# A Burst-Dependent Algorithm for Neuromorphic On-Chip Learning of Spiking Neural Networks

by

Michael Stuck

Thesis submitted to the
University of Ottawa
In partial fulfillment of the requirements for the
Master of Science in Physics

Department of Physics
University of Ottawa
June 2024

**Abstract**

The field of neuromorphic engineering addresses the high energy demands of neural networks through brain-inspired hardware for efficient neural network computing. For on-chip learning with spiking neural networks, neuromorphic hardware requires a local learning algorithm that is able to solve complex tasks. Approaches based on burst-dependent plasticity have been proposed to address this requirement, but their ability to learn complex tasks has not been proven. Specifically, previous burst-dependent learning was demonstrated on a spiking version of the exclusive or (XOR) problem using a network of thousands of neurons. Here, we extend burst-dependent learning, termed 'Burstprop,' to address more complex tasks with hundreds of neurons. We evaluate Burstprop on a rate-encoded spiking version of the MNIST handwritten digit dataset, achieving low test classification errors, comparable to those obtained using backpropagation through time on the same architecture. Going further, we develop another burst-dependent algorithm based on the communication of two types of error-encoding events for the communication of positive and negative errors. We find that this new algorithm performs better on the image classification benchmark. We also tested our algorithms under various types of feedback connectivity, establishing that the capabilities of fixed random feedback connectivity are preserved in spiking neural networks. Lastly, we tested the robustness of the algorithm to weight discretization. Together, these results suggest that spiking Burstprop can scale to more complex learning tasks while maintaining efficiency, potentially providing a viable method for learning with neuromorphic hardware.

**Acknowledgements**

First, I would like to thank my supervisor, Richard Naud, who has been an amazing guide and mentor throughout my degree. Your support in my research and the many insightful conversations we've had have kept me headed in the right direction. I am grateful for your patience and your consistent feedback has been of great value to me in this process.

I would also like to thank my parents, Kathy and Eric, for their consistent love and support throughout my schooling.

Major gratitude to my friends and roommates, in Ottawa and elsewhere, for the large surplus of good times we've had and for giving me a sense of belonging during this time. This may have been the most two years of my life thus far thanks to you people.

I would like to thank my lab mates, who have served great as role models throughout my degree.

Lastly, I would like to thank everyone who has supported me in various ways large and tiny throughout this journey.

# Contents

# List of Figures

# List of Abbreviations

| | |
|---|---|
| **ANN** | Artificial Neural Network |
| **BPTT** | Backpropagation Through Time |
| **CPU** | Central Processing Unit |
| **GPU** | Graphics Processing Unit |
| **LIF** | Leaky Integrate-and-Fire |
| **MNIST** | Modified National Institute of Standards and Technology |
| **SGD** | Stochastic Gradient Descent |
| **SNN** | Spiking Neural Network |
| **STDP** | Spike-Time-Dependent Plasticity |
| **XOR** | Exclusive OR |

# Contributions

- **Stuck, M., & Naud, R. (2023). Burstprop for Learning in Spiking Neuromorphic Hardware**
  - Proceedings of the 2023 International Conference on Neuromorphic Systems (peer-reviewed), reporting initial results
  - DOI: https://doi.org/10.1145/3589737.360596
  - Contribution: conceived work, performed simulations, analyzed results, created figures, wrote the manuscript
- **Stuck, M., & Naud, R. (2024). A Burst-Dependent Algorithm for Neuromorphic On-Chip Learning of Spiking Neural Networks**
  - Submitted to IOPScience Neuromorphic Computing and Engineering
  - Preprint on bioRxiv: https://doi.org/10.1101/2024.07.19.604308
  - Contribution: conceived work, performed simulations, analyzed results, created figures, wrote the manuscript

# Chapter 1

# Introduction

## 1.1 Motivation

Machine learning with Artificial Neural Networks (ANNs) can learn and perform a wide range of useful tasks [1]. However, neural networks require significant energy, which is a major limitation for energy-constrained applications [2]. This energy inefficiency has led to interest in alternative computational methods.

Neuromorphic computing has emerged as a promising approach to reduce the energy costs associated with neural networks [2, 3, 4]. These systems, inspired by the brain's energy-efficient computation, aim to achieve substantial improvements in energy efficiency. Neuromorphic hardware that implements Spiking Neural Networks (SNNs), which are neural networks that use spiking neurons to more closely mimic biological neural networks, offers a viable solution to the energy inefficiency problem [5, 4]. However, to fully realize the potential of neuromorphic computing for energy-efficient training and inference, compatible learning algorithms are needed [2]. This thesis aims to address this gap in suitable learning algorithms for training SNNs in neuromorphic hardware.

## 1.2 Review of Concepts

### 1.2.1 Neural Networks

Artificial Neural Networks (ANNs) are powerful machine learning models designed to learn and perform specific tasks when trained on sufficient data. The term "Neural Network" is derived from their resemblance to the human brain's network of interconnected neurons. ANNs consist of nodes (neurons) with nonlinear activations and weights that connect these nodes with varying strengths. These networks are typically organized in hierarchical layers, and when multiple layers are present, they form deep neural networks [6].

Neural network learning methods, and machine learning in general, can be subcategorized into supervised learning, unsupervised learning, and reinforcement learning. Supervised learning aims to learn input-output relations from labelled data to predict outputs from inputs [7]. Unsupervised learning aims to identify patterns in unlabeled data for various purposes [8], while reinforcement learning aims to make decisions through trial and error by interacting with an environment and receiving feedback in the form of rewards [9]. This work focuses solely on supervised learning.

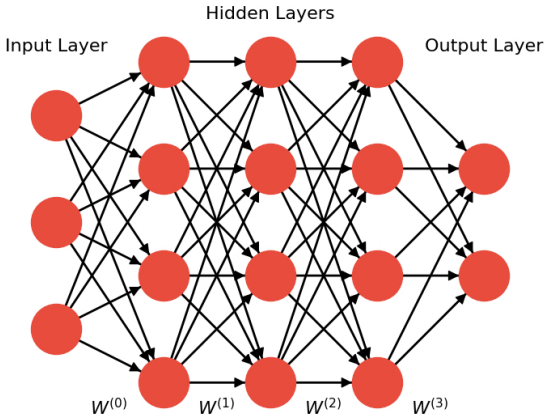


Figure 1.1: **Schematic of a feedforward neural network.** Feedforward neural network architecture with an input layer, 3 hidden layers, and an output layer.

### Feedforward Neural Networks

A feedforward neural network, as depicted in Figure 1.1 consists of an input layer, one or more hidden layers, and an output layer. Each layer contains neurons that are connected to the neurons in the previous and following layers by weighted edges [1].

Consider a feedforward neural network with $L$ layers. Let $x$ be the input vector, $W^{(l)}$ the weight matrix, and $b^{(l)}$ the bias vector for layer $(l)$. The output of the $(l)$-th layer, $a^{(l)}$, is computed as:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = \sigma(z^{(l)})$$

where $z^{(l)}$ is the input to layer $(l)$, $a^{(l-1)}$ is the activation from the previous layer, and $\sigma$ is the activation function. Common activation functions include the sigmoid function, ReLU (Rectified Linear Unit), and tanh [10].

### The Supervised Learning Problem

In supervised learning, the goal is to learn a mapping from inputs $x$ to outputs $y$ using a labelled dataset $\{(x_i, \hat{y}_i)\}_{i=1}^{N}$. The neural network is trained to minimize a loss function $\mathcal{L}$ that measures the difference between the predicted output $y$ and the true output $\hat{y}$ [11].

A general form of the loss function can be written as:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, \hat{y}_i)$$

where $\ell(y_i, \hat{y}_i)$ is the loss for a single prediction. This form can represent various specific loss functions, such as mean squared error (MSE) for regression or cross-entropy loss for classification [12].

## 1.2.2 Synaptic Plasticity

Synaptic plasticity is fundamental to learning in neural networks. The learning problem is a matter of finding the right values of synaptic weights in the network which is done by making changes to these weights to achieve an optimal network state. Synaptic plasticity refers to these changes in the strengths of synaptic weights in the network. The mechanisms of synaptic plasticity are what determine how a network learns.

### Backpropagation

To minimize the loss function, the backpropagation of error algorithm is used to compute the gradients of the loss with respect to the network parameters (weights and biases). These gradients are then used to update the parameters using an optimization algorithm, typically stochastic gradient descent (SGD) or one of its variants [13].

The gradient of the loss function with respect to the weights in layer $l$ is computed using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$

where $\delta^{(l)}$ is the error term for layer $l$:

$$\delta^{(l)} = \begin{cases} \frac{\partial \mathcal{L}}{\partial a^{(L)}} \odot \sigma'(z^{(L)}) & \text{if } l \text{ is the output layer} \\ (W^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)}) & \text{otherwise} \end{cases}$$

Here, $\odot$ denotes the element-wise multiplication, and $\sigma'$ is the derivative of the activation function [1].

Once the gradient is calculated, the weights are updated in the direction that minimizes the loss function, with magnitude determined by the learning rate, $\eta$:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}}$$

This is repeated until the loss function reaches a minimum and learning has converged.

#### Stochastic Gradient Descent

Gradient descent optimization can refer to either standard (batch) gradient descent or stochastic gradient descent (SGD).

Standard (Batch) Gradient Descent calculates the gradient of the full loss function, which is the average loss over the entire dataset:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, \hat{y}_i)$$

The weights are updated after processing all data points. While this method ensures stable convergence, it can be computationally expensive and slow, especially for large datasets and quickly becomes infeasible [14].

Stochastic Gradient Descent (SGD) calculates the gradient of the loss function using only a single data point at a time:

$$\mathcal{L}_i = \ell(y_i, \hat{y}_i)$$

The weights are updated after each data point. By updating the network after each data point rather than processing all data points, the network can converge faster with reduced computation and memory requirement, though this may introduce instability into the learning process [15].

### 1.2.3 Neuromorphic Hardware

Neuromorphic hardware is an approach to computing that aims to mimic the structure and function of biological neural networks. The field is driven by the need for more energy-efficient systems for running artificial neural networks. Neuromorphic systems aim to significantly reduce energy consumption without com-

promising computational capabilities, making them ideal for applications where energy efficiency is critical [5].

Neuromorphic computing encompasses a range of technologies and methodologies. These systems are typically associated with Spiking Neural Networks (SNNs) [3], though there are many examples of neuromorphic systems that do not implement SNNs [16]. This thesis is specifically relevant to spiking neuromorphic systems. Spiking neuromorphic systems running SNNs use discrete, binary events (spikes) to represent and transmit information. Inspired by the brain, spiking neuromorphic systems are event-driven, processing information only when events occur, rather than being clock-driven, to reduce power consumption. They also rely on parallel processing, performing operations in parallel to improve the speed and efficiency of computation [4].

Neuromorphic systems can be broadly categorized into three types: digital, analog, and hybrid. Digital neuromorphic systems, such as IBM's TrueNorth [17] and Intel's Loihi [18], use digital circuits to emulate neural behaviour. They offer high precision and are easier to scale but consume more power than analog systems. Analog neuromorphic systems, like BrainScaleS [19], use analog circuits to directly emulate neural dynamics. These systems have the potential for greater energy efficiency but face challenges in scaling and precision. Hybrid systems, such as the Tianjic chip [20], combine both digital and analog elements to leverage the advantages of both approaches.

Applications for neuromorphic hardware are emerging in fields where real-time, energy-efficient processing is essential. Neuromorphic systems are useful in edge computing, where computation is performed in devices rather than in centralized servers to reduce latency, making energy efficiency essential [21]. Brain-computer interfaces (BCIs) may benefit from neuromorphic hardware's ability to process neural signals efficiently [4, 22]. Adaptive control systems are another application where neuromorphic hardware can enable rapid, low-power decision-making for robotics and industrial automation [16].

Despite their potential, neuromorphic systems face several challenges. Scaling neuromorphic systems to incorporate larger numbers of artificial neurons while maintaining energy efficiency is a significant challenge in terms of hardware design [17]. Programming neuromorphic systems is another challenge, which has led to the development of specialized neuromorphic programming frameworks [18]. Another critical challenge, and the focus of this thesis, is the development of efficient learning algorithms for neuromorphic hardware. While traditional neural networks rely on backpropagation, this algorithm is not directly applicable to most neuromorphic systems due to its dependence on a separate memory to store information for the gradient calculation, necessitating the development of new, hardware-compatible learning approaches [23, 24].

### 1.2.4 Spiking Neural Networks

SNNs are a special type of ANN that can be run in spiking neuromorphic hardware. They share the fundamental structure of nodes connected by weights but differ significantly in operation from neural networks made of point neurons. Instead of continuous value activations, spiking neurons communicate using binary signals known as spikes, similar to action potentials in biological neurons. spiking neurons have a dynamic state represented by a membrane potential, unlike the static nonlinear activations in ANNs. While ANNs are simply functions that produce a static output given a static input, SNNs process time-varying inputs to produce time-varying outputs [25].

There are many types of spiking neurons which may be implemented with neuromorphic systems ranging in complexity including various integrate-and-fire neurons and more biologically realistic Hodgkin-Huxley neurons. Leaky integrate-and-fire (LIF) neurons are a common choice because they are relatively simple to simulate and implement in hardware [26].

#### Leaky-Integrate-and-Fire Neuron

LIF neurons integrate inputs and decay exponentially at a rate determined by their membrane time constant. The leaky integrator dynamics can be modelled by a basic RC circuit [27]. When the membrane potential reaches a threshold, the neuron fires a spike and undergoes reset dynamics.

Mathematically, the membrane potential $V(t)$ of a LIF neuron evolves according to the following differential equation:

$$\tau \frac{dV(t)}{dt} = -(V(t) - E_L) + \frac{I(t)}{g_L}$$

where $\tau$ is the membrane time constant, $g_L$ is the leak conductance, $E_L$ is the leak reversal potential, and $I(t)$ is the input current. When $V(t)$ reaches a threshold $V_{\text{th}}$, the neuron emits a spike, and the membrane potential is reset to a resting potential $V_{\text{reset}}$. Figure 1.2 illustrates the simulated time evolution of the membrane potential of a Leaky Integrate-and-Fire (LIF) with a noisy input current.

In a spiking neural network (SNN), the input current $I(t)$ to a LIF neuron is the weighted output of other neurons:

$$I(t) = \sum_j w_{ij} S_j(t)$$

where $w_{ij}$ represents the synaptic weight from neuron $j$ to neuron $i$, and $S_j(t)$ is the spike train of neuron $j$ [25].

In summary, the LIF neuron operates by integrating the weighted contributions from other neurons, decays over time, and generates spikes when its membrane potential reaches the threshold.

