2006 International Joint Conference on Neural Networks
Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada
July 16-21, 2006

Fast Modifications of the SpikeProp Algorithm

Sam McKennoch*, Dingding Liu, and Linda G. Bushnell
Department of Electrical Engineering
University of Washington
Seattle, Washington 98195-2500 USA
E-mail: skennoch@u.washington.edu

Abstract - **In this paper we develop and analyze Spiking Neural Network (SNN) versions of Resilient Propagation (RProp) and QuickProp, both training methods used to speed up training in Artificial Neural Networks (ANNs) by making certain assumptions about the data and the error surface. Modifications are made to both algorithms to adapt them to SNNs. Results generated on standard XOR and Fisher Iris data sets using the QuickProp and RProp versions of SpikeProp are shown to converge to a final error of 0.5 - an average of 80% faster than using SpikeProp on its own.**

I. Introduction

In recent years, the field of spiking neural networks (SNNs), a machine learning technique similar to traditional artificial neural networks (ANNs), has experienced an increasingly large amount of research attention. SNNs have been shown in theory to be at least as computationally powerful as ANNs, in part because of their ability to process and be trained on signals with a temporal component [8]. In [1], a training method analogous to error back-propagation for ANNs is developed for SNNs and is called *SpikeProp*. Further work has expanded on SpikeProp by verifying the derivation [10] and by investigating the effects of adding a momentum term to the SpikeProp algorithm [17].

In the case of supervised learning, a predescribed set of inputs and outputs, along with a training method are used to train the network to behave in a certain way. Spiking neural networks differ from traditional artificial neural networks in that the signals they process are pulse-coded rather than rate-coded. Feed-forward ANNs take in constant input values and instantly produce an output. Training methods such as standard error back-propagation and the faster RProp and QuickProp adjust the weights contained within the network to produce desired results [11].

A benefit of neural networks compared to classical machine learning techniques is the ability to generalize. That is, given inputs similar to inputs in a training set the outputs should be similar to the trained outputs. This result is exactly what occurs in neural networks. Some training methods such as error back-propagation use a local error gradient to approximate the global gradient and the traditional optimization process of steepest descent.

RProp is one of a number of learning rate adaptation techniques where the rate at which weights are adjusted is determined by the success of the previous change and other related factors. To the best of our knowledge, this paper is the first attempt to adapt RProp to SNNs.

QuickProp uses Newton's method for minimizing a one-dimensional function, the error-gradient in this case, to attempt to take one big step to the minimum. QuickProp then makes appropriate adjustments when the minimum is not directly reached. The work in [17] investigates QuickProp in terms of SNNs. Their approach, however, deviates from the

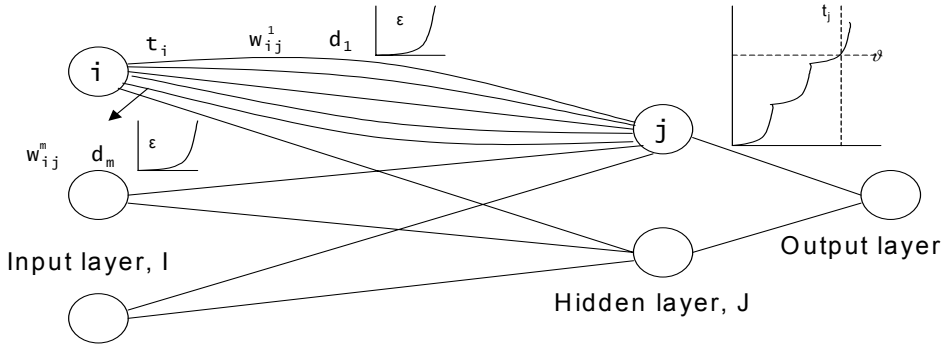

Fig. 1. Spiking Neural Network Diagram

SNNs possess greater biological realism than their ANN counterparts. Inputs to SNNs are pulse-coded. These spikes propagate through the network, effectively charging up neurons and eventually causing them to release their own output spike. Static delays in connections between neurons also play a role.

0-7803-9490-9/06/$20.00/©2006 IEEE

original QuickProp algorithm quite a bit as demonstrated in their equation 19. In this paper, we make adjustments to the QuickProp algorithm, but stay closer to the original concept behind QuickProp.

