Neural Network Learning Using The Robbins-Monro Algorithm

Donald L. Prados
Department of Electrical Engineering
University of New Orleans
New Orleans, LA 70148

Abstract

The purpose of this paper is to compare various learning algorithms for autoassociative Hopfield neural networks. Such neural networks store a set of patterns by making the patterns stable states of the network. All of the algorithms studied (except one) are basically gradient descent algorithms. They were compared by measuring how well they stored sets of randomly-generated patterns. There are certain trade-offs in choosing a learning algorithm for autoassociative Hopfield neural networks. If speed of convergence is of utmost importance, one may wish to use the perceptron algorithm. It is very fast, and its performance is not much worse than the slower algorithms. If high performance is essential, the modified LMSE algorithm, presented in this paper, can provide superior performance without too much of a decrease in speed of convergence. Because this algorithm minimizes the square error, its performance should be very difficult to beat. Also, like the perceptron algorithm, it has very high capacity.

Introduction

This paper compares various learning algorithms for autoassociative Hopfield neural networks. Such neural networks store a set of patterns by making the patterns stable states of the network. Since a neuron is not allowed to have input directly from itself, it must be able to determine its output by looking only at the other neurons. The neuron must actually separate the set of training patterns into two groups: those that have an output of 1 for that neuron and those that have an output of -1 (or 0) for that neuron. Since the neuron calculates a weighted sum of its inputs and then thresholds the result, the decision surface associated with the neuron is a hyperplane in state space. This hyperplane can be determined from the row of the weight matrix associated with the neuron. A neural-network training algorithm trains a net of N neurons by searching for N hyperplanes each of which separates the set of patterns into two groups. Also note that, since the neurons can be trained independently, they can be trained in parallel.

All of the algorithms studied (except one) are basically gradient descent algorithms. Such algorithms seek a minimum of a chosen energy (or error) function such that the minimum either successfully stores all patterns or attempts to minimize the probability of error. The direction of search is always down the energy landscape.

The algorithms were compared by measuring how well they stored sets of randomly-generated patterns. Each bit of each pattern generated was given an equal probability of being +1 or -1. After training the network for a set of patterns, the performance of the algorithm was tested by determining the next stable state for each possible input pattern and checking if that state was the closest in Hamming distance to the input pattern. The performance measurement is the percentage of input patterns that lead to the closest stable state.

Historical Algorithms

All algorithms to be compared assume that the output of a neuron i is calculated as follows:

$$x_i = \begin{cases} 1 \text{ if } \sum_{j=1}^{N} w_{ij}\, x_j > 0 \\ -1 \text{ otherwise} \end{cases} \qquad (1)$$

The first, and simplest, algorithm tested is based upon the Hebbian rule[1]. The weight matrix, $\mathbf{W}$, is calculated as the sum of the outer products of the m training patterns, $\mathbf{x}^s$, s = 1,..,m, (with the exception that all diagonal elements are set to 0):

$$w_{ij} = \sum_{s=1}^{m} x_i^s x_j^s \qquad (2)$$

where x_i^s is bit i of pattern s. This algorithm was chosen both because of its popularity and because the calculation can be made very quickly. This is the algorithm used by Hopfield in his famous 1982 paper[2]. It is the only algorithm tested not based upon gradient descent.

The second algorithm tested is known as the perceptron algorithm[3]. Unlike the Hebbian algorithm, this algorithm is an iterative algorithm that continually modifies the weight matrix until either all patterns are successfully stored or a limit placed upon the number of iterations is reached. Theoretically, if the set of patterns is linearly separable, this algorithm will eventually find a weight matrix that will successfully store the set of patterns. On the other hand, if the patterns are not linearly separable, the algorithm will not converge--it will enter a cycle.

The perceptron algorithm is a gradient descent algorithm. The energy of a state, $\mathbf{x}^s$, can be written as

$$J^s(\mathbf{W},\mathbf{x}) = -\sum_i \sum_j w_{ij}\, x_i^s\, x_j^s. \qquad (3)$$

This energy function, attributed to Hopfield[2], will be a local minimum whenever the network is in a stable state. The contribution of neuron i to the energy of a state is

$$J_i^s(\mathbf{w},\mathbf{x}) = -x_i^s \sum_j w_{ij}\, x_j^s. \qquad (4)$$

This component of the energy function will be negative if the next state of x_i is the same as the desired state, x_i^s.

