Elias Hajek, BSc

Training recurrent spiking neural networks with biologically inspired learning rules

Master's Thesis
to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor
Em.Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Wolfgang Maass

Institute for Theoretical Computer Science, Graz University of Technology,
Austria
Head: Assoc. Prof. Dipl.-Ing. Dr. techn. Robert Legenstein

Graz, August 2019

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

.....	.....
date	(signature)

Abstract

Neuromorphic hardware is a promising approach to overcome the von Neumann bottleneck and to run a range of machine learning algorithms on low power hardware, that can solve problems in fields like robotics, speech processing or image recognition. There is however still a lack of powerful learning algorithms, that allow to close the performance gap to machine learning on conventional hardware such as GPUs and which scale well with increasing network sizes.

In this thesis, a novel learning algorithm, "e-prop", is tested, that allows to efficiently train a network of recurrently connected spiking neural networks in an online-fashion. With e-prop, it is not required to first simulating a whole sequence, storing all past network states and inputs and then going "backwards in time" through the unfolded network to compute the weight updates like it would be necessary with state of the art algorithms like backpropagation. Since e-prop does not have this storage requirement, and also no need for additional offline processing, it could be implemented on neuromorphic hardware.

It is demonstrated, that this algorithm can solve a range of tasks with a performance that approaches backpropagation through time. Also a modification of this algorithm is discussed, which mitigates the weight transport problem, that is, the issue of the biological plausibility of a symmetric feedback path, that would be necessary for implementing backpropagation.

Kurzfassung

Neuromorphe Hardware ist ein vielversprechender Ansatz um den Von-Neumann-Flaschenhals zu überwinden und eine Reihe von maschinellen Lernalgorithmen zur Lösung von Problemen in Bereichen wie Robotik, Sprachverarbeitung und Bilderkennung auf Hardware mit geringem Stromverbrauch auszuführen. Es mangelt zurzeit jedoch noch an leistungsstarken Lernalgorithmen, welche den Leistungsunterschied zu maschinellen Lernalgorithmen auf traditioneller Hardware, wie etwa Grafikkarten, beseitigen können und welche auch mit steigender Netzwerkgröße gut skalieren.

In dieser Arbeit wird ein neuer Lernalgorithmus "e-prop" getestet, welcher es erlaubt, ein rekurrentes Netzwerk von gepulsten Neuronen online zu trainieren, also ohne zuerst die komplette Sequenz zu simulieren, den gesamten vergangenen Zustand sowie die Eingaben zu speichern und danach die Gewichtsänderungen mittels "Zurückgehen in der Zeit" durch das aufgefaltete Netzwerk zu berechnen, wie es mit dem aktuellen Stand der Technik, dem Fehlerrückführungs-Algorithmus, notwendig wäre. Da e-prop nicht diese Anforderung an den Speicher hat und auch ohne zusätzliche Offline-Berechnungen auskommt, könnte es auf neuromorpher Hardware implementiert werden.

Es wird gezeigt, dass dieser Algorithmus eine Reihe von Aufgaben mit einer Performance lösen kann, welche der Fehlerrückführung durch die Zeit sehr nahe kommt. Es wird auch eine Abänderung des Algorithmus diskutiert, welche das Gewichtstransportsproblem mildert, die Frage nach der biologischen Plausibilität eines symmetrischen Rückkopplungspfades, welcher für die Fehlerrückführung nötig wäre.

Acknowledgements

Prof. Wolfgang Maass for the opportunity of taking me on as his student and for his guidance and suggestions, without which this thesis would not have been possible
Prof. Robert Legenstein for his advice and support
Guillaume Bellec, Franz Scherr and Darjan Salaj for the countless things they were teaching me during the course of this thesis, their patience and of course all their previous work on which this thesis is based on.
All the great people at IGI, for the interesting discussions, being fun to be with at the workshops and also providing valuable advice.
My family, my girlfriend and my friends for supporting me and providing encouragement and stability during all stressful times.

Contents

1 Introduction and preliminaries **7**
- 1.1 Motivation . . . 7
- 1.2 Biological neurons . . . 7
- 1.3 Artificial neuron models . . . 8
- 1.4 Artificial recurrent neural networks . . . 9
  - 1.4.1 Definition . . . 9
- 1.5 Training recurrent neural networks . . . 10
  - 1.5.1 Gradient based methods . . . 10
  - 1.5.2 Reinforcement learning . . . 13
- 1.6 Spiking neuron models . . . 13
  - 1.6.1 Introduction . . . 13
  - 1.6.2 Network and neuron equations . . . 14
  - 1.6.3 Output computation . . . 16
- 1.7 Learning with spiking neurons . . . 17
  - 1.7.1 Spike timing dependent plasticity . . . 17
  - 1.7.2 Surrogate gradient method . . . 18

2 Problem statement and goal **19**
- 2.1 Problem statement . . . 19
  - 2.1.1 Locking problem . . . 19
  - 2.1.2 Weight transport problem . . . 19
- 2.2 Goal . . . 20

3 Related work **21**
- 3.1 Feedback alignment . . . 21
- 3.2 Broadcast alignment . . . 21
- 3.3 FORCE learning . . . 21
- 3.4 Other online learning algorithms . . . 22
  - 3.4.1 Real time recurrent learning (RTRL) . . . 22
  - 3.4.2 Unbiased Online Recurrent Optimization (UORO) . . . 22
  - 3.4.3 Kernel RNN Learning (KeRNL) . . . 23
  - 3.4.4 Random feedback local online learning (RFLO) . . . 23

4 Training algorithms **24**
- 4.1 Eligibility propagation (e-prop) . . . 24
  - 4.1.1 Introduction . . . 24
  - 4.1.2 Derivation . . . 25
  - 4.1.3 E-prop for LIF and ALIF neurons . . . 25

4.1.4 Computation of update rules 28
4.1.5 E-prop and the weight transport problem 30

5 Experiments **31**
5.1 Pattern generation task 31
5.1.1 Introduction 31
5.1.2 Task details 31
5.1.3 Details of network model 32
5.1.4 Details of training procedure 32
5.1.5 Results 32
5.2 Movie replay task 33
5.2.1 Introduction 33
5.2.2 Details of the movie replay task: 34
5.2.3 Details of the network model and of the input scheme: . . 34
5.2.4 Details of the learning procedure 35
5.2.5 Results 35
5.3 Evidence accumulation task 36
5.3.1 Introduction 36
5.3.2 Details of the network model and of the input scheme: . . 37
5.3.3 Details of the learning procedure: 38
5.3.4 Results 39

6 Analysis of random e-prop **41**
6.1 Introduction 41
6.1.1 Positive definiteness of a matrix 41
6.1.2 Trace 41
6.1.3 Alignment of angles 42
6.2 Results 43
6.2.1 Pattern generation 43
6.2.2 Store recall 43
6.2.3 Evidence accumulation task 46
6.2.4 Discussion 47

7 Conclusion **48**
7.1 Discussion 48
7.2 Further research 48

Chapter 1

Introduction and preliminaries

1.1 Motivation

Recurrent spiking neural networks form the basis of all information processing in human and animal brains. It is however still not quite understood, how exactly these networks learn to execute complex tasks such as motor control, memory, prediction, navigation or planning. Solving these tasks involves a challenging temporal credit assignment problem, since there are many time scales involved, and the firing of any neuron could have an influence on an outcome far in the future.

A better understanding of learning in recurrent spiking neural networks, could therefore allow us to implement very powerful algorithms on novel neuromorphic hardware such as Loihi (Davies et al. (2018)), Spinnaker (Furber et al. (2014)) or BrainScaleS (Schemmel et al. (2010)).

1.2 Biological neurons

The information of this section can be found in Bear et al. (2016).

Neurons are the main building blocks of our nervous system. They are nerve cells, that can communicate with each other using discrete events, the so called action potentials (APs). These APs are produced via an electrochemical process, where different concentrations of ions inside and outside the cell produce an electric voltage.

There exist many different kinds of neurons that each have certain functionality. One type are the sensory neurons, that detect incoming stimuli such as touch, light, sound or taste and relay this information to other neurons. There are also neurons that serve as connection between other neurons, the so called inter neurons. A third group are the motor neurons, which can can execute motor movement by stimulating muscles to contract.

The prototypical neuron consists of the cell body, also called the soma, which is surrounded by the neuronal membrane which separates it from the outside of the cell. To transmit information to other neurons, a neuron has a so called axon, which can range in length from a millimeter to up to over a meter long. From the soma there also extends a tree-like branch of so called dendrites, which are connected to other axons and carry information into the neuron.

When a neuron emits an action potential, it travels over the neurons axon to the presynaptic axon terminal, where it releases chemicals, the so called neurotransmitters. These neurotransmitters then travel through the synaptic cleft to the postsynaptic dendrite, which is connected to the soma of another neuron. This causes the membrane voltage of the receiving neuron to be slightly depolarized. Once a sufficient amount of these events happen, the neuron emits an action potential itself, which then again travels over the axon to other neurons.

1.3 Artificial neuron models

A simple way to model this, is the artificial neuron model. In this model, a neuron with index j computes an activation value h_j, which is a weighted sum of outputs from other neurons. The activation value is then sent into a nonlinear function to obtain the output. In the perceptron model from Rosenblatt (1958) this nonlinear function is a simple step function:

$$h_j = \sum_{i \neq j} \theta_{ji} z_i \tag{1.1}$$

and

$$z_j = \begin{cases} 0 & h_j < 0 \\ 1 & h_j \geq 0 \end{cases} \tag{1.2}$$

One could interpret the weights of the input as synaptic strengths and the step function as the discrete spiking function of a biological neuron. In most of nowadays neural networks however, this step function is replaced by a continuous function like the logistic sigmoid:

$$\sigma(a) = \frac{1}{1 + exp(-a)} \tag{1.3}$$

This is done to avoid the discontinuity of the step function to be able to compute gradients and use gradient descent based methods to optimize the network parameters.

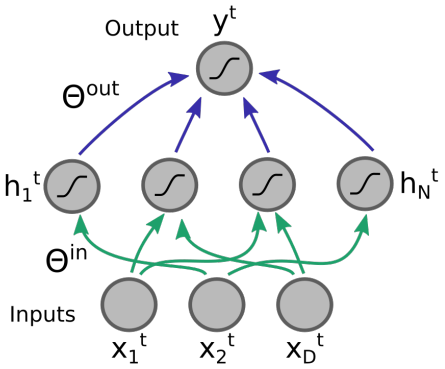

Figure 1.1: Graph of a one layer feedforward artificial neural network.

One then typically groups these neurons into layers, where each layer only receives input from the previous layer and sends its output to the next layer. The last layer is then used to compute the output of the network. Since there are no loops in the graph of these networks, they are also called feedforward neural networks.

1.4 Artificial recurrent neural networks

1.4.1 Definition

In contrast to regular feed forward artificial neurons, recurrent neural networks contain feedback connections and therefore form a loop in the network graph. This introduces a "time" dependency and enables the network to process temporal sequences (Goodfellow et al. (2016)).

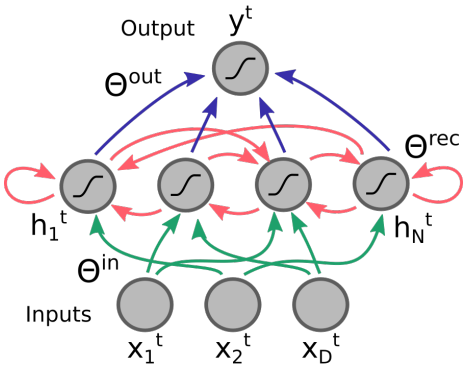

Figure 1.2: Sketch of a simple recurrent neural network.

A general recurrent neural network can be described with the following equations:

$$\boldsymbol{h}^t = g(\boldsymbol{x}^t, \boldsymbol{h}^{t-1}, \boldsymbol{z}^{t-1}) \tag{1.4}$$

$$\boldsymbol{z}^t = f(\boldsymbol{h}^t) \tag{1.5}$$

$$\boldsymbol{y}^t = \sigma(\boldsymbol{z}^t) \tag{1.6}$$

At every time step t, for each neuron a new hidden state value is calculated by some function of the hidden states from the previous time step $\boldsymbol{h}^{t-1}$, the network input $\boldsymbol{x}^t$ and the previous observable state of the network $\boldsymbol{z}^{t-1}$. The observable state $\boldsymbol{z}^t$ is then computed by some function on the hidden state. It is called hidden, because it is not accessible to other neurons.

Depending on the problem at hand, the observable sate is then sent into an output function $\sigma(\boldsymbol{z}^t)$. This could be a softmax function to obtain class probabilities in a classification setup, or the identity function for regression problems.

This network describes a dynamical system which is driven by an external input $\boldsymbol{x}^t$, updates its internal states $\boldsymbol{h}^t$ and emits some observable state (output) $\boldsymbol{z}^t$. The function $g()$ is usually parameterized with some parameter vector $\boldsymbol{\theta}$.

It is important to note, that the parameter vector $\boldsymbol{\theta}$ does not depend on time and is therefore shared across all time steps. Learning these parameters, allows the model to generalize across different sequence lengths. Similar to a convolutional neural network that has an inherent spatial translation invariance, a recurrent neural network is invariant with regard to the position in time.

1.5 Training recurrent neural networks

In a supervised learning setup, one typically wishes to minimize the discrepancy between a network's output and a predefined target. This is formalized with a function of the parameters that represents this error which is often called a loss function.

