TOWARDS NEUROMORPHIC GRADIENT DESCENT: EXACT GRADIENTS AND LOW-VARIANCE ONLINE ESTIMATES FOR SPIKING NEURAL NETWORKS

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF PHD.

By

Florian Bacho

September 2023

Abstract

Spiking Neural Networks (SNNs) are biologically-plausible models that can run on low-powered non-Von Neumann neuromorphic hardware, positioning them as promising alternatives to conventional Deep Neural Networks (DNNs) for energy-efficient edge computing and robotics. Over the past few years, the Gradient Descent (GD) and Error Backpropagation (BP) algorithms used in DNNs have inspired various training methods for SNNs. However, the non-local and the reverse nature of BP, combined with the inherent non-differentiability of spikes, represent fundamental obstacles to computing gradients with SNNs directly on neuromorphic hardware. Therefore, novel approaches are required to overcome the limitations of GD and BP and enable online gradient computation on neuromorphic hardware.

In this thesis, I address the limitations of GD and BP with SNNs by proposing three algorithms. First, I extend a recent method that computes exact gradients with temporally-coded SNNs by relaxing the firing constraint of temporal coding and allowing multiple spikes per neuron. My proposed method generalizes the computation of exact gradients with SNNs and enhances the tradeoffs between performance and various other aspects of spiking neurons. Next, I introduce a novel alternative to BP that computes low-variance gradient estimates in a local and online manner. Compared to other alternatives to BP, the proposed method demonstrates an improved convergence rate and increased performance with DNNs. Finally, I combine these two methods and propose an algorithm that estimates gradients with SNNs in a manner that is compatible with the constraints of neuromorphic hardware. My empirical results demonstrate the effectiveness of the resulting algorithm in training SNNs without performing BP.

To both my uncle and my granddad, who left us too soon.

Acknowledgements

I would like to express my gratitude:

My supervisor, Dr. Dominique Chu, for his guidance, patience, and expertise.

The University of Kent and its staff for providing me with a supportive academic environment.

My family and my partner Daniela for their love and support.

Finally, my friends and colleagues with whom I had lengthy discussions throughout my academic journey.

Publications

- Bacho, F. and Chu, D. (2023). Exploring Trade-Offs in Spiking Neural Networks. Neural Computation, 35(10), pp. 1627–1656. doi: 10.1162/neco_a_01609.
- Bacho, F. and Chu, D. (2024). Low-variance Forward Gradients using Direct Feedback Alignment and momentum. Neural Networks, 169, pp. 572–583. doi: 10.1016/j.neunet.2023.10.051.

Source Code

- `https://github.com/Florian-BACHO/bats`
- `https://github.com/Florian-BACHO/FDFA`
- `https://github.com/Florian-BACHO/SFDFA`

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Over the past decades, *Deep Neural Networks* (DNNs) trained with the well-established *Gradient Descent* (GD) and the *Error Backpropagation* (BP) algorithms have become the workhorse of modern machine learning. These powerful computational models have started a new era of artificial intelligence, enabling computers to perform various challenging tasks with state-of-the-art performance such as computer vision (Krizhevsky, Sutskever and Hinton 2012b; Szegedy, Toshev and Erhan 2013), natural language processing (Vaswani et al. 2017; Devlin et al. 2018; Brown et al. 2020), or reinforcement learning (Mnih et al. 2013, 2016). However, the hardware and energy cost of DNNs represent a significant challenge in terms of sustainability and restrict the practical applicability of deep learning in resource-limited environments such as low-powered edge devices (Daghero, Pagliari and Poncino 2021). Therefore, exploring more energy-efficient alternatives to DNNs is crucial to address the environmental cost of machine learning and provide sustainable solutions for edge computing.

One possible alternative to DNNs is *Spiking Neural Networks* (SNNs). Inspired by the dynamics of biological neurons and offering a greater computational power than DNNs (Maass 1997), SNNs process information through discrete spatio-temporal events known as *spikes* rather than continuous activations. More importantly, spikes enable efficient event-driven implementations of neural networks on massively parallel *neuromorphic hardware* that only consumes a fraction of the power required by DNNs on von Neumann computers (Furber et al. 2014; Höppner et al. 2022; Akopyan et al. 2015; Schemmel et al. 2010; Pehle et al. 2022; Davies et al. 2018; Frady et al. 2022). Therefore, SNNs represent promising alternatives to DNNs for energy-efficient machine learning and low-powered edge computing. However, the lack of general-purpose training algorithms that can match the performance of DNNs trained with GD and BP while also being compatible with neuromorphic hardware significantly limits the widespread adoption of SNNs (Grüning and Bohté 2014). Therefore, the exploration of novel training algorithms compatible with neuromorphic hardware represents one major step towards the adoption of neuromorphic computing for real-world machine learning applications.

One fundamental obstacle to the application of GD and BP to SNNs is the non-differentiability of spikes. Unlike DNNs where gradients can be computed by differentiating the continuous activations of neurons, the discrete activation of spiking neurons is generally not differentiable (Bohté, Kok and Poutré 2000; Shrestha and Orchard 2018). Many alternatives have been proposed to address this issue and compute approximate gradients of SNNs. For instance, surrogate derivatives can be used to replace the inexisting derivative between spikes and the membrane potential (Shrestha and Orchard 2018; Zheng et al. 2020; Wu et al. 2018). While these methods achieve performance close to that of DNNs, they often backpropagate errors even when no spike occurs, contradicting the event-based nature of neuromorphic hardware. In contrast, other methods compute approximate gradients by propagating errors through spike timings, making them particularly suitable for neuromorphic hardware (Göltz et al. 2021). However, the absence of a closed-form solution for spike timings represents a major obstacle in deriving exact gradients for SNNs (Mostafa 2016; Comsa et al. 2020; Göltz et al. 2021; Wunderlich and Pehle 2021; Lee, Haghighatshoar and Karbasi 2023) and event-based methods often have to rely on approximations (Bohté, Kok and Poutré 2000; Jin, Zhang and Li 2018).

In addition to the lack of differentiability of spikes, the BP algorithm represents one of the major obstacles to the computation of gradients on neuromorphic hardware. When computing gradients of SNNs with BP, the states of neurons are first stored during a forward pass and then unfolded through both time and space to propagate errors in reverse during a backward pass (Rumelhart, Hinton and Williams 1986; Shrestha and Orchard 2018; Göltz et al. 2021; Wunderlich and Pehle 2021). However, neuromorphic hardware operate in real-time and are thus unable to propagate information in reverse, as required by BP. Moreover, they exclusively perform computations with locally available information, such as pre-synaptic and post-synaptic spikes, thereby avoiding the bottlenecks created by data transfers between processing elements and memory (Sze et al. 2017; Schuman et al. 2022). Therefore, storing the states of neurons for the backward pass of BP not only violates the locality principle inherent to neuromorphic hardware but also substantially increases the risk of introducing memory bottlenecks causing processing latency and significant power consumption (Horowitz 2014; Sze et al. 2017; Schuman et al. 2022).

1.1 Thesis Contributions

The aim of this thesis is to address the challenges associated with the computation of gradients on neuromorphic hardware. More particularly, I propose three algorithms:

- I extend *Fast & Deep* (Göltz et al. 2021), an algorithm that isolates closed-form solutions for post-synaptic spike timings and computes exact gradients of temporally-coded SNNs. The proposed algorithm generalizes the computation of exact gradients to SNNs firing multiple spikes per neuron. By relaxing the spike constraint associated with temporal coding, my method achieves improved tradeoffs between performance, convergence, energy efficiency, latency and robustness to noise and weight quantization when compared to the original Fast &

Deep algorithm.

- I introduce the *Forward Direct Feedback Alignment* (FDFA), a novel method that combines *Activity-Perturbed Forward Gradients* (Ren et al. 2023) with *Direct Feedback Alignment* (Nøkland 2016) and *Momentum* to estimate the derivatives between output and hidden neurons as feedback connections. By computing low-variance gradient estimates without performing backpropagation, the proposed FDFA algorithm improves the performance and convergence of DNNs compared to other online and local alternatives to BP.

- I propose the *Spiking Forward Direct Feedback Alignment* (SFDFA) algorithm, a spiking adaptation of the FDFA algorithm. By using additional payloads carried by spikes called *grades* and by computing local gradients during the inference, the proposed SFDFA algorithm estimates gradients of SNNs in a local and online manner that complies with the constraints of neuromorphic hardware. When compared to Direct Feedback Alignment, the SFDFA algorithm offers advantages both in terms of performance and convergence.

These algorithms collectively aim to provide novel perspectives for training SNNs. By addressing the fundamental challenges associated with local and online gradient computations, my work represents a promising direction for fast, performant, energy-efficient and robust trainings of SNNs on neuromorphic hardware.

1.2 Thesis Outline

The rest of this thesis is structured as follows:

- In Chapter 2, I review the relevant literature forming the technical background of this thesis. I provide an overview of biological neurons, deep neural networks, and spiking neural networks including neuron models, network architectures, hardware accelerations neural coding, and training methods. Finally, I review various gradient-based optimization techniques and the two main approaches for automatic differentiation.

- In Chapter 3, I start by reviewing the *Fast & Deep* (Göltz et al. 2021) algorithm applied to temporally-coded SNNs. I then introduce my extension of Fast & Deep that relaxes the spike constraint of temporal coding and computes gradients with multiple spikes per neuron. I finally compare my proposed method with the original Fast & Deep algorithm on multiple criteria such as convergence, likelihood, performance, sparsity, classification latency, and robustness to noise and weight quantization. In addition, I investigate the role of weight initialization in Fast & Deep and highlight several tradeoffs between these different aspects.

- In Chapter 4, I first review the *Forward Gradient* (Baydin et al. 2022) algorithm and its variants for DNNs (Silver et al. 2021; Ren et al. 2023) as well as the *Direct Feedback Alignment* (Nøkland 2016) algorithm. I then introduce our *Forward Direct Feedback Alignment* (FDFA) algorithm and provide both theoretical and empirical results demonstrating the advantages of my method in terms of performance and convergence compared to other alternatives to BP.

- In Chapter 5, I review the application of Direct Feedback Alignment to SNNs and introduce our *Spiking Forward Direct Feedback Alignment* (SFDFA) algorithm. I describe how the local gradient of post-synaptic spikes can be computed in an online manner and show the existence of critical points causing gradient explosions. I then propose an ad-hoc solution to these critical points and derive an eligibility trace for neuromorphic hardware compatibility. Finally, I compare the proposed SFDFA algorithm with Direct Feedback Alignment and demonstrate the advantages of my method in terms of performance and convergence.

Chapter 2

Literature Review

In this chapter, we give an overview of the relevant literature that forms the technical background of this thesis. We start by providing a brief overview of biological neurons in Section 2.1. In Sections 2.2, we give a brief history of artificial neural networks, introduce different architectures used in modern deep learning as well as the possible hardware acceleration for DNNs. In Section 2.3, we describe several popular spiking neuron models, and discuss the different simulation techniques and hardware acceleration for SNNs. Then, we review methods of spike coding as well as unsupervised and supervised approaches for training SNNs. Finally, we review in Sections 2.4 and 2.5 gradient-based optimization and automatic differentiation techniques respectively.

2.1 The Brain

The human brain is capable of performing a broad range of complex cognitivee tasks such as vision, motor control, sound recognition, speech, or planning with a power consumption of only 20 Watts (Balasubramanian 2021). For comparison, standard computers consume, on average, 80 Watts. With its estimated 86 billion neurons and kilometers of connections per cubic millimeter, the human brain is a powerful and energy-efficient structure (Gerstner and Kistler 2002; Azevedo et al. 2009; Gerstner et al. 2014).

2.1.1 Structure and Dynamics of Biological Neurons

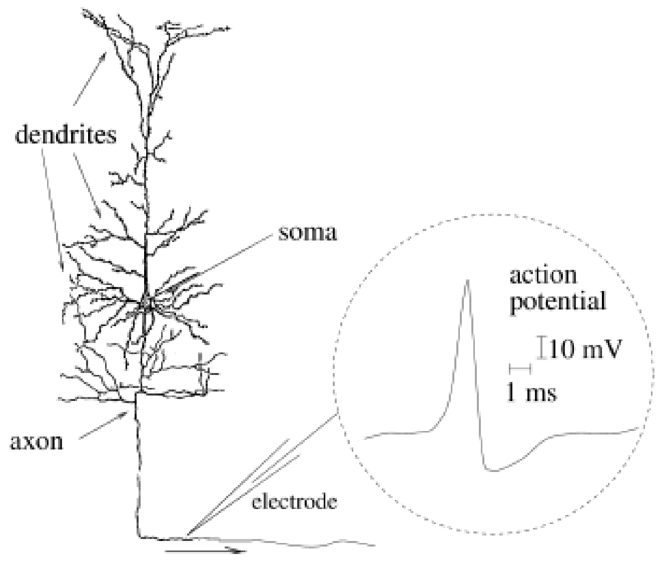


Figure 1: Reproduction of a drawing of a single neuron by Ramón y Cajal. This figure depicts the dendrites, the soma, and the axon of the neuron as well as a schematic of an action potential recorded by an electrode placed at the axon. Figure from (Gerstner et al. 2014).

Neurons are the fundamental processing units of the brain. They are composed of synapses, a cell body (or *soma*), and an axon. Synapses are the inputs of the neuron. They receive information from other neurons and transmit them to the soma through the dendrites. The soma integrates input stimuli and exhibits important non-linear dynamics. When input stimuli are strong enough to excite the soma past a certain threshold, an output signal is emitted and delivered to downstream neuron through the axon (Kandel, Schwartz and Jessell 1991; Gerstner and Kistler 2002; Gerstner et al. 2014). See Figure 1 for a schematic representation of a single biological neuron.

The signals transmitted between neurons are composed of short and discrete electrical pulses referred to as *action potentials* or *spikes* (see Figure 1 for a schematic representation). Occurring in time, sequences of action potentials emitted by pre-synaptic neurons (or *spike trains*) cause complex biochemical reactions that trigger extracellular ion influxes into the soma. This leads to voltage changes in the *membrane potential* of the neuron, depending on the strengths of its synaptic connections. In the absence of stimulus, the membrane potential relaxes to its resting potential due to ion leakage. Referred to as *Post-Synaptic Potential* (PSP), this voltage response to a pre-synaptic action potential can either be excitatory (i.e. induces a positive charge) or inhibitory (i.e. induces a negative charge). When the membrane potential reaches the firing threshold of the neuron, an action potential is released through the axon, and the membrane potential returns to its resting potential for further spike integration (Kandel, Schwartz and Jessell 1991; Gerstner and Kistler 2002; Gerstner et al. 2014).

2.1.2 Information Coding

Biologial neurons can exhibit a large range of spiking behaviors. However, it is still unclear how neurons encode information in the brain. Two main theories are at the center of debates: rate coding or spike time coding (Brette 2015; Gautrais and Thorpe 1998).

Rate Coding

In *rate coding*, the information carried by a neuron is defined by its firing rate over a period of time (Brette 2015). The firing rate f of a neuron is represented by the temporal average of the spike count n, such as:

$$f = \frac{n}{T} \tag{1}$$

where T is the time period of the measurement.

Alternatively, information can be encoded through the firing rate of a give population of neurons. (Brette 2015). This encoding mechanism is commonly referred to as *population coding*.

However, computing neurons firing rates requires a large temporal window T which is in contradiction with the fast responses observed in the brain (Thorpe, Delorme and

Van Rullen 2001a). For example, it has been shown that the brain of primates can respond to visual stimuli in only 100-150ms (Thorpe and Imbert 1989). As information about stimuli would cross approximately 10 layers of neurons, Thorpe and Imbert argued that each neuron would have to respond under 10ms which is too fast for their average firing rate ($\approx$ 100 Hz). This suggests that information might be carried by spike timings rather than firing rates.

Temporal Coding

In the theory of *temporal coding*, the firing rate defines the rate of information rather than encoding stimuli (Brette 2015). Instead, spikes encode stimuli through their relative timings (Brette 2015; Thorpe, Delorme and Van Rullen 2001a). Such a coding scheme maximizes the amount of information carried by individual spikes as only a single action potential per neuron is sufficient to process inputs (Thorpe, Delorme and Van Rullen 2001a). This also allows fast information processing that could explain the observed response time of the brain (Thorpe and Imbert 1989; Thorpe, Delorme and Van Rullen 2001a).

For example, in *First Time Latency* coding or *Time-To-First-Spike* (TTFS), information is encoded through the relative latency between spikes and stimuli, as observed in both the auditory (Furukawa and Middlebrooks 2002) and visual (Reich, Mechler and Victor 2001) systems. However, most studies of TTFS assume an independent stimulus onset in the brain which, in most situations, does not exist (Phillips 1998).

Other studies have suggested that the order in which neurons spike could encode information (Gautrais and Thorpe 1998). Thus referred to as *Rank Order Coding* (ROC), such coding scheme is decorrelated from the stimulus onset and is believed to be more robust to spike jitter than TTFS (Gautrais and Thorpe 1998).

Finally, *Phase-of-Firing* is another form of temporal coding where information is temporally coded with respect to the oscillations of the neural population. In this case, the synchronous population firing is considered to be a reference offset for the relative timings of spikes (Chase and Young 2007).

2.1.3 Biological Plasticity

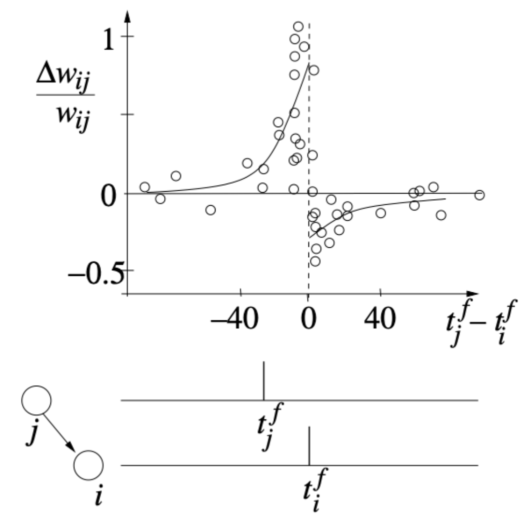


Figure 2: Temporal requirements of synaptic changes Δw_{ij} between a pre-synaptic neuron j firing at time t_j^f and a post-synaptic neuron i firing at time t_i^f in cultured hippocampal neurons. *Long-Term Potentiation* (LTP) occurs if the pre-synaptic spike shortly precedes the post-synaptic spike. *Long-Term Depression* (LTD) occurs if the pre-synaptic spike is fired after the post-synaptic spike. Figure from (Gerstner and Kistler 2002) and data points from (Bi and Poo 1998).

When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A 's efficiency, as one of the cells firing B, is increased"

Hebb (1949)

The underlying mechanisms of learning in the brain is not yet fully understood to this day and remain a topic of ongoing research. However, numerous theories and in vivo studies have provided valuable insights into fundamental plasticity mechanisms of the brain that are considered to be essential for learning. For example, Hebb introduced in 1949 the *Cell Assembly Theory* or *Hebbian Learning* whereby neurons that exhibit synchronous firing undergo synaptic strengthening, highlighting an important mechanism underlying learning in the brain.

While Hebb's postulate describes the conditions in which synaptic strengthening occurs, it neglects the temporal aspects of plasticity Gerstner and Kistler (2002). Multiple in vivo studies of synaptically connected neurons have enabled high-resolution recordings of the temporal requirements of Hebbian learning in the brain (Levy and Steward 1983; Magee and Johnston 1997; Bi and Poo 1998). Figure 19 depicts these temporal dependencies between pre-synaptic and post-synaptic neurons recorded in cultured hippocampal neurons (Bi and Poo 1998). It shows that positive change of weights or *Long-Term Potentiation* (LTP) occurs if the pre-synaptic spike occurs shortly before the post-synaptic spike. In contrast, a decrease in synaptic efficiency or *Long-Term Depression* (LTD) occurs if the pre-synaptic spike is fired after the post-synaptic spike. More importantly, the strength of these changes depends on the relative timing between the pre-synaptic and post-synaptic spikes, exponentially decreasing as the relative difference in timing increases. Known as *Spike Time-Dependent Plasticity* (STDP), these observations align with Hebb's theory of learning as only pre-synaptic spikes that occur before action potentials contribute to synaptic growth (Gerstner and Kistler 2002).

2.2 Artificial Neural Networks

The idea of reproducing the brain's computational behaviors to give computers the capability to learn has been a topic of interest since as early as 1943. Since, multiple abstracted mathematical models of neurons and training algorithms have been proposed. In this section, we give a brief history and a detailed description of these neuron models.

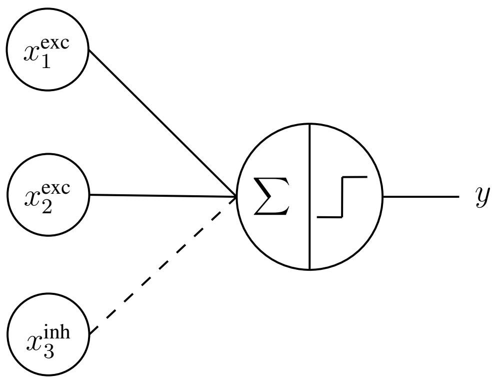


Figure 3: Illustration of the McCulloch-Pitts neuron model. Here, the neuron receives three binary inputs x_1, x_2, and x_3 that are summed and then passed through a threshold function to produce a binary output y.

2.2.1 The McCulloch-Pitts Neuron

In 1943, McCulloch and Pitts proposed the first computational model of a neuron that implements boolean logic functions. Named the *McCulloch-Pitts* (MCP) neuron, it is composed of n excitatory inputs $\boldsymbol{x}^{\text{exc}} \in \{0,1\}^n$, m inhibitory inputs $\boldsymbol{x}^{\text{inh}} \in \{0,1\}^m$ that are aggregated then passed through a linear threshold gate function $\Theta(x)$ producing a single binary output $y \in \{0,1\}$ (McCulloch and Pitts 1943), such as:

$$\begin{aligned} s &:= \sum_{j=1}^{n} x_j^{\text{exc}} - \sum_{j=1}^{m} x_j^{\text{inh}} \\ y &:= \Theta\left(s - \vartheta\right) \end{aligned} \tag{2}$$

where $\vartheta \in \mathbb{R}^+$ is the threshold of the neuron and

$$\Theta(x) := \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

is the Heavyside step function. The McCulloch-Pitts neuron model is capable of computing simple linear binary functions such as the OR and AND gates but is incapable of performing non-linear functions such as the exclusive OR (or XOR). Therefore, its computational power is only limited to manually designed and linearly separable boolean tasks. While the MCP neuron has represented an important step towards the development of artificial neural networks, no learning mechanism has been introduced with the model, limiting its applicability to complex problems.

2.2.2 The Perceptron

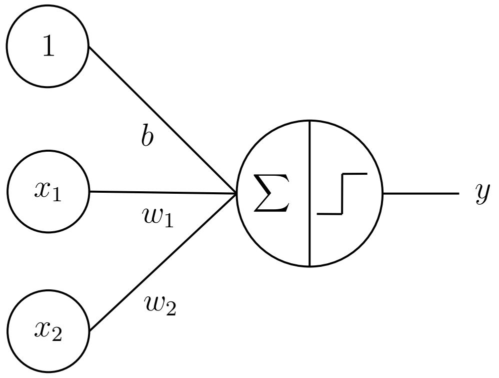


Figure 4: Illustration of the Perceptron. Here, the neuron receives two real-valued inputs x_1 and x_2 and a bias b is added (represented as an additional connection with a constant input of 1). All inputs are scaled by their corresponding weights, summed together then passed through a threshold function to produce a binary output y.

Algorithm 1: Perceptron algorithm

```
1:  Input: Training data $\mathcal{D} = \{\boldsymbol{x}_i, \widehat{y}_i\}_{i=1}^{n_{\text{data}}}$
2:  Randomly initialize $\boldsymbol{w}$ and $b$.
3:  repeat
4:    {Loop through training samples}
5:    for $i = 1, n_{\text{data}}$ do
6:      $y \leftarrow \Theta(\boldsymbol{w} \cdot \boldsymbol{x}_i + b)$ {Infer neuron}
7:      if $y_i = 1$ and y = 0 then
8:        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{x}_i$ {False negative}
9:      end if
10:     if $y_i = 0$ and y = 1 then
11:       $\boldsymbol{w} \leftarrow \boldsymbol{w} - \boldsymbol{x}_i$ {False positive}
12:     end if
13:   end for
14: until converged
```

In 1958, Rosenblatt proposed the Perceptron based on the MCP neuron model and

Hebb findings on synaptic efficiency and plasticity. In contrast with the MCP neuron model, the Perceptron uses real-valued inputs $\boldsymbol{x} \in \mathbb{R}^n$ and weights $\boldsymbol{w} \in \mathbb{R}^n$ between the inputs and the neuron, representing synaptic strengths. Unlike the MCP, the perceptron does not use absolute inhibition, meaning that synapses are capable of freely transitioning between being inhibitory and excitatory, and vice versa. Moreover, a bias $b \in \mathbb{R}$ is introduced in the model, which corresponds to the negative of the MCP threshold ϑ. The result of the weighted sum between the inputs and the weights is then passed through the Heavyside step function to produce a binary output, such as:

$$y := \Theta\left(\sum_{j=1}^{n} w_j \; x_j + b\right) \tag{4}$$

In vector form, Equation 4 is written as:

$$y = \Theta\left(\boldsymbol{w}\boldsymbol{x} + b\right) \tag{5}$$

Note that the threshold ϑ of the MCP model (Equation 2) is replaced by the trainable bias b in the equation of the Perceptron.

In addition to these modifications, Rosenblatt proposed a training algorithm called the *Perceptron algorithm* (see Algorithm 1). Intuitively, the Perceptron algorithm modifies the weights when samples are misclassified. When a false negative prediction occurs (i.e. the target is positive but the prediction is negative), the dot product between the inputs $\boldsymbol{x}$ and the weights $\boldsymbol{w}$ is negative but needs to be positive. Therefore, the inputs $\boldsymbol{x}$ are added to the weights to reinforce the result of the dot product $\boldsymbol{w} \cdot \boldsymbol{x}$. Inversely, when a false positive prediction occurs (i.e. the target is negative but the prediction is positive), the dot product between the inputs $\boldsymbol{x}$ and the weights $\boldsymbol{w}$ is positive but needs to be negative. Therefore, the inputs $\boldsymbol{x}$ are subtracted from the weights, which increases the result of the dot product $\boldsymbol{w} \cdot \boldsymbol{x}$.

While the Perceptron algorithm is capable of learning to perform binary classification, it is limited to data that is linearly separable by a hyperplane (Rosenblatt 1958;

Minsky and Papert 2017; Bishop 1995). To enable the Perceptron to classify non-linear data, successive layers of neurons can be stacked, resulting in what is known as a *Multi-Layer Perceptron* (MLP) (see Figure 5 for an example). In theory, the MLP has the capability to approximate any boolean function as long as the hidden layers are sufficiently large (Bishop 1995). However, the original Perceptron algorithm was designed for single-layer networks and does not contain a learning rule for hidden neurons, as the contributions of hidden neurons to the network outputs are not evident. Commonly referred to as the *credit assignment problem*, this issue can be overcome by using differentiable activation functions instead of the Heaviside step function.

2.2.3 Continuous Neurons and Error Backpropagation

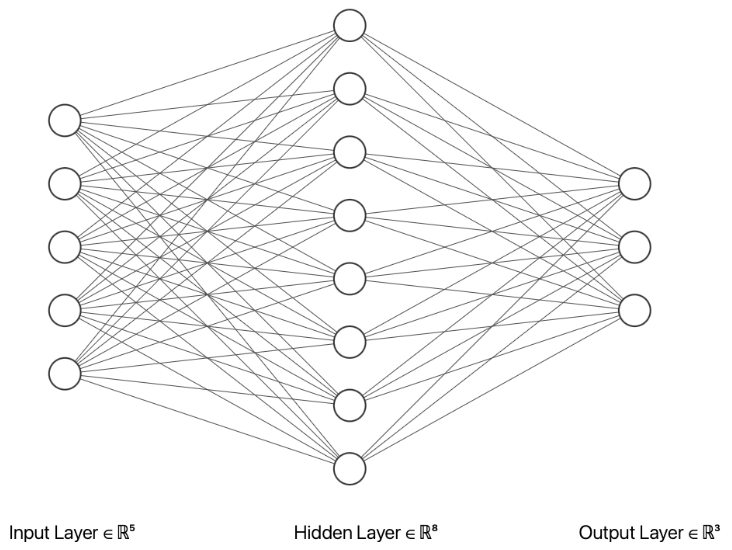

Input Layer $\in \mathbb{R}^5$
Hidden Layer $\in \mathbb{R}^8$
Output Layer $\in \mathbb{R}^3$

Figure 5: Example of a fully-connected neural network with two layers (MLP).

To overcome the credit assignment problem in MLPs, Rumelhart, Hinton and Williams proposed in 1986 the Backpropagation (BP) algorithm, which uses the chain rule with continuously-activated neurons to backpropagate output errors to hidden neurons.

Formally, we denote by $\boldsymbol{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ the weights and by $\boldsymbol{b}^{(l)} \in \mathbb{R}^{n_l}$ the biases for the l^{th} layer with n_l neurons. In contrast to the original Perceptron model, which

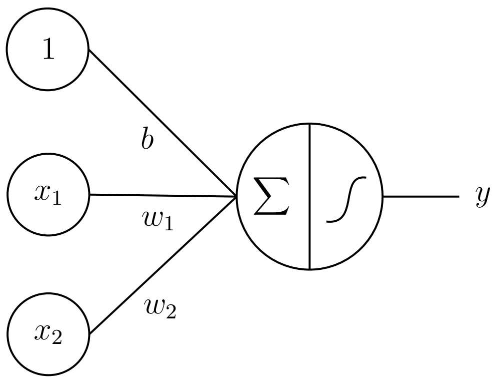


Figure 6: Illustration of a continuously-activated artificial neuron. Here, the neuron receives two real-valued inputs x_1 and x_2 and a bias b is added (represented as an additional connection with a constant input of 1). All inputs are scaled by their corresponding weights, summed together then passed through a continuous and differentiable activation to produce a real-valued output y.

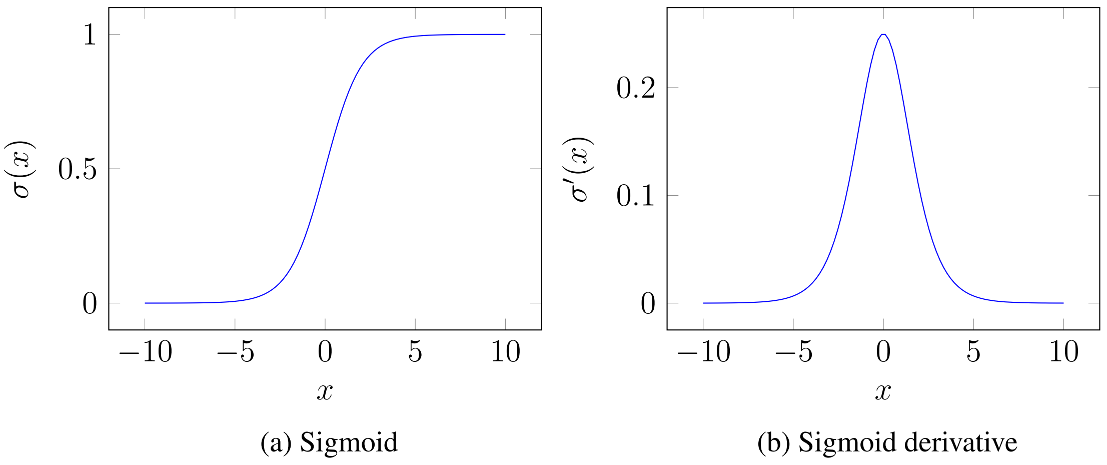


Figure 7: Sigmoid activation function (Figure 7a) and its first order derivative (Figure 7b). The sigmoid function monotonically increases and saturates to 0 and 1. This causes its derivative to vanish as x tends towards $-\infty$ and ∞.

Algorithm 2: Error Backpropagation (BP) algorithm given a single training sample. Here $\odot$ is the Hadamard product (or elementwise product) and $\sigma'(x)$ is the first order derivative of $\sigma(x)$.

```
1: **Input:** Training sample $\{\boldsymbol{x}, \widehat{\boldsymbol{y}}\}$
2: $\boldsymbol{y}^{(l)} \leftarrow \boldsymbol{x}$ {Set inputs}
3: {Forward pass}
4: for $l = 1, L$ do
5:     $\boldsymbol{y}^{(l)} \leftarrow \sigma\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$ {Infer each layer}
6: end for
7: $\boldsymbol{\delta}^{(L)} \leftarrow \frac{\partial\mathcal{L}(\boldsymbol{x})}{\partial\boldsymbol{y}^{(L)}}$ {Compute output errors}
8: {Backward pass}
9: for $l = L, 1$ do
10:     $\boldsymbol{\nabla W}^{(l)} \leftarrow \boldsymbol{\delta}^{(l)} \odot \sigma'\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\right)\left(\boldsymbol{y}^{(l-1)}\right)^T$ {Compute weights gradient}
11:     $\boldsymbol{\nabla b}^{(l)} \leftarrow \boldsymbol{\delta}^{(l)} \odot \sigma'\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$ {Compute bias gradient}
12:     $\boldsymbol{\delta}^{(l-1)} \leftarrow \left(\boldsymbol{W}^{(l)}\right)^T \boldsymbol{\delta}^{(l)} \odot \sigma'\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$ {Backpropagate errors to
        upstream layer}
13: end for
14: return $\left\{\boldsymbol{\nabla W}^{(1)}, \boldsymbol{\nabla W}^{(2)}, \cdots, \boldsymbol{\nabla W}^{(L)}\right\}$ and $\left\{\boldsymbol{\nabla b}^{(1)}, \boldsymbol{\nabla b}^{(2)}, \cdots, \boldsymbol{\nabla b}^{(L)}\right\}$
```

uses a binary activation function, the neurons introduced in (Rumelhart, Hinton and Williams 1986) use a continuous and differentiable activation function $\sigma(x)$. Using a vector representation, the activations $\boldsymbol{y}^{(l)}$ for the l^{th} layer are defined as:

$$\boldsymbol{y}^{(l)} := \sigma\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\right) \tag{6}$$

where $\boldsymbol{y}^{(0)} := \boldsymbol{x}$ are the inputs and $\sigma(\boldsymbol{x})$ is elementwise. In MLPs, the activation function $\sigma(x)$ is defined as a logistic function, such as the sigmoid:

$$\sigma(x) := \frac{1}{1 + e^{-x}} \tag{7}$$

See Figure 7 for a visualization of the sigmoid function.

By defining a differentiable loss function $\mathcal{L}(\boldsymbol{x})$ measuring the prediction errors, a gradient $\boldsymbol{\nabla W}^{(l)}$ of the loss function can be computed, such as:

$$\begin{aligned}\boldsymbol{\nabla W}^{(l)} &:= \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{W}^{(l)}} \\ &= \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(L)}} \frac{\partial \boldsymbol{y}^{(L)}}{\boldsymbol{y}^{(l)}} \frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{W}^{(l)}} \\ &= \boldsymbol{\delta}^{(l)} \frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{W}^{(l)}}\end{aligned} \tag{8}$$

where $\boldsymbol{\delta}^{(l)}$ are the error for the neurons of the layer l. For the output layer L, these errors are defined as the derivative of the loss function:

$$\boldsymbol{\delta}^{(L)} := \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(L)}} \tag{9}$$

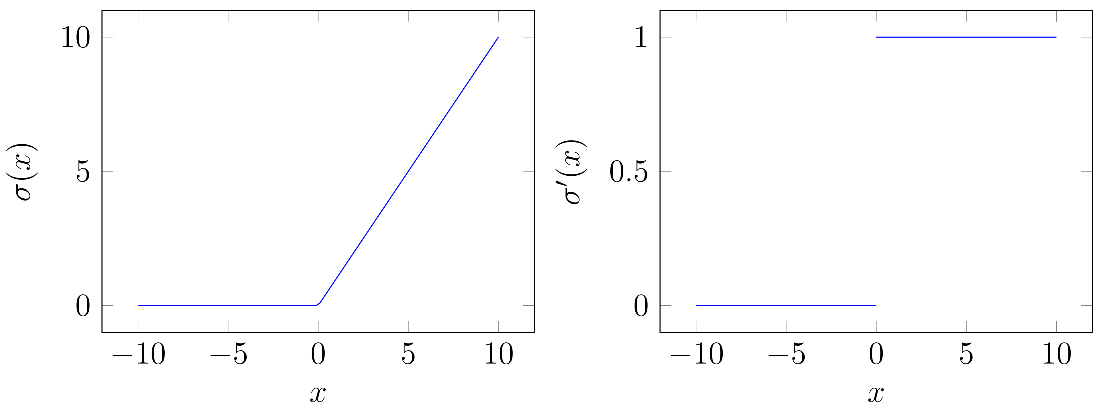
For hidden layers $l < L$, output errors can be sequentially backpropagated to hidden layers using the chain rule, such as:

$$\begin{aligned}\boldsymbol{\delta}^{(l)} &= \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(L)}} \frac{\partial \boldsymbol{y}^{(L)}}{\boldsymbol{y}^{(l)}} \\ &= \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(L)}} \frac{\partial \boldsymbol{y}^{(L)}}{\boldsymbol{y}^{(L-1)}} \cdots \frac{\partial \boldsymbol{y}^{(l+1)}}{\boldsymbol{y}^{(l)}} \\ &= \boldsymbol{\delta}^{(l+1)} \frac{\partial \boldsymbol{y}^{(l+1)}}{\boldsymbol{y}^{(l)}}\end{aligned} \tag{10}$$

See Algorithm 2 for the full algorithm given a single training sample. Finally, the weights are updated using a gradient-based optimization algorithm — see Section 2.4 for details.

By effectively solving the credit assignment problem in MLPs, the error backpropagation algorithm has represented a significant turning point in the development of artificial neural networks. Nowadays, BP remains widely used and is considered to be the workhorse of modern deep learning. Its popularity and widespread usage attest its fundamental importance in training neural networks.

2.2.4 The Fundamental Deep Learning Problem



(a) ReLU (b) ReLU derivative

Figure 8: ReLU activation function (Figure 8a) and its first order derivative (Figure 8b). The sigmoid function monotonically increases and saturates to 0 and 1. This causes its derivative to vanish as x tends towards $-\infty$ and ∞.

Until the 1990s, artificial neural networks were mostly kept shallow, as in MLPs (Schmidhuber 2015). In fact, several works in the late 1980s have established universal approximation theorems stating that any multivariate continuous function can be approximated by a network of sigmoid neurons with bounded depths and arbitrary width (Cybenko 1989; Hornik, Stinchcombe and White 1989; Hecht-Nielsen 1989; Hornik 1991). Moreover, while BP is able to train neural networks that contain many layers — thus referred to as *Deep Neural Networks* (DNNs) — it was found to be difficult in practice (Schmidhuber 2015). In 1991, Hochreiter identified that, when using logistic activation functions in DNNs, cumulative backpropagated errors either grow out of bound or shrink rapidly, causing the now famous *exploding* or *vanishing gradients* problem (Hochreiter 1991), also referred to as the *Fundamental Deep Learning Problem* (Schmidhuber 2015).

One of the main causes of vanishing gradients is the nature of logistic functions and, more precisely, their derivatives. For example, Figure 7a shows that the sigmoid function saturates in 0 and 1 as its input tends towards $-\infty$ and ∞ respectively. This causes its derivative (Figure 7b) to vanish, decreasing towards 0 in these limits. Therefore, errors backpropagated to upstream neurons also vanish and prevent the weights from changing their values (Basodi et al. 2020).

Many solutions to the vanishing gradient problem have been proposed (Schmidhuber 1992; Nair and Hinton 2010; Glorot, Bordes and Bengio 2011; Ioffe and Szegedy 2015; Basodi et al. 2020). In particular, the choice of the activation function seems to be important to avoid vanishing gradients. For example, the use of the Rectified Linear Unit (ReLU) activation function and its variants — e.g. Leaky ReLU or the Scaled Exponential Linear Unit (SELU) — have empirically demonstrated major improvements over the logistic functions in DNNs (Nair and Hinton 2010; Glorot, Bordes and Bengio 2011) and are, to this day, widely used in modern deep learning.

Formally, the ReLU activation function is defined as follows:

$$\sigma(x) := \max(0, x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{11}$$

Figure 8 visually shows this function and its derivative. In essence, the ReLU activation is linear for positive inputs and outputs zero for negative inputs. This causes its derivative to only saturate in negative values and output a constant for positive values, thus reducing the impact of vanishing gradients (Nair and Hinton 2010; Glorot, Bordes and Bengio 2011; Basodi et al. 2020).

Another technique widely used in modern DNNs to avoid exploding gradients is *Batch Normalization* (BN) (Ioffe and Szegedy 2015). In BN, the inputs of each layer are normalized to reduce their variance. This prevents the neurons from receiving inputs with large bounds and, consequently, avoids saturating the activation function (Ioffe and Szegedy 2015; Basodi et al. 2020).

By effectively solving the vanishing and exploding gradient problem, these methods have enabled the exploration and the successful training of deeper neural architectures, including *Convolutional Neural Networks* and *Recurrent Neural Networks*.

2.2.5 Convolutional Neural Networks

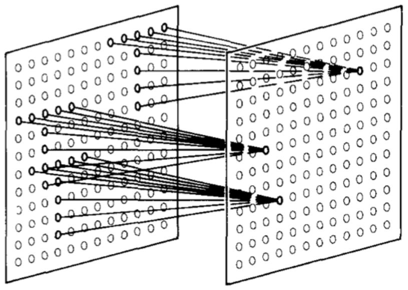


Figure 9: Illustration of the two dimensional neuron layout in the Neocognitron model. Here, individual neurons are connected to a sub-area of their two-dimensional inputs, locally detecting spacial features. Figure from (Fukushima 1988).

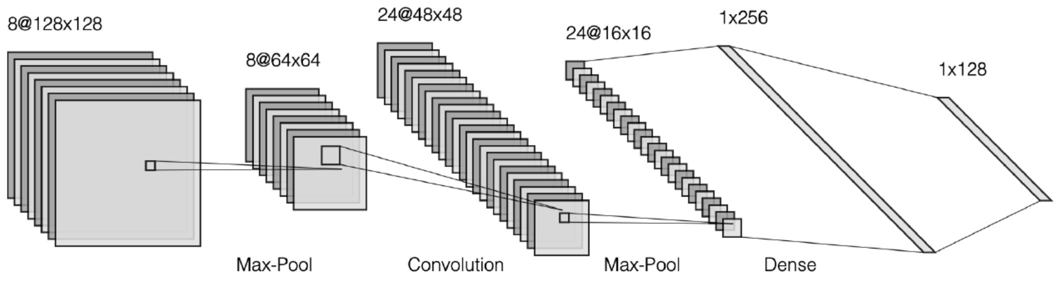


Figure 10: Example of a convolutional neural network. This network contains two convolutional layers, each followed by a max pooling layer. The outputs of the last pooling layer are then flattened and fed to two fully connected (dense) layers.

Inspired by the fact that neurons in the visual cortex individually respond to small regions of the visual receptive field (Hubel and Wiesel 1968), the principle of locally extracting spatial features in input patterns was first introduced with the Neocognitron in 1980 (Fukushima 1988). The Neocognitron model organizes neurons in a two-dimensional grid. These neurons then extract visual features from specific sub-areas of their inputs, similar to receptive fields found in the visual cortex (see Figure 9). By successively stacking such layers, the Neocognitron forms a network that closely resembles the hierarchical model of the visual cortex proposed by Hubel and Wiesel, thereby laying the foundation for the development of modern *Convolutional Neural*

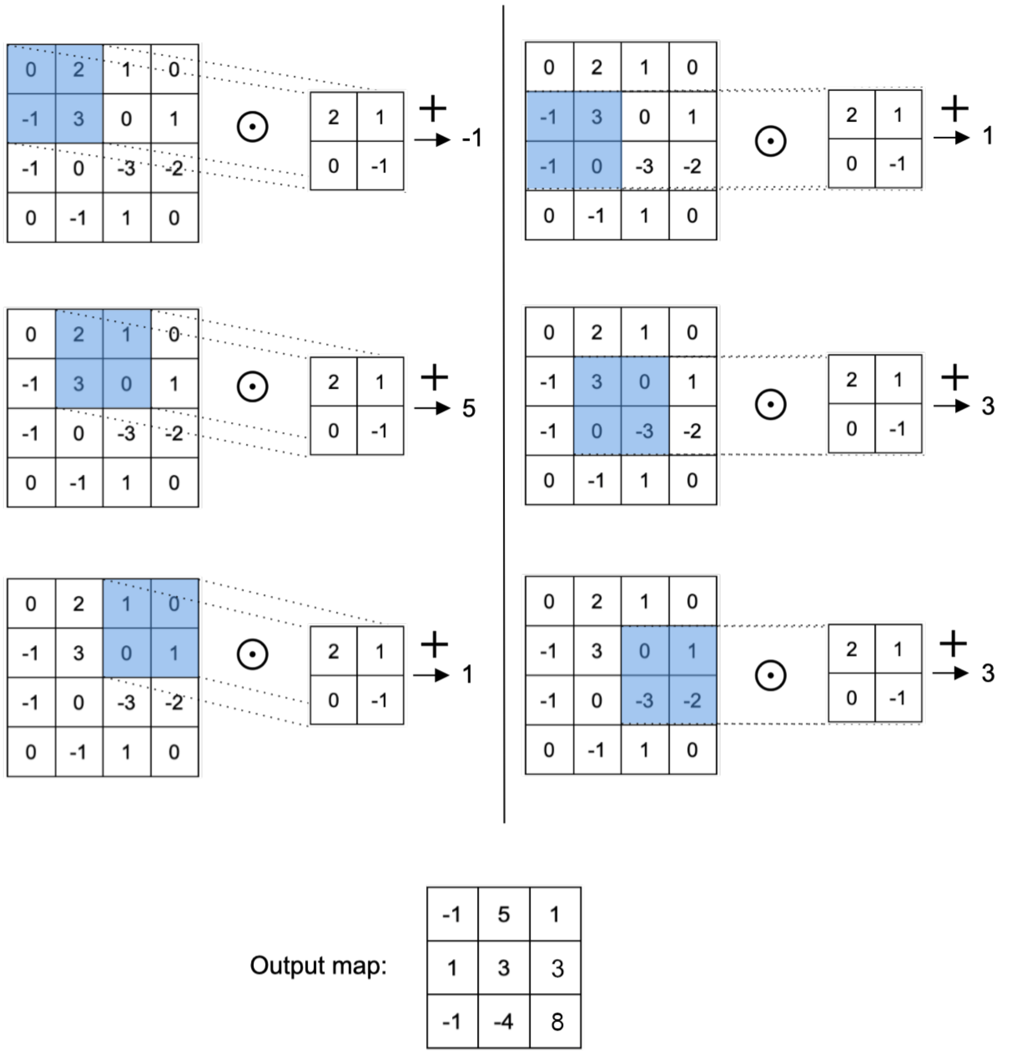


Figure 11: Example of a two-dimensional convolution performed with a single $2 \times 2 \times 1$ filter on a $4 \times 4 \times 1$ input map with a stride of 1 and no padding, no bias, and a linear activation function. This convolution results in a $3 \times 3 \times 1$ output map of activations.

Networks (CNNs).

In 1989, Yann Le Cunn successfully trained the first competition-winning CNN for classifying handwritten digits (MNIST) (LeCun, Cortes and Burges 2010) using BP (LeCun et al. 1989) which was then used to recognize hand-written ZIP code numbers. The following years, the same CNN was used for medical image segmentation (Zhang et al. 1991) as well as breast cancer detection (Zhang et al. 1994), establishing CNNs as a fundamental approach in the field of computer vision.

Often used in signal processing, convolution is a mathematical operation expressing the relation between a signal function and a kernel function. Denoted by $*$, convolution is defined as the integral of the product between the kernel function $g(t)$ and the signal $f(t)$ (Smith 1997), such as:

$$(f * g)(t) = \int_{-\infty}^{\infty} g(\tau) f(t - \tau) \, \mathrm{d}\tau \tag{12}$$

In essence, the convolution operation filters the signal with the kernel function given a shift t. This represents the base principle of convolutional layers in CNNs.

In a convolutional layer with index l, n_l trainable three-dimensional discrete filters $G^{(l)} \in \mathbb{R}^{k_l \times k_l \times d_{l-1}}$ (also known as *convolution kernels*) of width and height k_l and depth d_{l-1} are individually convolved along the horizontal and vertical axis of a discrete input *map* $Y^{(l-1)} \in \mathbb{R}^{w_{l-1} \times h_{l-1} \times d_{l-1}}$ of width w_{l-1}, height h_{l-1} and depth d_{l-1} (O'Shea and Nash 2015; Alzubaidi et al. 2021). For the first convolutional layer of a CNN, $Y^{(l-1)} = X$ is defined as the input image, and d_{l-1} represents the number of channels in the input image, which is 1 for grayscale and 3 for RGB images. A *stride* s_l is also defined to represent the number of pixels that are skipped when convolving filters, thus controlling the overlap of kernels during the convolution. Moreover, a *padding* of size p_l is added at the border of the input map to control the size of the output map (O'Shea and Nash 2015; Alzubaidi et al. 2021). It is common to fill these additional pixels with zeros but other methods exist such as *average padding*.

