Roland Renner, BSc

Training spiking neural networks to perform elementary computational operations

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Assoc.Prof. Dipl.-Ing. Dr.techn. Robert Legenstein

Institute of Theoretical Computer Science
Head: Assoc.Prof. Dipl.-Ing. Dr.techn. Robert Legenstein

Graz, March 2020

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

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

Zusammenfassung

Gepulste neuronale Netzwerke sind eine energieeffiziente Alternative zu künstlichen Netzwerken. Ein Modell dieser gepulsten neuronalen Netzwerke sind die Long Short-term Memory Spiking Neural Networks (LSNN). Ein LSNN ist eine Weiterentwicklung von einem Recurrent Network of Spiking Neurons (RSNN). In einem LSNN wird zusätzlich neuronale Adaptivität modelliert und dieser zusätzliche Prozess steigert die Rechenleistung erheblich.

In der folgenden Arbeit wird untersucht ob RSNNs auf elementare rechnerische Operationen trainiert werden können und ob im weiter Verlauf diese trainierten Netzwerke als Komponente in einem größeren Netzwerk inkludiert werden können. Als elementare Operationen wurden die Softmax Funktion und die K-winner-takes-all Funktion ausgewählt. Zuerst wurde gezeigt, dass RSNNs diese Funktion mit einer hohen Genauigkeit erlernen können. Danach wurde eine Methode vorgestellt diese vortrainierten Netzwerke in eine LSNN einzubinden und danach das LSNN zu trainieren. Diese Methode wurde erfolgreich auf einen Memory-Task eine temporale Mustererkennungsaufgabe angewandt.

Abstract

Spiking neural networks have been proposed as a promising energy-efficient alternative to artificial neural networks. One type of spiking neural network is the Long short-term memory spiking neural network (LSNN). An LSNN is a recurrent network of spiking neurons (RSNN) equipped with a neural adaptation process that enhances its computational power significantly.

Usually, LSNNs are trained from scratch, i.e., networks are largely unstructured before training. Here, we investigated whether we can train an RSNN to perform elementary computational operations and whether we could use such pre-trained networks as one component in a larger network structure. As example elementary operations, we chose the softmax function and the K-winner-takes-all function. We show that RSNNs can be trained to perform these computational operations with high accuracy. We further show that such pre-trained networks can be used as modules in LSNNs that are trained to perform a memory task or a temporal pattern recognition task.

Contents

List of Figures

1 Introduction

Artificial neural networks are generally brain-inspired, but there are significant differences. The most obvious difference lies in the way the neurons communicate information with each other. Artificial neurons send and receive continuous values, whereas biological neurons work on a time scale and communicate witch trains of action potentials. These action potentials are also known as spikes because they are short electric impulses. They transfer information in their timing and frequency. Furthermore, the spikes are sparse in time. Therefore a single spike contains high information content and allows the brain to function with very little energy. Spiking neural networks aim to emulate these abilities by mimicking the use of spikes. Therefore spiking neural networks show great potential in both computational power and energy efficiency (Tavanaei et al., 2019).

However, simulations of spiking neural networks on classical computation hardware (von Neumann architecture) are inefficient, because the asynchronous nature of the spiking neural network stands in direct contrast to the sequential and central data processing of von Neumann architecture. Neuromorphic hardware aims at solving this problem. It mimics the structure of neurons and synapses, which allows parallel processing and locality of the data. Additionally, the use of spikes to propagate information through the network creates energy-efficient systems (Pfeiffer and Pfeil, 2018).

A concrete model for a spiking neural network is the Long short-term memory Spiking neural networks (LSNN). Bellec, Salaj, et al., 2018 have equipped a recurrent spiking neural network (RSNN) with neural adaptation to develop LSNN. This extension enhances the learning capabilities of LSNNs greatly compared to RSNNs (Bellec, Salaj, et al., 2018).

In the case of LSNNs, readout neurons generate a non-spiking output. The network uses these outputs in a loss-function in combination with a softmax (Bellec, Salaj, et al., 2018). The softmax function is a normalization function, which scales a discrete input into probability space. LSNN uses the softmax function to normalize the activation of the readout neurons.

Our goal is to replace the softmax function with a spiking neural network and move one step closer to a fully spike-based network. Therefore we train an RSNN with Backpropagation through time (BPTT) to perform the calculation of a softmax function. To then test the usefulness of this trained softmax network, we replace a softmax function in an LSNN with the softmax network and train the LSNN for the given task. To further exhibit the general applicability, we trained an RSNN with BPTT to perform the K-winner-takes-all function. The K-winner-takes-all function only actuates the outputs corresponding to the K larges input values and the other outputs become zero. Neural networks use this function to implement decision making based on network activation.

In the following, we present first the background to RSNNs, LSNNs and BPTT. Then we give a short outlook to other approaches for spiking neural networks. Afterwards we describe the approach for the training of the softmax and the results of the training. We then insert a pre-trained softmax network in a working memory task and a classification task on the sequential MNIST dataset. In the last part, we describe our approach for training the K-winner-takes-all selection and report the achieved results.

2 Background

2.1 Spiking neurons

Spiking neurons are strongly inspired by biological neurons, as they also use spikes to communicate information to each other. In biological neurons, spikes are brief electric impulses. Spiking neurons receive spikes as input, which translates to an input current. This input current changes the membrane potential of the neuron. If the membrane potential reaches the threshold of the neuron, the neuron generates an action potential (spike). After the neuron generated a spike, the membrane potential resets to the reset potential. Then the neuron enters a refractory period, where it can not spike for a given time interval. The connections between neurons are called synapses. Synapses only work in one direction, where the post-synaptic neuron receives the spikes of a pre-synaptic neuron as input.

We use the leaky integrate and fire (LIF) neuron model as the theoretical basis to describe the membrane potential. The LIF neuon model describes the membrane potential of a neuron as the voltage across an RC circuit with a battery for the resting potential. The resistor and the capacitor are placed parallel to each other (Gerstner et al., 2014). This leads to the following differential equation

$$I(t) = \frac{u(t) - u_{\text{rest}}}{R} + C\frac{du(t)}{dt}, \tag{2.1}$$

where $I(t)$ in the input current, $u(t)$ is the membrane potential and u_{rest} is the voltage from the battery. Furthermore, R is the resistor coefficient and C is the capacitor coefficient. The first term models the current through the resistor and the second term models the capacitive current. We reformulate

the equation into a differential equation which describes the change in the membrane potential as

$$\tau_m \frac{du(t)}{dt} = -(u(t) - u_{\text{rest}}) + RI(t), \tag{2.2}$$

where $\tau_m = RC$ is the membrane time constant.

2.2 RSNN model

An RSNN consists of a population of LIF neurons, which are recurrently connected and receive input from input neurons. We use weights to scale the synapses between the neurons. These synaptic weights are the trainable parameters of the network.

We simulate the RSNN in discrete time with a timestep δt. We model the spike trains as binary sequences $z_i^{in}, z_j^{rec} \in \{0, \frac{1}{\delta t}\}$, where z_i^{in} is the spike train generated by the ith input neuron and z_j^{rec} is the spike train generated by the jth neuron of the RSNN population.

We assume the input current of the neuron is constant in between timesteps. We solve the equation 2.2 by integrating over it and assuming $I(t)$ is a constant input current I_c and $u_{\text{rest}} = 0$. This yields the following equation

$$u(\delta t) = \exp\left(-\frac{\delta t}{\tau_m}\right) u_{\text{init}} + \left(1 - \exp\left(-\frac{\delta t}{\tau_m}\right)\right) RI_c, \tag{2.3}$$

for the membrane potential $u(\delta t)$ for the time period of zero to δt. u_{init} is the initial condition.

We use equation 2.3 to model a discrete time step from time t to time $t + \delta t$ of the membrane potential $u_j(t + \delta t)$ for the jth neuron in our population. We use the membrane potential at time t $u_j(t)$ as the initial condition and the input at time $I_j(t)$ as the constant current. If a spike occured at time t

we reset the membrane potential. Therefore we subbract the term $\gamma z_j^{rec}(t)\delta t$, where γ is the threshold and $z_j^{rec}(t)$ becomes $\frac{1}{\delta t}$ if a spike occurres at time t. This yields the following equation

$$u_j(t+\delta t) = \exp\left(-\frac{\delta t}{\tau_m}\right)u_j(t) + \left(1-\exp\left(-\frac{\delta t}{\tau_m}\right)\right)RI_j(t) - \gamma z_j^{rec}(t)\delta t. \quad (2.4)$$

If the membrane potential of a neuron crosses the threshold and this neuron is not in the refactory period, a spike in the output spike train z_j^{rec} of this neuron j occurs. After the neuron emitted a spike, it enters a refractory period with a duration of κ. In this period the neuron can not spike again. We model this with

$$z_j^{rec}(t) = \begin{cases} \frac{1}{\delta t} & \text{if } u_j(t) \geq \gamma \text{ and } \sum_{k=1}^{\kappa} z_j^{rec}(t-k) = 0 \\ 0 & \text{else} \end{cases}. \quad (2.5)$$

The network consists of a population of recurrently connected neurons of size N_{rec}. Additionally, the recurrently connected neurons receive input from N_{in} input neurons. Therefore, the input current of a neuron $I_j(t)$ is the weighted sum of all incoming spikes at time t. $I_j(t)$ reads the following

$$I_j(t) = \sum_{i \in N_{in}} W_{ji}^{in} z_i^{in}(t-d) + \sum_{\substack{i \in N_{rec} \\ i \neq j}} W_{ji}^{rec} z_i^{rec}(t-d), \quad (2.6)$$

where $\mathbf{W}^{in}$ are the synaptic weights that scale the input from the input neurons, $\mathbf{W}^{rec}$ are the synaptic weights that scale the input from recurrent neurons and d is the synaptic delay.

2.3 LSNN

RSNNs do not achieve similar performances ranges as LSTM networks (Bellec, Salaj, et al., 2018). LSTM cells have three gates, which control the information flow of the hidden state of the cell. These gates allow the LSTM cells to store information. (Hochreiter and Schmidhuber, 1997). Therefore Bellec, Salaj, et al., 2018 developed the Long short-term memory Spiking

Neural Network (LSNN) model, which is an extension to the RSNN model by implementing neural adaptation. Neural adaptation is one of the various firing patterns shown by biological neurons. This pattern describes the desensitization of a neuron to a constant stimulus (Gerstner et al., 2014). Therefore neural adaptation equips the network additionally with a dynamic process, which works on a time scale in contrast to the static nature of the synaptic weights similar to the memory capabilities of LSTM cells.

Bellec, Salaj, et al., 2018 model the process of adaptation by increasing the firing threshold of a neuron by a fixed amount β after this neuron spikes. Otherwise, the threshold decays back to the baseline threshold b^0 with the time constant τ_a. The following equation describes this relationship.

$$\tau_a \frac{db_j(t)}{dt} = -(b_j(t) - b^0) + \beta z_j^{rec}(t)\delta t, \tag{2.7}$$

where b_j is the threshold of neuron j at time t. $z_j^{rec}(t)$ is the spike train of the neuron j at time t, which we use to determine if the neuron spiked. To obtain an equation for b_j for a discrete timestep δt, we integrate over the equation 2.7 from 0 to δt by assuming $z_j^{rec}(t)$ is a constant. Then we set the initial condition to $b_j(t)$ and we get

$$b_j(t+\delta t) = b^0 + \exp\left(-\frac{\delta t}{\tau_a}\right) b_j(t) + \left(1 - \exp\left(-\frac{\delta t}{\tau_a}\right)\right) \beta z_j^{rec}(t). \tag{2.8}$$

In the equation for the membrane potential, we model the threshold now with the function $b_j(t)$ instead of the constant γ, which changes equation 2.4 to

$$u_j(t+\delta t) = \exp\left(-\frac{\delta t}{\tau_m}\right) u_j(t) + \left(1 - \exp\left(-\frac{\delta t}{\tau_m}\right)\right) R I_j(t) - b_j(t) z_j^{rec}(t)\delta t \tag{2.9}$$

and the spike generation for a neurons j to

$$z_j^{rec}(t) = \begin{cases} \frac{1}{\delta t} & \text{if } u_j(t) \geq b_j(t) \text{ and } \sum_{k=1}^{\kappa} z_j^{rec}(t-k) = 0 \\ 0 & \text{else} \end{cases}. \tag{2.10}$$

2.4 BPTT

We use backpropagation through time (BPTT) to optimize the network. BPTT unrolls the recurrent network into a feedforward network and performs backpropagation on the unrolled feedforward network (Werbos, 1990). Figure 2.1 illustrates how different timesteps influence each other.