Common loss functions are the mean squared error for regression problems:

$$E(\boldsymbol{\theta}) = ||\boldsymbol{y}(\theta) - \boldsymbol{y}_{target}||_2^2 \tag{1.7}$$

or the cross entropy error in a binary sequence classification setup:

$$E(\boldsymbol{\theta}) = -\sum_{t=1}^{T} y_{target}^t \log(y^t(\boldsymbol{\theta})) + (1 - y_{target}^t) \log(1 - y^t(\boldsymbol{\theta})) \tag{1.8}$$

The loss function $E(\boldsymbol{\theta})$ is in general not convex and therefore it is hard to find a global minimum. Also it is usually not possible to find a closed form solution, so the most common way of optimizing neural networks is with gradient based methods, which under certain assumptions are guaranteed to find a local minimum.

1.5.1 Gradient based methods

Introduction

Gradient based methods are optimization methods that utilize the gradient of the cost function in some way in order to reach a local minimum. They use

the fact, that the gradient of a function points in the direction where a function value is increasing the strongest, while the negative gradient points in the direction of the strongest function value decrease. Since many cost functions don't allow a closed form solution in the parameters, this will be the only way to obtain a solution.

In general, optimization theory tells us that a sufficiently small step along a descent direction, (a direction which forms an angle of less than 90 degrees with the negative gradient) will give a decrease in the function value. More formally:

Theorem 1 *if* $\nabla f^T \boldsymbol{d} < 0$ *there exists an interval* $(0, \delta]$ *such that* $f(\boldsymbol{x} + \alpha \boldsymbol{d}) < f(\boldsymbol{x})$ *for all* $\alpha \in (0, \delta]$.

This can be utilized by an iterative algorithm, that makes small enough steps along a descent direction until a local minimum is reached. The most common form is to choose the negative gradient as a descent direction $\boldsymbol{d} = -\nabla f$ which is also known as steepest descent. The size of these steps is controlled by a so called learning rate, which is often decreased over time to ensure convergence.

Gradient for a recurrent neural network

To optimize the loss function, one needs to compute the total derivative of the loss with respect to the network parameters $\boldsymbol{\theta}$, i.e. $\frac{dE}{d\boldsymbol{\theta}}$. It should be noted, that it is relevant here to make a distinction between the partial and the total derivative, as will be illustrated in the case of a simple example.

Suppose we have a function $f(z)$ which is of the following form:

$$f(z) = x + z \tag{1.9}$$

where $x = g(z)$ The partial derivative is simply computed as:

$$\frac{\partial f(z)}{\partial z} = 1 \tag{1.10}$$

The total derivative now takes into account all variables that depend on z and gives a measure that relates the change in z to the total change in $f(z)$:

$$\frac{df(z)}{dz} = \frac{\partial f(z)}{\partial z} + \frac{\partial f(z)}{\partial x}\frac{dx}{dz} \tag{1.11}$$

The loss at time t is usually a function of the current network state, which itself is a function of the network parameters. We can express this gradient via the chain rule:

$$\frac{dE^t}{d\boldsymbol{\theta}} = \frac{dE^t}{d\boldsymbol{z}^t}\frac{d\boldsymbol{z}^t}{d\boldsymbol{h}^t}\frac{d\boldsymbol{h}^t}{d\boldsymbol{\theta}} \tag{1.12}$$

The first two parts of this gradient are rather simple to compute, and depend in general on the type of output and loss function that is used. Harder to compute is the total derivative $\frac{d\boldsymbol{h}}{d\boldsymbol{\theta}}$, since $\boldsymbol{h}(t)$ is a function that depends on $\boldsymbol{h}(t-1)$ and $\boldsymbol{\theta}$ which itself depends on $\boldsymbol{h}(t-2)$ and so on.

We can write this total derivative as:

$$\frac{d\boldsymbol{h}^t}{d\boldsymbol{\theta}} = \frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{\theta}} + \frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{h}^{t-1}} \frac{d\boldsymbol{h}^{t-1}}{d\boldsymbol{\theta}} \tag{1.13}$$

The jacobian $\frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{h}^{t-1}} = J_h$ depends on the used network model and does not change with time, therefore we can write:

$$\prod_{t'}^{t-1} \frac{\partial \boldsymbol{h}^{\boldsymbol{t'+1}}}{\partial \boldsymbol{h}^{\boldsymbol{t'}}} = \prod_{t'}^{t-1} J_h = J_h^{t-t'} = \frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{h}^{t'}} \tag{1.14}$$

Now the recursive definition of the total derivative can be simplified as:

$$\frac{d\boldsymbol{h}^t}{d\boldsymbol{\theta}} = \sum_{t'=1}^{t} \frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{h}^{t'}} \frac{\partial \boldsymbol{h}^{t'}}{\partial \boldsymbol{\theta}} \tag{1.15}$$

Using the assumption that the overall error E, is the unweighted sum of the errors E^t we can now obtain the desired derivative:

$$\frac{dE}{d\boldsymbol{\theta}} = \sum_{t=1}^{T} \frac{\partial E^t}{\partial \boldsymbol{z}^t} \frac{\partial \boldsymbol{z}^t}{\partial \boldsymbol{h}^t} \sum_{t'=1}^{t} \frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{h}^{t'}} \frac{\partial \boldsymbol{h}^{t'}}{\partial \boldsymbol{\theta}} \tag{1.16}$$

Backpropagation through time

Rumelhart et al. (1986) introduced an effective method called backpropagation for computing the gradients of a neural network by recursively applying the chain rule. This algorithm is one of the main contributors that enabled the recent success of deep neural networks.

Werbos (1990) gives a detailed account of how to apply this very general algorithm to the training of recurrent neural networks with a method that is called backpropagation through time.

Contemporary machine learning algorithms use automatic differentiation software such as Tensorflow (Abadi et al. (2015)) to also very efficiently compute these gradients on an unrolled computational graph (see figure 1.3).

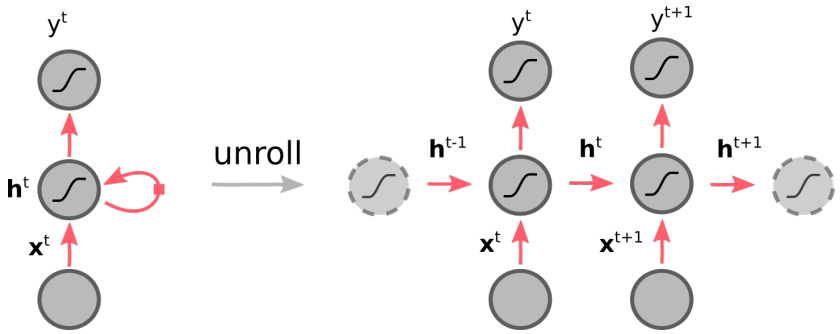

Figure 1.3: Unrolling of the computational graph for a recurrent neural network. Figure adapted from Goodfellow et al. (2016)

To compute this gradient, first the network is simulated forward in time, computing the outputs and errors for all time steps. At the next step, the gradients are computed by propagating these errors backwards in the unrolled network. This is a very efficient method, but it requires that the inputs and hidden states of the network are stored for the whole sequence.

1.5.2 Reinforcement learning

Introduction

Reinforcement learning is formalized in (Sutton and Barto (1998), chapter 5) in the context of finite Markov decision problems. The setup consists of an actor at time t who is in a certain state $S_t \in S$ and takes actions $A_t \in A(s)$, which will bring him to the next state S_{t+1} along with a reward $R_{t+1} \in R$ based on the taken actions. The goal is to find the action in each state, which will maximize the expected rewards.

Policy gradient methods

To chose the right actions, the actor will follow a certain policy, which in the context of this thesis, will be parameterized by a recurrent spiking neural network. The policy π is defined according to the following equation (Sutton and Barto (1998), chapter 13):

$$\pi(a|s, \boldsymbol{\theta}) = p(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}) \tag{1.17}$$

A simple policy gradient method will then try to maximize the probability that a good action was taken, by using the gradient of the policy w.r.t the parameters $\boldsymbol{\theta}$. The weight update will be of the form:

$$\Delta\boldsymbol{\theta}_t = -\eta(R^t - V^t)\nabla \log(\pi(A^t|S^t, \boldsymbol{\theta^t})) \tag{1.18}$$

Here V^t is some estimate of the value function which serves as a baseline to reduce the gradient variance, and in the simplest case could just be a constant reward baseline.

This learning rule will increase the probability of a certain action, if the received reward was greater than the chosen reward baseline and decrease it otherwise.

1.6 Spiking neuron models

1.6.1 Introduction

In this section, a very simple mathematical model for a biological neuron as described in section 1.2 is introduced. This model is chosen, because it captures essential features of biological models, while still allowing a mathematical analysis that does not get out of hand. The model described in the following sections is introduced in Bellec et al. (2018b).

1.6.2 Network and neuron equations

Two spiking neural models are used in this thesis: the leaky integrate and fire model (LIF) and the adaptive leaky integrate and fire neuron (ALIF). The LIF neuron is a dynamical system, which has a voltage $\boldsymbol{v}$, also called the membrane potential as its hidden state variable $\boldsymbol{h}$. A neuron with index j integrates incoming action potentials (spikes z_i) from other neurons, and emits a spike z_j itself, once a certain threshold is reached. These spikes make up the observable state. The membrane potential is also decaying exponentially (leaking) over time back to a defined baseline voltage, as controlled by the decay factor $\alpha \in (0, 1)$.

In all further equations, the time is assumed to be discretized, with the step from t to $t + 1 = 1$ ms. The equations in discretized time for an LIF neuron with index j are as follows (Bellec et al. (2019a)):

$$v_j^{t+1} = \alpha v_j^t + \sum_{i \neq j} \theta_{ji}^{rec} z_i^t + \sum_i \theta_{ji}^{in} x_i^{t+1} - z_j^t v_{th} \tag{1.19}$$

θ_{ji}^{rec} are the synaptic weights from network neuron i to network neuron j which are also called recurrent weights. θ_{ji}^{in} are the synaptic weights from input neuron i to network neuron j and are called input weights. The inputs are here assumed to be discrete spikes $x_i \in \{0, 1\}$.

The observable state is computed as:

$$z_j^t = \mathcal{H}(v_j^t - v_{th}) \tag{1.20}$$

Where $\mathcal{H}$ is the heaviside step function defined as:

$$\mathcal{H}(v) = \begin{cases} 0 & v < 0 \\ 1 & v \geq 0 \end{cases} \tag{1.21}$$

Here we again see a discontinuity in the spiking function. One could again replace this with a logistic sigmoid or any other continuous function, but this would come with a drawback: The all or nothing nature of a spike allows a very efficient computation of the network dynamics, since most of the time a neuron does not spike and therefore no computation is necessary. This temporal sparsity is one of the reasons why neuromorphic hardware can be made very energy efficient.

In this simple neuron model, the baseline voltage is assumed to be zero. Figure 1.4 shows the time course of the membrane voltage in the case where the weighted sum of network spikes is replaced by a constant input value. In the first 300 ms the neuron is integrating the input current and emitting a spike, every time the membrane voltage reaches the threshold and then resets back to the baseline voltage of zero. When the input current stops after 300 ms, the voltage decays exponentially back to zero with a time constant of 20 ms.

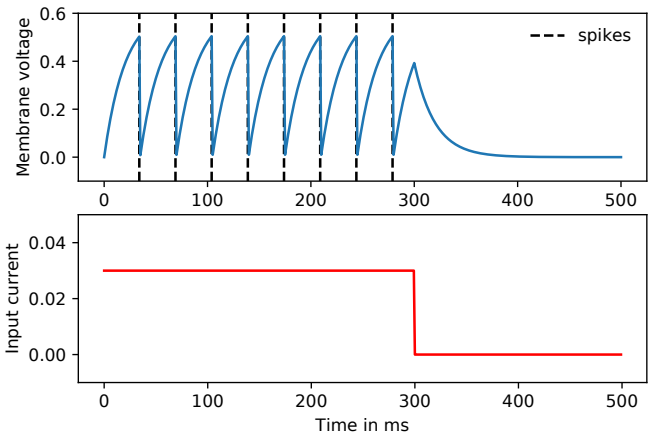

Figure 1.4: Time course of the membrane voltage for 500 ms with a constant input current of 0.03 for the first 300 seconds. The threshold voltage is set to $v_{th} = 0.5$ and the decay constant α is set to $\exp(-\frac{1}{20})$

.

To obtain richer dynamics, this neuron model is extended by introducing another state variable a_j, which then controls the spiking threshold (Bellec et al. (2019a)):

$$A_j^t = v_{th} + \beta a_j^t \tag{1.22}$$

$$a_j^{t+1} = \rho a_j^t + z_j^t \tag{1.23}$$

and

$$z_j^t = \mathcal{H}(v_j^t - A_j^t) \tag{1.24}$$

This state variable now increases every time the neuron emits a spike, and exponentially decays back to the baseline threshold v_{th} with the multiplicative factor $\rho \in (0, 1)$. In further experiments and text, this kind of neuron will be referred to as ALIF (adaptive leaky integrate and fire) neuron. It should be noted, that in the case of $\beta = 0$ this model reduces to the basic LIF neuron.

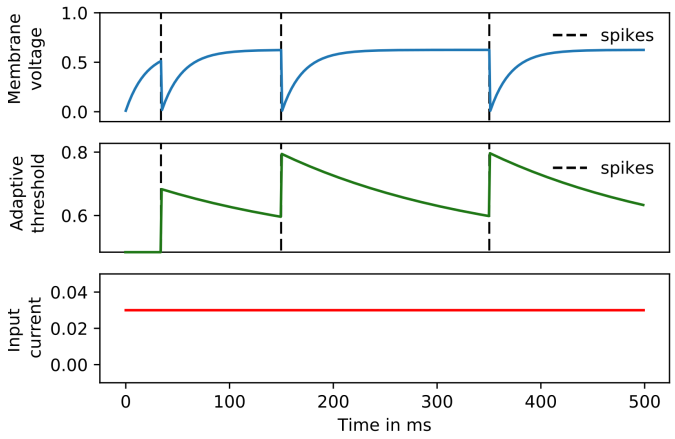

Figure 1.5: Time course of the membrane voltage and adaptive threshold for 500 ms with a constant input current of 0.03 for the first 300 seconds. The threshold voltage is set to $v_{th} = 0.5$ and the decay constant α is set to $\exp(-\frac{1}{20})$. The threshold adaptation strength β is set to 0.2 and the adaptation decay factor ρ is set to $\exp\left(-\frac{1}{200}\right)$

.

