Optimization Schemes for Neural Networks

T.T. Jervis and W.J. Fitzgerald

CUED/F-INFENG/TR 144

August 24, 1993

Cambridge University Engineering Department
Trumpington Street
Cambridge CB2 1PZ
England

Email: ttj10@eng.cam.ac.uk

Optimization Schemes for Neural Networks

Technical Report CUED/F-INFENG/TR 144

T.T. Jervis W.J. Fitzgerald

August 24, 1993

Abstract

Training neural networks need not be a slow, computationally expensive process. The reason it is seen as such might be the traditional emphasis on gradient descent for optimization.

Conjugate gradient descent is an efficient optimization scheme for the weights of neural networks. This work includes an improvement to conjugate gradient descent that avoids line searches along the conjugate search directions. It makes use of a variant of backprop (Rumelhart et al., 1986), called rbackprop (Pearlmutter, 1993), which can calculate the product of the Hessian of the weights and an arbitrary vector. The calculation is exact and computationally cheap.

The report is in the nature of a tutorial. Gradient descent is reviewed and the backpropagation algorithm, used to find the gradients, is derived. Then a number of alternative optimization strategies are described:

- Conjugate gradient descent
- Scaled conjugate gradient descent
- Delta-bar-delta
- RProp
- Quickprop

All six optimization schemes are tested on various tasks and various types of networks. The results show that scaled conjugate gradient descent and quickprop are expedient optimization schemes for a variety of problems.

Contents

1 Introduction

Neural networks map input vectors to output vectors. They are interesting because they have the following properties:

- They perform non-linear mappings
- Input and output can be vector-valued
- They can be trained from examples
- They can generalize (interpolate and extrapolate) from examples
- They operate on real-valued data
- Their structure is amenable to fast parallel processing

Neural networks are parametrized by a set of weights[1]. The task of an optimization scheme for a neural network is to find a set of weights that makes the network perform the desired mapping. This mapping might classify some input data, predict the next state of a discretized system based on the current state and the control action, produce a control action based on the current state of a system, and so on.

For completeness, a brief description of the kind of neural networks covered here will be given below. We do not consider radial basis function (RBF) networks, Hopfield networks, recurrent networks or Boltzmann machines. For a fuller tutorial of neural networks, see, for example, Rumelhart et. al. (Rumelhart and McClelland, 1986). For a good exposition of other neural network training schemes, see Schiffmann et. al. (Schiffmann et al., 1992).

A neural network comprises a set of weighted links and a set of nodes (see figure 1). A node forms a weighted sum x of its input, and has an activation y, a function of x. The function may be linear or non-linear. For nodes in the hidden layer, two candidate functions are the tanh function and the log-sigmoid function:

$$y = \frac{1}{1+\exp(-x)}. \tag{1}$$

Nodes in the output layer often have a linear function ($y = x$) so that the outputs are not limited in size.

1.1 The optimization problem

A neural network optimization scheme must maximize the utility of the parameters of a neural network model, the weights. Weights are found that minimize an error, to be defined below.

A training set comprises a set of patterns p, each pattern being a pair of input and target vectors. When the target vectors for each input are available, the training is *supervised*. For each pattern p, the difference between the output produced by the network $\mathbf{o}_p$ and the target $\mathbf{t}_p$ is an error vector $\mathbf{e}_p$:

$$\mathbf{e}_p = \mathbf{o}_p - \mathbf{t}_p. \tag{2}$$

A scalar measure of the difference between the output and the target is the sum-squared error:

$$e_p = \frac{1}{2}\sum_i(\mathbf{o}_p(i) - \mathbf{t}_p(i))^2 = \frac{1}{2}\mathbf{e}_p^T\mathbf{e}_p \tag{3}$$

[1]Network nodes often have a bias to give non-zero values when there is zero input. A bias term for each node in the network is just like a weighted link from a hypothetical node with unit activation, and can be adapted like any other weight. Using this interpretation means we need only talk of the weights of a network, rather than having to say 'the weights and biases'.

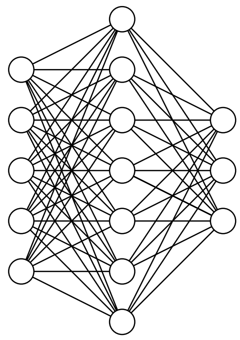


Figure 1: A feedforward neural network for a task mapping a 5-dimensional input space to a 3-dimensional output space. The connections for this network are layered (there are no connections directly between the input and output layer)

where $\mathbf{o}_p(i)$ is element i of the output vector for pattern p and $\mathbf{t}_p(i)$ is the target for that value, the sum being taken over all the elements in the output vector. The error over the whole training set is the sum of the errors e_p over the patterns p:

$$e = \sum_p e_p \tag{4}$$

or equivalently

$$e = \frac{1}{2}\mathrm{Trace}\{E^T E\} \tag{5}$$

where E is a matrix made up of columns of error vectors for each pattern in the training set.

The error over the training set is a function of the weights of the network. The error surface in weight space is a surface whose height at a point is the error e over the training set when the network weights are set to the coordinates of the point in weight space. The concept of the error surface is crucial to understanding optimization strategies. To aid visualization a simple error surface is given in figure 2.

Figure 2 can help to visualize the motivation behind optimization schemes, but should not be taken as being representative of all error surfaces in weight space. It is easy to make very different surfaces for even this simple system.

1.2 Generalization

Part of the utility of neural networks comes from their ability to extrapolate from and interpolate between the training data used to set the weights. Setting the weights of a network so that this generalization is sensible may be done in two ways.

One way is to use cross-validation. These methods use part of the data as a training set, and part as a validation set. One cross-validation method is called *early stopping*. The network is trained for a while on the training set, and then tested on the validation set. As training progresses, the error on the validation set should fall. However, if there are unmodelled dynamics in the data, such as noise, the validation error will eventually start to rise, and the network is being over-trained. Training is stopped when a minimum validation error is reached, even though proceeding might give lower training errors. Unmodelled dynamics mean that it is not desirable for a network to be trained to the point where it can faithfully reproduce the training set, because it would generalize poorly.

Another way to ensure good generalization is to take a Bayesian procedure when training the weights (MacKay, 1991; Thodberg, 1993). This uses a prior probability distribution for the weights and a noise model for the network mapping to find a posterior distribution for the weights in the

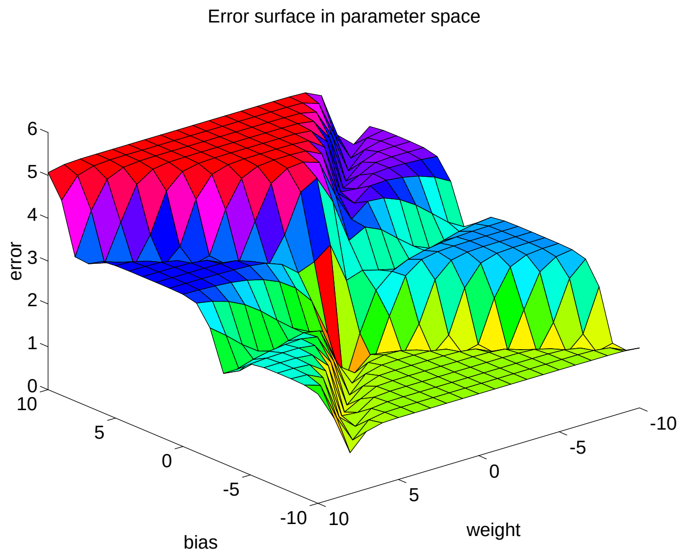


Figure 2: An error surface for the simplest 'network' possible. The network has one node, with one weight and a bias. This is the only system for which an error surface can be plotted. The node has a hyperbolic tangent non-linearity. Three training inputs were used: 0.3, 0.8 and 1.2. The three targets were -0.23, -0.4 and -0.3 respectively.

light of the data. The method uses a modified error surface made up from the data error e defined above and a 'weight error', related to the prior distribution of the weights. This is known in the statistics community as *ridge regression*.

Generalization ability is not a feature of an optimization strategy, however. Both these methods of generalization can benefit from fast optimization strategies, but gradient descent, for example, does not by itself necessarily lead to good generalization.

1.3 On-line and off-line optimization

Off-line optimization methods update the weights after a complete pass of the training data, using the error e over the whole training set. An *epoch* refers to a complete pass through the training data. Thus, off-line methods update the weights after each epoch. This is also known as batch learning.

On-line optimization updates the weights after a single training pattern (a pair of input and target vectors). The error e_p for the single training pattern p is used. On-line methods therefore update the weights several times during an epoch.

An off-line optimization procedure can often be transformed into an on-line procedure by using the error for the current pattern, e_p, instead of the error over the whole training set, e. Weights may also be updated after any number of input/target presentations, so on-line and off-line learning are extremes. Introducing these changes, however, means using different error surfaces and thus complicates the learning dynamics.

Schiffmann et. al. (Schiffmann et al., 1992) report that on-line gradient descent is superior to the advanced techniques running in batch mode when training with large data sets. The problem of training networks on large data sets (compared with the size of the network) is an issue not covered by this report, and is linked with the problem of generalization. The results of this report are valid for a training set of similar size to the number of parameters in the network. For large data sets with redundancy, one could prune the data set or consider an on-line learning scheme.

1.4 Backprop: the chain rule

A resurgence of interest in neural networks occurred when the back-propagation algorithm (commonly referred to as *backprop*) was introduced (Rumelhart et al., 1986). This algorithm is an application of the chain rule to neural networks. The algorithm is able to compute the partial derivative, $\frac{\partial e}{\partial w}$, of the error e with respect to each weight w in the network. $\frac{\partial e}{\partial w}$ is a column vector of n elements, where n is the number of weights in the network:

$$\frac{\partial e}{\partial w} = \nabla e = \left[\frac{\partial e}{\partial w_1}, \frac{\partial e}{\partial w_2}, \cdots, \frac{\partial e}{\partial w_n}\right]^T. \qquad (6)$$

It is important to see back-propagation as an algorithm for computing $\frac{\partial e}{\partial w}$. It is not an optimization procedure in itself, although it is a simple step to implement a gradient descent optimization procedure from the derivatives furnished by backprop. However, other optimization procedures can use the gradients computed by backprop. These optimization procedures can work faster than steepest gradient descent.

The backprop algorithm is derived as follows. Let node i in the network have an input from the other nodes given by x_i and an input from the external world given by I_i. The output, or *activation*, of the node, y_i, is denoted by:

$$y_i = \sigma_i(x_i) + I_i \qquad (7)$$

where σ_i is the node function (e.g. linear, log-sigmoid or hyperbolic tangent). Thus, for nodes in the input layer of the network, $x_i = 0$, and I_i is an element of the input pattern to the network. For nodes not in the input layer, $I_i = 0$ and x_i is given by:

$$x_i = \sum_j w_{ji} y_j \qquad (8)$$

where w_{ji} is the weight[2] of the link between node j and node i.

Equations 7 and 8 define the forward pass of the backprop algorithm for a feedforward network. A feedforward network is one where the connections are such that the values of x_i and y_i may be computed in an order that does not lead to recursion.

We are looking for an expression for $\frac{\partial e}{\partial w}$ for each weight w. We proceed by tracing the effect that changing each weight has on the following chain of dependence: the effect of the weights on the inputs to the nodes; the effect of the inputs to the nodes to the output of the nodes; the effect of the output of the nodes to the error. Firstly, the effect that changing the weight w_{ij} has on the input to node j gives, using equation 8:

$$\frac{\partial e}{\partial w_{ij}} = \frac{\partial e}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} = \frac{\partial e}{\partial x_j} y_i. \qquad (9)$$

An expression for $\frac{\partial e}{\partial x_j}$ in terms of the node activations y_i comes from equation 7:

$$\frac{\partial e}{\partial x_i} = \frac{\partial e}{\partial y_i}\frac{\partial y_i}{\partial x_i} = \frac{\partial e}{\partial y_i}\sigma_i'(x_i) \qquad (10)$$

and finally $\frac{\partial e}{\partial y_i}$ is given by:

$$\frac{\partial e}{\partial y_i} = \frac{de}{dy_i} + \sum_j w_{ij}\frac{\partial e}{\partial x_j} \qquad (11)$$

where $\frac{de}{dy_i}$ is the direct derivative of the error with respect to the activation y_i. $\frac{de}{dy_i}$ is zero for any non-output node. For an output node and the sum-squared error criterion, the right-hand term is zero, and $\frac{de}{dy_i}$ is $y_i - t_i$, where t_i is the target value for the output node i.

