Supervised Learning in Spiking Neural Networks

M. B. Büller

TU Delft

Supervised Learning in Spiking Neural Networks

by

M. B. Büller

to obtain the degree of Master of Science
at the faculty of Aerospace Engineering,
Department of Control & Simulation
Delft University of Technology

Student number:	4166566	
Project duration:	April 08, 2019 – May 25, 2020	
Thesis committee:	Prof. dr. G. C. H. E. de Croon	TU Delft, supervisor
	MSc F. Paredes Vallés	TU Delft, supervisor
	Dr. O.A. Sharpanskykh	TU Delft
	Ir. C. de Wagter	TU Delft

TU Delft

Acknowledgments

First, I would like to thank my supervisors Guido de Croon and Federico Paredes Vallés. Not only did you provide invaluable guidance during my thesis, but you also inspired challenged me to look for creative and new solutions. My time with you has been very valuable and educational, and I am thankful for that.

I would also like to thank my dear friends and family for their unwavering support and patience. You have helped me through the emotional ups and downs during my thesis. I might not have always shown it, but know that I am grateful for it.

Most importantly, I would like to give a special thank you to my parents. You have always supported my choices in life, and you have made it possible for me to get to where I am now.

M. B. Büller
Heemstede, May 16^{th} *2020*

Abstract

Spiking neural networks are notoriously hard to train because of their complex dynamics and sparse spiking signals. However, in part due to these properties, spiking neurons possess high computational power and high theoretical energy efficiency. This thesis introduces an online, supervised, and gradient-based learning algorithm for spiking neural networks. It is shown how gradients of temporal signals that influence spiking neurons can be calculated online as an eligibility trace. The trace represents the temporal gradient as a single scalar value and is recursively updated at each consecutive iteration. Moreover, the learning method uses approximate error signals to simplify their calculation and make the error calculation compatible with online learning. In several experiments, it is shown that the algorithm can solve spatial credit assignment problems with short-term temporal dependencies in deep spiking neural networks. Potential approaches for improving the algorithm's performance on long-term temporal credit assignment problems are also discussed.

Besides the research on spiking neural networks, this thesis includes an in-depth literature study on the topics of neuromorphic computing and deep learning, as well as extensive evaluations of several learning algorithms for spiking neural networks.

Contents

Acknowledgments iii

Abstract v

List of Symbols ix

List of Acronyms xi

List of Figures xiii

List of Tables xv

1 Introduction 1
1.1 Motivation and Research Question . . . 2
1.2 Structure of This Work . . . 2

I Scientific Paper 5

II Literature Study 29

2 Deep Learning 31
2.1 Artificial Neural Networks . . . 31
2.1.1 Fundamentals . . . 31
2.1.2 Basic Architectures . . . 34
2.1.3 Neuromorphic Artificial Neural Networks . . . 36
2.1.4 Deep Learning Frameworks and Hardware . . . 37
2.2 Spiking Neural Networks . . . 38
2.2.1 Biological Background . . . 38
2.2.2 Neuron Models . . . 39
2.2.3 Learning Rules . . . 41
2.2.4 Supervised Learning Comparison . . . 45
2.2.5 STDP and Back-Propagation . . . 46
2.2.6 Neuromorphic Hardware and Software . . . 47

3 Reinforcement Learning 49
3.1 Basics . . . 49
3.2 Credit Assignment Problem . . . 52
3.3 SNN and RL . . . 53

4 Literature Synthesis 55
4.1 Neuromorphic Computing . . . 55
4.2 Reinforcement Learning . . . 56
4.3 Designing a Learning Rule . . . 56

III Preliminary Experiments 59

5 Methodology 61
5.1 Datasets . . . 61
5.2 Event Discretization Kernel . . . 62
5.3 Phased LSTM . . . 64
5.4 Spike Layer Error Reassignment in Time . . . 66
5.5 Reward Modulated Spike Timing Dependent Plasticity . . . 68

6 Experimental Results **71**
6.1 Even Discretization Kernel . . . 71
6.2 Phased LSTM . . . 72
6.3 Spike Layer Error Reassignment in Time . . . 74
6.4 RSTDP for classification . . . 77

7 Experiments Discussion **81**
7.1 Datasets . . . 81
7.2 Artificial Neural Networks . . . 81
7.3 Spiking Neural Networks . . . 82

Bibliography **85**

A Results for SLAYER Experiments **93**

List of Symbols

Greek Symbols

γ	Learning rate
η	Activation of eligibility trace
Θ	Spiking threshold of spiking neuron
η, ϵ, κ	Kernel functions for the influence of spikes on a neuron's membrane voltage
τ	Time constant in a decaying function

Roman Symbols

A	Action of agent
$A_{+/-}$	Scaling factor of voltage difference due to a presynaptic or postsynaptic spike
b	Bias vector for the connections between two layer of a neural network
E	Error value for neural network performance
$f_{i/j}$	Spike during of discrete timestep
G	Expected return
I	Electrical current flowing into a spiking neuron membrane
$P_{i/j}$	Trace of presynaptic or postsynaptic spiking activity
R	Electrical resistance of of a spiking neuron membrane
r	Reward value
R	Reward of agent
S	State of agent
v	Membrane voltage of a spiking neural network neuron
v_{rest}	Resting membrane voltage of a spiking neural network neuron
δw	Weight update value
W	Weight matrix for the connections between two layer of a neural network
w_{init}	Initialization value of synaptic weight
x, y	Pixel coordinates in an image
y	Network output
$\hat{y}$	Desired network output
z	Eligibility trace

Subscripts

i	Postsynaptic neuron index
j	Presynaptic neuron index

Superscripts

f	Presynaptic spike index
n	Postsynaptic spike index

List of Acronyms

ANN	Artificial Neural Network
API	Application Programming Interface
ATIS	Asynchronous Time Based Image Sensor
BP-STDP	Backpropagation Spike-timing-Dependent Plasticity
BPTT	Backpropagation Through Time
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DVS	Dynamic Vision Sensor
FC	Fully Connected
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
IF	Integrate-and-Fire Neuron
LIF	Leaky Integrate-and-Fire Neuron
LSTM	Long Short-Term Memory
MARL	Multi-Agent Reinforcement Learning
MAVLab	Micro Air Vehicle Laboratory
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
MSTDPET	Reward Modulated Spike Timing Dependent Plasticity with Eligibility Traces
N-Caltech101	Neuromorphic Caltech101s
N-MNIST	Neuromorphic MNIST
NCars	Neuromorphic Cars
ODE	Ordinary Differential Equation
PDF	Probability Density Function
PLSTM	Phased LSTM
R-STDP	Reward-Modulated Spike-Timing-Dependent Plasticity
RAM	Random-Access Memory
ReLU	Rectified Linear Unit
ReSuMe	Remote Supervised Method

RL	Reinforcement Learning
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SHL	Supervised Hebbian Learning
SL	Supervised Learning
SLAYER	Spike Layer Error Assignment Rule
SNN	Spiking Neural Network
SOTA	State of the Art
SRM	Spike Response Model
STDP	Spike-Timing-Dependent Plasticity
TCA	Temporal Creddit Assignment
TPU	Tensor Processing Unit
UL	Unsupervised Learning

List of Figures

2.1 (a): Feed forward multilayer neural network. (b) Back-propagation in feed forward network. Both figures adapted from LeCun et al. [53] . . . 32
2.2 Sigmoid . . . 33
2.3 Hyperbolic tangent . . . 33
2.4 ReLU . . . 33
2.5 Convolution operation in neural network. Adapted from [101] and [77]. . . . 35
2.6 Single recurrent neuron. Adapted from Olah [75] . . . 35
2.7 Three step LSTM. The σ is a sigmoidal activation function, $\times$ is a pointwise multiplication, $+$ is a pointwise addition, $tanh$ is a hyperbolic tangent, an arrow is a vector transfer, C_t the cell state, h_t the hidden state, x_t the input activation. Adapted from Olah [75] . . . 36
2.8 Overview of the McCulloch Pitts neuron, it only receives and sends out binary signals. Adapted from Lagandula [50] . . . 38
2.10 Neuronal spike. Adapted from Gerstner [26] . . . 39
2.11 Spike-Timing-Dependent Plasticity (STDP) proportional weight scaling based on relative timing of pre- and post-synaptic spikes. Adapted from Sjostrom and Gerstner [96]. . . . 42

3.1 The agent-environment interaction in a Markov decision process. Adapted from Sutton and Barto [99] . . . 50

5.1 The saccades performed with the ATIS sensor for converting the MNIST dataset to an event-based format. Adapted from [70] . . . 62
5.2 An arbitrary spiking rate distribution for a set of six input neurons. . . . 63
5.3 Linear interpolation of an event its contribution to two adjacent discretization points, which is based on how close the event is to the discretization point in time. . . . 63
5.4 Example of transforming a linear indexing into regular, dimension based indexing for a three dimensional matrix. Each cell in the matrix is assigned a unique integer index. Adapted from $https://nl.mathworks.com/help/matlab/ref/ind2sub.html$ - accessed 10-8-2019 . . . 65
5.5 Training and testing accuracy of the Multi-Layer Perceptron (MLP) for Neuromorphic MNIST (N-MNIST) classification. Adapted from Shrestha and Orchard [94] . . . 66
5.6 Training and testing accuracy of the Convolutional Neural Network (CNN) for N-MNIST classification. Adapted from Shrestha and Orchard [94] . . . 66
5.7 SLAYER probability distribution function. The gradient along the curve is used as an approximate gradient of the non-differentiable spikes within a spiking neural network. Adapted from Shrestha and Orchard [94] . . . 67

6.1 Accuracy curves on training and testing datasets for (a) regular 34 layer ResNet trained on frame-based Caltech101 dataset and (b) event discretization kernel prepended to 34 layer ResNet network trained on Neuromorphic Caltech101s (N-Caltech101) dataset. . 72
6.2 Training and testing accuracies on the event-based N-MNIST dataset during training of the (a) Phased LSTM (PLSTM) and (b) Long Short-Term Memory (LSTM) network. . . . 73
6.3 Mean of gradients for three layer fully connected Spiking Neural Network (SNN) during training. . . . 74
6.4 Standard deviation of gradients for three layer fully connected SNN during training. . . . 74
6.5 Mean of gradients for three layer fully connected Artificial Neural Network (ANN) during training. . . . 75
6.6 Standard deviation of gradients for three layer fully connected ANN during training. . . . 75
6.7 Mean of gradients for four layer convolutional SNN during training. . . . 75
6.8 Standard deviation of gradients for four layer convolutional SNN during training. . . . 75

6.9 Mean of gradients for four layer convolutional ANN during training. 76
6.10 Standard deviation of gradients for four layer convolutional ANN during training. 76
6.11 Weight distributions after training of the XOR network with (a) regular Reward Modulated Spike Timing Dependent Plasticity with Eligibility Traces (MSTDPET), (b) threshold MSTDPET, and (c) with threshold MSTDPET and connections not split into predefined positive and negative weights. 78
6.12 Activity of the hidden neurons during a complete epoch, *after* training of the XOR network with (a) regular MSTDPET, (b) threshold MSTDPET, and (c) threshold MSTDPET and connections not split into predefined positive and negative weights. 78
6.13 Thresholds distribution after training of the XOR with (a) threshold MSTDPET, and (b) threshold MSTDPET and connections not split into predefined positive and negative weights. 78
6.14 Activity of the output neurons during a complete epoch, *after* training of the network (50 input, 30 hidden) with *threshold* MSTDPET. Figure (a) shows the output of the network with the best final performance at 89.5% accuracy, (b) shows the worst performing initialization with an accuracy of 21.5%. 79
6.15 Statistics for a 80 input, 50 hidden network that was trained using MSTDPET and achieved 12% accuracy. (a) The neuronal activity of the hidden layer, and (b) the weight distributions after training. 80
6.16 Statistics for a 30 input, 15 hidden network that was trained using MSTDPET and achieved 60% accuracy. (a) The neuronal activity of the hidden layer, and (b) the weight distributions after training. 80

A.1 Mean of gradients for 2 layer fully connected SNN during training. 94
A.2 Standard deviation of gradients for 2 layer fully connected SNN during training. 94
A.3 Mean of gradients for 2 layer fully connected ANN during training. 94
A.4 Standard deviation of gradients for 2 layer fully connected ANN during training. 94
A.5 Mean of gradients for four layer fully connected SNN during training. 95
A.6 Standard deviation of gradients for four layer fully connected SNN during training. . . . 95
A.7 Mean of gradients for four layer fully connected ANN during training. 95
A.8 Standard deviation of gradients for four layer fully connected ANN during training. . . . 95
A.9 Mean of gradients for five layer fully connected SNN during training. 96
A.10 Standard deviation of gradients for five layer fully connected SNN during training. 96
A.11 Mean of gradients for five layer fully connected ANN during training. 96
A.12 Standard deviation of gradients for five layer fully connected ANN during training. 96
A.13 Mean of gradients for three layer convolutional SNN during training. 97
A.14 Standard deviation of gradients for three layer convolutional SNN during training. 97
A.15 Mean of gradients for three layer convolutional ANN during training. 97
A.16 Standard deviation of gradients for three layer convolutional ANN during training. 97
A.17 Mean of gradients for five layer convolutional SNN during training. 98
A.18 Standard deviation of gradients for five layer convolutional SNN during training. 98
A.19 Mean of gradients for five layer convolutional ANN during training. 98
A.20 Standard deviation of gradients for five layer convolutional ANN during training. 98
A.21 Mean of gradients for six layer convolutional SNN during training. 99
A.22 Standard deviation of gradients for six layer convolutional SNN during training. 99
A.23 Mean of gradients for six layer convolutional ANN during training. 99
A.24 Standard deviation of gradients for six layer convolutional ANN during training. 99

List of Tables

2.1 Basic neural network components and their relational biases. From [8] 34

4.1 Learning rules for optimizing spiking neural networks and their strengths and weaknesses as derived from literature. 58

5.1 Advised standard hyper parameters for the Adam optimization algorithm 64
5.2 Layerwise number of neurons for the fully connected SNNs trained with the SLAYER algorithm 67
5.3 Layerwise parameters for the convolutional SNNs trained with the SLAYER algorithm. Following convention for describing a convolutional layer: k(kernel), s(stride), p(padding, always even), c(channels). 67
5.4 Network design and hyper parameters for the XOR and classification experiments. . . . 70
5.5 Layerwise number of neurons for the fully connected networks trained with the additive and multiplicative versions of the MSTDPET algorithm. 70

6.1 Performance comparison of 34 layer ResNet trained on the Caltech101 dataset and the 34 layer ResNet prepended with a learnable discretization kernel trained on the Neuromorphic Caltech101 dataset. 72
6.2 Training and testing results for the PLSTM and LSTM experiments performed on a subset of the N-MNIST dataset, the results from the original paper [70], and a reference experiment on frame-based MNIST data. Event-based = Ev., Frame-based = Fr. 73
6.3 Training and testing results for the three layer fully connected and four layer convolutional networks. The first two columns show the results for the networks trained on event-based data with the SLAYER algorithm, the last two columns for the network trained on frame-based data with regular backpropagation. 76
6.4 Best and worst classification performance for each network and learning rule combination. 79

A.1 Training and testing results for the two and five layer fully connected networks trained on event-based data with the SLAYER algorith, and trained on frame-based data with regular backpropagation. 93
A.2 Each network and the reference to the figures containing the statistics (mean and standard deviation) of its gradients during training. 93

1 Introduction

For a long time, humans have been fascinated by their own ability to learn. Young children start learning as soon as they are born, whereas experienced adults possess both the ability to reason about their environment or be guided by their intuition, almost without fault. A good example is an experienced driver that knows an accident is about to take place before it actually occurs. This ability to learn from experience and examples is something that many researchers strive to replicate, for example in robots.

The past decade saw a surge of popularity for learning algorithms that can infer patterns in data by showing it examples together with the desired outcome [49, 53], or even without telling the network what the desired outcome is [32, 34]. These self-learning methods mostly rely on deep learning, a family of machine learning methods that are roughly based on how biological brains process information by using a large network of interconnected neurons [33], called an Artificial Neural Network (ANN). While deep learning has been very successful in many pattern recognition tasks like classifying images, or speech recognition, most of the successful model required immense amounts of computational power. To illustrate, a small neural network, for modern standards, often requires at least one high-end Graphics Processing Unit (GPU) to be trained and used effectively. These devices use 250W of power[1] or more, making them highly ineffective for use in low-power applications like small scale robotics.

Combining the learning capabilities of ANNs with energy efficiency is where Spiking Neural Network (SNN) come in to play [26]. While ANNs are loosely inspired by animal brains, SNNs stay closer to the way animal brains work. Compared to an ANN, SNNs have the potential of being several orders of magnitude more energy-efficient because of the differences in the way they encode information. While ANNs communicate with scalar values and perform many large scale, synchronous matrix multiplications, SNNs send signals in the form of binary spikes where the time of arrival carries the information. Each neuron in a SNN performs this operation asynchronously from the others. It is this sparsity of the spiking signal that makes SNNs more energy efficient.

Due to their design, SNNs can be implemented on neuromorphic hardware. These are chips designed to leverage the sparse and asynchronous signals used in SNNs, and are a large reason for the reduction in energy usage compared to ANNs. As mentioned earlier, this makes SNNs ideal for low power applications like robotics. In addition, SNNs work together very well with event-based cameras [56]. These cameras detect changes in light intensity, instead of absolute values of light intensity. Information is communicated in the form of binary spikes, and each pixel works completely asynchronous from the other pixels. This is the same data format as used in SNNs. As a result, these cameras can have a temporal resolution of as little as 2 μs, and do not suffer from any motion blur. Even though these cameras are not a central topic to this thesis, their suitability for robotics applications and their synergy with SNNs is in itself a good reason for researching and developing SNNs.

Whereas there are theoretical advantages to using SNNs, they have not reached a point of maturity where they are truly useable in common applications. The main reason for this is that there currently exists no algorithm that can solve the credit assignment problem [64] in SNNs as efficiently as supervised backpropagation [86] can in ANNs. The credit assignment problem is the problem of determining

[1] https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/ - accessed on 21-02-2020

which of the previous actions lead to a certain outcome, and supervised learning is the process of finding patterns in data that strongly correlate with the desired output signal.

In contrast to optimizing ANNs where supervised learning is most common, SNNs are mostly trained with unsupervised learning algorithms. These algorithms work by extracting the most salient patterns present in the data. The type of method that is mostly used for training SNNs is called STDP [96]. It finds its origin in how animal brains perform optimization. A clear downside of unsupervised learning is that it cannot relate an arbitrary target signal to an arbitrary input signal like supervised learning can. Yet, supervised learning rules are a lot less common for optimizing SNNs, and the ones that do exist put several constraints on the network and still perform less than ANNs. Efficiently solving the credit assignment problem in SNNs is one of the largest outstanding challenges in the field of neuromorphic computing. Because of that, this thesis will focus on contributing to solving it.

To date, there are roughly two ways of solving the credit assignment problem, through supervised learning, or through Reinforcement Learning (RL). Reinforcement learning is best described as learning through trial-and-error [99]. By intelligently exploring the possible options and rewarding ones that lead to the desired result, one eventually can learn to solve complex problems. The reinforcement learning framework has roots in machine learning, but also in neuroscience [88]. Animal brains make extensive use of reinforcement learning by modulating its operations with for example neurotransmitters. Besides in the brain, many animals also make use of reinforcement learning at an organism level. One of the best examples is young children that learn through playing.

Supervised learning plays a crucial role in modern deep learning. Reinforcement learning is also a very active research area in the machine learning community, but also has many applications within neuroscience and biology. In addition, these are the two main methods for solving a credit assignment problem currently in use. Because of these reasons, this preliminary thesis will explore both methods as means to solve the credit assignment problem in spiking neural networks.

1.1. Motivation and Research Question

The Micro Air Vehicle Laboratory (MAVLab) at the Delft University of Technology solves fundamental technological challenges of micro air vehicles to maximize their utility and safety.[2] The most important factor in achieving this is to make the micro air vehicles autonomous, for which one of the main research fields is vision-based collision avoidance algorithms. For example, recent work towards this goal includes learning to predict distance using a monocular camera [16], or to estimate optical flow in an unsupervised manner using efficient neuromorphic computing [78].

Because SNNs have such a low (theoretical) power consumption they are ideal for low power applications like micro air vehicles, especially when paired with an event-based camera. In addition, there is a lot of research and development taking place in designing neuromorphic chips that can be used to efficiently simulate SNNs. Yet, the main thing that is missing is an efficient algorithm for solving the credit assignment problem in spiking neural networks. A supervised and online learning algorithm would be ideal for solving the credit assignment problem because it can be used while the SNN is in use.

According to this motivation, the main research question for this thesis is as follows:

Can a supervised learning algorithm be designed that can optimize spiking neural networks in an online manner?

1.2. Structure of This Work

This thesis is divided into three parts. The first part presents the main contribution of this thesis in the form of a scientific paper. The paper presents a new online, supervised, gradient-based learning method for SNNs. In addition to the design of the learning method, the paper presents the results of the experimental analysis of the learning method, as well as discussing several ways of improving it. Also, this paper can be read as a standalone document.

The second part consists of an in-depth review of literature on *deep learning*, *neurmorphic computing*, and *reinforcement learning*. Chapter 2 covers deep learning in ANNs and SNNs. For ANNs, this includes their basic functioning as well as applications in event-based vision processing. The chapter concludes with information on SNNs, treating their origin in biology, fundamentals of SNNs, and the

[2] http://mavlab.tudelft.nl/ - retrieved on 21-06-2019

current state of network and learning rule design. In chapter 3 the basics of Reinforcement Learning (RL) are presented with a focus on its relation and applicability to SNNs. Lastly, chapter 4 concludes this part with a synthesis of the literature presented in the preceding chapters and guidelines for working with and designing SNNs.

The third part describes experiments that are conducted as part of the preliminary thesis. The objective was to compare the performance of ANNs and SNNs when processing event-based data. Besides comparing ANNs and SNNs, the experiments also served as a practical introduction to neuromorphic computing. Chapter 5 starts with a description of the datasets that were used during the experiments and concludes with the methodology and setup for each experiment. Next, chapter 6 presents the results for each of the experiments. Finally, chapter 7 provides a discussion on the datasets, experimental setups, main results, and the conclusions that contribute towards the main thesis.

I

Scientific Paper

Online supervised learning in spiking neural networks through eligibility traces

Bas Büller*, Federico Paredes-Vallés†, Guido C.H.E. de Croon†
Control & Simulation Section, Control & Operations Department, Faculty of Aerospace Engineering
Delft University of Technology, Delft, The Netherlands

***Abstract*—Spiking neural networks are notoriously hard to train because of their complex dynamics and sparse spiking signals. However, in part due to these properties, spiking neurons possess high computational power and high theoretical energy efficiency. We introduce an online, supervised, and gradient-based learning algorithm for spiking neural networks. We show how gradients of temporal signals that influence spiking neurons can be calculated online as an eligibility trace. The trace represents the temporal gradient as a single scalar value and is recursively updated at each consecutive iteration. Moreover, the learning method uses approximate error signals to simplify their calculation and make the error calculation compatible with online learning. In several experiments, we show that our algorithm can solve spatial credit assignment problems with short-term temporal dependencies in deep spiking neural networks. We also address ways to improve the algorithm's performance on long-term temporal credit assignment problems.**

***Index Terms*—spiking neural networks, supervised learning, online learning, neuromorphic computing**

I. Introduction

Animals are capable of extraordinary feats that artificial systems have not rivaled yet. For example, humans are capable of seamlessly transitioning from walking to running without falling over, almost instantly react to a car that suddenly appears in the corner of their sight, or collaborate with thousands of other people to put a human on the moon. In part, these capabilities result from their remarkably powerful, yet efficient brain, nervous system, and senses. For instance, the human brain performs all its tasks by operating 10^{11} neurons with 10^{15} synaptic connections [1] at just 20W [2].

These astonishing capabilities and numbers can partly be attributed to the way neurons in the brain operate and how they communicate with each other. Biological brains are essentially a large number of interconnected neurons, called a *neural network*. By optimizing the strength of their connections, neural networks are capable of solving a wide range of tasks. The neurons itself are characterized by their *spiking function* [3]. If a neuron is excited enough by other neurons over time, it generates a spike and almost instantly resets itself. As a result of this function, biological neurons communicate through signals consisting of asynchronous, spatiotemporal, sparsely distributed spikes [3].

Biological brains have inspired research towards many types of artificial neural networks [4], and the research field is called *deep learning*. The most common neural networks use a simplified neuron model optimized for large-scale parallel computing. These networks have achieved many impressive results in the past decade [5]–[8] and are generally referred to as Artificial Neural Networks (ANNs). On the contrary, the field of neuromorphic computing researches artificial neural networks that closely mimic their biological counterpart. These are Spiking Neural Networks (SNNs), referring to the spikes that are the result of the spiking function of a neuron. Compared to the more common ANNs, SNNs have a higher computational power per neuron [3], [9], as well as lower theoretical energy requirements. As a result, SNNs are often considered the third generation of neural networks [9]. Besides their theoretical advantages, SNNs have received increased attention due to recent advances in SNN related hardware. Especially neuromorphic chips [10], [11] for highly efficient simulation of SNNs, and event-based cameras [12] that communicate with the same spike-based signals as SNNs.

One of the main aspects preventing SNNs from seeing widespread adoption is the absence of a useful and generally applicable learning algorithm. There are two main approaches to training SNNs, gradient-based methods, and correlation-based methods. The workhorse of deep learning, Backpropagation (BP) [13], forms the basis for the gradient-based methods. Gradient-based methods are state of the art for training SNNs [14], [15] and thus widely used. However, they come with several practical disadvantages. Firstly, BP is not directly applicable to SNNs because the spike generation function is non-differentiable. Secondly, BP requires completely separates network operations and updating the network's parameters, called *offline* training.

Correlation-based learning rules [3], [16] are more readily applicable to SNNs because they do not require the operations of a network to be differentiable. In general, correlation-based methods are based on Hebb's pos-

*MSc student, †Supervisor.

tulate [17], [18], summarized as follows: "Neurons wire together if they fire together." As a result, correlation-based methods put minimal constraints on the neural network and its applications. In addition to not requiring differentiable operations, most correlation-based learning rules can update a network's parameters while it is active, called *online* training [19]. However, correlation-based methods often have inferior performance compared to gradient-based methods [14], [20].

Combining the advantageous properties of gradient and correlation-based methods into a single learning rule would significantly improve the applicability of SNNs. Such a new learning rule has to be able to identify and compare temporal activity patterns online, which historically has been somewhat difficult [21]. However, eligibility traces can be used to determine these temporal relations [22]. Eligibility traces describe a temporal process in a single scalar value by increasing in value if the temporal signal is active, and gradually decaying back to zero in the absence of activity. Used initially in Reinforcement Learning (RL) applications to perform online learning in possibly infinite episodes, they were quickly adopted for use in SNNs [23], [24]. SNNs often use eligibility traces as a short-term memory of neuronal activity by discounting (or decreasing) a spike's influence as it is more distant in time.

Inspired by the effectiveness of gradient-based methods, the flexibility and online learning of correlation-based methods, and the ability of eligibility traces to capture temporal dependencies, the contributions of this work are as follows:

- Show that gradients of temporal network operations can be calculated in the form of eligibility traces. As a result, these gradients can be calculated while operating a network.
- An approximate error propagation scheme that is comparable to backpropagation.
- Trace propagation, a learning algorithm that combines eligibility traces and approximate error propagation to perform online supervised learning in SNNs. The method is named *trace propagation* because of its extensive use of eligibility traces in optimizing SNNs.

Section II starts with background information on SNNs and training methods. This section is followed by an overview of related works in section III. Next, trace propagation is presented in section IV. Section V presents the experiments that have been conducted to validate the learning rule. Lastly, this paper ends with its conclusion in section VI.

To the authors regret, a very comparable work [25] has recently been made available for preprint. The particular method was published at a time such that we could have read it before conducting this study. However, this work was performed independently, and [25] was only discovered after our study was concluded. So, this paper presents our findings as if [25] is not published. The differences between the methods are discussed in section IV-F.

II. Background

This section presents the relevant background information for the training of SNNs. Firstly, a description of SNNs and the dynamics of spiking neurons is presented in section II-A. Secondly, the basics of gradient-based learning in SNNs is found in section II-B. Thirdly, the general working of correlation-based learning can be found in section II-C. Lastly, eligibility traces and their use in temporal learning problems are described in section II-D.

A. Spiking Neural Networks

The dynamics of spiking neurons are primarily inspired by those of biological neurons [3]. As a result, spiking neurons communicate with sparse, spatiotemporal signals, and spiking neurons can operate asynchronously from each other. A network of such neurons becomes an efficient and powerful learning model [9].

SNNs belong to the fields of *neuromorphic computing* and *deep learning*. Compared to regular ANNs [4], SNNs have a similar architectural topology, but they greatly differ in their neuron dynamics. Figure 1 visualizes the dynamics of a spiking neuron, and is a reference for the following explanation. Spiking neurons have a state that at least consists of a *membrane potential*, which changes as the neuron receives input. If the membrane potential surpasses a *threshold* value, the neuron generates a binary output, called a *spike*. After generating a spike, the membrane potential resets to its resting value. At this moment, the neuron enters a *refractory* period during which the membrane potential cannot change, and thus, the neuron cannot spike. Because of their state and spiking signals, spiking neurons always operate in the temporal dimension, communicate with sparse binary signals called *spike trains*, function asynchronously from each other, and have more computational power per single neuron [9], [26] than linear artificial neurons.

Roughly two groups of neuron models exist, biophysical models and phenomenological models. Biophysical models [28] intend to model biological neurons with high accuracy, but this comes at the cost of high computational complexity. On the other hand, phenomenological models make a compromise between accuracy and computational complexity. Because this work uses phenomenological models the most common version, the LIF [3], [29], is treated next. This model builds on the assumption that spike timing captures all information in a spiking signal, and variations in spike

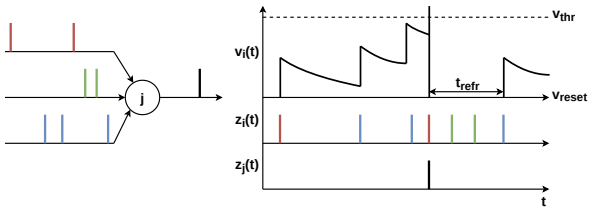


Figure 1: Dynamics of a Leaky Integrate-and-Fire (LIF) neuron. On the left is an illustration of the neuron with incoming spikes in color and an outgoing spike in black. The top row in the diagram on the right shows the progression of the membrane potential over time. The middle row shows presynaptic spikes aligned with the corresponding increase in membrane potential. The bottom row shows the postsynaptic spike, which aligns with the moment the membrane potential surpasses the firing threshold and resets. From [27].

amplitude and shape are unimportant. As a result, the LIF communicates with binary signals but retains rich temporal dynamics. Besides the LIF neuron, the Spike Response Model (SRM) [30] and the Izhikevich neuron [31] are commonly used.

Neurons can be freely composed into a neural network by connecting them. It is common practice to group neurons into layers and then stacking several layers into a network. Figure 2a shows an example of such a network. The network processes the input it receives one layer at a time. Once a layer is done processing, it communicates its output through its connections to the next layer. Neurons in a sending layer are called *presynaptic* neurons, neurons in a receiving layer *postsynaptic* neurons, and the connections between neurons are called *synapses*. Besides propagation information between neurons, synapses also can multiply the signal with a *synaptic weight*. The ability for an SNN to adapt these weights to change its dynamics is referred to as *plasticity*. A particular form of plasticity is the adaptive threshold [3], [21], where the firing threshold of a neuron temporarily increases every time it emits a spike. The name of this process is *Short-Term Plasticity (STP)*, referring to the non-permanent change in neuronal dynamics.

B. Gradient-Based Learning for SNNs

The state of the art learning methods for SNNs [14], [15], [32] are based on the BP algorithm [13]. BP was designed for optimizing regular ANNs and is the go-to learning method for such networks. However, there are two issues to overcome when applying BP to SNNs. The non-differentiable spiking function and the fact that spiking neurons operate in the temporal domain cause these issues. BP is explained next, followed by how to make SNN compatible with BP.

For the purpose of explaining BP, a neural network like depicted in fig. 2a is used. The neurons in the network do not have a state like spiking neurons do. Let $i = 1, 2, ..., n^{l-1}$ denote a set of presynaptic neurons in layer $l-1$ and $j = 1, 2, ..., n^l$ a set of postsynaptic neurons in layer l. Now, a postsynaptic neuron's output z_j is a function of the presynaptic output z_i and synaptic weights θ_{ij}.

BP performs gradient descent on a network's loss function by adjusting the network's weights. The algorithm does so in two separate phases, the forward pass and the backward pass. During the forward pass, the network processes its input, like in fig. 2a. While processing the input, the gradients of the network's operations are stored in memory. Once the network generates an output, it is analyzed using a loss function L, at which point the forward pass ends.

Next is the backward pass, where the network's parameters are updated to minimize the loss function. This phase starts with "backpropagating" information from the network's loss to all neurons in the network like shown in red in fig. 2b. The backward pass accounts for all paths through which an operation in the network influences the loss function. The ingenuity of the BP method is that it uses the chain rule to compute the *error signal* $\frac{\delta L}{\delta z_j}$ efficiently for each neuron in a network. For the neurons in the output layer, the error signal $\frac{\delta L}{\delta z_j}$ depends on the loss function. However, for a presynaptic neuron i in a non-output layer l, its error signal $\frac{\delta L}{\delta z_i^l}$ is based on the error signals of all postsynaptic neurons j in layer $l+1$ it is connected to. BP calculates the error signal as follows:

$$\frac{\delta L}{\delta z_i^l} = \sum_j \frac{\delta L}{\delta z_j^{l+1}} \frac{\delta z_j^{l+1}}{\delta z_i^l} \tag{1}$$

This process is repeated layer by layer until the algorithm reaches the input layer.

The backward pass finished by updating the network's free parameters θ_{ij}. To minimize the loss function, the update is performed in the opposite direction of the gradient of the loss function with respect to the network's parameters:

$$\Delta\theta_{ij} = -\mu \frac{\delta L}{\delta z_j} \frac{\delta z_j}{\delta \theta_{ij}} \tag{2}$$

Before updating the parameters, the gradients are multiplied with the learning rate $\mu << 1$. Its value is considerably smaller than one to prevent abrupt changes to the network.

As mentioned earlier, there are two issues with applying BP to SNN. Firstly, the spike generation function of a spiking neuron is non-differentiable due to the

reset of a neuron's state after spiking. Therefore, most gradient-based methods for SNNs make use of surrogate derivatives, some of which are presented in section III-A.

Secondly, the backward pass of regular BP does not work with explicit or implicit recurrent connections. BP in recurrent connections leads to self-reinforcing patterns where the outcome of eq. (1), $\frac{\delta L}{\delta z_j}$, is dependent on itself. A recurrent-connection is visualized in the left part of fig. 3. Neurons with recurrent-connections use their output, or state, of the previous timestep $t-1$ as part of their input for the current timestep t. These connections can be used when the data is sequential; i.e., it consists of multiple timesteps. Spiking neurons have an implicit recurrent connection since a neuron's state at timestep t depends on their state and output of the previous timestep $t-1$. Backpropagation Through Time (BPTT) [34] was introduced to deal with these recurrent connections. It "unrolls" a Recurrent Neural Network (RNN) in its temporal dimension, effectively creating a single layer for each timestep, as shown in the right part of fig. 3. After unrolling regular BP can be applied to the network to "backpropagate the error through time." Depending on the number of timesteps involved, unrolling of networks can lead to deep networks consisting of many layers.

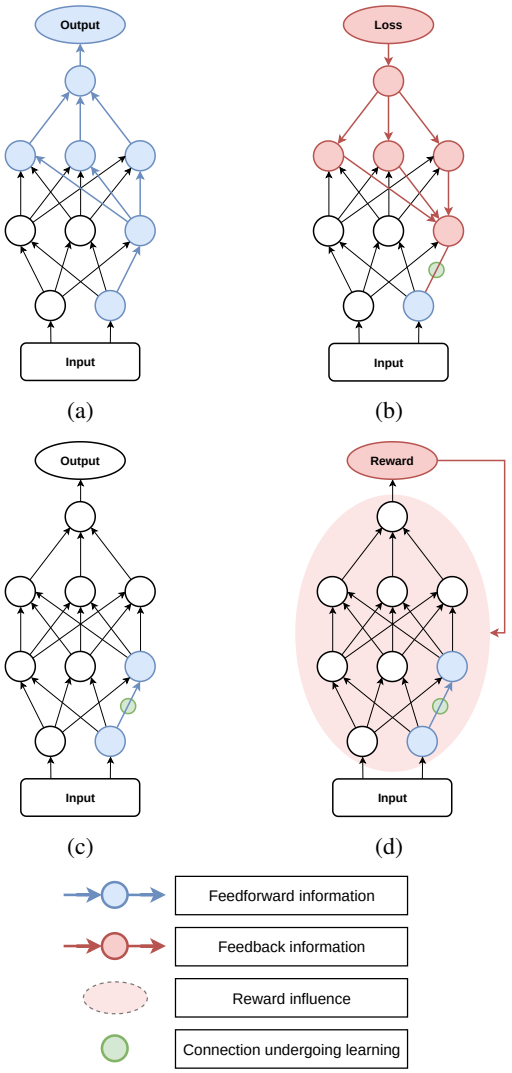


Figure 2: Visual explanation of the flow of information for parameter updates in BP and correlation-based methods. **(a)** Propagation of information from a single connection during the forward pass. **(b)** The backward pass of BP corresponding to the forward pass in **(a)**. **(c)** Information sources for correlation-based learning during the forward pass. **(d)** Information sources for correlation-based learning with a global reward or error signal during the forward pass. Adapted from [33].

C. Correlation Based Learning for SNNs

A defining property of correlation-based learning rules is that they do not require the operations in a neural network to be differentiable, making them well-suited for training SNNs. In their most elementary form, they strengthen connections between neurons that share a synaptic connection and whose signals are correlated, hence the name correlation-based methods. Besides, most of these methods are local, meaning they only rely on information that is directly present in the synaptic connections of a neuron, as visualized in fig. 2c. Because all computations required for the weight updates occur operating a network, there is no backward pass, which simplifies online learning. More information on different variations and existing correlation-based methods is presented in section III.

The most basic and popular correlation-based learning rule is Spike-Timing-Dependent Plasticity (STDP) [36], [37], which forms the basis for most existing and new correlation-based methods. All STDP rules work based on a difference in timing between presynaptic and postsynaptic spikes. Whenever a presynaptic spike arrives at a postsynaptic neuron *before* that neuron spikes, the synaptic weight of the connection carrying the presynaptic spike is increased. If the order of spikes is reversed, thus presynaptic *after* postsynaptic, the weight is decreased.

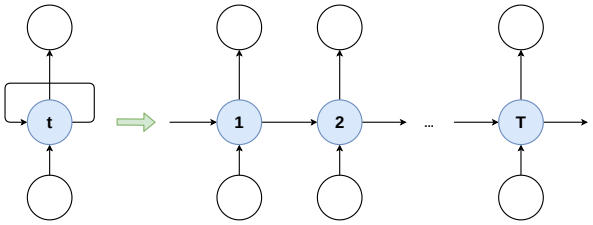


Figure 3: Unrolling a recurrent neural network in time. Adapted from [35].

Lastly, the smaller the timing difference between the two spikes, the larger the absolute weight change.

A common modification of STDP is to use it in a RL setting with a global reward or error signal. The global signal modifies the weight updates by changing its sign. Figure 2d shows a graphical interpretation of this reward-based learning. One of the main concerns for RL methods is to account for the time delay between related events. For example, a reward signal often occurs later than the action that caused the generation of this reward [38], [39]. Exact determination of this time difference is almost always intractable, and how methods deal with it is one of the main differences between them.

D. Eligibility Traces

One of the difficulties of training SNNs is the need for learning temporal dependencies, both in the data and in network activity. Learning temporal dependencies is especially difficult for online learning without explicitly storing past activities in memory. To (partly) alleviate this problem, eligibility traces were designed [22]–[24]. In general, eligibility traces express the activity of a process within a limited time-window as a scalar value. While doing so, they emphasize recent activities over those further in the past. These properties make eligibility traces a primary tool for determining temporal dependencies in SNNs [23]–[25].

An eligibility trace increases in value when the process it describes (like the activity of a neuron) is excited or generates an output, i.e.:

$$\epsilon^{t+1} = \gamma\epsilon^t + x^t,\ with\ \gamma < 1 \tag{3}$$

The parameter ϵ^t represents the trace at time t, γ is the decay rate, and signals that increase the trace at time t are indicated by x^t.