Gradient descent algorithms increment weights in the direction of the negative gradient of the energy function:

$$w_{ij}(k+1) = w_{ij}(k) - c\,(\partial J/\partial w_{ij}) \qquad (5)$$

where k refers to the iteration. Applying Equation 5 to Equation 4 gives

$$w_{ij}(k+1) = w_{ij}(k) + c\, x_i^s\, x_j^s. \qquad (6)$$

A neural network can be trained using the perceptron algorithm as follows. Since the output of neuron i depends only upon row i of the weight matrix and the input pattern, each row of the matrix can be found independently. For each pattern, calculate the output of neuron i and compare it to the desired output of neuron i. If they are the same, no change in row i of **W** is made. If they differ, row i is changed according to Equation 6. Incorporating this condition into Equation 6 gives

$$w_{ij}(k+1) = w_{ij}(k) + c\,(x_i^s - x_i)\, x_j^s. \qquad (7)$$

where x_i^s is the desired output of neuron i when pattern s is input to the net and x_i is the actually output calculated. If one writes row i of the weight matrix **W** as the weight vector $\mathbf{w}_i$, Equation 7 becomes

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + c\,(x_i^s - x_i)\, \mathbf{x}^s. \qquad (8)$$

The algorithm cycles through the set of patterns until either neuron i successfully separates the set of patterns or the maximum number of iterations is reached. This procedure is, of course, applied to each neuron to obtain the entire weight matrix **W**.

The main advantage of this method is that, if the patterns are linearly separable, it is very fast. This is partly do to the fact that the algorithm ends as soon as a weight matrix is found that successfully stores the set of patterns. Its speed is also due to the fact that the weights are either incremented or decremented by 1 depending on the signs of the desired and actual outputs.

Two disadvantages of this algorithm are 1) that it will not converge if the patterns are not linearly separable and 2) that it makes no attempt to find an optimal weight matrix--it simply tries to find a weight matrix that will successfully store the set of patterns. Both of these problems are resolved by the next two algorithms.

Stochastic Approximation Methods

As mentioned above, a neural-network training algorithm searches for N hyperplanes, each hyperplane associated with both a neuron and a row of the weight matrix. The goal of the training algorithm can thus be thought of as finding suitable decision functions. Since the Bayes decision functions minimize the average cost of misclassification as well as yielding the lowest probability of error[4], they are a logical candidate. The Bayes decision functions for m classes are

$$d_i(\mathbf{x}) = p(\mathbf{x}/\mathbf{x}\epsilon c_i)\; p(\mathbf{x}\epsilon c_i),\quad i = 1,..,m \qquad (9)$$

where $p(\mathbf{x}\epsilon c_i)$ is the probability that pattern $\mathbf{x}$ belongs to class c_i. A pattern classifier will place a pattern $\mathbf{x}$ into class c_i if $d_i(\mathbf{x}) > d_j(\mathbf{x})$ for all j not equal to $\mathbf{x}$. Applying Bayes rule to Equation 9 gives

$$d_i(\mathbf{x}) = p(\mathbf{x}\epsilon c_i/\mathbf{x})\; p(\mathbf{x}),\quad i = 1,..,m \qquad (10)$$

Since the term $p(\mathbf{x})$ is independent of i, it may be dropped from the equation yielding,

$$d_i(\mathbf{x}) = p(\mathbf{x}\epsilon c_i/\mathbf{x}) \qquad i = 1,..,m \qquad (11)$$