Instead of using rate-coded signals, SNNs use pulse-coded signals, which is a closer model of actual biological neurons. By incorporating this additional biological realism the power of SNNs increases dramatically. In [10], it is shown that SNNs are at least as computationally powerful as ANNs for learning any given function. SNNs are well-suited to process both analog and digital outputs [16]. SNNs are especially well suited to process natural signals that develop over time. For example, by using pulse-coding instead of rate-coding, SNNs are able to multiplex information such as frequency and amplitude of sound [6].

SNNs take trains of spikes as inputs. Once a neuron receives an input spike, that input connection begins to charge up according to the spike-response function, ε_{ij}^{k}.

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

Subscripts i and j and superscript k signify that this is the spike response between neurons i and j with delay k. As in [1], we create a number of connections between each neuron pair, each having its own static delay, d. This arrangement allows for a number of different weight-delay combinations with only having to train the weights. τ is the time constant that helps determine how quickly a neuron will respond to an input spike.

The spike-responses for all of the inputs are modulated by weights and delays before being combined at a neuron according to the neuron state variable equation, $a_j(t)$,

$$a_j(t) = \sum_{i \in \Gamma_j} \sum_{k} w_{ij}^{k} \varepsilon_{ij}^{k}(t - t_i - d_{ij}^{k}) \tag{2}$$

where Γ_j is the set of input neurons to neuron j, and t_i is the time of the input spike from neuron i to j.

Once this combination of signals exceeds an activation threshold, ϑ, the neuron is activated and produces an output spike at time t_j.

$$t_j \mid a_j(t_j) = \vartheta \wedge \forall t < t_j \; ; \; a_j(t) < \vartheta \tag{3}$$

Just as in feed-forward ANNs, the spikes are propagated from layer to layer and eventually result in a time series of spikes at the output layer. Figure 1 gives further detail on how SNNs process signals.

One major distinction between types of SNNs is in regard to the number of times a neuron can fire. If a neuron can fire only once, there is no need to model what happens to a neuron after it fires, thus greatly simplifying the mathematical modelling. In the more complex case, if a neuron can fire more than once, then typically a kernel is used to model the period of absolute and relative refractoriness. *Refractoriness* refers to the inability of a neuron to immediately fire again once it has initially fired [16]. The original version of SpikeProp was developed for SNNs that fire only once. In [2], a modified version of SpikeProp that can train SNNs in which the neurons can fire multiple times is developed. This type of SNN is more efficient at learning and takes more fully advantage of the temporal information processing ability of SNNs.

Other researchers have also tried to speed up SNNs. For example, in [13] and [14], they develop methods for training the delays, activation threshold, and response curve time constants in addition to the method already developed for the weights. The method requires many less neurons, since multiple neurons with static delays are needed when delays are not trained in order to accommodate different weight/delay combinations.

In [17] they experiment with using a momentum term to speed up the results. Momentum is typically used to help move out of local minimums on the error surface by taking into account previous movements on this surface. We also use momentum in this paper in combination with RProp and QuickProp.

The field of SNN research as it relates to actual biological neurons has been well-studied and is very mature [9], [3]. Typical problems include developing models to functionally simulate parts of the brain. Another example is better modelling of the neuron itself as it relates ion concentrations, rather than a whole network of neurons. The engineering perspective, as with ANNs, is to pragmatically use SNNs to perform engineering tasks regardless of strict biological realism. Training algorithms, such as SpikeProp, have little relation to what actually occurs in biological neurons, but nevertheless they achieve the desired result.

This paper applies the principles of RProp and QuickProp to the original SpikeProp algorithm in order to establish meaningful baseline comparisons. RProp and QuickProp could also be applied equally well to the training methods used in [13], [14] and [2], which is left for future work.

In Section II of this paper we discuss and derive parts of the original SpikeProp algorithm, RProp, QuickProp and the combination of SpikeProp with RProp and QuickProp. Section III details how we set up our SNN simulation, and the reasoning behind our choice of parameters. Section IV contains simulation results and analysis of the SNN trained with regular SpikeProp, RProp and QuickProp on the XOR data set and a simplified Fisher Iris dataset, as well as equivalent results generated in an ANN trained with the same methods. Summary and future work follow in Section V.

II. Training Algorithms

This section begins by summarizing the SpikeProp algorithm. Next the learning-rate adaptation method of RProp is described and analyzed for how best to combine it with SpikeProp. The same description and analysis is then presented for QuickProp.