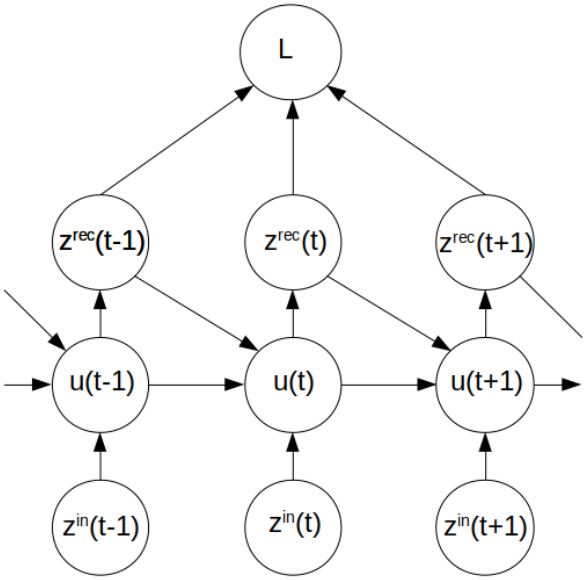

Figure 2.1: **BPTT graph** The graph shows the dependence of the variables of the network with time. u is the state of the membrane potentials, z^{rec} is the state of the output spike train of the recurrent neurons, z^{in} is the spike train of the input and L is the loss function.

Backpropagation is an algorithm which arises from the use of gradient descent on the network LeCun et al., 1988. Gradient descent is an optimization algorithm, which uses the gradient of a function to iteratively find parameters for a local minimum of the function. The gradient can be interpreted as the direction and rate of the fastest increase. Therefore we subtract the gradient to find parameters closer to a local minimum.

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \mu \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_k), \tag{2.11}$$

where $\boldsymbol{\theta}_k$ is the current or kth state of the parameter and $\boldsymbol{\theta}_{k+1}$ is the next or the $(k+1)$th state of the parameter. μ is the learning rate, which is a constant that scales the step size of the parameter update. $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_k)$ is the gradient of the function f and the kth state of the parameter $\boldsymbol{\theta}$. In our case, the function is the loss function of the neural network. The loss function measures the degree of error by the network and reaches its minimum if the network made no error. Therefore we use gradient descent to find parameters for the network that minimizes this function. As loss function, we used the mean squared error, see section 4.1.1 and section 4.2.1.

For our simulations, we used an extension of the gradient descent algorithm, which is called Adam (Kingma and Ba, 2014). Adam still a first-order iterative optimization algorithm. The algorithm uses two ways to enhance the speed of gradient descent. The first is momentum, which uses the past of the gradients to average out fast-changing gradients. The second is an adaptive learning rate, which computes an individual learning rate per parameter. It divides the learning rate by the root of a momentum term for the squared gradients. This division decreases the learning rate for big gradients and increases the learning rate for small gradients.

To use this algorithms the gradient of the loss function is needed. Therefore the various states of the network have to be differentiable, which the spike generation in equation 2.5 and equation 2.10 is not. To deal with this problem, we use the pseudo-derivative proposed in Bellec, Salaj, et al., 2018, for the RSNN.

$$\frac{dz_j(t)}{du_j(t)} = \nu \max\left\{0, 1 - \left|\frac{u_j(t) - \gamma}{\gamma}\right|\right\}, \tag{2.12}$$

where ν acts as a damping factor. The damping factor helps to achieve stable performances for large unrolled networks (Bellec, Salaj, et al., 2018). The pseudo derivative is a hat function of the normalized difference between the membrane potential $u_j(t)$ and the threshold γ. Therefore the derivative becomes ν at time t if a spike happened at time t. In the case of LSNN, the pseudo derivative uses the adapting threshold $b_j(t)$ at time t instead of the

constant threshold γ. This changes the pseudo derivative to

$$\frac{dz_j(t)}{du_j(t)} = \nu \max\left\{0, 1 - \left|\frac{u_j(t) - b_j(t)}{b_j(t)}\right|\right\}, \tag{2.13}$$

2.4.1 DEEP R

DEEP R is an algorithm for training neural networks with sparse connectivity in combination with BPTT. The algorithm maintains a certain connectivity level through training and also dynamically deactivates and activates connections (Bellec, Kappel, et al., 2017).

In the beginning, the algorithm fixes each neuron typ either to inhibitory or excitatory. Then DEEP R performs gradient descent updates on the weights of the active connections. DEEP R deactivates a connection if the connection changes from inhibitory to excitatory or vise versa during an update. Then the algorithm randomly chooses an inactive connection and activates it. To ensure that connectivity changes, DEEP R uses a noise term in the gradient update step to achieve high weights and sparse connections. Also, the algorithm uses an L1-regularization to enforce sparse connectivity. Instead of gradient descent, DEEP R also works in combination with Adam or other first-order iterative optimization algorithms.

3 Related Work

There are also other ways to deal with the problems of training a spiking neural network. The approaches range from different pseudo derivatives (Wu et al., 2018, Zenke and Ganguli, 2018, Shrestha and Orchard, 2018) over a modified description of the spike function (Huh and Sejnowski, 2018) to mapping the spiking neural network into an artificial neural network (Liu, Chen, and Furber, 2017, Jug et al., 2012).

Wu et al., 2018 take a similar approach as described in chaper 2. They also discretize the spiking neural network and train the network similar to BPTT. In their work, they additionally tested different pseudo derivatives for the spike generation. They used a rectangular function, a polynomial function, a sigmoid function and a Gaussian cumulative distribution function.

Zenke and Ganguli, 2018 use the spike response model to model a spiking neuron. They use the negative side of the fast sigmoid function to account for the nondifferentiability of the spike generation. For the derivative of the spike response model, they neglect the self kernel, which is the influence of the past membrane potential on the derivative. They do not use backpropagation to train the network. They use an error signal for the visible neurons and a feedback signal for the hidden neuron. For the feedback signal, they describe and tested three different ways. They weigh the error signal of influenced visible neurons either symmetric, random or unform for the feedback signal.

Shrestha and Orchard, 2018 also use the spike response model in their approach, but they take the self kernel into account and train the network in a backpropagation-like manner. Instead of the negative side of the fast

sigmoid function, they use the spike escape rate function as a pseudo derivative for the spike generation.

Instead of using a pseudo derivative, Huh and Sejnowski, 2018 uses a different model for the spike generation. They use a gate function, which works on the presynaptic membrane potential instead of working with a threshold. The gate function allows the synaptic current to be activated gradually, rather than in an abrupt and nondifferentiable manner. They use BPTT to train the neurons.

Jug et al., 2012 and Liu, Chen, and Furber, 2017 take a different approach to the previously discussed ones. Instead of training the spiking neural network, they convert the network in a shadow network. Then they train the shadow network like an artificial neural network. Jug et al., 2012 use Siegert neurons in the shadow network. They map the firing rates of the spiking neurons to the Siegert neuron. The firing rates of the spiking neurons are the input and output of the Siegert neuron. Therefore the Siegert neuron tries to model the response of the spiking neuron to the firing rates. Liu, Chen, and Furber, 2017 is an extension to this work. They use a different neuron function in the shadow network to better match the behavior of the spiking neural network. The disadvantage of this model is that the spiking neural network uses a firing-rate code by assumption, which is typically inefficient as many spike events are generated.

4 Results

In this chapter, we train an RSNN to implement the softmax function or the K-winner-takes-all function. We have the goal to replace these functions in a neural network setting, with an RSNN, which performs the same task. In the first section 4.1, we tested if we can train an RSNN to learn a softmax function and also a softmax with different linear mappings of input vector, see Figure 4.1.

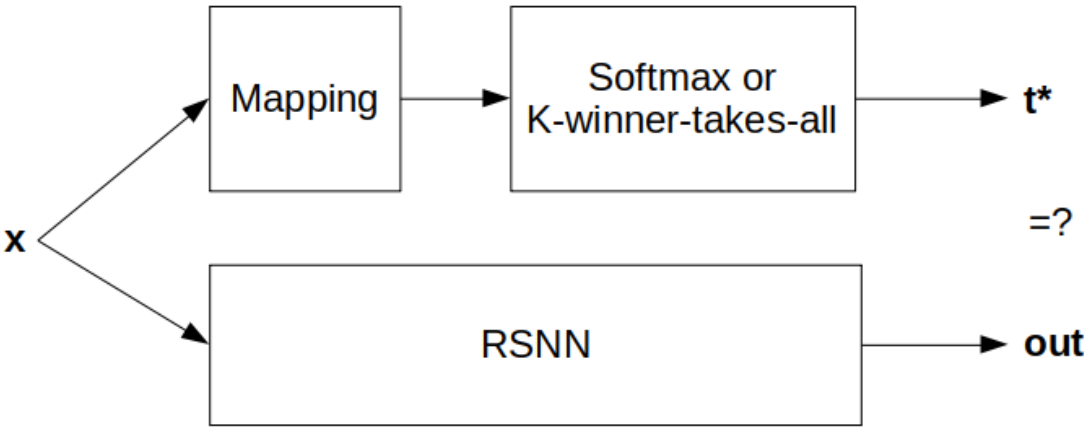

Figure 4.1: **Test setup** Given an input vector **x**, we want to train an RSNN to generate the same output as a softmax or K-winner-takes-all function with linear mapping in front. We denote the output vector of the RSNN with **out** and the target vector or the output vector of the function to train with $\mathbf{t}^*$

In the second part of chapter (section 4.2), we replaced a softmax with such an RSNN in two LSNNs and trained the LSNNs to perform their given task. The first LSNN performed a working memory task (Store-Recall). The second LSNN performed classification on the sequential MNIST dataset. In

the last section 4.3, we examine if we can train an RSNN to perform the functionallity of a K-winner-takes-all function.

4.1 Softmax

The softmax function is a normalization function, which takes N real numbers $x_1, \cdots, x_N$ as an input and normalizes them into a probability distribution

$$p_n(x_1, \cdots, x_N) = \frac{\exp(x_n)}{\sum_{j=1}^{N} \exp(x_j)}. \tag{4.1}$$

Therefore after applying the function, all componets sum up to one ($\sum_{n=1}^{N} p_n = 1$) and are in the intervall between zero and one ($p_n \in [0, 1]$).

4.1.1 RSNN setup

The network we used to learn the softmax tasks has a recurrent layer and an input layer. The recurrent layer consist of N output subpopulations and K hidden neurons, see Figure 4.2.

Input Layer

Every input m of the input layer in Figure 4.2 consists of a subpopulation of input neurons G_m^{in}. For our simulations, we draw the input samples $x_1, \cdots, x_N$ from an exponential distribution, see 4.1.3, 4.1.5 or 4.1.6. To achieve useful rates for an RSNN, we multiply every input sample x_n with a constant scale frequency ρ_{scale}. Then we model the spike train of a single neuron of every input subpopulation G_m^{in} with a Poisson process. Consider a spike train $z_{i,m}^{in}$ of a neuron i of the subpopulation G_m^{in} in discrete time with a time step of δt, where $z_{i,m}^{in}(t) \in \{0, \frac{1}{\delta t}\}$. If $z_{i,m}^{in}$ was generated by a Poisson process with rate $\rho_{\text{scale}} x_m$, it holds for all t:

$$P[z_{i,m}^{in}(t) = \frac{1}{\delta t} | \rho_{\text{scale}} x_m] = \rho_{\text{scale}} x_m \delta t. \tag{4.2}$$

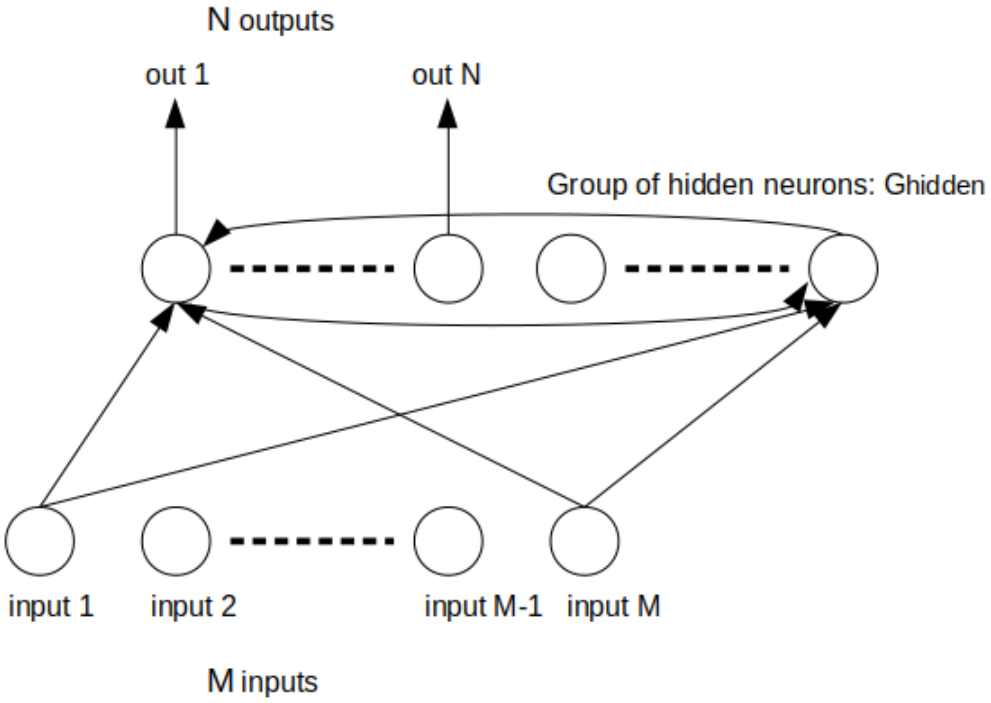

Figure 4.2: **RSNN structure** The abstract structure of the network we use for learning the softmax function.

