Back-propagation learning in deep Spike-By-Spike networks

David Rotermund and Klaus R. Pawelzik

March 6, 2019

University of Bremen, Institute for Theoretical Physics, Bremen, Germany

Correspondence: davrot@neuro.uni-bremen.de; Tel.: +49-421-218-62003

1 Abstract

Neural networks are important building blocks in artificial intelligence applications. These technical systems rely on noiseless continuous signals in stark contrast to the discrete action potentials exchanged among the neurons in real brains. A promising approach towards bridging this gap are the Spike-by-Spike (SbS) networks, which were demonstrated to reliably process information based on few stochastic spikes given suitable synaptic weights are provided. What is missing are algorithms for finding weight sets that would optimize the output performance of acyclically connected SbS networks.

Here, a learning rule for hierarchically organized SbS networks is derived. It is inspired by the error back-propagation algorithms used in feed-forward neural networks. The properties of this approach are investigated and its functionality demonstrated by simulations. In particular, a Deep Convolutional SbS network for classifying handwritten digits (MNIST) is presented. The resulting classification performance reaches values achieved by standard deep networks that have a similar structure and also use only simple gradient descent. We expect that this new SbS back-prop learning rule will provide a basis for building spiking neuronal network models for research in neuroscience and technical applications, especially when it will become combined with state-of-the-art optimization techniques and implemented on specialized computational hardware.

2 Introduction

Fueled by the huge improvements in computational power by CPUs, GPUs, and special hardware, deep neuronal networks ([1]) brought a massive improvement for the field of expert systems as well as artificial intelligence ([2, 3, 4, 5, 6]). These networks started out as simple perceptrons ([7]) which where extended into multi-layer networks by a learning rule that utilizes the chain rule to propagate the error, between the actual and the desired output of the network, back from the output layer to the input. This allows to train all weights in such a network based on this back-propagated error. This learning rule is called backprop ([8]).

In real biological neuronal networks, however, information typically is exchanged between neurons by discrete stereotyped signals, the action potentials. Combining deep networks with these spikes would open the door for new opportunities ([9]), among which are biologically more realistic neuronal networks for studying and describing the information processing in the brain as well as interesting technical approaches for improving the operation of such networks (e.g. low power consumption, fast inference, event-driven information processing, and massive parallelization) ([10]). [11] presented a type of shallow neuronal network that is based on non-negative generative models ([12, 13]). Compared to other models with spiking neurons, this Spike-By-Spike (SbS) network model provides a type of network that requires relatively low additional computational requirements for using spikes as a mean for transmitting information to other neurons. This is due to the fact that in this model time only progresses from one spike to the next, thereby skipping the real time in between spikes. The SbS network has interesting properties like inbuilt sparseness which connects it to compressed sensing and the ability to

massively parallelize the network. However, until now constructing deep SbS networks was not possible since no suitable learning rule existed.

For a network consisting of one input layer and only one hidden layer, an iterative algorithm can be derived ([11]) which has the goal to maximize the likelihood between the observed spikes, generated by the input layer, and an internal representation of the input based on hidden latent variables. The input layer is defined by an input probability distribution $p_\mu(s)$ for input pattern μ to generate a (next) spike at input neuron s. For a given pattern μ in every time step t a spike is drawn at input neuron $s_{t,\mu}$ from $p_\mu(s)$. Time advances from spike to spike, jumping over the real time that has elapsed between these spikes. Based on these observed spikes, the realized input probability distribution $\hat{p}_\mu(s)$ can be calculated over T observed spikes through

$$\hat{p}_\mu(s) = \frac{1}{T}\sum_t^T \delta_{s,s_{t,\mu}} \,. \tag{1}$$

The internal representation $r(s)$ is defined by

$$r_\mu(s) = \sum_i h_\mu(i) W(s|i) \,, \tag{2}$$

where $h_\mu(i)$ are the hidden latent variables and $W(s|i)$ are corresponding weights between the input and the hidden layer. Both commodities are normalized, positive numbers ($0 \leq h(i) \leq 1$ with $\sum_i h_\mu(i) = 1$ and $0 \leq W(s|i) \leq 1$ with $\sum_s W(s|i) = 1$). Summarizing [11], the negative logarithm of the likelihood

$$L = C - \sum_{\mu=1}^{M}\sum_{s=1}^{S} \hat{p}_\mu(s) \log\left(r_\mu(s)\right) \,, \tag{3}$$

is optimized under the condition of positive and normalized elements. C is a constant that is independent of the internal representation.

Optimization of L by gradient descent can be used to derive learning rules for training the weights between the input and hidden layer. More importantly, it also leads to an update rule for the latent variables that are updated with each incoming spike $s_{t,\mu}$:

$$h_\mu^t(i) = \left(\frac{1}{1+\epsilon}\right) \cdot \left(h_\mu^{t-1}(i) + \epsilon \frac{h_\mu^{t-1}(i) W(s_t|i)}{\sum_j h_\mu^{t-1}(j) W(s_t|j)}\right) \tag{4}$$

with ϵ ($\epsilon > 0$) as smoothing constant which results in different level of sparseness on $h_\mu(i)$ depending of its value. Compared to other spiking neuron models ([14]), here the required computational effort for each spike is very low. This is a direct result of completely peculating the temporal structure in between spikes.

The resulting sparseness of $h(i)$ is mainly a result of using non-negative elements. It is known that using non-negative elements is connected to so-called compressed sensing ([15]). Compressed sensing (CS) ([16, 17, 18, 19, 20]) is a method used in technical applications to reconstruct underlying causes from data if these causes are sparse. CS is used to find solutions to otherwise under-determined linear systems using only a minimum of measurements.

While [11] provides suitable learning rules for one layer networks which can also be extended to two layer networks, it doesn't offer learning rules for networks with arbitrary many layers. In this paper a learning rule capable of training the weights for much deeper networks is derived based on the idea of error back-propagation ([8]) which takes into account the presented update rule for the latent variables. After that, the functionality of this new learning rule is demonstrated through several examples with increasing complexity, up to a deep convolutional network used for classifying handwritten digits.

3 Results

3.1 Derivation of the SbS backprop learning rule

For derivation of the learning rule, the error function is defined by the cross entropy

$$E = -\sum_{\mu}^{M}\sum_{d}^{D} t_\mu(d)\log(h^t_{y,\mu}(d)) \tag{5}$$

with $t_\mu(d)$ as the desired output for neuron d in the output layer of the network and $h^t_{y,\mu}(d)$ as the actual output of the network at time t. It is assumed that the output layer has D output neurons as well as that the error is optimized over an ensemble of M patterns. The individual pattern is denoted by μ. Furthermore, $\sum_d^D t_\mu(d) = 1$ and $\sum_d^D h^t_{y,\mu}(d) = 1$ is expected.
In addition, the update rule for $h^t_{y,\mu}(d)$ ([11]) is given by

$$h^t_{y,\mu}(d) = \left(\frac{1}{1+\epsilon}\right)\cdot\left(h^{t-1}_{y,\mu}(d) + \epsilon\sum_{s}^{S}\frac{h^{t-1}_{y,\mu}(d)W^{y-1\to y}(s|d)}{\sum_j^D h^{t-1}_{y,\mu}(j)W^{y-1\to y}(s|j)}h^{t-1}_{y-1,\mu}(s)\right) \tag{6}$$

with $t-1$ denoting latent variables one update step back in the past and $y-1$ for the layer directly before the output layer, which contains S neurons. $W^{y-1\to y}(s|d)$ describe the weights between the layer before the output layer $(y-1)$ and the output layer (y) itself. These weights, like all weights in a Spike-By-Spike (SbS) network, are normalized via $\sum_s W^{y-1\to y}(s|d) = 1$.
Using equation 6 in equation 5 leads to

$$E = -\sum_{\mu}^{M}\sum_{d}^{D} t_\mu(d)\log\left(\left(\frac{1}{1+\epsilon}\right)\cdot\left(h^{t-1}_{y,\mu}(d) + \epsilon\sum_{s}^{S}\frac{h^{t-1}_{y,\mu}(d)W^{y-1\to y}(s|d)}{\sum_j^D h^{t-1}_{y,\mu}(j)W^{y-1\to y}(s|j)}h^{t-1}_{y-1,\mu}(s)\right)\right). \tag{7}$$