Since a neuron has only two possible outputs (m = 2), there are two decision functions for each of the N neurons. For the time being, let us assume an output of 1 for class c_1 and an output of 0 for class c_2. One can combine these two decision functions into one to get a decision function for neuron i:

$$\begin{aligned} d_i(\mathbf{x}) &= d_1(\mathbf{x}) - d_2(\mathbf{x}), \qquad i = 1,..,N \qquad (12)\\ &= p(x_i=1/\mathbf{x}) - p(x_i=0/\mathbf{x})\\ &= p(x_i=1/\mathbf{x}) - [1 - p(x_i=0/\mathbf{x})]\\ &= 2p(x_i=1/\mathbf{x}) - 1 \end{aligned}$$

This may be expressed in the form: if $p(x_i=1/\mathbf{x}) > 1/2$, then, assign $\mathbf{x}$ to class 1 (set $x_i = 1$); if $p(x_i=1/\mathbf{x}) < 1/2$, then, assign $\mathbf{x}$ to class 2 (set $x_i = 0$).

The key problem is the estimation of $p(x_i=1/\mathbf{x})$. The only information that is available during training is the desired output of each neuron for each pattern vector. For each neuron, let us define a *random classification variable*, $r_i(\mathbf{x}^s)$, with the following property:

$$r_i(\mathbf{x}^s) = \begin{cases} 1 \text{ if } x_i^s = 1 \\ 0 \text{ otherwise} \end{cases} \qquad (13)$$

Since we desire knowledge of $p(x_i=1/\mathbf{x})$ only for classification purposes, let us interpret $r_i(\mathbf{x})$ as being a *noisy* observation of $p(x_i=1/\mathbf{x})$:

$$r_i(\mathbf{x}) = p(x_i=1/\mathbf{x}) + n \qquad (14)$$

where n is a noise factor which is assumed to have zero expected value. We now seek an approximation to $p(x_i=1/\mathbf{x})$ of the form $\mathbf{w}_i^T\mathbf{x}$ by observing values of $r_i(\mathbf{x})$. Consider the energy function

$$J(\mathbf{w}_i,\mathbf{x}) = E\{\ |r_i(\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}|\ \} \qquad (15)$$

where $E\{.\}$ is the expected value operator. The minimum of this function occurs when $\mathbf{x}$ is correctly classified. Assuming that $E\{r_i(\mathbf{x})\} = E\{p(x_i=1/\mathbf{x})\}$, J can be expressed as

$$J(\mathbf{w}_i,\mathbf{x}) = E\{\ |p(x_i=1/\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}|\ \}. \qquad (16)$$

According to this equation, finding the minimum of J corresponds to finding an average approximation to $p(x_i=1/\mathbf{x})$.

In order to find a minimum of a $J(\mathbf{w},\mathbf{x})$, we find the root of its derivative. The partial derivative of $J(\mathbf{w},\mathbf{x})$ with respect to $\mathbf{w}_i$ is calculated as

$$\partial J(\mathbf{w}_i,\mathbf{x})/\partial\mathbf{w}_i = E\{\ \partial|r_i(\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}|/\partial\mathbf{w}_i\ \}. \qquad (17)$$

The root of $\partial J(\mathbf{w}_i,\mathbf{x})/\partial\mathbf{w}_i$ can now be successfully estimated by invoking the Robbins-Monro algorithm[5].

The Robbins-Monro algorithm is used to find the root of a function g(w) given only noisy values of the function, indicated by h(w). Several assumptions about the data must hold for the Robbins-Monro algorithm to be successful. First, it is assumed that the noise has zero mean:

$$E\{h(w)\} = g(w). \qquad (18)$$

The second assumption is that the variance of h(w) from g(w) is finite for all values of w:

$$\sigma^2(w) = E\{[g(w) - h(w)]^2\} < L \qquad (19)$$

for all w, where L is a finite positive constant. If these two assumptions are valid, the Robbins-Monro algorithm can be used to iteratively seek the root of g(w). The estimate of the root of w is updated at the kth iterative step according to the relation

$$w(k+1) = w(k) - \alpha_k h[w(k)] \qquad (20)$$

where α_k is a member of a sequence of positive numbers which satisfy the following conditions