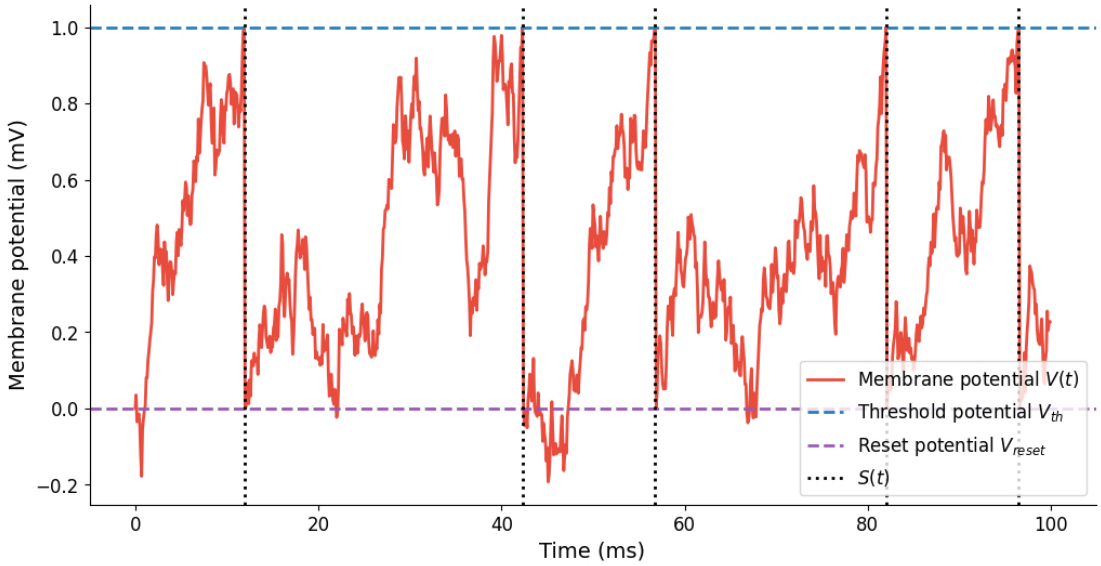


Figure 1.2: **Time Evolution of a Leaky Integrate-and-Fire (LIF) Neuron with Noisy Input.** Membrane potential $V(t)$ of an LIF neuron for 100 ms, subject to noisy input current. The parameters used in the simulation are: membrane time constant $\tau = 10.0$ ms, leak reversal potential $E_L = 0.0$ mV, threshold potential $V_{\text{threshold}} = 1.0$ mV, reset potential $V_{\text{reset}} = 0.0$ mV, leak conductance $g_L = 1.0$ µS, mean input current $I_e = 1.0$ nA, and standard deviation of the noise noise_std $= 5.0$ nA.

### Neural Coding

A fundamental aspect of Spiking Neural Networks (SNNs) is how they encode and transmit information through spikes. When designing an SNN algorithm, both the encoding of input information and the decoding of output information must be considered. Common coding methods include rate coding, where information is represented by the frequency of spikes, and latency coding, where information is represented by the timing of specific spikes [28]. Other encoding methods include phase coding and burst coding [25, 29].

Rate coding is more robust to noise. It is conceptually similar to traditional neural networks since neuron activation can be interpreted as analogous to neuron firing rates. Latency coding, while potentially more information-dense and energy-efficient, typically requires more specialized learning approaches [30]. This work focuses on training rate-coded SNNs.

### 1.2.5 Simulating SNNs

While the motivation for developing SNNs is for implementation in spiking neuromorphic hardware, these networks can be simulated on CPUs or GPUs.

To simulate SNNs, the continuous-time dynamics of the neurons are approximated in discrete timesteps. This discretization allows for the numerical integration of the differential equations governing the neuron's behaviour. The forward Euler method is a simple and commonly used numerical integration method [31].

Using the forward Euler method, the membrane potential $V(t)$ of a LIF neuron at a discrete time step $t + \Delta t$ can be updated as follows:

$$V(t + \Delta t) = V(t) + \frac{\Delta t}{\tau} \left( -(V(t) - E_L) + \frac{I(t)}{g_L} \right)$$

where $\Delta t$ is the timestep size, $\tau$ is the membrane time constant, $E_L$ is the leak reversal potential, $I(t)$ is the input current, and $g_L$ is the leak conductance. Figure 1.2 illustrates a simulated LIF neuron over time.

In a feedforward spiking neural network, the input to layer $l$ at time $t$ can be set either as the output of layer $l - 1$ at the same time $t$ or the previous time step $t - 1$, which may impact the behaviour of the network. In this thesis, the former option is used in simulations.

### 1.2.6 Training SNNs

There are various methods for training SNNs based on backpropagation and synaptic plasticity.

#### Backpropagation for SNNs

Training Spiking Neural Networks (SNNs) with backpropagation is more complex than for non-spiking Artificial Neural Networks (ANNs) due to the discontinuity of spikes, which prevents the definition of an exact gradient for credit assignment. However, several methods have been developed to enable the use of backpropagation for SNNs.

Artificial neural network (ANN)-SNN Conversion involves training a non-spiking ANN and then converting the network parameters to an SNN. The nonlinear activations of the ANN are interpreted as firing rates in the SNN. This approach leverages the straightforward training of ANNs and transfers the learned parameters to the SNN, maintaining similar performance [32].

Backpropagation can be done in SNNs by considering the timing of spikes.

Spikeprop performs backpropagation using the precise spike times to calculate gradients [33].

Backpropagation-through-time, a method for training recurrent neural networks where the error is calculated backwards through the duration of the forward pass, can also be applied to SNNs with Surrogate Gradient Descent. This method accomplishes this by approximating the discontinuous gradient of spike events with a smooth surrogate gradient [34].

While these methods are effective, they are not suitable for training SNNs with neuromorphic hardware since they involve nonlocal computations. As a result, alternative approaches must be explored to train SNNs in spiking hardware that operate on local, spike-based learning rules [23].

### Local Learning and the Weight Transport Problem

The principle of locality is crucial for learning in neuromorphic systems. Local learning rules operate under the constraint that updates to synaptic weights are based only on information available at the location of the synapse at the time of plasticity.

Backpropagation violates this principle in two significant ways. It requires global information from the forward pass about the state of neurons and spike times to calculate the gradient in the backward pass, thus violating temporal locality. Further, backpropagation assumes that the exact weights used during the forward pass are available during the backward pass to compute the gradients. This is known as the weight transport problem, which arises because the weights $W^{(l)}$ must be accessible across the network for accurate gradient calculation [35].

Mathematically, the gradient of the loss function $\mathcal{L}$ with respect to a weight $w_{ij}^{(l)}$ connecting neurons $i$ and $j$ in layer $l$ is given by:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} \cdot a_i^{(l-1)}$$

where $\delta_j^{(l)}$ is the error signal at neuron $j$ in layer $l$, and $a_i^{(l-1)}$ is the activation of neuron $i$ from the previous layer. This calculation necessitates that the weights $w_{ij}^{(l)}$ used in the forward pass are available during the backward pass, requiring a global communication mechanism that is impractical for neuromorphic systems [35].

Neuromorphic systems must rely on local synaptic plasticity, where learning rules adjust synaptic weights based solely on local information, such as the timing of pre and postsynaptic spikes and the state of the synapse. If error propagation is required, it must be done in a way that bypasses the weight transport problem [23].

### Spike-Time-Dependent Plasticity

Spike-time-dependent plasticity (STDP) is a well-established family of local learning rules, derived from Hebbian learning, that determines synaptic plasticity based on the precise timing of pre- and postsynaptic spikes [36]. However, STDP does not incorporate the backpropagation of credit-carrying error information typically used for supervised learning. Therefore, additional methods are needed for local supervised learning in neuromorphic hardware. One such method is burst-dependent learning, which combines the principles of STDP with backward error propagation to enable supervised learning in these systems [24].

### Burst-Dependent Learning

Burst-dependent learning, or Burstprop, is a method for training hierarchical rate-encoded spiking neural networks (SNNs) using single-phase local learning. It approximates backpropagation by performing credit assignment through two-compartment neurons, each with a separate soma and apical dendrite. The soma integrates feedforward signals, while the apical dendrite handles feedback error signals. Burstprop operates by multiplexing distinct error signals, or bursts, which are transmitted backward through the network and integrated by the apical compartment. The error information guides synaptic plasticity as the network processes feedforward information [24, 37].

While Burstprop is a promising learning algorithm for training spiking neural networks (SNNs) in neuromorphic hardware, its experimental applications have been limited. It has only demonstrated success in solving a simple spiking exclusive or (XOR) task. XOR is an operation where the output is true if and only if the two inputs differ and serves as a simple problem to demonstrate the nonlinear learning capabilities of the algorithm. It is also unclear whether it can be implemented without violating the weight transport problem. To extend its utility and potential to facilitate learning in neuromorphic hardware, further improvements are needed to enable Burstprop to address more complex learning problems.

## 1.3 Statement of Problem

Burstprop has shown preliminary success in coordinating local learning in SNNs. While initial experiments solving a spiking XOR task yielded positive results, this task is far simpler than most real-world applications of neuromorphic learning. Additionally, solving this task with Burstprop required a network of thousands of spiking neurons, which is highly inefficient for such a simple problem [24, 37].

These initial experiments leave several questions unanswered that are crucial for evaluating the potential usefulness of Burstprop in neuromorphic learning applications. Specifically, can Burstprop learn more efficiently with networks of fewer neurons, and does it scale to more complex supervised learning problems?

Furthermore, can these improvements be achieved while avoiding weight transport and using low-resolution weights? This thesis aims to improve the Burstprop algorithm to address these questions.

## 1.4 Proposed Solutions

To address the limitations of the Burstprop algorithm, this thesis proposes several enhancements aimed at improving its efficiency, scalability, and practical applicability in neuromorphic learning systems.

Firstly, we simplify network simulation by using a marked point process to replace bursts and spike firing patterns. This approach eliminates the need to simulate short-term synaptic plasticity dynamics and allows the use of a simpler neuron model. Additionally, we introduce a local regularization method to reduce network activity and enhance the stability of the learning process. Furthermore, a ternary burst signal is implemented to improve the precision of error propagation and synaptic plasticity.

To address the weight transport problem, we incorporate feedback alignment and direct feedback alignment methods into the Burstprop algorithm. We also develop a method enabling Burstprop to learn on a network with low-resolution synaptic weights.

For evaluation, Burstprop is tested on a rate-encoded spiking version of the MNIST handwritten digit dataset, a commonly used dataset for training and testing machine learning algorithms, to demonstrate its learning capabilities. The learning process is simulated on a CPU using Python, using discrete time steps and the forward Euler method to calculate neuron dynamics.

## 1.5 Outline of Thesis

The outline of the thesis is as follows: Related Work and relevant background on neural network learning methods are described in Chapter 2. The methods describing the specific details of the Burstprop learning algorithm are described in Chapter 3. The experimental results of the algorithm applied to the spiking MNIST handwritten digit dataset are outlined in Chapter 4. The conclusion and future work are discussed in Chapter 5. Chapters 2-4 are included in the article by the same title as this thesis, submitted to IOPScience Neuromorphic Computing and Engineering.

# Chapter 2

# Background

## 2.1 Learning in SNNs

Multiple methods for training SNNs have been developed. We first distinguish methods relying on non-local plasticity rules from those using local ones. The local plasticity rules can be further separated into burst-dependent plasticity and spike-timing-dependent plasticity.

### 2.1.1 Non-Local Learning Methods

Nonlocal learning methods for SNNs include ANN-SNN conversion [38, 39, 40, 41] and backpropagation-through-time (BPTT) [42, 43]. With ANN-SNN conversion, a rate-based neural network is trained using the backpropagation algorithm which is then converted to a spiking neural network. In real applications, imperfect conversions will strongly alter performance, but this can be remedied with further fine-tuning [44]. With BPTT, backpropagation is performed on an SNN by using surrogate gradients to propagate the gradient through the spiking nonlinearity. To backpropagate gradients through time, the gradient must keep track of the network state and activity at each previous time step. This algorithm is therefore highly nonlocal (but local approximations have been derived [45]). These methods do not address the problem faced with training neuromorphic online, we will use these methods to provide a point of reference for the performance metrics.

### 2.1.2 Spike-Time-Dependent Plasticity Learning Methods

Spike-time-dependent plasticity (STDP) alters connectivity according to the relative timing of spikes in the presynaptic and postsynaptic neurons and is therefore fully local. One local learning algorithm, proposed by Kappel et al. [46] combines a STDP plasticity rule and winner-take-all dynamics and recurrent excitation to

approximate hidden Markov model learning in SNNs. Relatedly, Diehl and Cook [47] combined STDP with winner-take-all dynamics to train a non-recurrent network. Reading out the resulting representation with a supervised plasticity rule, they showed that a single layer could achieve 95% accuracy on MNIST. As only the readout is supervised, this method is sometimes referred to as unsupervised. Another STDP approach combines a trace set by relative coincidence and a global reinforcement by reward-associated signals [48, 49, 50], but such three-factor learning rules tend to scale poorly on problems having a high number of dimensions. Also, how these STDP approaches can scale to multiple layers in a hierarchy remains unknown.

### 2.1.3 Burstprop Learning Methods

Another class of local learning algorithms depend on separate somatic and dendritic compartments to represent both feedback error signals and inference signals within the neuron, often through a rate-based representation [51, 52, 53, 54]. For spiking neural networks, burst-dependent algorithms exploit the target-dependence of short-term plasticity [55, 56, 57], dendrite-dependent bursting [58, 59] and burst-dependent plasticity [60, 61, 62, 63] to perform credit assignment in a way that approximates backpropagation [64, 37, 65]. These burst-dependent learning methods, referred to as 'Burstprop' algorithms, can in principle allow for multilayer local learning in SNNs.

The original formulation [64] of the algorithm learns online but with signals coming in two phases. An initial phase in which synaptic plasticity was disabled was required to determine each neuron's baseline burst probability which was then used in the second phase to calculate synaptic plasticity, violating temporal locality. A second formulation of the algorithm (called BurstCCN [37]) adapted the learning rule for single-phase learning by including a second feedback pathway that communicates spike signals such that a baseline burst probability can be preset, avoiding the need for two phases. Addressing complex problems with these algorithms has been almost entirely limited to rate-based networks. Spiking proof-of-principles were done on the XOR problem.

## 2.2 Weight-Transport-Free Learning

Approaches approximating the backpropagation of error algorithm face a problem known as the weight-transport problem. To accurately approximate the gradient, the feedback pathways must have synaptic weights that are symmetric to feedforward weights (symmetric feedback). Such direct weight-transport involves nonlocal communication weight values between synapses and must be avoided for learning in spiking neuromorphic hardware. Many solutions have been proposed [66, 67, 68, 69, 70, 64, 71, 72], including imposing random but static feedback connectivity (feedback alignment [66, 67]) and learning rules that seek to transport

the weights by exploiting correlations in the activity patterns [69, 64, 71].

### 2.2.1 Feedback Alignment Methods

It has been shown that feedback weights need not be initialized as symmetrical to feedforward weights, nor to be kept tightly symmetrical throughout learning to effectively train networks using backpropagation [66]. As long as feedforward weights are initialized such that, if the weights were represented as a vector, it is within 90° of the feedback weight vector, feedforward weights trained with backpropagation tend to align with feedback weights, leading to accurate credit assignment and effective learning. Since feedforward synapses must align with feedback synapses, this method is referred to as feedback alignment (FA) [66]. Building upon this work, direct feedback alignment (DFA) uses random feedback weights directly connecting network outputs to each layer and has also been shown to be effective for learning with backpropagation [67]. While these methods are effective for training shallow, rate-based neural networks, their efficacy tends to decrease with increasing network depth [73, 74, 75].

