frontiers in Computational Neuroscience

ORIGINAL RESEARCH
published: 13 August 2019
doi: 10.3389/fncom.2019.00055

Back-Propagation Learning in Deep Spike-By-Spike Networks

David Rotermund and Klaus R. Pawelzik*

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

Artificial neural networks (ANNs) are important building blocks in technical applications. They rely on noiseless continuous signals in stark contrast to the discrete action potentials stochastically exchanged among the neurons in real brains. We propose to bridge this gap with Spike-by-Spike (SbS) networks which represent a compromise between non-spiking and spiking versions of generative models. What is missing, however, are algorithms for finding weight sets that would optimize the output performances of deep SbS networks with many layers. Here, a learning rule for feed-forward SbS networks is derived. The properties of this approach are investigated and its functionality is demonstrated by simulations. In particular, a Deep Convolutional SbS network for classifying handwritten digits achieves a classification performance of roughly 99.3% on the MNIST test data when the learning rule is applied together with an optimizer. Thereby it approaches the benchmark results of ANNs without extensive parameter optimization. We envision this learning rule for SBS networks to provide a new basis for research in neuroscience and for technical applications, especially when they become implemented on specialized computational hardware.

Keywords: deep network (DN), spiking network model, sparseness, compressed sensing (CS), error back propagation (BP) neural network

OPEN ACCESS

Edited by:
Anthony N. Burkitt,
The University of Melbourne, Australia

Reviewed by:
Timothée Masquelier,
Centre National de la Recherche Scientifique (CNRS), France
Thomas Pfeil,
Robert Bosch, Germany

****Correspondence:***
David Rotermund
davrot@neuro.uni-bremen.de

Received: *11 March 2019*
Accepted: *24 July 2019*
Published: *13 August 2019*

Citation:
Rotermund D and Pawelzik KR (2019) Back-Propagation Learning in Deep Spike-By-Spike Networks. Front. Comput. Neurosci. 13:55. doi: 10.3389/fncom.2019.00055

1. INTRODUCTION

Fueled by the huge progress in computational power by CPUs (central processing units), GPUs (graphics processing units), and the availability of special hardware (Lacey et al., 2016; Sze et al., 2017; Jouppi et al., 2018), deep neuronal networks (Schmidhuber, 2015) brought a massive improvement to the field of expert systems as well as artificial intelligence (Azkarate Saiz, 2015; Mnih et al., 2015; Gatys et al., 2016; Guo et al., 2016; Silver et al., 2016). These networks started out as simple perceptrons (Rosenblatt, 1958) which where extended into multi-layer networks by a learning procedure 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 so-called back-propagation rule (Rumelhart et al., 1986) allows to train all weights in a network based on the back-propagated error.

In real biological neuronal networks, however, information is typically exchanged between neurons by discrete stereotyped signals, the action potentials. Combining deep networks with spikes offers new opportunities (Lee et al., 2016; Anwani and Rajendran, 2018; Tavanaei et al., 2018; Wu et al., 2018), 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; Pfeiffer and Pfeil, 2018).

Ernst et al. (2007) presented a framework for networks of stochastically spiking neurons that is related to non-negative generative models (Lee and Seung, 1999, 2001). Updates in these Spike-By-Spike (SbS) networks occur only when a new spike occurs, such that time progresses only from one spike to the next. Thereby, these networks have relatively low additional computational requirements for using spikes as a mean for transmitting information to other neurons.

This makes SbS akin to event-based neuron models (Brette, 2006, 2007; Lagorce et al., 2015; Serrano-Gotarredona et al., 2015) but with populations of stochastically firing neurons that perform inference. While the goal of these inference populations (IPs) is to represent the input as best as possible by their latent variables also their sparseness becomes optimized: An IP has one latent variable $h(i)$ for each of its N neurons ($i = 1, ..., N$) and the update of the latent variables (here termed h-dynamic) finds solutions where the $h(i)$'s are sparse over the populations of neurons. Sparseness is mainly fueled by using non-negative elements which connects this approach to compressed sensing (CS) (Candes et al., 2006; Bruckstein et al., 2008; Lustig et al., 2008; Ganguli and Sompolinsky, 2010, 2012; Wiedemann et al., 2018), a method used in technical applications to reconstruct underlying causes from data if these causes are sparse. The level of sparseness is further influenced by a parameter ϵ (Ernst et al., 2007) that represents the temporal integration rate (the larger, the more sparse).

Furthermore, SbS networks allow for massive parallelization. In particular, in layered architectures each layer consists of many IPs which can be simulated independently while the communication between the IPs is organized by a low bandwidth signal—the spikes. While this natural parallelization into populations or single neurons is a property of most neuronal networks with spiking neurons, the bandwidth required for representing a typical spiking neuron model is higher when the integration time needs to be segmented into small time steps. In the case of the SbS model, where time progresses from one spike to the next only the identity of the spiking neuron and information about the corresponding IP is transferred. Technically speaking, this allows to build special hardware (Rotermund and Pawelzik, 2018) dealing with the SbS IPs where one application specific integrated circuit (ASIC) can host a sub-network of IPs which can be arranged into larger networks by connecting such ASICs via exchanging the low bandwidth spike information. This is furthered by the ability to run the internal h-dynamics of an IP asynchronously from the spike exchange process.

In summary, a SbS network is fundamentally different from usual artificial neuronal networks since (a) the building block of the network are SbS inference populations which realize an optimized generative representation of the IP's inputs with only non-negative elements, (b) time progresses from one spike to the next, while conserving the property of stochastically firing neurons, and (c) a SbS network has only a small number of parameters, making it easy to use. With respect to biological realism and computational effort to simulate neural networks these properties place a SbS network in between non-spiking neural networks and networks of stochastically spiking neurons.

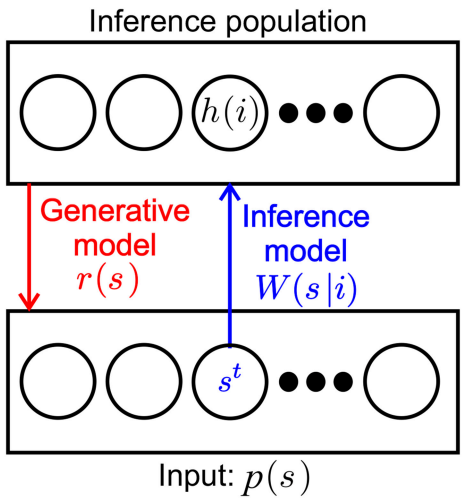

FIGURE 1 | SbS inference population (IPs). SbS IP that processes incoming stochastic spikes s^t generated from an input population $p(s)$, where s denotes the index of the input neuron. The non-negative weights $W(s|i)$ and the latent variables $h(i)$ of the neurons (i enumerates the N_H neurons in that IP) are used in a generative model to optimally represent the observed input regarding the loss function.

However, until now optimizing deep SbS networks for specific tasks was not possible since only local learning rules for the weights existed.

For convenience we recapitulate the inner workings of one SbS inference population (Ernst et al., 2007) by taking a look at the most simple network, constructed from one input population and one hidden inference population (see **Figure 1**): For setting up the framework we initially assume that the input neurons fire independently according to a Poisson point process with a firing rate $\rho_\mu(s)$, where s denotes one of the N_S input neurons and μ the momentary input pattern from a set of M available input patterns. The input pattern can be e.g., pixel images, waveforms or other types of rate coded information. The normalized external input $p_\mu(s) = \frac{\rho_\mu(s)}{\sum_{s'}^{N_S} \rho_\mu(s')}$ is the probability of input neuron s^t to be the next to fire a spike where t denotes the time step when this happens.

The inference population has one latent variable $h_\mu(i)$ for each of its neurons, where i denotes the identity of the neuron. The latent variables of an IP can form a probability distribution with $0 \leq h(i) \leq 1$ since they are normalized according to $\sum_i^{N_H} h_\mu(i) = 1$. The purpose of representing its input as a generative model with its latent variables is laid upon the inference population. The internal representation $r_\mu(s)$ of the normalized external input $p_\mu(s) = \frac{\rho_\mu(s)}{\sum_{s'}^{N_S} \rho_\mu(s')}$ is defined by

$$r_\mu(s) = \sum_i h_\mu(i) W(s|i)\,, \tag{1}$$

where $W(s|i)$ are the corresponding weights between the input population and the hidden population, which are also

normalized and non-negative numbers ($0 \leq W(s|i) \leq 1$ with $\sum_s W(s|i) = 1$).

