Variations on Backpropagation

Batch Mode vs. On-Line Mode

- On-Line is also called “Stochastic”, referring to the \textit{random selection} of training samples.
- In On-Line, weight update is done after each sample.

- In Batch mode, weight update is done only after all samples have been applied.
- Equivalently, the gradient could be averaged for all samples, then used for the weight update.
- The would-be updates are accumulated, then averaged for the actual update.
- In batch mode, it is not necessary to use a random order, because all samples will be evaluated before any update.
Backprop Variations

• Heuristic Modifications
  – Variable Learning Rate
  – Momentum
  – Quickprop

• Classical Optimization
  – Conjugate Gradient
  – Newton’s Method
  – Levenberg-Marquardt Method

Common Theme

• Gradient Descent, as used in ordinary backpropagation, will find a minimum, but it can be very slow, especially for problems with a large number of weights.

• These techniques try to speed up the gradient descent algorithm.
Error Surface Example

Network Architecture

Nominal Function

Parameter Values

$$w^1_{1,1} = 10 \quad w^1_{2,1} = 10 \quad b^1_1 = -5 \quad b^1_2 = 5$$
$$w^2_{1,1} = 1 \quad w^2_{1,2} = 1 \quad b^2 = -1$$

MSE vs. $w^1_{1,1}$ and $w^2_{1,1}$ (holding other weights fixed)

superscript is the layer number
subscripts are neuron, input
(actual surface is 7-dimensional)
MSE vs. $w_{1,1}^1$ and $b_1^1$

MSE vs. $b_1^1$ and $b_2^1$
Different Projections Demo nnD12sd1

Backpropagation Convergence Example
Learning Rate Too Large

wide oscillations ("rattling")

Learning Rate Demo nnd12sd2
Momentum Backpropagation

Steepest Descent Backpropagation (SDBP)

\[
\Delta W^m(k) = -\alpha s^m (a^{m-1})^T \\
\Delta b^m(k) = -\alpha s^m
\]

Momentum Backpropagation (MOBP)

\[
\Delta W^m(k) = \gamma \Delta W^m(k-1) - (1 - \gamma) \alpha s^m (a^{m-1})^T \\
\Delta b^m(k) = \gamma \Delta b^m(k-1) - (1 - \gamma) \alpha s^m
\]

\( \gamma = 0.8 \)

\( \gamma = \text{Momentum} \)

Without vs. With Momentum

Without

With
Variable Learning Rate (VLBP)

\( \alpha = \text{Learning rate}, \gamma = \text{Momentum are both variable} \)

- There are three additional parameters: \( \zeta, \rho \) both between 0 and 1 and \( \eta > 1 \).
- Uses batch mode: weight updates are at the end of epoch.
- If \( \text{MSE decreases} \) after a weight update, then:
  - The weight update is accepted and
  - Learning rate \( \alpha \) is multiplied by factor \( \eta > 1 \).
  - If \( \gamma = 0 \), it is reset to its original value \( \gamma_0 \).
- If \( \text{MSE increases by less than} \ \zeta \) after a weight update, then the weight update is accepted, and learning rate \( \alpha \) and momentum \( \gamma \) are unchanged.
- If \( \text{MSE increases by more than} \ \zeta \) after a weight update, then:
  - The weight update is discarded,
  - Learning rate \( \alpha \) is multiplied by factor \( \rho (0 < \rho < 1) \), and
  - Momentum \( \gamma \) is set to 0.
VLBP Example

Straight Momentum
\( \gamma = 0.8 \)

VBLP \( \gamma_0 = 0.8 \)

MSE Increase Threshold \( \zeta = 4\% \)
Learning Rate Increase Factor \( \eta = 1.05 \)
Learning Rate Decrease Factor \( \rho = 0.7 \)

MSE \( W_{1,1} \)
Learning Rate \( W_{1,1} \)

Iteration Number

VLBP Demo

Network DESIGN Variable LR Backpropagation

Initial Learning Rate: 16.3
Increase Rate: 1.06
Decrease Rate: 0.7

Chapter 12
Conjugate Gradient Backprop

Rather than pure gradient descent,
controls **direction** of search by “conjugation”.

