

Improving SpikeProp: Enhancements to An Error-Backpropagation Rule for Spiking Neural Networks

Benjamin Schrauwen, Jan Van Campenhout
Ghent University ELIS-PARIS
St. Pietersnieuwstraat 41, 9000 Gent, Belgium
{bschrau, jvc}@elis.ugent.be

Abstract—In this paper, enhancements to the SpikeProp learning algorithm [1] are presented. SpikeProp is an error-backpropagation learning rule suited for supervised learning of spiking neurons that use exact spike time coding. These enhancements provide additional learning rules for the synaptic delays and time constants and for the neurons’ thresholds. This results in smaller network topologies. The simple XOR problem for example needs up to 10 times less weights and learning convergence is up to two times faster.

I. INTRODUCTION

Spiking Neural Networks (SNN) are a hot topic in recent neural network literature [6]. Although spiking neural networks were known even before perceptrons or sigmoidal neurons [2], they have always been overshadowed by them. This is mainly due to computational considerations; spiking neural networks are computationally intensive on traditional sequential architectures. But it has recently been shown that spiking neural networks are very suited for parallel implementation in digital [7] and analog hardware [10]. Another major breakthrough came when was shown that spiking neurons are computationally stronger than sigmoidal networks [5]. Due to these advantages, spiking neural networks are a very promising alternative for sigmoidal networks, and are even called “Third Generation” neural networks [4].

Classic neural networks use analog values to communicate information between neurons. Spiking neural networks on the other hand use the biologically more correct spikes as means of communication. These spike are only identified by the time they occur. A classic neuron computes a non-linear function of the weighted sum of the analog input values. A spiking neuron can generally be viewed as a leaky integrator. All incoming spikes are temporally integrated (with a certain weight), but the integrand decays in time. This integrand is also called membrane potential. Whenever the membrane potential crosses a certain threshold, the neuron emits a spike, and its membrane potential is reset.

One of the open issues in SNN research is how the networks need to be trained. Much research has been done

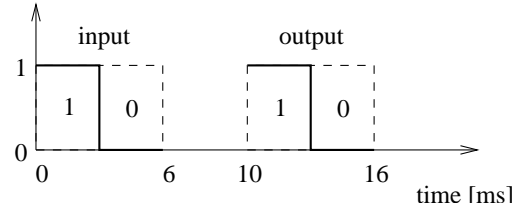


Fig. 1. Binary time to first spike coding.

on biologically inspired local learning rules [3], but these rules can only implement unsupervised learning. The networks can thus not be trained to perform a given task. Classic neural network research became famous because of the error-backpropagation learning rule. Due to this a neural network can be trained to solve a problem which is specified by a representative set of examples. In [1] a supervised learning rule for spiking neural networks was presented. This learning rule, called SpikeProp, operates on networks of spiking neurons that use exact spike time temporal coding. This means that the exact spike time of input and output spikes encode the input and output values. Note that several other coding hypothesis are used in spiking neural network literature. SpikeProp can only operate on one of these hypotheses.

In this paper we chose to encode the input and output values as can be seen in Figure 1. Input values are encoded by firing a spike in the range of 0 to 6 ms, where 0 ms represents a 1 and 6 ms represents a 0. The output of the network is expected in the range 10 ms to 16 ms, again where 10 ms represents a 1 and 16 ms represents a 0. In this paper we use the binary variant where only a 1 or a 0 is encoded, but a analog variant is also possible where for example the input range [0 1] is linearly encoded in the interval [0 6] milliseconds.

SpikeProp assumes a special network topology that was introduced in [9]. Globally the network looks like a classical feed-forward network, but every connection consists of a fixed number of, what is called, delayed synaptic terminals, each having different weights and delays. However, the delays are fixed (1 ms to 16 ms for each of the

16 synaptic terminals per connection) and only the weights can be trained. Because of this the network topology has to be largely over-specified to make all possible weight/delay combinations possible.

This paper extends the SpikeProp algorithm such that the delay and time constant of every connection and the threshold of the neurons can be trained. Because the delays, which are normally fixed, can now be trained, fewer synaptic terminals are necessary, effectively reducing the size of the architecture and thus also the simulation and training time.