We can see the in figure 1.5 how this new state variable can now give rise to more complex behavior. When the network emits the first spike, the adaptive threshold jumps to a higher value. Since the input current is not strong enough to raise the membrane voltage above this new threshold, the neuron does not spike until the time when the adaptive threshold is sufficiently decayed back to the baseline threshold. After this spike, the threshold increases again which prevents the neuron from spiking for even longer.

This resulting behavior is known as spike-frequency adaptation and is also observed in biological neurons (see review paper of Ha and Cheong (2017))

To account for a feature that is also seen in biological neurons, we used a so called refractory period, where after a neurons spikes, it needs a certain time to recover, before it can spike again. This can be easily implemented by extending the state of the neuron by a clipped counter r, that is incremented by a certain number (e.g. $r^{t+1} = r^t + 4$) every time the neuron spikes and then decremented by 1 in every subsequent time step. The counter is clipped to avoid negative values in the counter. The spike function then has to implement a check if this counter is greater than 1 and in this case it should not emit a spike. During this period, the neuron is said to be refractory.

1.6.3 Output computation

As defined in equation 1.6, the output of a recurrent network is some function of the observable state $\boldsymbol{z}^t$. In the case of spiking neurons, this needs to be adapted a little bit, since $\boldsymbol{z}^t$ can only assume the discrete values $\{0, 1\}$ since we are

usually interested in obtaining a real valued output.

To obtain this, one could for example use the timing of $\boldsymbol{z}^t$ with respect to a reference signal, which is also known as temporal coding. This method is however very sensitive to noise since it is very hard in a stochastic setup to produce an exact spike timing for a single neuron.

The method that will be used here is some measure of the spike frequency of a neuron. This is done by introducing a new variable defined as:

$$\bar{\boldsymbol{z}}^t = \kappa \bar{\boldsymbol{z}}^{t-1} + \boldsymbol{z}^t \tag{1.25}$$

with $\kappa = \exp(-\frac{1}{\tau_{out}})$.
This quantity gives a measure of the instantaneous firing frequency of a neuron j and is a real-valued output which will then be sent into the output function $\sigma(\bar{\boldsymbol{z}}^t)$.
The output function is usually a weighted sum of these filtered values, that is then sent into a readout function

$$y_k = \sigma(\sum_{k=1}^{K}\sum_{j=1}^{N} \theta_{kj}^{out} \bar{z}_j^t) = \sigma(\boldsymbol{\theta}^{out} \bar{\boldsymbol{z}}^t) \tag{1.26}$$

One can think of this quantity inside the readout-function σ as the membrane potential of a so called readout neuron k, which is non-spiking, has a time constant τ_{out} and is connected to the network via a set of output synapses $\boldsymbol{\theta}^{out}$.

1.7 Learning with spiking neurons

1.7.1 Spike timing dependent plasticity

Donald Hebb postulated in his book "The Organization of Behavior" in 1949 that "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" (Hebb (1949)).

According to Caporale and Dan (2008), there is strong experimental evidence that the causality and spike timing of two neurons can determine the modification of their synaptic strengths. In spike timing dependent plasticity in the case of two excitatory neurons, when neuron A fires before neuron B, then the connection from A to B will be reinforced, and if neuron A fires after neuron B, this connection will be depressed.

Legenstein et al. (2005) show that under certain assumptions, neurons can learn any map from input to output spike trains. However STDP does not make use of the postsynaptic membrane voltage, and does not account for the occurrence of spike pairings that don't come in pairs. STDP is also an unsupervised and local learning algorithm, which in itself will usually not be sufficiently powerful for the types of problems that will be of interest later in this thesis.

1.7.2 Surrogate gradient method

When trying to apply the work horse of deep learning, gradient descent, to the training of spiking neural networks, one encounters the problem, that because of the use of a step function as the spiking function, the gradient of the loss function with respect to the parameters is not defined.

However, Bellec et al. (2018b) showed, that recurrent networks of spiking neurons can still be trained with backpropagation by using a surrogate gradient for the spike function $\mathcal{H}(v)$. They showed, that networks of spiking neural networks can achieve a performance on tasks like the sequential MNIST, that is almost on par with state-of-the-art LSTM (Hochreiter and Schmidhuber (1997)) networks. This almost means that we can eat our cake and have it too, since we get to keep the desirable sparsity properties of an all-or-nothing spike function, while being able to use gradient descent methods to optimize the network.

The pseudo-derivative $\psi(v^t)$ that will be used in this thesis is defined as follows:

$$\psi(v^t) = \gamma \max(0, 1 - |\frac{v^t - A_j^t}{v_{th}}|) \tag{1.27}$$

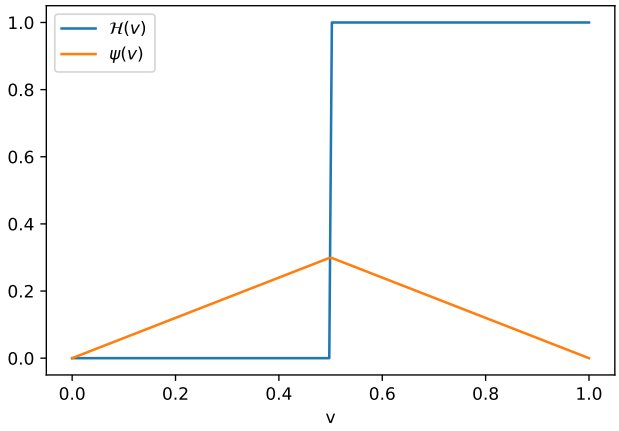

Figure 1.6: Pseudo derivative used to deal with the discontinuity of the spike function for a threshold voltage of $v_{thr} = 0.5$ and dampening factor $\gamma = 0.3$

.

It should be noted here, that in the case of a refractory period, the pseudo-derivative will be set to zero while a neuron is refractory, since in this case no spike can happen.

Chapter 2

Problem statement and goal

2.1 Problem statement

2.1.1 Locking problem

In order to compute the gradients with backpropagation through time, the activations of each neuron and the network inputs for the whole sequence have to be stored in memory during the forward evaluation of the network. This makes it infeasible to learn very long sequences. It also prevents the algorithm from beeing implemented in neuromorphic hardware due to strict constraints on memory usage.

There exists algorithms, that would allow to compute BPTT in a recursive and online fashion, however they are computationally very expensive and therefore not implementable for any large scale problem. This will be discussed further in the related work section.

Therefore an online algorithm is needed, which allows to compute the gradient in the feedforward pass, without the need of storing the whole input and activation sequence and that has a computational complexity, that would allow it to run on neuromorphic hardware.

2.1.2 Weight transport problem

Another problem that is described in Lillicrap et al. (2016), is that the backwards flow of errors with backpropagation uses an exact copy of the feedforward weights, which is argued to be biologically implausible.

They show this in the example of a one layer linear feedforward artificial neural network that has the following equations:

$$\boldsymbol{h} = \boldsymbol{\theta}_{in}\boldsymbol{x} \tag{2.1}$$

$$\boldsymbol{y} = \boldsymbol{\theta}_{out}\boldsymbol{h} \tag{2.2}$$

The gradient of a mean squared loss w.r.t input weights $E = \boldsymbol{e}^T\boldsymbol{e}$, with error signal ($\boldsymbol{e} = \boldsymbol{y}^{target} - \boldsymbol{y}$) for a given target vector $\boldsymbol{y}^{target}$ is then given as:

$$\nabla_{\theta_{in}} E = \boldsymbol{\theta}_{out}^T \boldsymbol{e}\boldsymbol{x}^T \tag{2.3}$$

This means, that the error signal flows back to the input weights through the transpose of the readout weights $\boldsymbol{\theta}_{out}^T$. The authors argue in the paper, that there is no known mechanism in the brain, that would allow such a symmetrical retrograde transmission of these errors.

2.2 Goal

The first goal of this thesis is to test the performance of e-prop, a learning algorithm that offers a solution to the locking problem of backpropagation through time. To do so, it will be necessary to find a range of tasks that involve learning complex patterns and memory.

Lillicrap et al. (2016) give a solution for the weight transport problem in feedforward neural networks, we are also interested if and how this could be applied also to a recurrent neural network. The second goal is therefore, to look at modifications of e-prop that allow avoiding the weight transport problem, and analyzing why and how these methods work.

Chapter 3

Related work

3.1 Feedback alignment

To address the weight transport problem in feedforward networks Lillicrap et al. (2016) introduced a mechanism that replaces the symmetric feedback weights used in backpropagation with a backwards path that uses a matrix of random feedback weights. They show that this approach leads to the same performance as backpropagation for a linear problem and also approaches the performance of backpropagation in a more complex non-linear function approximation task.

The authors call this method feedback alignment, since they found, that the feedback matrices align in their angles with the pseudo-inverse of the forward weight matrices during the course of training.

3.2 Broadcast alignment

Later Samadi et al. (2017) and Nøkland (2016) discovered, that it is even possible to directly broadcast the error signal back to each layer of the neural network, without going through each layer through the random feedback path like in feedback alignment.

Their experiments show, that this method performs equally good during training as backpropagation and feedback alignment on the MNIST and CIFAR-10 data set.

3.3 FORCE learning

Sussillo and F Abbott (2009) introduce a method for training recurrent neural network to produce complex output patterns that they call FORCE training.

The general setup for this algorithm is a recurrently neural network, whose neuron activities are combined via a linear readout to obtain the network output:

$$y(t) = \boldsymbol{w}^T \boldsymbol{z}(t) \tag{3.1}$$

The output of the network is then subtracted from a given target function $f(t)$ to obtain an error signal that is fed back to the network. It is essential for this algorithm, that the errors that are fed back to the network are sufficiently small in magnitude. This is ensured by updating the readout weights with the recursive least squares algorithm. This ensures that the output errors stay small and also that the changes in the recurrent weights that are needed to keep the error small will also decrease over time.

The authors evaluate the method on a range of difficult pattern-generation tasks such as reproducing songbird singing, or short movie-clips and achieve a very impressive performance. FORCE learning is however not argued to be biologically plausible, since it only modifies the readout weights, and the corresponding learning rule is not local. (see Bellec et al. (2019a)) It is also not clear, how well this method generalizes to tasks that go beyond the storage and generation of patterns.

3.4 Other online learning algorithms

3.4.1 Real time recurrent learning (RTRL)

Williams and Zipser (1989) described in 1989 already a method of computing the gradient of a recurrent network in an online manner, by recursively computing the gradient by multiplying the Jacobian $\frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{h}^{t-1}}$ of the network state, with the derivative of the state w.r.t the weights $\frac{\partial \boldsymbol{h}^{t-1}}{\partial \boldsymbol{\theta}}$ as defined in section 1.5.1. They call this real time recurrent learning (RTRL).

The main drawback of RTRL is however, that it is of complexity $\mathcal{O}(N^4)$ for a network of N recurrent neurons. This will be shown in section 4.1.2, where the recursive computation of the gradient is outlined. This makes it unfeasible to train very large networks.

3.4.2 Unbiased Online Recurrent Optimization (UORO)

Tallec and Ollivier (2017) use an approximation on top of RTRL which reduces its memory requirements to $\mathcal{O}(N^2)$. The idea of the algorithm is to assume the existence of an unbiased estimator $\tilde{\boldsymbol{G}}^t$ for $\frac{d\boldsymbol{h}^t}{d\boldsymbol{\theta}}$, that can be decomposed as an outer product $\tilde{\boldsymbol{h}}^t \otimes \tilde{\boldsymbol{\theta}}^t$ where $\tilde{\boldsymbol{h}}^t$ and $\tilde{\boldsymbol{\theta}}^t$ are of the same shape as $\boldsymbol{h}^t$ and $\boldsymbol{\theta}$.

Propagating $\tilde{\boldsymbol{G}}^t$ to the next time step using equation 1.13 will give an unbiased estimator of a higher rank at time $t+1$, which is then approximated and transformed back to rank-one with the so called rank-one trick. The authors give an recursive update rule for $\tilde{\boldsymbol{h}}^t$ and $\tilde{\boldsymbol{\theta}}^t$ which are of the same computational complexity as simulating the network forward in time.

Their learning rule is unbiased, however the approximation comes at the cost of injecting noise. The authors only provide experiments with small network sizes of up to 64 units and therefore it is not clear how this variance affects the training of larger networks. Their implementation also makes use of non-

local operations like vector norms to achieve minimum variance estimates, which could be argued to be biologically implausible.

3.4.3 Kernel RNN Learning (KeRNL)

Roth et al. (2019) use a rank-2 approximation of the Jacobian $J_h^\tau = \frac{\partial \boldsymbol{h}^t}{\partial \boldsymbol{h}^{t-\tau}}$ that assumes the form $J_h^\tau \approx \beta_{ij} K(\tau, \gamma_j)$. Since the Jacobian gives a measure of how much the activity of a neuron in the past influences the neuron in the future, they call this term the sensitivity. They interpret β_{ij} as an averaged measure of how much neuron j affects neuron i and $K(\tau, \gamma_j)$, which is a temporal kernel with coefficients γ_j, determines how far back into time this influence reaches.

The values for β_{ij} and γ_j are learned by analyzing how a small perturbation on the hidden state in the past affects the hidden state in the future. They set up an objective function where β_{ij} and γ_j are used to predict the perturbed hidden state in the future and the update rules come from gradient descent on this cost function. For this it is necessary to carry an additional perturbed copy $\tilde{\boldsymbol{h}}^t$ of the hidden state $\boldsymbol{h}^t$ forward in time.