Using these parameters, the width w_l — and, similarly, the height h_l — of the output map $Y^{(l)}$ is calculated as follows:

$$w_l = \frac{w_{l-1} - k_l + 2p_l}{s_l} + 1 \tag{13}$$

The depth $d_l = n_l$ of the output map then equals the number of filters n_l in the layer. Assuming that the stride is $s_l = 1$ is one, the activation $y^{(l)}_{i,j,z}$ of the neuron at the position i, j and z in the output map is:

$$y^{(l)}_{i,j,z} = \sigma\left(\sum_{u=0}^{k_l}\sum_{v=0}^{k_l}\sum_{w=0}^{d_{l-1}} y^{(l)}_{i+u,\, j+v,\, w}\; g^{(l)}_{u,\, v,\, w} + b^{(l)}_z\right) \tag{14}$$

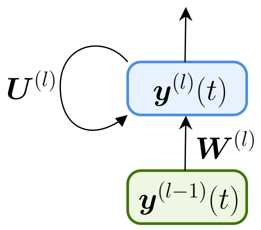
where $\boldsymbol{b}^{(l)} \in \mathbb{R}^{n_l}$ are trainable biases. See Figure 11 for an example of two-dimensional convolution performed by a convolutional filter.

Convolutional layers are commonly followed by *pooling* layers that aim to gradually reduce the dimensionality of the representation (O'Shea and Nash 2015; Alzubaidi et al. 2021). Pooling layers also perform a two-dimensional convolution operation but, unlike convolutional layers, they do not contain any trainable parameter as they solely sub-sample the input map. Several types of pooling layers have been introduced (Alzubaidi et al. 2021) including the widely used *max pooling* and *average pooling*. Pooling is often performed with a kernel size of 2 and a stride of 2, scaling down the input map by 25% of its original size (O'Shea and Nash 2015).

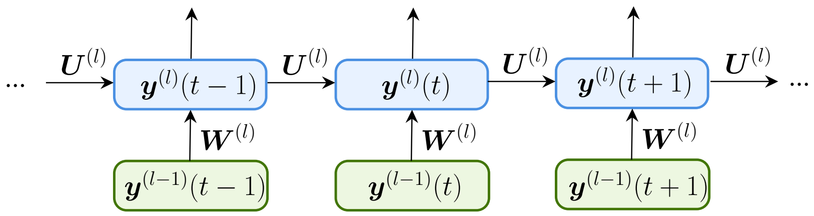
CNNs are therefore built by successively stacking convolutional and pooling layers to extract spatial features. To perform classification, fully connected layers can then be connected to the flattened output of the last convolutional — or pooling — layer. See Figure 10 for an example of CNN.

One of the main advantages of CNNs is that weights from filters are shared within the neuron of a convolutional layer, significantly reducing the memory requirements compared to fully connected layers. Moreover, convolutional layers are *translation invariant* — or *shift invariant* — meaning that, unlike fully connected DNNs, the model is not affected by a translation of its inputs. This makes CNNs better at generalizing on spacial data than fully connected networks and less likely to overfit on visual data (Alzubaidi et al. 2021). However, because the weights are shared between neurons and are no longer local, CNN are not considered as biologically plausible.

2.2.6 Recurrent Neural Networks



(a) RNN


...
$\boldsymbol{U}^{(l)}$
$\boldsymbol{y}^{(l)}(t-1)$
$\boldsymbol{U}^{(l)}$
$\boldsymbol{y}^{(l)}(t)$
$\boldsymbol{U}^{(l)}$
$\boldsymbol{y}^{(l)}(t+1)$
$\boldsymbol{U}^{(l)}$
...
$\boldsymbol{W}^{(l)}$
$\boldsymbol{W}^{(l)}$
$\boldsymbol{W}^{(l)}$
$\boldsymbol{y}^{(l-1)}(t-1)$
$\boldsymbol{y}^{(l-1)}(t)$
$\boldsymbol{y}^{(l-1)}(t+1)$

(b) Unfolded RNN

Figure 12: Computational graph of a recurrent layer in a compressed form (Figure 12a) and unfolded through time (Figure 12b). At each time step, neurons receive inputs from upstream layers as well as the states of neurons from the previous time step.

Both standard DNNs and CNNs are successful in processing static data such as images. However, they are incapable of efficiently processing sequences of data such as text, sound, or videos (Lipton, Berkowitz and Elkan 2015; Yu et al. 2019) – at least in their standard forms. In contrast, *Recurrent Neural Networks* (RNNs) are cyclic architectures that are specifically designed to process temporal sequences.

RNNs are composed of standard artificial neurons augmented with an internal state (i.e. memory), introducing a discrete notion of time (Lipton, Berkowitz and Elkan 2015; Yu et al. 2019). Formally, the activations $\boldsymbol{y}^{(l)}(t)$ of the recurrent layer l at time step t is:

$$\boldsymbol{y}^{(l)}(t) = \sigma\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)}(t) + \boldsymbol{U}^{(l)}\boldsymbol{y}^{(l)}(t-1) + \boldsymbol{b}^{(l)}\right) \tag{15}$$

where $\boldsymbol{U}^{(l)} \in \mathbb{R}^{n_l \times n_l}$ are the recurrent connections. Here, both the inputs $\boldsymbol{y}^{(l-1)}(t)$ at the current time and the activations $\boldsymbol{y}^{(l)}(t-1)$ of the layer at the previous time step are taken into account in the computation, giving recurrent neurons the capability of memorizing information from the past. See Figure 12 for a visual representation of a recurrent layer.

When trained using BP, the dynamics of RNNs are typically unfolded through time (see Figure 12b). Therefore, errors flow backward both spatially (i.e. from post-synaptic neurons to pre-synaptic neurons) and temporally (i.e. from the current time step to previous time steps). In this context, error backpropagation is often referred to as *Backpropagation Through Time* (BPTT) (Werbos 1990).

Despite their relative successes (Karpathy, Johnson and Fei-Fei 2015; Yu et al. 2019), standard recurrent neurons are only capable of handling short-term dependencies. As the time gap between the current time step and the relevant input increases, they face increasing difficulty in retaining information, making them less suitable for long-term memory tasks (Yu et al. 2019; Hochreiter and Schmidhuber 1997). Moreover, recurrent cells are particularly subject to the vanishing and exploding gradients problem due to their large temporal depth when trained with BPTT (Hochreiter 1991; Hochreiter and Schmidhuber 1997).

To address these issues, the *Long Short-Term Memory* (LSTM) (Hochreiter and Schmidhuber 1997) and the *Gated Recurrent Unit* (GRU) (Cho et al. 2014) cells have been proposed and successfully applied to many practical sequence learning tasks such as speech recognition (He and Droppo 2016; Graves, Jaitly and Mohamed 2013) and machine translation (Cho et al. 2014; Sutskever, Vinyals and Le 2014). To improve the memory capacity of RNNs, LSTM and GRU cells introduce specific *gates* that control what information should be stored, forgotten, and output (Hochreiter and Schmidhuber 1997; Cho et al. 2014; Lipton, Berkowitz and Elkan 2015; Yu et al. 2019). In essence, these gates help to mitigate the vanishing and exploding gradients problem by allowing neurons to propagate unchanged gradients that are not exponentially decayed (Hochreiter and Schmidhuber 1997).

2.2.7 Hardware Acceleration

DNNs are computationally demanding to evaluate, especially during training, due to the large numbers of *Multiply-Accumulate* (MAC) operations required by matrix multiplications. Furthermore, the sequential nature of multicore Central Processing Units (CPUs) makes them inherently inefficient for evaluating DNNs, leading to computational bottlenecks. Therefore, many massively parallel and dedicated hardware are used in practice to infer and train DNNs.

Graphical Processing Units

Graphical Processing Units (GPUs) have emerged as a reference hardware for deep learning (Shawahna, Sait and El-Maleh 2019). While originally designed for computer graphics applications, modern GPUs have been widely adopted in deep learning for their highly parallel architecture. Modern GPUs contain multiple parallel cores, each capable of executing thousands of parallel threads simultaneously (Bhattacharya 2021; Talib et al. 2020). For example, standard DNNs can be accelerated up to 10 times on GPUs when compared to CPUs (Steinkraus, Buck and Simard 2005; Buber and Diri 2018; Gyawali 2023). This large parallelization capability enables GPUs to efficiently handle the computational requirements of deep learning algorithms, making them the primary choice for training large-scale DNNs despite their significant energy consumption (Shawahna, Sait and El-Maleh 2019). Consequently, many deep learning frameworks and libraries have adopted GPUs as hardware acceleration (Abadi et al. 2015; Paszke et al. 2019).

Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are dynamic devices containing programmable arrays of logic gates as well as reconfigurable connections. Configurable using a *Hardware Description Language* such as VHDL or Verilog, they offer great degrees of parallelism, execution speed, and versatility, as required by DNNs (Shawahna, Sait and El-Maleh 2019; Bhattacharya 2021; Talib et al. 2020). Moreover, FPGAs benefit from lower energy consumption and lower cost than GPUs, making them particularly suitable for deep learning applications on low-powered devices (Shawahna, Sait and El-Maleh 2019; Bhattacharya 2021). However, the specialized expertise required for hardware design significantly limits the widespread adoption of FPGAs for deep learning.

Application-Specific Integrated Circuits

Often prototyped with FPGAs, *Application-Specific Integrated Circuits* (ASICs) are high-density integrated circuits for custom applications (Bhattacharya 2021). Due to their high degree of parallelism, low surface area, and low energy consumption, ASICs are particularly suitable for low-powered edge devices (Talib et al. 2020). However, the high costs associated with designing and manufacturing ASICs make them less accessible than FPGAs and GPUs. In addition, ASICs cannot be altered or improved after fabrication, further limiting their practicality (Talib et al. 2020).

Recently, Google has developed and released in the cloud the *Tensor Processor Unit* (TPUs), an ASIC processor dedicated to efficient machine learning computation (Jouppi et al. 2017). More specifically, TPUs are designed to perform high-speed MAC operations, making the inference and training of DNNs up to 10 times faster than GPUs (Wang, Wei and Brooks 2019).

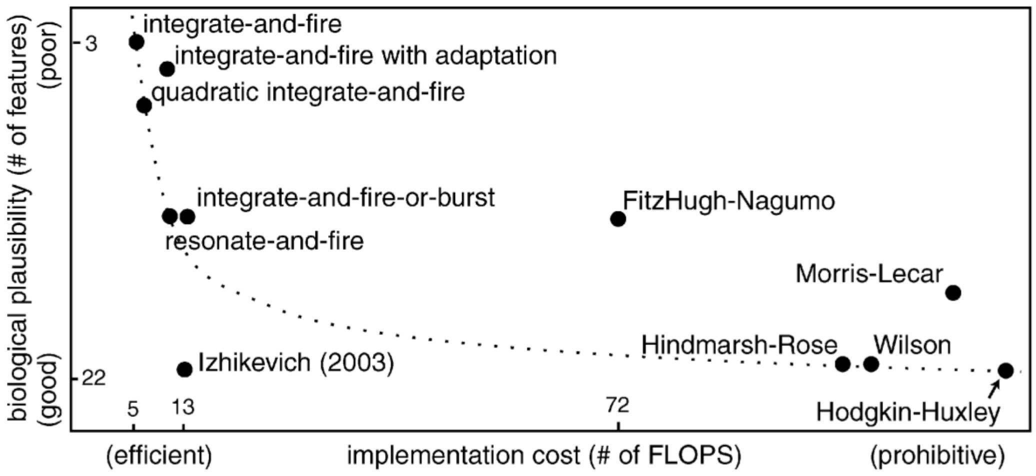


Figure 13: Comparison of different spiking neuron models according to their biological plausibility and computational complexity. In this figure ”# of FLOPS” refers to the numbers of floating point operations required to simulate 1ms with the model. Figure from (Izhikevich 2004).

2.3 Spiking Neural Networks

Unlike neurons used in DNNs, *spiking neurons* are biologically plausible neuron models that aim to capture the dynamics of real cortical neurons. Instead of processing information through abstracted rate codes, *Spiking Neural Networks* (SNNs) process information through precise sequences of asynchronous events over time called *spikes* (Gerstner and Kistler 2002; Brette 2015).

Originally designed for neuroscience purposes to accurately model the intricate dynamics of biological neurons (Hodgkin and Huxley 1952), SNNs recently gained attention and interest with the development of ultra low-powered *neuromorphic hardware*. Referred to as the third generation of artificial neural networks (Maass 1997), SNNs have also been proven to be universal approximators (Zhang and Zhou 2022) and are known to have at least the same computational power as DNNs (Maass 1997) making them attractive alternatives to DNNs. Therefore, the exploration of SNNs extends beyond modeling biological neurons and aims to exploit their distinct characteristics for energy-efficient and bio-inspired machine learning.

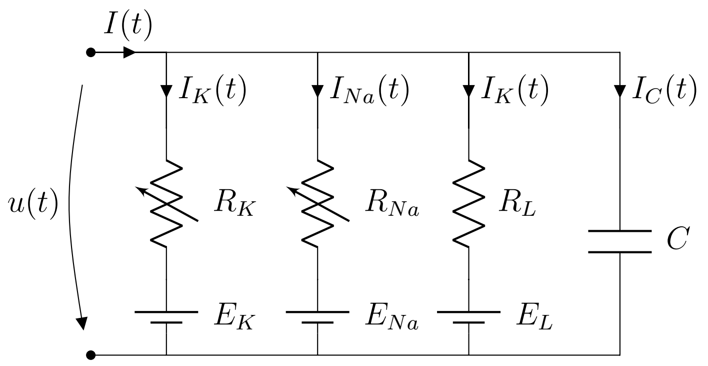


Figure 14: Schematic diagram of the circuit defining the Hodgkin-Huxley neuron model (Hodgkin and Huxley 1952). The membrane potential $u(t)$ is represented by the voltage across the capacitor C. Here, $R_{Na} = \frac{1}{g_{Na}}$, $R_K = \frac{1}{g_K}$ and $R_{Na} = \frac{1}{g_L}$ are the resistance values for the sodium, potassium, and leak channels respectively.

In this section, we review diverse spiking neuron models with varying levels of biological plausibility and computational complexity (see Figure 13 for a comparison) and discuss the software simulation and hardware acceleration of SNNs. Finally, we introduce different spike coding schemes and review both unsupervised and supervised learning algorithms.

2.3.1 Hodgkin–Huxley Model

Proposed in 1952 by Hodgkin and Huxley, the Hodgkin–Huxley (HH) neuron is a conductance-based model that represents the voltage-gated ion channels and leak channel of neurons (Hodgkin and Huxley 1952). It is essentially defined by the electrical circuit shown in Figure 14. Formally, by denoting by C the capacitance of the neuron, by g_{Na}, g_K and g_L the sodium, potassium and leak conductance and by E_{Na}, E_K and E_L their respective equilibrium potentials, the dynamic of the membrane potential of an HH neuron is defined by the follow system of Ordinary Differential Equations (ODEs) (Hodgkin and Huxley 1952):

$$C\frac{\mathrm{d}u(t)}{\mathrm{d}t} = -g_{Na}m(t)^3h(t)\left(u(t) - E_{Na}\right) - g_K n(t)^4\left(u(t) - E_K\right) - g_L\left(u(t) - E_L\right) + I(t) \quad (16)$$

where

$$\begin{aligned}\frac{\mathrm{d}n(t)}{\mathrm{d}t} &= \alpha_n(t)\,[1-n(t)] - \beta_n(t)\; n(t)\\ \frac{\mathrm{d}m(t)}{\mathrm{d}t} &= \alpha_m(t)\,[1-m(t)] - \beta_m(t)\; m(t)\\ \frac{\mathrm{d}h(t)}{\mathrm{d}t} &= \alpha_h(t)\,[1-h(t)] - \beta_h(t)\; h(t)\end{aligned} \tag{17}$$

and

$$\begin{aligned}\alpha_n(t) &= 0.01\frac{u(t)+10}{\exp\left(\frac{u(t)+10}{10}\right)-1}\\ \beta_n(t) &= 0.125\exp\left(\frac{u(t)}{80}\right)\\ \alpha_m(t) &= 0.1\frac{u(t)+25}{\exp\left(\frac{u(t)+25}{10}\right)-1}\\ \beta_m(t) &= 4\exp\left(\frac{u(t)}{18}\right)\\ \alpha_h(t) &= 0.07\exp\left(\frac{u(t)}{20}\right)\\ \beta_h(t) &= \frac{1}{\exp\left(\frac{u(t)+30}{10}\right)+1}\end{aligned} \tag{18}$$

Here, the time-dependent variables $n(t)$, $m(t)$, and $h(t)$ describe the opening and closing of the voltage-dependent channels. All functions in Equation 18 are defined as in the original HH model paper (Hodgkin and Huxley 1952). However, simpler variations of the model exist (Koslow and Subramaniam 2005).

The HH neuron is considered to be the most biologially plausible neuron model making it particularly relevant for investigating the dynamics of biological neurons (Izhikevich 2004). However, it is one of the most computationally expensive neurons to implement, with around 1200 Floating Point Operations (FLOPS) required for only 1 millisecond of simulation (Izhikevich 2004) – see Figure 13 for a comparison with other neuron models. Therefore, the HH model is too complex to be efficiently used in the context of machine learning.

2.3.2 Izhikevich Model

Proposed in 2003, the Izhikevich neuron model is a popular simplification of the Hodgkin–Huxley neuron that retains a high degree of biological plausibility while being computationally more efficient (see Figure 13) (Izhikevich 2003). Formally, it is defined by the following system of ODEs:

$$\begin{aligned} \frac{\mathrm{d}u(t)}{\mathrm{d}t} &= 0.04\ u(t)^2 + 5\ u(t) + 140 - v(t) + I(t) \\ \frac{\mathrm{d}v(t)}{\mathrm{d}t} &= a\,[b\ u(t) - v(t)] \end{aligned} \tag{19}$$

where $u(t)$ is the membrane potential in volt, $v(t)$ is the membrane recovery variable and $I(t)$ is the input current in in volt per second (Izhikevich 2003). When the membrane potental reaches 30 mV, both $u(t)$ and $v(t)$ are reset as follows:

$$\text{if } u(t) \geq 30 \text{ mV, then } \begin{cases} u(t) \leftarrow c \\ v(t) \leftarrow v(t) + d \end{cases} \tag{20}$$

Here, a describes the time scale of the recovery variable (a typical value is $a = 0.02$), b is the sensitivity of the recovery variable (a typical value is $b = 0.2$), c is the reset value of the membrane potential after a post-synaptic spike (a typical value is $c = -65\mathrm{mV}$) and d is the reset value of the recovery variable (a typical value is $d = 2$) (Izhikevich 2003).

By choosing different values of its parameters, the Izhikevich neuron model can exhibit various firing patterns (Izhikevich 2003). Moreover, it only requires 13 FLOPS per millisecond of simulation which is 92 times more computationally efficient than the HH neuron model (Izhikevich 2004). The Izhikevich model thus represents a good compromise between computational cost and biological plausibility.

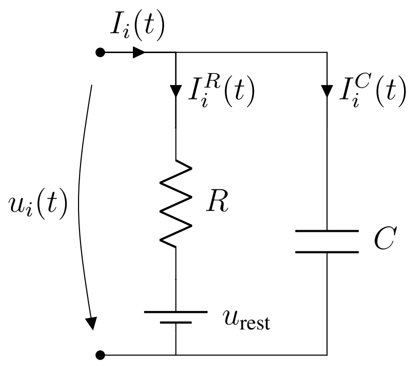


Figure 15: Schematic diagram of the parallel Resistor-Capacitor circuit that defines the membrane potential behaviour of the Leaky Integrate-and-Fire neuron model.

2.3.3 Leaky Integrate-and-Fire Model

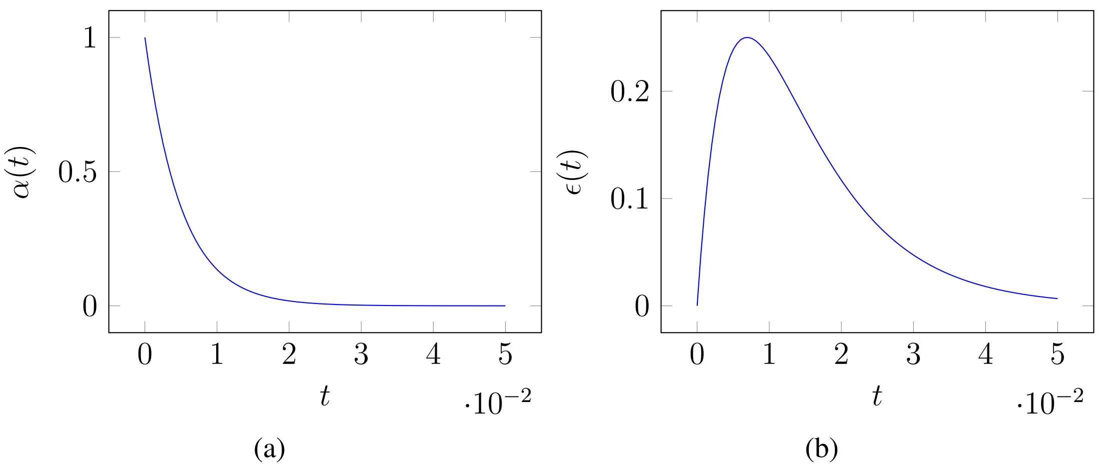


Figure 16: Illustration of the exponential α function (Figure 16a) and its corresponding PSP kernel ϵ (Figure 16b). Here, $\tau = 10\text{ms}$, $\tau_s = 5\text{ms}$ and $C = \frac{\tau\tau_s}{\tau-\tau_s} = 10\text{ms}$.

The *Leaky Integrate-and-Fire* (LIF) model is one of the least biologically plausible spiking neuron models — see Figure 13. However, due to its simplicity and computational efficiency, it is one of the most widely used models in computational neurosciences (Izhikevich 2004). It is electrically represented by a parallel *Resistor-Capacitor* circuit driven by a time-dependent input current $I(t)$ – see Figure 15. The membrane potential $u(t)$ of the LIF neuron is defined by the voltage across the capacitor C. When a current $I(t)$ is injected into the circuit, it charges the capacitor leading to an increase in voltage. When this input current vanishes, the capacitor C discharges through the parallel resistor R until its voltage $u(t)$ reaches some resting potential u_{rest}, provided by an optional battery (Gerstner et al. 2014).

By using the law of current conservation, the current $I_i(t)$ injected into the neuron i splits as follows:

$$I_i(t) = I_i^C(t) + I_i^R(t) \tag{21}$$

where

$$I_i^R(t) = \frac{u_i(t) - u_{\text{rest}}}{R} \tag{22}$$

is, according to the Ohm's law, the current flowing through the resistor R and

$$I_i^C(t) = C\,\frac{\mathrm{d}u_i(t)}{\mathrm{d}t} \tag{23}$$

is the current flowing through the capacitor C.

By combining Equations 21, 22 and 23, we find the following ODE describing the dynamic of the membrane potential $u_i(t)$ over time (Gerstner et al. 2014):

$$\tau \frac{\mathrm{d}u_i(t)}{\mathrm{d}t} = -\left[u_i(t) - u_{\text{rest}}\right] + R\,I_i(t) \tag{24}$$

where $\tau = RC$ is the membrane time constant of the neuron expressed in seconds.

Similarly to DNNs, SNNs are typically connected through synapses represented by weights. By denoting $\mathcal{P}_i = \{j \mid j \text{ is connected to } i\}$ as the set of pre-synaptic neuron indices connected to the post-synaptic neuron i and $\mathcal{T}_j = \left\{t_j^k \mid 1 < k < n_j\right\} = \{t \mid u_j(t) = \vartheta\}$ as the set of n_j spike timings fired by j, the synaptic current $I_i(t)$ injected into the post-synaptic neuron i is defined according to the spatio-temporal activity of its pre-synaptic events, such as:

$$I_i(t) := \sum_{j \in \mathcal{P}_i} w_{i,j} \sum_{t_j^k \in \mathcal{T}_j} \Theta\left(t - t_j^k\right) \alpha\left(t - t_j^k\right) \tag{25}$$

where

$$\Theta(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

is the Heavyside step function that only triggers events that occurred in the past, $w_{i,j} \in \mathbb{R}$ is the strength of the synaptic connection between the post-synaptic neuron i and the pre-synaptic neuron j, and α is a function that defines the temporal impact of input spikes onto the current. One common choice for α is the following low-pass filter:

$$\alpha(t) := A \exp\left(\frac{-t}{\tau_s}\right) \tag{27}$$

where τ_s is the synaptic time constant controlling the duration of the pre-synaptic spike integration and A is the current induced by the pre-synaptic spike, expressed in Amps. Figure 16 illustrates the shapes produced by Equations 27 and the membrane potential after receiving a pre-synaptic spike. The PSP in the LIF neuron is characterized by two phases: a short raising phase where the membrane potential increases rapidly, and a decay phase where the membrane potential slowly decreases.

Finally, when the value of the membrane potential $u_i(t)$ reaches a given threshold $\vartheta \in \mathbb{R}$ at time t, a post-synaptic spike at time is emitted at the output of the neuron and the membrane potential is reset to its resting potential, such as:

$$\text{if } u_i(t) \geq \vartheta, \text{ then } u_i(t) \leftarrow u_{\text{rest}} \tag{28}$$

With only 5 FLOPS required per millisecond of simulation, the LIF neuron is the most popular model for machine learning applications (Izhikevich 2004). Because of its simplicity and resemblance with the continuously-activated neurons of DNNs, it is widely implemented in hardware implementations of SNNs.

2.3.4 Spike Response Model

The *Spike Response Model* (SRM) is a generalization of the LIF neuron. Rather than making the membrane potential voltage-dependent, the SRM model directly depends on time, making derivations with the membrane potential easier than differential equations.

Formally, the membrane potential $u_i(t)$ of a SRM neuron i is defined as Gerstner and Kistler (2002):

$$\begin{aligned} u_i(t) = & u_{\text{rest}} + \sum_{j \in \mathcal{P}_i} w_{i,j} \sum_{t_j^z \in \mathcal{T}_j} \Theta\left(t - t_j^z\right) \epsilon(t - t_j^z) \\ & - \sum_{t_i^k \in \mathcal{T}_i} \Theta\left(t - t_i^k\right) \eta(t - t_i^k) \end{aligned} \tag{29}$$

where t is the current time, t_j^z is the k^{th} spike fired by the pre-synaptic neuron i, t_i^k is the k^{th} post-synaptic spike of the neuron i, η models the reset after firing, $\Delta_\vartheta = \vartheta - u_{\text{rest}}$ is the amplitude of the reset and ϵ is the PSP kernel describing the temporal response of the membrane potential to pre-synaptic spikes.

Because the LIF neuron is a special case of SRM, it is possible to map the former in the latter Gerstner and Kistler (2002). By integrating Equation 24 with the initial condition $u_i(0) = u_{\text{rest}}$, the following kernels are found Gerstner and Kistler (2002):

$$\begin{aligned} \epsilon(t) &= \frac{1}{C} \int_0^\infty \exp\left(\frac{-t'}{\tau}\right) \alpha\left(t - t'\right) \mathrm{d}t' \\ \eta(t) &= \Delta_\vartheta \exp\left(\frac{-t}{\tau}\right) \end{aligned} \tag{30}$$

where $\Delta_\vartheta = \vartheta - u_{\text{rest}}$.

Finally, by defining the α function as in Equation 27, the PSP kernel ϵ takes the following bi-exponential form:

$$\epsilon(t) = \frac{A}{C} \frac{\tau \tau_s}{\tau - \tau_s} \left[\exp\left(\frac{-t}{\tau}\right) - \exp\left(\frac{-t}{\tau_s}\right)\right] \tag{31}$$

See Figure 16b for a visualization of this function.

2.3.5 Simulations

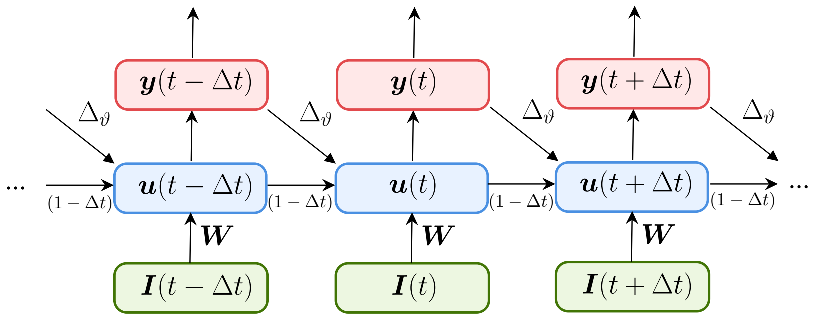

$\boldsymbol{y}(t-\Delta t)$
$\boldsymbol{y}(t)$
$\boldsymbol{y}(t+\Delta t)$
Δ_ϑ
$\boldsymbol{u}(t-\Delta t)$
$\boldsymbol{u}(t)$
$\boldsymbol{u}(t+\Delta t)$
$(1-\Delta t)$
$\boldsymbol{W}$
$\boldsymbol{I}(t-\Delta t)$
$\boldsymbol{I}(t)$
$\boldsymbol{I}(t+\Delta t)$
...

Figure 17: Computational graph of a clock-based simulation of neurons' membrane potentials. At each time step t, the membrane potential $u(t-\Delta t)$ at the previous time step $t-\Delta t$ is decayed, and the input current $I(t)$ is integrated. If an output spike $y(t-\Delta t)$ was produced at the previous time step, the membrane potential is reset using a negative current Δ_ϑ.

Due to their intrinsic temporal dimension, spiking neurons inherently require more computational resources compared to traditional DNNs. Furthermore, conventional Von Neumann computers are not explicitly designed to efficiently solve systems of ODEs. As a result, two main approaches have been developed to efficiently simulate SNNs, namely *clock-based* and *event-based* simulations (Brette et al. 2007).

Clock-based simulations are the most common approaches. These techniques involve discretizing the continuous dynamics of the membrane potential, typically using the Euler method for the numerical integration of ODEs. For example, the LIF neuron (see Equation 24) can be discretized to form the following recurrent function:

$$u_i(t) = (1-\Delta t)\ u_i(t-\Delta t) + \frac{1}{C}\,\Delta t\ I_i(t) - \Delta_\vartheta\ y(t-\Delta t) \tag{32}$$

where $\Delta t > 0$ is the size of the time step and

$$y(t) = \begin{cases} 1 & \text{if } u(t) \geq \vartheta \\ 0 & \text{otherwise} \end{cases} \tag{33}$$

is the threshold function. Figure 17 illustrates the recurrent computation of the discrete LIF neuron.

By dividing the dynamic into a finite number of time steps, the resulting discrete membrane potential takes a recurrent form that resembles the recurrent cells of RNNs — see Section 2.2.6 — making clock-based simulations relatively simple to implement. However, clock-based neurons suffer from inaccuracies due to the local truncation error of the Euler method (Butcher 2016), especially if the time step is chosen to be large. In addition, spike timings are inherently constrained to a grid of discrete times, thus losing in temporal precision (Mo and Tao 2022). To make clock-based simulations more accurate, the size of the time step can thus be chosen small, but at the cost of an increased number of time steps and, consequently, an increase in computational requirements (Brette et al. 2007; Mo and Tao 2022). Moreover, discrete simulations update the states of neurons even when no event occurs. Because SNNs are highly sparse, this leads to many unnecessary operations that are performed during the simulation.

In contrast, *event-based* simulations are asynchronous algorithms developed for precise evaluations of SNNs (Brette et al. 2007; Mo and Tao 2022). These methods typically iterate through the ordered set of spikes, update the states of neurons at each event, and calculate the precise timings of post-synaptic spikes (Brette et al. 2007). Event-based simulations offer numerous advantages such as exact spike timing computations and efficient processing of sparse data. However, implementing event-based simulations can be more complex than clock-based approaches. Moreover, the time complexity of event-based simulations scales with the number of events received and emitted, which can result in slower execution for SNNs with complex connections (Brette et al. 2007).

Therefore, while event-based simulations provide precise temporal computations and efficient processing for sparse SNNs, they may not always be the optimal choice. The selection of simulation methods should consider the specific requirements, network structure, and tradeoffs between accuracy and computational efficiency for the given model.

2.3.6 Hardware Acceleration

Because of their increased complexity, SNNs are inherently more computationally demanding than DNNs. Therefore, specific hardware can be used to take advantage of the parallel aspect of SNNs and accelerate their simulations and training.

Graphical Processing Units

Similarly to DNNs, SNNs can benefit from the massively parallel aspect of GPUs for fast evaluations (Brette and Goodman 2012). In particular, discretized SNNs are easily implementable as RNNs using standard deep learning libraries — for instance, see the PyTorch (Paszke et al. 2019) implementation of SLAYER (Shrestha and Orchard 2018). GPUs thus offer enough flexibility to experiment with new spiking neuron models, simulation techniques, and training algorithms. However, as SNNs are computationally more expensive than DNNs, their implementations on GPUs usually consume considerable amounts of power when compared to other alternative hardware.

Field-Programmable Gate and Analog Arrays

FPGAs are also popular approaches for low-powered hardware acceleration of spiking neurons. Many FPGA implementations of SNNs and training algorithms have been proposed (Neil and Liu 2014; Fang et al. 2019; Lee et al. 2020; Pham et al. 2021), sometimes leading to a 10 times speedup and up to 200 times gain in energy consumption when compared to GPUs (Fang et al. 2019). Alternatively, *Field Programmable Analog Arrays* (FPAAs) can be used for SNN simulations (Rocke et al. 2008). Due to their analog aspect, FPAAs are naturally capable of solving complex systems of ODEs at high speed and with lower power consumption than FPGAs (Hasler 2020), making them particularly suitable for SNNs. However, FPAAs remain relatively uncommon, primarily due to their inherent complexity and the limited number of manufacturers producing them.

Neuromorphic Hardware

SNNs can be implemented in non-Von Neumann, massively parallel, and scalable dedicated ASICs known as *neuromorphic hardware*. These ultra low-powered hardware can either be digital, like Intel's Loihi (Davies et al. 2018) and its newer version Loihi 2 (Frady et al. 2022), IBM Truenorth (Akopyan et al. 2015), SpiNNaker (Furber et al. 2014), and SpiNNaker 2 (Höppner et al. 2022) from the University of Manchester, or analog/mixed-signal, such as BrainScaleS-1 (Schemmel et al. 2010) and BrainScaleS-2 (Pehle et al. 2022) developed by a European consortium. Neuromorphic chips are composed of several neurocores, each containing a certain amount of addressable neurons and synapses that receive and transmit spikes through the *Address Event Representation* (AER) protocol (Sivilotti 1991). More importantly, neuromorphic hardware are event-driven. They update the states of neurons only when spikes are received, and collocate processing elements with memory to mitigate memory bottlenecks and reduce costly data movements inherent to Von Neumann architectures (Bouvier et al. 2019; Schuman et al. 2022). For example, it has been shown that neuromorphic hardware are capable of being more than 1000x more energy-efficient than GPUs on certain tasks (Rao et al. 2022). Due to the simplicity of some models (e.g. see the LIF neuron in Section 2.3.3), the resources necessary to build spiking neurons are reduced to adders for spike integration, comparators for thresholds, and a local memory for storing the membrane potentials (Bouvier et al. 2019). Moreover, recent neuromorphic hardware implement new features to enhance the computational power of SNNs. For example, in Loihi 2 Frady et al. (2022) or SpiNNaker Furber et al. (2014); Höppner et al. (2022), spikes are not only restricted to binary values but can also carry an additional 32-bit payload. Referred to as *graded spikes*, they can be used to encode the amplitude of spikes and increase the amount of transmitted information or for learning purposes. Therefore, by combining event-based processing with high parallelism, locality, and scalability, neuromorphic hardware represent promising alternatives for low-powered machine learning.

2.3.7 Spike Coding

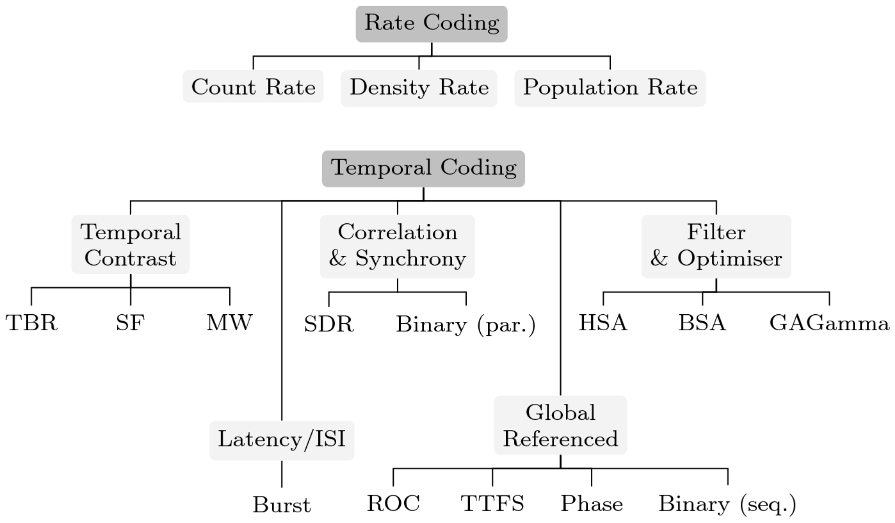


Figure 18: Different rate and temporal spike coding schemes. Figure from (Auge et al. 2021).

Unlike DNNs, SNNs receive spike trains as inputs instead of real numbers. However, many real-world data are represented by real numbers and thus required to be converted into spikes before being used for inference. SNNs also produce spike trains at their outputs which require to be converted back to real numbers for post-processing. Like the brain — see Section 2.1.2 for details —, a large variety of spike coding schemes are available — see Figure 18 — and many are better suited for some types of data than others (Auge et al. 2021). All spike coding schemes for SNNs can be classified into two main categories: rate coding and temporal coding schemes. In the following, we describe two of the most popular spike coding schemes used in SNNs, namely *Spike Count* and *Time-to-First-Spike* coding.

Spike Count Coding

In *Spike Count* coding, real-valued information are proportionally encoded by the number of spikes fired by input neurons. Formally, spikes can be generated at regular intervals inversely proportional to the input firing rate:

$$\Delta t = \frac{1}{f} \tag{34}$$

Here, Δt is the Inter-Spike Interval (ISI) and f is the firing rate of the input neuron, calculated as:

$$f = f_{\min} + x\ g \tag{35}$$

where $x \in [0, 1]$ is the normalized input signal and $f_{\min} > 0$ is the minimum firing rate and $g > 0$ is a gain scaling the firing rate.

Alternatively, input spikes can be generated following a homogeneous Poisson random process (Heeger et al. 2000). In this case, the probability density function for the ISI Δt is (Heeger et al. 2000):

$$p\left(\Delta t\right) = f e^{-f\Delta t} \tag{36}$$

For decoding, the output firing rate of a neuron can be converted to a positive real value as follows:

$$\begin{aligned} y &= y_{\min} + f\ g \\ &= y_{\min} + \frac{n}{T}\ g \end{aligned} \tag{37}$$

where y is the decoded value, f is the firing rate of the output neuron, $y_{\min}$ is a bias, n is the number of spikes fired by the neuron, T is the simulation duration and g is a gain scaling the decoded value.

Due to its redundance of spikes, rate coding has the advantage of being robust to noise (Guo et al. 2021). However, processing rate codes with spiking neurons requires the integration of several spikes over time which increases the inference time of SNNs (Guo et al. 2021).

Time-To-First-Spike Coding

Time-To-First-Spike (TTFS) coding is one of the most popular temporal coding schemes in SNN. In TTFS, real values are encoded by the timing of spikes, with early spikes representing higher values than late spikes.

Formally, the spike time t encoding the normalized real value $x \in [0, 1]$ can be computed as:

$$t = T\,(1 - x) \tag{38}$$

where T is the duration of the encoding window.

For decoding, the output spike time t can be converted to a real value y as follows:

$$y = y_{\max} - t\;g \tag{39}$$

where $y_{\max}$ is the maximum decoded value and $g > 0$ is a gain. Note that, in this equation, high values of t results in low values of y.

TTFS benefits from several advantages when compared to spike count coding. For example, it is known to be particularly fast to process as most of the information is carried by early spikes. Moreover, because TTFS encodes information with a single spike, it has the advantage of producing sparse and energy-efficient (Guo et al. 2021). Finally, from a pure information point of view, temporal coding schemes such as TTFS are capable of encoding more bits of information than rate codes (Thorpe, Delorme and Van Rullen 2001b). However, this single spike emitted by TTFS makes the coded information sensitive to noise as even a small perturbation in spike timing can drastically change the encoded value (Guo et al. 2021; Park, Lee and Yoon 2021).

2.3.8 Unsupervised Learning

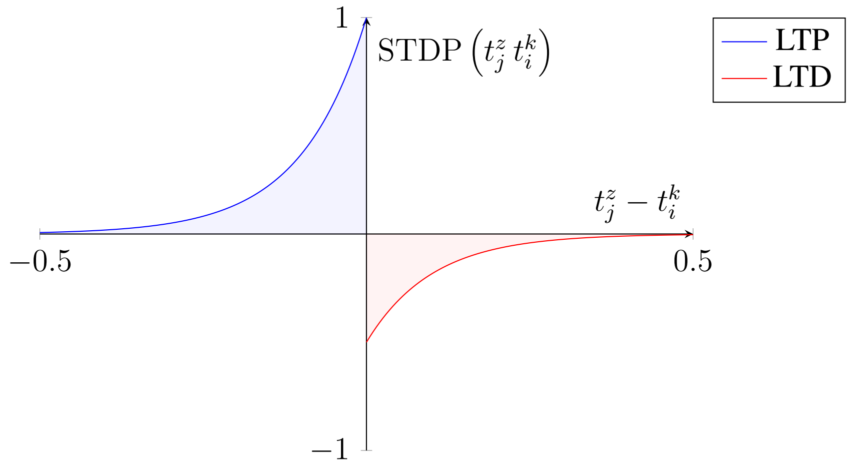


Figure 19: Illustration of STDP according to the relative temporal distance between a pre-synaptic spike fired at time t_j^z and a post-synaptic spike fired at time t_i^k. In this figure, $A_+ = 1$, $A_- = 0.5$ and $\tau_+ = \tau_- = 0.1$. The blue line corresponds to LTP, increasing the connection between the two neurons, and the red line corresponds to LTD, decreasing the connection.

Unsupervised Learning has emerged as a prominent approach to achieve biologically plausible learning in SNNs. Neurons trained with unsupervised learning rules aim to identify patterns from their inputs and learn meaningful representations of unlabeled data. The features learned through unsupervised learning can then be used as inputs to other tasks such as supervised classification (Diehl and Cook 2015; Deng and Li 2021; Lee et al. 2018) or simply be useful by themselves for data clustering (Smith 2020; Diamond, Schmuker and Nowotny 2019).

Spike Time-Dependent Plasticity (STDP) is by far the most popular approach for unsupervised learning in SNNs. Inspired by Hebb's postulate on synaptic strengthening as well as in-vivo observation of plasticity – see Section 2.1.3 for details – the STDP rule describes the changes of weight that occur at synapses according to the relative timings between pre-synaptic and post-synaptic spikes (Gerstner and Kistler 2002).

Formally, given the timing t_j^z of the z^{th} spike fired by the pre-synaptic neuron j and the timing t_i^k of the k^{th} spike fired by the post-synaptic neuron i, the change of weight caused by this pair of spike timings is defined by the following rule (Gerstner and Kistler 2002):

$$\mathrm{STDP}\left(t_j^z, t_i^k\right) = \begin{cases} A_+ \exp\left(\frac{t_i^k - t_j^z}{\tau_+}\right) & \text{if } t_j^z \leq t_i^k \\ -A_- \exp\left(\frac{t_j^z - t_i^k}{\tau_-}\right) & \text{otherwise} \end{cases} \tag{40}$$

where A_+ and A_- are two parameters controlling the strength of LTP and LTD respectively and τ_+ and τ_- are time constants controlling the size of the temporal window. See Figure 19 for a visual representation of this learning rule.

Using the rule defined in Equation 40, the total change of weight $\Delta w_{i,j}$ between the post-synaptic neuron i and the pre-synaptic neuron j is:

$$\Delta w_{i,j} = \sum_{t_j^z \in \mathcal{T}_j} \sum_{t_i^k \in \mathcal{T}_i} \mathrm{STDP}\left(t_j^z, t_i^k\right) \tag{41}$$

STDP reinforces or weakens connections based on the causal relationship between the timing of pre and post-synaptic spikes. However, STDP cannot be applied repeatedly in the same direction without making the network activity fall into complete silence — also referred to as the *dead neuron problem* — or into an epileptic state where all neurons fire at a high frequency due to overly large weights. Therefore, additional mechanisms such as bounds on weights, weight normalization, synaptic competition, or homeostatic control are often required to keep weights and firing rates within reasonable ranges (van Rossum, Bi and Turrigiano 2000; Diehl and Cook 2015; Deng and Li 2021; Lee et al. 2018; Dong et al. 2023; Mozafari et al. 2019).

Finally, due to its biological plausibility and its online aspect, many implementations of STDP have been proposed in neuromorphic hardware (Schemmel et al. 2010; Davies et al. 2018; Pehle et al. 2022), giving SNNs the capability of performing feature learning on the edge.

2.3.9 Supervised Learning

In contrast with unsupervised learning, *Supervised Learning* aims to learn a general mapping function between given inputs and desired outputs. However, the computation of gradients of SNNs is not as straightforward as it is with DNNs due to the non-differentiable nature of spikes. Therefore, supervised learning with SNNs is notably more challenging as in DNNs.

In this context, many supervised learning techniques have been proposed with different degrees of biological plausibility and compatibility with neuromorphic hardware.

Reward-Modulated Spike Time-Dependent Plasticity

The unsupervised STDP rule — see Section 2.3.8 — can be adapted into a three-factor learning rule that modulates the change of weights with reward signals (Mozafari et al. 2019; Juarez-Lora et al. 2022). Originally designed for biologically plausible reinforcement learning (Juarez-Lora et al. 2022), *Reward-Modulated STDP* (RSTDP) is also used to solve supervised learning tasks (Mozafari et al. 2019; Nobukawa, Nishimura and Yamanishi 2019).

Instead of directly changing the weights, RSTDP accumulates the STDP traces into an eligibility trace $e_{i,j}$ associated with the weight $w_{i,j}$, such as:

$$e_{i,j} = \sum_{t_j^z \in \mathcal{T}_j} \sum_{t_i^k \in \mathcal{T}_i} \mathrm{STDP}\left(t_j^z, t_i^k\right) \tag{42}$$

When a reward signal r_i is emitted, a change of weight is triggered by scaling the eligibility trace with the reward, such as:

$$\Delta w_{i,j} = r_i \; e_{i,j} \tag{43}$$

As STDP, RSTDP has the benefit of being compatible with neuromorphic hardware.

However, the RSTDP rule is often only applied to output neurons to perform classification as assigning rewards to hidden neurons is inherently more complex than output neurons. Therefore, SNNs trained with RSTDP strongly rely on unsupervised learning in their hidden layers, causing a significant performance gap with DNNs trained with BP (Mozafari et al. 2019; Nobukawa, Nishimura and Yamanishi 2019).

Surrogate Gradients

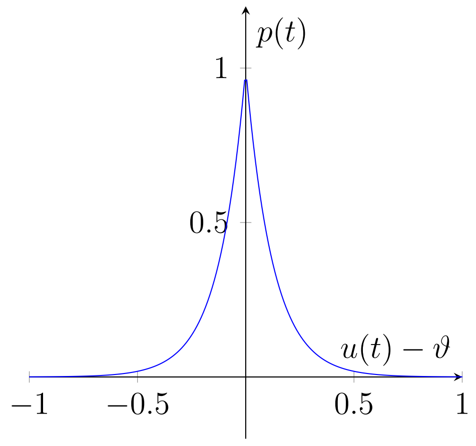


Figure 20: The spike escape rate function $p(t)$ defines the probability of state change (from spiking to non-spiking or non-spiking to spiking) as a function of the difference between the membrane potential and the threshold as a time t. Here $\alpha = 1$ and $\beta = 8$.

Many gradient-descent methods for SNNs adopt clock-based strategies where the discrete spatio-temporal dynamics of SNNs are unfolded into RNNs (Shrestha and Orchard 2018; Zheng et al. 2020; Wu et al. 2018) — see Section 2.2.6. Such methods use BPTT to backpropagate errors backward in time and use various surrogate derivatives to overcome the discontinuous aspect of spikes and spatially backpropagate errors (Shrestha and Orchard 2018; Zheng et al. 2020; Wu et al. 2018). For example, SLAYER (Shrestha and Orchard 2018) defines the surrogate derivative of the neuron output with respect to the internal membrane potential as the spike escape rate function $p(t)$, such as:

$$p(t) = \frac{1}{\alpha} \exp\left(-\beta \left|u(t) - \vartheta\right|\right) \tag{44}$$

where $\alpha > 0$ and $\beta > 0$ are parameters respectively controlling the amplitude and the decay rate of the function. See Figure 20 for an illustration of this function. This function represents the probability density function for the state change of the neuron — i.e. from spiking to non-spiking or non-spiking to spiking – at a given time. It increases when the membrane potential $u(t)$ is near the threshold ϑ as a slight disturbance in the membrane potential is likely to induce a state change. However, the function decreases as the membrane potential deviates further from the threshold, as it would require larger perturbations to trigger a change of state (Shrestha and Orchard 2018).