The general layout of the network can be seen in Figure 2. Input nodes are in a set called I , hidden neurons in set H , and output neurons in set O . The spike generation process is modeled by the Spike Response Model (SRM) [3]. The SRM consists of three processes: first there is the process that models the effect of an input spike on the membrane potential, this is called the post synaptic potential (PSP)

$$a_j(t) = \sum_{i \in \Gamma_j} \sum_k w_{ij}^k \varepsilon_{ij}^k(t - t_i - d_{ij}^k).$$

Here w_{ij}^k is the weight of synaptic terminal k of the connection between neuron i and j , $\varepsilon_{ij}^k(t)$ is the actual PSP, t_i is the fire time of neuron i and d_{ij}^k is the delay of the synaptic terminal. This is already an extension of what is defined by the standard SRM [3] or what was used by Bohte et al. [1]. Every connection consists of several synaptic terminals, each having their own weights, delays and PSP's, which all can be learned, as will be shown in Section III. In the rest of this paper we assume, without loss of generality, that all PSP's are α -functions:

$$\varepsilon_{ij}^k(t) = \frac{t}{\tau} e^{1 - \frac{t}{\tau}}.$$

The only restriction for the $\varepsilon_{ij}(t)$'s is that their derivatives exist.

Secondly, spike generation is modeled by a threshold process. Whenever the neuron's membrane potential crosses a given threshold ϑ_j , a spike is fired.

The third process modeled by the original SRM is the return of the membrane potential to its resting potential after a spike has been produced. Because we use exact spike time coding, we are only interested in the first spike a neuron produces. The rest of the time course of the neuron is irrelevant. This is why we can leave out this third process. Whenever a neuron fires one spike, it isn't allowed to fire any more spikes.

Note that although time is present while calculating the spiking neural network, the function that is calculated by the network is not a function of time. This function can be

written as $y = f(x)$. During computation of this function both x and y are represented using a virtual time axis: the computation time. The function f , the input x and the output y are themselves time independent.

The rest of this paper is organized as follows: in Section II we give an overview of the important formulas from the SpikeProp algorithm needed in the rest of this paper. In Section III we present the enhancements to the SpikeProp algorithm. Next, in Section IV we will test the proposed techniques on the classical XOR problem. Finally, in Section V we draw some conclusions.

II. SPIKEPROP

SpikeProp is an error-backpropagation training algorithm. The error to minimize is defined on the spike times of the output neurons t_j^a (with $j \in O$) and the desired spikes times t_j^d . Generally the mean squared error is used:

$$E = \frac{1}{2} \sum_{j \in O} (t_j^a - t_j^d)^2.$$

For the exact derivation of this algorithm, please consult [1]. We will only provide those formulas that are needed in the rest of this paper.

The general form of the error derived by a connection's weight is:

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}^k} &= \frac{\partial E}{\partial t_j} (t_j^a) \frac{\partial t_j}{\partial a_j(t)} (t_j^a) \frac{\partial a_j(t)}{\partial w_{ij}^k} (t_j^a) \\ &= \varepsilon_{ij}^k(t_j^a - t_i^a - d_{ij}^k) \delta_j. \end{aligned}$$

Here δ_j is different for neurons in the output layer or neurons in the hidden layers. We use the following notation: the set Γ_j represents all the direct pre-synaptic neurons of neuron j , while the set Γ^j represent all the direct successors of neuron j . For a neuron in the output layer ($j \in O$), δ_j is equal to:

$$\delta_j = \frac{-(t_j^a - t_j^d)}{\sum_{i \in \Gamma_j} \sum_k w_{ij}^k \frac{\partial \varepsilon_{ij}^k}{\partial t} (t_j^a - t_i^a - d_{ij}^k)}.$$

For hidden neurons ($j \in H$) δ_j is equal to:

$$\delta_j = \frac{\sum_{i \in \Gamma^j} \delta_i \sum_k w_{ij}^k \frac{\partial \varepsilon_{ij}^k}{\partial t} (t_i^a - t_j^a - d_{ij}^k)}{\sum_{i \in \Gamma_j} \sum_k w_{ij}^k \frac{\partial \varepsilon_{ij}^k}{\partial t} (t_j^a - t_i^a - d_{ij}^k)}.$$

Here the error-backpropagation is clearly visible because δ_j is dependent on all the δ_i 's of the successors of neuron j .