Uses gradient descent as a first step.

Conjugate Gradient Backpropagation

Weights are here denoted by vector $\mathbf{x}$. Subscript $k$ is step number.

weight change $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$

$p_k$ - Search Direction
$\alpha_k$ - Learning Rate

or

$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k$

$\alpha_k \mathbf{p}_k$

$\mathbf{x}_{k+1}$

$\mathbf{x}_k$

Direction, tempered by learning rate.
Pure Gradient Descent

Choose the next step so that the function decreases:

\[ F(x_{k+1}) < F(x_k) \]

For small changes in \( x \) we can approximate \( F(x) \):

\[ F(x_{k+1}) = F(x_k + \Delta x_k) = F(x_k) + g_k \Delta x_k \]

where

\[ g_k = \nabla F(x)|_{x=x_k} \]

If we want the function to decrease:

\[ g_k^T \Delta x_k = \alpha_k g_k^T p_k < 0 \]

(learning rate * gradient * direction)

We can maximize the decrease by choosing:

\[ p_k = -\nabla F(x) \]

(direction = neg. gradient)

\[ x_{k+1} = x_k - \alpha_k g_k \]

Example for an Analytic Function

\[ F(x) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1 \]

\[ x_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, \quad \alpha = 0.1 \]

\[ \nabla F(x) = \begin{bmatrix} \frac{\partial F(x)}{\partial x_1} \\ \frac{\partial F(x)}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \]

\[ g_0 = \nabla F(x)|_{x=x_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix} \]

\[ x_1 = x_0 - \alpha g_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.1 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix} \]

\[ x_2 = x_1 - \alpha g_1 = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix} - 0.1 \begin{bmatrix} 1.8 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.08 \end{bmatrix} \]
Pure Gradient Descent Plot

Note:
Lots of small steps toward end

Steepest Descent  Conjugate Gradient

CG is a batch-mode algorithm
Gradient is based on all samples.
CG Basic Idea

- Don’t take multiple small steps in roughly the same direction.
- Instead take big steps in roughly-orthogonal directions.
- “Conjugate” is the roughly-orthogonal aspect.

Conjugate vs. Orthogonal

Orthogonal Directions

Conjugate Directions

CG Accelerates by Minimizing MSE Function Along a Line

\( p_k \) is some chosen line direction, e.g., \(-g_k\) (negative gradient)

Compute learning rate \( \alpha_k \) to minimize \( F(x_k + \alpha_k p_k) \)

Take derivative wrt \( \alpha_k \) of multivariate Taylor expansion:

\[
\frac{d}{d\alpha_k} (F(x_k + \alpha_k p_k)) = \nabla F(x_k)^T p_k + \alpha_k \nabla^2 F(x_k) p_k
\]

**Set derivative to 0 and solve for \( \alpha_k \)**

\[
\alpha_k = -\frac{\nabla F(x_k)^T p_k}{p_k^T \nabla^2 F(x_k) p_k} = -\frac{g_k^T p_k}{p_k^T A_k p_k}
\]

where

\[
A_k = \nabla^2 F(x_k) |_{x_k = x_k}
\]

This is the **analytic** version, which assumes we know the Hessian, but we often don’t. Later, we show how to minimize by search.

---

2D Taylor’s Series Demo nnd8ts2
Hessian/Quadratic Demo nn8qf

Analytic 2D Example for Illustration Purposes

Quadratic function

\[ F(x) = \frac{1}{2}x'Ax + d'x + c \]

Starting Point

\[ x_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \]

Gradient

\[ \nabla F(x) = \begin{bmatrix} \frac{\partial F(x)}{\partial x_1} \\ \frac{\partial F(x)}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \]

Initial Direction

\[ p_0 = -\nabla F(x) \bigg|_{x = x_0} = \begin{bmatrix} -3 \\ -3 \end{bmatrix} \]

Solve

\[ \alpha_0 = -\begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix} = 0.2 \]

New Point

\[ x_1 = x_0 - \alpha_0 p_0 = \begin{bmatrix} 0.5 - 0.2 \cdot 3 \\ 0.5 - 0.2 \cdot 3 \end{bmatrix} = \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix} \]
CG does Successive Line Minimizations with different directions

How to choose directions?