A. SpikeProp

SpikeProp, derived in [1], uses the same weight update

method as error back-propagation.

$$\Delta w = -\eta \frac{\partial E}{\partial w} \tag{4}$$

The calculation of the error gradient, however, is substantially different reflecting the nature of SNNs. Error is defined by a sum of squared errors (SSE) as applied to the actual firing time t_j^a, and the desired firing time t_j^d, for neuron j, contained in the set of output neurons, O.

$$E = \frac{1}{2} \sum_{j \in O} (t_j^a - t_j^d)^2 \tag{5}$$

As with ANN error back-propagation, the error gradient is separated into separate derivatives.

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial t_j}(t_j^a) \frac{\partial t_j}{\partial a_j(t)}(t_j^a) \frac{\partial a_j(t)}{\partial w_{ij}^k}(t_j^a) \tag{6}$$

The last two partial derivative terms are calculated by making a linear approximation of a around $t=t_j$. Using this assumption, (6) reduces to the following, where δ_j is a complex function that differs for output and hidden neurons.

$$\frac{\partial E}{\partial w_{ij}^k} = \varepsilon_{ij}^k (t - t_i - d_{ij}^k) \delta_j \tag{7}$$

With momentum, (4) becomes the following, where α is the momentum parameter.

$$\Delta w(t) = -\eta \frac{\partial E}{\partial w}(t) + \alpha \Delta w(t-1) \tag{8}$$

SpikeProp is typically used as a batch method, where the error gradient for each input pattern is found and then summed together to adjust the weights.

B. RProp with SpikeProp

RProp is short for resilient propagation. It is one of many learning-rate adjustment algorithms for training neural networks. Other popular ones include Vogl's Method and Delta-Bar-Delta [11]. RProp is unique in that the learning rate adjustments depend only on the sign of the gradient, not the magnitude. If the error surface is highly non-linear and complex, the gradient magnitude becomes unpredictable and is a poor measure of how the learning rate should be adjusted.

The learning rate in RProp is updated according to the following formulas in (9) and (10).

$$\Delta_{ij}(t) = \begin{cases} \eta^+ \cdot \Delta_{ij}(t-1), \text{ if } \frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ \eta^- \cdot \Delta_{ij}(t-1), \text{ if } \frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) < 0 \\ \Delta_{ij}(t-1), \text{ otherwise} \end{cases} \tag{9}$$

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t), \text{ if } \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ +\Delta_{ij}(t), \text{ if } \frac{\partial E}{\partial w_{ij}}(t) < 0 \\ 0, \text{ otherwise} \end{cases} \tag{10}$$

The first equation says that the learning rate can be increased if no minimum is jumped over, and likewise if a minimum is jumped over (resulting in a change of signs in the error term) then the learning rate should be decreased as the minimum is narrowed in on. The second equation simply determines the direction to adjust the weights based upon the direction to the minimum. In ANNs, choosing the initial value of Δ_{ij} is important to avoid node saturation. For additional details on RProp refer to [11], [12].

In order to combine RProp with SpikeProp, a number of issues need to be considered. Some of these issues are described in [10], where the SpikeProp algorithm is implemented with details that were left out of the original paper, [1]. One of the main difficulties we encountered was making sure the parameters were set in such a way that at least initially every neuron fires. A neuron that does not fire makes no contribution at all to the final result, greatly decreasing the effective network size. In the worst case, if a whole layer of neurons which only fire once, does not fire, the SNN will fail to produce an output. Also for the once-firing SNN, after a reasonable amount of time has gone by and the last input neuron to another neuron has fired, if that second neuron has not yet fired then it never will. If it were possible for its input neurons to fire again, then a future firing would still remain possible. Thus in reasoning about using RProp and QuickProp with SpikeProp, the most important consideration in choosing setup parameters is to make sure that all input neurons will initially fire and that non-input neurons will also fire subsequent to the input neurons firing.

A neuron will not fire or will fire too late if its input weights are too small, so it is desirable to decrease weights slowly. On the other hand, if the weights get too large, the neuron will fire immediately, effectively becoming saturated. So weights should increase slowly as well.