$$\lim_{k\to\infty} \alpha_k = 0 \qquad (21a)$$

$$\sum_{k=1}^{\infty} \alpha_k = \infty \qquad (21b)$$

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty \qquad (21c)$$

An example of such a sequence is the harmonic sequence $\{1/k\} = \{1, 1/2, 1/3, \ldots\}$. In order to prevent large overcorrections it is assumed that g(w) is bounded by a straight line on either side of the root. This assumption is valid as long as the root lies in some finite interval. If this condition is satisfied, along with the conditions of Equations 18, 19 and 21, Robbins and Monro[5] have shown that the algorithm of Equation 20 converges to the root, w^*, of g(w) in the mean-square sense:

$$\lim_{k\to\infty} \{E[(w(k) - w^*)^2]\} = 0. \qquad (22)$$

One can estimate the root of $\partial J(\mathbf{w}_i,\mathbf{x})/\partial\mathbf{w}_i$ using the Robbins-Monro algorithm as follows. Let

$$h[\mathbf{w}_i(k)] = \partial|r_i(\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}|/\partial\mathbf{w}_i. \qquad (23)$$

Using Equation 20, we obtain

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) - \alpha_k\partial|r_i(\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}|/\partial\mathbf{w}_i \qquad (24)$$

where $\mathbf{w}_i(1)$ may be chosen arbitrarily. The partial derivative is $-\mathbf{x}\ \mathrm{sgn}(\ r_i(\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}\ )$, where the function sgn(.) simply takes the sign of its argument. Substituting this into Equation 24, we obtain what has been referred to as the increment-correction algorithm[4]:

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + \alpha_k\ \mathbf{x}\,\mathrm{sgn}\{r_i(\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}\} \qquad (25)$$

The increment-correction algorithm seeks an approximation to $p(x_i=1/\mathbf{x})$ in absolute value. A least-mean-square-error (LMSE) algorithm[4] can be derived by using the energy function

$$J(\mathbf{w}_i,\mathbf{x}) = 1/2\ E\{\ [r_i(\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}]^2\ \}. \qquad (26)$$

Taking the partial derivative of J with respect to $\mathbf{w}_i$ yields

$$\partial J(\mathbf{w}_i,\mathbf{x})/\partial\mathbf{w}_i = E\{\ -\mathbf{x}[r_i(\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}]\ \} \qquad (27)$$

and the training algorithm

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + \alpha_k\ \mathbf{x}\ (r_i(\mathbf{x}) - \mathbf{w}_i^T\mathbf{x}). \qquad (28)$$

Note that both the increment-correction algorithm and the LMSE algorithm make a correction on $\mathbf{w}_i$ at every iterative step. Note also that these two statistical algorithms will converge to a solution regardless of whether the classes are strictly separable, while the perceptron algorithm oscillates in nonseparable situations. The price paid for the guaranteed convergence is the slowness with which these algorithms achieve this convergence.

The reason that these algorithms do not oscillate is that α_k is decreased at every iteration leading to smaller and smaller changes in the weights. The algorithm can be terminated as soon as the magnitudes of these weight changes fall below a chosen accuracy.

Two modifications to the Robbins-Monro algorithm that address the slowness with which it achieves convergence are presented in this paper. The first involves keeping α_k constant during steps in which h[w(k)] has the same sign[4]. This method is based on the fact that changes in the sign of h[w(k)] tend to occur more often in the vicinity of the root. For points far away from the root large corrections are desired, whereas these corrections should be smaller and smaller as the root is approached. Since Equations 25 and 28 are iteratively applied to each training pattern, determining when to change α_k is not trivial. Experimental results show that maintaining a separate α_k^s for each pattern s and updating α_k^s only when $h[\mathbf{w}_i(k)]$ changes sign, significantly speeds up convergence without decreasing the performance of the algorithms. As the algorithm iterates through the set of patterns, the sign of $h[\mathbf{w}_i(k)]$ for each pattern is saved. If, for a particular pattern, the sign of $h[\mathbf{w}_i(k)]$ changes from the last time that that pattern was applied, α_k^s is decreased.