$p_\mu(s)$ can not be observed directly by the inference population. Only the spikes s^t emitted by the input population are visible. By counting the spikes, $p_\mu(s)$ could be estimated through

$$\hat{p}_\mu(s) = \frac{1}{T}\sum_t^T \delta_{s,s^t_\mu}\,, \tag{2}$$

after observing T spikes. The goal of the IP is to minimize the difference between $\hat{p}_\mu(s)$ and its own representation $r_\mu(s)$. As an objective measure we use the cross-entropy:

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

Derivatives of E with respect to $h_\mu(i)$ can be used for deriving iterative algorithms that shall estimate optimal values for the latent variables $h_\mu(i)$ from the spikes generated by the input pattern $\hat{p}_\mu(s)$ weighted by given $W(s|i)$.

A particularly simple iterative update algorithm for the latent variables $h_\mu(i)$ is obtained for processing only one spike s^t_μ at a time. It results in the so called h-dynamic (Ernst et al., 2007):

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

where $\epsilon > 0$ is a smoothing parameter. ϵ corresponds to a rate constant in a low-pass filter that regulates the impact of the contribution for one spike onto the latent variables which also controls the level of sparseness on $h_\mu(i)$ depending of its value (the larger ϵ the sparser the representation).

The transition to processing only the momentary spike in the h-dynamic allows to replace the original Poisson rate coded input populations by Bernoulli processes based on the probability distribution $p_\mu(s)$. This is possible because now the only important information is which of the input neurons fires next (for the formal derivation see Ernst et al., 2007).

This simple example with one input population and one SbS inference population can be extended to all kinds of network structures (Rotermund and Pawelzik, 2019b) by exploiting the fact that the latent variables of an IP themselves form a probability distribution $h^t_\mu(s)$. This distribution is then used in a Bernoulli process to also produce spikes and act as an input population for other IPs. While the $p_\mu(s)$ of the input is assumed to be constant over time for a given pattern, $h^t_\mu(s)$ will change with every spike that IP processes. This allows to transport information through the whole network.

While Ernst et al. (2007) provides suitable learning rules for networks with one hidden layer, it doesn't offer learning rules for networks with arbitrary many layers. In this paper a learning rule capable of training the weights for deep feed-forward networks is derived in analogy to error back-propagation (Rumelhart et al., 1986). It differs substantially from the usual back-prop rule since it needs to take into account the dynamics for the latent variables presented above. After that, the functionality of this new learning rule is demonstrated with several examples of increasing complexity. Finally a spike-by-spike version of a deep convolutional network for classifying handwritten digits is examined. It turns out, that it can be built exclusively from IPs, in particular, the pooling layers do not require computationally different modules. Here we also introduce an optimizer which significantly improves the network's performance as compared to a method that is based only on gradient descent.

2. RESULTS

The source code used for simulating the presented networks can be found in the **Supplementary Materials**.

2.1. SbS Backprop Learning Rule

In the following a short summary of the learning rule is given. A detailed derivation of the learning rule can be found in the **Supplementary Material**. Since the equations are heavily laden with indices, **Figure 2** gives an overview of the equations for an example segment of a network with an output layer and three hidden layers.

The goal is to update the old weights W to get new weights W^{new}, using the gradient $\frac{\partial E}{\partial W}$ on the objective function E

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

with $\zeta_\mu(q)$ as the desired output for neuron q in the output layer of the network and $h^t_{y,\mu}(q)$ as the actual output of the network at time t.

In the following the output layer y is used as reference from which all the other layers are counted backwards. The update for the weights $W^{l\to l+1}(q|q')$ (or short $W^l(q|q')$) between layer l (with Q_l neurons in an IP) and layer $l+1$ can be calculated via

$$\begin{aligned} V^l(q|q') &= W^l(q|q')\left(1-\frac{\gamma}{S^l}\frac{\partial E}{\partial W^l(q|q')}\right)\\ W^{new,l}(q|q') &= \frac{V^l(q|q')}{\sum_j V^l(j|q')}\,. \end{aligned} \tag{6}$$

γ is a learning rate ($0 < \gamma < 1$) and S is a scaling factor for preserving the non-negativity of the weight values. I.e., S needs to ensure that $\frac{\gamma}{S}\frac{\partial E}{\partial W} \leq 1$ for all components of the weight matrix. This can be done by using

$$S^l = \max\left(\left|\frac{\partial E}{\partial W^l(q|q')}\right|\right) \tag{7}$$

with calculating the maximum over all components of $\frac{\partial E}{\partial W(q|q')}$.

The gradient itself is a sum over all contributions from a set of several pattern μ (i.e., a batch or mini-batch of input pattern) and, in the case of a convolutional layer, a sum over different spatial positions:

$$-\frac{\partial E}{\partial W^l(q|q')} = \sum_{\mu}^{M}\omega^l_\mu(q|q') \tag{8}$$

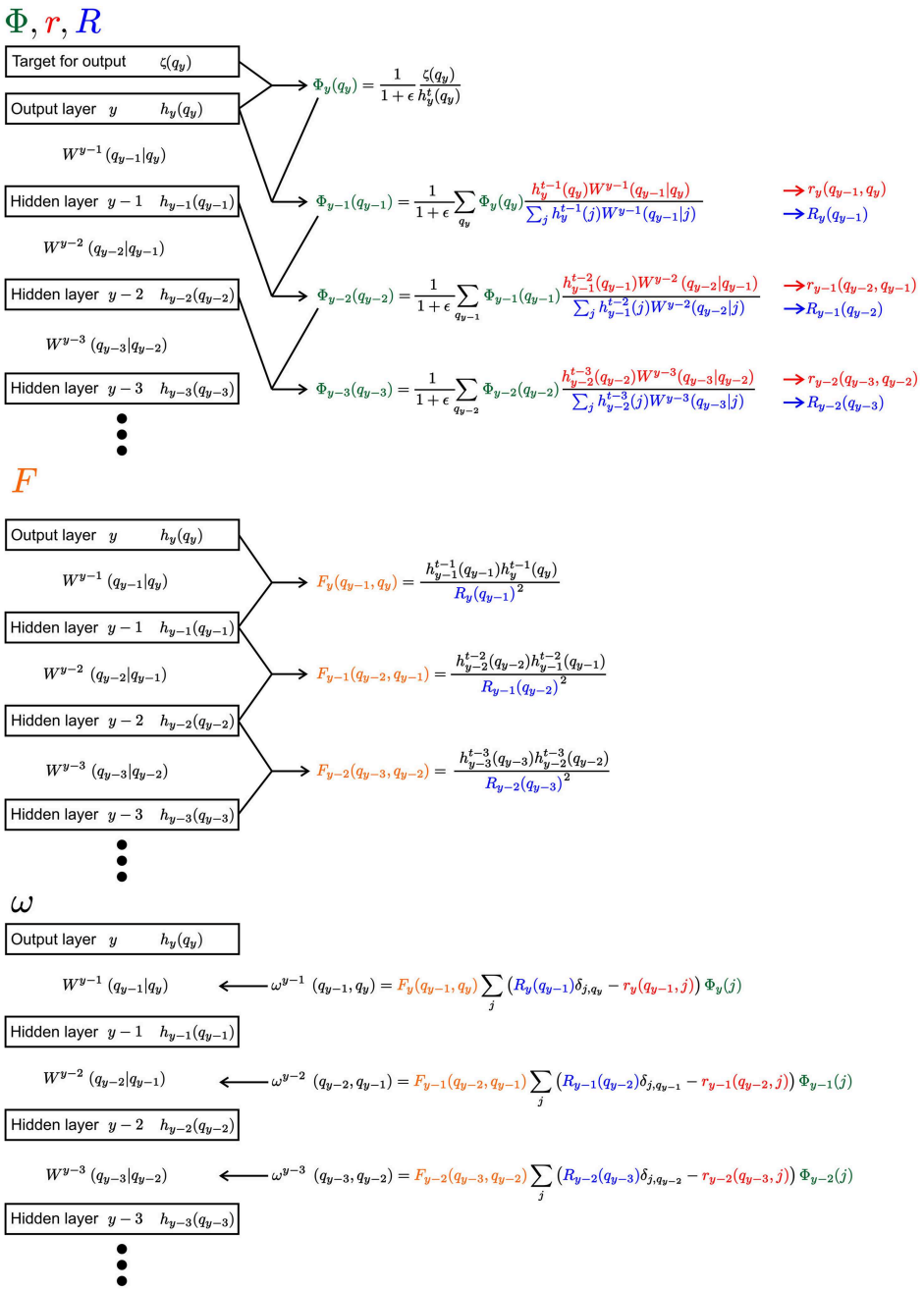

FIGURE 2 | Visualization of the learning rule and its parts. The Equation 9 and its sub-equations are shown for the output layer and three hidden layers. Everything, except the weights W, would have the same index μ, which is not shown for convenience. We use the different colors to highlight the reoccurring parts of the equations (Φ, r, R, and F).

