Improved p-Delta Learning Algorithm

R. MIRSU V. TIPONUT L. PETROMANJANC Z. HARASZY
Applied Electronics, "POLITEHNICA" University of Timisoara
Timisoara, Str. Vasile Parvan, Nr.2
ROMANIA
radu.mirsu@etc.upt.ro virgil.tiponut@etc.upt.ro lilijana@gmail.com
zoltan.haraszy@etc.upt.ro

Abstract: - This paper presents a modified version of the p-Delta learning algorithm. The algorithm can be used for training a parallel perceptron regardless of the application. There are three significant changes from the original algorithm: an adaptive learning rate, a conscience mechanism and an adaptive margin enhancement mechanism. The changes offer an improved speed, stability and noise margin at the expense of complexity. The higher complexity can be a drawback if the algorithm is intended to be implemented in hardware.

Key-Words: - perceptrons, parallel perceptrons, perceptron delta learning rule, liquid state machines, noise margin, adaptive learning

1 Introduction

The p-Delta Learning Algorithm was introduced by Peter Auer in [1] as a direct solution to designing the readout units of a Liquid State Machine [2]. In general, the learning rule can be used for training any parallel perceptron regardless of the application. The original algorithm was developed assuming that it will be applied to a system implemented exclusively in hardware. Inside such a system communication between individual neurons and the control unit can be a major difficulty. Therefore, the algorithm had one important constraint: simplicity. In this paper a new version of the algorithm is presented. It introduces a few modifications that make the algorithm faster, more stable and with a higher noise margin. However, the changes complicate the algorithm making it less suitable for a hardware implementation. Currently, our team is using a software model for the spiking neural network and is not aiming towards a software independent implementation. This allows a more complicated algorithm to be easily implemented.

2 Parallel Perceptron

A single perceptron, as introduced by McCulloch-Pitts, is a gate that computes an averaged sum of all inputs. If the sum is greater than the threshold ***TH*** the perceptron outputs "1" otherwise "0". Mathematically, this is written as in formula (1).

$$y(p) = \begin{cases} 1, wp > TH \\ 0, wp < TH \end{cases} \tag{1}$$

where ***w*** is the synaptic weight vector and ***p*** is the input. The perceptron model can be easily implemented by a spiking neuron [3] if p is considered to be the rate of the spike train. The output is "1" when the neuron fires and "0" otherwise.

A parallel perceptron is a group of N single perceptrons that are fed with the same input ***p***. The output of the parallel perceptron ***Y*** is produced by a squashing function ***S*** that counts the number of active neurons and maps this number onto a continuous value, as in formula (2).

$$Y(p) = S\left(\sum_{i=1}^{N} y_i(p)\right) \tag{2}$$

The squashing function ***S*** can be any monotonous continuous function. However, for this paper the linear function from formula (3) was used.

$$S(n) = \frac{n}{N} * \left(Y_{\max} - Y_{\min}\right) + Y_{\min} \tag{3}$$

where $Y_{\max}, Y_{\min}$ are the boundaries of the output range and *n* is number of active neurons.

The next chapter will present the original learning rule for the parallel perceptron.

3 "p-Delta" Training Algorithm

The algorithm can be efficiently used for training a parallel perceptron to map a set of given input data ***p*** to a desired target output ***t***.

3.1 The Single Perceptron Delta Rule

This is the simplest learning rule that can be applied to a single perceptron. Let ***p, y*** and ***t*** be the input, output and target data respectively. If the output ***y*** is '0' and the target ***t*** is '1' it means that the dot product ***wp*** is too small in comparison to the desired threshold ***TH***. In order for the dot product to increase, the weight vector ***w*** needs to move toward the data vector ***p***, hence the angle between the two vectors will decrease. If the output ***y*** is '1' and the target ***t*** is '0' it means that the dot product ***wp*** is too large and so the weight vector needs to move away from the data vector. If the output ***y*** matches the target ***t*** no change is done. The rule can be mathematically presented as by formula (4).

$$w \Leftarrow \begin{cases} (1-\lambda) * w + \lambda * \begin{cases} +p, t > y \\ -p, t < y \end{cases} \\ w, t = y \end{cases} \tag{4}$$

where λ is the learning rate.