Such a modification, however, introduces an additional problem. The algorithm may enter a cycle during which none of the α_k^s are updated. To address this problem, the algorithm was divided into two phases. During the first phase, α_k^s was conditionally decreased as discussed above. If the weight changes did not fall below the chosen accuracy within a chosen number of iterations, then α_k^s was decreased every iteration until the chosen accuracy was reached. Such a method not only produces faster convergence, but also guarantees convergence within a reasonable time frame. This algorithm will be referred to as the modified LMSE algorithm.

Results

Tests were run to compare the performances and capacities of the Hebbian, perceptron, increment-correction, and modified LMSE algorithms. The capacities are given in terms of the probability of successfully storing a set of m randomly-generated patterns of length N = 11. The performances are given in terms of the percentage of the 2^N possible input patterns that converge to the closest stable state. A test consisted of generating m random patterns of length N = 11, each bit of each pattern having an equal probability of being +1 or -1. The two algorithms to be compared were then used to attempt to store the set of input patterns by making each of them stable states of the network. The performances of the algorithms were checked by determining, for each of the 2^N possible input patterns, whether it was a stable state (either one of the randomly-generated patterns, a complement of one of the randomly-generated patterns, or a spurious state), whether it was part of a cycle, and whether the closest stable state in Hamming distance to the pattern was the next stable state of the network when the pattern was input to the network.

As expected, the capacity of the Hebbian algorithm was very poor. Figure 1 shows that, for N = 11 neurons, one has only about a 90 percent chance of storing three arbitrary patterns. The capacity of the iterative algorithms is far superior. The performance of the Hebbian algorithm, however, was not too much worse than that of the other algorithms (see Figure 2), although it did exhibit the worst performance.

Figure 1. Capacity Comparison

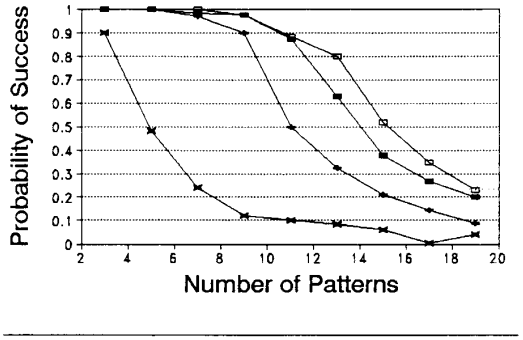
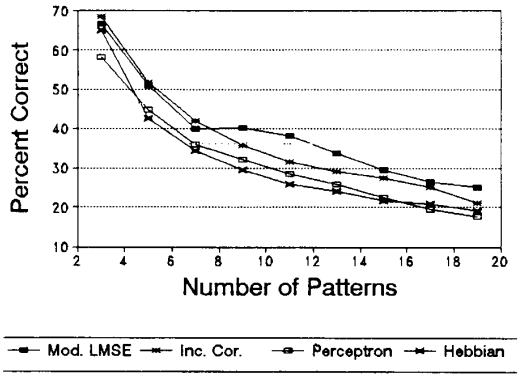


Figure 2. Performance Comparison



The perceptron algorithm exhibited the highest capacity as can be seen in Figure 1. This may be misleading since the perceptron algorithm usually converges faster than the other algorithms when the set of patterns is linearly separable. The increment-correction algorithm and the modified LMSE algorithm were more likely to reach the limit on the maximum number of iterations allowed, at which point convergence was accelerated. Thus, especially when the number of patterns was large, these two algorithms suffered more than did the perceptron algorithm from the necessity of having a limit on the number of iterations. While the capacity of the perceptron algorithm is very good, its performance, in terms of pattern classification of unknown patterns, was not as good as the slower algorithms, as can be seen in Figure 2.

The increment-correction algorithm displayed a significantly poorer ability to successfully store sets of patterns than did the modified LMSE and perceptron algorithms (Figure 1). Again, time constraints seem to have been a problem for this very slow algorithm. Although it demonstrated superior performance to the Hebbian and perceptron algorithms, it was somewhat inferior to the modified LMSE algorithm.