In this work, we initially propose $\eta^+ = 1.2$ and $\eta^- = 0.5$ as RProp parameters that meet this criteria. Rather than specify maximum and minimum values of $\Delta_{ij}(t)$ as is normally done with RProp, for SNNs it makes more sense to look at maximum and minimum values of input weights. If these limits are exceeded, $\Delta_{ij}(t)$ is not changed. Working from (1), (2), and (3), we derive:

$$w_{max} = \frac{\tau \vartheta}{|d| |\Gamma_i| t_{min}} e^{\frac{t_{min}}{\tau} - 1} \tag{11}$$

$$w_{min} = \frac{\tau \vartheta}{|d| |\Gamma_i| t_{max}} e^{\frac{t_{max}}{\tau} - 1} \tag{12}$$

The variables t_{min} and t_{max} are the minimum and maximum change in time allowed between when a neuron receives an input spike and when it produces one, after any applicable delay. $|d|$ is the number of delay connections between any two neurons. $|\Gamma_i|$ is the number of input neurons for a given layer. As an example, the XOR values given in Table 1 are used, with t_{min}=0.1 and t_{max}=10. These values are based on network size and input and output spike-time range. Note that the times are relative to the firing time of the previous layer. Using these values, w_{min} = 0.08 and w_{max} =

2.04. Equations (11) and (12) are used to initialize the network weights as well. In erring on the side of caution, we initialize our weights with a normal distribution of:

$$\mu = \frac{2}{3}(w_{max} - w_{min}) + w_{min} \quad (13)$$

$$\upsilon = \frac{1}{6}(w_{max} - w_{min}) \quad (14)$$

Simulations described in Section IV show the result of using a number of different learning rate values.

C. QuickProp with SpikeProp

QuickProp is a training method that tries to approximate the global error surface by examining the local one for each weight. There are two underlying assumptions:

- The error for each weight can be approximated by a parabola opening upward.
- The second derivative of the error with respect to one weight is not affected by the other weights that change at that time.

Given these assumptions, Newton's method for minimizing a one-dimensional function is used as follows:

$$\Delta x = -\left(\frac{f'(x)}{f''(x)}\right) \text{ iterate to minimize } f(x) \quad (15)$$

$$\Delta w(t) = \frac{\frac{\partial E}{\partial w}(t)}{\frac{\partial E}{\partial w}(t-1) - \frac{\partial E}{\partial w}(t)} \Delta w(t-1) = \beta \Delta w(t-1) \quad (16)$$

Equation (16) uses a finite-difference approximation to calculate the second derivative. Specific rules are used in addition to this equation to prevent infinite steps or steps up the error surface, which can take place when the original assumptions are violated. For example, when $(\partial E/\partial w)(t)$ and $(\partial E/\partial w)(t-1)$ have the same sign, then a simple gradient descent term of $-\eta(\partial E/\partial w)(t)$ is added to (16). A variety of other add-on heuristics are used to further speed up training and prevent saturation. For additional details on QuickProp refer to [5], [11].

In applying QuickProp to SNNs it is useful to consider the sensitivity to weight changes on the error. Sensitivity calculations are also used in network pruning algorithms. For a sigmoidal ANN, sensitivity assuming one input neuron is:

$$\frac{\partial a}{\partial w} = \frac{i e^{-iw}}{(1 + e^{-iw})^2} \quad (17)$$

Regardless of the initial value in w, any change from that initial value on a will be bounded. For SNNs, if we use a first-order Taylor series approximation of the exponential in (1), the sensitivity becomes:

$$\frac{\partial t_j}{\partial w} = \frac{-(t_j - t_i - d_k)(t_j - t_i - 2d_k)}{w(2 - d_k)} \quad (18)$$

Thus we see that for small initial weights, changes to these weights can have a large impact on the output firing time. QuickProp bases the current weight change on previous weight changes, which can have the effect of amplifying this sensitivity. In our experiments we have found the following changes to be helpful:

- Multiply β by a factor between 0 and 1 to diminish its effect.
- In the case where the risky assumptions have been violated (the current gradient and the previous gradient have the same sign, and the current gradient is larger or the same size than the previous gradient), set $\beta = 0$ rather than using a maximum growth factor to limit growth.

These changes ensure that changes to the output firing time happen at a measured pace and decrease the likelihood of moving in the wrong direction from the error minimum. Weight initialization takes place in the same way as that described in Section IIB.

III. Simulation Setup

Simulations of SpikeProp and its RProp and QuickProp modifications described in Section II were conducted using Matlab™ due to its ability to rapidly prototype. Functions were built to create a SNN structure, to run a SNN (find the outputs given inputs), and to train the SNN network based on pure SpikeProp or its RProp or QuickProp modifications.