For optimizing the weights, the gradient $-\frac{\partial E}{\partial W^{y-1\to y}(u|v)}$ is of special interest. Part of this gradient is

$$\begin{aligned}&\frac{\partial}{\partial W^{y-1\to y}(u|v)}\sum_{s}^{S}\frac{h^{t-1}_{y,\mu}(d)W^{y-1\to y}(s|d)}{\sum_j^D h^{t-1}_{y,\mu}(j)W^{y-1\to y}(s|j)}h^{t-1}_{y-1,\mu}(s) = \\ &\frac{h^{t-1}_{y,\mu}(v)h^{t-1}_{y-1,\mu}(u)}{\left(\sum_j^D h^{t-1}_{y,\mu}(j)W^{y-1\to y}(u|j)\right)^2}\left(\left(\delta_{d,v}\sum_{j'}^{D}h^{t-1}_{y,\mu}(j')W^{y-1\to y}(u|j')\right) - \left(h^{t-1}_{y,\mu}(d)W^{y-1\to y}(u|d)\right)\right)\end{aligned} \tag{8}$$

with $\delta_{d,v}$ as Kronecker delta.
For keeping the further nomenclature shorter,

$$F^t_{a\to b,\mu}(u,v) = \frac{h^t_{a,\mu}(u)h^t_{b,\mu}(v)}{\left(\sum_j h^t_{b,\mu}(j)W^{a\to b}(u|j)\right)^2} \tag{9}$$

$$T^t_\mu(d) = \frac{t_\mu(d)}{h^t_{y,\mu}(d)} \tag{10}$$

$$r^t_{y-1\to y,\mu}(u,d) = h^t_{y,\mu}(d)W^{y-1\to y}(u|d) \tag{11}$$

$$R^t_{y-1\to y,\mu}(u) = \sum_{j}^{D} r^t_{y-1\to y,\mu}(u,j) \tag{12}$$

are introduced.
This leads to the following gradient for optimizing the weights $W^{y-1\to y}$:

$$-\frac{\partial E}{\partial W^{y-1\to y}(u|v)} = \frac{\epsilon}{1+\epsilon}\sum_{\mu}^{M}F^{t-1}_{y-1\to y,\mu}(u,v)\cdot\sum_{d}^{D}T^t_\mu(d)\left(R^{t-1}_{y-1\to y,\mu}(u)\delta_{d,v} - r^{t-1}_{y-1\to y,\mu}(u,d)\right) \tag{13}$$

For updating the weights one layer down $W^{y-2\to y-1}(u|v)$, in equation 7 $h^{t-1}_{y-1,\mu}(s)$ will be replaced by its update rule

$$h^{t-1}_{y-1,\mu}(s) = \left(\frac{1}{1+\epsilon}\right)\cdot\left(h^{t-2}_{y-1,\mu}(s) + \epsilon\sum_k^K \frac{h^{t-2}_{y-1,\mu}(s)W^{y-2\to y-1}(k|s)}{\sum_j^S h^{t-2}_{y-1,\mu}(j)W^{y-2\to y-1}(k|j)} h^{t-2}_{y-2,\mu}(k)\right). \tag{14}$$

This results in

$$\begin{aligned} E &= -\sum_\mu^M\sum_d^D t_\mu(d)\log\left(\left(\frac{1}{1+\epsilon}\right)\cdot\left(h^{t-1}_{y,\mu}(d) + \epsilon\sum_s^S \frac{h^{t-1}_{y,\mu}(d)W^{y-1\to y}(s|d)}{\sum_j^D h^{t-1}_{y,\mu}(j)W^{y-1\to y}(s|j)}\cdot\right.\right. \\ &\left.\left.\left(\frac{1}{1+\epsilon}\right)\cdot\left(h^{t-2}_{y-1,\mu}(s) + \epsilon\sum_k^K \frac{h^{t-2}_{y-1,\mu}(s)W^{y-2\to y-1}(k|s)}{\sum_j^S h^{t-2}_{y-1,\mu}(j)W^{y-2\to y-1}(k|j)} h^{t-2}_{y-2,\mu}(k)\right)\right)\right). \end{aligned} \tag{15}$$

This delivers the basis for calculating the gradient for the weight values $W^{y-2\to y-1}(u|v)$ as follows:

$$-\frac{\partial E}{\partial W^{y-2\to y-1}(u|v)} = \sum_\mu^M\sum_s\left(\frac{\partial h^{t-1}_{y-1,\mu}(s)}{\partial W^{y-2\to y-1}(u|v)}\right)\left(\sum_d \Psi^{t-1}_{y-1\to y,\mu}(s,d)T^t_\mu(d)\right) \tag{16}$$

with

$$\Psi^{t-1}_{y-1\to y,\mu}(s,d) = \left(\frac{\epsilon}{1+\epsilon}\right)\frac{h^{t-1}_{y,\mu}(d)W^{y-1\to y}(s|d)}{\sum_j h^{t-1}_{y,\mu}(j)W^{y-1\to y}(s|j)} \tag{17}$$

and

$$\begin{aligned} &\frac{\partial h^{t-1}_{y-1,\mu}(s)}{\partial W^{y-2\to y-1}(u|v)} = \left(\frac{\epsilon}{1+\epsilon}\right)\frac{\partial}{\partial W^{y-2\to y-1}(u|v)}\sum_k^K \frac{h^{t-2}_{y-1,\mu}(s)W^{y-2\to y-1}(k|s)}{\sum_j^S h^{t-2}_{y-1,\mu}(j)W^{y-2\to y-1}(k|j)} h^{t-2}_{y-2,\mu}(k) \\ &= \left(\frac{\epsilon}{1+\epsilon}\right)F^{t-2}_{y-2\to y-1,\mu}(u,v)\left(R^{t-2}_{y-2\to y-1,\mu}(u)\delta_{s,v} - r^{t-2}_{y-2\to y-1,\mu}(u,s)\right). \end{aligned} \tag{18}$$

This procedure can be iterated until the input layer is reached. This delivers the gradient for the weights

$$\begin{aligned} &-\frac{\partial E}{\partial W^{y-m-1\to y-m}(u|v)} = \sum_\mu^M F^{t-m-1}_{y-m-1\to y-m}(u,v)\cdot \\ &\sum_{q_{y-m}}^{Q_{y-m}}\left(R^{t-m-1}_{y-m-1\to y-m,\mu}(u)\delta_{q_{y-m},v} - r^{t-m-1}_{y-m-1\to y-m,\mu}(u,q_{y-m})\right)\Phi^{t-m}_{y-m}(q_{y-m}) \end{aligned} \tag{19}$$

with for $m > 0$

$$\Phi^{t-m}_{y-m}(q_{y-m}) = \left(\frac{\epsilon}{1+\epsilon}\right)\sum_{q_{y-m+1}}^{Q_{y-m+1}}\Phi^{t-m+1}_{y-m+1}(q_{y-m+1})\frac{h^{t-m}_{y-m+1,\mu}(q_{y-m+1})W^{y-m\to y-m+1}(q_{y-m}|q_{y-m+1})}{\sum_j h^{t-m}_{y-m+1,\mu}(j)W^{y-m\to y-m+1}(q_{y-m}|j)} \tag{20}$$

and for $m = 0$

$$\Phi^t_y(q_y) = \left(\frac{\epsilon}{1+\epsilon}\right)\frac{t_\mu(q_y)}{h^t_{y,\mu}(q_y)} \tag{21}$$

as well as

$$\begin{aligned} F^{t-m-1}_{y-m-1\to y-m,\mu}(u,v) &= \frac{h^{t-m-1}_{y-m-1,\mu}(u)h^{t-m-1}_{y-m,\mu}(v)}{\left(\sum_j h^{t-m-1}_{y-m,\mu}(j)W^{y-m-1\to y-m}(u|j)\right)^2} \\ r^{t-m-1}_{y-m-1\to y-m,\mu}(u,q_{y-m}) &= h^{t-m-1}_{y-m,\mu}(q_{y-m})W^{y-m-1\to y-m}(u|q_{y-m}) \end{aligned}$$

$$R^{t-m-1}_{y-m-1\to y-m,\mu}(u) \quad = \quad \sum_{j}^{Q_{y-m}} r^{t-m-1}_{y-m-1\to y-m,\mu}(u,j)\,.$$

Like in other backprop rules, the error is back-propagated from the output layer to the input layer but here also in time (one iteration back in time for each layer closer to the input).
Using this gradient, the weights in all the examples in this paper are updated as follows

$$\begin{aligned} V^{y-m-1\to y-m}(u|v) &= W^{y-m-1\to y-m}(u|v)\left(1-\frac{\gamma}{S}\frac{\partial E}{\partial W^{y-m-1\to y-m}(u|v)}\right) \\ W^{new,y-m-1\to y-m}(u|v) &= \frac{V^{y-m-1\to y-m}(u|v)}{\sum_j V^{y-m-1\to y-m}(j|v)} \end{aligned} \tag{22}$$

with $0<\gamma<1$ as learning rate and

$$S=\max\left(\left|\frac{\partial E}{\partial W^{y-m-1\to y-m}(u|v)}\right|\right) \tag{23}$$

as scaling factor for ensuring the non-negativity of the weight values. Alternatively the gradient could be used for L4 ([21]) or Adam ([22]). However, in the following examples we focus on the pure gradient for understanding its properties instead of attempting to optimize the classification performances of the networks.

3.2 Learning the XOR function

The first example is a two layer network (see figure 1a) with four neurons in the input layer, four neurons in the hidden layer and two neurons in the output layer. The task of the network is to realize the XOR function, which receives two bits of input and outputs zero if the values of both bits are the same or one if both bits have different values. The input layer consists of two bits while every bit is represented by a group of two neurons. The first neuron in a group is only active if the input bit has a value of zero. The second input neuron of the group is only active if the input bit has a value of one. In every time step one spike is drawn from the input pattern distribution which is represented by the input neurons and send to the hidden layer. The hidden layer consists of one normalization group (for which $\sum_i h(i)=1$) with four neurons. In the figure, the non-zero weights between the input and the hidden layer are shown and have a value of 0.5 each. Given these weights and the SbS dynamics for $h(i)$, after processing enough input spikes, only one hidden neuron will remain active due to the competition within a normalization group. The corresponding input patterns, which lead to an activation of the neurons is listed in the hidden neurons in figure 1a. The probability distribution stored in the hidden neurons is also used to draw one spike in every time step. The spikes from the hidden layer are send to the output layer. The output layer processes incoming spikes according to the h-dynamic using the weight values between the hidden and output layer as shown in the figure. For decoding the result of the information processing, the output neuron with the higher value in its latent variable is selected. The first output neuron represents an output of zero and the second output neuron an output of one.
For the first test, the weights in this network were randomly initialized. This was done by first setting all values to one, then adding 1% of positive random uniform noise and subsequently normalizing the weights. Before presentation of an input pattern $p_X(i)$, the latent variables of the hidden neurons $h_{H1}(i)$ and the output neurons $h_{HY}(i)$ were set to their uniform distributions. Then for every time step, one spike each is drawn from p_X and h_{H1}. This is done by comparing the cumulative sum of these probability distributions with random numbers drawn from an uniform distribution with $[0,1[$. These spikes are then used to update $h_{H1}(i)$ and $h_{HY}(i)$ (using $\epsilon=0.1$ in the update process). The cycle of drawing spikes and updating the latent variables of the hidden and output layer is performed with the same input pattern until a given number of spikes has been processed. Using equation 19 the gradient for this pattern is calculated and stored. After collecting these contributions for all four patterns, equation 22 is applied to update the weights. However, before normalizing $V^{y-m-1\to y-m}(u|v)$, it is ensured that the smallest value in V is 0.0001. This prohibits that equation 22 is getting stuck by zero values, due to its multiplicative nature.