---

[2] Sometimes it will be convenient to index the network weights twice, to indicate the source and destination nodes of the weighted link. At other times it will be convenient to number the weights with a single integer, treating the weights as a vector. The convention will be clear from the context.

Equations 9, 10 and 11 define the backward pass of the backprop algorithm. It is so-called because the direct derivatives of the errors at the output nodes (equation 11) are evaluated first, and the results propagated backwards through the network towards the inputs.

2 Steepest gradient descent

The simplest approach to finding suitable weights is to follow the gradient of the error surface in weight space. Let the set of weights at iteration i be denoted by $\mathbf{w}(i)$. Then the update rule for gradient descent is given by:

$$\mathbf{w}(i+1) = \mathbf{w}(i) - \eta \left.\tfrac{\partial e}{\partial w}\right|_i \tag{12}$$

where $\left.\frac{\partial e}{\partial w}\right|_i$ is the vector of gradients of the error e with respect to each weight w evaluated at iteration i, and η is a learning rate parameter. The value of the learning rate parameter, or step size, is crucial for the success of the algorithm.

2.1 Setting the step size

A small step size leads to slow learning and the possibility of getting trapped in local minima of the error surface. A large step size can overshoot the minimum. The weights may then be set to a point in weight space on a high plateau of the error surface. A plateau in the error surface will exist where the nodes have saturated activations. Here, the gradient is small. The small gradient then means the large step size is irrelevant and the network learns slowly again, but this time from a high error. Thus both small and large step sizes are undesirable.

The best step size depends on the error surface, itself a function of the architecture of the network (the number of nodes and the connections between them) and the training data. The following heuristic to set η includes a dependence on the network architecture and the training data:

$$\eta = \frac{0.01}{\left\| \frac{\partial e}{\partial w} \right\|^2} \tag{13}$$

where $\left\| \frac{\partial e}{\partial w} \right\|$ is the Euclidean norm of the error gradient at the first iteration, so

$$\left\| \tfrac{\partial e}{\partial w} \right\|^2 = \sum_w \left(\frac{\partial e}{\partial w} \right)^2 . \tag{14}$$

Equation 13 has been used in the optimization schemes below to set a step size when one is not specified by the optimization strategy. In practice it appears useful for a variety of different architectures and training tasks.

2.2 Line searches

A line search is a search for a minimum of error along a line. It avoids the problem of setting a fixed step size. The search may be exact, or inexact, depending on the method and the stopping criterion.

An inexact line search samples the error at a number of points along the search direction. The search returns the point with the lowest error from among the samples. Samples at points nearer and further than the length of the default step size given in equation 13 might be chosen. The dynamic range of the steps can be modified after each iteration, based on the position of the minimum sample. Such a procedure avoids sticking to a fixed step size, but is inefficient in its sampling strategy. The same number of error evaluations might be done at more appropriate places, based on the results of previous evaluations.

A more efficient line search first finds an interval, along the search direction, in which a minimum exists, and then finds a minimum within the interval. The interval is characterized by three points.

The central point has a lower error than the outer two. The interval therefore brackets a minimum. The interval comprises two sections. A point within the larger section is then sampled, giving four samples. From the four samples, three can be found which form another, narrower, interval that brackets a minimum. The procedure may be continued until the interval is small.

A line search inevitably requires several calculations of error to find a minimum. Gradient information can also be used to improve the search, at further computational cost. The value of a good line search has to be judged against its computational cost. There is a playoff between the effort that should be made to minimize the error in the line search, and the effort that should be made searching in other directions.

2.3 Adding momentum

Adding a momentum term to gradient descent alters the search direction by adding some of the previous search directions to the current gradient. In practice, adding momentum speeds convergence.

Firstly, define the weight change $\boldsymbol{\delta}\mathbf{w}(i)$ for iteration i:

$$\mathbf{w}(i+1) = \mathbf{w}(i) + \boldsymbol{\delta}\mathbf{w}(i). \tag{15}$$

Momentum is used as follows:

$$\boldsymbol{\delta}\mathbf{w}(i) = -\eta \left.\frac{\partial e}{\partial w}\right|_i + \alpha \boldsymbol{\delta}\mathbf{w}(i-1) \tag{16}$$

where $\alpha = 0.9$ is the momentum term and $\boldsymbol{\delta}\mathbf{w}(i-1)$ is the previous weight change. The value of α is usually set to 0.9 independent of the problem and the architecture of the network.

3 Second order methods

Second order optimization methods make use of second derivatives of the error in weight space. The key to second order methods is the Hessian matrix H.

The Hessian matrix is a matrix of second derivatives of e with respect to the weights w:

$$H = \begin{bmatrix} \frac{\partial_2 e}{\partial w_1^2} & \frac{\partial_2 e}{\partial w_1 \partial w_2} & \cdots & \frac{\partial_2 e}{\partial w_1 \partial w_n} \\ \frac{\partial_2 e}{\partial w_2 \partial w_1} & \frac{\partial_2 e}{\partial w_2^2} & \cdots & \frac{\partial_2 e}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial_2 e}{\partial w_n \partial w_1} & \frac{\partial_2 e}{\partial w_n \partial w_2} & \cdots & \frac{\partial_2 e}{\partial w_n^2} \end{bmatrix}. \tag{17}$$

The Hessian contains information about how the gradient changes in different directions in weight space. It answers the question "How does the gradient change if I move off from this point in that direction?". Let the Hessian H be evaluated at a point w in weight space. Let us then consider a direction $\mathbf{v}$ from this point in weight space. The product $H\mathbf{v}$ is the rate of change of the gradient along the direction $\mathbf{v}$ from the point w.

The Hessian is useful for calculating both search directions and step sizes for optimization schemes. It is also large and difficult to compute. Fortunately there are a number of techniques that can implicitly calculate the Hessian, avoiding the need to calculate it or store it explicitly.

3.1 Conjugate gradient descent

The idea behind conjugate[3] gradients is to choose search directions that complement each other. This is meant to avoid the possibility of 'undoing' the minimisation of previous iterations, by choosing appropriate search directions.

Assume the error surface is a quadratic in weight space:

[3] One definition for 'conjugate' from Webster: 'having features in common but opposite or inverse in some particular'

$$e(\mathbf{w}) = \mathbf{c}^T\mathbf{w} + \frac{1}{2}\mathbf{w}^T H\mathbf{w} \tag{18}$$

where $\mathbf{c}$ is a constant and H is a Hessian matrix, constant throughout the weight space. The search direction at iteration i is denoted by $\mathbf{v}(i)$. The searches directions are then conjugate in the following sense: for all $i \neq j$,

$$\mathbf{v}(i)^T H\mathbf{v}(j) = 0. \tag{19}$$

It is not necessary to compute the Hessian to find a set of conjugate directions. Conjugate directions may be generated as follows. Set the first direction, $\mathbf{v}(1)$, to minus the gradient $-\left.\frac{\partial e}{\partial w}\right|_1$. Set subsequent directions using:

$$\mathbf{v}(i) = -\left.\frac{\partial e}{\partial w}\right|_i + \beta(i)\mathbf{v}(i-1) \tag{20}$$

where $\beta(i)$ is a scalar value like an adaptive momentum term. β may be defined in a number of ways, each of which produces conjugate directions. The differences between the definitions show themselves when the error surface is not quadratic. Two definitions for β are given below. The Polak-Ribiere rule (Hertz et al., 1991) for β is given by:

$$\beta(i) = \frac{\left(\left.\frac{\partial e}{\partial w}\right|_i - \left.\frac{\partial e}{\partial w}\right|_{i-1}\right)^T \left.\frac{\partial e}{\partial w}\right|_i}{\left.\frac{\partial e}{\partial w}\right|_{i-1}^T \left.\frac{\partial e}{\partial w}\right|_{i-1}}. \tag{21}$$

The Hestenes-Stiefel rule (Møller, 1993) for β is given by:

$$\beta(i) = \frac{\left(\left.\frac{\partial e}{\partial w}\right|_{i-1} - \left.\frac{\partial e}{\partial w}\right|_i\right)^T \left.\frac{\partial e}{\partial w}\right|_i}{\mathbf{v}(i)^T \left.\frac{\partial e}{\partial w}\right|_{i-1}} \tag{22}$$

and is reported to be more robust to non-quadratic error surfaces (Møller, 1993). In both equations, $\left.\frac{\partial e}{\partial w}\right|_i$ is the column vector of partial derivatives of the error e by each weight w in the network at iteration i.

Conjugate gradient descent will reach the minimum of a quadratic error surface in, at most, as many steps as there are dimensions of the weight space. For non-quadratic error surfaces, the minimum may not have been reached after this number of steps. Instead, the algorithm is restarted by setting β to zero for a step, and the procedure continued as before.

Conjugate gradients give only a search direction, not a step size. Furthermore, the power of conjugate gradients is only apparent if the error is minimized along the current search direction. This is usually done with a line search, the difficulties of which have been outlined above.

3.1.1 Avoiding the line search

If the error function is quadratic, or has a slowly varying Hessian, a good approximation to the optimal step size for a search direction can be found.

If the error function is minimized along a search direction, the gradient of the error function at the minimum is perpendicular to the search direction. Hence:

$$\mathbf{v}^T \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}+r\mathbf{v}} = 0. \tag{23}$$

where $\left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}+r\mathbf{v}}$ is a column vector of partial derivatives of e with respect to each weight evaluated at $\mathbf{w}+r\mathbf{v}$, $\mathbf{v}$ is the search direction and r is the step size along the search direction to the minimum.

Assuming a quadratic surface, equation 18 can be used to find an expression for the gradient:

$$\left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}} = \mathbf{c} + H\mathbf{w} \tag{24}$$

so the gradient at $\mathbf{w} + r\mathbf{v}$ is given by:

$$
\begin{aligned}
\left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}+r\mathbf{v}} &= \mathbf{c} + H(\mathbf{w} + r\mathbf{v}) && (25)\\
&= \mathbf{c} + H\mathbf{w} + rH\mathbf{v} && (26)\\
&= \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}} + rH\mathbf{v}. && (27)
\end{aligned}
$$

Substituting equation 27 into equation 23:

$$
\begin{aligned}
\mathbf{v}^T \left(\left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}} + rH\mathbf{v} \right) &= 0 && (28)\\
\mathbf{v}^T \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}} + r\mathbf{v}^T H\mathbf{v} &= 0. && (29)
\end{aligned}
$$

Rearranging equation 29 gives an expression for r, the distance to the minimum along the current search direction $\mathbf{v}$:

$$
r = -\frac{\mathbf{v}^T \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}}}{\mathbf{v}^T H\mathbf{v}}. \qquad (30)
$$

Equation 30 has been avoided in the past due to the effort of finding the Hessian H. The Hessian H may be approximated by a method of finite differences, or the product $H\mathbf{v}$ may be approximated by taking a difference of gradients. However, the equation becomes useful again in the light of the RBackprop algorithm, given in the appendix. RBackprop gives the product $H\mathbf{v}$ computationally cheaply, and exactly, although this is not necessarily worthwhile for non-quadratic error surfaces.

3.2 Scaled conjugate gradient

Møller (Møller, 1993) has introduced a variation on conjugate gradient descent that takes some account of the non-quadratic nature of the error surface in weight space. This method also benefits from RBackprop.

The quadratic optimal step size given in equation 30 gives the distance to the turning point of the error along the search direction under the quadratic assumption. This turning point may be a maximum or a minimum. The Hessian can be used to find whether the turning point is a maximum or a minimum.

The product $H\mathbf{v}$ of the Hessian H and a direction $\mathbf{v}$ is the rate of change of gradient in the direction $\mathbf{v}$. The expression $\mathbf{v}^T H\mathbf{v}$ is negative if the gradient is increasing along the direction $\mathbf{v}$, and positive if the gradient is decreasing along the direction $\mathbf{v}$. Within the quadratic assumption, the Hessian is constant, and the sign of the change of gradient along the direction is constant. If the gradient is increasing, the graph of the error along $\mathbf{v}$ is a cup, and there is a minimum, otherwise there is a maximum.

To monitor the sign of the product $\mathbf{v}^T H\mathbf{v}$, and therefore the type of the turning point, define δ by

$$
\delta = \mathbf{v}^T H\mathbf{v}. \qquad (31)
$$

If $\delta \leq 0$ for non-zero $\mathbf{v}$, there is a minimum along the direction $\mathbf{v}$. But for non-quadratic error surfaces, it may be that $\delta \leq 0$ and yet there is still a minimum to find, since H changes along $\mathbf{v}$. Møller's scaled conjugate gradient method takes account of this.