We simulate the network in discrete time with 1 ms timesteps. Equation 4.2 with $\delta t = 1$ gives the probability for an input spike ($z^{in}_{i,m}(t) = 1, z^{in}_{i,m}(t) \in \{0, 1\}$) at some timestep t.

Recurrent Layer

The neurons of the recurrent layer are recurrently connected. The input layer is in a feed-forward way connected to the recurrent layer. We divide the recurrent layer into two population: the hidden population and the output population.

The output population consists of N subpopulations G^{out}_n. Each of these

N-subpopulation of the output population in Figure 4.2 consists of a group of LIF neurons.

The hidden population G_{hidden} consists of an additional pool of LIF neurons of size $|G_{\text{hidden}}|$ to improve the computational power of the network.

Network output

We denote the final output of the network as $\text{out}_1, \cdots, \text{out}_N$ with $\text{out}_n \in \mathbb{R}_0^+$. The nth output out_n is given by the average spike frequency of the nth subpopulation of the output neurons. By defining G_n^{out} as the set of indices of this population, we can write

$$\text{out}_n = \frac{1}{(T - T_{init})|G_n^{out}|} \sum_{j \in G_n^{out}} \sum_{t=T_{init}}^{T} z_j^{rec}(t). \tag{4.3}$$

We record the spikes of a neuron z_j^{rec}, sum the spikes up and divide them by the recorded time $(T - T_{init})$ in seconds to calculate the spiking frequency. We give the network an initialization time after we start the simulation, therefore the recorded time is the absolute simulation time T minus the initialization time T_{init}.

Training

We use back-propagation-through-time with Adam and a learning rate of 0.001. We use the mean squared error between the output value $\text{out}_{n,i}$ of the output instance n and sample i and their target values $t_{n,i}^*$ as the loss function, which results for one batch B_s in the following:

$$L_{\text{mse}} = \frac{1}{2|B_s|} \sum_{i \in B_s} \sum_{n=1}^{N} (\text{out}_{n,i} - t_{n,i}^*)^2. \tag{4.4}$$

We refer to a batch B_s as the sth subset of a dataset D ($B_s \subseteq D$). For training, we used a training set with 10000 random samples and a batch size $|B_s|$ of

50. We trained the network for 50 epochs. We used a validation set of 1000 random samples to identify the best performing parameters. After training, we used a test set of 1000 random samples to measure the final performance of the network.

General Parameters

We list all parameters of the network in Table 4.1, which where kept the same over all tasks. For the subpopulation of the input and output we used the same size for every subpopulation

$$\begin{aligned} |G_m^{in}| &= |G^{in}| \quad \text{for} \quad m = 1, \cdots, M \\ |G_n^{out}| &= |G^{out}| \quad \text{for} \quad n = 1, \cdots, N. \end{aligned} \tag{4.5}$$

simulation time T	300 ms
initialization time T_{init}	30 ms
input subpopulation size $\lvert G^{in}\rvert$	10
output subpopulation size $\lvert G^{out}\rvert$	10
ρ_{scale}	200 Hz
hidden population size $\lvert G_{\text{hidden}}\rvert$	50
timestep δt	1 ms
damping factor ν	0.3
threshold γ	0.5
synaptic delay d	1 ms
refactory period κ	1 ms
membrane time constant τ_m	20 ms

Table 4.1: General parameters over all tasks, for the softmax training

4.1.2 Tasks

We performed three tasks in sections 4.1.3,4.1.5 and 4.1.6 with different input sizes with the RSNN. In the first task section 4.1.3, we tested if we can train the RSNN to perform the softmax calculation. We summarized the results in Table 4.2. Additionally, we tested the effects of regularization on the hidden population for the first task in section 4.1.4. In the second task section 4.1.5, we asked if the RSNN can additionally learn a linear mapping applied to the input vector before the softmax is applied, see Table 4.4. In the third task section 4.1.6, we introduced a mapping also to represent negative values. Table 4.5 shows the results.

For the accuracy rate, we considered the output of a sample as misclassified if the arg max of the input and output differ. The reason is, that this is the most important task of a softmax in a neural network setting. Otherwise, the output becomes false because the function interprets the activation of the network false.

4.1.3 Task Softmax with exponentially distributed input samples

In the first task, we want to train the network to perform the softmax function. Therefore we draw samples from an exponential probability function to get input samples. The exponential distribution has a high probability for small values. We assume mostly, one activation is highly active for a well-performing network. Therefore we obtain samples, which closely mimic the activation of a neural network performing a classification task. In the following equation, we describe the drawing process for one input sample $\mathbf{x}$, which leads to

$$\mathbf{x} = (x_1, \cdots, x_M)^T \quad \text{with} \quad x_i \sim \text{EXP}_\beta. \tag{4.6}$$

The density function f_β of the exponential distribution EXP_β is given by equation 4.7 where we used $\beta = 0.5$

$$f_\beta(x) = \begin{cases} \beta e^{-\beta x} & x \geq 0 \\ 0 & x < 0 \end{cases}. \tag{4.7}$$

From these input samples, we calculate the softmax function and multiply them by ρ_{scale} to get the target rates

$$\mathbf{t}^* = (t_1^*, \cdots, t_N^*)^T \tag{4.8}$$

for the sample $\mathbf{x}$. This means that the input size M is the same as the output size N, which leads to

$$t_n^* = \rho_{\text{scale}} \frac{\exp(x_n)}{\sum_{j=1}^{M} \exp(x_j)}. \tag{4.9}$$

Results

We performed the task for an input size of 4, 8, 10, 12, 16, 24 and 32. We hypothesized that the performance of the network would drop for larger input sizes. Table 4.2 shows that this drop in performance occurred. Figures 4.3 and 4.5 show two samples with an input size of 4 and 10. Figures 4.4 and show the training loss over time and the final test error for the input size of 4 and 10.

input size M	output size N	accuracy
4	4	0.957
8	8	0.858
10	10	0.942
12	12	0.859
16	16	0.851
24	24	0.802
32	32	0.829

Table 4.2: Task Softmax with exponentially distributed input samples accuracy for different input output sizes.

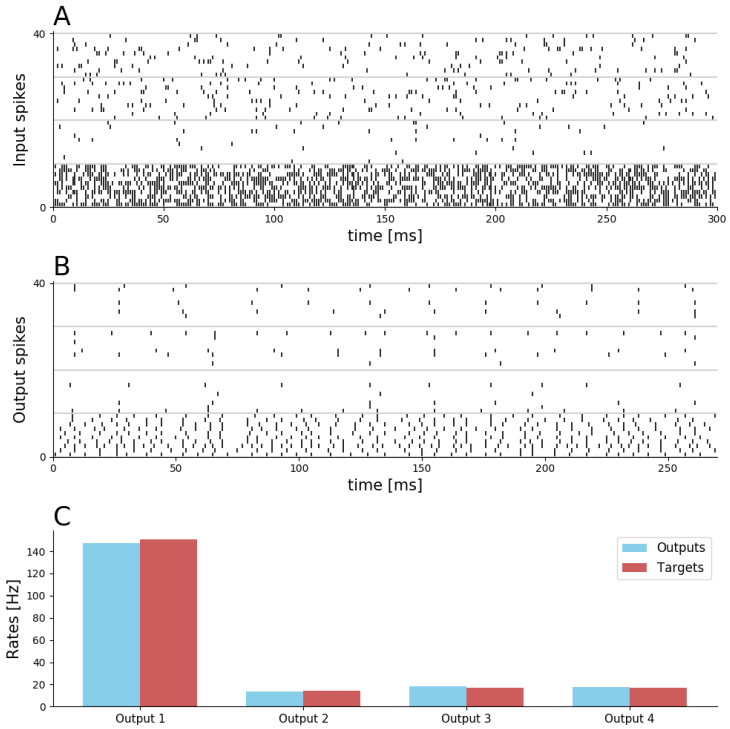

Figure 4.3: **Task Softmax with exponentially distributed 4 dimensional input sample A**) Spikes of the Poission processes which function as input to the network. **B**) Spiking activity of the output neurons. **C**) Comparison the averaged output rates to the target rates.

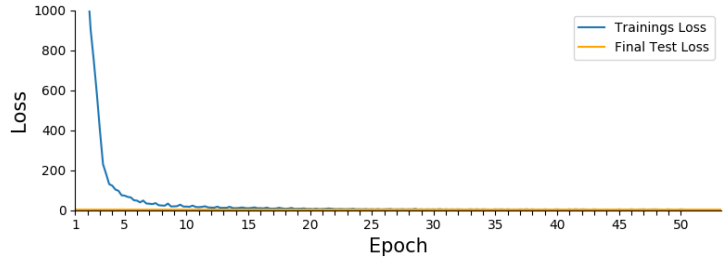

Figure 4.4: **Task Softmax with exponentially distributed 4 dimensional input loss over time** The plot shows the evolution of the training loss over time and the final test error.

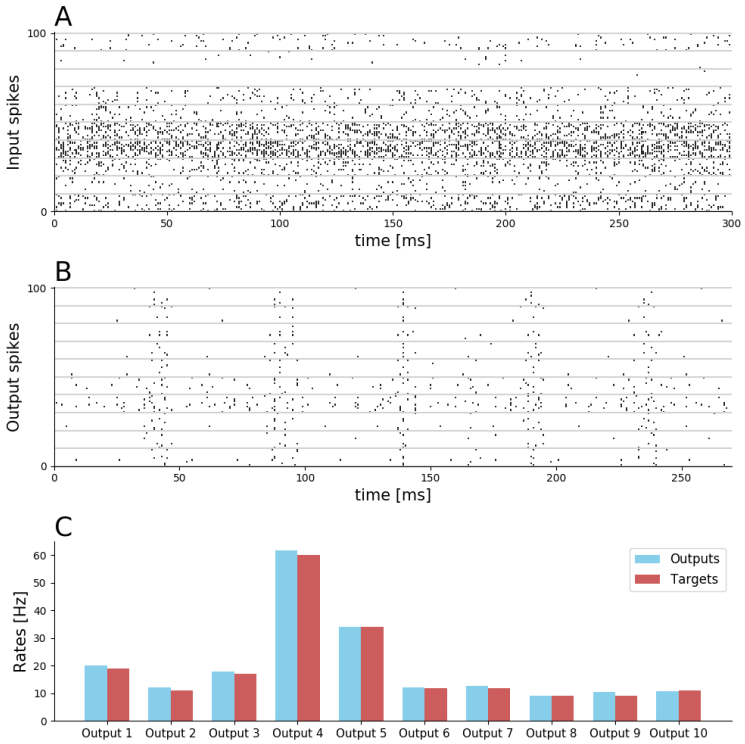

Figure 4.5: **Task Softmax with exponentially distributed 10 dimensional input sample** **A**) Spikes of the Poission processes which function as input to the network. **B**) Spiking activity of the output neurons. **C**) Comparison the averaged output rates to the target rates.

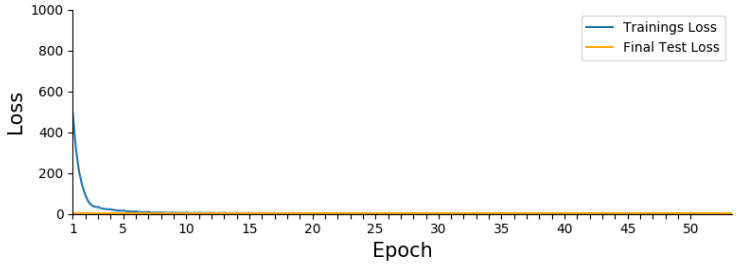

Figure 4.6: **Task Softmax with exponentially distributed 10 dimensional input loss over time** The plot shows the evolution of the training loss over time and the final test error.

4.1.4 Task Softmax task with firing rate regularization

In task 4.1.3, we performed our simulations without controlling the spiking frequencies of the neurons in the hidden population. Therefore these neurons had a high average spiking frequency, see Table 4.3. We want to achieve smaller spiking frequencies because this would allow to run these networks energy-efficiently on neuromorphic hardware.

We control the spiking frequencies of the neurons contributing to the output with the loss-function because these spiking frequencies are the direct output of the network. There are no constraints on the neurons in the hidden population G_{hidden}, in task 4.1.3. To change that, we add a regularization part to the loss function, which punishes differences from a chosen spiking frequency over the set of hidden neurons G_{hidden}

$$L_{\text{reg}} = \frac{\lambda}{|G_{\text{hidden}}|} \sum_{k \in G_{\text{hidden}}} (\rho^* - \rho_k)^2 \quad \text{with}$$
$$\rho_k = \frac{1}{T} \sum_{t=1}^{T} S_k(t). \tag{4.10}$$

λ determines the influence of the regularization, which is the mean-squared-error between the chosen frequency ρ^* and the averaged frequencies of the neurons in the hidden population ρ_k over the simulation time T.

The only difference to task 4.1.3 is that we add the regularization part to the loss-function.

Results

In Table 4.3, we compare the test loss and the average spiking frequency of neurons in the hidden population for different λs. The target frequency for the hidden neurons was 50 Hz.