Two datasets were used. The first dataset was the binary XOR dataset, in the form used in [1]. For the most part, identical parameters to those in [1] were used. Parameters used in this test and the ones to follow are summarized in Table 1. Where used in Table 1, the notation *x:y:z* refers to a range of values that vary from *x* to *z* at intervals of *y*. The notation used for network size refers to the number of neurons in the input, hidden, and outputs layers. Weights were initialized as discussed in Section II with the additional rule that 10% of the weights were chosen to be inhibitory (negative) initially. We place no restriction on mixing inhibitory and excitatory weights. Further biological realism dictates that because these connections in nature are generally separate, they should be separate in our networks as well. However, we've not found any deleterious effect from mixing the two types.

Because of the simplicity of the XOR dataset, no coding is needed. Rather spiking times at 0 and 6 represent the binary values of 0 and 1 respectively. Likewise for the outputs, firing times 10 and 16 are assigned to be equivalent to binary values 0 and 1 respectively. These times are chosen in [1] to allow binary 0 and 1 to be easily separable and to allow sufficient time for the signals to propagate through the network based on other parameters such as τ.

Because the inputs to the Fisher Iris data set are not binary, a more complex encoding scheme must be used than that used for the XOR data set. As with [1] we use population coding. In population coding, multiple local receptive fields are used as a way of distributing the rate-coded input values into multiple pulse-coded input firing times. We used four overlapping Gaussian receptive fields evenly distributed throughout the expected range of rate-coded input values to

TABLE 1: Network Simulation Parameters

	XOR	Fisher Iris
Inputs	Binary XOR inputs with firing times 0 and 6 signifying binary 0 and 1, plus a zero-valued referenced input firing time	10% Random sample of the 150 sample dataset using population coding
Outputs	Binary XOR outputs with 10 and 16 signifying binary 0 and 1	5; 10; and 15 for the three iris types
Delays	1:1:16	1:1:16
Network Size	[3 5 1]	[16 7 1]
Sim Time	0:0.025:50	0:0.05:50
Tau	7	7
Activation Threshold	5	5
Alpha	0.1	0.1
BackProp Learning Rate	0.05:0.05:0.5	0.15
RProp Positive Learning Rate	1.05:0.05:1.5	1.2
RProp Negative Learning Rate	0.4:0.1:0.6	0.5
RProp Initial Del	0.01	0.01
QuickProp Learning Rate	0.05:0.05:0.2	0.1

code each variable. For example, an input of 10, might produce a list of spike times of [7 4 ∞ ∞]. The last two receptive fields in this case are not sufficiently activated and translate to two input neurons that are not activated. Population coding allows for a more efficient use of neurons, and also has the effect of normalizing the input data. The use of population coding for spiking neural networks has been well studied [4], [18]. Other potential uses of population coding include using Gaussian receptive fields of different sizes to make the coding more insensitive to the scale of the input data [1]. The outputs firing times 5, 10 and 15 are assigned to be equivalent to the three iris types. These times are again chosen to allow iris types to be easily separable and to allow sufficient time for the signals to propagate through the network based on other parameters such as τ.

IV. Simulation Results

In [17], momentum was found to have an oscillatory effect on error from epoch to epoch. We also encountered this result, but also found that momentum helped the system converge quicker on the whole. As noted in [11], momentum increases the effective learning rate and can have a destabilizing effect in ANNs. On average, however, we found that simulations run with momentum, as opposed to those without momentum, regardless of the training method used, converged to an mean squared error (MSE) of 0.5 about 25% quicker.

Figure 2 and Figure 3 show plots of the same networks trained with RProp over a variety of values of momentum and simulation time interval. For all cases in these two figures, the error to which the network was to be trained was set at 0.01, a relatively low value. Additional oscillation was encountered when attempting to train a network to a very low error. Error is measured in terms of spike timing. The resolution to which spike timings can be measured is determined a priori by the simulation time interval. If a network is attempted to be trained to an error level lower than the simulation time interval allows, then the error will tend to oscillate back and forth between two values, without any overall decrease in the error. Also, the time interval value affects the path taken over the error surface. For higher time intervals, steps taken are larger, which also increases the effective learning rate, leading to not only decreased convergence time but also decreased stability.

A comparison of typical MSE curves over different training methods is presented in Figure 4. RProp and QuickProp training finishes approximately 80% quicker than SpikeProp. This percentage holds across the many simulations we did.