The recursive formulation of eq. (3) can be expanded between an arbitrary previous timestep t' and current time t as follows:

$$\epsilon^{t+1} = \gamma(\gamma(...\epsilon^{t'}) + x^t) + x^t \tag{4}$$

Because an eligibility trace decays back to zero, it has a time-window within which it operates. After a certain amount of recursive updates, the original input x^t is decreased to a negligible value. The decay rate γ determines the duration of this time window [22]:

$$\tau_\epsilon \approx \frac{1}{1-\gamma} \tag{5}$$

III. Related Work

The current section covers learning methods from the literature that are relevant to this work. Section III-A describes several existing gradient-based methods for training SNNs. Next, correlation-based methods are discussed in section III-B. Lastly, a couple of methods that researched the importance of accurate learning signals are presented in section III-C.

A. Gradient Based Learning in SNNs

Many methods have made SNNs compatible with BP and BPTT. The resulting learning rules depend on a surrogate gradient for the non-differentiable spiking function or circumvent it altogether. Some of the more prominent approaches include linearizing the relationship between neuronal input and output around the time a neuron spikes [40], replacing the instant spike generation and reset with a piece-wise smooth curve [15], directly differentiating on the neuron's membrane potential [32], [41], [42], using the gradients of a surrogate function [14], or using the alpha function te^{-t} for shaping a spike that allows for deriving exact gradients with respect to spike times [43].

The advantages of the previously mentioned methods include that they unlock many of the benefits of BP. Examples are accurate learning and mature software frameworks. In part because of these advantages, several gradient-based methods are state of the art for spatial and temporal credit assignment tasks for SNNs [14], [15], [43]. However, these methods perform offline training and have considerable memory requirements. Moreover, SNNs often are still outperformed by regular ANNs, especially with larger and deeper networks [7], [21]. There are two likely reasons for that. The first is the fact that a surrogate gradient is used. The second is that "unrolling" an SNN along its temporal dimension can result in networks of hundreds of layers. Training such deep networks historically has caused difficulties in training ANNs [44], [45], and likely also causes troubles for SNNs.

B. Correlation Based Learning in SNNs

In its basic form, STDP [37] has several limitations. It is unsupervised, unstable in its performance, and only applicable in a minimal number of situations. This section treats several variations of STDP that resolve the previous issues, grouped in unsupervised, supervised, and RL based methods.

Basic STDP performs weight updates without accounting for the network output or network performance. As a result, STDP creates bimodal weight distributions where the weights take on large absolute values [36] and making the learning and network performance unstable. Roughly two types of solutions exist. Using hard bounds on the weight values [46]–[48], or using soft bounds that gradually decrease the size of weight updates as the absolute value of weights increase [27], [49], [50]. These solutions are relatively simple to implement and place little to no restrictions on the network or data. However, the network's performance does not influence weight

updates in any way. As a result, purely unsupervised STDP is incapable of learning non-trivial input-output relations [51] and thus unsuited for supervised learning as well as some unsupervised learning tasks.

There are only a few supervised, correlation-based learning methods, of which ReSuMe [52] is likely the most well-known. ReSuMe requires a target spike train for each neuron in a network. If a connection's presynaptic signal correlates strongly with the target signal, its weight is potentiated. On the contrary, if a connection's presynaptic signal correlates more strongly with the neuron's actual output signal, its weight is depressed. The downside of ReSuMe is that setting a target spiking signal for each neuron in a network is complicated and unfeasible.

What separates most RL methods from each other is how they correlate network actions with a delayed reward signal. In [53], the authors use stochastic neurons that fire with Poisson statistics. These neurons effectively fire at a constant rate but have a fluctuating time between consecutive spikes. Fluctuations in the spiking statistics can now be correlated with the reward signal, indicating which parts of the network contributed to the desired output. However, neurons that fire with a fixed, average rate are considered inefficient because it prevents the neurons from using sparse signals. In Reward Modulated Spike Timing Dependent Plasticity with Eligibility Traces (MSTDPET) [23], eligibility traces [22] are used to represent recent activity of a neuron as a scalar value. The traces allow for postponing weight updates until a reward signal is received and also improve the method's capability of correlating the reward with the action that caused it. In [20], [54], a combination of MSTDPET and regular STDP is used to perform classification of MNIST images. This combination was the first successful application of MSTDPET to a visual recognition task of the scale of MNIST.

Reinforcement learning and STDP combine easily and requires little adjustments to a network or training environment. Moreover, the combination is relatively flexible since it can perform both RL and supervised learning. However, many of these methods suffer from common RL disadvantages. Sample inefficiency and bad scaling with network size [22] are the two most important ones.

C. Accuracy of Error Signals

Perpendicular to defining surrogate gradients for the spiking function is research that focusses on finding an alternative to the backward pass of BP. This research is mainly driven by the fact that connections between biological neurons send information in only one direction [55], [56]. However, the results that approximate error signals can effectively train moderately sized ANNs just as effectively as BP can have exciting implications for both SNNs and ANNs.

In [57] the precise backward pass of BP is replaced with random, fixed gradients. Experiments showed that this replacement resulted in almost no change in the trained network's performance. Because the random feedback gradients are fixed, the network's forward (regular) weights approximately aligned with the feedback gradients. This alignment allowed the network to learn as effectively as with BP. Random feedback gradients were tested on only moderately complex ANNs, and no experiments were performed on a large state of the art network.

The authors of [58] present a different approach. They prove that the network's error signal can be factorized into a local error signal for each neuron in the network by separating positive and negative weight connections from each other. By adding a global error signal like used in RL, the method minimizes the loss function for binary classification tasks. As a result, backpropagation of the error signal is not necessary, yet the method still achieves performance comparable to that of normal BP. The downside of this method is that it is only applicable to binary classification tasks.

In general, these works indicate that supervised learning in ANNs does not always require precise error propagation. The results are far from conclusive, but they stimulate further research and experimentation. Especially for SNNs, where exact backpropagation is not possible.

IV. Methods

As described in section II, gradient-based methods for training SNNs commonly use BPTT to deal with the network's temporal signals and implicit recurrent connections. However, this requires unrolling of the network in time, storing of gradients for each timestep, and offline learning. In this section, we present *trace propagation*, a learning rule that performs online and gradient-based optimization of SNNs. Trace propagation performs gradient descent on the network's loss. The weight updates are based on an adjusted version of eq. (2) that accounts for a neuron's state s and the timestep t of the weight update:

$$\frac{\delta L^t}{\delta \theta_{ij}} = \frac{\delta L^t}{\delta z_j^t} \frac{\delta z_j^t}{\delta s_j^t} \frac{\delta s_j^t}{\delta \theta_{ij}} \tag{6}$$

Equation (6) is a lot like the version used in BPTT for training SNNs. However, there are three distinct differences in the way trace propagation works compared to BPTT based methods:

- Trace propagation performs online training, so it evaluates the network's loss function and updates the network's parameters update with eq. (6) after each timestep. As a result, trace propagation can train networks for any amount of timesteps.
- Trace propagation calculates the gradient $\frac{\delta s_j^t}{\delta \theta_{ij}}$ online while the network processes its input. A neuron's state is dependent on temporal signals, so the same goes for gradients of that state. Its online calculation prevents the need for storing the gradient's previous values for use with the chain rule. The online calculation of $\frac{\delta s_j^t}{\delta \theta_{ij}}$ is possible because we show how the gradient can be calculated as an eligibility trace.
- The error signals $\frac{\delta L}{\delta z_j^t}$ for each neuron in a network are approximate values based on the eligibility trace representing the spiking activity of each neuron. Therefore, computations are easier than when using spiking signals.

Also, due to these properties, trace propagation does not have to unroll an SNN in the temporal dimension. Thus, it is simpler to use, requires less memory, and is more flexible than BPTT based methods.

The following sections are based on the neuron model and network architectures that we use in this work. However, trace propagation is compatible with a larger variety of neurons and architectures. Firstly, trace propagation applies to any neuron model for which eligibility trace calculation (see section IV-B) is valid. So, trace propagation works with all neuron models based on the SRM. Secondly, trace propagation can be used with and without an adaptive threshold (see section IV-D), because the threshold does not take part in the error calculations or error propagation. Thirdly, trace propagation can be applied to all feedforward and recurrent network architectures since error propagation is compatible with both.

The neuron model is often the starting point of an SNN design and thus the first topic in section IV-A. Next, section IV-B covers the online calculation of the gradient of a neuron's state with respect to its presynaptic parameters $\frac{\delta s_j^t}{\delta \theta_{ij}}$. Section IV-C discusses the surrogate gradient that is used for the spiking function $\frac{\delta z_j^t}{\delta s_j^t}$. Approximate error propagation used to calculate the error signal $\frac{\delta L}{\delta z_j^t}$ for each neuron in a network is discussed in section IV-D. The loss function L used in this work, as well as a short discussion of other loss functions that can be used with trace propagation are presented in section IV-E. Lastly, an overview of the differences between trace propagation and the eprop method of [25] can be found in section IV-F.

A. Neuron Model

This work uses the standard LIF neuron model [3] because of its rich temporal dynamics, computational simplicity, and extensive use in practice. The neuron also uses an adaptive threshold to regularize its spiking activity.

The LIF model was designed based on the assumption that the shape and amplitude of a spike do not carry information. So, the LIF neuron and its purely binary spikes are sufficient for propagating information while the neuron retains rich temporal dynamics. Its recursive dynamics are as follows:

$$v_j^{t+1} = e^{-dt/\tau_v}(v_j^t - v_{rest}) + \alpha \sum_i \theta_{ji} x_i^t \qquad (7)$$

The decay of the voltage of postsynaptic neuron j (indicated by subscript) is determined by its adaptation time constant τ_v. The binary-valued x_i^t represents incoming spikes at time t from presynaptic neuron i. Incoming spikes are multiplied with synaptic weight θ_{ij}, and an optional input scaling factor α. At the event of a postsynaptic spike z_j^t, the membrane potential is reset to its resting value v_{rest}. Lastly, the refractory period lasts for t_{refr} timesteps. During this period, the state of the neuron is constant, and the neuron cannot spike.

The LIF neuron is extended with an adaptive threshold [21] as a form of regularization. An adaptive threshold regularizes a neuron by increasing the neuron's firing threshold each time it spikes and prevents it from becoming increasingly active. The adaptive threshold consists of a lower-bound threshold value v_{thr} and an adaptive term ζ:

$$\xi^t = v_{thr} + \zeta^t \qquad (8)$$

The recursive dynamics of ζ^t are the same as for a basic eligibility trace:

$$\zeta^{t+1} = e^{-dt/\tau_{thr}}\,\zeta^t + \alpha_{thr} z \qquad (9)$$

B. Online Gradient Calculation

In this section we show how the gradient $\frac{\delta s_{ij}^t}{\delta \theta_{ij}}$ for eq. (6) can be calculated online by rewriting the offline formulation as an online eligibility trace.

The state of a neuron at time t depends on the neuron's input at that time, and indirectly on the neuron's input from previous timesteps through the neuron's state. Figure 4 visualizes a pair of connected neurons over T timesteps. The grey dotted lines indicate that a signal from neuron i at time t influences the state of neuron j at times $t' > t$. In addition, fig. 4 shows that an error signal is provided to the postsynaptic neuron at each timestep. Because a neuron's state depends on inputs from an

unknown number of previous timesteps, gradients of the neuron's state have to take all timesteps into account:

$$\frac{\delta s_j^t}{\delta \theta_{ij}} = \frac{\delta s_j^t}{\delta \theta_{ij}^t} + \frac{\delta s_j^t}{\delta s_j^{t-1}} \frac{\delta s_j^{t-1}}{\delta \theta_{ij}^{t-1}} + ... + \frac{\delta s_j^t}{\delta s_j^0} \frac{\delta s_j^0}{\delta \theta_{ij}^0} \tag{10}$$

One can interpret this gradient as the change in the current neuron state s_j due to changing the synaptic weight θ_{ij} within the time window τ_v. The terms where both the numerator and denominator have a superscript, like $\frac{\delta s_j^t}{\delta \theta_{ij}^t}$, stand for the derivative of the neuronal state with respect to its presynaptic parameters at time t. The term without a superscript in the denominator $\frac{\delta s_j^t}{\delta \theta_{ij}}$ accounts for the influence of the parameter change at all timesteps.

Equation (10) consists of two parts. Firstly, the immediate impact of a parameter change at time t. Secondly, the indirect impact parameter changes at previous timesteps have on the neuron state at the current time. A more compact summation form of eq. (10) is as follows:

$$\frac{\delta s_j^t}{\delta \theta_{ij}} = \sum_{t'=0}^{t} \frac{\delta s_j^t}{\delta s_j^{t'}} \frac{\delta s_j^{t'}}{\delta \theta_{ij}^{t'}} \tag{11}$$

Further expansion of the derivative $\frac{\delta s_j^t}{\delta s_j^{t'}}$ clearly shows how a neuron's state at time t' influences a neuron's state at time t:

$$\frac{\delta s_j^t}{\delta s_j^{t'}} = \frac{\delta s_j^t}{\delta s_j^{t-1}} \frac{\delta s_j^{t-1}}{\delta s_j^{t-2}} \cdots \frac{\delta s_j^{t'+1}}{\delta s_j^{t'}} = \prod_{k=t'}^{t-1} \frac{\delta s_j^{k+1}}{\delta s_j^k} \tag{12}$$

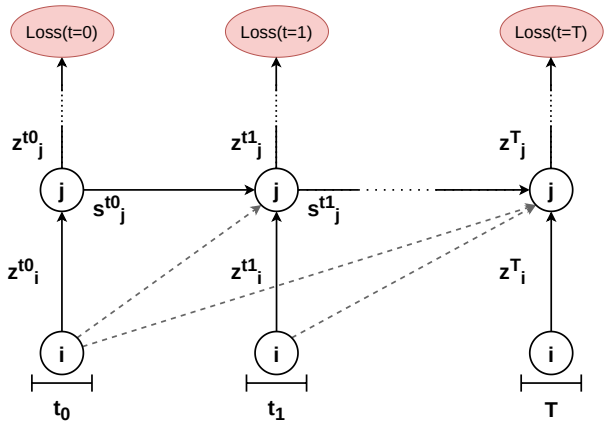


Figure 4: Computational graph for a presynaptic neuron i and postsynaptic neuron j and their connection that is trained with trace propagation. The dashed grey arrows represent the indirect influence z_i^t has on later states s_j^t.

The important point is that due to its summation over all timesteps, eq. (11) can be written in recursive formulation:

$$\frac{\delta s_j^t}{\delta \theta_{ij}} = \frac{\delta s_j^t}{\delta s_j^{t-1}} \frac{\delta s_j^{t-1}}{\delta \theta_{ij}} + \frac{\delta s_j^t}{\delta \theta_{ij}^t} \tag{13}$$

Thus, it shows that eq. (13) captures the gradient of a neuron's state with respect to a temporal signal.

Evaluating $\frac{\delta s_j^t}{\delta s_j^{t-1}}$ and $\frac{\delta s_j^t}{\delta \theta_{ij}}$ for the LIF neuron model in eq. (7) leads to the following:

$$\frac{\delta s_j^t}{\delta \theta_{ij}} = e^{-dt/\tau_v} \frac{\delta s_j^{t-1}}{\delta \theta_{ij}} + \alpha x_i^t \tag{14}$$

With the recursive formulation of eq. (14), $\frac{\delta s_j^t}{\delta \theta_{ij}}$ can be calculated online at each timestep t without storing all the gradients of eq. (10) in memory. If postsynaptic neuron j receives an input spike x_i^t from presynaptic neuron i the gradient increases in value. In addition, the gradient's value from the previous timestep decays at the same rate as membrane potential decay rate e^{-dt/τ_v} of neuron j. The form of eq. (14) is almost equivalent to that of the eligibility trace that describes the output activity of presynaptic neuron i, as defined in eq. (3). The only differences are the optional scaling factor α and the fact that the decay rate e^{-dt/τ_v} depends on the postsynaptic neuron's τ_v, and not that of the presynaptic neuron.

C. Surrogate Gradient

The surrogate gradient for a neuron's output with respect to its state $\frac{\delta z_j^t}{\delta s_j^t}$ is adapted from [21]:

$$\frac{\delta z_j^t}{\delta s_j^t} = 0.3 \cdot max\left(a, 1 - \left|\frac{v_j^t - v_{rest}}{v_{rest}}\right|\right) \tag{15}$$

The a parameter is a minimum value for the gradient, before multiplying it with 0.3. This gradient is also used in [25], which is the work that is very comparable to this one (see section IV-F for a comparison of the methods). However, the gradient itself is introduced in [21]. The works have the same first author, but [21] trains SNNs with BPTT. Equation (15) is chosen as the gradient for the spiking function because of its state of the art performance. Equation (15) can be calculated online at each timestep because it only depends on the neuron's voltage at that time.

The original derivative in eq. (15) uses $a = 0$. It has state of the art performance and is easy to use with regular LIF models. However, a common phenomenon in SNNs is that of "dead neurons" [59]. Dead neurons do not fire and thus do not participate in training. In this situation where a neuron never gets excited, its gradients remain zero. As a result, its synaptic weights do not change. This problem of "non-flowing gradients" is also

common in ANNs, where empirical results suggest that setting the minimum gradient to 0.2 improves learning [60]. Following this rationale, this work uses a slightly modified version of eq. (15) with $a = 0.2$.

D. Error Propagation

Here we introduce approximate error propagation for SNNs. It calculates an approximation for the error signal $\frac{\delta L}{\delta z_j}$ in eq. (6) at each timestep, for each neuron in a network. Error signals are propagated from layer to layer, starting at the output layer of an SNN, just as with BP. However, three main factors set the current method apart from BP. Firstly, error signals are based on the eligibility trace ϵ representing a neuron's activity. Secondly, the gradients for propagating the error signal to neurons are for use with eligibility traces instead of spiking signals. Lastly, there is no need for unrolling the SNN along the temporal dimension like with BPTT. A description of the logic behind error propagation is presented next, followed by discussions of the three defining factors of error propagation in the same order as they were just presented.

Error propagation works based on the premise that an increase in presynaptic activity for a connection with a positive weight results in an increase in postsynaptic activity. Conversely, an increase in presynaptic for a connection with a negative weight results in a decrease in postsynaptic activity. For both situations, the opposite holds if the presynaptic activity decreases. Lastly, it is also possible that there is no change in postsynaptic activity. A constant postsynaptic activity could happen because the change in presynaptic activity is not significant enough to cause a change in the postsynaptic spiking signal, or the postsynaptic neuron is already operating at its maximum firing rate. Table I extends this logic by also taking the sign of the loss signal for the postsynaptic neuron into account, as well as the desired change in the presynaptic weight's magnitude.

For this work, it was decided to determine the network error L and error signals for each neuron based on the eligibility trace ϵ representing a neuron's activity, instead of using a neuron's spiking signal z. The first reason is that eligibility traces express a signal as a scalar value, which is more convenient in computations. The second is that eligibility traces are less susceptible to differences in spike timing than pure spiking signals [22]. The substitution is based on the results where training ANNs with approximate error signals is just as accurate as training ANNs with exact BP error signals (see section III-C). Substituting ϵ for z in error propagation means that the error signal in eq. (6) is approximated as follows:

$$\frac{\delta L^t}{\delta z_j^t} \approx \frac{\delta L^t}{\delta \epsilon_j^t} \tag{16}$$

This approach can be used for applications where the network encodes information in the intensity of its output signal, i.e., the number of spikes and how recently the spikes took place. However, trace-based loss signals are not suited for exact spike-timing or first-to-spike applications. Eligibility traces are sensitive to spike timing, but not in an exact manner.

Communicating error signals to all neurons in an SNN is done recursively from layer $l+1$ to layer l, just like regular BP. However, it does so with a modified version of eq. (1) that accounts for the state of spiking neurons s, the timestep t, and eligibility traces ϵ instead of the neuron's output z:

$$\frac{\delta L^t}{\delta \epsilon_i^{t,l}} = \sum_j \frac{\delta L^t}{\delta \epsilon_j^{t,l+1}} \frac{\delta \epsilon_j^{t,l+1}}{\delta s_j^{t,l+1}} \frac{\delta s_j^{t,l+1}}{\delta \epsilon_i^{t,l}} \tag{17}$$

Equation (17) applies to all layers except for the output layer. For neurons in the output layer the error signal $\frac{\delta L^t}{\delta \epsilon_j^t}$ depends on the loss function. The superscript t denoting the timestep is omitted for the rest of this section to improve clarity.

As a result of the loss signal depending on eligibility traces, the gradients calculated while operating the network like $\frac{\delta z_j^l}{\delta s_j^l}$, cannot be used for error propagation. For example, $\frac{\delta z_j^l}{\delta s_j^l}$ in eq. (15) is related to $\frac{\delta \epsilon_j^l}{\delta s_j^l}$. However, $\frac{\delta z_j^l}{\delta s_j^l}$ applies to the generation of a single spike, whereas error propagation applies to eligibility traces that can span multiple spikes. Based on the logic in table I, we decided to set $\frac{\delta \epsilon_j^l}{\delta s_j^l}$ for error propagation equal to one:

$$\frac{\delta \epsilon_j^l}{\delta s_j^l} = 1 \tag{18}$$

The value of one is chosen because of the complexity of defining a proper non-linear relation between pre and postsynaptic activity traces. Whereas the size of $\frac{\delta \epsilon_j^l}{\delta s_j^l}$ is certainly wrong, the sign of the gradient is valid as long as the logic in table I holds. For table I (and thus eq. (18)) to hold, the parameters t_{refr} and τ_v for all neurons in a network have to satisfy:

$$\frac{t_{refr}}{\tau_v} \geq 0.1 \tag{19}$$

The proof that leads to the condition in eq. (19) can be found in appendix A. We leave improving $\frac{\delta \epsilon^l}{\delta s^l}$ to accurately capture the dynamics between pre and postsynaptic neuron's eligibility traces, as well as respecting a neuron's maximum and minimum firing rates, for future work.

Table I: The general relation between a postsynaptic error signal, the presynaptic weight, and the change in presynaptic activity and presynaptic weight magnitude required to lower the error signal. Because a spiking signal can only be zero or positive, the sign of the synaptic weight and the sign of the postsynaptic error signal determines the sign of the required change in presynaptic activity.

	Increase postsynaptic activity $\epsilon^l \uparrow$		Decrease postsynaptic activity $\epsilon^l \downarrow$	
	Presynaptic activity change ϵ^{l-1}	Presynaptic weight magnitude change $\|w\|$	Presynaptic activity change ϵ^{l-1}	Presynaptic weight magnitude change $\|w\|$
Positive presynaptic weight w^+	$\epsilon^{l-1} \uparrow$	$\|w\| \uparrow$	$\epsilon^{l-1} \downarrow$	$\|w\| \downarrow$
Negative presynaptic weight w^-	$\epsilon^{l-1} \downarrow$	$\|w\| \downarrow$	$\epsilon^{l-1} \uparrow$	$\|w\| \uparrow$

For eq. (17), only $\frac{\delta s_j^l}{\delta \epsilon_i^{l-1}}$ is left to define. The trace based gradient is related to the spike based gradient $\frac{\delta s_j^l}{\delta z_i^{l-1}}$. Evaluating $\frac{\delta s_j^l}{\delta \epsilon_i^{l-1}}$ for the neuron model in eq. (7) results in the following:

$$\frac{\delta s_j^l}{\delta \epsilon_i^{l-1}} = \theta_{ij} \tag{20}$$

Now that both $\frac{\delta \epsilon_j^l}{\delta s_j^l}$ and $\frac{\delta s_j^l}{\delta \epsilon_i^{l-1}}$ are in place, error propagation can be performed from layer $l+1$ to layer l:

$$\frac{\delta L}{\delta \epsilon_i^l} = \sum_j \frac{\delta L}{\delta \epsilon_j^{l+1}} \cdot 1 \cdot w_{ij} \tag{21}$$

The conclusion that the gradients for a neuron's output with respect to its state in the forward phase $\frac{\delta z_j^l}{\delta s_j^l}$, and error propagation phase $\frac{\delta \epsilon_j^l}{\delta s_j^l}$ need to be different is also experimentally tested in section V.

Lastly, error signals are not propagated through recurrent connections because trace propagation does not unroll recurrent connections. As stated in section II-B, propagating error signals through recurrent connections would lead to self-reinforcing patterns with eq. (17). Unrolling is done by BPTT to propagate the network's loss signal back in time. Yet, because trace propagation evaluates a network's loss function and weight updates at each timestep, unrolling is unnecessary.

E. Loss Functions

Trace propagation requires a scalar loss signal. Depending on the desired output, readout neurons [61] or spiking neurons can be used. Readout neurons do not spike, and their membrane potential represents the output without any conversion. On the contrary, when using a spiking neuron, it is necessary to convert the spike train to an eligibility trace using eq. (3), or a comparable method.

The experiments in section V are all classification problems that use the hinge loss with margin [62]. Spiking signals can vary considerably in intensity over time. A hinge loss with margin was chosen to generate a relatively steady output trace value, by enforcing a margin m between the target neuron t and the other output neuron their traces at all times. For the *target* output neuron t, thus the neuron that should be most active since it represents the class of the input signal, the loss is as follows:

$$L_t = max(0, m + \max_{j \neq t} \epsilon_j - \epsilon_t) \tag{22}$$

and for the *non-target* output neurons j:

$$L_j = max(0, m + \epsilon_j - \epsilon_t) \tag{23}$$

where m is a positive margin between the output eligibility trace ϵ_t of target neuron t and the output trace ϵ_j of all other neurons j. The total loss is the sum of the losses for all output neurons.

The gradient of eq. (22) with respect to the output of target neuron t is as follows:

$$\frac{\delta L}{\delta \epsilon_t} = \begin{cases} -1 & \text{if } \epsilon_t < m + \max_{j \neq t} \epsilon_j \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

and the gradient for a non-target neuron:

$$\frac{\delta L}{\delta \epsilon_j} = \begin{cases} 1 & \text{if } \epsilon_j > \epsilon_t - m \\ 0 & \text{otherwise} \end{cases} \tag{25}$$

However, the actual loss function can be any other commonly used one, like mean squared error or cross-entropy [19].

F. Comparison Trace propagation and Eprop

As mentioned at the end of section I, the method presented in [25] closely resembles the method presented in this paper. Because of that, the main similarities and differences are discussed next.

Overall, both trace propagation and the method in [25] are online, supervised, gradient-based learning methods for training SNNs. The main aspects that are the same are most of the operations that are performed during the forward pass of operating the network; these include:

- Calculation of $\frac{\delta s}{\delta \theta}$ as an eligibility trace, thus online without using BPTT.
- The surrogate gradient for the spike generation function $\frac{\delta z}{\delta s}$ as in eq. (15).
- The basic LIF neuron model.

The most important differences are in how error signals $\frac{\delta L}{\delta z}$ are calculated. The method in [25] compares several different error propagation schemes, but none is the same as presented in this work. The differences are detailed in table II.

V. Experiments

The following experiments are designed to test trace propagation's ability to train SNNs for spatial and temporal credit assignment problems. Also, the performance in networks of increasing depth and recurrent networks is tested.

The experiments were performed with the PySNN framework[1], which was developed during this thesis and available under an open-source license. Existing SNN software frameworks are either inflexible since they were developed for neuroscience research, or difficult to script with because they are written in C++ and CUDA [27]. With these shortcomings in mind, the PySNN framework is designed to be modular and easy to use by building it on top of PyTorch. PySNN makes use of PyTorch its excellent GPU acceleration, and the ease of scripting in Python. The PySNN API is kept syntactically as similar as possible as PyTorch to make learning the new framework relatively easy.

All of the experiments were performed on a desktop computer equipped with an Intel 4790K i7 quad-core processor clocked at 4 GHz per core, 16 GB of RAM, and an NVIDIA RTX 2070 GPU.

A. Configurations

The experiments in section V-B and section V-C partially share the same configuration. In these experiments, we tested two versions of $\frac{\delta \epsilon_j}{\delta s_j}$ during error propagation. The first version is $\frac{\delta \epsilon_j}{\delta s_j} = 1$ and the second is:

$$\frac{\delta \epsilon_j^t}{\delta s_j^t} = 0.3 \cdot max\left(a, 1 - \left|\frac{v_j^t - v_{rest}}{v_{rest}}\right|\right) \tag{26}$$

[1]PySNN is available at https://github.com/BasBuller/PySNN

Equation (26) is simply an adaptation of eq. (15), but for use during the error propagation phase. Equation (26) from now on is referred to as Bellec's derivative. The two versions are compared to experimentally verify that the deduction of $\frac{\delta z_j}{\delta s_j} = 1$ in section IV-D is valid.

More information about the hyperparameters for the networks, data, and optimizers is presented in appendix B.

B. Spatial Pattern Classification

Trace propagation was tested in two spatial credit assignment problems. The first is about classifying data where the distribution of firing rates over the input neurons follows a Gaussian distribution. A Gaussian was chosen because it allows expressing the difficulty of separating two samples with the Kullback-Leibler Divergence (KLD). For the second experiment, spiking MNIST [63] images are classified because this dataset is one of the go-to datasets for spatial credit assignment problems.

Data and Networks

Both a regular Gaussian distribution and MNIST images are scalar signals instead of spiking. A Poisson process converts a scalar signal to a spiking signal by using the scalar signal as its rate. Using a Poisson process adds variance to the sample and allows for generating a slightly different spiking signal each time the sample is used during training.

The Gaussian data samples are created by taking a Gaussian distribution and multiplying it with the maximum firing rate. Conversion of this distribution to a spiking signal was done by sampling the Gaussian distribution and using that as the rate for the Poisson process. Next, The KLD between two different samples was controlled by varying the distance between the mean of the distributions and their standard deviations. When doing so, the interval of possible values was fixed to keep a consistent frame of reference. Each version of the Gaussian dataset consists of ten classes. The different stages of this process are shown in fig. 10 and fig. 11 in appendix C.

For the grey-scale MNIST images, each pixel value was multiplied with the maximum firing rate and successively converted to a spiking signal. Two example of a spiking MNIST number can be found in fig. 16 and

Table II: Most important differences between the learning methods present in this work and [25].

Difference	This paper	Method of [25]
Signal used in evaluating loss function	Eligibility trace of neuron's activity	Voltage of readout neurons
Error propagation / communication (1)	Layer to layer	Random feedback gradients
Error propagation / communication (2)	-	Learned teacher network, or learning to learn
Threshold involved in weight update	No	Yes
Applied to ANNs	No	Yes, for improving online gradient calculation in RNNs

fig. 17, in appendix C. The dataset consists of 10 classes. It is worth noting that for the MNIST experiments, a subset of 30.000 images was used to train the network over a single epoch. Comparable methods use the entire dataset of 60.000 images for 10 to 100 epochs [14], [20]. This sample efficiency is an advantage of applying weight updates online, at every timestep.

All networks are fully connected, and classification is done based on the output neuron that generates the highest number of spikes.

Results

Results for the experiments with the Gaussian can be found in table III. In general, trace propagation adequately solves the spatial credit assignment problems, but its performance declines with increasing network depth.

In the MNIST results in table IV, the performance also decreases with increasing network depth. Besides, there is a noticeable difference in performance in favor of $\frac{\delta\epsilon}{\delta s} = 1$ over Bellec's derivative. This difference is a confirmation that the error propagation phase requires a different $\frac{\delta\epsilon}{\delta s}$ derivative than the one used during gradient calculation. However, the difference between $\frac{\delta\epsilon}{\delta s} = 1$ and Bellec's derivative is not present in the Gaussian results in table III, likely because the data samples are easier to separate. A relatively small network is enough to solve the Gaussian classification problem, compared to the MNIST classification problem. As a result, we hypothesize that error propagation plays a less important role in the Gaussian experiments.

In addition to the performance of our experiments, table IV also shows the results for three related methods for comparison. Whereas these methods all outperform our method, all of them use a considerably larger number of epochs during training. Also, except for the last method, all use BPTT based learning methods for training the network. However, it also shows that there is still room for improvement in trace propagation.

The difference between $\frac{\delta\epsilon}{\delta s} = 1$ and Bellec's derivative was explored in more detail by comparing how the two gradients performed error propagation. Error signals are compared on a layerwise basis by taking the mean of the error signals of all neurons in a layer. The layerwise mean of the error signals is inspected because performance dropped with increasing network depth. Figure 5 shows that for $\frac{\delta\epsilon}{\delta s} = 1$, the mean error per neuron in a layer tends to grow as more layers are passed. Conversely, fig. 6 shows that for Bellec's derivative the reverse is true. The vanishing of gradients is likely because in Bellec's derivative (see eq. (15)), the gradient takes a value between 0.2 and 1.0. Next, the gradient is multiplied with a value of 0.3. As a result, the mean error per neuron shrinks. In the case

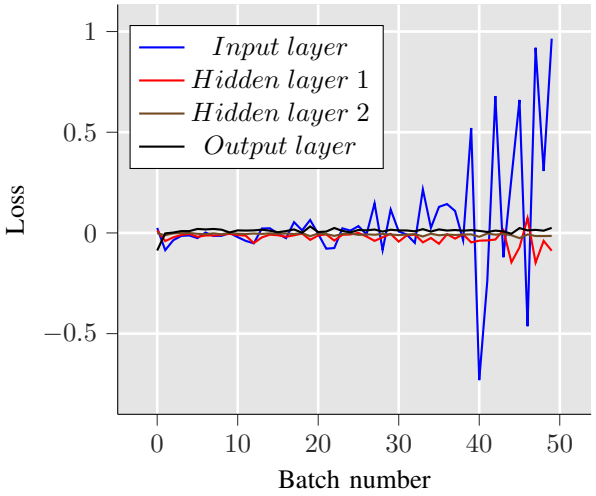


Figure 5: Error signal averaged over all neurons per layer during training. Data belongs to a four-layer network trained with $\frac{\delta\epsilon}{\delta s} = 1$ in the error propagation phase.

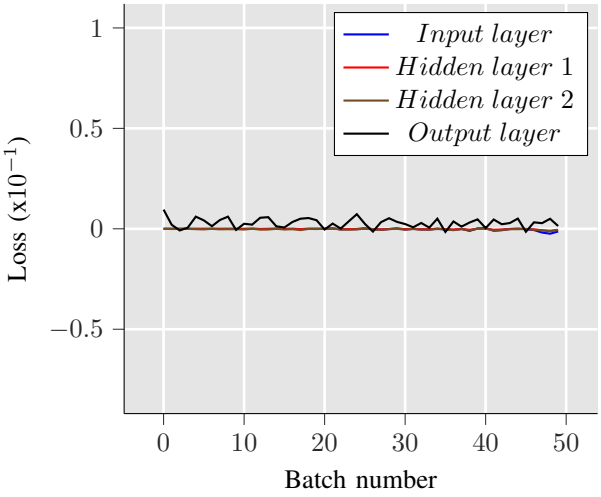


Figure 6: Error signal averaged over all neurons per layer, during training. Data belongs to a four-layer network trained with $\frac{\delta\epsilon}{\delta s} = 0.3 \cdot max(0.2, |\frac{v - v_{rest}}{v_{rest}}|)$ in the error propagation phase.

of $\frac{\delta\epsilon}{\delta s} = 1$, there is no scaling of the error except for θ_{ij} (see eq. (21)). As a result, error per neuron grows with each layer.

With the vanishing of error signals, learning becomes slower, or possibly even results in negligible parameter updates. Exploding error signals are not desirable either, but at least the network is provided a signal that results in noticeable parameter updates. We conjecture that the vanishing of error signals for networks trained with Bellec's derivative causes lesser performance in the MNIST experiments compared to networks trained with $\frac{\delta\epsilon}{\delta s} = 1$.

Table III: Training (first entry) and testing (second entry) accuracy for different network and data configurations for spatial Gaussian classification. The rows indicate different combinations of the $\frac{\delta\epsilon}{\delta s}$ gradient used during error propagation and of network configuration. The last three columns indicate three different versions of the Gaussian dataset. The datasets differ by the fact that the KLD becomes lower per column; thus, the samples are more alike. The actual values of the mean and standard deviation only have value when comparing samples with each other.

Error propagation	Network configuration	Mean sep. 2.0 Std. 3.0 KLD 0.22	Mean sep. 1.0 Std. 3.0 KLD 0.056	Mean sep. 1.0 Std. 5.0 KLD 0.02
$\frac{\delta\epsilon}{\delta s}$: 1	$200-160-120-10$	100.0%/100.0%	100.0%/100.0%	100.0%/100.0%
	$200-160-120-80-10$	100.0%/100.0%	100.0%/100.0%	95.0%/75.0%
$\frac{\delta\epsilon}{\delta s}$: Bellec	$200-160-120-10$	100.0%/100.0%	100.0%/100.0%	100.0%/90.0%
	$200-160-120-80-10$	100.0%/95.0%	100.0%/95.0%	98.0%/85.0%

Table IV: Training (first entry) and testing (second entry) accuracy on the MNIST dataset for the different $\frac{\delta\epsilon}{\delta s}$ used during error propagation. The first column shows the network configuration, whereas the second and third columns show the $\frac{\delta\epsilon}{\delta s}$ derivative used in error propagation. The last column presents accuracies achieved by some related works treated in section III. From the related works SLAYER [14] and Alpha Kernel [43] are BP based methods, whereas the network in the last related work [54] is trained with correlation-based methods.

Network configuration	$\frac{\delta\epsilon}{\delta s}$: 1	$\frac{\delta\epsilon}{\delta s}$: Bellec	Other method
784-512-10	95.7%/93.8%	94.4%/93.0%	-
784-512-256-10	95.6%/93.0%	91.5%/88.2%	-
SLAYER (BP) [14]	-	-	99.36%/97.0%
Alpha function (BP) [43]	-	-	99.96%/97.96%
MSTDPET and STDP [54]	-	-	97.2%

Whereas $\frac{\delta\epsilon}{\delta s} = 1$ outperforms Bellec's derivative in the MNIST experiments, it is not perfect. Improvements can be made by designing a gradient that more closely adheres to the dynamics of eligibility traces and takes neuronal firing limits into account. The current form $\frac{\delta\epsilon}{\delta s} = 1$ does neither. As a result, the adaptations will likely prevent exploding error signals.

C. Temporal Pattern Classification

Two temporal classification experiments were designed to test trace propagation. The first experiment tests the capacity of trace propagation to identify different intensity signals in a single neuron. The second experiment tests trace propagation to see if it can identify signals that vary over time according to different patterns.

Data and Networks

The first dataset consists of samples with one of the following four constant firing rates: $20, 60, 100, 140$ Hz. The second dataset consists of signals where a square wave $x(t)$ was used to modulate a fixed spike rate:

$$x(t) = sign(sin\ 2\pi ft) \tag{27}$$

The sign of the square wave $x(t)$ determines whether a spike at time t is assigned to input neuron one or neuron two. The frequencies f of the waves were: $2, 5, 10, 20$ Hz. All of the samples in the square wave dataset share the same base spiking rate; the only difference lies in the modulating wave frequency. For both datasets spiking signals were created using a Poisson process. Examples of the constant firing rate dataset are shown in fig. 12 and for the square wave signals in fig. 15, in appendix C.

Since the goal is to perform classification based on the temporal characteristics of the signal, the networks have a single input node for the constant rate data. For the square wave modulated signals, the networks have two input neurons to account for the two halves of a square wave's period. In both experiments, the input layer is followed by either a regular feedforward layer or a recurrent layer. For the pure feedforward network, the goal is to see if neuronal dynamics are sufficient for identifying (simple) temporal signals. If neuronal dynamics are enough, different neurons in a single layer should learn to respond to different incoming spiking frequencies. Classification is performed based on the total number of spikes.

Results

Table V shows the results for the same learning rule configurations as in section V-B. Only in the case of the feed-forward network applied to the constant rate dataset; some form of learning was achieved, yet it was far from perfect. Figure 7 shows how neurons in the hidden layer respond noticeably different to two different input frequencies for the constant rate dataset.

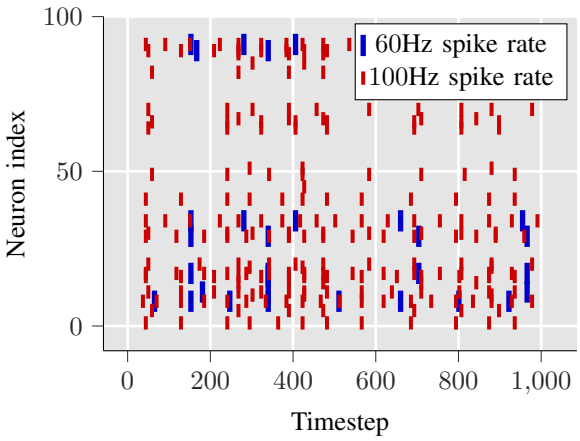


Figure 7: Hidden layer activity of a feedforward network for two different constant rate input signals. The network was trained with $\frac{\delta\epsilon}{\delta s} = 1$ during error propagation.