λ	Test loss	misclassified	Average Spiking Frequency
0	2.011 ± 0.200	57.2 ± 3.867	133.872 ± 10.620
0.001	1.928 ± 0.314	48.6 ± 6.406	132.090 ± 11.105
0.01	1.976 ± 0.157	57.6 ± 4.586	129.950 ± 7.371
0.1	2.051 ± 0.113	54.2 ± 6.852	116.818 ± 8.829
1	2.168 ± 0.289	53.6 ± 3.136	81.773 ± 3.754
10	2.121 ± 0.212	61 ± 3.949	53.679 ± 2.123
20	2.004 ± 0.162	53.8 ± 2.785	52.673 ± 1.614
50	1.960 ± 0.060	53.6 ± 8.138	49.773 ± 0.723
100	1.919 ± 0.194	55.4 ± 7.657	50.443 ± 0.378
200	2.082 ± 0.086	58.0 ± 6.260	50.340 ± 1.168

Table 4.3: Task Softmax task with firing rate regularization averaged over five runs

The regularization part does not interfere with the performance of the network, but controls the average spiking frequency of the neuron in the hidden population, see Table 4.3. Although we averaged the values over five runs, there is a small variance in the test loss and the misclassified samples. This is because we draw a new test and training set with every simulation.

4.1.5 Task softmax with a random linear mapping

For the this task, the network has to learn in addition a random linear mapping applied to the input prior to the softmax function. Usually, a softmax occurs with a weight matrix used to combine the input. Therefore, we want to examine if we can train an RSNN to learn the weight matrix additionally. For this task, the input size M is different from the output size N ($M > N$). We draw an input sample $\mathbf{x}$ form an exponential probability function EXP_β with a density function f_β in Equation 4.7 ($\beta = 0.5$)

$$\mathbf{x} = (x_1, \cdots, x_M)^T \quad \text{with} \quad x_i \sim \text{EXP}_\beta. \tag{4.11}$$

At the next step, we draw a random weight matrix $\mathbf{W_{target}}$ of size $(N \times M)$ with each component from a Gaussian distribution with mean of 20 and a standard deviation of 5. We first calculate the vector $\mathbf{a}$ and then compute

the softmax of this vector as in the previous task, to calculate the final target rates $\mathbf{t}^* = (t_1^*, \cdots, t_N^*)^T$ with

$$\begin{aligned} \mathbf{a} &= \mathbf{W}_{\mathbf{target}} * \mathbf{x} \\ \mathrm{t}_n^* &= \rho_{\mathrm{scale}} \frac{\exp(a_n)}{\sum_{j=1}^{N} \exp(a_j)}. \end{aligned} \tag{4.12}$$

Results

We performed the task for input size and output size combinations of (8,4), (14,10), (16,12), (16,12), (24,16) and (32,24). Table 4.4 shows that a drop in performance for larger input und output sizes occured. Figures 4.8 and 4.10 show two samples with an input size and output size combination of (4,8) and (14,10). Figures 4.7 and show the training loss over time and the final test error for the input size and output size combination of (4,8) and (14,10).

input size M	output size N	accuracy
8	4	0.975
14	10	0.958
16	12	0.872
24	16	0.887
32	24	0.846

Table 4.4: Task softmax with a random linear mapping accuracy for different input output sizes.

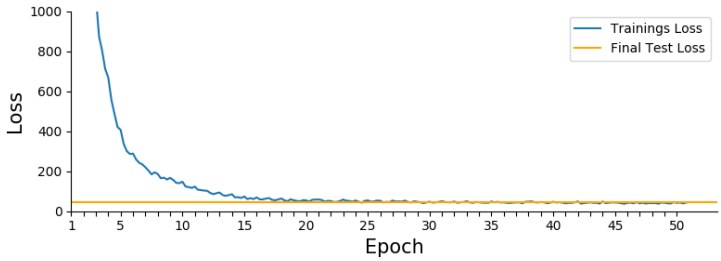

Figure 4.7: **Task softmax with a random linear mapping with 8 dimensional input and 4 dimensional output loss over time** The plot shows the evolution of the training loss over time and the final test error.

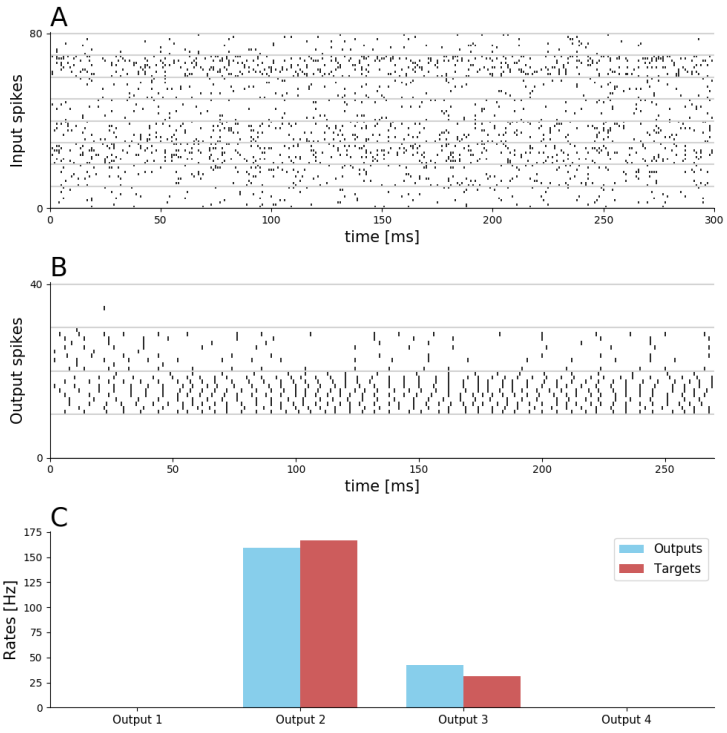

Figure 4.8: **Task softmax with a random linear mapping with 8 dimensional input and 4 dimensional output sample A**) Spikes of the Poission processes which function as input to the network. **B**) Spiking activity of the output neurons. **C**) Comparison of the averaged output rates to the target rates.

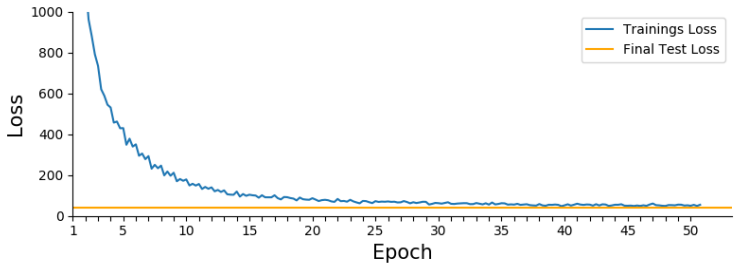

Figure 4.9: **Task softmax with a random linear mapping with 14 dimensional input and 10 dimensional output loss over time** The plot shows the evolution of the training loss over time and the final test error.

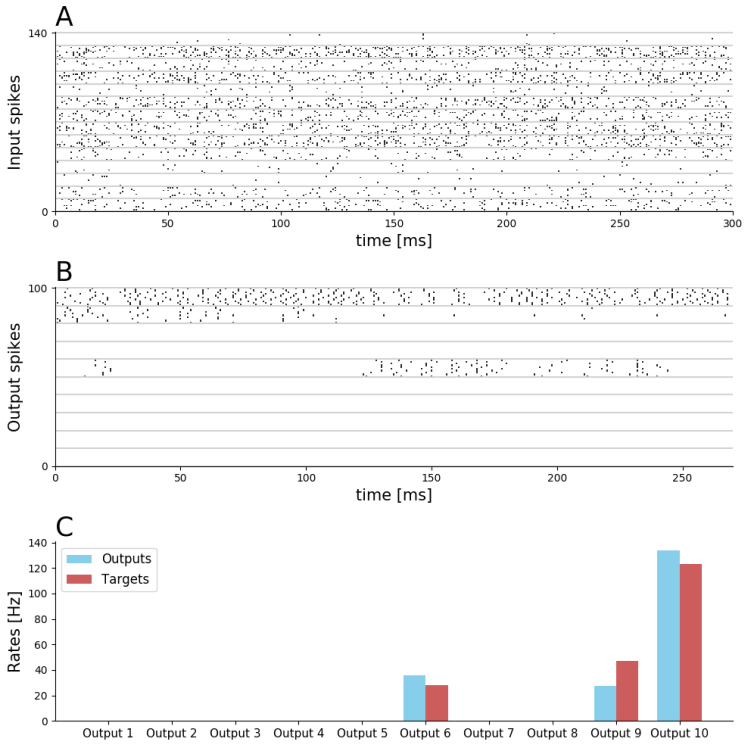

Figure 4.10: **Task softmax with a random linear mapping with 14 dimensional input and 10 dimensional output sample A**) Spikes of the Poission processes which function as input to the network. **B**) Spiking activity of the output neurons. **C**) Comparison of the averaged output rates to the target rates.

4.1.6 Task softmax with negative input values

In the previous tasks, we were only able to model positive input values because we encode the input values with spike rates. As rates are non-negative, this encoding scheme does not encode negative input values. To overcome this problem, we use two input values and subtract them to receive the actual input value. The subtraction map is up to the network to learn. Therefore we increase the input size M by a factor of two. We take the subtraction only in the target generation into account.

An input value of an input sample $\mathbf{x}$ has a positive part x_m^+ and a negative part x_m^-. We draw x_m^+ and x_m^- from an exponential probability distribution EXP_β with a density function f_β in Equation 4.7 ($\beta = 0.5$)

$$\begin{aligned} \mathbf{x} &= (x_1^+, x_1^-, \cdots, x_M^+, x_M^-)^T \\ x_m^+, x_m^- &\sim \text{EXP}_\beta. \end{aligned} \tag{4.13}$$

In the target generation for a target value t_n, we subtract the negative part from the positive part. Then we calculate the target values $\mathbf{t}^* = (t_1^*, \cdots, t_N^*)^T$ the same as in the previous tasks

$$\mathrm{t}_n^* = \rho_{\text{scale}} \frac{\exp(x_n^+ - x_n^-)}{\sum_{j=0}^{N} \exp(x_j^+ - x_j^-)}. \tag{4.14}$$

Results

We performed the task for an input size of 8, 20, 32, 12, 42, and 42. Note that the output size is always a factor two smaller than the input size because of the mapping for the negative values. Table 4.5 shows a drop performance occurred for larger input sizes. Figures 4.11 and 4.13 show two samples with an input size of 8 and 20. Figures 4.12 and show the training loss over time and the final test error for the input size of 8 and 20.

input size M	output size N	accuracy
8	4	0.952
20	10	0.907
32	16	0.755
42	24	0.886
64	32	0.877

Table 4.5: Task softmax with negative input values accuracy for different input output sizes.

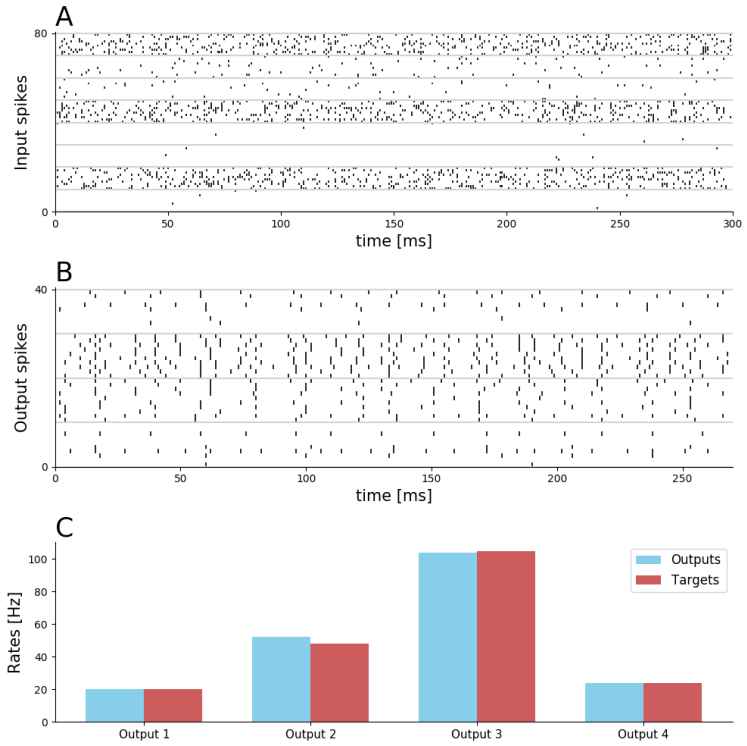

Figure 4.11: **Task softmax with negative input values with 8 dimensional input and 4 dimensional output sample A**) Spikes of the Poission processes which function as input to the network. **B**) Spiking activity of the output neurons. **C**) Comparison of the averaged output rates to the target rates.