Each of the contribution ω is calculated from four sub-equations Φ, r, R and F (see **Figure 2**):

$$\omega_\mu^l(q|q') = F_{l+1,\mu}(q,q')\cdot\sum_j^{Q_{l+1}}\left(R_{l+1,\mu}(q)\delta_{j,q'} - r_{l+1,\mu}(q,j)\right)\Phi_{l+1,\mu}(j) \tag{9}$$

Φ is akin to the back propagating error in non-spiking neural networks. At the output layer y, the distribution over the latent variables $h_\mu^t(q)$ is compared to the desired target $\zeta_\mu(q)$:

$$\Phi_{y,\mu}(q) = \left(\frac{\epsilon}{1+\epsilon}\right)\frac{\zeta_\mu(q)}{h_{y,\mu}^t(q)} \tag{10}$$

From there Φ is propagated backward in direction on the input (with $m \geq 1$ as the distance measured to the output layer y):

$$\Phi_{y-m,\mu}(q) = \left(\frac{\epsilon}{1+\epsilon}\right) \sum_{q'}^{Q_{y-m+1}} \Phi_{y-m+1}(q') \frac{h^{t-m}_{y-m+1,\mu}(q') W^{y-m}(q|q')}{\sum_j h^{t-m}_{y-m+1,\mu}(j) W^{y-m}(q|j)}$$
$$= \left(\frac{\epsilon}{1+\epsilon}\right) \sum_{q'}^{Q_{y-m+1}} \Phi_{y-m+1}(q') \frac{r_{y-m+1,\mu}(q,q')}{R_{y-m+1,\mu}(q)} . \quad (11)$$

However, unlike in the typical back-prop rule Φ is not only propagating from the output layer to the input layer, it also travels back in time. This is due to the fact that Φ is updated with latent variables from further in the past the further one gets from the output layer.

For the other three components r, R, and F the equations are:

$$r_{y-m,\mu}(q,q') = h^{t-m-1}_{y-m,\mu}(q') W^{y-m-1}(q|q') \quad (12)$$

$$R_{y-m,\mu}(q) = \sum_{j}^{Q_{y-m}} r_{y-m,\mu}(q,j) \quad (13)$$

$$F_{y-m,\mu}(q,q') = \frac{h^{t-m-1}_{y-m-1,\mu}(q) h^{t-m-1}_{y-m,\mu}(q')}{R_{y-m,\mu}(q)^2} . \quad (14)$$

2.2. Learning the XOR Function

The first example is a two layer network (see **Figure 3A**) 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 population of two neurons. The first neuron in a population is only active if the input bit has a value of zero. The second input neuron of that population 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. This is done using a Bernoulli processes.

The hidden layer consists of one inference population (for which $\sum_i h(i) = 1$) with four neurons. In the figure, an exemplary solution is shown. It shows only the non-zero weights between the input and the hidden layer which have a value of 0.5 each. Given these weights and the SbS dynamics for $h(i)$ (also called h-dynamic), after processing enough input spikes, only one hidden neuron will remain active due to the competition within an inference group. The corresponding input patterns, which lead to an activation of the neurons is listed in the hidden neurons in **Figure 3A**.

The probability distribution formed by the latent variables in the hidden neurons is also used to draw one spike in every time step according a Bernoulli process. 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 according to

$$V(q,q') = 1 + 0.01 \cdot \eta(q,q') \quad (15)$$

$$W(q|q') = \frac{V(q,q')}{\sum_j V(j,q')} \quad (16)$$

with $\eta(q,q')$ as random numbers drawn from a uniform distribution $[0,1]$. Before presentation of a new input pattern $p_X(i)$, the latent variables of the hidden neurons and the output neurons are always set to $h_{H1}(i) = \frac{1}{N_{H1}}$ and $h_{HY}(i) = \frac{1}{N_{HY}}$, respectively. Then, for every time step of the simulation, one spike each (i.e., the index of the neuron which fires next) is drawn from p_X and h_{H1}. This is done by using these two probability distributions for Bernoulli processes.

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 9 the gradient for this pattern is calculated and stored. After collecting these contributions for all four input patterns (Equation 8), Equation 6 is applied to update the weights.

However, before normalizing $V^l(q|q')$, it is ensured that the smallest value in V is $\Theta = 0.0001$:

$$\tilde{V}^l(q|q') = \max\left(\left\{V^l(q|q'), \Theta\right\}\right)$$
$$W^{new,l}(q|q') = \frac{\tilde{V}^l(q|q')}{\sum_j \tilde{V}^l(j|q')} .$$

This prevents the multiplicative update Equation 6 from getting stuck in zero values. **Figure 3B** shows the quality of the output during learning, averaged over 250 initial weights. Using 1,024 spikes per input pattern & learning step as well as $\gamma = 0.025$, an error of 0 is reached after 32 learning steps.

Analyzing the magnitude of the learned weight values, reveals that the weights have only changed a small amount from the randomly initialized weights. 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 **Supplementary Materials** we show the weights and present a procedure how to force the learning rule to produce weights more similar to the ideal binary weights shown in **Figure 3A**.

Furthermore, we show in the **Supplementary Materials** that the learning rule is able to ignore non task relevant inputs that are not correlated to the goal of the training procedure.

2.3. Learning the 4 Bit Parity Function

The 4 bit parity function can be understood as 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 4A**): Input layer X with

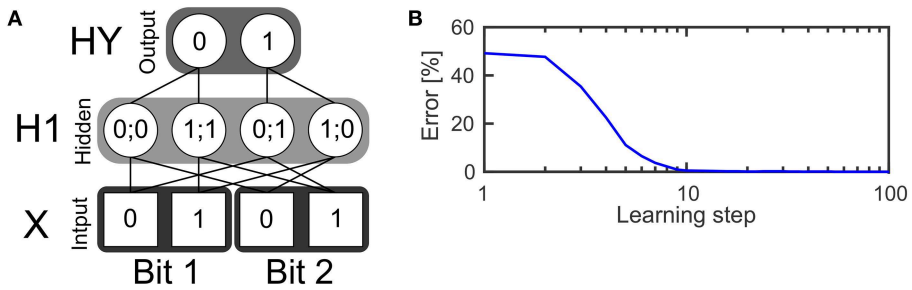

FIGURE 3 | Learning the XOR function. **(A)** Structure of a SbS network for solving the XOR function. **(B)** Error of the network's output during learning the weights, averaged over 250 initial conditions.