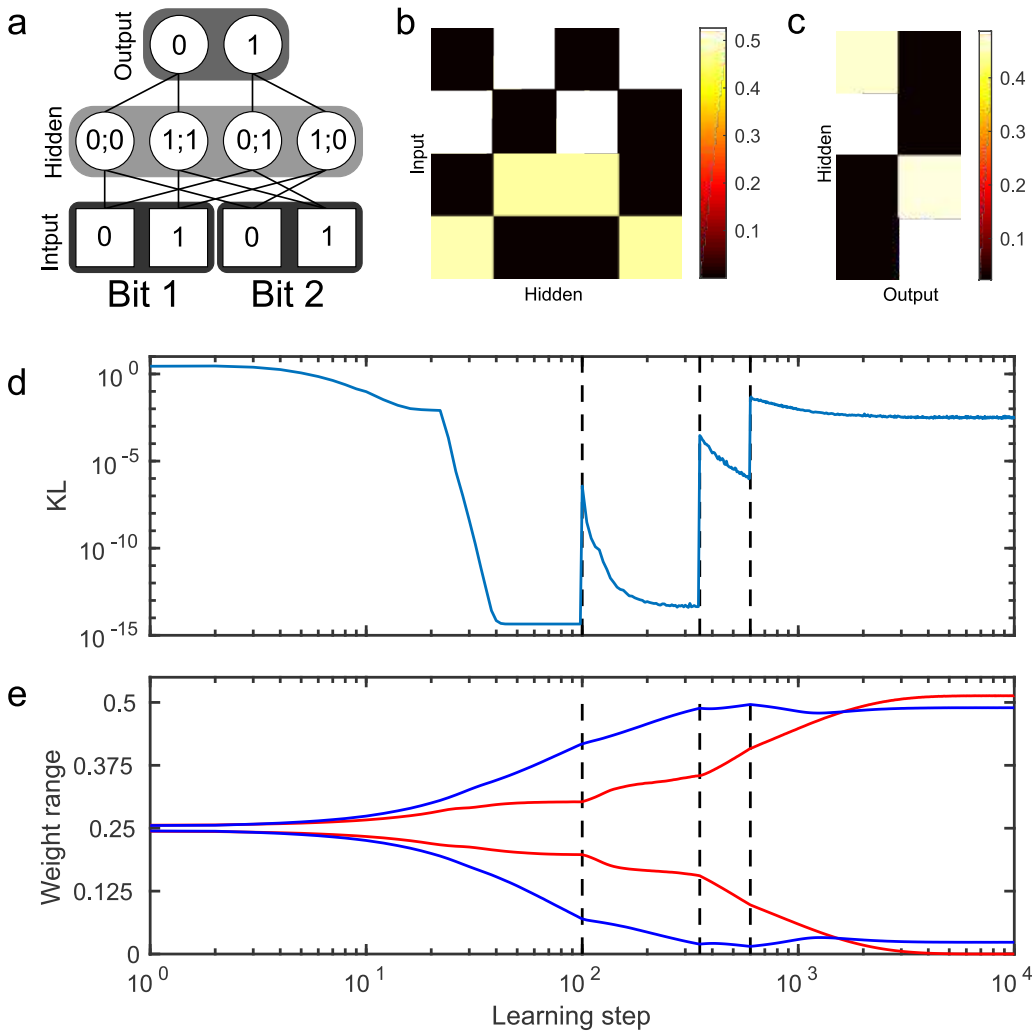

Figure 1: **Learning the XOR function.** a.) Structure of a SbS network for solving the XOR function. b.) - e.) Starting with random weights and using the backprop rule for SbS networks, weights are learned. b.) and c.) One example for learned weights. d.) Kullback-Leibler divergence over all possible input/output patterns between the correct output and the probability distribution of the output neurons. e.) Minimum and maximum values of the weight matrices between the input and the hidden layer (red lines) and the hidden and output layer (blue lines). The vertical dashed lines show when parameters of the network are changed. These parameter changes are done to increase the range of used weight values. Otherwise the network is, due to its spareness properties, able to solve the task perfectly with small changes from the initial random weights. For d.) and e.) the curves are an average over 250 different initial conditions, which all lead to prefect results.

Figure 1d shows the quality of the output during learning. As measure the Kullback-Leibler (KL) divergence between the expected output and the real output is calculated.

$$KL = \sum_{\mu}^{M} \sum_{d}^{D} t_{\mu}(d) \log \left(\frac{t_{\mu}(d)}{h_{HY,\mu}(d)} \right) \tag{24}$$

Compared to the cross entropy measure, which was used for developing the SbS backprop rule, the KL adds a reference which results in zero if and only if both distributions are the same. Due to limiting the smallest weight value to 0.0001, it is not possible to reach a KL divergence of 0. Instead, here the smallest KL is $\approx 4 \cdot 10^{-15}$. Using 1024 spikes per input pattern and learning step as well as $\gamma = 0.025$, we reach the minimum KL after 14 learning steps (see figure 1d). Analyzing the magnitude of the weight values (figure 1e), reveals that the weights only changed a small amount from randomly initialized weights. The changes for the weights between the hidden layer and the output layer (blue lines, after 100 learning steps: $\Delta = 0.35$) are stronger than the changes for the weights between the input and the hidden layer(red lines, after 100 learning steps: $\Delta = 0.1$). This is a result of the competition inside of the hidden as well as the output layer. Already small asymmetries can be used to solve

the task correctly.

In the following we wanted to know if the weights would approach the mathematically optimal weights (see figure 1a) if the task requires the network to be more robust against noise in its information processing. Thus we reduced the number of spikes (as well as γ) by half after 100, 350 and 600 learning steps (denoted by the vertical dashed lines in figure 1e and d). Due to the increased amount of noise, the KL doesn't decrease as much as before. However, the weights values start to spread over a larger range of values. In the end the weights are approaching the mathematical weights. Figure 1b and c show an example of weights as a result of the learning procedure.

We tested 250 initial conditions (i.e. seeds for the simulations). Figure 1e and d are an average over all these 250 instances. Furthermore, we re-checked all 250 weight sets with 1024 spikes per pattern and found that all resulted in the smallest KL.