The life history of a weight value is shown in Figure 5 that is typical of the three training methods. As indicated, weights more often decreased than increased, thereby increasing the

TABLE 2: Number of Epochs to 0.5 MSE with RProp on XOR data for values of η^+ (horizontal) and η^- (vertical)

	1.05	1.1	1.15	1.2	1.25	1.3	1.35	1.4	1.45	1.5	Mean
0.4	95	53	84	40	35	27	25	36	36	53	52
0.5	64	51	45	104	44	32	44	159	34	59	64
0.6	75	91	46	140	32	25	29	53	25	50	56
Mean	78	65	58	95	37	28	33	89	32	54	

TABLE 3: Number of Iterations to 0.5 MSE with binary XOR dataset

	ANN	SNN
Back-Propagation	2750	127
RProp	386	29
QuickProp	51	31

TABLE 4: Number of Iterations to 0.5 MSE with partial Fisher dataset

	ANN	SNN
Back-Propagation	1370	222
RProp	76	25
QuickProp	79	53

firing time.

As noted in Table 1, learning parameters were varied to be able to analyze their effects on convergence in the SNN domain. Table 2 displays the results for training with RProp on the XOR data while the learning parameters are varied. Each entry represents the average number of epochs for 10 different simulations. Generally, high positive learning rates appear to help convergence along. It should be noted, however, that if we were to train to a lower error value, such as an MSE of 0.01, higher learning rates would decrease the chances of convergence. RProp is inherently an adaptable learning rate training method, but the rate at which the learning is adapted could also be scheduled to decrease over time to maximize early performance and also maximize the probability of convergence.

Table 3 and Table 4 summarize the results of training on the XOR and Fisher Iris data set for both ANNs and SNNs over all three training methods. Recall, SpikeProp is back-propagation for SNNs. Each entry represents an average of ten or more simulations generated with the range of learning parameters described in Table 1. The exception is RProp, which based on the results in Table 2 was generated using $\eta^+ = 1.3$ and $\eta^- = 0.5$. The results are compared to results generated using ANNs. ANN results were generated using the Qwiknet tool [7]. Identical network size and parameters were used relative to the SNN training. A learning rate 0.1 was used for both QuickProp and SpikeProp. SNNs performed an average of 68.5% quicker than their ANN equivalents. Also, RProp and QuickProp are shown to train an average of 80% quicker than back-propagation. RProp performed slightly better than QuickProp for SNNs, while QuickProp was much better than RProp was for the XOR dataset and equivalent for the Fisher Iris dataset.

V. Conclusion and Future Work

In this paper we have demonstrated the value of using RProp and QuickProp algorithms in Spiking Neural Networks. We have shown that with only minor modifications to the network parameters, these algorithms can be ported directly from the non-spiking world. Quantitatively, we've shown an average improvement over SpikeProp of 84.5% and 76% respectively with RProp and QuickProp over two standard data sets. We have also shown an improvement of 68.5% of SNNs over ANNs for all three training methods discussed. The effect of momentum was briefly looked at in the context of these training methods as were the impacts of

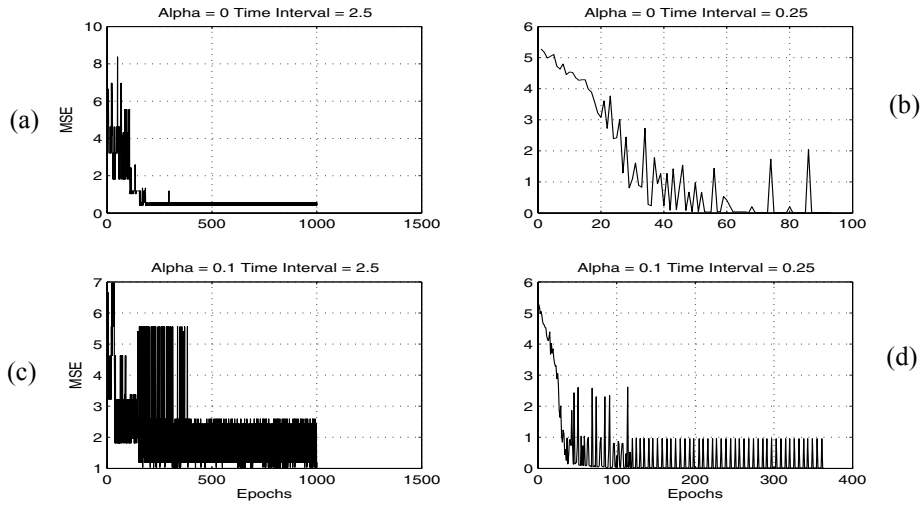

Fig. 2. MSE for the same network trained with RPRop with and without momentum and with time intervals 2.5 and 0.25

Decreased time interval in (a) to (b) decreases the effective learning rate, but allows the network to be trained to a lower error. The time interval in (a) and (c) is too high to allow the network to be trained. In both (c) and (d), the effect of momentum is to increase the learning rate, but to also decrease stability.

