AL-Qadisiya Journal for science Vol. 15 No. 2 year 2010

On Training Feed Forward Neural Networks For Approximation Problems

Luma. N. M.Tawfiq & Alaa. K. J. AL-Mosawi
Department of Mathematics, College of Education-Ibn Al-Haitham,University of Baghdad

Abstract:

In this paper, many modified and new algorithms have been proposed for training feed-forward neural networks; many of them having a very fast convergence rate for reasonable size networks.

We examine the similarities and differences between different training methods and compare the performance of training with each representation applied to the approximation problem.

In all of these algorithms we use the gradient of the performance function (energy function, error function) to determine how to adjust the weights such that the performance function is minimized, where the back propagation algorithm has been used to increase the speed of training. The above algorithms have a variety of different computation and thus different type of form of search direction and storage requirements; however none of the above algorithms has a global properties which suited to all problems.

1. Introduction

Back propagation (BP) process can train multilayer Feed Forward Neural Networks (FFNN's). With differentiable transfer functions, to perform a function approximation to continuous function $f \in R^N$, pattern association and pattern classification. The term of back propagation to the process by which derivatives of network error with respect to network weights and biases, can be computed. This process can be used with a number of different optimization strategies.

2. Training Algorithms for Neural Networks

Any non-linear optimization method, a local or global one, can be applied to the optimization of feed-forward neural networks weights. Naturally, local searches are fundamentally limited to local solutions, while global ones attempt to avoid this limitation. The training performance varies depending on the objective function (energy function or error function) and underlying error surface for a given problem and network configuration.

Since the *gradient* information of error surface is available for the most widely applied network configurations, the most popular optimization methods have been variants of *gradient* based back-propagation algorithms. Of course, this is sometimes the result of an inseparable combination of network configuration and training algorithm which limits the freedom to choose the optimization method.
Widely applied methods are, for example, modified back-propagation [1], back propagation using the conjugate-gradient approach [2], scaled conjugate-gradient and its stochastic counterpart [3], the Marquadt algorithm [4], and a concept learning

based back-propagation [5]. Many of these *gradient* based methods are studied and discussed even for large networks in [6]. Several methods have been proposed for network configurations where the *gradient* information is not available, such as simulated annealing for networks with non-differentiable transfer functions [7].

In many studies only small network configurations are considered in training experiments. Many *gradient* based methods and especially the Levenberg-Marquadt method are extremely fast for small networks (few hundreds of parameters), thus, leaving no room or motivation for discussion of using evolutionary approaches in the cases where the required *gradient* information is available. The problem of local minima can be efficiently avoided for small networks by using repeated trainings and randomly initialized weight values. Nevertheless, evolutionary based global optimization algorithms may be useful for validation of an optimal solution achieved by a *gradient* based method.

For large FFNNs, consisting of thousands of neurons, the most efficient training methods (Levenberg-Marquadt, Quasi –Newton, etc.) demand an unreasonable amount of computation due to their computational complexity in time and space. One possibility could be a hybrid of traditional optimization methods and evolutionary algorithms as studied in [8]. Unfortunately, it seems that none of the contemporary methods can offer superior performance over all other methods on all problem domains. It seems that no single solution appears to be available for the training of artificial neural networks.

Now, we introduce training rules (algorithms) for FFNN:

2.1.*Gradient* (Steepest) Descent (*taringd*)

A standard back propagation algorithm is a *gradient* descent algorithm (as in the Widrow-Hoff learning rule) .For the basic steepest (*gradient*) descent algorithm, the weights and biases are moved in the direction of the negative gradient of the performance function.

For the method of gradient descent, the weight update is given by :

$$W_{k+1}=W_k + \alpha_k(-g_k) \quad \text{.............................}(1)$$

where α_k is a parameter governing the speed of learning, named *learning rate*, controlling the distance between W_{k+1} and W_k and g_k is the *gradient* of the error surface at W_k, W_k is the weight at iteration k .[9],[10]

The convergence condition is satisfied by choosing: $0 < \alpha_k < \dfrac{1}{2\lambda_{max.}}$ where $\lambda_{max.}$ is the largest *eigenvalue* of weight matrix.