3.3 Ignoring irrelevant information

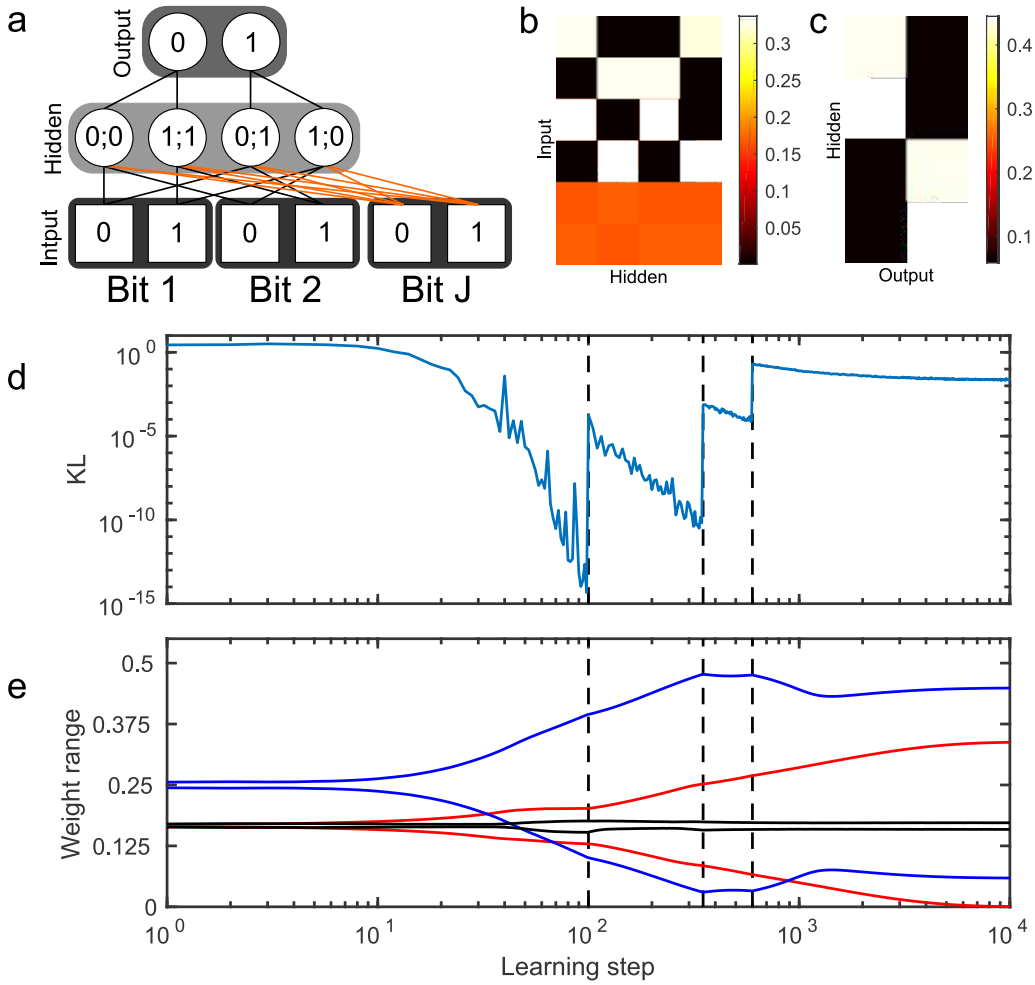

Figure 2: **Learning the XOR function with an additional uncorrelated input bit.** Similar to Figure 1. However, in this instance, two additional input neurons are present which represent an input bit J that is uncorrelated to the XOR task. Applying the same learning procedure shown in Figure 1, the learning process results in weights shown in b.) and c.). The main difference is that the weight values, corresponding to the bit J, deliver the same input to all the neurons in the hidden layer. Thus the competition between the hidden neurons ignores this kind of irrelevant input. e.) The black lines represent the development of the minimum and the maximum of the weight values corresponding to bit J and the red lines represent the minimum and maximum of the weight values for bit 1 and bit 2. For d.) and e.) the curves are an average over 250 different initial conditions, which all lead to prefect results.

In a second test we investigated if the SbS backprop rule can be used to learn a task correctly even if part of the input is not contributing to solving the task. Instead this part of the input introduces random unrelated

information into the system that needs to be ignored. Thus we revisited the XOR network and added one random bit J to the input pattern. Every time the real input bit pattern is changed also this bit is set to a new random value (equal random chance for zero or one). Figure 2 shows the modified network structure. The same learning procedure as well as the same analysis from the XOR example was performed.
The result is that the weights between the two neurons encoding bit J and the hidden neurons show roughly the same values (see figure 2b and c for example weights). In terms of the winner-take-all competition inside the hidden layer, a uniform input is ignored due to the normalization of the latent variables. The development of KL divergence and the used weight value range (see figure 2d and e) shows similarities to figure 1, albeit slower (1/3 of all input spikes are lost to the meaningless bit J) and noisier.

3.4 Learning the 4 bit parity function

The 4 bit parity function is an extension of the 2 bit XOR function. This function counts the number of its input bits with value one. Then it outputs one if the count is odd or zero if the count is even. A SbS network able to realize this function has four layers (see figure 3a): Input layer X with 8 neurons, which encodes the input in a similar fashion to the XOR network but via four groups with two neurons each. Hidden layer $H1$ with also 8 neurons and hidden layer $H2$ with four neurons as well as the output layer with two neurons. Besides the network structure, the procedure is as described in the example for the XOR network.
Concerning learning of weights, for 250 different initial seeds independent simulations were performed. Again, starting with randomly initialized weights the learning process goes through different combinations of numbers of spikes per pattern and γ learning rates. Subsequently 1024, 512, 256 and 128 spikes per pattern were used for 7000 learning steps each. During learning, γ is divided by 2 after every 1000 learning steps and every time the number of spikes per pattern is changed γ is reset to 0.03. Figure 4 shows the development of the average KL divergence as well as the average weight value range for the three weight matrices over the described learning process.
If learning was successful, the minimal KL divergence will be at $\approx 1.8 \cdot 10^{-14}$. This is due to the restriction on the smallest weights values (see the XOR example above for details). Figure 3i shows the KL distribution for the trials that didn't end up with weights that can solve the task correctly. After training with 1024 spikes per pattern and 7000 learning steps, still 135 of the initial seeds haven't found a correct solution (blue line). Taking these weights as starting point and running another 7000 learning steps but with 512 spikes per pattern, reduces this number to 16 erroneous sets of weights (orange line; for a fair comparison we tested the quality of the weights always with 1024 spikes per pattern). Using the weight sets from the 512 spikes per pattern run for an additional 7000 learning steps with 256 spikes per pattern ends in an increase of bad weight sets (error count is 28; yellow line). And finally these weights were used for another 7000 learning steps with 128 spikes per pattern. This ends up in 139 broken weight sets. In summary, it is important to optimize the number of spikes per pattern according to the task. Figure 3b (weights between input layer X and hidden layer $H1$), c (weights between hidden layer $H1$ and hidden layer $H2$) and d (weights between hidden layer $H2$ and output layer HY) shows example weights for a working set of weights after 14000 learning steps. Comparing the structure of these weights with the mathematical solution shown in figure 3a, revels that the competition within the layers $H1$, $H2$ and HY allows for additional solutions.
For the next test we modified the network structure (see figure 3e). Analyzing the original network structure shown in figure 3a reveals that the hidden layer $H1$ can be split into two normalization groups $H1a$ and $H1b$ which contain four neurons each. Every group has its own competition by normalization. With the old structure three spike were drawn from X, $H1$ and $H2$ for every time step. With the new structure five spikes are drawn in every time step: One spike from Xa, Xb, $H1a$, $H1b$ and $H2$ each. The weights between Xa and $H1a$ as well as the weights between Xb and $H1b$ are the same. The contributions from SbS backprop learning are averaged. One example for these shared weights after the whole learning process (4x 7000 learning steps) are shown in figure 3f. The corresponding weights between hidden layer $H1$ ($H1a$ and $H1b$ joined by concatenation and divided by 2 for maintaining the normalization condition) and hidden layer $H2$ are shown in figure 3g as well as the weights between hidden layer $H2$ and output layer HY are shown in 3h. Figure 5 shows the development

of the average KL divergence as well as the average weight value range for the three weight matrices during learning. In contrast to the original network structure, this convolution inspired structure reacts favorably to additional learning runs with lower number of spikes per pattern (see figure 3j). After the run with 1024 spikes, only 87 of 250 weight sets are erroneous. Adding a second run with 512 spike per pattern reduces this number to 17. An additional run with 256 spikes reduces the number even further to 7. And finally the run with 128 spikes ends up at only 4 bad weights sets.

As final test for the 4 bit parity function, we investigated how important the retardation of the H-values during learning is. Instead of using the H-values from the appropriate earlier times like it is defined by the derived SbS backprop rule, we only used the H-values from the same time step (after all spikes for that pattern have been processed into updated H-values). We made this small modification to the simulation for the network structure shown in figure 3e and re-run the simulations. Figure 3k shows the results. Besides the first run with 1024 spikes per pattern which resulted in 80 of 250 bad weights sets, we found a small decrease in performance. This can be seen in 30 (after learning with 512 spikes per pattern), 12 (after learning with 256 spikes), and 8 (after learning with 128 spikes) erroneous weights sets respectively. However, this difference is small and indicated that the retardation of H can be neglected if this is advantageous due to other factors.