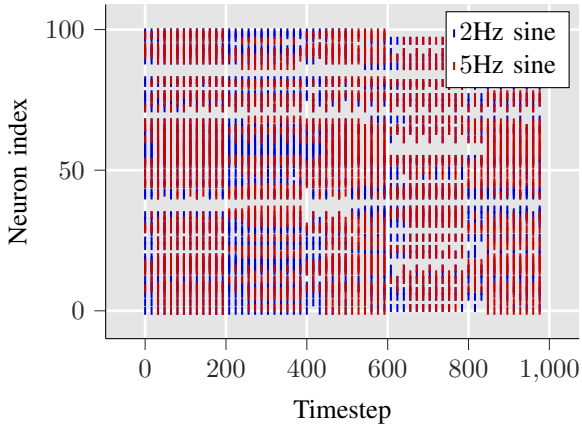


Figure 8: Hidden layer activity for a feedforward network trained for classifying spiking signals modulated with a square wave. Because of the highly active neurons, there is almost no difference in the network's hidden layer response to the two signals. The network was trained with $\frac{\delta\epsilon}{\delta s} = 1$ during error propagation.

Whereas there is a difference in the spiking patterns, the large difference in total spiking activity outweighs the difference in patterns. Hence, the network is unable to classify all different spiking rates accurately. In contrast, fig. 8 shows the hidden layer response for two square wave modulated signals. Here no specific pattern emerged but only overly active hidden neurons, which is why the network's performance is only as good as random guessing. The reasoning for why this might happen is that trace propagation is unable to capture long-term temporal dependencies. This inability is likely because the gradients have the same time-window as the membrane potential of τ_v ms (see eq. (5)). So, there is a limited period within which a signal can influence the gradient.

Table V: Training accuracy (first entry) and testing accuracy (second entry) in temporal classification experiments. The results are subdivided based on the network architecture and $\frac{\delta\epsilon}{\delta s}$ derivative used during error propagation. The networks trained with the constant rate dataset had the following architecture: $1 - 100 - 4$. The networks trained with the square wave dataset had the following architecture: $2 - 100 - 4$.

Error propagation	Dataset	Feedforward	Recurrent
$\frac{\delta\epsilon}{\delta s}$: 1	Constant rate square wave modulation	100.0/62.5% 62.5/25.0%	50.0/37.5% 0.0/0.0%
$\frac{\delta\epsilon}{\delta s}$: Bellec	Constant rate square wave modulation	87.5/75.0% 62.5/25.0%	75.0/25.0% 37.5/0.0%

Table V also shows the results for the recurrently connected networks. Whereas recurrent networks do not have to rely on neuronal dynamics to capture temporal dependencies, trace propagation in its current form was unable to optimize these networks. The "exploding" or "vanishing" activity of the networks caused a lack of positive results. A likely cause of these extreme reactions is the over or underestimation of error signals, as also described in section V-B.

VI. Conclusion

In this paper, we introduced trace propagation, a learning algorithm that performs online, supervised, gradient-based learning in SNNs. The algorithm is defined by the fact that it calculates gradients of temporal signals that influence spiking neurons as eligibility traces. Moreover, trace propagation uses approximate error signals based on eligibility traces instead of pure spiking signals. We demonstrate that trace propagation is capable of solving spatial credit assignment problems with short-term temporal relations, including classification of spiking MNIST images

We are hopeful that the combination of eligibility traces and approximate error propagation can form the basis for more successful learning methods. Research into using additional eligibility traces with long duration time windows will likely improve the identification and learning of long-term temporal dependencies. Moreover, research into accurately modeling the relation between the eligibility traces for presynaptic and postsynaptic neuron's activity is likely to prevent exploding or vanishing error signals.

REFERENCES

[1] S. Herculano-Houzel, "The human brain in numbers: A linearly scaled-up primate brain," *Frontiers in Human Neuroscience*, vol. 3, 2009.

[2] D. Drubach, *The Brain Explained*. Prentice Hall, 2000.

[3] W. Gerstner, *Neuronal Dynamics: From Single Neurons To Networks And Models Of Cognition*, uk ed. edition ed. Cambridge, United Kingdom: Cambridge University Press, Sep. 2014.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[6] "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II," 2019.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385 [cs]*, Dec. 2015.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805 [cs]*, Oct. 2018.

[9] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, Dec. 1997.

[10] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018.

[11] G. Indiveri, B. Linares-Barranco, T. J. Hamilton, A. van Schaik, R. Etienne-Cummings, T. Delbruck, S.-C. Liu, P. Dudek, P. Häfliger, S. Renaud, J. Schemmel, G. Cauwenberghs, J. Arthur, K. Hynna, F. Folowosele, S. Saighi, T. Serrano-Gotarredona, J. Wijekoon, Y. Wang, and K. Boahen, "Neuromorphic Silicon Neuron Circuits," *Frontiers in Neuroscience*, vol. 5, May 2011.

[12] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128× 128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor," *Solid-State Circuits, IEEE Journal of*, vol. 43, pp. 566–576, Mar. 2008.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, p. 533, Oct. 1986.

[14] S. B. Shrestha and G. Orchard, "SLAYER: Spike Layer Error Reassignment in Time," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 1412–1421.

[15] D. Huh and T. J. Sejnowski, "Gradient Descent for Spiking Neural Networks," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 1433–1443.

[16] T. J. Sejnowski and G. Tesauro, "The Hebb Rule for Synaptic Plasticity: Algorithms and," p. 10, 1989.

[17] D. Hebb, *The Organization of Behaviour*. John Wiley & Sons, Inc., 1949.

[18] S. Lowel and W. Singer, "Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity," *Science*, vol. 255, no. 5041, pp. 209–212, Jan. 1992.

[19] C. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. New York: Springer-Verlag, 2006.

[20] M. Mozafari, S. R. Kheradpisheh, T. Masquelier, A. Nowzari-Dalini, and M. Ganjtabesh, "First-Spike-Based Visual Categorization Using Reward-Modulated STDP," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 12, pp. 6178–6190, Dec. 2018.

[21] G. Bellec, D. Salaj, A. Subramoney, R. Legenstein, and W. Maass, "Long short-term memory and Learning-to-learn in networks of spiking neurons," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 787–797.

[22] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, Cambridge, MA, 2018.

[23] R. V. Florian, "Reinforcement Learning Through Modulation of Spike-Timing-Dependent Synaptic Plasticity," *Neural Computation*, vol. 19, no. 6, pp. 1468–1502, Jun. 2007.

[24] W. Gerstner, M. Lehmann, V. Liakoni, D. Corneil, and J. Brea, "Eligibility Traces and Plasticity on Behavioral Time Scales: Experimental Support of NeoHebbian Three-Factor Learning Rules," *Frontiers in Neural Circuits*, vol. 12, 2018.

[25] G. Bellec, F. Scherr, E. Hajek, D. Salaj, R. Legenstein, and W. Maass, "Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets," *arXiv:1901.09049 [cs]*, Jan. 2019.

[26] A. Gidon, T. A. Zolnik, P. Fidzinski, F. Bolduan, A. Papoutsi, P. Poirazi, M. Holtkamp, I. Vida, and M. E. Larkum, "Dendritic action potentials and computation in human layer 2/3 cortical neurons," *Science*, vol. 367, no. 6473, pp. 83–87, Jan. 2020.

[27] F. Paredes-Vallés, K. Y. W. Scheper, and G. C. H. E. de Croon, "Unsupervised Learning of a Hierarchical Spiking Neural Network for Optical Flow Estimation: From Events to Global Motion Perception," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2019.

[28] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, Aug. 1952.

[29] R. B. Stein, "A Theoretical Analysis of Neuronal Variability," *Biophysical Journal*, vol. 5, no. 2, pp. 173–194, Mar. 1965.

[30] W. M. Kistler, W. Gerstner, and J. L. van Hemmen, "Reduction of the Hodgkin-Huxley Equations to a Single-Variable Threshold Model," *Neural Computation*, vol. 9, no. 5, pp. 1015–1045, Jul. 1997.

[31] E. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, Nov. 2003.

[32] J. C. Thiele, O. Bichler, and A. Dupret, "SpikeGrad: An ANN-equivalent Computation Model for Implementing Backpropagation with Spikes," *arXiv:1906.00851 [cs]*, Jun. 2019.

[33] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, "Backpropagation and the brain," *Nature Reviews Neuroscience*, pp. 1–12, Apr. 2020.

[34] P. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, Oct. 1990.

[35] J. Kvita, "Visualizations of RNN units," Apr. 2016.

[36] N. Caporale and Y. Dan, "Spike Timing–Dependent Plasticity: A Hebbian Learning Rule," *Annual Review of Neuroscience*, vol. 31, no. 1, pp. 25–46, 2008.

[37] J. Sjostrom and W. Gerstner, "Spike-timing dependent plasticity," *J. Sj*, p. 18, 2010.

[38] E. M. Izhikevich, "Solving the distal reward problem through linkage of STDP and dopamine signaling," *Cerebral Cortex (New York, N.Y.: 1991)*, vol. 17, no. 10, pp. 2443–2452, Oct. 2007.

[39] N. Frémaux, H. Sprekeler, and W. Gerstner, "Functional Requirements for Reward-Modulated Spike-Timing-Dependent Plasticity," *Journal of Neuroscience*, vol. 30, no. 40, pp. 13 326–13 337, Oct. 2010.

[40] S. Bohte, J. Kok, and J. Poutré, "SpikeProp: Backpropagation for networks of spiking neurons." in *ESANN*, vol. 48, Jan. 2000, pp. 419–424.

[41] F. Zenke and S. Ganguli, "SuperSpike: Supervised learning in multi-layer spiking neural networks," *Neural Computation*, vol. 30, no. 6, pp. 1514–1541, Jun. 2018.

[42] A. Tavanaei and A. S. Maida, "BP-STDP: Approximating Backpropagation using Spike Timing Dependent Plasticity," *arXiv:1711.04214 [cs]*, Nov. 2017.

[43] I. M. Comsa, K. Potempa, L. Versari, T. Fischbacher, A. Gesmundo, and J. Alakuijala, "Temporal coding in spiking neural networks with alpha synaptic function," *arXiv:1907.13223 [cs, q-bio]*, Jul. 2019.

[44] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[45] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997.

[46] R. Kempter, W. Gerstner, and J. L. van Hemmen, "Hebbian learning and spiking neurons," 1999.

[47] S. Song, K. D. Miller, and L. F. Abbott, "Competitive Hebbian learning through spike-timing-dependent synaptic plasticity," *Nature Neuroscience*, vol. 3, no. 9, p. 919, Sep. 2000.

[48] W. Gerstner, R. Kempter, J. L. van Hemmen, and H. Wagner, "A neuronal learning rule for sub-millisecond temporal coding," *Nature*, vol. 383, no. 6595, pp. 76–78, Sep. 1996.

[49] M. C. W. van Rossum, G. Q. Bi, and G. G. Turrigiano, "Stable Hebbian Learning from Spike Timing-Dependent Plasticity," *Journal of Neuroscience*, vol. 20, no. 23, pp. 8812–8821, Dec. 2000.

[50] J. E. Rubin, R. C. Gerkin, G.-Q. Bi, and C. C. Chow, "Calcium Time Course as a Signal for Spike-Timing–Dependent Plasticity," *Journal of Neurophysiology*, vol. 93, no. 5, pp. 2600–2613, May 2005.

[51] P. Baldi and P. Sadowski, "A theory of local learning, the learning channel, and the optimality of backpropagation," *Neural Networks*, vol. 83, pp. 51–74, Nov. 2016.

[52] F. Ponulak and A. Kasiński, "ReSuMe learning method for spiking neural networks dedicated to neuroprostheses control," Jan. 2006.

[53] X. Xie and H. S. Seung, "Learning in neural networks by reinforcement of irregular spiking," *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics*, vol. 69, no. 4 Pt 1, p. 041909, Apr. 2004.

[54] M. Mozafari, M. Ganjtabesh, A. Nowzari-Dalini, S. J. Thorpe, and T. Masquelier, "Bio-Inspired Digit Recognition Using Spike-Timing-Dependent Plasticity (STDP) and Reward-Modulated STDP in Deep Convolutional Networks," *arXiv:1804.00227 [cs, q-bio]*, Mar. 2018.

[55] F. Crick, "The recent excitement about neural networks," *Nature*, vol. 337, no. 6203, pp. 129–132, Jan. 1989.

[56] S. Grossberg, "Competitive Learning: From Interactive Activation to Adaptive Resonance," *Cognitive Science*, vol. 11, no. 1, pp. 23–63, 1987.

[57] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning," *Nature Communications*, vol. 7, p. 13276, Nov. 2016.

[58] D. Balduzzi, H. Vanchinathan, and J. Buhmann, "Kickback cuts Backprop's red-tape: Biologically plausible credit assignment in neural networks," *arXiv:1411.6191 [cs, q-bio]*, Nov. 2014.

[59] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training Deep Spiking Neural Networks Using Backpropagation," *Frontiers in Neuroscience*, vol. 10, 2016.

[60] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical Evaluation of Rectified Activations in Convolutional Network," *arXiv:1505.00853 [cs, stat]*, Nov. 2015.

[61] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations," *Neural Computation*, vol. 14, no. 11, pp. 2531–2560, Nov. 2002.

[62] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. J. Huang, "A Tutorial on Energy-Based Learning," p. 59, 2006.

[63] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[64] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Mar. 2010, pp. 249–256.

APPENDIX A
ERROR PROPAGATION PROOF

Deriving the conditions for which table I is always valid comes down to proving that an increase in a presynaptic activity trace for a connection with a positive (negative) weight results in a constant or increased (decreased) postsynaptic activity. The proof compares trace values for a set of n spikes and a set of $n+1$ spikes. A difference of only one spike is chosen since this results in the smallest change in the corresponding eligibility trace value. The goal with these spike sets is to show that the *average* and *maximum* trace values for any permutation in spike timings of the set of $n+1$ spikes are larger than the trace values of the set of n spikes. This way, the spike set with the larger eligibility trace always leads to more excitation of the postsynaptic neuron receiving that signal. In turn, this postsynaptic neuron becomes more active, and thus its activity trace increases in value. So, the logic for table I holds.

The eligibility traces for n and $n+1$ spikes have the following dynamics:

$$\epsilon^{t+1} = e^{(-dt/\tau_t)}\epsilon^t + x^t \tag{28}$$

The average trace value for $n+1$ spikes is always higher than the average for n spikes. Equation (28) is rewritten to calculate the trace's value at any time:

$$\epsilon^t = \sum_{i=1}^{k} exp\left(-\frac{t-t_i}{\tau_t}\right) \tag{29}$$

The summation in eq. (29) covers all spikes of a set of k spikes. Spike timing is indicated by t_i. Equation (29) shows that trace increase and decay due to one spike never influences the gain or decay due to other spikes. Thus, more spikes always result in a higher average trace.

Determining whether the set of n or $n+1$ spikes has the largest maximum trace value is a bit more complicated. The maximum trace value is not only determined by the number of spikes, but also by the spike timings. So, the goal is to determine the conditions such that the maximum trace value for $n+1$ spikes is always the highest for any permutation of spike timings. During this analysis, the spikes take place in a time window of τ_t ms. This window is approximately the time window of the trace (see section IV-B). However, larger time windows are possible.

The highest possible trace value for any spike set occurs when spikes occur as close to each other as possible in time. Conversely, the lowest maximum trace value occurs maximally spread spikes. For the LIF neuron used in this work, the minimum time between two consecutive spikes is equal to the refractory duration t_{refr}. So, the maximum trace for n spikes that are t_{refr} ms apart (minimum time between spikes) has to be lower than the maximum trace value for $n+1$ spikes that are τ_t/n ms apart (maximum time between spikes). Comparing the conditions after substituting them in eq. (29) leads to the following expression:

$$\sum_{i=0}^{n-1} exp\left(-\frac{i \cdot t_{refr}}{\tau_t}\right) \leq \sum_{j=0}^{n} exp\left(-\frac{j \cdot \tau_t}{\tau_t \cdot n}\right) \tag{30}$$

The ratio $\frac{t_{refr}}{\tau_t}$ and the number of spikes n together determine whether the inequality in eq. (30) is true. This relation can be interpreted as follows. By increasing $\frac{t_{refr}}{\tau_t}$, either t_{refr} is increased, resulting in larger minimum timing difference between spikes. As a result the eligibility trace is more "flat" and has a lower potential maximum value. The other option is to decrease τ_t. In this situation the decay rate $exp(-dt/\tau_t)$ increases and the eligibility trace its maximum value decreases. Since there is no way to nicely solve this equation in an analytical way for variable n and $\frac{t_{refr}}{\tau_t}$, it is solved numerically. Figure 9 shows the results for the analysis. The left hand side of eq. (30) has been evaluated for four different $\frac{t_{refr}}{\tau_t}$ values, for which the curves are shown in blue. The evaluation of the right hand side of eq. (30) is shown in red. It becomes clear that for $\frac{t_{refr}}{\tau_t} \geq 0.1$, the maximum trace value for the least favorable spike timing permutation of $n+1$ spikes is equal or higher than the maximum value for the most favorable timing permutation of n spikes, for all values of n. Thus, under this condition, table I holds.

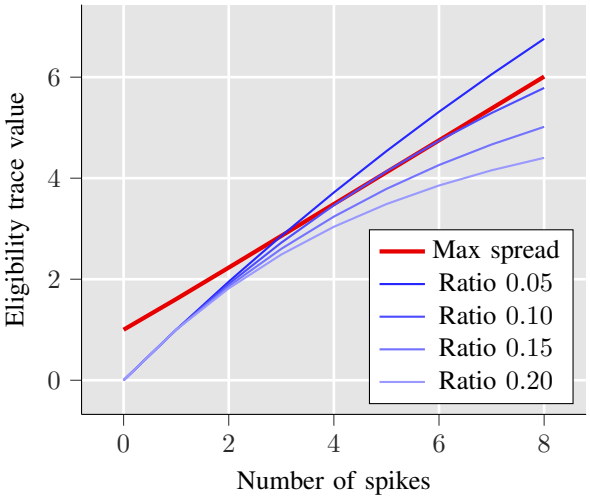


Figure 9: Maximum eligibility trace values for maximally spread spikes, as well as for minimal spread spikes for different $\frac{t_{refr}}{\tau_t}$ values.

APPENDIX B
EXPERIMENT CONFIGURATIONS

This appendix contains details about the network and hyperparameter configurations for the experiments described in section V. Hyperparameters that are specific for the Gaussian dataset are found in table VI, the MNIST dataset in table VII, and the temporal datasets in table VIII and table IX. Table X contains the hyperparameters for the networks, optimizers, and training configurations for all three experiments.

Table VI: Gaussian data parameters

Parameter	Description	Value
Firing rate	Rate provided to the Poisson process	50
Mean separation	Distance between the centers of the Gaussian distributions	1.0
Standard deviation	Standard deviation of each Gaussian distribution	$1.0, 3.0, 5.0$
Interval	Interval the Gaussian distributions are sampled from	$-5, 15$
Duration	Duration of a single sample	300 ms

Table VII: MNIST data parameters

Parameter	Description	Value
Firing rate	Rate provided to the Poisson process	50
Duration	Duration of a single sample	300 ms

Table VIII: Constant rate parameters, temporal data

Parameter	Description	Value
Firing rates	Rates provided to the Poisson process, one per sample	$20, 60, 100, 140$
Duration	Duration of a single sample	1000 ms

Table IX: square wave parameters, temporal data

Parameter	Description	Value
Firing rate	Rate provided to the Poisson process, same for all samples	60
Sine frequency	Frequencies of the modulating square waves, one per sample	$2, 5, 10, 20$
Duration	Duration of a single sample	1000 ms

Table X: Network, optimizer, and training configurations for all three experithe ments

Parameter	Description	Gaussian	MNIST	Temporal
τ_v	Membrane potential adaption time constant	40 ms	40	50
α_v	Input activation potential scaling factor	1	1	1
$\tau_{threshold}$	Threshold adaptation time constant	150 ms	150	300
$\alpha_{threshold}$	Adaptive threshold input scaling factor	$1/30$	$1/30$	$1/60$
v_{rest}	Membrane potential resting value	0 V	0	0
$v_{threshold}$	Spiking threshold on membrane potential	1 V	1	1
t_{refr}	Duration of the refractory period	4 ms	4 ms	5 ms
dt	Duration of simulation timestep	1 ms	1	1
weight init	Network weight initialization scheme	Glorot [64]	Glorot	Glorot
lr	Learning rate	0.0005	0.001	0.0002
βs	Momentum parameter for stochastic gradient descent with momentum	0.9	0.9	0.9
hinge loss margin	Desired trace difference between target class and most offending output	5.0	5.0	3.0
epochs	Number of training epochs	10	1	20
batch size	Number of samples in a single batch	20	100	8
samples per batch	Total number of samples in a batch	100	30.000	80
testing samples	Total number of samples in the testing dataset	20	1000	16

Appendix C
Data Examples

This appendix contains examples of the artificial datasets introduced in section V. Figure 10 and fig. 11 show two different perspectives on the Gaussian distribution dataset. Figure 12 and fig. 13 show examples of the constant firing rate dataset, whereas fig. 15 shows a square wave example. For the temporal signals, there is only a spiking representation since summing over a single neuron would provide no information. Lastly, two examples of the MNIST dataset in spiking format is shown in fig. 16 and fig. 17.

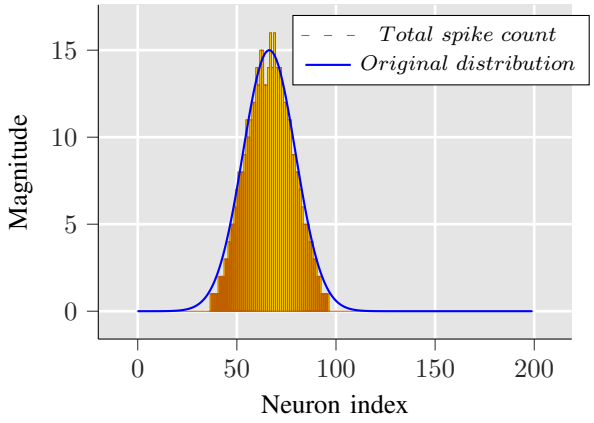


Figure 10: Data sample where the firing rate follows a Gaussian distribution. The continuous line shows the parametric distribution of the firing rates, whereas the bars indicate the number of spikes per neuron generated by a Poisson process.

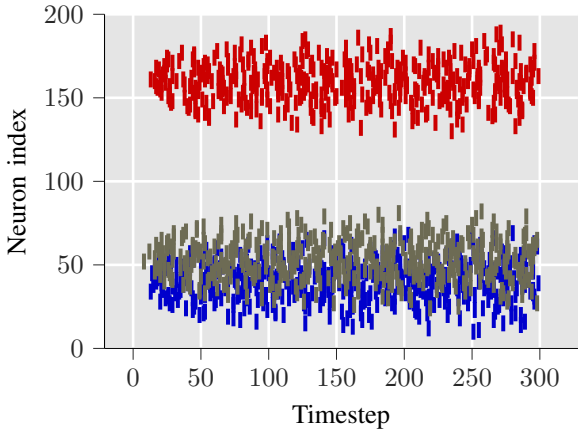


Figure 11: The blue spikes correspond to the distribution in fig. 10, but represented in spiking format. The grey and red distributions are intended to provide an indication of the minimum and maximum separation between the mean of two distributions.

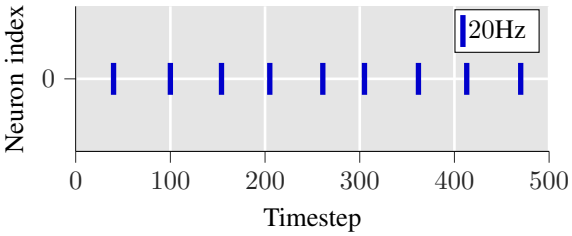


Figure 12: Constant rate sample in spiking format. The inconsistent spacing between consecutive spikes is because the spike trains are generated by a Poisson process.

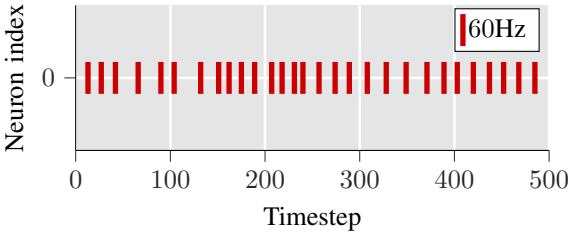


Figure 13: Another constant rate sample in spiking format.

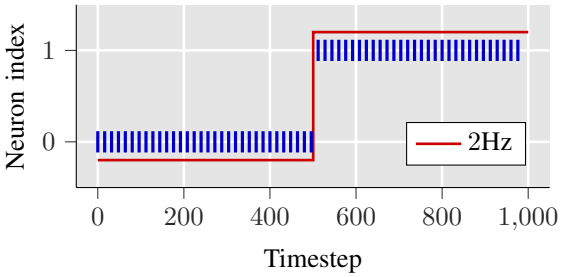


Figure 14: Square wave sample in spiking format, together with the square wave that was used to modulate the spike train.

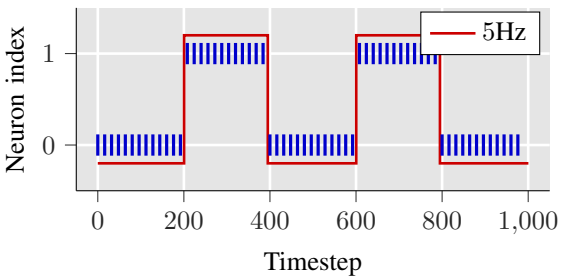


Figure 15: Another square wave sample in spiking format together with the modulating square wave.

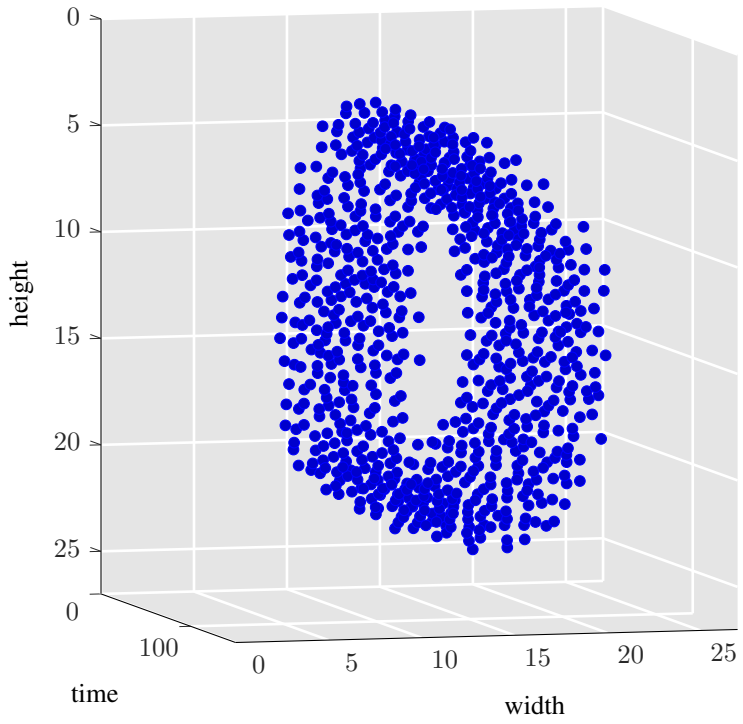


Figure 16: A number 0 from the MNIST dataset in its spiking format.

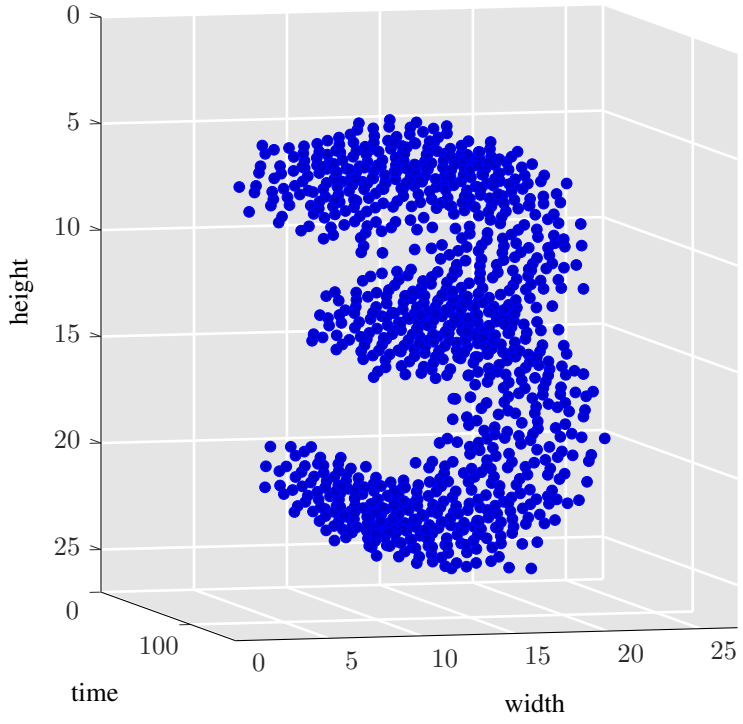


Figure 17: A number 3 from the MNIST dataset in its spiking format.

II

Literature Study

2

Deep Learning

Possibly the most defining and important moment for deep learning to current date took place during the ILSVRC - ImageNet[1] in 2012. It was during this competition that the work of Krizhevsky et al. [49] blew past its competition in the tasks of classifying images of 1000 different classes, outperforming the second place by more than 10% accuracy. This marked the moment where deep learning entered the spotlights and interest in the subject grew quickly. Over time ANNs were being used in many applications like computer vision, natural language processing, robotics, bioinformatics, weather forecasting, algorithmic trading, video games and many more. Part of the reason for this wide applicability is the ”universal approximation theorem” [5]. This states that every continuous function can be approximated by a large enough parallel perceptron [84] (single layer neural network of many parallel neurons) with arbitrary accuracy, given it has infinite data.

Artificial neural networks are considered state-of-the-art algorithms in machine learning in general. Yet, they still deal with the issue of being computationally intensive with some models costing several millions of dollars in energy and hardware to train. Partly as an answer to this problem and partly because of their biological plausibility SNNs have recently been enjoying increased attention. Spiking neural networks are largely inspired by how animal brains work. Where modern ANN models achieve super-human performance on very specific tasks, the human brain is still incredibly efficient at performing a large set of different tasks while only requiring 20W to operate [19].

This chapter consists of an overview of ANNs in section 2.1, and for SNNs in section 2.2.

2.1. Artificial Neural Networks

A good description of deep learning is quoted from LeCun et al. [53]: ”Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex functions can be learned.” The large advantage that deep learning methods provide is not having to hand-craft feature extractors that normally require lots of expertise and domain knowledge since the neural network can learn them by itself.

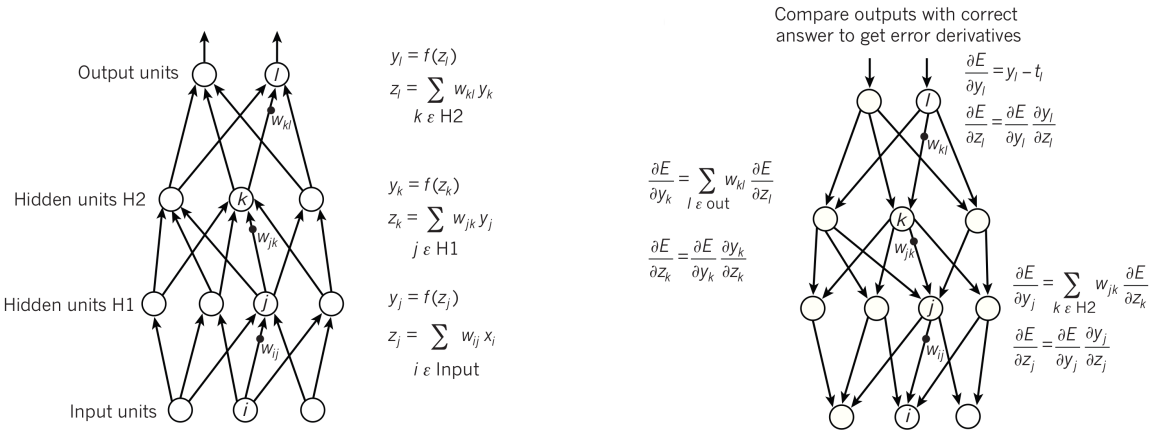
Starting, in section 2.1.1 the fundamentals of deep learning are presented. Next, basic architectures within deep learning are presented in section 2.1.2. This is followed by an overview of ANNs applied to event-based vision in section 2.1.3. Finally, this section is concluded with an overview of hardware and software used for deep learning in section 2.1.4. For a more in-depth description of deep learning the reader is referred to Sze et al. [101] or Goodfellow et al. [33].

2.1.1. Fundamentals

A simple four-layer network composed of an input, two hidden and an output layer is presented in fig. 2.1a and will be used as a reference. Circles in the figure represent neurons, and the arrows represent connections between them. At their core, all neural networks are trained (optimized) to

[1] http://www.image-net.org/challenges/LSVRC/2012/ - accessed 27-07-2019

minimize a loss function or maximize a reward function. This is done by adjusting the strength of each of these connections. This is done in two phases between which is consistently altered. The first phase is called the forward pass and the second the backward pass. The forward pass is the inference phase in which the network computes its output based on the provided input, ”propagating the input forward through the network”. During the backward pass, the network performs the actual training by updating its weights using the back-propagation algorithm, ”propagating the error signal backward through the network”. Both of these passes will be further elaborated in the following sections.



(a) (b)

Figure 2.1: (a): Feed forward multilayer neural network. (b) Back-propagation in feed forward network. Both figures adapted from LeCun et al. [53]

To perform the backward pass the learning algorithm needs an error signal (or loss function) to tell it in which direction it has to update the network’s weights and by how much such that it minimizes the error. The type of loss function determines the learning mode that is being used. The definitions of these modes and the requirements they impose on the data and network are as follows:

- **Supervised Learning (SL)** is the most common learning mode where a network is provided both training examples and the desired output for that example. The desired output can range from class labels to image transformations. The weights of the network are adjusted to make the difference between the network output and the desired output as small as possible. The network is trained to find patterns in the input data that have the strongest correlation with the target signal. This often leads to outstanding performance on a test task as long as the inputs after training are much like the training examples. The clear downside is having to provide the desired outcome for each input, which often results in humans manually annotating thousands or even millions of examples.

- **Unsupervised Learning (UL)** requires only raw training examples. Because of the absence of a clear target output, the network will try to extract patterns that are most common within the training data. The advantage of this is that no manual labeling is needed allowing for using large, automatically gathered datasets. The downside is that it is considerably harder to let the network learn a very specific pattern.

- **Reinforcement Learning (RL)** is best described as learning through interaction (See chapter 3 for more information), or trial and error. The interaction can be with a real environment, a simulator or a mathematical model. During the interaction, the network receives reward signals from its environment and has as its goal to maximize it. The ability to learn from pure interaction without supervision is the strong point of this learning method. The downside is that in most cases the reward signal contains very little information compared to the information required to update the large number of parameters of the network, thus leading to inefficient learning.

Forward Pass

Many applications of deep learning use feedforward neural network architectures like in fig. 2.1a, which learn to map a fixed-size input (for example, an image) to a fixed-size output (for example, the network's estimate for the probability the image contains an object from a specific class) [53]. Processing in an ANN is performed layer per layer. To go from one layer to the next, the neurons in a layer compute a weighted sum of their inputs from the previous layer and pass the result through a non-linear function, often called the activation function. This process for a single neuron is represented in eq. (2.1), where $W_{ij},\ x_i,\ y_j,\ b$ are the weights, input activation, output before the activation function, and bias respectively. The non-linear function $f(y)$ is added after each neuronal computation such that the network is not just a large, linear operator. Some of the most used non-linear activation functions are the sigmoid (fig. 2.2 and eq. (2.2)), hyperbolic tangent (fig. 2.3 and eq. (2.3)) and Rectified Linear Unit (ReLU) (fig. 2.4 and eq. (2.4)) [69]. From these activation functions, the ReLU is the most used one as it has well-defined gradients even when the input x_i gets large. For comparison, the sigmoid's (fig. 2.2) gradients go towards zero for high and low x_i values as it has an almost flat slope. The ReLU activation always has a large slope for positive values of x_i (fig. 2.4), which often results in better performance.

$$y_j = f\left(\sum_i^n W_{ij} \times x_i + b\right) \tag{2.1}$$

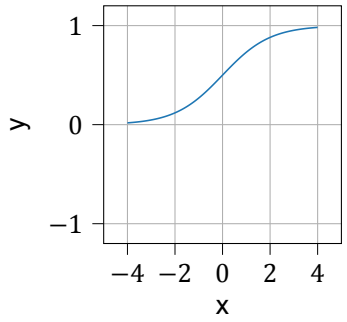


Figure 2.2: Sigmoid

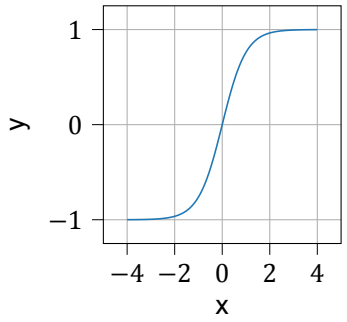


Figure 2.3: Hyperbolic tangent

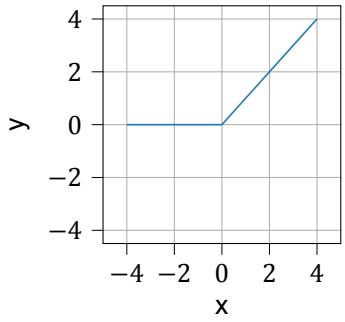


Figure 2.4: ReLU

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.3}$$

$$f(x) = max(0, x) \tag{2.4}$$

Backward Pass

The form of back-propagation used in most current-day ANNs was popularized by Rumelhart et al. [87] and its description is the basis for this section. The task of the back-propagation algorithm is to find the optimal set of connection weights w_i such that the network maximizes its task performance, like classifying cats in a set of pictures. To achieve this, back-propagation minimizes an error signal E. In the case of reconstruction of a signal a suitable loss function is the Mean Squared Error (MSE) loss as in eq. (2.5). Here $\hat{y}_i$ is the network output and y_i the desired output for each output neuron i.

$$E = \frac{1}{2}\sum_1^n (y_i - \hat{y}_i)^2 \tag{2.5}$$

Back-propagation minimizes the error signal E by optimizing the weights w_i through gradient descent. To do this it is necessary to compute the partial derivative of E for each weight w_i. This derivative for the weight is the sum of the derivatives resulting from each possible path along the connections of the network from the output layer to weight w_i. Calculating the derivative of a single path is done by applying the chain rule, multiplying the partial derivatives of all consecutive operations with each other. The resulting weight update is mathematically described in eq. (2.6). The result of the chain rule for all intermittent steps is collected in the term net_i. An example of the calculation process can also be

found in fig. 2.1b. Under this formulation, inter-layer connections and connections that go from higher to lower layers are forbidden, since that would result in an infinite loop in the chain rule.

$$\frac{\delta E}{\delta w_i} = \sum_t \frac{\delta E}{\delta net_i} \frac{\delta net_t}{\delta w_i} \tag{2.6}$$

In recent years back-propagation grew to be the staple learning rule for current state-of-the-art ANNs, yet it also brings certain downsides with it of which the biggest two are mentioned next. Firstly, all operations within a neural network have to be differentiable as otherwise a gradient cannot be calculated. Secondly, due to the separate backward pass for updating the network weights, a network is traversed twice for each example during training, increasing computational cost.

2.1.2. Basic Architectures

The architectural design of neural networks varies greatly depending on the network's objective and the author's creativity and insight into the task. Even so, most of the current day neural networks make use of at least one or more of the following architectural components: fully connected, convolutional, recurrent. Table 2.1 from Battaglia et al. [8] gives an overview of the most important properties of each type of component. Entities are the input elements to the component, the relation is what type of information the component extracts, relational inductive bias is the assumption the component makes on the structure that is present in its input data, and invariance indicates the type of translation the component is insensitive to. Further explanation of each component follows next.

Table 2.1: Basic neural network components and their relational biases. From [8]

Component	Entities	Relations	Relational inductive bias	Invariance
Fully connected	Units	All-to-all	Weak	-
Convolutional	Grid Elements	Local	Locality	Spatial translation
Recurrent	Timesteps	Sequential	Sequentiality	Time translation

Fully Connected

The Fully Connected (FC) layer is the first and simplest type of neural network as shown in fig. 2.1a, which are often also referred to as MLP. In a FC layer each output activation is composed of a weighted sum of all output activations of the previous layer, e.g. all neurons in the preceding layer are connected to all neurons in the following layer [101]. Because all connections carry a separate weight these layers require a significant amount of storage and compute. A fully connected layer is the most general type of layer as it does not make any assumptions about the form or properties of its input. Because of this property, it is often used as the final layer of a classification network.

Convolution

The CNN was first introduced by LeCun et al. [52]. The convolutional operation detects an instance of local patterns anywhere in its input plane. The detection of a particular feature is done using the weight sharing technique introduced in Rumelhart and McClelland [86]. The weighted sum for each output activation is computed using only a small neighborhood of input activations, and where the same set of weights are shared for each output as shown in fig. 2.5 [101]. Because the same set of weights "slides" over the input activations looking for a specific pattern a convolutional layer is invariant to the translation of features and is naturally suited to applications where the data is known to be ordered in a grid-like topology. This property is especially powerful for image processing, where convolutional neural networks are the state of the art [35, 43, 49, 95, 102]. In addition to being translation-invariant the sharing of weights also greatly reduces the number of weights and thus network size and the required amount of computing.