2.2.*Gradient* Descent With Momentum (*traingdm*) [11]

There is another training algorithm for FFNN that often provides faster convergence. The weight update formula for gradient descent with momentum is given by: $W_{k+1}=W_k + \alpha_k(-g_k) + \mu(W_k - W_{k-1})$

That is: $W_{k+1}=W_k + \alpha_k(-g_k) + \mu\Delta W_k$

i.e. $\Delta W_{k+1}=\alpha_k(-g_k) + \mu\Delta W_k \quad \text{.......................}(2)$

Where the momentum parameter μ is constrained to be in the range (0 , 1). Momentum allows the ANN to make reasonably large weight adjustments, while using a smaller learning rate to prevent a large response to the error from any one of training pattern.

The *gradient* is constant (g_k = const). Then, by applying iteratively (2) :

$$\Delta W = - \alpha g_k (1+ \mu + \mu^2 + \ldots) \longrightarrow - \frac{\alpha}{1-\mu} g_k$$

(because $\mu \in (0,1)$ and then $\lim\limits_{n\to\infty} \mu^n = 0$), i.e. the learning rate effectively increases from α to $\dfrac{\alpha}{(1-\mu)}$.

Remark

There are several issues on *gradient descent* training algorithms:

1. When the learning rate α is too small, the learning algorithm converges very slowly. However, when α is too large, the algorithm becomes unstable and diverges.
2. Another peculiarity of the error surface that impacts the performance of the *gradient descent* training algorithm is the presence of local minima [12]. It is undesirable that the learning algorithm stops at a local minimum if it is located far above a global minimum.
3. Neural network may be over-trained by using *gradient descent* algorithms and obtain worse generalization performance. Thus, validation and suitable stopping methods are required in the cost function minimization procedure.
4. *Gradient*-based training is very time-consuming in most applications.

The aim of this paper is to solve the above issues related with *gradient*-based algorithms and propose an efficient training algorithm for FFNNs

3. Faster Trining

In this section, we will discuss several high performance algorithms fall into two main categories. The first category uses heuristic techniques, which were developed from an analysis of the performance of the standard *gradient* descent algorithm. Another heuristic modification is the momentum technique, variable learning rate and resilient back propagation. The second category of fast algorithms uses standard numerical optimization techniques such as: conjugate gradient, quasi-Newton and Levenberg-Marquardt .

3.1.Variable Learning Rate

With standard *gradient* descent, the learning rate is held constant through out training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm becomes unstable. If the learning rate is too small, the algorithm will take too long to converge. Our numerical results and [13] shows that it is not practical to determine the optimal setting for the learning rate before training and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

D. G. Luenberger, 1991 shows that the optimal learning rate for quadratic error surface :

$$\eta_k = \frac{E_k{}^T E'_k}{\rho_k^T E''_k \rho_k}$$, where ρ_k is the search direction.

The performance of the steepest descent algorithm can be improved if we allow the learning rate to change during the training process. Back propagation training with an adaptive learning rate is implemented with the function '*traingda*'. The function '*traingdx*' combines adaptive learning rate with momentum training. [13]

3.2.Resilient Back Propagation (*trainrp*) [14]

The resilient back propagation training algorithm eliminates the harmful effect of having a small slope at the extreme ends of sigmoid transfer functions in hidden layers. Only the sign of the derivative of the transfer function is used to determine the direction of the weight update: the magnitude value of the derivative has no effect on the weight update. Our results show the resilient back propagation is generally much faster than the standard *gradient* descent algorithm. Also it has a nice property that it requires only a modest increase in memory requirements, and thus we do need to store the update values for each weight and bias.

3.3.Quasi-Newton Algorithms [16]

Quasi-Newton (or *secant*) methods are based on Newton's method but we require calculation of second derivatives (*Hessian* matrix) at each step. They update an approximate *Hessian* matrix at each iteration of the algorithm.