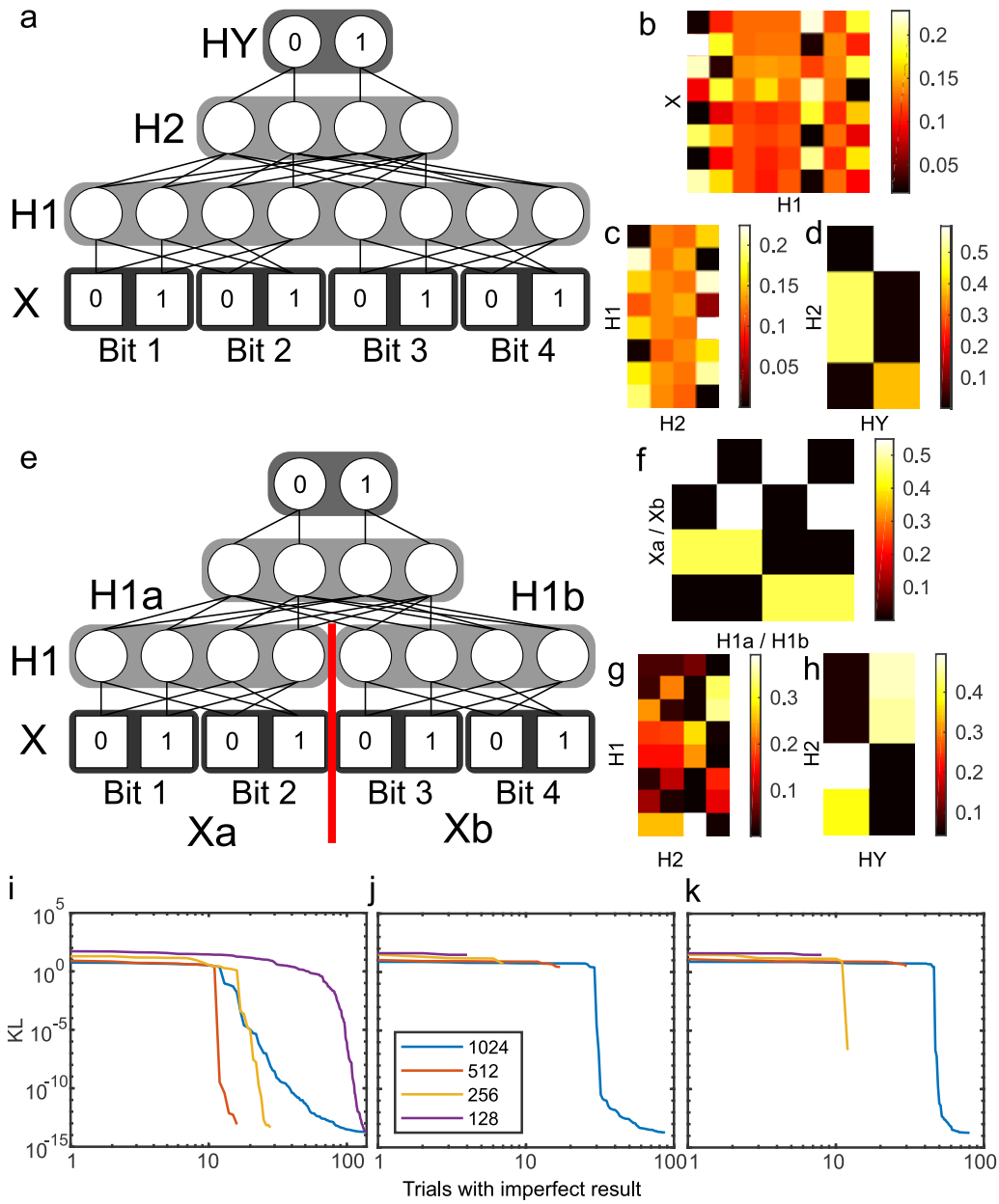

Figure 3: **4 bit parity function.** a.) Structure of a SbS network able to solve the 4 bit parity task. One set of example weights after 14000 learning steps, that solve the task correctly: b.) weights between input layer X and hidden layer $H1$, c.) weights between $H1$ and $H2$ and d.) weights between $H1$ and HY. e.) Variation of the structure of the network. The hidden layer $H1$ was split into two normalization groups $H1a$ and $H1b$ with 4 neurons each. One set of example weights after the whole learning process (4x 7000 learning steps), that solve the task correctly: f.) weights between input layer X and hidden layer parts $H1a$ as well as $H1b$ (both normalization groups share the same weights), g.) weights between the whole $H1$ layer and $H2$ and h.) weights between $H1$ and output layer HY. i.) - k.): KL divergence for the weight sets that didn't result in a minimal KL value, ordered by size. The weights were tested with 1024 spikes per pattern. i.) for the network structure a.). j.) and k.) for the network structure e.). For learning the weights for k.), the temporal part of the SbS backprop rule was removed and all the H-values used during learning were taken after the last spike for a pattern was processed.

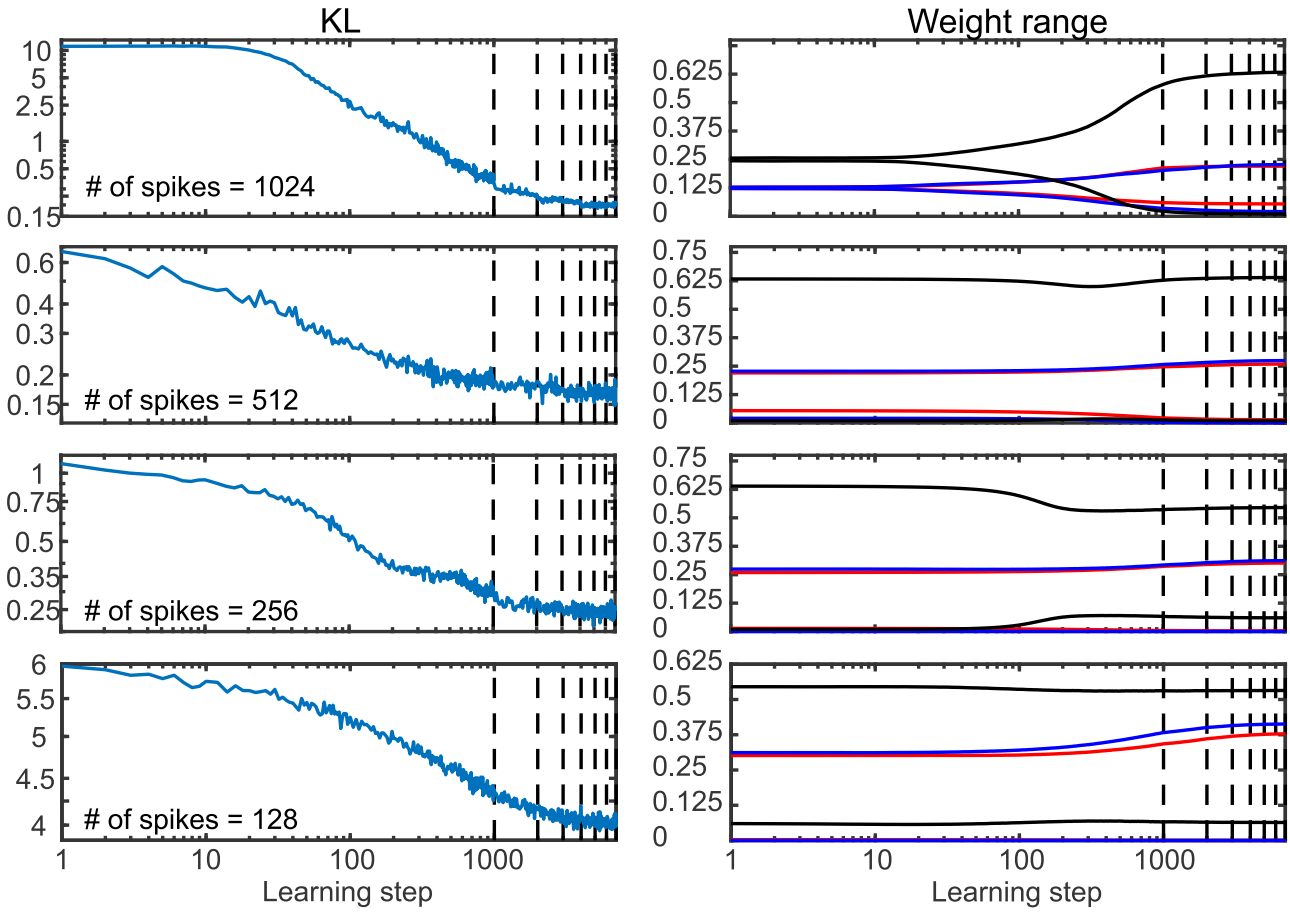

Figure 4: **Learning the 4 bit parity function with structure shown in figure 3a.** (left) Average KL divergence over 250 different initial seeds. In the upper row, the learning starts with randomly initialized weight sets then 7000 learning steps are done with 1024 spikes per pattern. For the first 1000 learning steps, γ was set to 0.03. Then γ was divided by 2. Again, after another 1000 learning steps, γ was divided by 2. This procedure is continued for the remaining 5000 learning steps, where always after another 1000 learning steps γ is divided by 2. The vertical dashed lines symbolize the points during learning when γ was divided by 2. The weights that result after 7000 learning steps were then used as starting weights for another 7000 learning steps with the same γ development but with 512 spikes per pattern. The corresponding KL divergence curve is shown in the second row. Learning was continued in the same fashion with 256 and then 128 spikes per pattern which resulted in the two lower rows, respectively. (right) Development of the maximal and minimal weights value for the three weight sets ($X \to H1$: red line, $H1 \to H2$: blue line, $H2 \to HY$: solid black line) during the learning process, averaged over the 250 initial seed conditions.