Conjugate Vectors

\[ F(x) = \frac{1}{2} x^T A x + d^T x + c \]

A set of vectors \( p_j \) is mutually *conjugate* with respect to a positive definite Hessian matrix \( A \) provided

\[ p_i^T A p_j = 0 \quad k \neq j \]

*Conjugate means “orthogonal with respect to the above products”.*


**One set** of conjugate vectors consists of the eigenvectors of \( A \).

\[ z_i^T A z_j = \lambda_i \lambda_j z_i^T z_j = 0 \quad k \neq j \]

(The eigenvectors of symmetric matrices are orthogonal.)
For Quadratic Functions

\[ \nabla F(x) = Ax + d \]
\[ \nabla^2 F(x) = A \]

The change in the gradient at iteration \( k \) is
\[ \Delta g_k = g_{k+1} - g_k = (Ax_{k+1} + d) - (Ax_k + d) = A \Delta x_k \]

where
\[ \Delta x_k = (x_{k+1} - x_k) = \alpha_k p_k \]

The conjugacy conditions can then be rewritten
\[ \alpha_k p_k^T A p_j = (\Delta x_k)^T A p_j = \Delta g_k^T p_j = 0 \quad k \neq j \]

the last term not requiring knowledge of the Hessian matrix \( A \).

Forming Conjugate Directions

Choose the initial search direction as the negative of the gradient:
\[ p_0 = -g_0 \]

Choose subsequent search directions:
\[ p_k = -g_k + \beta_k p_{k-1} \]

where \( \beta_k \) is chosen according to one of these formulae:
\[ \beta_k = \frac{\Delta g_k^T g_k}{\Delta g_k^T p_{k-1}} \quad \text{or} \quad \beta_k = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}} \quad \text{or} \quad \beta_k = \frac{\Delta g_{k-1}^T g_k}{g_{k-1}^T g_{k-1}} \]

Hestenes & Steifel Formula
Fletcher-Reeves Formula
Polak & Ribiere Formula
Summary Conjugate Gradient Algorithm
Analytic Version (if A were known)

- The first search direction is the negative of the gradient.

\[ \mathbf{p}_0 = -\mathbf{g}_0 \]

- Select the learning rate to minimize along the line.

\[ \alpha_k = -\frac{\nabla F(\mathbf{x})^T_{\mathbf{x}=\mathbf{x}_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 F(\mathbf{x})_{\mathbf{x}=\mathbf{x}_k} \mathbf{p}_k} = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T A_k \mathbf{p}_k} \] (e.g. for quadratic functions only.)

- Select the next search direction using

\[ \mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1} \]  

\[ \beta_k \text{ is from previous slide} \]

- If the algorithm has not converged, return to second step.

- If the function were quadratic, it would be minimized in \( n \) steps, where \( n \) is the number of dimensions.

Problem: A is *not* known.
Previous Analytic Example, with Follow-On

Hessian

\[ F(x) = \frac{1}{2} x^T \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} x + \begin{bmatrix} 1 & 0 \end{bmatrix} x \quad x_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \]

\[ \nabla F(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(x) \\ \frac{\partial}{\partial x_2} F(x) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \quad p_0 = -g_0 = -\nabla F(x) \bigg|_{x = x_0} = \begin{bmatrix} -3 \\ -3 \end{bmatrix} \]

\[ \alpha_0 = -\begin{bmatrix} 3 & 3 \\ -3 & -3 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix} = 0.2 \quad x_1 = x_0 - \alpha_0 g_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \frac{2}{3} \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix} \]