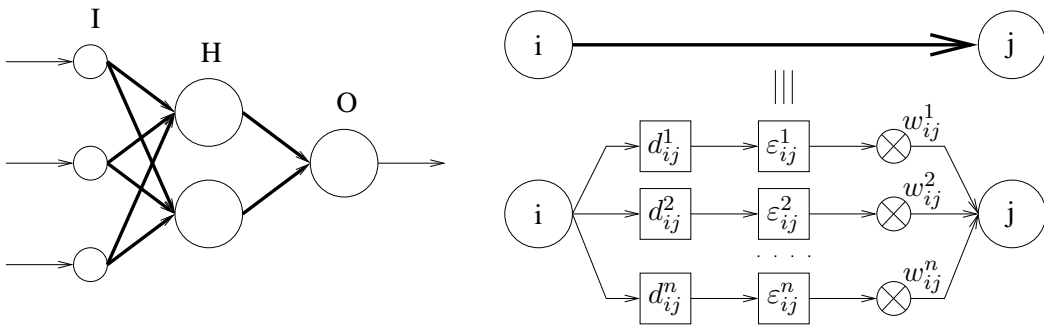


Fig. 2. The network model used by SpikeProp.

The weights are changed according to

$$\Delta w_{ij}^k = -\eta_w \frac{\partial E}{\partial w_{ij}^k}$$

where η_w is the learning rate for the weights. The error is thus minimized by changing the weight according to the negative local gradient. Note that small steps along the gradient are taken because the local gradient is only an approximation of the actual gradient surface.

III. IMPROVEMENTS

We introduce three new learning rules for synaptic delays, synaptic time constants and neuron thresholds.

A. Learning synaptic delays

We take the partial derivative of the error function to d_{ij}^k :

$$\frac{\partial E}{\partial d_{ij}^k} = \frac{\partial E}{\partial t_j}(t_j^a) \frac{\partial t_j}{\partial a_j(t)}(t_j^a) \frac{\partial a_j(t)}{\partial d_{ij}^k}(t_j^a).$$

The two first terms are the same as for the weight update rule, only the last term is different:

$$\begin{aligned} \frac{\partial a_j(t)}{\partial d_{ij}^k}(t_j^a) &= -w_{ij}^k \frac{\partial \epsilon_{ij}^k}{\partial t}(t_j^a - t_i^a - d_{ij}^k) \\ &= -w_{ij}^k \epsilon_{ij}^k(t_j^a - t_i^a - d_{ij}^k) \\ &\quad \left[\frac{1}{t_j^a - t_i^a - d_{ij}^k} - \frac{1}{\tau_{ij}^k} \right]. \end{aligned}$$

If we substitute this, and use the definition of δ_j (which is different for output neurons and hidden neurons) we get:

$$\frac{\partial E}{\partial d_{ij}^k} = -w_{ij}^k \frac{\partial \epsilon_{ij}^k}{\partial t}(t_j^a - t_i^a - d_{ij}^k) \delta_j. \quad (1)$$

The final update rule for the delays is

$$\Delta d_{ij}^k = -\eta_d \frac{\partial E}{\partial d_{ij}^k}$$

where η_d is the learning rate for the delays.

B. Learning synaptic time constants

This time we derive the error function to the time constant of the α -function:

$$\frac{\partial E}{\partial \tau_{ij}^k} = \frac{\partial E}{\partial t_j}(t_j^a) \frac{\partial t_j}{\partial a_j(t)}(t_j^a) \frac{\partial a_j(t)}{\partial \tau_{ij}^k}(t_j^a).$$

The third term can be written as:

$$\begin{aligned} \frac{\partial a_j(t)}{\partial \tau_{ij}^k}(t_j^a) &= w_{ij}^k \frac{\partial \epsilon_{ij}^k}{\partial \tau_{ij}^k}(t_j^a - t_i^a - d_{ij}^k) \\ &= w_{ij}^k \epsilon_{ij}^k(t_j^a - t_i^a - d_{ij}^k) \\ &\quad \left[\frac{(t_j^a - t_i^a - d_{ij}^k)}{(\tau_{ij}^k)^2} - \frac{1}{\tau_{ij}^k} \right]. \end{aligned}$$

When we substitute this and use the definition of δ_j we get:

$$\begin{aligned} \frac{\partial E}{\partial \tau_{ij}^k} &= w_{ij}^k \epsilon_{ij}^k(t_j^a - t_i^a - d_{ij}^k) \\ &\quad \left[\frac{(t_j^a - t_i^a - d_{ij}^k)}{(\tau_{ij}^k)^2} - \frac{1}{\tau_{ij}^k} \right] \delta_j. \end{aligned}$$

The final update rule for the synaptic time constants becomes

$$\Delta \tau_{ij}^k = -\eta_\tau \frac{\partial E}{\partial \tau_{ij}^k}$$

where η_τ is the learning rate for the synaptic time constants.