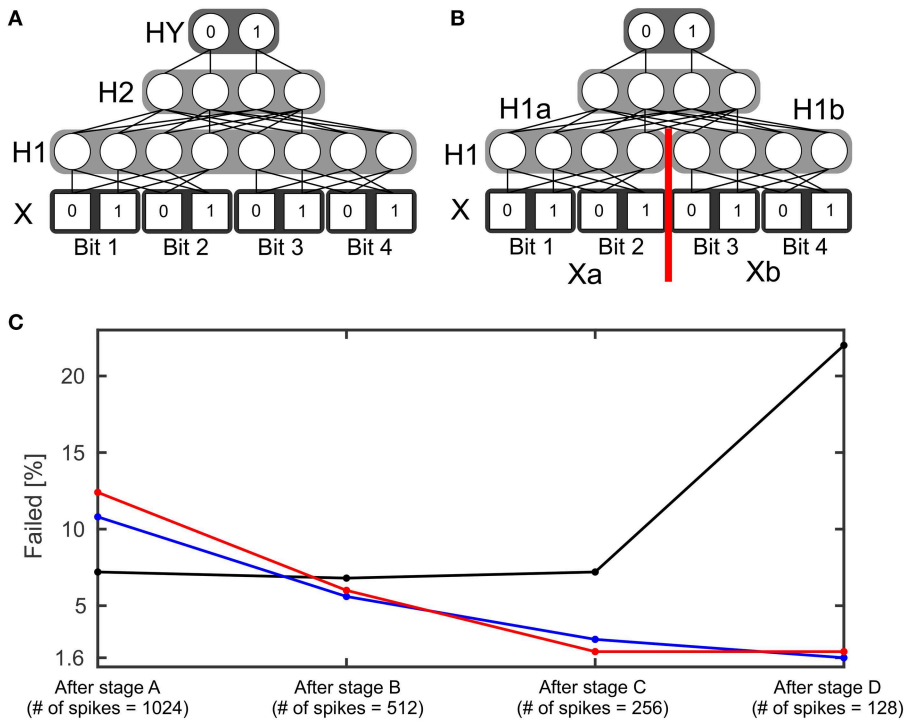

FIGURE 4 | Four bit parity function. **(A)** Structure of a SbS network able to solve the 4 bit parity task. **(B)** Variation of the structure of the network. The hidden layer *H*1 was split into two normalization groups *H*1*a* and *H*1*b* with 4 neurons each. **(C)** Failure rate of learning the 4 bit parity function with the network structure shown in **(A)** (black) and using the network structure **(B)** (blue and red). For the red curve, learning was performed without the retardation of the latent variables as formally required from the derivation of the learning rule. Instead the values obtained after the most recent spike were used. The failure rate is shown for the weights after the four stages of learning.

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 for simulating this network is as described in the example for the XOR network. Again, learning started with randomly initialized weights (see XOR example for how the weights were initialized). 250 simulations with different initial seeds were performed.

In the following, if two performances are called significantly different then a one-sided Fisher's exact test with p-level of 1% was used to determine this statement.

In summary, learning the 4 bit parity function shows that the learning algorithm has a problem with local minima.

We tried to minimize the amount of failed attempts (i.e., only if all 16 possible outputs of the network are classified correctly then it is not a failed attempt) by using different combinations of numbers of spikes per pattern and learning rates γ during learning. Such a procedure showed fruitful results in the XOR example where it lead to weights that looked more like the ideal weights (see **Supplementary Material** for the details). Thus we also applied it to the 4 bit parity function.

Learning went through four stages with every stage consisting of 7,000 learning steps. Furthermore, the subsequent stages

started with the final weights from the stage before, while the first stage started with random weights. For the stage A, 1,024 spikes per pattern were used. The following stages reduced the amount of spikes by a factor of 2 (512 spikes for stage B, 256 spikes for stage C, and 128 spikes for stage D) for learning. However, the performance was tested with 1,024 spikes per pattern, to keep the results comparable.

The reasoning behind this approach is as follows: Reducing the number of spikes increases the noise in the system by deteriorating the representation of inputs and latent variables which are transmitted to other inference populations. The learning process is forced to find a refined set of weights that is more robust against noise. However, starting directly with a high noise level ends often in non-functional set of weights. Thus we start with a low noise situation, which allows the learning algorithm find suitable weights. After learning, typically we found the range of used weights values to be rather shallow because with a low amount of noise the competition in an inference group can solve the task with small difference in the weights. Then the increase in noise forces the learning rule to increase the value range used in the weights.

In addition to reducing the number of spikes from one stage to the next, the learning rate gamma is reduced during a stage. Every stage starts with $\gamma = 0.03$. After every 1,000 learning steps, γ is divided by 2 ($\gamma_{\{1,\ldots 999\}} = 0.03$, $\gamma_{\{1000,\ldots 1999\}} = \frac{0.03}{2}$, $\gamma_{\{2000,\ldots 2999\}} = \frac{0.03}{4}$, $\gamma_{\{3000,\ldots 3999\}} = \frac{0.03}{8}$, $\gamma_{\{4000,\ldots 4999\}} = \frac{0.03}{16}$, $\gamma_{\{5000,\ldots 5999\}} = \frac{0.03}{32}$, and $\gamma_{\{6000,\ldots 7000\}} = \frac{0.03}{64}$). The idea behind this procedure is that in the beginning the weights are allowed to change strongly and then they are supposed to settle at their correct values. In the **Supplementary Materials**, detailed learning curves for the four stages are shown. The black line in **Figure 4C** shows the amount of initial conditions that failed to learn after the stages. **Figure 4C** reveals that the first three stages don't improve the failure rate. Furthermore, stage D even significantly increases the number of failed learning attempts.

A putative source for this problem could be that for a successfully operating network two neurons in $H1$ need to be active simultaneously. However, learning a SbS IP tends to favor sparse solutions. Thus, the failed learning attempts could be a result of over-sparsification. Hence, we modified the network structure (see **Figure 4B**) and split the hidden layer $H1$ into two normalization groups $H1a$ and $H1b$. Now a successful network needs only one active neuron per SbS IP. $H1a$ gets only input from the first two bits (Xa) of the Input X and $H1b$ sees the spikes from the latter two input bits (Xb).

With the new structure five spikes are drawn in every time step: One spike each from Xa, Xb, $H1a$, $H1b$ and $H2$. The weights between Xa and $H1a$ as well as the weights between Xb and $H1b$ are the same, like it would be in a convolutional neuronal network. During learning, the two contributions from the SbS backprop learning rule are averaged. **Figure 4C** (blue line) shows that there is now a significant difference between the failure rate after stage A and after stage D. Also a significant difference of the failure rate after stage D between the network with the split $H1$ layer (1.6%) vs. the original network (22.0%) was found.

As final test with the 4 bit parity function, we used the network with the split in $H1$ and investigated how important the retardation of the latent variables during learning is. Instead of using the h-values from the earlier spikes—like it is required by the derived SbS backprop rule—, we only used the h-values after processing the last spike for every input pattern. **Figure 4C** (red line) shows here no significant difference. Thus it might be an interesting alternative (e.g., for saving memory) to neglect the retardation on the latent variables.

Examples for successfully working weights sets for these three tests are shown in the **Supplementary Materials**.

2.4. Deep Convolutional Network (MNIST)

The MNIST database is a benchmark for machine learning (see http://yann.lecun.com/exdb/mnist/). It consists of handwritten digits with 28 × 28 pixels and 256 gray values per pixel. The database contains 60,000 examples for training the network and 10,000 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. Replacing the simple gradient decent by Adam (Kingma and Ba, 2014) and dropout learning (Srivastava et al., 2014), allows this network to reach a performance of 99.2% classifications correct (both source codes can be found in the **Supplementary Materials**).

The network in the TensorFlow tutorial is structured as follows: The input layer X consists of 28 × 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 × 28 pixel after the convolution) through a kernel with the size of 5 × 5 with 32 filters. $H1$ is followed by a max pooling layer $H2$, which calculated the maximum over a 2 × 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 2 × 2 max pooling layer to $H3$. After $H4$, a fully connected layer $H5$ with 1,024 neurons is positioned. And finally the output of the network can be read out 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 (see **Figure 5**). 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 × 28 pixels but layer $H1$ decreases to 24 × 24 with 32 filters. $H2$ halves the size to 12 × 12. Convolution layer $H3$ compressed the output of $H2$ to 8 × 8 with 64 filters. Its max pooling layer halves it again to 4 × 4. We keep the 1,024 neurons for the fully connected layer $H5$ as well as the 10 neurons for the output layer. In the **Supplementary Materials**, the structure of the SbS