3.2 The Parallel Perceptron p-Delta Rule

In theory the approximation error of the parallel perceptron can be as small as half the size of the quantization step. Therefore, the algorithm could theoretically set the desired accuracy ε to the value presented in formula (5), where $Y_{\min}$ and $Y_{\max}$ are the same as in formula (3).

$$\varepsilon = \frac{Y_{\max} - Y_{\min}}{2 * N} \tag{5}$$

However, reaching this error level is not guaranteed. This is because the algorithm can get stuck in a local error minimum and so it will not find the global minimum that satisfies formula (5). Therefore, from now on it is considered that the accuracy ε is set by the application and that the number of neurons N is sufficient for the accuracy constraint to be met.

Given the input data ***p***, the output of the parallel perceptron ***Y(p)*** is computed with formula (3). If the weights of the parallel perceptron are correct the output should be as close to the target ***t*** as constrained by ε. This is expressed in formula (6).

$$|Y(p) - t| < \varepsilon \tag{6}$$

If the output is greater than the target it means that too many neurons are active and so the weights of "some" of the active neurons should move away from the data. If the output is too small compared to the target, too few neurons are active and so "some" of the inactive neurons should move their weights towards the data.

The term "some" is flexible and represents the answer to the question: "how many and which neurons should be chosen for weight modification?" The authors of [1] suggest that all active neurons should be updated if the output is greater than the target and also that all inactive neurons should be updated if the output is smaller than the target. This approach does not offer a great convergence speed or stability. However, it minimizes communication between neuron units if a hardware implementation is preferred. In [1] it is also suggested that the stability and convergence speed could be improved if only a few neurons (or one [5]) are chosen for weight modification. Those neurons should be the ones that have a dot product ***wp*** that is closest to the threshold. This approach on the other hand increases communication as the neuron units would need to broadcast their dot product to the central unit.

Because this paper uses a software implementation of the neural circuitry, communication bandwidth is not a constraint. Therefore, it was chosen that each training iteration updates the weights of only one neuron which is declared winner. A neuron is declared winner if it has a dot product that is closest to the threshold and also if is on right side of the threshold. Therefore, the learning rule can be mathematically expressed as in formula (7).

$$w_k \Leftarrow (1-\lambda) * w_k - \eta * \left(\|w_i\|^2 - 1\right) * w_i + \\ + \lambda * \begin{cases} -p, Y(p) - t > \varepsilon \\ +p, t - Y(p) > \varepsilon \end{cases} \tag{7}$$

where:

- λ learning rate
- η normalization rate
- k winning neuron
- *i =1….N*

The middle term that contains the norm of the weight vector is a correction that is performed for each neuron on all iterations. This correction preserves the angle of the weight but brings the length of the vector to unit length. The correction is important because the dot product ***wp*** signifies the angle between the two vectors only if the lengths of the vectors remain bounded.

4. Adaptive learning rate

The first modification to the original algorithm is the introduction of an adaptive learning rate. The learning rate is recomputed at each iteration as in formula (8).

$$\lambda \Leftarrow \lambda_{\max} * \frac{msq\_error}{mean\_msq\_error} \qquad (8)$$

The learning rate starts from a maximum value $\lambda_{\max}$ and then decreases as the parallel perceptron starts to approximate the data well.

5. Greedy vs. NotGreedy

The second modification to the algorithm is the implementation of a conscience mechanism. A statistical study was done to see how fast the algorithm converges.

The algorithm is considered to have converged when the parallel perceptron approximates the target with an error smaller than ε for every data point in the training set. 10000 simulations were performed for every data point ***p*** and target ***t***. The target ***t*** is the result of a randomly chosen linear function that takes ***p*** as input variable. Each simulation starts with different initial weights for the neurons, records the number of epochs that the algorithm needs to converge and places it in a convergence histogram. Such a histogram is illustrated in figure 1.