The question of whether these techniques also scale to spiking neural networks has been partially addressed in Samadi et al. [68], where a spiking feedforward inference phase was combined with an analog learning phase that follows the backpropagation of error algorithm. Zenke and Ganguli [76] have similarly paired a spiking inference phased with analog, non-local, BPTT learning. Relatedly, Zhao et al. [77] have studied SNNs receiving direct but analog gradient information. Xing et al. [78] have used a signed spiking communication for the feedforward inference phase, but again paired with analog BPTT communication of gradients. In Payeur et al. (2021) [64] a fully spike-based communication of both feedforward and learning signals, but the feedback alignment approach was only applied to the XOR problem. The present work focuses on MNIST in a fully spiking communication of both feedforward and learning signals.

### 2.2.2 Learned Feedback

Several methods implement a learning rule on feedback synapses, allowing them to align with feedforward weights. With the Kolen-Pollack method [79], weight symmetry is achieved by assigning the same plasticity rule to feedforward and feedback synapses and including a weight decay term. While this is effective in aligning weights, the necessity of weight decay may be a problem for some learning methods. Phaseless Alignment Learning (PAL) aligns feedback weights to feedforward weights by learning feedback weights from neuronal noise [71] outside of the presentation of examples. Another algorithm proposed by Burbank [80] uses temporally opposed STDP learning rules on feedforward and feedback synapses to successfully train a spiking autoencoder network. Similarly, Payeur et al. [64] provide a spike-based learning rule for learning the alignment of feedback synapses, which have not been tested on SNNs. These methods form a conceptual rationale

for implementing symmetry as a partially accurate bypass of the learned feedback method.

# Chapter 3

# Methods

## 3.1 Neuron and Synapse Model

### 3.1.1 Somatic Compartment

Each neuron's soma follows Leaky Integrate and Fire (LIF) dynamics:

$$\tau_s \frac{dV_s(t)}{dt} = -(V_s(t) - E_L) + \frac{I_s(t)}{g_L} \tag{3.1}$$

where $V_s(t)$ represents the somatic membrane potential, $\tau_s$ the membrane time constant, $E_L$ the resting potential, $g_L$ the leak conductance, and $I_s(t)$ the input current. In our experiments, the somatic membrane time constant was set to $\tau_s = 10\,\text{ms}$. The neuron fires an event when $V_s(t)$ exceeds the threshold potential $V_{\text{threshold}}$, following which $V_s(t)$ resets to $V_{\text{reset}}$:

$$\text{if } V_s(t) \geq V_{\text{threshold}}, \text{ then } V_s(t) \to V_{\text{reset}}. \tag{3.2}$$

Rescaling to dimensionless variables , we define $\tilde{I}_s = I_s/g_L(V_{\text{threshold}} - E_L)$ and $\tilde{V}_s = V_s - E_L/(V_{\text{threshold}} - E_L)$, equation (1) becomes

$$\tau_s \frac{d\tilde{V}_s(t)}{dt} = -\tilde{V}_s(t) + \tilde{I}_s(t). \tag{3.3}$$

Every time the somatic voltage crosses the threshold, an event is emitted. This event can be either a single spike or a burst (see next section). Although in biology a burst means several spikes at a high frequency, here they are modelled as a marked point process, tagging every event with a type.

### 3.1.2 Eligibility Trace

We use eligibility traces, $\tilde{E}_j(t)$, to indicate the eligibility of individual synapses connecting from a lower-order neuron $j$ to some other higher-order neuron:

$$\tau_{\tilde{E}} \frac{d\tilde{E}_j(t)}{dt} = -\tilde{E}_j(t) + S_j(t). \tag{3.4}$$

Here the event train $S_i(t) = \sum_k \delta(t - t_k^{(i)})$, where $t_k^{(i)}$ is the $k^{th}$ event time from neuron $i$. The eligibility trace time constant was to $\tau_{\tilde{E}} = 10\,\text{ms}$.

### 3.1.3 Dendritic Compartment and Burst Generation

Dendrites come with different properties to the neuron's soma [81]. Here the dendritic compartment refers to an apical-like dendrite with a segregated membrane potential that can nonlinearly integrate feedback information to modulate neuron burst generation. It follows leaky integrator dynamics, using dimensionless inputs and membrane potential:

$$\tau_d \frac{d\tilde{V}_d(t)}{dt} = -\tilde{V}_d(t) + \tilde{I}_d(t). \tag{3.5}$$

where $\tilde{V}_d(t)$ represents the re-scaled dendritic membrane potential, $\tau_d$ is the dendritic membrane time constant, and $\tilde{I}_d$ is the re-scaled net dendritic input. We set the dendritic membrane time constant to $\tau_d = 10\,\text{ms}$.

There is no threshold in the dendritic compartment, instead, every time the soma emits an event, the state of the dendrite is used to determine if this event is a burst or a single spike. We consider two bursting models.

In the unsigned bursting model, the probability that an event is made into a burst is proportional to the dendritic potential $\tilde{V}_d$, we use a sigmoidal relationship $P_B(\tilde{V}_d(t)|\text{event at } t) = \sigma(\tilde{V}_d(t)) = \frac{1}{1+e^{-\tilde{V}_d(t)}}$.

In the signed bursting model, two types of bursts can be emitted: long bursts and short bursts. Either type of burst can only be generated upon the firing of an event from the soma. Given a somatic event, a burst is generated according to the symmetric function $P_B(\tilde{V}_d(t)|\text{event at } t) = \tanh(\text{abs}(\tilde{V}_d))$. If a burst is said to be generated, the sign of $V_d(t)$ is used to determine its type. If $\tilde{V}_d < 0$, a short burst is generated. If on the other hand $\tilde{V}_d > 0$, a long burst is generated. Because they represent the sign of the dendritic input, we refer to short and long bursts as negative and positive bursts, respectively. In this model, burst generation triggers a reset of the dendritic potential to the baseline, comparable to the somatic membrane potential reset upon a neuron spiking. This adds some refractoriness allowing the algorithm to better approximate backpropagation.

## 3.2 Network Architecture

The SNNs consist of an input layer of 784 units, corresponding to the number of pixels in each sample image in the MNIST handwritten digit dataset. An example of an image sample can be found in 3.1. There are 1, 2, 3 or 4 hidden layers consisting of 100, 200, 400 or 800 hidden neurons as specified in the figures and associated text. The output layer of 10 neurons corresponds to the 10 classes of the MNIST dataset. Except for input units, each unit follows the two-compartment model described above. The network inputs (see Sect. 3.8), as well as any connection between hidden layers or from the hidden layer to the output layer follow the fully connected configuration. Weights are allowed to be positive or negative. The label corresponding to the network's class is used in calculating the network's output error and teaching signal (see Sect. 3.4.1).

## 3.3 Synaptic Communication and Multiplexing

In our fully connected hierarchical SNNs, the credit-carrying information is communicated by multiplexing feedforward signals using events and feedback error signals using bursts. For the feedforward signals, we assume that synapses operate instantaneous changes in postsynaptic membrane potential upon every event. The input $I_s^{(i)}(t)$ to the soma of neuron $i$ is defined by the equation:

$$I_s^{(i)}(t) = \sum_j w_{ij} S_j(t). \tag{3.6}$$

In this equation, $w_{ij}$ is the feedforward synaptic weight from lower-order neuron $j$ to higher-order neuron $i$ and $S_j(t)$ is the event train of neuron $j$.

### 3.3.1 Unsigned burst Communication

For the feedback signals, we again assume that synapses operate instantaneous changes in postsynaptic membrane potential, but upon every burst. We further enforce that these feedback connections target the apical dendrite and therefore only affect the dendritic input $I_d^{(i)}(t)$ of a given neuron $i$. In the unsigned burst model, we have

$$I_d^{(i)}(t) = \sum_j b_{ij} B_j(t) + I_{\text{rec}}^{(i)}. \tag{3.7}$$

where $b_{ij}$ are the feedback synaptic weight from higher-order neuron $j$ to lower-order neuron $i$ and $B_j(t)$ is the burst train of neuron $j$. The additional current $I_{\text{rec}}$ appears due to recurrent inputs, see Sect. 3.4.2.

### 3.3.2 Signed burst Communication

In the signed burst model, we assume the type of burst can dictate the sign of the post-synaptic effect:

$$I_d^{(i)}(t) = \sum_j b_{ij}(B_j^+(t) - B_j^-(t)) + I_{\text{rec}}^{(i)}. \tag{3.8}$$

We note that since the same feedback weight $b_{ij}$ operates on two distinct outputs of a neuron, this model requires weight transport.

### 3.3.3 Multiplexing

In both signed and unsigned bursting models, when a neuron fires an event, the probability of that neuron sending a positive or negative error signal backward in the network is modulated by the dendritic potential of the neuron, $V_d$, as demonstrated in Figure 3.2. Similarly, the event rate is proportional to the integrated somatic input (proportional to $V_s$ if there were no reset). Thus the burst probability and the event rate represent two different signals simultaneously. Since each neuron integrates error signals from all higher-order neurons in its dendritic compartment and through the burst generation mechanism, the burst probability signal represents the signal that started with the input onto the apical dendrites at the top of the hierarchy and then backpropagates through it. This allows for the simultaneous online communication of credit-carrying error information from higher to lower-order neurons and feedforward, inference signals from lower to higher-order neurons.

## 3.4 Teaching Signals and Target Rates

For dendritic inputs to be conceived as providing error signals, we inject a teaching signal at the top of the hierarchy (Sect. 3.4.1) and a layerwise regularisation signal (Sect. 3.4.2).

### 3.4.1 Teaching Signals

Teaching signals are used for supervision, applying them recurrently to the dendritic compartments of output neurons such that $I_{\text{rec}}^{(i)}(t) = T_i(t)$ for these neurons. These signals are derived from the exponential moving average of the neuron's firing rate, $\bar{E}(t)$, using the same time constant as for the eligibility trace (see Sect. 3.1.2). The teaching signal, $T(t)$ is calculated as the derivative of the local error, $\varepsilon_T(\bar{E}(t), \hat{E})$, of the neuron's instantaneous firing rate, $\bar{E}(t)$ with respect to the neuron's target rate, $\hat{E}$. For the neuron corresponding to the correct output class,

a high target rate of $\hat{E} = 200\,\text{Hz}$ is set, and for all other neurons, a low rate of $\hat{E} = 20\,\text{Hz}$ is used. With this target, the teaching signal becomes

$$T_i(t) = \gamma_T \frac{\partial \varepsilon_T}{\partial \bar{E}(t)} = -2\gamma_T(\hat{E}_i(t) - \bar{E}_i(t)), \tag{3.9}$$

where $\epsilon_T$ corresponds to a square error loss. Here $\gamma_T$ controls the strength of the teaching signal. This differential in target rates will backpropagate and control synaptic plasticity through the network, ensuring the correct output neuron becomes more active while other output neurons are suppressed. We found that a teaching signal strength of $\gamma_T = 10^{-3}$ produced a strong enough teaching signal to steer plasticity while avoiding error signal saturation.

### 3.4.2 Hidden target rates

Similar to the teaching signals applied to output neurons, hidden neurons receive recurrent regularization signals, such that $I_{\text{rec}}^{(i)}(t) = R_i(t)$ for all neurons except input neurons and output neurons. We use a uniformly low target rates $\hat{E} = 20\,\text{Hz}$:

$$R_i(t) = \gamma_R \frac{\partial \varepsilon_R}{\partial \bar{E}(t)} = -2\gamma_R(\hat{E} - \bar{E}_i(t)). \tag{3.10}$$

for $\epsilon_R$ a squared error loss. This $R_i(t)$ affects the apical dendrite of the same layer as $\bar{E}_i(t)$. We set $\gamma_R = 10^{-5} \ll \gamma_T$ such that the regularization does not suppress the error information due to the teaching signal. This approach regulates the firing rates of hidden neurons, thereby promoting sparsity and stability in the network's learning process.

## 3.5 Plasticity Rule

The synaptic plasticity rule is the mechanism by which changes are made to the strengths of synaptic weights in the network, allowing for learning. Our algorithm utilizes a local spike-based learning rule to approximate backpropagation-based gradient descent. The synaptic weight changes are calculated as

$$\frac{dw_{ij}}{dt} = -\eta \tilde{E}_j(t)(B_i^+(t) - B_i^-(t)) \tag{3.11}$$

for the signed bursting model, and

$$\frac{dw_{ij}}{dt} = -\eta \tilde{E}_j(t)(B_i(t) - B_P S_i(t)) \tag{3.12}$$

for the unsigned bursting model. Here, $B_P$ is the baseline burst probability. The parameter $\eta$ represents the learning rate and was set to $\eta = 10^{-4}$ to achieve stable learning.

For the signed bursting model, synaptic weight changes, occur upon postsynaptic burst generation, with the sign of the weight change determined by the type

of burst. For the unsigned bursting model negative weight changes occur upon postsynaptic spikes and positive weight changes occur upon postsynaptic bursts. In both models, the magnitude of plasticity is proportional to the presynaptic neuron's eligibility trace and the postsynaptic event rate. Plasticity, then, depends on three factors: the activity of the presynaptic neuron (via the eligibility trace), the postsynaptic neuron's activity (triggering plasticity), and the integrated upstream error (via the burst fraction or burst type fraction). These three factors are essential to emulate the backpropagation of error algorithm, where weight adjustments of a given unit depend on three factors: the incoming activity, the derivative of the unit's activity with respect to the sum of its inputs, and the integrated downstream error.

It is worth noting the distinct behaviour of signed and unsigned conditions in the absence of any postsynaptic error ($\tilde{V}_d(t) = 0$). For the unsigned burst model, this condition results in a burst probability of 0.5. Because $\tilde{V}_d = 0$ corresponds to the absence of dendritic input, we call the resulting output burst probability the baseline burst probability. In the associated plasticity rule, the parameter, $B_P = 0.5$, is chosen to match this baseline burst probability, such that there is a balance between positive and negative weight changes. We note that although this ensures that $\Delta w = 0$ on average when $\tilde{V}_d(t) = 0$, the fluctuations around this can cause drifts in the learned state over time. For the signed burst model, plasticity is more stable because the probability of burst generation is 0 when $\tilde{V}_d(t) = 0$ so no plasticity can occur.

## 3.6 Low-Resolution Weights

To implement low-resolution synaptic weights, $N^2 - 1$ evenly spaced possible weight values were determined ranging from -1 and 1, where $N$ is the weight resolution in bits. The magnitude of the interval between possible weight values is referred to as $\epsilon$. This low resolution was imposed on initial weights and required an adaptation of the plasticity rule.

Initial weights (see Sect. 3.7) were rounded to the nearest possible weight value. During learning, an additional plasticity accumulator variable is added for each synapse. Plasticity is calculated according to the learning rule and added to the accumulator variable. Once a synaptic accumulator reaches $\pm 0.5\epsilon$, the synaptic weight potentiates or depresses by $\epsilon$ and the accumulator variable weight resets. In this way, the network can learn with low-resolution weights while still having an effective learning rate of less than $\epsilon$.