The optimum weight value can be computed in an iterative manner by writing:

$$W_{k+1} = W_k - \eta_k H^{-1} g_k \quad \text{……..….. (3)}$$

where η_k is the learning rate, g_k is the *gradient* of the error surface with respect to the W_k and H is the *Hessian* matrix (second derivatives of the error surface with respect to the W_k) [15]. We can show that the Quasi-Newton's method converges to the optimal weight W*. Now rewrite the equation of Newton's method as:

$$W^* = W_k - \frac{1}{2} H^{-1} g_k \quad \text{……………………………(4)}$$

Therefore, from equations (3) and (4), we get :

$$W_{k+1} = W_k - 2\eta_k(W_k - W^*) = W_k(1 - 2\eta_k) + 2\eta_k W^*$$

Starting with an initial weight of W_0 , we get :

$$W_1 = W_0(1 - 2\eta_k) + 2\eta_k W^* = W^* + (1 - 2\eta_k)(W_0 - W^*)$$

$$W_2 = W_1(1 - 2\eta_k) + 2\eta_k W^* = W_0(1 - 2\eta_k)^2 + 2\eta_k W^*(1 - 2\eta_k) + 2\eta_k W^*$$

$$= W^* + (1 - 2\eta_k)^2(W_0 - W^*)$$

$$W_k = W^* + (1 - 2\eta_k)^m(W_0 - W^*)$$

Since $W_0 - W^*$ is fixed, W_k converges to W^*, provided :

$$0 < 2\eta_k \le 1 \quad , \quad \text{i.e., } 0 < \eta_k \le ½ \quad .$$

We see that in the quasi-Newton method the steps do not proceed along the direction of the *gradient*. Now we introduce two quasi-Newton algorithms :

3.3.1.BFGS Quasi-Newton Algorithm (*trainbfg*) [13]

This algorithm requires more computation for each iteration and our results shows more storage require than the CG methods, although, generally, converges in fewer iterations. For a very large ANN it may be better to use resilient back propagation or one of the CG algorithms. For smaller ANN, however, BFGS quasi-Newton algorithm can be used as an efficient training function.

3.3.2.One Step Secant Algorithm (*trainoss*) [13]

Since the BFGS algorithm requires more storage and computation in each iteration than the CG algorithms, there is need for a *secant* approximation with smaller storage and computation requirements. The one step *secant* (OSS) method is an attempt to bridge the gap between the CG algorithms and the quasi-Newton (*secant*) algorithms .

This algorithm does not store the complete *Hessian* matrix; it assumes that at each iteration the previous *Hessian* was the identity matrix. This has the additional advantage that the new search direction can be calculated without computing a matrix inverse.

3.4.Levenberg-Marquardt Algorithm (*trainlm*) [13]

The Levenberg-Marquardt algorithm was designed to approach second order training speed without having to compute the *Hessian* matrix. When the performance function has the form of a sum of squares, then the *Hessian* matrix can be approximated as $H = J^TJ$ and the *gradient* can be computed as $g = J^Te$, where J is the *Jacobian* matrix, which contains first derivatives of the network errors with respect to the weights and biases, and *e* is a vector of network errors. The Levenberg-Marquardt algorithm uses this approximation to the *Hessian* matrix in the following Newton update: $W_{k+1} = W_k - [J^TJ + \mu I]^{-1}J^Te$

when the scalar $\mu=0$, this is just Newton's method. When μ is large, this becomes *gradient* descent with a small step size.

3.5.Conjugate Gradient Algorithms (*traincg*)

The conjugate gradient algorithms perform a search along conjugate directions, which produces generally faster convergence than *gradient* descent directions [Hagan and Beale, 1996]. The CG algorithms start out by searching in the *gradient* descent direction (negative of the *gradient*) on the first iteration, $\rho_0 = -g_0$.

Then the next search direction is determined so that it is conjugate to previous search directions, that is : [12]

$$W_{k+1} = W_k + \eta_k \rho_k \text{ . Where } \rho_k = -g_k + \beta_k \rho_{k-1}.$$

The various versions of CG are distinguished by the manner in which the β_k is computed.

In this paper, we will present different variations of CG algorithms with a comparison between them. In most of the training algorithms a learning rate is used to determine the length of the weight update (step size).

In most of the CG algorithms, the step size is adjusted at each iteration. A search is made along the CG direction to determine the step size, which will minimize the performance function along that line search. The CG algorithms that usually used in ANN as a training algorithm is much faster than variable learning rate back propagation, and are sometimes faster than Resilient back propagation, although the results will vary from one problem to another.