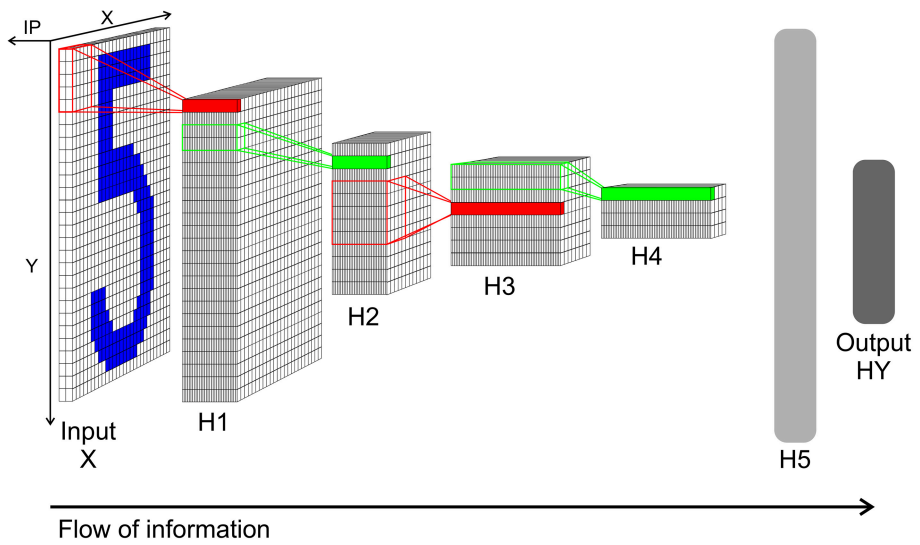

FIGURE 5 | Network structure of the convolution network for the MNIST data. Input *X*: Input layer with 28 × 28 normalization modules for 28 × 28 input pixel. Each module has two neurons realizing a simplified version of on/off cells for enforcing positive activity also for low pixel values. From this layer spikes are send to layer *H*1. *H*1: Convolution layer *H*1 with 24 × 24 IPs with 32 neurons each. Every IP processes the spikes from 5 × 5 spatial patches of the input pattern (x and y stride is 1). *H*2: 2 × 2 pooling layer *H*2 (x and y stride is 2) with 12 × 12 IPs with 32 neurons each. The weights between *H*1 and *H*2 are not learned but set to a fixed weight matrix that creates a competition between the 32 features of *H*1. *H*3: 5 × 5 convolution layer *H*3 (x and y stride is 1) with 8 × 8 IPs. Similar to *H*1 but with 64 neuron for each IP. *H*4: 2 × 2 pooling layer *H*4 (x and y stride is 2) with 4 × 4 IPs with 64 neurons each. This layer is similar to layer *H*2. *H*5: Fully connected layer *H*5. 1,024 neurons in one big IP which are fully connected to layer *H*4 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.

network is discussed in detail, especially that all layers—including the pooling layer—realize the same algorithm.

We used TensorFlow to train this reduced network with a simple gradient descent optimization and got a classification correct performance of 97.1%. Furthermore, we got still 99.2% for a version using Adam and dropout learning. In Rotermund and Pawelzik (2019b) we showed that—with 97.8% classification correct—this SbS network structure using a only local learning rule (based on simple gradient descent and bi-directional flow of spikes) can match the performance TensorFlow example with the simple gradient descent optimization.

In the following we will present the SbS back-prop learning rule combined with a new optimizer applied to the MNIST benchmark.

For the input to the SbS network, a so called on/off split was made (Ernst et al., 2007) 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 is defined 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.

Furthermore, instead of using max functions, the pooling layers *H*2 & *H*4 use only the inherent competition implemented by the SbS update rule. For the pooling IPs, the weights are fixed and not learned. E.g., a 2 × 2 spatial patch from convolution layer *H*1 (with its 2 × 2 × 32 neurons) delivers input to one pooling IP in *H*2 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 IPs) 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 *H*2 normalization group. The same happens for pooling layer *H*4 but with 64 neurons per IP.

A widely known experience from learning neural networks is that using a simple gradient descent is prone to end in a low performance. Thus optimizers like Adam (Kingma and Ba, 2014) are used. Transferring optimizers like Adam to the SbS world is problematic due to the normalization and non-negativity boundary conditions on the latent variables and the weights. As an alternative we developed an optimizer for SbS networks inspired by L4 (Rolinek and Martius, 2018). First of all, we used random mini-batches with 10% of the whole training data set. We smoothed the gradients from our SbS back-prop learning rule as well as the Kullback-Leibler divergence *KL* (measured between the desired and the actual distribution of the latent variables of the output layer *HY*) over the mini-batch according the equation from Rolinek and Martius (2018):

$$\hat{Y}(L) = \hat{Y}(L-1)\left(1 - \frac{1}{\tau}\right) + \frac{Y(L)}{\tau} \tag{17}$$

$$Y^*(L) = \frac{\hat{Y}(L)}{1 - \left(1 - \frac{1}{\tau}\right)^L} \tag{18}$$

where $Y(L)$ is the entity which needs smoothing and with L as learning step (i.e., number of used mini-batches) ($L \in \{1, \ldots L_{Max}\}$). Since we use mini-batches with 10% of the pattern randomly selected from the whole training data set, we selected

$\tau = 10$. In the following we will denote low-pass filtered variables with a $*$.

Furthermore, we modulated the learning rate γ with the smoothed Kullback-Leibler divergence KL^* over the training data via

$$\gamma(L) = \gamma_0 \sqrt{\frac{KL^*(L)}{KL_{Max}}} \tag{19}$$

with $\gamma_0 = 0.05$ (which was our guess for a good initial learning rate),

$$KL_{Max} = \max\left(\left\{KL^*(L), KL_{Max}\right\}\right) \tag{20}$$

and

$$KL = \sum_{\mu}^{M} \sum_{q}^{Q_{HY}} \zeta_\mu(q) \log\left(\frac{\zeta_\mu(q)}{h_{y,\mu}(q)}\right) \tag{21}$$

where M is the size of the mini-batch. In Equation (19) we chose to use the square root, compared to a linear function in Rolinek and Martius (2018). Our reasoning was that this keeps the learning rate higher in the beginning. We didn't compare the performances that both choices would result in.

The updates of the weights are done according to

$$V^l(q|q') = W^l(q|q') + \frac{\gamma(L)}{S} Z^{*,l}(q|q') \tag{22}$$

$$\tilde{V}^l(q|q') = \max\left(\left\{V^l(q|q'), \Theta, \frac{W^l(q|q')}{2}\right\}\right) \tag{23}$$

$$W^{new,l}(q|q') = \frac{\tilde{V}^l(q|q')}{\sum_j \tilde{V}^l(j|q')}. \tag{24}$$

using $\Theta = 0.0001$ and

$$Z^l(q|q') = -\frac{\partial E}{\partial W^l(q|q')} W^l(q|q') \tag{25}$$

as well as

$$S = \min\left(\left\{\max\left(\left|Z^{*,l}(q|q')\right|\right), 1\right\}\right) \tag{26}$$

with calculating the maximum over all components of $Z^*(q|q')$. For the SbS network with the back-prop learning rule, a three stage learning procedure is applied. This sequence of stages is shown in **Figure 6A**. In **Figure 6B** the development of the classification error over the these stages is summarized. In the **Supplementary Materials** detailed plots are shown that present how the training error (measured with the Kullback-Leibler divergence), learning rate, and the classification error on the test data set develops.

During learning weights are set to random values, this is done by

$$V(q, q') = 1 + 0.01 \cdot \eta(q, q') \tag{27}$$

$$W(q|q') = \frac{V(q, q')}{\sum_j V(j, q')} \tag{28}$$

with $\eta(q, q')$ as random numbers drawn from a uniform distribution $[0, 1]$.

For every input pattern, every input or SbS inference population generates 1,200 spikes over the course of simulating the SbS network for this pattern.

For the first stage of learning, we selected $\epsilon_0 = 0.1$ which is our standard value that typically works. For the latter two stages we reduced it to $\epsilon_0 = \frac{0.1}{2}$. The reasoning behind this decision was that we feared that otherwise the higher layer could might get too sparse in the distribution of latent variables in the SbS IPs. From ϵ_0 we derive ϵ values for the SbS IPs in the different layers. The underlying idea is that while IPs in different layers get a different amount of spikes in every time step of the simulations, we wanted to equalize the change on the latent variables of all IPs. Thus we divided ϵ_0 by the amount of spikes an IP receives in one time step. This results in: $\epsilon_{H1} = \epsilon_0$, $\epsilon_{H2} = \frac{\epsilon_0}{4}$, $\epsilon_{H3} = \frac{\epsilon_0}{25}$, $\epsilon_{H4} = \frac{\epsilon_0}{4}$, $\epsilon_{H5} = \frac{\epsilon_0}{16}$, and $\epsilon_{HY} = \epsilon_0$. After 1,000 spikes ϵ_0 is divided by 25. Since ϵ acts like a low-pass filter parameter that is controlling the impact one spike can have on the latent variables, reducing ϵ results in a reduction on the fluctuations of the latent variables. First, we want to allow the network to reach some "good" state with the first 1,000 spikes, then we use the reduction on ϵ and the next 200 spikes to implement an implicit averaging of the latent variables over the incoming spikes.

It is important to note, that we didn't optimize these ϵ values or ϵ_0 due to missing computational power. The parameters stem from a mere guess which parameters might work. The same ϵ values were used for learning the weights and testing the classification performance.