## 3.7 Weight Initialization

Feedforward synaptic weights were initialized from a normal distribution with standard deviation proportional to the square root of the number of presynaptic

neurons to ensure stable learning [82]. It was found that to ensure error signals do not saturate, feedback weights had to be scaled by the number of higher-order neurons divided by the number of lower-order neurons in the layers connected by these weights. For the symmetric feedback implementation, feedback weights are aligned to feedforward weights. For FA and DFA, feedback weights were initialized from a normal distribution with the standard deviation scaled according to the number of higher-order neurons divided by the number of lower-order neurons while ensuring initial feedback weight matrices are within 90° of feedforward weights, as is done in [66].

## 3.8 Input Encoding

In our study, we encoded the MNIST Digit Classification Dataset images for a spiking neural network. The dataset consists of 60,000 training and 10,000 test grayscale images, each 28x28 pixels [83]. The images were normalized so that the average pixel intensity of each image was constant. Pixel intensities, ranging from 0 to 1, were linearly mapped to firing rates between 20 Hz and 200 Hz following a Poisson process. Accordingly, each pixel in each image was converted into an input unit spike train of a duration of 100 ms.

## 3.9 Simulation Methods

We simulated the SNNs using a CPU with Python. Neuron dynamics were simulated using the forward Euler method with a 1 ms time step. For each time step, the network was updated from the input layer to the output layer. Feedforward inputs to a layer were taken from the lower-order outputs at the same time step and feedback inputs to a layer were taken from the higher-order outputs at the previous time step.

### 3.9.1 Training

During training, samples from the MNIST dataset are presented to the network sequentially. Teaching signals are applied continuously, and plasticity is engaged throughout the 100 ms. Neuronal state variables ($\tilde{V}_d$ and $\tilde{V}_s$), eligibility traces ($\tilde{E}$) and rate estimates ($\bar{E}$) are reset to zero at the end of each sample presentation.

### 3.9.2 Testing

During testing, test samples are presented the same way as in the training phase, but there are no inputs to the dendrites and plasticity is off. The most active output neuron over the sample duration decides the network's predicted class.

## 3.10 Backpropagation-Through-Time

We compare our results with a backpropagation-through-time method using surrogate gradients [42]. We use the sigmoid function to approximate the spike discontinuity in the backward pass. The neurons follow the same dynamics, with the same parameters as the somatic neuron compartments previously described. We trained the network on the same input encoding, with each sample presented also for 100 ms and used a learning rate of $\eta = 2 \times 10^{-4}$ to balance stability and speed of learning.

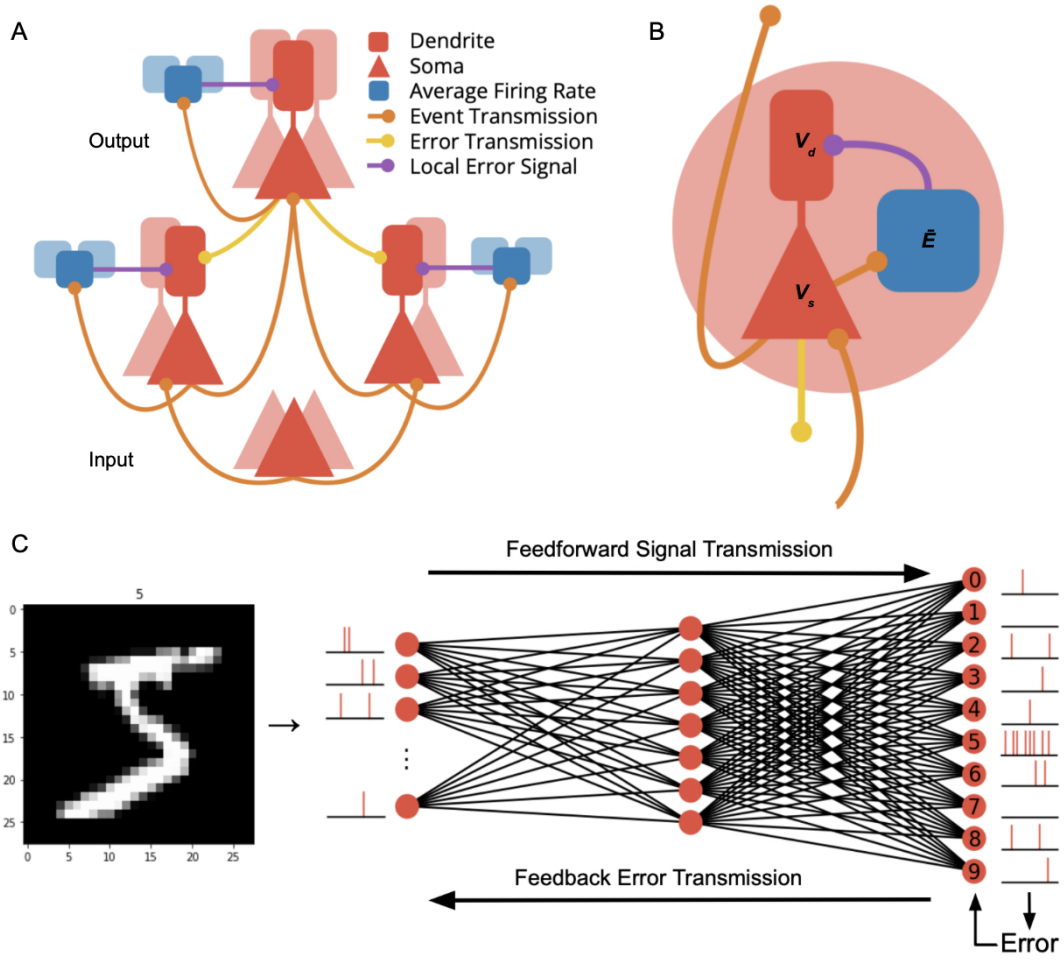


Figure 3.1: **Overview of Burstprop connectivity and signal transmission:** A. Schematic of the core spiking neural network architecture. Note that feedforward inputs target somas and feedback inputs target feedback. B. Schematic of the core unit. Neuron receives feedforward inputs to its somatic compartment, transmits events to higher-order neurons in the network and bursts/events to lower-order neurons in the network. The neuron's local error is calculated from its average firing rate and target rate and the error signal applied to the neuron's apical dendrite, combined with error signals received from the higher-order neurons. C. Schematic of the network spiking neural network used in the MNIST classification task. A sample image is converted to input Poisson spike trains which are propagated forward through the network. Each red circle corresponds to a neuron with a soma, a dendrite and a self connection as in panel B. The output layer provides the prediction according to the highest firing neuron. An error is continuously calculated and fed back to the output layer. Black lines are meant to indicate the presence of a connection in both directions, according to panel A.

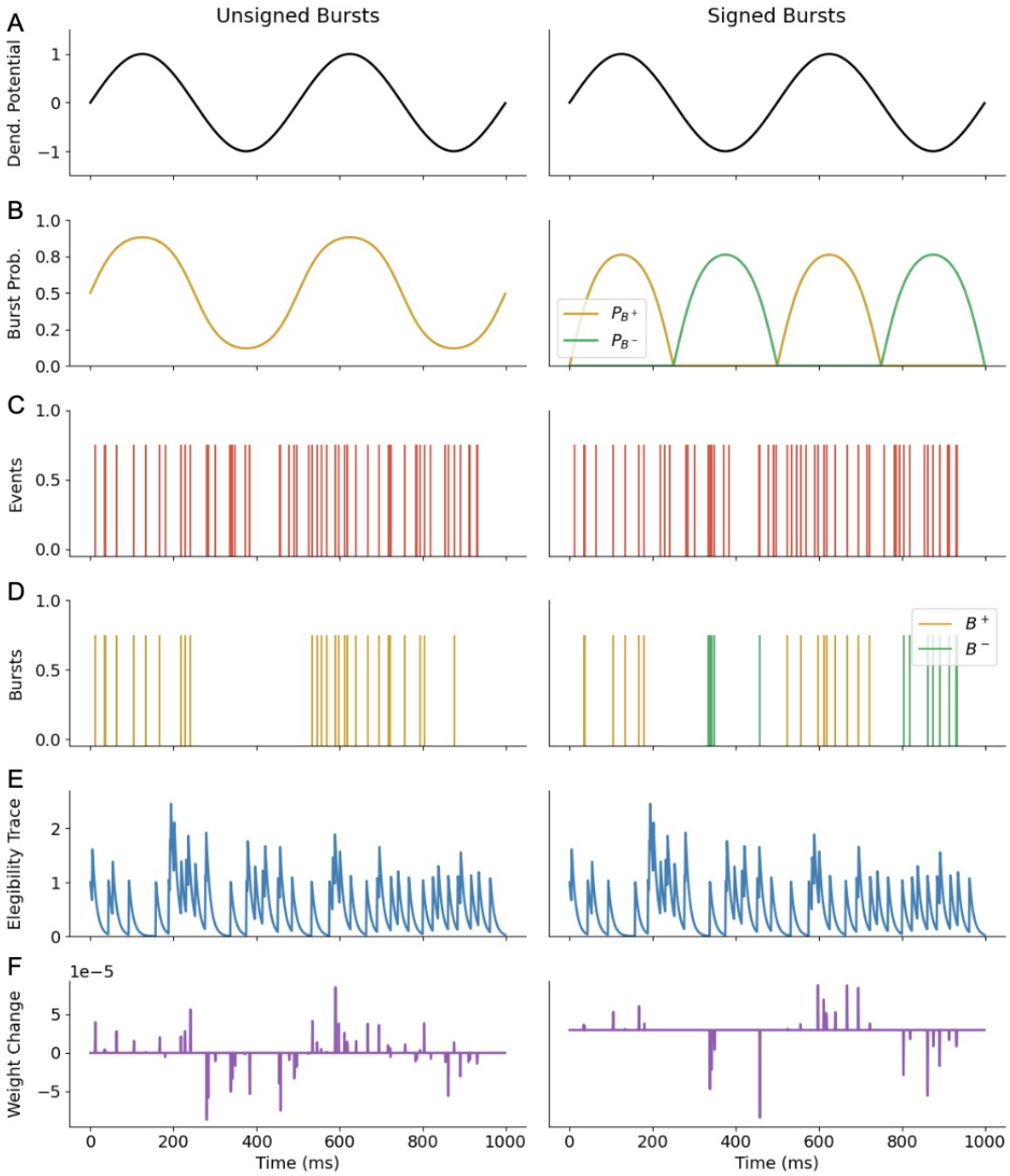


Figure 3.2: **Dendritic potential steers direction of plasticity for synapses to neurons.** A. Time-varying dendritic potential following a sine curve. B. Burst probabilities. Unsigned Burstprop, $P_B = \sigma(V_d)$. Signed Burstprop, $P_{B^+} = \max(0, \tanh(V_d))$ and $P_{B^-} = \max(0, -\tanh(V_d))$. C. Poisson spike train, with 50 Hz firing rate. D. Burst train generated from spike train and burst probability. E. Eligibility trace of a presynaptic neuron. F. Resultant synaptic plasticity.

# Chapter 4

# Results

We trained SNNs with Burst-CCN (Burstprop, for short [64, 37]) on a rate-coded version of the MNIST dataset. The algorithm uses two-compartment neuron models with a LIF somatic compartment and dendrite-dependent bursting implemented as a marked point process. Single spikes are sent to somata up the hierarchy while dendrite-dependent bursts are sent to dendrites down the hierarchy, thus multiplexing feedforward signals with feedback signals. Feedback signals are made credit-carrying by letting the dendrites of output units receive the classification error. Plasticity follows a local, burst-dependent rule chosen to approximate the backpropagation of error algorithm.

While Burstprop uses increases and decreases of burst probability to communicate positive and negative signs of the error signals (Fig. 3.2), here we also considered an alternative model where the different sign of the error signal triggers different types of bursts. Physiologically, this 'Signed Burst' model could correspond to bursts of different durations engaging different short-term plasticity [57] and long-term plasticity mechanics. By construction, the signed burst model is more stable and less subject to drifts in plasticity than the original Burstprop model.

## 4.1 Deep Learning on MNIST Dataset

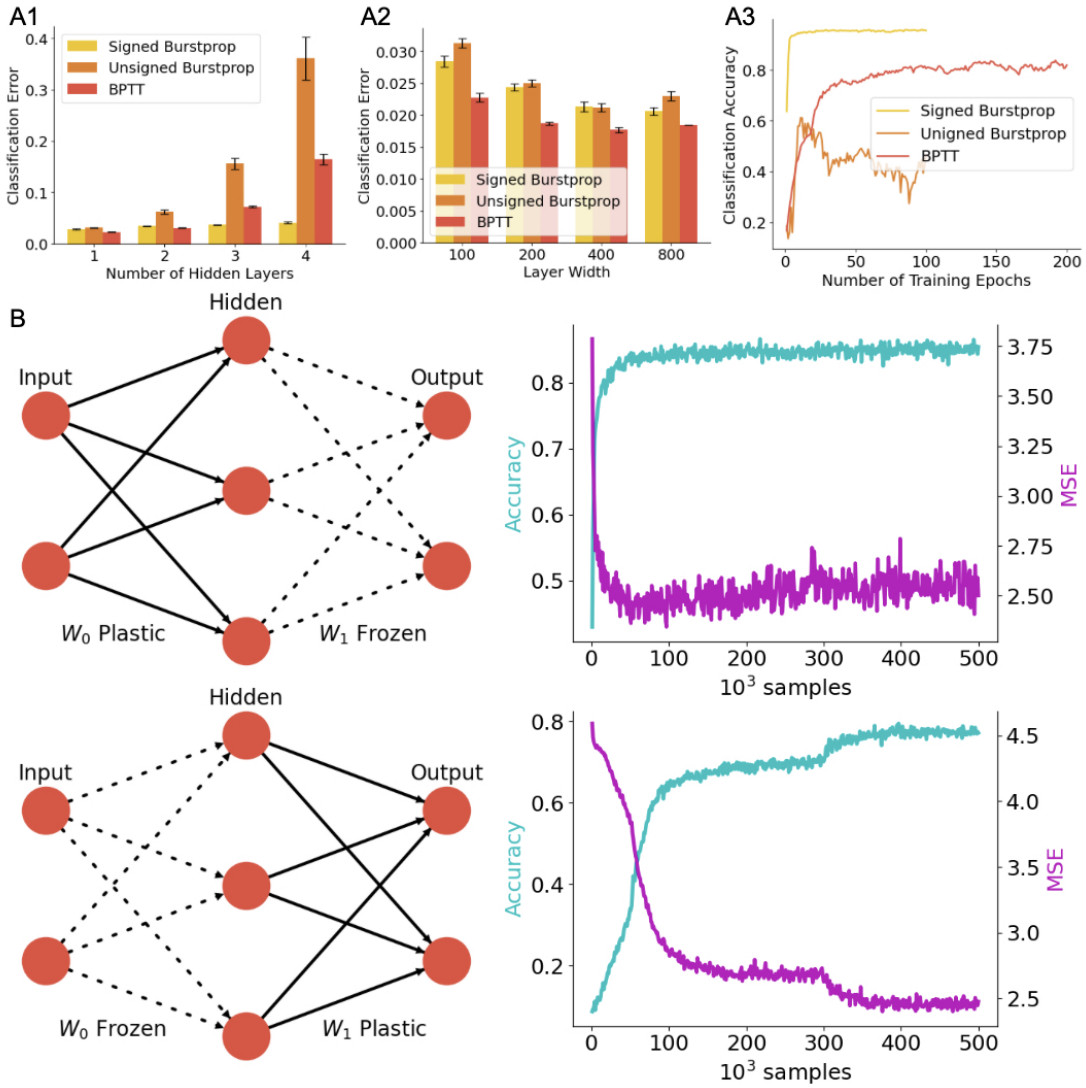


Figure 4.1: **Burstprop coordinates multilayer learning to solve MNIST.** A. Comparing test classification error on spiking MNIST dataset for Signed Burstprop, Unsigned Burstprop, and BPTT with symmetric feedback for networks with an increasing number of hidden layers, 100 neurons per layer (A1), and networks with increasing layer width, 1 hidden layer (A2). Burstprop algorithms were trained for 100 epochs and BPTT was trained for 200 epochs. A3. Example learning curves for each algorithm on a network of 4 hidden layers, each with 100 neurons. Error bars represent the standard deviation of minimum test error for 10 simulations. B. Signed Burstprop test accuracy and mean square error (MSE) per $10^3$ samples of a single hidden layer neural network of 100 hidden neurons. Row 1: $W_0$ learns and $W_1$ is frozen. Row 2: $W_1$ learns and $W_0$ is frozen.