Møller introduces two new variables, λ and $\bar{\lambda}$, to define an altered value of δ, $\bar{\delta}$. These variables are charged with ensuring that $\bar{\delta} > 0$. Although this does not affect the error surface, and the Hessian with the quadratic approximation will still suggest there is a maximum along the search direction, his method produces a step size that shows good results in practice.

$\bar{\delta}$ is defined as follows:

$$
\bar{\delta} = \delta + (\bar{\lambda} - \lambda)\mathbf{v}^T\mathbf{v}. \qquad (32)
$$

The requirement for $\bar{\delta} > 0$ gives a condition for $\bar{\lambda}$:

$$\bar{\lambda} > \lambda - \frac{\delta}{\mathbf{v}^T\mathbf{v}}. \tag{33}$$

Møller then sets $\bar{\lambda} = 2\left(\lambda - \frac{\delta}{\mathbf{v}^T\mathbf{v}}\right)$ to satisfy equation 33 and so ensures $\bar{\delta} > 0$. This allows for a substitution in equation 32 to give:

$$\bar{\delta} = -\delta + \lambda \mathbf{v}^T\mathbf{v}. \tag{34}$$

Subsequently, $\bar{\delta}$ is substituted for expressions that would otherwise involve δ. Thus the step size r is found by substituting $\bar{\delta}$ for δ in equation 30:

$$r = -\frac{\mathbf{v}^T \frac{\partial e}{\partial w}}{\bar{\delta}} = \frac{\mathbf{v}^T \frac{\partial e}{\partial w}}{\delta - \lambda \mathbf{v}^T\mathbf{v}}. \tag{35}$$

It is now necessary to specify how to update the scale λ. This is based on a measure of fit between the quadratic approximation and the real surface. The fit is measured by the variable Δ, and is a ratio between the actual change in error e produced by stepping along the search direction $\mathbf{v}$ by an amount r, and the predicted change in that error based on the quadratic approximation:

$$\Delta = \frac{e|_{\mathbf{w}+r\mathbf{v}} - e|_{\mathbf{w}}}{e_q|_{\mathbf{w}+r\mathbf{v}} - e|_{\mathbf{w}}} \tag{36}$$

where $e_q|_{\mathbf{w}+r\mathbf{v}}$ is the quadratic approximation of the error at $\mathbf{w} + r\mathbf{v}$, given by the first three terms of the Taylor expansion:

$$e_q|_{\mathbf{w}+r\mathbf{v}} = e|_{\mathbf{w}} + r \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}}^T \mathbf{v} + \frac{1}{2} r^2 \bar{\delta} \tag{37}$$

where a term that would naturally be $\mathbf{v}^T H|_{\mathbf{w}}\mathbf{v}$ has been replaced by the 'new' value for δ, $\bar{\delta}$.

Substituting equation 37 into equation 36 gives:

$$\Delta = \frac{e|_{\mathbf{w}+r\mathbf{v}} - e|_{\mathbf{w}}}{\left(e|_{\mathbf{w}} + r \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}}^T \mathbf{v} + \frac{1}{2} r^2 \bar{\delta}\right) - e|_{\mathbf{w}}} \tag{38}$$

$$\Delta = \frac{e|_{\mathbf{w}+r\mathbf{v}} - e|_{\mathbf{w}}}{r \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}}^T \mathbf{v} + \frac{1}{2} r^2 \bar{\delta}} \tag{39}$$

and substituting for r (equation 35) gives:

$$\Delta = \frac{e|_{\mathbf{w}+r\mathbf{v}} - e|_{\mathbf{w}}}{\left(-\frac{\mathbf{v}^T \frac{\partial e}{\partial w}}{\bar{\delta}}\right) \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}}^T \mathbf{v} + \frac{1}{2}\left(-\frac{\mathbf{v}^T \frac{\partial e}{\partial w}}{\bar{\delta}}\right)^2 \bar{\delta}} \tag{40}$$

$$\Delta = \frac{e|_{\mathbf{w}+r\mathbf{v}} - e|_{\mathbf{w}}}{-\frac{\left(\mathbf{v}^T \frac{\partial e}{\partial w}\right)^2}{\bar{\delta}} + \frac{1}{2}\frac{\left(\mathbf{v}^T \frac{\partial e}{\partial w}\right)^2}{\bar{\delta}}} \tag{41}$$

$$\Delta = \frac{2\bar{\delta}\left(e|_{\mathbf{w}} - e|_{\mathbf{w}+r\mathbf{v}}\right)}{\left(\mathbf{v}^T \frac{\partial e}{\partial w}\right)^2}. \tag{42}$$

λ is then updated as follows:

$$\lambda(i+1) = \begin{cases} \frac{1}{4}\lambda(i) & \text{if } \Delta(i) > 0.75 \\ \lambda(i) + \frac{\mathbf{v}(i)^T H(i) \mathbf{v}(i)(1-\Delta(i))}{\mathbf{v}(i)^T\mathbf{v}(i)} & \text{if } 0 < \Delta(i) < 0.25 \\ \lambda(i) & \text{otherwise} \end{cases} \tag{43}$$

where $\lambda(i)$ is the value of λ at iteration i.

Finally, a step is only taken if $\Delta > 0$. If $\Delta \leq 0$, the next iteration is started at the current point in weight space, in a new conjugate direction. The algorithm is started by setting the first value of λ, $\lambda(1)$, to be small. For the comparisons below, $\lambda(1) = 10^{-6}$.

3.3 A place for RBackprop

Møller uses a difference of gradients to approximate δ above. His approximation for δ, $\tilde{\delta}$, is given by:

$$\tilde{\delta} = \frac{\mathbf{v}^T \left(\frac{\partial e}{\partial w}\Big|_{\mathbf{w}+\eta\mathbf{v}} - \frac{\partial e}{\partial w}\Big|_{\mathbf{w}} \right)}{\eta} \tag{44}$$

where η is a scaled step along the direction $\mathbf{v}$, given by:

$$\eta = \frac{\sigma}{\mathbf{v}^T\mathbf{v}} \tag{45}$$

where $0 < \sigma \leq 10^{-4}$.

RBackprop (given in the appendix) calculates $H|_w \mathbf{v}$ exactly. This avoids the need to substitute $\tilde{\delta}$ for δ above, and eliminates the variables σ and η.

4 Local optimization methods

The methods covered so far have grouped the weights together to find a search direction and step size. The weights were updated in proportion to their contribution to the global search direction and step size.

Local optimization methods consider local changes for each weight. The methods are not gradient descent methods. The step in weight space is not necessarily along an error gradient. Each weight is treated as though the others did not exist. Because the methods below make use of the previous step's local gradients, they are still second order methods.

4.1 Delta-bar-delta

This technique uses gradient descent for the search direction, and then applies individual step sizes for each weight, which means the actual direction taken in weight space is not necessarily along the line of the steepest gradient. It was introduced by Jacobs (Jacobs, 1988).

The basic idea is as follows: if the weight updates between consecutive iterations are in opposite directions, the step size is decreased, otherwise it is increased. This is prompted by the idea that if the weight changes are oscillating, the minimum is between the oscillations, and a smaller step size might find that minimum. The step size may be increased again once the error has stopped oscillating.

Let $\boldsymbol{\eta}(i)$ be a vector of step sizes, one for each weight w_k, at iteration i. Then $\boldsymbol{\eta}$ is updated as:

$$\boldsymbol{\eta}(i+1) = \boldsymbol{\eta}(i) + \boldsymbol{\delta\eta}(i) \tag{46}$$

where $\boldsymbol{\delta\eta}(i)$ is a vector of changes for each learning rate, given by:

$$\boldsymbol{\delta\eta}_k(i) = \begin{cases} \kappa & \text{if } \bar{\delta}_k(i-1) \frac{\partial e}{\partial w_k}\Big|_i > 0 \\ -\phi\boldsymbol{\eta}_k(i) & \text{if } \bar{\delta}_k(i-1) \frac{\partial e}{\partial w_k}\Big|_i < 0 \\ 0 & \text{otherwise} \end{cases} \tag{47}$$

where $\boldsymbol{\delta\eta}_k(i)$ is the learning rate for weight w_k at iteration i, and

$$\bar{\delta}_k(i) = (1-\theta) \frac{\partial e}{\partial w_k}\Big|_i + \theta\bar{\delta}_k(i-1). \tag{48}$$

This procedure introduces three extra parameters that need to be set: κ, ϕ and θ. Jacobs finds values of the parameters that work well for particular tasks, but for the simulations below $\phi = 0.1$, $\theta = 0.7$, and κ was set to the same value as the initial learning rate for each weight, a vector of multiple copies of the default step size given in equation 13.

4.2 RProp

Schiffmann et. al. (Schiffmann et al., 1992) found RProp to be the fastest training algorithm they tested in their review of training methods.

RProp uses different step sizes for each weight, like delta-bar-delta. However, it only uses the sign of the local gradient $\frac{\partial e}{\partial w_k}$ when updating the weight w_k, not its magnitude.

The vector of step sizes $\boldsymbol{\eta}$ is defined as follows:

$$\boldsymbol{\eta}_k(i) = \begin{cases} \min(1.2\boldsymbol{\eta}_k(i-1), \eta_{\max}) & \text{if } \left.\frac{\partial e}{\partial w_k}\right|_i \left.\frac{\partial e}{\partial w_k}\right|_{i-1} > 0 \\ \max(0.5\boldsymbol{\eta}_k(i-1), \eta_{\min}) & \text{if } \left.\frac{\partial e}{\partial w_k}\right|_i \left.\frac{\partial e}{\partial w_k}\right|_{i-1} < 0 \\ \boldsymbol{\eta}_k(i-1) & \text{otherwise} \end{cases} \tag{49}$$

where $\eta_{\max}$ and $\eta_{\min}$ limit the size of the step above and below. The values used in the tests below were $\eta_{\max} = 1$ and $\eta_{\min} = 10^{-7}$. The maximum value was chosen to be about a tenth of the expected range of the weights. The minimum value was chosen to be about the expected necessary resolution of the weights. Reducing $\eta_{\min}$ to 10^{-40} produced a lower final error for some tasks, but did not lead to faster or slower training otherwise. The step sizes were initialised using the default step size given in equation 13.

The weights were updated as follows:

$$\boldsymbol{\delta}\mathbf{w}_k(i) = \begin{cases} -\text{sign}\left(\left.\frac{\partial e}{\partial w_k}\right|_i\right)\boldsymbol{\eta}_k(i) & \text{if } \left.\frac{\partial e}{\partial w_k}\right|_i \left.\frac{\partial e}{\partial w_k}\right|_{i-1} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{50}$$

such that $\mathbf{w}(i+1) = \mathbf{w}(i) + \boldsymbol{\delta}\mathbf{w}(i)$. A further detail concerns the stored value of the previous gradient $\left.\frac{\partial e}{\partial w_k}\right|_{i-1}$. If, for a particular weight, $\left.\frac{\partial e}{\partial w_k}\right|_i \left.\frac{\partial e}{\partial w_k}\right|_{i-1} < 0$, the value of the stored gradient for that weight would be set to 0 for the next time step.

4.3 Quickprop

Quickprop (Fahlman, 1988) assumes each weight has a quadratic error curve. It implicitly assumes the Hessian is diagonal. The technique works well in practice, but requires adjustment to avoid instability.

Consider any single weight in the network. This weight is assumed to affect the error independently of the others. Furthermore, the graph of the error against the value of the weight is assumed to be a quadratic. A quickprop step sets the weight to the minimum of the quadratic approximation.

This step may not be possible. Firstly, the quadratic may have a maximum and not a minimum. Secondly, there may not be enough information to approximate the quadratic. This can happen when the procedure is first started, but also if there has been no weight change between consecutive epochs. This is because the quadratic is approximated using a difference of gradients between the current and the previous epoch.

When the quickprop step breaks down, a gradient descent step is used to keep the procedure moving.