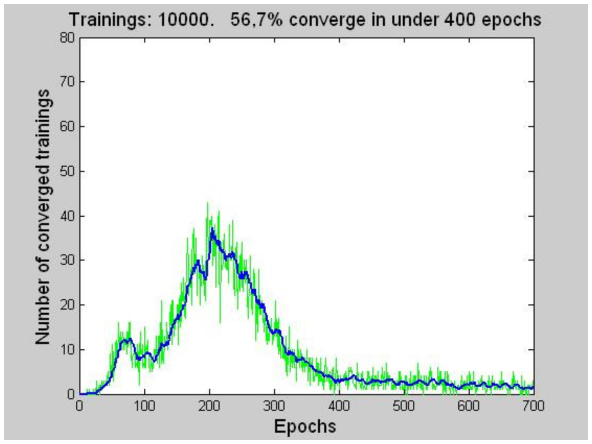

Fig. 1. PDelta Convergence histogram

It is seen that most trials converge in less than 400 epochs (aprox. 56.7%). Some trials converge in more than 400 epochs but it is most likely that their convergence is caused by chaotic effects and therefore is unreliable. Because the convergence percentage is not very high it was interesting to see what prevents the other trials from converging.

An activity monitor variable was attached to each neuron forming the parallel perceptron. The activity variable counts the number of times the weight of a neuron is updated during the current epoch. Then, it divides the count to the total number of updates performed during the epoch for all of the neurons. After the epoch is finalized the activity variable reflects a percentage of how often was a neuron declared winner. Figure 2 plots the activity traces for all the neurons during a trial that did not converge (each neuron is plotted in a different color).

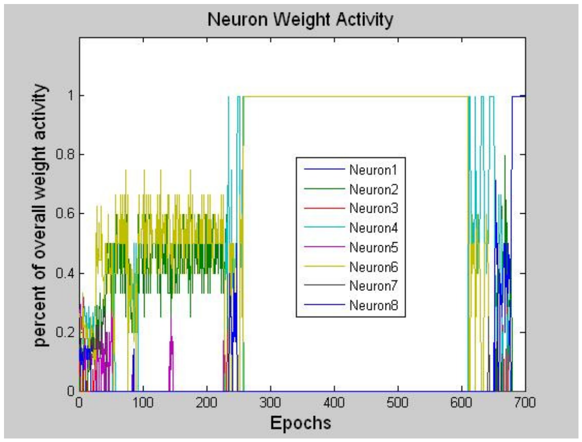

Fig. 2. Distribution of weight activity (“greedy” approach)

It is seen that initially several neurons have their weights updated. However, at some point, only one neuron is chosen exclusively for weight modification. This “greedy” behavior occurs when a neuron reaches a region that is densely populated with data and no other neuron is in the same region. In order for the minimum error to be reached it is required that several neurons are present in this region such that the quantization is smoother. Unluckily, no other neuron is close enough to the data and so the single isolated neuron will always win the competition preventing other neurons to approach the region.

In order to avoid this greedy behavior a conscience mechanism is inserted in the scoring function that is responsible for selecting the winner neuron. The scoring function calculates two scores: a proximity score ***PS*** and an activity score ***AS***. Both scores are sub-unitary and reflect the probability of a neuron to be declared winner. The proximity score ranks the neurons based on dot product comparison. ***PS*** will be 1 for the neuron with a dot product that is closest to the threshold ***TH*** and 0 for the neuron that is furthest away. The activity score is computed by monitoring the activity trace of each neuron *i* inside a window of given size ***WS***. The activity score ***AS*** is computed at any time *t* as given by formula (9).

$$AS_i(t) = 1 - \frac{1}{WS} * \sum_{k=1}^{WS} activity\_trace_i(t-k) \tag{9}$$

The overall score is the product of the two scores ***PS*** and ***AS***. The neuron with the highest overall score is declared winner. The neuron weight activity trace for the "not greedy" approach is illustrated in figure 3. It is seen that in this case no neuron dominates as all neurons change weights throughout the epochs of the algorithm.

