HIGH PERFORMANCE ALGORITHMS IN FEEDFORWARD NEURAL NETWORKS BACKPROPAGATION TRAINING

Luminița Giurgiu, Assistent Professor, Land Forces Academy "Nicolae Bălcescu", Sibiu, Revoluției 3-5, tel. 0040269432990/1413, fax 0040269215554, email: lumigee@actrus.ro

Abstract. **Neural networks, as a field, are becoming an important and useful tool for a wide variety of problem areas. As our understanding of the methodology increases through research, we become better equipped to address tasks on the scale of practical, real-world problems. Research contributions resulting from the study of biological systems, while extremely valuable in finding new research directions, place additional demands and burdens on the task of training. Thus, efficient training methods are essential for the field to progress. In this paper we will discuss several high performance algorithms used in backpropagation training of the feedforward neural networks and present comparative results obtained in MATLAB implementation.**

INTRODUCTION

The two classic backpropagation training algorithms: gradient descent and gradient descent with momentum are often too slow for practical problems. High performance algorithms can converge from ten to one hundred times faster than the classic algorithms. These faster algorithms fall into two main categories. One category uses heuristic techniques, which were developed from an analysis of the performance of the standard steepest descent algorithm, and the other category of fast algorithms uses standard numerical optimization techniques. In the first category are included the heuristic techniques of variable learning rate backpropagation and resilient backpropagation; in the second category are included the techniques of conjugate gradient, quasi-Newton algorithms and Levenberg Marquardt algorithm.

HEURISTIC TECHNIQUES

The performance of the steepest descent algorithm can be improved if we allow the learning rate to change during the training process. An adaptive learning rate will attempt to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.

An adaptive learning rate requires some changes in the training procedure

used by *traingd*. First, the initial network output and error are calculated. At each epoch new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio (max_perf_inc, typically 1.04), the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by lr_dec = 0.7). Otherwise, the new weights, etc., are kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by lr_inc = 1.05).

Backpropagation training with an adaptive learning rate is implemented with the function traingda, which is called just like *traingd*, except for the additional training parameters max_perf_inc, lr_dec, and lr_inc.

A two-layer network is called to train with this code:

```
x = [-1 -1 2 2;0 5 0 5];
y = [-1 -1 1 1];
net=newff(minmax(x),[3,1],{'tansig','purelin'},'traingda');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,x,y);
```

In the command window the result is:

TRAINGDA, Epoch 0/300, MSE 0.69466/1e-005, Gradient 2.29478/1e-006

TRAINGDA, Epoch 30/300, MSE 8.57235e-006/1e-005, Gradient 0.00369012/1e-006

TRAINGDA, Performance goal met.

The result of the simulation

res=sim(net, x) is:

```
res =
  -1.0008   -0.9997    1.0051    0.9972
```

The function *traingdx* combines adaptive learning rate with momentum training. It is invoked in the same way as *traingda*, except that it has the momentum coefficient mc as an additional training parameter.

Consider the same code with the additional parameter:

net.trainParam.mc=0.9

The training response will be:

TRAINGDX, Epoch 0/300, MSE 1.71149/1e-005, Gradient 2.6397/1e-006

TRAINGDX, Epoch 50/300, MSE 0.00191135/1e-005, Gradient 0.0627063/1e-006

TRAINGDX, Epoch 56/300, MSE 6.39208e-006/1e-005, Gradient 0.00353153/1e-006

TRAINGDX, Performance goal met.

The training response in the case of *traingd* function is for the same network:

TRAINGD, Epoch 0/300, MSE 1.21966/1e-005, Gradient 1.77008/1e-010
TRAINGD, Epoch 50/300, MSE 0.40967/1e-005, Gradient 0.370667/1e-010
TRAINGD, Epoch 100/300, MSE 0.0193925/1e-005, Gradient 0.129972/1e-010
TRAINGD, Epoch 150/300, MSE 0.00354209/1e-005, Gradient 0.0450151/1e-010
TRAINGD, Epoch 200/300, MSE 0.00091316/1e-005, Gradient 0.0217073/1e-010
TRAINGD, Epoch 250/300, MSE 0.000255977/1e-005, Gradient 0.0112875/1e-010
TRAINGD, Epoch 300/300, MSE 7.39654e-005/1e-005, Gradient 0.00602179/1e-010
TRAINGD, Maximum epoch reached, performance goal was not met.

Multilayer networks use sigmoid transfer functions in the hidden layers. Since they compress an infinite input range into a finite output range these functions are often called "squashing" functions. The slope of sigmoid functions must approach zero as the input gets large and this causes a problem when training with steepest descent: the gradient can have a very small magnitude, this cause small changes in the weights and biases and the weights and biases will be far from their optimal values.

The purpose of the resilient backpropagation (Rprop) training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives.

The magnitude of the derivative has no effect on the weight update and only the sign of the derivative is used to determine the direction of the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor delt_inc whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor delt_dec whenever the derivative with respect that weight changes sign from the previous iteration. If the derivative is zero, then the update value remains the same. Whenever the weights are oscillating the weight change will be reduced. If the weight continues to change in the same direction for several iterations, then the magnitude of the weight change will be increased.