The details of the algorithm are as follows. A quickprop step for iteration i, $\boldsymbol{\delta}\mathbf{w}_{qp}(i)$, is defined by:

$$\boldsymbol{\delta}\mathbf{w}_{qp}(i) = -\frac{\left.\frac{\partial e}{\partial w}\right|_i \boldsymbol{\delta}\mathbf{w}(i-1)}{\left.\frac{\partial e}{\partial w}\right|_i - \left.\frac{\partial e}{\partial w}\right|_{i-1}} \tag{51}$$

where $\boldsymbol{\delta}\mathbf{w}(i-1)$ is the weight update at the previous epoch, not necessarily the same as $\boldsymbol{\delta}\mathbf{w}_{qp}(i-1)$, since adjustments are necessary to improve convergence. When the denominator of equation 51 is zero for a weight, the quickprop step for that weight is set to zero.

The next steps mend the quickprop approximation. Firstly, the weight changes are limited by setting them to be the minimum of the quickprop weight change and 1.75 times the previous weight change. Secondly, steps are not taken that find a maximum of the quadratic approximation. These steps are set to 1.75 time the previous weight change. Finally, a gradient descent step is found to add to the weight changes found above. This starts the algorithm when all the quickprop steps are zero, and helps to change weights whose previous changes were zero.

The adjustments amount to a hybrid weight change $\boldsymbol{\delta}\mathbf{w}(i)$, given by:

$$\boldsymbol{\delta}\mathbf{w}_k(i) = \begin{cases} \boldsymbol{\delta}\mathbf{w}_{qp_k}(i) - \eta\frac{\partial e}{\partial w_k}(i) & \text{if } \frac{\partial e}{\partial w_k}(i)\frac{\partial e}{\partial w_k}(i-1) < 0 \\ \boldsymbol{\delta}\mathbf{w}_{qp_k}(i) & \text{otherwise} \end{cases} \tag{52}$$

where η is a fixed step size, that may be set using the default size of equation 13.

A number of alternatives around the basic idea of the quickprop step can be used. For example, the gradient may be adjusted by adding 0.1 to the values of $\sigma'_k(x)$ used in the backprop algorithm used to calculate $\frac{\partial e}{\partial w}$ (see equation 10). The motivation for this step is to avoid the flat areas of the derivative of the node activation functions slowing convergence. One can also use a modified error function. Fahlman details both these modifications (Fahlman, 1988).

5 Comparisons

The power of an optimization scheme rests on the appropriateness of the approximations made about the error surface. For example, if the error surface is well approximated by a quadratic, conjugate gradient methods may be expected to perform well. If the error surface is not quadratic, conjugate gradient will perform poorly. This is not to say that conjugate gradient descent is a poor optimization scheme. It says that the performance of an optimization scheme will depend on the match between the assumptions inherent in the scheme and the error surface of the task.

In practice it is desirable to know how robust an optimization scheme is, that is, how well an optimization scheme works on different training tasks. An indication of the robustness of the various schemes discussed above is given in a comparison of their performance for different tasks.

Among the factors that will affect the convergence of a neural network optimization scheme are:

1. The function to be mapped,
2. The network's node functions,
3. The architecture of the network,
4. The initial weight settings and the starting point in weight space.

The comparisons below vary the first three factors. The forth was not studied in detail. Five training sets are used to show variation over the function to be approximated. Networks with tanh hidden units and log-sigmoid (equation 1) hidden units are compared. Finally, networks with and without direct connections from input layer to output layer are compared.

5.1 The tasks

To measure the performance of the training schemes described here, five different training tasks have been set. They are:

sin Sine function

abs Absolute value function

xor Exclusive-or

10encoder 10-5-10 encoder

simpole 5 input, 4 output real-valued control plant data

The sine and absolute value tasks have been included because good performance on the sine task was not necessarily followed by good performance on the absolute value task.

The sine task used input values $\mathbf{x} = [-3, -2.8, -2.6, \cdots, 2.8, 3]$. The targets were $\sin(\mathbf{x})$. The absolute value task used the same input values. The targets for this task were the absolute values of the inputs, $|x|$.

Exclusive-or is a standard task for neural networks, of little practical importance but often quoted in benchmark tests. The four input patterns were $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$. The targets were respectively 0.1, 0.9, 0.9 and 0.1.

The 10-5-10 encoder task required the network to produce a description of the 10 input nodes in the hidden layer, so that the input could be reproduced at the output. Ten input patterns were used, the ten rows of a 10 by 10 identity matrix, so the inputs alternately turned on a single node of the input. The targets were identical to the inputs for this task. This task is trivial for a fully connected network, but it is still a valid training task to compare different optimization schemes.

The control plant data was generated from a simulation of an inverted pendulum system (Barto et al., 1983). The state was set to random positions in the four-dimensional state space. A force was applied, and the state 0.02 seconds after the force was applied was found by Euler integration of the accelerations. The initial state and control action are the inputs to the network, the change in the state is the target. 270 input/target pairs were used.

5.2 The networks

Different networks were tried for all the tasks. Some networks were not naturally suited to the tasks, for example the encoder problem is trivial with a fully-connected feedforward network. However, each task forms an error surface for a network, and so the optimization schemes may still be assessed on their relative performance.

The networks were varied around a default network. The default network had a single hidden layer of five tanh nodes, and an output layer of linear nodes. The number of input and output nodes was defined by each task. The connections linked the input layer to the hidden layer, and the hidden layer to the output layer. This is a layered connectivity.

Comparisons were then made by modifying the default network by using 2 and then 8 hidden nodes. Then the performance was compared by modifying the default network to use log-sigmoid instead of tanh hidden nodes. Finally, results for a default network modified for full connectivity were obtained. Full connectivity means layered connectivity with an extra set of weights linking input and output nodes. Full connectivity makes the encoder problem trivial, but it is also the sensible connectivity for a network designed to predict the next state of continuous control plant given the current state and control action, since in this case the target is only a perturbed version of the state component of the input vector.

The initial weights for all the networks were sampled from a zero-mean Gaussian distribution with a variance of 0.04.

5.3 The training schemes

The training schemes compared in the results are:

gdmom Gradient descent with momentum

hscgrbp Hestenes-Stiefel conjugate gradient descent with RBackprop

scgrbp Scaled conjugate gradient descent with RBackprop

rprop RProp

deltabardelta Delta-bar-delta

qp Quickprop

gdmom used a momentum parameter $\alpha = 0.9$. Section 6.1 discusses tuning the parameters of this scheme.

The Hestenes-Stiefel conjugate gradient scheme used the Hestenes-Stiefel rule (equation 22 to find conjugate search directions. The step size was set by the quadratic-optimal step size given in equation 30. RBackprop was used to calculate the product of the Hessian and the search direction.

The scaled conjugate gradient method also used RBackprop to calculate the product of the Hessian and the search direction. The details are given by Møller (Møller, 1993), save for the substitution of RBackprop in place of Møller's approximation to compute the product of the Hessian and the current search direction. Although the accuracy of RBackprop is not strictly required (since the product is used in an approximation), it eliminates an extra parameter from the algorithm.

6 Results

The results plot the training error for each optimization scheme over a series of iterations for the different tasks and networks. Each scheme was run for 10^6 floating-point operations. This is why plots of the error against the number of epochs have different lengths.

The majority of the results are given as plots of training error against the number of floating-point operations used by each optimization algorithm. Using flop-counts rather than execution time allows the data to be independent of variations in the computing load not directly related to the optimization, and gives an indication of the performance that might be expected using careful programming.

Figure 3 shows the difference between plotting against computational effort, and plotting against the number of epochs. Gradient descent is computationally cheap per epoch. For the same number of floating-point operations, more epochs of gradient descent may be completed. The graphs show, however, that the extra epochs do not make up the difference. Quickprop is also able to do many iterations for the same computational effort. However, quickprop's extra epochs allow it to compete with the conjugate gradient techniques, often winning, given the same computational constraints. The graphs are not identical in length, because the granularity of the floating-point requirements of the various procedures in the training loop interfere with the stopping criterion of 10^6 flops.

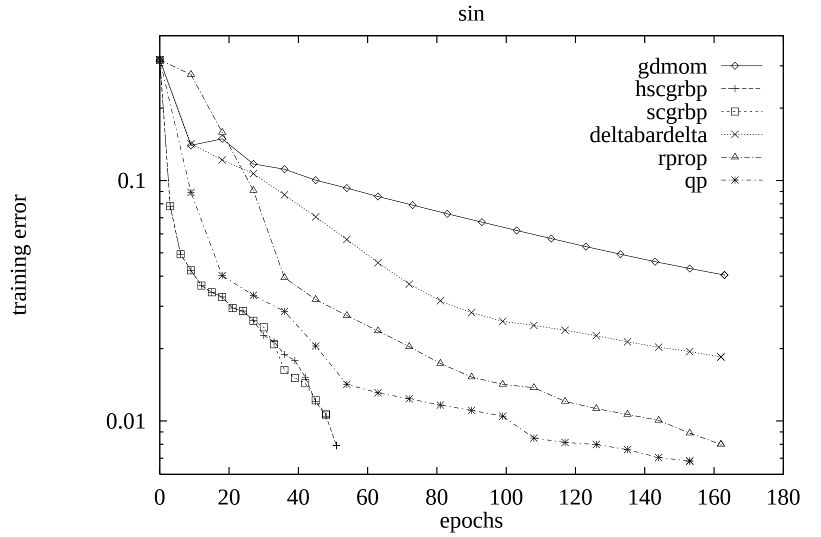
The link between floating-point operations and cpu time is shown in figure 4. As should be expected, the curves are similar. Plotting the training error against flop-count rather than cpu time allows the results to be independent of the available computational power, and to a certain extent the efficiency of the schemes' implementations.

Figure 5 gives the relative performances of the optimization schemes for the single-input, single-output sine and absolute value tasks on the default network of 5 tanh nodes in the hidden layer, and a layered connectivity. These tasks were chosen for comparison because schemes that worked well on sine were found to work less well on the absolute value task. Figure 6 gives the relative performances of the optimization schemes for the xor, encoder and plant data tasks.

Figures 7 and 8 plot the results of the training schemes on the training tasks for a default network modified by using only 2 nodes in the hidden layer. Figures 9 and 10 plot the results using 8 hidden layer nodes.

The effect of changing the non-linearity in the hidden layer is shown in Figures 11 and 12. These figures plot the training error for the tasks and the optimization schemes using the default network with 5 log-sigmoid hidden layer nodes.

Finally, the effect of changing the connectivity is shown in figures 13 and 14. These figures plot the results when a set of weighted links is added between the input and the output nodes.



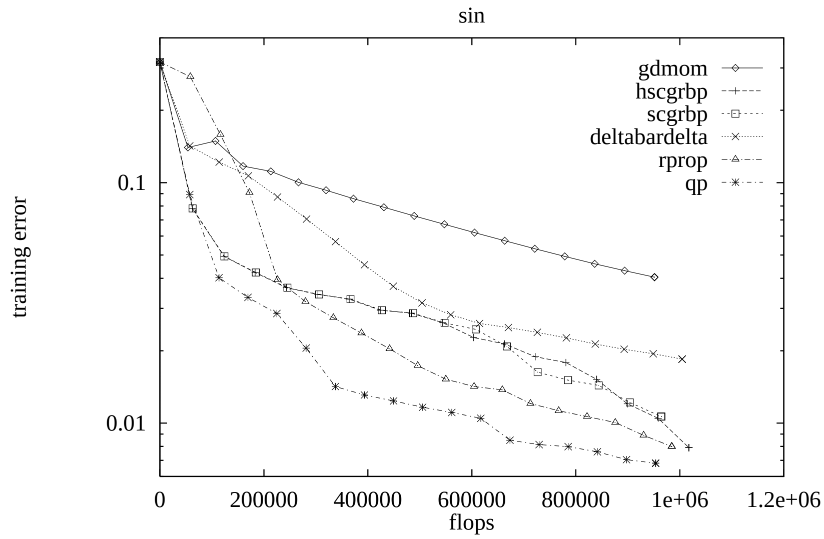


Figure 3: The upper graph plots the training error against epoch for the sine task on the default network. The lower graph plots the training error against the number of floating-point operations. The advantage of the advanced schemes over gradient descent appears diminished in the lower graph, but it is still clear that they are more efficient.