The first stage of learning starts with all the weights set to random values, except the pooling layer which are pre-set and not changed or learned at all. Learning the weights for 160 mini-batches, we reached a classification error on the test data set of 2.6%. Then the weights $W^{H2\to H3}$, $W^{H4\to H5}$, and $W^{H5\to HY}$ are set to random values again. Only the weights $W^{X\to H1}$ are kept and not learned during stage B. After 150 mini-batches the error goes down to 1.2%. Then again, the weights $W^{H4\to H5}$ and $W^{H5\to HY}$ are set to random values again. $W^{X\to H1}$ and $W^{H2\to H3}$ are kept constant.

When learning is performed again, the error goes down to 0.83% (see **Supplementary Materials** for detailed learning curves). However, sparseness on the latent variables can get a problem in the layer $H5$ with its 1,024 neurons. Selected neurons can show very large values in their latent variables which results—due to the competition in the SbS IP—in suppressing the other neurons. During learning this can get a self-reinforcing process. For compensating this behavior, we devised a strategy inspired from dropout learning (Srivastava et al., 2014).

Normally, in the beginning of the simulation all the latent variables start with the value $\frac{1}{N}$, were N is the number of neurons in a SbS IP. A simple way to include dropout in the SbS model is to initialize selected neuron's latent variables with the value zero. Since the update rule for the latent variables is multiplicative, such a neuron is disabled for that simulation. We use this aspect of the h-dynamic to implement dropout for layer $H5$ and to control how any neurons in $H5$ are active at the same time during learning.

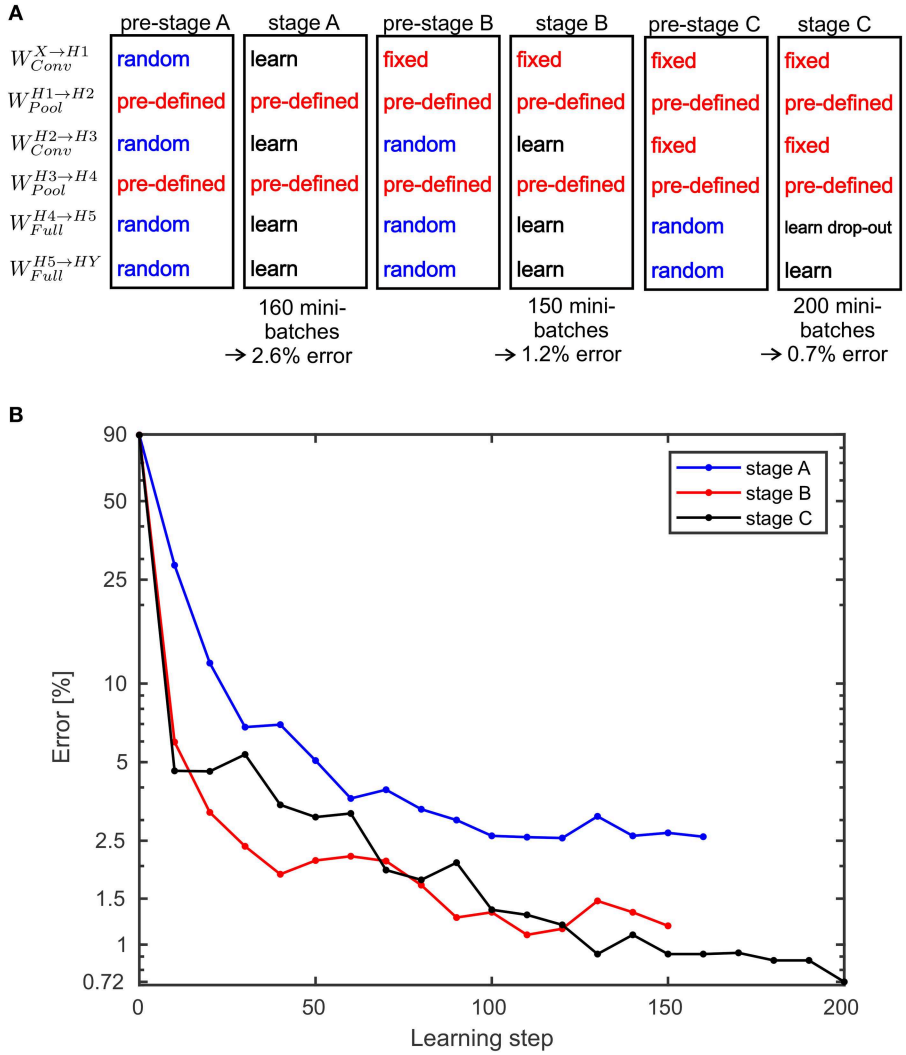

FIGURE 6 | Performance values for the MNIST benchmark. **(A)** A three stage learning process was used. First, in pre-stage A all weights are set to random values except the weights for the pooling layers which are pre-defined and not learned at all during any of the stage. In stage A the weights are trained for 160 mini-batches. This results in a classification error on the test data of 2.6%. Then in pre-stage B, the weights for the second convolutional layer and the fully connected layer are reset to random values. In stage B these random weights are trained again for 150 mini-batches which yields an error of 1.2%. In pre-stage C, the weights for the fully connected layers are replaced by random weights and in stage C these weights are learned again. However, this time a drop-out procedure for the layer *H5* is applied. After processing 200 mini-batches an error of 0.72% is reached. Alternatively, stage C can be done without drop-out and then yields in an error of 0.83% (see **Supplementary Materials** for the learning curves). **(B)** Development of the error on the MNIST test data set for the three stages over the performed learning steps.

In the beginning of the learning process we want only a few neurons to be active in $H5$ during learning (We started with 16 neurons of 1,024). Thus we calculated for every input pattern $\eta(i) \leq \beta = \frac{16}{1024}$ with $\eta(i)$ as random numbers drawn from a uniform distribution $[0, 1]$. For every $H5$ neuron i that fulfilled this condition, its latent variable $\tilde{h}_{H5}(i)$ was set to one. Otherwise the latent variable $\tilde{h}_{H5}(i)$ was set to zero. Afterwards, the latent variables of the IPs are normalized $h_{H5}(i) = \frac{\tilde{h}_{H5}(i)}{\sum_j \tilde{h}_{H5}(j)}$ and used for the simulation as initial values. After every 30 mini-batches the number of active neurons is double (i.e., β is doubled), until β reaches a value of one.

Applying dropout learning over a duration of 200 mini-batches yields an classification error of 0.72% (see **Supplementary Materials** for detailed learning curves).

For the 10% mini-batches used in our simulations, we didn't notice any over-fitting. Thus the Kullback-Leibler divergence on the training data was a reliable estimator for the expected classification performance on the test data. However, it needs to be noted that this might not be the case if the mini-batches get bigger. Then it might be necessary to separate a part of the training data as validation data set.

In **Figure S9**, we show the classification error in dependency of the amount of processed spikes. Furthermore, **Figure S10**

examines the distribution of the values over the output neurons for the 72 remaining wrongly classified patterns. This figure shows that if a pattern is classified correctly, then the correct output neuron concentrates the activity nearly exclusively on itself (see **Figure S10A**). In the case of wrongly classified pattern, the situation is more varying (see **Figure S10B**).

3. DISCUSSION

The present work is based on a framework where the basic computational units are local populations rather than individual neurons. These so called inference populations iteratively perform inference on the potential causes of their inputs from each stochastic spike impinging on the population (Ernst et al., 2007). While the corresponding dynamics might appear artificial, it captures the essence of models with more biologically realistic neurons (Rozell et al., 2008; Moreno-Bote and Drugowitsch, 2015; Zhu and Rozell, 2015) that perform sparse efficient coding, which is a leading hypothesis for understanding coding in the brain (Olshausen and Field, 2006; Spanne and Jörntell, 2015; Zhu and Rozell, 2015; Capparelli et al., 2019).

Hierarchical generative networks built from inference modules are used in technical approaches as e.g., for image generation (Ghosh et al., 2019). There are many papers concerned with learning such deep generative models with error back-propagation (e.g., Oh and Seung, 1998; Lee and Seung, 1999, 2001; Bengio et al., 2014; Rezende et al., 2014; Salakhutdinov, 2015; Guo and Zhang, 2017) as well as publications on training similar networks that use non-negative matrix factorization (e.g., Ahn et al., 2004; Zeng et al., 2016). In contrast to these approaches, the present framework conserves the specific spike driven update dynamics from Ernst et al. (2007) which is of special interest because it results in a very simple yet biologically plausible neuronal network using only spikes as signals. It allows to build special computational hardware that can be massively parallelized (Rotermund and Pawelzik, 2018) and exhibits sparse representations known from compressed sensing (Ganguli and Sompolinsky, 2010, 2012).