In contrast to the learning rule that will be evaluated in this paper, KeRNL does not make explicit use of the neuron model in the approximation of the Jacobian. For the estimation of β_{ij} and γ_j it is necessary to include non-local communication, since every neuron needs to know all perturbations to the hidden state of all other neurons in the network.

3.4.4 Random feedback local online learning (RFLO)

Murray (2018) suggests a learning rule that is very similar to the one that will be outlined further in the next chapter. He is approximating the gradient by only using the terms that are local to a synapse (i.e. pre- and postsynaptic activities). To address the weight transport problem, he also uses a random projection of the error back to the network.

The algorithm is tested on a continuous output task, where the network learns to produce a one-dimensional periodic output, and it is trained to learn a sequence of actions by concatenating multiple short sequences.

The paper however restricts itself to leaky sigmoidal neurons and does not give a theory for how this algorithm could be applied to spiking neurons, or other neurons with more complex behaviors, such as LSTM or ALIF neurons. There is also no comparison to BPTT on common benchmarks for RNNs.

Chapter 4

Training algorithms

4.1 Eligibility propagation (e-prop)

4.1.1 Introduction

In Bellec et al. (2019a) we introduced an algorithm, that allows to compute an online approximation of the true gradient of a recurrent spiking or non-spiking network, with a computational complexity of $\mathcal{O}(N^2)$ and which can be extended by a mechanism similar to broadcast alignment, to mitigate the weight transport problem.

The basic idea of this algorithm is to factorize the gradient $\frac{dE^t}{d\boldsymbol{\theta}}$ into two parts: a loss dependent learning signal L_j and a synapse specific eligibility trace e_{ji} that carries information about past events forward in time, which is then used later to inform the weight update.

$$\frac{dE}{d\theta_{ji}} = \sum_t L_j^t e_{ji}^t \tag{4.1}$$

In the case of backpropagation, L_j^t corresponds to the total derivative of the loss, with respect to the networks observable state, i.e. $\frac{dE}{dz_j^t}$ and the eligibility trace e_{ji}^t corresponds to the derivative of the observable state, with respect to a change of the parameters $\frac{\partial z_j^t}{\partial \theta_{ij}}$.

It is important to note here, that $\frac{dE}{dz_j^t}$ is a total derivative, and that the observable state at time t, might influence the error at a later time step, i.e. the future. Since we are interested in obtaining an online learning algorithm, future values of the loss function will not be available at the computation time of the weight update.

One can however use a different approach and think of L_j^t as an instantaneous learning signal $\frac{\partial E}{\partial z_j^t}$ and therefore in an ideal implementation of gradient descent, e_{ji}^t must be equivalent to $\frac{dz_j^t}{d\theta_{ij}}$. This is a measure of how the observable state at

time t changes, when the parameters are changed. This is the view that will be taken on in the e-prop algorithm alongside with a suitable approximation of $\frac{dz_j^t}{d\theta_{ij}}$.

4.1.2 Derivation

We can apply the chain rule to obtain:

$$\frac{dz_j^t}{d\theta_{ij}} = \frac{\partial z_j^t}{\partial \boldsymbol{h}_j^t} \frac{d\boldsymbol{h}_j^t}{d\theta_{ij}} \tag{4.2}$$

Further decomposing $\frac{d\boldsymbol{h}_j^t}{d\theta_{ij}}$ yields:

$$\frac{d\boldsymbol{h}_j^t}{d\theta_{ij}} = \frac{\partial \boldsymbol{h}_j^t}{\partial \theta_{ij}} + \sum_{t' \leq t} \sum_k \frac{d\boldsymbol{h}_j^{t'}}{d\boldsymbol{h}_k^{t'-1}} \frac{d\boldsymbol{h}_k^{t'-1}}{d\theta_{ij}} \tag{4.3}$$

This is problematic, because for each of the N neurons in the network, it is necessary to do N multiplications (one with every other neuron) with the Jacobian $\frac{d\boldsymbol{h}_j^{t'}}{d\boldsymbol{h}_k^{t'-1}} = \boldsymbol{J}_{j,k}^{t'}$ and $\frac{d\boldsymbol{h}_k^{t'-1}}{d\theta_{ij}}$, (where each multiplication has complexity $\mathcal{O}(N^2)$) which results in a total time complexity of $\mathcal{O}(N^4)$. This is not feasible for any larger network.

The key assumption of e-prop is now that we can neglect inter-neuron dependencies by only considering the Jacobians $\frac{\partial \boldsymbol{h}_j^{t'}}{\partial \boldsymbol{h}_j^{t'-1}}$. This now reduces thes time complexity to $\mathcal{O}(N^2)$ since we are only multiplying with $D \times D$ Jacobians (where D is the dimension of the neuron state, which in the case of LIF or ALIF neurons is either 1 or 2).

This also results in a neuron specific term which allows us now to obtain a simple recursive equation for the eligibility trace by introducing a new variable $\boldsymbol{\epsilon}_{ji}^t$ that makes use of these simplified Jacobians. The variable is the so called eligibility vector, that can be computed locally in each synapse:

$$\boldsymbol{\epsilon}_{ji}^t = \frac{\partial \boldsymbol{h}_j^t}{\partial \boldsymbol{h}_j^{t-1}} \boldsymbol{\epsilon}_{ji}^{t-1} + \frac{\partial \boldsymbol{h}_j^t}{\partial \theta_{ij}} \tag{4.4}$$

and finally the eligibility trace:

$$e_{ji}^t = \frac{\partial z_j^t}{\partial \boldsymbol{h}_j^t} \boldsymbol{\epsilon}_{ji}^t \tag{4.5}$$

As we are now working with an approximation of the true gradient, the gradient computed with e-prop will be denoted as $\widehat{\frac{dz_j^t}{d\theta_{ij}}}$

4.1.3 E-prop for LIF and ALIF neurons

Let us now analyze the computation of L_j^t and e_{ji}^t in the case of LIF and ALIF network models in the framework of regression and classification problems, in

which most machine learning tasks can be formulated. It should be noted here, that the eligibility trace does not depend on the loss function and therefore is the same for both problems.

To account for the exponential filtering in the output neuron, we will use a slight modification and compute as the learning signal L_j^t the quantity $\frac{\partial E}{\partial \bar{z}_j^t}$ instead of $\frac{\partial E}{\partial z_j^t}$. $\bar{z}_j^t$ is used here instead of z_j^t to avoid introducing terms that depend on future time steps in the equations.

This will allow us, to move the exponential filtering of the readout into the eligibility trace:

$$\bar{e}_{ji}^t = \kappa \bar{e}_{ji}^{t-1} + e_{ji}^t \tag{4.6}$$

The approximated total derivative is therefore decomposed as:

$$\widehat{\frac{dz_j^t}{d\theta_{ij}}} = \frac{\partial E}{\partial \bar{z}_j^t} \frac{d\bar{z}_j^t}{d\theta_{ji}} = \sum_t L_j^t \bar{e}_{ji}^t \tag{4.7}$$

This can be easily verified by expanding:

$$\widehat{\frac{d\bar{z}_j^t}{d\theta_{ji}}} = \frac{\partial \bar{z}_j^t}{\partial z_j^t} e_{ji}^t + \frac{\partial \bar{z}_j^t}{\partial z_j^{t-1}} e_{ji}^{t-1} + ... \tag{4.8}$$

and using the derivative:

$$\frac{\partial \bar{z}_j^t}{\partial z_j^{t-t'}} = \kappa^{t-t'} \tag{4.9}$$

To obtain the equations for the weight updates in case of LIF and ALIF neurons, one needs now to compute the Jacobian $\frac{\partial \boldsymbol{h}_j^t}{\partial \boldsymbol{h}_j^{t-1}}$, the derivative of the spike function $\frac{\partial z_j^t}{\partial \boldsymbol{h}_j^t}$ and the partial derivative of the hidden state, w.r.t the weights $\frac{\partial \boldsymbol{h}_j^{t'}}{\partial \theta_{ij}}$.

In the case of LIF neurons, the hidden state only consists of a single variable, the membrane potential $\boldsymbol{v}$. From equation 1.19 we obtain the Jacobian:

$$\frac{\partial \boldsymbol{h}_j^t}{\partial \boldsymbol{h}_j^{t-1}} = \frac{\partial v_j^t}{\partial v_j^{t-1}} = \alpha \tag{4.10}$$

and as the second multiplier for the case of the input weights:

$$\frac{\partial \boldsymbol{h}_j^t}{\partial \theta_{ij}^{in}} = \frac{\partial v_j^t}{\partial \theta_{ij}^{in}} = x_i^t \tag{4.11}$$

The derivative of the spike function is obtained from the definition in 1.27:

$$\frac{\partial z_j^t}{\partial \boldsymbol{h}_j^t} = \frac{\partial z_j^t}{\partial v_j^t} = \psi(v_j^t) = \psi_j^t \tag{4.12}$$

We can now compute the component of the eligibility vector by solving the geometric series defined in 4.4:

$$\epsilon_{ji} = \sum_{t' \leq t-1} \alpha^{t-t'} x_i^{t'} = \psi_j^t \bar{x}_i^{t-1} \tag{4.13}$$

By combining this we now have the complete eligibility trace:

$$e_{ji}^t = \psi_j^t \bar{x}_i^{t-1} \tag{4.14}$$

In case of ALIF neurons, the state vector $\boldsymbol{h}_j^t$ now also contains the threshold adaptation variable a_j. The eligibility vector therefore contains two components:

$$\boldsymbol{\epsilon}_{ji} = \begin{pmatrix} \epsilon_{ji,v} \\ \epsilon_{ji,a} \end{pmatrix} \tag{4.15}$$

We will now also need another derivative:

$$\frac{\partial z_j^t}{\partial \boldsymbol{h}_j^t} = \begin{pmatrix} \frac{\partial z_j^t}{\partial v_j^t} & \frac{\partial z_j^t}{\partial a_j^t} \end{pmatrix} = \begin{pmatrix} \psi_j^t & -\beta \psi_j^t \end{pmatrix} \tag{4.16}$$

The Jacobian $\frac{\partial \boldsymbol{h}_j^t}{\partial \boldsymbol{h}_j^{t-1}}$ is a 2×2 matrix with entries:

$$\frac{\partial \boldsymbol{h}_j^t}{\partial \boldsymbol{h}_j^{t-1}} = \begin{pmatrix} \frac{\partial v_j^t}{\partial v_j^{t-1}} & \frac{\partial v_j^t}{\partial a_j^{t-1}} \\ \frac{\partial a_j^t}{\partial v_j^{t-1}} & \frac{\partial a_j^t}{\partial a_j^{t-1}} \end{pmatrix} = \begin{pmatrix} \alpha & 0 \\ \psi_j^t & \rho - \beta \psi_j^t \end{pmatrix} \tag{4.17}$$

To compute the eligibility vector we now need to know:

$$\frac{\partial \boldsymbol{h}_j^t}{\partial \theta_{ij}^{in}} = \begin{pmatrix} \frac{\partial v_j^t}{\partial \theta_{ij}^{in}} & \frac{\partial a_j^t}{\partial \theta_{ij}^{in}} \end{pmatrix}^T \tag{4.18}$$

For the second component of this vector, $\frac{\partial a_j^t}{\partial \theta_{ij}^{in}}$ we can use that

$$\frac{\partial a_j^t}{\partial \theta_{ij}^{in}} = 0 \tag{4.19}$$

Using 4.4 we now obtain the following equations for the components of the eligibility vector:

$$\epsilon_{ji,v}^t = \begin{pmatrix} \alpha & 0 \end{pmatrix} \begin{pmatrix} \epsilon_{ji,v}^{t-1} \\ \epsilon_{ji,a}^{t-1} \end{pmatrix} + x_i^t = \bar{x}_i^{t-1} \tag{4.20}$$

and

$$\epsilon_{ji,a}^t = \begin{pmatrix} \psi_j^t & \rho - \beta \psi_j^t \end{pmatrix} \begin{pmatrix} \epsilon_{ji,v}^{t-1} \\ \epsilon_{ji,a}^{t-1} \end{pmatrix} + \frac{\partial a^t}{\partial \theta_{ij}^{in}} = \psi_j^t \bar{x}_i^{t-1} + (\rho - \beta \psi_j^t) \epsilon_{ji,a}^{t-1} \tag{4.21}$$

The full eligibility trace can now be computed as:

$$e_{ji}^t = \frac{\partial z_j^t}{\partial \boldsymbol{h}_j^t} \boldsymbol{\epsilon}_{ji}^t = \begin{pmatrix} \psi_j^t & -\beta \psi_j^t \end{pmatrix} \begin{pmatrix} \bar{x}_i^{t-1} \\ \epsilon_{ji,a}^t \end{pmatrix} = \psi_j^t (\bar{x}_i^{t-1} - \beta \epsilon_{ji,a}^t) \tag{4.22}$$

4.1.4 Computation of update rules

Since we are using the partial derivative $L_j^t = \frac{\partial E}{\partial \bar{z}_j^t}$ as the learning signal, this is equivalent to $\frac{\partial E^t}{\partial \bar{z}_j^t}$ i.e. the instantaneous error.