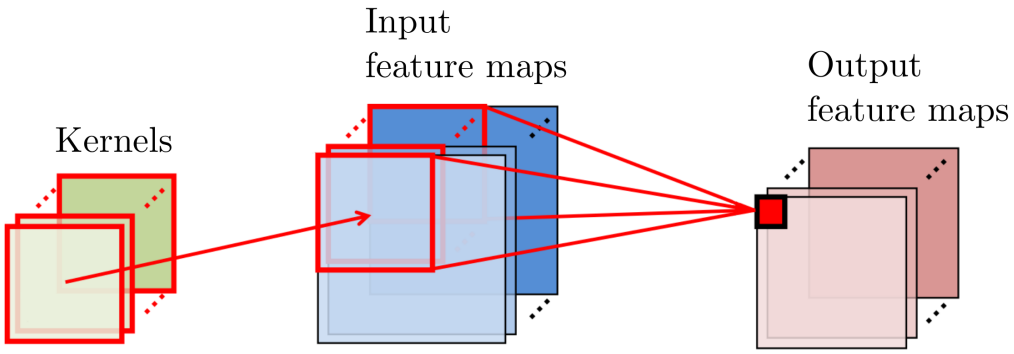


Figure 2.5: Convolution operation in neural network. Adapted from [101] and [77].

Recurrent

Recurrent neural networks are a family of neural networks for processing sequential data [33, 87]. Just like a CNN, Recurrent Neural Networks (RNNs) make use of the concept of weight sharing by applying an operation with the same weights across different parts of the data. The core concept behind a RNN is the addition of a feedback connection to the cells of a neural network that allow information to persist over multiple steps, as illustrated in fig. 2.6. It is because of this persistent state (somewhat like a memory mechanism) that RNNs are very well suited for sequential data processing tasks like natural language processing, speech recognition, image captioning or sensor data processing. Even though RNNs have seen great results lately, the basic model in fig. 2.6 has the issue of often exploding or vanishing back-propagation gradients [9]. As a result, this leads to oscillating weights, long training times, or entire collapsing of the network [40]. Vanishing or exploding gradients are the results of multiplying many very small or very large gradients with each other in the chain rule, leading to weights moving towards zero or becoming very large. To deal with this problem models like the LSTM and Gated Recurrent Unit (GRU) were designed.

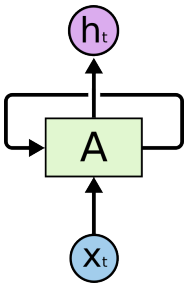


Figure 2.6: Single recurrent neuron. Adapted from Olah [75]

Long Short-Term Memory

The LSTM is the most used, and currently most successful, form of RNN. Its authors state in [40] that the LSTM is designed to deal with the problem of vanishing or exploding gradients [9] and can learn to bridge time intervals over 1000 time steps without loss of short period capabilities. A LSTM has the same structure as a RNN of repeating the same unit over multiple timesteps, but uses a repeating unit consisting of several neurons instead of one. For a LSTM this unit is called a memory cell and is composed of 4 neurons as shown in fig. 2.7, all having a dedicated function.

The four neurons in a cell work together to provide a LSTM cell with both long and short-term memory. Each sigmoidal neuron in fig. 2.7 indicated as σ acts as a gate. The output of the sigmoid operation always falls between [0-1] and when multiplied with a scalar or vector determines which portion of the incoming data is ”passed through” the gate. The key component in a LSTM is the cell-state C_t that flows through the entire chain of cells [75]. Each cell has to ability to scale the incoming cell-state and possibly perform an addition or subtraction. Both of these operations are modulated with a sigmoidal gate, which takes the hidden-state h_t as its input activation. The first gate is called the ”input gate” and the second the ”forget gate”. These respective operations are presented as the

first two gates acting on the cell-state C_t flowing through the chain of cells. The third and final gate modulates the output of a cell and fittingly is called the ”output gate”. This is the rightmost gate shown in fig. 2.7. The part of the cell-state that will be used as output is determined by the sigmoidal gate, whereas the hyperbolic tangent is used to force the cell-state between [-1, 1].

It is because of this ability to control which part and how much of the incoming information a cell remembers while retaining it for many (>1000) timesteps that the LSTM is the go-to RNN module in networks for processing sequential data. Many small variations of the LSTM exist [70, 107]

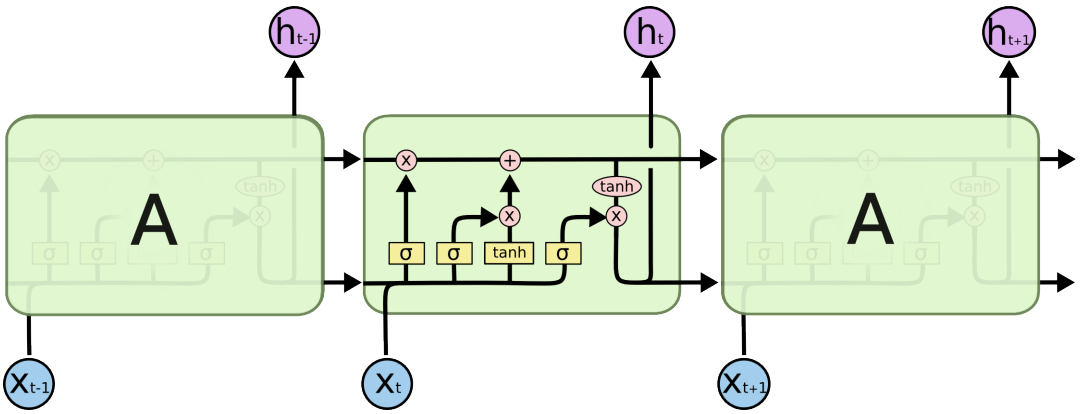


Figure 2.7: Three step LSTM. The σ is a sigmoidal activation function, $\times$ is a pointwise multiplication, $+$ is a pointwise addition, $tanh$ is a hyperbolic tangent, an arrow is a vector transfer, C_t the cell state, h_t the hidden state, x_t the input activation. Adapted from Olah [75]

2.1.3. Neuromorphic Artificial Neural Networks

Recently several ANNs have been designed to perform inference on event-based vision data. This section focuses on how each network processes event-based data and will not consider design choices that lead to the paper’s desired inference patterns.

The first work is called the Phased LSTM by Neil et al. [71]. The PLSTM is a LSTM extended with a time gate. The time gate opens and closes based on a parameterized oscillation. Only during the short open phase the memory cell can perform updates to its state and weights. Even with only the sparse update phase, the PLSTM achieves faster convergence than a regular LSTM. Because of the recurrent structure of a LSTM they can directly operate on asynchronous input data, such as the event stream of a Dynamic Vision Sensor (DVS). The downside of using a LSTM based architecture is the fact that it becomes hard to make use of convolutional architectures. Another downside is the fact that LSTMs are computationally expensive and processing events in a sequential manner can become expansive fast.

The second network is designed by Sekikawa et al. [92]. The focus of this research was to process high rate, event-based data streams that are variable length and spread non-uniform. The authors decided on a two-module architecture, where the first module updates an internal state every time an event is fed into the network. This updating of the state (or latent space) is performed based on the incoming event and its current internal state, making this module recurrent. The actual processing is done using a MLP. The second module is the read-out module which computes the output of the network on-demand, again using a lightweight MLP. To speed up the network at inference time the first event processing module is replaced with an efficient lookup table. This split of event-processing and on-demand output results in a lightweight network that can process up to 10^6 events per second on a standard Central Processing Unit (CPU).

The third network is introduced in Zhu et al. [111]. In this work, data from an event-based camera is used to predict depth, optical flow, and ego-motion in an unsupervised manner. To infer the three types of information the event-based feed is discretized into a frame-based representation. Discretization is performed along the temporal dimension through linear interpolation. This divides the weight of an event that falls between two discretization points based on its distance to each of these points. This method was a step forward from simply binning events over the temporal domain. The code for this work is not available, but for the predecessor of this network, it is available on github[2].

[2]https://github.com/daniilidis-group/EV-FlowNet - accessed 04-08-2019

The last work is presented in Gehrig et al. [23]. Where the work of Zhu et al. [111] performed linear interpolation during discretization, the networks of Gehrig et al. [23] learn the discretization as part of the network. Discretization is still done with equidistant spacing, but the amount an event contributes to the discretization point directly before or after its time of occurrence is a learned non-linear function. The interpolation kernel consists of a two-layer MLPs of 30 units each. This MLP kernel is applied to each event to determine its contribution to the nearest discretization points. After training the MLP is replaced by a look-up table for efficiency. The authors state performance improvements of approximately 12% on optical flow and classification tasks over state-of-the-art event-based networks.

2.1.4. Deep Learning Frameworks and Hardware

A big reason for the improvement in the performance of neural networks was the increasing availability of cheap computational power and large datasets. When implementing an ANN it is in its simplest form a sequence of linear algebra or matrix operations that are very well suited for parallel computation. Due to the rising interest in video games, the computing power of Graphics Processing Units (GPUs) increased significantly. This was also convenient for deep learning since GPUs are designed to perform very large matrix operations in parallel. At the current date especially NVIDIA[3] GPUs are well suited for training ANNs due to mature deep learning libraries relying on their CUDA programming language[4]. More recently Google has been putting effort into designing hardware accelerators specifically designed for deep learning, called Tensor Processing Unit (TPU)[5]. These TPUs are available in their cloud computing service.

Besides hardware, the availability of stable and mature software frameworks has also made it possible for more people to work with deep learning. All of the following frameworks have the Python language[6] as their main scripting Application Programming Interface (API), but often their back-end is written in more efficient languages like C++ and CUDA. The first and most used is TensorFlow by Google[7]. This is the most extended framework providing a large number of tools and predefined functions for writing production-ready ANN and Machine Learning (ML) code. It supports several programming languages including Python, C++, Java, Swift and more. The second most popular framework is PyTorch by Facebook[8]. It was designed as a reaction to the release of TensorFlow and quickly gained traction among both academic and industrial researchers due to its easy and flexible Python API. Built on top of frameworks like TensorFlow, the discontinued Theano[9], Microsoft Cognitive Toolkit[10] and PlaidML[11] is the high level framework Keras[12]. Keras was made to make deep learning accessible and easy by abstracting away a large portion of the low level and complex operations present in most frameworks. Besides being intuitive it is also the most flexible by letting its users choose out of the four previously mentioned frameworks as the engine behind their project. These three frameworks are currently the most used ones, but besides the frameworks supported by Keras the following are also worth mentioning: MXNet[13] by the Apache software foundation and the preferred deep learning framework for Amazon Web Services, Chainer[14] developed by Preferred Network Inc. and written in pure Python.

[3] https://www.nvidia.com/en-us/ - accessed 25-07-2019
[4] https://www.geforce.com/hardware/technology/cuda accessed 25-07-2019
[5] https://cloud.google.com/tpu/ - accessed 25-07-2019
[6] https://www.python.org/ - accessed
[7] https://www.tensorflow.org/ - accessed 25-07-2019
[8] https://pytorch.org/ - accessed 25-07-2019
[9] http://deeplearning.net/software/theano/
[10] https://docs.microsoft.com/en-us/cognitive-toolkit/ - accessed 25-07-2019
[11] https://github.com/plaidml/plaidml - accessed 25-07-2019
[12] https://keras.io/ - accessed 25-07-2019
[13] https://mxnet.apache.org/ - accessed 25-07-2019
[14] https://chainer.org/ - accessed 25-07-2019

2.2. Spiking Neural Networks

Spiking neural networks are more closely inspired by how neurons in animals work compared to the ANNs described in section 2.1 and thus is an interest shared by many neuroscientists as well as artificial intelligence researchers. Some researchers even consider them the ”third generation of neural networks” [59]. Spiking neural networks mainly differ from ANNs in that they have a notion of time due to maintaining an internal state from time step to time step. It is because of this that the theoretical efficiency of SNN neurons is higher than that of an ANN [59]. Yet, in practice ANNs still outperform SNNs on most tasks. The second important difference is that SNNs neurons communicate data between each other by sending trains of discrete spikes (also called action potentials), in contrast to the scalar values send between neurons from ANNs. There is still a discrepancy between the theoretical and practical power and efficiency of SNNs and ANNs, which is mainly caused by the fact that SNNs are harder to train. Because SNNs communicate through discrete spikes the network operations are not differentiable and the back-propagation algorithm from section 2.1 cannot be used for training. Thus, searching for stable learning rules is one of the main priorities in this field of research.

Firstly, a more in-depth biological background on SNNs is provided. Secondly, the most common and important neuron models are discussed. Thirdly, the most used learning rules are presented. Lastly, the chapter is concluded with an overview of neuromorphic hardware that can be used to efficiently simulate SNNs.

2.2.1. Biological Background

The first mathematical model (and generation) of a neuron was presented in McCulloch and Pitts [62], its graphical representation can be found in fig. 2.8. This simple neuron sends out a binary signal of either 0 or 1, depending on whether its summed input signal has a value that surpasses its firing threshold. The McCulloch Pitt neuron was inspired by the binary signals of animal neurons. These started the so-called ”first generation of neural networks models”.

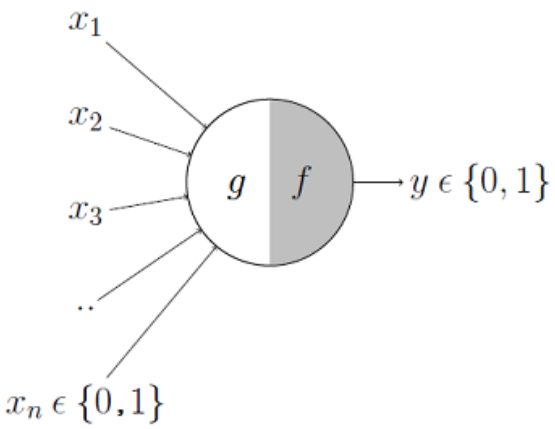


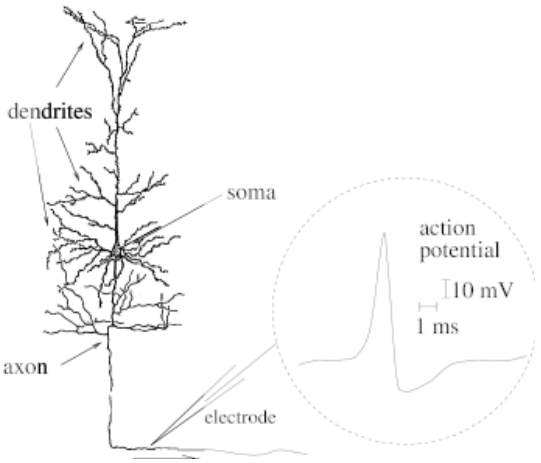
Figure 2.8: Overview of the McCulloch Pitts neuron, it only receives and sends out binary signals. Adapted from Lagandula [50]

As previously mentioned, SNNs are sometimes considered the third generation of neural network models. Out of all the three generations, the SNN models are the most biologically plausible. A general description of the biological origin of SNNs is adapted from the thorough overviews and explanations in Gerstner [26].

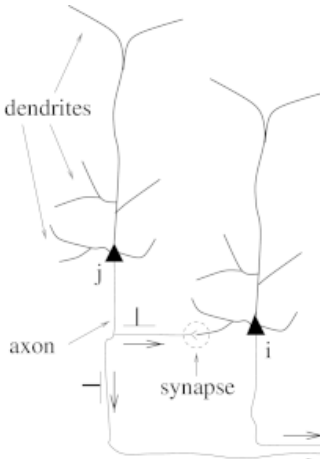
A biological spiking neuron consists of three distinct parts as seen in fig. 2.9a, the dendrite (input device), the soma (cell body or computing node) and the axon (output device). Connections between an axon and a dendrite are called *synapses* as in fig. 2.9b. The neuron that is sending a signal is called the *pre-synaptic* cell, the neuron that is receiving the signal is called the *post-synaptic* cell. The soma’s main characteristic is its *membrane potential* $v(t)$, which is the voltage difference between the cell’s internals and its surroundings. If the cell receives no input signals it is at a constant resting potential v_{rest}. Once the soma receives enough input signals such that its membrane potential exceeds a threshold called the *firing threshold* θ the neuron generates an output *action potential* (or *spike*) and propagates it through its dendrites to the connected post-synaptic neurons. After a neuron has emitted

a spike it enters a *refractory period* during which it cannot spike and resets its cell voltage to its resting state. A short period after its refractory state a neuron enters a phase of relative *refractoriness* where it is more difficult to excite. Each neuronal connection (synapse) can either have a positive or negative impact on the membrane potential of its post-synaptic neurons. These synapses are accordingly called *excitatory* or *inhibitory* synapses.

The neuronal signal consists of short electrical pulses as seen in fig. 2.10, called an action potential or spike. Each spike has an amplitude of about $100mV$ and a width of about $1 - 2ms$, and the form of the pulse does not change while propagating along an axon. Since the shape of the spikes does not differ, all information is carried in the time of arrival at the post-synaptic neuron. Multiple consecutive spikes are called a spike train. Spikes within a single output spike train never overlap even in case of a very active neuron. All ANNs ("second generation") make use of a rate-encoding scheme, where the values passed between neurons represent the firing rate instead of the exact firing time as in real neurons [2, 3]. Spiking neural networks have the option of using both a rate and/or timing encoding scheme.



(a) Schematic representation of a single spiking neuron. Adapted from Gerstner [26]



(b) Schematic representation of a signal passing between two neurons. Adapted from Gerstner [26]

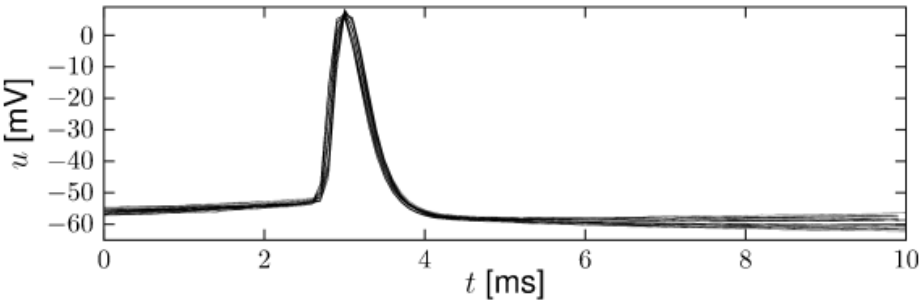


Figure 2.10: Neuronal spike. Adapted from Gerstner [26]

2.2.2. Neuron Models

Choosing a neuronal model for use within a SNN often is one of the first to make and can have a large impact on the functioning of the network, as well as which learning rule should be applied (see section 2.2.3). Most of the properties that biological neurons exhibit have been adapted into several mathematical models. Depending on the model, certain properties have been left out and the representation of each property can differ too.

The most biologically precise and general neuron model is the "Hodgkin-Huxley model" [41]. Named after its authors, they performed experiments on the giant axon of a squid and were able to both measure and influence the action potentials running through the axons. Besides measuring, they captured

the dynamics in a set of differential equations that to date are still considered the most accurate model out there. For their work, Hodgkin and Huxley received the Nobel prize in Physiology or Medicine in 1963. Despite its accuracy, the model is too complex to be feasible for use in large scale simulations. Because of these characteristics, this model serves as the starting point for most neuron models that try to retain as much of its dynamics while lowering the computational burden.

This section first covers the mathematically simplest leaky integrate-and-fire model. This is followed by the spike response model, and the section concludes with the Izhikevich model. For more unconventional models the reader is referred to the following works: Ahmed et al. [4], Florian [21], Nessler et al. [72], Shrestha et al. [93]

Leaky Integrate-and-Fire Model

The most commonly used neuron model is the Leaky Integrate-and-Fire Neuron (LIF). Its predecessor, the integrate-and-fire, was introduced by Brunel and van Rossum [14]. The defining assumption in the model is that real neuronal spikes have a consistent shape and their information is encoded in the exact timing or absence of a spike. The simplest way to still encode information is then to model spikes as single events using the *Kronecker-delta function* $\delta(t)$. As the name integrate-and-fire suggests, the model integrates incoming spikes until its membrane potential surpasses the firing threshold θ from below, it fires a spike, resets its membrane potential to v_{rest}, and enters a refractory period. Equation (2.7) shows an addition of a decay term to the Integrate-and-Fire Neuron (IF) model by Stein [98] based on elementary electrical laws. I is the incoming current from the pre-synaptic neurons, R is a linear resistor, C is a capacitor modeling the capacity of the membrane and $\tau_v = RC$ is the voltage-time decay constant. Because of the formulation of the membrane potential as a linear differential equation, over time the membrane potential will exponentially move towards its resting potential v_{rest}. Additionally, the model may incorporate an absolute refractory period during which the neuron cannot change its state or fire (described in section 2.2.1).

$$\tau_v \frac{dv}{dt} = -[v(t) - v_{rest}] + RI(t) \tag{2.7}$$

Spike Response Model

Gerstner and van Hemmen [28] introduced the Spike Response Model (SRM) as a generalization of the LIF. It differs from the LIF in two ways. Firstly, the model makes use of a different mathematical description, instead of the differential equations as in the LIF model the SRM makes use of parametric functions of time called *signal response functions* or *kernels*. All pre and post-synaptic spikes in the past time are convolved with a separate kernel respectively before their influence on the membrane potential is calculated. Secondly, a SRM can include a phase of relative refractoriness (described in section 2.2.1). Each post-synaptic spike influences the membrane potential of the signaling neuron by convolving it with a refractory kernel η. Action potential generation in a SRM is the same as with a LIF, where an action potential is generated when the membrane potential crosses a firing threshold from below.

Equation (2.8) describes the membrane potential of a SRM neuron i at time t, with the neuron's last post-synaptic spike at $\hat{t}$. To determine the influence of presynaptic spikes over time the SRM sums over all past pre-synaptic spikes from pre-synaptic neurons j and convolves them with kernel ϵ, after which it multiplies the spikes from each synapse with their respective synaptic weight w_{ij}. Next, the model expresses the influence of its post-synaptic spikes on its membrane potential through kernel η. This kernel can capture both the absolute and relative refractory periods (section 2.2.1), allowing for non-spiking periods and periods of increased spiking difficulty. This is what sets the SRM apart from the LIF. Lastly, the model allows for the incorporation of external stimulation of the membrane potential in the form of the integral over I convolved with the kernel κ. For most simulations, this term is likely zero. The response functions (or kernels) $\eta,\ \epsilon,\ \kappa$, can take many forms and are open to its user to decide on which form suits their needs best. It is because of these additions that the SRM is the most general linear neuron model.

$$v_i(t) = \sum_f \eta(t - t_i^{(f)}) + \sum_j w_{ij} \sum_f \epsilon_{ij}(t - \hat{t}_i, t - t_j^{(f)}) + \int_0^\infty \kappa(t - \hat{t}_i, s) I(t - s) ds + v_{rest} \tag{2.8}$$

Izhikevich Neuronal Model

The Izhikevich neuronal model [44] is a biologically plausible and precise neuron model that is heavily inspired by the Hodgkin-Huxley model, yet its computational complexity is comparable to that of the LIF model. Due to these properties, this neuron model is well suited for high-fidelity simulations of animal brains. Izhikevich [44] reduces the complex Hodgkin-Huxley model to a 2D system of Ordinary Differential Equations (ODEs), as in eq. (2.9) and eq. (2.10). Here, v and u are dimensionless variables, and a, b, c, and d are dimensionless parameters. The variable v represents the membrane potential and u represents a membrane recovery variable. After the spike reaches its peak ($+30mV$), the membrane voltage and the recovery variable are reset according to the eq. (2.11). Synaptic currents, or injected dc-currents, are delivered via the variable I.

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{2.9}$$

$$\frac{du}{dt} = a(bv - u) \tag{2.10}$$

$$\text{if } v \geq 30mV, \text{ then} \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{2.11}$$

The parameter a describes the time scale of recovery variable u, smaller values result in slower recovery. The parameter b describes the sensitivity of the recovery variable u to the subthreshold fluctuations of the membrane potential v. Greater values couple v and u more strongly resulting in possible subthreshold oscillations and low-threshold spiking dynamics. The parameter c describes the after-spike reset value of the membrane potential v. Lastly, the parameter d describes the after-spike reset of the recovery variable u.

2.2.3. Learning Rules

Possibly the most important algorithm in making a neural network perform well is the learning algorithm. Whereas for ANNs backpropagation (section 2.1.1) is the staple algorithm for almost every implementation, for SNNs there is no clear go-to algorithm. As mentioned in section 2.2, signals in the forms of spikes are non-differentiable and thus derivative-based algorithms are not usable without certain assumptions or modifications to spiking neurons. This forces its users in the direction of correlation-based learning rules. First, the unsupervised learning algorithm STDP is treated. This is followed by a discussion of an augmented form called Reward-Modulated Spike-Timing-Dependent Plasticity (R-STDP), and the section is concluded with an overview of supervised learning rules for SNNs.

Spike-Timing-Dependent Plasticity

The most used learning algorithm for SNNs is called Spike-Timing-Dependent Plasticity (STDP), which is based on Hebb's postulate:

"Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability. ... When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased. [37]"

The following description of STDP has been adapted from Sjostrom and Gerstner [96]. For more information, reviews, and overviews of STDP the reader is referred to Abbott and Nelson [1], Sejnowski and Tesauro [91], Sjostrom and Gerstner [96].

STDP is an *unsupervised* and *local* learning rule. The concept of unsupervised learning is described in section 2.1.1. Local means that the weights update Δw is purely based on information present in and between the pre- and post-synaptic layers of neurons. The weight changes Δw_j of a synapse from a pre-synaptic neuron j depends on the relative timing between pre-synaptic spike arrivals and post-synaptic spike times. A connection weight is increased (decreased) if the pre-synaptic spike arrives at the post-synaptic cell just before (after) the pre-synaptic spike takes place. The further the spikes are apart in time, the smaller the weight update in absolute value. Because the weight update trends towards zero for large separations in time the update is also called the learning window. Pre-synaptic

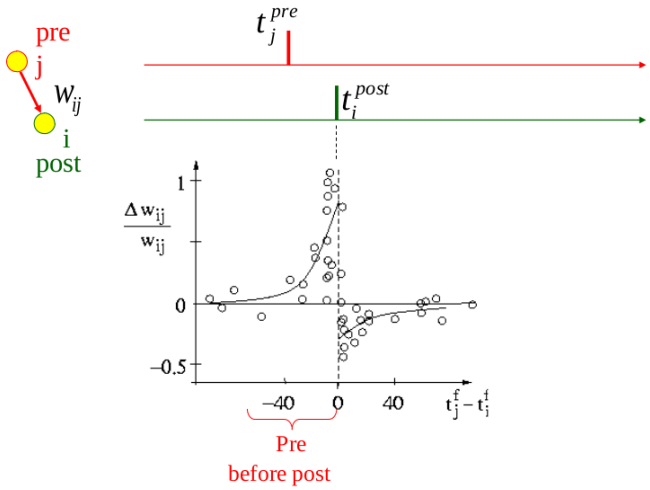


Figure 2.11: STDP proportional weight scaling based on relative timing of pre- and post-synaptic spikes. Adapted from Sjostrom and Gerstner [96].

arrival times at synapse j are indicated as t_j^f where $f = 1, 2, 3, \ldots$ is the count of pre-synaptic spikes and similarly n is the count of post-synaptic spikes. Such a set of spikes for a single synapse is also called a *trace*. The total weight change Δw is determined by eq. (2.12), where $W(x)$ denotes one of the STDP functions illustrated in fig. 2.11. The x-axis depicts the difference in time between the post and pre-synaptic spike $t_j^f - t_i^n$, positive values mean the pre-synaptic spike occurred after the post-synaptic spike. As mentioned previously, this should result in a negative weight update, as is shown on the y-axis. The weight updates are strongest when the spikes occur near each other in time. Also, positive and negative weight updates can be of different magnitude.

$$\Delta w_j = \sum_{j=1}^{N} \sum_{n=1}^{N} W(t_i^n - t_j^f) \tag{2.12}$$

A popular choice for the STDP function is shown in eq. (2.13) and eq. (2.14), which has been used to fit experimental data [109]. The parameters A_+ and A_- can be dependent on the synaptic weight w_j and the time constants τ_+, τ_- are in the order of 10ms. Equation (2.13) and eq. (2.14) are the most common learning window functions, but other forms can be used.

$$W(x) = A_+ e^{-x/\tau_+} \quad for\ x > 0 \tag{2.13}$$

$$W(x) = -A_- e^{x/\tau_-} \quad for\ x < 0 \tag{2.14}$$

Many variations and extensions to the basic STDP formulation above exist. An interesting and often made addition is to make the algorithm perform online weight updates. This is based on the assumption that each presynaptic spike arrival leaves a trace $x_j(t)$ which is updated by an amount $a_+(x)$ at the moment of spike arrival and decays exponentially in the absence of spikes according to eq. (2.15). The same logic holds for the post-synaptic trace y, the trace increases by an amount $a_-(y)$ at the moment of a post-synaptic spike according to eq. (2.16).

$$\tau_+ \frac{dx_j}{dt} = -x + a_+(x) \sum_j \delta(t - t_j^f) \tag{2.15}$$

$$\tau_- \frac{dy}{dt} = -y + a_-(y) \sum_j \delta(t - t^n) \tag{2.16}$$

The total weight change is given by eq. (2.17). This shows that the weight is increased by an amount proportional to the pre-synaptic trace x and depressed proportional to the post-synaptic trace y.

$$\frac{dw_j}{dt} = A_+(w_j)x(t)\sum_n \delta(t - t^n) - A_-(w_j)y(t)\sum_f \delta(t - t_j^f) \tag{2.17}$$

There is a relatively large number of variations of STDP, but the most important discerning factor between different rules is whether the rule is multiplicative or additive [29, 105]. Additive STDP, as shown in eq. (2.18), is mathematically the simplest form where the weight update Δw only depends on the relative timing of the pre- and post-synaptic spikes. The advantage of this method is its ability to quickly change synaptic weights in the desired direction, but with the downside that this causes bi-modal distributions. Since STDP reinforces already strong connections and depresses already weak ones it forces the weights to their upper or lower boundaries. In comparison, multiplicative (also called weight dependent) STDP, takes the current strength of the connection into account in its weight updates. An example of such a rule is taken from Paredes-Vallés et al. [78] and simplified for illustrative purposes. The resulting weight update is shown in eq. (2.19). Here the $e^{-|w_t - w_{init}|}$ term scales Δw, with w_{init} acting as the reference weight value. Weight updates for already strong connections are smaller than for weak connections under the assumption that all other variables are equal. This causes the method to converge slower, yet it can satisfy a uni-modal distribution and thus it is better at guaranteeing convergence by keeping the synaptic weights bounded. It should be noted that more advanced versions are possible. For example, an adaption can be made where weight update towards w_{init} are large, but small in the other direction.

$$w_{t+1} = w_t + \Delta w_t \tag{2.18}$$

$$w_{t+1} = w_t + \Delta w_t e^{-|w_t - w_{init}|} \tag{2.19}$$

As a last note, almost all of the modern implementations of STDP within an engineering setting only allow for adjusting synaptic weights. Parameters that have a significant influence on the functionality of a network like the transmission time or firing threshold of a cell are almost always considered to be fixed or tuned through hyperparameter optimization. It appears that these parameters in animal brains can be learned just like the synaptic weights. Research providing evidence for the adaptive learning of firing thresholds in the human brain can be found in Daoudal and Debanne [17], Zhang and Linden [110]. Further research on this topic might result in improved learning performance.

Reward Modulated Spike-Timing-Dependent Plasticity

One of the biggest outstanding problems within training SNNs is the fact that STDP is purely unsupervised, while there are no stable and reliable algorithms that allow supervised learning without gradient-based techniques. Luckily, progress is being made towards this goal in the form of Reward-Modulated Spike-Timing-Dependent Plasticity (R-STDP), which in its simplest form is a combination of reinforcement learning and STDP. In its essence reinforcement learning can be defined as goal-directed learning from interaction [99], or put even simpler, learning through trial and error. This interaction can be with other humans, animals, complete environments, simulations or even single reward functions. More information on reinforcement learning can be found chapter 3.

One of the first successful and well known scientific experiments towards organism level reinforcement learning is research done by Pavlov [79]. Besides behavioral experiments research has also been conducted towards the presence of reinforcement learning like operations in animal brains. The works of Bartlett and Baxter [7], Reynolds and Wickens [83], Schultz [88], Schultz et al. [90] show experimental support for the presence of R-STDP in the brain. It currently is widely accepted by researchers that R-STDP plays an important role in the learning of animals and for an overview of the modern view of biological R-STDP the reader is referred to [80, 89]. The precise functioning of R-STDP in the brain is still unknown, and from an engineering perspective, no algorithm is considered precise and effective. Because of this, there is a lot of room for improvement, and improvements might open up ways of implementing both reinforcement learning and supervised learning methods in SNNs. For the remaining of this work, the focus will be on R-STDP from an engineering perspective.

As mentioned before, R-STDP can theoretically be used to perform both reinforcement learning and supervised learning in spiking neural networks. Using RL in a SL setting can be as simple as restating the reward signal to have a target reward value it has to attain. Xie and Seung [106] where the first to formally state R-STDP's usefulness in doing so. One of the first formulations of R-STDP for an engineering application was presented in Florian [20] and is to the current date still quite influential. This work together with Gerstner [26] will be taken as a reference and used to explain the basics of R-STDP.

Reward-modulated STDP, as its name suggests, is simply modulating STDP based on a reward or error signal such that the network learns based on maximizing (minimizing) its reward (error) signal. Regular STDP weight updates are based on a Hebbian correlation term $H(pre_j, post_i)$ between the pre- and post-synaptic trace, like in eq. (2.12), eq. (2.13) and eq. (2.14). As an addition to the Hebbian term, R-STDP adapts the weight update according to a modulation signal M, resulting in eq. (2.20). One of the more common modulation signals is 'reward minus expected reward' as in eq. (2.21). This signal has the nice property of "forcing" the network's output towards the desired output, analogous to weight updates done by backpropagation in ANNs.

$$\Delta w_j = \sum_{j=1}^{N} \sum_{n=1}^{N} M \cdot W(t_i^n - t_j^f) \tag{2.20}$$

$$M(t) = R(t) - ER \tag{2.21}$$

Within literature, variations on this algorithm exist [20, 67, 68]. As far as the author's knowledge goes, these implementations make changes in the Hebbian correlation function, but the scaling of the Hebbian correlation through multiplication with the modulation signal M is retained in all these works.

Changing the actual impact of the modulation signal on the weight updates could be an interesting research avenue since research in the field of neuroscience shows that the functioning and shape of the STDP window are changed through reward modulation with e.g. the neuromodulator dopamine[15] [80]. Also, Frémaux et al. [22] showed that a reward signal that on average is not zero (e.g. there is a baseline reward value that the network will receive) will induce a bias in the R-STDP rule towards performing regular STDP. This originates from the fact that R-STDP guides regular STDP weight updates Δw_{ij} in the correct direction based on a reward signal. As long as the average reward is around zero the method has an equal capability of potentiating (positive reward) or depressing (negative reward) the synaptic connections. The further the average reward shifts away from zero, the more the general orientation of the weight updates due to high and low rewards remains the same. This means that the variations in the weight updates are mostly due to differences in the STDP term, thus expressing a bias towards unsupervised weight updates even when using R-STDP. Multiple methods exist to alleviate this problem, which can be as simple as subtracting the average reward. This is only possible for a single task as the reward might differ per task. A more robust and elegant solution would be to use an actor-critic algorithm. In this case, the critic learns to predict the expected reward for the next time step. In place, the critic tells the network which weight updates it has to perform. This intermediate step of the critic both removes the bias towards unsupervised learning updates and reduces the variance in the weight updates [22].

Gradient Based Learning

Multiple methods exist that approximate gradient based learning in SNNs [13, 94, 103, 104, 108]. In order to obtain an approximate derivative of a spiking signal there are generally two kinds of assumptions made in literature [108].

The first is to perform optimization based on the membrane voltage of a neuron instead of the spiking signal it generates. The voltage is relatively easy to approximate with a differentiable function. A good example of this are BP-STDP [103], and SpikeGrad [104]. Both of the works approximate the membrane voltage of an IF neuron as the activation value of the ReLU (eq. (2.4)) operation. This relies on the fact that a *non-leaky IF neuron* integrates its incoming action potentials. Because the neuron always needs an equivalent amount of incoming potential its output activity over time scales linearly with the incoming potential, as long as it surpasses its firing threshold.

[15] https://www.medicinenet.com/script/main/art.asp?articlekey=14345 - retrieved 18-07-2019

The second type of method does perform optimization based on the output signal of a neuron, by approximating the spike generation mechanism with a differentiable function. The first work to do this was SpikeProp [13] by linearizing the relationship between the postsynaptic input and the resulting spike timing. The Spike Layer Error Assignment Rule (SLAYER) algorithm [94] approximates the gradient of a spiking signal as the gradient of a Probability Density Function (PDF) that expresses the probability of the neuron changing its firing state based on its current membrane potential. The work of Zenke and Ganguli [108] combines both a stochastic firing neuron to generate a differentiable distribution over the spike timing and optimization directly on the membrane voltage of the neuron.

In order to try and leverage the hardware and algorithms available for ANNs completely, several works have converted a fully trained ANN to a SNN in order to gain the efficiency and energy benefits of a SNN [18, 74]. This was done by replacing the activation function (section 2.1.1) with spiking neurons and performing parameter optimization. These types of algorithms will not be discussed further.

2.2.4. Supervised Learning Comparison

Finding a supervised learning rule for SNNs that is somewhat comparable to backpropagation is highly desired by the SNN community. Two main approaches currently allow for supervised learning. The first allows for using backpropagation by using approximations and assumptions to make the blocking SNN operations differentiable. The second is based around designing a supervised, correlation-based learning function. This section will go over and compare the available supervised learning algorithms for SNNs.

The first supervised learning rule to be considered is Remote Supervised Method (ReSuMe) [81]. ReSuMe is based on the concept of Supervised Hebbian Learning (SHL), extensively analyzed in Legenstein et al. [54]. SHL revolves around stimulating neurons through the injection of an external voltage to make the neurons spike at desired times. The downside of this method is that it does not depress connections leading to unwanted activations. To resolve this ReSuMe potentiates synaptic connections that show a correlation between pre-synaptic spikes and a pre-defined desired post-synaptic spike train. To account for unwanted post-synaptic spikes ReSuMe always depresses connections that show a correlation between pre-synaptic spikes and the actual post-synaptic spikes generated by the neuron. The potentiation (based on the desired signal) and depression (based on the actual signal) cancel each other out when the signals match, resulting in a weight update equal to zero. The biggest shortcoming of the ReSuMe algorithm is that it needs a pre-defined desired signal for every neuron in the network. This often is impossible to determine as only the output for the final layer is known.

A different and more recent work from Mozafari et al. [68] applies a combination of R-STDP and STDP for supervised classification. The authors use a convolutional encoding network for image classification. Each class is assigned a single output neuron and the first output neuron to spike indicates the network's prediction. The network uses regular STDP for its first layers (2 in the experiments) and R-STDP for the final layers (1 in the experiment). R-STDP weight updates are only applied to the output neuron that spikes first, ignoring the other neurons, and performs potentiation (depression) in case of a correct (incorrect) classification. The work is quite simple and shows the possible efficiency and capabilities of R-STDP in supervised learning by outperforming STDP methods. On the other hand, the presented implementation is quite simplistic and restricted to just classification as it performs R-STDP based on just the first output spike.

As mentioned in section 2.2.3, the first real R-STDP formulation was presented in Florian [20] (called modulate spike timing dependent plasticity) and is still considered one of the better performing algorithms. The algorithm used eligibility traces (see chapter 3 for more information) as a kind of memory for recent neuronal activity. Weight updates were a summation of two correlations, the pre-synaptic trace P_{pre} and a post-synaptic spike f_{post}, and the post-synaptic trace P_{post} and a pre-synaptic spike f_{pre}. Furthermore, regular modulation of the weight updates with a reward signal r of +1 or -1 was used. Equation (2.22) shows the definition of the weight update.

$$\Delta w = r(P_{pre} f_{post} - P_{post} f_{pre}) \tag{2.22}$$

Within the available gradient-based methods several variations exist in how many assumptions and approximations are made. From the perspective of keeping the algorithm mathematically sound and general the lesser is assumed or approximated, the better. Both from this perspective, as well as published performance quantitatively and qualitatively the SLAYER algorithm presented in Shrestha and Orchard [94] as discussed in section 2.2.3, seems like the best choice to the author. Aside from

the approximation of the spike gradient with the slope of a suitable PDF, the algorithm does not make any further assumptions. It can assign errors back through time and its source code is available as a library built on PyTorch (see also section 2.2.6). Other methods like Backpropagation Spike-timing-Dependent Plasticity (BP-STDP) [103], SpikeProp [13], SpikeGrad [104], or SuperSpike [108] are more confined in their possible applications. The first two works are restricted to using purely non-leaky IF neurons, whereas the last work is more flexible in the type of neurons but makes multiple assumptions about the generation of a spiking signal and its derivative. Both are reasons to favor the SLAYER algorithm.