While deep convolutional networks were successful in predicting responses of neurons in primate visual cortex investigations into the potential of deep generative models for explaining natural computation in the brain are just beginning (Bengio et al., 2014, 2015). But even if formal approaches along these lines were phenomenologically successful, realistic models are still required to elucidate how the striking performances of brains in terms of speed and precision are realized with the spikes that in reality are available as sole signals and in cortex known to have a high degree of stochasticity.

For exploring the potential of SbS networks as alternative to deep convolutional networks with respect to performance in technical applications as well as models for real neuronal networks suitable learning rules are missing. As a first step toward this goal we present an error back-propagation based learning rule for training multi-layer feed-forward SbS networks.

Having a supervised method for optimizing the weights in deep SbS networks allows not only to explore the potential of the rather unusual SbS framework but also to compare it to other approaches toward learning in these networks as e.g., a local rule for unsupervised learning in recurrent SbS networks (Rotermund and Pawelzik, 2019b).

Simple examples with Boolean functions show that the backpropagation algorithm presented here can learn the weights in a given forward network architecture from scratch (i.e., randomly initialized weights) as well as to ignore non task relevant information. Apparently, the algorithm can be used to train weights several layers away from the output. Furthermore, architectures with convolutional and pooling layers composed of the same basic SbS elements are proposed. In particular, no special functionality was required for setting up the pooling layers, in contrast to usual deep convolutional networks.

We used the MNIST benchmark to investigate deeper convolutional SbS networks. Beside the input and the output layer, the network has five internal layers comprising two pooling layers and two convolutional layers as well as one fully connected layer. Overall, the network contains 1,378 populations with 57,994 neurons. During the simulation of one input pattern, a total of $1200 \cdot 1378$ spikes are generated (28.5 spikes per neuron). On the MNIST benchmark test data, our SbS network achieved up to 99.3% classification correct performance. Using the same architecture for a non-spiking convolutional neural network we measured 99.2%. However, comparing performance values with other networks (e.g., see Tavanaei et al., 2018 and for a list of MNIST networks http://yann.lecun.com/exdb/mnist) is not trivial. In our case we didn't optimized the network structure for the use of SbS inference populations. We rather decided to re-use the network structure associated with the Tensor Flow Tutorial because this gave us a base-line for a network design which our computer cluster was just been able to simulate. Furthermore, we didn't use any input distortion methods (e.g., shifting, scaling, or rotating the input pictures) for increasing the size of the training data set, methods that are known to lead to performance values of up to 99.8% (Wan et al., 2013). The reason was simply that this would have been too much for our computer cluster, like it would have been to optimize the parameters used in the SbS MNIST network. Or in other words: The performances shown for the MNIST SbS network certainly do not reflect what a fully optimized SbS network might be capable to deliver.

The SbS-approach avoids the real time dynamics required for simulation of noisy leaky integrate-and-fire (IaF) neurons where the membrane potential needs to become updated every time step dt which is often in the range of sub-milliseconds. The number of updates between two spikes depends on the firing rate of that neuron. If for example $dt = 0.1$ms and the firing rate is 10Hz this would translate roughly into 1,000 updates of the membrane potential between two spikes. Compared to that, a SbS neuron would perform one update. While the different types of spiking neuron models (Izhikevich, 2004) have varying number of computations for one update, in a SbS population with N neurons $3N$ multiplications, $2N$ summations, and one division are used for one update of the whole population. This reduction in computational requirement is traded in for a decrease in biological realism.

An approach akin to the SbS's removal of real time is available for integrate-and-fire neurons. These so called event-based neuronal networks (e.g., Brette, 2006, 2007; Lagorce et al., 2015; Serrano-Gotarredona et al., 2015) use analytic solutions of the neuron's dynamics to bridge the time between two spikes. However, this approach gets problematic with stochastic neurons (Brette, 2007). This is similar to the problem of finding an analytically solution for the first passage time (Burkitt, 2006a,b) of neurons in a network of neurons with stochastic inputs, which is a hard problem. In the case of the SbS network, the stochasticity is an important aspect because it is a type of importance sampling of the input as well as of the latent variables. This acts as a filter in order to capture the more dominant information in the network and suppress noise.

Concerning the parallelization of non-spiking neural networks compared to the SbS model: A traditional deep convolutional neuronal network (DNN) implements several different types of layers (e.g., convolutional layer, pooling layers, and dense layers) for which it requires a variety of optimized hardware elements (Sze et al., 2017). In the case of the SbS model, all these different type of layers are represented by the same dynamics of the latent variables. A hardware solution for SbS networks (Rotermund and Pawelzik, 2018) can be understood more like a pool of SbS inference populations that are shaped into the desired network structure by just organizing the flow of the spikes. Furthermore, it is less easy to extend one layer of a non-spiking network over several ASICs. This is a consequence from the high amount of information that needs to be exchanged in a typical non-spiking DNN. In the SbS case, the computation is already compartmentalized into IPs that can be operated also asynchronously having a low bandwidth communication among each other.

In summary, we presented an error back-propagation based learning rule for training multi-layer feed-foreward SbS networks. This significantly extends earlier work (Ernst et al., 2007) such that for the first time supervised training of deep Spike-By-Spike based networks is possible. These results show that a novel network type consisting of inference populations as basic computational elements instead of single neurons can have a competitive performance. These networks use spikes and require comparatively little computational effort. The non-negativity inherent in this approach is an other desirable property since it induces sparseness, makes a link to non-negative matrix factorization and compressed sensing and matches the fact that long range interactions in cortex are excitatory. Large networks with convolutional architectures can be built from a unique type of inference populations without requiring different computational modules for the pooling layers, which we consider more elegant and biologically plausible. Combined with optimized and massively parallel computational hardware (Rotermund and Pawelzik, 2018) having an efficient learning rule will open the door for future investigations of this conceptually simple spike based neuronal network which we believe has interesting properties as a generative model with built-in sparseness for both technical applications as well as model for natural computations in real brains.

DATA AVAILABILITY

Publicly available datasets were analyzed in this study. This data can be found here: http://yann.lecun.com/exdb/mnist.

AUTHOR CONTRIBUTIONS

DR and KP contributed conception and design of the study, wrote the first draft of the manuscript, and contributed to manuscript revision, read, and approved the submitted version. DR programmed and performed the simulations.

ACKNOWLEDGMENTS

This manuscript has been released as a Pre-Print at www.biorxiv.org (Rotermund and Pawelzik, 2019a). We thank Anne Kremer from the University of Bremen's Schreibwerkstatt MINT for helping us with improving the readability of the paper.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fncom.2019.00055/full#supplementary-material

REFERENCES

Ahn, J.-H., Choi, S., and Oh, J.-H. (2004). "A multiplicative up-propagation algorithm," in *Proceedings of the Twenty-First International Conference on Machine Learning* (Banff, AB: ACM), 3.

Anwani, N., and Rajendran, B. (2018). Training multilayer spiking neural networks using normad based spatio-temporal error backpropagation. *arXiv:1811.10678*.

Azkarate Saiz, A. (2015). Deep learning review and its applications. Available online at: https://addi.ehu.es/handle/10810/15792

Bengio, Y., Lee, D.-H., Bornschein, J., Mesnard, T., and Lin, Z. (2015). Towards biologically plausible deep learning. *arXiv[Preprint].arXiv:1502.04156*.