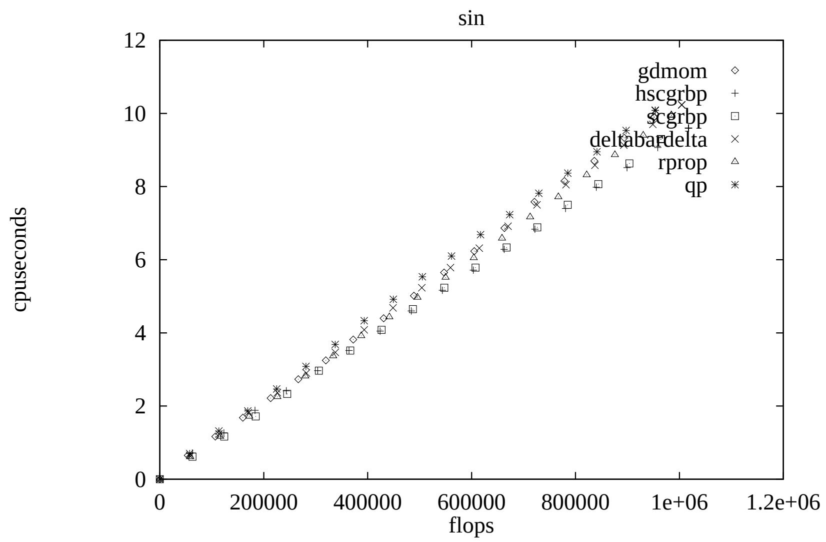


Figure 4: The linear relationship between flops and cpu seconds verifies that the floating-point count is linearly related to the computation time.

6.1 Gradient descent with momentum

Gradient descent with momentum is a commonly-used training algorithm, and so we devote a small section to it here. Figure 15 shows the results of training the default network on the simulated pole data using the following schemes:

gd Steepest gradient descent;

gdmom Gradient descent with momentum, $\alpha = 0.9$;

gdmom0.1 Gradient descent with momentum, $\alpha = 0.1$;

gdmomB Gradient descent with momentum, equation 53, $\alpha = 0.9$;

gdmomB0.1 Gradient descent with momentum, equation 53, $\alpha = 0.1$;

qp Quickprop.

All the gradient descent schemes used the default step size given by equation 13. Updating with equation 53 sets the weight change $\boldsymbol{\delta}\mathbf{w}$ by:

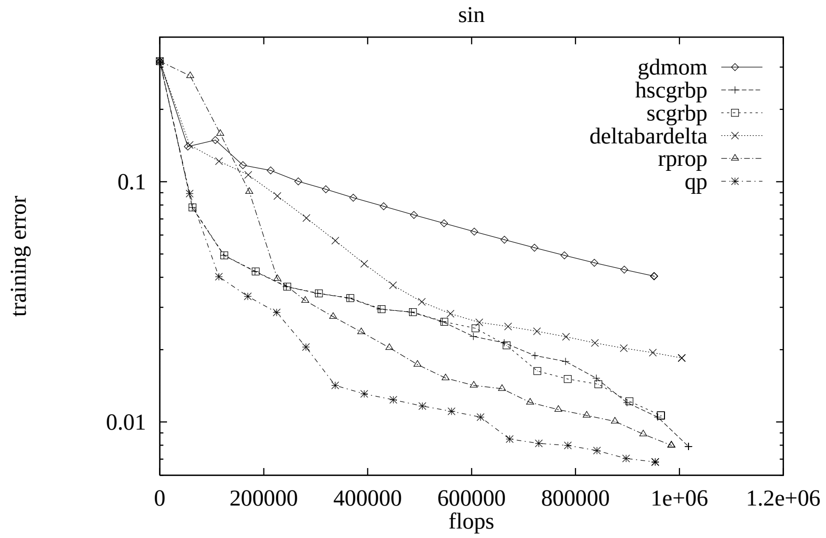
$$\boldsymbol{\delta}\mathbf{w}(i) = -(1-\alpha)\eta \left.\frac{\partial e}{\partial w}\right|_i + \alpha\boldsymbol{\delta}\mathbf{w}(i-1), \tag{53}$$

differing from the conventional update given in equation 16 by an added $(1-\alpha)$ term.

The details of the quickprop scheme have been given above. It is included in the graph to emphasize its improvement over standard gradient descent.

The graph shows that gradient descent with momentum, setting $\alpha = 0.9$ and using equation 16, is the best first-order gradient descent method tried. It was also the best first-order method when tested with the other tasks.

The results show that the second-order methods do better. It is possible that a first-order method could be found with more appropriate step size and momentum parameters, which might approach the performance of the second-order methods. However, the effort required to tune the parameters seems pointless, since the second-order methods already work.



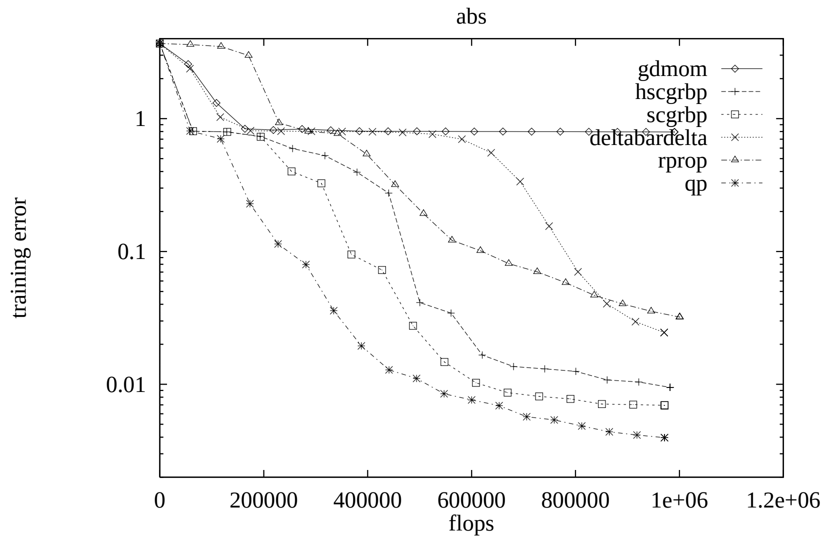


Figure 5: The default network trained to perform (upper) the sine mapping and (lower) the absolute value mapping.

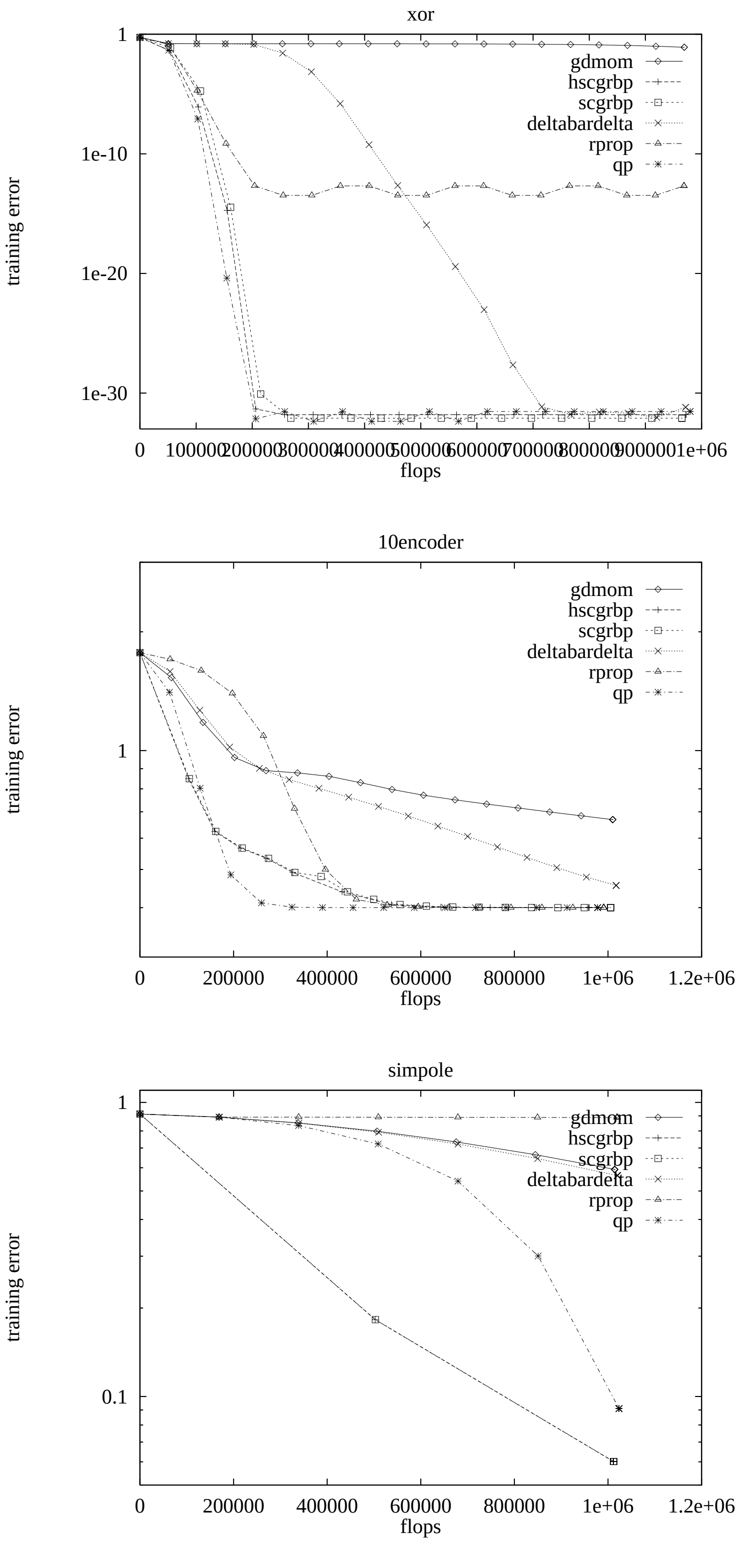


Figure 6: The default network trained on xor, an encoder problem and a simulated control system.

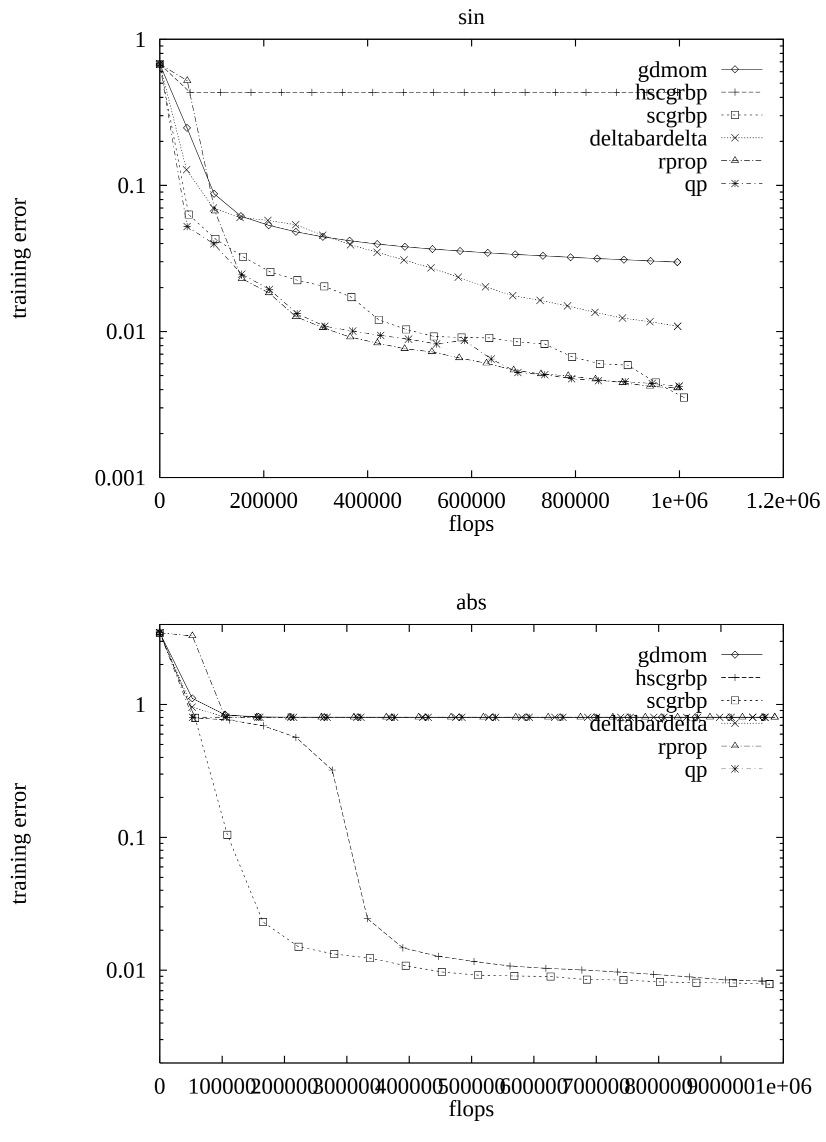


Figure 7: The default network with 2 hidden nodes trained on sine and absolute value functions.