Follow-On

Hessian

\[ g_1 = \nabla F(x) \bigg|_{x = x_1} = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.6 \end{bmatrix} \]

Fletcher-Reeves Formula

\[ \beta_1 = \frac{g^T_1 g_1}{g^T_0 g_0} = \frac{[0.6] [0.6]}{[3] [3]} = \frac{0.72}{18} = 0.04 \]

\[ p_1 = -g_1 + \beta_1 p_0 = \begin{bmatrix} -0.6 \\ 0.6 \end{bmatrix} + 0.04 \begin{bmatrix} -3 \\ 3 \end{bmatrix} = \begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix} \]

\[ \alpha_1 = -\begin{bmatrix} 0.6 & -0.6 \\ -0.72 & 0.48 \end{bmatrix} \begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix} = \begin{bmatrix} -0.72 \\ 0.576 \end{bmatrix} = 1.25 \]
Conjugate Gradient vs. Steepest Descent Plots

\[ x_2 = x_1 + \alpha_1 p_1 = \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix} + 1.25 \begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix} = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix} \]

Conjugate Gradient

Steepest Descent

Problem: A is not known.

Conjugate Gradient:
Line Minimization Searches
for General Functions
(not necessarily quadratic)
Line Search

- The purpose of line search is to locate a minimum along a specific direction.

- Generally we want to do this by evaluating the MSE at as few points as possible, for computational efficiency.
No Single Line Search is Best

Matlab NN Toolbox:

**Line Search Routines**

Several of the conjugate gradient and quasi-Newton algorithms require that a line search be performed. In this section we describe five different line searches which can be used. In order to use any of these search routines you simply set the training parameter `srchPen` equal to the name of the desired search function, as has been described in previous sections. It is often difficult to predict which of these routines will provide the best results for any given problem, but we have set the default search function to an appropriate initial choice for each training function, so you may never need to modify this parameter.

**Available Searches**

- Golden Section (`srchgol`)
- Brent’s (`srchbre`)
- Hybrid Bisection Cubic (`srchhyb`)
- Charalambous (`srchcha`)
- Backtracking (`srchbac`)

Example: Golden Section Search for Minimum

- **Part 1:** **Locate** an interval containing the minimum.
- **Part 2:** **Reduce** the interval’s width successively, until the interval is sufficiently small that we are close enough to the minimum.
Part 1: Interval Location to Bracket Minimum

Evaluate \( F(x_0 + n\varepsilon) \), for \( n \) successively doubling ...

Stop when the function increases. Minimum is bracketed. Proceed to Part 2.

Part 2: Interval Reduction: Divide & Conquer

What doesn’t help:
- Dividing in two parts

What does:
- Dividing in three parts

Bracketing interval not reduced.

\( c > d \Rightarrow \) minimum is between \( c \) and \( b \).
\( d > c \Rightarrow \) minimum is between \( a \) and \( d \).
Golden Section Search

- Rather than dividing interval into three *equal* parts, using two additional evaluations at each step,

- do it in such a way that one of the two evaluations will be reused at the next step,

- so that only *one additional* evaluation is required for each sub-dividing step.

Golden Section Search Logic

From the diagram above, it is seen that the **new search interval** will be either between \(x_1\) and \(x_4\) with a length of \(a+c\), or between \(x_2\) and \(x_3\) with a length of \(b\).

**The golden section search requires that these intervals be equal: \(b = a+c\).**

If they are not, a run of "bad luck" could lead to the wider interval being used many times, thus slowing down the rate of convergence.

To ensure that \(b = a+c\), the algorithm should choose \(x_4 = x_1 - x_2 + x_3\).
Golden Section Search Logic, continued

However there still remains the question of where $x_2$ should be placed in relation to $x_1$ and $x_3$. The golden section search chooses the spacing between these points in such a way that these points have the same proportion of spacing as the subsequent triple $x_1, x_2, x_4$ or $x_2, x_4, x_4$. By maintaining the same proportion of spacing throughout the algorithm, we avoid a situation in which $x_2$ is very close to $x_1$ or $x_3$, and guarantee that the interval width shrinks by the same constant proportion in each step.