3.5.1.Fletcher-Reeves update (*traincgf*)

The general procedure for determining the new search direction is to combine the new *gradient* descent direction with the previous search direction :

$\rho_k = -g_k + \beta_k \rho_{k-1}$.For Fletcher-Reeves update procedure [14] : $\beta_k = \dfrac{g_k^T g_k}{g_{k-1}^T g_{k-1}}$

The training parameters for '*traincgf*' are: *epochs, show, goal, time, min-grad, srchFcn.*

The training status will be displayed every show iterations of the algorithm. The other parameters determine when the training is stopped. The training will stop when the number of iterations exceeds an *epochs*, if the performance function drops below *goal*, if the magnitude of the *gradient* is less than *mingrad* or if the training time is longer than *time* in seconds. The parameter *srchfcn* is the name of the line search function. *traincgf* generally converges in fewer iterations than Resilient back propagation (*trainrp*) (although there is more computation required in each iteration).

3.5.2.Polak-Ribiere update (*traincgp*)

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiere [16]. For the Polak-Ribiere update, the constant β_k is computed from :

$$\beta_k = \frac{\Delta g_{k-1}^T g_k}{g_{k-1}^T g_{k-1}}$$

The *traincgp* routine has performance similar to traincgf. It is difficult to predict which algorithm will perform best on a given problem. The storage requirements for Polak-Ribiere (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

3.5.3.Dixon update (*traincgd*)

We propose another version of the conjugate gradient algorithm, which derive from classical method proposed by Dixon [16].

For the Dixon update, the constant β_k is computed by: $\beta_k = \dfrac{-g_k^T g_k}{\rho_{k-1}^T g_{k-1}}$

The training parameters for *traincgd* are: *epochs,* show, *goal, time, min-grad, max-fail, srchFcn, scal-tol, alpha, beta, delta, gama, low-lim, up-lim, maxstep, minstep, bmax*.

The training status will be displayed every show iterations of the algorithm. The other parameters determine when the training is stopped. The training will stop if the number of iterations exceeds *epochs*, if the performance function drops below *goal*, if the magnitude of the *gradient* is less than *mingrad*, or if the training time is longer than *time* seconds, *max-fail* which is associated with the early stopping technique.

The parameter *srchFcn* is the name of the line search function. The remaining parameters are associated with specific line search routines. The default line search routine *srchcha* is used.

The *traincgd* routine has performance, which is some what better than *traincgp* for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Dixon algorithm (three vectors).

3.5.4.Al-Assady and Al-Bayati update (*traincga*)

We use another version of the conjugate gradient algorithm, when the classical method proposed by Al-Assady and Al-Bayati [16].

For the Al-Assady and Al-Bayati update, the constant β_k is computed by:

$$\beta_k = \frac{-g_k^T \Delta g_{k-1}}{\rho_{k-1}^T g_k}$$

The training parameters for *traincga* are: *epochs*, *show*, *goal, time*, *min-grad*, *max-fail, srchFcn*. The storage requirements for the Al-Assady and Al-Bayati algorithm (four vectors).

3.5.5.Hestenes-Stiefel update (*traincgh*)

We will consider another version of the CG algorithm, when the classical method proposed by Hestenes-Stiefel [12].

For the Hestenes-Stiefel update, the constant β_k is computed by :

$$\beta_k = \frac{g_k^T \Delta g_{k-1}}{\rho_{k-1}^T \Delta g_{k-1}}$$

The *traincgh* routine has performance similar to *traincgd*.

The storage requirements for the Hestenes-Stiefel algorithm (four vectors)

3.5.6.Reyadh-Luma update (*traincgr*)

We propose a new version of the CG algorithm when the search direction at each iteration is determined by: $\rho_k = -\square g_k + \beta_k \rho_{k-1}$

Where the constant β_k is computed by: $\beta_k = \dfrac{g_k^T \Delta g_{k-1}}{\rho_{k-1}^T g_{k-1}}$

The training parameters for *traincgr* are: *epochs*, *show*, *goal*, *time*, *min-grad*, *max-fail*, *sigma*, *lambda*.[13]

3.5.7.Line Search Routines (*SRCHCHA*)

The method of *srchcha* was designed to be used in a combination with a CG algorithm for ANN training. We have used this routine as the default search for most of the CG algorithms, since it appears to produce excellent results for many different problems. It does require the computation of the derivatives (back propagation) in addition to the computation of performance function, but it over comes this limitation by locating the minimum with fewer steps.