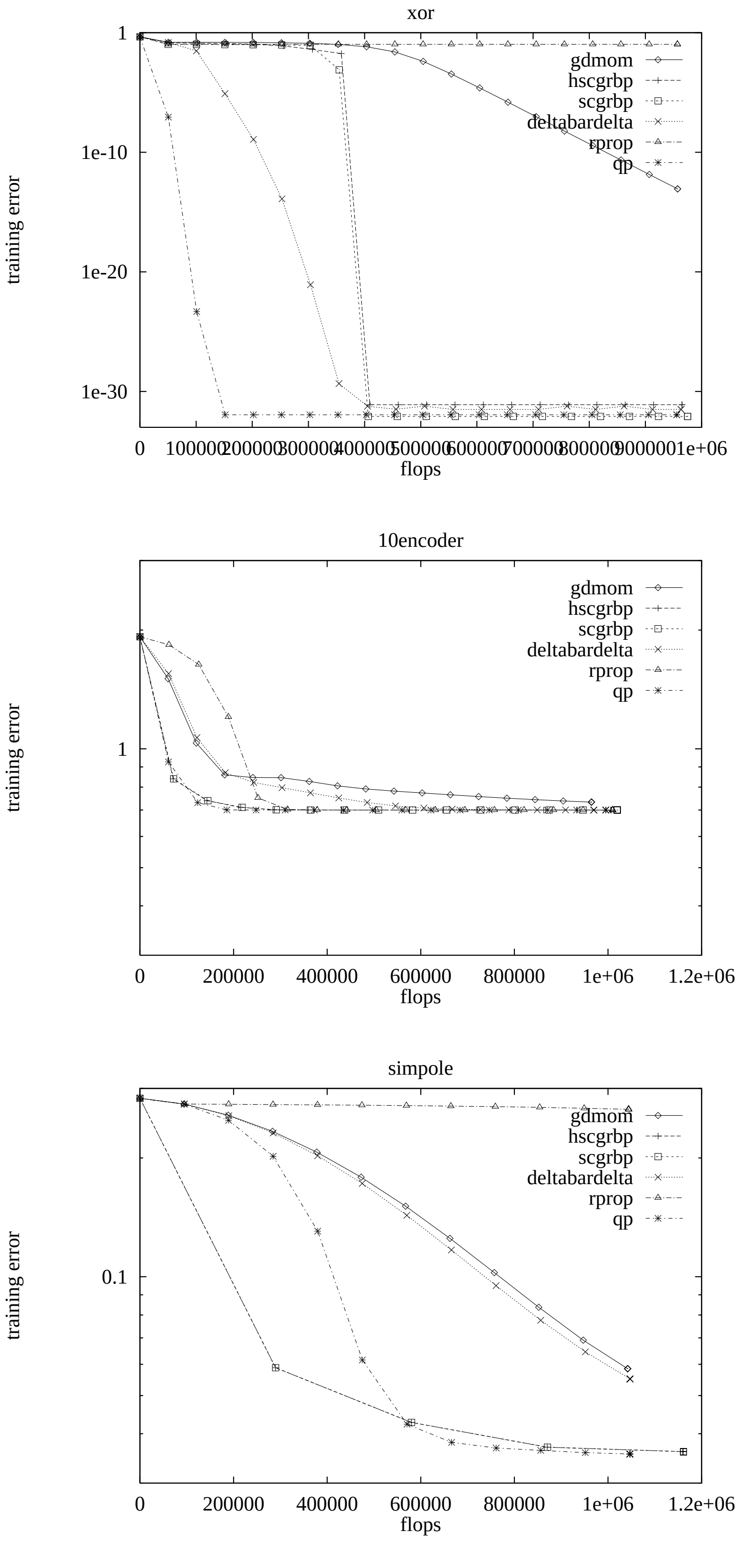


Figure 8: The default network with 2 hidden nodes trained on xor, an encoder problem and a simulated control system.

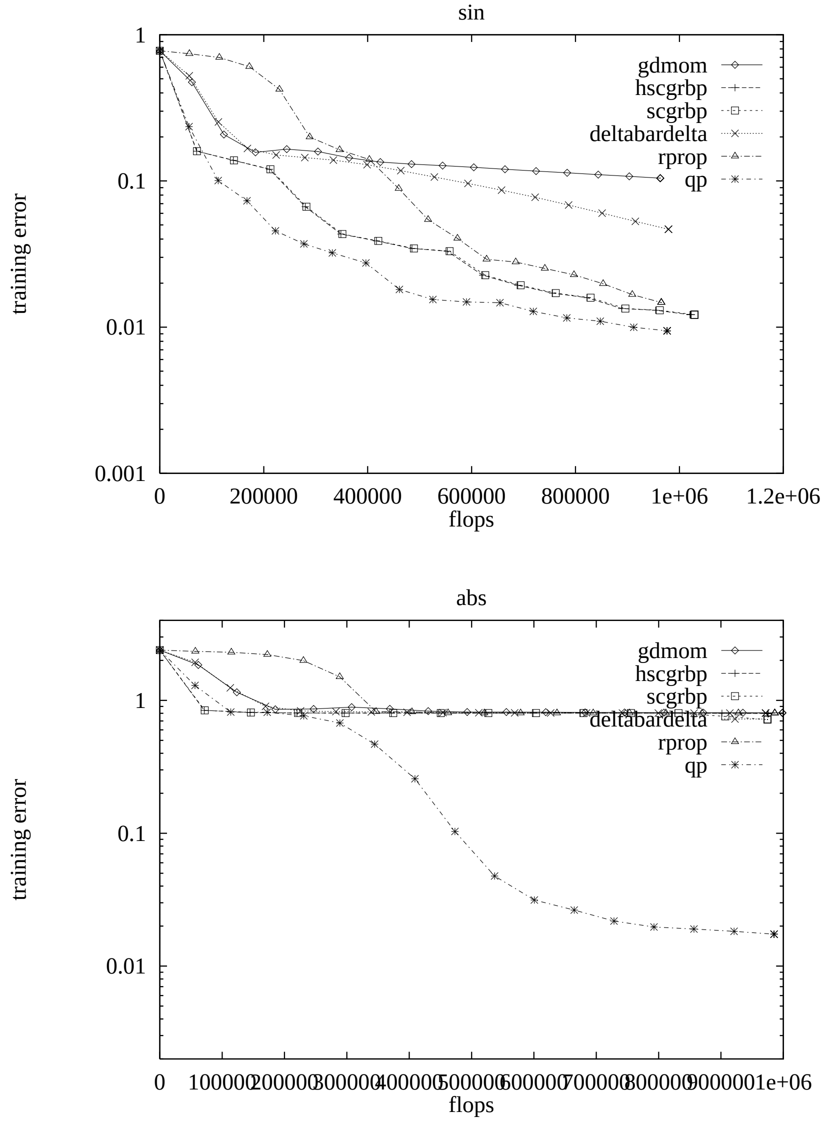


Figure 9: The default network with 8 hidden nodes trained on sine and absolute value functions.

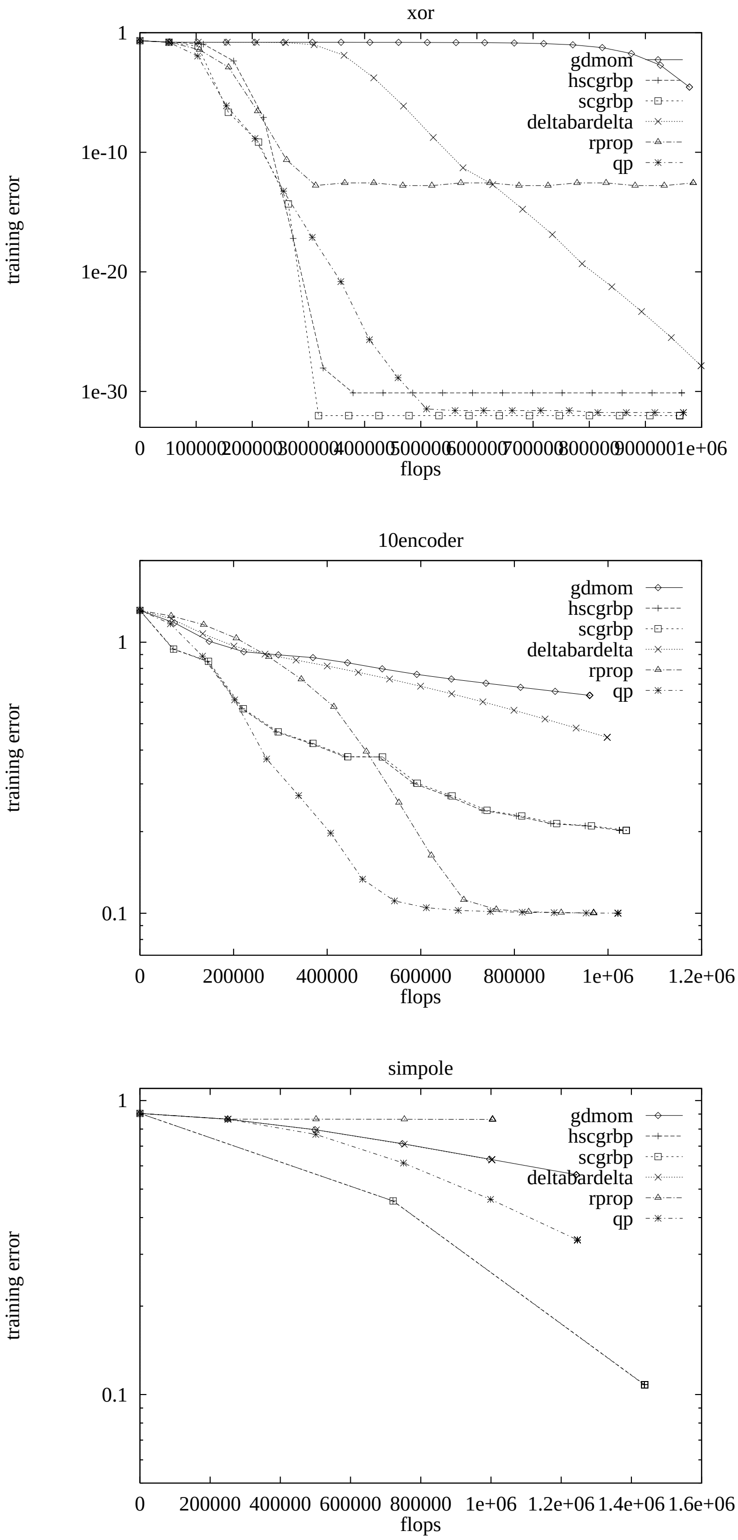
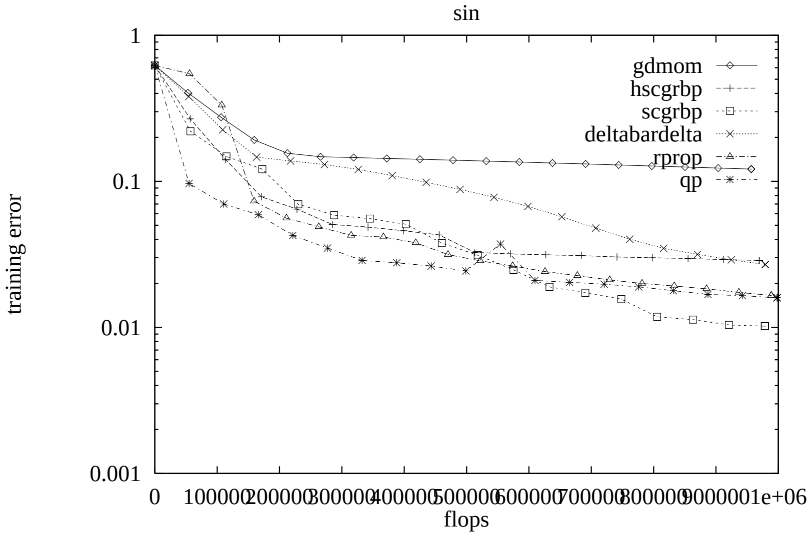


Figure 10: The default network with 8 hidden nodes trained on xor, an encoder problem and a simulated control system.