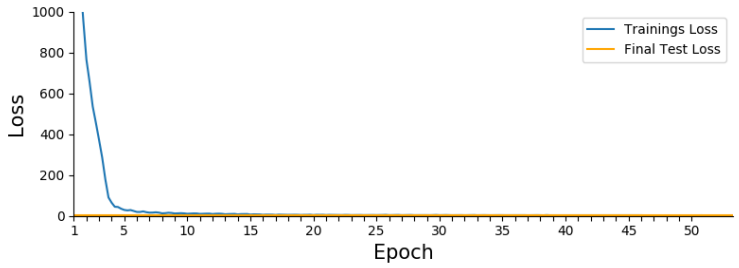

Figure 4.12: **Task softmax with negative input values with 8 dimensional input and 4 dimensional output loss over time** The plot shows the evolution of the training loss over time and the final test error.

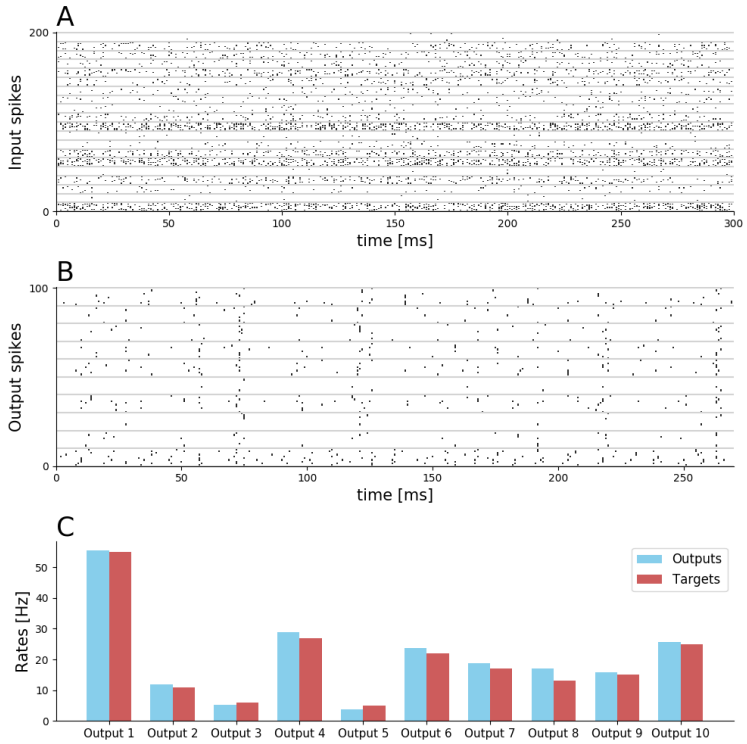

Figure 4.13: **Task softmax with negative input values with 20 dimensional input and 10 dimensional output sample A**) Spikes of the Poission processes which function as input to the network. **B**) Spiking activity of the output neurons. **C**) Comparison of the averaged output rates to the target rates.

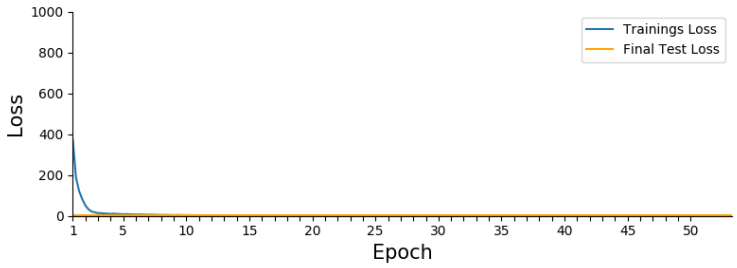

Figure 4.14: **Task softmax with negative input values with 20 dimensional input and 10 dimensional output loss over time** The plot shows the evolution of the training loss over time and the final test error.

4.2 Embedding of spiking softmax in an LSNN

Spiking Neural Networks often use the softmax function as their output, because it scales the activation into an easy to interpret probability vector. Additionally, the use of the softmax in combination with the cross entropy as loss function results in a simple and easy to handle gradient for back-propagation. Therefore we asked if our trained softmax network can replace a softmax function in the setting of an LSNN.

We use the store-recall task from Bellec, Scherr, et al., 2019 and Sequential MNIST similar to Bellec, Salaj, et al., 2018 for the experiments. The store-recall task is a simple working memory task. Sequential MNIST is a classification task on the MNIST dataset, where we show the network an image pixel by pixel.

4.2.1 Replacing the softmax

In both tasks, we use a pool of LIF neurons and adaptive LIF neurons (ALIF) as the central part of the network. Then we encode the output of this neuron pool with the membrane potential of linear readout neurons. Then we normalize these values with a softmax function.

To replace the softmax with our network, we have to change the output calculation because we need spike trains as an input for our network. Therefore we replace the part with the linear readout neurons in Figure 4.15 and 4.19 with an additional layer of LIF neurons. These input LIF neurons act as the input neurons for the softmax network and replace the Poisson processes in the softmax network structure. Otherwise, the softmax network part has the same structure as the network we used for our previous tasks.

In the original LSNN with artificial softmax output, cross entropy was used as the loss function. We change the cross-entropy to the mean-squared-error because the gradient is different with the softmax network. The cross-entropy L_{ce} with a softmax for the network activations a_i is

$$\begin{aligned} L_{\text{ce}} &= \sum_{i=1}^{N} t_i^* \log out_{i,\text{softmax}} \quad \text{with} \\ out_{i,\text{softmax}} &= \frac{\exp(a_i)}{\sum_{j=1}^{N} \exp(a_j)}. \end{aligned} \tag{4.15}$$

We denote t_i^*as a target value for the network output value $out_{i,\text{softmax}}$. The gradient of L_{ce} with respect to an activation a_i of the network is

$$\frac{\partial L_{\text{ce}}}{\partial a_i} = out_{i,\text{softmax}} - t_i^*. \tag{4.16}$$

In our case, we use the activation of the network directly as the output value, since our pre-trained network performs the softmax calculation. The activation of our network and also the output value out_i, is the average firing frequency of the associated output neuron group, see equation 4.3. Therefore the cross-entroy-loss is

$$L_{\text{ce}} = \sum_{i=1}^{N} t_i^* \log out_i. \tag{4.17}$$

The gradient of L_{ce} with respect to the activation value and also the output out_i is

$$\frac{\partial L_{\text{ce}}}{\partial out_i} = \frac{t_i^*}{out_i}. \tag{4.18}$$

In order to obtain a gradient of the form shown in equation 4.16, we use the mean-squarred-error

$$L_{\text{mse}} = \frac{1}{2}\sum_{i=1}^{N}(out_i - t_i^*)^2. \tag{4.19}$$

The gradient of L_{mse} with respect to the activation and output out_i is now

$$\frac{\partial L_{\text{ce}}}{\partial out_i} = out_i - t_i^*. \tag{4.20}$$

For training, we first train the weights for the softmax network part. Therefore we use the same protocol as in task normal input 4.1.3. Then we train the whole network and fix the weights of the softmax on the given task.

4.2.2 Store-recall

The store-recall task is a simple working memory task. The network receives different input values over time. The network has to remember the value at the time of a store command and has to output this value at the time of a recall command. Bellec, Scherr, et al., 2019 used two input values. We increased the input values to four because we want to justify the use of a softmax function.

The original network received four inputs. The inputs were store, recall, value 0 and value 1. Dedicated input neuron groups encoded the input to the network, with a groups size of 25. They defined a command or a value for a time period of 200 ms. In this time frame the input neurons of this command or value spiked with a frequency of 50 Hz. The main part of the

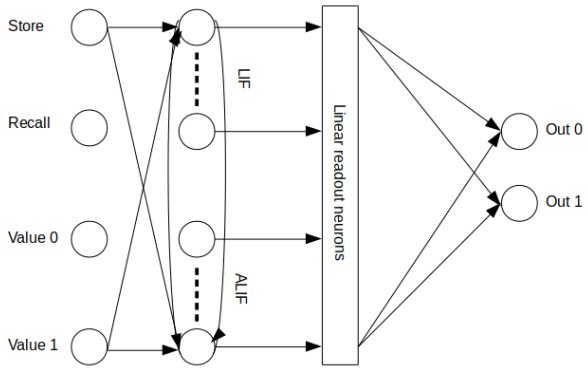

Figure 4.15: **Bellec et al. network setup for store-recall** The network consisted of input neurons pools for every input signal, a pool of LIF and ALIF neurons and linear readout neurons.

network is a pool of recurrently connected LIF neurons and adaptive LIF neurons. The pool contained 10 LIF neurons and 10 adaptive LIF neurons. Figure 4.15 shows the structure of the network.

To compute the output of the LSNN, Bellec, Scherr, et al., 2019 use the membrane potentials of the linear readout neurons over the whole simulation time and a weight matrix to sample them down to the desired output size of 2. Afterward, they average the sampled-down membrane potentials in the period of the recall command, which gives them one value for every output. Finally, they use the softmax to scale these two values into probability space and get the final output values of the LSNN. They use the cross-entropy of a target value (0 or 1 in their case) and this output values as the loss function.

We increased the number of values that we show to the store-recall task to four, which also increased the number of outputs to four. Then we changed the linear readout neurons to the LIF neurons and replaced the softmax with our network, as discribed in section 4.2.1. Figure 4.16 illustrates the new network structure. As the final output values we condider the output values of the softmax network in the time periode of the recall command.

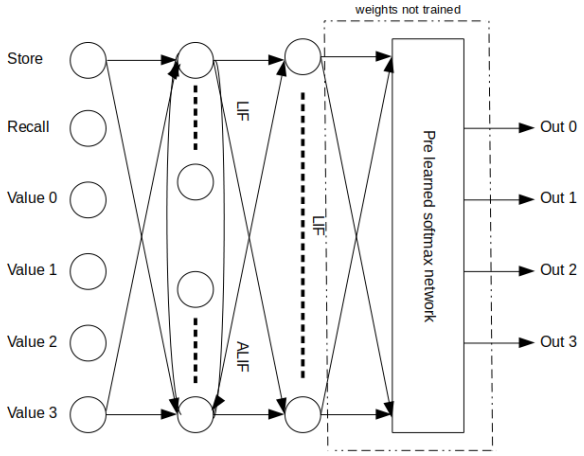

Figure 4.16: **Network setup for store-recall** We increased the possible input values to four and replaced the linear readout neurons with LIF neurons and a pre-trained softmax network. We do not train the weights of the softmax network.

Training

The input and output size of the softmax network is four (M=4 and N=4 in Figure 4.2). We use an input group size of 10 neurons. This means we have 40 LIF, which act as input neurons to the softmax network, see Figure 4.16. We increased the number of neurons in the recurrently connected neuron pool of LIF and ALIF neurons to 40 (20 LIF and 20 ALIF). We did this because we do not want to upscale between the neuron pool and the LIF input neurons and we also have more possible input values. We used regularization for training the softmax network, see section 4.1.4. We set the target frequency of the hidden neurons to 50 Hz and used a regularization factor of 50.

For training the whole network, we used the same training protocol as Bellec, Scherr, et al., 2019. We created randomly training samples which were randomly generated inputs with a length of 2.4 s. We showed the network every command (store or recall) or value (value 0, value 1, value 2 or value 3) for a time period of 200 ms. In each such period, we showed the network a command with a probability of $\frac{1}{6}$. The first command is a store command and the next command is a recall command, see Figure 4.17 for

an example spike train of a training sample.

We randomly generated a new training set with a size of 128 and a new validation set with a size of 128 for every new iteration. We used the validation set to evaluate the performance of the network.

Results

The network is able to learn the store-recall task. The wrongly classified samples in the training and valitation set converge to around zero, see Figure 4.18, 4.17.