2.2.5. STDP and Back-Propagation

Regardless of the incompatibility of back-propagation with SNNs (see section 2.2.3), many researchers have been trying to reconcile the two through approximations or adjustments to SNNs. The reason for this is very well explained by the following quote from Lillicrap and Santoro [57]:

> "Ultimately we expect that agents and animals alike will not adhere to strict formulations of Backpropagation Through Time (BPTT). This does not imply that BPTT should not remain a canonical guide to Temporal Creddit Assignment (TCA); even when full differentiation through time is not possible, innovation should be guided towards its approximation, and progress should be gauged with the bar set by its hypothetical possibility." [57]

The ideal situation would be to provide the power of back-propagation through an algorithm that is directly applicable to SNNs. Whether that is even possible remains an open question, yet some research indicates that the methods share certain similarities. The first is that STDP resembles a temporal derivative filter [46], providing its user with the time derivative of its signal. This exact concept is also used in back-propagation, but in that case, it is for the error signal. In the presentation of Hinton [39] it is stated that the inspiration for back-propagation was attained from exactly this concept. A more mathematical approach to comparing STDP with back-propagation is presented in Bengio et al. [11]. The work is still very exploratory but might provide an interesting avenue for future research.

Possibly the most important difference between the two methods is also stated by Lillicrap and Santoro [57]. It is the fact that back-propagation makes use of *explicit computations* that are highly precise. The influence of a single parameter is exactly known due to the deterministic functions used. On the other hand STDP is a *correlation* based method which can have high variance, especially if the network becomes large. Under the assumption of having enough samples, the correlation can be computed quite accurately, converging to the actual and thus desired value. In that case, the correlation can be a decent expression of how much a parameter influences the error signal. The concept of minimizing variance is also used in modern variations of the back-propagation algorithm, specifically all methods based on Stochastic Gradient Descent (SGD). The batching of weight updates originating from multiple training samples lowers the variance of the combined weight updates. This makes it more likely that the update is performed in the desired direction within the parameter space of the network, and thus speeding up the convergence of training.

The second important difference is that performing supervised learning within a SNN is considerably less reliable than with a ANN. Propagating an error derivative, or a surrogate, in a SNN is still an active research field and can currently only be done indirectly. This includes tuning firing thresholds of presynaptic neurons (see section 2.2.3), or multiple forms of R-STDP (see section 2.2.3). In Baldi and Sadowski [6] it was shown that purely local STDP cannot learn to extract specific or desired patterns. For correlation-based methods to be able to do so they require some form of a backward channel that tells each neuron in which direction its weights should move to reduce its contribution to the error. Preferably this also includes the magnitude of the weight update, in which case the rule starts to converge to acting like backpropagation.

One of the biggest downsides of back-propagation (and even more so recurrent networks) are exploding or vanishing gradients often occurring after many successive steps of the chain rule. As discussed in section 2.1.2, there are many methods that try to deal with this issue through smart architectural design. Another good example that has not been mentioned before is the use of skip-connections that were first successfully used in He et al. [35]. The resulting Resnet is still considered a State of the Art (SOTA) network since its introduction in 2015. The local property of STDP based learning rules (see section 2.2.3) makes it that they do not suffer from this problem as the weight updates are only dependent on the activity of the pre and post-synaptic layers.

It can be concluded that back-propagation through time and STDP share certain important characteristics, mainly the fact that they use the temporal derivative in their computations. But, for STDP to achieve performance comparable to that of back-propagation, researchers need to do two main things. Find methods that can consistently assign errors to the correct neurons in previous layers and achieve low variance weight updates while remaining relatively sample efficient.

2.2.6. Neuromorphic Hardware and Software

To date most of the simulations and implementations of SNNs are performed on von Neumann architecture machines, which is the architecture used in most modern-day computers [85]. Even though some interesting results have been produced, this architecture is not well suited for simulating SNNs because of the limited number of processing units compared to neurons in a network and the transferring of data between the Random-Access Memory (RAM) and CPU/GPU. Inspired by animal brains and their enormous amount of separate computational units in the form of neurons (the human brain contains about 10^{11} neurons [45], and about 10^{15} synapses), researchers have set out to design chips closer to how animal brains function. These chips are naturally very well suited for running SNNs while also being several orders of magnitude more efficient when it comes to power consumption per calculation. On top of all this, most of these chips also support asynchronous, parallel operations. In short, they are an ideal candidate for running SNNs.

A non-exhaustive list of some of the most well known neuromorphic chips follows:

- **Loihi**[16] is a neuromorphic chip designed and made by Intel.
- **TrueNorth**[17] is a neuromorphic chip designed and made by IBM.
- **SpiNNaker**[18] is a neuromorphic chip designed by the University of Manchester.
- **Braindrop**[19] is a neuromorphic chip designed by Stanford university.
- **DYNAP**[20] is a neuromorphic chip designed and made by aiCTX.

Several different software frameworks and simulators are freely available to construct, train, and simulate SNNs. There are quite some differences between many of them, some support GPU acceleration, some are intended to precisely simulate biological processes in brains, while others are intended as an extendable engineering framework. In the current context a *simulator* is considered to be high level, allows for easy and fast implementation of conventional neural network architectures but on the downside is not very flexible and does not support the implementation of novel concepts out of the box. An engineering framework, on the other hand, requires more effort to define a complete neural network due to exposing mainly low-level functionalities and operations to its user. This does provide the advantage of having fine-grained control over the neuron and network design, making it well suited for experimentation through implementing novel concepts and fine-tuning the efficiency of each network. The following is a list of some of the most used simulators and frameworks and their defining characteristics:

- **BindsNet**: BindsNet[21] is based on the PyTorch[22] deep learning platform. BindsNet is a SNN simulation library that is geared towards machine learning and reinforcement learning applications. It supports simulation on both CPU and GPU. BindsNet places itself between a simulator and an engineering framework, making it relatively quick at simulating small-scale networks and making use of some of the benefits PyTorch provides. The downside is that implementing networks that don't fit well to the predefined simulation pipeline can require quite some modifications to the source code.

[16] https://www.intel.com/content/www/us/en/research/neuromorphic-computing.html
[17] http://www.research.ibm.com/articles/brain-chip.shtml
[18] http://apt.cs.manchester.ac.uk/projects/SpiNNaker/
[19] https://neuroscience.stanford.edu/events/braindrop-mixed-signal-neuromorphic-system-presents-clean-abstractions-kwabena-boahen
[20] https://aictx.ai/technology/
[21] BindsNet documentation at https://bindsnet-docs.readthedocs.io/
[22] PyTorch documentation at https://pytorch.org

- **Brian**: The Brian simulator[23] is written in pure Python and designed to be an easy to use and accurate simulator of spiking neural networks. Because it is a simulator it is more suited towards research on biological SNNs and less so for ML research towards SNNs. Also, the lack of GPU acceleration is a downside in the case of large datasets.
- **SpykeTorch**: SpykeTorch[24] is a SNN framework built on top of the PyTorch deep learning framework. Compared to BindsNet, SpykeTorch is more a framework and makes more use of PyTorch conventions and GPU acceleration. This allows its users more freedom in designing their networks. A big downside of the framework is the infrequent updates and the immature state of the package, limiting its current use cases.
- **cuSNN**: The cuSNN library[25] is a C++ library for the design and simulation of SNNs introduced in Paredes-Vallés et al. [78]. It separates itself from the other frameworks in that it is written in the potentially highly efficient CUDA[26] platform. This allows for strong GPU acceleration that is specifically tailored towards SNN simulation, whereas the PyTorch based frameworks rely on functions optimized for regular deep learning. One of the downsides of the library is that it is written using the C++ language (same language used in CUDA). Even though the language is known for its speed, it is a low-level language compared to Python, requiring more lines of code for the same result and its user has to be aware of potential memory and runtime issues.
- **SLAYER**: The SLAYER framework[27] was introduced accompanying the Shrestha and Orchard [94] paper. The framework was originally written in C++/CUDA, but currently also supports a PyTorch wrapper. The library is aimed at training SNNs using the well-established backpropagation algorithms present in PyTorch by making a SNN differentiable by approximation, as is done in the paper.

[23] Brian documentation at http://briansimulator.org/
[24] SpykeTorch framework documentation at https://github.com/miladmozafari/SpykeTorch
[25] cuSNN framework documentation at https://github.com/tudelft/cuSNN
[26] https://developer.nvidia.com/cuda-zone
[27] SLAYER documentation at https://github.com/bamsumit/slayerPytorch

3

Reinforcement Learning

Learning by doing is something everyone is familiar with and for many people even the preferred way of learning. A young child learns to coordinate its body while playing, an athlete perfects his movements over thousands of hours of practice, and a chess grandmaster can judge a board state in the blink of an eye based on the millions of situations he or she has seen and played before. Besides using reinforcement learning on the scale of a complete entity, the human brain also makes use of reinforcement learning on a micro-scale. In the case where a specific condition is fulfilled the brain can release neuromodulators that can alter the learning performed by STDP, resulting in R-STDP (see section 2.2.3). In general, reinforcement learning plays an important role in the human brain by allowing it to perform both reinforcement and supervised learning, in contrast to the purely unsupervised learning regular STDP.

Reinforcement learning can prove useful for training SNNs for two reasons. Firstly, the framework does not enforce differentiability and thus applies to SNNs. Secondly, reinforcement learning is concerned with performing credit-assignment over long time periods and delays, which can act as a reference and provide inspiration for dealing with the time-delays in SNNs.

The current chapter will focus on theoretical RL that can be applied specifically to SNNs. Firstly, RL basics will be treated in section 3.1. Secondly, the (temporal) credit assignment that is central to RL problems will be covered in section 3.2. Lastly, the possible use cases and parallels for RL and SNN design are considered in section 3.3. For an excellent reference work on RL the reader is referred to Sutton and Barto [99], this work will be used as a reference for most of the current chapter. For more information on RL in neuroscience and biological systems the reader is referred to Bartlett and Baxter [7], Pawlak et al. [80], Reynolds and Wickens [83], Schultz [88, 89], Schultz et al. [90].

3.1. Basics

This section is largely an adaption of the work of Sutton and Barto [99]. The two most important characteristics (and also problems) of RL are trial-and-error search and delayed reward. As such, a rigorous mathematical/computational model is needed for both of those, which is what this section will deal with. A more in-depth analysis of the delayed reward is provided in section 3.2.

Central to any reinforcement learning application is the ability to (partly) perceive the state of the environment, interact with and influence the state through actions, and lastly, to have a goal or goals relating to the state space. All of these characteristics are captured in a Markov Decision Process (MDP) [42], and any algorithm that is suited to solve it can be considered a reinforcement learning algorithm. The learner and decision-maker is called the *agent*. This is limited to anything that can be arbitrarily controlled by the agent. The thing it is interacting with is called the *environment*. Think of an action like the electrical current sent to an engine of a drone, this is part of the agent. The rotor blades can be considered part of the environment since it will respond to the current but also the air surrounding it.

To completely describe and solve a RL problem three things are needed: a MDP describing the interaction between the agent and the environment which includes the reward function, a value function indicating the expected reward the agent can receive from each state, and a policy for determining which action an agent will take in each state it is in. The final result of reinforcement learning is a policy, as

this can be applied to new situations that are comparable to one the agent has been trained in, but the MDP and value function are prerequisites for completing the training.

Markov Decision Process

The simple goal of an agent is to maximize the total reward it receives. Thus maximizing immediate reward is not the task, but the cumulative long-term reward. To obtain a reward the agent is continuously interacting back and forth with its environment in an iterative process which is explained in fig. 3.1. Based on its current state S_t, the agent performs an action A_t and as a result of that will move to state S_{t+1}. At this point, the environment provides the agent with a reward R_{t+1}, which is a numerical value.

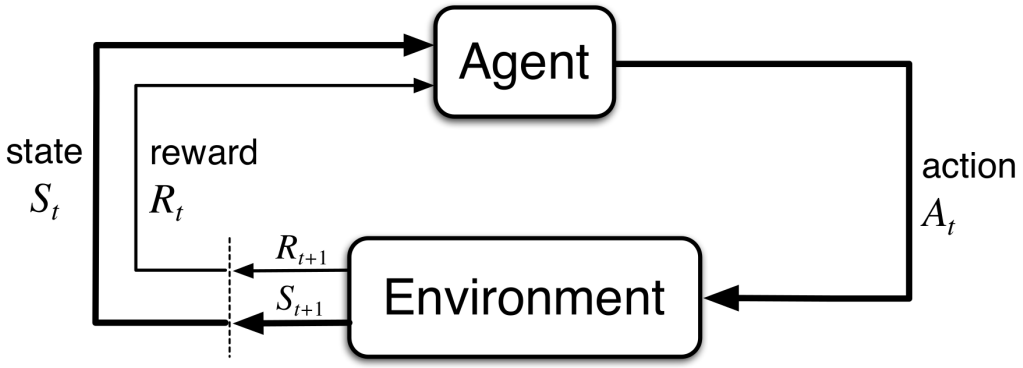


Figure 3.1: The agent-environment interaction in a Markov decision process. Adapted from Sutton and Barto [99]

As mentioned before, this process can be captured by a MDP. In the case of a *finite* MDP the sets of states, actions, and rewards all have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probabilities dependent only on the current state and action. This is best viewed not as a restriction on the decision process but on the state, it must contain all needed information of previous states. Under this assumption, the state satisfies the Markov property [61], which will be assumed true for the rest of this chapter. Equation (3.1) expresses this relation between the current state and action with the future state and reward. Equation (3.1) completely defines the *dynamics* of the system and thus summing over all its possible parameter values results in eq. (3.2). Equation (3.1) is the central equation for a MDP process and can be used to compute any other information about the environment.

$$p(s', r|s, a) \doteq Pr\left[S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\right] \tag{3.1}$$

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1, \; for \; all \; s \in S, a \in A(s) \tag{3.2}$$

Reward Function

In order to maximize cumulative reward a mathematical formulation of this objective is needed. This objective is called the *expected return* and is a function of the reward sequence $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$ generated by the interaction of the agent with its environment. Its recursive formulation is found in eq. (3.3). The equation expresses its current expected return G_t as the sum of its reward for the next time time step R_{t+1}, plus the expected return of the next time step G_{t+1} multiplied by a discount factor γ. The discount factor lies within the range $[0, 1]$ and makes sure the expected reward remains bounded in the situation of an infinite number of time steps $T = \infty$, under the assumption the R_t is bounded too. The expected return remains bounded according to the limit in eq. (3.4) for a constant reward value of +1. A more general expression for eq. (3.3) is found in eq. (3.5) as it expresses the expected return for every desired time step with respect to the initialization time t_0 and is agnostic to the duration of the episode.

$$G_t \doteq R_{t+1} + \gamma G_{t+1} \tag{3.3}$$

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{3.4}$$

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{3.5}$$

Equation (3.3) and eq. (3.5) express the relationship between current and future rewards. They express a preference for instant reward in favor of future rewards by discounting them with γ. A gamma value close to 0 states that future rewards are non-important, whereas a value close to 1 puts an equal emphasis on instant and future rewards. A common value for γ is 0.99.

Policies and Value Functions

To complete the RL model both a *value function* and *policy* is required. A *value function* is an estimation of how good (or desirable) it is for an agent to be in a given state (or state-action pair). The value function expresses how good the current state is in terms of the expected return that can be achieved. The way an agent selects which action to take in each respective state is what is called the policy. Formally, a policy is a mapping from a state to the probabilities of selecting each possible action. Reinforcement learning methods specify how the agent's policy changes due to its experience.

The agent has to learn to estimate the value function over time. The mathematical formulation is found in eq. (3.6). It expresses the expected return of a state s under the current policy π. Estimating the value function is an important task and often the most complicated part of solving a RL problem.

$$v_\pi(s) \doteq \mathbb{E}[G_t | S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \; for\ all\ s \in S \tag{3.6}$$

The expectation in eq. (3.6) is a function of the random variables of the reward R and state S and indirectly of the action variable A. Its expression can be expanded to more clearly express the relationship between the policy, the dynamics of the environment, and the received rewards. This is done in eq. (3.7) and it is called the *Bellman equation*. It expresses a relationship between the value of a state and the values of its successor states as a sum of the rewards for each state weighted by the total probability of reaching that state and reward. It takes into account the probability of moving into a specific state under a specific action as $p(s', r|s, a)$ and the probability of selecting each action under the current policy as $\pi(a|s)$. The Bellman equation plays a central role in RL since its properties need to be accurately estimated.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_{pi}(s')], \; for\ all\ s \in S \tag{3.7}$$

Solving Reinforcement Learning Problems

Solving a reinforcement learning task roughly comes down to finding a policy that obtains a lot of reward over the time it is active. A policy π is considered better than policy π' if its expected return is equal or greater for every state than the other policy, e.g. the following holds $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v'_\pi(s)$ for all $s \in S$. The best policy of all is called the *optimal policy* denoted as π_*, although there might be more than one optimal policy. All optimal policies share the same optimal value function v_* as in eq. (3.8). Once the optimal value function is found it is relatively easy to find an optimal policy, meaning the problem generally revolves around optimally solving the Bellman's equation for the value function in eq. (3.7).

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{3.8}$$

The last thing that is needed to complete a RL problem is a function that can represent the dynamics of the value function and policy, and that can also be learned from experience (trial and error with the environment). There are two main approaches to solve this problem, of which the first is tabular methods that work on discrete problems or continuous problems that have been discretized. In order

to optimize these functions Monte Carlo methods [12] are used. These methods rely on visiting each state as often as possible and averaging the rewards it receives for each state, resulting in an estimate of the value function. These methods greatly suffer from the curse of dimensionality and thus scale badly as the number of states increases. The other methods consists of parameterized functions like Gaussian mixture models [12] or neural networks (see chapter 2). These methods are optimized using algorithms like maximum-likelihood estimation, backpropagation, or STDP, which indirectly minimize the error of the model whereas Monte Carlo methods do so directly. In this work RL will be used in conjunction with parameterized models, specifically, ANNs or SNNs, and thus from now on will be the only methods considered.

3.2. Credit Assignment Problem

An important problem within RL that is also especially relevant for SNNs is the *credit assignment problem*. This can be stated as follows: ”In case of a successful period how does one assign credit (or discredit in case of failure) for the success among the multitude of decisions [64]?”. Especially in a SNN where many synapses take part in obtaining the reward it becomes a hard problem to assign credit to the correct synapses. To add to this the signals sent out by synapses and neurons have varying time delays before they actually impact the final result and might also influence multiple reward signals due to the persistent voltage in neurons. In short, a large number of free parameters in the synaptic weights and the varying time delay before signals reach the final layer makes the credit assignment problem in SNNs a rather difficult one.

Many approaches to try and solve the credit assignment problem make use of the concept of *eligibility traces*. An eligibility trace can be seen as a temporary record of the occurrence of an event, like spiking activity of a specific neuron, or the reward received in the past couple of time steps. This way the traces work as a memory that marks a specific event, weight or object as eligible for assigning credit or blame for the received reward or error. Thus eligibility traces are a basic mechanism for temporal credit assignment. Many neuron models in SNNs make use of this concept by expressing the recent activity of a neuron or synapse as a single scalar value trace (see section 2.2.2).

As a mathematical example, the application of an eligibility trace to a reward signal will be provided, but the concept can just as easily be applied to any temporal signal. The general formulation of an expected reward G eligibility trace can be found in eq. (3.9). The important factor in eq. (3.9) is the term λ, which should fall within the range $[0, 1]$ to make sure the trace remains bounded. The λ expresses with how much the trace from the previous timestep will decay. By doing so it indirectly expresses the temporal window that is considered as relevant within the trace. In case of $\lambda = 0$ only the current timestep t is considered. In the case of $\lambda = 1$ the trace is equivalent to the Monte Carlo result where the trace is tracked from the initiation till the end of the period. The term $(1 - \lambda)$ is used to ensure that the weights of the λ terms sum to 1 in the case of infinite duration.

$$G_t^{\lambda} \doteq (1 - \lambda) \sum_{n=0}^{t} \lambda^{n-1} G_{0:n} \tag{3.9}$$

Some of the advantages of eligibility traces include that they are incremental. They can be applied in an online learning algorithm like R-STDP. The method is bootstrapped, meaning that it uses its previous estimate to improve the accuracy of its estimate in the following time step.

Actor-Critic

As mentioned in section 3.1, solving a RL problem revolves around maximizing reward and one of the hardest parts in this is accurately estimating the value function (eq. (3.6)). Once the value function is approximated well enough it is relatively easy to find a decent working policy. Many methods try and combine estimating both of these functions into a single model, or simply perform greedy action selection based on the value function. A set of more advanced methods is called *actor-critic* methods [48, 100]. The main concept behind this is that the critic learns the value function and ”teaches” the actor what actions to perform in each state.

Several of the advantages of an actor-critic method are as follows. The algorithm most of the time outperforms just an actor or just a critic method in accuracy, time till convergence and success rate of converging. It reduces variance in the performed weight updates. It allows for learning a parametric

representation of both the value function and the policy. After training, just the policy can be used for interacting with an environment. An actor-critic algorithm also comes closer to a form of supervised or self-supervised learning (see section 2.1.1) since the critic is used to predict what will be the expected reward for the next state. It is actively trying to minimize a loss with a clear lower bound of an error of zero. This differentiates itself from more regular RL algorithms that just try to maximize their total reward that has no clear upper bound.

Besides these practical advantages, there is also ample evidence in neuroscience literature that suggests that parts of the human brain make use of an actor-critic learning model [22, 73, 88]. This should not be a reason in itself to make use of actor-critic algorithms, but it is an indication that there might be value in using these methods to train SNNs.

3.3. SNN and RL

From a biological perspective, there is an obvious link between RL and SNNs as it is widely used in the human brain. From an engineering perspective, there is also an important link between the two subjects that many forms of R-STDP try to capture. This is the fact that by using RL one can perform a form of supervised learning in SNNs without using differentiable (and thus back-propagation) operations. Reinforcement learning would provide the weights of synaptic connections with information about their performance concerning a common goal like classifying images. This is a form of the backward channel described in section 2.2.5 [6].

An important factor in this approach is the fact that each neuron can be considered a single RL agent, thus making this into a Multi-Agent Reinforcement Learning (MARL) [15], specifically, a *cooperative* one where all agents try to maximize the same reward function. It is this fact together with the time delay between an input signal and the output signals of each neuron that makes RL in SNNs a *structural credit assignment problem*. E.g., credit has to be assigned to both each specific neuron and also to its action at a specific time. Luckily, even though many separate agents cannot observe the complete state of the environment (all other neurons and the reward function), as long as all agents work towards maximizing the same reward signal and each agent is equipped with an algorithm that allows it to learn to increase the reward signal, the collective will also learn to increase the total reward [99].

An important note when using RL in SNN learning is that the design of the reward function and definition of the expected reward (see eq. (3.5)) is paramount and differs greatly between a continuous control task and a classification task. During a classification task, there is a clear number of steps to be taken, which is the number of discrete time steps in the input sample. Because of this, the reward can be guaranteed to be bounded without a discount γ. This would also imply that every instance of classification is valued just as important, which is a valid assumption for most classification tasks. On the other hand, during a control task, it is desirable to implement a discount factor in the expected reward formulation. Otherwise, the reward could become infinite, but this also takes into account that immediate reward is more desirable than a reward that has a large delay (see section 3.1).

Pure reinforcement learning that is using a single reward signal is highly unlikely to solve a complex problem like optimizing even a medium-size SNN. This is due to the vast number of free parameters and only a single scalar value reward telling the network if it is performing well or not. Even so, RL might be able to play a role in solving the SNN problem by combining it with other learning algorithms. The first example is simply R-STDP that combines RL with STDP. One could also think of using RL to optimize a critic function or perform online hyperparameter tuning. One thing is clear though, there is likely a role for RL within optimizing SNNs, as it is naturally fit to solve credit assignment problems and humans are the living proof of its capabilities.

4

Literature Synthesis

The first part of this document provided a literature study on the topic of supervised learning in spiking neural networks. Attention was paid to three specific subjects: deep learning and neuromorphic computing (see chapter 2), and reinforcement learning with a focus on spiking neural networks (see chapter 3). These subjects were chosen as they provided valuable information in deciding whether ANNs or SNNs should be chosen for modeling, but also in the design of a learning rule for SNNs. A direct comparison between the methods was hard since they provide different strengths and weaknesses. This is due to their different computational paradigms where ANNs are highly sequential and based on large, synchronous matrix operations, whereas SNNs are highly parallel and asynchronous in nature. Each of the following sections provides a summary of the literature and concludes with a synthesis of the relevance of the related findings towards the goal of this thesis. Combined with the experimental results presented in chapter 5, chapter 6, and chapter 7 a decision was made between ANNs and SNNs to focus on SNNs. After the overview sections, this chapter will end with a collection of relevant conclusions, points of attention, and ideas for developing a supervised learning rule for a SNN.

4.1. Neuromorphic Computing

Neuromorphic computing is seen as the field of research that focuses on implementing artificial neural networks that are biologically inspired. These networks are called spiking neural networks and posses advantages like asynchronous operations, sparse signals, and low power consumption [59]. Whereas there is an increasing amount of knowledge on the functioning of the human brain, the actual reproduction of a large scale artificial brain-like architecture is still to be achieved. The key factor in SNNs is that information is communicated in the form of discrete spikes. Using SNNs for processing event-based data is a natural fit, yet it currently lacks the learning capabilities of the more common ANNs. This leads to an important goal within the SNN community to find learning algorithms comparable in power to the back-propagation algorithm.

Artificial neural networks form the state-of-the-art in most machine learning and deep learning applications. They perform powerful sequential pattern matching [53] for many tasks like classification and data generation while using varying architectures like convolutional and recurrent networks. Yet, for neuromorphic computing, they seem sub-optimal because of the mismatch between their sequential and matrix operation based processing paradigm when applied to sparse and asynchronous data streams. Some works exist that process event-based data using a regular ANN like Gehrig et al. [24], Sekikawa et al. [92], Zhu et al. [111], or through converting an ANN to a SNN like [18]. Information is lost during the conversion process of either the event-based to frame-based format or the network from ANN to SNN. There is also the inconvenience of not being able to train these networks on actual neuromorphic chips. Although ANNs are not the most suited for neuromorphic computing, they do offer insight in devising learning rules as back-propagation is considered the upper limit of learning efficiency [6, 57]. Together with a rich amount of architectures that currently outperform SNNs in most tasks, and a large and active research community, ANNs can provide a wealth of information for the SNN community.

As mentioned before, SNNs are inherently suited for processing event-based data. There are several (theoretical) advantages to using SNNs like greater computational power per neuron and more efficient energy usage. Currently, these have not yet been achieved due to a lack of mature hardware acceleration and software frameworks, and stable supervised learning rules. In the search for better supervised learning rules, two main approaches exist, correlation and gradient-based (network conversion not considered). Correlation-based learning methods are biologically inspired and (mostly) biologically plausible. Its most basic form is local and unsupervised STDP [96]. Several extensions exist that try to add a supervision signal through reinforcement learning techniques, called R-STDP [20, 68]. Some interesting and promising first results were generated using these algorithms, but none of them seem to have a thorough mathematical justification. Other works try to configure their network in such a way that the desired patterns emerge while using STDP [78, 81]. Although they succeed at their respective tasks, the algorithms are not very flexible and cannot learn arbitrary input-output mappings, for that a specific error or reward signal moving back through the network is needed [6].

The second approach is to apply gradient-based methods to SNNs. This requires making approximations and/or assumptions on how the SNN behave because of the non-differentiable spiking signals. The currently most effective algorithm, called SLAYER [94], makes a single assumption on the spike generation mechanisms of a neuron. The limited amount of assumptions makes it that SLAYER uses a lot of the power of the back-propagation algorithm. It integrates into the PyTorch framework and performs well on at least smaller networks. It does require an even larger amount of GPU memory compared to regular ANNs due to the added time dimensions to the input tensors. It also has the added downside of only being able to train a network offline.

A trade-off between correlation and gradient-based methods for training SNNs shows that correlation-based methods are outperformed by gradient-based methods on both accuracy and applicability to different network architectures. They do have a natural fit to SNNs resulting in a high theoretical ceiling in both accuracy and efficiency of resources. An overview of the learning rules mentioned in this work, together with their most important pro and con, and an example of an implementation of the method is found in table 4.1.

4.2. Reinforcement Learning

Only a limited part of reinforcement learning was covered in this work. The basics of estimating a value function and policy through long term reward maximization were treated [99]. The focus was on the applicability of RL in training SNNs, both for reinforcement and supervised learning tasks. There is a large correspondence between the two subjects (both in the theoretical/mathematical and biological sense). Both heavily make use of traces as a memory for the recent activity of a (stochastic) variable and reward-based modulation of the training algorithm to solve a (temporal) credit assignment problem. Practically this means that eligibility traces can be used to express pending weight updates for synapses, recent activity in neurons, and recently received reward signals. This is a large number of applications compared to regular deep RL where traces are only really applied to the reward signal itself. The best example of RL in SNNs is R-STDP, the RL based learning rule that is also used in human brains for learning arbitrary and precise spiking patterns. An interesting direction of research is to take more inspiration from the extensive body of computational neuroscience literature on the subject of R-STDP since there are multiple works on showing and proving the stability of R-STDP. It seems that there has been relatively little work translating those results to the engineering of SNNs.

4.3. Designing a Learning Rule

Based on previously presented literature and the research question for this thesis (see chapter 1), three parts for designing a supervised learning rule in SNNs have been identified. The three parts consist of neuronal dynamics, plasticity updates, and credit assignment. Each of the parts their scope, reference work, and some ideas on how to address the corresponding difficulties in a supervised learning rule are treated next.

- **Neuronal dynamics**: This contains the dynamics of both neuron cells and their synapses like cell voltage, refractory period or spike transmission delays. There is quite an extensive list of mathematical neuron models that can be used like the IF, LIF, SRM and more [25, 27, 45]. Neuronal dynamics are quite well known from both an engineering and neuroscience perspective allowing

for a large variety and inspiration.

- **Plasticity updates**: This includes the algorithm for updating synaptic weights. Both from an engineering and neuroscience perspective there is decent amount of work on this topic [20, 78, 96, 99]. An important focus point should be assuring that synaptic weights remain bounded and ideally also follow a uni-modal distribution [29, 105]. Updating parameters other than synaptic weights are also an interesting research avenue for their influence on the stability of a learning rule, as it might add instability or bias to an otherwise stable STDP rule [22].
- **Credit assignment**: This is the problem of correctly assigning credit (or discredit) to operations causing a reward (or error) signal. Compared to the other two points, this is the hardest problem to solve since there is no clear theory on how the human brain does this. Besides the inability of directly differentiating SNN operations, this problem also becomes harder due to the time delay between a signal entering a network and it influencing the output layer. Solving this problem can use a lot of creativity by for example using RL methods, evolutionary methods, reciprocal connections (connections feeding back to previous layers) or even random feedback weights [58]. In the end, this problem comes down to feeding reward/error information back to earlier layers in the network.

Table 4.1: Learning rules for optimizing spiking neural networks and their strengths and weaknesses as derived from literature.

	Sub-classification	Positive	Negative	Example
STDP	Additive	Fast weight adaptations	Bi-modal weight distributions	Song et al. [97]
	Multiplicative	Uni-modal weight distributions	Slow weight updates	Paredes-Vallés et al. [78]
	Local in time	Aggresive weight adaptations	No delayed weight updates	Song et al. [97]
	Eligibility Traces	Delayed weight updates/rewards	Trace decay needs tuning	MSTDPET [20]
	Reward modulation	Supervised form of STDP	Learning based on trial-and-error	Mozafari et al. [68]
Gradient methods	Cell voltage based	Small approximations	Converges to rate based weight updates	SpikeGrad [104]
	Event based	Can learn specific spike timings	Assumes Kronecker delta derivative	SLAYER [94]
Conversion		Methods and tools available for ANNs	Limited to ANN designs	Diehl et al. [18]

III

Preliminary Experiments

5

Methodology

This preliminary research acts as a precursor to the main research question: **"Can a supervised learning algorithm be designed that can optimize spiking neural networks in an online manner?"**. Because this problem is rather complicated and too large for a preliminary analysis the scope is decreased to: **"How do supervised learning rules for event-based ANNs and SNNs compare against each other in both training time and testing accuracy?"** This question is rather qualitative and no decisive answer can likely be given at the end of these experiments.

In order to compare ANNs and SNNs roughly two types of experiments are performed. The first set tries to indicate the trade-off between network accuracy and computational overhead while using ANNs for processing event-based data. This is done by looking into discretization methods and sequential processing methods. The second set of experiments tries to provide insight into the performance of supervised learning rules applied to SNNs. Both a gradient-based and a correlation-based method will be tested.

Section 5.1 will first provide an overview of the event-based datasets that have been used during the experiments. This is followed by a description of the four experiments in section 5.2, section 5.3, section 5.4, and section 5.5 respectively.

5.1. Datasets

During the experiments three datasets have been used, the N-MNIST dataset, the N-Caltech101 dataset, and a simple spiking intensity distribution dataset. Each of the datasets is an event-based dataset and does not contain any further data except for labels. A description of the data in terms of dimensions, the number of samples and the purpose of the datasets are provided next.

N-MNIST

The N-MNIST dataset was made and presented by Orchard et al. [76]. It is based on the MNIST dataset [51], which is a staple dataset within the computer vision community for fast prototyping and testing of vision algorithms. The original dataset consists of approximately 60.000 training and 10.000 test images of handwritten numbers of 0 to 9. To convert the MNIST dataset to an event-based representation an ATIS event-based camera [82] camera was placed in front of an LCD screen which displayed the MNIST images. To make the camera register events it was mounted on a robotic arm that performed three small saccades to move the camera in a predefined pattern as shown in fig. 5.1. Each saccade lasted for about 100 ms. Using this automated process, all MNIST images were converted into event-based representations of $34x34$ pixels, with a duration of 300 ms, and polarity dimension of 2.

N-Caltech101

The N-Caltech101 dataset was presented in the same work as the N-MNIST dataset [76]. This dataset is based on the more complex Caltech101 dataset [55]. The original dataset contains images of 101 different classes, all of about $300x200$ pixels. Each class consists of $40 - 800$ images with an average of about 50 per class. Conversion of the original images to their event-based representation was

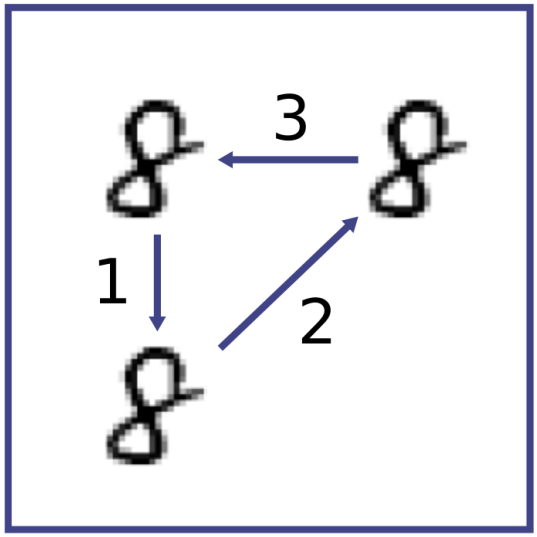


Figure 5.1: The saccades performed with the ATIS sensor for converting the MNIST dataset to an event-based format. Adapted from [70]

done using the same method as for the N-MNIST dataset. The N-Caltech101 images are also roughly $300x200$ pixels in size, have a duration of 300 ms, and polarity dimension of 2.

Spike Rate Distributions

The spiking intensity dataset is a simple dataset that defines the *spike rate per second* for each neuron in the input layer of a network. Each of the neurons then generates spikes according to a *homogeneous Poisson process* [38]. This means a neuron generates spikes with a constant probability of spiking at each time step. On average, the total amount of spikes generated by a neuron is constant, yet the time between consecutive spikes varies.

All variations of the dataset share the fact that they are one-dimensional and small to keep the datasets simple. Datasets differ in their spiking rates and how the spiking rates are distributed among different neurons. The firing rate for the input neurons varies relatively smoothly from neuron to neuron, and each sample has its highest spike rate placed at a different input neuron. An example is found in fig. 5.2. This dataset can easily be adapted to a various number of classes, and also vary in the number of neurons that are present in a single sample. Each time a sample is drawn from the dataset it is generated based on the Poisson process. Because of this, each sample varies slightly from the previous ones.

5.2. Event Discretization Kernel

One of the more intuitive ways of using event-based images in regular ANNs is by converting the event voxel (height x width x time) to a frame-based image. This requires the discretization of the time axis into a small number of channels, like discretizing a $300ms$ event-based sample into a 4 channel frame-based image. Most methods do this by simply assigning an event to the closest time discretization point, or through a form of simple interpolation [60, 66, 111]. Figure 5.3 shows an example of linear interpolation where the event's contribution to the discretization points is based on how close the event is to each point in time. All values are normalized for ease of interpretation. In Gehrig et al. [23], a method for learning to discretize an event-stream into a frame-based representation is presented. The novelty lies in the fact that the discretization kernel is learned together with the downstream ANN, making the discretization kernel task and network specific.

In their work, the author presents classification performance for the Neuromorphic Cars (NCars) dataset of 92.5% accuracy, and for the N-Caltech101 of 81.7% accuracy. The network used during these experiments consisted of the discretization kernel followed by a 34 layer ResNet [36]. The 152 layer ResNet architecture holds the crown as the best performing network on the ImageNet competition. The 34 layer ResNet showed an accuracy of 78.5% on the ImageNet dataset, which is very comparable to

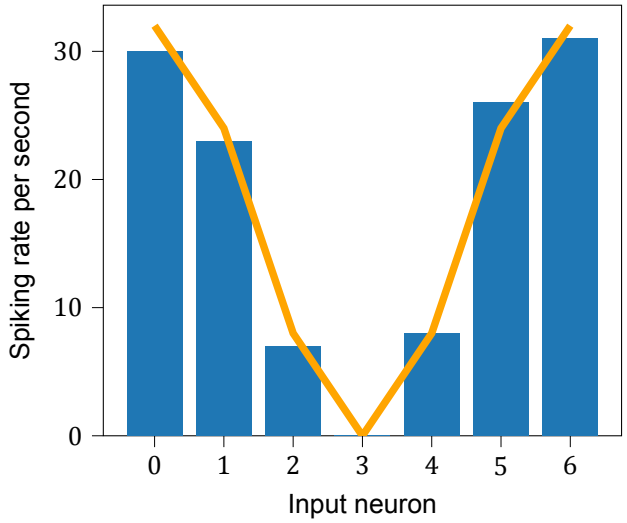


Figure 5.2: An arbitrary spiking rate distribution for a set of six input neurons.

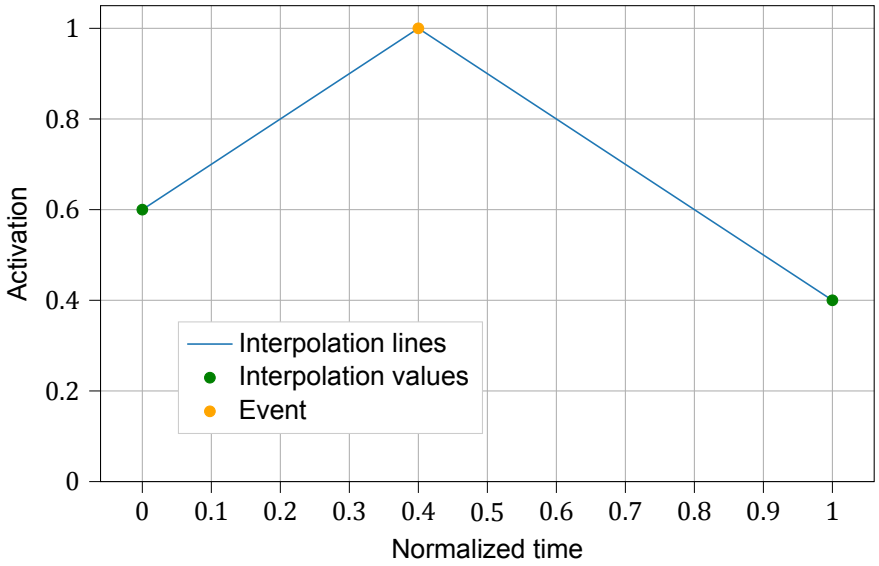


Figure 5.3: Linear interpolation of an event its contribution to two adjacent discretization points, which is based on how close the event is to the discretization point in time.

the performance of the event-based network which runs on a dataset of 101 classes. Next, the authors mention replacing the discretization kernel with a hash-map after learning to increase the inference time of the network. Whereas inference time decreases, the network becomes rather rigid since the hash-map is a fixed representation. Also, a decrease in performance might occur.