To test the performance of our SNNs, we have trained both signed and unsigned versions of networks using symmetric feedback, varying width and depth on the rate encoded MNIST data (Fig. 4.1). We found by inspecting the resulting test-set

errors that both signed and unsigned models were able to effectively train spiking neural networks with one layer with the signed model achieving $97.2 \pm 0.1\%$ and $96.8 \pm 0.1\%$ test classification accuracy respectively on a network with a single hidden layer of 100 neurons. For deeper networks, the signed model outperformed the unsigned one (Fig. 4.1A1), with the signed model achieving $95.9 \pm 0.1\%$ accuracy compared to $64 \pm 1\%$ accuracy for the unsigned model on a 4 hidden layer network. We did not observe an increase but rather a decrease in performance with the number of hidden layers. While there was no performance gain with depth, increased width from 100 to 800 neurons was associated with an increase in performance (Fig. 4.1A2). Our algorithms could thus successfully train SNNs to perform MNIST with a performance that increases with layer width. Furthermore, we conclude that the signed burst model helps the algorithm to remain successful in deep networks.

To contextualize the learning ability, we compared the test-set prediction accuracy of our local learning algorithm to a state-of-the-art, non-local, BPTT-based learning method [42]. We found that in shallow networks, BPTT training applied on the same SNN architecture provided slightly better performance, with $97.7 \pm 0.1\%$ accuracy on a network with a single hidden layer of 100 neurons. This improvement was maintained across the different network widths (Fig. 4.1A2). Networks trained with BPTT showed a decrease in performance with network depth ($84 \pm 1\%$ accuracy on a 4 hidden layer network), an observation that echoes previous tests [82]. Because the choice of hyperparameters is likely to influence the comparison of these three algorithms, we conservatively conclude that SNNs with local burst-dependent plasticity rules can achieve good performance on MNIST, even with a small number of neurons and with multiple layers.

That the network performance does not increase with network depth may suggest that the algorithm is only training the output weights. To investigate this, and to determine whether the algorithm is effectively assigning credit to earlier layers, we examine how freezing either the output weights or the input weights affects learning in a single hidden layer network. If learning were limited to the output layer, freezing plasticity in this layer should result in the network being unable to learn. Instead, what we see is that the network remains able to learn effectively when the output layer is frozen (Fig. 4.1B). To take another approach, forcing the plasticity to only occur in the output weights should preserve the performance if credit signals were not reaching the lower-order connection. We found that freezing the output weights has a smaller effect on learning capabilities than freezing the input weights. Freezing the input weights made learning much slower (Fig. 4.1). Together, our burst-dependent SNN can successfully propagate credit-carrying information to neurons and accurately steer plasticity beyond the output layer. This suggests that our algorithm is effectively approximating backpropagation in the way it assigns errors.

## 4.2 Feedback Alignment

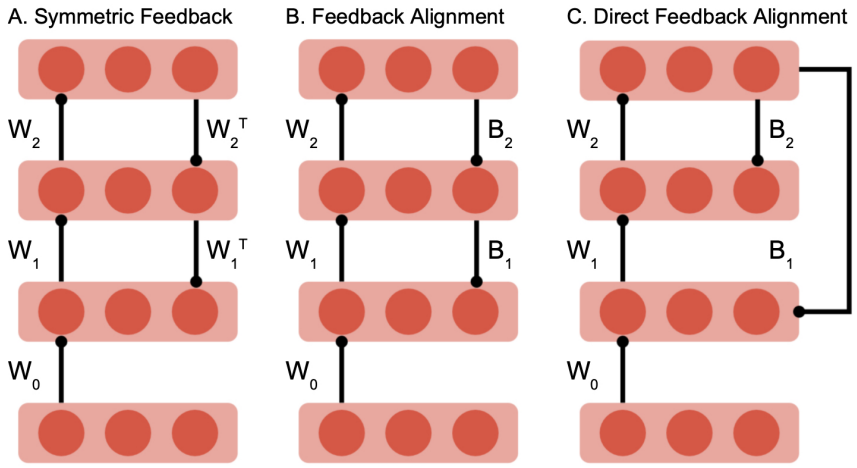


Figure 4.2: **Feedback Connectivity Schematics:** A. Symmetric connectivity: the strength of feedback synapses connecting a higher-order neuron to a lower-order neuron to higher-order neurons are proportional to the synapses connecting lower-order neurons to higher-order neurons. B. Feedback alignment: feedback synapses are static and randomly initialized. C. Direct feedback alignment: Feedback synapses are static and randomly initialized, connecting output neurons to all hidden neurons.

While we have shown that Burstprop can successfully train an SNN on MNIST, we did this by enforcing symmetry between feedforward and feedback weights. To allow for neuromorphic learning, and to address biological plausibility, our algorithm must remain effective with weight-transport-free learning. We thus tested the performance of Burstprop on our spiking MNIST dataset using feedback alignment (FA) and direct feedback alignment (DFA), two algorithms that avoid the weight-transport problem by using random but fixed feedback weights (see Methods and Figure 4.2). If Burstprop can learn from the data while using random but static feedback weights, this would suggest that weight transport is not necessary for learning in SNNs. We found that both signed (Figure 4.3B1) and unsigned (Figure 4.3B2) versions of the algorithm can achieve good performance, achieving $95.6 \pm 0.1\%$ accuracy and $85 \pm 1\%$ accuracy respectively on a 4 hidden layer network with feedback alignment. The decrease in performance with the number of layers was lessened by switching to FA for the unsigned burst model but unaltered for the signed burst model.

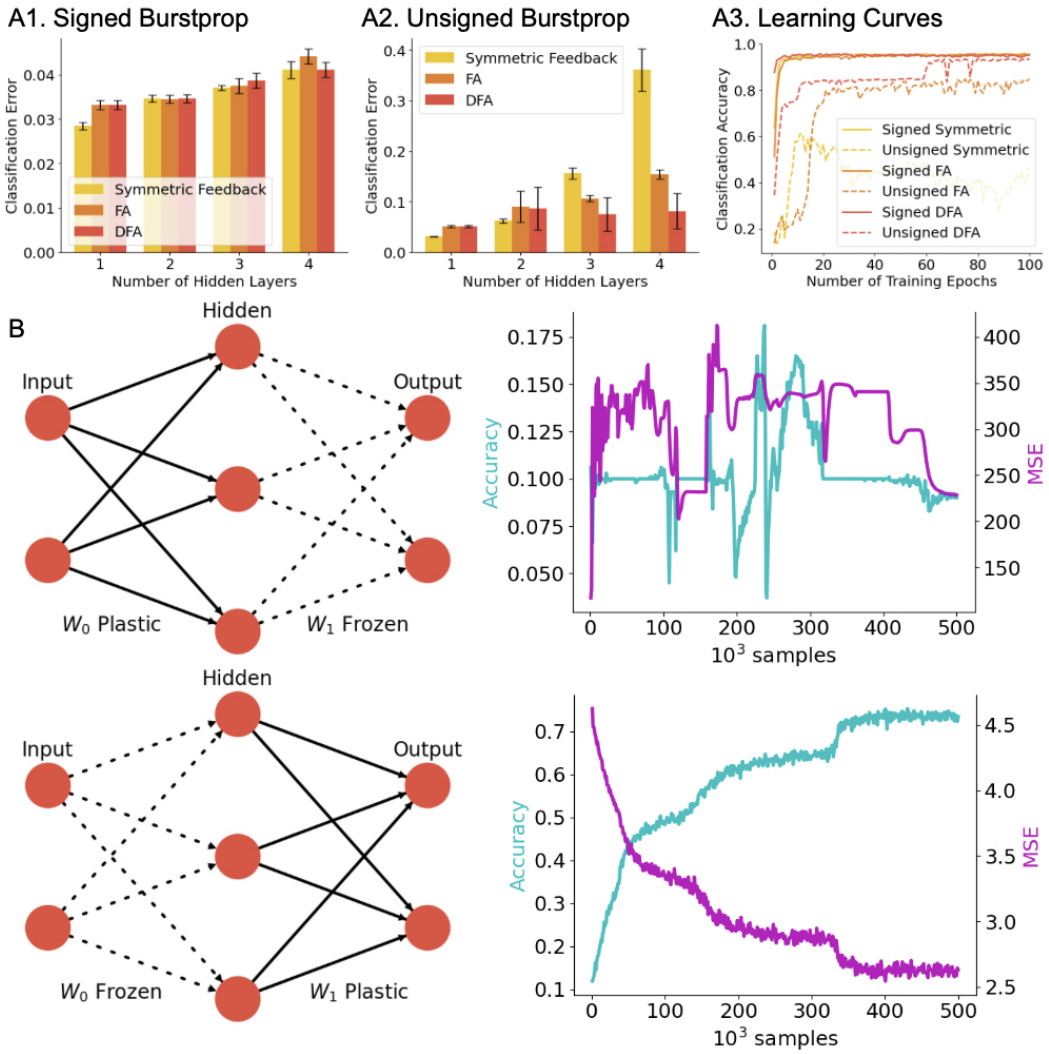


Figure 4.3: **Burstprop learns with feedback alignment methods:** A. Comparing test classification error on spiking MNIST dataset for different feedback error transportation methods with the Signed Burstprop (B1) and Unsigned Burstprop (B2) algorithms with increasing number of hidden neurons, 100 neurons per layer. Each trained for 100 epochs. A3. Example learning curves for each algorithm on a network of 4 hidden layers, each with 100 neurons. B. Signed Burstprop with FA test accuracy and mean square error (MSE) per $10^3$ samples of a single hidden layer neural network of 100 hidden neurons. Row 1: $W_0$ learns and $W_1$ is frozen. Row 2: $W_1$ learns and $W_0$ is frozen.

Next, we again investigated how freezing either the first or second layer impacts learning. In principle, learning the set of weights should not be possible if the higher-order layers have not had a chance to learn and align with the feedback [66]. Accordingly, if we freeze the output weights, learning should be strongly affected. This is indeed what we see (Fig. 4.3B), whereby freezing the output weights in an FA scenario completely blocked learning. Thus using random but fixed feedback weights can implement weight-transport-free learning in SNNs using local learning rules.

## 4.3 Low-Resolution Synaptic Weights

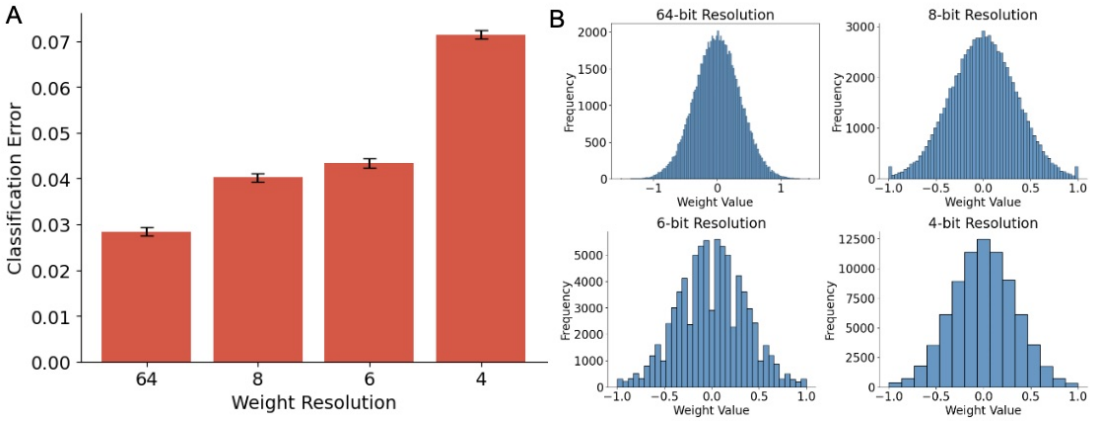


Figure 4.4: **Burstprop is robust to low-resolution synaptic weights.** A. Comparing test classification error on spiking MNIST dataset for networks of a single hidden layer of 100 neurons, trained with Signed Burstprop, with varying weight resolutions. B. Initial weight distributions of input weights for varying weight resolutions.

An important consideration for implementing SNNs in neuromorphic hardware is the number of distinct values that synaptic weights can take [84]. Spiking neuromorphic hardware may only be able to support small bit resolutions for their weights, in the range of 4 to 8 bits [84]. Here we investigate the impact of decreased synaptic weights on learning performance. Figure 4.4 shows that learning performance decreases with weight resolution as expected. The decrease in performance, however, is not substantial. With 6-bit resolution, the algorithm retains a 95.6% classification accuracy and with 4-bit resolution, a 92.8% classification accuracy. This result indicates that Burstprop is robust to low bit resolution in the synaptic weights.

# Chapter 5

# Conclusion and Future Work

## 5.1 Summary of Results

From the positive results of training Burstprop on the rate-encoded MNIST dataset, we conclude that we have achieved our goal of improving the Burstprop algorithm to scale from the spiking XOR problem to a more complex learning task. Moreover, Burstprop achieved comparable performance to BPTT, which is notable because it does so without relying on nonlocal learning. By experimenting with freezing weight layers during learning, we conclude that the algorithm is able to accurately perform credit assignment beyond the output layer. The experiments training the algorithm with FA and DFA show that Burstprop can successfully perform weight-transport-free learning. The results also show the algorithm is robust to low-resolution synaptic weights [24, 85].

## 5.2 Conclusion

The aim of this work was to develop an algorithm suitable for neuromorphic learning. This goal was achieved by incorporating features of biological learning to develop a local learning algorithm for SNNs, something that had previously never been accomplished. These positive results open the door for the principles of this algorithm to be used in a hardware implementation of neuromorphic learning, addressing the critical challenge of energy efficiency in AI systems.

Furthermore, the development of Burstprop as a viable method for neuromorphic learning has broader implications for the field of artificial intelligence and neuromorphic engineering. The ability to perform supervised learning directly on neuromorphic hardware could lead to significant energy savings and open up new applications for AI in energy-constrained environments, including applications in edge computing, where low-power devices are required to perform complex tasks locally without relying on cloud-based resources [2].