Surrogate gradients can train SNNs with performance comparable to DNNs (Shrestha and Orchard 2018; Zheng et al. 2020; Wu et al. 2018). Moreover, they are less prone to the dead neuron problem since spiking activity is not necessary to obtain a gradient. However, there are certain drawbacks associated with their use. First, surrogate gradients require unfolding network dynamics into discrete RNNs, leading to redundant and memory-intensive computations. This inefficiency arises because errors are backpropagated even in the absence of spikes. Second, surrogate methods use BPTT to backpropagate errors backward through space and time. Therefore, they cannot be directly implemented in neuromorphic systems as they do not permit backward computation. Instead, they can be used in the loop, where membrane potentials are recorded at regular intervals during the inference on neuromorphic hardware, and gradients are computed offline on a traditional von Neumann computer (Cramer et al. 2022b). However, when it is supported by the neuromorphic hardware, recording membrane potentials significantly increases the amount of data movements to off-chip DRAM memory leading to increased energy consumption (Christensen et al. 2022).

Exact Event-Based Gradients via Implicit Functions

In contrast with clock-based surrogate gradients, event-based methods compute gradients by propagating errors from spike timing to spike timing Bohté, Kok and Poutré (2000); Jin, Zhang and Li (2018); Göltz et al. (2021); Wunderlich and Pehle (2021).

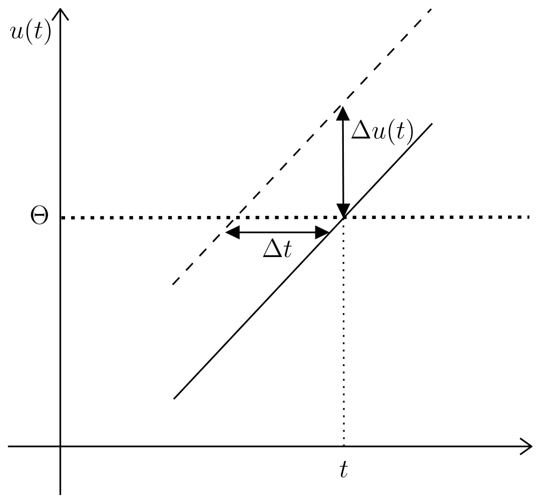


Figure 21: The linear relationship between the spike timing and the membrane potential assumed in SpikeProp (Bohté, Kok and Poutré 2000). Here, $u(t)$ is the membrane potential, t the spike time, Θ the threshold, $\Delta u(t)$ the change in membrane potential and Δt the change in spike timing. The solid and dashed lines respectively represent the membrane potential before and after applying the perturbation.

Due to their event-based aspect, these methods only require spike timings to compute gradients, thus requiring less memory and making them particularly suitable for in-the-loop training with neuromorphic hardware (Göltz et al. 2021). However, the lack of a closed-form solution for the spike timings represents a major obstacle to the computation of explicit gradients in SNNs Göltz et al. (2021); Wunderlich and Pehle (2021). Nevertheless, the *Implicit Function Theorem* (Margossian and Betancourt 2022) — see Theorem 1 — allows for an implicit expression of the gradient.

Theorem 1 (Implicit Function Theorem (Margossian and Betancourt 2022))**.** *Let $F(x, y)$ be a differentiable function in the neighborhood of $(x_0, y_0) \in R^2$. We suppose that $F(x, y) = 0$. If $\frac{\partial F(x,y)}{\partial y} \neq 0$,then there exists a function $g(x)$ defined on an interval I containing x_0 such as $F(x, g(x)) = 0$, $g(x) = y$, and*

$$\frac{\partial g(x)}{\partial x} = -\frac{\frac{\partial F(x,y)}{\partial x}}{\frac{\partial F(x,y)}{\partial y}} \tag{45}$$

By application of Theorem 1, the gradient of spike timings can implicitly be computed without knowing the true relationship between the membrane potential and the spike time, such as:

$$\frac{\partial t_i^k}{\partial w_{i,j}} = -\frac{1}{\frac{\partial u_i\left(t_i^k\right)}{\partial t_i^k}} \frac{\partial u_i\left(t_i^k\right)}{\partial w_{i,j}} \tag{46}$$

The implicit gradient defined in Equation 46 was first proposed in SpikeProp (Bohté, Kok and Poutré 2000) and its variants (Schrauwen and Van Campenhout 2004; Booij and tat Nguyen 2005; McKennoch, Liu and Bushnell 2006; Ghosh-Dastidar and Adeli 2009). They assumed that, for an infinitesimal region ϵ around the post-synaptic spike, the membrane potential could be approximated by a linear function of time — see Figure 21 for a visual representation of this linear relationship. However, recent works (Wunderlich and Pehle 2021; Lee, Haghighatshoar and Karbasi 2023) have demonstrated that Equation 46 is not an approximation but is an exact gradient of spike timings that can be evaluated by adjoint dynamical systems during the backward pass (Wunderlich and Pehle 2021) or in an event-based manner (Zhang and Li 2020; Lee, Haghighatshoar and Karbasi 2023).

Exact Event-Based Gradients via Time Constants Constraints

Instead of implicitly computing gradients, recent works have successfully derived closed-form solutions of spike timings by constraining the time constants of the LIF neuron (Mostafa 2016; Comsa et al. 2020; Göltz et al. 2021). The method, named Fast & Deep, proposes three different constraints to isolate analytical expressions of spikes and compute exact gradients.

The first constraint consists of making the membrane time constant $\tau \to \infty$ tend towards infinity (Göltz et al. 2021). By doing so, the first term of the PSP kernel ϵ — see Equation 31 — tends towards one and the function reduces to PSP kernel of the non-Leaky Integrate and Fire (nLIF) neuron:

$$\epsilon(t) = \frac{A}{C}\tau_s \left[1 - \exp\left(\frac{-t}{\tau_s}\right)\right] \tag{47}$$

which can easily be solved for post-synaptic spikes (Mostafa 2016; Göltz et al. 2021).

The second constraint sets the membrane potential time constant $\tau = \tau_s$ equal to the synaptic time constant τ_s (Göltz et al. 2021). According to Hôpital's rule, in the limit of $\tau \to \tau_s$, the PSP kernel of the LIF neuron reduces to the following alpha function (Comsa et al. 2020; Göltz et al. 2021):

$$\epsilon(t) = \frac{A}{C} t \exp\left(\frac{-t}{\tau}\right) \tag{48}$$

By adopting this PSP kernel, a differentiable closed-form solution for the spike timing can be found by solving the equation $u(t) = \vartheta$ using the main branch of the Lambert W function (Corless et al. 1996; Comsa et al. 2020; Göltz et al. 2021).

Finally, the last constraint consists of setting the membrane time constant $\tau = 2\tau_s$ as twice the synaptic time constant τ_s (Göltz et al. 2021). By using such constraint, we can show that:

$$\exp\left(\frac{-t}{\tau_s}\right) = \exp\left(\frac{-2t}{2\tau_s}\right) = \exp\left(\frac{-2t}{\tau}\right) = \exp\left(\frac{-t}{\tau}\right)^2 \tag{49}$$

meaning that the PSP kernel ϵ reduces to a second degree polynomial:

$$\epsilon(t) = \frac{A}{C} \frac{\tau\tau_s}{\tau - \tau_s} \left[\exp\left(\frac{-t}{\tau}\right) - \exp\left(\frac{-t}{\tau}\right)^2\right] \tag{50}$$

which can be solved using the quadratic formula (Göltz et al. 2021).

These three constraints have mainly been proposed in the context of temporally-coded SNNs (Mostafa 2016; Comsa et al. 2020; Göltz et al. 2021) and have demonstrated promising performances on simple benchmark datasets. Moreover, the existence of analytical solutions for the post-synaptic spikes makes these methods particularly suitable for efficient simulations of SNNs as no numerical solver is required. Finally, their event-based nature also makes them suitable for in-the-loop training where only spike timings are available (Göltz et al. 2021). However, they cannot be directly implemented on neuromorphic hardware due to the non-local and reverse aspects of BP.

2.4 Gradient-Based Optimization

Gradient-based algorithms are the most popular methods to perform optimization of differentiable functions and, by far, the most common training techniques for artificial neural networks (Ruder 2017). In this section, we introduce the Gradient Descent algorithm and some of its variants that are used to update the weights of neural networks.

2.4.1 Gradient Descent

The *Gradient Descent* (GD) algorithm — also referred to as *Batch Gradient Descent* or *Vanilla Gradient Descent* – is the simplest form of first-order optimization algorithms.

Let $f(\boldsymbol{x}, \boldsymbol{y}) : \mathbb{R}^m \rightarrow \mathbb{R}$ be a differentiable parametrized function with parameters $\boldsymbol{\theta} \in \mathbb{R}^n$, inputs $\boldsymbol{x}$ and target $\boldsymbol{y}$ from a dataset $\mathcal{D} = \{(\boldsymbol{x_i}, \boldsymbol{y_i})\}_{i=1}^{n_{\text{data}}}$ of size n_{data}. We also denote by $\boldsymbol{\nabla\theta} = \left[\frac{\partial f(\boldsymbol{x},\boldsymbol{y})}{\partial \theta_1}, \frac{\partial f(\boldsymbol{x},\boldsymbol{y})}{\partial \theta_2}, \cdots, \frac{\partial f(\boldsymbol{x},\boldsymbol{y})}{\partial \theta_n}\right]$ the gradient of $f(\boldsymbol{x}, \boldsymbol{y})$, defined by all the first order derivatives of the function with respect to its parameters. In GD, parameters are iteratively updated by taking steps in the opposite direction of the expected gradient (Ruder 2017), such as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \underset{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{D}}{\mathbb{E}} [\boldsymbol{\nabla\theta}] \tag{51}$$

where $\eta \in \mathbb{R}_+$ is a learning rate controlling the step size of the parameters update.

However, evaluating the expected gradient $\underset{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{D}}{\mathbb{E}} [\boldsymbol{\nabla\theta}]$ at every step of GD is computationally expensive, particularly if the size of the dataset $\mathcal{D}$ is large (Ruder 2017).

2.4.2 Stochastic Gradient Descent

To reduce the computational cost of GD, parameters can be updated in a stochastic manner by choosing a random sample from the dataset and taking a step in the opposite direction of the gradient (Ruder 2017). Thus referred to as *Stochastic Gradient Descent* (SGD), weights are updated as follows:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \, \boldsymbol{\nabla\theta} \tag{52}$$

Note that, in contrast to GD, SGD does not average gradients. In addition to being more computationally tractable, SGD thus benefits from a greater convergence rate than GD (Bubeck 2015). However, while SGD is guaranteed to converge with sufficiently small learning rates (Robbins and Monro 1951), theoretical analyses have shown that the convergence rate of SGD depends on the variance of the estimates (Bubeck 2015; Bottou, Curtis and Nocedal 2018; Murata 1999; Moulines and Bach 2011; Needell, Srebro and Ward 2016; Chee and Toulis 2018; Gower et al. 2019; Faghri et al. 2020). Gradients exhibiting low variance have less variability and are more consistent, leading to better convergence than those with high variance. Moreover, another limitation of SGD is that it has difficulties converging when the surface of the function curves more sharply in some dimensions than others (i.e. *ravines*) (Sutton 1986; Ruder 2017). This often results in strong oscillation in the parameter space and, consequently, a slow convergence towards the local optimum (Sutton 1986; Ruder 2017).

2.4.3 Mini-Batch Gradient Descent

To reduce the impact of gradient variance on the rate of convergence while retaining the computational tractability of SGD, an expected gradient can be computed over a random *mini batch* of samples $\mathcal{B} \subset \mathcal{D}$, such as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \underset{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{B}}{\mathbb{E}} [\boldsymbol{\nabla\theta}] \tag{53}$$

Thus referred to as *Mini-Batch Gradient Descent* (Mini Batch GD), this method combines the computational advantages of SGD with the stability of GD (Ruder 2017) and typically benefits from better convergence than SGD (Bubeck 2015).

2.4.4 Stochastic Gradient Descent with Momentum

Inspired by Newton's laws of motion, *Stochastic Gradient Descent with Momentum* (SGD-M) adds *momentum* to the gradient to reduce the oscillations in SGD (Qian 1999).

Formally, SGD-M is performed by introducing a first-order moment $\boldsymbol{m}$ that computes an exponential average of the gradient over the previous updates, such as:

$$\begin{aligned} \boldsymbol{m} &\leftarrow \alpha\, \boldsymbol{m} + \boldsymbol{\nabla\theta} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta\, \boldsymbol{m} \end{aligned} \tag{54}$$

where $0 < \alpha < 1$ is the friction coefficient, typically set to 0.9.

Intuitively, the moment term of SGD-M increases in dimensions with consistent directions and reduces in oscillating dimensions (Ruder 2017). By reducing oscillations and accelerating the gradient, SGD-M benefits from a greater convergence rate than SGD (Bubeck 2015) but introduces a bias in the weight updates (Defazio 2020).

2.4.5 RMSProp

Root Mean Squared Propagation (RMSProp) is a gradient-based optimization technique that introduces adaptive learning rates for each parameter (Ruder 2017).

Formally, RMSProp computes a moving average of the squared gradient and scales the gradient by the square root of this moving average, such as:

$$\begin{aligned} v_i &\leftarrow \beta\, v_i + (1-\beta)\, \nabla\theta_i^2 \\ \theta_i &\leftarrow \theta_i - \frac{\eta}{\sqrt{v_i + \epsilon}}\, \nabla\theta_i \end{aligned} \tag{55}$$

where $0 < \beta < 1$ is the friction coefficient and ϵ is a smoothing term used to avoid divisions by zero, typically set to 10^{-8}.

By scaling individual learning rates by the root mean squared gradient, RMSProp essentially ensures that each parameter is updated with a similar magnitude, making the method invariant to the scaling of gradients (Ruder 2017).

2.4.6 Adam

The *Adaptive Moment Estimation* (Adam) is a popular method for training neural networks that combines the benefits of both SGD-M and RMSProp (Kingma and Ba 2017). In addition to computing the first-order moment of the gradient, Adam also stores an exponentially decaying average of the gradient magnitude, as in RMSProp. Moreover, the author observed that both the first and second moments of the gradient are biased toward zero during the initial steps. Therefore, they introduce a bias correction for each moment to counteract the effect of initialization biases. Finally, weights are updated by following the direction of the first moment of the gradient and scaling the learning rate by the estimated magnitude, such as:

$$\begin{aligned} m_i &\leftarrow \beta_1\, m_i + (1-\beta_1)\, \nabla\theta_i \\ v_i &\leftarrow \beta_2\, v_i + (1-\beta_2)\, \nabla\theta_i^2 \\ \widehat{m}_i &= \frac{m_i}{1-\beta_1^t} \\ \widehat{v}_i &= \frac{v_i}{1-\beta_2^t} \\ \theta_i &\leftarrow \theta_i - \frac{\eta}{\sqrt{\widehat{v}_i}+\epsilon}\widehat{m}_i \end{aligned} \tag{56}$$

where m_i and v_i are respetively the first and second moment of the gradient, β_1 and β_2 are their respective decay factors, $\widehat{m}_i$ and $\widehat{v}_i$ are the bias-corrected first and second moment of the gradient, t is the current update step, and ϵ is the smoothing term used to avoid divisions by zero. The author also proposed default values of $\beta_1 = 0.9$, $\beta_2 =$

0.999 and $\epsilon = 10^{-8}$ (Kingma and Ba 2017).

2.5 Automatic Differentiation

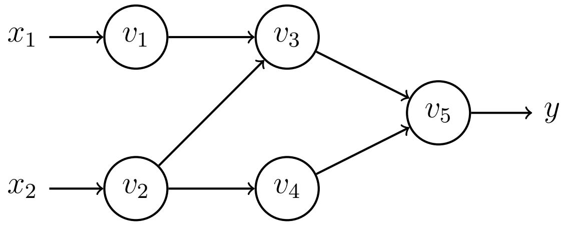


Figure 22: Computational graph of $y = f(x_1, x_2) = x_1x_2 - \ln(x_2)$, inspired from Baydin et al. (2017). See the Forward Primal Trace column in Table 1 for the definitions of the intermediate variables.

Automatic Differentiation (AD) is an ensemble of techniques used to efficiently and accurately evaluate derivatives of functions. Widely used in machine learning for gradient computation, the basic idea of AD is to represent complex functions, such a neural networks, into sequences of simple arithmetic operations (such as addition, subtraction, multiplication, and division) and elementary functions (Margossian 2019). This allows efficient and accurate computations of derivatives by repeatedly applying the chain rule to these simple operations.

We take as an example the following function of two inputs x_1 and x_2:

$$y = f(x_1, x_2) = x_1x_2 - \ln(x_2) \tag{57}$$

This function can be represented as a computational graph of elementary operations (Bauer 1974), as illustrated by Figure 22. This computational graph is evaluated by a *Forward Primal Trace* or *Evaluation Trace*, represented by a list of elementary operations called a *Wengert List* (Wengert 1964). The left column of Table 1 shows how Equation 57 is represented as such.

To compute derivatives of y with respect to x_1 and x_2, two main approaches can be used: Reverse-Mode AD and Forward-Mode AD.

2.5.1 Reverse-Mode Automatic Differentiation

Table 1: Reverse-Mode Automatic Differentiation of $y = f(x_1, x_2) = x_1x_2 - \ln(x_2)$ evaluated at $x_1 = 7$ and $x_2 = 3$. The arrows on the sides of the table indicate the order of evaluation. Table inspired from (Baydin et al. 2017).

Forward Primal Trace $\Downarrow$		Reverse Adjoint Trace $\Uparrow$	
$v_1 = x_1$	$= 7$	$x'_1 = v'_1$	$= 3$
$v_2 = x_2$	$= 3$	$x'_2 = v'_2$	$= 7.333$
$v_3 = v_1 \times v_2$	$= 7 \times 3$	$v'_1 = v'_3 \frac{\partial v_3}{\partial v_1}$	$= v_2 = 3$
		$v'_2 = v'_2 + v'_3 \frac{\partial v_3}{\partial v_2}$	$= 0.333 + v_1 = 7.333$
$v_4 = \ln(v_2)$	$= \ln(3)$	$v'_2 = v'_4 \frac{\partial v_4}{\partial v_2}$	$= \frac{1}{v_2} = 0.333$
$v_5 = v_3 + v_4$	$= 21 + 1.098$	$v'_3 = v'_5 \frac{\partial v_5}{\partial v_3}$	$= v'_5 \times 1 = 1$
		$v'_4 = v'_5 \frac{\partial v_5}{\partial v_4}$	$= v'_5 \times 1 = 1$
$y = v_5$	$= 22.098$	$v'_5 = y'$	$= 1$

Reverse-Mode AD — also known as the *Adjoint Method* — is the most widely used type of automatic differentiation in modern machine learning. It propagates derivatives backward through the computational graph, from outputs to inputs (Baydin et al. 2017; Margossian 2019). Reverse-Mode AD is therefore a two-phase process. First, the function is evaluated during the forward pass, saving intermediate variables v_i and recoding their dependencies. Then, derivatives are computed by propagating, in reverse, adjoint variables $v'_i = \frac{\partial y_j}{\partial v_i}$ (Baydin et al. 2017). See Table 1 for an example of Reverse-Mode AD applied to the function defined in Equation 57. Nowadays, Reverse-Mode AD is widely used to perform error backpropagation in neural networks.

2.5.2 Forward-Mode Automatic Differentiation

In *Forward-Mode AD*, each evaluation of the function $f(x_1, x_2)$ also computes a tangent trace with respect to a single input by applying the chain rule as the computational graph is evaluated. This can be performed by evaluating the function using *Dual Numbers* (Baydin et al. 2017). Formally, a dual number has the following form:

$$v + v'\epsilon \tag{58}$$

Table 2: Forward-Mode Automatic Differentiation of $y = f(x_1, x_2) = x_1x_2 - \ln(x_2)$ evaluated at $x_1 = 7$ and $x_2 = 3$. The arrows on the sides of the table indicate the order of evaluation. Table inspired from (Baydin et al. 2017).

Forward Primal Trace		Forward Tangent Trace	
$v_1 = x_1$	$= 7$	$v'_1 = x'_1$	$= 0\epsilon$
$v_2 = x_2$	$= 3$	$v'_2 = x'_2$	$= 1\epsilon$
$v_3 = v_1 \times v_2$	$= 7 \times 3$	$v'_3 = x'_1 \frac{\partial v_3}{\partial v_1} + x'_2 \frac{\partial v_3}{\partial v_2}$	$= 0\epsilon \times 3 + 1\epsilon \times 7 = 7\epsilon$
$v_4 = \ln(v_2)$	$= \ln(3)$	$v'_4 = v'_2 \frac{\partial v_4}{\partial v_2}$	$= 1\epsilon \times \frac{1}{3} = 0.333\epsilon$
$v_5 = v_3 + v_4$	$= 21 + 1.098$	$v'_5 = x'_3 \frac{\partial v_5}{\partial v_3} + x'_4 \frac{\partial v_5}{\partial v_4}$	$= 7\epsilon + 0.333\epsilon = 7.333\epsilon$
$y = v_5$	$= 22.098$	$y' = v'_5$	$= 7.333\epsilon$

where, $v \in \mathbb{R}$ is the primal number, $v' \in \mathbb{R}$ is the tangent value and ϵ is a nilpotent number such as $\epsilon \neq 0$ and $\epsilon^2 = 0$. In practice, dual numbers are represented by a pair (v, v') in memory. Simple operations with dual numbers thus conveniently perform first-order differentiation. For example:

$$\begin{aligned} (v + v'\epsilon) + (u + u'\epsilon) &= (v + u) + (v' + u')\epsilon \\ (v + v'\epsilon)(u + u'\epsilon) &= (vu) + (vu' + v'u)\epsilon \end{aligned} \tag{59}$$

and applying a differentiable function f to a dual number leads to the following transformation:

$$f(v + v'\epsilon) = f(v) + f'(v)v'\epsilon \tag{60}$$

where $f'(v)$ is the first order derivative of f.

Table 2 shows an example of Forward-Mode AD applied to the function defined in Equation 57. In this example, the tangent trace for the input variables v_1 and v_2 are initialized to 0 and 1 respectively. By computing the tangent trace along with the primal trace using dual numbers, the derivative $\frac{\partial y}{\partial x_2}$ is produced at the output, without having to propagate derivatives backward (Baydin et al. 2017). However, unlike Reverse-Mode AD, only a single tangent is produced per evaluation, which means that n evaluations of the function are required to compute the full gradient, where n is the number of inputs to the function.

2.6 Chapter Summary

In Section 2.1, we briefly reviewed the structure and dynamics of biological neurons, introduced the two main theories of information coding, and discussed plasticity in the brain.

Moving to Section 2.2, we explored the historical evolution of artificial neural networks, from the McCulloch-Pitts Neuron to the continuously activated neuron model used in modern deep learning. More specifically, we introduced the BP algorithm, which plays a crucial role in computing gradients to update the weights of DNNs through gradient-based optimization techniques, as described in Section 2.4. Often implemented using Reverse-Mode AD, as detailed in Section 2.5, BP effectively addresses the error credit assignment problem and is considered to be the workhorse of high-performance deep learning. In addition, we discussed the fundamental deep learning problem causing vanishing that hinders the convergence of DNNs and introduced various solutions available to overcome this issue and enable the training of deeper networks. We then described various network architectures that have been made possible by these solutions, namely CNNs, and RNNs. Finally, we discussed the different hardware that can be used to accelerate the inference and training of DNNs such as GPUs, FPGAs, and ASICs.

In Section 2.3, we introduced the more biologically plausible SNNs as an alternative to DNNs for low-powered computation. More precisely, we introduced various spiking neuron models with different complexities, including the computationally efficient LIF neuron and its mapping to the SRM model. We then explored the range of hardware options available for the acceleration SNNs. SNNs become particularly advantageous when implemented on energy-efficient neuromorphic hardware. Finally, we discuss the benefits and disadvantages of spike count coding over TTFS and introduce several unsupervised and supervised methods for training SNNs.

Current state-of-the-art results with SNNs are achieved through various adaptations of BP and GD to SNNs that overcome the non-differentiability of spikes. For instance, the recent Fast & Deep algorithm, introduced in Section 2.3.9, uses specific constraints to isolate closed-form solutions of spike timings and compute exact gradients of SNNs. However, the Fast & Deep algorithm has only been benchmarked on temporally-coded networks and is yet to be generalized to SNNs firing multiple spikes per neuron. One other challenge associated with GD in SNNs is the non-local and reverse aspects of BP that are incompatible with the local and online constraints of neuromorphic hardware. Because of this limitation, GD is primarily suited for unadapted von Neumann architectures, resulting in computationally expensive and energy-consuming training of SNNs.

Chapter 3

Exact Gradients of Spiking Neural Networks

The contributions of this chapter are:

- I introduce an extension of the Fast & Deep algorithm that relaxes the firing constraint of temporally-coded SNNs and computes the exact gradients of unconstrained SNNs.
- I compare temporally-coded SNNs trained with Fast & Deep and unconstrained SNNs trained with the proposed method on several criteria such as convergence likelihood, performance, rate of convergence, classification latency, and robustness to noise and weight quantization.
- I investigate the role of weight initialization in temporally-coded SNNs and highlight tradeoffs between performance, sparsity, classification latency, and robustness.

3.1 Introduction

Inspired by DNNs, supervised learning in SNNs is mostly performed with various adaptations of the gradient descent and BP algorithms. However, the discontinuous aspect of spikes makes the computation of exact gradients of SNNs more challenging than DNNs.

Surrogate gradients are popular approaches to overcome the non-differentiability of the threshold function in SNNs (Shrestha and Orchard 2018; Zheng et al. 2020; Wu et al. 2018). These methods typically model the probability of state transition (i.e. from spiking to non-spiking or non-spiking to spiking) to allow errors to be temporally backpropagated through the membrane potential dynamics using BPTT (Shrestha and Orchard 2018) — see Section 2.3.9 for more details. While these methods allow for high-performance training of SNNs, they are known to perform many unnecessary steps and require recording the states of neurons at each time step, which is generally not provided by neuromorphic hardware or, when possible, represents a major source of energy consumption. Therefore, surrogate gradient methods are mostly limited to offline training.

In contrast, event-based gradient descent techniques for SNNs compute gradients by propagating errors through spike timings. The inherently event-driven nature of these methods makes them particularly suitable for in-the-loop training with neuromorphic hardware as they exclusively rely on spike timings to compute gradients (Göltz et al. 2021). However, the absence of a closed-form solution for spike timings represents a major obstacle to the computation of exact gradients (Mostafa 2016; Comsa et al. 2020; Göltz et al. 2021; Wunderlich and Pehle 2021; Lee, Haghighatshoar and Karbasi 2023).

Solutions have been proposed to either sidestep the lack of analytical solution by computing implicit gradients (Wunderlich and Pehle 2021; Lee, Haghighatshoar and Karbasi 2023) or isolate specific analytical solutions for spike timings imposing constraints on the model parameters (Mostafa 2016; Comsa et al. 2020; Göltz et al. 2021).

For instance, the Fast & Deep algorithm (Göltz et al. 2021) introduces closed-form solutions for spike timings by using various time constant constraints. This allows for the computation of exact gradients and enables the training of fast, energy-efficient, and robust SNNs. However, in its original version, the Fast & Deep algorithm uses SNNs temporally coded with TTFS — see Section 2.3.7 for details — which are known for their advantages in terms of inference speed and energy-efficiency but also for their various limitations.

First, the firing constraint imposed on neurons using TTFS coding limits their capacity to transmit information. In the spike-based theory of neural coding (Brette 2015), each spike carries some amount of information. Firing rates relates to the rates of information that are transmitted by neurons. Therefore, limiting the number of spikes that neurons can fire also limits their information capacity. For this reason, SNNs using a single spike per neuron do not perform as well as SNNs allowing multiple spikes per neuron (Jin, Zhang and Li 2018; Zhang and Li 2019; Shrestha and Orchard 2018; Lee, Delbruck and Pfeiffer 2016; Zhang et al. 2022). Fast & Deep is no exception to this limitation as it does not reach the performance of ANNs on simple benchmark datasets such as MNIST (Göltz et al. 2021).

Second, limiting neurons to fire only once also makes them unable to respond to several different patterns in time. SNNs are capable of responding to many spatio-temporal patterns occurring at different timings making them computationally powerful. Intuitively, neurons that fire only once are not capable of responding to recurrent patterns in time once their single spike has been fired. This limitation becomes particularly significant since certain types of data contain many repetitive patterns, such as sound or videos. Therefore, by constraining neurons to only fire a single spike, Fast & Deep restricts the computational capabilities of SNNs.

Finally, temporal coding schemes such as TTFS are particularly subject to the effects of substrate noise of analog neuromorphic hardware (Guo et al. 2021; Park, Lee and Yoon 2021). More particularly, the only spike fired by neurons in temporally-coded SNNs is strongly affected by *spike jitter* where spikes are shifted by small temporal perturbations. This weakness can have a drastic impact on the performance of temporally-coded SNNs if not addressed (Park, Lee and Yoon 2021).

Motivated by these limitations, I propose to relax the single spike constraint imposed by Fast & Deep to allow the generation of multiple spikes per neuron in SNNs, which I refers to as *unconstrained SNNs*. To understand the benefits of relaxing this constraint, I conduct several empirical experiments to explore the relationships between performance, energy consumption, speed, and noise robustness when using temporally-coded and unconstrained SNNs. The rest of this chapter is therefore structured as follows:

- I first review the underlying cause behind the absence of a general closed-form solution for spike timings as well as one of the solutions proposed by Fast & Deep (Göltz et al. 2021). In addition, I review how the Fast & Deep algorithm backpropagates errors through spikes and computes exact gradients in temporally-coded SNNs.

- Then, I introduce my generalization of the Fast & Deep algorithm to unconstrained SNNs. By taking into account all the spatio-temporal dependencies between spikes, I highlight the fact that errors are backpropagated through two distinct paths due to the addition of reset in the computational graph and describe how errors are backpropagated in unconstrained SNNs to compute exact gradients.

- I describe the experimental settings used to produce my empirical results including network architectures, loss functions, weight initializations, hyperparameters, and hardware acceleration.

- I finally empirically compare Fast & Deep with the proposed method on multiple criteria such as convergence likelihood, performance on benchmark datasets, convergence rate, sparsity, classification latency, and robustness to noise and weight quantization. Moreover, I investigate the role of weight initialization in TTFS SNNs and highlight several tradeoffs between these different aspects.

3.2 The Lack of Closed-Form Solution for Spike Timings

I now consider the LIF neuron model as described in Section 2.3.3 and, more specifically, its mapping in the SRM model as described in Section 2.3.4. Using the α function defined in Equation 27, the temporal impact of incoming pre-synaptic spikes onto the membrane potential is described by the following bi-exponential PSP kernel:

$$\epsilon(t) = \Theta\left(t\right) \frac{A}{C} \frac{\tau \tau_s}{\tau - \tau_s} \left[\exp\left(\frac{-t}{\tau}\right) - \exp\left(\frac{-t}{\tau_s}\right) \right] \tag{61}$$

For simplicity, we will assume that $\frac{A}{C}\frac{\tau\tau_s}{\tau-\tau_s} = 1 \text{ Volt}$. Therefore, using Equation 31 as PSP kernel and the constants constraint, the membrane potential $u_i(t)$ of the i^{th} neuron, as defined by the SRM model, becomes:

$$\begin{aligned} u_i(t) = &u_{\text{rest}} + \sum_{j \in \mathcal{P}_i} w_{i,j} \sum_{t_j^z \in \mathcal{T}_j} \Theta\left(t - t_j^z\right) \left[\exp\left(\frac{t_j^z - t}{\tau}\right) - \exp\left(\frac{t_j^z - t}{\tau_s}\right) \right] \\ &- \Delta_\vartheta \sum_{t_i^k \in \mathcal{T}_i} \Theta\left(t - t_i^k\right) \exp\left(\frac{t_i^k - t}{\tau}\right) \end{aligned} \tag{62}$$

where $\Delta_\vartheta = \vartheta - u_{\text{rest}}$.

To be able to differentiate spike timings and compute exact gradients of SNNs, we first need to isolate a closed-form solution of each post-synaptic spike t_i^k when the membrane potential crosses the threshold, such as:

$$\vartheta = u_i(t_i^k) \tag{63}$$

However, it appears that the PSP kernel defined in Equation 31 has no general closed-form solution. This is because it is defined as a difference of exponential functions, which is known to have no general closed-form expression. The lack of a closed-form solution for post-synaptic spike timings is therefore the major obstacle to the computation of exact gradients in SNNs (Wunderlich and Pehle 2021).

3.3 Closed-Form Solution by Time Constant Constraints

Until recently, the lack of a closed-form solution for spike timings was seen as a major obstacle to the differentiation of spike timings. However, recent works have successfully derived exact expressions of spikes allowing the computation of exact gradients (Mostafa 2016; Comsa et al. 2020; Göltz et al. 2021). These methods use constraints between the membrane and synaptic time constants to isolate a closed-form solution of post-synaptic spike timings. In particular, the recent Fast & Deep (Göltz et al. 2021) method proposed three different constraints: $\tau \rightarrow \infty$ (i.e. no leak), $\tau = \tau_s$ and $\tau = 2\tau_s$. In the following, I will only focus on the last constraint $\tau = 2\tau_s$ because of its simplicity.

Formally, by constraining the membrane time constant $\tau = 2\tau_s$ as twice the synaptic time constant, the following Equation holds:

$$\exp\left(\frac{-t}{\tau}\right)^2 = \exp\left(\frac{-t}{\tau_s}\right) \tag{64}$$

Therefore, Equation 63 can be written as a quadratic equation (Göltz et al. 2021), such as:

$$
\begin{aligned}
&\vartheta = u_i\left(t_i^k\right) \\
&\Leftrightarrow \vartheta = u_{\text{rest}} + \sum_{j\in\mathcal{P}_i} w_{i,j} \sum_{t_j^z\in\mathcal{T}_j} \Theta\left(t_i^k - t_j^z\right)\left[\exp\left(\frac{t_j^z - t_i^k}{\tau}\right) - \exp\left(\frac{t_j^z - t_i^k}{\tau_s}\right)\right] \\
&\qquad - \Delta_\vartheta \sum_{t_i^z\in\mathcal{T}_i} \Theta\left(t_i^k - t_i^z\right)\exp\left(\frac{t_i^z - t_i^k}{\tau}\right) \\
&\Leftrightarrow 0 = \sum_{j\in\mathcal{P}_i} w_{i,j} \sum_{t_j^z\in\mathcal{T}_j} \Theta\left(t_i^k - t_j^z\right)\left[\exp\left(\frac{t_j^z}{\tau}\right)\exp\left(\frac{-t_i^k}{\tau}\right) - \exp\left(\frac{t_j^z}{\tau_s}\right)\exp\left(\frac{-t_i^k}{\tau_s}\right)\right] \\
&\qquad - \Delta_\vartheta \sum_{t_i^z\in\mathcal{T}_i} \Theta\left(t_i^k - t_i^z\right)\exp\left(\frac{t_i^z}{\tau}\right)\exp\left(\frac{-t_i^k}{\tau}\right) + u_{\text{rest}} - \vartheta \\
&\Leftrightarrow 0 = \left[\sum_{j\in\mathcal{P}_i} w_{i,j} \sum_{t_j^z\in\mathcal{T}_j} \Theta\left(t_i^k - t_j^z\right)\exp\left(\frac{t_j^z}{\tau}\right) - \Delta_\vartheta \sum_{t_i^z\in\mathcal{T}_i} \Theta\left(t_i^k - t_i^z\right)\exp\left(\frac{t_i^z}{\tau}\right)\right]\exp\left(\frac{-t_i^k}{\tau}\right) \\
&\qquad - \left[\sum_{j\in\mathcal{P}_i} w_{i,j} \sum_{t_j^z\in\mathcal{T}_j} \Theta\left(t_i^k - t_j^z\right)\exp\left(\frac{t_j^z}{\tau_s}\right)\right]\exp\left(\frac{-t_i^k}{\tau}\right)^2 - \Delta_\vartheta \\
&\Leftrightarrow\ -a_i^k \exp\left(\frac{-t_i^k}{\tau}\right)^2 + b_i^k \exp\left(\frac{-t_i^k}{\tau}\right) - \Delta_\theta
\end{aligned}
\tag{65}
$$

where a_i^k, b_i^k are two coefficients associated with the post-synaptic spike at time t_i^k, defined as:

$$
a_i^k := \sum_{j\in\mathcal{P}_i} w_{i,j} \sum_{t_j^z\in\mathcal{T}_j} \Theta\left(t_i^k - t_j^z\right)\exp\left(\frac{t_j^z}{\tau_s}\right) \tag{66}
$$

and

$$
b_i^k := \sum_{j\in\mathcal{P}_i} w_{i,j} \sum_{t_j^z\in\mathcal{T}_j} \Theta\left(t_i^k - t_j^z\right)\exp\left(\frac{t_j^z}{\tau}\right) - \Delta_\vartheta \sum_{t_i^z\in\mathcal{T}_i} \Theta\left(t_i^k - t_i^z\right)\exp\left(\frac{t_i^z}{\tau}\right) \tag{67}
$$

Because, Equation 65 is a polynomial of degree 2, we can use the quadratic formula to find a closed-form solution for t_i^k. Therefore, by choosing the solution that corresponds to the rise of the membrane potential (Göltz et al. 2021), Equation 65 has the following solution:

$$\begin{aligned}\exp\left(\frac{-t_i^k}{\tau}\right) &= \frac{b_i^k + x_i}{2a_i^k} \\ \Leftrightarrow t_i^k &= \tau \ln\left(\frac{2a_i^k}{b_i^k + x_i^k}\right)\end{aligned} \tag{68}$$

where

$$x_i^k := \sqrt{\left(b_i^k\right)^2 - 4a_i^k \Delta_\vartheta} \tag{69}$$

This demonstrates that specific analytical solutions for spike timings can be isolated. In particular, these solutions are differentiable. Therefore, exact gradients of SNNs can be computed.

3.4 Exact Gradients of Temporally-Coded Spiking Neural Networks

In addition to the closed-form solution of spike timings previously introduced, the Fast & Deep algorithm computes exact gradients of temporally-coded SNNs (Göltz et al. 2021). More precisely, it computes exact gradients of spiking neurons by differentiating the closed-form solution of their spike timings and backpropagating errors backward through space and time from spike to spike.

Formally, given a differentiable loss $\mathcal{L}$, the gradient is defined by all the partial derivative $\frac{\partial \mathcal{L}}{\partial w_{i,j}}$ with respect to the weights, such as:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{i,j}} &= \frac{\partial \mathcal{L}}{\partial t_i}\frac{\partial t_i}{\partial w_{i,j}} \\ &= \delta_i \frac{\partial t_i}{\partial w_{i,j}}\end{aligned} \tag{70}$$

where

$$\delta_i := \frac{\partial \mathcal{L}}{\partial t_i} \tag{71}$$

is the error backpropagated to the spike time $t_i = t_i^1$. Here, I dropped the spike index for conciseness.

Because a closed-form solution for spike timings exists if $\tau = 2\tau_s$, it is possible to differentiate the post-synaptic spike t_i with respect to the weight $w_{i,j}$, such as:

$$\begin{aligned}\frac{\partial t_i}{\partial w_{i,j}} &= \frac{\partial t_i}{\partial a_i}\frac{\partial a_i}{\partial w_{i,j}} + \frac{\partial t_i}{\partial b_i}\frac{\partial b_i}{\partial w_{i,j}} \\ &= \Theta\left(t_i - t_j\right)\left[\frac{\partial t_i}{\partial a_i}\exp\left(\frac{t_j}{\tau_s}\right) + \frac{\partial t_i}{\partial b_i}\exp\left(\frac{t_j}{\tau}\right)\right]\end{aligned} \tag{72}$$

where

$$\frac{\partial t_i}{\partial a_i} = \frac{\tau}{a_i}\left[1 + \frac{\Delta_\vartheta}{x_i}\exp\left(\frac{t_i}{\tau}\right)\right] \tag{73}$$

and

$$\frac{\partial t_i}{\partial b_i} = -\frac{\tau}{x_i} \tag{74}$$

3.4.1 Error Backpropagation in Temporally-Coded Spiking Neural Networks

To compute the error δ_i associated to the hidden spike time t_i, the output errors $\delta_o = \frac{\partial \mathcal{L}}{\partial t_o}$ is recursively backpropagated backward through time and space using BP. By denoting $\mathcal{H}_i = \{j \mid i \text{ is connected to } j\}$ as the set of downstream neurons connected to the neuron i, the error δ_i for the hidden neuron i is computed as:

$$\delta_i = \sum_{j \in \mathcal{H}_i} \frac{\partial t_j}{\partial t_i}\delta_j \tag{75}$$

where

$$\begin{aligned}\frac{\partial t_j}{\partial t_i} &= \frac{\partial t_j}{\partial a_j}\frac{\partial a_j}{\partial t_i} + \frac{\partial t_j}{\partial b_j}\frac{\partial b_j}{\partial t_i} \\ &= \Theta\left(t_j - t_i\right) w_{i,j}\left[\frac{\partial t_j}{\partial a_j}\frac{\exp\left(\frac{t_i}{\tau_s}\right)}{\tau_s} + \frac{\partial t_j}{\partial b_j}\frac{\exp\left(\frac{t_i}{\tau}\right)}{\tau}\right]\end{aligned} \tag{76}$$

After computing the gradient, weights can thus be updated using a gradient-based optimization algorithm — see Section 2.4 for details.

By using TTFS coding, Fast & Deep is capable of training low-latency SNNs producing fast predictions and promising results on benchmark datasets. Moreover, Fast & Deep is particularly suitable for in-the-loop training as it only requires spike timings. In addition, the algorithm has shown robustness to the substrate noise of the mixed-signal Brainscale-2 neuromorphic hardware during in-the-loop training (Göltz et al. 2021). Therefore, Fast & Deep represents a promising approach for supervised learning in SNNs.

3.5 Exact Gradients of Unconstrained Spiking Neural Networks

In this section, I introduce my extension of the Fast & Deep algorithm that relaxes the constraint on the firing of temporal coding and describe how errors are backpropagated through spikes in unconstrained SNNs.

I first derive the exact gradient of unconstrained SNNs, defined by all the partial derivatives $\frac{\partial \mathcal{L}}{\partial w_{i,j}}$ of a differentiable loss function $\mathcal{L}$ with respect to the weights. In an unconstrained SNN, each post-synaptic spike t_i^k of a neuron i receives errors and contributes to the computation of the gradient. Therefore, the partial derivatives $\frac{\partial \mathcal{L}}{\partial w_{i,j}}$ composing the gradient are defined as the sum over the post-synaptic spikes, such as:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{i,j}} &= \sum_{t_i^k \in \mathcal{T}_i} \frac{\partial \mathcal{L}}{\partial t_i^k} \frac{\partial t_i^k}{\partial w_{i,j}} \\ &= \sum_{t_i^k \in \mathcal{T}_i} \delta_i^k \frac{\partial t_i^k}{\partial w_{i,j}}\end{aligned} \tag{77}$$

where

$$\delta_i^k := \frac{\partial \mathcal{L}}{\partial t_i^k} \tag{78}$$

is the error backpropagated to the k^{th} spike time t_i^k of the neuron i.

By using the closed-form solution for the spike time (Equation 68), we can thus differentiate each post-synaptic t_i^k with respect to the weight $w_{i,j}$:

$$\begin{aligned}\frac{\partial t_i^k}{\partial w_{i,j}} &= \frac{\partial t_i^k}{\partial a_i^k}\frac{\partial a_i^k}{\partial w_{i,j}} + \frac{\partial t_i^k}{\partial b_i^k}\frac{\partial b_i^k}{\partial w_{i,j}} \\ &= \sum_{t_j^z \in \mathcal{T}_j} \Theta\left(t_i^k - t_j^z\right)\left[\frac{\partial t_i^k}{\partial a_i^k}\exp\left(\frac{t_j^z}{\tau_s}\right) + \frac{\partial t_i^k}{\partial b_i^k}\exp\left(\frac{t_j^z}{\tau}\right)\right]\end{aligned} \tag{79}$$

where

$$\frac{\partial t_i^k}{\partial a_i^k} = \frac{\tau}{a_i^k}\left[1 + \frac{\Delta_\vartheta}{x_i^k}\exp\left(\frac{t_i^k}{\tau}\right)\right] \tag{80}$$

and

$$\frac{\partial t_i^k}{\partial b_i^k} = -\frac{\tau}{x_i^k} \tag{81}$$

3.5.1 Error Backpropagation in Unconstrained Spiking Neural Networks

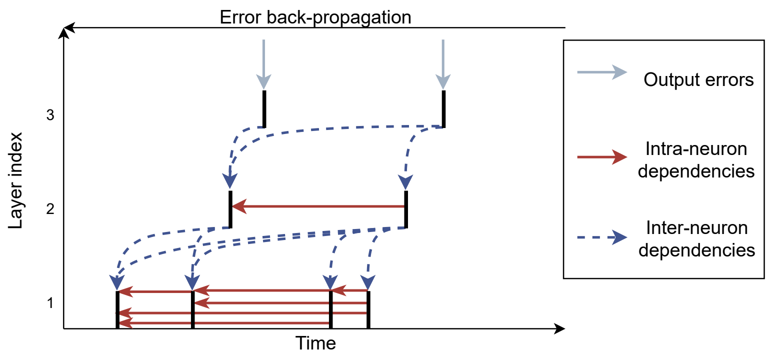


Figure 23: Illustration of error backpropagation through spikes. This figure represents a three-layered network where the spike trains of only one neuron per layer are shown. The grey arrows represent the error coming from the loss function, the dashed blue arrows are the errors backpropagated from the downstream spikes (i.e. inter-neuron dependencies) and the red arrows are the error backpropagated from the future activity of the neuron due to the recurrence of the reset function (i.e. intra-neuron dependencies).

As in Fast & Deep, output errors can recursively be backpropagated backward through time and space using the error backpropagation algorithm. However, additional dependencies between post-synaptic spikes are present in unconstrained neurons due to the reset of the membrane potential. More precisely, errors are backpropagated through two distinct paths: from post-synaptic to pre-synaptic spikes (or *inter-neuron dependencies*), and through post-synaptic to post-synaptic spikes (or *intra-neuron dependencies*).

Inter-neuron dependencies arise due to the causal interactions between pre-synaptic and post-synaptic neurons. In essence, errors are spatially backpropagated to pre-synaptic spikes that have a causal relationship with post-synaptic spikes — see the dashed blue arrows in Figure 23. On the other hand, intra-neuron dependencies arise due to the causal interactions between post-synaptic spikes. More precisely, the reset of the membrane potential caused by post-synaptic spikes temporally influences subsequent spikes. Errors are therefore backpropagated within the post-synaptic spikes of the same neuron — see red arrows in Figure 23.

Formally, the error δ_i^k computed through error backpropagation can be decomposed into two error components:

$$\begin{aligned} \delta_i^k :=& \frac{\partial \mathcal{L}}{\partial t_i^k} \\ =& \underbrace{\sum_{j \in \mathcal{H}_i} \sum_{t_j^z \in \mathcal{T}_j} \frac{\partial \mathcal{L}}{\partial t_j^z} \frac{\partial t_j^z}{\partial t_i^k}}_{\text{Inter-Neuron}} + \underbrace{\sum_{t_i^z \in \mathcal{T}_i} \frac{\partial \mathcal{L}}{\partial t_i^z} \frac{\partial t_i^z}{\partial t_i^k}}_{\text{Intra-Neuron}} \\ =& \phi_i^k + \mu_i^k \end{aligned} \tag{82}$$

where ϕ_i^k is the error backpropagated through the synapses (inter-neuron dependencies) and μ_i^k is the error backpropagated through the reset of the membrane potential (intra-neuron dependencies). In contrast with my method, Fast & Deep only propagates errors through inter-neuron dependencies as the membrane potential of constrained neurons is never reset.

Inter-Neuron Dependencies

The inter-neuron error ϕ_i^k has different definitions for output neurons and hidden neurons. For output neurons o, the inter-neuron error ϕ_o^k received by the k^{th} spike of the neuron o is defined as the derivative of the loss function as no inter-neuron backpropagation is required:

$$\phi_o^k := \frac{\partial \mathcal{L}}{\partial t_o^k} \tag{83}$$

For hidden neurons, the inter-neuron error ϕ_i^k is defined as the sum of all errors backpropagated from the downstream spikes that have been emitted since the time of firing t_i^k, such as:

$$\phi_i^k = \sum_{j \in \mathcal{H}_i} \sum_{t_j^z \in \mathcal{T}_j} \frac{\partial t_j^z}{\partial t_i^k} \delta_j^k \tag{84}$$

where

$$\begin{aligned} \frac{\partial t_j^z}{\partial t_i^k} &= \frac{\partial t_j^z}{\partial a_j^z} \frac{\partial a_j^z}{\partial t_i^k} + \frac{\partial t_j^z}{\partial b_j^z} \frac{\partial b_j^z}{\partial t_i^k} \\ &= \Theta\left(t_j^z - t_i^k\right) w_{i,j} \left[\frac{\partial t_j^z}{\partial a_j^z} \frac{\exp\left(\frac{t_i^k}{\tau_s}\right)}{\tau_s} + \frac{\partial t_j^z}{\partial b_j^z} \frac{\exp\left(\frac{t_i^k}{\tau}\right)}{\tau} \right] \end{aligned} \tag{85}$$

Essentially, the way errors backpropagate through inter-neuron dependencies in unconstrained SNNs is similar to the original Fast & Deep algorithm. The only difference lies in the fact that, in unconstrained SNNs, errors are backpropagated from multiple spikes instead of a single one.