Note that if a $\epsilon_{ij}^k(t)$ is used with different parameters, other (but similar) training rules needs to be derived.

C. Learning neuron thresholds

Last we derive the error function for the neuron's threshold:

$$\frac{\partial E}{\partial \vartheta_j} = \frac{\partial E}{\partial t_j}(t_j^a) \frac{\partial t_j}{\partial \vartheta_j}(t_j^a).$$

The second term is:

$$\begin{aligned} \frac{\partial t_j}{\partial \vartheta_j}(t_j^a) &= \frac{1}{\frac{\partial a_j(t)}{\partial t}(t_j^a)} \\ &= \frac{1}{\sum_{i \in \Gamma_j} \sum_k w_{ij}^k \frac{\partial \varepsilon_{ij}^k}{\partial t}(t_j^a - t_i^a - d_{ij}^k)}. \end{aligned}$$

If we add the first term, we get the negative of δ_j as learning rule:

$$\frac{\partial E}{\partial \vartheta_j} = -\delta_j.$$

The final update rule for the neuron threshold is

$$\Delta \vartheta_j = -\eta_\vartheta \frac{\partial E}{\partial \vartheta_j}$$

where η_ϑ is the learning rate for the neuron threshold.

D. Solving problems with non-spiking neurons, large synaptic delays and weight initialization

During the training of a network, it is possible that certain neurons stop firing because their membrane potential is unable to reach the threshold. From that moment on, learning the parameters of that neuron gets stuck because all the update rules fail once a neuron stops firing. This is why an extra mechanism was introduced that multiplies the neurons' threshold by 0.9 whenever it stops firing. This process actively lowers the threshold, hoping that the neuron starts firing again, and hence continues its learning cycle. Note that lowering the threshold is equivalent to increasing all the weights.

This mechanism is technically not needed when the learning rate of the parameters is very small. This means that the gradient is followed by very small steps. But if a good learning speed is required, the learning rates are chosen much larger, due to this it is possible that a step is taken that forces a neuron to stop firing. Once this happens, learning gets stuck. Lowering the threshold forces the neuron to fire again.

A second problem occurs when the synaptic delays get too large; it is possible that delayed input spikes reach the neuron *after* an output spike is produced. Therefore the second term in equation 1 can yield zero, causing the delay of that connection to get stuck. This was solved by constraining the delays such that delayed input spikes never surpass output spikes.

Weight initialization was already a problem with the original SpikeProp algorithm. In the original publication this topic was hardly mentioned, but in [8] this point was made clear. The weight initialization is a critical factor for a good operation of the learning rule. The conclusion of

[8] was that weights should be initialized in such a way that every neuron initially fires, and that its membrane potential doesn't surpass the threshold too much. We chose to use a very simple variant where the initial weights are chosen randomly (normal distribution with mean 1.5 and variance 1). Afterwards one weight is set such that the sum of all the weights is equal to 1.5.

IV. RESULTS

As a simple test for the improvements of SpikeProp we train the binary XOR function which was also used in the original publication of SpikeProp [1]. Bohte et al. used a network with three inputs, five hidden neurons and one output neuron. Every connection consisted of 16 synaptic terminals each having fixed delays ranging from 1-16 ms. This gave a total of 320 weights that had to be trained. Note that positive and negative weights could not be mixed, therefore one inhibitory neuron was provided. All connections originating from this inhibitory neuron are negative. The XOR function could be trained with an average of 250 training cycles with a MSE of 1.0 ms. The optimal learning rate was 10^{-2} . Note that all spike times were expressed in milliseconds.

Not only the two inputs of the binary XOR function is provided to the network, but also a third input which acts as a reference or bias input. It will always receive a spike at 0 ms. This is needed to provide a reference start time. Otherwise the network would not be able to differentiate between the double zero or double one input pattern.