Additionally, the robustness of Burstprop to low-resolution synaptic weights suggests that it can be implemented on a wider range of neuromorphic hardware platforms, including those with limited precision capabilities. This increases the versatility and potential for adoption of the algorithm [24].

In summary, the results of this study demonstrate the potential of Burstprop as an efficient, biologically inspired learning algorithm for SNNs, paving the way for advancements in neuromorphic computing and energy-efficient AI.

## 5.3 Limitations and Future Work

We have shown that feedback alignment methods can be used effectively with SNNs using Burstprop to solve weight transport between feedforward and feedback weights. Our algorithm, however, requires some symmetry between two sets of feedback weights. These symmetric pathways are required to transmit positive and negative error signals. In our experiments, we have been enforcing this symmetry through weight transport; however, there are many methods that could be implemented to align these weights. Further work must be done to find a suitable solution to address this problem. For example, investigating learned feedback methods [86] or biologically inspired mechanisms for weight alignment could provide more insights.

Another weakness of our algorithm is that its fitting capabilities were not improved by increasing network depth. This is common for spiking neural networks, and it is unclear whether this is related to the input encoding or could be a fundamental limitation of the algorithm. While neural networks often rely on deep structures to fit complex functions, even a single-layer spiking neural network is a universal function approximator, so this may not be an important issue. Future research should explore the impact of different input encoding schemes and network architectures to fully understand this limitation [30].

One challenge in experimenting with this algorithm was finding suitable network hyperparameters. The algorithm requires the specification of many parameters pertaining to neuron dynamics, error generation, and learning. The parameters used for the experimental results were satisfactory for generating positive results though certainly not optimal. More work is required to optimize the network parameters for performance.

An important limitation of our results worth mentioning is that we limited our experimentation to the MNIST dataset. MNIST is a fairly simple benchmark problem for supervised learning algorithms and many non-neural network algorithms reach high test accuracy on this task. For instance, a linear classifier can achieve 7.6% error [87], a random forest algorithm can achieve 2.4% error, and an SVM can achieve 0.56% error [88]. While our specific spiking version of the dataset is not as simple of a problem, it is still solving the same underlying problem. The goal of this work was not to achieve state-of-the-art performance on this benchmark but to show that Burstprop can be applied to solve a complex

learning task. Further experimentation with more complex problems is required to see how the algorithm scales.

Future work will focus on addressing the limitations identified in this study, such as optimizing network hyperparameters and exploring more complex datasets beyond MNIST. Investigating alternative methods for aligning feedback weights without relying on symmetry will also be crucial for further improving the algorithm's effectiveness.

# Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[2] Catherine D. Schuman, Shruti R. Kulkarni, Maryam Parsa, J. Parker Mitchell, Prasanna Date, and Bill Kay. Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science*, 2:10–19, January 2022.

[3] Nitin Rathi, Indranil Chakraborty, Adarsh Kosta, Abhronil Sengupta, Aayush Ankit, Kaushik Roy, and Priyadarshini Panda. Exploring neuromorphic computing based on spiking neural networks: Algorithms to hardware. *ACM Computing Surveys*, 55(12):1–49, March 2023.

[4] Kaushik Roy, Akhilesh Jaiswal, and Priyadarshini Panda. Towards spike-based machine intelligence with neuromorphic computing. *Nature*, 575(7784):607–617, 2019.

[5] Giacomo Indiveri and Shih-Chii Liu. Memory and information processing in neuromorphic systems. *Proceedings of the IEEE*, 103(8):1379–1397, 2015.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[7] Matthieu Cord and Pádraig Cunningham. Supervised learning. In Matthieu Cord and Pádraig Cunningham, editors, *Machine Learning Techniques for Multimedia*, pages 21–49. Springer, 2023.

[8] Happiness Ugochi Dike, Yimin Zhou, Kranthi Kumar Deveeraseddy, and Qingtian Wu. Unsupervised learning based on artificial neural network: A review. In *2018 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1043–1048. IEEE, 2018.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[10] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 4(12):310–316, April 2020.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[12] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The elements of statistical learning: Data mining, inference, and prediction. *Springer Series in Statistics*, 2009.

[13] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[14] Léon Bottou. Large-scale machine learning with stochastic gradient descent. *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT)*, pages 177–186, 2010.

[15] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[16] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, John D. Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*, 2017.

[17] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[18] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Shweta H Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Sumeet Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[19] Johannes Schemmel, Daniel Bruderle, Andreas Grubl, Michael Hock, Karlheinz Meier, and Sebastian Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1947–1950, 2010.

[20] Jiaxin Pei, Lei Deng, Sen Song, Mingsheng Zhao, Yi Zhang, Shuangchen Wu, Guangyu Wang, Zhixin Zou, Sen Wu, Weinan He, et al. Towards artificial general intelligence with hybrid tianjic chip architecture. *Nature*, 572(7767):106–111, 2019.

[21] Olga Krestinskaya, Alex Pappachen James, and Leon Ong Chua. Neuromemristive circuits for edge computing: A review. *IEEE Transactions on Neural Networks and Learning Systems*, 31(1):4–23, 2020.

[22] Jerald Yoo and Mahsa Shoaran. Neural interface systems with on-device computing: machine learning and neuromorphic architectures. *Current Opinion in Biotechnology*, 72:95–101, December 2021.

[23] Yoshua Bengio, Dong-Hyun Lee, Jorg Bornschein, Thomas Mesnard, and Zhouhan Lin. Towards biologically plausible deep learning. *arXiv preprint arXiv:1502.04156*, 2015.

[24] Alexandre Payeur, Jordan Guerguiev, Friedemann Zenke, Blake A. Richards, and Richard Naud. Burst-dependent synaptic plasticity can coordinate learning in hierarchical circuits. *Nature Neuroscience*, 24(7):1010–1019, 2021.

[25] Wulfram Gerstner and Werner M. Kistler. Spiking neuron models: Single neurons, populations, plasticity. 2002.

[26] Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, 2003.

[27] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. Neuronal dynamics: From single neurons to networks and models of cognition, 2014.

[28] Jason K. Eshraghian, Max Ward, Emre O. Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. Training spiking neural networks using lessons from deep learning. *Proceedings of the IEEE*, 2023.

[29] Wenzhe Guo, Mohammed E. Fouda, Ahmed M. Eltawil, and Khaled Nabil Salama. Neural coding in spiking neural networks: A comparative study for robust neuromorphic systems. *Frontiers in Neuroscience*, 15, March 2021.

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

[31] Jan Hahne, David Dahmen, Jannis Schuecker, Andreas Frommer, Matthias Bolten, Moritz Helias, and Markus Diesmann. Integration of continuous-time dynamics in a spiking neural network simulator. *Frontiers in Neuroinformatics*, 11, May 2017.

[32] Peter U. Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. *Proceedings of the 2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2015.

[33] Sander M. Bohte, Joost N. Kok, and Han La Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1-4):17–37, 2002.

[34] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks. *IEEE Signal Processing Magazine*, 36(6):61–63, 2019.

[35] Stephen Grossberg. Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science*, 11(1):23–63, 1987.

[36] Guo-Qiang Bi and Mu-Ming Poo. Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of Neuroscience*, 18(24):10464–10472, 1998.

[37] Will Greedy, Heng Wei Zhu, Joseph Pemberton, Jack Mellor, and Rui Ponte Costa. Single-phase deep learning in cortico-cortical networks. *Advances in Neural Information Processing Systems*, 35, 2022.

[38] Ruohong Zhou. A method of converting ann to snn for image classification. In *2023 IEEE 3rd International Conference on Neuromorphic Computing (ICNC)*. IEEE, 2023.

[39] Haoran Gao, Junxian He, Haibing Wang, Tengxiao Wang, Zhengqing Zhong, Jianyi Yu, Ying Wang, Min Tian, and Cong Shi. High-accuracy deep ann-to-snn conversion using quantization-aware training framework and calcium-gated bipolar leaky integrate and fire neuron. *Frontiers in Neuroscience*, 17, March 2023.

[40] Eric Hunsberger and Chris Eliasmith. Training spiking deep networks for neuromorphic hardware. *arXiv preprint arXiv:1611.05141*, 2016.

[41] Sebastian Schmitt, Johann Klähn, Guillaume Bellec, Andreas Grübl, Maurice Guettler, Andreas Hartel, Stephan Hartmann, Dan Husmann, Kai Husmann, Sebastian Jeltsch, et al. Neuromorphic hardware in the loop: Training a deep spiking network on the brainscales wafer-scale system. In *2017 international joint conference on neural networks (IJCNN)*, pages 2227–2234. IEEE, 2017.

[42] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks. *arXiv preprint arXiv:1901.09948*, 2019.

[43] Sumit Bam Shrestha and Garrick Orchard. Slayer: Spike layer error reassignment in time. *arXiv preprint arXiv:1810.08646*, 2018.

[44] Benjamin Cramer, Sebastian Billaudelle, Simeon Kanya, Aron Leibfried, Andreas Grübl, Vitali Karasenko, Christian Pehle, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, et al. Surrogate gradients for analog neuromorphic computing. *Proceedings of the National Academy of Sciences*, 119(4):e2109194119, 2022.

[45] Benjamin Ellenberger, Paul Haider, Jakob Jordan, Kevin Max, Ismael Jaras, Laura Kriener, Federico Benitez, and Mihai A. Petrovici. Backpropagation through space, time, and the brain. *arXiv preprint arXiv:2403.16933*, March 2024.

[46] Bernhard Nessler David Kappel and Wolfgang Maass. Stdp installs in winner-take-all circuits an online approximation to hidden markov model learning. *PLoS Computational Biology*, 10(3):e1003511, March 2014.

[47] Peter U. Diehl and Matthew Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, 9, August 2015.

[48] Robert Legenstein, Dejan Pecevski, and Wolfgang Maass. A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback. *PLOS Comput. Biol.*, 4:e1000180, 2008.

[49] R. V. Florian. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19:1468–1502, 2007.

[50] Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Functional requirements for reward-modulated spike-timing-dependent plasticity. *J Neurosci*, 30(40):13326–13337, 2010.

[51] Konrad P Körding and Peter König. Supervised and unsupervised learning with two sites of synaptic integration. *J Comput. Neurosci.*, 11(3):207–215, 2001.

[52] João Sacramento, Rui Ponte Costa, Yoshua Bengio, and Walter Senn. Dendritic cortical microcircuits approximate the backpropagation algorithm. In *Advances in Neural Information Processing Systems*, pages 8721–8732, 2018.

[53] Jordan Guergiuev, Timothy P Lillicrap, and Blake A Richards. Towards deep learning with segregated dendrites. *eLife*, 2017.

[54] Walter Senn, Dominik Dold, Mihai A. Petrovici, et al. A neuronal least-action principle for real-time learning in cortical circuits. *eLife*, v2, 2024. Reviewed Preprint, Revised by authors on April 12, 2024.

[55] Alex Reyes, Rafael Lujan, Andrej Rozov, Nail Burnashev, Peter Somogyi, and Bert Sakmann. Target-cell-specific facilitation and depression in neocortical circuits. *Nat. Neurosci.*, 1(4):279–285, 1998.

[56] Massimo Scanziani, Beat H Gähwiler, and Serge Charpak. Target cell-specific modulation of transmitter release at terminals from a single axon. *Proceedings of the National Academy of Sciences*, 95(20):12004–12009, 1998.

[57] John Beninger, Julian Rossbroich, Katalin Toth, and Richard Naud. Functional subtypes of synaptic dynamics in mouse and human. *Cell Reports*, 2024.

[58] ME Larkum, J Zhu, and B Sakmann. A new cellular mechanism for coupling inputs arriving at different cortical layers. *Nature*, 398:338–341, 1999.

[59] Zachary Friedenberger and Richard Naud. Dendritic excitability controls overdispersion. *Nature Computational Science*, 4:19–28, 2024. Published 27 December 2023.

[60] Katie C Bittner, Aaron D Milstein, Christine Grienberger, Sandro Romani, and Jeffrey C Magee. Behavioral time scale synaptic plasticity underlies ca1 place fields. *Science*, 357(6355):1033–1036, 2017.

[61] Johannes J. Letzkus, Bjorn M. Kampa, and Greg J. Stuart. Learning rules for spike timing-dependent plasticity depend on dendritic synapse location. *J. Neurosci.*, 26(41):10420–10429, October 2006.

[62] Lea Caya-Bissonnette, Richard Naud, and Jean-Claude Beique. Cellular substrate of eligibility traces during behavioral timescale synaptic plasticity. *BioRxiv*, 2023.

[63] Zachary Friedenberger, Emerson Harkin, Katalin Tóth, and Richard Naud. Silences, spikes and bursts: Three-part knot of the neural code. *The Journal of Physiology*, 601(23):5165–5193, December 2023.

[64] Alexandre Payeur, Jordan Guerguiev, Friedemann Zenke, Blake A Richards, and Richard Naud. Burst-dependent synaptic plasticity can coordinate learning in hierarchical circuits. *Nature Neuroscience*, pages 1–10, 2021.

[65] Mike Stuck and Richard Naud. Burstprop for learning in spiking neuromorphic hardware. In *Proceedings of the 2023 International Conference on Neuromorphic Systems (ICONS)*, pages 1–5, August 2023.

[66] Timothy P. Lillicrap, Daniel Cownden, Douglas B. Tweed, and Colin J. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7:13276, November 2016.

[67] Arild Nøkland. Direct feedback alignment provides learning in deep neural networks. *arXiv preprint arXiv:1609.01596*, 2016.

[68] Arash Samadi, Timothy P Lillicrap, and Douglas B Tweed. Deep learning with dynamic spiking neurons and fixed feedback weights. *Neural Comput.*, 29(3):578–602, 2017.

[69] Mohamed Akrout, Collin Wilson, Peter Humphreys, Timothy Lillicrap, and Douglas B Tweed. Deep learning without weight transport. *Advances in neural information processing systems*, 32, 2019.

[70] Li Ji-An and Marcus K Benna. Deep learning without weight symmetry. *arXiv preprint arXiv:2405.20594*, 2024.

[71] Kevin Max, Laura Kriener, Garibaldi Pineda García, Thomas Nowotny, Ismael Jaras, Walter Senn, and Mihai A. Petrovici. Learning efficient backprojections across cortical hierarchies in real time. *arXiv preprint arXiv:2212.10249*, February 2024.

[72] Bill Podlaski and Christian K Machens. Biological credit assignment through dynamic inversion of feedforward networks. *Advances in Neural Information Processing Systems*, 33:10065–10076, 2020.

[73] Sergey Bartunov, Adam Santoro, Blake Richards, Luke Marris, Geoffrey E Hinton, and Timothy Lillicrap. Assessing the scalability of biologically-motivated deep learning algorithms and architectures. *Advances in neural information processing systems*, 31, 2018.

[74] Theodore H Moskovitz, Ashok Litwin-Kumar, and LF Abbott. Feedback alignment in deep convolutional networks. *arXiv preprint arXiv:1812.06488*, 2018.

[75] Albert Jiménez Sanfiz and Mohamed Akrout. Benchmarking the accuracy and robustness of feedback alignment algorithms. *arXiv preprint arXiv:2108.13446*, 2021.