3.6. Error Surfaces

Generally the error may be represented as a surface E = E(W) into the $N_W + 1$ space where N_W is the total number of weights. The goal is to find the minima of error function, where g = 0; however note that this condition is not enough to find the absolute minima because it is also true for local minimums, maximums and saddle-points.

In general it is not possible to find the solution W in a closed form. Then a numerical approach is taken, to find it by searching the weights space in incremental steps (k = 1, …) of the form $W_{k+1} = W_k + \Delta W_k$. However, usually, the algorithm does not guarantee for the finding of absolute minima and even a saddle-point may stick them.

On the other hand the weight space have a high degree of symmetry and thus many local and global minimums which give the same value for the error function; then a relatively fast convergence may be achieved starting from a random point.

3.7. Initialization and Termination of Training

Usually the weights are initialized with random values to avoid problems due to weight space symmetry. However there are two restrictions:

- If the initial weights are too big then the activation functions *f* will have values into the saturation region (e.g. *sigmoidal* activation function) and their derivatives *f* ' will be small, leading to a small error *gradient* as well, i.e. an approximatively at error surface and, consequently, a slow training.

- If the initial weights are too small then the activation functions *f* will be linear and their derivatives will be quasi-constant, the second derivatives will be small and then the *Hessian* will be small meaning that around minimums the error surface will be approximatively at end, consequently, a slow training.

We suggest the method to determinate the weights by the following:

3.7.1.Determination of weights by Computation

For a linear FFNN's let actual output vector y=WX and the desired output vector is d , then the total error E(W) over all the L input/output pattern pairs is given by :

$$E(W)=\frac{1}{L}\sum_{i=1}^{L}\| d_i - Wx_i \|^2$$

We can write

$$E(W)=\frac{1}{L}\sum_{i=1}^{L}\| D - WX \|^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5)$$

Using the definition that the *trace* of a square matrix *S* is the sum of the main diagonal entries of *S*, it is easy to see that: $E(W)=\frac{1}{L}\mathrm{tr}(S)$,

where the matrix *S* is given by: *S*=(D-WX) (D-WX)^T ,and tr(*S*) is the *trace* of the matrix *S*.

Using the definition for *pseudo inverse* of a matrix, i.e. $A^+ = A^T(AA^T)^{-1}$, we get the matrix identities $A^+AA^T=A^T$ and $AA^T(A^+)^T=A$.

Using these matrix identities we get:

$$S = (D–WX)(D–WX)^T=(DX^{-1}X–WX)(DX^{-1}X–WX)^T=(DX^{-1}-W)\, XX^T\, (DX^{-1}-W)$$
$$= (W-DX^T(X^T)^{-1}X^{-1})XX^T(W-DX^T(X^T)^{-1}X^{-1})^T+DD^T- DD^T$$
$$= (W-DX^T(XX^T)^{-1})XX^T(W-DX^T(XX^T)^{-1})^T+DD^T-DX^T(XX^T)^{-1}XD^T$$
$$S = (W-DX^+)XX^T(W-DX^+)^T+D(I-X^+X)D^T \quad \ldots\ldots\ldots\ldots\ldots\ldots(6)$$

It can be seen that the *trace* of the first term in equation (6) is always nonnegative, as it is in a quadratic form of the real symmetric matrix XX^T.

It becomes zero for $W=DX^+$. The *trace* of the second term is a constant, independent of W. Since the *trace* of sum of matrices is the sum of *traces* of the individual matrices, the error E(W) is minimum when $W=DX^+$.

The minimum error is obtained by substituting $W=DX^+$ in equation (5) and is given by:

$$E_{min}=\frac{1}{L}\| D - DX^+X \|^2$$
$$=\frac{1}{L}\mathrm{tr}[(D(I-X^+X))(\, D(I-X^+X))^T] = \frac{1}{L}\mathrm{tr}[D(I-X^+X)(I-X^+X)^TD^T]$$
$$=\frac{1}{L}\mathrm{tr}[D(I-X^+X)(I-(X^+X)^T)D^T]$$
$$=\frac{1}{L}\mathrm{tr}[D(I-(X^+X)-(X^+X)^T+(X^+X)(X^+X)^T)D^T]$$

$$= \frac{1}{L}\,\mathrm{tr}[D(I-X^{+}X-X^{T}(X^{+})^{T}+X^{+}X))D^{T}] = \frac{1}{L}\,\mathrm{tr}[D(I-(X^{+}X)^{T})D^{T}]$$

$$= \frac{1}{L}\,\mathrm{tr}[D(I-X^{+}X)D^{T}] \quad \text{………………………………………………}(7)$$

where I is an L×L identity matrix. The above simplification is obtained by using the following matrix identities $(X^{+}X)^{T}=X^{T}(X^{+})^{T}$ and $XX^{T}(X^{+})^{T}=X$

Note

We use *pseudo inverse* since we can not be compute the inverse of matrix and we use Singular Value Decomposition (SVD) method to compute *pseudo inverse*.