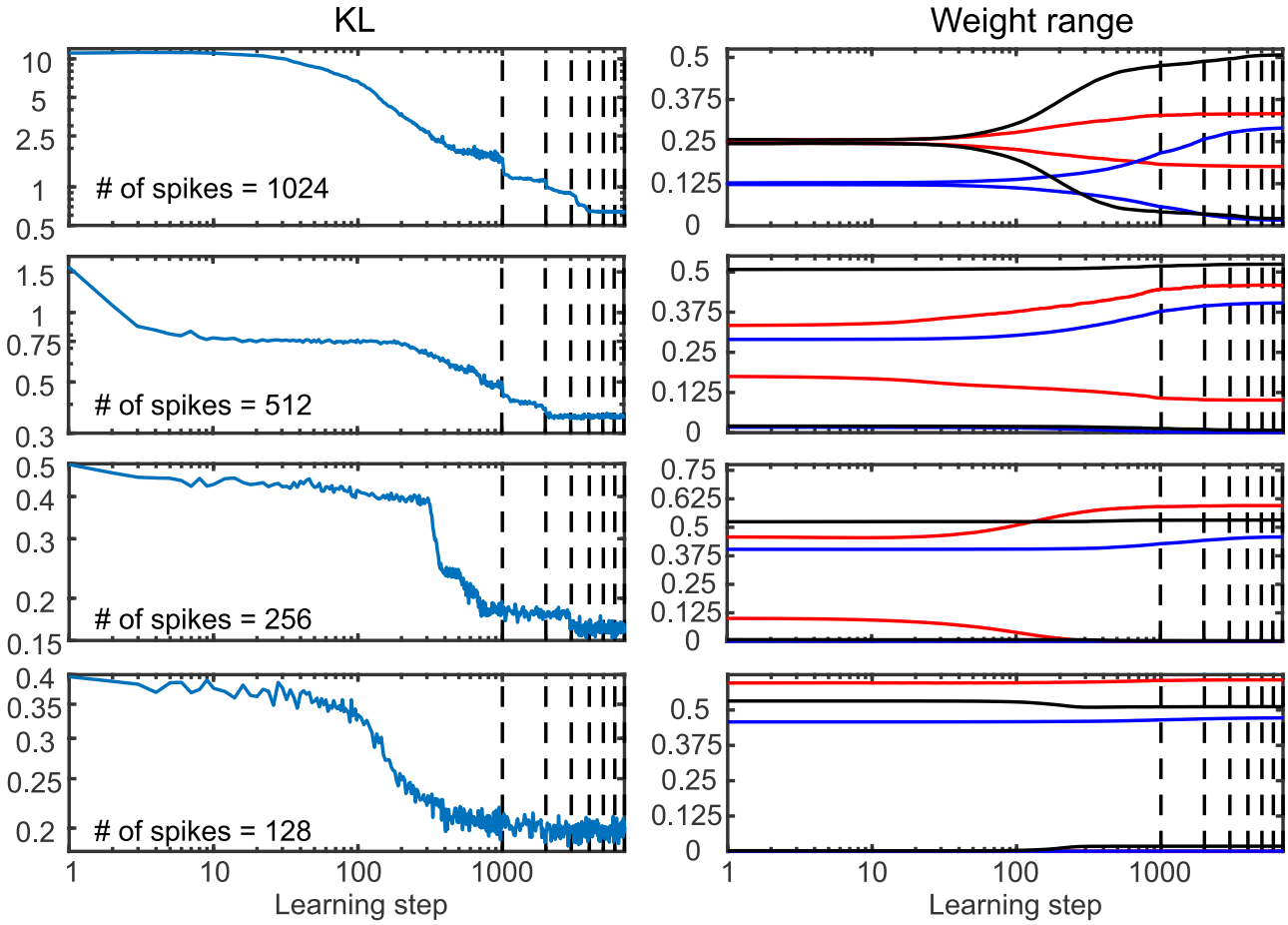

Figure 5: **Learning the 4 bit parity function with structure shown in figure 3e.** Like figure 4 but simulated with the network structure shown in figure 3e.

3.5 Deep convolutional network (MNIST)

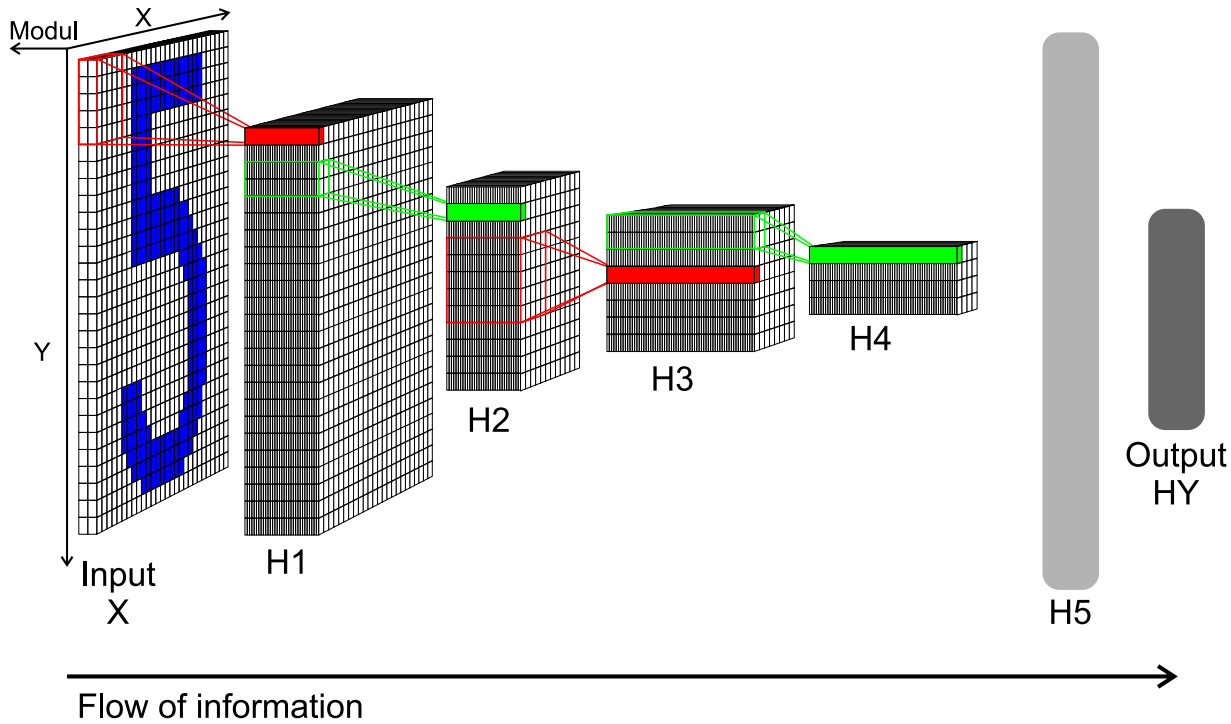

Figure 6: **Network structure of the convolution network for the MNIST data.** Input X: Input layer with 28 x 28 normalization modules for 28 x 28 input pixel. Each module has two neurons realizing a simplified version of on/off cells for enforcing positive activity also for low pixel values. From this layer spikes are send to the layer $H1$. $H1$: Convolution layer $H1$ with 24 x 24 normalization groups with 32 neurons each. Every normalization group processes the spikes from 5 x 5 blocks of normalization groups from the input layer (x and y stride is 1). $H2$: 2 x 2 pooling layer $H2$ (x and y stride is 2) with 12 x 12 normalization groups with 32 neurons each. The weights between $H1$ and $H2$ are not learned but set to a fixed weight matrix that creates a competition between the 32 features of $H1$. $H3$: 5 x 5 convolution layer $H3$ (x and y stride is 1) with 8 x 8 normalization groups. Similar to $H1$ but with 64 neuron for each normalization group. $H4$: 2 x 2 pooling layer $H4$ (x and y stride is 2) with 4 x 4 normalization groups with 64 neurons each. This layer is similar to layer $H2$. $H5$: Fully connected layer $H5$. E.g. 1024 neurons in one big normalization group which are fully connected to layer $H4$ and output layer HY. HY: Output layer HY with 10 neurons for the 10 types of digits. For decoding the identity of the neuron with the highest activity is selected.

The MNIST database is a benchmark for machine learning (see `http://yann.lecun.com/exdb/mnist/`). It consists of handwritten digits with 28 x 28 pixels and 256 gray values per pixel. The database contains 60000 examples for training the network and 10000 examples for testing the performance of the network. As part of the tutorial for the Google TensorFlow machine learning software, a convolutional neuron network for classifying these handwritten digits is presented `https://www.tensorflow.org/tutorials/estimators/cnn`. In their example they present a simple network learned via back-propagation based on a simple gradient descent optimization method. The performance for this network is listed with 97.3% classifications correct (running this simulation will result in variation around this value due to randomness). Replacing the simple gradient decent by L4 ([21]) or Adam ([22]) as well as creating more training examples by shifting and rotating the existing training data, allows this network to reach a performance of over 99% classifications correct.
The network in the TensorFlow tutorial is structured as follows: The input layer X consists of 28 x 28 elements representing the picture of a handwritten digit. The input layer is followed by a first hidden layer $H1$ which performs a convolution (with stride 1 and zero padding for keeping the size at 28 x 28 pixel after the convolution) through a kernel with the size of 5 x 5 with 32 filters. $H1$ is followed by a max pooling layer $H2$, which calculated the maximum over a 2 x 2 segment with stride of 2 from $H1$'s output. Layer $H3$ is again a convolution layer like $H1$ but looking at the output of $H2$ and with 64 filters instead. $H4$ is a max pooling layer to $H3$. After $H4$, a fully connected layer $H5$ with 1024 neurons is positioned. And finally the output of the network can be read out

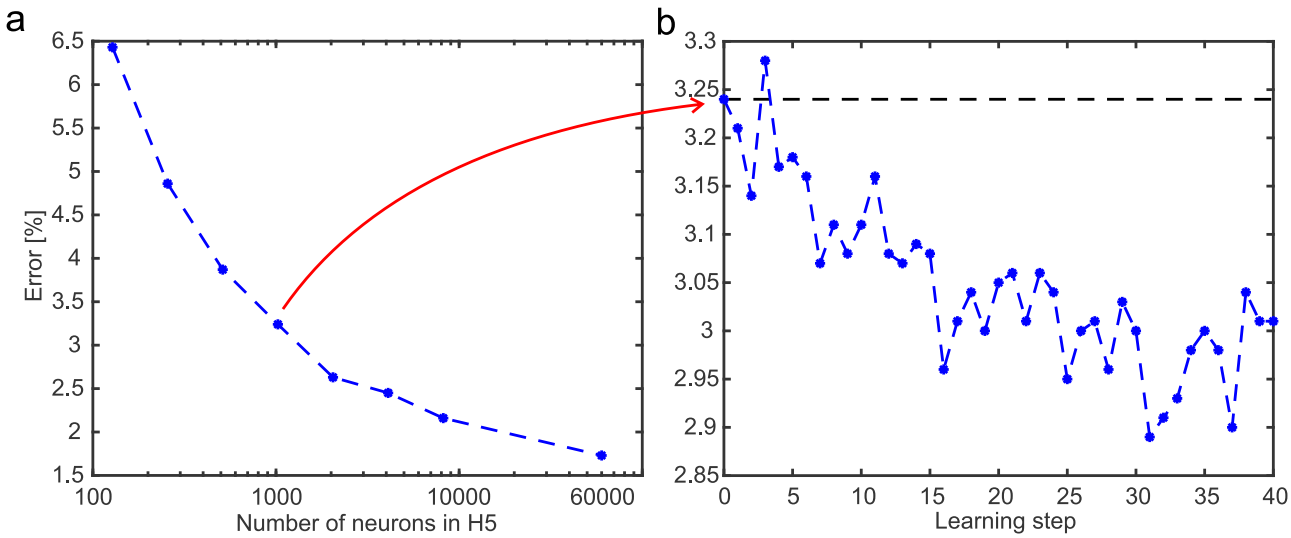

Figure 7: **Performance values for the MNIST benchmark.** a.) Error after pre-learning, in dependence of the number of neurons in layer $H5$. b.) Error for the network with 1024 neurons in $H5$ under SbS backprop learning. At step 0 (as well as by the dashed line) the error from the pre-learning is shown as reference.

from the output layer HY. HY consists of 10 neurons, where each neuron represents one class of the 10 digits. The classification result is decoded by calculating the argmax from HY's neurons.