[76] Friedemann Zenke and Surya Ganguli. Superspike: Supervised learning in multilayer spiking neural networks. *Neural Comput.*, 30(6):1514–1541, 2018.

[77] Dongcheng Zhao, Yi Zeng, Tielin Zhang, Mengting Shi, and Feifei Zhao. Glsnn: A multi-layer spiking neural network based on global feedback alignment and local stdp plasticity. *Frontiers in Computational Neuroscience*, 14, November 2020.

[78] Xingrun Xing, Shitao Xiao, Yequan Wang, Zheng Zhang, Yiming Ju, Jiajun Zhang, Ziyi Ni, Siqi Fan, and Guoqi Li. Spikelm: Towards general spike-driven language modeling via elastic bi-spiking mechanisms. *arXiv preprint arXiv:2406.03287v1*, June 2024.

[79] J.F. Kolen and J.B. Pollack. Backpropagation without weight transport. In *Proceedings of the 1994 IEEE International Conference on Neural Networks (ICNN)*, pages 1375–1380. IEEE, 1994.

[80] Kendra S. Burbank. Mirrored stdp implements autoencoder learning in a network of spiking neurons. *PLOS Computational Biology*, 11(12):e1004566, December 2015.

[81] Alexandre Payeur, Jean-Claude Béïque, and Richard Naud. Classes of dendritic information processing. *Current Opinion in Neurobiology*, 58:78–85, October 2019.

[82] Julian Rossbroich, Julia Gygax, and Friedemann Zenke. Fluctuation-driven initialization for spiking neural network training. *arXiv preprint arXiv:2206.10226*, 2022.

[83] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[84] Lorenz K. Müller and Giacomo Indiveri. Rounding methods for neural networks with low resolution synaptic weights. *arXiv preprint arXiv:1504.05767*, April 2015.

[85] Will Greedy, Heng Wei Zhu, Joseph Pemberton, Jack Mellor, and Rui Ponte Costa. Single-phase deep learning in cortico-cortical networks. *arXiv preprint arXiv:2206.11769*, 2022.

[86] Timothy P. Lillicrap, Daniel Cownden, Douglas B. Tweed, and Colin J. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7:13276, 2016.

[87] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[88] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

# Appendix: Burstprop Python Code

```
"""
Burstprop Code written by Michael Stuck
"""

from numba import jit
from numba.typed import List, Dict
import numpy as np
import matplotlib.pyplot as plt
import math

import time as time
from torchvision import datasets

def normalize_data(data, constant_sum=100):

    # Normalize so that the sum of inputs is constant
    sums = data.sum(axis=1, keepdims=True)
    normalized_data = (data / sums) * constant_sum

    return normalized_data

data_path = '/data/mnist'

mnist_train = datasets.MNIST(data_path, train=True, download=True)
mnist_test = datasets.MNIST(data_path, train=False, download=True)

mnist_data_x = mnist_train.data.numpy() / 256  # Convert to float
↪  and scale to [0, 1]
mnist_data_y = mnist_train.targets.numpy()

# Apply the normalization transformation
mnist_data_x =
↪  normalize_data(mnist_data_x.reshape(mnist_data_x.shape[0], -1))

train_x = mnist_data_x[:50000]
train_y = mnist_data_y[:50000]