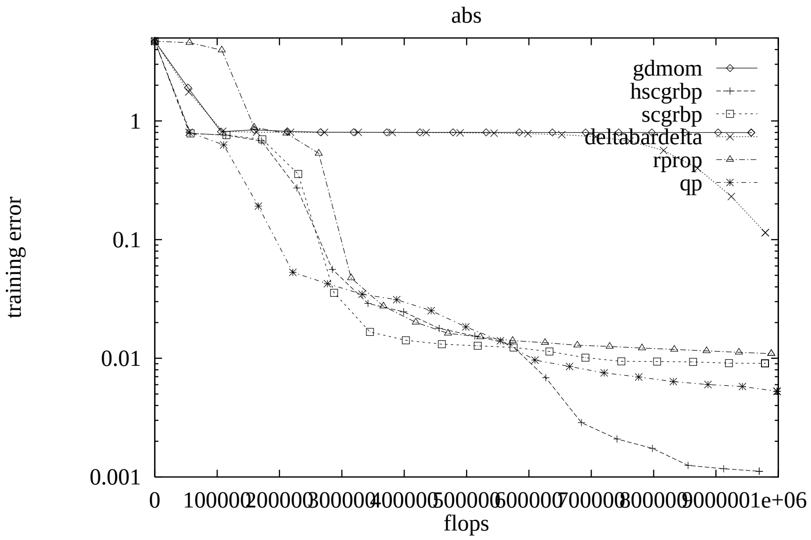


Figure 11: The default network with log-sigmoid non-linearities trained on sine and absolute value functions.

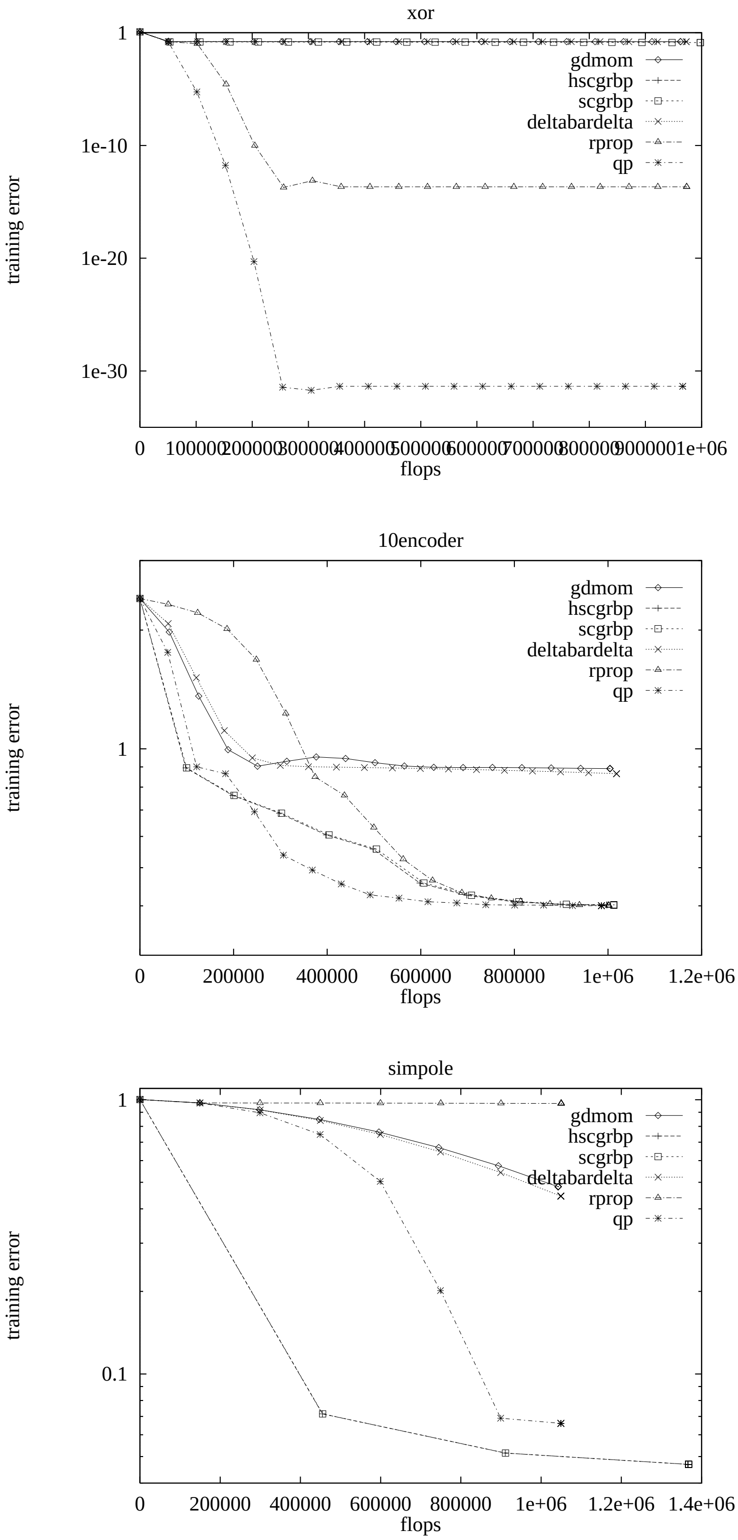


Figure 12: The default network with log-sigmoid non-linearities trained on xor, an encoder problem and a simulated control system.

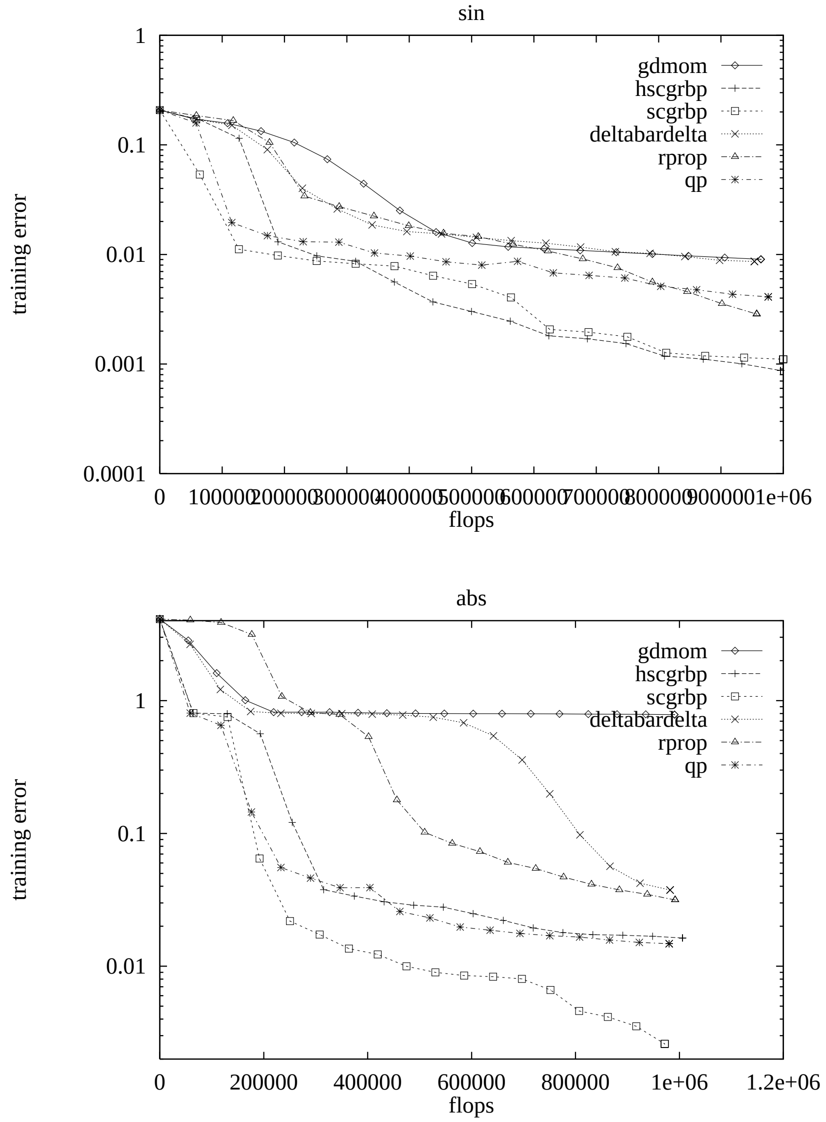


Figure 13: The default network with full connectivity trained on sine and an absolute value functions.

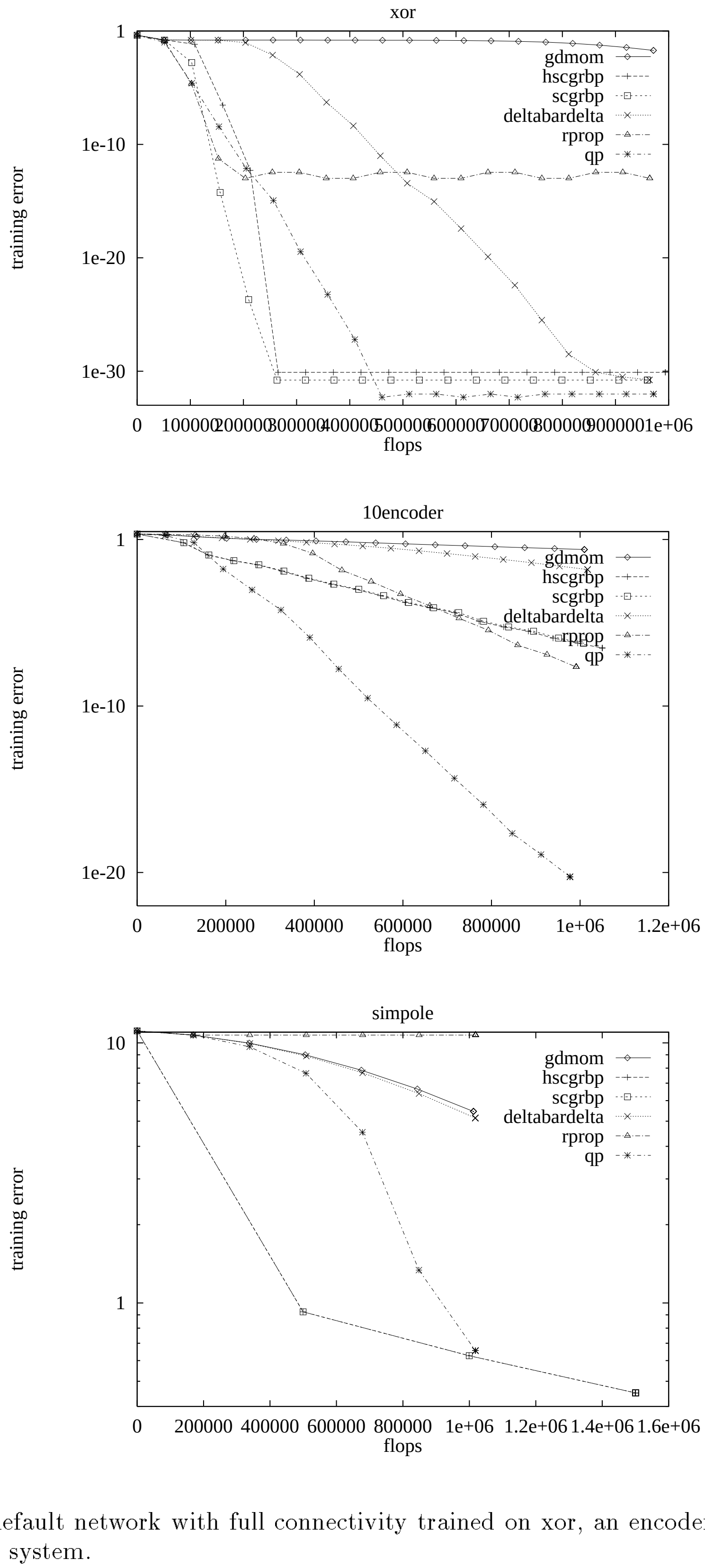


Figure 14: The default network with full connectivity trained on xor, an encoder problem and a simulated control system.

6.2 The effect of RBackprop on scaled conjugate gradient

Figure 16 plots the training error for the same tasks comparing scaled conjugate gradient (scg) using Møller's Hessian approximation, and scaled conjugate gradient using RBackprop (scgrbp). The results show similar performance for all the problems.

7 Summary

We have described the principles of some neural network training techniques. Sources of more detail for the schemes have been given so that the interested reader might implement the schemes. The schemes have been compared for a variety of small tasks and a variety of networks, with varying sizes, connectivity and non-linear elements. The RBackprop algorithm, re-introduced by Pearlmutter (Pearlmutter, 1993), has been described and used within the scaled conjugate gradient algorithm (Møller, 1993).