Supervised regression

The output of the spiking neural network in a regression setup with an output dimension of K, is simply given by the weighted sum of the exponentially filtered spike trains:

$$y_k^t = \sum_{k=1}^{K} \sum_{j=1}^{N} \theta_{kj}^{out} \bar{z}_j^t \tag{4.23}$$

Under an additive Gaussian noise model for the target value and a maximum likelihood approach, one obtains the mean squared error loss function:

$$E = \sum_{t=1}^{T} E^t = \frac{1}{2} \sum_{t=1}^{T} \sum_{k=1}^{K} (y_k^{target,t} - y_k^t)^2 \tag{4.24}$$

The derivative of this loss is readily obtained:

$$L_j^t = \frac{\partial E^t}{\partial \bar{z}_j^t} = \sum_{k=1}^{K} (y_k^{target,t} - y_k^t) \theta_{kj}^{out} \tag{4.25}$$

Combining this learning signal with the eligibility trace and a learning rate η yields the following final update rules:

For input and recurrent weights:

$$\Delta \theta_{ji}^{in/rec} = \eta \sum_t (\sum_k \theta_{kj}^{out} (y_k^{target,t} - y_k^t) \bar{e}_{ji}^t \tag{4.26}$$

The update term for the readout weights is easily obtained as:

$$\Delta \theta_{kj}^{out} = -\eta \nabla_{\theta_j^{out}} E = \eta \sum_t (y_k^{target,t} - y_k^t) \bar{z}_j^t \tag{4.27}$$

Supervised classification

When applying a Bayesian framework to the classification problem, one obtains the functional form of the softmax function for the class probabilities:

$$p(class = k | time = t) = \pi_k^t = softmax_k(y_1^t, ..., y_K^t) = \frac{\exp y_k^t}{\sum_{k'=1}^{K} \exp y_{k'}^t} \tag{4.28}$$

$$y_k^t = \sum_{k=1}^{K} \sum_{j=1}^{N} \theta_{kj}^{out} \bar{z}_j^t \tag{4.29}$$

By using a maximum likelihood approach for a given labeled data set one obtains the so called cross-entropy loss function:

$$E = \sum_{t=1}^{T} E^t = -\sum_{t=1}^{T}\sum_{k=1}^{K} \pi_k^{target,t} \log \pi_k^t \tag{4.30}$$

The loss-dependent learning signal can then be computed as:

$$L_j = \frac{\partial E}{\partial \bar{z}_j^t} = \sum_k (\pi_k^{target,t} - \pi_k^t)\theta_{kj}^{out} \tag{4.31}$$

Again we combine the learning signal with the eligibility trace to get the final update rules:

Input and recurrent weights:

$$\Delta\theta_{ji}^{in/rec} = \eta \sum_t (\sum_k \theta_{kj}^{out}(\pi_k^{target,t} - \pi_k^t))\bar{e}_{ji}^t \tag{4.32}$$

Readout weights:

$$\Delta\theta_{kj}^{out} = -\eta \nabla_{\theta_j^{out}} E = \eta \sum_t (\pi_k^{target,t} - \pi_k^t)\bar{z}_j^t \tag{4.33}$$

Binary classification with reinforcement learning

It is also possible to apply e-prop in a reinforcement learning setup. The idea of this is to modify the computation of gradients, like it is done in a standard policy gradient method, by using e-prop. We consider here a so called actor-critic method as described in Sutton and Barto (2018). The following update rules are also described in the supplement of Bellec et al. (2019b).

The loss function in this algorithm consists of two terms: the so called actor-critic loss, which gives a measure of how good the current policy is, and a reward-prediction loss, which makes sure the estimate of the reward baseline is correct. The reward was 1 when the correct class was chosen and 0 otherwise. In the context of this thesis, this loss is simplified, because the analyzed task only requires a binary action A^T at time step T at the end of the sequence. The actor then finds itself in state S^T and receives a single reward $R^T \in \{0, 1\}$ after taking the action. The actor-critic loss is therefore defined as:

$$E = -(R^T - V^T)\log(\pi(A^T|S^T, \boldsymbol{\theta})) + C_{val}(R^T - V^T)^2 \tag{4.34}$$

We will treat the two components of the loss separately and use the two weight update rules additively as defined in Bellec et al. (2019a).

The update rule for the value estimation (critic) is given by:

$$\Delta\theta_{ji}^{in} = \eta C_{val}(R^T - V^T)\theta_j^{out,critic}\bar{e}_{ji}^T \tag{4.35}$$

The update rule for the policy (actor) is defined as:

$$\Delta\theta_{ji}^{in/rec} = \eta(R^T - V^T)\sum_k \theta_{kj}^{out,action}(\mathbb{1}_{k=A^T} - \pi_k^T)\bar{e}_{ji}^T \tag{4.36}$$

where $\mathbb{1}_{k=A^T}$ is the indicator function that is 1 if $k = A^T$ and 0 otherwise. The readout weight update for the actor part is given by:

$$\Delta\theta_{kj}^{out} = \eta(R^T - V^T)(\mathbb{1}_{k=A^T} - \pi_k^T)\bar{z_j}^T \tag{4.37}$$

and for the value function:

$$\Delta\theta_{critic,j}^{out} = \eta C_{val}(R^T - V^T)\bar{z_j}^T \tag{4.38}$$

4.1.5 E-prop and the weight transport problem

We consider here three variants of e-prop: Symmetric, random and adaptive e-prop.

The most straightforward version is the symmetric e-prop, with the update rule as stated in equation 4.32. This learning rule makes use of the network readout weights θ_{kj}^{out} to send the error signals back to the synapses. This raises the issue of a symmetric feedback path, which is argued to be biologically implausible (Lillicrap et al. (2016)).

It is however possible, to replace the feedback weights θ_{kj}^{out} with a matrix B_{kj} of randomly sampled values which then stays constant during the course of the training. We call this method *random e-prop*. This was inspired by the idea of broadcast alignment (Nøkland (2016) and Samadi et al. (2017)) where the modulator signal (error) is sent back to each hidden layer in the feedforward network via a different random feedback matrix. This works, because the readout matrix will align its angle with the transpose of the feedback matrix during the course of training.

A key difference here is that we are working with a recurrent neural network and therefore the weights of the unrolled network are the same for each "layer". We will see later in the experiments, that it is beneficial to use the same feedback matrix B_{kj} for each time step, and not vary it over time like it could be expected from a straightforward application of broadcast alignment to the unrolled network.

In case the random feedback is not sufficient, there is also an option to decouple the feedback from the forward path with a modification of the Kolen-Pollack algorithm, as described in Akrout et al. (2019). In this algorithm, we initialize a random feedback matrix B_{kj} like in *random e-prop*. Now the feedback matrix is not fixed, but is also changing during the course of training. The same weight updates that are applied to θ_{kj}^{out} are also applied to B_{kj} and additionally both matrices receive a weight decay of the form:

$$\Delta\theta_{kj}^{out} = -\lambda\theta_{kj}^{out} \tag{4.39}$$

and

$$\Delta B_{kj} = -\lambda B_{kj} \tag{4.40}$$

with a small factor $0 < \lambda < 1$ This also solves the weight transport problem, but in its straightforward implementation then raises the issue of how the weight changes are transported. We call this version *adaptive e-prop*.

Chapter 5

Experiments

5.1 Pattern generation task

5.1.1 Introduction

As a first task, we consider a simple pattern generation task to compare our algorithm to learning rules like the FORCE training Nicola and Clopath (2017). In this task, the network should reproduce a fixed pattern by only having a clock-like signal available as input.

5.1.2 Task details

The network was tasked to produce a weighted sum of four sinusoids with different fixed frequencies. The target sequence had a duration of 1000ms and the four sinusoids had fixed frequencies of 1Hz, 2Hz, 3Hz and 5Hz. The amplitudes or weights of each sinusoidal component were drawn from a uniform distribution with values ranging from 0.5 to 2. Each component was also randomly phase-shifted with a phase that was also sampled uniformly from 0 to 2π. The network received a clock signal with five steps at the input. The error was defined as mean of the squared differences between network output and target.

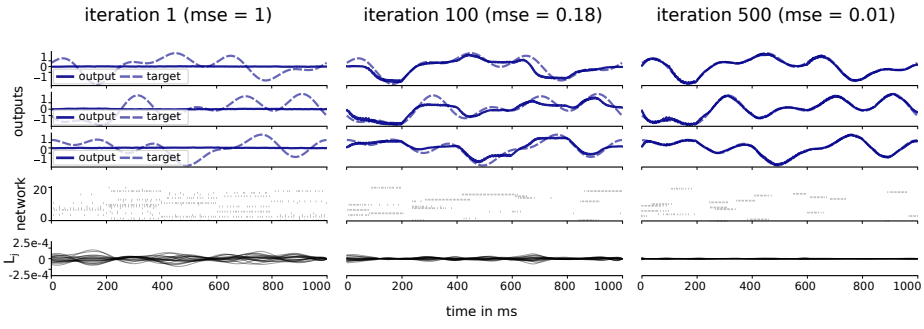

Figure 5.1: Spike raster and network outputs for the pattern generation task that was trained with e-prop1. (from (Bellec et al., 2019a))

5.1.3 Details of network model

The simulation time step was 1ms. 600 recurrently connected LIF neurons were used for solving this task. These neurons were fully connected to the 20 input neurons that produced the clock signal and also to a single readout neuron. The output at each time step was then given by the membrane potential of the readout neuron.

The input signal was encoded by 20 input neurons, that fired in groups in 5 successive time steps with a length of 200ms each. The membrane time constants were set to $\tau_m = 20$ ms and the firing threshold was set to $v_{th} = 0.41$. The neurons had a refractory period of 3 ms. The readout neuron had a time constant $\tau_{out} = 20ms$.

5.1.4 Details of training procedure

The input, recurrent and output weights of the network were trained for 1000 iterations with a learning rate of 0.003 and the Adam optimizer. After every 100 iterations, the learning rate is decayed with a multiplicative factor of 0.7. A dampening factor of $\gamma = 0.3$ for the pseudo-derivative of the spiking function was used. A batch size of a single sequence is used for training and testing. To avoid an implausible high firing rate, a regularization term is added to the loss function, that keeps the neurons closer to a target firing rate of 10Hz.

We also tried variations of random e-prop, where we varied the feedback matrix over time. In the case of a change every 1ms, in every time step, a different feedback matrix was used. We sampled the matrices once before the training was started and therefore the feedback matrix for any particular time step stayed constant during the whole training.

5.1.5 Results

Figure 5.2 shows the networks mean squared error over the course of the training. After around 700 iterations, all versions of random e-prop converge to a mean squared error below 0.03. However there is still a small gap remaining to backpropagation through time, which is able to drive the error down to almost zero. As a control, we trained a network with BPTT, but disabled its recurrent connections. This version is not able to solve the task very well.

In figure 5.3 the final training performance for the different algorithms are shown. An additional control experiment was to replace the random feedback weight matrix with a matrix of constant values $\frac{1}{\sqrt{N}}$. This would correspond to a global learning signal. This did not work very well for this task, and performed even worse than the network with no recurrent connections.

Varying the feedback matrix over time also influences the result: The best results are achieved when the feedback matrix stays fixed during the whole sequence length. However the network is still able to learn the sequence with a

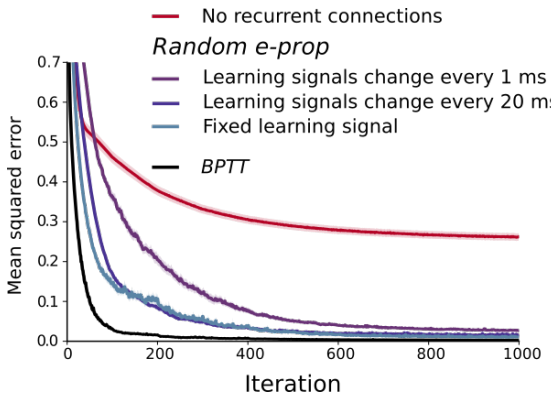

Figure 5.2: Averaged mse over training iterations for different variants of random e-prop.(from (Bellec et al., 2019a))

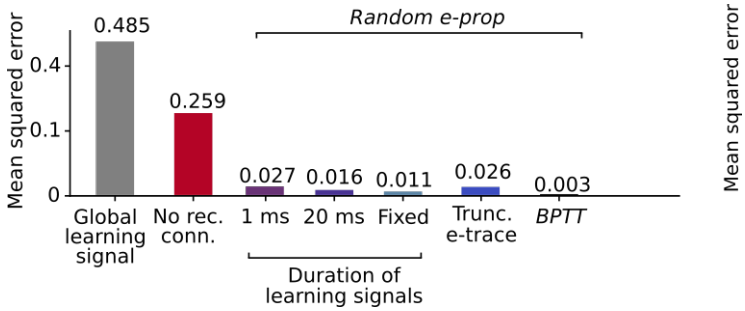

Figure 5.3: Bar plot of final performance for the pattern generation task with different training methods.(from (Bellec et al., 2019a))

feedback matrix that is changing every 20 ms or every 1 ms, although with a slightly higher final error.

One drawback of this task is, that it is not needed to learn any dependencies far backwards in time. This is due to the fact, that there is an error signal at each time-step and almost no information from the past is needed to reproduce the pattern. This can be seen in the bar with the label "trunc. e-trace" where the eligibility trace is cut off to only reach one time step backwards and the network can still learn to reproduce the pattern with a low error.

5.2 Movie replay task

5.2.1 Introduction

This task is an extension of the movie replay task as described by (Clopath 2017). The task was to reproduce at each time-step the pixel values, that correspond to a short movie-clip. As an extension, the network had to replay one out of three possible movie sequences, where the target movie was indicated by an additional group of input neurons. Similar to the pattern generation task,

the network received a clock-like input signal.