This network structure is mimicked by a SbS network. Since the computational complexity of the SbS network is orders of magnitude bigger, we simplified the TensorFlow example network. We removed the zero padding in both convolutional layers of the network. Thus layer X still has 28 x 28 pixels but layer $H1$ decreases to 24 x 24 with 32 filters. $H2$ halves the size to 12 x 12. Convolution layer $H3$ compressed the output of $H2$ to 8 x 8 with 64 filters. Its max pooling layer halves it again to 4 x 4. We keep the 1024 neurons for the fully connected layer $H5$ as well as the 10 neurons for the output layer. We used TensorFlow to train this reduced network with a simple gradient descent optimization and got a classification correct performance of 97.1%.

For transferring this reduced network into a SbS network (see figure 6), the following adjustments were made:

Input layer X: For the input a so called on/off split was made ([11]), which results in two channels per pixel. This is very similar to the representation of a bit by two neurons in the XOR example. This transformation can be described by $I_{ON}(x,y) = f(2P(x,y)-1)$ and $I_{Off}(x,y) = f(1-2P(x,y))$ with $f(\cdot)$ as a threshold linear function which sets all negative values to 0 and passes on all positive values without change. $P(x,y)$ is the pixel values at the position x and y with $P(x,y) \in [0,1]$. Next we used a 5 x 5 selection window and moved it with stride 1 in x and y over $I_{ON}(x,y)$ and $I_{Off}(x,y)$. This resulted in 24 x 24 groups of with 50 values each. In a last step we normalized the values in every one of these groups individually to get 24 x 24 input probability distributions $p_X(s,x,y)$ (with $s \in [1,50]$ and $\sum_s p_X(s,x,y) = 1$). In every time step, one spike is drawn from each of these input probability distributions generating 24 x 24 input spikes per time step which are sent to layer $H1$.

Convolution layer $H1$: The hidden layer $H1$ consists of 24 x 24 independent normalization groups with 32 neurons per group each ($h_1(i,x,y)$ with $i \in [1,32]$ and $\sum_i h_1(i,x,y,) = 1$). These 24 x 24 normalization groups share the same weights. Every one of these $H1$ groups receives spikes from its direct counterpart from the input layer at the same x and y coordinates (The 5 x 5 convolution with stride one shown in figure 6 is executed by building 24 x 24 groups from the original picture in input layer $X1$.). In every time step, one spike each is drawn from every one of the 24 x 24 groups, delivering 24 x 24 spikes to layer $H2$.

Pooling layer $H2$: Instead of using max functions, the pooling layer uses only the inherent competition implemented by the SbS update rule. This is done via the kernel's 2 x 2 weight matrix $W^{H1\to H2}(s,x,y|i) = \frac{\delta_{s,j}}{4}$ (with $x \in [1,2]$ and $y \in [1,2]$) using the h-update rule. These weights are fixed and not learned. A 2 x 2 spatial patch from $H1$ with its 2 x 2 x 32 neurons delivers input to one normalization group in $H2$ which has also 32

neurons. The structure of the weights ensures that the input from different features (i.e. the 32 features that are represented by the 32 neurons in the normalization groups) do not mix. Thus the combined spatial inputs from the 32 features compete against each other. Only the features with strong inputs are represented in the corresponding $H2$ normalization group. This pooling kernel is applied with stride 2, which results in 12 x 12 spatially non-overlapping normalization groups with 32 neurons each. The next layer $H3$ gets 12 x 12 spikes per time step as input.

Convolution layer $H3$: While the convolution in $H1$ was simply done by reorganizing the input, in layer $H3$ the convolution is realized by using a 5 x 5 spatial region of layer $H2$ as input. This funnels the spikes generated from these 5 x 5 x 32 neurons from $H2$ into one normalization group of $H4$ with 64 neurons. Due to a stride of one, this results in a total of 8 x 8 normalization groups in $H3$. Thus layer $H3$ delivers 8 x 8 spikes to layer $H4$. The weight matrix is shared among all the normalization groups in this layer.

Pooling layer $H4$: Layer $H4$ is similar to layer $H2$. The difference is that in $H4$ there are 64 neurons in each normalization group. $H4$ consists of 4 x 4 normalization groups, delivering 4 x 4 spikes per time step to layer $H5$.

Fully connected layer $H5$: Layer $H5$ is one large normalization group with e.g. 1024 neurons, which generates one spike per time step. It processes all spikes produced by the whole layer $H4$.

Output layer HY: The output layer is one normalization group with 10 neurons, where each neuron represents one class of the handwritten digits. This group takes the spikes from layer $H5$ as its input and updates its h-values according the h update rule. For decoding the results of the classification, the neuron with the highest h value is selected. The corresponding digits connected to this neuron is used as result of the classification task.

Pre-learning: Before applying the SbS backprop learning rule, we pre-learned the weights in an unsupervised manner starting with randomly initialized weights (see the XOR example for details). During pre-learning only one set of weights is trained at a time. This is done by reducing the whole network into several networks with one input layer and one hidden layer ([11]). Beginning with the weights between the input layer X and the first hidden layer $H1$, for each of the 60000 input pattern this one layer network ran for 1000 spikes per input group with $\epsilon = 0.11111$ and then for 200 additional spikes with $\epsilon = \frac{0.11111}{25}$ for smoothing the h - distribution. As result we got a h - vector for every one of the training patters. Then we applied the following self-organizing learning rule from [11] for updating the weights:

$$V(s|i) = \sum_{\mu,x,y} \hat{I}_{\mu,x,y}(s) \frac{W(s|i)h_\mu(i,x,y)}{\sum_i W(s|i)h_{\mu,x,y}(i)} \tag{25}$$

$$W^{new}(s|i) = \frac{V(s|i)}{\sum_s V(s|i)} \tag{26}$$

$I_{\mu,x,y}(s)$ represents the input distribution which is used to generate the spikes that are sent to the normalization group in $H1$ with their $h_{\mu,x,y}(i)$ (from now on denoted by $h_{H1,\mu,x,y}(i)$ to keep track of the layers). $\hat{I}_{\mu,x,y}(s)$ is calculated by counting the input spikes drawn from $I_{\mu,x,y}(s)$ and then converting it into a probability distribution by normalization. Beside of averaging over all 60000 patterns μ, the contributions from all the 24 x 24 spatial positions are averaged too. The minimum value for V was set to 0.05 divided by the number of input neurons. Learning is repeated for 20 steps. After that the network is simulated a 21th time and the resulting $h_{H1,\mu,x,y}(i)$ are stored. Now this collection of 60000 $h_{H1,\mu,x,y}(i)$ is used as input $I_{H1,\mu,x,y}(s)$ for a second one layer network with layer $H2$ ($h_{H2,\mu,x,y}(i)$) as hidden layer. Running this network once with the fixed pooling weights propagates the input $I_{H1,\mu,x,y}(s)$ one layer further to $h_{H2,\mu,x,y}(i)$. Again we use $h_{H2,\mu,x,y}(i)$ as $I_{H2,\mu,x,y}(s)$ in a third one layer network where the hidden layer is $H3$. These weights are trained for 20 learning steps and then with the 21th run of the simulation, the input $I_{H3,\mu,x,y}(s)$ is calculated. $I_{H3,\mu,x,y}(s)$ is propagated further via pooling layer $H4$ with its fixed weights into $I_{H4,\mu,x,y}(s)$. Up to this layer the learning procedure was unsupervised.

For the last two layers $H5$ and HY, pre-learning is slightly modified. $I_{H4,\mu,x,y}(s)$ defines the input into $H4$ and the digit for a given pattern defines $h_{HY,\mu,x,y}(i)$. All $h_{HY,\mu,x,y}(i)$ are zero except the one neuron which is associated to the digit for that paper. This neuron's $h_{HY,\mu,x,y}(i)$ is set to one. For training, like shown in [11], such a two layer network can be converted into a one layer network by folding down $h_{HY,\mu,x,y}(i)$ and

using it as additional input $I_{HY,\mu,x,y}(s)$ to $H5$. In every time step, we now draw 4 x 4 spikes from $I_{H4,\mu,x,y}(s)$ and one spike from $I_{HY,\mu,x,y}(s)$. This allows us to train the weights $W^{H4\to H5}$ and $W^{HY\to H5}$ simultaneously for 20 learning steps like it was done for the other layers before. However, for testing the performance of this network the weights $W^{H5\to HY}$ are required. We calculate them from $W^{HY\to H5}$ by transposing the matrix and re-normalizing it afterward.