3.7.2. Singular Value Decomposition (SVD)

The following singular value decomposition (SVD) of an m×n matrix X is used to compute the *pseudo inverse* and to evaluate the minimum error.
A singular value and corresponding singular vectors of a rectangular matrix $X \in R^{m\times n}$ are a scalar σ and a pair of vectors u and v that satisfy:

$$X\,v = \sigma\,u \quad \& \quad X^{T}u = \sigma\,v$$

With the singular values on the diagonal of a diagonal matrix S and the corresponding singular vectors forming the columns of two orthogonal matrices U and V, we have:

$$X\,V = U\,S \quad \& \quad X^{T}U = V\,S$$

Where U and V are orthogonal. The above decomposition of X is called the singular value decomposition (S V D) : $X = U\,S\,V^{T}$

The singular value decomposition of an m× n matrix, X, involves the computation of an m ×m matrix, U, an m× n matrix, S, and an n× n matrix, V.

In other wards, U and V are both square and S is the same size as X.

If X has many more rows than columns, the resulting U can be quite large, but most of its columns are multiplied by zeros in S. In this situation, the economy sized decomposion saves both time and storage by producing an m× n matrix, U, an n× n matrix, S, and the same V.

The *eigenvalue* decomposion is the appropriate tool for analyzing a matrix when it represents a mapping from a vector space in to itself. On the other hand, the SVD is the appropriate tool for analyzing a mapping from one vector space in to another vector space, possibly with a different dimension.

Most systems of simultaneous linear equations fall into the last category.

If X is square, symmetric and positive definite, then its *eigenvalue* and SVD are the same. But, as X departs from symmetry and positive definiteness, the difference between the two decompositions increases.

In particular, the SVD of a real matrix is always real, but the eigenvalue decomposion of real, non symmetric matrix might be complex.

Now, we can provide a simple explanation for the well known phenomenon reported in many practical studies with Ann's. This is the observation that better results may well be obtained if the iteration is not continued to converge.

These problems are closely related to the issue of non spanning patterns which we have already encountered. The linear least squares (L.S) problem of minimizing $||XW - Y||_2$ always has a solution. The solution is unique if and only if null(X) = 0, that is, linear least squares has a unique solution when X has linear independent columns (X^TX non-singular, even if X is singular) that is null(X) = 0 if and only if X has linearly independent columns.

Now $X \in R^{m\times n,}$ if $n > m$, then null(X) $\neq 0$.

Then we may have in this case (null(X) $\neq$ 0) near linear dependent among possibly the last columns and in this case we cant use L.S.S. because one is unsure about the rank (X) and in this case a remedy for this problem is to use a new technique, singular value decomposition (SVD) and this technique used as follow :

Split X in to USV^T, where U and V are *orthogonal* and S is diagonal (but not necessarily square). That is $UU^T = I_m$, $VV^T = I_n$. Then:

$$S = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ 0 & 0 & \cdots & \sigma_n \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

The matrices U and V consist of the *orthonormalize eigenvectors* of X^TX, XX^T respectively. σ_i are the square root, $\sigma_i = \sqrt{\lambda_i(X^TX)}$ and said singular values of X.

Now, if the rank(X) = r, then : $\sigma_{r+1} = \sigma_{r+2} = \sigma_{r+3} = \ldots = \sigma_n = 0$

Remark

Another way to improve network performance is to train multiple instances of the same network, but with a different set of initial weights, and choosing among those who give best results. This method is called committee of networks.

The criteria for stopping the training process may be one of the following:

- Stop after a fixed number of steps.
- Stop when the error function had become smaller than a specified amount.
- Stop when the change in the error function (ΔE) had become smaller than a specified amount.
- Stop when the error on an (independent) validation set begins to increase.