Intra-Neuron Dependencies

When firing, each post-synaptic spike affects the membrane potential of the neuron due to the reset. This creates temporal intra-neuron dependencies between causal post-synaptic spikes. Consequently, computing the intra-neuron error μ_i^k requires backpropagating errors from all the future spike activity of the neuron. Formally, μ_i^k is defined as a sum over all errors backpropagated from the post-synaptic spikes occurring after t_i^k, such as:

$$\mu_i^k = \sum_{t_i^z \in \mathcal{T}_i} \frac{\partial t_i^z}{\partial t_i^k} \delta_i^z \tag{86}$$

where

$$\begin{aligned} \frac{\partial t_i^z}{\partial t_i^k} &= \frac{\partial t_i^z}{\partial a_i^z} \frac{\partial a_i^z}{\partial t_i^k} + \frac{\partial t_i^z}{\partial b_i^z} \frac{\partial b_i^z}{\partial t_i^k} \\ &= \Theta\left(t_i^z - t_i^k\right) \frac{\Delta_\vartheta}{x_i^z} \exp\left(\frac{t_i^k}{\tau}\right) \end{aligned} \tag{87}$$

If evaluated as written, Equation 86 implies a quadratic complexity $\mathcal{O}\left(n^2\right)$ where n is the number of post-synaptic spikes fired by the neuron. However, every reset has the same impact on the membrane potential i.e. an instantaneous negative voltage of size $\Delta_\theta = \vartheta - u_{\text{rest}}$. Therefore, the computation of μ_i^k can be factorized as follows:

$$\begin{aligned} \mu_i^k =& \Delta_\vartheta \exp\left(\frac{t_i^k}{\tau}\right) \sum_{t_i^z \in \mathcal{T}_i} \Theta\left(t_i^z - t_i^k\right) \frac{\delta_i^z}{x_i^z} \\ =& \Delta_\vartheta \exp\left(\frac{t_i^k}{\tau}\right) \beta_i^{k+1} \end{aligned} \tag{88}$$

where

$$\beta_i^k = \beta_i^{k+1} + \frac{\delta_i^k}{x_i^k} \tag{89}$$

is a recursive term that can be accumulated in linear time $\mathcal{O}(n)$ during the backward pass. Therefore Equation 88 allows for a linear error backpropagation through intra-neuron dependencies that significantly reduces the computational cost of the algorithm compared to Equation 86.

3.6 Experimental Settings

In this section, I describe the experimental settings used to produce my empirical results, including the benchmark datasets, encoding and decoding techniques, network architectures, loss functions, the different hyperparameter values as well as my custom event-based simulator for GPU.

3.6.1 Benchmark Datasets

Table 3: Descriptions of the benchmark datasets used in my experiments.

Dataset Name	Train Samples	Test Samples	Classes	Description
MNIST	60 000	10 000	10	28x28 images of handwritten digits
EMNIST	112 800	18 800	47	28x28 images of handwritten digits and letters
Fashion MNIST	60 000	10 000	10	28x28 images of clothes
SHD	8 156	2 264	20	700 channels recordings of spoken digits by an artificial cochlea

To evaluate the performance of the proposed method and compare it with the original Fast & Deep, I selected four benchmark datasets, namely the Modified National Institute of Standards and Technology (MNIST) dataset (LeCun, Cortes and Burges 2010), the Balanced Extended MNIST (EMNIST) dataset (Cohen et al. 2017), the Fashion MNIST dataset (Xiao, Rasul and Vollgraf 2017) and the Spiking Heidelberg Dataset (SHD) (Cramer et al. 2022a). See Table 3 for a detailed description of each dataset.

3.6.2 Encoding

To encode the pixel values of static images (e.g. MNIST, EMNIST, and Fashion MNIST) into spikes that can be used as inputs of SNNs, I applied the TTFS coding scheme — see Section 2.3.7 for details — with an encoding window of $T = 100$ ms. For pixel values of 0 (i.e. black), no spikes were generated. In this way, TTFS therefore produces sparse temporal encoding of grayscale images that are fast to process in my event-based simulator. For the SHD dataset, I used the spike trains as provided by the dataset.

3.6.3 Network Architectures

To benchmark the proposed method, I used two types of architectures, namely fully-connected SNNs and convolutional SNNs. Fully connected SNNs are, like their DNNs counterparts, composed of successive layers where each neuron is connected to all the neurons of the previous layer.

Convolutional SNNs are also similar to CNNs with the exception of their pooling layers. Weights are spatially shared between the neurons of convolutional spiking layers. However, max pooling and average pooling are not directly applicable to spike trains. Therefore, I adopt the aggregation pooling method (Shrestha and Orchard 2018) which consists of aggregating the spike trains of a sub-region into a single spike train.

3.6.4 Decoding and Loss Functions

Because of its constraint on spikes, class predictions in Fast & Deep are given by the first spike emitted by the output layer of SNNs, such as:

$$p = \arg\min_{o} t_o \tag{90}$$

where p is the prediction label and t_o is the only spike fired by the output neuron o.

During the training of TTFS SNNs, I adopted the same *Softmax Cross Entropy* loss function as the original Fast & Deep work (Göltz et al. 2021). It is defined as:

$$\begin{aligned}\mathcal{L} &:= -\sum_o \widehat{y}_o \log\left[\frac{\exp\left(\frac{-t_o}{\tau_\mathcal{L}}\right)}{\sum_{o'}\exp\left(\frac{-t_{o'}}{\tau_\mathcal{L}}\right)}\right]\\ &= \log\left[\frac{\sum_o \exp\left(\frac{-t_o}{\tau_\mathcal{L}}\right)}{\exp\left(\frac{-t_{\hat{y}}}{\tau_\mathcal{L}}\right)}\right]\\ &= \log\left[\sum_o \exp\left(-\frac{t_o - t_{\hat{y}}}{\tau_\mathcal{L}}\right)\right]\end{aligned} \tag{91}$$

where $\hat{y}$ is the target label, $\tau_\mathcal{L}$ is the loss time constant and

$$\widehat{y}_o = \begin{cases} 1 & \text{if } o = \hat{y} \\ 0 & \text{otherwise}\end{cases} \tag{92}$$

In contrast with Fast & Deep, neurons in my unconstrained SNNs can fire multiple spikes, allowing more freedom in the choice of spike decoding schemes and loss functions. Therefore, I adopted a spike count decoding for the outputs of my unconstrained SNNs, leading to the following predictions:

$$p = \arg\max_o n_o \tag{93}$$

where p is the predition label and n_o is the spike count for the output neuron o.

Finally, I used the following *Mean Squared Error* loss function to train my SNNs to match a desired firing count, such as:

$$\mathcal{L} := \frac{1}{2}\sum_o \left(\widehat{y}_o - n_o\right)^2 \tag{94}$$

where $\widehat{y}_o$ is the spike count target for the output neuron o.

3.6.5 Update Method and Hyperparameters

I used the Adam (Kingma and Ba 2017) algorithm — see Section 2.4.6 for details — to update the weights of SNNs with the computed gradients. I used the default values of $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ (Kingma and Ba 2017), a batch size of 50 for fully connected SNNs, a batch size of 20 for convolutional SNNs and used a learning rate of $\lambda = 0.003$ for image classification and $\lambda = 0.001$ for audio classification with the SHD dataset.

Experimental conditions were standardized for both Fast & Deep and my method, except for weight distributions and thresholds. I used a synaptic time constant of $\tau_s = 130$ ms for image classification and $\tau_s = 100$ ms for the SHD dataset. Two uniform weight distributions ($w_{i,j} \sim U(-1, 1)$ and $w_{i,j} \sim U(0, 1)$) were used to evaluate Fast & Deep, to measure the effect of initial weight distributions on the different evaluation criteria. My method was solely assessed using $w_{i,j} \sim U(-1, 1)$, as using only positive weights lead to excessive spiking activity, hindering computational and energy efficiency. Thresholds were manually tuned to find the best-performing networks and kept fixed during training.

I used a time constant of $\tau_{\mathcal{L}} = 5$ ms for the softmax cross entropy loss function of Fast & Deep. For my method, I choose a spike count target of 15 for the target label and 3 for the others.

All these parameters were choosen manually during my experiments to obtain the best performing SNNs.

3.6.6 Event-Based Simulations on GPU

Equation 68 provides a closed-form solution of the spike time that can be used to infer the spike trains of the neurons. In configurations where there is no lateral interaction within a layer, neurons' spike trains can be computed in parallel. Therefore, I have developed my own custom event-based Python simulator accelerated with GPUs.

In this simulator, spike trains are evaluated for the entire simulation layer after layer to be able to provide the full set of spike trains to the next one. Pre-synaptic events are therefore sorted in time which allows the computation of post-synaptic spikes by iterating through the ordered set of pre-synaptic events. However, there is no known method to predict the number of spikes that neurons will fire. Memory must therefore be allocated before the inference as GPUs do not allow dynamic memory allocation during kernel executions. For this reason, a spike train buffer of size n_{max} is allocated for each neuron before the inference, thus allowing a given maximum number of spikes per neuron. In my experiments with my method, $n_{\text{max}} = 30$ was choosen sufficiently large to allow multiple spikes per neuron. For Fast & Deep, $n_{\text{max}} = 1$ was set to one to constraint neurons to fire at most one spike.

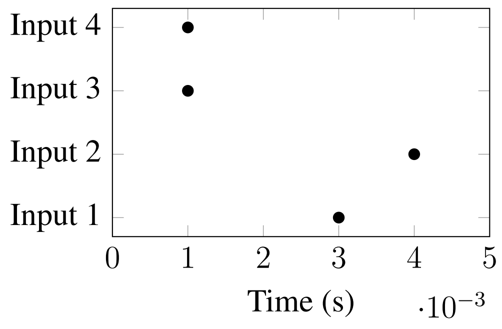
All the inference and backpropagation logic was implemented in custom Cuda kernels in C and interfaced with Python using CuPy (Okuta et al. 2017), a Numpy (Harris et al. 2020) compatible library for GPU-accelerated computing in Python.

3.7 Empirical Results

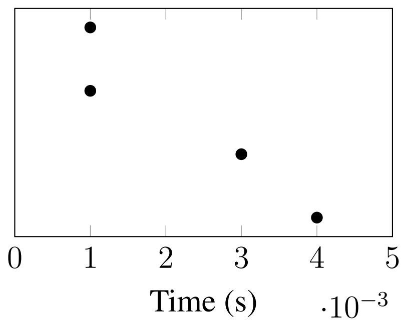
In this section, I compare the proposed method with Fast & Deep, initialized with two different weight distributions (i.e. $w_{i,j} \sim U(-1, 1)$ and $w_{i,j} \sim U(0, 1)$). This evaluation was conducted based on multiple criteria including convergence likelihood, performance on benchmark datasets, convergence rate, sparsity, classification latency, as well as robustness to noise and weight quantization.

3.7.1 Convergence Likelihood

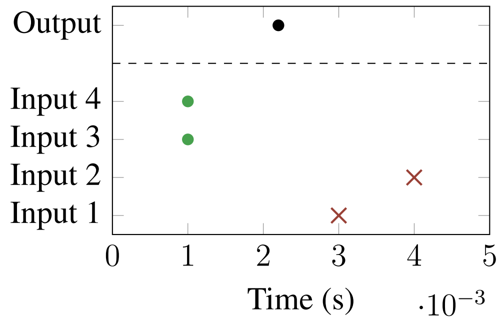
When a temporally-coded SNN is used to perform classification, only the first spike of each neuron contributes to the computation. Therefore, only the information carried by pre-synaptic spikes occurring before the first spike of a neuron is captured by the



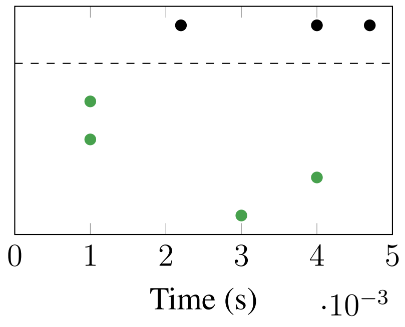
(a) Class 1



(b) Class 2

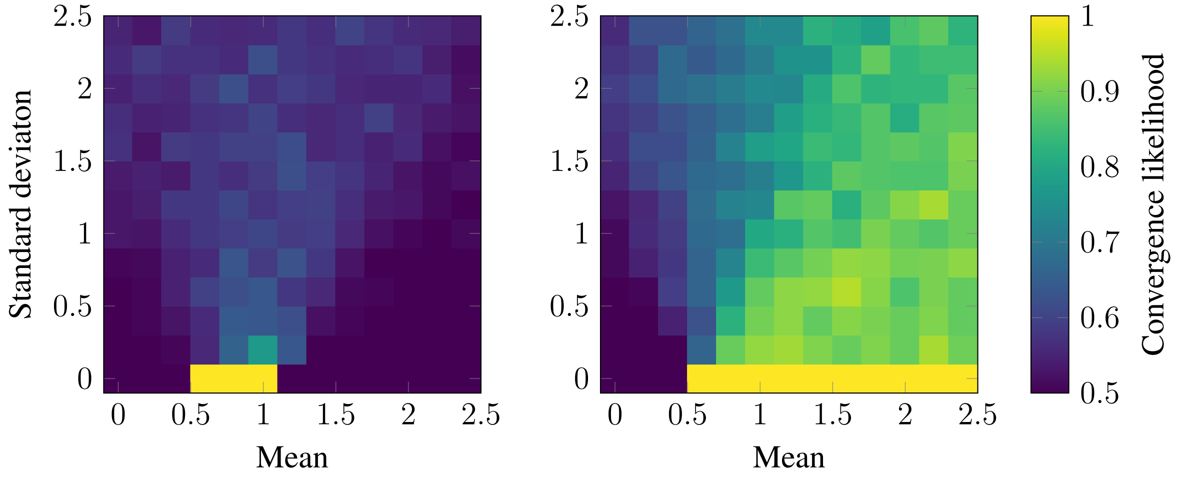


(c) TTFS neuron



(d) Unconstrained neuron

Figure 24: Example of classification of two simple patterns (Figures 24a and 24b) where relevant information is carried by the two last spikes. When using TTFS coding (Figure 24c), neurons can miss the relevant information if they spike too early and never converge. By using unconstrained SNNs (Figure 24d), neurons can fire several spikes, thus increasing the likelihood of convergence.



(a) TTFS neuron (b) Unconstrained neuron

Figure 25: Average accuracy achieved by different initial weight distributions for a single-spike model (Figure 25a) and a multi-spike model (Figure 25b) on the simple spatio-temporal pattern classification problem shown in Figure 24.

gradient. If during training – or simply at the initialization – neurons fire before the relevant information of the input pattern, learning is never achieved because the important features are not captured – see Figure 25a. On the other hand, a neuron model that fires multiple times is more likely to spike after the relevant information and thus achieve learning – see Figure 25b. Therefore, unconstrained SNNs are, in theory, more likely to converge than TTFS models.

To empirically validate this phenomenon, I measured the convergence likelihood of both TTFS and unconstrained models on a simple spatio-temporal pattern classification problem with different initial weight distributions. In this task, four input neurons generate a single spike each forming two different spatio-temporal patterns that must be classified. The first two spikes are always generated at the same timings in both patterns while the last two spikes are emitted at different timings and thus contain all the relevant information about the class — see Figures 24a and 24b. For each evaluated distribution, I trained two spiking neurons initialized with a normal weight distribution to classify these spatio-temporal patterns (i.e. one neuron per class) 100 times and computed the probability that neurons correctly classify the patterns. Figure 25 shows my results. To achieve convergence using a single spike per neuron, the distribution of the weights must be restricted to a small region of the weight distribution plane, otherwise, the weights are either too low to produce a spike or too high which makes the neurons fire earlier than the relevant spikes. However, the multi-spike model shows better convergence properties as it can discharge after the last input spikes, even if it already fired. This shows that, under some circumstances, multi-spike neurons may have better convergence capabilities than single-spike neurons as they can learn with a broader range of weight distribution.

3.7.2 Performance

Table 4: Performances comparison between Fast & Deep and my method on the MNIST, EMNIST, Fashion-MNIST, and Spiking Heidelberg Digits (SHD) datasets. The initial weight distribution used in each column is specified in the first row of the table. Conv. refers to a convolutional SNN with the following architecture: 15C5-P2-40C5-P2-300-10.

Dataset	Architecture	Fast & Deep $U(-1,1)$	Fast & Deep $U(0,1)$	My Method $U(-1,1)$
MNIST	800-10	$96.76 \pm 0.17\%$	$97.83 \pm 0.08\%$	**$98.88 \pm 0.02\%$**
	Conv.	$99.01 \pm 0.16\%$	$99.22 \pm 0.05\%$	**$99.38 \pm 0.04\%$**
EMNIST	800-47	$69.56 \pm 6.70\%$	$83.34 \pm 0.27\%$	**$85.75 \pm 0.06\%$**
Fashion MNIST	400-400-10	$88.14 \pm 0.08\%$	$88.47 \pm 0.20\%$	**$90.19 \pm 0.12\%$**
SHD	128-20	$33.84 \pm 1.35\%$	$47.37 \pm 1.65\%$	**$66.8 \pm 0.76\%$**

Table 5: Performances of several methods on the MNIST dataset. Results for Fast & Deep and my method are highlighted in bold.

	Method	Arch.	Test accuracy
TTFS	Fast & Deep (Göltz et al. 2021)	350	97.1 ± 0.1%
	Wunderlich & Pehle (Wunderlich and Pehle 2021)	350	97.6 ± 0.1%
	Alpha Synapses (Comsa et al. 2020)	340	97.96%
	S4NN (Kheradpisheh and Masquelier 2020)	400	97.4 ± 0.2%
	BS4NN (Kheradpisheh, Mirsadeghi and Masquelier 2021)	600	97.0%
	Mostafa (Mostafa 2016)	800	97.2%
	STDBP (Zhang et al. 2022)	800	98.5%
	Fast & Deep (Göltz et al. 2021) (my implementation)	**800**	**97.83 ± 0.08%**
Unconstrained	eRBP (Neftci et al. 2017a)	2x500	97.98%
	Lee et al (Lee, Delbruck and Pfeiffer 2016)	800	98.71%
	HM2-BP (Jin, Zhang and Li 2018)	800	98.84 ± 0.02%
	This work	**800**	**98.88 ± 0.02%**

Table 6: Performances of several methods on the EMNIST dataset. Results for Fast & Deep and my method are highlighted in bold.

	Method	Arch.	Test accuracy
TTFS	**Fast & Deep (Göltz et al. 2021) (my implementation)**	**800**	**83.34 ± 0.27%**
Unconstrained	eRBP (Neftci et al. 2017a)	2x200	78.17%
	HM2-BP (Jin, Zhang and Li 2018)	2x200	84.31 ± 0.10%
	HM2-BP (Jin, Zhang and Li 2018)	800	85.41 ± 0.09%
	This work	**800**	**85.75 ± 0.06%**

Table 7: Performances of several methods on the Fashion-MNIST dataset. Results for Fast & Deep and my method are highlighted in bold. The recurrence column indicates if the model contains recurrent connections.

	Method	Arch.	Reccurent	Test accuracy
TTFS	S4NN (Kheradpisheh and Masquelier 2020)	1000	✗	88.0%
	BS4NN (Kheradpisheh, Mirsadeghi and Masquelier 2021)	1000	✗	87.3%
	STDBP (Zhang et al. 2022)	1000	✗	88.1%
	Fast & Deep (Göltz et al. 2021) (my implementation)	**2x400**	✗	**88.28 $\pm$ 0.41%**
Unconstrained	HM2-BP (Jin, Zhang and Li 2018)	2x400	✗	88.99%
	TSSL-BP (Zhang and Li 2020)	2x400	✗	89.75 $\pm$ 0.03%
	ST-RSBP (Zhang and Li 2019)	2x400	✓	90.00 $\pm$ 0.14%
	This work	**2x400**	✗	**90.19 $\pm$ 0.12%**

Table 8: Network architectures used in Table 9. 15C5 represents a convolution layer with 15 5x5 filters and P2 represents a 2x2 pooling layer.

Network Name	Architecture
Net1	32C5-P2-16C5-P2-10
Net2	12C5-P2-64C5-P2-10
Net3	15C5-P2-40C5-P2-300-10
Net4	20C5-P2-50C5-P2-200-10
Net5	32C5-P2-32C5-P2-128-10
Net6	16C5-P2-32C5-P2-800-128-10

Table 9: Performances of several methods on the MNIST dataset with Spiking Convolutional Neural Networks. The network topologies are given in Table 8. The DA column indicates if the model has been trained with data augmentation to improve the performance.

	Method	Arch.	DA	Test accuracy
TTFS	Zhou et al (Zhou et al. 2021)	Net1	✓	99.33%
	STDBP (Zhang et al. 2022)	Net6	✓	99.4%
	Fast & Deep (my implementation)	**Net3**	✓	**99.22 $\pm$ 0.05%**
	Fast & Deep (my implementation)	**Net3**	✓	**99.46 $\pm$ 0.01%**
Unconstrainned	Lee et al (Lee, Delbruck and Pfeiffer 2016)	Net4	✓	99.31%
	HM2-BP (Jin, Zhang and Li 2018)	Net3	✓	99.42% $\pm$ 0.11%
	TSSL-BP (Zhang and Li 2020)	Net3	✗	99.50 $\pm$ 0.02%
	ST-RSBP (Zhang and Li 2019)	Net2	✓	99.50 $\pm$ 0.03%
	ST-RSBP (Zhang and Li 2019)	Net3	✓	99.57 $\pm$ 0.04%
	This work	**Net3**	✗	**99.38 $\pm$ 0.04%**
	This work	**Net3**	✓	**99.60 $\pm$ 0.03%**

Table 10: Performances of several methods on the Spiking Heidelberg Digits (SHD) dataset. Results for Fast & Deep and my method are highlighted in bold. The recurrence column indicates if the model contains recurrent connections.

	Method	Arch.	Recurrence	Test accuracy
TTFS	**Fast & Deep (Göltz et al. 2021) (my implementation)**	**128**	✗	**47.37 $\pm$ 1.65%**
Unconstrained	Cramer et al. (Cramer et al. 2022a)	128	✗	48.10 $\pm$ 1.6%
	Cramer et al. (Cramer et al. 2022a)	128	✓	71.4 $\pm$ 1.9%
	This work	**128**	✗	**66.79 $\pm$ 0.66%**

To assess the performance of the proposed method, I trained fully connected SNNs

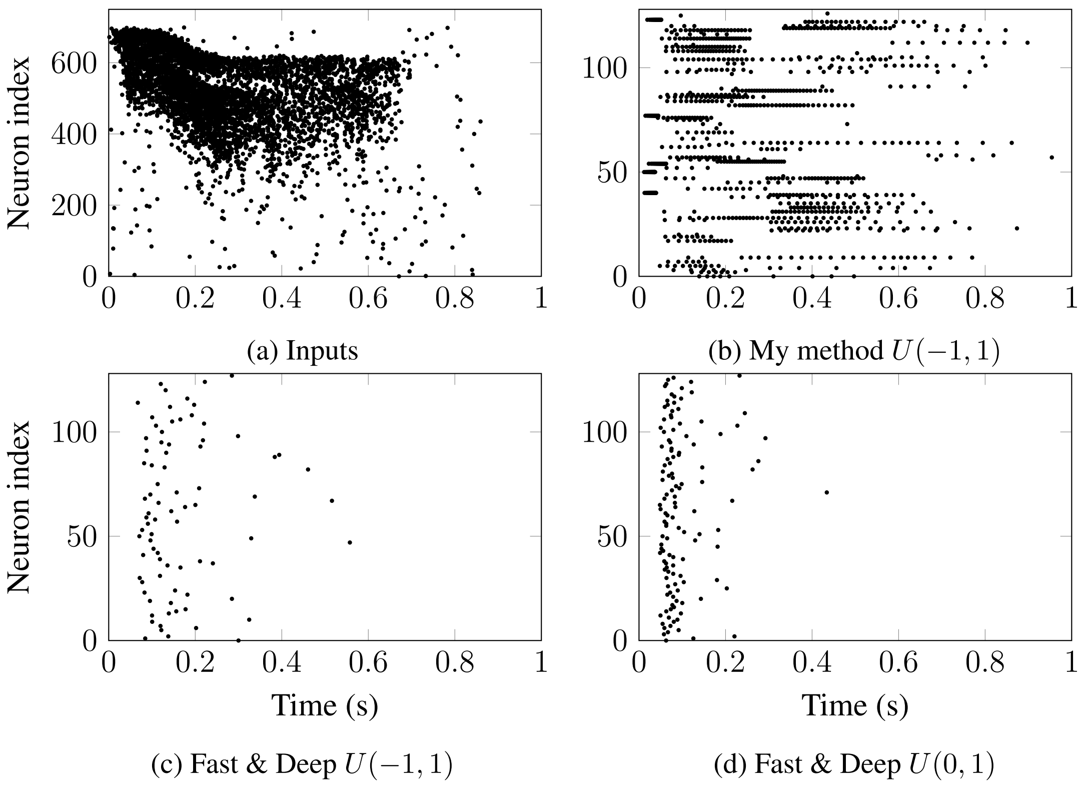


Figure 26: Spiking activity of hidden neurons in an SNN trained with my method (Figure 26b) and Fast & Deep (Figures 26c and 26d) given a spoken ”zero” (Figure 26a) from the SHD dataset. TTFS neurons in Fast & Deep mainly respond to early stimuli, missing most of the input information. In contrast, my method allows for multiple spikes per neuron which enables them to capture all the information from the inputs. This demonstrates the importance of relaxing the spike constraint of TTFS when processing temporal data.

on the MNIST (LeCun, Cortes and Burges 2010), EMNIST (Cohen et al. 2017), Fashion-MNIST (Xiao, Rasul and Vollgraf 2017) datasets the Spiking Heidelberg Digits (SHD) dataset (Cramer et al. 2022a) as well as Convolutional SNNs on MNIST. I compared my results to those obtained using the original Fast & Deep algorithm. Table 4 summarizes the average test accuracies of both methods given the considered initial weight distributions. For completeness, a comparison of Fast & Deep and my method with other spike-based BP algorithms can be found in Tables 5, 6, 7, 10 for each dataset.

Firstly, it should be noted that the Fast & Deep algorithm generally achieves better performance when the weights are initialized with positive values, which is consistent with the choice of weight distribution made by Göltz et al. (2021). Secondly, the proposed method demonstrates superior performance compared to the Fast & Deep algorithm, with improvement margins ranging from 1.05% on MNIST to up to 2.41% on the more difficult EMNIST dataset. This is not surprising given that unconstrained SNNs are known to perform better compared to TTFS networks (Jin, Zhang and Li 2018; Zhang and Li 2019; Shrestha and Orchard 2018; Lee, Delbruck and Pfeiffer 2016; Zhang et al. 2022). With random elastic deformation (Castro, Cardoso and Pereira 2018) data augmentation, my model is capable to improve its performance from 99.38% to 99.60% on MNIST, thus slightly outputforming the 99.57% achieved by ST-RSBP with the same architecture (see Figure 9). Moreover, my method outperforms by up to 20% Fast & Deep on the SHD dataset with a single hidden layer of 128 neurons. Note that no recurrent connection was used in these experiments. To understand the reason for this performance gap, I analyzed the spiking activity in the hidden layers after training. Figure 26 shows that TTFS neurons only respond to early stimuli. In temporal coding, high-valued information is encoded by early spikes. As demonstrated in Section 3.7.1, training with a one-spike constraint is energy efficient but tends to make SNNs spike as early as possible, thus missing the information occurring later. In contrast, neurons trained without spike constraint can respond throughout the duration of the sample, thus capturing all the information despite an increased total of spikes fired.

This highlights the importance of firing more than once and demonstrates that a tradeoff exists between performance and energy consumption when processing temporal data.

3.7.3 Convergence Rate

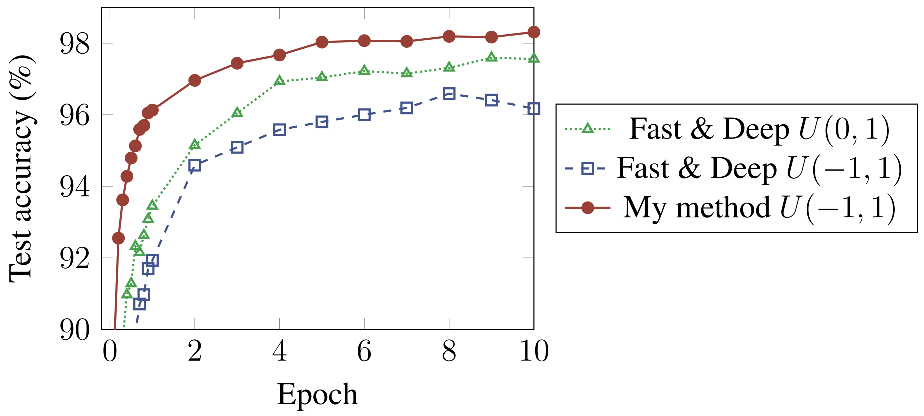


Figure 27: Evolution of the test accuracy of both Fast & Deep and my method on the MNIST dataset with the same learning rate. The unconstrained SNN trained with my method benefits from a higher convergence rate than the temporally-coded networks trained with Fast & Deep.

The rate of convergence, or *convergence rate*, measures how quickly the loss function is minimized by a training algorithm. To understand the difference in training speed of each algorithm, I measured in Figure 27 the evolution of the test accuracy of TTFS SNNs trained with Fast & Deep and an unconstrained SNN trained with my method on the MNIST dataset.

We can observe that the temporally-coded SNN initialized with positive weights converges faster than the TTFS network initialized with both negative and positive weights. This suggests that the initialization of weights in temporally-coded SNNs not only influences their performance but also their convergence rate.

In contrast, the unconstrained SNN trained with my method converges significantly faster than both TTFS SNNs. This shows that my method is capable of reaching desired accuracies in fewer epochs compared to Fast & Deep. This also suggests that there is

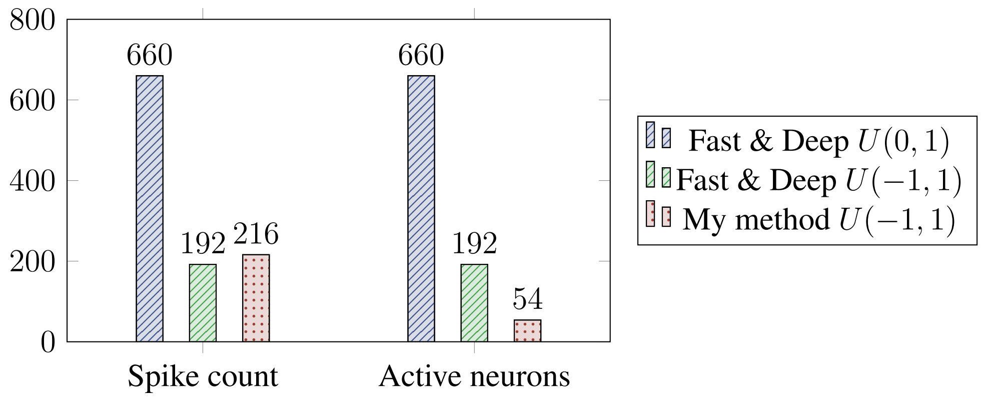


Figure 28: Comparison of the population spike count and the number of active neurons in fully connected SNNs trained using Fast & Deep and the proposed method on the MNIST dataset. These results indicate that the sparsity of SNNs after training depends on the weight distribution, with the SNN initialized with only positive weights appearing to be less sparse than those initialized with both negative and positive weights. Additionally, the proposed method demonstrates a similar level of sparsity and fewer active neurons as Fast & Deep for the same initial weight distribution, despite the relaxed constraint on neuron spike counts.

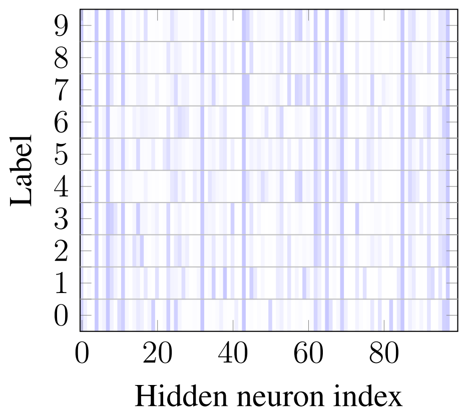
a direct correlation between the convergence rate and performance and implies that discriminative features are likely learned earlier during training.

3.7.4 Sparsity

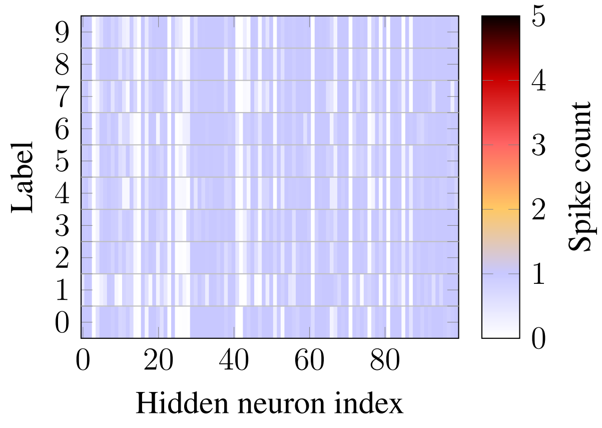
Achieving high sparsity in trained SNNs is critical for energy efficiency, as neuromorphic hardware only consumes energy at spike events.

Figure 28 shows the population spike counts of fully connected SNNs trained on MNIST using both methods. It appears that the initial weight distribution plays an important role in the final sparsity level of the trained SNNs. In the analyzed case, SNNs initialized with positive weights appear to be less sparse after training than SNNs initialized with both negative and positive weights.

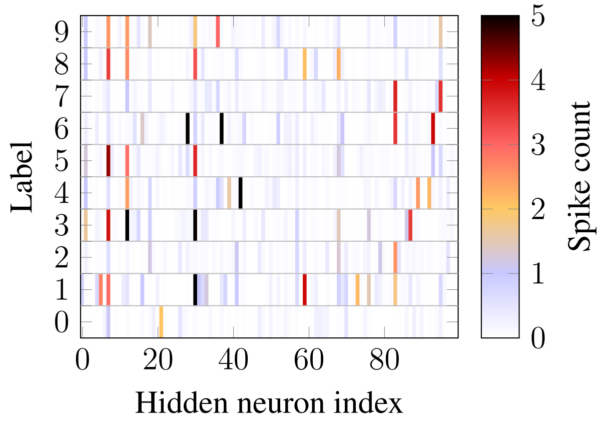
While the proposed method allows for an increased number of spikes per neuron, which implies more energy consumption, I found that it can achieve similar sparsity as



(a) Fast & Deep $U(-1,1)$

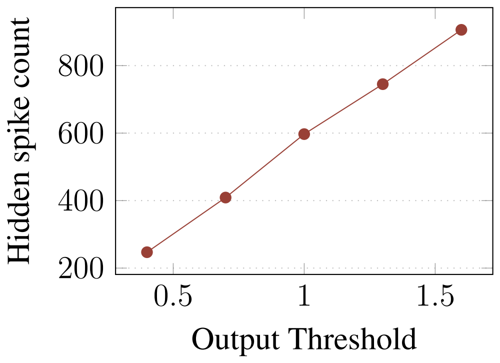


(b) Fast & Deep $U(0,1)$

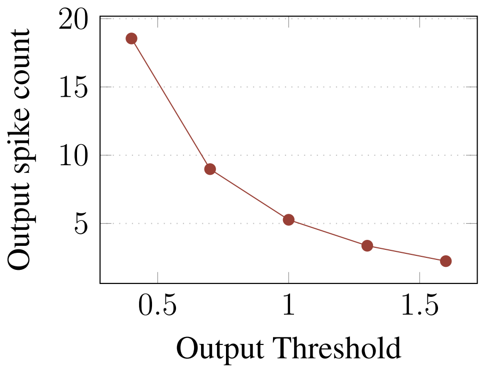


(c) My method $U(-1,1)$

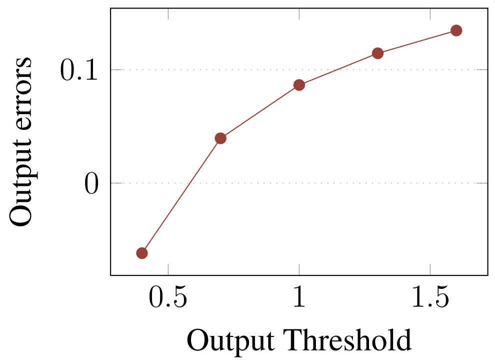
Figure 29: Figures 29a and Figure 29b show the average spike count of hidden neurons trained with Fast & Deep on the MNIST dataset, while Figure 29c shows the average spike count of hidden neurons trained with the proposed method. Each row corresponds to the average activity over all the test samples of a particular digit. As TTFS networks mainly encode information temporally, we observe that neurons trained with Fast & Deep fire indiscriminately in response to stimuli, making it difficult to differentiate the labels from the mean spike count, regardless of the initial weight distribution. However, the proposed SNN training method results in a different distribution of firing activity. More precisely, key neurons respond selectively to particular digits, while most of the other neurons remain mostly silent.



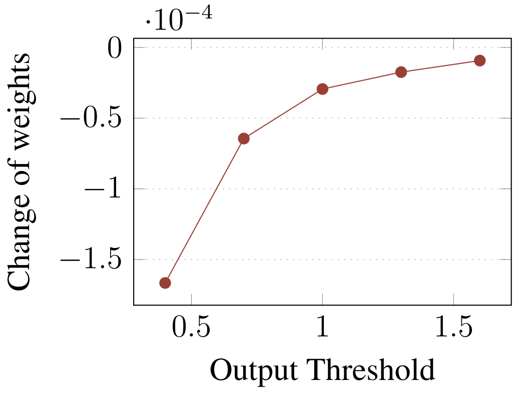
(a)



(b)



(c)



(d)

Figure 30: Influence of the output threshold on the sparsity of a 2-layer SNN trained on MNIST with my method. Figure 30a illustrates that a lower output threshold results in fewer spikes generated after 1 epoch. Figure 30b indicates that decreasing the output threshold increases the initial activity in the output layer, thereby leading to a greater number of negative errors transmitted during the backward pass (as shown in Figure 30c). This, in turn, leads to a decrease in the weights in the hidden layer, as depicted in Figure 30d.

TTFS networks initialized with both negative and positive weights while performing better than TTFS networks initialized with positive weights only (see Figure 28 and Table 4). This suggests that the proposed method can offer improved tradeoffs between accuracy and sparsity.

To understand why my method can achieve such levels of sparsity despite not imposing any constraints on neuron firing, I analyzed the average activity in each network. Figure 29 shows that neurons trained with Fast & Deep fire indiscriminately in response to any input digit, which is characteristic of temporal coding where information is represented by the timing of spikes, rather than the presence or absence of spikes. In contrast, SNNs trained with my method exhibit a different distribution of firing activity, with certain key neurons selectively responding to specific digits. Figure 28 indicates that only 7% of neurons trained with my method are active during inference (i.e. neurons that fire at least once). In comparison, the SNN trained with Fast & Deep and initialized with both negative and positive weights has 24% of its neurons firing. Therefore, the reduced proportion of active units in my method compensates for the increased number of spikes per neuron, leading to fewer spikes emitted in the network.

I found during my experiments that the sparsity of SNNs trained with my method was also influenced by the choice of threshold values. More precisely, we observed that decreasing the output threshold resulted in a reduction of activity in the network, as illustrated in Figure 30a. This decrease in activity occurred because the output neurons fired more often, which is illustrated in Figure 30b. The increased number of output spikes caused the loss function to produce more negative errors, which resulted in negative changes in weights in the hidden layer. However, lowering the threshold has a dual impact on activity: it increases the firing rates at initialization but also contributes to producing more negative errors, which can decrease activity during learning. While controlling sparsity using thresholds is trivial in shallow SNNs, a fine balance between thresholds has to be found to control sparsity in Deep SNNs, making it challenging to

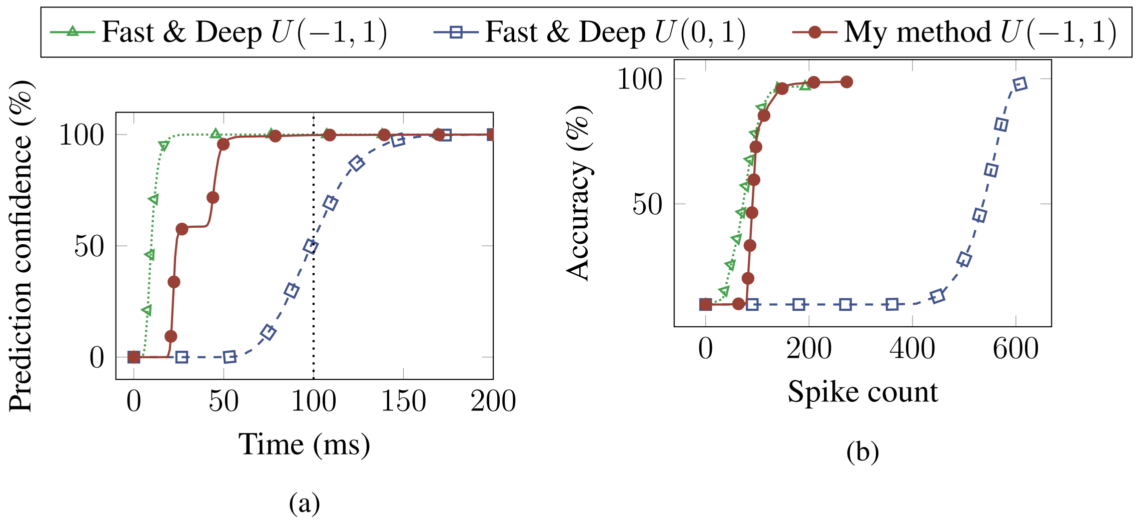


Figure 31: Figure 31a shows the evolution of the average prediction confidence during simulations on the MNIST test set. To produce this figure, I measured the probability of predictions at a time t being equal to the final predictions at the end of the simulations. The vertical dotted line represents the end of input spikes. The initial weight distribution seems to have a crucial impact on the latency of predictions. More precisely, negative initial weights produce confidence earlier than positive initial weights. Therefore, simulation time can be reduced to further improve sparsity. Figure 31b shows the relationship between spike count and accuracy as the simulation time increases. It demonstrates that the duration of simulation can be used as a post-training method to further reduce energy consumption while maintaining high performance.

achieve sparsity with larger architectures. However, my findings suggest that threshold values play a crucial role in determining the sparsity of unconstrained SNNs and can be seen as a way to control activity without the need for firing rate regularization.

3.7.5 Prediction Latency

I also investigated the prediction latency of SNNs trained using different methods. This corresponds to the amount of simulation time needed by the models to reach full confidence in their predictions and determines the simulation duration required to achieve high accuracy.

Figure 31a shows the averaged prediction confidence over time for SNNs trained using each method. I found that when initialized with only positive weights, Fast &

Deep achieved high confidence on predictions after more than 150ms. However, when initialized with both negative and positive weights, both Fast & Deep and the proposed method achieved confidence earlier (in 20ms and 50ms, respectively). This suggests that initialization has a significant impact on prediction latency. However, SNNs trained with the proposed method are slightly slower than TTFS networks trained with Fast & Deep for the same weight distribution due to the increased spike count per neuron.

Despite this difference, the short latency of these networks allows for shorter simulation durations, which can further improve sparsity without affecting performance. In Figure 31b, I show the relationship between population spike count and accuracy for each SNN as simulation time increases from 0 to 200 milliseconds. This demonstrates that, by reducing the simulation time, SNNs can become more sparse while maintaining high performance. Therefore, the sparsity-accuracy tradeoff can be further improved after training by adjusting the simulation duration. This also demonstrates that my method can align with the level of the sparsity of Fast & Deep while still performing better.

3.7.6 Robustness to Noise and Weight Quantization

Analog neuromorphic hardware are inherently noisy and are often limited to specific ranges and resolutions of weights. Having a model that is robust to noise and weight quantization is therefore important to achieve high performance on such systems. To assess the robustness of each method, I measured the impact of spike jitter, weight clipping, and weight precision on their accuracy. In these experiments, performance was normalized with the maximum accuracy of each method to better compare variations.

Figure 32a shows the impact of spike jitter on the performance of each method. These results were produced by artificially adding noise to spike timings with a normal distribution $\mathcal{N}(0, \sigma)$ during training. When initialized with both negative and positive weights, Fast & Deep appears to be less robust than using only positive weights. With negative weights, only a fraction of neurons transmit information which leads to an

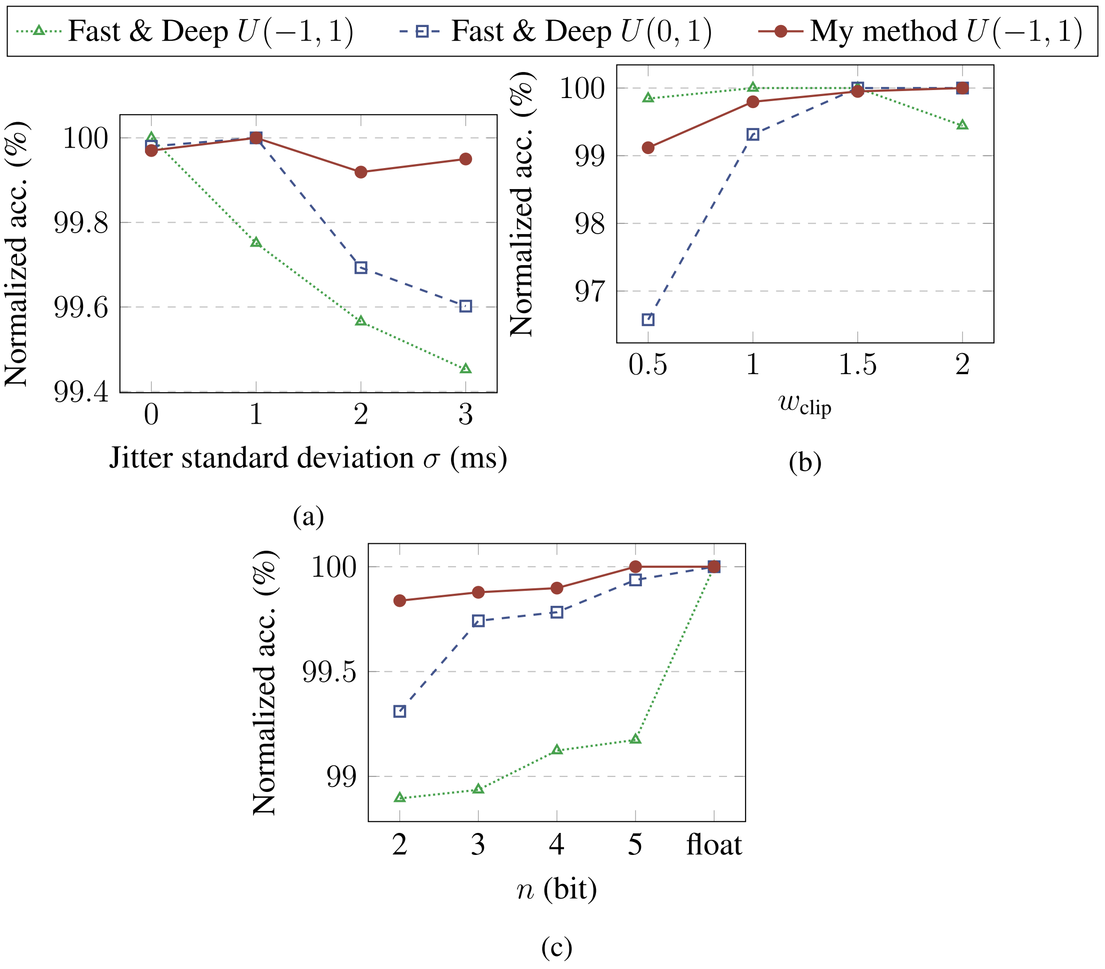


Figure 32: Figure 32a shows the effect of spike jitter on the performance of each method. This was achieved by introducing artificial noise to the spike timings, following a normal distribution $\mathcal{N}(0, \sigma)$. Figure 32b displays the impact of weight clipping, which involved restricting weights to the range $[-w_{\text{clip}}, w_{\text{clip}}]$ during training. Lastly, Figure 32c demonstrates the effect of weight precision, which was obtained by discretizing weights into $2^{n+1} - 1$ bins (n bits plus one bit for the sign of the synapse) within the range $[-1, 1]$. Overall, my method was found to be more resilient to noise and reduced weight precision than Fast & Deep.

increased sparsity, as illustrated in Figure 28. Therefore, introducing noise to spike timings significantly impacts performance. In contrast, positive weights ensure consistent network activity and redundancy in transmitted information. SNNs initialized with positive weights are thus less affected by spike jitter. However, SNNs trained with Fast & Deep remain susceptible to noise, even with positive initial weights. Perturbations in spike timing still have a critical impact on temporal coding. In contrast, the proposed method demonstrates greater robustness to spike jitter than Fast & Deep, with minimal variation observed. This is a result of the redundancy created by the multiple spikes fired by neurons.