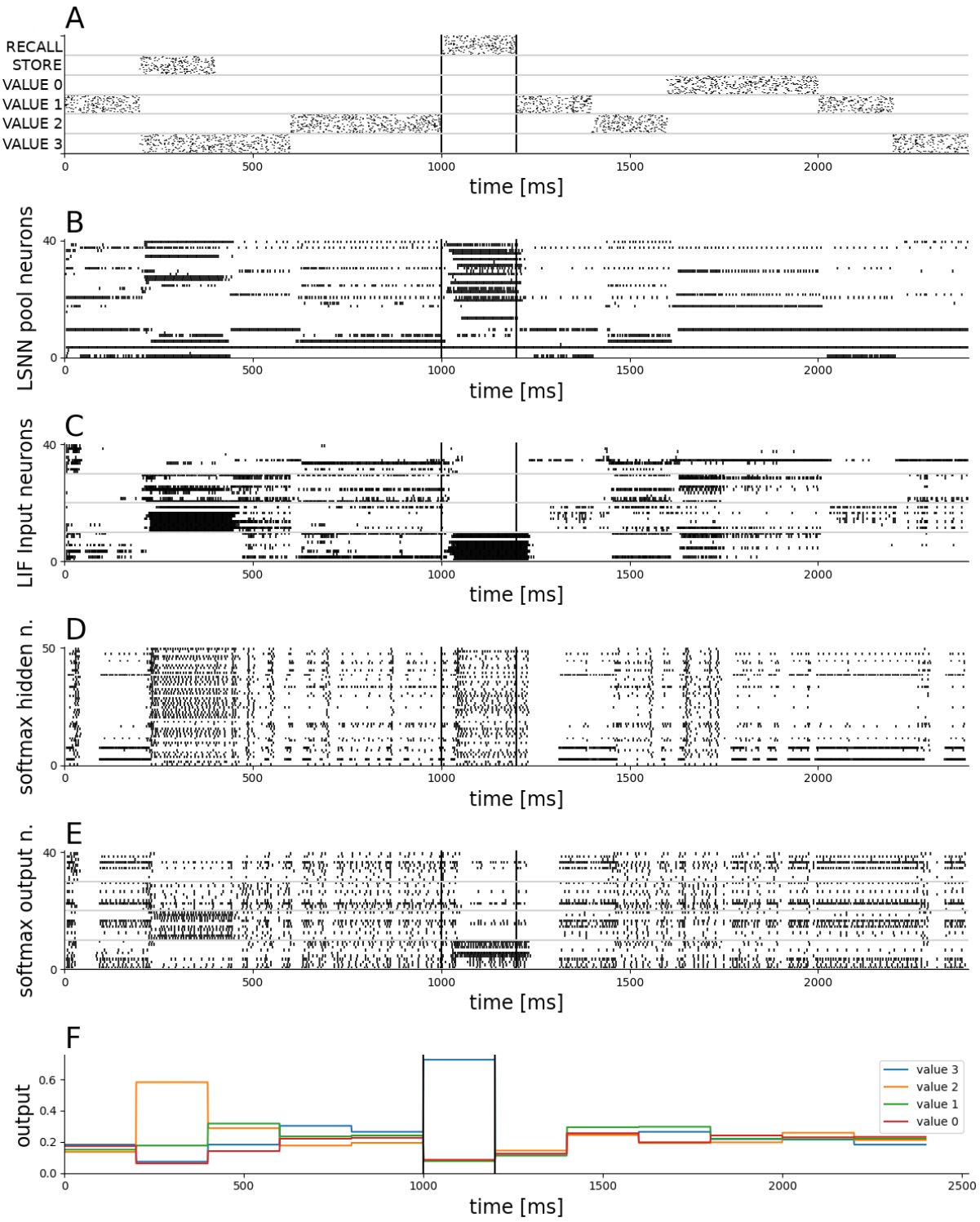
A
RECALL
STORE
VALUE 0
VALUE 1
VALUE 2
VALUE 3
0
500
1000
1500
2000
time [ms]
B
LSNN pool neurons
40
0
time [ms]
C
LIF Input neurons
40
0
time [ms]
D
softmax hidden n.
50
0
time [ms]
E
softmax output n.
40
0
time [ms]
F
output
0.6
0.4
0.2
0.0
2500
time [ms]
value 3
value 2
value 1
value 0

Figure 4.17: **Testing the softmax network in the store recall task A**) The activity of the input neurons to the LSNN network. **B**) The activity of the LSNN pool of the network. **C**) The spiking behavior of the input neurons to the softmax part of the network and **D** The activity of the hidden neurons in the softmax part of the network. **E**) The spiking behavior of the output neurons of the softmax part of the network. **F**) The averaged output values in 200 ms time windows. We marked the time window of the recall command with black lines.

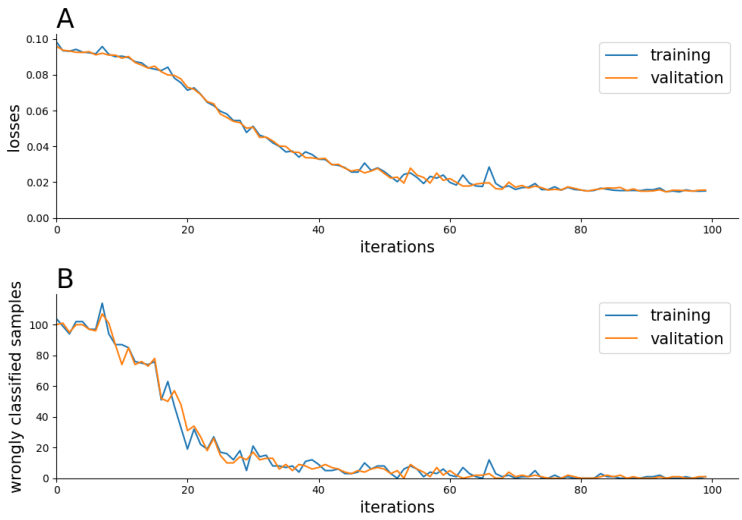

Figure 4.18: **Loss and wrong classified samples of the store-recall task A**) The top plot shows the time course of the loss of the training set and the validation set. **B**) The bottom plot shows the time course of the number of wrongly classified samples in the training and validation set. The training set and the validation set have a samples size of 128.

4.2.3 Sequential MNIST

In this task, we replace the softmax with our softmax network in an LSNN that learns the sequential MNIST. The goal of sequential MNIST is to classify the images of handwritten digits of the MNIST dataset, similar to Bellec, Salaj, et al., 2018.

The images consist of 28x28 pixels. We show the pixels to the network by going through the image row by row. We scale the grey values of the image into the interval $[0, 1]$. For every timestep (1 ms), we show the network one scaled pixel value, which adds up to a total simulation time of 784 ms (784 pixels).

The LSNN contains a total of 240 neurons, where 140 of these neurons are adaptive LIF neurons. Instead of spikes from input neurons, these neurons receive the scaled pixel value with a weight vector $\mathbf{w} \in \mathbb{R}^{N_{rec}}$ as input. This changes equation 2.6 for the input current $I_j(t)$ of a neurons j at time t to

$$I_j(t) = w_j p_{\text{scaled}}(t) + \sum_{\substack{i \in N_{rec} \\ i \neq j}} W_{ji}^{rec} z_i^{rec}(t - d), \tag{4.21}$$

where $p_{\text{scaled}}(t)$ is the tth scaled pixel of the image. Figure 4.19 shows the whole structure of the network.

We replaced the softmax, which normalizes the membrane potential of the readout neurons with our pre-trained softmax network, as described section 4.2.1.
Our pre-trained softmax network computes the softmax of a spike train over a given time. We use the last 400ms of the simulation time, which is equal to the last 400 pixels for the softmax computation. Figure 4.20 shows the new network structure.

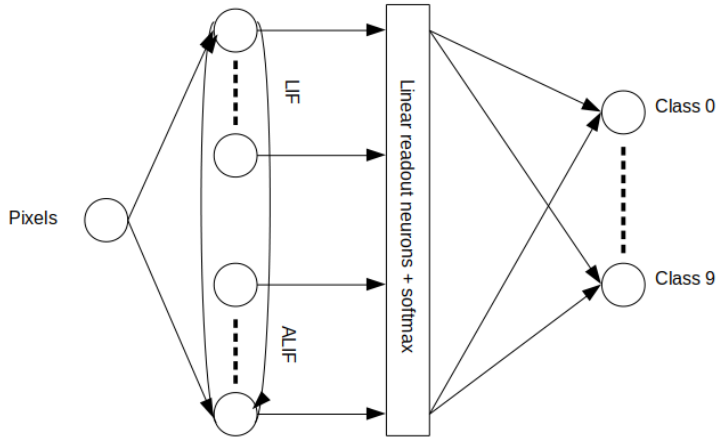

Figure 4.19: **Network setup for Sequential MNIST with softmax** The network consists a pool of LIF and ALIF neurons, which receive the pixels as input. Linear readout neurons compute the output of this pool.

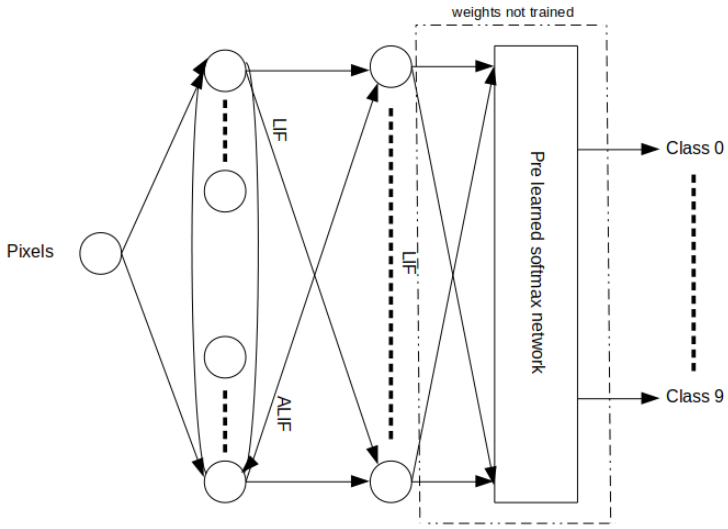

Figure 4.20: **Network setup for Sequential MNIST with softmax network** We replaced the linear readout neurons with LIF neurons and a pre-trained softmax network. We do not train the weights of the softmax network.

Training

We use a group of 10 neurons per input to the softmax. Therefore we have 100 LIF input neuron for the softmax network.

We again train the softmax beforehand and then fix the weights while training the whole network to perform the given task. In the original setup the network was optimized with BPTT and Deep R, similar to Bellec, Salaj, et al., 2018. In the new setup, we only use DEEP R on the pool of LIF and ALIF neurons and not the input neurons for the softmax network or the softmax network. As in the original setup, we use a connectivity precentage of 30%, 0.01 as the regularization coefficient and no noise for DEEP R.

Results

The original setup (softmax function) achieved a test set accuracy of 0.89. The new setup (softmax network) achieved a test set accuracy of 0.91. Figure 4.21 shows the network activity for a sample of the new setup. We took the mean over five runs for the accuracy. We think that the softmax network performs better since the softmax network computes the output over a time period. The last pixels of most images have a scaled pixel value of zero. A scaled pixel value of zero translates to no input. Therefore the network stops spiking because there is no new input to the network. We think because of the additional layers and early timesteps, which contribute actively to the output, the LSNN with the softmax network can maintain information for a longer period in the network.

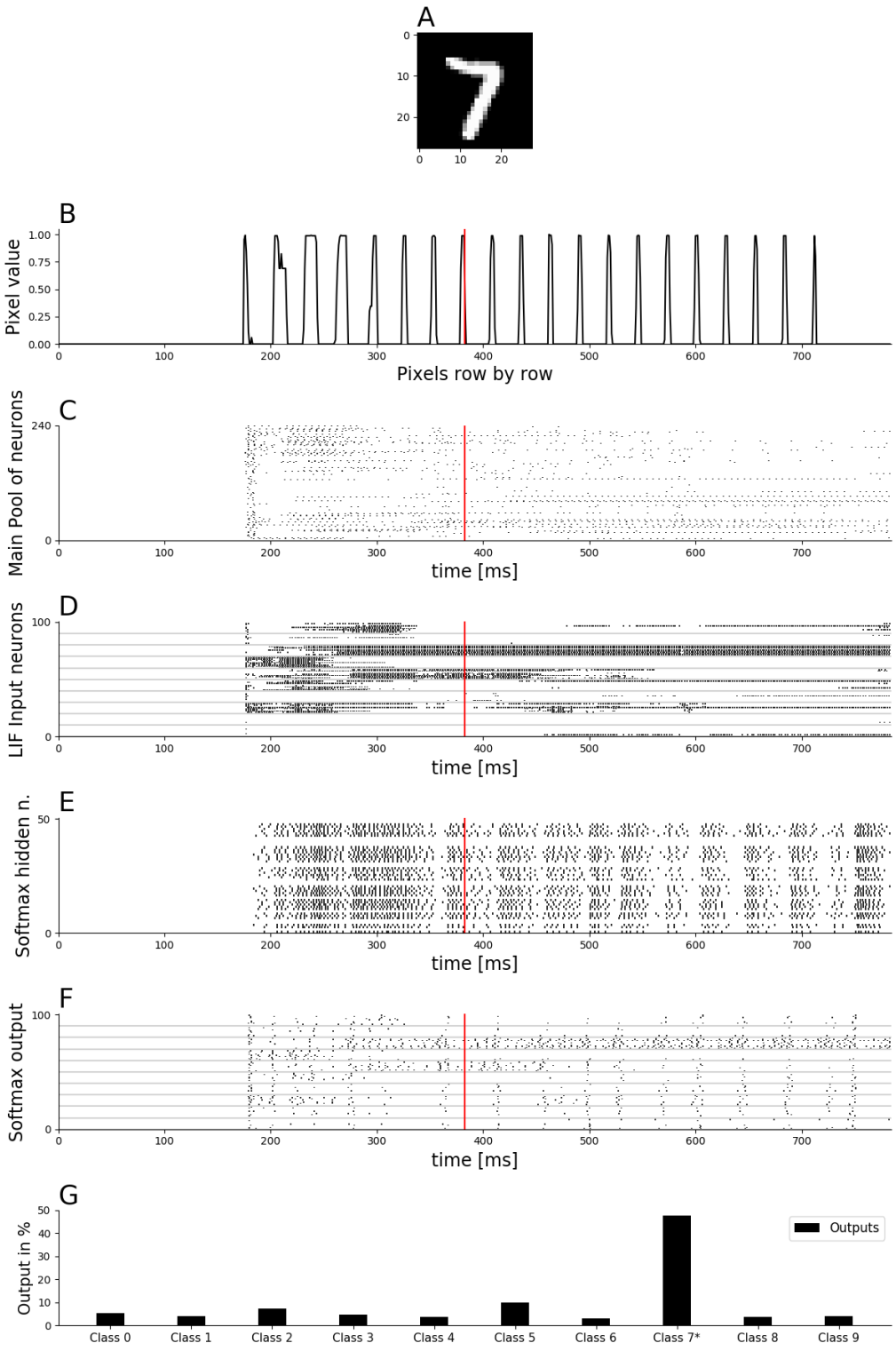
A
B
Pixel value
Pixels row by row
C
Main Pool of neurons
time [ms]
D
LIF Input neurons
time [ms]
E
Softmax hidden n.
time [ms]
F
Softmax output
time [ms]
G
Output in %
Outputs
Class 0
Class 1
Class 2
Class 3
Class 4
Class 5
Class 6
Class 7*
Class 8
Class 9

Figure 4.21: **Testing the softmax network in the sequential MNIST task A**) The input image. **B**) The scaled pixel values over in sequential order (row by row). The red line marks the starting time of the output calculation. **C**) The activity of the LSNN pool of the network. **D**) The spiking behavior of the input neurons to the softmax part of the network and **E** The activity of the hidden neurons in the softmax part of the network. **F**) The spiking behavior of the output neurons of the softmax part of the network. **G**) The output of the whole network for every class. The correct class is labeled with a star "*".