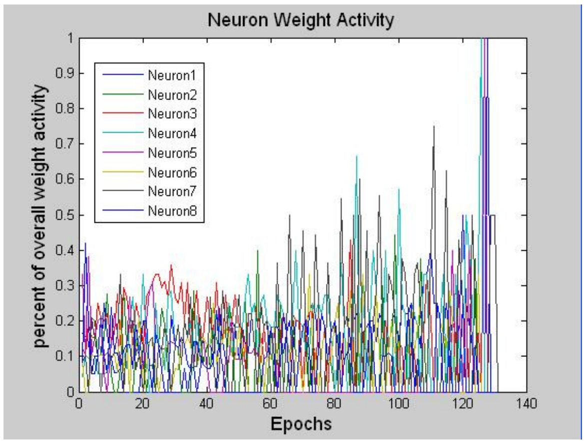

Fig. 3. Distribution of weight activity ("not greedy" approach)

It is seen that with this approach the algorithm converges a lot faster (140 epochs). In order to graphically compare the "greedy" and "not greedy" methods a similar histogram as the one in figure 1 was computed. Figure 4 plots the cumulated sums of several such histograms. The blue trace represents the cumulated sum of the "greedy" histogram in figure 1. The other traces are cumulated sums of histograms obtained with the "not greedy" approach for several values of the window size ***WS***. It is seen that the size of the averaging window ***WS*** does not significantly influence the convergence speed of the algorithm. However, it is also seen that the "not greedy" approach converges a lot faster than the "greedy" approach and also that the number of un-converged trials is significantly reduced. The same data is numerically available in table 1.

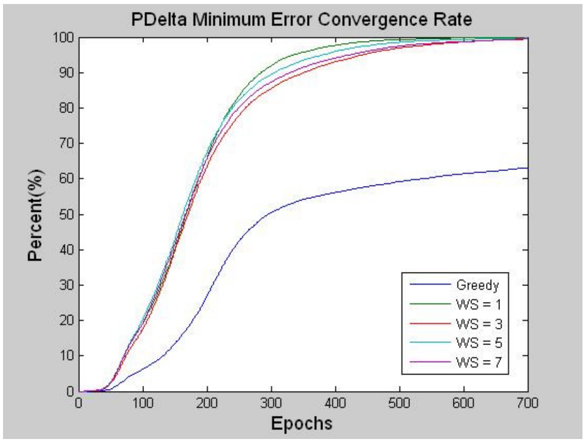

Fig. 4. Convergence Rate

Method	***Converge under x epochs***			***Never converge***
	100ep	200ep	400ep	
Greedy	6.21%	26.94%	56.13%	36.89%
Not Greedy	17.56%	63.38%	93.12%	0.53%

Table 1. Greedy vs. Not Greedy Convergence Statistic

6. Adaptive Noise Margin Control

In that presented above the algorithm will stop when the error of the parallel perceptron is below the desired accuracy ε. This happens, when for any data point in the training set all neurons will have a suitable dot product ***wp*** relative to the threshold ***TH***. The problem with this approach is that some of the neurons could have a dot product ***wp*** that is very close to the threshold. In this case, any noise affecting the data can flip one of the neurons thus causing an undesired change at the output of the squashing function. More details on the necessity of a high noise margin can be found in [4].

The original algorithm presents a solution to this problem by inserting a mechanism that produces a reasonable amount of margin between the thresholds and the ***wp*** products of all neurons for any data point. This is done by adding another term in the learning process as presented in formula (10).

$$w_k \Leftarrow (1-mlr) * w_k + mlr * \\ * \begin{cases} -p, -M < Wp - TH < 0 \\ +p, 0 < Wp - TH < M \end{cases} \text{ and } \quad \|Y(p) - t\| < \varepsilon \tag{10}$$

This applies only when the output is within the desired accuracy but the dot product ***wp*** is closer to the threshold ***TH*** than a specified margin ***M***. In this case the weight vector ***W*** is moved towards or away from the data with the margin learning rate ***mlr*** such that the margin is increased.

The only problem with this idea is the constant learning rate ***mlr*** and the constant margin ***M***. Choosing a margin beforehand can be tricky because the maximum obtainable margin is dependent on the distribution of the data within the input space. Also, a learning rate that is too big can lead to instability and also to the inability to reach the maximum margin even though this margin might has been guessed or computed beforehand. As a solution to this problem our approach introduces an adaptable learning rate ***mlr*** and an adaptable margin level ***M***.