In Figure 32b, I demonstrate the effect of weight range on performance by clipping weights between the range $[-w_{\text{clip}}, w_{\text{clip}}]$ during training. The performance of Fast & Deep initialized with positive weights degrades when w_{clip} is lower than 1.5. However, both my method and Fast & Deep exhibit robustness to reduced weight ranges when initialized with both negative and positive weights. This suggests that weight distribution may play a role in the network's resilience to limited weight ranges.

Finally, Figure 32c shows the performance of each method with reduced weight resolutions from 2 to 5 bits (results with float precision are also given as a reference). It highlights that Fast & Deep is less robust to reduced weight precision than my method, particularly with negative weights. In contrast, my approach is only slightly impacted by the decreased precision, even when reduced to as low as 2 bits.

3.8 Discussion

In this chapter, I proposed an extension of the Fast & Deep algorithm that relaxes the constraint on firing in TTFS and generalizes the computation of exact gradients of SNNs. I explored how the initial weight distribution affects the tradeoffs between performance and various other aspects of SNNs such as sparsity, classification latency, and robustness to noise and weight quantization. I finally compared the improvements of the proposed method with the original Fast & Deep algorithm on those tradeoffs.

First, I found that initializing Fast & Deep with positive weights leads to better generalization capabilities compared to initializing with both negative and positive weights. This observation is consistent across the benchmarked datasets, as shown in Table 4. However, relaxing the spike constraint improves the overall performance and convergence rate of SNNs, at least on the benchmark problems I considered. This result was expected before my experiments since BP methods that use multiple spikes per neuron generally perform better than methods that impose firing constraints (Jin, Zhang and Li 2018; Zhang and Li 2019; Shrestha and Orchard 2018; Lee, Delbruck and Pfeiffer 2016; Zhang et al. 2022).

My experiments also demonstrate that the weight distribution significantly influences the sparsity of Fast & Deep. We observed that SNNs with positive weight initialization tend to be less sparse than those initialized with weights between -1 and 1. However, the former consistently outperforms the latter in terms of performance. This highlights the accuracy-sparsity tradeoff often observed when training SNNs (Yin et al. 2023; Li et al. 2021). The quasi-dense activity provided by positive weights explains the difference in sparsity, as shown in Figure 29b. In contrast, initializing Fast & Deep with both negative and positive weights leads to fewer active neurons due to the inhibition provided by negative weights. Additionally, neurons trained with Fast & Deep fire indiscriminately to stimuli, suggesting a pure temporal representation of information, whereas neurons trained with the proposed method selectively respond to their inputs and exhibit a different distribution of activity, as shown in Figure 29c. the proposed method allows for a different distribution of the spike activity, whereby key neurons can fire more often than others, while irrelevant neurons may not spike at all. This enables my method to achieve a level of sparsity comparable to Fast & Deep, as illustrated in Figure 28.

To achieve a high degree of sparsity without firing rate regularization, thresholds can be tuned to indirectly influence spiking activity through learning. Decreasing thresholds increases the firing rate of downstream layers, resulting in more negative errors at outputs and consequently, negative weight changes in hidden layers, as shown in Figure 30. This mechanism offers a natural way to control sparsity in unconstrained SNNs without requiring any regularization technique. By leveraging this principle, I was able to train shallow SNNs with a sparsity level similar to Fast & Deep while achieving higher performance. This implies that allowing multiple spikes per neuron has the potential to enhance the accuracy-sparsity tradeoff and prompts further investigation into the effectiveness of TTFS in achieving efficient computation. However, finding thresholds that lead to high sparsity is more difficult when networks become deeper due to the fluctuations in the firing rates of each layer. The factors that influence sparsity in unregularized SNNs are currently not fully understood and present an opportunity for future research to investigate how to naturally achieve sparsity in deep architectures.

In addition to performance and sparsity, I measured the prediction latency of each method, which is the waiting time required before the system can make reliable predictions. I found that the speed of classification was primarily driven by the weight distribution. Figure 31a shows that both Fast & Deep and my method achieve similar latencies when initialized between -1 and 1, with a slight advantage for Fast & Deep. However, when initialized with only positive values, Fast & Deep requires more simulation time to achieve full confidence in predictions. Low latency is advantageous not only in terms of inference speed but also in improving energy efficiency. If predictions occur early enough, the duration of simulations can be significantly reduced, limiting the number of spikes fired by neurons. In Figure 31b, I demonstrated that reducing the simulation time can lead to a reduction in computational cost for both Fast & Deep and my method while maintaining the same performance. This shows that prediction latency, energy consumption, and performance are closely related and that unconstrained SNNs trained with the proposed method can offer better tradeoffs between these aspects than TTFS SNNs trained with Fast & Deep. By reducing the latency, the sparsity of SNNs can therefore be increased, leading to potential gains in energy efficiency on neuromorphic hardware.

The final characteristic that I investigated is the robustness to noise and weight quantization that are inherent to analog neuromorphic hardware. The timing of spikes is a critical factor for the performance of TTFS SNNs as it carries most of the information. Therefore, perturbations in these timings and weight constraints can significantly affect the reliability of the feature extraction. In contrast, the proposed method benefits from an increased number of spikes per neuron, providing redundancy that enhances resilience to noise and weight constraints. For instance, Figure 32a demonstrates that my method is less impacted by perturbations in spike timings than Fast & Deep. This suggests that the proposed method has the potential to provide more stable learning on analog neuromorphic hardware than Fast & Deep.

Chapter 4

Low-Variance Gradient Estimates without Backpropagation

The contributions of this chapter are:

- I propose the *Forward Direct Feedback Alignment* algorithm, a method that uses *Activity-Perturbed Forward Gradients* and *Momentum* to estimate the derivatives between output and hidden neurons as direct feedback connections and train DNNs without performing error backpropagation.
- I analytically demonstrate that the momentum introduced by the proposed method quadratically reduces the estimation variance of the gradient. I show that, compared to other *Forward Gradient methods*, my algorithm produces gradient estimates that exhibit significantly lower variance, thus improving the rate of convergence.
- I empirically validate my theoretical results, demonstrating the effectiveness of the FDFA algorithm in reducing the variance of gradient estimates and enabling faster convergence than other *Forward Gradient* methods.
- I compare the FDFA algorithm with *Forward Gradient* and *Direct Feedback*

Alignment methods and demonstrate performance that my algorithm achieves performance that is closer to BP with both DNNs and CNNs.

- I empirically demonstrate that gradient estimates of FDFA better follow the direction of steepest descent than random *Direct Feedback Alignment* and the *Direct Kolen-Pollack* algorithms, thus explaining the differences in convergence rate observed in my experiments.

4.1 Introduction

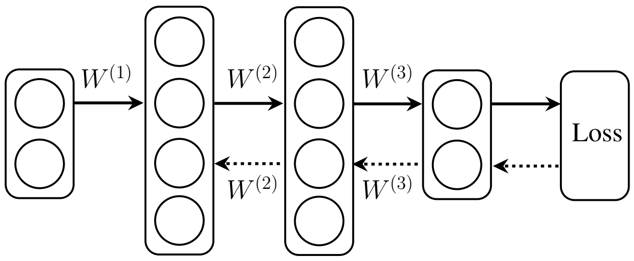


Figure 33: Illustrations of the error backpropagation (Figure 33) in a DNN. Solid arrows represent forward inference paths and dotted arrows represent error backpropagation paths. During the backward pass, errors are sequentially backpropagated to hidden layers.

The error Backpropagation (BP) algorithm is the most widely used method to compute the gradients of neural networks. Despite its success, BP has certain limitations that restrict its efficiency, scalability, and, more importantly, its applicability to neuromorphic hardware.

First, the sequential propagation of errors in BP — see Figure 33 — significantly constrains the parallelization of the backward pass and limits the full exploitation of computing resources. Often referred to as *Backward Locking* (Nøkland 2016; Launay et al. 2020; Huo et al. 2018), this problem inherently leads to time-consuming gradient computation and strongly limits the scalability of BP.

Second, during the backward pass, the BP algorithm reuses the (forward) weights of the network to propagate errors to upstream layers. Naively, this can be implemented by delivering error signals through a decoupled feedback circuit that has the same weights as the forward connections (Lillicrap et al. 2020). However, this approach requires the weights to be transported from the forward to the feedback network before propagating errors. Known as the *weight transport problem*, this is the main reason for the biological implausibility of BP (Grossberg 1987; Lillicrap et al. 2020, 2016; Nøkland 2016; Akrout et al. 2019) and represents one of the major sources of energy consumption during learning on DNN accelerators due to the costly transports of information (Kwon et al. 2020).

Finally, when applied to SNNs, BP backpropagates errors backward through time and space, requiring saving spike events in memory and unfolding neuron dynamics during the backward pass. However, the biological constraints inherent to neuromorphic hardware only allow learning rules that use information that is locally available to the neurons at a given time and location. As a result, such hardware is unable to propagate information backward through time and space, making BP impractical for neuromorphic learning.

The limitations of the BP algorithm thus represent major challenges for the efficient training of neural systems. As such, there is a growing need for local, parallel, scalable alternatives to BP that could train neural systems without the need for global knowledge of the network. More importantly, the development of such algorithms could support the design of novel learning rules compatible with neuromorphic hardware.

In this context, many alternatives to BP have been proposed (Hjelm et al. 2019; Löwe, O' Connor and Veeling 2019; Belilovsky, Eickenberg and Oyallon 2019; Nøkland and Eidnes 2019; Mostafa, Ramesh and Cauwenberghs 2018; Jaderberg et al. 2017; Hinton 2022; Jabri and Flower 1992; Le Cun, Galland and Hinton 1988; Wen et al. 2018; Baydin et al. 2022; Silver et al. 2021; Ren et al. 2023; Lillicrap et al. 2016; Nøkland 2016; Webster, Choi and Ahn 2021; Han and jun Yoo 2019; Crafton et al.

2019). Among these alternatives, feedback-based learning methods, such as *Direct Feedback Alignment* (Nøkland 2016), and techniques using Forward-Mode Automatic Differentiation (Forward-Mode AD) — see Section 2.5.2 for details —, such as *Forward Gradients* (Baydin et al. 2022; Silver et al. 2021; Ren et al. 2023), have recently gained attention.

Forward Gradient algorithms (Baydin et al. 2022) represent promising alternatives to BP. They are capable of computing unbiased gradient estimates of DNNs along with the forward pass without requiring error backpropagation, thus overcoming many of the limitations associated with the BP algorithm. However, in large DNNs, Forward Gradients suffer from high variance which has a detrimental effect on convergence. (Ren et al. 2023; Silver et al. 2021). While SGD is guaranteed to converge for unbiased gradient estimators with sufficiently small learning rates (Robbins and Monro 1951), theoretical analyses have shown that its convergence rate directly depends on the variance of the gradient estimates (Bubeck 2015; Bottou, Curtis and Nocedal 2018; Murata 1999; Moulines and Bach 2011; Needell, Srebro and Ward 2016; Chee and Toulis 2018; Gower et al. 2019; Faghri et al. 2020). Therefore, gradient estimators with low variance have less variability and are more consistent, leading to better convergence than those with high variance.

One approach for reducing the variance of Forward Gradients is to perturb neuron activations instead of weights, as in Activity-Perturbed Forward Gradients (Ren et al. 2023). Because DNNs typically have fewer neurons than weights, perturbing neurons result in estimating fewer derivatives through forward gradients, leading to lower variance. While this approach reduces the variance of the estimates, the large number of neurons in DNNs still affects convergence. As a solution, local greedy loss functions can be used to further reduce the variance, where each loss function trains only a small portion of the network. However, this method circumvents large variances by ensuring that each local loss function only trains a small number of neurons. In the case where layers contain large numbers of neurons — as in convolutional layers —, the method still suffers from high variance. Therefore, alternative solutions should be found to better reduce the variance of forward gradients in DNNs.

In this chapter, I propose the Forward Direct Feedback Alignment (FDFA) algorithm, a method that combines Activity-Perturbed Forward Gradient (Ren et al. 2023) with Direct Feedback Alignment (Nøkland 2016) and momentum to compute low-variance gradient estimates. My method addresses the limitations of BP by avoiding sequential backpropagation of error, solving the weights transport problem, and computing low-variance gradient estimates in a way that could be compatible with neuromorphic hardware. The rest of this chapter is structured as follows:

- I first review the Forward Gradient algorithm (Baydin et al. 2022) and its DNN applications (Silver et al. 2021; Ren et al. 2023) as well as Direct Feedback Alignment (Nøkland 2016), which form the technical foundation of the FDFA algorithm.

- I then describe the proposed FDFA algorithm that uses forward gradients to estimate derivatives between output and hidden neurons as direct feedback connections.

- I provide theoretical results showing that the gradient estimates computed by the proposed FDFA method exhibit lower variance than other Forward Gradient methods.

- I describe the experimental settings used to produce my empirical results.

- Finally, I provide empirical results demonstrating the effectiveness of FDFA in reducing the variance of gradient estimates and enabling fast convergence with DNNs. I show that, compared to other Forward Gradient and Direct Feedback Alignment methods, FDFA achieves better performance with both fully connected and convolutional neural networks.

4.2 Forward Gradient

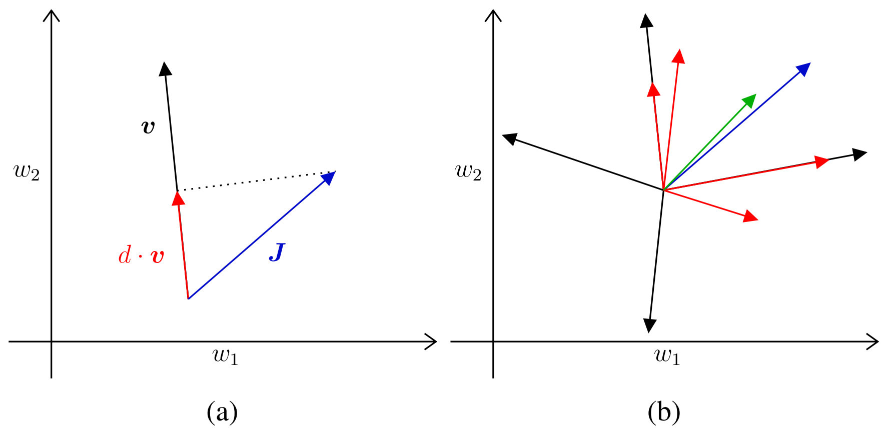


Figure 34: Figure 34a: Projection of the Jacobian J at $\boldsymbol{w}$ onto a given direction $\boldsymbol{v}$. The vector $d \cdot \boldsymbol{v}$ is obtained by scaling the direction $\boldsymbol{v}$ by the directional derivative d evaluated at $\boldsymbol{w}$ in the direction of $\boldsymbol{v}$. Figure 34b: The expected directional derivative (green arrow), computed by averaging directional gradients (red arrows) over many random directions (black arrows), is an unbiased estimate of the true gradient (blue arrow).

The Forward Gradient (FG) algorithm (Baydin et al. 2022; Silver et al. 2021) is a recent weight perturbation technique that uses Forward-Mode AD (Margossian 2019) — see Section 2.5.2 — to estimate gradients without backpropagation. Consider a differentiable function $\boldsymbol{f} : \mathbb{R}^m \mapsto \mathbb{R}^n$ and a vector $\boldsymbol{v} \in \mathbb{R}^m$, Forward-Mode AD evaluates the directional gradient $\boldsymbol{d} = \boldsymbol{J} \cdot \boldsymbol{v}$ of $\boldsymbol{f}$ in the direction $\boldsymbol{v}$. Here, $\boldsymbol{J} \in \mathbb{R}^{n \times m}$ is the Jacobian matrix of f, and $\boldsymbol{d}$ is obtained by computing the matrix-vector product between $\boldsymbol{J}$ and $\boldsymbol{v}$ during the function evaluation.

By sampling each element of the direction vector $v_i \sim \mathcal{N}(0, 1)$ from a standard normal distribution and multiplying back each computed directional derivative by $\boldsymbol{v}$, an unbiased estimate of the Jacobian is computed:

$$\mathbb{E}\left[\boldsymbol{d} \otimes \boldsymbol{v}\right] = \mathbb{E}\left[(\boldsymbol{J} \cdot \boldsymbol{v}) \otimes \boldsymbol{v}\right] = \boldsymbol{J} \tag{95}$$

Here, $\otimes$ is the outer product. See (Baydin et al. 2022) or Theorem 2 for the proof of unbiasedness and Figure 34 for a visual representation of forward gradients.

Theorem 2. *Unbiasedness of Forward Gradients (Baydin et al. 2022) Let $\boldsymbol{x} \in \mathbb{R}^n$ be a vector of size n and $\boldsymbol{v} \in \mathbb{R}^n$ be a random vector of n independent variables. If $\boldsymbol{v} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ follows a multivariate standard normal distribution, then:*

$$\mathbb{E}\left[(\boldsymbol{x} \cdot \boldsymbol{v})\, \boldsymbol{x}\right] = \boldsymbol{x} \tag{96}$$

Proof. Focusing on the i^{th} element, we have:

$$\begin{aligned}
\mathbb{E}\left[(\boldsymbol{x} \cdot \boldsymbol{v})\, v_i\right] &= \mathbb{E}\left[\sum_{j=1}^{n} x_j v_j v_i\right] \\
&= \mathbb{E}\left[x_i v_i^2\right] + \sum_{\substack{j=1 \\ j \neq i}}^{n} \mathbb{E}\left[x_j v_j v_i\right] \\
&= x_i \mathbb{E}\left[v_i^2\right] + \sum_{\substack{j=1 \\ j \neq i}}^{n} x_j \mathbb{E}\left[v_j\right] \mathbb{E}\left[v_i\right]
\end{aligned} \tag{97}$$

However, we know that $v_i \sim \mathcal{N}(0, 1)$. Therefore, $\mathbb{E}[v_i] = 0$ and $\mathbb{E}[v_i^2] = \mathbb{E}[v_i]^2 + \mathrm{Var}[v_i] = 1$. Using these properties, Equation 97 reduces to:

$$\mathop{\mathbb{E}}_{\boldsymbol{x}, \boldsymbol{v}}\left[(\boldsymbol{x} \cdot \boldsymbol{v})\, v_i\right] = x_i \tag{98}$$

and

$$\mathop{\mathbb{E}}_{\boldsymbol{x}, \boldsymbol{v}}\left[(\boldsymbol{x} \cdot \boldsymbol{v})\, \boldsymbol{v}\right] = \boldsymbol{x} \tag{99}$$

which concludes the proof. □

4.3 Weight-Perturbed Forward Gradient

When the FG algorithm is applied to DNNs, a random perturbation matrix $\boldsymbol{V}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ is drawn from a standard normal distribution for each weight matrix $\boldsymbol{W} \in$

Algorithm 3: Weight-Perturbed Forward Gradient algorithm (Baydin et al. 2022) with a fully-connected DNN.

```
1: **Input:** Training data $\mathcal{D}$
2: Randomly initialize $w_{ij}^{(l)}$ for all $l$, $i$ and $j$.
3: **repeat**
4:    $\{$Inference (sequential)$\}$
5:    **for** $\boldsymbol{x}$ **in** $\mathcal{D}$ **do**
6:       $\boldsymbol{y}^{(0)} \leftarrow \boldsymbol{x}_s$
7:       $\boldsymbol{d}^{(0)} \leftarrow \mathbf{0}$
8:       **for** $l = 1$ **to** $L$ **do**
9:          Sample $\boldsymbol{V}^{(l)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
10:         $\boldsymbol{a}^{(l)} \leftarrow \boldsymbol{W}^{(l)} \boldsymbol{y}^{(l-1)}$
11:         $\boldsymbol{y}^{(l)} \leftarrow f\left(\boldsymbol{a}^{(l)}\right)$
12:         $\boldsymbol{d}^{(l)} \leftarrow \left(\boldsymbol{W}^{(l)} \boldsymbol{d}^{(l-1)} + \boldsymbol{V}^{(l)} \boldsymbol{y}^{(l-1)}\right) \odot \sigma'\left(\boldsymbol{a}^{(l)}\right)$
13:      **end for**
14:      $d \leftarrow \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(L)}} \boldsymbol{d}^{(L)}$
15:      $\{$Weights updates (parallel)$\}$
16:      **for** $l = 1$ **to** $L$ **do**
17:         $\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \lambda \boldsymbol{V}^{(l)} d$
18:      **end for**
19:   **end for**
20: **until** $\mathbb{E}\left[\mathcal{L}(\boldsymbol{x})\right] < \epsilon$
```

$\mathbb{R}^{n_l \times n_{l-1}}$ of each layer $l \leq L$. Because weights are perturbed, the algorithm is thus referred to as *Weight-Perturbed Forward Gradient* (FG-W) (Ren et al. 2023). By doing so, Forward-Mode AD computes the directional derivative d^{FG} of the loss $\mathcal{L}(\boldsymbol{x})$ along the drawn perturbation, such as:

$$d^{\text{FG-W}} = \sum_{l=1}^{L} \sum_{i=1}^{n_l} \sum_{j=1}^{n_{l-1}} \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(l)}} v_{i,j}^{(l)} \tag{100}$$

where $v_{i,j}^{(l)}$ is the element of the perturbation matrix $V^{(l)}$ in row i and column j, associated with the weight $w_{i,j}^{(l)}$.

Thus refered to as Weight-Perturbed Forward Gradient (FG-W) (Ren et al. 2023), the gradient estimate $g^{\text{FG}-\text{W}}(\boldsymbol{W}^{(l)})$ for the weights of the layer l is obtained by scaling its perturbation $\boldsymbol{V}^{(l)}$ matrix with the directional derivative $d^{\mathrm{FG-W}}$:

$$g^{\mathrm{FG-W}}\left(\boldsymbol{W}^{(l)}\right) := d^{\mathrm{FG-W}}\boldsymbol{V}^{(l)} \tag{101}$$

See Algorithm 3 for the full algorithm applied to a fully connected DNN.

It is essential to understand that the directional derivative $d^{\mathrm{FG-W}}$, takes the form of a *scalar*. This scalar is defined as the result of the vector product between the true gradient and the given direction that is implicitly evaluated with Forward-Mode AD. Unlike Reverse-Mode AD, where the individual explicit derivatives constituting the gradient are accessible, Forward-Mode AD does not provide them directly. Therefore, multiplying the directional derivative by the given direction is an essential step for computing an unbiased approximation of the gradient. For more details, refer to (Baydin et al. 2022).

However, it has been previously shown that the variance of FG-W scales poorly with the number of parameters in DNNs which impacts the convergence of SGD (Ren et al. 2023).

4.4 Activity-Perturbed Forward Gradient

To address the variance issues of Weight-Perturbed FG, Ren et al. proposed the *Activity-Perturbed FG* (FG-A) algorithm that perturbs neurons activations instead of weights. A perturbation vector $\boldsymbol{u}^{(l)} \in \mathbb{R}^{n_l}$ is drawn from a multivariate standard normal distribution for each layer $l \leq L$. Forward-Mode AD thus computes the following directional derivative:

$$d^{\mathrm{FG-A}} = \sum_{l=1}^{L}\sum_{i=1}^{n_l} \frac{\partial \mathcal{L}\left(\boldsymbol{x}\right)}{\partial y_i^{(l)}} u_i^{(l)} \tag{102}$$

where $\boldsymbol{y}^{(l)}$ are the activations of the l^{th} layer. Note that directional derivatives computed in the FG-A algorithm are defined as the sum of all neurons rather than all weights. The

Algorithm 4: Activity-Perturbed Forward Gradient algorithm (Ren et al. 2023) with a fully-connnected DNN.

```
1: **Input:** Training data $\mathcal{D}$
2: Randomly initialize $w_{ij}^{(l)}$ for all $l$, $i$ and $j$.
3: **repeat**
4:   $\{$Inference (sequential)$\}$
5:   **for all** $x$ **in** $\mathcal{D}$ **do**
6:     $\boldsymbol{y}^{(0)} \leftarrow \boldsymbol{x}_s$
7:     $\boldsymbol{d}^{(0)} \leftarrow \boldsymbol{0}$
8:     **for** $l = 1$ **to** $L$ **do**
9:       Sample $\boldsymbol{v}^{(l)} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
10:      $\boldsymbol{a}^{(l)} \leftarrow \boldsymbol{W}^{(l)} \boldsymbol{y}^{(l-1)}$
11:      $\boldsymbol{y}^{(l)} \leftarrow \sigma\left(\boldsymbol{a}^{(l)}\right)$
12:      $\boldsymbol{d}^{(l)} \leftarrow \left(\boldsymbol{W}^{(l)} \boldsymbol{d}^{(l-1)}\right) \odot \sigma'\left(\boldsymbol{a}^{(l)}\right)$
13:      **if** $l < L$ **then**
14:        $\boldsymbol{d}^{(l)} \leftarrow \boldsymbol{d}^{(l)} + \boldsymbol{v}^{(l)}$
15:      **end if**
16:    **end for**
17:    $d \leftarrow \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(L)}} \boldsymbol{d}^{(L)}$
18:    $\{$Weights updates (parallel)$\}$
19:    **for** $l = 1$ **to** $L$ **do**
20:      $\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \lambda\left(d^{(L)} \boldsymbol{v}^{(l)} \odot \sigma'\left(\boldsymbol{a}^{(l)}\right)\right) \otimes \boldsymbol{y}^{(l-1)}$
21:    **end for**
22:  **end for**
23: **until** $\mathbb{E}\left[\mathcal{L}(\boldsymbol{x})\right] < \epsilon$
```

activity-perturbed forward gradient $g^{\mathrm{FG-A}}\left(\boldsymbol{W}^{(l)}\right)$ is then:

$$g^{\mathrm{FG-A}}\left(\boldsymbol{W}^{(l)}\right) := \left(d^{\mathrm{FG-A}} \boldsymbol{u}^{(l)}\right) \frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{W}^{(l)}} \tag{103}$$

where $\frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{W}^{(l)}}$ are local gradients computed by neurons with locally-available information. See Algorithm 4 for the full algorithm applied to a fully connected DNN.

This method reduces the number of derivatives to estimate since the number of neurons is considerably lower than the number of weights (see Table 11 for an example). Consequently, the method leads to lower variance than FG-W (Ren et al. 2023).

FG-A demonstrated improvements over FG-W on several benchmark datasets (Ren

Table 11: Number of neurons and number of parameters in fully-connected DNNs with different depths. In this example, I consider a 784 input network with hidden layers of 800 neurons and 10 outputs.

Depth	N. Neurons	N. Params
2 Layers	810	636K
3 Layers	1610	1.2M
4 Layers	2410	1.9M

et al. 2023) but still suffers from high variance as the number of neurons in DNNs remains large. To avoid this issue, Ren et al. proposed the Local Greedy Activity-Perturbed Forward Gradient (LG-FG-A) method, which uses local loss functions to partition the gradient computation and decrease the number of derivatives to estimate. By adopting this local greedy strategy, LG-FG-A greatly improved the performance of Local MLP Mixers, a specific architecture that uses shallow multi-layer perceptrons to perform vision without having to use convolution. However, no results were reported for conventional fully-connected or convolutional neural networks, where the number of neurons per layer is larger than in MLP Mixers.

4.5 Direct Feedback Alignment

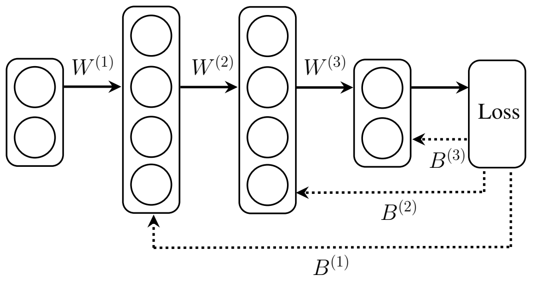


Figure 35: Illustrations of the Direct Feedback Alignment algorithm applied to a DNN. Solid arrows represent forward inference paths and dotted arrows represent error back-propagation paths.

While BP relies on symmetric weights to propagate errors to hidden layers, it has been shown that weight symmetry is not mandatory to achieve learning (Lillicrap et al. 2016). For example, FA has proven that random fixed weights can also be used for backpropagating errors and still achieve learning (Lillicrap et al. 2016). Therefore, by not reusing the network weights, FA effectively overcomes the weight transport problem. However, the algorithm still backpropagates errors sequentially which limits its parallelization.

DFA takes the idea of FA one step further by directly projecting output errors to hidden layers using fixed linear feedback connections (Nøkland 2016) — see Figure 35. Feedback matrices $\boldsymbol{B}^{(l)} \in \mathbb{R}^{n_L \times n_l}$ replace the derivatives $\frac{\partial \boldsymbol{y}^{(L)}}{\partial \boldsymbol{y}^{(l)}}$ of output neurons with respect to hidden neurons. The approximate gradient $g^{\mathrm{DFA}}\left(\boldsymbol{W}^{(l)}\right)$ for the weights of the hidden layer l is then computed as follows:

$$g^{\mathrm{DFA}}\left(\boldsymbol{W}^{(l)}\right) := \frac{\partial \mathcal{L}\left(\boldsymbol{x}\right)}{\partial \boldsymbol{y}^{(L)}} \boldsymbol{B}^{(l)} \frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{W}^{(l)}} \tag{104}$$

In DFA, feedback matrices for hidden layers are chosen to be random and kept fixed during training. For the output layer, the identity matrix is used as no propagation of errors is required.

The success of DFA depends on the alignment between the forward and feedback weights, which results in the alignment between the approximate and true gradient (Lillicrap et al. 2016; Nøkland 2016; Refinetti et al. 2021). When the angle between these gradients is within 90 degrees, the direction of the update is descending (Lillicrap et al. 2016; Nøkland 2016).

DFA can scale to modern deep learning architectures such as Transformers (Launay et al. 2020) but is unable to train deep convolution layers (Launay, Poli and Krzakala 2019) and fails to learn challenging datasets such as CIFAR-100 or ImageNet without the use of transfer learning (Bartunov et al. 2018; Crafton et al. 2019). However, recent methods to learn symmetric feedback such as the Direct Kolen-Pollack (DKP) algorithm (Webster, Choi and Ahn 2021) showed promising results with convolutional neural networks due to improved gradient alignments.

Algorithm 5: Forward Direct Feedback Alignment algorithm with a fully-connected DNN.

```
1: Input: Training data $\mathcal{D}$
2: Randomly initialize $w_{ij}^{(l)}$ for all $l$, $i$ and $j$.
3: Initialize $\boldsymbol{B}^{(l)} = \boldsymbol{0}$ for all $l$.
4: repeat
5:     {Inference (sequential)}
6:     for all $\boldsymbol{x}$ in $\mathcal{D}$ do
7:         $\boldsymbol{y}^{(0)} \leftarrow \boldsymbol{x}_s$
8:         $\boldsymbol{d}^{(0)} \leftarrow \boldsymbol{0}$
9:         for $l = 1$ to $L$ do
10:            Sample $\boldsymbol{v}^{(l)} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
11:            $\boldsymbol{a}^{(l)} \leftarrow \boldsymbol{W}^{(l)} \boldsymbol{y}^{(l-1)}$
12:            $\boldsymbol{y}^{(l)} \leftarrow \sigma\left(\boldsymbol{a}^{(l)}\right)$
13:            $\boldsymbol{d}^{(l)} \leftarrow \left(\boldsymbol{W}^{(l)} \boldsymbol{d}^{(l-1)}\right) \odot \sigma'\left(\boldsymbol{a}^{(l)}\right)$
14:            if $l < L$ then
15:                $\boldsymbol{d}^{(l)} \leftarrow \boldsymbol{d}^{(l)} + \boldsymbol{v}^{(l)}$
16:            end if
17:        end for
18:        $\boldsymbol{e} \leftarrow \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(L)}}$
19:        {Weights updates (parallel)}
20:        $\boldsymbol{W}^{(L)} \leftarrow \boldsymbol{W}^{(L)} - \lambda\, \boldsymbol{e} \otimes \boldsymbol{y}^{(L-1)}$
21:        for $l = 1$ to $L - 1$ do
22:            $\boldsymbol{B}^{(l)} \leftarrow (1 - \alpha)\, \boldsymbol{B}^{(l)} - \alpha \left(\boldsymbol{d}^{(L)} \otimes \boldsymbol{v}^{(l)}\right)$
23:            $\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \lambda \left(\boldsymbol{e} \boldsymbol{B}^{(l)} \odot \sigma'\left(\boldsymbol{a}^{(l)}\right)\right) \otimes \boldsymbol{y}^{(l-1)}$
24:        end for
25:    end for
26: until $\mathbb{E}\left[\mathcal{L}(\boldsymbol{x})\right] < \epsilon$
```

4.6 Forward Direct Feedback Alignment

In this section, I describe the proposed Forward Direct Feedback Alignment (FDFA) algorithm which uses forward gradients to estimate derivatives between output and hidden neurons as direct feedback connections.

Similarly to FG-A, we sample perturbation vectors $\boldsymbol{u}^{(l)} \in \mathbb{R}^{n_l}$ for each layer $l \leq L$ from a multivariate standard normal distribution. However, in contrast to FG-A, we use the directional derivatives that are computed at the output layer rather than at the loss function, such as:

$$\boldsymbol{d}^{\mathrm{FDFA}} = \sum_{l=1}^{L-1} \sum_{i=1}^{n_l} \frac{\partial \boldsymbol{y}^{(L)}}{\partial y_i^{(l)}} u_i^{(l)} \tag{105}$$

which produces a vector of directional derivatives. Note that only partial derivatives of the network outputs with respect to the activations of hidden neurons are considered, as represented by feedback connections in DFA. Output neurons are updated using a fixed identity feedback matrix as no error backpropagation is required.

Rather than relying solely on the most recent forward gradient, as in FG-W and FG-A, it is possible to obtain a more accurate estimate of $\frac{\partial \boldsymbol{y}^{(L)}}{\partial \boldsymbol{y}^{(l)}}$ by averaging the forward gradient over the past training steps.

Formally, we define an update rule for the direct feedback connections of DFA that performs an exponential average of the estimates, such as:

$$\boldsymbol{B}^{(l)} \leftarrow (1-\alpha)\, \boldsymbol{B}^{(l)} + \alpha\, \boldsymbol{d}^{\mathrm{FDFA}} \otimes \boldsymbol{u}^{(l)} \tag{106}$$

where $0 < \alpha < 1$ is the feedback learning rate. Equation 106 can also be re-written in a form compatible with the stochastic gradient descent algorithm:

$$\boldsymbol{B}^{(l)} \leftarrow \boldsymbol{B}^{(l)} - \alpha \nabla \boldsymbol{B}^{(l)} \tag{107}$$

where

$$\nabla \boldsymbol{B}^{(l)} = \boldsymbol{B}^{(l)} - \boldsymbol{d}^{\mathrm{FDFA}} \otimes \boldsymbol{u}^{(l)} \tag{108}$$

In essence, Equation 107 minimizes the following Mean Squared Error (MSE) function:

$$\mathcal{L}_B(\boldsymbol{x}) = \mathbb{E}\left[\sum_{l=1}^{L-1} \sum_{o=1}^{n_L} \sum_{i=1}^{n_l} \left(B_{o,j}^{(l)} - d_o^{\mathrm{FDFA}} u_i^{(l)}\right)^2\right] \tag{109}$$

where $d_o^{\mathrm{FDFA}} u_i^{(l)}$ is the target value for the optimized feedback connection $B_{o,j}^{(l)}$. The global minimum of this loss function occurs at the point where all feedback connections are equal to the expected derivative between output and hidden neurons, such as:

$$B_{o,i}^{(l)} = \mathbb{E}\left[d_o^{\mathrm{FDFA}} u_i^{(l)}\right] = \mathbb{E}\left[\frac{\partial y_o^{(L)}}{\partial y_i^{(l)}}\right] \tag{110}$$

for all output o and all hidden neuron i of all hidden layers $l < L$. Therefore, the FDFA algorithm is a dual optimization procedure where both loss functions $\mathcal{L}(\boldsymbol{x})$ and $\mathcal{L}_B(\boldsymbol{x})$ are minimized concurrently: weights are updated with direct feedbacks to minimize prediction errors and feedbacks connections converge towards the derivatives between output and hidden neurons. In the case where output neurons are linear, the penultimate feedback matrix becomes symmetric with the output weight matrix:

$$\mathbb{E}\left[\boldsymbol{B}^{(L-1)}\right] = \mathbb{E}\left[\frac{\partial \boldsymbol{y}^{(L)}}{\partial \boldsymbol{y}^{(L-1)}}\right] = \boldsymbol{W}^{(L)} \tag{111}$$

In this particular case, gradient estimates become mathematically equivalent to BP. For lower hidden layers, feedback matrices linearly approximate the non-linear derivatives between output and hidden neurons, which introduces a bias in the gradient estimates. This is because the feedback learning rule introduced in FDFA acts as momentum for these derivatives (see Equation 106), which is known to mitigate the effect of gradient variance on convergence at the cost of increased bias (Defazio 2020).

Finally, the FDFA gradient estimate $g^{\mathrm{FDFA}}\left(\boldsymbol{W}^{(l)}\right)$ for the weights $\boldsymbol{W}^{(l)}$ of the hidden layer l is computed as in DFA. Formally, output errors are linearly projected onto hidden neurons using the feedback matrix $B^{(l)}$, such as:

$$g^{\mathrm{FDFA}}\left(\boldsymbol{W}^{(l)}\right) := \frac{\partial \mathcal{L}\left(\boldsymbol{x}\right)}{\partial \boldsymbol{y}^{(L)}} \boldsymbol{B}^{(l)} \frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{W}^{(l)}} \tag{112}$$

The FDFA gradient estimate is then used to update the weights of the network with stochastic gradient descent, such as:

$$\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \lambda\, g^{\mathrm{FDFA}}\left(\boldsymbol{W}^{(l)}\right) \tag{113}$$

Table 12: Theoretical gradient estimation variance produced by the FG-W, FG-A and FDFA algorithm given a single input sample.

Algorithm	Local	Estimation Variance
BP	✗	0
FG-W	✓	$O(n_1 n_0)$
FG-A	✓	$O(n_1)$
DFA	✓	0
FDFA ($\alpha \to 0$)	✓	0

where $\lambda > 0$ is a learning rate. Note that other gradient-based optimization techniques can also be applied for both feedback and forward weight updates in place of SGD, such as Adam (Kingma and Ba 2017) — see Section 2.4 for details about other gradient-based optimization algorithms.

The full algorithm applied to a fully connected DNN is given in Algorithm 5.

4.7 Theoretical Results

In this section, I investigate and compare the theoretical variance of FG-W, FG-A, and the proposed FDFA algorithm to understand their potential differences in terms of convergence. All my theoretical results have been summarized in Table 12.

I first start by decomposing the variance of unbiased gradient estimates.

4.7.1 Variance Decomposition of Unbiased Gradient Estimates

Following (Wen et al. 2018) and (Ren et al. 2023), it can be shown that the variance of unbiased gradient estimates can be decomposed into two terms.

Proposition 3. *(Wen et al. 2018; Ren et al. 2023) Let $\boldsymbol{W}^{(1)} \in \mathbb{R}^{n_1,n_0}$ be the hidden weights of a two-layer fully-connected neural network. We denote by $\boldsymbol{x}$ the independent input samples and by $\boldsymbol{v}^{(1)}$ the independent random perturbations used to estimate the gradients of the hidden layer. The variance of a gradient estimate $g\left(w_{i,j}^{(1)}\right)$ for the* *hidden weight $w_{i,j}^{(1)}$ can be decomposed into two parts:*

$$\mathrm{Var}\left[g\left(w_{i,j}^{(1)}\right)\right] = z_1 + z_2 \tag{114}$$

where

$$z_1 := \underset{\boldsymbol{x}}{\mathrm{Var}}\left[\underset{\boldsymbol{v}^{(1)}}{\mathbb{E}}\left[g\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right]\right] \tag{115}$$

and

$$z_2 := \underset{\boldsymbol{x}}{\mathbb{E}}\left[\underset{\boldsymbol{v}^{(1)}}{\mathrm{Var}}\left[g\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right]\right] \tag{116}$$

The first term z_1 defined in Proposition 3 corresponds to the gradient variance from data sampling. This term vanishes with the size of the batch in the case of mini-batch learning (Wen et al. 2018; Ren et al. 2023). The second term z_2 corresponds to the expected additional variance produced by the stochastic estimation of the gradient, which scales differently for each algorithm. In the case of BP and DFA, this term equals zero as both algorithms are deterministic. In (Ren et al. 2023), a third term z_3 was considered, which corresponds to the correlation between gradient estimates. However, this term is zero in the case where perturbations are independent (Ren et al. 2023).

The only terms that differ between the variance of FG-W, FG-A, and FDFA is z_2. Therefore, we are interested in comparing the gradient estimation variance z_2 of each algorithm with the proposed FDFA algorithm to gain insight about their convergence rates.

4.7.2 Estimation Variance of Weight-Perturbed Forward Gradients

To find the theoretical estimation variance of FG-W, we first start by deriving the variance of forward derivatives.

Lemma 4. *Let* $\boldsymbol{x} \in \mathbb{R}^n$ *be a vector of size* n *and* $\boldsymbol{v} \in \mathbb{R}^n$ *be a random vector of* n *independent variables. If* $\boldsymbol{v} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ *follows a multivariate standard normal distribution, then:*

$$\mathrm{Var}\left[(\boldsymbol{x} \cdot \boldsymbol{v})\, v_i\right] = x_i^2 + \|\boldsymbol{x}\|_2^2 \tag{117}$$

Proof. Because the elements of $\boldsymbol{x}$ are considered as constants and all the elements of $\boldsymbol{v}$ are independant from each other, the variance of $(\boldsymbol{x} \cdot \boldsymbol{v})\, v_i$ decomposes as follows:

$$\begin{aligned} \mathrm{Var}\left[(\boldsymbol{x} \cdot \boldsymbol{v})\, v_i\right] =& \mathrm{Var}\left[\sum_{j=1}^{n} x_j v_j v_i\right] \\ =& \sum_{j=1}^{n} x_j^2 \, \mathrm{Var}\left[v_j v_i\right] \end{aligned} \tag{118}$$

We can show that if $j \neq i$:

$$\begin{aligned} \mathrm{Var}\left[v_j v_i\right] =& \mathbb{E}\left[(v_j v_i)^2\right] - \mathbb{E}\left[v_j v_i\right]^2 \\ =& \mathbb{E}\left[v_i^2\right] \mathbb{E}\left[v_j^2\right] - \mathbb{E}\left[v_i\right]^2 \mathbb{E}\left[v_j\right]^2 \\ =& 1 \end{aligned} \tag{119}$$

and if $j = i$:

$$\begin{aligned} \mathrm{Var}\left[v_i^2\right] =& \mathbb{E}\left[\left(v_i^2\right)^2\right] - \mathbb{E}\left[v_i^2\right]^2 \\ =& \mathbb{E}\left[v_i^4\right] - \mathbb{E}\left[v_i^2\right]^2 \\ =& 2 \end{aligned} \tag{120}$$

as $\mathbb{E}\left[v_i^2\right]^2 = \mathrm{Var}\left[v_i\right]^2 = 1$ and $\mathbb{E}\left[v_i^4\right] = 3\mathrm{Var}\left[v_i\right] = 3$.

Therefore, by plugging Equations 119 and 120 into Equation 118, we find:

$$\begin{aligned}
\mathrm{Var}\left[(\boldsymbol{x}\cdot\boldsymbol{v})\, v_i\right] =& \sum_{j=1}^{n} x_j^2 \ \mathrm{Var}\left[v_j v_i\right] \\
=& x_i^2 \mathrm{Var}\left[v_i^2\right] + \sum_{\substack{j=1 \\ j\neq i}}^{n} x_j^2 \mathrm{Var}\left[v_j v_i\right] \\
=& 2x_i^2 + \sum_{\substack{j=1 \\ j\neq i}}^{n} x_j^2 \\
=& x_i^2 + \sum_{j=1}^{n} x_j^2 \\
=& x_i^2 + \|\boldsymbol{x}\|_2^2
\end{aligned} \tag{121}$$

which concludes the proof. □

By applying Lemma 4, we can derive the theoretical estimation variance of FG-W.

Proposition 5. *Estimation Variance of Weight-Perturbed Forward Gradients*
Let $\boldsymbol{W}^{(1)} \in \mathbb{R}^{n_1,n_0}$ be the hidden weights of a two-layer fully connected neural network evaluated with an input sample $\boldsymbol{x} \in \mathbb{R}^{n_0}$. We denote by $g^{\mathrm{FG-W}}\left(w_{i,j}^{(1)}\right)$ the weight-perturbed forward gradient for the weight $w_{i,j}^{(l)}$. We also assume that all derivatives $\left(\frac{\partial\mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(l)}}\right)^2 \leq \beta$ are bounded and that all the elements of the perturbation matrix $\boldsymbol{V}^{(2)}$ for the weights of the output layer are 0. If each element $v_{i,j}^{(1)} \sim \mathcal{N}(0,1)$ of $\boldsymbol{V}^{(1)}$ follows a standard normal distribution, then:

$$\begin{aligned}
\underset{\boldsymbol{v}^{(1)}}{\mathrm{Var}}\left[g^{\mathrm{FG-W}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] =& \left(\frac{\partial\mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(1)}}\right)^2 + \left\|\frac{\partial\mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{W}^{(1)}}\right\|_2^2 \\
=& O\left(n_1 n_0\right)
\end{aligned} \tag{122}$$

in the limit of large n_1 and large n_0.

Proof. By application of Lemma 4, we know that:

$$\begin{aligned}\underset{\boldsymbol{v}^{(1)}}{\operatorname{Var}}\left[g^{\mathrm{FG-W}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] =& \underset{\boldsymbol{v}^{(1)}}{\operatorname{Var}}\left[\sum_{k=1}^{n_1}\sum_{l=1}^{n_0}\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{k,l}^{(1)}} v_{k,l}\, v_{i,j}\right] \\ &= \left(\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(1)}}\right)^2 + \left\|\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{W}^{(1)}}\right\|_2^2\end{aligned} \tag{123}$$

However, in the limit of large n_0 and large n_1:

$$\left\|\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{W}^{(1)}}\right\|_2^2 = \sum_{i=1}^{n_1}\sum_{j=1}^{n_0}\left(\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(1)}}\right)^2 = O\left(n_1 n_0\right) \tag{124}$$

Therefore, we conclude that:

$$\underset{\boldsymbol{v}^{(1)}}{\operatorname{Var}}\left[g^{\mathrm{FG-W}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] = O\left(n_1 n_0\right) \tag{125}$$

□

Proposition 5 shows that the variance estimate of FG-W scales linearly with the number of parameters in the network. This highlights the convergence issues of FG-W in large DNNs.

4.7.3 Estimation Variance of Activity-Perturbed Forward Gradients

I now derive the estimation variance of the FG-A algorithm.

Proposition 6. *Estimation Variance of Activity-Perturbed Forward Gradients*
Let $\boldsymbol{W}^{(1)} \in \mathbb{R}^{n_1,n_0}$ be the hidden weights of a two-layer fully connected neural network evaluated with an input sample $\boldsymbol{x} \in \mathbb{R}^{n_0}$. We denote by $g^{\mathrm{FG-A}}\left(w_{i,j}^{(1)}\right)$ the activity-perturbed forward gradient for the weight $w_{i,j}^{(l)}$. We also assume that all derivatives $\left(\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(l)}}\right)^2 \leq \beta$ are bounded and that all the elements of the perturbation vector $\boldsymbol{u}^{(2)}$ for

the activations of the output layer are 0. If each element $u_i^{(1)} \sim \mathcal{N}(0,1)$ of $\boldsymbol{u}^{(1)}$ follows a standard normal distribution, then:

$$\begin{aligned}\operatorname*{Var}_{\boldsymbol{v}^{(1)}}\left[g^{\mathrm{FG-A}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] &= \left[\left(\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial y_i^{(1)}}\right)^2 + \left\|\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(1)}}\right\|_2^2\right]\left(\frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}\right)^2 \\ &= O\left(n_1\right)\end{aligned} \tag{126}$$

in the limit of large n_1 and large n_0.