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

Brette, R. (2006). Exact simulation of integrate-and-fire models with synaptic conductances. *Neural Comput.* 18, 2004–2027. doi: 10.1162/neco.2006.18.8.2004

Brette, R. (2007). Exact simulation of integrate-and-fire models with exponential currents. *Neural Comput.* 19, 2604–2609. doi: 10.1162/neco.2007.19.10.2604

Bruckstein, A. M., Elad, M., and Zibulevsky, M. (2008). On the uniqueness of nonnegative sparse solutions to underdetermined systems of equations. *IEEE Trans. Informat. Theory* 54, 4813–4820. doi: 10.1109/TIT.2008.929920

Burkitt, A. N. (2006a). A review of the integrate-and-fire neuron model: I. homogeneous synaptic input. *Biol. Cybern.* 95, 1–19. doi: 10.1007/s00422-006-0068-6

Burkitt, A. N. (2006b). A review of the integrate-and-fire neuron model: II. Inhomogeneous synaptic input and network properties. *Biol. Cybern.* 95, 97–112. doi: 10.1007/s00422-006-0082-8

Candes, E. J., Romberg, J. K., and Tao, T. (2006). Stable signal recovery from incomplete and inaccurate measurements. *Commun. Pure Appl. Math.* 59, 1207–1223. doi: 10.1002/cpa.20124

Capparelli, F., Pawelzik, K., and Ernst, U. (2019). Constrained inference in sparse coding reproduces contextual effects and predicts laminar neural dynamics. *bioRxiv*. doi: 10.1101/555128

Ernst, U., Rotermund, D., and Pawelzik, K. (2007). Efficient computation based on stochastic spikes. *Neural Comput.* 19, 1313–1343. doi: 10.1162/neco.2007.19.5.1313

Ganguli, S., and Sompolinsky, H. (2010). Statistical mechanics of compressed sensing. *Phys. Rev. Lett.* 104:188701. doi: 10.1103/PhysRevLett.104.188701

Ganguli, S., and Sompolinsky, H. (2012). Compressed sensing, sparsity, and dimensionality in neuronal information processing and data analysis. *Annu. Rev. Neurosci.* 35, 485–508. doi: 10.1146/annurev-neuro-062111-150410

Gatys, L. A., Ecker, A. S., and Bethge, M. (2016). "Image style transfer using convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Las Vegas, NV), 2414–2423.

Ghosh, P., Sajjadi, M. S. M., Vergari, A., Black, M., and Schölkopf, B. (2019). From variational to deterministic autoencoders. Available online at: https://arxiv.org/abs/1903.12436v2

Guo, Y., Liu, Y., Oerlemans, A., Lao, S., Wu, S., and Lew, M. S. (2016). Deep learning for visual understanding: a review. *Neurocomputing* 187, 27–48. doi: 10.1016/j.neucom.2015.09.116

Guo, Z., and Zhang, S. (2017). Sparse deep nonnegative matrix factorization. *arXiv[Preprint].arXiv:1707.09316*.

Izhikevich, E. M. (2004). Which model to use for cortical spiking neurons? *IEEE Trans. Neural Netw.* 15, 1063–1070. doi: 10.1109/TNN.2004.832719

Jouppi, N., Young, C., Patil, N., and Patterson, D. (2018). Motivation for and evaluation of the first tensor processing unit. *IEEE Micro* 38, 10–19. doi: 10.1109/MM.2018.032271057

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

Lacey, G., Taylor, G. W., and Areibi, S. (2016). Deep learning on fpgas: past, present, and future. *arXiv[Preprint].arXiv:1602.04283*.

Lagorce, X., Stromatias, E., Galluppi, F., Plana, L. A., Liu, S.-C., Furber, S. B., et al. (2015). Breaking the millisecond barrier on spinnaker: implementing asynchronous event-based plastic models with microsecond resolution. *Front. Neurosci.* 9:206. doi: 10.3389/fnins.2015.00206

Lee, D. D., and Seung, H. S. (1999). Learning the parts of objects by non-negative matrix factorization. *Nature* 401:788. doi: 10.1038/44565

Lee, D. D., and Seung, H. S. (2001). "Algorithms for non-negative matrix factorization," in *Advances in Neural Information Processing Systems* (Vancouver, BC), 556–562.

Lee, J. H., Delbruck, T., and Pfeiffer, M. (2016). Training deep spiking neural networks using backpropagation. *Front. Neurosci.* 10:508. doi: 10.3389/fnins.2016.00508

Lustig, M., Donoho, D. L., Santos, J. M., and Pauly, J. M. (2008). Compressed sensing MRI. *IEEE Signal Process. Mag.* 25, 72–82. doi: 10.1109/MSP.2007.914728

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., et al. (2015). Human-level control through deep reinforcement learning. *Nature* 518:529. doi: 10.1038/nature14236

Moreno-Bote, R., and Drugowitsch, J. (2015). Causal inference and explaining away in a spiking network. *Sci. Rep.* 5:17531. doi: 10.1038/srep17531

Oh, J.-H., and Seung, H. S. (1998). "Learning generative models with the up propagation algorithm," in *Advances in Neural Information Processing Systems* (Vancouver, BC), 605–611.

Olshausen, B. A., and Field, D. J. (2006)."What is the other 85 percent of v1 doing," in *23 Problems in Systems Neuroscience*, eds J. L. van Hemmen and T. J. Sejnowsk 182–211. Available online at: https://www.oxfordscholarship.com/view/10.1093/acprof:oso/9780195148220.001.0001/acprof-9780195148220

Pfeiffer, M., and Pfeil, T. (2018). Deep learning with spiking neurons: opportunities and challenges. *Front. Neurosci.* 12:774. doi: 10.3389/fnins.2018.00774

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

Rolinek, M., and Martius, G. (2018). L4: practical loss-based stepsize adaptation for deep learning. *arXiv[Preprint].arXiv:1802.05074*.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychol. Rev.* 65:386. doi: 10.1037/h0042519

Rotermund, D., and Pawelzik, K. R. (2018). Massively parallel fpga hardware for spike-by-spike networks. *bioRxiv [preprint]*. doi: 10.1101/500280

Rotermund, D., and Pawelzik, K. R. (2019a). Back-propagation learning in deep spike-by-spike networks. *bioRxiv [preprint]*. doi: 10.1101/569236

Rotermund, D. and Pawelzik, K. R. (2019b). Biologically plausible learning in a deep recurrent spiking network. *bioRxiv [preprint]*. doi: 10.1101/613471

Rozell, C. J., Johnson, D. H., Baraniuk, R. G., and Olshausen, B. A. (2008). Sparse coding via thresholding and local competition in neural circuits. *Neural Comput.* 20, 2526–2563. doi: 10.1162/neco.2008.03-07-486

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature* 323:533. doi: 10.1038/323533a0

Salakhutdinov, R. (2015). Learning deep generative models. *Annu. Rev. Stat. Its Appl.* 2, 361–385. doi: 10.1146/annurev-statistics-010814-020120

Schmidhuber, J. (2015). Deep learning in neural networks: an overview. *Neural Netw.* 61, 85–117. doi: 10.1016/j.neunet.2014.09.003

Serrano-Gotarredona, T., Linares-Barranco, B., Galluppi, F., Plana, L., and Furber, S. (2015). "Convnets experiments on spinnaker," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)* (Lisbon: IEEE), 2405–2408.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature* 529:484. doi: 10.1038/nature16961

Spanne, A., and Jörntell, H. (2015). Questioning the role of sparse coding in the brain. *Trends Neurosci.* 38, 417–427. doi: 10.1016/j.tins.2015.05.005

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 1929–1958.

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. (2017). Efficient processing of deep neural networks: a tutorial and survey. *Proc. IEEE* 105, 2295–2329. doi: 10.1109/JPROC.2017.2761740

Tavanaei, A., Ghodrati, M., Kheradpisheh, S. R., Masquelier, T., and Maida, A. (2018). Deep learning in spiking neural networks. *Neural Netw* 111, 47–63. doi: 10.1016/j.neunet.2018.12.002

Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., and Fergus, R. (2013). "Regularization of neural networks using dropconnect," in *International Conference on Machine Learning* (Atlanta, GA), 1058–1066.

Wiedemann, T., Manss, C., and Shutin, D. (2018). Multi-agent exploration of spatial dynamical processes under sparsity constraints. *Auton. Agents Multi-Agent Syst.* 32, 134–162. doi: 10.1007/s10458-017-9375-7

Wu, Y., Deng, L., Li, G., Zhu, J., and Shi, L. (2018). Spatio-temporal backpropagation for training high-performance spiking neural networks. *Front. Neurosci.* 12:331. doi: 10.3389/fnins.2018.00331

Zeng, X., He, Z., Yu, H., and Qu, S. (2016). Bidirectional nonnegative deep model and its optimization in learning. *J. Optimizat.* 2016:8. doi: 10.1155/2016/5975120

Zhu, M., and Rozell, C. J. (2015). Modeling inhibitory interneurons in efficient sensory coding models. *PLoS Comput. Biol.* 11:e1004353. doi: 10.1371/journal.pcbi.1004353

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.