4.3 K-winner-takes-all

K-winner-takes-all identifies the K-largest components of an input vector $\mathbf{x} \in \mathbf{R}^N$ of size N.

$$y_i(x_1, \cdots, x_N) = \begin{cases} 1 & \text{rank}_i(\mathbf{x}) < \text{K} \\ 0 & else \end{cases} \quad \text{with} \quad \text{rank}_i(\mathbf{x}) = \Big|\{j | x_j > x_i\}\Big|, \tag{4.22}$$

where $\mathbf{x}$ is the input and y_i is the ith output. The function rank_i gives the number of components larger than the ith component. If x_i is in the K-largest numbers of $\mathbf{x}$, y_i is one.

4.3.1 RSNN setup

We use almost the same network setup as for the softmax tasks, compare section 4.1.1. The input layer again consists of subpopulations G_n^{in} of input neurons. The output layer also consists of subpopulation G_n^{out} of output neurons. To increase the computational power of the network, we again use the population of hidden neurons G_{hidden}. The neurons of the hidden population and the neurons of the subpopulation for the output are recurrently connected. Figure 4.22 shows the whole structure.

This time, we only use an identity matrix as mapping on the input vector. Therefore the output size N is equal to the input size. We again use a Poisson

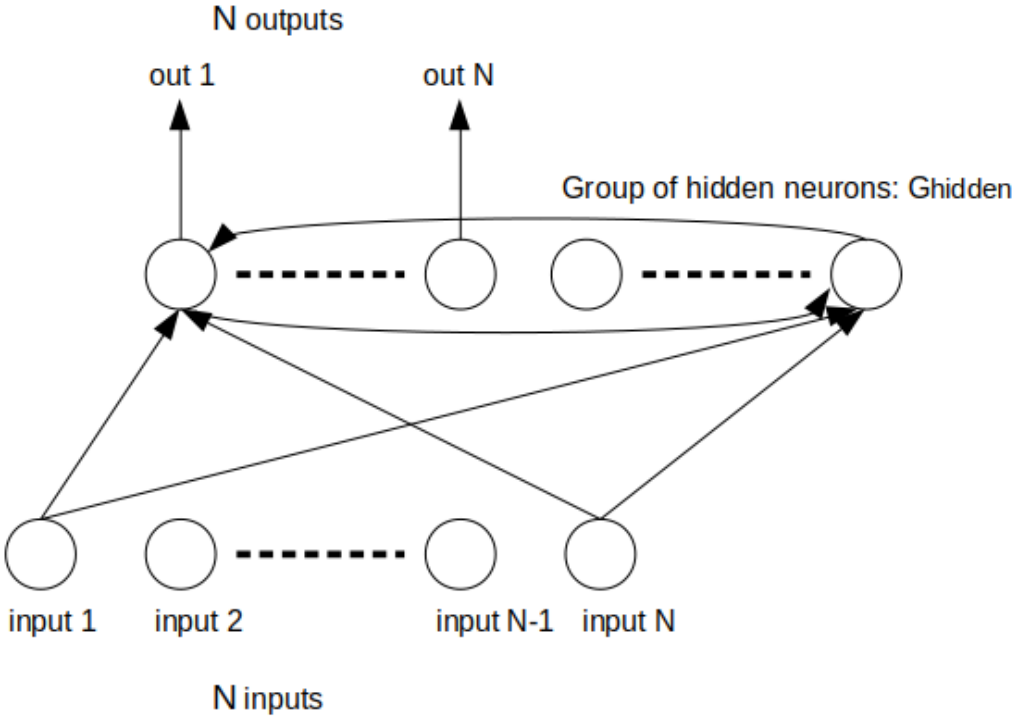

Figure 4.22: **RSNN structure** The abstract structure of the network we use for learning the K-winner-takes-all function.

process to create spike trains for the neurons of the input populations, see equation 4.2. To calculate the output values $\text{out}_1, \cdots, \text{out}_N$ of the network, we again use the average spiking frequency of their dedicated subpopulation, see equation 4.3.

Training

We do not use the mean squared error as a loss function anymore. For the K-winner-takes-all function, we do not need exact output rates. We only need to distinguish between zero or one. Therefore we choose a frequency that acts as a border. Every output with a frequency higher than the border, we interpret as one. Every output with a frequency equal or lower, we

interpret as a zero. Therefore the mean squared error is a bad fit since the mean squared error punishes every deviation form a chosen target. Instead, we use a loss function similar to the hinge loss.

We linearly punish outputs, which are on the wrong side of their desired border. In the loss function, we use two different borders: ρ_{zero} and ρ_{one}. This choice creates outputs, which better visually resemble a K-winner-takes-all function, see Figure 4.24, 4.30, 4.28 and 4.26. This decision had no impact on the performance of the network. The following equation shows the loss for one sample.

$$L = \sum_{n=0}^{N} (1 - t_n^*) \max\{0, out_n - \rho_{\text{zero}}\} + t_n^* \max\{0, \rho_{\text{one}} - out_n\}, \quad (4.23)$$

where N is the output size, ρ_{zero} is the maximum for a zero in Hz and ρ_{one} is the minimum for a one in Hz. out_n is the nth output of the network. t_n^* is the target for the nth output ($t_n^* \in 0, 1$).

Additionally, we use the regularization of section 4.1.4, which is the mean squared error between a choosen frequency and the average frequency of the neurons in the hidden population G_{hidden}, see equation 4.10. Since our loss function equation 4.23 does not punish high frequencies if the output should be a one, we want to control high frequencies in hidden neurons.

We use back-propagation-through-time with Adam and a decaying learning rate. We start with a learning rate of 0.01.We decrease the learning rate every 10 epochs by a factor of 0.8.

We used a training set with 10000 random samples and a batch size of 50 for learning. We used a validation set of 1000 random samples to identify the best performing parameters. After learning, we used a test set of 1000 random samples to measure the final performance of the network.

Generel Parameters

We listed all parameters for the network in Table 4.6.

simulation time T	400 ms
initialization time T_{init}	100 ms
input subpopulation size $\lvert G^{in} \rvert$	10
output subpopulation size $\lvert G^{out} \rvert$	10
ρ_{scale}	100 Hz
ρzero	20 Hz
ρone	50 Hz
hidden population size $\lvert G_{\text{hidden}} \rvert$	50
timestep δt	1 ms
damping factor ν	0.3
threshold γ	0.5
synaptic delay d	1 ms
refactory period κ	1 ms
membrane time constant τ_m	20 ms

Table 4.6: General parameters for K-winner-takes-all training

4.3.2 Task K-winner-take-all

We draw samples from an exponential probability function to get input samples.

$$\mathbf{x} = (x_1, \cdots, x_N)^T \quad \text{with} \quad x_i \sim \text{EXP}_\beta, \tag{4.24}$$

where $\mathbf{x}$ is the drawn input samples from EXP_β. EXP_β has the probability density function descirbed in equation 4.7 and we set β to 0.5.

We use the equation 4.22 to generate the target values for a given input sample.

Results for 1 K-winner

In this example we choose 1 winner (K = 1). We performed the task with different input sizes. Then we compared the performances of the network in Table 4.7. We consider an output to be correct, if the rates are on the right side of the border ρ_{zero}. This means if the output should be one the rate has to be greater than ρ_{zero} and Vise versa. The network achieves great results with small input sizes but struggles with larger. Figure 4.24 and 4.26 show two samples as examples and Figure 4.23 and 4.25 show the training loss over time and the final test loss for a 5 and 10 dimenional input.

input size N	accuracy
5	0.952
10	0,84
16	0.83
24	0.649
32	0.487

Table 4.7: Task K-winner takes all with 1 Winner (K = 1) accuracy for different input sizes.

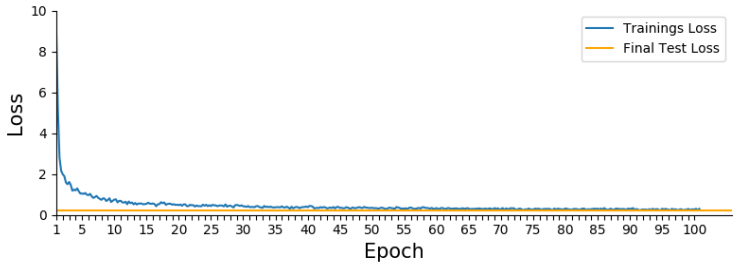

Figure 4.23: **Task K-winner-takes-all 5 dimensional input and 1 Winner loss over time** The plot shows the evolution of the training loss over time and the final test error.

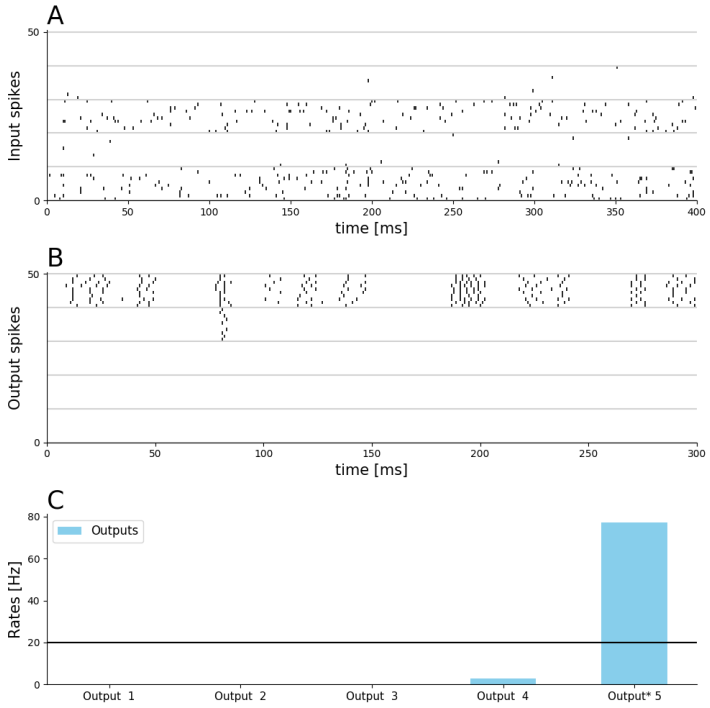

Figure 4.24: **Task K-winner-takes-all 5 dimensional input and 1 Winner sample A**) Spikes of the Poission processes which function as input to the network. **B**) The spiking activity of the output neurons. **C**) The averaged output rates and the border ρ_{zero}. The star "*" labels the output, which should have won.

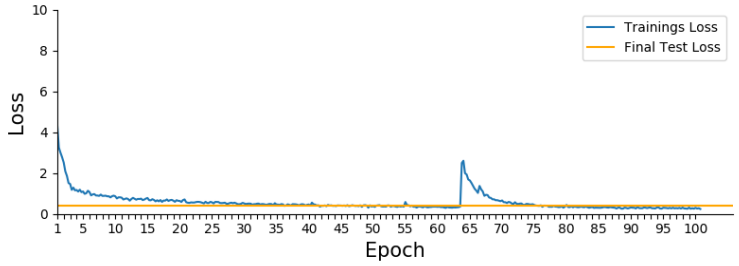

Figure 4.25: **Task K-winner-takes-all 10 dimensional input and 1 Winner loss over time** The plot shows the evolution of the training loss over time and the final test error.