Proof. Starting from the variance of $g^{\mathrm{FG-A}}\left(\boldsymbol{W}^{(1)}\right)$, we have:

$$\begin{aligned}\operatorname*{Var}_{\boldsymbol{v}^{(1)}}\left[g^{\mathrm{FG-A}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] =& \operatorname*{Var}_{\boldsymbol{v}^{(1)}}\left[\left(\sum_{k=1}^{n_1}\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial y_k^{(1)}}u_k^{(1)}\right)u_i^{(1)}\ \frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}\right] \\ =& \operatorname*{Var}_{\boldsymbol{v}^{(1)}}\left[\sum_{k=1}^{n_1}\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial y_k^{(1)}}u_k^{(1)}u_i^{(1)} \mid \boldsymbol{x}\right]\left(\frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}\right)^2\end{aligned} \tag{127}$$

as the partial derivative $\frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}$ is considered as constant.
By applying Lemma 4, we obtain:

$$\begin{aligned}\operatorname*{Var}_{\boldsymbol{v}^{(1)}}\left[g^{\mathrm{FG-A}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] &= \left[\left(\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial y_i^{(1)}}\right)^2 + \left\|\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(1)}}\right\|_2^2\right]\left(\frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}\right)^2 \\ &= \left(\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(1)}}\right)^2 + \left\|\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(1)}}\right\|_2^2\left(\frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}\right)^2\end{aligned} \tag{128}$$

Finally, we can show that

$$\left\|\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{y}^{(1)}}\right\|_2^2 = \sum_{i=1}^{n_1}\left(\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial y_i^{(1)}}\right)^2 = O\left(n_1\right) \tag{129}$$

Therefore:

$$\operatorname*{Var}_{\boldsymbol{v}^{(1)}}\left[g^{\mathrm{FG-A}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] = O\left(n_1\right) \tag{130}$$

□

Proposition 6 shows that the estimation variance of the FG-A algorithm only scales linearly with the number of neurons in the network and is invariant to the number of weights per neuron. This represents a significant improvement compared to the FG-W algorithm which scales with the number of parameters in the network.

4.7.4 Estimation Variance of Forward Direct Feedback Alignment Gradients

I now derive the estimation variance of the proposed FDFA algorithm.

Proposition 7. *Estimation Variance of Forward Direct Feedback Alignment Gradients Let $\boldsymbol{W}^{(1)} \in \mathbb{R}^{n_1, n_0}$ be the hidden weights of a two-layer fully connected neural network evaluated with an input sample $\boldsymbol{x} \in \mathbb{R}^{n_0}$. We denote by $g^{\text{FDFA}}\left(w_{i,j}^{(1)}\right)$ the FDFA gradient estimate for the weight $w_{i,j}^{(l)}$ and assume that all the elements of the perturbation vector $\boldsymbol{u}^{(2)}$ for the activations of the output layer are 0. We also assume that all derivatives $\left(\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(l)}}\right)^2 \leq \beta$ are bounded and that the feedback matrix $\boldsymbol{B}^{(1)}$ converged to $\frac{\partial \boldsymbol{y}^{(2)}}{\partial \boldsymbol{y}^{(1)}} = \boldsymbol{W}^{(1)}$, making the gradient estimates unbiased. If each element $u_i^{(1)} \sim \mathcal{N}(0,1)$ of $\boldsymbol{u}^{(1)}$ follows a standard normal distribution, then:*

$$\operatorname*{Var}_{\boldsymbol{v}^{(1)}}\left[g^{\text{FDFA}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] = \alpha^2 \operatorname*{Var}_{\boldsymbol{v}^{(1)}}\left[\sum_{k=1}^{n_1} \frac{\partial y_o^{(1)}}{\partial y_k^{(1)}} u_k^{(1)} u_i^{(1)} \mid \boldsymbol{x}\right] \left(\frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}\right)^2 \tag{131}$$

and

$$\lim_{\alpha \to 0} \operatorname*{Var}_{\boldsymbol{v}^{(1)}}\left[g^{\text{FDFA}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] = 0 \tag{132}$$

Proof. Starting from the gradient estimate $g^{\mathrm{FG-A}}\left(w_{i,j}^{(1)}\right)$, we have:

$$\begin{aligned} g^{\mathrm{FDFA}}\left(w_{i,j}^{(1)}\right) &= \sum_{o=1}^{n_2} \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial y_o^{(2)}} b_{o,i}^{(1)} \frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}} \\ &= \sum_{o=1}^{n_2} \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial y_o^{(2)}} \left(\alpha \sum_{k=1}^{n_1} \frac{\partial y_o^{(1)}}{\partial y_k^{(1)}} u_k^{(1)} u_i^{(1)} + (1-\alpha) \frac{\partial y_o^{(2)}}{\partial y_i^{(1)}} \right) \frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}} \\ &= (1-\alpha) \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(1)}} + \alpha \sum_{k=1}^{n_1} \frac{\partial y_o^{(1)}}{\partial y_k^{(1)}} u_k^{(1)} u_i^{(1)} \frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}} \end{aligned} \tag{133}$$

Because both $\frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(1)}}$ and $\frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}$ are constants, the variance of $g^{\mathrm{FDFA}}\left(w_{i,j}^{(1)}\right)$ given an input sample $\boldsymbol{x}$ reduces to:

$$\begin{aligned} \underset{\boldsymbol{v}^{(1)}}{\mathrm{Var}}\left[g^{\mathrm{FDFA}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] =& \underset{\boldsymbol{v}^{(1)}}{\mathrm{Var}}\left[(1-\alpha) \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(1)}} + \alpha \sum_{k=1}^{n_1} \frac{\partial y_o^{(1)}}{\partial y_k^{(1)}} u_k^{(1)} u_i^{(1)} \frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}} \mid \boldsymbol{x}\right] \\ =& \underset{\boldsymbol{v}^{(1)}}{\mathrm{Var}}\left[(1-\alpha) \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial w_{i,j}^{(1)}} \mid \boldsymbol{x}\right] \\ &+ \underset{\boldsymbol{v}^{(1)}}{\mathrm{Var}}\left[\alpha \sum_{k=1}^{n_1} \frac{\partial y_o^{(1)}}{\partial y_k^{(1)}} u_k^{(1)} u_i^{(1)} \frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}} \mid \boldsymbol{x}\right] \\ =& \alpha^2 \underset{\boldsymbol{v}^{(1)}}{\mathrm{Var}}\left[\sum_{k=1}^{n_1} \frac{\partial y_o^{(1)}}{\partial y_k^{(1)}} u_k^{(1)} u_i^{(1)} \mid \boldsymbol{x}\right] \left(\frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}\right)^2 \end{aligned} \tag{134}$$

Therefore:

$$\begin{aligned} \lim_{\alpha \to 0} \underset{\boldsymbol{v}^{(1)}}{\mathrm{Var}}\left[g^{\mathrm{FDFA}}\left(w_{i,j}^{(1)}\right) \mid \boldsymbol{x}\right] &= \lim_{\alpha \to 0} \alpha^2 \underset{\boldsymbol{v}^{(1)}}{\mathrm{Var}}\left[\sum_{k=1}^{n_1} \frac{\partial y_o^{(1)}}{\partial y_k^{(1)}} u_k^{(1)} u_i^{(1)} \mid \boldsymbol{x}\right] \left(\frac{\partial y_i^{(1)}}{\partial w_{i,j}^{(1)}}\right)^2 \\ &= 0 \end{aligned} \tag{135}$$

which concludes the proof. □

Proposition 7 demonstrates that the estimation variance of the FDFA algorithm scales quadratically with the feedback learning rate α. This means that, in the limit

Table 13: Descriptions of the benchmark datasets used in my empirical experiments.

Dataset Name	Train Samples	Test Samples	Classes	Description
MNIST	60 000	10 000	10	28x28 images of handwritten digits
Fashion MNIST	60 000	10 000	10	28x28 images of clothes
CIFAR10	60 000	10 000	10	32x32 images of vehicles and animals
CIFAR100	60 000	10 000	100	32x32 images of various vehicles, animals, plants, objects...
Tiny ImageNet 200	100 000	10 000	200	64x64 images of various vehicles, animals, plants, objects...

of $\alpha \to 0$, the estimation variance of FDFA quadratically decreases towards zero. This gives the FDFA algorithm the potential to benefit from significantly lower variance than FG-W and FG-A and, consequently, greater convergence rates.

4.8 Experimental Settings

I now describe the experimental settings used to produce the empirical results, including the benchmark datasets, the network architectures, the loss function, the different hyperparameter values as well as the software and hardware setup.

4.8.1 Benchmark Datasets

To evaluate the performance of the proposed FDFA algorithm and compare it with other alternatives to BP, I selected several datasets widely adopted as standard benchmarks and expressing increasing levels of difficulty, namely MNIST (LeCun, Cortes and Burges 2010), Fashion MNIST (Xiao, Rasul and Vollgraf 2017), CIFAR10 and CIFAR100 (Krizhevsky, Hinton et al. 2009), and Tiny ImageNet 200 (Le and Yang 2015). See Table 13 for a detailed description of each dataset. We used the original train/test split of each dataset.

4.8.2 Network Architecture

I benchmarked each training method with several architectures including fully-connected DNNs and CNNs. More precisely, I trained fully-connected DNNs with 1, 2, and 3 hidden layers. In addition, I trained shallow CNNs with the following architecture: 15C5-P2-40C5-P2-128-10 where 15C5 represents 15 5x5 convolutional layers and P2 represents 2x2 max pooling layers. Finally, I trained AlexNet (Krizhevsky, Sutskever and Hinton 2012a) networks, composed of the 8 following layers:
64C11-P3-192C5-P3-384C3-256C3-256C3-P3-4096-4094-100

I removed the dropout layer in AlexNet as I found that it negatively affects feedback learning. Finally, I also added batch normalization (Ioffe and Szegedy 2015) after each convolutional layer of AlexNet to help training.

Neurons in each benchmarked DNN use the ReLU activation function in the hidden layers and linear activations in the output layer. For LG-FG-A, additional local linear outputs with their loss function were added after each layer $l < L - 1$ to perform local greedy learning (Ren et al. 2023)

4.8.3 Loss function

In every DNN, I applied a softmax operation at the outputs, such as:

$$p_i := \frac{\exp\left(y_i^{(L)}\right)}{\sum_{j=1}^{n_L} \exp\left(y_j^{(L)}\right)} \tag{136}$$

where $p_i^{(L)}$ represent the class probability for the output i. I then defined a *Cross-Entropy* loss function using the output class probabilities computed by the softmax operation, such as:

$$\mathcal{L}(\boldsymbol{x}) = -\sum_{i=1}^{n_L} \widehat{y}_i \log(p_i) \tag{137}$$

where $\widehat{y}_i$ one if i is the index of the target label and zero otherwise. This function corresponds to the *Softmax Cross-Entropy* loss widely used for classification in modern deep learning. However, note that any differentiable loss functions can be used instead of the softmax cross entropy, such as the MSE.

4.8.4 Update Method and Hyperparameters

In BP, FG-W, FG-A, LG-FG-A, DKP, and FDFA, forward weights were initialized with the uniform Kaiming initialization (He et al. 2015). For FDFA, feedback connections were initialized to 0. For Random DFA, forward weights were initialized to 0 and feedback weights were drawn from a uniform Kaiming distribution and kept fixed during training.

I train 5 times each network over 100 epochs with Adam (Kingma and Ba 2017) — see Section 2.4.6 for details. I also used Adam for the updates of feedback matrices in the FDFA and DKP algorithms. Adam was used because it is invariant to rescaling the gradient (Kingma and Ba 2017), making it a good optimization method to benchmark convergence with gradient estimators that exhibit different scales.

I used a learning rate of $\lambda = \alpha = 10^{-4}$ and the default values of the parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ in Adam. No regularization or data augmentation has been applied in any experiment. Finally, I used learning rate decay with a decay rate of 0.95 after every epoch.

In this work, all hyperparameters, including α and λ were manually tuned. I found that, in many cases, better convergence behaviors were achieved if $\alpha \leq \lambda$. This motivated my choices for the learning rates of FDFA.

4.8.5 Software and Hardware

All my experiments have been performed with the PyTorch (Paszke et al. 2019) library in Python. BP was implemented by using reverse-mode AD. The FG-W, FG-A, and

LG-FG-A algorithms were implemented using the forward-mode AD recently released in PyTorch. DFA and DKP were implemented by creating specific PyTorch functions with custom backward behaviors. Finally, my FDFA algorithm was implemented by combining the custom function used for implementing DFA with forward-mode AD. All experiments were accelerated using GPUs.

4.9 Empirical Results

I now present my empirical results with the proposed FDFA algorithm and other local alternatives to BP. I compare the performance and convergence rate of the proposed FDFA method with various local alternatives to BP related to my algorithm, including weight-perturbed Weight-Perturbed Forward Gradient (FG-W), Activity-Perturbed Forward Gradient (FG-A), Local-Greedy Activity-Perturbed Forward Gradient (LG-FG-A), Random Direct Feedback Alignment (DFA) and the Direct Kolen-Pollack (DKP) algorithm. In addition, I provide numerical verification of my theoretical results — see Section 4.7 — and analyze the relationship between the variance of gradient estimates and the convergence rate. Finally, I compare the gradient alignment occurring in DFA, DKP, and the proposed FDFA algorithm.

4.9.1 Performance

For each method, average test performance over 10 training runs is reported in Tables 14, 15, and 16.

Overall, the weight-perturbed FG-W algorithm achieves poor generalization compared to BP. More importantly, its performance significantly decreases with the size of the network and the complexity of the task. For example, the method is unable to converge with AlexNet on the CIFAR100 or Tiny ImageNet 200 datasets with 3.43% and 0.70% test accuracy respectively. The FG-A method slightly improves the generalization of forward gradients but still fails to match the performance of BP. The greedy

Table 14: Performance of fully-connected DNNs with different depths. LG-FG-A was not evaluated with two-layers DNNs as these networks are not deep enough to require greedy learning.

Depth	Method	Local	MNIST	Fashion MNIST	CIFAR10
2 Layers	BP	✗	98.25 ± 0.04%	89.37 ± 0.09%	56.20 ± 0.12%
	FG-W	✓	85.87 ± 0.25%	77.72 ± 0.18%	30.47 ± 0.32%
	FG-A		93.39 ± 0.06%	84.73 ± 0.14%	47.39 ± 0.23%
	DFA		98.04 ± 0.07%	88.67 ± 0.12%	54.80 ± 0.22%
	DKP		98.12 ± 0.05%	89.09 ± 0.11%	55.41 ± 0.18%
	FDFA		**98.21 ± 0.02%**	**89.27 ± 0.07%**	**55.74 ± 0.14%**
3 Layers	BP	✗	98.39 ± 0.06%	90.01 ± 0.08%	56.66 ± 0.18%
	FG-W	✓	82.36 ± 0.48%	73.94 ± 0.57%	27.52 ± 0.34%
	FG-A		90.86 ± 0.17%	82.21 ± 0.13%	43.39 ± 0.14%
	LG-FG-A		91.64 ± 0.07%	82.41 ± 0.13%	43.08 ± 0.18%
	DFA		98.18 ± 0.06%	89.19 ± 0.13%	54.20 ± 0.11%
	DKP		98.28 ± 0.06%	89.63 ± 0.12%	55.15 ± 0.24%
	FDFA		**98.30 ± 0.04%**	**89.82 ± 0.13%**	**55.70 ± 0.15%**
4 Layers	BP	✗	98.46 ± 0.05%	89.99 ± 0.15%	56.92 ± 0.19%
	FG-W	✓	80.03 ± 0.62%	71.72 ± 0.63%	25.17 ± 0.32%
	FG-A		83.40 ± 1.33%	77.62 ± 0.25%	35.53 ± 0.21%
	LG-FG-A		86.89 ± 0.90%	77.98 ± 0.21%	34.56 ± 0.34%
	DFA		98.15 ± 0.05%	89.15 ± 0.09%	53.61 ± 0.16%
	DKP		98.27 ± 0.05%	89.48 ± 0.09%	54.35 ± 0.21%
	FDFA		**98.32 ± 0.05%**	**89.56 ± 0.11%**	**54.97 ± 0.19%**

Table 15: Performance of a shallow CNN (15C5-P2-40C5-P2-128-10 where 15C5 represents 15 5x5 convolutional layers and P2 represents a 2x2 max pooling layer) on the MNIST, Fashion MNIST and CIFAR10 datasets.

Method	Local	MNIST	Fashion MNIST	CIFAR10
BP	✗	99.32 ± 0.03%	92.10 ± 0.16%	68.11 ± 0.57%
FG-W	✓	87.63 ± 1.20%	74.93 ± 0.55%	33.10 ± 0.37%
FG-A		96.63 ± 0.31%	82.71 ± 0.89%	42.51 ± 0.32%
LG-FG-A		95.42 ± 0.72%	79.96 ± 0.94%	37.74 ± 2.41%
DFA		98.80 ± 0.08%	89.69 ± 0.22%	58.14 ± 0.94%
DKP		99.08 ± 0.03%	90.39 ± 0.25%	60.30 ± 0.34%
FDFA		**99.16 ± 0.03%**	**91.54 ± 0.11%**	**66.96 ± 0.70%**

Table 16: Performance of AlexNet (Krizhevsky, Sutskever and Hinton 2012a) trained on the CIFAR100 and Tiny ImageNet 200 datasets with the official train-test split.

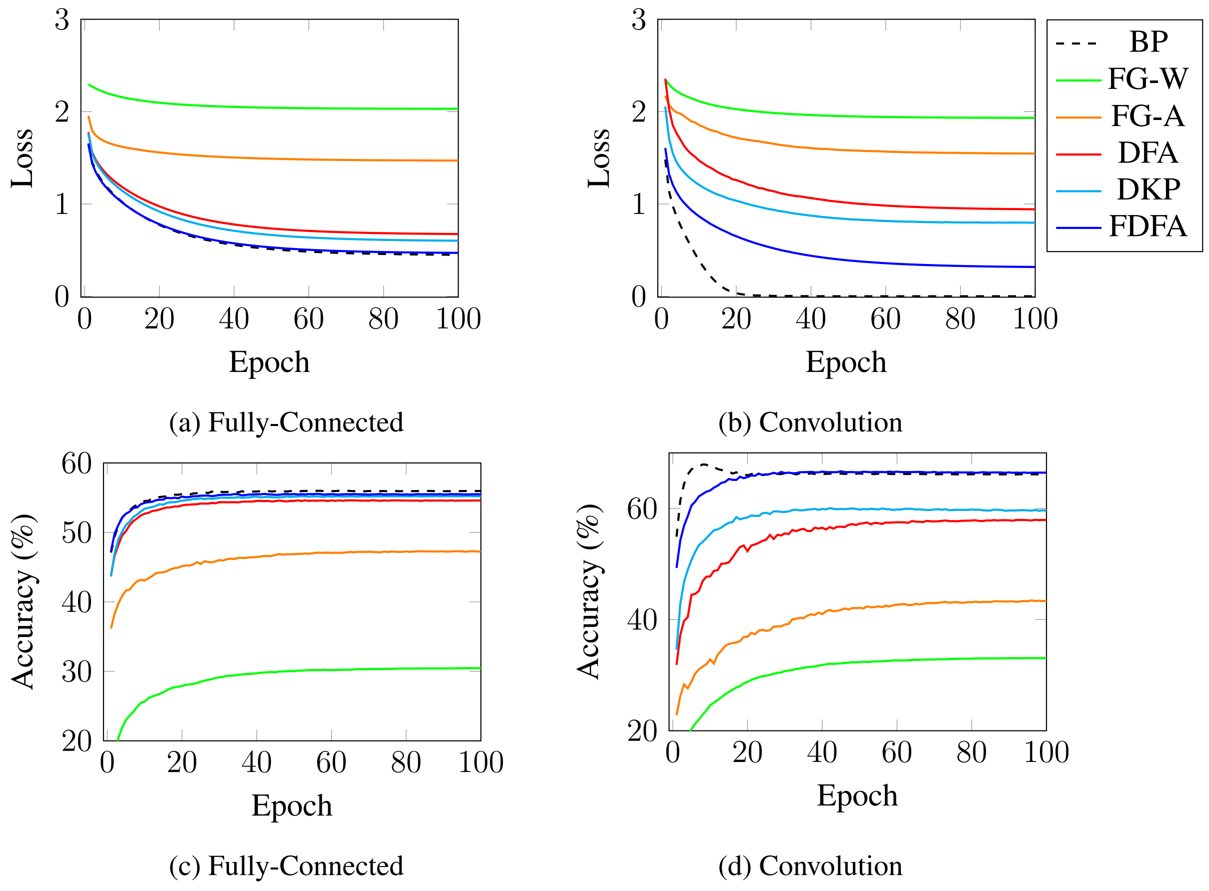
Method	Local	CIFAR100	Tiny ImageNet
BP	✗	60.43 ± 0.35%	40.55 ± 0.31%
FG-W	✓	3.43 ± 0.43%	0.70 ± 0.09%
FG-A		15.84 ± 0.32%	3.52 ± 0.32%
LG-FG-A		15.27 ± 0.29%	2.90 ± 0.14%
DFA		35.75 ± 0.58%	17.47 ± 0.34%
DKP		49.15 ± 0.34%	25.36 ± 0.82%
FDFA		**57.27 ± 0.11%**	**36.47 ± 0.40%**

approach used in LG-FG-A further improves performance when the number of neurons per layer is relatively low, as in fully connected networks for MNIST and Fashion MNIST. However, the method does not perform as well as FG-A on networks that contain large layers such as convolution. This suggests, that LG-FG-A is most suitable for specific architectures where each local loss function sends error signals to a small number of neurons.

DFA achieves performance closer to BP than forward gradient methods. However, it fails to scale to AlexNet on CIFAR100 and Tiny ImageNet 200, as observed in previous work (Launay, Poli and Krzakala 2019; Bartunov et al. 2018; Crafton et al. 2019). The performance of DFA is increased by the DKP algorithm but a gap still exists with BP, especially on difficult tasks such as CIFAR100 and Tiny ImageNet 200. In contrast, the proposed FDFA method performs closer to BP than DFA and DKP on all benchmarked networks and datasets. For example, my method doubles the test accuracy of DFA and improves by at least 10% the performance of DKP on Tiny ImageNet 200.

4.9.2 Convergence

To evaluate the convergence improvements of my method, I measured the evolution of the training loss and test accuracy during the training process. As shown in Figure 36, the FG-W algorithm converges slowly compared to both DFA and BP. Although FG-A slightly improves the convergence of FG-W, it was still unable to converge as quickly as



(a) Fully-Connected

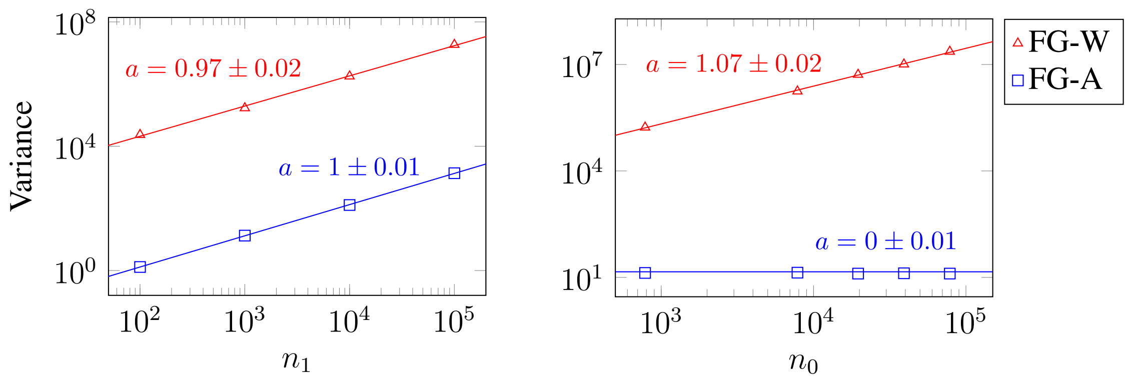
(b) Convolution

(c) Fully-Connected

(d) Convolution

Figure 36: Training loss and test accuracy of a 2-layer fully-connected network (Figures 36a and 36c) and a CNN (Figures 36b and 36d) trained on the CIFAR10 dataset. FDFA has a similar convergence rate as BP on fully connected networks. With CNNs, FDFA is not able to overfit the training data. However, my method has the highest convergence rate compared to FG-W, FG-A, and DFA.

BP and DFA. Both the DKP and FDFA algorithms showed better convergence rates than FG-W, FG-A, and DFA. However, DKP seems to be unable to reduce the loss as low as BP with convolutional layers. Finally, the proposed FDFA algorithm achieved a similar convergence rate as BP with fully connected networks and substantially improved the convergence rate of DFA and DKP with CNNs.



(a) (b)

Figure 37: Variance of the FG-W (red triangles) and FG-A (blue squares) gradient estimates as a function of the number of neurons n_1 (Figure 37a) and number of inputs n_0 (Figure 37b) in a two-layer fully connected network. Each point was produced by computing the variance of gradient estimates over 10 iterations on the MNIST dataset. The pixels of input images were duplicated, to increase the number of inputs n_0. The red and blue lines were fitted using linear regression. The slope a and the asymptotic standard error of each line are given with the same color. These figures show that the variance of FG-W scales linearly with the number of neurons and inputs while the variance of FG-A only scales linearly with the number of neurons.

4.9.3 Variance

Numerical verifications for the scaling of the gradient estimate variance in FG-W and FG-A (Propositions 5 and 6 respectively) are provided in Figure 37. These values were computed by varying the number of neurons as well as the number of inputs in a fully connected DNN and computing the variance of gradient estimates over 10 iterations on the MNIST dataset. A linear regression was then performed on the log-scaled values to determine the power relationship between the number of neurons or the number of inputs and the variance of the gradient estimates.

Figure 37 shows that the variance of FG-W and FG-A scales differently with the number of neurons and parameters in the network. FG-W scales linearly with the number of parameters and FG-A scales linearly with the number of neurons. This indicates that, in large DNNs, gradient estimates provided by the FG-W algorithm have a larger

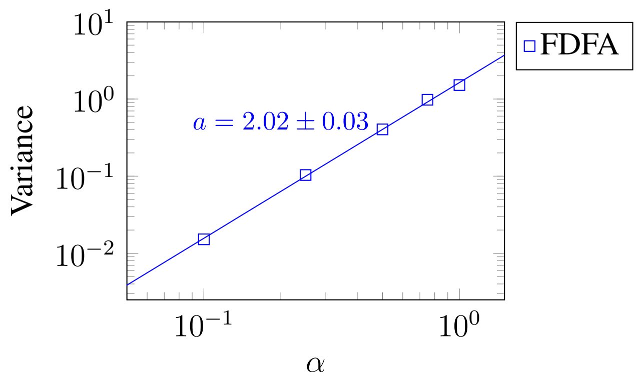


Figure 38: The variance of FDFA gradient estimates as a function of the feedback learning rate α in a two-layer fully connected network. Each point was produced by computing the variance of gradient estimates over 10 iterations of the MNIST dataset. The blue line was fitted using linear regression. The slope a and the asymptotic standard error is given with the same color. This figure shows that the variance of FDFA scales quadratically with α.

variance than FG-A estimates. However, as DNNs also contain large numbers of neurons, the variance of FG-A remains high.

In addition, numerical verifications for the scaling of the gradient estimate variance in FDFA (Propositions 7) are given in Figure 38. Similarly to Figure 37, these values were computed by varying the feedback learning rate α in a fully connected DNN and computing the variance of gradient estimates over 10 iterations on the MNIST dataset. A linear regression was then performed on the log-scaled values to determine the power relationship between α and the variance of the gradient estimates.

Figure 38 empirically shows that the gradient estimation variance of FDFA estimates quadratically vanishes as α tends towards zero. Therefore, by choosing values of α close to zero, the variance of FDFA estimates approaches the variance of the true gradient. This confirms the theoretical result found in Propositon 7.

To better understand the impact that the gradient variance has on the convergence rate of SGD, I conducted an experiment where I measured the training loss of DNNs after one epoch as a function of variance. I simulated the variance of forward gradients by injecting Gaussian noise into the gradient computed with BP. By increasing the

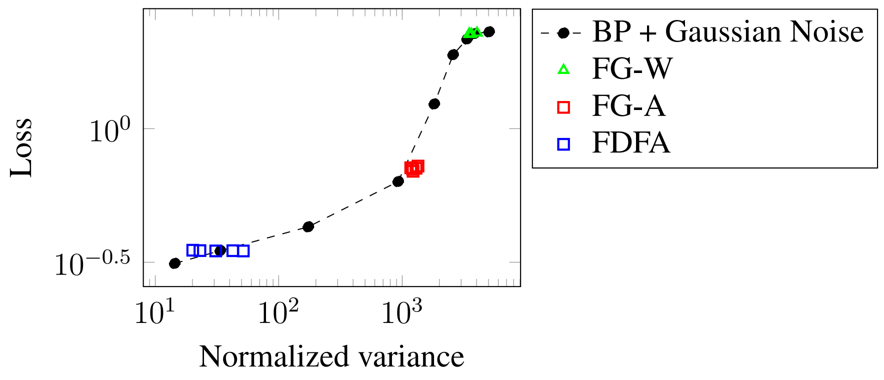


Figure 39: Correlation between the normalized variance of gradient estimates and the loss of a two-layer network with 1000 hidden neurons, following a single training epoch on the MNIST dataset. The variance of BP was artificially increased by adding Gaussian noise to the gradients to simulate the stochasticity of forward gradients. All gradient variances were normalized with the expected squared norm of the gradient estimates to ensure invariance to the amplitude. Pairs of variance-loss for the FG-W, LGA, and FDFA algorithms are represented in green, red, and blue, respectively. This figure shows that the differences in convergence are solely due to the variance of the gradient estimates.

standard deviation of the injected noise, I artificially generated gradients with varying levels of variance. I then trained a two-layer DNN with these noisy gradients and measured the loss after one epoch of training on the MNIST dataset.

Figure 39 illustrates that the variance strongly influences the training loss achieved after one epoch, as low-variance gradients tend to converge towards lower loss values compared to high-variance gradients. Hence, in this context, variance is the determining factor for convergence. Additionally, Figure 39 shows the variance and loss values associated with the FG-W, FG-A, and FDFA algorithms. Notably, all data points align with the line formed by the noisy BP. This shows that the differences in gradient variance are solely responsible for the differences in convergence among the FG-W, FG-A, and FDFA algorithms.

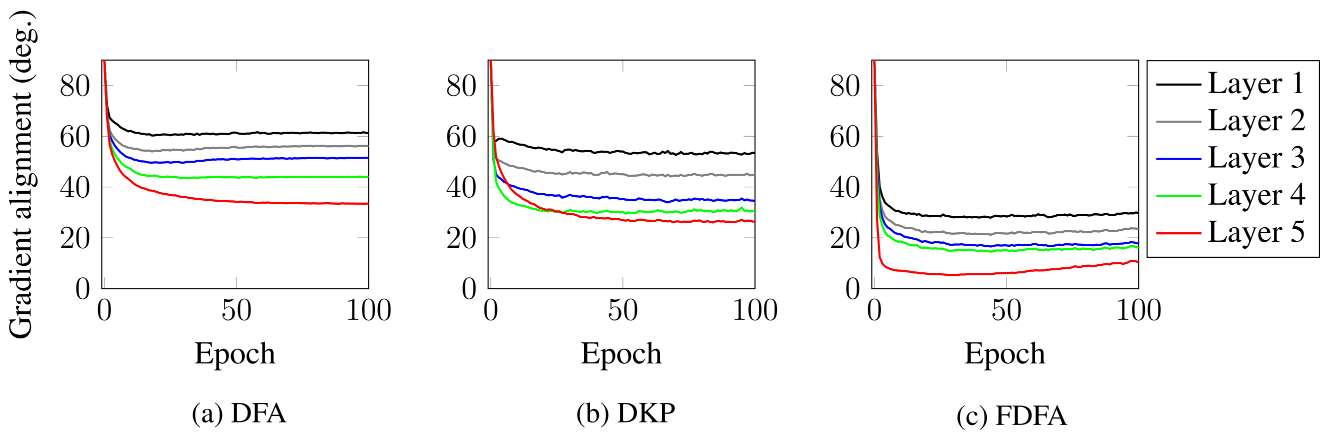


Figure 40: Layerwise alignment between gradient estimates and the true gradient computed using BP. These figures show that the proposed FDFA method (Figure 40c) produces gradient estimates that better align with the true gradients than DFA (Figure 40a) which suggests improved descending directions.

4.9.4 Gradient Alignment in Feedback Methods

In my experiments, I observed that the proposed FDFA algorithm exhibits better convergence compared to DFA, despite having similar variance. This suggests that their gradient estimation variance alone does not fully explain the difference in convergence between these two algorithms.

Both FDFA and DFA exhibit different degrees of biasedness. On the one hand, the FDFA algorithm introduces bias through its feedback learning rule, which acts as momentum for the derivatives between output and hidden neurons. On the other hand, DFA uses random feedback connections, resulting in strongly biased gradient estimates. Therefore, I conjecture that the disparity in convergence between DFA and the proposed FDFA algorithm can be attributed to their respective bias.

To test this, I measured the bias of each method during training by recording the angle between the true gradient and the averaged gradient estimates produced by DFA, FDFA, and the DKP algorithm. This measure, often referred to as *gradient alignment* (Refinetti et al. 2021; Webster, Choi and Ahn 2021), indicates the extent to which the expected gradient estimate deviates from the true gradient.

Figure 40 shows the evolution of the layer-wise gradient alignment of each algorithm in a 5-layer fully connected DNN trained on the MNIST dataset during 100 epochs. This figure shows that the proposed FDFA method achieves faster alignment with the true gradient compared to DFA and DKP, which suggests earlier descending updates. Remarkably, the alignment of the gradient in FDFA is globally enhanced by 30 degrees in comparison to DFA and DKP. This significant improvement suggests that my method provides less biased gradient estimates. Thus, these findings strongly support the fact that the differences in convergence between DFA, DKP, and FDFA can be attributed to their respective levels of biasedness.

4.10 Discussion

The various limitations of BP, and particularly its incompatibility with neuromorphic hardware, highlight the need for alternative training methods. As a result, the exploration of local, parallel, and online alternatives to BP has gained significant importance.

In this chapter, I proposed the FDFA algorithm that combines the FG-A algorithm with DFA and momentum to train DNNs without relying on BP. The algorithm involves a dual optimization process where weights are updated with direct feedback connections to minimize classification errors, and derivatives are estimated as feedback using activity-perturbed forward gradients. My experiments showed that the FDFA gradient estimate closely aligns with the true gradient, particularly in the last layers. However, the sole differences between the FDFA estimate and the true gradient are the feedback connections replacing the derivatives between outputs and hidden neurons. Hence, the convergence of feedback connections toward these derivatives is the only factor responsible for this gradient alignment. We can thus conclude that the proposed feedback learning rule is capable of learning the relevant derivatives to approximate BP. Moreover, the averaging process introduced in my feedback learning rule acts as momentum for the derivatives estimates between output and hidden neurons. My results demonstrated that the introduced momentum significantly reduces the gradient estimation variance compared to other forward gradient methods. Consequently, my method provides more accurate gradient estimates, leading to improved convergence and better performances on several benchmark datasets and architectures.

Using the proposed method, feedback connections in the penultimate layer become equal to the output weights. This results in gradient estimates that are equivalent to BP. However, in deeper layers, feedback connections linearly approximate the non-linear derivatives between output and hidden neurons, introducing a bias. To address this bias, the FDFA algorithm could be adapted to Feedback Alignment, where feedback matrices replace the weights during the backward pass of BP, rather than direct connections between output and hidden layers. Although this modification would lower biasedness and better approximate BP without requiring weight transport, it would come at the cost of increased computation time due to backward locking. This highlights a particular tradeoff between computation cost and biasedness in FDFA. Future work could explore novel ways to leverage this biasedness without hindering the parallel aspect of FDFA.

It is also important to note that, while both DFA and the proposed FDFA algorithm similarly compute gradient estimates, they differ fundamentally in their respective definitions of the feedback connections. In DFA, feedback connections are initialized randomly and remain fixed throughout training, thereby introducing a substantial bias into its pseudo-gradient. In contrast, my FDFA algorithm learns to represent the derivatives between output and hidden neurons as feedback connections. This distinction results in gradient estimates that align more closely with the steepest descent direction compared to DFA. This difference in biasedness was empirically verified in Figure 40 by measuring the angle between the gradient estimate of each algorithm and the true gradient computed with BP. As shown in Table 12, DFA and FDFA have the same variance in the limit of small feedback learning rates (i.e. 0). Hence, their divergence lies solely in their respective biases. I observed in my experiments that this difference in bias consistently influences both the rate of convergence and the performance of DNNs on various benchmark datasets. Therefore, by providing an update direction closer to the direction of steepest descent, the proposed FDFA algorithm demonstrates the ability to converge faster and achieve greater performance than DFA.

Finally, while methods relying on feedback connections showed better performance than forward gradients methods, the use of feedback matrices increases the number of weights to store in memory and ultimately poses a significant memory limitation, especially as networks grow larger. This highlights another important tradeoff between memory usage and performance. Future work could explore mechanisms such as sparse feedback matrices (Crafton et al. 2019) or reduced weight precision (Han and jun Yoo 2019) to mitigate the memory impact of the proposed FDFA algorithm, further enhancing its practicality and scalability.

The FDFA algorithm thus represents a promising local alternative to error backpropagation, effectively resolving backward locking and the weight transport problem. Its ability to approximate backpropagation with low variance not only opens new possibilities for the creation of efficient and scalable training algorithms but also holds significant importance in the domain of neuromorphic computing. By solely propagating information in a forward manner, the FDFA algorithm aligns with the online constraints of neuromorphic systems, presenting new prospects for developing algorithms specifically tailored to meet the requirements of these hardware. Therefore, the implications of my findings highlight the potential of FDFA as a promising direction for online learning in neuromorphic systems.

Chapter 5

Online Gradient Estimates of Spiking Neural Networks

The contributions of this chapter are:

- I propose the *Spiking Forward Direct Feedback Alignment* (SFDFA) algorithm, an adaption of *Forward Direct Feedback Alignment* to SNNs that estimates the weights between output and hidden neurons as feedback connections.
- I describe how exact local gradients of spikes can be computed in an online manner while taking into account the intra-neuron dependencies between post-synaptic spikes and derive a dynamical system for neuromorphic hardware compatibility.
- I show the existence of critical points where local gradients explode numerically. I identify the cause of the problem and propose an ad-hoc solution to avoid gradient explosions.
- I compare the SFDFA algorithm with DFA and show that the proposed algorithm achieves higher performance and convergence rates.

5.1 Introduction

In the context of SNNs, random feedback learning, and particularly the DFA algorithm — see Section 4.5 for details — are popular alternatives to BP for online gradient approximation (Neftci et al. 2017b; Bellec et al. 2020; Rostami et al. 2022; Lee et al. 2020; Shrestha et al. 2019, 2021). For example, the *Event-Driven Random Backpropagation* (eRBP) (Neftci et al. 2017b) and the *e-prop* (Bellec et al. 2020; Rostami et al. 2022) algorithms use random feedback connections to send error signals to hidden neurons at every time steps, thus continuously accumulating gradients during the inference. However, these algorithms constantly project errors to hidden neurons which can lead to significant computation and energy consumption on hardware.

Alternatives such as the *Spike-Train Level Direct Feedback Alignment* (ST-DFA) (Lee et al. 2020) or the *Error-Modulated Spike-Timing-Dependent Plasticity* (EMSTDP) (Shrestha et al. 2019, 2021) algorithms accumulate local changes of weights during the inference and send a single error signal to hidden neurons through random direct feedbacks connections. These approaches are ultimately more energy-efficient as the error projection is only performed once, thus requiring fewer operations. Moreover, because they solve many of the limitations of BP, spiking DFA algorithms have been successfully implemented on various neuromorphic hardware and demonstrated promising performance and lower energy consumption compared to offline training on GPUs (Lee et al. 2020; Shrestha et al. 2021; Rostami et al. 2022).

I demonstrated in Chapter 4 that the strong bias introduced by the random feedback connections affects the convergence rate of DFA. While the algorithm relies on the alignment between weight and feedback connections (Nøkland 2016; Refinetti et al. 2021), my experiments demonstrated that practical alignment tends to be limited due to the lack of flexibility in the feedback connections, thus leading to biased approximate gradients that deviate from the direction of steepest descent — see Section 4.9.4. In contrast, the FDFA algorithm proposed in Chapter 4 allows for greater flexibility in feedback connections by introducing a learning mechanism that estimates the derivatives between output and hidden neurons, leading to gradient estimates that better follow the direction of steepest descend and improved convergence rates. Therefore, similar feedback learning mechanisms could be proposed to improve the convergence of DFA in SNNs.

In this chapter, I propose the *Spiking Forward Direct Feedback Alignment* (SFDFA) algorithm, an adaptation of the FDFA for online training of SNNs. Using graded spikes to propagate additional information during the inference, the proposed SFDFA algorithm estimates the derivatives between output and hidden neurons as feedback connections, thus improving convergence when compared to spiking DFA. The rest of this chapter is structured as follows:

- I first review how the DFA algorithm can be used to train SNNs without relying on error backpropagation.
- Based on my findings in Chapters 3 and 4, I introduce the SFDFA algorithm, a spiking adaptation of the FDFA algorithm to compute gradient estimates of SNNs without error backpropagation. I then describe how the local gradients of spikes can be computed in an online manner and demonstrate the existence of critical points causing gradient explosions. I thus propose an ad hoc solution to these critical points and derive a dynamical system for the computation of local gradients on neuromorphic hardware.
- I describe the experimental settings used to produce my empirical results.
- Finally, I investigate the cause of the critical points in the exact local gradient of spikes and demonstrate the effectiveness of the modified local gradient in avoiding gradient explosions. I then compare the performance and convergence rate of the SFDFA algorithm with DFA and BP on several benchmark datasets as well as the weight and gradient alignment of DFA and SFDFA.

5.2 Spiking Direct Feedback Alignment

The computation of spike errors in SNNs relies on information that is not available in real-time, thus representing an obstacle for online computation. However, an approximation of BP can be computed in an online manner by using a spiking implementation of DFA.

Similarly to DNNs — see Section 4.5 for details —, we define a random matrix $\boldsymbol{B}$ representing the feedback connections between output and hidden neurons. Assuming that a single error signal is sent at the end of the inference, the error δ_i^{DFA} received by the hidden neuron i is computed as (Neftci et al. 2017b; Bellec et al. 2020; Rostami et al. 2022; Lee et al. 2020; Shrestha et al. 2019, 2021):

$$\delta_i := \sum_o b_{o,i} \, \delta_o \tag{138}$$

where δ_o is the errors attributed to the output neuron o. Note that because feedbacks are neuron-specific and not spike-specific, all post-synaptic spikes fired by a neuron receive the same error signal from feedbacks, therefore $\delta_i^k := \delta_i \; \forall \; t_i^k \in \mathcal{T}_i$. For conciseness, I thus dropped the spike index from δ_i^k. In essence, Equation 138 computes a linear projection of output errors with the feedback matrix $\boldsymbol{B}$. As in DNNs, the elements of this feedback matrix are randomly drawn and kept fixed during training.

Finally, using the projected errors δ_i^{DFA}, the approximate gradient $g^{\text{DFA}}\left(w_{i,j}\right)$ computed by DFA is computed as follows (Neftci et al. 2017b; Bellec et al. 2020; Rostami et al. 2022; Lee et al. 2020; Shrestha et al. 2019, 2021):

$$g^{\text{DFA}}\left(w_{i,j}\right) = \delta_i \nabla_{\text{loc}} w_{i,j} \tag{139}$$

Here, $\nabla_{\text{loc}} w_{i,j}$ is the total local gradient for the weight $w_{i,j}$, locally computed by the neuron in an online manner during inference. It is defined as follows in event-based settings:

$$\nabla_{\text{loc}} w_{i,j} = \sum_{t_i^k \in \mathcal{T}_i} \nabla_{\text{loc}} w_{i,j}^k \tag{140}$$

where $\nabla_{\text{loc}} w_{i,j}^k$ is the local gradient for the spike timing t_i^k. Many definitions exist for these local gradients of spikes. For example, the gradient of the membrane potential at spike times (Lee et al. 2020) or surrogate local gradients (Bellec et al. 2020) can be used. Other works, such as EMSTDP (Shrestha et al. 2021), use biologically-plausible local gradients like STDP — see Section 2.3.8 — or simple accumulations of the number of pre-synaptic spikes as in eRBP (Neftci et al. 2017b).

DFA sidesteps the need for backpropagating errors backward through space and time and allows for online error computation. In addition, because all post-synaptic spikes receive the same projected error, a local gradient of spikes can be locally computed during the inference without immediately requiring an error signal. For this reason, DFA has been successfully implemented in several neuromorphic hardware (Lee et al. 2020; Shrestha et al. 2021; Rostami et al. 2022).

5.3 Spiking Forward Direct Feedback Alignment

When applied to DNNs, the FDFA algorithm propagates directional derivatives alongside neuron activations which are used to estimate derivatives between output and hidden neurons as direct feedback connections. However, the propagation of information differs significantly between spiking and continuously-activated neurons. In SNNs, spikes represent the sole carriers of information while rate-based neurons are continuously activated. Therefore, it becomes necessary to adapt the FDFA algorithm from its original design for continuously activated neurons to one suitable for discrete spiking activations. In the following, I introduce the SFDFA algorithm. I then describe how local gradients can be computed in an online manner, propose an ad hoc solution to gradient explosions caused by critical points in the local gradient, and derive a dynamical system for compatibility with neuromorphic hardware.

Algorithm 6: Inference of a single hidden neuron i with the Spiking Forward Direct Feedback Alignment algorithm.

```
1: Input: The neuron index $i$, the neuron weights $\boldsymbol{w}$, the reset current $\Delta_\vartheta = \vartheta - u_{\text{rest}}$, the random perturbation $p \sim \mathcal{N}(0,1)$ and the set $\mathcal{T}_{\text{pre}}$ of pre-synaptic graded spikes $\left(j, t_j^z, d_j^z\right) \in \mathcal{T}_{\text{pre}}$ sorted in time where $j$ is a pre-synaptic neuron index, $t_j^z$ is a timing and $d_j^z$ is a spike grade.
2: Initialize: the factors $a = 0$ and $b = 0$, the spike count $k$, the local gradient $\boldsymbol{\nabla}_{\text{loc}}\boldsymbol{w} = \boldsymbol{0}$, the set of output events $\mathcal{T}_{\text{post}} = \{\}$, the internal directional derivatives $s = 0$, and the local derivatives synaptic traces $\boldsymbol{f} = \boldsymbol{0}$ and $\boldsymbol{h} = \boldsymbol{0}$.
3: for all $\left(j, t_j^z, d_j^z\right)$ in $\mathcal{T}_{\text{pre}}$ do
4:     $f_j \leftarrow f_j \exp\left(\frac{t_j^z}{\tau_s}\right)$ {Update local derivative traces}
5:     $h_j \leftarrow h_j \exp\left(\frac{t_j^z}{\tau}\right)$
6:     $a \leftarrow a + w_j \exp\left(\frac{t_j^z}{\tau_s}\right)$ {Integrate pre-synaptic spike into quadratic factors}
7:     $b \leftarrow b + w_j \exp\left(\frac{t_j^z}{\tau}\right)$
8:     $s \leftarrow s + w_j\ d_j^z$ {Update internal directional derivative}
9:     if the neuron fires a valid spike then
10:        continue
11:    end if
12:    $t \leftarrow \tau \ln\left(\frac{2a}{b + \sqrt{b^2 - 4a\Delta_\vartheta}}\right)$ {Compute post-synaptic spike timing}
13:    $\boldsymbol{t'} \leftarrow -\frac{\tau}{a_i^k \exp\left(\frac{-t}{\tau_s}\right)} \left[\boldsymbol{h} \exp\left(\frac{-t}{\tau}\right) - \boldsymbol{f} \exp\left(\frac{-t}{\tau_s}\right)\right]$ {Compute spike derivative}
14:    $\boldsymbol{\nabla}_{\text{loc}}\boldsymbol{w} \leftarrow \boldsymbol{\nabla}_{\text{loc}}\boldsymbol{w} + \boldsymbol{t'}$ {Update local gradient}
15:    $\boldsymbol{h} \leftarrow \boldsymbol{h} - \frac{\Delta_\vartheta}{\tau} \exp\left(\frac{t}{\tau}\right) \boldsymbol{t'}$ {Apply recurrence}
16:    $b \leftarrow b - \Delta_\vartheta \exp\left(\frac{t}{\tau}\right)$ {Reset of membrane potential}
17:    $d \leftarrow s + p$ if $k = 0$ else $s$ {Compute directional derivative as spike grade}
18:    add $(i, t, d)$ to $\mathcal{T}_{\text{post}}$
19: end for
20: return $\mathcal{T}_{\text{post}}$, $\boldsymbol{\nabla}_{\text{loc}}\boldsymbol{w}$
```

Algorithm 7: Weights update of a single hidden neuron i with the Spiking Forward Direct Feedback Alignment algorithm.

```
1: Input: the neuron weights $\boldsymbol{w}$, the local gradient $\boldsymbol{\nabla}_{\text{loc}}\boldsymbol{w}$, the feedback connections $\boldsymbol{b}$, the random perturbation $p$, the output directional derivatives $\boldsymbol{d}$, the output errors $\boldsymbol{\delta}$ and the learning rates $\lambda$ and $\alpha$.
2: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \lambda\ \boldsymbol{\delta}\boldsymbol{b}\boldsymbol{\nabla}_{\text{loc}}\boldsymbol{w}$ {Update weights}
3: $\boldsymbol{b} \leftarrow (1-\alpha)\,\boldsymbol{b} + \alpha\ p\boldsymbol{d}$ {Update feedback conneections}
4: return $\boldsymbol{w}$, $\boldsymbol{b}$
```

When using DFA with SNNs, feedback connections essentially replace the inter-neuron dependencies of spikes. As FDFA estimates the expected derivatives between hidden and output neurons, a spiking adaptation of the algorithm would estimate the expected derivative between output and hidden spike timings, such as:

$$\mathop{\mathbb{E}}_{\boldsymbol{x},\, t_o^z,\, t_i^k}\left[\frac{\partial t_o^z}{\partial t_i^k}\right] = \mathop{\mathbb{E}}_{\boldsymbol{x},\, t_o^z,\, t_i^k}\left[\Theta\left(t_o^z - t_i^k\right) w_{o,i}\left[\frac{\partial t_o^z}{\partial a_o^z}\frac{\exp\left(\frac{t_i^k}{\tau_s}\right)}{\tau_s} + \frac{\partial t_o^z}{\partial b_o^z}\frac{\exp\left(\frac{t_i^k}{\tau}\right)}{\tau}\right]\right] \tag{141}$$

Here, $\boldsymbol{x}$ is an input sample from a dataset. It is important to note that the expectation is not solely taken over the dataset but also encompasses the post-synaptic spikes of both the neuron i and the output neuron o. This distinction arises from the fact that neurons can generate multiple spikes, yet they possess only a single linear feedback connection shared by all their spikes.