Golden Section Search Logic, continued

To ensure that the spacing after evaluating $f(x_4)$ is proportional to the spacing prior to that evaluation, if $f(x_4)$ is $f_{4a}$ and our new triplet of points is $x_1, x_2, x_4$ then we want $c/a = a/b$. 
However, if \( f(x_4) = f_{ab} \) and our new triplet of points is \( x_2, x_4, \) and \( x_3 \) then we want \( c/(b-c) = a/b. \)

We want \( c/a = a/b \) and \( c/(b-c) = a/b, \) and need to solve for \( a/b. \)

Eliminating \( c \) from these two simultaneous equations yields:

\[
\left( \frac{b}{a} \right)^2 = \frac{b}{a} + 1
\]

or

\[
\frac{b}{a} = \varphi
\]

where \( \varphi \) is the golden ratio:

\[
\varphi = \frac{1 + \sqrt{5}}{2} = 1.618033988 \ldots
\]
Conjugate Gradient BP (CGBP) Demo

Intermediate Steps (showing line searches)

Complete Trajectory

Note semi-orthogonal direction changes
Methods Involving Solving Equations to get the Next Point

• Quickprop
• Newton’s Method
• Gauss-Newton Method
• Levenberg-Marquardt Method

Quickprop (QP, Qprop)
Scott Fahlman, CMU, 1988

• This is a batch-training method.

• Local optimization of backpropagation is based on using a **parabolic estimate** of the MSE is used to determine the weights for the next step.

• The focus of QP is on **updating a single weight at a time**.

• It is applied when, between two successive steps, the **gradient** has **decreased in magnitude** and has **changed sign**.
Quickprop, step k, in 1 dimension

Quickprop

- Assume J is parabolic as a function of w:
  \[ J(w) = aw^2 + bw + c \]

- First derivative of J is a line:
  \[ \frac{\partial J}{\partial w} = 2aw + b \]
  abbreviate \( \frac{\partial J}{\partial w} \) as \( J'(w) \).

- To find: value of \( w(k+1) \) such that \( J'(w(k+1)) = 0 \).
Quickprop

- To find: value of $w(k+1)$ such that $J'(w(k+1)) = 0$.

- We have
  
  $J'(w(k)) = 2aw(k) + b$
  
  $J'(w(k-1)) = 2aw(k-1) + b$

- Solving for $a$ and $b$ in terms of the other quantities:
  
  $2a = \left[ J'(w(k)) - J'(w(k-1)) \right] / \Delta w(k-1)$
  
  $b = J'(w(k)) - \left[ (J'(w(k)) - J'(w(k-1)))w(k) / \Delta w(k-1) \right]$

  where $\Delta w(k-1) = w(k) - w(k-1)$

Quickprop

- Set $J'(w(k+1)) = 0$, since we are looking for the parabolic minimum.

- Then $2a w(k+1) + b = 0$, i.e. $w(k+1) = -b/2a$.

- Substituting in previous equations, we get the weight update:

  $w(k+1) =$

  $w(k) + [J'(w(k)) \Delta w(k-1)] / [J'(w(k-1)) - J'(w(k))]$
Performance Comparisons

1997 A practical comparison between Quickprop and back-propagation
Sam Wangh* and Anthony Adams†
Artificial Neural Network Research Group
Department of Computer Science, University of Tasmania

2004 Quickprop Neural Network Short-Term Forecasting Framework for a Database Intrusion Prediction System

P. Ramaubramanian and A. Kannan
Department of Computer Science and Engineering
Anna University, Chennai 600025, Indin.

The idea is interesting, but uniform superiority of Quickprop over ordinary backpropagation (BP) has not been established. Nonetheless, related ideas were in subsequent improvements to BP.