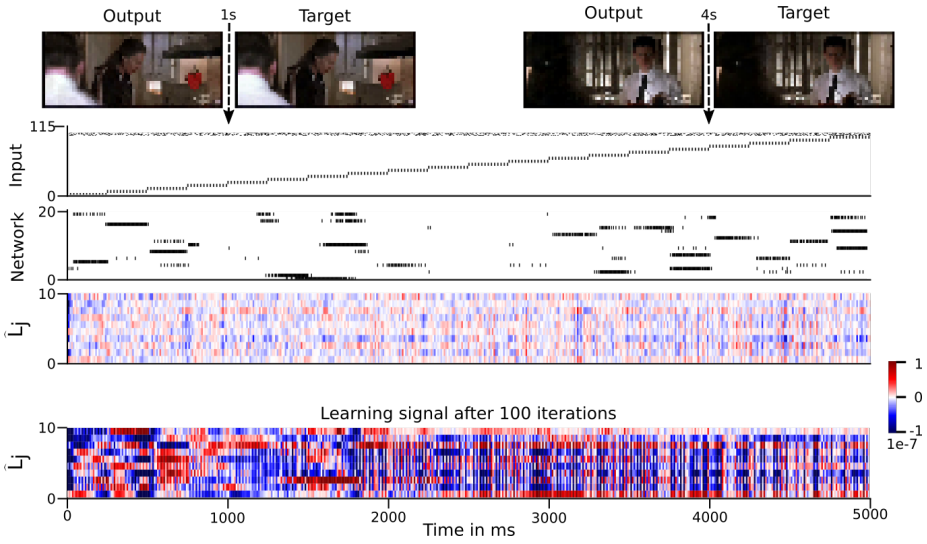

Figure 5.4: Spike raster for the movie replay task that was trained with *random e-prop*. The bottom two rows show the learning signals for 10 randomly chosen neurons. After training, the magnitude of the learning signals is very small, an indication of a low error.

5.2.2 Details of the movie replay task:

The target video signal had a height 28, a width of 66 and 3 color channels for a total of 5544 dimensions. The target values were normalized to be in the range from 0 to 1 in each color channel.

We chose three movies from the Hollywood 2 data set (Marszałek et al. (2009)). We used the first 5 seconds from the movies with the file names "sceneclipautoautotrain000[19,61,71].avi". The movies were then spatially subsampled to the desired resolution of 66×28. The original movies had 25 frames per second, which were extended with linear interpolation to 1000 frames per second to match our simulation time step of 1ms.

5.2.3 Details of the network model and of the input scheme:

The network consisted of 700 LIF and 300 ALIF neurons that were recurrently connected. Each neuron had a membrane time constant of $\tau_m = 20$ ms and a refractory period of 5 ms. The time constant for the threshold adaptation was 500 ms for every ALIF neuron. The firing threshold was set to $v_{\text{th}} = 0.62$. The network outputs were provided by the weighted sum of the membrane potential of 5544 readout neurons with a time constant $\tau_{out} = 4$ ms.

The network received input from 115 input neurons, divided into 23 groups of 5 neurons. The first 20 groups indicated the current phase of the target

sequence similar to (Nicola and Clopath (2017)). Neurons in group $i \in \{0, 19\}$ produced 50 Hz regular spike trains during the time interval $[250 \cdot i, 250 \cdot i + 250)$ ms and were silent at other times. The remaining 3 groups encoded the selection of the movie by having the group that corresponded to a given movie fire with a poisson spike train of 50 Hz, while the other 2 groups remained silent.

5.2.4 Details of the learning procedure

For learning, we carried out 5 second simulations, where the network produced a 5544 dimensional output pattern. In each simulation one of the three movies was uniformly sampled as the target which was then indicated to the network by a 50 Hz Poisson spike train from the corresponding input group.

We applied synaptic plasticity using *random e-prop*, where the random feedback weights B_{kj} were sampled from a Gaussian distribution with a mean of 0 and a variance of 1. Weight updates were applied once after every 8 trials and the gradients were accumulated during those trials additively. The parameter updates were implemented using Adam (Kingma and Ba (2014)) with a learning rate of $2 \cdot 10^{-3}$ and default hyperparameters. After every 100 weight iterations, the learning rate was decayed by a factor of 0.95. To avoid an excessively high firing rate, a regularization term was added to the loss function to achieve a target firing rate of $f^{\text{target}} = 10$ Hz.

5.2.5 Results

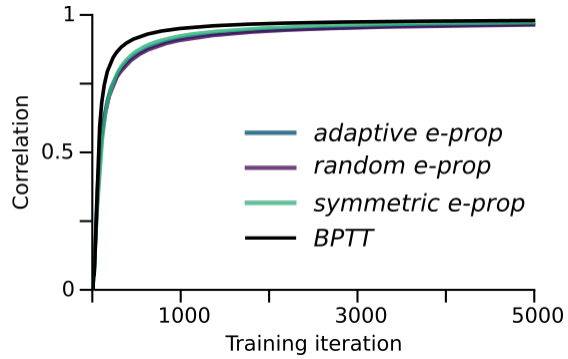

Figure 5.5: Performance curves for different varieties of e-prop and BPTT.

As a measure of similarity between the target movie and the output, we report the Pearson correlation coefficient averaged over a batch of 8 trials, between the target and the output. Figure 5.5 shows the correlation over the course of the training.

	Correlation
BPTT	0.980
Random e-prop	0.964
Adaptive e-prop	0.966
Symmetric e-prop	0.973

Table 5.1: Final performance comparison after training.

It can be seen, that all three version of e-prop perform very similarly to backpropagation through time. Table 5.1 summarizes the final performance after training. All of the tested algorithms managed to solve this task nearly perfectly with a correlation over 0.96.

5.3 Evidence accumulation task

5.3.1 Introduction

This task was inspired by the evidence-accumulation task done by Morcos and Harvey (2016). In their task, mice were head-fixed and ran down a T-shaped maze in a virtual-reality environment. They were presented with 6 visual cues on fixed locations that appeared on either the right or left wall. At the T-section the mice had to turn their head in the direction which had more cues.

To simulate this, visual cues on the right and left wall were encoded by two groups of input neurons (represented as red and blue), that fired when a cue was active. To allow for an unambiguous target label, 7 instead of 6 cues were used. After a delay of around $1s$ another group of input neurons signaled the time to make the decision with a recall cue.

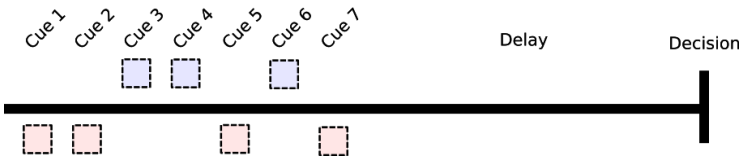

Figure 5.6: Sketch of the evidence accumulation task. During the evidence period, the network has to keep track of the number of cues on the left and right side, remember these values during the delay period and finally make the right decision.

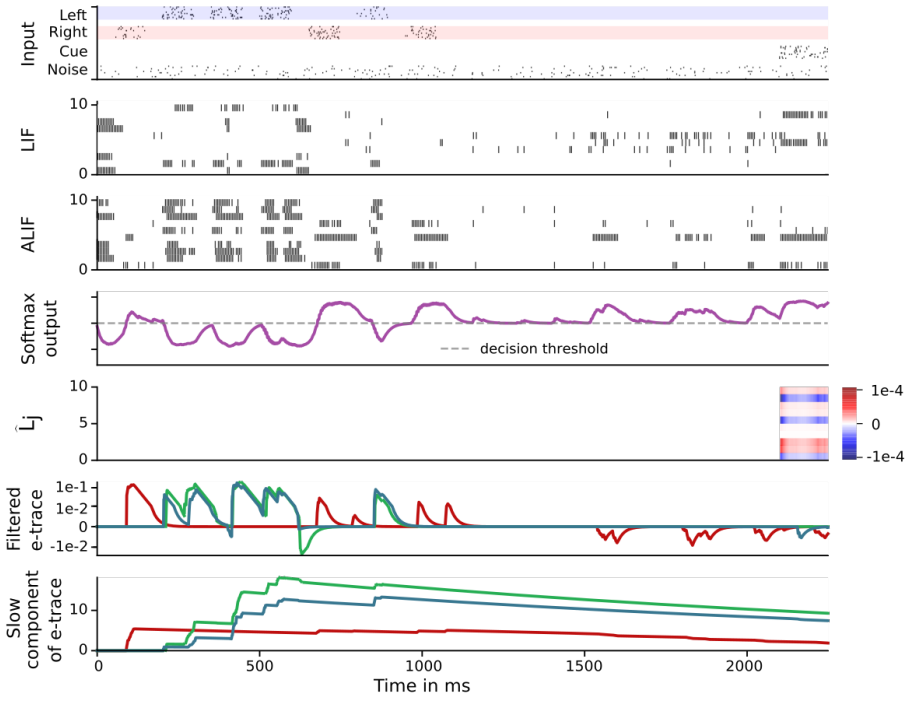

Figure 5.7: Spike raster for the click count task that was trained with *random e-prop*. The top row shows the four input channels consisting of 10 neurons each, which are encoding the two input groups (left and right group), the recall cue and the background noise. The row below the input is the spike raster of the network. 50 recurrent and 50 adaptive neurons were used for the task. The networks output is shown below the spike raster. The output is given as the softmax over two output neurons. Below this, the learning signals for 10 selected neurons are shown. The second to last row shows the eligibility trace for 3 selected synapses. The last row shows the slow component of the eligibility trace $\epsilon_{ji,a}$ for the same synapses.

5.3.2 Details of the network model and of the input scheme:

We encoded the network input by four groups of 10 neurons each. All groups emit 40 Hz Poisson spike trains when active, except for the last group that continuously emits a 10 Hz Poisson spike train, to represent background noise. The first two groups encode the cues by giving of a 40 Hz Poisson spike train that lasts for 100 ms when a cue is selected. The cues are separated by 50 ms. After the 7 cues, there is a delay period of 1050 ms, where only the noise group is active. This is followed by a 150 ms recall cue, where the third group gives of a 40 Hz Poisson spike train.

For this task, we used a recurrent LSNN network consisting of 50 standard LIF neurons and 50 LIF neurons with adaptive thresholds. One network simulation had a total duration of 2250 ms with a simulation time-step of $dt = 1$

ms. The neurons were connected in an all-to-all fashion. All neurons had a membrane time constant of $\tau_m = 20$ ms and a baseline threshold of $v_{\text{th}} = 0.6$. Adaptive neurons linearly spanned a range of threshold adaptation time constants $\tau_{a,j}$ between 2000 and 4000 ms. We used a refractory period of 5 ms. The threshold adaptation strength was calculated for each adaptive neuron with the formula $\beta_j = 1.7 \frac{1-\exp(-dt\tau_{a,j}^{-1})}{1-\exp(-dt\tau_m^{-1})}$.

In the supervised setup, the output is computed during the recall period (the last 150 ms) with the help of two readout neurons with time constant $\tau_{out} = 20$ ms that are being sent into a softmax function to obtain normalized output probabilities. The decision is made at the end of the sequence by averaging all output probabilities during the recall period. This allows us to use an instantaneous loss for the learning signal.

5.3.3 Details of the learning procedure:

Supervised setup

The input, recurrent and output weights of the network were trained with a learning rate of 0.005 and the Adam algorithm with default hyperparameters (Kingma and Ba, 2014). Training was stopped when a misclassification rate below 0.08 was reached. The distribution of the random feedback weights B_{kj} was sampled from a normal distribution with mean 0 and variance 1. We used a batch size of 64 trials, which means the network was shown 64 trials, during which the weight updates were accumulated additively and then applied at the end. We also used an additional regularization term to avoid firing rates above 10 Hz.

Reinforcement learning setup

The network setup was the same as in the supervised case, with the only difference being in the output computation. The output is an action that is sampled at the last timestep, according to the probability that is given by the softmax over the readout neurons at the last timestep. To do this, the readout time constant τ_{out} was increased to 150 ms.

The weight update is also computed after the network was shown 64 trials and the learning rates and used optimizer were the same as in the supervised setup. We used the actor-critic setup as described in section 4.1.4 where the loss term corresponding to the value function estimate is weighted with a factor $C_{val} = 0.1$.

5.3.4 Results

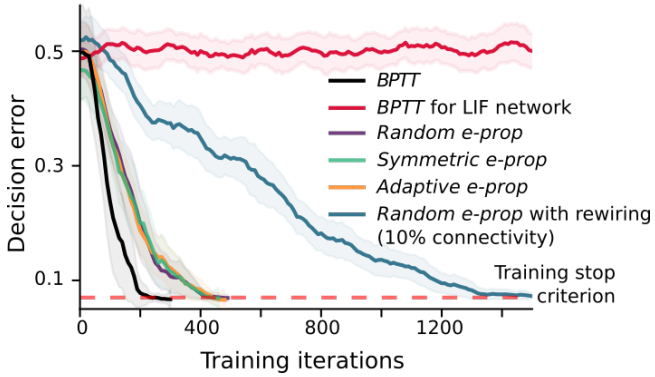

Figure 5.8: Performance plot for the supervised setup. Random e-prop solved the task on average after 308 iterations while with BPTT it was 156.

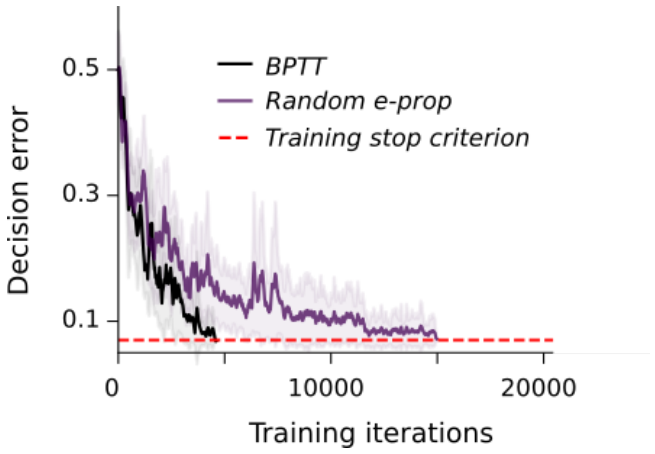

Figure 5.9: Performance plot for the reinforcement setting.