A neural network optimization scheme makes assumptions about the error surface in weight space. The success of a scheme will depend on the validity of the assumptions. For different problems, the assumptions will be appropriate to varying degrees. This means there is no optimization scheme that will perform better than all other schemes on all possible tasks.

We do not recommend any particular optimization scheme. For a particular kind of task, a number of optimization schemes should be compared to find which ones are efficient. Candidate schemes should include scaled conjugate gradient and quickprop.

The benefits of using an appropriate optimization scheme are substantial. For some tasks, the best schemes produce errors that are orders of magnitude lower than gradient descent for the same computational effort. This kind of improvement is the kind necessary to make neural networks more useful, for example, in the field of on-line adaptive control where data and time are precious.

Scaled conjugate gradient descent and quickprop are expedient optimization schemes. Using RBackprop for scaled conjugate gradient avoids an extra parameter and is computationally cheap.

A RBackprop

RBackprop is an algorithm for calculating the product of the hessian $H = \frac{\partial_2 e}{\partial w^2}$ (equation 17) and an arbitrary vector $\mathbf{v}$. The name of the algorithm comes from the notation used by Pearlmutter in his exposition of the technique (Pearlmutter, 1993). The algorithm is significant because not only does it give an exact result, but it is also computationally cheap, especially if the gradient has already been computed (as is the case, for example, in conjugate gradient descent). It can be used to find the exact Hessian for a network by taking multiple products of vectors that extract consecutive columns of the Hessian.

Let $\mathbf{w}$ be a point in weight space, r a small scalar and $\mathbf{v}$ an arbitrary vector in weight space. Then

$$\left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}+r\mathbf{v}} = \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}} + r \left.\frac{\partial_2 e}{\partial w^2}\right|_{\mathbf{w}} \mathbf{v} + \mathcal{O}\left(r^2\right) \tag{54}$$

where $\mathcal{O}\left(r^2\right)$ represents terms of order r^2 and higher.

Substituting $H = \left.\frac{\partial_2 e}{\partial w^2}\right|_{\mathbf{w}}$, rearranging and dividing by r gives:

$$H\mathbf{v} = \frac{\left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}+r\mathbf{v}} - \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}}}{r} + \mathcal{O}\left(r\right). \tag{55}$$

The exactness comes from taking the limit as $r \rightarrow 0$:

$$H\mathbf{v} = \lim_{r\rightarrow 0} \frac{\left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}+r\mathbf{v}} - \left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}}}{r} = \left.\frac{\partial}{\partial r}\left(\left.\frac{\partial e}{\partial w}\right|_{\mathbf{w}+r\mathbf{v}}\right)\right|_{r=0}. \tag{56}$$

The term on the right hand side is the key to the algorithm. Defining the following differential operator $\mathcal{R}_v\left\{.\right\}$, from which RBackprop gets its name:

$$\mathcal{R}_v\left\{f(w)\right\} = \frac{\partial}{\partial r} f(\mathbf{w} + r\mathbf{v})\Big|_{r=0} \tag{57}$$

gives us $H\mathbf{v} = \mathcal{R}_v\left\{\frac{\partial e}{\partial w}\right\}$. Thus applying the operator to the gradient produced by backprop gives the result we are looking for. Since the operator is a normal differential operator, it can be applied to all the steps in the backprop algorithm.

The algorithm is therefore as follows. In the equations below, the vector $\mathbf{v}$ to be multiplied by H has been indexed in the same way as the weight vector. The forward pass:

$$\mathcal{R}_v\left\{x_i\right\} = \sum_j (w_{ji}\mathcal{R}_v\left\{y_i\right\} + v_{ji}y_j) \tag{58}$$

$$\mathcal{R}_v\left\{y_i\right\} = \mathcal{R}_v\left\{x_i\right\}\sigma_i'(x_i) \tag{59}$$

and the backward pass:

$$\mathcal{R}_v\left\{\frac{\partial e}{\partial w_{ij}}\right\} = y_i\mathcal{R}_v\left\{\frac{\partial e}{\partial x_j}\right\} + \mathcal{R}_v\left\{y_i\right\}\frac{\partial e}{\partial x_j} \tag{60}$$

$$\mathcal{R}_v\left\{\frac{\partial e}{\partial x_i}\right\} = \sigma_i'(x_i)\mathcal{R}_v\left\{\frac{\partial e}{\partial y_i}\right\} + \mathcal{R}_v\left\{x_i\right\}\sigma_i''(x_i)\frac{\partial e}{\partial y_i} \tag{61}$$

$$\mathcal{R}_v\left\{\frac{\partial e}{\partial y_i}\right\} = \frac{d_2 e}{dy_i^2}\mathcal{R}_v\left\{y_i\right\} + \sum_j (w_{ij}\mathcal{R}_v\left\{\frac{\partial e}{\partial x_j}\right\} + v_{ij}\frac{\partial e}{\partial x_j}) \tag{62}$$

where $\frac{d_2 e}{dy_i^2}$ is the second direct derivative of the error with respect to the activation y_i. This is 1 for output nodes, and 0 other nodes.

The algorithm has a number of variables in common with the backprop calculation, a feature that can be used to speed up the calculation at a point in weight space for which the error gradient has already been found using backprop.

References

Barto, A., Sutton, R., and Anderson, C. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC–13:834–846.

Fahlman, S. E. (1988). An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, CMU.

Hertz, J. A., Palmer, R. G., and Krogh, A. S. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley, 350 Bridge Parkway, Redwood City, CA 94065, U.S.A.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1:295–307.

MacKay, D. (1991). *Bayesian Methods for Adaptive Models*. PhD thesis, California Institute of Technology, Pasadena, California.

Møller, M. (1993). A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533.

Pearlmutter, B. A. (1993). Fast exact multiplication by the hessian. to appear in *Neural Computation*.

Rumelhart, D. E., Hinton, G., and Williams, R. (1986). Learning internal representations by error propagation. In Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors, *Parallel Distributed Processing*, volume 1, pages 318–364. The MIT Press, Cambridge, Mass., U.S.A.

Rumelhart, D. E. and McClelland, J. L. (1986). *Parallel Distributed Processing*, volume 1. The MIT Press, Cambridge, Mass., U.S.A.

Schiffmann, W., Joost, M., and Werner, R. (1992). Optimization of the backpropagation algorithm for training multilayer perceptrons.

Thodberg, H. H. (1993). Ace of bayes: Application of neural networks with pruning. Technical report, The Danish Meat Research Institute, Maglegaardsvej 2, DK-4000 Roskilde, Denmark.

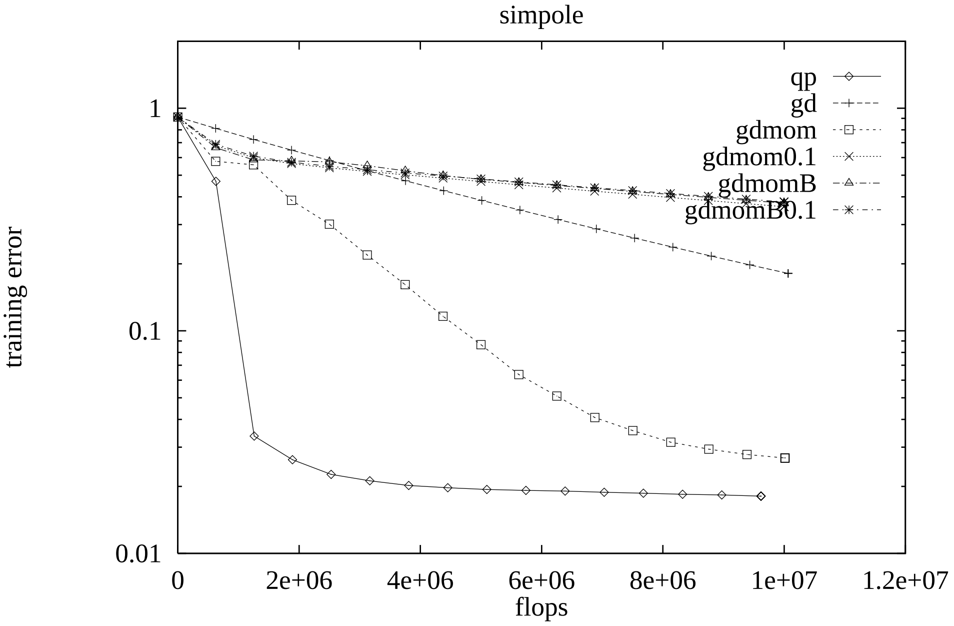


Figure 15: Gradient descent schemes training the default network on the simulated pole data. The runs are for 10^7 flops.

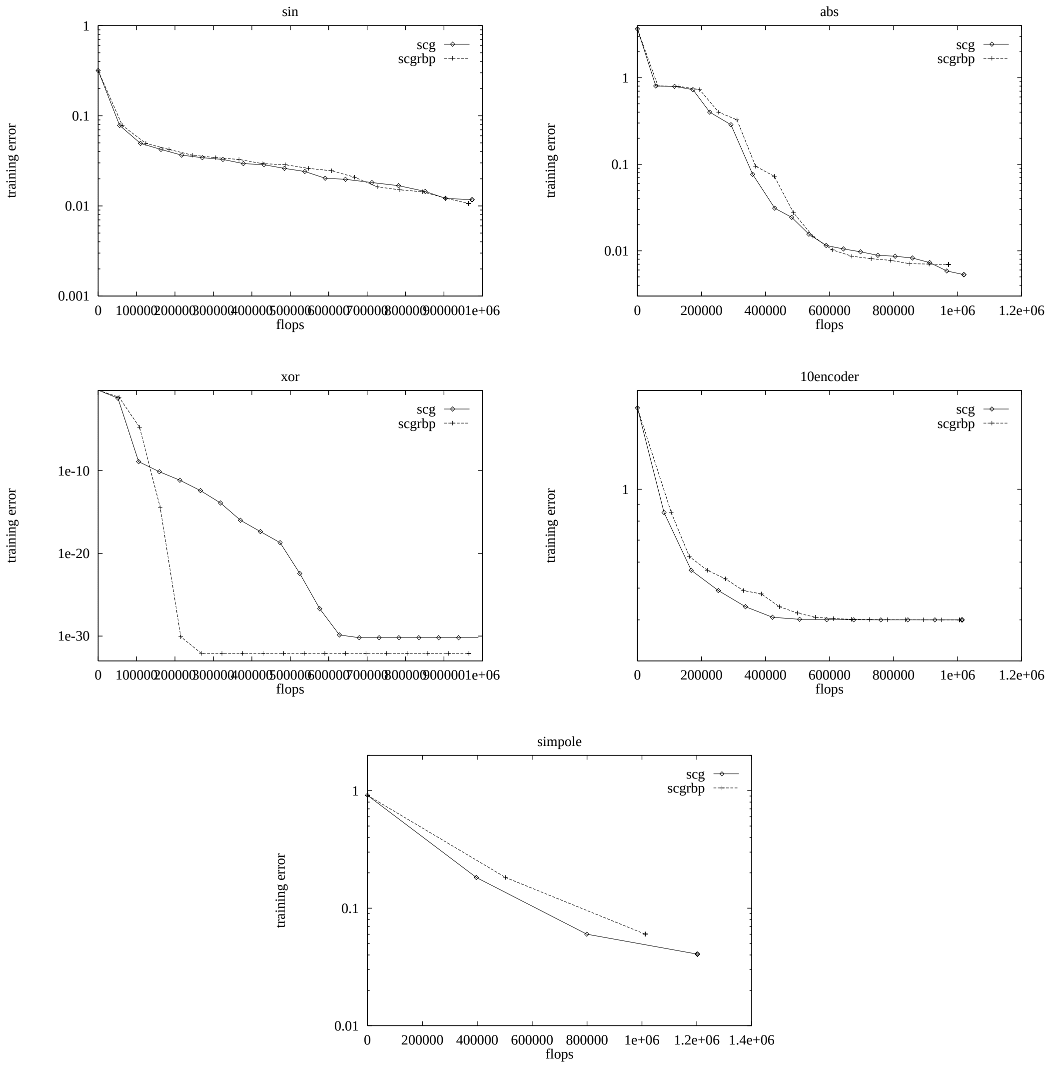


Figure 16: A comparison of scaled conjugate gradient (scg) with the original approximation used by Møller, and scg with RBackprop. The results of training the default network on each of the five tasks are shown.