If the neuron i shares a direct connection with the output neuron o, Equation 141 decomposes as follows:

$$\begin{aligned}\mathop{\mathbb{E}}_{\boldsymbol{x},\, t_o^z,\, t_i^k}\left[\frac{\partial t_o^z}{\partial t_i^k}\right] &= w_{o,i}\mathop{\mathbb{E}}_{\boldsymbol{x},\, t_o^z,\, t_i^k}\left[\Theta\left(t_o^z - t_i^k\right)\left[\frac{\partial t_o^z}{\partial a_o^z}\frac{\exp\left(\frac{t_i^k}{\tau_s}\right)}{\tau_s} + \frac{\partial t_o^z}{\partial b_o^z}\frac{\exp\left(\frac{t_i^k}{\tau}\right)}{\tau}\right]\right] \\ &= w_{o,i}\mathop{\mathbb{E}}_{\boldsymbol{x}}\left[f\left(\mathcal{T}_o, \mathcal{T}_i\right)\right]\end{aligned} \tag{142}$$

where

$$f\left(\mathcal{T}_o, \mathcal{T}_i\right) = \mathop{\mathbb{E}}_{t_o^z \in \mathcal{T}_o}\mathop{\mathbb{E}}_{t_i^k \in \mathcal{T}_i}\left[\Theta\left(t_o^z - t_i^k\right)\left[\frac{\partial t_o^z}{\partial a_o^z}\frac{\exp\left(\frac{t_i^k}{\tau_s}\right)}{\tau_s} + \frac{\partial t_o^z}{\partial b_o^z}\frac{\exp\left(\frac{t_i^k}{\tau}\right)}{\tau}\right]\right] \tag{143}$$

is a function of the spike trains $\mathcal{T}_o$ and $\mathcal{T}_i$ accounting for the temporal relationship between spikes. Due to the inherent variability in spike counts and spike timings, this function exhibits significant variance, which can impact the convergence of the derivative estimates. Therefore, to mitigate the influence of large variance on the convergence of feedback connections, I exclude the temporal relationships between spikes from the derivative estimates and only estimate the weights between hidden and output neurons.

Formally, we draw a random perturbation $p_i^1 \sim \mathcal{N}(0,1)$ for the first spike t_i^1 of each hidden neuron i. For all the other post-synaptic spikes, i.e. when $k > 1$, the spike perturbations $p_i^k = 0$ are set to zero. We then denote as d_i^k the grade of the spike — i.e. an additional payload value carried by spike events, see Section 2.3.6 for more details — fired at time t_i^k by the neuron i. It is defined as:

$$d_i^k = s_i\left(t_i^k\right) + p_i^k \tag{144}$$

where $s_i(t)$ is a variable defined for each neuron, updated as follows when a pre-synaptic spike is received at time t_j^z from a neuron j:

$$s_i\left(t_i^z\right) \leftarrow s_i\left(t_i^z\right) + w_{i,j}\ d_j^k \tag{145}$$

and is reset to zero when a post-synaptic spike is received.

In essence, $s_i(t)$ accumulates the spike grades d_j^z propagated by pre-synaptic neurons over time, during inference. When a post-synaptic spike is fired, $s_i(t)$ is then propagated to downstream neurons through spike grades, along with perturbations p_i^k. Spike perturbations are only drawn for the first spikes of neurons as they are more likely to have a causal relationship with downstream spikes.

Therefore, a directional derivative d_o can be computed for each output neuron o by summing all the output spike grades. In the case of two-layers fully connected SNN and assuming that all hidden spikes are causal to output spikes, it is:

$$\begin{aligned} d_o &= \sum_{t_o^k \in \mathcal{T}_o} d_o^k \\ &= \sum_{i \in \mathcal{P}_o} w_{o,i} \sum_{t_i^z \in \mathcal{T}_i} d_i^z \\ &= \sum_{i \in \mathcal{P}_o} w_{o,i} \sum_{t_i^k \in \mathcal{T}_i} p_i^k \\ &= \sum_{i \in \mathcal{P}_o} w_{o,i}\, p_i^1 \end{aligned} \tag{146}$$

Note the similarity with the directional derivative computed in FDFA — see Equation 105.

Multiplying d_o by the perturbation p_i^1 thus leads to an unbiased estimate of the weight $w_{o,i}$, such as:

$$\begin{aligned} \mathbb{E}\left[d_o\, p_i^1\right] =& \mathbb{E}\left[\sum_{j \in \mathcal{P}_o} w_{o,j}\, p_j^1\, p_i^1\right] \\ =& \sum_{j \in \mathcal{P}_o} w_{o,j}\, \mathbb{E}\left[p_j^1\right] \mathbb{E}\left[p_i^1\right] \\ =& w_{o,i} \end{aligned} \tag{147}$$

As in FDFA, we define an update rule for the feedback connection $b_{o,i}$ that performs an exponential average of the weight estimate, such as:

$$b_{o,i} \leftarrow (1-\alpha)\, b_{o,i} + \alpha\, d_o\, p_i^1 \tag{148}$$

where $0 < \alpha < 1$ is the feedback learning rate.

This update rule can also be written in a form compatible with stochastic gradient descent:

$$b_{o,i} \leftarrow b_{o,i} - \alpha \nabla b_{o,i} \tag{149}$$

where

$$\nabla b_{o,i} = b_{o,i} - d_o \; p_i^k \tag{150}$$

The feedback matrix is used to linearly project output errors and compute error signals for hidden neurons, such as:

$$\delta_i := \sum_o b_{o,i} \; \delta_o \tag{151}$$

The gradient estimate $g^{\text{SFDFA}}\left(w_{i,j}\right)$ is then computed as in DFA, such as:

$$g^{\text{SFDFA}}\left(w_{i,j}\right) = \delta_i \; \nabla_{\text{loc}} w_{i,j} \tag{152}$$

where

$$\nabla_{\text{loc}} w_{i,j} = \sum_{t_i^k \in \mathcal{T}_i} \nabla_{\text{loc}} w_{i,j}^k \tag{153}$$

and $\nabla_{\text{loc}} w_{i,j}^k$ is the local gradient for the spike timing t_i^k.

Finally, the computed gradient estimate is used to update the weights using any gradient-based optimization algorithm such as SGD:

$$w_{i,j} \leftarrow w_{i,j} - \lambda \; g^{\text{SFDFA}}\left(w_{i,j}\right) \tag{154}$$

where $\lambda > 0$ is the learning rate. See Algorithms 6 and 7 for more details about the inference and weight updates with the SFDFA algorithm.

5.3.1 Online Local Gradients

As described in Chapter 3, errors in BP are spatially backpropagated from post-synaptic to pre-synaptic spikes through inter-neuron dependencies due to synaptic interactions between neurons and temporally propagated from post-synaptic to post-synaptic spikes through intra-neuron dependencies due to the reset of the membrane potential. As they cannot be evaluated in an online manner, the inter-neuron dependencies are essentially replaced by the direct feedback connections in SFDFA. On the other hand, the recurrent intra-neuron dependencies can be included in the computation of the local gradients of spikes without violating the online computation requirements, such as:

$$
\begin{aligned}
\nabla_{\text{loc}} w_{i,j}^k :=& \frac{\partial t_i^k}{\partial w_{i,j}} + \sum_{t_i^z \in \mathcal{T}_i} \frac{\partial t_i^k}{\partial t_i^z} \left[\frac{\partial t_i^z}{\partial w_{i,j}} + \sum_{t_i^y \in \mathcal{T}_i} \frac{\partial t_i^z}{\partial t_i^y} \left[\frac{\partial t_i^y}{\partial w_{i,j}} + \dots \right] \right] \\
=& \frac{\partial t_i^k}{\partial w_{i,j}} + \sum_{t_i^z \in \mathcal{T}_i} \frac{\partial t_i^k}{\partial t_i^z} \nabla_{\text{loc}} w_{i,j}^z \\
=& \frac{\tau}{a_i^k} \left[1 + \frac{\Delta_\vartheta}{x_i^k} \exp\left(\frac{t_i^k}{\tau} \right) \right] \sum_{t_j^z \in \mathcal{T}_j} \Theta\left(t_i^k - t_j^z \right) \exp\left(\frac{t_j^z}{\tau_s} \right) \\
& - \frac{\tau}{x_i^k} \sum_{t_j^z \in \mathcal{T}_j} \Theta\left(t_i^k - t_j^z \right) \exp\left(\frac{t_j^z}{\tau} \right) + \sum_{t_i^z \in \mathcal{T}_i} \Theta\left(t_i^z - t_i^k \right) \frac{\Delta_\vartheta}{x_i^k} \exp\left(\frac{t_i^z}{\tau} \right) \nabla_{\text{loc}} w_{i,j}^z \\
=& \frac{\tau}{a_i^k} \left[1 + \frac{\Delta_\vartheta}{x_i^k} \exp\left(\frac{t_i^k}{\tau} \right) \right] f_i^k - \frac{\tau}{x_i^k} h_i^k
\end{aligned}
\tag{155}
$$

where

$$
f_i^k = \sum_{t_j^z \in \mathcal{T}_j} \Theta\left(t_i^k - t_j^z \right) \exp\left(\frac{t_j^z}{\tau_s} \right) \tag{156}
$$

and

$$
h_i^k = \sum_{t_j^z \in \mathcal{T}_j} \Theta\left(t_i^k - t_j^z \right) \exp\left(\frac{t_j^z}{\tau} \right) - \sum_{t_i^z \in \mathcal{T}_i} \Theta\left(t_i^k - t_j^z \right) \frac{\Delta_\vartheta}{\tau} \exp\left(\frac{t_i^z}{\tau} \right) \nabla_{\text{loc}} w_{i,j}^z \tag{157}
$$

only contain sums over previous pre-synaptic and post-synaptic events which satisfies the conditions for online computation. However, to reduce the computational requirements of the local gradient evaluation, Equation 155 can be further simplified by reintroducing the post-synaptic spike time t_i^k into the equation:

$$\begin{aligned}\nabla_{\text{loc}} w_{i,j}^k =& \frac{\exp\left(\frac{-t_i^k}{\tau_s}\right)}{\exp\left(\frac{-t_i^k}{\tau_s}\right)} \frac{\tau}{a_i^k} \left[1 + \frac{\exp\left(\frac{-t_i^k}{\tau}\right)}{\exp\left(\frac{-t_i^k}{\tau}\right)} \frac{\Delta_\vartheta}{x_i^k} \exp\left(\frac{t_i^k}{\tau}\right)\right] f_i^k - \frac{\exp\left(\frac{-t_i^k}{\tau}\right)}{\exp\left(\frac{-t_i^k}{\tau}\right)} \frac{\tau}{x_i^k} h_i^k \\ =& \frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)} \left[1 + \frac{\Delta_\vartheta}{x_i^k \exp\left(\frac{-t_i^k}{\tau}\right)}\right] f_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - \frac{\tau}{x_i^k \exp\left(\frac{-t_i^k}{\tau}\right)} h_i^k \exp\left(\frac{-t_i^k}{\tau}\right)\end{aligned} \tag{158}$$

Then we can isolate an expression for $x_i^k \exp\left(\frac{-t_i^k}{\tau}\right)$, such as:

$$\begin{aligned} & t_i^k = \tau \ln\left(\frac{2a_i^k}{b_i^k + x_i^k}\right) \\ \Leftrightarrow & \exp\left(\frac{t_i^k}{\tau}\right) = \frac{2a_i^k}{b_i^k + x_i^k} \\ \Leftrightarrow & \exp\left(\frac{t_i^k}{\tau}\right) \exp\left(\frac{-t_i^k}{\tau}\right) = \exp\left(\frac{-t_i^k}{\tau}\right) \frac{\exp\left(\frac{-t_i^k}{\tau}\right)}{\exp\left(\frac{-t_i^k}{\tau}\right)} \frac{2a_i^k}{b_i^k + x_i^k} \\ \Leftrightarrow & \exp\left(\frac{t_i^k - t_i^k}{\tau}\right) = \frac{2a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)}{\left(b_i^k + x_i^k\right) \exp\left(\frac{-t_i^k}{\tau}\right)} \\ \Leftrightarrow & 1 = \frac{2a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)}{\left(b_i^k + x_i^k\right) \exp\left(\frac{-t_i^k}{\tau}\right)} \\ \Leftrightarrow & x_i^k \exp\left(\frac{-t_i^k}{\tau}\right) = a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - \Delta_\vartheta \end{aligned} \tag{159}$$

as of the constraint between the synaptic and membrane time constant $\tau = 2\tau_s \Leftrightarrow \exp\left(\frac{-t_i^k}{\tau}\right)^2 = \exp\left(\frac{-t_i^k}{\tau_s}\right)$ — see Chapter 3 for details — and

$$\begin{aligned} u(t_i^k) = b_i^k \exp\left(\frac{-t_i^k}{\tau}\right) - a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) = \Delta_\vartheta \\ \Leftrightarrow b_i^k \exp\left(\frac{-t_i^k}{\tau}\right) = a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) + \Delta_\vartheta \end{aligned} \tag{160}$$

Therefore, the local gradient $\nabla_{\text{loc}} w_{i,j}^k$ of spikes simplifies to:

$$\begin{aligned} \nabla_{\text{loc}} w_{i,j}^k =& \frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)} \left[\frac{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - \Delta_\vartheta}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - \Delta_\vartheta} + \frac{\Delta_\vartheta}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - \Delta_\vartheta} \right] f_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) \\ &- \frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - \Delta_\vartheta} h_i^k \exp\left(\frac{-t_i^k}{\tau}\right) \\ =& \frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - \Delta_\vartheta} \left[f_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - h_i^k \exp\left(\frac{-t_i^k}{\tau}\right) \right] \end{aligned} \tag{161}$$

which requires 8 floating point operations (FLOPS) to evaluate instead of 10 in Equation 155. In this Equation, both factors $\frac{\tau}{a_i^k}\left[1 + \frac{\Delta_\vartheta}{x_i^k} \exp\left(\frac{t_i^k}{\tau}\right)\right]$ and $\frac{\tau}{x_i^k}$ of f_i^k and h_i^k respectively reduce to the same factor $\frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - \Delta_\vartheta}$ which is thus computed only once without requiring x_i^k.

5.3.2 Critical Points of Local Gradients

I now show the existence of critical points where the local gradients of spikes diverge towards infinity and propose a simple ad-hoc solution to avoid gradient explosions.

Knowing that $x_i^k \exp\left(\frac{-t_i^k}{\tau}\right) \geq 0 \Leftrightarrow a_i^k \exp\left(\frac{-t_i^k}{\tau}\right) \geq \Delta_\vartheta$, the factor $\frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - \Delta_\vartheta}$ in Equation 161 contains a critical point where its value diverges towards infinity as

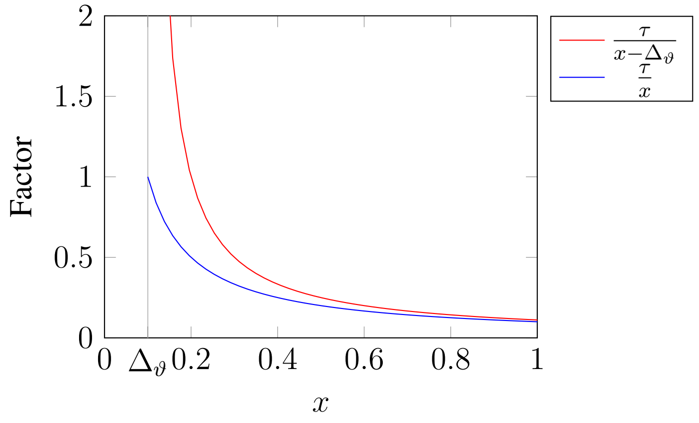


Figure 41: Original factor $\frac{\tau}{x-\Delta_\vartheta}$ (red line) and the modified factor $\frac{\tau}{x}$ as a function of $x = a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)$. As $a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)$ approaches Δ_ϑ, the original factor diverges towards infinity while the modified factor is bounded.

$a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)$ approaches Δ_ϑ (see Figure 41). This situation occurs when the membrane potential barely crosses the threshold, resulting in the quadratic form of $u_i(t) = \vartheta$ having a single real solution — i.e. $x_i^k = 0$. This phenomenon introduces critical points in the parameter space, where the gradient grows exponentially. These critical points have been identified in other works on exact gradients of spikes (Wunderlich and Pehle 2021; Takase et al. 2009a,b). A common solution to mitigate gradient explosion is adding additional mechanisms to limit the size of the gradient such as *gradient clipping* (Wunderlich and Pehle 2021; Hong et al. 2020). However, gradient clipping requires computing the norm of the gradient, which cannot be locally performed on neuromorphic hardware. In this work, I propose a simple modification of the local gradient to remove the cause of the critical points.

Formally, we drop Δ_ϑ from Equation 161, such as:

$$\nabla_{\text{loc}} w_{i,j}^k := \frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)} \left[f_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - h_i^k \exp\left(\frac{-t_i^k}{\tau}\right) \right] \tag{162}$$

Here, $\frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)} \leq \frac{\tau}{\Delta_\vartheta}$ has an upper bound and does not diverge towards infinity (see Figure 41). Dropping the term Δ_ϑ from the equation provides a natural bound without requiring additional mechanisms such as gradient clipping, which should improve the stability of the weights updates. While this creates an additional bias in the gradient estimates, this should not differ from the bias introduced by the feedback connections.

5.3.3 Neuromorphic Hardware Compatibility

The local gradient in Equation 162 is adapted for computation on von Neumann computers. Therefore, to make the SFDFA algorithm compatible with neuromorphic hardware, we derive an eligibility trace that can be evaluated with a system of ordinary differential equations on hardware.

Formally, Equation 162 can be written as:

$$\begin{aligned}\nabla_{\text{loc}} w_{i,j}^k =& \frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)} \left[f_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right) - h_i^k \exp\left(\frac{-t_i^k}{\tau}\right)\right] \\ =& -\frac{\tau}{a_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)} \left[h_i^k \exp\left(\frac{-t_i^k}{\tau}\right) - f_i^k \exp\left(\frac{-t_i^k}{\tau_s}\right)\right] \\ =& -\frac{\tau}{I_i(t_i^k)} e_{i,j}(t_i^k)\end{aligned} \tag{163}$$

where

$$\begin{aligned}I_i(t) =& a_i^k \exp\left(\frac{-t}{\tau_s}\right) \\ =& \sum_{j \in \mathcal{P}_i} w_{i,j} \sum_{t_j^z \in \mathcal{T}_j} \Theta\left(t - t_j^z\right) \alpha\left(t - t_j^z\right)\end{aligned} \tag{164}$$

is the synaptic current at the spike time — see Equation 25 in Section 2.3.3 —, which is computed by the LIF neuron, and

$$\begin{aligned} e_{i,j}(t) = &\sum_{t_j^z \in \mathcal{T}_j} \Theta\left(t - t_j^z\right) \underbrace{\left[\exp\left(\frac{t_j^z - t}{\tau}\right) - \exp\left(\frac{t_j^z - t}{\tau_s}\right)\right]}_{\epsilon\left(t - t_j^z\right)} \\ &- \sum_{t_i^z \in \mathcal{T}_i} \Theta\left(t - t_j^z\right) \underbrace{\Delta_\vartheta \exp\left(\frac{t_i^z - t}{\tau}\right)}_{\eta\left(t - t_i^z\right)} \frac{\nabla_{\text{loc}} w_{i,j}^z}{\tau} \end{aligned} \tag{165}$$

is an eligibility trace computed over time that coincides with the desired expression required by the local gradient at the spike time t_i^k. We can observe that Equation 165 has a form that is similar to the mapping of the LIF neuron to the SRM model — see Section 2.3.4 for details. Therefore, this expression can be implemented on neuromorphic hardware using a LIF model — see Section 2.3.3 for details — at each synapse and locally be used at post-synaptic spike times for the computation of local gradients.

5.4 Experimental Settings

In this section, I describe the experimental settings used to produce my empirical results, including the benchmark datasets, encoding and decoding, network architectures, the loss function, hyperparameters as well as the software and hardware settings.

5.4.1 Benchmark Datasets

I evaluate the performance of the proposed SFDFA algorithm as well as BP and DFA in SNNs with the same four benchmark datasets as in Chapter 3, namely MNIST, EMNIST, Fashion MNIST and the SHD dataset — see Section 3.6.1 for details about these datasets.

5.4.2 Encoding

I used the same encoding as in Chapter 3. See Section 3.6.2 for details.

5.4.3 Network Architectures

I benchmarked my method with fully-connected SNNs of different sizes. For MNIST and EMNIST, I used two-layer SNNs with 800 hidden neurons, for Fashion MNIST, a three-layer SNN with 400 hidden neurons per hidden layer, and a two-layer SNN with 128 hidden neurons for the SHD dataset.

5.4.4 Decoding and Loss Function

I used a spike count strategy to decode the output spike counts of the SNNs. I then used a spike count *Mean Squared Error* loss function for training. See Section 3.6.4 for more details about spike count decoding and spike count Mean Squared Error loss function.

5.4.5 Firing Rate Regularization

Without additional constraint, neurons may exhibit high firing rates to achieve lower loss values which could increase energy consumption and computational requirements. To prevent this issue, I implemented a firing rate regularization that drives the mean firing rate of neurons towards a given target during training. By incorporating firing rate regularization, the neural network is encouraged to find a balance between learning from the data and avoiding high firing rates. This can lead to improved generalization, reduced energy consumption, and enhanced stability.

Formally, a penalty term for high firing rates is added to the loss function, such as:

$$\mathcal{L}_{\text{reg}}(\boldsymbol{x}) = \mathcal{L}(\boldsymbol{x}) + \beta \sum_{i} \left(\mathop{\mathbb{E}}_{\boldsymbol{x}} \left[n_i \right] - \widehat{n} \right)^2 \tag{166}$$

where $\beta > 0$ is a constant defining the strength of the regularization and $\mathop{\mathbb{E}}_{\boldsymbol{x}} [n_i]$ is the mean firing rate of the hidden neuron i. The mean firing rate can be estimated in an online manner by computing a moving average or an exponential average of the firing rate over the last inference. If batch learning is used, a mean firing rate can be computed from the batch.

5.4.6 Update Method and Hyperparameters

I used the Adam (Kingma and Ba 2017) algorithm — see Section 2.4.6 for details — to update both the weights and the feedback connections. I used the default values of $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ (Kingma and Ba 2017) and a batch size of 50 for fully-connected SNNs. I used a learning rate of $\lambda = 0.003$ for image classification and $\lambda = 0.001$ for audio classification with the SHD dataset. In addition, I used a feedback learning rate of $\alpha = 10^{-4}$ in every experiment. These hyperparameters were choosen manually to produce the best performance on each task.

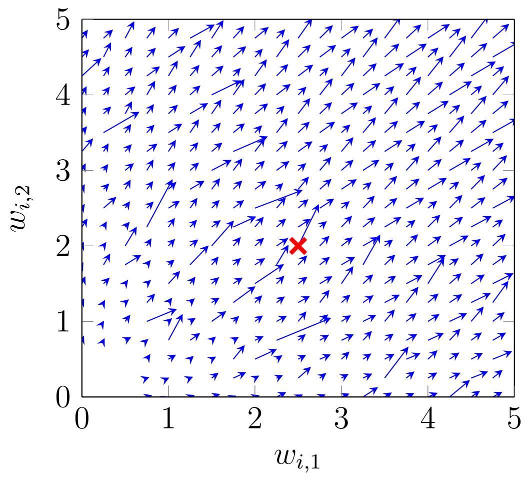
Experimental conditions were standardized for BP, DFA and SFDFA. I used the same hyperparameters as the method proposed in Chapter 3 — see Section 3.6.5 for details.

5.4.7 Event-Based Simulations on GPU

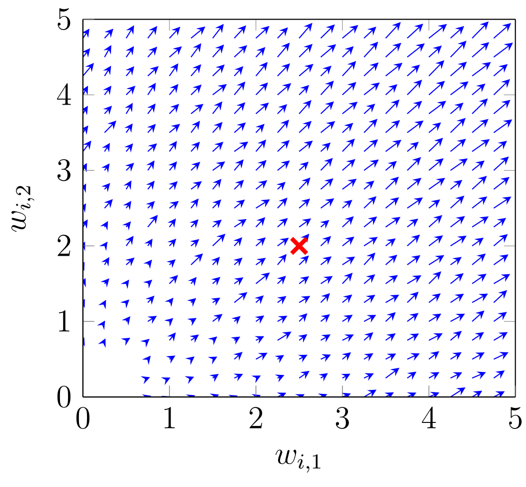
To simulate and train SNNs, I reused the event-based simulator introduced in Chapter 3 — see Section 3.6.6 for details. I adapted the GPU kernels related to the inference to compute local gradients as well as directional derivatives in an online manner. Moreover, I replaced the code performing error backpropagation with feedback learning. Similarly to neuromorphic hardware, my simulator never backpropagates errors backward through time and performs all computations in an online manner.

5.5 Empirical Results

I now present my empirical results with the proposed SFDFA algorithm. I empirically highlight the critical points in the exact local gradient of spikes and demonstrate the effectiveness of the modified local gradient in preventing gradient explosions. I then



(a) Exact local gradient



(b) Modified local gradient

Figure 42: Gradient fields of a single neuron computed using the exact local gradients (Figure 42a) and the modified local gradients (Figure 42b). I can observe that the exact local gradient contains critical points where the norm is abnormally large compared to neighboring regions. However, with the modified local gradients, the norm of the gradient is consistent throughout the weight space, mitigating the gradient explosion caused by the critical points. The red crosses at $w_{i,1} = 2.5$ and $w_{i,2} = 2.0$ correspond to the critical point visualized in Figure 43.

compare the performance and convergence rate of SFDFA with DFA and BP on several benchmark datasets as well as the weight and gradient alignment of DFA and SFDFA.

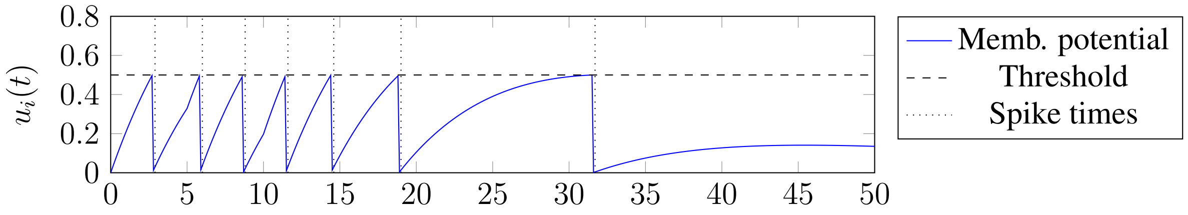
5.5.1 Critical Points Analysis

I now analyze the stability of the exact local gradient of spikes and compare it with the modified local gradient, defined in Equation 162.

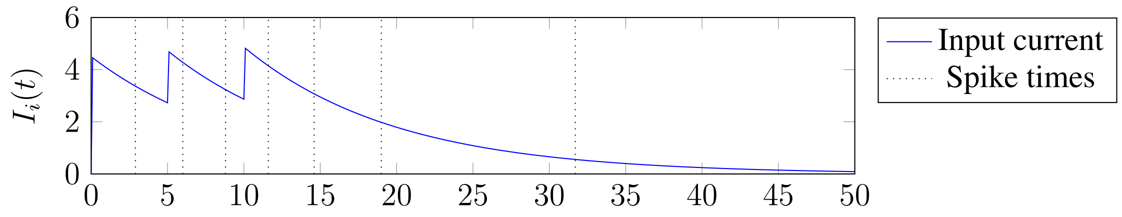
Figure 42 shows the respective gradient fields computed using the exact and modified local gradients. These figures have been produced by varying the weights of a two-input neuron receiving four pre-synaptic spikes and evaluating the local gradient $\nabla_{\text{loc}}\boldsymbol{w}$ with the exact derivative of spikes (Equation 161) and with the modified local gradient (Equation 162). It can be observed in Figure 42a that the exact local gradient contains several critical points where its norm is significantly larger than neighboring vectors.

To investigate the cause behind these large norms, I examined the neuron's internal state over time at one of these critical points (marked in red in Figure 42). Figure 43 depicts the temporal evolution of the membrane potential, input current, factor $\frac{\tau}{I_i(t)-\Delta_\vartheta}$, and the computed local gradient. Notably, it can be observed that the last post-synaptic spike fired by the neuron narrowly crosses the threshold (i.e. hair trigger) due to a low input current. This low current leads to the factor $\frac{\tau}{I_i(t)-\Delta_\vartheta}$ becoming large due to its divergence when $I_i(t)$ approaches Δ_ϑ (see Figure 41). Consequently, the local gradient at this spike time significantly increases.

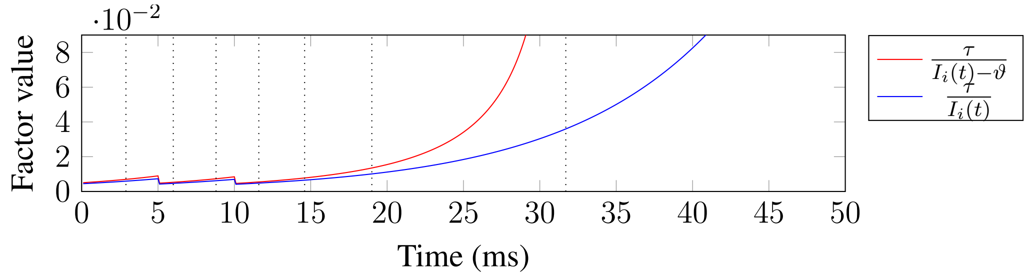
On the other hand, Figure 42 demonstrates that the proposed modified local gradient maintains a consistent norm across the weight space without any instability points where the norm deviates abnormally from neighboring vectors. Additionally, Figure 43 illustrates that, at the spike time when the membrane potential narrowly reaches the threshold, the modified factor $\frac{\tau}{I_i(t)}$ exhibits a significantly lower value compared to the original factor $\frac{\tau}{I_i(t)-\Delta_\vartheta}$. This is because the modified factor is upper-bounded, as shown in Figure 41. As a result, the contribution of this spike to the modified local gradient is substantially reduced in comparison to its contribution to the exact local gradient.



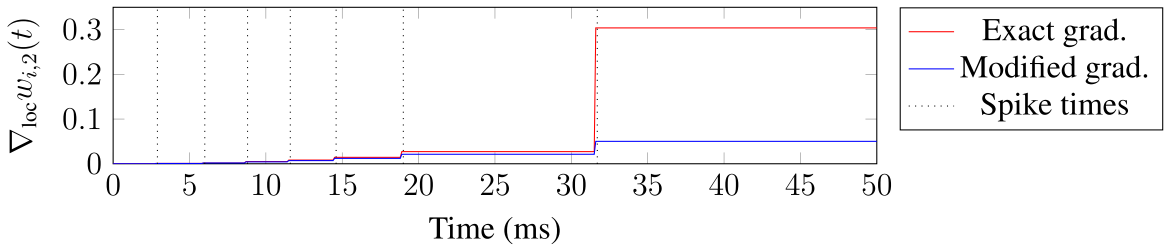
(a) Membrane potential



(b) Input current



(c) Factors

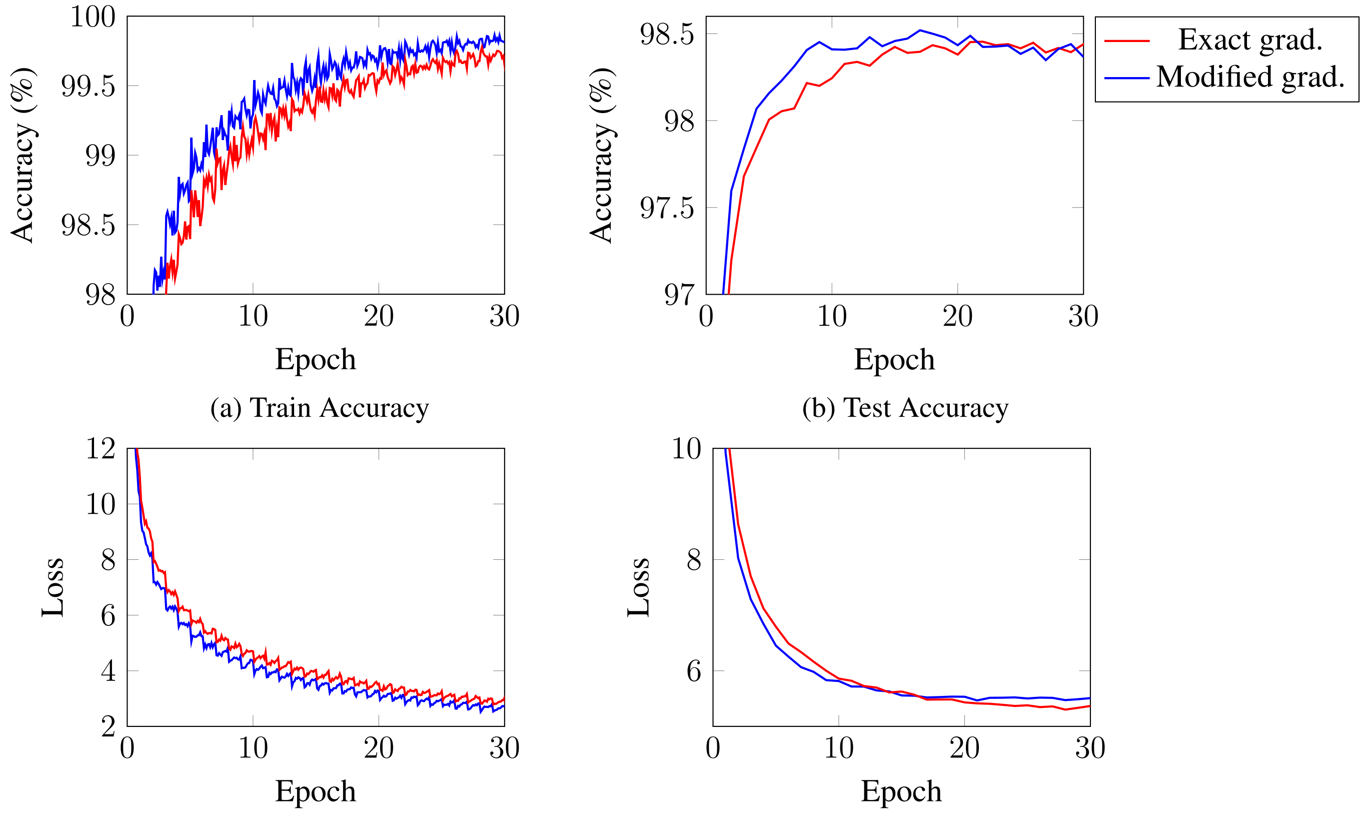


(d) Local gradient

Figure 43: Evolution over time of the membrane potential (Figure 43a), input current (Figure 43b), gradient factors (Figure 43c) and local gradients (Figure 43d) at an instability point ($w_{i,1} = 2.5$ and $w_{i,2} = 2.0$ in Figure 42). Here all vertical dotted lines correspond to post-synaptic spike times. The red lines in both Figures 43c and 43d represent the original factor and local gradient, while the blue lines represent the modified factor and local gradient. These figures show that the last post-synaptic spike fired by the neuron occurs when the membrane potential narrowly reaches the threshold. This narrow threshold crossing is attributed to the low input current $I_i(t)$ which causes large factors $\frac{\tau}{I_i(t)-\Delta_\vartheta}$ and consequently large local gradients. In contrast, the modified factor $\frac{\tau}{I_i(t)}$ restricts the amplitude of the factor, thereby moderating the scale of the gradient.

To assess the impact of critical points on the convergence of SNNs, I conducted an experiment using the MNIST dataset. Identical SNNs were trained, both containing two layers of fully connected neurons and sharing the same hyperparameters. These networks were initialized with identical weights, differing only in the type of local gradient used for training. One network was trained with the exact local gradient, while the second network was trained with the modified gradient defined in Equation 162.

Figure 44 shows the evolution of the training loss, training accuracy, test loss, and test accuracy for each network during the training of each network. It can notably be observed that the network trained with the modified local gradient converges slightly faster than the SNN trained with the exact local gradient. This suggests that the modified gradient mitigates the impact of instabilities on convergence, leading to enhanced learning.



(a) Train Accuracy (b) Test Accuracy

(c) Train Loss (d) Test Loss

Figure 44: Evolution of the training accuracy (Figure 44a), test accuracy (Figure 44b), train loss (Figure 44c) and test loss (Figure 44d) during the training of a two-layers SNN on the MNIST dataset. Red lines correspond to the metrics of the SNN updated using the exact local gradient while blue lines correspond to the metrics of the SNN updated using the modified local gradient defined in Equation 162. These figures show that the modified gradient converges slightly faster than the exact local gradient.

5.5.2 Performance

In order to evaluate the efficiency of the proposed SFDFA method, I conducted a comparative analysis of its performance against BP and DFA. To ensure a fair comparison, I maintained uniformity across all hyperparameters, maintaining consistency with the experiments conducted in Chapter 3.

Table 17 summarizes the performance of each method on each dataset. We can

Table 17: Average best test performance of BP, DFA and the proposed SFDFA on the MNIST, EMNIST, Fashion MNIST and SHD datasets.

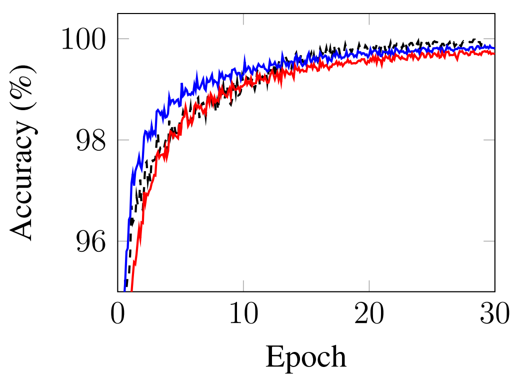
Dataset	Architeture	BP	DFA	SFDFA
MNIST	800-10	$98.88 \pm 0.02\%$	$98.42 \pm 0.06\%$	$98.56 \pm 0.04\%$
EMNIST	800-47	$85.75 \pm 0.06\%$	$79.48 \pm 0.11\%$	$82.33 \pm 0.10\%$
Fashion MNIST	400-400-10	$90.19 \pm 0.12\%$	$89.41 \pm 0.12\%$	$89.73 \pm 0.17\%$
SHD	128-20	$66.79 \pm 0.66\%$	$52.70 \pm 2.30\%$	$54.63 \pm 1.16\%$

observe that both the DFA and SFDFA algorithms achieve test performance similar to that of BP for the MNIST, EMNIST, and Fashion MNIST datasets. However, for the SHD dataset, neither algorithm attains a level of accuracy comparable to BP. This indicates that the direct feedback learning approach with a single error signal fails to effectively generalize when applied to temporal data.

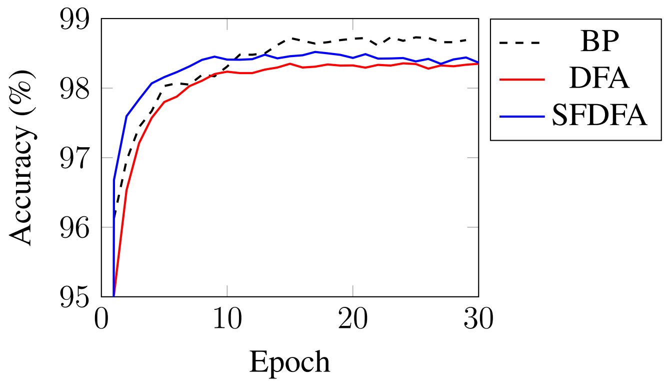
Despite this limitation, the proposed SFDFA algorithm consistently outperforms DFA across all benchmarked datasets. However, there remains a noticeable gap between the performance of SFDFA and BP, particularly as the complexity of the task increases. These results thus align with the observations made regarding the FDFA algorithm in Chapter 4.

5.5.3 Convergence

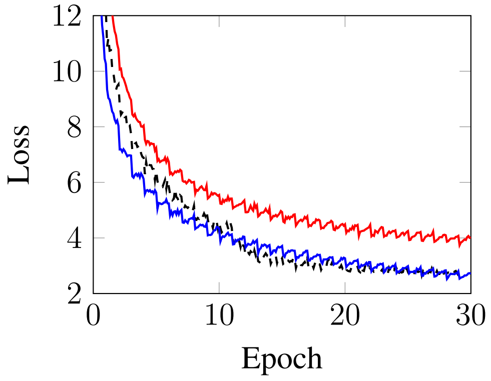
The FDFA algorithm introduced in Chapter 4 demonstrated improvements in terms of convergence compared to DFA. To evaluate if this is also the case in SNNs, I recorded the train loss, train accuracy, test loss, and test accuracy of SNNs trained with the DFA and SFDFA algorithms on each dataset.



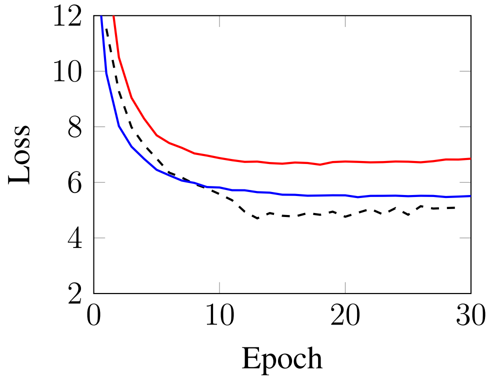
(a) Train Accuracy



(b) Test Accuracy

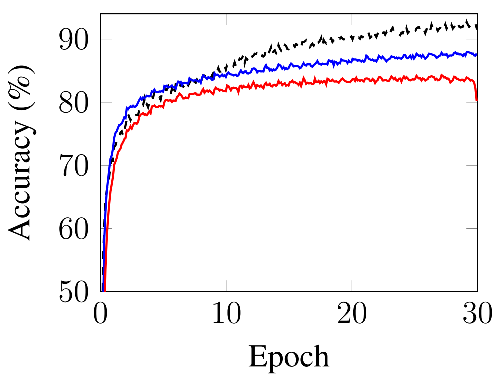


(c) Train Loss

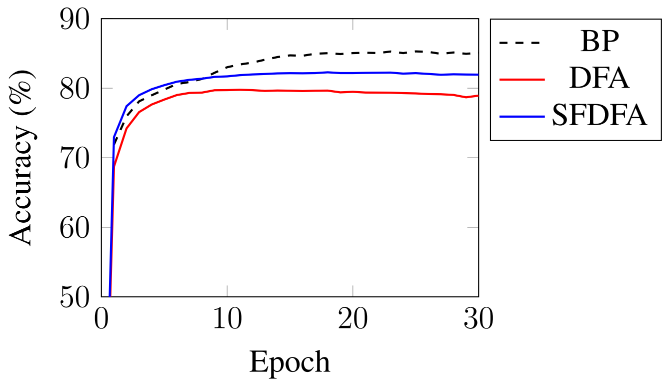


(d) Test Loss

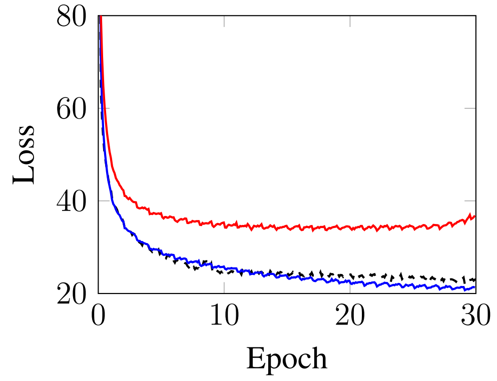
Figure 45: Evolution of the averaged train accuracy (Figure 45a), test accuracy (Figure 45b), train loss (Figure 45c) and test loss (Figure 45d) during the training of two-layers SNNs on the MNIST dataset. Black dashed lines correspond to BP. Blue and red solid lines correspond to the SFDFA and DFA algorithms respectively.



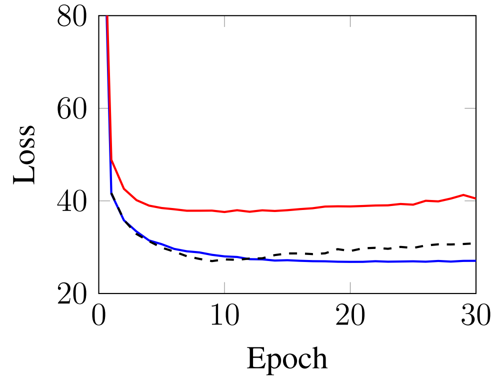
(a) Train Accuracy



(b) Test Accuracy

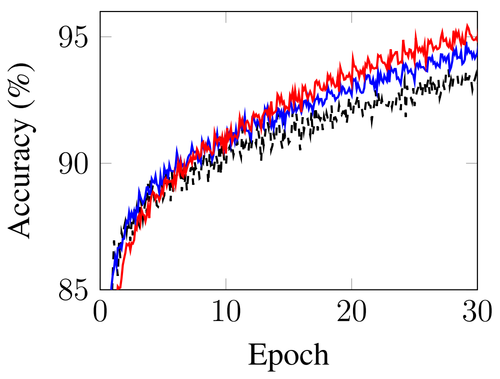


(c) Train Loss

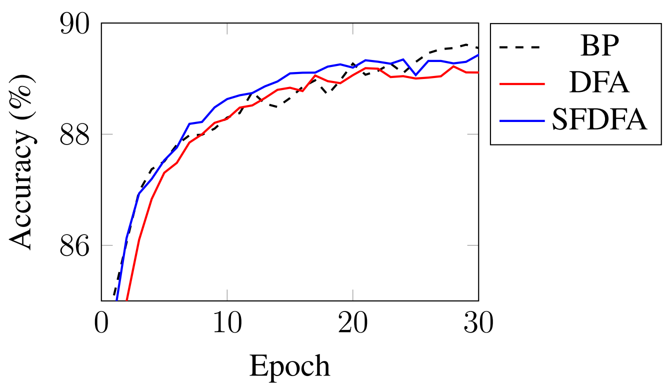


(d) Test Loss

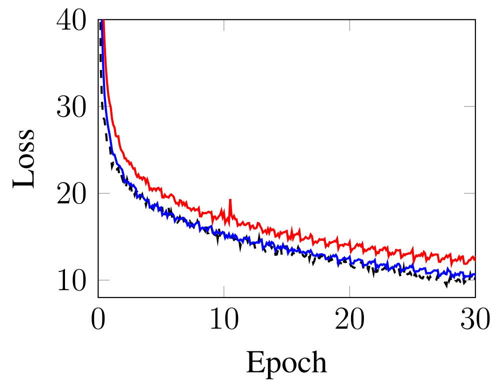
Figure 46: Evolution of the averaged train accuracy (Figure 45a), test accuracy (Figure 45b), train loss (Figure 45c) and test loss (Figure 45d) during the training of two-layers SNNs on the EMNIST dataset. Black dashed lines correspond to BP. Blue and red solid lines correspond to the SFDFA and DFA algorithms respectively.