	Iterations until task solved
BPTT	156
Random e-prop	308
Adaptive e-prop	331
Symmetric e-prop	309
Random e-prop with rewiring (10% connectivity)	1219
Reinforcement learning with BPTT	3050
Reinforcement learning with random e-prop	8880

Table 5.2: Average number of iterations needed until the task was solved.

As can be seen in figure 5.8, all variants of e-prop manage to solve the task. As a control, we also tried using a network with 100 LIF neurons that was trained

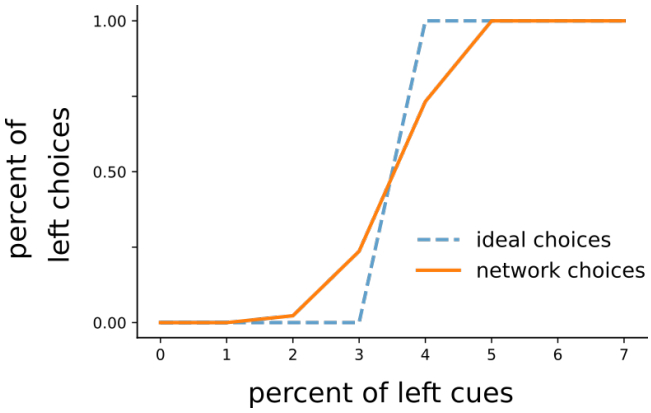

Figure 5.10: Percentage of choices of the network trained with e-prop1 for the red group as a function of the number of red cues. In the easiest cases where there is a red:blue split of 0:7, 6:1 or 5:2, the network has an accuracy of almost 100% which drops down to around 70% for the harder cases of a 3:4 split.

with BPTT. It did not manage to achieve an error below the baseline of 50%. Table 5.2 gives an overview about the average training iterations that are needed to solve the task for each variant.

The three e-prop variants perform almost the same on this task. When using rewiring (Bellec et al. (2018a)), the network still manages to solve the task, although it needs around four times as many training iterations. The task can also be solved using reinforcement learning. In figure 5.9 we can see that random e-prop can solve the task in about three times as many iterations as when using BPTT.

Figure 5.10 shows the error distribution as a function of the number of left cues. It can be seen, that the network produces almost all errors in the cases, where the number of left and right cues follows a 3:4 split. In the cases where all cues are on one side, or the cues are split 5:2, the network makes almost no errors. This was also found in the experiments of Morcos and Harvey (2016).

Chapter 6

Analysis of random e-prop

6.1 Introduction

To aid the following analysis, a few principles form linear algebra are introduced in this section.

6.1.1 Positive definiteness of a matrix

A symmetric square matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is called positive definite if for all non-zero vectors $\boldsymbol{z} \in \mathbb{R}^n$ it holds that $\boldsymbol{z}^T \boldsymbol{A} \boldsymbol{z} > 0$. The matrix is called positive semi-definite if $\boldsymbol{z}^T \boldsymbol{A} \boldsymbol{z} \geq 0, \forall \boldsymbol{z} \neq \boldsymbol{0}$. Analogously the matrix is negative definite if $\boldsymbol{z}^T \boldsymbol{A} \boldsymbol{z} < 0, \forall \boldsymbol{z} \neq \boldsymbol{0}$ and negative semi-definite if $\boldsymbol{z}^T \boldsymbol{A} \boldsymbol{z} \leq 0, \forall \boldsymbol{z} \neq \boldsymbol{0}$.

This can be checked by looking at the eigenvalues of the matrix. If all eigenvalues are strictly positive, the matrix is positive definite, if all eigenvalues are strictly negative the matrix is negative definite. Since the matrices of interest in this analysis, will not always be symmetric we need to extend this definition. This is done by rewriting the quadratic form as $\boldsymbol{z}^T \boldsymbol{A} \boldsymbol{z}$ as $\boldsymbol{z}^T \frac{\boldsymbol{A}^T + \boldsymbol{A}}{2} \boldsymbol{z}$ and then analyzing this symmetric matrix. The equivalence can be easily verified by transposing the scalar term $(\boldsymbol{z}^T \boldsymbol{A}^T \boldsymbol{z})^T = \boldsymbol{z}^T \boldsymbol{A} \boldsymbol{z}$. The matrix $\frac{\boldsymbol{A}^T + \boldsymbol{A}}{2}$ corresponds to the symmetric part of the matrix $\boldsymbol{A}$.

One geometric interpretation of this definition that also holds in the extended case, would be that every vector $\boldsymbol{z}$ that was transformed with a positive definite matrix A forms an angle of less than 90°with the original vector.

6.1.2 Trace

The trace $tr(\boldsymbol{A})$ of a square matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is defined as the sum of its diagonal elements.

$$tr(\boldsymbol{A}) = \sum_{i=1}^{n} A_{ii} \tag{6.1}$$

with A_{ij} denoting the ith row and jth column of $\boldsymbol{A}$. The trace of a diagonalizable matrix is equivalent to the sum of its eigenvalues:

$$tr(\boldsymbol{A}) = \sum_{i=1}^{n} \lambda_i \tag{6.2}$$

A square matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is diagonalizable iff $\exists \boldsymbol{S}, \boldsymbol{S^{-1}} \in \mathbb{R}^{n \times n}$ such that $\boldsymbol{S^{-1}AS}$ is a diagonal matrix. This allows us to decompose $\boldsymbol{A}$ as

$$\boldsymbol{A} = \boldsymbol{SDS^{-1}} \tag{6.3}$$

where $\boldsymbol{D}$ is a diagonal matrix whose diagonal entries are the eigenvalues λ_i of $\boldsymbol{A}$. We can easily verify equation 6.2 by using the cyclic property of the trace:

$$tr(\boldsymbol{A}) = tr(\boldsymbol{SDS^{-1}}) = tr(\boldsymbol{DS^{-1}S}) = tr(\boldsymbol{D}) = \sum_{i=1}^{n} \lambda_i \tag{6.4}$$

This decomposition requires the dimension of the eigenspace of $\boldsymbol{A}$ to be equal to n, which in our analysis with random matrices will almost always be the case.

Since the trace can be written as the sum of all eigenvalues it follows, that a positive definite matrix will always have a positive trace since all eigenvalues are positive, and therefore a negative trace tells us that the matrix can not be positive definite.

6.1.3 Alignment of angles

Lillicrap et al. (2016) introduce a so called modulator vector:

$$\boldsymbol{\delta}_{bp} = \boldsymbol{\theta}^T \boldsymbol{e} \tag{6.5}$$

which is a weighted sum of the error signal with the readout weights which is used in the weight update rule (see section 2.1.2). Since in the case of feedback alignment, the modulator vector is replaced by $\delta_{ba} = \boldsymbol{Be}$ (see Lillicrap et al. (2016)) we can look at the inner product between these two vectors which gives a measure of the angle between them. A positive value will mean that the vectors are within 90° of each other, 0 means the vectors are exactly orthogonal and a negative value means an angle of more than 90°.

$$(\boldsymbol{\theta}^T \boldsymbol{e})^T (\boldsymbol{Be}) = \boldsymbol{e}^T \boldsymbol{\theta B e} \tag{6.6}$$

This inner product is a quadratic form whose sign will solely be determined by the matrix $\boldsymbol{\theta B}$, independent from the error vector $\boldsymbol{e}$. As described before in the definition of positive definiteness of a matrix, this product will only be positive if all eigenvalues of $\boldsymbol{\theta B}$ are positive. It is therefore a necessary condition that the trace of $\boldsymbol{\theta B}$ is also positive. A positive trace can therefore be an easily computable indicator that the modulator signals are within 90° to each other.

As was explained in section 1.5.1, it is sufficient that the direction of the weight update is within 90° of the negative gradient to improve on the previous loss and in a convex loss function, with a properly chosen learning rate, eventually achieve convergence. Therefore if we observe that the modulator vectors during the course of training form an angle that is less than 90°, this would give an explanation why random e-prop can work.

6.2 Results

6.2.1 Pattern generation

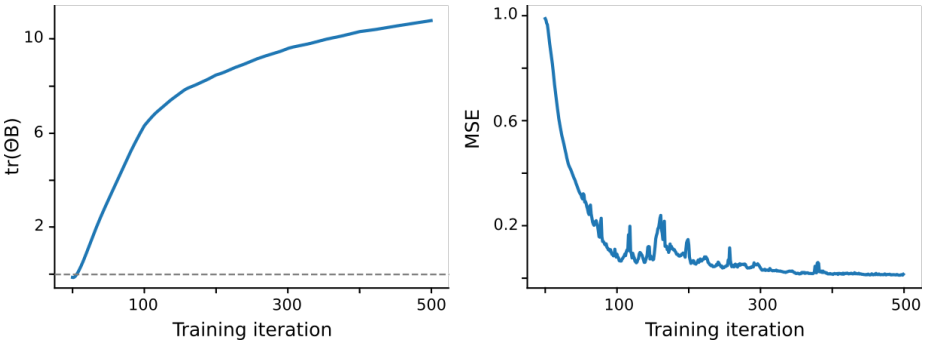

Figure 6.1: (left) Alignment of angles between readout and feedback weights. (right) Loss over the course of training.

Figure 6.1, supports the idea of angle alignment. Before training, the trace of $\boldsymbol{\theta B}$ has a value close to 0, which is expected for two randomly initialized matrices. However during the course of training, the angles align, which is visible in an ever increasing trace value. Interestingly this alignment of angles is then also highly correlated with the development of the loss.

6.2.2 Store recall

We also wanted to analyze the dynamics of this weight alignment, by tracking the development of the readout weights during the training process.

For this analysis, the so called store-recall task is used, which in its basic setup is very similar to the evidence accumulation task. 10 LIF and 10 ALIF neurons are used. Figure 6.2 shows the input encoding for the task. The network receives a random input from two channels, along with a store and a recall command. The task is to remember which input channel was active during the store command and then output this value during the recall command. The delay between the store and recall command is uniformly sampled from $200 * \mathcal{U}(1, 6)$ ms.

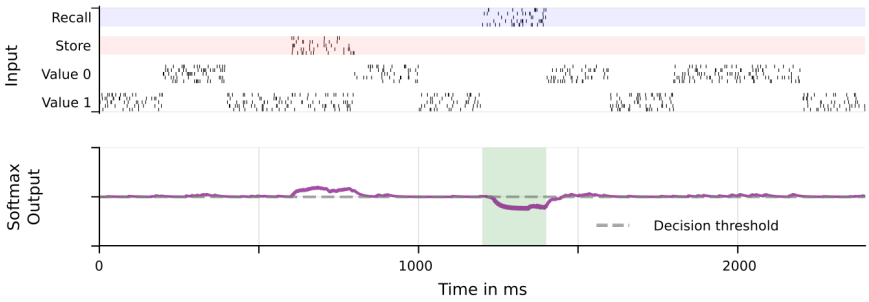

Figure 6.2: Input and output of the store recall task.

We will now analyze the mechanism by which the readout weights align with the feedback. Since the update term for the readout weights does not depend directly on the feedback weights (see equation 4.33), it helps to look at the firing activities of the network neurons, which will ultimately determine the readout weights.

This is done by computing the firing frequency for each neuron during the time a store command comes and value 0 or value 1 is stored. These frequencies are denoted as f_{store0} and f_{store1}. The difference between these frequencies will give a measure of the preference for a neuron to storing either 0 or 1.

In the left panel of figure 6.3 we can see, that the difference of the two feedback weights that connect to each neuron, determine the preference of a neuron. For this figure an ensemble of trained networks was used. Before training, there is no preference to either input. After training, the neurons exhibit a strong preference to either input 0 or input 1. In the right panel the readout weights are then plotted as a function of the preferences. We can see a negative alignment, that is caused by the fact, that when an adaptive neuron fires a lot during a store command, its threshold will get raised and during the recall when the outputs are read out, it will not fire anymore.

Figure 6.4 shows that the network neurons learn to exhibit a preference to either input 0 or input 1 during the store command which is shown on the left panel. In the right panel the development of the readout weights is shown. We can see, that the readout weights follow the preference of each neuron.

In figure 6.5 we see, that this tuning is produced by a strong input weight to the store input and an inhibitory connection to the opposite input, which will make a neuron fire either during store and input 0 or store and input 1. Additionally most ALIF neurons have a strong positive weight to the recall input, which will make them fire during the recall period if their threshold was not increased in the past during the store period.

The assignment of these roles, seems to depend in general mostly on the difference between the feedback weights $B_{j,value0}$ and $B^{j,value1}$. The initial tuning of a neuron does not seem to play a role, since there are many crossings of the zero line as can be seen in figure 6.4. This will be also confirmed when analyzing

the evidence accumulation task.

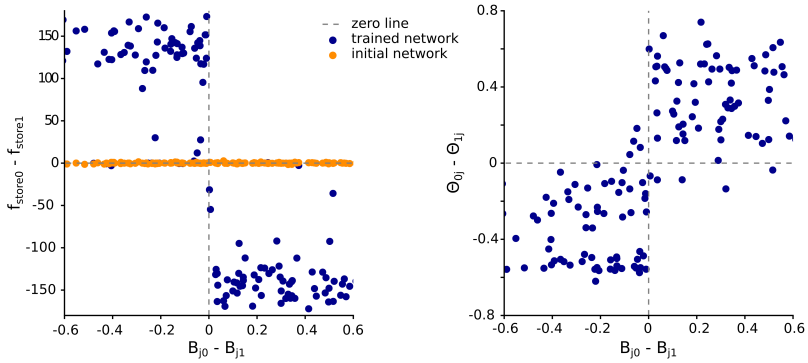

Figure 6.3: Left panel: firing preferences of neurons during store command as a function of the feedback weight difference. Right panel: Readout weight difference as a function of feedback weight difference.