In figure 7a shows the classification error that the full multi-layer network, using the pre-learned weights, achieves. For determining the performance 1200 spikes per each of the 10000 test pattern were used. ϵ was changed to a base value of 0.4 and hence compensated for the amount of spikes received by the layers in one time step. Thus we used $\epsilon = \frac{\hat{\epsilon}}{1-\hat{\epsilon}}$ with $\hat{\epsilon}_{X\to H1} = 0.4$, $\hat{\epsilon}_{H1\to H2} = \frac{0.4}{4}$, $\hat{\epsilon}_{H2\to H3} = \frac{0.4}{25}$, $\hat{\epsilon}_{H3\to H4} = \frac{0.4}{4}$, $\hat{\epsilon}_{H4\to H5} = \frac{0.4}{16}$, and $\hat{\epsilon}_{H5\to HY} = 0.4$ for the first 1000 spikes. And $\epsilon = \frac{1}{25}\frac{\hat{\epsilon}}{1-\hat{\epsilon}}$ was used for the last 200 spikes for smoothing the h-distributions. Every normalization group in this network produced one spike each in every time step. In figure 7a the error values are shown for hidden layers $H5$ with different number of neurons. For the TensorFlow example with 1024 neurons we get a classification performance of 96.8%. For the special case of 60000 neurons in layer $H5$ the number of neurons matches the number of patterns to be classified. Here, learning of the final layers of weights can be omitted. For each pattern μ we selected a different $H5$ neuron and copied the input distributions $I_{H4,\mu,x,y}(s)$ into its input weights which were subsequently normalized. Thereby the pattern number was used to select each of the 60000 neurons in $H5$ as a template which in turn was then connected to the corresponding class in HY and these weights were also normalized.

SbS Backprop Learning: Based on the weights from pre-learning with 1024 neurons in layer $H4$, several learning steps with the SbS backprop rule were done. For each weights update, the network ran with the 60000 training pattern in the full network with the parameters used for figure 7a. All the weights (except the fixed pooling weights) were optimized at the same time, while the minimum value for V was 0.0001, as in the earlier examples. Similar to the last example of learning the 4 bit parity function, we only used all the h - values after processing the last spike (not the time retarded h - values from earlier spikes). The learning rate γ was set to 0.025. We tested the classification error of the network during learning with the test data in the described fashion. Figure 7b shows the development of the classification error over the learning steps. Compared to the results of the corresponding TensorFlow network (2.9% error) which also only used the simple gradient, performances after SbS backprop are similar. Here it needs to be stressed that, due to the high computational requirements, neither γ (used from the XOR example) nor ϵ (0.4 was used instead of 0.1 because the information is transmitted faster through the network but under the caveat of stronger fluctuations in the h-distributions) were optimized.

4 Discussion

In this paper we presented an algorithm for learning weights in deep Spike-by-Spike networks ([11]) via error back-propagation. We were able to show that the algorithm can be used to learn the weights with a given network structure from scratch (i.e. randomly initialized weights) as well as to ignore non task relevant information. We showed that the algorithm can be used to train weights that were several layers away from the output.

Besides the new SbS backprop method, we showed how to build neuronal networks with convolutional and pooling layers with the same basic elements. No special maximum functionality as usual in deep convolutional networks was required. Furthermore, we presented a strategy to pre-learn weights in a convolutional neuronal network using an unsupervised method before applying the SbS backprop rule.

The SbS network is a generative model with latent variables as well as purely non-negative components which uses only spikes to transmit information. In the literature there are many papers concerned with learning (deep) generative models with error back-propagation (e.g. [12, 13, 23, 24, 25, 26, 27]) as well as similar papers for training networks that use non-negative matrix factorization (e.g. [28, 29]). In contrast to these approaches, the present framework conserves the specific spike driven update dynamics from [11]. This update rule is of special interest because it results in an extremely simple yet biologically plausible neuronal network that uses only spikes as signals. This allows to build special computational hardware that can be massively parallelized ([30]) and exhibits sparse representations that are know from compressed sensing ([19, 20]) and are a leading

hypothesis for understanding coding in the brain ([31, 32]).
When run on typical CPUs or GPUs without special hardware, the SbS network is computationally very demanding. For this paper this forced us to stick with the simple gradient optimization method which is known to result in less classification performances compared to more elaborated optimization methods like Adam ([22]) or L4 ([21]). Furthermore, the long running times for our simulations prohibited a thorough search for optimal parameters. Nevertheless, for the example of the MNIST benchmark data we could show that a SbS convolutional network is able to reach a classification performance of 97.1% correct with SbS backprop and 1024 neurons in layer $H5$ (resp. 98.3% with 60000 neurons in layer $H5$ using only unsupervised pre-learning).
In summary, we presented an error back-propagation based learning rule for training multi-layer SbS networks. This significantly extends earlier work ([11]) such that for the first time it makes supervised training of deep Spike-By-Spike based networks possible. Combined with optimized and massively parallel computational hardware ([30]) this will open the door for future investigations of this simple spike based neuronal network which has interesting properties as a generative model with in-built sparseness.

References

[1] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[2] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, and M. S. Lew, "Deep learning for visual understanding: A review," *Neurocomputing*, vol. 187, pp. 27–48, 2016.

[3] A. Azkarate Saiz, "Deep learning review and its applications," 2015.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[6] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2414–2423, 2016.

[7] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

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

[9] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*, 2018.

[10] M. Pfeiffer and T. Pfeil, "Deep learning with spiking neurons: Opportunities and challenges," *Frontiers in neuroscience*, vol. 12, 2018.

[11] U. Ernst, D. Rotermund, and K. Pawelzik, "Efficient computation based on stochastic spikes," *Neural computation*, vol. 19, no. 5, pp. 1313–1343, 2007.

[12] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, p. 788, 1999.

[13] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in neural information processing systems*, pp. 556–562, 2001.

[14] E. M. Izhikevich, "Which model to use for cortical spiking neurons?," *IEEE transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[15] A. M. Bruckstein, M. Elad, and M. Zibulevsky, "On the uniqueness of nonnegative sparse solutions to underdetermined systems of equations," *IEEE Transactions on Information Theory*, vol. 54, no. 11, pp. 4813–4820, 2008.

[16] E. J. Candes, J. K. Romberg, and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements," *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, vol. 59, no. 8, pp. 1207–1223, 2006.

[17] M. Lustig, D. L. Donoho, J. M. Santos, and J. M. Pauly, "Compressed sensing mri," *IEEE signal processing magazine*, vol. 25, no. 2, pp. 72–82, 2008.

[18] T. Wiedemann, C. Manss, and D. Shutin, "Multi-agent exploration of spatial dynamical processes under sparsity constraints," *Autonomous Agents and Multi-Agent Systems*, vol. 32, no. 1, pp. 134–162, 2018.

[19] S. Ganguli and H. Sompolinsky, "Compressed sensing, sparsity, and dimensionality in neuronal information processing and data analysis," *Annual review of neuroscience*, vol. 35, pp. 485–508, 2012.

[20] S. Ganguli and H. Sompolinsky, "Statistical mechanics of compressed sensing," *Physical review letters*, vol. 104, no. 18, p. 188701, 2010.

[21] M. Rolinek and G. Martius, "L4: Practical loss-based stepsize adaptation for deep learning," *arXiv preprint arXiv:1802.05074*, 2018.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[23] J.-H. Oh and H. S. Seung, "Learning generative models with the up propagation algorithm," in *Advances in Neural Information Processing Systems*, pp. 605–611, 1998.

[24] Z. Guo and S. Zhang, "Sparse deep nonnegative matrix factorization," *arXiv preprint arXiv:1707.09316*, 2017.

[25] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic backpropagation and approximate inference in deep generative models," *arXiv preprint arXiv:1401.4082*, 2014.

[26] Y. Bengio, E. Laufer, G. Alain, and J. Yosinski, "Deep generative stochastic networks trainable by backprop," in *International Conference on Machine Learning*, pp. 226–234, 2014.

[27] R. Salakhutdinov, "Learning deep generative models," *Annual Review of Statistics and Its Application*, vol. 2, pp. 361–385, 2015.

[28] J.-H. Ahn, S. Choi, and J.-H. Oh, "A multiplicative up-propagation algorithm," in *Proceedings of the twenty-first international conference on Machine learning*, p. 3, ACM, 2004.

[29] X. Zeng, Z. He, H. Yu, and S. Qu, "Bidirectional nonnegative deep model and its optimization in learning," *Journal of Optimization*, vol. 2016, 2016.

[30] D. Rotermund and K. R. Pawelzik, "Massively parallel fpga hardware for spike-by-spike networks," *bioRxiv*, p. 500280, 2018.

[31] B. A. Olshausen and D. J. Field, "What is the other 85 percent of v1 doing," *L. van Hemmen, & T. Sejnowski (Eds.)*, vol. 23, pp. 182–211, 2006.

[32] A. Spanne and H. Jörntell, "Questioning the role of sparse coding in the brain," *Trends in neurosciences*, vol. 38, no. 7, pp. 417–427, 2015.