The margin level ***M*** is recomputed at each iteration as being the P% percentile of all margin levels for all neurons. Values for P between 5% and 20% have proven to work very well. This approach guarantees that the margin constraint ***M*** is not higher than what the p-perceptron can obtain considering the given data. It also assures that the algorithm adapts and increases the constraint ***M*** once the average margin increases. This leads the algorithm towards obtaining the highest possible margin even though the margin is not known beforehand.

Several attempts were made until an appropriate control rule for adapting the learning rate ***mlr*** was found. The first attempt was to increase the learning rate whenever the derivative of the average margin is high. This meant that the current average margin is still significantly small compared to the maximum margin so faster changes can be done. Whenever the derivative of the average margin is small or negative the learning rate ***mlr*** is decreased because the maximum margin is close or already reached. The problem with this method is that predicting the approach to the maximum margin by monitoring changes in the average margin can lead to a late prediction. If the learning rate is very high at this point the algorithm can become temporally unstable and loose whatever progress accumulated.

The second attempt tried to fix this problem by setting positive and negative boundaries for the learning rate. This assured that the algorithm will not get out of control. Unfortunately the boundaries were also data dependent and could only be set experimentally.

The third approach was more successful. At each step of the algorithm the learning rate ***mlr*** is modified with formula (11). K is a percentage with range -1 to 1 given by formula (12), where Δm is the changes of the average margin during the last training epoch.

$$mlr \Leftarrow (1+K) * mlr \tag{11}$$

$$K = A * \Delta m + B \tag{12}$$

A normal learning regime is one where the margin increase Δm is equal to an estimated increase Δm_{est}. In this case ***mlr*** should be constant hence K should be zero. Δm_{est} is computed with formula (13). In practice, Δm will not equal Δm_{est} but will randomly move inside a small interval around it. This will create small opposite changes in ***mlr*** that will average down to zero.

$$\Delta m_{est} = -\frac{B}{A} \tag{13}$$

Whenever Δm is constantly larger than Δm_{est} it is considered that the learning process allows a faster increase of the margin. Because in this case K is constantly positive the learning rate ***mlr*** will increase. In order to avoid an excessive increase of the learning rate the algorithm enters an adaptive regime where the estimated value Δm_{est} is reevaluated with formula (14).

$$B = B + \xi * \left(\Delta m_{est} - \Delta m + \frac{dB}{dt} \right) \tag{14}$$

This moves Δm_{est} towards the new average value of Δm. This is graphically illustrated in figure 5.

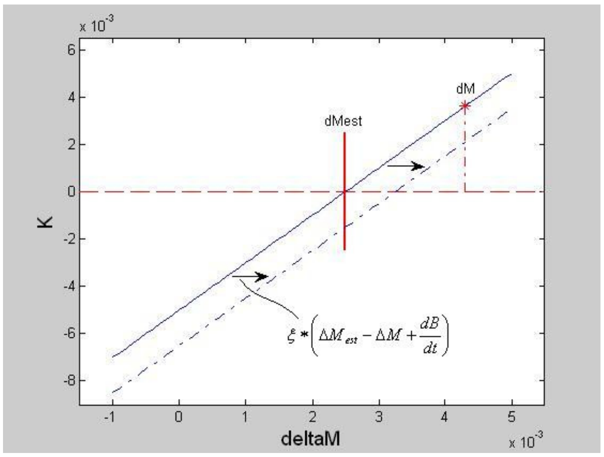

Fig. 5. K Control Rule

Whenever Δm is constantly smaller than Δm_{est} the algorithm adapts in the opposite direction. Figures 6 and 7 illustrate the values of the average margin and of ***mlr*** respectively over the epochs. Please note that the margin enhancement mechanism is inhibited until the mean square error of the parallel perceptron reaches the desired accuracy level. This can be seen in the fact that until epoch 70 the average margin changes randomly as a result of the error minimizing learning.