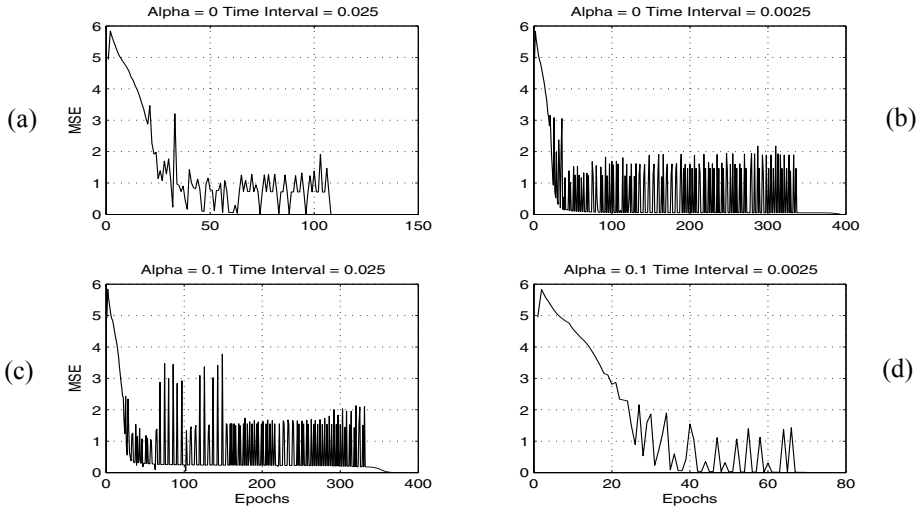

Fig. 3. MSE for the same network trained with RPRop with and without momentum and with time intervals 0.025 and 0.0025

Decreased time interval in (a) to (b) decreases the effective learning rate, and leads to slow convergence due to decreased stability. Increased Momentum in (a) to (c) causes a slightly quicker initial drop in error, but at the expense of decreased stability and increased oscillations. In both increasing the momentum and decreasing the time interval in (d), the forces on the effective learning rate balance each other in a beneficial way, and convergence happens 27% quicker than in (a).

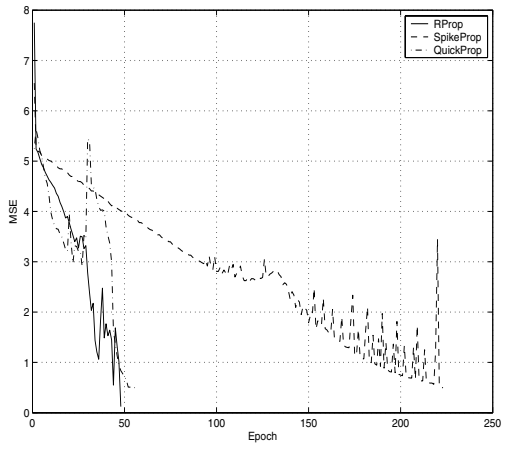

Fig. 4. Typical MSE for networks trained with SpikeProp, RProp, and QuickProp

RProp and QuickProp both converge around 50 epochs, while SpikeProp takes 227 epochs. Oscillations are due both to lack of infinite time resolution and the presence of momentum. Training parameters used were η =0.1 for QuickProp and SpikeProp, η^+ = 1.2 and η^- = 0.5 for RProp.

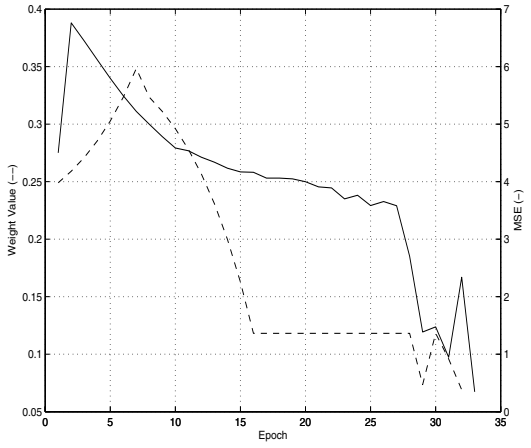

Fig. 5. Weight Lifecycle vs MSE

Weights were more likely to decrease than increase over the training time. Changes in weight values often directly translated into changes in MSE showing in part the large sensitivity of error to changes in the weights.

weight changes and initialization.