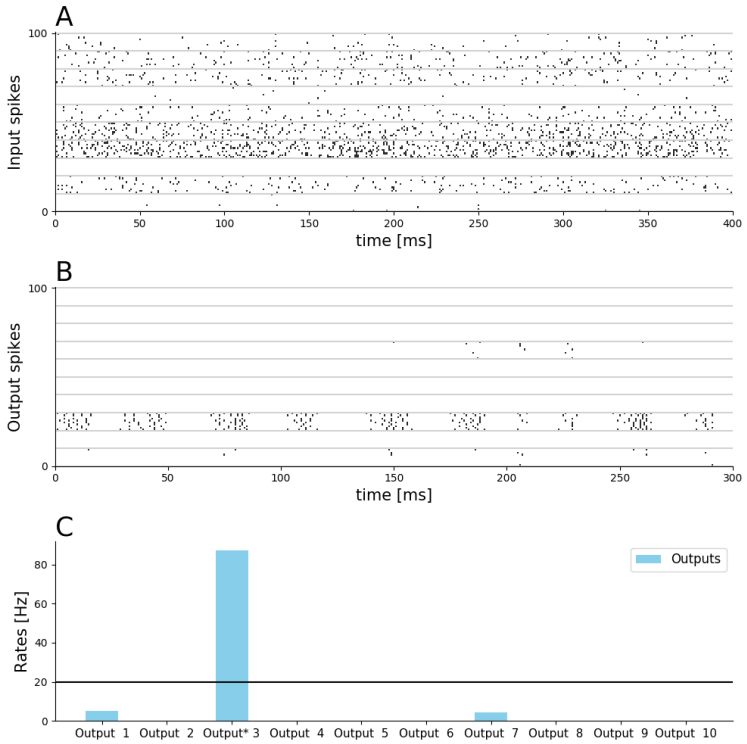

Figure 4.26: **Task K-winner-takes-all 10 dimensional input and 1 Winner sample** **A**) Spikes of the Poission processes which function as input to the network. **B**) The spiking activity of the output neurons. **C**) The averaged output rates and the border ρ_{zero}. The star "*" labels the output, which should have won.

Results for 3 K-winners

In this example we choose 3 winners (K = 3). We performed the task with different input sizes. Then we compared the performances of the network in Table 4.8. The network achieves great results with small input sizes but struggles with larger. Figure 4.30 and 4.28 show two samples as examples and Figure 4.29 and 4.27 show the training loss over time and the final test loss for a 5 and 10 dimenional input.

input size N	accuracy
5	0,94
10	0,858
16	0.648
24	0.433
32	0.225

Table 4.8: Task K-winner takes all K=3 accuracy for different input sizes.

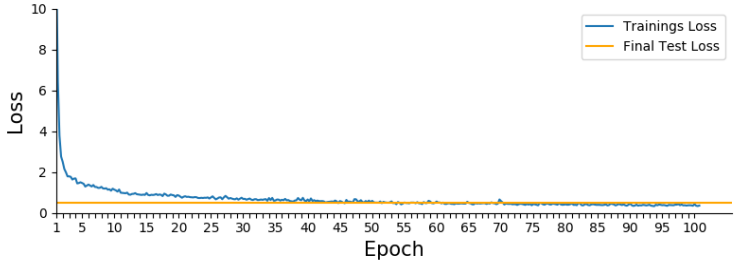

Figure 4.27: **Task K-winner-takes-all 10 dimensional input and 3 Winners loss over time** The plot shows the evolution of the training loss over time and the final test error.

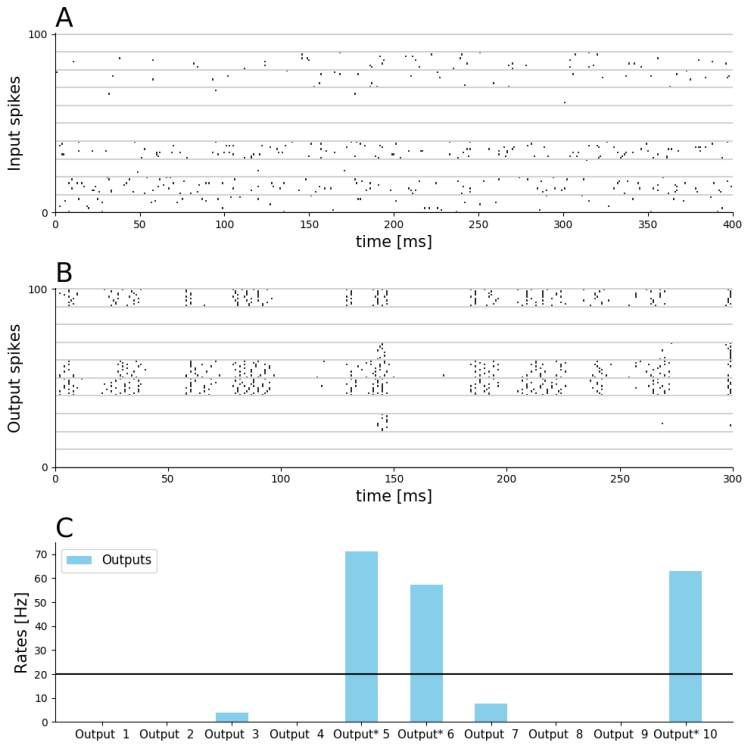

Figure 4.28: **Task K-winner-takes-all 10 dimensional input and 3 Winners sample** **A**) Spikes of the Poission processes which function as input to the network. **B**) The spiking activity of the output neurons. **C**) The averaged output rates and the border ρ_{zero}. The star "*" labels the output, which should have won.

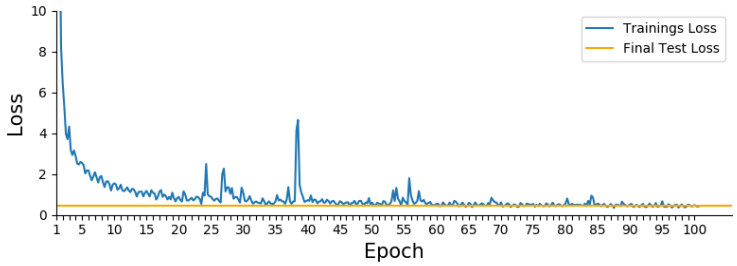

Figure 4.29: **Task K-winner-takes-all 5 dimensional input and 3 Winners loss over time** The plot shows the evolution of the training loss over time and the final test error.

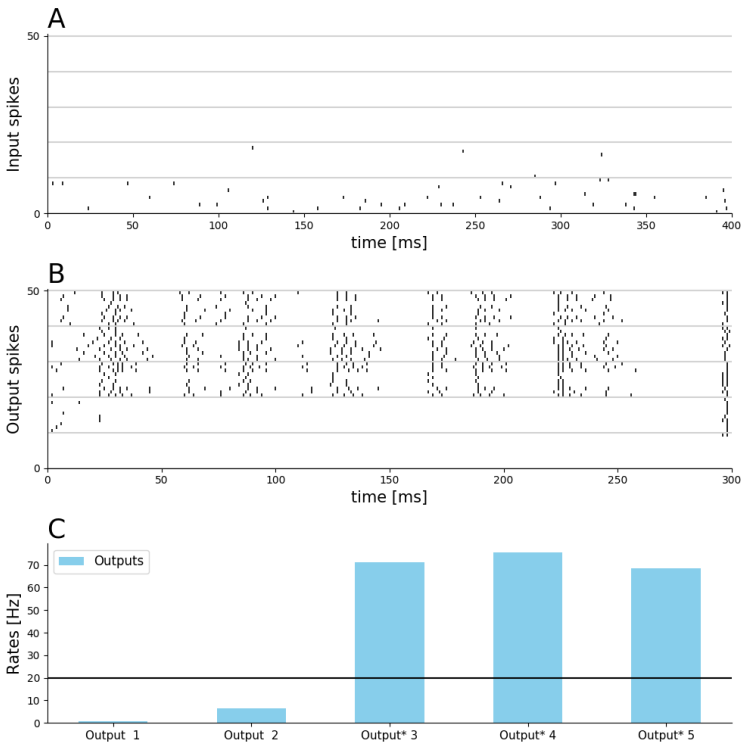

Figure 4.30: **Task K-winner-takes-all 5 dimensional input and 3 Winners sample A**) Spikes of the Poission processes which function as input to the network. **B**) The spiking activity of the output neurons. **C**) The averaged output rates and the border ρ_{zero}. The star "*" labels the output, which should have won.

5 Discussion

We demonstrated that one can train an RSNN to learn the softmax function and the K-winner-takes-all function. Furthermore, we replaced the softmax function with our softmax network in two LSNNs and successfully trained the LSNNs afterward. The first LSNN performed a working memory task, see section 4.2.2. The second performed classification on the sequential MNIST dataset, see section 4.2.3.

We show the RSNN can learn the softmax combined with an additional linear mapping of the input vector, see section 4.1. The first additional mapping was a random matrix on the input vector. The second was a mapping that allowed us to represent negative values. Although the RSNN achieved good results, the performance of the network dropped for larger input sizes, see Tables 4.2,4.4 and 4.5.

In the second part, we replaced a softmax function with our pre-trained network. The replacement is rather straight forward. Instead of using the membrane potential of the readout neurons as activations, we use the spike trains of the readout neurons as input to our softmax network, see section 4.2.1. We also change the loss function from the cross-entropy to the mean squared error because the removal of the softmax function changed the gradient.

The first LSNN in which we replaced the softmax is a network performing a working memory task (Store-recall). The goal of the task is to achieve an accuracy of over 0.95 on a validation set. Our LSNN with the softmax network was able to achieve this, see section 4.2.2. The second task is a classification task on the sequential MNIST dataset. In this secenario, the LSNN with our

softmax network achieved a better accuracy on the test set as the LSNN with the softmax function, see section 4.2.3. These two results are promising for the softmax network. In the case of the sequential MNIST task, the softmax network can help to maintain information in the network longer, which ultimately helped with the performance of the network. Although to fully understand the effects of the softmax network on the performance of the network, more experiments are necessary.

For the K-winner-takes-all functions, we changed the loss function from the means squared error to a function similar to the hinge loss, see section 4.3.1. The choice of output interpretation made the mean squared error unpractical to use. The RSNN managed to achieve good performances for small input sizes, but Tables 4.7 and 4.8 show a noteworthy decline in accuracy for larger input sizes.

5.1 Outlook

Altogether our results show an interesting first look on this topic. We achieved good results with almost straight forward implementations. However, we think the performance of the softmax network can be inproved because we did not fine-tuning the hyperparameters of the RSNN. The same holds for the K-winner-takes-all network. Also, the effects of the softmax network in an LSNN have to be further studied. The fact that the LSNN with the softmax network outperformed the LSNN with the softmax function will probably not hold for every scenario. Furthermore, we did not integrate the K-winner-takes-all network in an LSNN. It would be interesting to see how the network compares to a K-winner-takes-all function in an LSNN in terms of performance.

Appendix

Bibliography

Bellec, Guillaume, David Kappel, et al. (2017). "Deep rewiring: Training very sparse deep networks." In: *arXiv preprint arXiv:1711.05136* (cit. on p. 9).

Bellec, Guillaume, Darjan Salaj, et al. (2018). "Long short-term memory and learning-to-learn in networks of spiking neurons." In: *Advances in Neural Information Processing Systems*, pp. 787–797 (cit. on pp. 1, 2, 5, 6, 8, 31, 39, 41).

Bellec, Guillaume, Franz Scherr, et al. (2019). "Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets." In: *arXiv preprint arXiv:1901.09049* (cit. on pp. 31, 33–35).

Gerstner, Wulfram et al. (2014). *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press (cit. on pp. 3, 6).

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory." In: *Neural computation* 9.8, pp. 1735–1780 (cit. on p. 5).

Huh, Dongsung and Terrence J Sejnowski (2018). "Gradient descent for spiking neural networks." In: *Advances in Neural Information Processing Systems*, pp. 1433–1443 (cit. on pp. 11, 12).

Jug, Florian et al. (2012). "Spiking networks and their rate-based equivalents: does it make sense to use Siegert neurons?" In: *Swiss Society for Neuroscience* (cit. on pp. 11, 12).

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (cit. on p. 8).

LeCun, Yann et al. (1988). "A theoretical framework for back-propagation." In: *Proceedings of the 1988 connectionist models summer school*. Vol. 1. CMU, Pittsburgh, Pa: Morgan Kaufmann, pp. 21–28 (cit. on p. 7).

Liu, Qian, Yunhua Chen, and Steve Furber (2017). "Noisy Softplus: an activation function that enables SNNs to be trained as ANNs." In: *arXiv preprint arXiv:1706.03609* (cit. on pp. 11, 12).

Pfeiffer, Michael and Thomas Pfeil (2018). "Deep learning with spiking neurons: opportunities and challenges." In: *Frontiers in neuroscience* 12 (cit. on p. 1).

Shrestha, Sumit Bam and Garrick Orchard (2018). "SLAYER: Spike layer error reassignment in time." In: *Advances in Neural Information Processing Systems*, pp. 1412–1421 (cit. on p. 11).

Tavanaei, Amirhossein et al. (2019). "Deep learning in spiking neural networks." In: *Neural Networks* 111, pp. 47–63 (cit. on p. 1).

Werbos, Paul J (1990). "Backpropagation through time: what it does and how to do it." In: *Proceedings of the IEEE* 78.10, pp. 1550–1560 (cit. on p. 7).

Wu, Yujie et al. (2018). "Spatio-temporal backpropagation for training high-performance spiking neural networks." In: *Frontiers in neuroscience* 12 (cit. on p. 11).

Zenke, Friedemann and Surya Ganguli (2018). "Superspike: Supervised learning in multilayer spiking neural networks." In: *Neural computation* 30.6, pp. 1514–1541 (cit. on p. 11).