References:-

[1] G.-B. Huang, (4, JULY 2006), " Real -Time Learning Capability of Neural Networks ", IEEE
Transactions on Neural Networks, VOL. 17, NO.
[2] W. W. Hsieh, (August 31, 2008), "Machine Learning Methods in Environmental Sciences Neural Networks and Kernels", Cambridge University Press.
[3] G. Weir-Smith and C.A.Schwabe, (2002), "Spatial interpolation vs neural network propagation as a method of extrapolating from field surveys", GIS Centre, HSRC (Human Sciences Research Council), Pretoria.
[4] J.M. Turmon, (August 1995), "Assessing Generalization of Feed forward Neural Networks", phD. thesis, University of Cornell,.
[5] M.A.Ali, S.D.Gore and M.AL-Sarierah, (2005),"The Use of Neural Network to Recognize the parts of the Computer Motherboard", Journal of Computer Sciences 1(4), pp. 477- 481.
[6] J.ILONEN, J.-K.KAMARAINEN and J.LAMPINEN, (2003), "Differential Evolution Training Algorithm for Feed-Forward Neural Networks", Neural Processing Letters 17:pp. 93–105.
[7] S. Breutel, (2004), "Analysing the Behaviour of Neural Networks", PhD thesis, Queensland University of Technology, Brisbane.
[8] T. Su, J. Jhang and C. Hou, (September 2008), "A Hybrid Artificial Neural Networks and Particle Swarm Optimization for Function Approximation", International Journal of Innovative Computing, Information and Control ICIC International , Volume 4, Number 9, pp. 2363—2374.
[9] A.Pinkus, (1999), "Approximation theory of the MLP model in neural networks", Acta Numerica, pp.143-195.
[10] L.N.M.Tawfiq and Q.H.Eqhaar, (2007), "On Feed forward neural network with Ridge basis function", Journal Al-Qadisiya for Pure Science, Vol. 12, No.4.
[11] T.Poggio and F.Girosi, (July 1989), "A Theory of Networks for Approximation and Learning ,"Massachusetts Institute of Technology Artificial Intelligence Laboratory", A.I.Memo No.1140, C.B.I.P Paper No.31.
[12] R. M. Hristev, (1998), "The ANN Book", Edition 1.
[13] L.N.M.Tawfiq, (2004), "On Design And Training of Artificial Neural Networks For Solving Differential Equations", phD.Thesis, College of Education Ibn Al-Haitham,Bahgdad University.
[14] G.P.Jaya Prakash and TRBstaff Representative, (December1999), "Use of Artificial Neural Networks In Geomechanical And Pavement System", Transportation Research Circular, Number E-co12.
[15] N.Stanevski and D.Tsvetkov, (2004), "On the Quasi-Newton Training Method for Feed-Forward Neural Networks", International Conference on Computer System and Technologies.
[16] L.N.M.Tawfiq and R.S.Naoum, (2005), "On Training of Artificial Neural Networks", AL-Fath Jornal, No 23.

لمسائل التقريب

لمى ناجي محمد توفيق و علاء كامل جابر
قسم الرياضيات – كلية التربية –أبن الهيثم – جامعة بغداد

الخلاصة:-

في هذا البحث اقترحنا عدد من الخوارزميات المطورة والجديدة لتدريب الشبكات العصبية ذات التغذية التقدمية البعض منها تمتلك سرعة تقارب جيدة للشبكات ذات التركيب المعقول واختبرنا أوجه التشابه والاختلاف بين طرق التدريب المختلفة وقارنا الأداء للتدريب لكل تمثيل طبق على مسائل التقريب .
في كل تلك الخوارزميات استخدمنا انحدار دالة الأداء (دالة الخطأ، دالة الطاقة) لتحديد كيفية ضبط الأوزان بحيث تكون دالة الأداء أقل ما يمكن . حيث استخدمنا خوارزمية الانتشار المرتد لتسريع التدريب
جميع الخوارزميات أعلاه تتنوع من حيث اختلاف الحسابات وبالتالي اختلاف الأنواع حسب الصيغ لاتجاه التفتيش والخزن الذي تقتضيه وكل الخوارزميات أعلاه لا تمتلك خواص رئيسية تجعلها مناسبة لكل المسائل .