The following experiment has got two focus points. The first is to provide insight into the increase in inference time by adding the learnable kernel to a network (not the hash-map). The second is to gain insight into the possible loss of information/performance of this method compared to performing inference using a regular frame-based data representation. This will be done by comparing a 34 layer ResNet trained on the frame-based Caltech101 datasets and a 34 layer ResNet prepended with the learnable event discretization trained on the event-based N-Caltech101 dataset.

Experiment Setup

The experiment consists of two parts. The first is setting up the reference test for the regular frame-based dataset and network. For this experiment the Caltech101 [55] dataset is used together with a pre-trained 34 layer ResNet. To fit the pre-trained network to the new Caltech101 dataset the final fully

connected layer was replaced with a new layer consisting of the 101 necessary nodes for classification. During training, only the final layer of the network was optimized to speed up training and make optimal use of the extensive training performed by the creators of the network. This concept where a large part of a pre-trained network is reused while only retraining a small set of layers is called transfer learning [10].

The second part consists of testing the classification and duration performance of the learnable discretization kernel followed by a pretrained 34 layers ResNet, repeating the experiment from [23]. This time the dataset is the N-Caltech101 and for the ResNet both the first convolutional layer as well as the final fully connected layer are replaced and trained from scratch. The first convolutional layer has to be retrained since the number of channels for the input image is increased from 3 to 18 to not lose too much information from discretizing the time dimension of the event-based images. The discretization kernel follows the design presented in the original paper and thus consists of two fully connected layers of 30 nodes each. After processing each event with the kernel the result is folded into an image of the following dimensions: "channel: 18, height: 224 and width: 224". In this representation, the polarity dimension of the spike tensor is unfolded along the new channel dimension. This means that both the sets of positive and negative polarity spikes are discretized at 9 points along the time axis.

The following settings and/or algorithms are shared by both the experiments. Cross entropy is used as the loss function since this is a multi-class classification problem [33]. For optimization, the Adam algorithm [47] is used. During the frame-based experiment standard hyperparameters for the Adam algorithm are used, which can be found in the first column of table 5.1. For the event-based experiment the hyperparameters as presented in Gehrig et al. [23] are used and can be found in the last column of table 5.1.

Table 5.1: Advised standard hyper parameters for the Adam optimization algorithm

	Standard parameters	Discretization task
Learning rate α	1^{-3}	1^{-5}
Running average coefficient β_1	0.9	0.9
Running average squared coefficient β_2	0.99	0.99
Denominator addition (prevent division by zero) ϵ	1^{-8}	1^{-8}

5.3. Phased LSTM

The PLSTM [70] was mentioned in section 2.1.3 and is an extension of the LSTM by adding a time gate. This gate allows for sparse updates of the weights of a cell which results in improved performance. The PLSTM was introduced in Neil et al. [70] where one of the experiments consisted of classifying (event-based) N-MNIST images. As mentioned in section 2.1.3, the biggest advantage of using variations of RNNs is that it allows an ANN to directly process event-based data without first discretizing the time dimension like in section 5.2. A clear downside is that this forces the user to process the data as an event-stream, thus losing the spatial representation of an image. Even so, the paper's results show an improved sample efficiency compared to networks that use regular LSTMs, with an accuracy of 97.3%. The goal for the PLSTM experiment is the same as for the event discretization kernel in section 5.2, to gain insight in the classification accuracy and inference time when applying the PLSTM to event-based and frame-based data.

Experiment Setup

This experiment consists of two parts. Firstly, the PLSTM is used to perform classification on the event-based N-MNIST dataset, complemented with a control experiment. Secondly, the PLSTM is applied to the frame-based MNIST dataset, again with an added control experiment. First, the architecture as presented in Neil et al. [70] is covered. This will act as a reference for both experiments. Differences in architecture for each experiment will be covered after explaining the original network.

The network consists of three layers, an embedding layer, a layer of 110 PLSTM cells, and a fully connected layer of 10 nodes as the final layer. In the first layer, each event is fed into an embedding

operation [63] which outputs a 40 element long embedding vector. The embedding is a learnable hash-map that maps a high dimensional, discrete input to a set of unique lower-dimensional vectors, one for each possible input. This operation is performed to decrease the dimensionality of the input data. Embeddings were originally developed for high dimensional language datasets. Besides now being of a lower dimension, the words are also represented as scalar values and thus can be processed by ANNs. An added benefit is that semantically similar words end up close to each other in the vector space, like *king* and *queen*. This property might make it that the sequential network retains some of the spatial order of the images. To apply an embedding to a spiking image containing events, this image first has to be converted to a vector. Each element stands for a unique combination of indices along the pixel height, pixel width, time, and polarity dimensions of the spiking image. This assumes the event-based image is already discretized at a very high temporal resolution by the camera, e.g. a temporal spacing of $10\mu s$. The indexing is equivalent to linear indexing of a tensor where each point in the *height x width x time x polarity* voxel of a spiking image is converted to an embedding integer id according to eq. (5.1). In this equation i indicates a single pixel (or cell) in a four-dimensional tensor. A graphical explanation is shown in fig. 5.4. The three-dimensional tensor on the left shows linear indexing, whereas the tensor on the right shows regular dimension based indexing. The vector resulting from the embedding operation is a latent space representation of each event, which then gets passed to a 110 cell PLSTM. Afterward, the output of the PLSTM cell is processed by a fully connected layer of 10 elements, the output of which is considered the classification output of the network.

$$embedding\ id = (i_{polarity} \cdot i_{time} \cdot i_{height} \cdot i_{width})(i_{time} \cdot i_{height} \cdot i_{width}) + (i_{height} \cdot i_{width}) + i_{width} \quad (5.1)$$

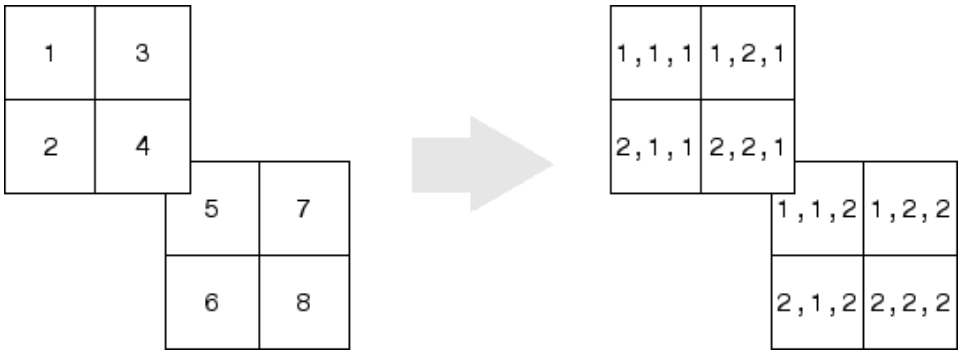


Figure 5.4: Example of transforming a linear indexing into regular, dimension based indexing for a three dimensional matrix. Each cell in the matrix is assigned a unique integer index. Adapted from $https$: $//nl.mathworks.com/help/matlab/ref/ind2sub.html$ - accessed 10-8-2019

The first experiment tries to reconstruct the original PLSTM experiment. For its control experiment, the only difference was that PLSTM cells were replaced with regular LSTM cells. This indicates the possible increase in performance due to using a PLSTM. In the second experiment, the PLSTM is applied to the frame-based MNIST dataset, the embedding layer is removed from the architecture and just the PLSTM and the fully connected layers remain. Since the MNIST images are grayscale (meaning only one channel) and 28x28 pixels each, there is no need to represent them in a lower-dimensional space. For the control experiment, the same reasoning applies as with the event-based data, just the PLSTM cells are replaced with LSTM cells.

Implementations of the PLSTM in multiple frameworks exist. For this experiment, the Keras implementation was used[1]. All the networks were trained using the same hyperparameters, which follow next. Training was performed on a subset of the N-MNIST dataset consisting of 1000 training images and 100 test images. This was used instead of the 60.000 training images to keep the training duration within two to four hours. Training lasted for 100 epochs, with a batch size of 100 images to maximize GPU usage. For the optimizer, the Adam algorithm [47] was chosen with standard hyperparameters since these were used in the original experiment. The parameters again are as shown in table 5.1.

[1] https://github.com/fferroni/PhasedLSTM-Keras - accessed 10-08-2019

5.4. Spike Layer Error Reassignment in Time

The SLAYER algorithm [94] is the state of the art when it comes to applying back-propagation to SNNs, which makes it relevant for performing supervised learning. In the paper (and on github as well[2]) three experiments are shown. Out of these the MLP and CNN used for classifying N-MNIST data will be discussed briefly. The accuracy of the two layer MLP during training and testing is presented in fig. 5.5 and shows an average testing accuracy of about 94% and peaks at around 97%. The accuracies for the four layer CNN can be found in fig. 5.6. It shows an average testing accuracy of about 60% and peaks at about 70%. This is remarkable since a CNN is inherently better suited for computer vision tasks. This difference in performance will be explored next.

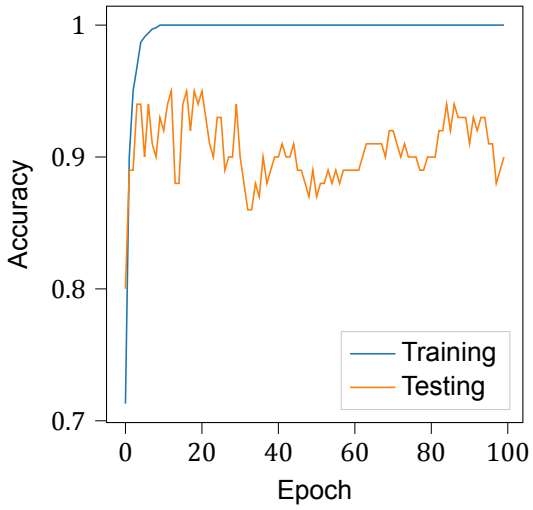


Figure 5.5: Training and testing accuracy of the MLP for N-MNIST classification. Adapted from Shrestha and Orchard [94]

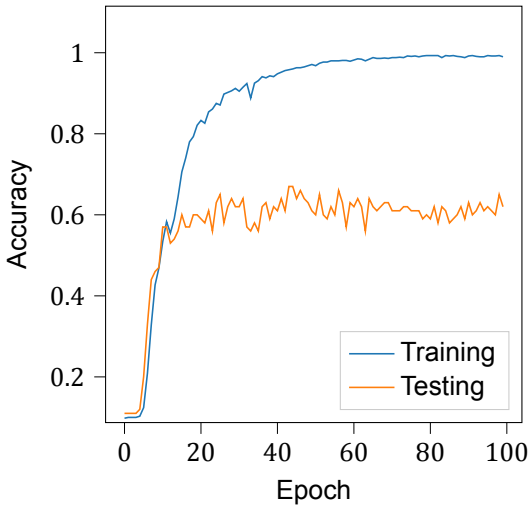


Figure 5.6: Training and testing accuracy of the CNN for N-MNIST classification. Adapted from Shrestha and Orchard [94]

Besides the difference that one network is a MLP and the other a CNN, another significant difference is in the number of layers. The MLP consists of just two densely connected layers, whereas the CNN consists of four-layers, out of which 3 are sparsely connected (the convolutional layers) and a single fully connected layer. This raises the question of how viable the SLAYER algorithm is for training deeper and/or sparsely connected networks. Back-propagation is known for having issues with vanishing and/or exploding gradients, which is especially true for deep networks and RNNs. This is because in those types of networks there is the possibility of multiplying many small (or large) gradients with each other, either causing them to vanish or blow up. This gradient issue occurs more often within ANNs that use of the sigmoid activation function (see section 2.1.1 and fig. 2.2) where the gradient of the sigmoid activation is practically zero in either of its tails. To solve this problem the ReLU (see fig. 2.4) was introduced [31]. Figure 5.7 shows the PDF which the SLAYER algorithm uses to approximate the gradient of the non-differentiable spikes of a SNN. It is clear that the gradient of this PDF also approaches zero as the cell voltage moves away from the firing threshold at the vertical line. These observations resulted in the following hypothesis:

The SLAYER algorithm suffers from vanishing gradients when applied to moderately sized and/or sparsely connected networks.

Experiment Setup

To test the hypothesis of section 5.4 a set of experiments with progressively increasing network depths has been performed. Just as in the paper [94], a set of MLPs and a set of CNNs is used. For the MLPs four networks are compared, the original two-layer network, and three networks with each consecutive network having an extra layer compared to the previous one. An overview of the fully connected

[2]https://github.com/bamsumit/slayerPytorch/tree/master/example - accessed 06-09-2019

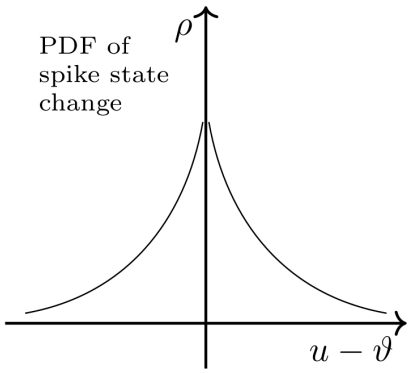


Figure 5.7: SLAYER probability distribution function. The gradient along the curve is used as an approximate gradient of the non-differentiable spikes within a spiking neural network. Adapted from Shrestha and Orchard [94]

networks can be found in table 5.2. For the CNNs, also four different networks are compared. In this situation, only the number of convolutional layers varies while the final fully connected layer stays the same. The four networks include a three-layer network, the original four-layer network, and a five and six-layer network. As a control experiment, for each of the SNNs an ANN counterpart with the same architecture will be trained. An overview of the convolutional architectures is shown in table 5.3. During all these experiments the N-MNIST dataset will be used for the SNNs, and the MNIST dataset will be used for the ANNs. For optimization, the Adam optimizer with standard parameter (as in table 5.1) is used, and training will last for 100 epochs.

Table 5.2: Layerwise number of neurons for the fully connected SNNs trained with the SLAYER algorithm

	2 Layer MLP	3 Layer MLP	4 Layer MLP	5 Layer MLP
Layer 1 (neurons)	512	512	512	512
Layer 2 (neurons)	10	256	256	256
Layer 3 (neurons)	–	10	128	128
Layer 4 (neurons)	–	–	10	64
Layer 5 (neurons)	–	–	–	10

Table 5.3: Layerwise parameters for the convolutional SNNs trained with the SLAYER algorithm. Following convention for describing a convolutional layer: k(kernel), s(stride), p(padding, always even), c(channels).

Layer/Network	3 Layer CNN	4 Layer CNN	5 Layer CNN	6 Layer CNN
Convolutional 1	k5x5 s1 p1 c16	k5x5 s1 p1 c16	k5x5 s1 p1 c16	k5x5 s1 p1 c16
Convolutional 2	-	-	k3x3 s1 p1 c32	k3x3 s1 p1 c32
Pool 1	k2x2 s2	k2x2 s2	k2x2 s2	k2x2 s2
Convolutional 3	k3x3 s1 p1 c32	k3x3 s1 p1 c32	k3x3 s1 p1 c64	k3x3 s1 p1 c64
Convolutional 4	-	-	-	k3x3 s1 p1 c64
Pool 2	-	k2x2 s2	k2x2 s2	k2x2 s2
Convolutional 5	-	k3x3 s1 p1 c64	k3x3 s1 p1 c64	k3x3 s1 p1 c64
Fully connected 1	10	10	10	10

After each epoch of training the network's accuracy is measured on the testing set. To confirm or reject the hypothesis several basic statistics about the gradients of each layer are tracked at each weight update. Specifically, the mean of all the gradients in a single layer, together with its standard deviation

will be used to provide information on how gradients propagate through the networks. Besides the gradients the networks their performances during training will also be useful for a qualitative indication of the performance of the SLAYER algorithm in progressively deeper networks.

5.5. Reward Modulated Spike Timing Dependent Plasticity

As mentioned in section 2.2.3, R-STDP is the correlation-based learning rule that gets closest to training SNNs in a supervised manner. Since this thesis' focus is on supervised learning, the R-STDP from Florian [20] (called MSTDPET) will be used as a reference for the following experiments. In section 2.2.3, section 2.2.5, and section 4.3 two important points of critique for correlation-based learning rules were mentioned. Firstly, using additive STDP learning rules results in bi-modal weight distribution, whereas multiplicative learning rules tend to be slow at converging. Secondly, correlation-based learning rules currently do not have an effective way of dealing with the credit assignment problem, also referred to as lacking an effective backward channel for the error or reward function. Two experiments will be performed in which the performance of MSTDPET concerning these two points will be tested, as well as applying some changes to try and improve its performance.

Next, the basics of MSTDPET will be covered. As the name suggests, the rule combines STDP based weight updates with reward modulation to determine the sign of the weight updates. It also makes use of traces (see section 3.2) to act as a memory for the recent activity of neurons. It is an additive rule, meaning weight updates are performed without taking the current value of the weight into account, as in eq. (5.2). Here i is the index of the post-synaptic neurons, j is the index of the pre-synaptic neuron, δt is the duration of a single discrete time step, γ is the learning rate, r is the reward, and w_{ij} and z_{ij} are the weight and eligibility trace of a single synaptic connection. Equation (5.3) shows the dynamics of the eligibility trace z_{ij}, which are defined by a recursive formula where z_{ij} decays over time due to $0 < \beta < 1$. Excitation of the eligibility trace due to neuronal activity is captured in η_{ij} and scaled by the time scale constant τ_z, which is discussed next.

$$w_{ij}(t + \delta t) = w_{ij}(t) + \gamma r(t + \delta t) z_{ij}(t + \delta t) \tag{5.2}$$

$$z_{ij}(t + \delta t) = e^{(-\delta t / tau_z)} z_{ij}(t) + \eta_{ij}(t)/\tau_z \tag{5.3}$$

The dynamics of η_{ij} are described in eq. (5.4), which in its essence multiplies the pre-synaptic trace P_j^+ with the post-synaptic activity of the last time step f_i. Next, it subtracts the product of the post-synaptic trace P_i^- with the pre-synaptic activity f_j. Both f_i and f_j are either 0 or 1, depending on whether the neuron spiked or not. The synaptic traces P_i^- and P_j^+ are updated after every time step according to eq. (5.5) and eq. (5.6). Again, this is a recursive and decaying function, but this time the decay is modulated by the exponential function $e^{(\delta t/\tau_{+/-})}$, where the terms $\tau_{+/-}$ are time constants. The last part of the equations covers the excitation of the traces, which at each time step only happens if target neuron i/j has spiked. The increase of the trace is scaled by the scalar terms $A_{+/-}$. The traces z_{ij}, η_{ij}, P_j^+, and P_i^- are updated after every time step, whereas the actual weight updates are performed only when the network receives a reward signal.

$$\eta_{ij}(t) = P_j^+(t) f_i(t) - P_i^-(t) f_j(t) \tag{5.4}$$

$$P_j^+(t) = P_j^+(t - \delta t) e^{\delta t/\tau_+} + A_+ f_j(t) \tag{5.5}$$

$$P_i^-(t) = P_i^-(t - \delta t) e^{\delta t/\tau_-} + A_- f_i(t) \tag{5.6}$$

To test the limits of MSTDPET for classification tasks while also trying to alleviate the credit assignment problem two experiments will be conducted. During these experiments two learning rules will be used, regular MSTDPET, while the second rule is an extension of MSTDPET that also updates spiking thresholds. One of the issues with MSTDPET is that for the final layer of the network, each neuron can be provided with a separate reward. But, earlier layers all have to share the same, single scalar reward value. This situation suffers from the curse of dimensionality, a single scalar value for learning possibly thousands of weights. To (partly) try and alleviate this problem, MSTDPET will be extended to also learn the spiking thresholds for the pre-synaptic neurons of each connection, except for the thresholds

of the input layer. Inspiration for this mechanism was found in biological experiments [17, 110], as previously mentioned in section 2.2.3. The reasoning behind this is as follows. Regular weight updates only make changes in the scaling of spikes arriving at the post-synaptic set of neurons, but it cannot directly adjust the number of spikes arriving. Thus, the learning rule is only able to adjust the activity of the post-synaptic neurons, yet the pre-synaptic activity remain unchanged. As an example, let's assume that for the final layer L an increase of its weights is performed, but the spiking threshold and thus the pre-synaptic activity remains unchanged. When moving to the layer before the final layer, $L-1$, the original pre-synaptic activity is now the post-synaptic activity for the current layer $L-1$. Since this is unchanged, no information from the upper layer is reaching the current layer $L-1$. The weight update for layer $L-1$ then only depends on the change in activity in its pre-synaptic neurons. However, in the situation where the spiking threshold of the pre-synaptic neurons is updated based on the correlation between the pre- and post-synaptic activity, the pre-synaptic activity changes along with the weights. This then also changes the post-synaptic activity of layer $L-1$, and information from the weight updates of the final layer indirectly "flows back through the network".

The implementation of this learning rule is rather simple. In case the weights of a synaptic connection are increased, the spiking threshold of the pre-synaptic neurons is decreased. This leads to increased pre-synaptic activity, thus reinforcing the connection. The mechanism is reversed in case of decreasing weights. Increasing weights and pre-synaptic activity of a connection is a self-reinforcing concept, meaning that both will likely increase the tendency of the learning rule to generate bi-modal distributions. The actual spiking threshold updates are performed according to eq. (5.7).

$$\Theta(t+\delta t) = \Theta(t) - \gamma r(t+\delta t) z_{ij}(t+\delta t) \tag{5.7}$$

Experiment Setup

Both the regular MSTDPET and the version that also adapts thresholds are tested on the same set of classification experiments. The first experiment consists of the exclusive OR (XOR) problem. Here the network has to predict the output of the XOR logic gate based on two binary inputs. If only one of the two inputs is true, the output of the network should also be true $[0,1] \rightarrow 1; [1,0] \rightarrow 1$. If neither or both of the inputs are true, the output of the network should be false $[0,0] \rightarrow 0; [1,1] \rightarrow 0$. This is a common and small, yet not trivial, experiment [65]. It was also used as an experiment in the original MSTDPET publication [20]. Because it was used in the original paper, the XOR experiment is mainly intended as a sanity check to test whether the adaptation of learning spiking thresholds does not break the learning rule and if there are any clear differences directly showing up.

The network is a two-layer network with 16 input neurons, 16 hidden neurons, and a single output neuron. For the 16 input neurons, the first 8 are assigned to represent the first binary input, whereas the other 8 represent the second binary input. An input of 1 is represented by a Poisson spike train of 40Hz, and an input of 0 with a spike train of 4Hz. At the initialization of the network, halve of the input neurons are randomly selected for which all outgoing synapses are assigned to only negative weights, thus ranging in weight between $[-5,0]$. The other 8 neurons are limited to have only positive outgoing weights in the range $[0,5]$. This is done to support the learning rule in training the inhibitive weights that are needed to solve the problem. Weights in the final layer are all initialized as positive. Weight updates are performed each time the network generates an output spike. In the situation where the output is supposed to be true, the network is rewarded with a positive reward of 1. If the output is supposed to be false but the network does spike, it is rewarded with a negative reward of -1. After training, the network output is considered correct if the network presented at least 10 more spikes for true inputs ($[0,1]$ and $[1,0]$) than for the false inputs ($[0,0]$, $[1,1]$). The network and hyper-parameters can all be found in table 5.4. A second experiment is performed, where instead of actively splitting weights of all connections in the first layer in positive and negative weights, the weights are initialized randomly. This should result in a harder task for both the regular and the threshold MSTDPET rules.

As the third experiment, each learning rule will be tested on a small set of fully connected architectures of increasing complexity. Here the task is to perform classification on a set of spiking intensity distributions from section 5.1. The distributions have a maximum Poisson spiking rate of 40Hz and a linear decay away from the maximum intensity neuron. The different architectures, as well as the number of samples the network has to classify, can be found in table 5.5. To compare regular MSTDPET with the version that also learns thresholds, all networks are two layers deep such that the set of hidden neurons' their thresholds can be adjusted. Input layers do not have variable thresholds since

Table 5.4: Network design and hyper parameters for the XOR and classification experiments.

Parameter	XOR	Classification
Spiking threshold	2	2
Minimum spike thresh.	1	1
Maximum spike thresh.	3	2
Minimum weight	-5	-5
Maximum weight	5	5
Refractory duration	$10ms$	$10ms$
Spike trace time constant (τ_+/τ_-)	$20ms$	$20ms$
Eligibility trace time constant (τ_z)	$25ms$	$25ms$
Input spike rate	$40Hz$	$40Hz$
Sample duration	$500ms$	$250ms$
Epochs	200	250
Learning rate	0.1	0.1

they directly forward the input to the following layer.

Weight updates are done in the same manner as for the XOR experiment, once the network generates an output a weight update is initiated. The main difference is the fact that for classification the network has multiple output neurons, instead of one. For each neuron it is known whether it is desirable for it to spike, so each output neuron is provided a separate reward. If the neuron that indicates the class of the input is spiking it is rewarded with a positive reward of 1. Each other neuron that might also be spiking is provided a negative reward of -1. Assigning neuron-specific rewards is not possible for layers that are not the output layer. For those layers, a single scalar reward is provided, which is positive in case the network correctly classifies the input, and negative otherwise. The hyperparameters for training are presented in table 5.4.

Table 5.5: Layerwise number of neurons for the fully connected networks trained with the additive and multiplicative versions of the MSTDPET algorithm.

	Network 1	Network 2	Network 3
Input neurons	30	50	80
Hidden neurons	15	30	50
Number of input samples	5	5	5

6

Experimental Results

The current section presents the results for the experiments described in chapter 5 together with the conclusions that can directly infer based on these results. The experiments are treated in the same order as in chapter 5. All of the experiments were performed on a desktop computer equipped with an Intel 4790K i7 quad-core processor clocked at 4 GHz per core, 16 GB of RAM, and an NVIDIA RTX 2070 GPU.

6.1. Even Discretization Kernel

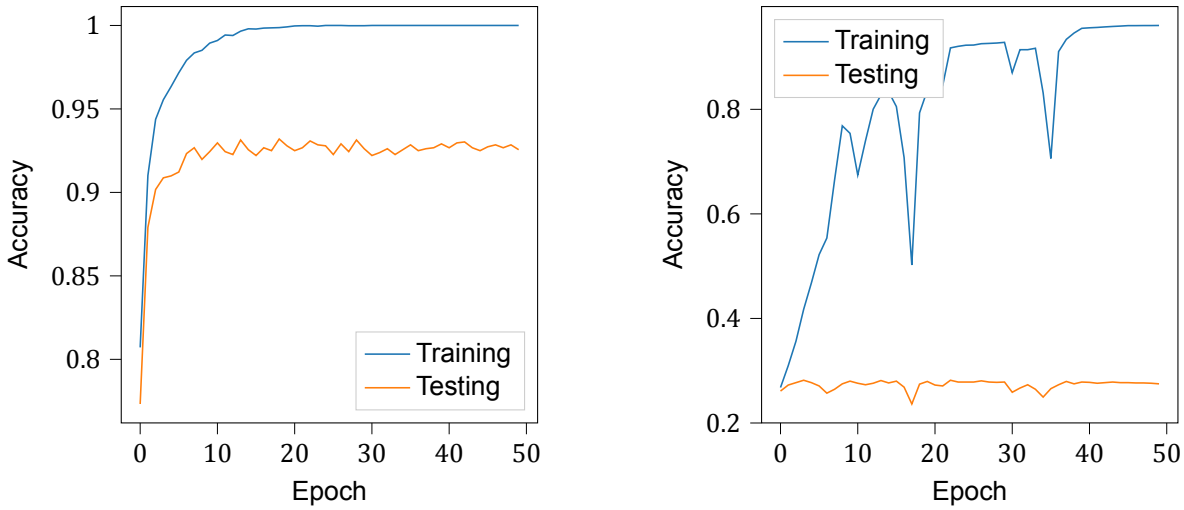
The first part of this experiment consisted of training a 34 layer ResNet on the Caltech101 dataset as a reference experiment. Only the final layer had to be trained since a pretrained network was used, which resulted in rather fast training times of an average of 46 seconds per epoch. Accuracy results during training can be found in fig. 6.1a. The network converged to a consistent accuracy of approximately 93.2% after 50 epochs, with an average inference time of 2.3 milliseconds. The inference time was averaged over the entire training dataset. A summation of these results can be found in table 6.1.

The second part of this experiment revolved around training another 34 layer ResNet, yet this time it was prepended with an event discretization kernel such that the network could process the N-Caltech101 dataset. Training and testing accuracies are shown in fig. 6.1b for comparison with the frame-based ResNet. The accuracies show that the event-based network never generalizes to good results on the test set. The final performance in both classification and inference time is presented in table 6.1, with a training accuracy of 96% and testing accuracy of 28.2%. For clarity, the results from the original paper with 81.7% testing accuracy are shown in column 3. In addition to training performance, the number of epochs is added to the last row. The number of epochs of the original paper is not mentioned explicitly, but since the authors mention adjusting the learning rate twice after 10.000 and 20.000 iterations an approximate number of epochs can be deduced. The authors use a batch size of 60 for the N-Caltech101 dataset which consists of roughly 7000 images, assuming an 80-20 train-test split. This means that after roughly $10.000/(7000/60) \approx 85$ epochs the learning rate is adjusted, and this is repeated after again 85 epochs. This makes it likely that the network in the original paper was trained for about $3 \cdot 85 \approx 250$ epochs.

Reproducing the original results as published in Gehrig et al. [23] was unsuccessful as can be seen in the contrast between the results published in Gehrig et al. [23] and the results of the author's experiment. For comparison with the frame-based results, the results for event-based data from Gehrig et al. [23] is used. Classification accuracies on the test set differ by about 12%, which is a considerable gap in performance. This difference in performance is likely because the ResNet architecture was designed for frame-based images and some information losses occur in the event discretization. The difference in the number of training epochs (50 for frame-based and an estimated 250 for the event-based) provides a skewed image since the ResNet that was used was extensively pretrained, meaning the effective number of training epochs is a lot higher than 50.

Regarding inference times, there is a considerable difference. The frame-based network's inference time is about 3 times faster than that of the event-based network. A difference in favor of the frame-based network was evident due to the addition of the event discretization step and the 18 channel

image being fed into the ResNet. It must be noted that the inference times are based on different hardware, where the event-based network was trained on a faster NVIDIA 2080 Ti GPU and a lower clocked 2.7GHz Intel i7 CPU of which the number of cores is unknown. Both of the systems used 16GB of RAM. Since most of the computations performed in training ANNs rely on the speed of the GPU, it is reasonable to assume that the inference time of the frame-based network on the desktop equipped with the NVIDIA 2080 Ti GPU would be comparable or faster than the inference time mentioned in table 6.1.



(a) (b)

Figure 6.1: Accuracy curves on training and testing datasets for (a) regular 34 layer ResNet trained on frame-based Caltech101 dataset and (b) event discretization kernel prepended to 34 layer ResNet network trained on N-Caltech101 dataset.

Table 6.1: Performance comparison of 34 layer ResNet trained on the Caltech101 dataset and the 34 layer ResNet prepended with a learnable discretization kernel trained on the Neuromorphic Caltech101 dataset.

	Frame-based	Event-based	Results from paper
Training Accuracy	100 %	96.0 %	–
Testing Accuracy	93.2 %	28.2 %	81.7 %
Average inference time	2.3 ms	14.7 ms	6.85 ms (with hashmap)
Training time	38 m	238 m	–
Epochs	50	50	~250

6.2. Phased LSTM

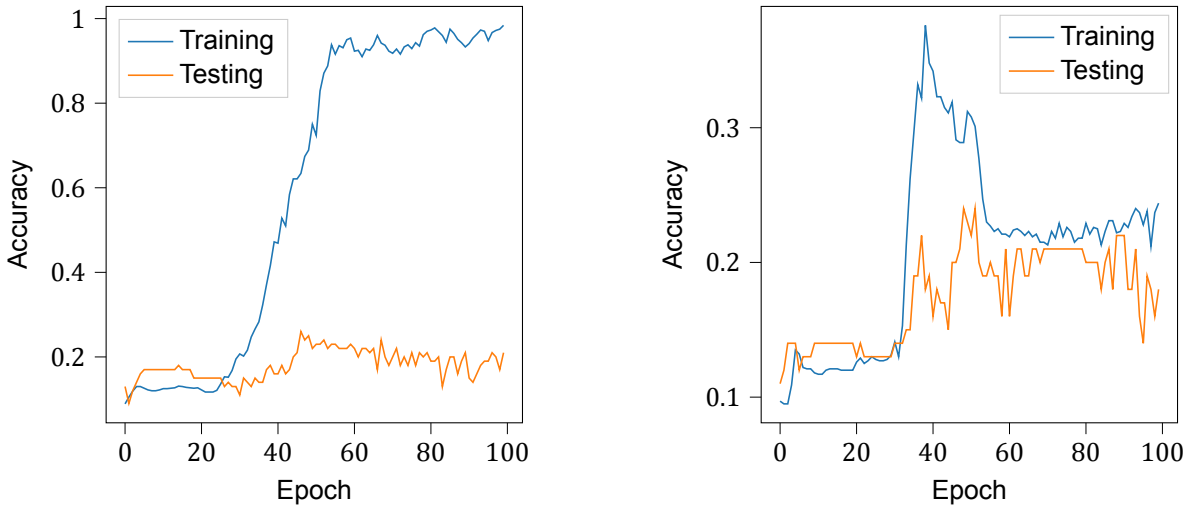
The accuracy progression for both training and testing of the PLSTM network trained of the event-based N-MNIST dataset is presented in fig. 6.2a. The accuracy progression for its LSTM based control experiment is shown in fig. 6.2b. In addition, table 6.2 provides an overview of the most important results for both experiments and the respective control experiments mentioned in section 5.3 and the testing accuracy as stated in Neil et al. [70]. There are three important observations that will follow next.

Firstly, it is observed that in both experiments the PLSTM network performed better than its LSTM counterpart in classification accuracy. For the event-based experiments this is seen in a training accuracy of 97% for the PLSTM and 23% for the LSTM, whereas for the performance on the test dataset the differences are absent with 20% accuracy for both networks. For the frame-based experiment the PLSTM network outperforms the LSTM both on the train (99% for the PLSTM and 15% for the LSTM)

and test (87% for the PLSTM and 11% for the LSTM) datasets. This confirms the observation in the original paper that the PLSTM is more sample efficient, and even by a large amount. In both situations the LSTM networks' their performance is slightly better than random guessing for 10 classes.

Secondly, it can be seen that the inference time per sample for the PLSTM is higher than that for the LSTM. This was expected since the PLSTM requires more computational steps to evaluate. When comparing these inference times to the convolutional networks of section 6.1 as shown in table 6.1, it is clear that computational time for both of the recurrent networks is very high. Even the frame-based LSTM, which has the lowest inference times, takes 0.2 seconds to evaluate. This is an important aspect to take into consideration when using any form of LSTM, since the networks presented here consisted of only two to three layers.

Thirdly, the author was unable to reproduce the performance presented in the original paper as seen in 77% difference in classification accuracy. Looking at fig. 6.2a, the event-based PLSTM network shows that it can learn on the training data by achieving 97% training accuracy yet is not able to generalize to new data. It is likely that to increase performance on the test dataset the network has to be trained for more epochs. Another explanation could be a mistake in the implementation, even though this seems less likely because of the good performance on the frame-based dataset. Because there is no clear reason to doubt the original results, further comparisons of the PLSTM with other methods will be based on the original paper's results.



(a) PLSTM

(b) LSTM

Figure 6.2: Training and testing accuracies on the event-based N-MNIST dataset during training of the (a) PLSTM and (b) LSTM network.

Table 6.2: Training and testing results for the PLSTM and LSTM experiments performed on a sub-set of the N-MNIST dataset, the results from the original paper [70], and a reference experiment on frame-based MNIST data. Event-based = Ev., Frame-based = Fr.

	Ev. PLSTM	Ev. LSTM	Fr. PLSTM	Fr. LSTM	Paper
Training Accuracy	97 %	23 %	99 %	15 %	–
Testing Accuracy	20 %	20 %	87 %	11 %	97.27 %
Average inference time	3.6 s	2.0 s	0.3 s	0.2 s	–
Training time	146 m	97 m	17 m	10 m	–
Epochs	100	100	100	100	–

6.3. Spike Layer Error Reassignment in Time

Experiments as presented in section 5.4 revolved around tracking gradients of all layers during training. Firstly, a description of the type of results is provided that applies to both the fully connected and convolutional network experiments. Secondly, the results of the fully connected networks are analyzed, followed by the convolutional networks' results. Thirdly, this section finishes with a conclusion about the hypothesis and a comparison between the two types of networks.

In total four different fully connected, and four different convolutional networks were trained for this experiment. Only the results for a three-layer fully connected network and four-layer convolutional network will be shown since these networks gave the clearest impressions of the results. Results for the other networks showed the same patterns in their gradients. The results for the other networks can be found in appendix A. Results for the SNNs will be compared with the results of their ANN counterpart.

Fully Connected Networks

All of the following plots concern the statistics of layer-wise gradients during training. The means and standard deviations of the two-layer, fully connected SNN can be found in fig. 6.3 and fig. 6.4. The mean and standard deviation for its ANN counterpart are shown in fig. 6.5 and fig. 6.5. The gradients for the SNN and ANN follow a comparable pattern over time, but the gradients of the SNN are several orders of magnitude larger than for the ANN. The mean moves towards zero in just a few iterations for the SNN, and for the ANN it starts very close to zero and stays around it. This indicates that the weights converge towards a consistent value. Another observation is the fact that the standard deviation for the gradients of the final layer is higher than those of the earlier layers throughout the entire training process, while for the earlier layers the magnitude of the gradients is very comparable. This makes sense from the perspective that final layers can be adjusted more aggressively since no other layers depend on their output statistics. For earlier layers, it is thought to be important to not move its weights too aggressive, as this can change the input distribution for the following layers and thus decrease performance.

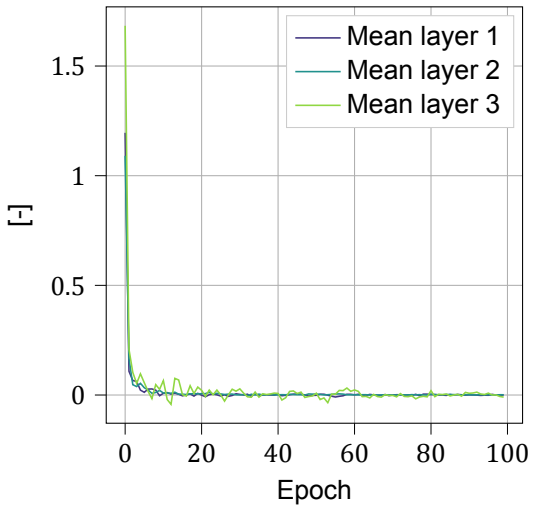


Figure 6.3: Mean of gradients for three layer fully connected SNN during training.

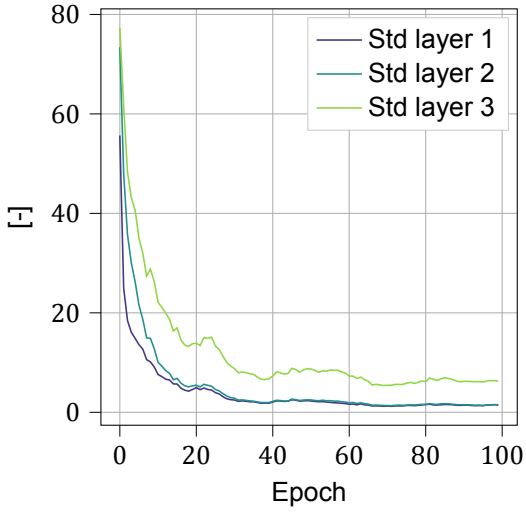


Figure 6.4: Standard deviation of gradients for three layer fully connected SNN during training.

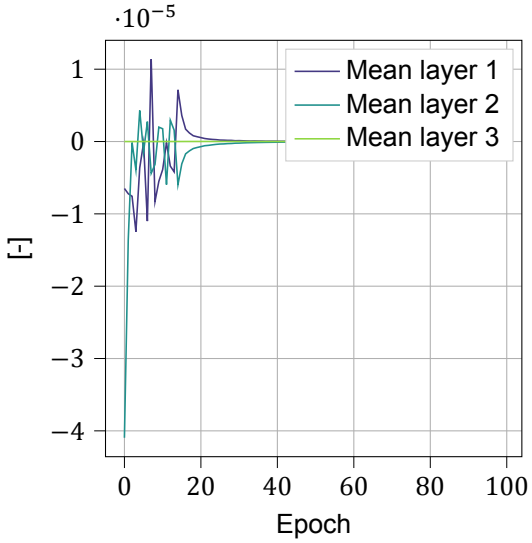


Figure 6.5: Mean of gradients for three layer fully connected ANN during training.