In the future, we will would like to apply RProp, QuickProp and a number of other standard ANN training algorithms to increasingly complex SNNs. Such networks would include those described in [14] where the delays and other parameters are trained. Another example is the SNN where a neuron is allowed to fire multiple times [2]. As neurons in SNNs are allowed to fire multiple times they become much more powerful, but also much more complex, thus showing the need for faster, more powerful training algorithms. Research into training the simulation time interval would be beneficial as well.

We also hope to apply SNNs to the domain of communication learning in multi-agent systems. The strengths of SNNs lend themselves well to this domain. More specifically, SNNs are powerful machine learning tools (both supervised and unsupervised) which are adept at dealing with natural spatio-temporal signals. Also SNNs are very powerful at handling event-driven systems of the kind found in multi-agent systems.

ACKNOWLEDGEMENTS

This work is supported in part by the US National Science

Foundation Grant No. ECS-0322618 and Grant No. ECS-0137422.

REFERENCES

[1] Bohte, S., "Spiking Neural Networks," Ph.D. dissertation, Centre for Mathematics and Computer Science (CWI), Amsterdam, 2003.

[2] Booij, O., and Nguyen, H., "A gradient descent rule for spiking neurons emitting multiple spikes," *Information Processing Letters*, Vol. 95, No. 6, 30 September 2005, pps. 552-558.

[3] Eliasmith C., and Anderson C., *Neural Engineering*. Cambridge, Massachusetts: The MIT Press, 2003.

[4] Eurich, C. W., and Wilke, S. D. "Multi-dimensional encoding strategy of spiking neurons," *Neural Computation*, Vol. 12, 2000, pp. 1519-1529.

[5] Fahlman, S. E., "Faster-learning variations of back-propagation: An empirical study," in Touretzky D. S., Hinton G., Sejnowski T., Eds., *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo: Morgan Kaufmann, 1988, pps. 38-51.

[6] Gerstner, W., Kempter, R., Leo van Hemmen, J. and Wagner, H., "Hebbian Learning of Pulse Timing in the Barn Owl Auditory System," in Maass, W. & Bishop, C. M., Eds., *Pulsed Neural Networks*, Cambridge, Massachusetts: The MIT Press, 1999.

[7] Jensen, C., "QwikNet: Professional Neural Network Software," http://qwiknet.home.comcast.net/, 2002.

[8] Maass, W., "Networks of Spiking Neurons: the Third Generation of Neural Network Models," *Proceedings of Australian Conference on Neural Networks*, Canberra, 1996, pp. 1-10.

[9] Mastebroek, H. and Vos J., Eds., *Plausible Neural Networks for Biological Modelling*. Norwell, MA, Kluwer Academic Publishers, 2001.

[10] Moore, S. C., "Back-Propagation in Spiking Neural Networks," M.S. thesis, University of Bath, 2002.

[11] Reed, R. D., and Marks II, R. J., *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, Massachusetts: The MIT Press, 1999.

[12] Reidmiller, M. and Braun H., "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *IEEE International Conference on Neural Networks*, San Francisco, Vol. 1, 1993, pp. 586-591.

[13] Schrauwen, B. and Van Campenhout, J., "Extending SpikeProp," *Proceedings of the International Joint Conference on Neural Networks*, 2004. pp. 471-476.

[14] Schrauwen, B. and Van Campenhout, J., "Improving SpikeProp: Enhancements to An Error-Backpropagation Rule for Spiking Neural Networks," *Proceedings of the 15th ProRISC Workshop*, 2004.

[15] Wennekers, T., "Analysis of Spatio-Temporal Patterns in Associative Networks of Spiking Neurons,", *Proceedings of ICANN*, 1999.

[16] Vreeken, J., "Spiking neural networks, an introduction," Technical Report UU-CS-2003-008, Institute for Information and Computing Sciences, Utrecht University, 2002.

[17] Xin, J., & Embrechts, M., "Supervised learning with spiking neural networks," *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, 2001, pp. 1772-1777.

[18] Zhang, K., and Sejnowski, T. "Neuronal tuning: To sharpen or broaden?," *Neural Computation*, Vol. 11, No. 1, 1999, pp. 75-84.