The modified LMSE algorithm showed the best performance in terms of pattern recognition. Its capacity was second to the perceptron by a small margin, but the difference may have been due mainly to the limit on the maximum number of iterations. Although not as fast as the perceptron algorithm, the modified LMSE algorithm is significantly faster than both the standard LMSE algorithm, in which α_k is decreased every iteration, and the increment-correction algorithm. Figures 3 and 4 show a comparison made between the modified LMSE algorithm and the standard LMSE algorithm. The standard LMSE algorithm's low probability of storing sets of patterns (see Figure 3) is partly due to the slowness with which it converges.

Figure 3. Modified LMSE vs. LMSE

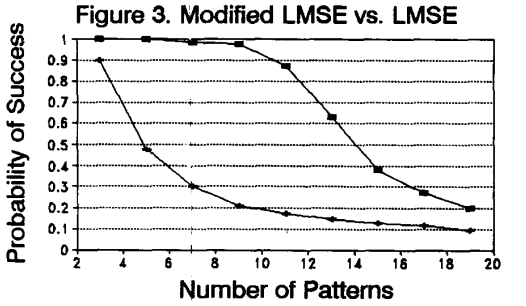
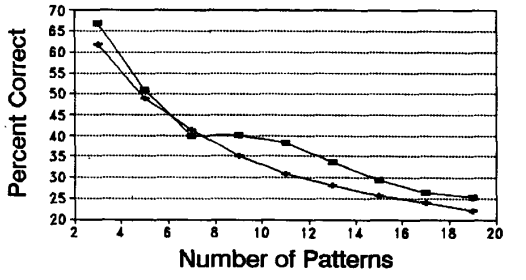


Figure 4. Modified LMSE vs. LMSE



Discussion

There are certain trade-offs in choosing a learning algorithm for autoassociative Hopfield neural networks. If speed of convergence is of utmost importance, one may wish to use the perceptron algorithm. For N larger than about 20, this algorithm has nearly a 100 percent chance of successfully storing an arbitrary set of N or less patterns[6]. In addition, its performance is not much worse than the slower algorithms discussed. Its biggest drawback is that it will not converge if the set of patterns is not linearly separable.

If high performance is essential, the modified LMSE algorithm can provide superior performance without too much of a decrease in the speed of convergence. Because this algorithm minimizes the square error, its performance should be very difficult to beat. Also, like the perceptron algorithm, it has very high capacity. For N larger than about 20, it should have little difficulty storing sets of N or less patterns. If the number of patterns is much larger than N, it still should have little difficulty provided the set of patterns is linearly separable.

Since the tests performed required calculating the next stable state of each of the 2^N possible input patterns, N had to be kept quite small. Obviously, if N is increased by 1, the time required for testing an algorithm is at least doubled. Time considerations also required setting a limit on the maximum number of iterations. For all tests the limit on the number of iterations was set to N^2m^2. For the algorithms based on the Robbins-Monro algorithm, convergence was excellerated when this limited was reached by decreasing α_k more quickly. This leads to the changes in the weights reaching the chosen accuracy more quickly. Further tests are under way using larger N and higher limits on the number of iterations. It is unlikely, however, that the relative performances of these algorithms will differ significantly in these new tests.

References

[1] D.O. Hebb, The Organization of Behavior, New York: Wiley, 1949.

[2] J.J. Hopfield, "Neural networks and physical systems with emergent collective computational ability," Proc. National Academy of Science, USA, vol. 79, pp. 2554-2558, 1982.

[3] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," Psychological Review, vol. 65, pp. 386-408, 1958.

[4] J.T. Tou and R.C. Gonzalez, Pattern Recognition Principles, Reading, MA: Addison-Wesley, 1974.

[5] H. Robbins and S. Monro, "A stochastic approximation method," Ann. Math. Stat, vol. 22, pp. 400-407, 1951.

[6] D. Prados, "The capacity of artificial neural networks using the delta rule," Ph.D. dissertation, Louisiana State University, August, 1989.

Dr. Prados received an M.S. in Biomedical Engineering from Tulane University in 1982 and a Ph.D. in Electrical Engineering from Louisiana State University in 1989. He is currently an assistant professor in Electrical Engineering at the University of New Orleans. His main areas of research are neural networks and pattern recognition.