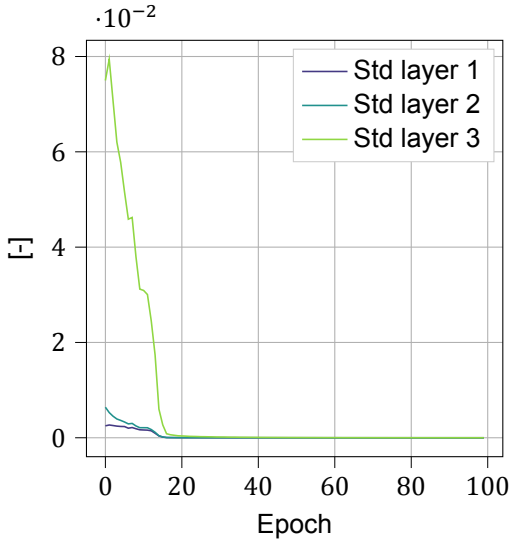


Figure 6.6: Standard deviation of gradients for three layer fully connected ANN during training.

Convolutional Networks

The following plots cover the statistics of the gradients of the convolutional neural networks during training. For the SNN, the means and standard deviations can be found in fig. 6.7 and fig. 6.8. For the ANN counterpart the means are presented in fig. 6.9 and the standard deviations in fig. 6.10. Just as with the fully connected networks, the gradients follow the same pattern during training (both the mean and standard deviation), but the gradients of the SNN are several orders of magnitude larger. A clear difference compared to the fully connected networks is the fact that for convolutional networks the first layer has the largest gradients over time, instead of the final layer. Also, these gradients of the first layer never seem to stabilize, visible from the consistently high standard deviations in fig. 6.8.

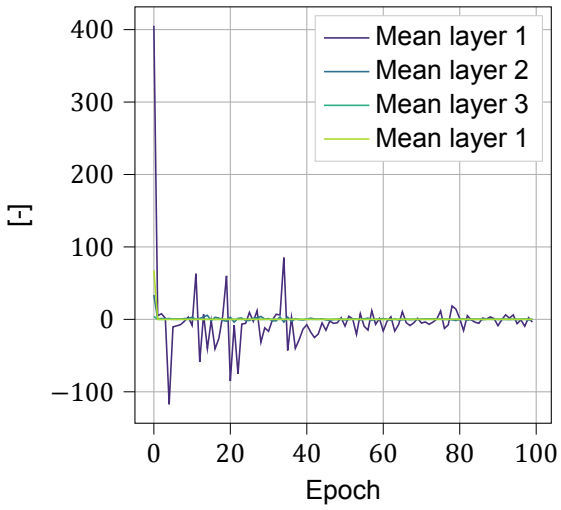


Figure 6.7: Mean of gradients for four layer convolutional SNN during training.

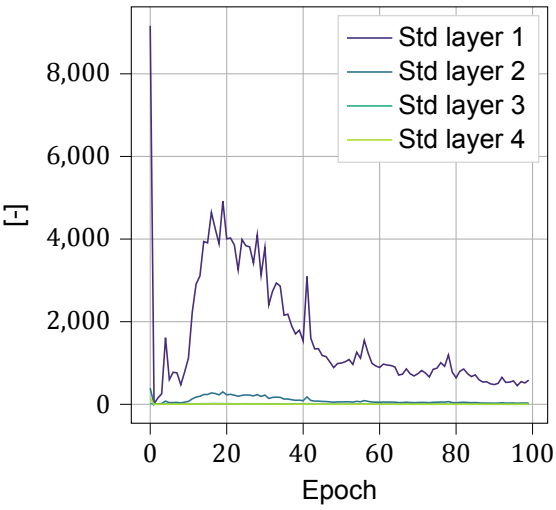


Figure 6.8: Standard deviation of gradients for four layer convolutional SNN during training.

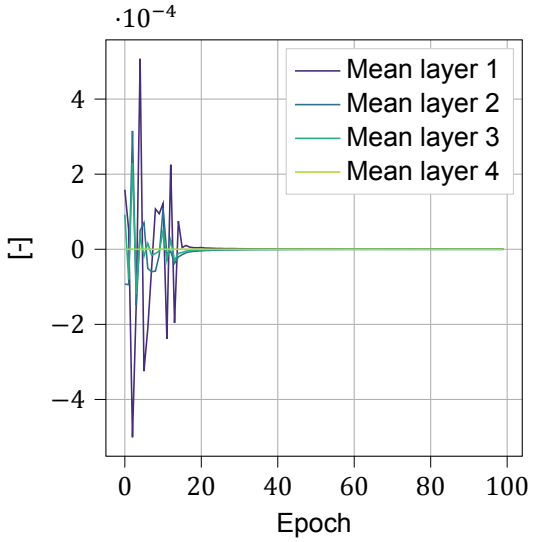


Figure 6.9: Mean of gradients for four layer convolutional ANN during training.

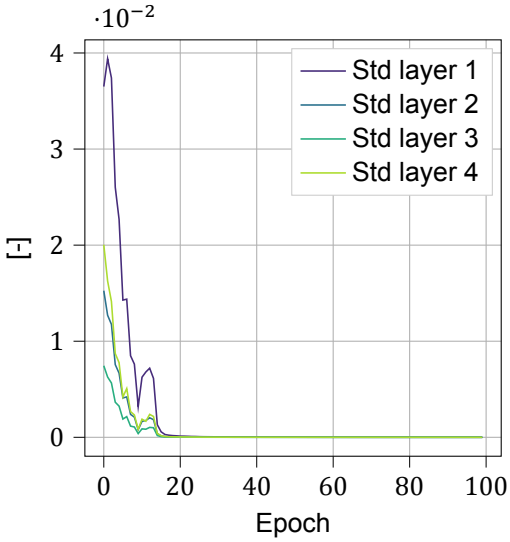


Figure 6.10: Standard deviation of gradients for four layer convolutional ANN during training.

General Results

To complement the previous results, the networks' accuracies after training are shown in table 6.3. It is interesting to see that the fully connected SNN outperforms the ANN on the test data, but for the convolutional networks, it is the other way around.

Table 6.3: Training and testing results for the three layer fully connected and four layer convolutional networks. The first two columns show the results for the networks trained on event-based data with the SLAYER algorithm, the last two columns for the network trained on frame-based data with regular backpropagation.

	Event-based		Frame-based	
	Training Acc.	Testing Acc.	Training Acc.	Testing Acc.
3 Layer Fully Connected	100%	93%	100%	90%
4 Layer Convolutional	100%	66%	100%	96%

Based on these results the following hypothesis is rejected: *The SLAYER algorithm suffers from vanishing gradients when applied to moderately sized and/or sparsely connected networks.* The gradients of the convolutional networks show the opposite pattern of what was expected, which is true for both SNNs and ANNs. All results together do show consistent patterns when comparing SNNs and ANNs, the gradients show similar patterns for equal architecture designs. The clear difference is the considerably larger magnitude gradients in SNNs. These very large (often called exploding gradients) might be the cause for the underperforming of the spiking CNN.

6.4. RSTDP for classification

As described in section 5.5, two R-STDP experiments were performed focused at comparing regular MSTDPET and an adaptation which also adjusts thresholds based on learning performance. The first experiment was to learn a network to act as a logic XOR gate, and the second experiment concerned learning to classify a small set of spiking intensity distributions. The results from these experiments will be presented next, followed by a general conclusion.

XOR

As mentioned in section 5.5, the XOR experiment was mainly performed to see if the adjustment of learning both weights and spiking thresholds would not break the learning rule. Hyperparameters were set such that the experiment converged to a decent performance of 98.5% accuracy using regular MSTDPET (not learning thresholds) and halve of the connections limited to positive weights, and the other halve to negative weights. See table 5.4 for the hyperparameters values. Next, the same settings were applied to the threshold learning rule, which resulted in an accuracy of 97.2%. For the third experiment, with all random weights, the accuracy of the network did not surpass 75.0% accuracy. This is a trivial result, since the network is now acting like a regular OR gate, resulting in the following pattern: $[0,0] \rightarrow 0; [0,1] \rightarrow 1; [1,0] \rightarrow 1; [1,1] \rightarrow 1$. It always responds with an output once it gets at least one input, and remains silent if it receives no input. As a side note, there was decent variability in performance between multiple runs, with final accuracies deviating up to 15%. For brevity, results for the best performing iterations are shown here.

Figure 6.11a and fig. 6.11b show the distributions of the weight values for both layers of the network after training with regular MSTDPET (fig. 6.11a) and the threshold MSTDPET (fig. 6.11b). Both figures show results for the networks where the weights are split into positive and negative weights before training. The distributions show very comparable shapes and densities for both the first and second layers, which indicates that they converged to comparable solutions. This is also visible in their accuracies. A last important point is that the weight distributions for the first layer (the blue distributions) show a slight bi-modal behavior. This is expected because of the split into positive and negative weights, and desirable, as this split is needed to represent an XOR gate. The activity of the hidden neurons during a single epoch, where each of the four inputs is shown once, can provide extra insight into the possible differences between the two learning rules. A summation of the total number of spikes emitted during an epoch for regular MSTDPET is found fig. 6.12a, and for threshold MSTDPET in fig. 6.12b. Same as for the weight distributions, the hidden layer neuron activity is very comparable for the two networks. From this, it can be concluded that learning spiking thresholds did not significantly impact training or final performance in the XOR experiment, compared to using regular MSTDPET.

In addition to the previous figures, fig. 6.11c shows the weight distribution of the network for which the weights were initialized at random. Compared to the other two figures the distributions are relatively flat, especially for the weights of the final layer (orange), and are more spread towards the limits of the weights. A large spread is an undesirable property since it indicates that the learning rule is defaulting to simply reinforcing (depressing) already strong (weak) connections. This correlates with that this network was not able to learn the XOR task sufficiently. Figure 6.12c shows the total activity of the hidden neurons during a single epoch for the randomly initialized network. It shows that the spiking activity of the hidden layer is about one and a half times as high compared to the successful networks. The free weight network is also less selective in its hidden layer, meaning more neurons have high activity in comparison to the other two networks.

For verification, the spiking threshold distributions for the networks trained with threshold learning MSTDPET are compared. Only the thresholds for the hidden layer were trained. The distribution for the network with split weights is shown in fig. 6.13a, and for the network with random weights in fig. 6.13b. The average threshold value is higher and more centered around a single value for the randomly initialized network. This means the thresholds cannot be the cause of the high activity of its hidden layer. Thus the weights being more spread towards high values (fig. 6.11c weights range up to five) may cause the lackluster performance of the network. This indicates that MSTDPET (likely R-STDP in general) is unable to recover from a bad initialization.

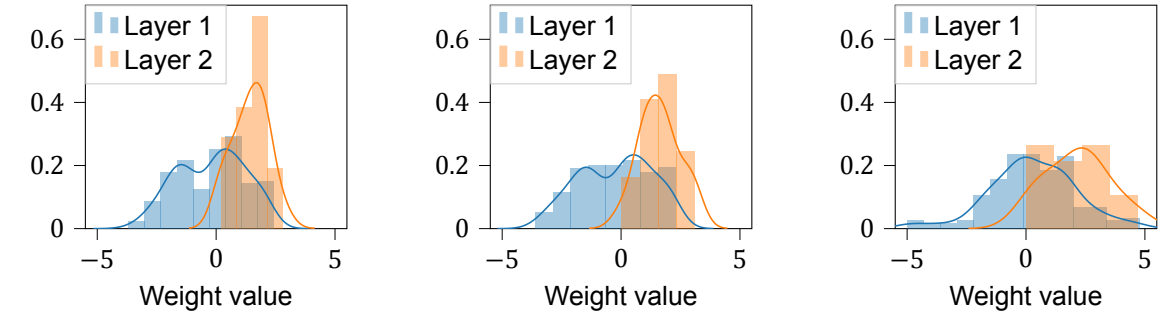


Figure 6.11: Weight distributions after training of the XOR network with (a) regular MSTDPET, (b) threshold MSTDPET, and (c) with threshold MSTDPET and connections not split into predefined positive and negative weights.

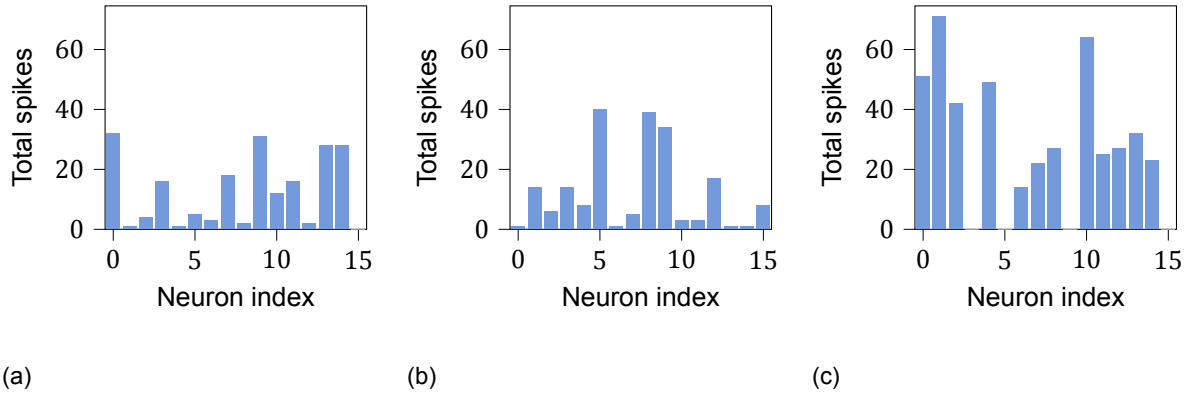


Figure 6.12: Activity of the hidden neurons during a complete epoch, *after* training of the XOR network with (a) regular MSTDPET, (b) threshold MSTDPET, and (c) threshold MSTDPET and connections not split into predefined positive and negative weights.

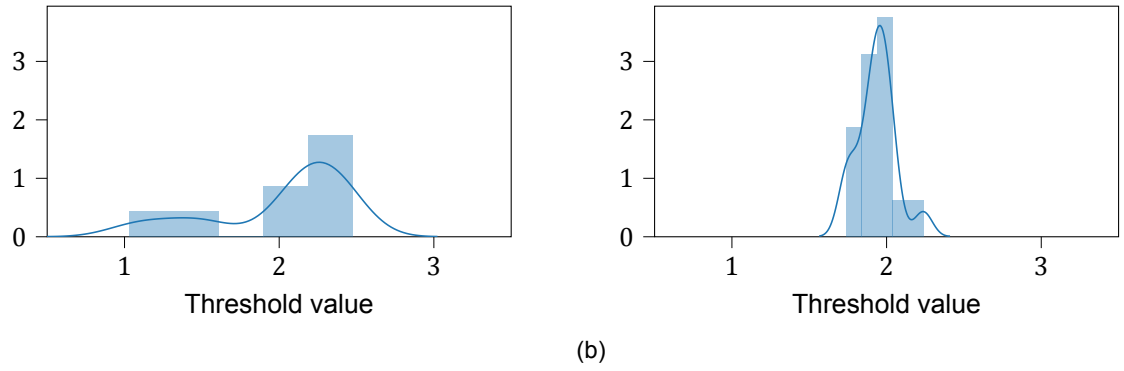


Figure 6.13: Thresholds distribution after training of the XOR with (a) threshold MSTDPET, and (b) threshold MSTDPET and connections not split into predefined positive and negative weights.

Classification

During the classification experiment as described in section 5.5, three different fully connected neural network architectures were tested (see table 5.5). All networks were trained with regular MSTDPET and threshold MSTDPET. To account for differences due to weight initialization all classification experiments were repeated up to five times each. Only results that are most illustrative for the observed patterns are shown in this section.

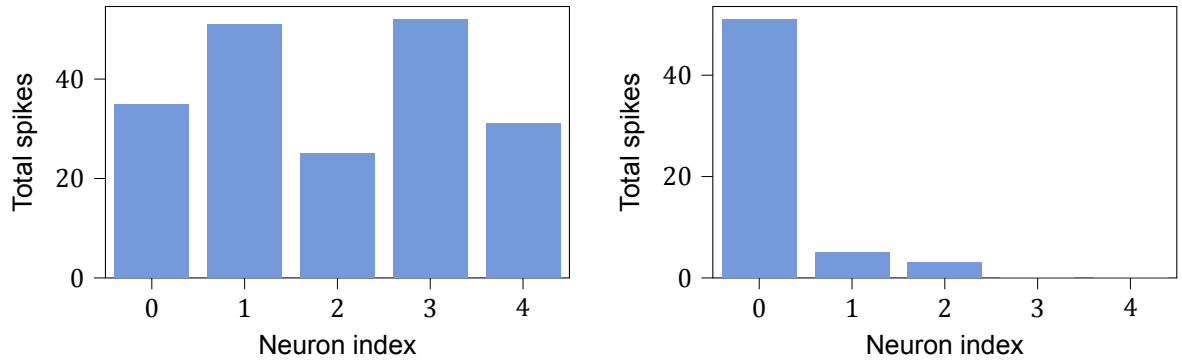
During all experiments, only a single network achieved an accuracy of over 80%, at 89.5%. The best and worst accuracies for each architecture and learning rule combination are shown in table 6.4. The threshold version of MSTDPET was able to achieve (almost) equal or better accuracies on all three

networks. Yet, it also has the worst performance for each of the networks. In general, the variation between different initializations was rather large for both methods, and thus might also be the cause for the differences between regular and threshold MSTDPET. Also, there is a significant difference in performance between the two smaller networks and the largest network. The following results will focus on exploring the differences between initializations, followed by an exploration of the difference in performance for varying network sizes.

Table 6.4: Best and worst classification performance for each network and learning rule combination.

	Regular MSTDPET			Threshold MSTDPET		
Input neurons - Hidden neurons	30-15	50-30	80-50	30-15	50-30	80-50
Best accuracy	60.0%	80.0%	12%	77.0%	89.5%	40%
Worst accuracy	40.0%	59.0%	8.0%	40.0%	21.5%	3.0%

To compare different initializations of a single network, the spiking behaviors over a single epoch were compared for both the hidden and output layers. A perfect network should have all its output neurons spike a roughly equal amount of times throughout an epoch. The activity of the output layer of the best performing network (89.5%) during a single epoch, after training, is shown in fig. 6.14a. This network consisted of 50 input neurons, 30 hidden neurons, and 5 output neurons (the final layer is the same for all networks). It was trained using threshold MSTDPET. This was the only network that had above 80% accuracy and is also the only network where all output neurons spike a considerable amount of times. For comparison, fig. 6.14b shows the output activity of the same network but initialized with a different seed. Only a single neuron shows relevant activity, two neurons spiking a few times, and two neurons not spiking at all. This pattern occurred in each of the experiments where one or more of the output neurons are almost or completely inactive after training. Once the neuron is dead, the learning rules are unable to reactivate it. This is a considerable issue since only one out of 30 experiments did not suffer from a dead output neuron.

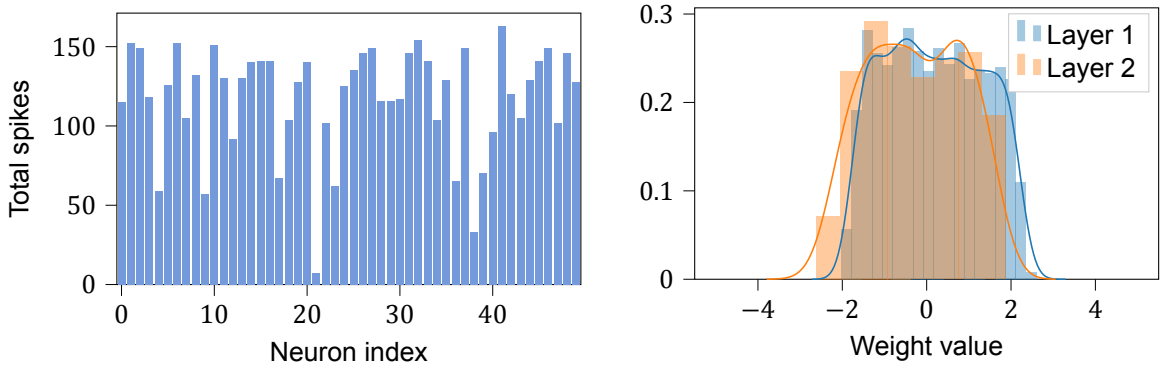


(a) (b)

Figure 6.14: Activity of the output neurons during a complete epoch, *after* training of the network (50 input, 30 hidden) with *threshold* MSTDPET. Figure (a) shows the output of the network with the best final performance at 89.5% accuracy, (b) shows the worst performing initialization with an accuracy of 21.5%.

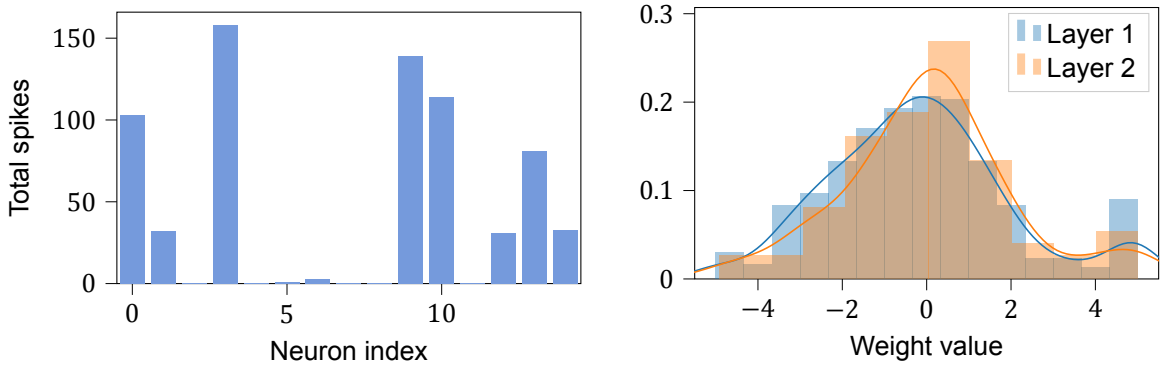
Lastly, the difference in performance due to network size is explored. The lackluster performance of most of the large networks (80 input neurons, 50 hidden neurons) is due to an almost complete absence of output spikes. This is the same pattern as described previously. Figure 6.15a shows the activity during one epoch of the hidden layer of large network trained with MSTDPET. This network achieved 12% accuracy. It shows that almost all of the hidden neurons are very active throughout a single epoch, with on average around 100 spikes per neuron. This is a lot, considering that the theoretical maximum number of spikes a neuron can emit under the settings used is around 200. For comparison, fig. 6.16a shows the hidden layer activity for the smallest network trained with MSTDPET. This network's accuracy is 60%. Whereas some of the neurons are rather active, it is only a few of them.

The over-activity of the hidden layer of the large network might be a cause for the bad performance. To complement these results, the weights after training for the large network are show in fig. 6.15b, and in fig. 6.16b for the small network. The weights of the large network are small compared to the other network, shown by the narrow distribution. The high activity of the hidden neurons is not due to large weights, but due to a large number of pre-synaptic connections per neuron. For the large network, this is 80 connections, whereas for the smaller network it is only 30. This is an indication that the weights should be scaled depending on the number of presynaptic connections to a neuron.



(a) (b)

Figure 6.15: Statistics for a 80 input, 50 hidden network that was trained using MSTDPET and achieved 12% accuracy. (a) The neuronal activity of the hidden layer, and (b) the weight distributions after training.



(a) (b)

Figure 6.16: Statistics for a 30 input, 15 hidden network that was trained using MSTDPET and achieved 60% accuracy. (a) The neuronal activity of the hidden layer, and (b) the weight distributions after training.

General Results

In general, it seems that the learning of thresholds amplifies both the strong and weak performance of MSTDPET. This is likely because the learning of thresholds further amplifies the tendency of additive R-STDP to reinforce strong connections and weaken weak ones. These results should be taken as an indication since the difference in performance might also be caused by MSTDPET its sensitivity to weight initialization and network size.

7

Experiments Discussion

In this preliminary analysis, experiments were performed to explore and compare the capabilities of ANNs and SNNs in analyzing asynchronous, event-based data in a supervised learning setting. Besides comparing ANNs with SNNs, the preliminary analysis also included two sub-goals that were aimed at identifying the best event processing paradigm for ANNs and SNNs respectively. This chapter presents the main findings and discussions for each of these goals, as well as the implications this preliminary analysis has on the final thesis.

Firstly, a brief discussion on the datasets is provided. Secondly, time discretization and direct event processing with RNNs in ANNs are treated. Thirdly, gradient-based learning methods and correlation-based learning methods for supervised learning in SNNs are compared. Lastly, this chapter concludes with a discussion on processing asynchronous event-based data with ANNs or SNNs and which approach will be used during the final thesis.

7.1. Datasets

During the experiments three different datasets were used, the N-MNIST dataset, the N-Caltech101 dataset, and a simple spiking intensity distribution dataset. The N-MNIST and N-Caltech101 datasets can be considered as the current go-to datasets for experimentation with event-based vision. The reason for this is that these datasets were captured with an actual Asynchronous Time Based Image Sensor (ATIS) camera. This is more realistic than converting frame-based images to an event-based representation by using Gabor filters or converting the pixel intensity to a Poisson spike rate. The second reason is that these datasets their frame-based representations are heavily used in many computer vision experiments, resulting in a lot of reference work to compare results with. A possible downside of these datasets is that they are not designed to challenge an event-based camera its weaknesses, like identifying smooth and/or large surfaces due to the absence of edges. Whereas this might be worth an investigation in itself, it is not within the scope of this work.

The spiking intensity distribution dataset proved useful in very small scale experiments. Especially for testing adjustments to, or entirely new iterations of a learning rule. It allowed for quickly changing the number of samples, and since the samples were generated using a stochastic process it forced the network to generalize at least somewhat. Yet, due to its one-dimensionality and very low variance between the samples it is only really suited for very quick iterations and not for thoroughly testing the limits of a network or learning rule.

7.2. Artificial Neural Networks

Regarding the experiments with ANNs, the goal was to get insight into the differences between time discretization techniques and direct processing of events through RNNs. For discretization, the algorithm from Gehrig et al. [23] was used and for direct processing the PLSTM from Neil et al. [70]. The author was not successful in replicating the results mentioned in either of the original papers as shown in section 6.1 and section 6.2, so for further analysis, the performances mentioned in the original papers were used. When doing so it became clear that for both experiments there was a considerable loss in

accuracy and an increase in inference time when processing event-based data instead of frame-based data. Inference times got as high as 15 milliseconds for the discretization kernel, and 3.6 seconds for the PLSTM. This difference in performance can have multiple causes, which will be treated separately for the discretization kernel + ResNet network and the PLSTM network.

For the discretization kernel and ResNet network, the architecture was likely not optimal. Especially the 34 layer ResNet was designed and pre-trained to process 3 channel (RGB) images and not a 20 channel image based on the feed of an event-based camera. Within these 20 channels, there is likely a difference between the first and last channel, simply because they represent a set of events that occurred the furthest apart in time. Since this time dimension is still present within the data, albeit of a rather short duration of about 300ms, a 3D convolution operation can become valuable because events close to each other in time are likely more correlated than those far apart. The second point of attention is that the mechanism for discretizing events into frames may be improved. Using a form of RNN might be useful since events that are close to each other in time are likely to be related in some way, possibly requiring a different form of discretization.

In the case of the PLSTM, possible improvements are different. Whereas the recurrent structure allows it to correlate events along the time dimension, the network discards a lot of spatial information. The translational invariance of the convolution operation is crucial in most, if not all, computer vision applications. There is no direct way of adding this to a PLSTM, but as mentioned earlier, the PLSTM might prove valuable in pre-processing events before feeding them into CNN. Secondly, the PLSTM was especially slow in inference time at 3.6 seconds per image. This is due to events being processed sequentially. Adding some kind of batching along the time dimension may speed up the process.

An observation that is shared among both of the networks, and possibly the most important one, is that both methods show a large increase in inference time compared to frame-based ANN. This was expected because of the added dimensions to the event network compared to the frame-based network. Inference times for discretization methods add up to about 14.7 milliseconds, and 3.6 seconds for the PLSTM in small to moderately sized networks for both types of networks. Unless a method is designed to take advantage of the sparsity of the event stream in ANNs, this is likely to remain an issue.

The last remark is that the physical phenomenon based on which event-based images are generated is different from that of frame-based images. Since event-based data is generated due to changes in the observed scene, this type of data is likely more suited for interpreting dynamic properties of the environment like the direction an object is moving in, the speed of motion, or ego-motion. This is in contrast to frame-based images which inherently provide rich details of the current state of the environment, yet have trouble capturing its dynamics. If this turns out to be true, direct performance comparisons between event-based networks and frame-based networks should be done with care. Ideally, one type of network excels at both, but this is likely only possible at the cost of a lot of computing and energy. In general, there remains somewhat of a paradigm mismatch between ANNs and event-based data, forcing an even larger trade-off between efficiency and accuracy than with frame-based images.

7.3. Spiking Neural Networks

Experiments with SNNs were intended to generate insight in to, and compare gradient-based and correlation-based supervised learning rules for SNNs. For gradient-based learning the SLAYER algorithm from Shrestha and Orchard [94] was used, and for correlation-based learning the R-STDP rule from Florian [20] acted as a starting point. The main observation is that for both methods there is room for improvement. This results in smaller experiments showing promising results, but the methods tend to break down for larger and more complex experiments. The SLAYER algorithm was able to train more complex networks compared to R-STDP. Yet, the fact that part of the experiments converges is promising and indicates that the methods are worth being researched further.

Gradient Based Learning

As stated in section 5.4, the SLAYER algorithm is the current state of the art when it comes to gradient-based learning in SNNs. Experiments were performed to test the following hypothesis: *The SLAYER algorithm suffers from vanishing gradients when applied to moderately sized and/or sparsely connected networks.* Based on the results the hypothesis was rejected. Whereas gradients decayed from output to input layers in fully connected networks, the exact opposite happened in convolutional networks.

Even more so, when using the SLAYER algorithm to train SNNs gradients tended to be orders of magnitude larger than in regular ANNs. It seems that the algorithm might have issues with so-called exploding gradients. A possible cause for this is the loss function that is used as this directly scales all gradients if it changes. A different cause could be that the gradient approximation of a spike results in large gradients. In section 5.4 it was mentioned that the gradient of the PDF of fig. 5.7 has very small gradients in its tails, yet it also has large gradients towards its peak. According to Shrestha and Orchard [94], the loss function stimulates infrequent random spiking of otherwise dormant neurons. This is done to make them easily excitable in case of an infrequent pattern. As a result, most neurons should be near a change in spiking state, which would result in high gradients due to the peak of the PDF. Exploring the relation between the loss function and the PDF would make for interesting research.

The second discussion point is more focused on the difference in performance between fully connected and convolutional networks, when implemented as ANNs or SNNs. The far worse performance of the convolutional SNN compared to the ANN is likely caused by the consistently large and varying gradients of the first layer, as seen in fig. 6.8. If the problem of the large gradients is solved, the performance of convolutional SNNs might improve. For the fully connected networks it is the other way around, the SNN outperforms the ANN with a small margin. A possible explanation for this is the fact that these experiments were performed with a small dataset of just 1000 training images. The added time dimension for SNNs makes it that there is a larger amount of data per sample to train on. Also, this added time dimension also likely is noisy, which could act as an important regularizer. Especially for a fully connected network with a large number of connections per neuron, this difference in data may cause a difference in performance.

Correlation Based Learning

Correlation-based learning rules, specifically MSTDPET [20], were tested for their ability to perform supervised learning. A downside of this experiment compared to the others is the fact that it was performed on very small networks and datasets, so a direct comparison is not possible. Three main conclusions can be drawn from the results presented in section 6.4. First and foremost is that MSTDPET seems to have issues with increasing network size. This was expected, since it quickly starts to suffer from the curse of dimensionality, due to using a single scalar reward value for updating thousands of weights.

The second conclusion is that performance is greatly dependent on the initialization of weights. An ”unlucky” initialization can result in that some neurons rarely spike during training. Since reinforcement learning methods rely on trial-and-error, meaning they only reinforce a desirable outcome once it occurs, it is not able to move the network’s weights in the correct direction if these desirable outcomes never occur. This weakness is further strengthened by the fact that additive formulations of STDP tend to result in bi-modal distributions, meaning the weakly initialized connections tend to be further depressed until the connection is almost dead. The solution to this problem knows two sides, having a learning rule that can actively seek for correct solutions and a good weight initialization scheme. The former will be treated first. Since it is not known what loss or reward function STDP optimizes for, the first solution would be to derive a correlation-based update rule that optimizes a known loss function. The second solution works in the opposite direction, finding a function STDP optimizes by trying many different loss functions and looking for the best fit. The last solution would be to engineer a feedback channel that locally influences weight updates, for example through the use of random feedback weights (see section 2.2.5).

The third and last conclusion is related to the initialization of weights, together with deciding on the magnitude of the many hyperparameters for training SNNs. Besides that spiking neural networks require many hyperparameters, several of these parameters also have different optimal values per layer. A good example is finding a good average initialization weight, which varies from layer to layer to excite all neurons roughly equally. With the number of pre-synaptic connections per neuron varying from layer to layer, the optimal average weight per connection also varies, assuming other parameters like the firing threshold are kept constant. This became clear since the average neuron activity varied considerably between layers of different sizes. For training regular ANNs the same observation was made, where the distribution for sampling weights during initialization is based on the dimensions of the layer [30].

This observation also holds during training, where the optimal average weight still varies from layer to layer. Setting the hyperparameters that control the weight range per layer (upper and lower bound

for additive STDP, or a target weight for multiplicative STDP) is undesirable or impossible, even with hyperparameter optimization. In many works homeostasis mechanisms are added to control and limit the activity of neurons, yet these are often network-specific or even layer-specific. Good examples are different firing thresholds or winner-takes-all mechanisms. To partly alleviate this problem it is suggested to set a reference spiking activity for each neuron. Since it regularizes a single neuron the mechanism should be insensitive to the layer dimensions. When combined with good weight initialization it should also be able to deal with a varying number of pre-synaptic connections per neuron. Also, it can act as a mechanism that keeps neurons from ”dying”. Last but not least, a low reference activity also acts as regularizing noise.

Conclusions Preliminary Analysis

This preliminary analysis lays the foundation for the research for the thesis following this work by exploring and comparing the processing processing of asynchronous, event-based data through ANNs and SNNs. This section presents the most important conclusions and which approach will be used during the rest of this thesis.

Firstly, because of the novelty and difficulty of processing event-based data with either ANNs or SNNs, relatively small and simple datasets were used during the experiments. The N-MNIST and N-Caltech101 [76] datasets are the event-based counterparts of the very common MNIST [51] and Caltech-101 datasets [55]. It is very likely that the N-MNIST dataset will be used to validate any model or learning rule during the final thesis, the use of the N-Caltech101 datasets depends on the progress that is made. The spiking intensity distribution dataset proved easy to work with, yet it will only be used for initial quick testing of new methods.

Secondly, two forms of ANNs for processing event-based vision data were tested, one based on discretizing events into a frame-based representation [23], and the other based on directly processing events with PLSTMs [70]. The mismatch between the frame-based paradigm of ANNs and event-based data is causing a decrease in accuracy and an increase in inference time compared to networks for frame-based images. Several improvements were suggested in section 7.2, but the author does not consider the ease of using ANNs to outweigh the increase in inference time. It is because of these reasons that ANNs will not be considered during the final thesis.

Thirdly, experiments with SNNs showed mixed results. Small networks showed promising results for the gradient-based learning rule SLAYER [94], whereas it showed difficulty converging for deeper networks. It forms the current state of the art, but the problems with exploding and vanishing gradients make it hard to train more complex architectures.

Fourthly, correlation-based methods in their current form showed lackluster results in simple classification tasks. Variations of the MSTDPET [20] learning rule were unable to reliably optimize a network due to a large sensitivity for weight initialization, a lack of proper homeostasis mechanisms, and the curse of dimensionality most reinforcement learning methods suffer from. The positive side of these experiments is that several points of improvement have been identified, as well as possible methods to do so.

Finally, is it decided to go forward with correlation-based methods for the rest of this thesis. Even though these methods are the least developed, they have a lot of opportunities for improvement and have fewer limitations compared to gradients based methods. This mostly concerns not requiring differential operations and the absence of the vanishing or exploding gradients problem. As a result, a successful correlation-based learning rule should be able to train a larger diversity of architectures. Also, correlation-based methods can be used to train SNNs online on a neuromorphic chip. This could result in greatly improved inference times and energy consumption.

Bibliography

[1] L. F. Abbott and Sacha B. Nelson. Synaptic plasticity: Taming the beast. *Nature Neuroscience*, 3(11):1178, November 2000. ISSN 1546-1726. doi: 10.1038/81453.

[2] M. Abeles. *Corticonics: Neural Circuits of the Cerebral Cortex*. Cambridge University Press, Cambridge ; New York, 1 edition edition, February 1991. ISBN 978-0-521-37617-4.

[3] M. Abeles, H. Bergman, E. Margalit, and E. Vaadia. Spatiotemporal firing patterns in the frontal cortex of behaving monkeys. *Journal of Neurophysiology*, 70(4):1629–1638, October 1993. ISSN 0022-3077. doi: 10.1152/jn.1993.70.4.1629.

[4] Khadeer Ahmed, Amar Shrestha, Qinru Qiu, and Qing Wu. Probabilistic inference using stochastic spiking neural networks on a neurosynaptic processor. In *2016 International Joint Conference on Neural Networks, IJCNN 2016*, pages 4286–4293. Institute of Electrical and Electronics Engineers Inc., October 2016. doi: 10.1109/IJCNN.2016.7727759.

[5] Peter Auer, Harald Burgsteiner, and Wolfgang Maass. A learning rule for very simple universal approximators consisting of a single layer of perceptrons. *Neural Networks*, 21(5):786–795, June 2008. ISSN 0893-6080. doi: 10.1016/j.neunet.2007.12.036.

[6] Pierre Baldi and Peter Sadowski. A theory of local learning, the learning channel, and the optimality of backpropagation. *Neural Networks*, 83:51–74, November 2016. ISSN 0893-6080. doi: 10.1016/j.neunet.2016.07.006.

[7] Peter L. Bartlett and Jonathan Baxter. A Biologically Plausible and Locally Optimal Learning Algorithm for Spiking Neurons. 2000.

[8] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv:1806.01261 [cs, stat]*, June 2018.

[9] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994. ISSN 1045-9227. doi: 10.1109/72.279181.

[10] Yoshua Bengio. Deep Learning of Representations for Unsupervised and Transfer Learning. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, pages 17–36, June 2012.

[11] Yoshua Bengio, Thomas Mesnard, Asja Fischer, Saizheng Zhang, and Yuhuai Wu. STDP-Compatible Approximation of Backpropagation in an Energy-Based Model. *Neural Computation*, 29(3):555–577, March 2017. ISSN 0899-7667, 1530-888X. doi: 10.1162/NECO_a_00934.

[12] Joseph K. Blitzstein and Jessica Hwang. *Introduction to Probability*. Chapman and Hall/CRC, Boca Raton, 1 edition edition, July 2014. ISBN 978-1-4665-7557-8.

[13] Sander Bohte, Joost Kok, and Johannes Poutré. SpikeProp: Backpropagation for networks of spiking neurons. In *ESANN*, volume 48, pages 419–424, January 2000.

[14] Nicolas Brunel and Mark C. W. van Rossum. Lapicque's 1907 paper: From frogs to integrate-and-fire. *Biological Cybernetics*, 97(5):337–339, December 2007. ISSN 1432-0770. doi: 10.1007/s00422-007-0190-0.

[15] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent Reinforcement Learning: An Overview. In Janusz Kacprzyk, Dipti Srinivasan, and Lakhmi C. Jain, editors, *Innovations in Multi-Agent Systems and Applications - 1*, volume 310, pages 183–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-14434-9 978-3-642-14435-6. doi: 10.1007/978-3-642-14435-6_7.

[16] Guido Croon. Monocular distance estimation with optical flow maneuvers and efference copies: A stability-based strategy. *Bioinspiration & Biomimetics*, 11, January 2016. doi: 10.1088/1748-3190/11/1/016004.

[17] Gaël Daoudal and Dominique Debanne. Long-term plasticity of intrinsic excitability: Learning rules and mechanisms. *Learning & Memory (Cold Spring Harbor, N.Y.)*, 10(6):456–465, 2003 Nov-Dec. ISSN 1072-0502. doi: 10.1101/lm.64103.

[18] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. C. Liu, and M. Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *IEEE International Joint Conference on Neural Networks (IJCNN)*, Piscataway, USA, July 2015. Neural Networks (IJCNN), 2015 International Joint Conference on. doi: info:doi/10.5167/uzh-121702.

[19] Daniel Drubach. *The Brain Explained*. Prentice Hall, 2000.

[20] Răzvan V. Florian. Reinforcement Learning Through Modulation of Spike-Timing-Dependent Synaptic Plasticity. *Neural Computation*, 19(6):1468–1502, June 2007. ISSN 0899-7667, 1530-888X. doi: 10.1162/neco.2007.19.6.1468.

[21] Razvan V. Florian. The chronotron: A neuron that learns to fire temporally-precise spike patterns. *Nature Precedings*, 2010. doi: 10.1038/npre.2010.5190.1.