The training code for the same network is:

```
x = [-1 -1 2 2;0 5 0 5];
y = [-1 -1 1 1];
net=newff(minmax(x),[3,1],{'tansig','purelin'},'trainrp');
net.trainParam.show = 10;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,x,y);
```

TRAINRP, Epoch 0/300, MSE 2.66524/1e-005, Gradient 4.39122/1e-006
TRAINRP, Epoch 10/300, MSE 0.00150934/1e-005, Gradient 0.110215/1e-006
TRAINRP, Epoch 14/300, MSE 9.64232e-006/1e-005, Gradient 0.00224347/1e-006
TRAINRP, Performance goal met.

NUMERICAL OPTIMIZATION TECHNIQUES

In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions. In most of the training algorithms that we discussed up to this point, a learning rate is used to determine the length (step size) of the weight update. In most of the conjugate gradient algorithms, the step size is adjusted at each iteration. To determine the step size, which minimizes the performance function along that line, a search is made along the conjugate gradient direction. There are different search functions who can be used interchangeably with a variety of the training functions. Some search functions are best suited to certain training functions, although the optimum choice can vary according to the specific application. An appropriate default search function is assigned to each training function, but this can be modified by the user.

All of the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

$$p_0=-g_0$$

A line search is then performed to determine the optimal distance to move along the current search direction:

$$x_{k+1}=x_k+\alpha_k p_k$$

The next search direction is determined so that it is conjugate to previous search directions.

$$p_k=-g_k+\beta_k g_{k-1}$$

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

The various versions of conjugate gradient are distinguished by the manner in

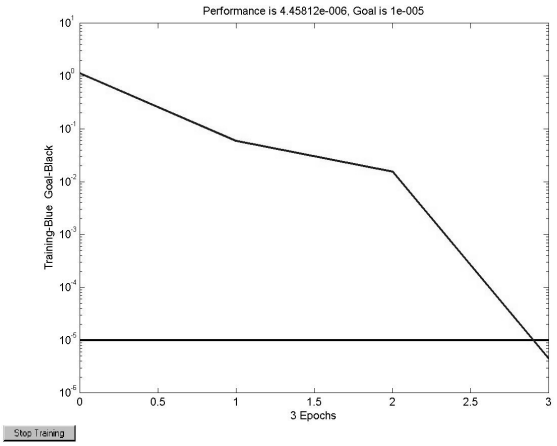
which the constant is computed. For the Fletcher-Reeves update the procedure is

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

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

The following code reinitialize the previous network and retrain it using the Fletcher-Reeves version of the conjugate gradient algorithm:

```
x = [-1 -1 2 2;0 5 0 5];
y = [-1 -1 1 1];
net=newff(minmax(x),[3,1],{'tansig','purelin'},'traincgf');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,x,y);
```



The conjugate gradient algorithms are usually much faster than variable learning rate backpropagation, and are sometimes faster than *trainrp*, although the results will vary from one problem to another.

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére and differs from the Fletcher-Reeves by the constant β_k who is computed by

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

This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient.

The previous code change only in line:

net=newff(minmax(x),[3,1],{'tansig','purelin'},'traincgp');

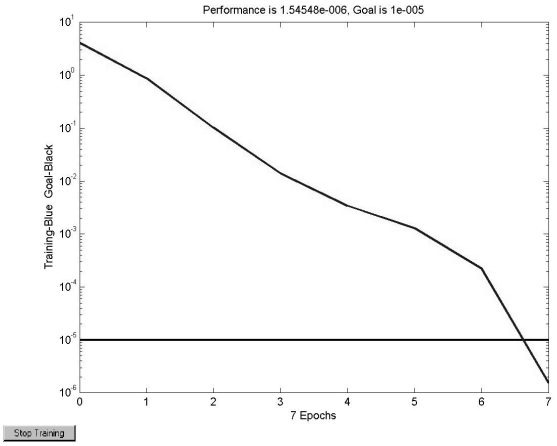
and the result is:

TRAINCGP-srchcha, Epoch 0/300, MSE 4.11391/1e-005, Gradient 4.5572/1e-006

TRAINCGP-srchcha, Epoch 5/300, MSE 0.00129061/1e-005, Gradient 0.0588631/1e-006

TRAINCGP-srchcha, Epoch 7/300, MSE 1.54548e-006/1e-005, Gradient 0.00317448/1e-006

TRAINCGP, Performance goal met.



The training parameters for *traincgp* are the same as those for *traincgf*. The default line search routine srchcha is used in this example.

The method of Charalambous srchcha was designed to be used in combination with a conjugate gradient algorithm for neural network training. It is a hybrid search and it uses a cubic interpolation, together with a type of sectioning.

The *traincgp* routine has performance similar to *traincgf* and it is difficult to predict which algorithm will perform best on a given problem.