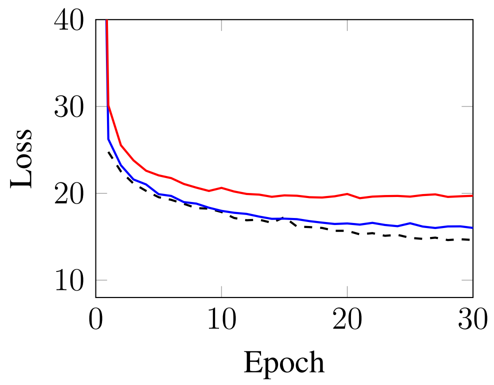
(a) Train Accuracy



(b) Test Accuracy

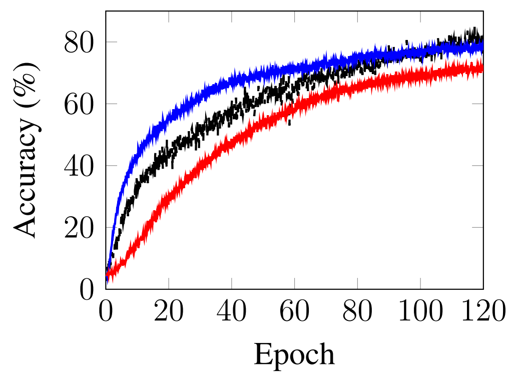


(c) Train Loss

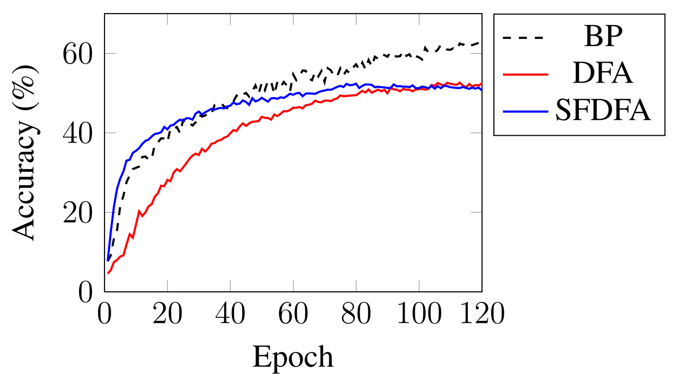


(d) Test Loss

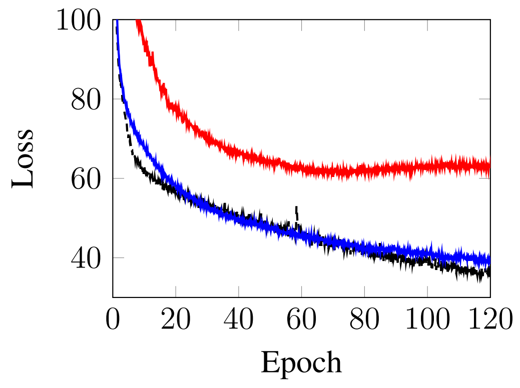
Figure 47: Evolution of the averaged train accuracy (Figure 45a), test accuracy (Figure 45b), train loss (Figure 45c) and test loss (Figure 45d) during the training of two-layers SNNs on the Fashion MNIST dataset. Black dashed lines correspond to BP. Blue and red solid lines correspond to the SFDFA and DFA algorithms respectively.



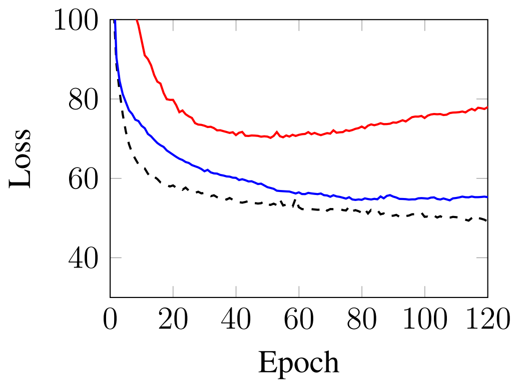
(a) Train Accuracy



(b) Test Accuracy



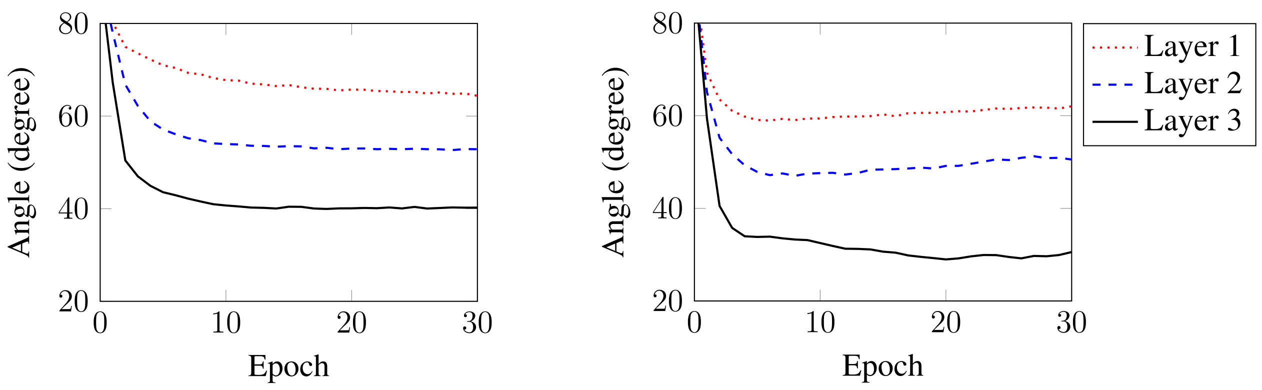
(c) Train Loss



(d) Test Loss

Figure 48: Evolution of the averaged train accuracy (Figure 45a), test accuracy (Figure 45b), train loss (Figure 45c) and test loss (Figure 45d) during the training of two-layers SNNs on the Spiking Heidelberg Digits dataset. Black dashed lines correspond to BP. Blue and red solid lines correspond to the SFDFA and DFA algorithms respectively.

Figures 45, 46, 47 and 48 show the evolution of the different metrics with the MNIST, EMNIST, Fashion MNIST and SHD datasets respectively. In these figures, the metrics recorded in Chapter 3 for BP are also given as a baseline. It can be observed that a gap exists in both the train and test loss between the BP and DFA algorithms. In contrast, the train and test loss for the proposed SFDFA algorithm closely follow the loss values of BP, indicating similar convergence rates. Interestingly, during the initial stage of training, both the train and test accuracies of the SFDFA algorithm increase faster than both BP and DFA. This behavior is particularly prominent in Figures 45 and



(a) DFA

(b) SFDFA

Figure 49: Layerwise alignment between spiking gradient estimates and the true gradient computed using BP. These figures show that the SFDFA algorithm (Figure 49b) aligns better with the true gradient than DFA (Figure 49a), especially in layers close to the outputs.

48. However, in the later stage of training, the test accuracy of SFDFA plateaus while the test accuracy of BP continues to improve. This suggests that the proposed SFDFA algorithm does not generalize as much as BP on test data. However, compared to DFA, the test accuracy of the SFDFA algorithm increases significantly faster, especially on temporal data (see Figure 48). These results on convergence align with the results observed with the FDFA algorithm on DNNs in Chapter 4. From a general point of view, the SFDFA algorithm seems to improve the rate of convergence of DFA, especially as the task increases in difficulty.

5.5.4 Weights and Gradient Alignment

Similarly to Section 4.9.4, I measured the bias of the gradient estimates by recording the layerwise alignment between the approximate gradients and the true gradients computed using BP. For this experiment, I trained a 4-layer SNN on the MNIST dataset for 30 epochs using both the DFA and SFDFA algorithms to compute gradient estimates. I then calculated the angles between these estimates and the true gradient computed by

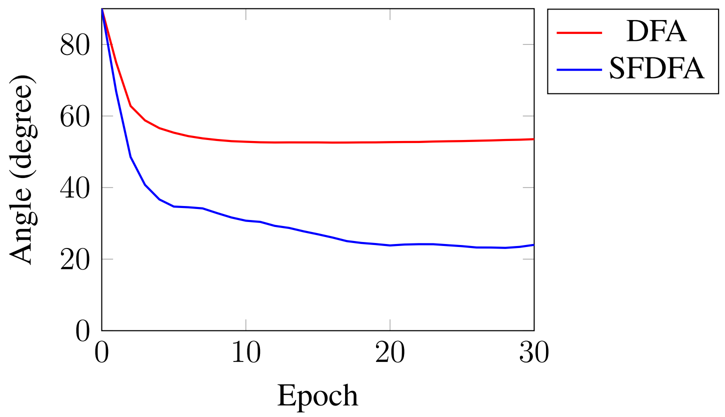


Figure 50: Evolution of the alignment between the output weights and last hidden layer feedbacks in a 4-layer SNN trained on MNIST with DFA (red line) and SFDFA (blue line). The weights and feedback connections align faster and better with the SFDFA algorithm than with DFA. Moreover, the weights alignment in SNNs correlates with the gradient alignment (see Figure 49).

BP.

The evolution of the resulting angles during training is given in Figure 49. It can be observed that the gradient estimates provided by FDFA align faster and exhibit a lower angle with respect to BP when compared with DFA. This indicates that the proposed SFDFA algorithm achieves a lower level of bias earlier than DFA, with weight updates that better follow BP. This result is consistent with the observations made with the FDFA algorithm in Chapter 4. However, it is important to note that the gap in alignment between SFDFA and DFA diminishes with the depth of the layer. Specifically, when comparing the alignments of the third layer (i.e. closest to the output) trained with SFDFA and DFA, the former demonstrates significantly better alignment. However, in the case of the first layer, both methods exhibit similar levels of alignment. This suggests that the benefits of SFDFA become more pronounced in layers close to the outputs.

In contrast with the FDFA algorithm which estimates derivatives, the SFDFA algorithm approximates the derivatives of spikes by estimating the weights connections between hidden and output neurons. Therefore, in addition to the layerwise gradient alignment, I measured the angle between the vectors represented by the output weights and the vector represented by the feedback connections for the last hidden layer. Also known as *weight alignment*, it is believed to be the main source of gradient alignment in DFA (Refinetti et al. 2021). Figure 50 shows the evolution of the alignment between the flattened output weights and the flattened feedback connections to the last hidden layer. The alignments for deeper hidden layers were ignored as the exact forward weights for these layers are unknown. We can see in this figure that the output weight and the feedback connections trained with the SFDFA algorithm align faster and better than those trained with DFA. Moreover, we can observe that the weight alignment correlates with the gradient alignment of the third layer in Figure 49. This suggests that estimating the forward weights as feedback connections makes the approximate gradients align with the true gradients.

5.6 Discussion

In this chapter, I proposed the SFDFA algorithm, a spiking adaptation of FDFA that trains SNNs in an online and local manner. The proposed algorithm computes local gradients of post-synaptic spikes by taking into account all intra-neuron dependencies and uses direct feedback connections to linearly project output errors to hidden neurons without unrolling the neuron's dynamics through space and time. Similarly to the FDFA algorithm, SFDFA estimates the derivatives between hidden and output neurons as feedback connections by propagating directional derivatives as spike grades during the inference. More precisely, SFDFA estimates the weights between output and hidden and ignores the temporal relationships between spikes to avoid large variances that could hinder the convergence of feedback connections.

I also demonstrated the existence of critical points where the norm of the exact local gradient diverges towards infinity. These critical points have previously been discovered in other formulations of exact gradients (Takase et al. 2009a,b; Wunderlich and Pehle 2021) of spikes. In my work, I identified the cause of these critical points in the computation of the exact local gradients and proposed a simple modification of the derivatives that suppresses gradient explosions. While this ad hoc solution introduces an additional bias to the gradient estimates, I showed that it enabled faster convergence of SNNs than the exact gradient when trained with SFDFA. This implies that the rate of convergence of my algorithm benefits from the improved stability of the modified local gradient.

My empirical results showed that the proposed SFDFA consistently converges faster and achieves higher performance than DFA on all benchmark datasets. However, while my method also performs better than DFA on the SHD dataset, a significant gap still exists with BP. This suggests that the learning procedure used in both DFA and SFDFA has limitations when applied to highly temporal data. Therefore, future work could explore alternative update methods to improve this performance gap with BP on temporal data.

In my experiments, I measured the alignment between the approximate gradients computed by DFA and SFDFA and the true gradient computed by BP. My results showed that the approximate gradients computed by the proposed SFDFA algorithm align faster with the true gradient than DFA. This suggests that weights are updated with steeper descending directions in SFDFA than in DFA. This could explain the increased convergence rate experienced by my algorithm. However, we observed that the gradient estimates in SNNs align less than the FDFA algorithm applied to DNNs. This weak alignment can be explained by two factors. First, by bounding the local derivatives, the modified local gradient of my method slightly changes the direction of the weight updates. Second, the complex temporal relationships between spikes are ignored in the computation of the directional derivatives to avoid introducing large variances in the feedback updates. Ignoring these temporal relationships could make the approximate gradients further deviate from the true gradient.

In addition to the gradient alignment, I measured the alignment between the network weights and feedback connections in both DFA and SFDFA. I observed a stronger difference in weight alignment between DFA and SFDFA than in gradient alignment. This could be explained by the fact that my method estimates the weight connections between output and hidden neurons as feedback rather than derivatives. In particular, we observed that the weight alignment of SFDFA correlates with the gradient alignment of the last layer, suggesting that estimating weights contributes to the gradient alignments. However, it is still unclear why deeper layers fail to align more than in DFA. Future work could therefore focus on improving the gradient alignment of deep layers to improve the rate of convergence as well as the performance of SNNs when trained with SFDFA.

From an engineering point of view, the local gradients of spikes can be locally computed by neurons by implementing dedicated circuits that evaluate the dynamical system derived in Section 5.3.3. Moreover, the computation and propagation of directional derivatives during the inference can be implemented through the grades of spikes. Spike grades are features that have recently been added to several large-scale neuromorphic platforms such as Loihi 2 (Frady et al. 2022) and SpiNNaker (Furber et al. 2014). Traditionally used to modify the amplitude of spikes for computational purposes, my work instead proposes the use of spike grades for learning purposes. Finally, direct feedback connections have widely been implemented on various neuromorphic platforms, thus supporting the hardware compatibility of SFDFA.

Therefore, by successfully addressing the limitations of BP, the proposed SFDFA algorithm represents a promising step towards the implementation of neuromorphic gradient descent. While there are still areas for improvement and exploration, my findings contribute to the growing body of knowledge aimed at improving the field of neuromorphic computing.

Chapter 6

Discussion

The lack of differentiable spikes and the limitations of BP represent the main obstacles to the application of gradient descent on neuromorphic hardware. This thesis addresses these challenges by extending *Fast & Deep* (Göltz et al. 2021) to SNNs producing multiple spikes per neuron and by proposing a novel algorithm that computes gradient estimates in an online manner without relying on BP. By combining these two solutions, I proposed an algorithm that addresses the online requirements and local constraints of neuromorphic hardware.

Introduced in Chapter 3, my first proposed method relaxes the spike constraint of temporal coding in Fast & Deep (Göltz et al. 2021) to compute exact gradients of unconstrained SNNs. By allowing multiple spikes per neuron instead of a single one, my extension both increases the rate of information propagated by neurons and the likelihood of convergence. As shown by my empirical experiments, relaxing the single spike constraint enables improved tradeoffs between accuracy, convergence, sparsity, classification latency, and robustness to noise and weight quantization when compared to temporally-coded SNNs trained with Fast & Deep. However, while my empirical experiments demonstrated promising results, the non-local and backward aspects of BP make its implementation too costly on neuromorphic hardware, if not entirely impractical. Therefore, the use of exact gradients is mainly limited to offline training or in-the-loop, where the inference is performed on neuromorphic hardware and the gradients are computed on von Neumann computers.

To overcome the limitations of BP, many alternatives have previously been proposed such as *Forward Gradients* (Baydin et al. 2022) or *Direct Feedback Alignment* (Nøkland 2016). However, these methods suffer from slow convergence, either because of high variance limiting the rate of convergence of SGD, or because of significant biases making the gradient estimates deviate from the direction of fastest descent. In Chapter 4, I proposed the *Forward Direct Feedback Alignment* (FDFA) algorithm, a novel local and online alternative to BP combining Forward Gradients with Direct Feedback Alignment and momentum. By using activity-perturbed Forward Gradients, the proposed FDFA algorithm learns the derivatives between output and hidden neurons as direct feedback connections. The feedback learning rule introduced in my approach acts as a momentum that reduces the variance of the gradient estimates. In addition to significantly improving the rate of convergence, the FDFA gradient estimates achieve greater performance and update weights in a direction that better follows the direction of steepest descent when compared to other alternatives to BP with DNNs.

Motivated by its performances with DNNs, I introduced in Chapter 5 a spiking adaptation of the FDFA algorithm. Referred to as *Spiking Forward Direct Feedback Alignment* (SFDFA), the proposed method computes local gradients of spikes in an online manner and propagates directional derivatives through the grade of spikes, an additional payload value carried by spike events. In addition, I demonstrated the existence of critical points causing gradient explosions. I thus proposed an ad-hoc solution that bounds the local gradients of spikes and improves convergence despite an additional bias. Finally, I derived a dynamical system for the computation of local gradients on neuromorphic hardware. my empirical results have demonstrated the advantages of the SFDFA algorithms over DFA both in terms of convergence and performance.

In conclusion, by addressing the inherent lack of differentiable spikes and the limitations of BP, this thesis introduces novel perspectives for neuromorphic-compatible gradient descent. Although a slight performance gap persists compared to backpropagation, the proposed method has demonstrated improved capabilities to train SNNs with higher rates of convergence and increased performance when compared with the widely-used DFA algorithm. Furthermore, my approach only performs computations in a local and online manner that complies with the constraints and features of neuromorphic hardware. As such, my work not only represents a novel perspective for training SNNs on neuromorphic hardware but also represents a promising energy-efficient alternative to DNNs trained with BP for high-performance learning in edge computing.

In conclusion, this thesis addresses the inherent lack of differentiable spikes and the fundamental challenges related to local and online gradient computations, presenting novel perspectives for neuromorphic-compatible gradient descent. While a slight performance gap persists compared to backpropagation, the proposed SFDFA algorithm thesis showed promising capabilities in training SNNs with higher rates of convergence. It also outperforms the widely-used DFA algorithm. Importantly, the proposed approach estimate gradients in a local and online manner, aligning with the constraints and features of neuromorphic hardware. Consequently, this work offers a unique perspective for SNN training on neuromorphic hardware.

6.1 Future Work

I now propose several directions of future research to further extend and improve the proposed method.

6.1.1 Improving Learning with Spatio-Temporal Data

In Section 5.5, I showed that both the DFA and SDFDA algorithms are capable of training SNNs on static image datasets with performance close to BP. However, both fail to reach the performance of BP on the SHD dataset which suggests a potential limitation in training spatio-temporal data with direct feedback connections. I hypothesize that this limitation arises due to the nature of the error signals. In my experiments, a single error signal was sent to neurons at the end of each simulation. However, previous works with DFA have shown error signals that are frequently emitted during the inference can improve learning with long-term dependencies (Bellec et al. 2020; Rostami et al. 2022). Therefore, future work could implement more frequent error signals to improve the performance of the SFDFA algorithm with spatio-temporal data.

6.1.2 Spiking Forward Direct Feedback Alignment with Spike Time-Dependent Plasticity

The SFDFA algorithm computes local gradients by accumulating the derivatives of post-synaptic spikes. However, several neuromorphic hardware already integrate biologically inspired learning mechanisms such as Spike Time-Dependant Plasticity (STDP) or Reward-Modulated STDP (RSTDP) (Schemmel et al. 2010; Pehle et al. 2022; Davies et al. 2018) — see Sections 2.3.8 and 2.3.9 for details. In addition to being biologically plausible, STDP and R-STDP have the advantage of being computationally efficient compared to the derivatives of post-synaptic spikes. Therefore, future work could explore the adaptation of the SFDFA algorithm with R-STDP. More particularly, local gradients could be replaced by STDP traces computed during the inference, which would then be modulated by the project error signals. This would reduce the computational complexity of local gradients of spikes while increasing the compatibility of SFDFA with neuromorphic hardware implementing biologically plausible learning rules.

6.1.3 Implementation on Neuromorphic Hardware

While the proposed SFDFA algorithm has been designed according to the constraints and features of neuromorphic hardware, all experiments of this thesis have been performed on von Neumann computers and accelerated with GPUs. Therefore, future work could focus on implementing the proposed SFDFA on neuromorphic hardware and benchmarking with real-world applications. Finally, because the derivative estimates between output and hidden neurons in SFDFA rely on the use of graded spikes, neuromorphic platforms already proposing graded events such as Loihi 2 (Frady et al. 2022) or SpiNNaker (Furber et al. 2014; Höppner et al. 2022) should be privileged.

If implemented on neuromorphic hardware, the SFDFA algorithm is anticipated to exhibit higher energy consumption than DFA. This is attributed to the computational demands associated with calculating directional derivatives and incorporating an additional feedback learning rule. However, the potential performance enhancement offered by SFDFA in comparison to DFA introduces a noteworthy trade-off between accuracy and energy consumption that could be the subject of future studies.

Additionally, there is an opportunity to enhance control over the firing rate of neurons with feedback learning. Investigating strategies to reduce the spike count of the neuron population is crucial, as the firing rate significantly influences energy consumption in event-based neuromorphic hardware. Discovering more effective methods to regulate the spike count holds the promise of advancing energy efficiency in these systems.

Bibliography

Abadi, M. et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Akopyan, F. et al. (2015). Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10), pp. 1537–1557.

Akrout, M., Wilson, C., Humphreys, P. C., Lillicrap, T. P. and Tweed, D. B. (2019). Deep learning without weight transport. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox and R. Garnett, eds., *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 974–982.

Alzubaidi, L. et al. (2021). Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, 8(1).

Auge, D., Hille, J., Mueller, E. and Knoll, A. (2021). A survey of encoding techniques for signal processing in spiking neural networks. *Neural Processing Letters*, 53(6), pp. 4693–4710.

Azevedo, F. A. C. et al. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *J Comp Neurol*, 513(5), pp. 532–541.

Balasubramanian, V. (2021). Brain power. *Proceedings of the National Academy of Sciences*, 118(32), p. e2107022118, `https://www.pnas.org/doi/pdf/10.1073/pnas.2107022118`.

Bartunov, S. et al. (2018). Assessing the scalability of biologically-motivated deep learning algorithms and architectures. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., NIPS'18, p. 9390–9400.

Basodi, S., Ji, C., Zhang, H. and Pan, Y. (2020). Gradient amplification: An efficient way to train deep neural networks. *Big Data Mining and Analytics*, 3(3), pp. 196–207.

Bauer, F. L. (1974). Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1), pp. 87–96.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A. and Siskind, J. M. (2017). Automatic differentiation in machine learning: A survey. *J Mach Learn Res*, 18(1), p. 5595–5637.

Baydin, A. G., Pearlmutter, B. A., Syme, D., Wood, F. and Torr, P. (2022). Gradients without backpropagation. `2202.08587`.

Belilovsky, E., Eickenberg, M. and Oyallon, E. (2019). Greedy layerwise learning can scale to imagenet. In *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of Machine Learning Research, pp. 583–593.

Bellec, G. et al. (2020). A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications*, 11(1).

Bhattacharya, G. (2021). From dnns to gans: Review of efficient hardware architectures for deep learning. `2107.00092`.

Bi, G.-q. and Poo, M.-m. (1998). Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of neuroscience*, 18(24), pp. 10464–10472.

Bishop, C. (1995). *Neural networks for pattern recognition*. Oxford University Press, USA.

Bohté, S. M., Kok, J. N. and Poutré, H. L. (2000). Spikeprop: backpropagation for networks of spiking neurons. In *The European Symposium on Artificial Neural Networks*.

Booij, O. and tat Nguyen, H. (2005). A gradient descent rule for spiking neurons emitting multiple spikes. *Information Processing Letters*, 95(6), pp. 552–558, applications of Spiking Neural Networks.

Bottou, L., Curtis, F. E. and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *SIAM Review*, 60(2), pp. 223–311.

Bouvier, M. et al. (2019). Spiking neural networks hardware implementations and challenges: A survey. *J Emerg Technol Comput Syst*, 15(2).

Brette, R. (2015). Philosophy of the spike: Rate-based vs. spike-based theories of the brain. *Front Syst Neurosci*, 9, p. 151.

Brette, R. and Goodman, D. F. M. (2012). Simulating spiking neural networks on gpu. *Network: Computation in Neural Systems*, 23(4), pp. 167–182.

Brette, R. et al. (2007). Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, 23(3), pp. 349–398.

Brown, T. et al. (2020). Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan and H. Lin, eds., *Advances in Neural Information Processing Systems*, vol. 33, Curran Associates, Inc., pp. 1877–1901.

Bubeck, S. (2015). Convex optimization: Algorithms and complexity. *Found Trends Mach Learn*, 8(3–4), p. 231–357.

Buber, E. and Diri, B. (2018). Performance analysis and cpu vs gpu comparison for deep learning. pp. 1–6.

Butcher, J. C. (2016). *Numerical Differential Equation Methods*, John Wiley and Sons, Ltd, chap. 2. pp. 55–142.

Castro, E., Cardoso, J. S. and Pereira, J. C. (2018). Elastic deformations for data augmentation in breast cancer mass detection. In *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*, pp. 230–234.

Chase, S. M. and Young, E. D. (2007). First-spike latency information in single neurons increases when referenced to population onset. *Proceedings of the National Academy of Sciences*, 104(12), pp. 5175–5180.

Chee, J. and Toulis, P. (2018). Convergence diagnostics for stochastic gradient descent with constant learning rate. In A. Storkey and F. Perez-Cruz, eds., *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, *Proceedings of Machine Learning Research*, vol. 84, Proceedings of Machine Learning Research, pp. 1476–1485.

Cho, K. et al. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. `1406.1078`.

Christensen, D. V. et al. (2022). 2022 roadmap on neuromorphic computing and engineering. *Neuromorphic Computing and Engineering*, 2(2), p. 022501.

Cohen, G., Afshar, S., Tapson, J. and van Schaik, A. (2017). Emnist: Extending mnist to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 2921–2926.

Comsa, I. M. et al. (2020). Temporal coding in spiking neural networks with alpha synaptic function: Learning with backpropagation.

Corless, R. M., Gonnet, G. H., Hare, D. E. G., Jeffrey, D. J. and Knuth, D. E. (1996). On the LambertW function. *Advances in Computational Mathematics*, 5(1), pp. 329–359.

Crafton, B., Parihar, A., Gebhardt, E. and Raychowdhury, A. (2019). Direct feedback alignment with sparse connections for local learning. *Frontiers in neuroscience*, 13, p. 525.

Cramer, B., Stradmann, Y., Schemmel, J. and Zenke, F. (2022a). The heidelberg spiking data sets for the systematic evaluation of spiking neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 33(7), pp. 2744–2757.

Cramer, B. et al. (2022b). Surrogate gradients for analog neuromorphic computing. *Proceedings of the National Academy of Sciences*, 119(4), p. e2109194119.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4), pp. 303–314.

Daghero, F., Pagliari, D. J. and Poncino, M. (2021). Chapter eight - energy-efficient deep learning inference on edge devices. In S. Kim and G. C. Deka, eds., *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*, *Advances in Computers*, vol. 122, Elsevier, pp. 247–301.

Davies, M. et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1), pp. 82–99.

Defazio, A. (2020). Momentum via primal averaging: Theoretical insights and learning rate schedules for non-convex optimization. `2010.00406`.

Deng, M. and Li, C. (2021). Stdp and competition learning in spiking neural networks and its application to image classification. In *2021 8th International Conference on Information, Cybernetics, and Computational Social Systems (ICCSS)*, pp. 385–389.

Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:181004805*.

Diamond, A., Schmuker, M. and Nowotny, T. (2019). An unsupervised neuromorphic clustering algorithm. *Biological Cybernetics*, 113(4), pp. 423–437.

Diehl, P. and Cook, M. (2015). Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, 9.

Dong, Y., Zhao, D., Li, Y. and Zeng, Y. (2023). An unsupervised stdp-based spiking neural network inspired by biologically plausible learning rules and connections. *Neural Networks*, 165, pp. 799–808.

Faghri, F., Duvenaud, D., Fleet, D. J. and Ba, J. (2020). A study of gradient variance in deep learning. `2007.04532`.

Fang, H., Shrestha, A., Zhao, Z., Li, Y. and Qiu, Q. (2019). An event-driven neuromorphic system with biologically plausible temporal dynamics. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8.

Frady, E. P. et al. (2022). Efficient neuromorphic signal processing with resonator neurons. *J Signal Process Syst*, 94(10), p. 917–927.

Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks*, 1(2), pp. 119–130.

Furber, S. B., Galluppi, F., Temple, S. and Plana, L. A. (2014). The spinnaker project. *Proceedings of the IEEE*, 102(5), pp. 652–665.

Furukawa, S. and Middlebrooks, J. C. (2002). Cortical representation of auditory space: information-bearing features of spike patterns. *Journal of neurophysiology*, 87(4), pp. 1749–1762.

Gautrais, J. and Thorpe, S. (1998). Rate coding versus temporal order coding: a theoretical approach. *Biosystems*, 48(1), pp. 57–65.

Gerstner, W. and Kistler, W. M. (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press.

Gerstner, W., Kistler, W. M., Naud, R. and Paninski, L. (2014). *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press.

Ghosh-Dastidar, S. and Adeli, H. (2009). A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection. *Neural Networks*, 22(10), pp. 1419–1431.

Glorot, X., Bordes, A. and Bengio, Y. (2011). Deep sparse rectifier neural networks. In G. Gordon, D. Dunson and M. Dudík, eds., *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, *Proceedings of Machine Learning Research*, vol. 15, Fort Lauderdale, FL, USA: PMLR, pp. 315–323.

Göltz, J. et al. (2021). Fast and energy-efficient neuromorphic deep learning with first-spike times. 3(9), pp. 823–835.

Gower, R. M. et al. (2019). Sgd: General analysis and improved rates. `1901.09401`.

Graves, A., Jaitly, N. and Mohamed, A.-r. (2013). Hybrid speech recognition with deep bidirectional lstm. In *2013 IEEE workshop on automatic speech recognition and understanding*, IEEE, pp. 273–278.

Grossberg, S. (1987). Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science*, 11(1), pp. 23–63.

Grüning, A. and Bohté, S. M. (2014). Spiking neural networks: Principles and challenges. In *The European Symposium on Artificial Neural Networks*.

Guo, W., Fouda, M. E., Eltawil, A. M. and Salama, K. N. (2021). Neural coding in spiking neural networks: A comparative study for robust neuromorphic systems. *Frontiers in Neuroscience*, 15.

Gyawali, D. (2023). Comparative analysis of cpu and gpu profiling for deep learning models. `2309.02521`.

Han, D. and jun Yoo, H. (2019). Efficient convolutional neural network training with direct feedback alignment. `1901.01986`.

Harris, C. R. et al. (2020). Array programming with NumPy. *Nature*, 585(7825), pp. 357–362.

Hasler, J. (2020). Large-scale field-programmable analog arrays. *Proceedings of the IEEE*, 108(8), pp. 1283–1302.

He, K., Zhang, X., Ren, S. and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, Los Alamitos, CA, USA: IEEE Computer Society, pp. 1026–1034.

He, T. and Droppo, J. (2016). Exploiting lstm structure in deep neural networks for speech recognition. *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5445–5449.

Hebb, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. New York: Wiley.

Hecht-Nielsen (1989). Theory of the backpropagation neural network. In *International 1989 Joint Conference on Neural Networks*, pp. 593–605 vol.1.

Heeger, D. et al. (2000). Poisson model of spike generation. *Handout, University of Standford*, 5(1-13), p. 76.

Hinton, G. (2022). The forward-forward algorithm: Some preliminary investigations. `2212.13345`.

Hjelm, R. D. et al. (2019). Learning deep representations by mutual information estimation and maximization. In *International Conference on Learning Representations*.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), pp. 1735–1780.

Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4), pp. 500–544.

Hong, C. et al. (2020). Training spiking neural networks for cognitive tasks: A versatile framework compatible with various temporal codes. *IEEE Transactions on Neural Networks and Learning Systems*, 31(4), pp. 1285–1296.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2), pp. 251–257.

Hornik, K., Stinchcombe, M. and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), pp. 359–366.

Horowitz, M. (2014). 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)*, pp. 10–14.

Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1), pp. 215–243.

Huo, Z., Gu, B., Yang, Q. and Huang, H. (2018). Decoupled parallel backpropagation with convergence guarantee. `1804.10574`.

Höppner, S. et al. (2022). The spinnaker 2 processing element architecture for hybrid digital neuromorphic computing. `2103.08392`.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, JMLR.org, ICML'15, p. 448–456.

Izhikevich, E. (2003). Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6), pp. 1569–1572.

Izhikevich, E. M. (2004). Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, 15(5), pp. 1063–1070.

Jabri, M. and Flower, B. (1992). Weight perturbation: an optimal architecture and learning technique for analog VLSI feedforward and recurrent multilayer networks. *IEEE Transactions on Neural Networks*, 3(1), pp. 154–157.

Jaderberg, M. et al. (2017). Decoupled neural interfaces using synthetic gradients. In D. Precup and Y. W. Teh, eds., *Proceedings of the 34th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, vol. 70, Proceedings of Machine Learning Research, pp. 1627–1635.

Jin, Y., Zhang, W. and Li, P. (2018). Hybrid macro/micro level backpropagation for training deep spiking neural networks. Red Hook, NY, USA: Curran Associates Inc.

Jouppi, N. P. et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, New York, NY, USA: Association for Computing Machinery, ISCA '17, p. 1–12.

Juarez-Lora, A., Ponce-Ponce, V. H., Sossa, H. and Rubio-Espino, E. (2022). R-STDP spiking neural network architecture for motion control on a changing friction joint robotic arm. *Frontiers in Neurorobotics*, 16.

Kandel, E. R., Schwartz, J. H. and Jessell, T. M., eds. (1991). *Principles of Neural Science*. New York: Elsevier, 3rd edn.

Karpathy, A., Johnson, J. and Fei-Fei, L. (2015). Visualizing and understanding recurrent networks. `1506.02078`.

Kheradpisheh, S. R. and Masquelier, T. (2020). Temporal backpropagation for spiking neural networks with one spike per neuron. *International Journal of Neural Systems*, 30.

Kheradpisheh, S. R., Mirsadeghi, M. and Masquelier, T. (2021). Bs4nn: Binarized spiking neural networks with temporal coding and learning. *Neural Processing Letters*, 54(2), p. 1255–1273.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization. `1412.6980`.

Koslow, S. H. and Subramaniam, S. (2005). *Databasing the brain: From Data to Knowledge (neuroinformatics)*. Wiley-Liss.

Krizhevsky, A., Hinton, G. et al. (2009). Learning multiple layers of features from tiny images.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012a). Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, Red Hook, NY, USA: Curran Associates Inc., NIPS'12, p. 1097–1105.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012b). Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou and K. Weinberger, eds., *Advances in Neural Information Processing Systems*, vol. 25, Curran Associates, Inc.

Kwon, H. et al. (2020). Understanding reuse, performance, and hardware cost of dnn dataflows: A data-centric approach using maestro. `1805.02566`.

Launay, J., Poli, I. and Krzakala, F. (2019). Principled training of neural networks with direct feedback alignment. `1906.04554`.

Launay, J., Poli, I., Boniface, F. and Krzakala, F. (2020). Direct feedback alignment scales to modern deep learning tasks and architectures. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., NIPS'20.

Le, Y. and Yang, X. (2015). Tiny imagenet visual recognition challenge. *CS 231N*, 7(7), p. 3.

Le Cun, Y., Galland, C. and Hinton, G. E. (1988). Gemini: Gradient estimation through matrix inversion after noise injection. In D. Touretzky, ed., *Advances in Neural Information Processing Systems*, vol. 1, Morgan-Kaufmann.

LeCun, Y., Cortes, C. and Burges, C. (2010). Mnist handwritten digit database. *ATT Labs [Online] Available: http://yannlecuncom/exdb/mnist*, 2.

LeCun, Y. et al. (1989). Handwritten digit recognition with a back-propagation network. In D. Touretzky, ed., *Advances in Neural Information Processing Systems*, vol. 2, Morgan-Kaufmann.

Lee, C., Panda, P., Srinivasan, G. and Roy, K. (2018). Training deep spiking convolutional neural networks with STDP-based unsupervised pre-training followed by supervised fine-tuning. *Frontiers in Neuroscience*, 12.

Lee, J., Zhang, R., Zhang, W., Liu, Y. and Li, P. (2020). Spike-train level direct feedback alignment: Sidestepping backpropagation for on-chip training of spiking neural nets. *Frontiers in Neuroscience*, 14.

Lee, J. H., Delbruck, T. and Pfeiffer, M. (2016). Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10, p. 508.

Lee, J. H., Haghighatshoar, S. and Karbasi, A. (2023). Exact gradient computation for spiking neural networks via forward propagation. In *International Conference on Artificial Intelligence and Statistics*, PMLR, pp. 1812–1831.

Levy, W. and Steward, O. (1983). Temporal contiguity requirements for long-term associative potentiation/depression in the hippocampus. *Neuroscience*, 8(4), pp. 791–797.

Li, Y. et al. (2021). Differentiable spike: Rethinking gradient-descent for training spiking neural networks. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang and J. W. Vaughan, eds., *Advances in Neural Information Processing Systems*, vol. 34, Curran Associates, Inc., pp. 23426–23439.

Lillicrap, T. P., Cownden, D., Tweed, D. B. and Akerman, C. J. (2016). Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7.

Lillicrap, T. P., Santoro, A., Marris, L., Akerman, C. J. and Hinton, G. (2020). Backpropagation and the brain. *Nature Reviews Neuroscience*, 21(6), pp. 335–346.

Lipton, Z. C., Berkowitz, J. and Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. `1506.00019`.

Löwe, S., O' Connor, P. and Veeling, B. (2019). Putting an end to end-to-end: Gradient-isolated learning of representations. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox and R. Garnett, eds., *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc.

Maass, W. (1997). Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9), pp. 1659 – 1671.

Magee, J. C. and Johnston, D. (1997). A synaptically controlled, associative signal for hebbian plasticity in hippocampal neurons. *Science*, 275(5297), pp. 209–213.

Margossian, C. C. (2019). A review of automatic differentiation and its efficient implementation. *WIREs Data Mining and Knowledge Discovery*, 9(4).

Margossian, C. C. and Betancourt, M. (2022). Efficient automatic differentiation of implicit functions. `2112.14217`.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), pp. 115–133.

McKennoch, S., Liu, D. and Bushnell, L. (2006). Fast modifications of the spikeprop algorithm. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pp. 3970–3977.

Minsky, M. and Papert, S. A. (2017). *Perceptrons: An Introduction to Computational Geometry*. The MIT Press.

Mnih, V. et al. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:13125602*.

Mnih, V. et al. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, PMLR, pp. 1928–1937.

Mo, L. and Tao, Z. (2022). EvtSNN: Event-driven SNN simulator optimized by population and pre-filtering. *Frontiers in Neuroscience*, 16.

Mostafa, H. (2016). Supervised learning based on temporal coding in spiking neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, PP.

Mostafa, H., Ramesh, V. and Cauwenberghs, G. (2018). Deep supervised learning using local errors.

Moulines, E. and Bach, F. (2011). Non-asymptotic analysis of stochastic approximation algorithms for machine learning. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira and K. Weinberger, eds., *Advances in Neural Information Processing Systems*, vol. 24, Curran Associates, Inc.

Mozafari, M., Ganjtabesh, M., Nowzari-Dalini, A., Thorpe, S. J. and Masquelier, T. (2019). Bio-inspired digit recognition using reward-modulated spike-timing-dependent plasticity in deep convolutional networks. *Pattern Recognition*, 94, pp. 87–95.

Murata, N. (1999). *A Statistical Study of On-Line Learning*, USA: Cambridge University Press. p. 63–92.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. Madison, WI, USA: Omnipress, ICML'10, p. 807–814.

Needell, D., Srebro, N. and Ward, R. (2016). Stochastic gradient descent, weighted sampling, and the randomized Kaczmarz algorithm. *Mathematical Programming*, 155(1), pp. 549–573.

Neftci, E. O., Augustine, C., Paul, S. and Detorakis, G. (2017a). Event-driven random back-propagation: Enabling neuromorphic deep learning machines. *Frontiers in Neuroscience*, 11, p. 324.

Neftci, E. O., Augustine, C., Paul, S. and Detorakis, G. (2017b). Event-driven random back-propagation: Enabling neuromorphic deep learning machines. *Frontiers in Neuroscience*, 11.

Neil, D. and Liu, S.-C. (2014). Minitaur, an event-driven fpga-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12), pp. 2621–2628.

Nobukawa, S., Nishimura, H. and Yamanishi, T. (2019). Pattern classification by spiking neural networks combining self-organized and reward-related spike-timing-dependent plasticity. *Journal of Artificial Intelligence and Soft Computing Research*, 9(4), pp. 283–291.

Nøkland, A. (2016). Direct feedback alignment provides learning in deep neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon and R. Garnett, eds., *Advances in Neural Information Processing Systems*, vol. 29, Curran Associates, Inc.

Nøkland, A. and Eidnes, L. H. (2019). Training neural networks with local error signals. In K. Chaudhuri and R. Salakhutdinov, eds., *Proceedings of the 36th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, vol. 97, Proceedings of Machine Learning Research, pp. 4839–4850.

Okuta, R., Unno, Y., Nishino, D., Hido, S. and Loomis, C. (2017). Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*.

O'Shea, K. and Nash, R. (2015). An introduction to convolutional neural networks. `1511.08458`.

Park, S., Lee, D. and Yoon, S. (2021). Noise-robust deep spiking neural networks with temporal information. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, IEEE Press, p. 373–378.

Paszke, A. et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., pp. 8024–8035.

Pehle, C. et al. (2022). The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity. *Frontiers in Neuroscience*, 16.

Pham, Q. T. et al. (2021). A review of snn implementation on fpga. In *2021 International Conference on Multimedia Analysis and Pattern Recognition (MAPR)*, pp. 1–6.

Phillips, D. (1998). Factors shaping the response latencies of neurons in the cat's auditory cortex. *Behavioural Brain Research*, 93(1), pp. 33–41.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1), pp. 145–151.

Ramón y Cajal, S. (????). *Histologie du système nerveux de l'homme et des vertébrés*, vol. v. 1. Paris, Maloine, 1909-11, https://www.biodiversitylibrary.org/bibliography/48637.

Rao, A., Plank, P., Wild, A. and Maass, W. (2022). A long short-term memory for AI applications in spike-based neuromorphic hardware. *Nature Machine Intelligence*, 4(5), pp. 467–479.

Refinetti, M., D'Ascoli, S., Ohana, R. and Goldt, S. (2021). Align, then memorise: the dynamics of learning with feedback alignment. In *International Conference on Machine Learning*, pp. 8925–8935.

Reich, D. S., Mechler, F. and Victor, J. D. (2001). Temporal coding of contrast in primary visual cortex: when, what, and why. *Journal of neurophysiology*, 85(3), pp. 1039–1050.

Ren, M., Kornblith, S., Liao, R. and Hinton, G. (2023). Scaling forward gradient with local losses. In *The Eleventh International Conference on Learning Representations*.

Robbins, H. and Monro, S. (1951). A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3), pp. 400 – 407.

Rocke, P., McGinley, B., Maher, J., Morgan, F. and Harkin, J. (2008). Investigating the suitability of fpaas for evolved hardware spiking neural networks. In G. S. Hornby, L. Sekanina and P. C. Haddow, eds., *Evolvable Systems: From Biology to Hardware*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 118–129.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6, pp. 386–408.

Rostami, A., Vogginger, B., Yan, Y. and Mayr, C. G. (2022). E-prop on SpiNNaker 2: Exploring online learning in spiking RNNs on neuromorphic hardware. *Frontiers in Neuroscience*, 16.

Ruder, S. (2017). An overview of gradient descent optimization algorithms. `1609.04747`.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323, pp. 533–536.

Schemmel, J. et al. (2010). A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1947–1950.

Schmidhuber, J. (1992). Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2), pp. 234–242.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61, pp. 85–117.

Schrauwen, B. and Van Campenhout, J. (2004). Extending spikeprop. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, vol. 1, pp. 471–475.

Schuman, C. D. et al. (2022). Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science*, 2(1), pp. 10–19.

Shawahna, A., Sait, S. M. and El-Maleh, A. (2019). Fpga-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7, pp. 7823–7859.

Shrestha, A., Fang, H., Wu, Q. and Qiu, Q. (2019). Approximating back-propagation for a biologically plausible local learning rule in spiking neural networks. In *Proceedings of the International Conference on Neuromorphic Systems*, New York, NY, USA: Association for Computing Machinery, ICONS '19.

Shrestha, A., Fang, H., Rider, D. P., Mei, Z. and Qiu, Q. (2021). In-hardware learning of multilayer spiking neural networks on a neuromorphic processor. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 367–372.

Shrestha, S. B. and Orchard, G. (2018). Slayer: Spike layer error reassignment in time. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., NIPS'18, p. 1419–1428.

Silver, D., Goyal, A., Danihelka, I., Hessel, M. and van Hasselt, H. (2021). Learning by directional gradient descent. In *International Conference on Learning Representations*.

Sivilotti, M. A. (1991). *Wiring considerations in analog VLSI systems, with application to field-programmable networks*. California Institute of Technology.

Smith, J. E. (2020). A neuromorphic paradigm for online unsupervised clustering. `2005.04170`.

Smith, S. W. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing*. USA: California Technical Publishing.

Steinkraus, D., Buck, I. and Simard, P. (2005). Using gpus for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pp. 1115–1120 Vol. 2.

Sutskever, I., Vinyals, O. and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, Cambridge, MA, USA: MIT Press, NIPS'14, p. 3104–3112.

Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Erlbaum.

Sze, V., Chen, Y.-H., Emer, J., Suleiman, A. and Zhang, Z. (2017). Hardware for machine learning: Challenges and opportunities. In *2017 IEEE Custom Integrated Circuits Conference (CICC)*, IEEE, pp. 1–8.

Szegedy, C., Toshev, A. and Erhan, D. (2013). Deep neural networks for object detection. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani and K. Q. Weinberger, eds., *Advances in Neural Information Processing Systems*, vol. 26, Curran Associates, Inc.

Takase, H. et al. (2009a). Obstacle to training spikeprop networks — cause of surges in training process —. pp. 3062–3066.

Takase, H. et al. (2009b). Obstacle to training spikeprop networks — cause of surges in training process —. In *2009 International Joint Conference on Neural Networks*, pp. 3062–3066.

Talib, M. A., Majzoub, S., Nasir, Q. and Jamal, D. (2020). A systematic literature review on hardware implementation of artificial intelligence algorithms. *The Journal of Supercomputing*, 77(2), pp. 1897–1938.

Thorpe, S., Delorme, A. and Van Rullen, R. (2001a). Spike-based strategies for rapid processing. *Neural Networks*, 14(6), pp. 715–725.

Thorpe, S., Delorme, A. and Van Rullen, R. (2001b). Spike-based strategies for rapid processing. *Neural Networks*, 14(6), pp. 715–725.

Thorpe, S. J. and Imbert, M. (1989). 1 biological constraints on connectionist modelling.

van Rossum, M. C. W., Bi, G. Q. and Turrigiano, G. G. (2000). Stable hebbian learning from spike timing-dependent plasticity. *Journal of Neuroscience*, 20(23), pp. 8812–8821.

Vaswani, A. et al. (2017). Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan and R. Garnett, eds., *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc.

Wang, Y. E., Wei, G.-Y. and Brooks, D. (2019). Benchmarking tpu, gpu, and cpu platforms for deep learning. `1907.10701`.

Webster, M. B., Choi, J. and Ahn, C. (2021). Learning the connections in direct feedback alignment.

Wen, Y., Vicol, P., Ba, J., Tran, D. and Grosse, R. (2018). Flipout: Efficient pseudo-independent weight perturbations on mini-batches. `1803.04386`.

Wengert, R. E. (1964). A simple automatic derivative evaluation program. *Commun ACM*, 7(8), p. 463–464.

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proc IEEE*, 78, pp. 1550–1560.

Wu, Y., Deng, L., Li, G., Zhu, J. and Shi, L. (2018). Spatio-temporal backpropagation for training high-performance spiking neural networks. *Frontiers in Neuroscience*, 12.

Wunderlich, T. C. and Pehle, C. (2021). Event-based backpropagation can compute exact gradients for spiking neural networks. *Scientific Reports*, 11(1).

Xiao, H., Rasul, K. and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

Yin, R. et al. (2023). Workload-balanced pruning for sparse spiking neural networks.

Yu, Y., Si, X., Hu, C. and Zhang, J. (2019). A review of recurrent neural networks: Lstm cells and network architectures. *Neural Computation*, 31(7), pp. 1235–1270.

Zhang, M. et al. (2022). Rectified linear postsynaptic potential function for backpropagation in deep spiking neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 33(5), pp. 1947–1958.

Zhang, S.-Q. and Zhou, Z.-H. (2022). Theoretically provable spiking neural networks. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho and A. Oh, eds., *Advances in Neural Information Processing Systems*, vol. 35, Curran Associates, Inc., pp. 19345–19356.

Zhang, W. and Li, P. (2019). Spike-train level backpropagation for training deep recurrent spiking neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox and R. Garnett, eds., *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc.

Zhang, W. and Li, P. (2020). Temporal spike sequence learning via backpropagation for deep spiking neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan and H. Lin, eds., *Advances in Neural Information Processing Systems*, vol. 33, Curran Associates, Inc., pp. 12022–12033.

Zhang, W., Hasegawa, A., Itoh, K. and Ichioka, Y. (1991). Image processing of human corneal endothelium based on a learning network. *Appl Opt*, 30(29), pp. 4211–4217.

Zhang, W. et al. (1994). Computerized detection of clustered microcalcifications in digital mammograms using a shift-invariant artificial neural network. *Medical Physics*, 21(4), pp. 517–524.

Zheng, H., Wu, Y., Deng, L., Hu, Y. and Li, G. (2020). Going deeper with directly-trained larger spiking neural networks. In *AAAI Conference on Artificial Intelligence*.

Zhou, S., Li, X., Chen, Y., Chandrasekaran, S. T. and Sanyal, A. (2021). Temporal-coded deep spiking neural network with easy training and robust performance. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(12), pp. 11143–11151.