[22] Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Functional Requirements for Reward-Modulated Spike-Timing-Dependent Plasticity. *Journal of Neuroscience*, 30(40):13326–13337, October 2010. ISSN 0270-6474, 1529-2401. doi: 10.1523/JNEUROSCI.6249-09.2010.

[23] Daniel Gehrig, Antonio Loquercio, Konstantinos G. Derpanis, and Davide Scaramuzza. End-to-End Learning of Representations for Asynchronous Event-Based Data. *arXiv:1904.08245 [cs]*, April 2019.

[24] Daniel Gehrig, Antonio Loquercio, Konstantinos G. Derpanis, and Davide Scaramuzza. End-to-End Learning of Representations for Asynchronous Event-Based Data. *arXiv:1904.08245 [cs]*, April 2019.

[25] Wulfram Gerstner. Time structure of the activity in neural network models. *Physical Review E*, 51(1):738–758, January 1995. doi: 10.1103/PhysRevE.51.738.

[26] Wulfram Gerstner. *Neuronal Dynamics: From Single Neurons To Networks And Models Of Cognition*. Cambridge University Press, Cambridge, United Kingdom, uk ed. edition edition, September 2014. ISBN 978-1-107-63519-7.

[27] Wulfram Gerstner and Werner M. Kistler. *Spiking Neural Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[28] Wulfram Gerstner and J. Leo van Hemmen. Associative memory in a network of 'spiking' neurons. *Network: Computation in Neural Systems*, 3(2):139–164, January 1992. ISSN 0954-898X. doi: 10.1088/0954-898X_3_2_004.

[29] Matthieu Gilson and Tomoki Fukai. Stability versus Neuronal Specialization for STDP: Long-Tail Weight Distributions Solve the Dilemma. *PloS one*, 6:e25339, October 2011. doi: 10.1371/journal.pone.0025339.

[30] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, March 2010.

[31] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Neural Networks. page 9, 2011.

[32] Clément Godard, Oisin Mac Aodha, and Gabriel J. Brostow. Unsupervised Monocular Depth Estimation with Left-Right Consistency. *arXiv:1609.03677 [cs, stat]*, September 2016.

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

[34] Ariel Gordon, Hanhan Li, Rico Jonschkowski, and Anelia Angelova. Depth from Videos in the Wild: Unsupervised Monocular Depth Learning from Unknown Cameras. *arXiv:1904.04998 [cs]*, April 2019.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015.

[37] D.O. Hebb. *The Organization of Behaviour*. John Wiley & Sons, Inc., 1949.

[38] Professor David Heeger. Poisson Model of Spike Generation. page 13, September 2000.

[39] Geoffrey Hinton. How to do backpropagation in a brain. page 22, 2014.

[40] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural computation*, 9: 1735–80, December 1997. doi: 10.1162/neco.1997.9.8.1735.

[41] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, August 1952. ISSN 0022-3751.

[42] Ronald A. Howard. Dynamic Programming. *Manage. Sci.*, 12(5):317–348, January 1966. ISSN 0025-1909. doi: 10.1287/mnsc.12.5.317.

[43] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:1602.07360 [cs]*, February 2016.

[44] E.M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14 (6):1569–1572, November 2003. ISSN 1045-9227. doi: 10.1109/TNN.2003.820440.

[45] Eugene M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. Computational Neuroscience. MIT Press, Cambridge, Mass, 2007. ISBN 978-0-262-09043-8.

[46] Richard Kempter, Wulfram Gerstner, and J. Leo van Hemmen. Hebbian learning and spiking neurons. 1999. doi: 10.1103/PhysRevE.59.4498.

[47] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014.

[48] Vijay R. Konda and John N. Tsitsiklis. Actor-Critic Algorithms. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1008–1014. MIT Press, 2000.

[49] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[50] Akshay Chandra Lagandula. McCulloch-Pitts Neuron — Mankind's First Mathematical Model Of A Biological Neuron. https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1, 1943.

[51] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

[52] Yann LeCun, B Bosker, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, and Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1989.

[53] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. ISSN 1476-4687. doi: 10.1038/nature14539.

[54] Robert Legenstein, Dejan Pecevski, and Wolfgang Maass. A Learning Theory for Reward-Modulated Spike-Timing-Dependent Plasticity with Application to Biofeedback. *PLOS Computational Biology*, 4(10):e1000180, October 2008. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1000180.

[55] Li Fei-Fei, R. Fergus, and P. Perona. Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories. In *2004 Conference on Computer Vision and Pattern Recognition Workshop*, pages 178–178, June 2004. doi: 10.1109/CVPR.2004.383.

[56] P. Lichtsteiner, C. Posch, and T. Delbruck. A 128 X 128 120db 30mw asynchronous vision sensor that responds to relative intensity change. In *2006 IEEE International Solid State Circuits Conference - Digest of Technical Papers*, pages 2060–2069, February 2006. doi: 10.1109/ISSCC.2006.1696265.

[57] Timothy P Lillicrap and Adam Santoro. Backpropagation through time and the brain. *Current Opinion in Neurobiology*, 55:82–89, April 2019. ISSN 0959-4388. doi: 10.1016/j.conb.2019.01.011.

[58] 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. ISSN 2041-1723. doi: 10.1038/ncomms13276.

[59] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, December 1997. ISSN 0893-6080. doi: 10.1016/S0893-6080(97)00011-7.

[60] Ana I. Maqueda, Antonio Loquercio, Guillermo Gallego, Narciso Garcia, and Davide Scaramuzza. Event-based Vision meets Deep Learning on Steering Prediction for Self-driving Cars. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5419–5427, June 2018. doi: 10.1109/CVPR.2018.00568.

[61] A. A. Markov. *Theory of Algorithms*. Academy of Sciences of the USSR, 1954.

[62] Warren S. McCulloch and Walter H. Pitts. A Logical Calculus of the Ideas Imanent in Nervous Activity. *Mathetmatical Bulletin of Biophysics*, 5, 1943.

[63] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[64] Marvin Minsky. Steps toward Artificial Intelligence. *Proceedings of the IRE*, 49(1):8–30, January 1961. ISSN 0096-8390. doi: 10.1109/JRPROC.1961.287775.

[65] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry, Expanded Edition*. The MIT Press, Cambridge, Mass, expanded, subsequent edition edition, December 1987. ISBN 978-0-262-63111-2.

[66] Diederik Paul Moeys, Federico Corradi, Emmett Kerr, Philip Vance, Gautham Das, Daniel Neil, Dermot Kerr, and Tobi Delbruck. Steering a Predator Robot using a Mixed Frame/Event-Driven Convolutional Neural Network. *arXiv:1606.09433 [cs]*, June 2016.

[67] M. Mozafari, S. R. Kheradpisheh, T. Masquelier, A. Nowzari-Dalini, and M. Ganjtabesh. First-Spike-Based Visual Categorization Using Reward-Modulated STDP. *IEEE Transactions on Neural Networks and Learning Systems*, 29(12):6178–6190, December 2018. ISSN 2162-237X. doi: 10.1109/TNNLS.2018.2826721.

[68] Milad Mozafari, Mohammad Ganjtabesh, Abbas Nowzari-Dalini, Simon J. Thorpe, and Timothée Masquelier. Bio-Inspired Digit Recognition Using Spike-Timing-Dependent Plasticity (STDP) and Reward-Modulated STDP in Deep Convolutional Networks. *arXiv:1804.00227 [cs, q-bio]*, March 2018.

[69] Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. page 8, 2010.

[70] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences. *arXiv:1610.09513 [cs]*, October 2016.

[71] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences. *arXiv:1610.09513 [cs]*, October 2016.

[72] Bernhard Nessler, Michael Pfeiffer, Lars Buesing, and Wolfgang Maass. Bayesian Computation Emerges in Generic Cortical Microcircuits through Spike-Timing-Dependent Plasticity. In *PLoS Computational Biology*, 2013. doi: 10.1371/journal.pcbi.1003037.

[73] Yael Niv. The Neuroscience of Reinforcement Learning, 2009.

[74] Peter O'Connor, Daniel Neil, Shih-Chii Liu, Tobi Delbruck, and Michael Pfeiffer. Real-time classification and sensor fusion with a spiking deep belief network. *Frontiers in Neuroscience*, 7:178, 2013. ISSN 1662-4548. doi: 10.3389/fnins.2013.00178.

[75] Christopher Olah. Understanding LSTM Networks – colah's blog. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[76] Garrick Orchard, Ajinkya Jayawant, Gregory K. Cohen, and Nitish Thakor. Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades. *Frontiers in Neuroscience*, 9, 2015. ISSN 1662-453X. doi: 10.3389/fnins.2015.00437.

[77] Federico Paredes Valles. Neuromorphic Computing of Event-Based Data for High-Speed Vision-Based Navigation. 2018.

[78] Federico Paredes-Vallés, Kirk Y. W. Scheper, and Guido C. H. E. de Croon. Unsupervised Learning of a Hierarchical Spiking Neural Network for Optical Flow Estimation: From Events to Global Motion Perception. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2019. ISSN 0162-8828, 2160-9292, 1939-3539. doi: 10.1109/TPAMI.2019.2903179.

[79] I. P. Pavlov. *Conditioned Reflexes: An Investigation of the Physiological Activity of the Cerebral Cortex*. Conditioned Reflexes: An Investigation of the Physiological Activity of the Cerebral Cortex. Oxford Univ. Press, Oxford, England, 1927.

[80] Verena Pawlak, Jeffery R. Wickens, Alfredo Kirkwood, and Jason N. D. Kerr. Timing is not Everything: Neuromodulation Opens the STDP Gate. *Frontiers in Synaptic Neuroscience*, 2, 2010. ISSN 1663-3563. doi: 10.3389/fnsyn.2010.00146.

[81] Filip Ponulak. ReSuMe - New Supervised Learning Method for Spiking Neural Networks. page 10, 2005.

[82] Christoph Posch, Daniel Matolin, and Rainer Wohlgenannt. A QVGA 143 dB Dynamic Range Frame-Free PWM Image Sensor With Lossless Pixel-Level Video Compression and Time-Domain CDS. *IEEE Journal of Solid-State Circuits*, 46:259–275, 2011. doi: 10.1109/JSSC.2010.2085952.

[83] John N. J Reynolds and Jeffery R Wickens. Dopamine-dependent plasticity of corticostriatal synapses. *Neural Networks*, 15(4):507–521, June 2002. ISSN 0893-6080. doi: 10.1016/S0893-6080(02)00045-X.

[84] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. ISSN 1939-1471(Electronic),0033-295X(Print). doi: 10.1037/h0042519.

[85] J. Rosenburg. Von Neumann Architecture - an overview. 2017.

[86] D. E. Rumelhart and J. L. McClelland. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. MITP, 1987. ISBN 978-0-262-29140-8.

[87] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, October 1986. ISSN 1476-4687. doi: 10.1038/323533a0.

[88] Wolfram Schultz. Predictive Reward Signal of Dopamine Neurons. *Journal of Neurophysiology*, 80(1):1–27, July 1998. ISSN 0022-3077. doi: 10.1152/jn.1998.80.1.1.

[89] Wolfram Schultz. Dopamine signals for reward value and risk: Basic and recent data. *Behavioral and Brain Functions*, 6(1):24, April 2010. ISSN 1744-9081. doi: 10.1186/1744-9081-6-24.

[90] Wolfram Schultz, Peter Dayan, and P Read Montague. A Neural Substrate of Prediction and Reward. page 8, 1997.

[91] Terrence J Sejnowski and Gerald Tesauro. The Hebb Rule for Synaptic Plasticity: Algorithms and. page 10, 1989.

[92] Yusuke Sekikawa, Kosuke Hara, and Hideo Saito. EventNet: Asynchronous Recursive Event Processing. *arXiv:1812.07045 [cs]*, December 2018.

[93] A. Shrestha, K. Ahmed, Y. Wang, and Q. Qiu. Stable spike-timing dependent plasticity rule for multilayer unsupervised and supervised learning. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1999–2006, May 2017. doi: 10.1109/IJCNN.2017.7966096.

[94] Sumit Bam Shrestha and Garrick Orchard. SLAYER: Spike Layer Error Reassignment in Time. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1412–1421. Curran Associates, Inc., 2018.

[95] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]*, September 2014.

[96] J Sjostrom and W Gerstner. Spike-timing dependent plasticity. *J. Sj*, page 18, 2010.

[97] Sen Song, Kenneth D. Miller, and L. F. Abbott. Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, 3(9):919, September 2000. ISSN 1546-1726. doi: 10.1038/78829.

[98] Richard B. Stein. A Theoretical Analysis of Neuronal Variability. *Biophysical Journal*, 5(2):173–194, March 1965. ISSN 0006-3495. doi: 10.1016/S0006-3495(65)86709-1.

[99] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, second edition, 2018. ISBN 978-0-262-03924-6.

[100] Richard S Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, 2000.

[101] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *arXiv:1703.09039 [cs]*, March 2017.

[102] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper With Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[103] Amirhossein Tavanaei and Anthony S. Maida. BP-STDP: Approximating Backpropagation using Spike Timing Dependent Plasticity. *arXiv:1711.04214 [cs]*, November 2017.

[104] Johannes Christian Thiele, Olivier Bichler, and Antoine Dupret. SpikeGrad: An ANN-equivalent Computation Model for Implementing Backpropagation with Spikes. *arXiv:1906.00851 [cs]*, June 2019.

[105] M. C. W. van Rossum, G. Q. Bi, and G. G. Turrigiano. Stable Hebbian Learning from Spike Timing-Dependent Plasticity. *Journal of Neuroscience*, 20(23):8812–8821, December 2000. ISSN 0270-6474, 1529-2401. doi: 10.1523/JNEUROSCI.20-23-08812.2000.

[106] Xiaohui Xie and H. Sebastian Seung. Learning in neural networks by reinforcement of irregular spiking. *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics*, 69(4 Pt 1):041909, April 2004. ISSN 1539-3755. doi: 10.1103/PhysRevE.69.041909.

[107] Kaisheng Yao, Trevor Cohn, Katerina Vylomova, Kevin Duh, and Chris Dyer. Depth-Gated LSTM. *arXiv:1508.03790 [cs]*, August 2015.

[108] Friedemann Zenke and Surya Ganguli. SuperSpike: Supervised learning in multi-layer spiking neural networks. *Neural Computation*, 30(6):1514–1541, June 2018. ISSN 0899-7667, 1530-888X. doi: 10.1162/neco_a_01086.

[109] Li I. Zhang, Huizhong W. Tao, Christine E. Holt, William A. Harris, and Mu-ming Poo. A critical window for cooperation and competition among developing retinotectal synapses. *Nature*, 395 (6697):37, September 1998. ISSN 1476-4687. doi: 10.1038/25665.

[110] Wei Zhang and David J. Linden. The other side of the engram: Experience-driven changes in neuronal intrinsic excitability. *Nature Reviews Neuroscience*, 4(11):885, November 2003. ISSN 1471-0048. doi: 10.1038/nrn1248.

[111] Alex Zihao Zhu, Liangzhe Yuan, Kenneth Chaney, and Kostas Daniilidis. Unsupervised Event-based Learning of Optical Flow, Depth, and Egomotion. *arXiv:1812.08156 [cs]*, December 2018.

A

Results for SLAYER Experiments

This appendix contains the results for the networks as described in section 5.4, table 5.2, and table 5.3 for which the results were not treated in section 6.3. Because of the large number of figures, the references for each figure are presented in table A.2. The accuracies during training and testing for each network are found in table A.1

Table A.1: Training and testing results for the two and five layer fully connected networks trained on event-based data with the SLAYER algorith, and trained on frame-based data with regular backpropagation.

	Event-based		Frame-based	
	Training Acc.	Testing Acc.	Training Acc.	Testing Acc.
2 Layer Fully Connected	100%	93%	100%	89%
4 Layer Fully Connected	100%	91%	100%	91%
5 Layer Fully Connected	100%	85%	100%	86%
3 Layer Convolutional	100%	65%	100%	91%
5 Layer Convolutional	100%	50%	100%	98%
6 Layer Convolutional	100%	56%	100%	96%

Table A.2: Each network and the reference to the figures containing the statistics (mean and standard deviation) of its gradients during training.

	Event-based		Frame-based	
	Mean	Standard Deviation	Mean	Standard Deviation
2 Layer Fully Connected	fig. A.1	fig. A.2	fig. A.3	fig. A.4
4 Layer Fully Connected	fig. A.5	fig. A.6	fig. A.7	fig. A.8
5 Layer Fully Connected	fig. A.9	fig. A.10	fig. A.11	fig. A.8
3 Layer Convolutional	fig. A.13	fig. A.14	fig. A.15	fig. A.16
5 Layer Convolutional	fig. A.17	fig. A.18	fig. A.19	fig. A.20
6 Layer Convolutional	fig. A.21	fig. A.22	fig. A.23	fig. A.24

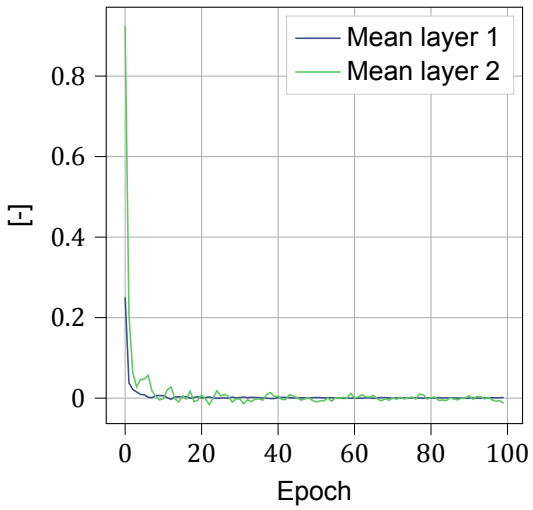


Figure A.1: Mean of gradients for 2 layer fully connected SNN during training.

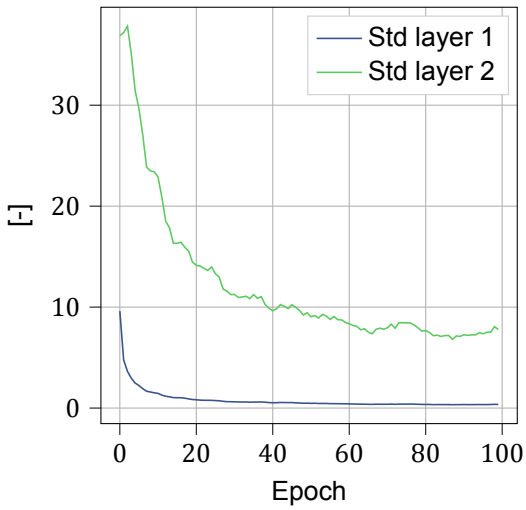


Figure A.2: Standard deviation of gradients for 2 layer fully connected SNN during training.

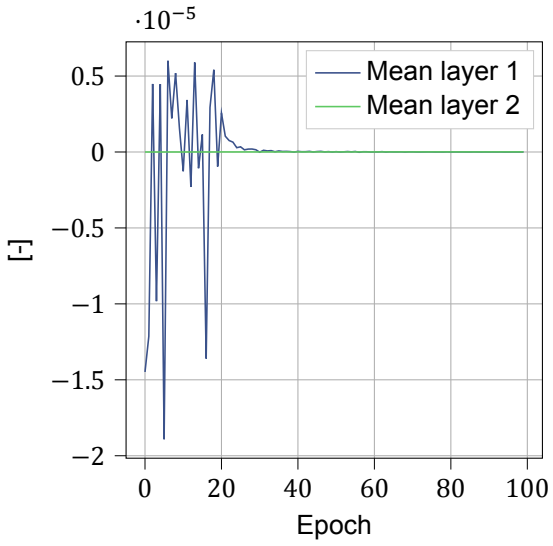


Figure A.3: Mean of gradients for 2 layer fully connected ANN during training.

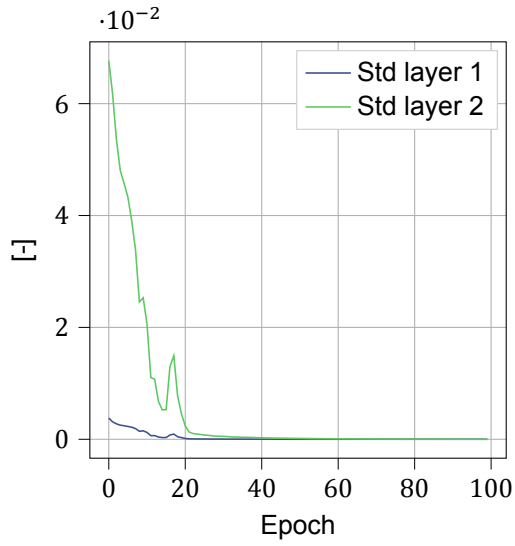


Figure A.4: Standard deviation of gradients for 2 layer fully connected ANN during training.

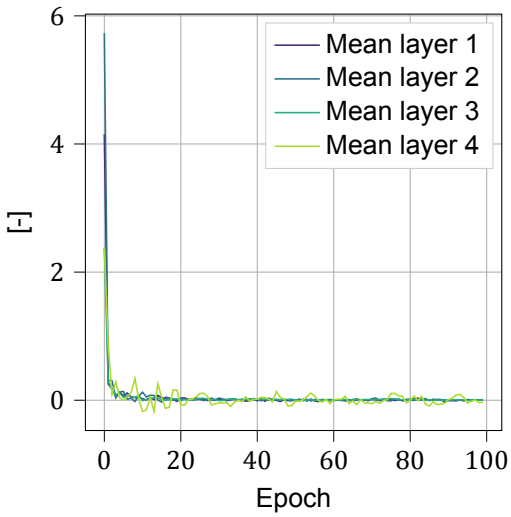


Figure A.5: Mean of gradients for four layer fully connected SNN during training.

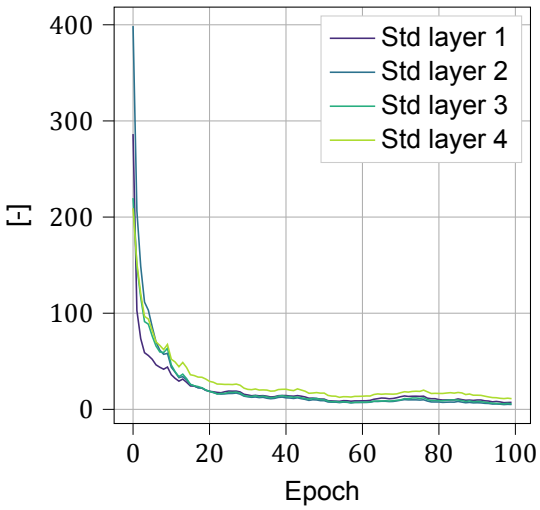


Figure A.6: Standard deviation of gradients for four layer fully connected SNN during training.

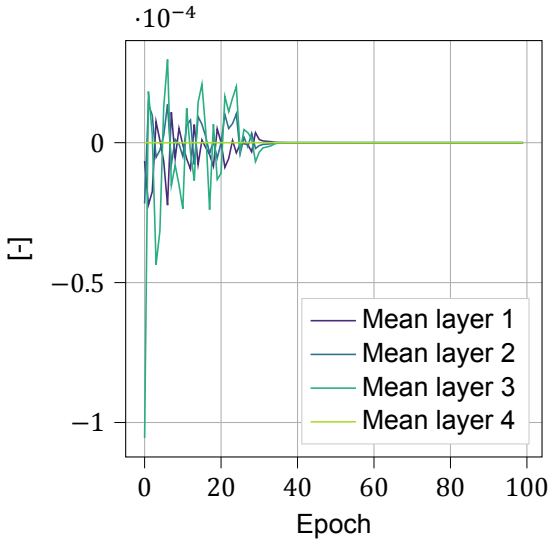


Figure A.7: Mean of gradients for four layer fully connected ANN during training.

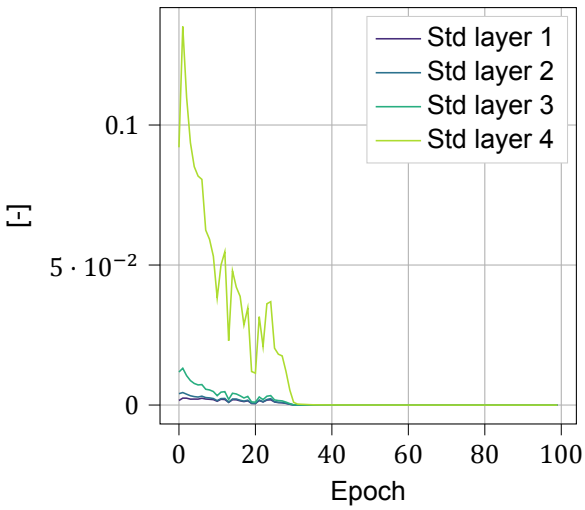


Figure A.8: Standard deviation of gradients for four layer fully connected ANN during training.

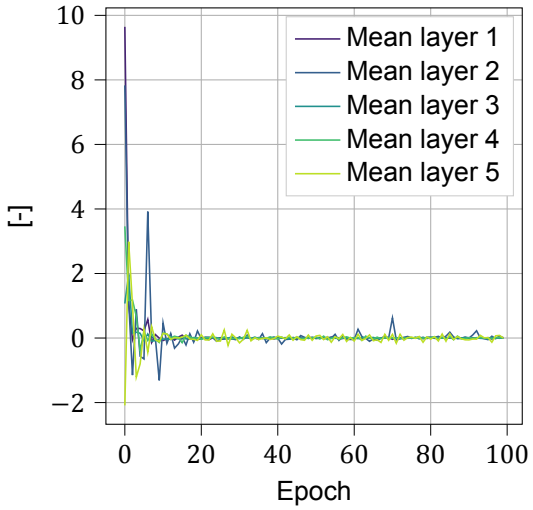


Figure A.9: Mean of gradients for five layer fully connected SNN during training.

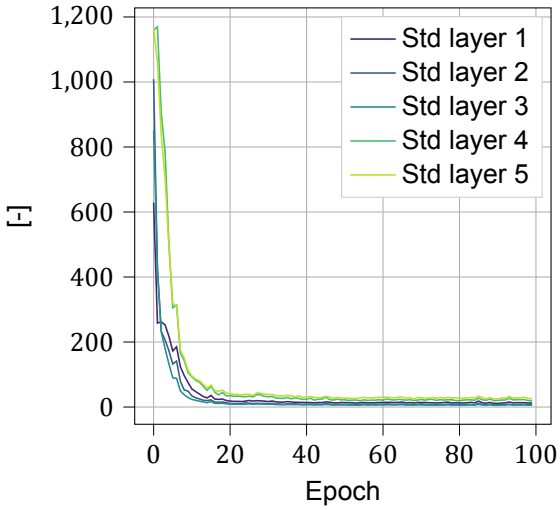


Figure A.10: Standard deviation of gradients for five layer fully connected SNN during training.

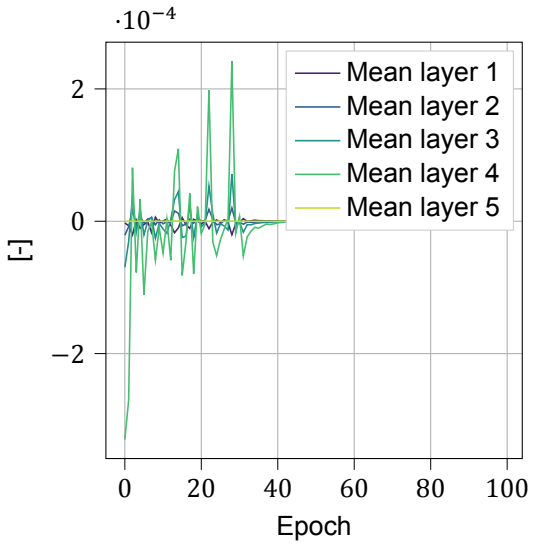


Figure A.11: Mean of gradients for five layer fully connected ANN during training.

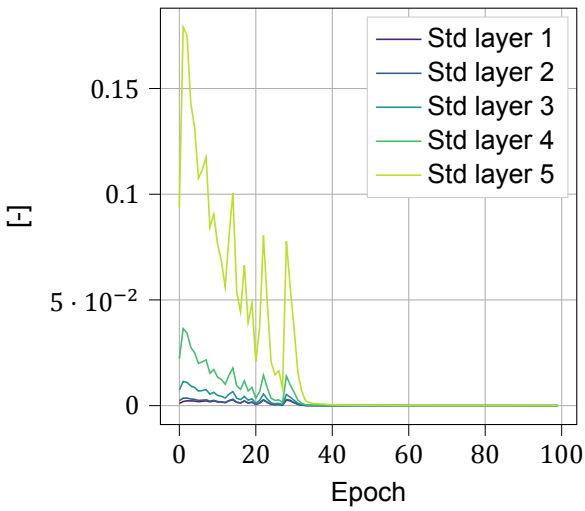


Figure A.12: Standard deviation of gradients for five layer fully connected ANN during training.

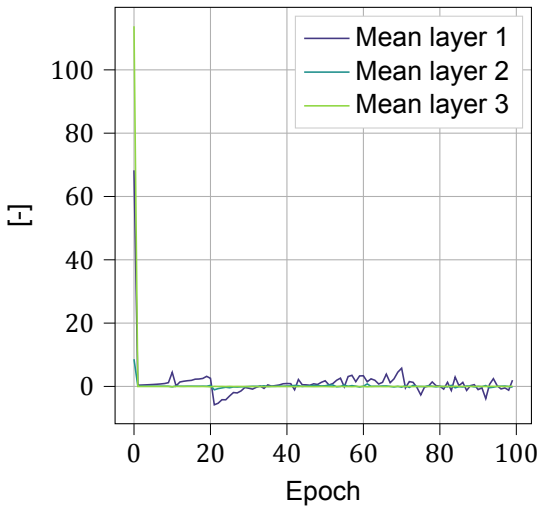


Figure A.13: Mean of gradients for three layer convolutional SNN during training.

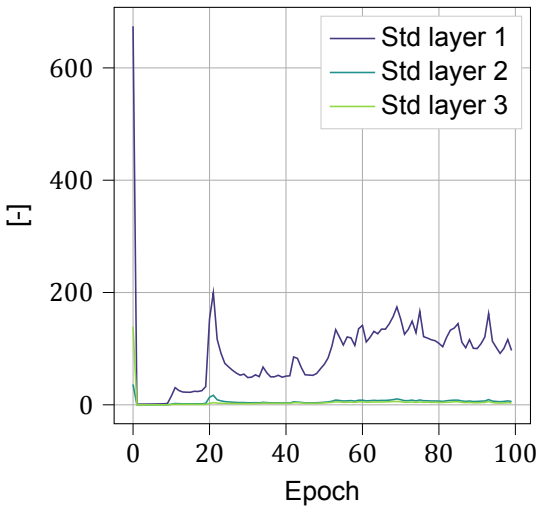


Figure A.14: Standard deviation of gradients for three layer convolutional SNN during training.

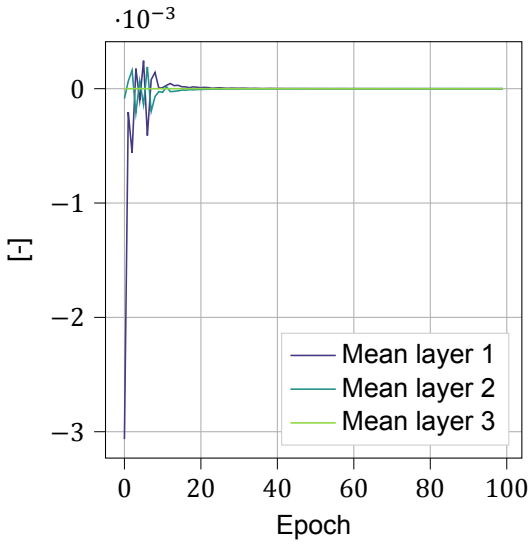


Figure A.15: Mean of gradients for three layer convolutional ANN during training.

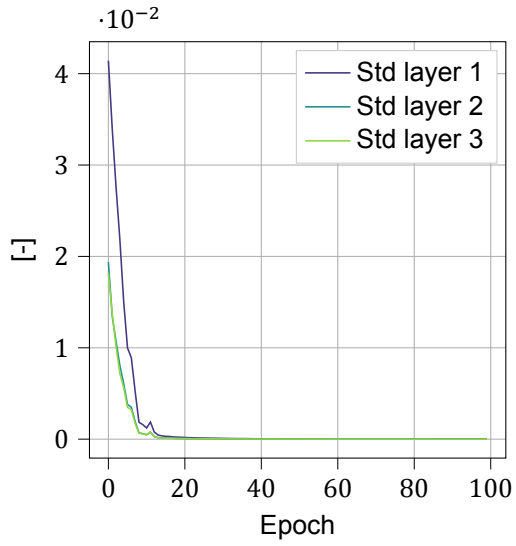


Figure A.16: Standard deviation of gradients for three layer convolutional ANN during training.

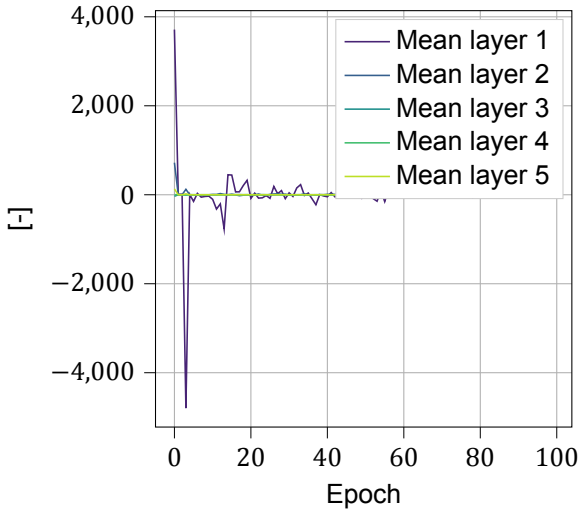


Figure A.17: Mean of gradients for five layer convolutional SNN during training.

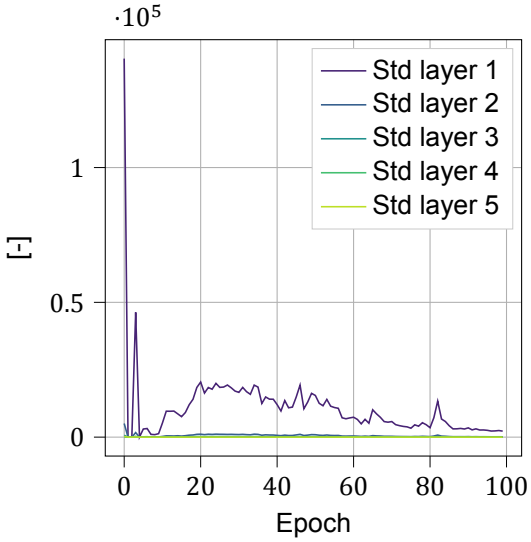


Figure A.18: Standard deviation of gradients for five layer convolutional SNN during training.

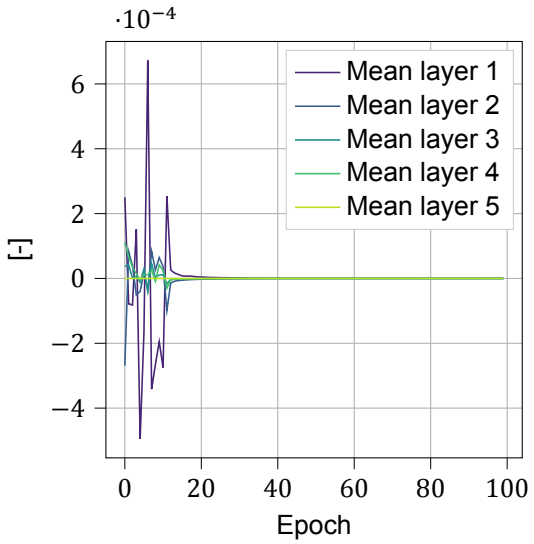


Figure A.19: Mean of gradients for five layer convolutional ANN during training.

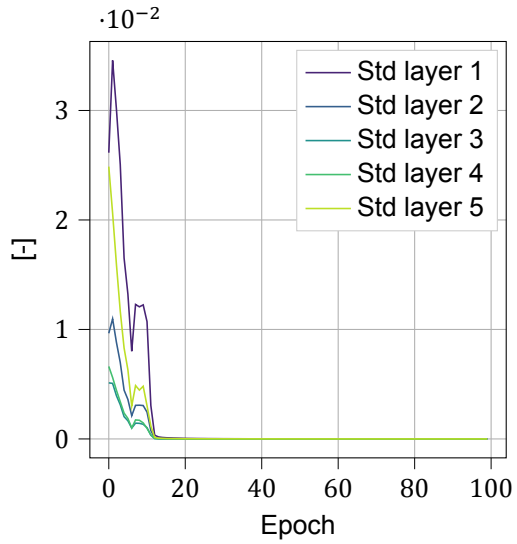


Figure A.20: Standard deviation of gradients for five layer convolutional ANN during training.

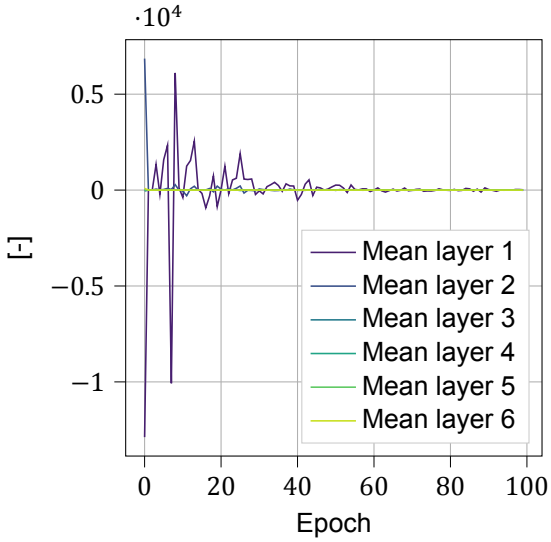


Figure A.21: Mean of gradients for six layer convolutional SNN during training.

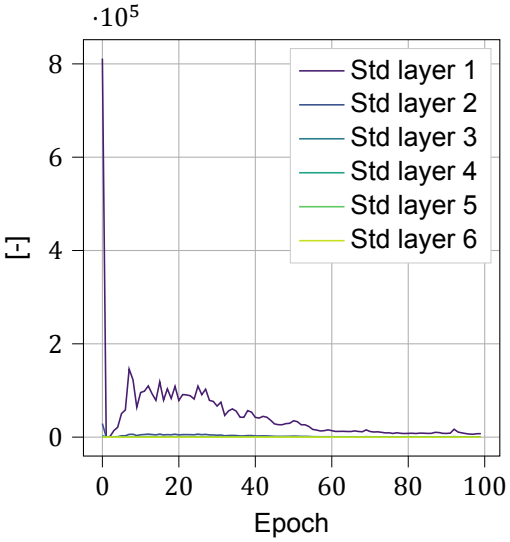


Figure A.22: Standard deviation of gradients for six layer convolutional SNN during training.

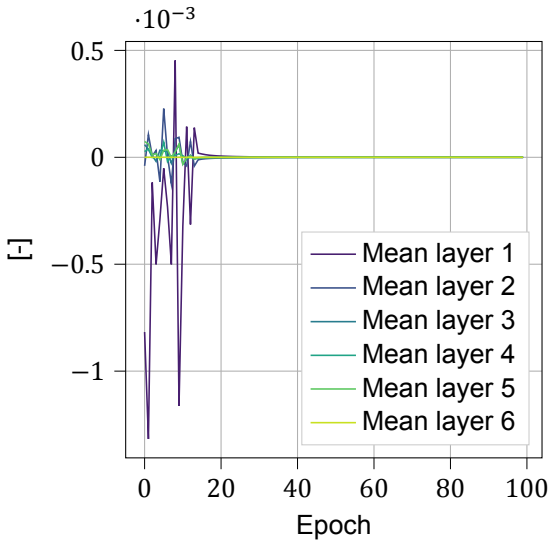


Figure A.23: Mean of gradients for six layer convolutional ANN during training.

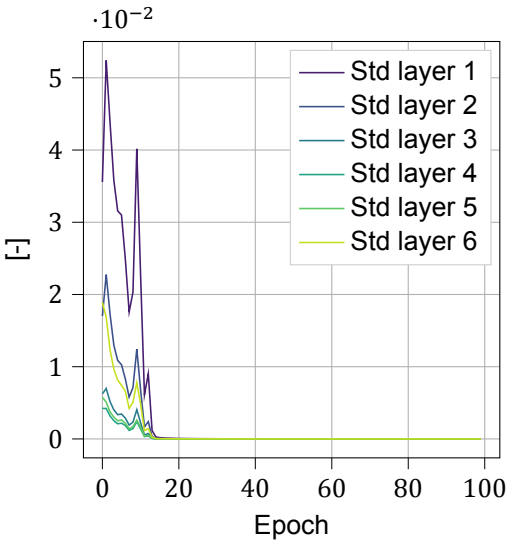


Figure A.24: Standard deviation of gradients for six layer convolutional ANN during training.