Due to the introduction of three new learning rules, we have four learning rates: η_w , η_d , η_τ and η_ϑ . After learning the XOR function with several network architectures, and for many combinations of the learning rates, we can conclude that good choices for the learning rates are: $\eta_w = 10^4$, $\eta_d = 1$, $\eta_\tau = 1$ and $\eta_\vartheta = 10^3$. Some of these learning rates might seem incredibly high, but in our case, all times are expressed in seconds. Due to this the δ 's we use are 10^{-6} times smaller than the δ 's used by Bohte et al. who chose to express all times in milliseconds. If we keep this in mind, we see that our optimal learning rate for the weights of 10^4 is actually just the same as the learning rate of [1]. The learning rates of the delays and time constants look normal, but this is because a millisecond range is used to learn values that are also in the millisecond range, while the weights and the threshold have values around unity, and are used to train values in the millisecond range.

In all our simulation we use mixed positive and negative weights without any problem. Unlike stated in [1] this does not jeopardize the learning. For the given learning rates we obtain a convergence of 80% over a wide range of architectures.

The architecture used by Bohte was a 3-5-1 architecture with 16 synaptic terminals. We got very good results for a 3-5-1 architecture with 2 synaptic terminals per connection. This means that 8 times less weights are needed. A convergence of 90% and 130 training cycles for the converged runs was achieved, this is only half the number of training cycles as in [1]. Convergence was defined by a MSE of 1.0 ms.

It was even possible to learn the XOR function with a 3-3-1 architecture with 2 synaptic terminals per connections with a convergence of 60% and an average of 320 training cycles for the converged runs. This last case uses only 24 weights, which is 13 times less than in [1].

Thresholds are initialized at 1, synaptic time constant at 0.007 seconds, and delays are chosen at random in the range [0 0.003] seconds. Note that the convergence could be further improved by using a more elaborate weight initialization as in [1].

V. CONCLUSIONS

We introduced several extensions to the SpikeProp learning algorithm that make it possible to learn not only the weights, but also the delays and synaptic time constants of the connections, and the thresholds of the neurons. Due to these enhancements, a smaller network architecture can be used. This is mainly due to the fact that delays can now be trained and need not be enumerated.

The simple XOR problem could be solved with the same precision as the original SpikeProp algorithm, but with 8 times less weights (making the simulation and learning phase of the network much faster) and an increased learning convergence.

Future work is necessary to test these extensions on real world problems. This simple XOR problem only shows the basic applicability of these extensions, but to reveal the full potential of these extensions, more elaborate tests are necessary. Also a thorough analysis of the weight initialization problem is required. The convergence rate seems to be pretty sensitive to this. Several techniques used in classic neural networks to speed up backpropagation learning (like quickprop and rprop) could be added to SpikeProp to further speed up learning.

REFERENCES

[1] S. M. Bohte, H. L. Poutré, and J. N. Kok. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1–4):17–37, November 2002. A short version of this paper has been published in the proceedings of ESANN’2000.

[2] P. Dayan and L. F. Abbott. *Theoretical Neuroscience*. MIT Press, Cambridge, MA, 2001.

[3] W. Gerstner and W. Kistler. *Spiking Neuron Models*. Cambridge University Press, Cambridge, England, 2002.

[4] W. Maass. Networks of spiking neurons: the third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

[5] W. Maass. Noisy spiking neurons with temporal coding have more computational power than sigmoidal neurons. In M. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, pages 211–217. MIT Press (Cambridge), 1997.

[6] W. Maass and C. M. Bishop. *Pulsed Neural Networks*. Bradford Books/MIT Press, Cambridge, MA, 2001.

[7] N. Mehrtaash, D. Jung, H. H. Hellmich, T. Schoenauer, V. T. Lu, and H. Klar. Synaptic plasticity in spiking neural networks (SP²INN): A systems approach. *IEEE Transactions on Neural Networks*, 14(5):980–992, September 2003.

[8] S. M. Moore. Back-propagation in spiking neural networks. Master’s thesis, University of Bath, 2002. Available online: <http://www.simonchristianmoore.co.uk/back.htm>.

[9] T. Natschläger and B. Ruf. Spatial and temporal pattern analysis via spiking neurons. *Network: Computation in Neural Systems*, 9(3):319–332, 1998.

[10] L. S. Smith and A. Hamilton, editors. *Neuromorphic Systems: Engineering Silicon from Neurobiology*. World Scientific, 1998.