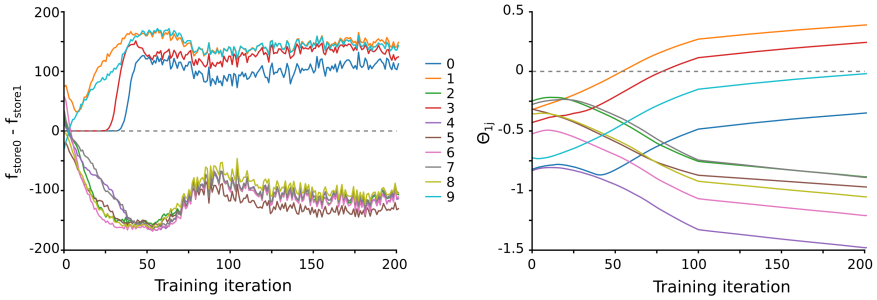

Figure 6.4: Development of readout weights and neuron tuning over the course of the training.

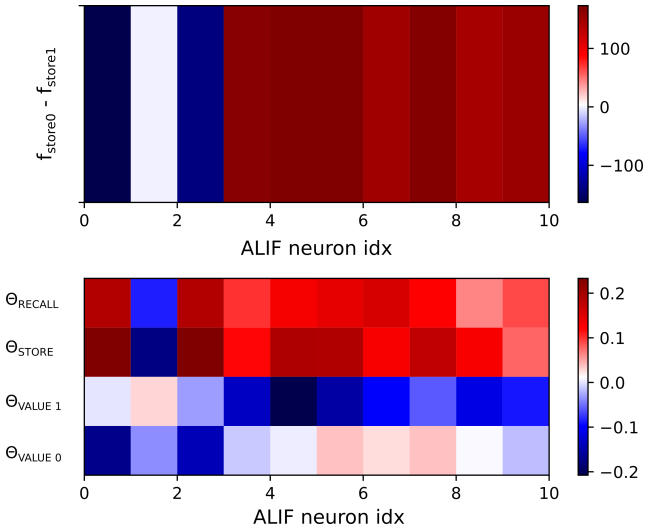

Figure 6.5: Firing rates differences for value 0 and value 1 during the store command. The network produces the tuning by a strong positive weight on the store input which is then inhibited by a negative connection to the opposite value.

6.2.3 Evidence accumulation task

Also in the evidence accumulation task an alignment of features happens during the training process. This is shown in figure 6.6. The orange curve plots the alignment of the feedback weights with the readout weights, while the blue curve shows the alignment of the neuron tuning with the feedback weights. We can see that the alignment of the tuning happens first, and then after this the alignment of the readout weights follows.

In figure 6.7 we can see, that the tuning of an ALIF neuron to either the right or left channel, correlates very strongly with the difference between the feedback weights $B_{j,left}$ and $B_{j,right}$.

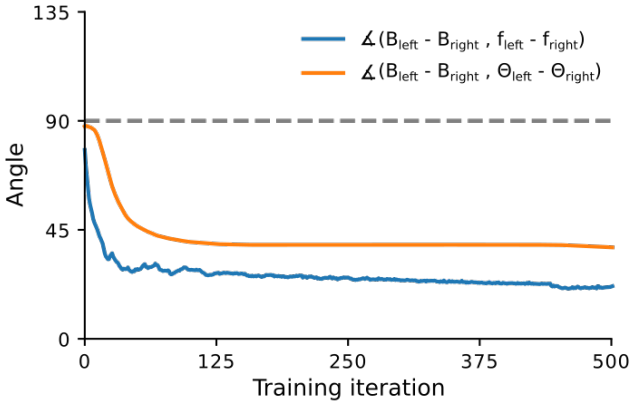

Figure 6.6: Alignment of the features ($f_{left} - f_{right}$) and angles of readout and feedback matrix.

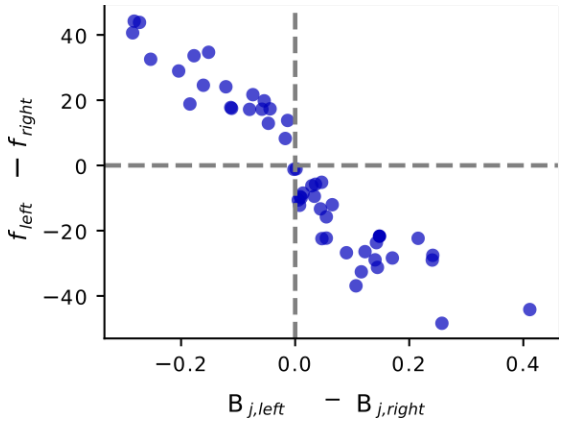

Figure 6.7: This figure shows the preference of the adaptive neurons for either the left input or the right input, in dependence of the feedback matrix difference.

6.2.4 Discussion

In all of the analyzed tasks, the readout weights were aligning with the feedback weights during the course of training. This alignment happened, because the readout weights were adjusting to make best use of the features, which were first produced with help of the error information coming through the feedback weights.

This seems to match with the analysis of Lillicrap et al. (2016), where it was found, that the information from the feedback weight first flows back to the hidden layer and from there into the readout weights.

Chapter 7

Conclusion

7.1 Discussion

In this work we evaluated a novel learning algorithm, e-prop, on the basis of a simple spiking neuron model, as defined in section 1.6.

We showed that e-prop can perform very similarly to backpropagation through time, on a range of tasks that demand challenging temporal credit assignment and memory.

We also showed, that e-prop can be modified in a way such that it uses an asymmetric feedback path for error signals. We found that similar as in the experiments of Lillicrap et al. (2016), the readout weights aligned their angle over time with the randomly chosen feedback weights. The asymmetric error feedback path and the synapse specific update dynamics could be seen as an argument in favor of the higher biological plausibility of e-prop as compared to BPTT.

Since e-prop is an online algorithm, it does not involve storage of network inputs and states for the course of the whole sequence. The memory requirements of e-prop are of the same order as simply simulating the network and there are no requirements for additional offline processing. This in principle allows the algorithm to be implemented on neuromorphic hardware.

7.2 Further research

At the moment it is not clear by how much and on which tasks the performance of e-prop is limited, or might even break down. The experiments done in this thesis only cover a very small fraction of the space of tasks that a recurrent spiking neural network could in principle solve.

Therefore, to obtain a better understanding of the power of e-prop, more experiments and a more diverse set of benchmark tasks are needed. In cases where e-prop performs worse than BPTT, it might be interesting to see, if and

how much this could be improved with the introduction of learning to learn, or synthetic gradients like it is done in (Bellec et al. (2019a)).

It would also be valuable to have e-prop implemented on neuromorphic hardware, such as Loihi Davies et al. (2018) and be deployed at a task that makes use of a large enough network, that could not be trained with BPTT anymore.

List of Figures

1.1 Graph of a one layer feedforward artificial neural network. 9
1.2 Sketch of a simple recurrent neural network. 9
1.3 Unrolling of the computational graph for a recurrent neural network. Figure adapted from Goodfellow et al. (2016) 12
1.4 Time course of the membrane voltage for 500 ms with a constant input current of 0.03 for the first 300 seconds. The threshold voltage is set to $v_{th} = 0.5$ and the decay constant α is set to $\exp(-\frac{1}{20})$ 15
1.5 Time course of the membrane voltage and adaptive threshold for 500 ms with a constant input current of 0.03 for the first 300 seconds. The threshold voltage is set to $v_{th} = 0.5$ and the decay constant α is set to $\exp(-\frac{1}{20})$. The threshold adaptation strength β is set to 0.2 and the adaptation decay factor ρ is set to $\exp(-\frac{1}{200})$ 16
1.6 Pseudo derivative used to deal with the discontinuity of the spike function for a threshold voltage of $v_{thr} = 0.5$ and dampening factor $\gamma = 0.3$ 18

5.1 Spike raster and network outputs for the pattern generation task that was trained with e-prop1. (from (Bellec et al., 2019a)) . . . 31
5.2 Averaged mse over training iterations for different variants of random e-prop.(from (Bellec et al., 2019a)) 33
5.3 Bar plot of final performance for the pattern generation task with different training methods.(from (Bellec et al., 2019a)) 33
5.4 Spike raster for the movie replay task that was trained with *random e-prop*. The bottom two rows show the learning signals for 10 randomly chosen neurons. After training, the magnitude of the learning signals is very small, an indication of a low error. . 34
5.5 Performance curves for different varieties of e-prop and BPTT. . 35
5.6 Sketch of the evidence accumulation task. During the evidence period, the network has to keep track of the number of cues on the left and right side, remember these values during the delay period and finally make the right decision. 36

5.7 Spike raster for the click count task that was trained with *random e-prop*. The top row shows the four input channels consisting of 10 neurons each, which are encoding the two input groups (left and right group), the recall cue and the background noise. The row below the input is the spike raster of the network. 50 recurrent and 50 adaptive neurons were used for the task. The networks output is shown below the spike raster. The output is given as the softmax over two output neurons. Below this, the learning signals for 10 selected neurons are shown. The second to last row shows the eligibility trace for 3 selected synapses. The last row shows the slow component of the eligibility trace $\epsilon_{ji,a}$ for the same synapses. 37
5.8 Performance plot for the supervised setup. Random e-prop solved the task on average after 308 iterations while with BPTT it was 156. 39
5.9 Performance plot for the reinforcement setting. 39
5.10 Percentage of choices of the network trained with e-prop1 for the red group as a function of the number of red cues. In the easiest cases where there is a red:blue split of 0:7, 6:1 or 5:2, the network has an accuracy of almost 100% which drops down to around 70% for the harder cases of a 3:4 split. 40

6.1 (left) Alignment of angles between readout and feedback weights. (right) Loss over the course of training. 43
6.2 Input and output of the store recall task. 44
6.3 Left panel: firing preferences of neurons during store command as a function of the feedback weight difference. Right panel: Readout weight difference as a function of feedback weight difference. 45
6.4 Development of readout weights and neuron tuning over the course of the training. 45
6.5 Firing rates differences for value 0 and value 1 during the store command. The network produces the tuning by a strong positive weight on the store input which is then inhibited by a negative connection to the opposite value. 46
6.6 Alignment of the features ($f_{left} - f_{right}$) and angles of readout and feedback matrix. 47
6.7 This figure shows the preference of the adaptive neurons for either the left input or the right input, in dependence of the feedback matrix difference. 47

List of Tables

5.1 Final performance comparison after training. 36
5.2 Average number of iterations needed until the task was solved. . 39

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Kaiser, L., Kudlur, M., Levenberg, J., and Zheng, X. (2015). Tensorflow : Large-scale machine learning on heterogeneous distributed systems.

Akrout, M., Wilson, C., Humphreys, P. C., Lillicrap, T., and Tweed, D. (2019). Deep Learning without Weight Transport. *arXiv:1904.05391 [cs, stat]*. arXiv: 1904.05391.

Bear, M. F., Conners, B. W., and Paradiso, M. A. (2016). *Neuroscience: Exploring the Brain - Fourth edition*. Wolters Kluwer.

Bellec, G., Kappel, D., Maass, W., and Legenstein, R. (2018a). Deep rewiring: Training very sparse deep networks. *International Conference for Learning Representations*.

Bellec, G., Salaj, D., Subramoney, A., Legenstein, R., and Maass, W. (2018b). Long short-term memory and learning-to-learn in networks of spiking neurons. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 787–797. Curran Associates, Inc.

Bellec, G., Scherr, F., Hajek, E., Salaj, D., Legenstein, R., and Maass, W. (2019a). Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets. *arXiv preprint arXiv:1901.09049*.

Bellec, G., Scherr, F., Subramoney, A., Hajek, E., Salaj, D., Legenstein, R., and Maass, W. (2019b). A solution to the learning dilemma for recurrent networks of spiking neurons. *bioRxiv*.

Caporale, N. and Dan, Y. (2008). Spike timing–dependent plasticity: A hebbian learning rule. *Annual Review of Neuroscience*, 31(1):25–46. PMID: 18275283.

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., Dimou, G., Joshi, P., Imam, N., Jain, S., et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99.

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Ha, G. E. and Cheong, E. (2017). Spike frequency adaptation in neurons of the central nervous system. *Experimental neurobiology*, 26(4):179–185.

Hebb, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. Wiley.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Legenstein, R., Naeger, C., and Maass, W. (2005). What can a neuron learn with spike-timing-dependent plasticity? *Neural computation*, 17:2337–82.

Lillicrap, T. P., Cownden, D., Tweed, D. B., and Akerman, C. J. (2016). Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7.

Marszałek, M., Laptev, I., and Schmid, C. (2009). Actions in context.

Morcos, A. and Harvey, C. (2016). History-dependent variability in population dynamics during evidence accumulation in cortex. *Nature Neuroscience*.

Murray, J. M. (2018). Local online learning in recurrent networks with random feedback. *bioRxiv*.

Nicola, W. and Clopath, C. (2017). Supervised learning in spiking neural networks with force training. *Nature Communications*, 8(1):2208–2208.

Nøkland, A. (2016). Direct feedback alignment provides learning in deep neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 1045–1053, USA. Curran Associates Inc.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.

Roth, C., Kanitscheider, I., and Fiete, I. (2019). Kernel RNN learning (keRNL). In *International Conference on Learning Representations*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

Samadi, A., Lillicrap, T. P., and Tweed, D. B. (2017). Deep learning with dynamic spiking neurons and fixed feedback weights. *Neural Computation*, 29(3):578–602. PMID: 28095195.

Schemmel, J., Briiderle, D., Griibl, A., Hock, M., Meier, K., and Millner, S. (2010). A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Circuits and systems (ISCAS), proceedings of 2010 IEEE international symposium on*, pages 1947–1950. IEEE.

Sussillo, D. and F Abbott, L. (2009). Generating coherent patterns of activity from chaotic neural networks. *Neuron*, 63:544–57.

Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Tallec, C. and Ollivier, Y. (2017). Unbiased online recurrent optimization. *CoRR*, abs/1702.05043.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280.