```

```
validate_x = mnist_data_x[50000:]
validate_y = mnist_data_y[50000:]

@jit(nopython=True)
def signed_burst_prob(V_d,beta=1):
    return np.tanh(V_d*beta)

@jit(nopython=True)
def calc_burst_prob(V_d,beta=1):
    return 1/(1+np.exp(-(beta*V_d)))

@jit(nopython=True)
def update_neurons(params,V_s,V_d,eligibility_trace,somatic_inputs
↪ ,dendritic_inputs):

    #detrmine number of neurons in layer
    num_neurons = len(V_s)

    #calculate new somatic and dendritic membrane potentials
    V_s += -(V_s/params.tau_mem)*params.dt + somatic_inputs
    V_d += -(V_d/params.tau_mem_d)*params.dt + dendritic_inputs

    #record somatic spikes
    spike_mask = V_s>params.V_threshold
    V_s[spike_mask] = params.V_reset
    spikes = np.where(spike_mask,1.,0.)

    if params.burst_type == 0:
        #determine burst (feedback spikes)
        burst_prob = np.zeros(num_neurons)
        burst_prob[spike_mask] =
        ↪ np.abs(signed_burst_prob(V_d[spike_mask],params.beta))
        burst_mask = np.random.rand(num_neurons)<burst_prob
        bursts = np.where(burst_mask,1.,0.)
        bursts *= np.sign(V_d)
        V_d[burst_mask] = params.V_reset

    elif params.burst_type == 1:
        #determine burst (feedback spikes)
        burst_prob = np.zeros(num_neurons)
        burst_prob[spike_mask] =
        ↪ calc_burst_prob(V_d[spike_mask],params.beta)
        burst_mask = np.random.rand(num_neurons)<burst_prob
        bursts = np.where(burst_mask,1.,0.)
        #V_d[spike_mask] = params.V_reset

    #increase eligibility trace by 1 for neurons that spiked
```

```
80      eligibility_trace +=
        ↪  -(eligibility_trace/params.tau_pre)*params.dt + spikes
81
82      return spikes,bursts
83
84  @jit(nopython=True)
85  def update_neurons_test(params,V_s,eligibility_trace,somatic_input
    ↪  s):
86
87      #detrmine number of neurons in layer
88      num_neurons = len(V_s)
89
90      #calculate new somatic and dendritic membrane potentials
91      V_s += -(V_s/params.tau_mem)*params.dt + somatic_inputs
92
93      #record somatic spikes
94      spike_mask = V_s>params.V_threshold
95      V_s[spike_mask] = params.V_reset
96      spikes = np.where(spike_mask,1.,0.)
97
98      #increase eligibility trace by 1 for neurons that spiked
99      eligibility_trace +=
        ↪  -(eligibility_trace/params.tau_pre)*params.dt + spikes
100
101     return spikes
102
103
104 @jit(nopython=True)
105 def reset_neurons(V_s,V_d,spikes,bursts,eligibility_trace,num_laye
    ↪  rs):
106     for i in range(num_layers):
107         #reset_neurons(V_s,V_d,elegibility_trace)
108         if i<num_layers-1:
109             V_s[i][:] = 0
110             V_d[i][:] = 0
111             spikes[i][:] = 0
112             bursts[i][:] = 0
113         #reset spikes
114         eligibility_trace[i][:] = 0
115
116 @jit(nopython=True)
117 def initialize_neuron_variable(params):
118
119     num_layers = len(params.num_neurons)
120
121     #neuron variables
122     V_s = List()
123     V_d = List()
```

```
124     spikes = List()
125     bursts = List()
126     eligibility_trace = List()
127 
128     for i in range(num_layers):
129         if i > 0:
130             V_s.append(np.zeros(params.num_neurons[i]))
131             V_d.append(np.zeros(params.num_neurons[i]))
132             spikes.append(np.zeros(params.num_neurons[i]))
133             bursts.append(np.zeros(params.num_neurons[i]))
134         eligibility_trace.append(np.zeros(params.num_neurons[i]))
135 
136     return V_s,V_d,eligibility_trace,spikes,bursts
137 
138 @jit(nopython=True)
139 def generate_sample(params,data_x,data_y,index):
140 
141     sample_y = data_y[index]
142     frequencies = data_x[index].flatten()*params.input_freq
143     probs = frequencies*params.dt
144     mask = np.random.rand(params.num_steps,784)
145     sample_x = np.where(mask<probs, 1., 0.)
146 
147     return sample_x,sample_y
148 
149 @jit(nopython=True)
150 def initialize_spike_rates(params):
151 
152     num_layers = len(params.num_neurons)
153 
154     #spike rate stats
155     layer_spike_rates = List()
156     layer_burst_rates = List()
157     layer_bp_rates = List()
158     layer_bm_rates = List()
159     layer_elegibility_trace = List()
160 
161     for i in range(num_layers-1):
162         layer_spike_rates.append(np.zeros(params.num_steps))
163         layer_burst_rates.append(np.zeros(params.num_steps))
164         layer_bp_rates.append(np.zeros(params.num_steps))
165         layer_bm_rates.append(np.zeros(params.num_steps))
166         layer_elegibility_trace.append(np.zeros(params.num_steps))
167     layer_elegibility_trace.append(np.zeros(params.num_steps))
168 
169     return layer_spike_rates,layer_burst_rates,layer_bp_rates,laye
    ↪  r_bm_rates,layer_elegibility_trace
170 
```

```
171 @jit(nopython=True)
172 def train_snn(params,weights_forward,weights_backward,data_x,data_
    ↪ y):
173 
174     num_layers = len(params.num_neurons)
175     pop_size = params.num_neurons[-1]//10
176 
177     #initialize neuron variables
178     V_s,V_d,eligibility_trace,spikes,bursts =
        ↪ initialize_neuron_variable(params)
179 
180     #initialize output rate and error
181     output_rates = np.zeros((params.num_neurons[-1]//pop_size))
182     error = np.zeros(params.num_neurons[-1])
183 
184     #layer rate stats
185     layer_spike_rates,layer_burst_rates,layer_bp_rates,layer_bm_ra
        ↪ tes,layer_elegibility_trace =
        ↪ initialize_spike_rates(params)
186 
187     #output rec
188     output_error_rec = np.zeros((2,params.num_steps))
189 
190     #set number of samples per epoch
191     if params.num_samples == 0:
192         num_samples = len(data_y)
193     else:
194         num_samples = min(params.num_samples,len(data_y))
195 
196     #gat sample indeces
197     indices = np.arange(num_samples); np.random.shuffle(indices)
198 
199     for n,index in enumerate(indices):
200 
201         #reset network variables
202         reset_neurons(V_s,V_d,spikes,bursts,eligibility_trace,num_
            ↪ layers)
203 
204         #reset output rates and error
205         output_rates[:] = 0
206         error[:] = 0
207 
208         #get data
209         sample_x, sample_y =
            ↪ generate_sample(params,data_x,data_y,index)
210 
211         #set target rates
```

```
212         target_rate = np.ones(params.num_neurons[-1]//pop_size)*pa
            ↪  rams.zero_freq*pop_size
213         target_rate[sample_y] = params.one_freq*pop_size
214
215         for t in range(params.num_steps):
216             for i in range(num_layers-1):
217                 if i == 0:#update first hidden layer
218
219                     #update input elegibility trace
220                     eligibility_trace[i] += -(eligibility_trace[i]
                        ↪  /params.tau_pre)*params.dt +
                        ↪  np.where(sample_x[t]==1,1.,0.)
221
222                     #hidden rate error
223                     hidden_error = 2*params.hidden_error_scale*(pa
                        ↪  rams.hidden_target-eligibility_trace[i+1]/
                        ↪  params.tau_pre)
224
225                     somatic_input = sample_x[t]@weights_forward[i]
226                     if params.feedback_type == 2: #dfa
227                         if params.burst_type == 0: #signed busts
228                             dendritic_input = (bursts[-1])@np.tran
                                ↪  spose(weights_backward[i]) +
                                ↪  hidden_error
229                         elif params.burst_type == 1: #bccn
230                             dendritic_input = (bursts[-1]-params.b
                                ↪  ase_prob*spikes[-1])@np.transpose(
                                ↪  weights_backward[i]) +
                                ↪  hidden_error
231                     else:
232                         if params.burst_type == 0: #signed busts
233                             dendritic_input = (bursts[i+1])@np.tra
                                ↪  nspose(weights_backward[i]) +
                                ↪  hidden_error
234                         elif params.burst_type == 1: #bccn
235                             dendritic_input = (bursts[i+1]-params.
                                ↪  base_prob*spikes[i+1])@np.transpos
                                ↪  e(weights_backward[i]) +
                                ↪  hidden_error
236
237                     spikes[i],bursts[i]  = update_neurons(params,V
                        ↪  _s[i],V_d[i],eligibility_trace[i+1],somati
                        ↪  c_input,dendritic_input)
238
239                 elif i < num_layers-2:#remaining hidden layers
240
241                     #hidden rate error
```

```
242                 hidden_error = 2*params.hidden_error_scale*(pa
                    ↪ rams.hidden_target-eligibility_trace[i+1]/
                    ↪ params.tau_pre)
243
244                 somatic_input = spikes[i-1]@weights_forward[i]
245                 if params.feedback_type == 2: #dfa
246                     if params.burst_type == 0: #signed busts
247                         dendritic_input = (bursts[-1])@np.tran
                            ↪ spose(weights_backward[i]) +
                            ↪ hidden_error
248                     elif params.burst_type == 1: #bccn
249                         dendritic_input = (bursts[-1]-params.b
                            ↪ ase_prob*spikes[-1])@np.transpose(
                            ↪ weights_backward[i]) +
                            ↪ hidden_error
250                 else:
251                     if params.burst_type == 0: #signed busts
252                         dendritic_input = (bursts[i+1])@np.tra
                            ↪ nspose(weights_backward[i]) +
                            ↪ hidden_error
253                     elif params.burst_type == 1: #bccn
254                         dendritic_input = (bursts[i+1]-params.
                            ↪ base_prob*spikes[i+1])@np.transpos
                            ↪ e(weights_backward[i]) +
                            ↪ hidden_error
255
256                 spikes[i],bursts[i] = update_neurons(params,V_
                    ↪ s[i],V_d[i],eligibility_trace[i+1],somatic
                    ↪ _input,dendritic_input)
257
258             else:#output layer
259
260                 somatic_input = spikes[i-1]@weights_forward[i]
261                 dendritic_input = error
262
263                 spikes[i],bursts[i]= update_neurons(params,V_s
                    ↪ [i],V_d[i],eligibility_trace[i+1],somatic_
                    ↪ input,dendritic_input)
264
265                 #calculate population output rates
266                 output_rates -=
                    ↪ (output_rates/params.tau_avg)*params.dt
267                 for p in range(pop_size):
268                     output_rates +=
                        ↪ spikes[-1][p::pop_size]/params.tau_avg
269
270                 #calculate output error (teaching signal)
```

```
271                     error = np.repeat(2*params.error_scale*(target
                        ↪  _rate-output_rates),pop_size)
272
273                 #weight update
274                 if params.burst_type == 0:
275                     condition = bursts[i]!=0
276                     weights_forward[i][:,condition] +=
                        ↪  params.eta*(np.outer(eligibility_trace[i],
                        ↪  bursts[i][condition]))
277                 elif params.burst_type == 1:
278                     condition = spikes[i]==1
279                     weights_forward[i][:,condition] +=
                        ↪  params.eta*(np.outer(eligibility_trace[i],
                        ↪  bursts[i][condition]-params.base_prob*spik
                        ↪  es[i][condition]))
280
281                 #update backward weights
282                 if i>0 and params.feedback_type == 0:
283                     weights_backward[i-1] = weights_forward[i].cop
                        ↪  y()*params.feedback_scale*params.num_neuro
                        ↪  ns[i]/(params.beta*params.num_neurons[i+1])
284
285                 #layer rates
286                 layer_spike_rates[i][t] += np.sum(spikes[i])
287                 layer_burst_rates[i][t] +=
                    ↪  np.sum(np.abs(bursts[i]))
288                 layer_bp_rates[i][t] += np.sum(bursts[i]>0)
289                 layer_bm_rates[i][t] += np.sum(bursts[i]<0)
290
291                 #elegibility trace rec
292                 layer_elegibility_trace[i][t] += np.sum(eligibilit
                    ↪  y_trace[i]/params.num_neurons[i])
293
294             layer_elegibility_trace[-1][t] += np.sum(eligibility_t
                ↪  race[-1]/params.num_neurons[-1])
295
296             output_error_rec[0,t] += output_rates[sample_y]
297             output_error_rec[1,t] += (np.sum(output_rates/1.)-outp
                ↪  ut_rates[sample_y])/(params.num_neurons[-1]-1)
298
299     #layer rates
300     for i in range(num_layers-1):
301         layer_spike_rates[i] *=
            ↪  1000/(num_samples*params.num_neurons[i+1])
302         layer_burst_rates[i] *=
            ↪  1000/(num_samples*params.num_neurons[i+1])
303         layer_bp_rates[i] *=
            ↪  1000/(num_samples*params.num_neurons[i+1])
```

```
304             layer_bm_rates[i] *=
                ↪  1000/(num_samples*params.num_neurons[i+1])
305
306         #elegibility trace
307         for i in range(num_layers):
308             layer_elegibility_trace[i] /= num_samples
309
310         #divide error by number of samples
311         output_error_rec /= num_samples
312
313         return layer_spike_rates, layer_burst_rates, layer_bp_rates,
            ↪  layer_bm_rates, layer_elegibility_trace, output_error_rec
314
315     @jit(nopython=True)
316     def test_snn(params,weights_forward,data_x,data_y):
317
318         num_layers = len(params.num_neurons)
319         pop_size = params.num_neurons[-1]//10
320
321         #initialize neuron variables
322         V_s,V_d,eligibility_trace,spikes,bursts =
            ↪  initialize_neuron_variable(params)
323
324         error = np.zeros((params.num_neurons[-1]))
325
326         #track network performance
327         spike_counter = np.zeros(params.num_neurons[-1]//pop_size)
328         num_correct = 0
329         average_error = 0
330
331         if params.num_samples == 0:
332             num_samples = len(data_y)
333         else:
334             num_samples = min(params.num_samples,len(data_y))
335
336         for n,sample_idx in enumerate(range(num_samples)):
337
338             #reset network variables
339             reset_neurons(V_s,V_d,spikes,bursts,eligibility_trace,num_
                ↪  layers)
340
341             #get data
342             sample_x, sample_y =
                ↪  generate_sample(params,data_x,data_y,sample_idx)
343
344             #get event rate target
345             target_rate = np.ones(params.num_neurons[-1]//pop_size)*pa
                ↪  rams.zero_freq*pop_size
```

```
346             target_rate[sample_y] = params.one_freq*pop_size
347
348             output_rates = np.zeros((params.num_neurons[-1]//pop_size))
349
350             for t in range(params.num_steps):
351                 for i in range(num_layers-1):
352
353                     if i == 0:#update first hidden layer
354
355                         somatic_input = sample_x[t]@weights_forward[i]
356                         spikes[i] = update_neurons_test(params,V_s[i],
                            ↪  eligibility_trace[i+1],somatic_input)
357
358                     elif i < num_layers-2:#remaining hidden layers
359
360                         somatic_input = spikes[i-1]@weights_forward[i]
361                         spikes[i] = update_neurons_test(params,V_s[i],
                            ↪  eligibility_trace[i+1],somatic_input)
362
363                     else:#output layer
364
365                         somatic_input = spikes[i-1]@weights_forward[i]
366                         spikes[i]= update_neurons_test(params,V_s[i],e
                            ↪  ligibility_trace[i+1],somatic_input)
367
368                         output_rates -=
                            ↪  (output_rates/params.tau_avg)*params.dt
369                         for p in range(pop_size):
370                             output_rates +=
                                ↪  spikes[-1][p::pop_size]/params.tau_avg
371                             spike_counter += spikes[-1][p::pop_size]
372
373                         average_error +=
                            ↪  np.sum(params.error_scale*(target_rate-out
                            ↪  put_rates)**2)/params.num_neurons[-1]
374
375             #determine classification
376             guess = np.argmax(spike_counter)
377             if guess == sample_y:
378                 num_correct += 1
379             spike_counter[:] = 0
380
381         accuracy = num_correct/num_samples
382         average_error = average_error/(num_samples*params.num_steps)
383
384         return accuracy, average_error
385
386     @jit(nopython=True)
```

```
387 def angle(u, v):
388   return np.arccos(u.dot(v)/(np.linalg.norm(u)*np.linalg.norm(v)))
389
390 def initialize_weights(params):
391
392     num_neurons = params.num_neurons
393     num_layers = len(num_neurons)
394
395     #set initial weights
396     weight_scale = 10
397     mean = 0
398
399     std = np.zeros(num_layers)
400
401     for i in range(num_layers):
402         std[i] = weight_scale/np.sqrt(num_neurons[i])
403
404     weights_forward = List()
405     weights_backward = List()
406
407     #random forward weights
408     for i in range(num_layers-1):
409         weights_forward.append(np.random.normal(mean,std[i],num_ne
        ↪  urons[i]*num_neurons[i+1]).reshape(num_neurons[i],num_
        ↪  neurons[i+1]))#weigths input to hidden
        ↪  layer
410
411     for i in range(num_layers-2):#no backward weigths on first
      ↪  hidden layer
412         if params.feedback_type == 0:
413             #copy forward weights
414             weights_backward.append(weights_forward[i+1].copy()*pa
            ↪  rams.feedback_scale*num_neurons[i+1]/(params.beta*
            ↪  num_neurons[i+2]))
415         elif params.feedback_type == 1:
416             #random feedback weigths within 90 degress of forward
              ↪  weights
417             weights_backward.append(np.random.normal(mean,std[i+1]
              ↪  ,num_neurons[i+1]*num_neurons[i+2]).reshape(num_ne
              ↪  urons[i+1],num_neurons[i+2])*params.feedback_scale
              ↪  *num_neurons[i+1]/(params.beta*num_neurons[i+2]))
418             while math.degrees(angle(weights_forward[i+1].flatten(
              ↪  ),weights_backward[i].flatten())) >
              ↪  90:
```

```
419                     weights_backward[-1] =
                        ↪  np.random.normal(mean,std[i+1],num_neurons[i+1
                        ↪  ]*num_neurons[i+2]).reshape(num_neurons[i+1],n
                        ↪  um_neurons[i+2])*params.feedback_scale*num_neu
                        ↪  rons[i+1]/(params.beta*num_neurons[i+2])
420             else:
421                 #direct feedback weigths
422                 if i == num_layers-3:
423                     #for output layer
424                     #random feedback weigths within 90 degress of
                        ↪  forward weights
425                     weights_backward.append(np.random.normal(mean,std[
                        ↪  i+1],num_neurons[i+1]*num_neurons[-1]).reshape
                        ↪  (num_neurons[i+1],num_neurons[-1])*params.feed
                        ↪  back_scale*num_neurons[-2]/(params.beta*num_ne
                        ↪  urons[-1]))
426                     while math.degrees(angle(weights_forward[i+1].flat
                        ↪  ten(),weights_backward[i].flatten())) >
                        ↪  90:
427                         weights_backward[-1] = np.random.normal(mean,s
                            ↪  td[i+1],num_neurons[i+1]*num_neurons[-1]).
                            ↪  reshape(num_neurons[i+1],num_neurons[-1])*
                            ↪  params.feedback_scale*num_neurons[-2]/(par
                            ↪  ams.beta*num_neurons[-1])
428                 else:
429                     #direct feedback weigths, within 90 degrees for
                        ↪  weights to output
430                     weights_backward.append(np.random.normal(mean,std[
                        ↪  i+1],num_neurons[i+1]*num_neurons[-1]).reshape
                        ↪  (num_neurons[i+1],num_neurons[-1])*params.feed
                        ↪  back_scale*num_neurons[-2]/(params.beta*num_ne
                        ↪  urons[-1]))
431
432     return weights_forward,weights_backward
433
434 from collections import namedtuple
435
436 Parameters = namedtuple('Parameters',[
437     #neuron parameters
438     'V_threshold',
439     'V_reset',
440     'tau_mem',
441     'tau_mem_d',
442     'tau_pre',
443     'tau_avg',
444     'base_prob',
445
446     #network parameters
```

```
447         'num_neurons',
448 
449         #training parameters
450         'input_freq',
451         'zero_freq',
452         'one_freq',
453         'hidden_target',
454         'eta',
455         'beta',
456         'error_scale',
457         'hidden_error_scale',
458         'feedback_scale',
459         'dt',
460         'num_steps',
461         'num_epochs',
462         'num_samples',
463 
464         #training type
465         'feedback_type',
466         'burst_type'
467 
468     ])
469 
470     params = Parameters(
471         #neuron parameters
472         V_threshold = 1,
473         V_reset = 0,
474         tau_mem = 10e-3,
475         tau_mem_d = 10e-3,
476         tau_pre = 10e-3,
477         tau_avg = 10e-3,
478         base_prob = 0.5,
479 
480         #network parameters
481         num_neurons = np.array([784,100,10]),
482 
483         #training parameters
484         input_freq = 200,
485         zero_freq = 20,#per neuron target 0
486         one_freq = 200,#per neuron target 1
487         hidden_target = 20,
488         eta = 1e-4,
489         beta = 1,
490         error_scale = 1e-3,
491         hidden_error_scale = 1e-5,
492         feedback_scale = 1,
493         dt = 1e-3,
494         num_steps = 100,
```

```
    num_epochs = 100,
    num_samples = 1000,

    #training type
    feedback_type = 1, # 0 = symmetric, 1 = fa, 2 = dfa
    burst_type = 0 # 0 = signed, 1 = bccn
)

print(params)

weights_forward,weights_backward = initialize_weights(params)

errors = []
accuracies = []

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Increase font size
plt.rcParams.update({'font.size': 20})

# Plot distribution
sns.histplot(weights_forward[0].flatten(), kde=False)
plt.title(f'{64}-bit Resolution')
plt.xlabel('Weight Value')
plt.ylabel('Frequency')
plt.show()

# Remove top and right spines
sns.despine()

plt.show()

mean_value = np.mean(weights_forward[0])
std_value = np.std(weights_forward[0])

print(f"Mean = {mean_value:.4f}")
print(f"Standard Deviation = {std_value:.4f}")

import seaborn as sns

for i, weight_layer in enumerate(weights_forward):

    #Plot distribution
    sns.histplot(weight_layer.flatten(), bins=30, kde=True)
    plt.title(f'Layer {i} Weight Distribution')
    plt.xlabel('Weight Value')
```

```
543         plt.ylabel('Frequency')
544         plt.show()
545 
546         mean_value = np.mean(weight_layer)
547         std_value = np.std(weight_layer)
548 
549         print(f"Mean = {mean_value:.4f}")
550         print(f"Standard Deviation = {std_value:.4f}")
551 
552     import seaborn as sns
553 
554     for i, weight_layer in enumerate(weights_backward):
555 
556         #Plot distribution
557         sns.histplot(weight_layer.flatten(), bins=30, kde=True)
558         plt.title(f'Layer {i} Weight Distribution')
559         plt.xlabel('Weight Value')
560         plt.ylabel('Frequency')
561         plt.show()
562 
563         mean_value = np.mean(weight_layer)
564         std_value = np.std(weight_layer)
565 
566         print(f"Mean = {mean_value:.4f}")
567         print(f"Standard Deviation = {std_value:.4f}")
568 
569     start = time.time()
570     for i in range(params.num_epochs):
571         epoch_time = time.time()
572 
573         layer_spike_rates, layer_burst_rates, layer_bp_rates,
            ↪  layer_bm_rates, layer_elegibility_trace, output_error_rec =
            ↪  train_snn(params,weights_forward,weights_backward,train_x,
            ↪  train_y)
574         accuracy, average_error =
            ↪  test_snn(params,weights_forward,validate_x,validate_y)
575         accuracies.append(accuracy)
576         errors.append(average_error)
577 
578         print("Epoch =",len(accuracies),"Accuracy =",accuracy,"Error
            ↪  =",average_error,"Time
            ↪  =",(time.time()-epoch_time)/60,"CumTime
            ↪  =",(time.time()-start)/60)
579         print("Average Neuron Rates =",
            ↪  np.mean(np.array(layer_spike_rates),axis = 1))
```

```
580     print("Average Positive Burst Rate
        ↪  =",np.mean(np.array(layer_bp_rates),axis = 1),"Average
        ↪  Negative Burst Rate
        ↪  =",np.mean(np.array(layer_bm_rates),axis = 1))
581     print("Average Burst Fraction
        ↪  =",np.mean(np.array(layer_burst_rates),axis =
        ↪  1)/np.mean(np.array(layer_spike_rates),axis = 1))
582
583     time_steps = np.arange(params.num_steps)
584     for j in range(len(params.num_neurons)-1):
585         plt.plot(time_steps, layer_spike_rates[j], label=f'Layer
            ↪  {j + 1}')
586     plt.xlabel("Time (ms)")
587     plt.ylabel("Average Spike Rate (Hz)")
588     plt.legend()
589     plt.show()
590
591     for j in range(len(params.num_neurons)-1):
592         plt.plot(time_steps, layer_burst_rates[j], label=f'Layer
            ↪  {j + 1}')
593     plt.xlabel("Time (ms)")
594     plt.ylabel("Average Burst Rates (Hz)")
595     plt.legend()
596     plt.show()
597
598     colors = ['b', 'g', 'r', 'c', 'm']  # Define a list of colors
599
600     print("Solid = Positive, Dashed = Negative")
601     for j in range(len(params.num_neurons)-1):
602         color = colors[j % len(colors)]  # Choose color for layer
            ↪  j
603         plt.plot(time_steps, layer_bp_rates[j], color=color,
            ↪  label=f'Layer {j + 1}')
604         plt.plot(time_steps, layer_bm_rates[j], '--', color=color)
605
606     plt.xlabel("Time (ms)")
607     plt.ylabel("Average Positive/Negative Burst Rate (Hz)")
608     plt.legend()
609     plt.show()
610
611     for j in range(len(params.num_neurons)):
612         plt.plot(time_steps, layer_elegibility_trace[j],
            ↪  label=f'Layer {j + 1}')
613     plt.xlabel("Time (ms)")
614     plt.ylabel("Average Elegibility Trace")
615     plt.legend()
616     plt.show()
617
```

```
618     plt.plot(time_steps, output_error_rec[0],label = "1")
619     plt.plot(time_steps, output_error_rec[1],label = "0")
620     plt.xlabel("Time (ms)")
621     plt.ylabel("Average Output Rates")
622     plt.legend()
623     plt.show()
624
625     epochs = np.arange(len(accuracies))+1
626     fig,ax = plt.subplots()
627     ax.plot(epochs,accuracies,color="c",marker="o")
628     ax.set_ylabel("Accuracy",color="c")
629     ax2=ax.twinx()
630     ax2.plot(epochs,errors,color="m",marker="o")
631     ax2.set_ylabel("Error",color="m")
632     ax.set_xlabel("Epochs")
633     plt.show()
```