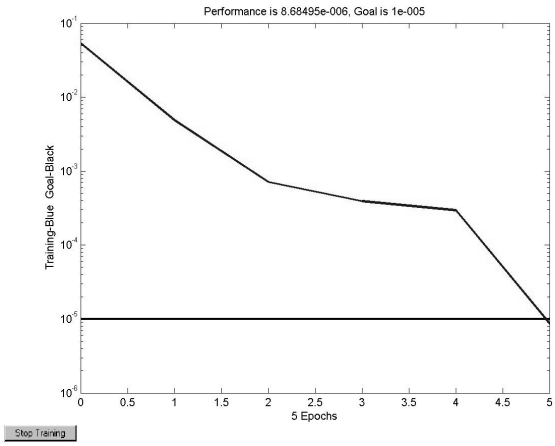
An alternative to the conjugate gradient methods for fast optimization is Newton's method. The basic step of Newton's method is:

$$x_{k+1}=x_k-A_k^{-1}g_k$$

where is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases.

There is a class of algorithms that is based on Newton's method, but which doesn't require calculation of second derivatives. These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. The quasi-Newton method that has been most successful in published studies is the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update. This algorithm has been implemented in the *trainbfg* routine.

The previous network is reinitialized and retrained using the BFGS quasi-Newton algorithm.



TRAINBFG-srchbac, Epoch 0/300, MSE 0.0535683/1e-005, Gradient 0.569755/1e-006

TRAINBFG-srchbac, Epoch 5/300, MSE 8.68495e-006/1e-005, Gradient 0.00641884/1e-006

TRAINBFG, Performance goal met.

This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations.

Like the quasi-Newton methods, 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 (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

$$H = J^T J$$

and the gradient can be computed as

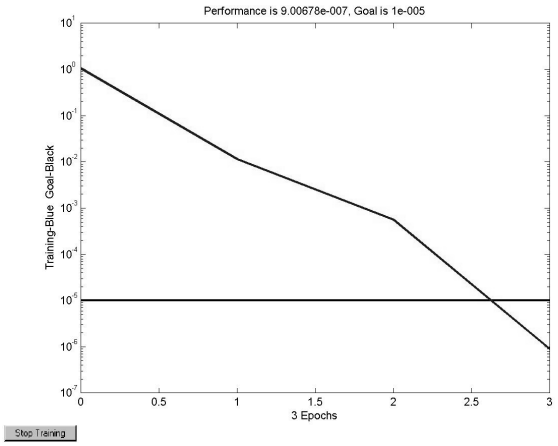
$$g = J^T e$$

where J is the Jacobian matrix that 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-like update:

$$x_{k+1} = x_k - \left[J^T J + \mu I\right]^{-1} J^T e$$

When the scalar μ is zero, this is just Newton's method, using the approximate Hessian matrix. When μ is large, this becomes gradient descent with a small step size. μ is decreased after each successful step and is increased only when a tentative step would increase the performance function: the performance function will always be reduced at each iteration of the algorithm.
The training parameters for trainlm are epochs, show, goal, time, min_grad, max_fail, mu, mu_dec, mu_inc, mu_max, mem_reduc. The parameter mu is the initial value for μ. This value is multiplied by mu_dec whenever the performance function is reduced by a step. It is multiplied by mu_inc whenever a step would increase the performance function. If mu becomes larger than mu_max, the algorithm is stopped. The parameter mem_reduc is used to control the amount of memory used by the algorithm.

```
net=newff(minmax(x),[3,1],{'tansig','purelin'},'trainlm');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,x,y);
```



TRAINLM, Epoch 0/300, MSE 1.05016/1e-005, Gradient 6.23194/1e-010
TRAINLM, Epoch 3/300, MSE 9.00678e-007/1e-005, Gradient 0.00542298/1e-010
TRAINLM, Performance goal met.
This algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has a very efficient MATLAB implementation, since the solution of the matrix equation is a built-in function, so its attributes become even more pronounced in a MATLAB setting.

CONCLUSIONS

It is very difficult to know which training algorithm will be the fastest for a given problem. It will depend on many factors, including the complexity of the problem, the number of data points in the training set, the number of weights and biases in the network, the error goal, and whether the network is being used for pattern recognition (discriminant analysis) or function approximation (regression). In general, on networks which contain up to a few hundred weights the Levenberg-Marquardt algorithm will have the fastest convergence; this advantage is noticeable if very accurate training is required. The next fastest algorithms are the quasi-Newton methods on moderate size networks. The BFGS algorithm does require storage of the approximate hessian matrix, but is generally faster than the conjugate gradient algorithms. Rprop algorithm do not require a line search and have small storage requirements; he is reasonably fast, and very useful for large problems. The variable learning rate algorithm is usually much slower than the others methods, has about the same storage requirements and can be usefull for some problems when using early stopping is better to converge more slowly.

REFERENCES

Dumitrescu D., 2003, Principiile inteligenței artificiale, Editura Albastră, Cluj Napoca
Năstac D. I., 2002, Rețele neuronale artificiale, Editura Printech, Bucureşti
http://www.mathworks.com
http://www.math.uvt.ro