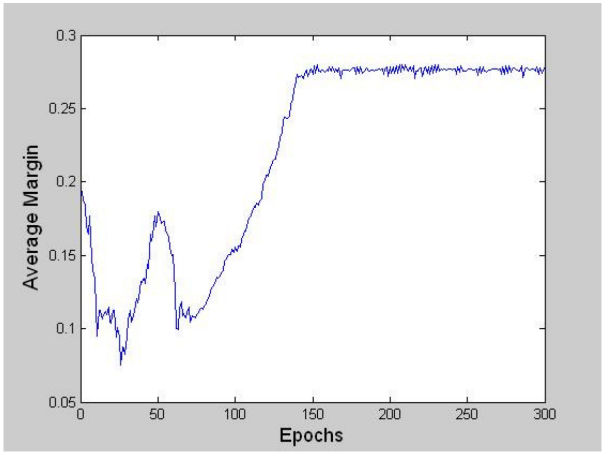

Fig. 6. Average margin during training

It is seen that as the margin approaches its maximum value the learning rate ***mlr*** decreases. The margin reaches its maximum value some time before epoch 150. It is seen that until this point the learning rate ***mlr*** is sufficiently small such that any additional changes do not make the learning process unstable or loose any of the gained progress.

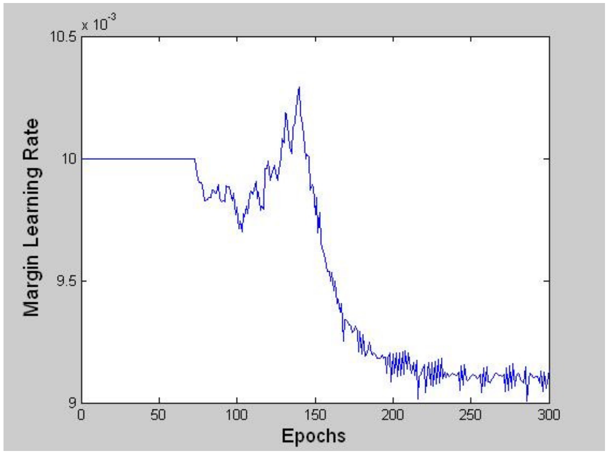

Fig. 7. Margin learning rate ***mlr*** during training

The value for A was experimentally set to 2 and it reflects the sensitivity of K over Δm. The initial value for the learning rate ***mlr*** was also set experimentally. Anyway, it is preferred to have a very small value for ***mlr*** in the beginning in order to avoid instability. The algorithm will quickly adapt ***mlr*** to a proper level.

7. Conclusion and Future Work

The new version of the algorithm offers higher speed, stability and noise margin at the expense of complexity.

The future of the project is towards designing a liquid state machine that is able to analyze and characterize the movement of a person. The input to the liquid state machine is a set of trajectories collected from markers displaced on the human body. The algorithm will be used at training the readout units of the liquid state machine. Further on, if the application proves to be successful, an attempt will be done to export the model on a platform that is PC independent. One possible option is a GPU.

8. Acknowledgement

This work was partially supported by the strategic grant POSDRU 6/1.5/S/13, (2008) of the Ministry of Labour, Family and Social Protection, Romania, co-financed by the European Social Fund – Investing in People.

This work was supported in part by the following grants: UEFISCSU 599/19.01.2009 and ANCS 222/15.04.2009.

References:

[1] Peter Auer, Harald M. Burgsteiner, Wolfgang Maass. The p-Delta Learning Rule for Parallel Perceptron, 2002.

[2] Wolgang Maass, Thomas Natschlager, Henry Markram. Real-Time Computing without stable states: A new Framework for Neural Computation based on Perturbation, 2001.

[3] Wulfram Gerstner, Coding Properties of Spiking Neurons: reverse and cross-correlations. *Neural Networks*, Vol.14, Lausanne, 2001, pp. 599-610.

[4] Freund Y., Schapire R. E. Large margin classification using the Perceptron algorithm. *Machine Learning*, 37(3), 1999, pp. 277-296.

[5] Wolfgang Maass. Neural Computation with Winner-Take-All as the only Nonlinear Operation, *Advances in Neural Information Processing Systems*, vol. 12, MIT Press, Cambridge, 2000, pp. 293-299.