Example


Globally Convergent Modification of the Quickprop Method

MICHAEL N. KRAHATIS1,3, GEORGE D. MAGOULAS2,3, and VASSILIS P. PLAGIANAKOS1,3
1Department of Mathematics, University of Patras, GR-261 10 Patras, Greece. 2Department of Informatics, University of Athens, GR-157 84 Athens, Greece. 3University of Patras Artificial Intelligence Research Center-UPAIRe.
e-mail: vrahatis@math.upatras.gr

In this Letter the convergence of the Qprop method has been considered. A modification of the classical Qprop algorithm has been presented and a strategy for alleviating the use of highly problem-dependent heuristic learning parameters that are necessary in order to secure the stability of the classical algorithm have been proposed. A new theorem that guarantees the convergence of the proposed modified Qprop has been proved. This modified Qprop scheme exhibits rapid convergence and provides stable learning and therefore, a greater possibility of good performance.
The Modification

QP based on Secant Methods

Instead of:

\[ w(k+1) = w(k) + \frac{\Delta w(k-1)}{J'(w(k-1)) - J'(w(k))} \]

Use:

\[ w(k+1) = w(k) + \frac{\Delta E(w(k))}{\| \Delta E(w(k)) \|^2} \]

Examples (Modified Qprop)
Newton’s Method for finding zero of a function

To find a zero:

The process is repeated until a sufficiently accurate value is reached:

\[ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \]

Newton’s Method for finding minimum of a function

To find a minimum, apply the method to the function’s derivative:

\[ x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}. \]
Newton’s Method

Use Taylor’s series approximation to $F$ about $x_k$

$$ F(x_{k+1}) = F(x_k + \Delta x_k) \approx F(x_k) + g_k^T \Delta x_k + \frac{1}{2} \Delta x_k^T A_k \Delta x_k $$

Set $F(x_{k+1}) - F(x_k) = 0$ and solve to find the minimum (more accurately, a stationary point):

$$ g_k + A_k \Delta x_k = 0 $$

$$ \Delta x_k = -A_k^{-1} g_k $$

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

**Iteration formula**

For the $x_i$’s are weights, not inputs)

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

**Hessian** = vector $2^{nd}$ derivative

$$ A_k = \nabla^2 F(x) \bigg|_{x = x_k} $$

**Gradient** = vector $1^{st}$ derivative

$$ g_k = \nabla F(x) \bigg|_{x = x_k} $$

For the SSE function, $v_i$ = error in $i^{th}$ sample:

$$ F(x) = \sum_{i=1}^{N} v_i^2(x) = v^T(x)v(x) $$

the $j^{th}$ element of the gradient is

$$ [\nabla F(x)]_j = \frac{\partial F(x)}{\partial x_j} = 2 \sum_{i=1}^{N} v_i(x) \frac{\partial v_i(x)}{\partial x_j} $$
Example

\[ F(x) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1 \]

\[ x_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \]

\[ \nabla F(x) = \begin{bmatrix} \frac{\partial F(x)}{\partial x_1} \\ \frac{\partial F(x)}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \]

\[ g_0 = \nabla F(x) \big|_{x=x_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix} \]

\[ A = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \]

\[ x_{k+1} = x_k - A_k^{-1} g_k \]

\[ x_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \frac{3}{3} \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}^{-1} \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 1 & -0.5 \\ -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 1.5 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix} \]

Plot of Newton’s Method for a Quadratic
Solve for minimum in one step
Non-Quadratic Case:
Use Quadratic Approximation Iteratively

\[ F(x) = (x_2 - x_1)^4 + 8x_1x_2 - x_1 + x_2 + 3 \]

Stationary Points:
\[ x^1 = \begin{bmatrix} -0.42 \\ 0.42 \end{bmatrix} \quad x^2 = \begin{bmatrix} -0.13 \\ 0.13 \end{bmatrix} \quad x^3 = \begin{bmatrix} 0.55 \\ -0.55 \end{bmatrix} \]

Different Initial Conditions
and Quadratic Approximations to Each
Using Matlab Nnet Toolbox for Backpropagation


load house_dataset

Loading this file creates two variables. The input matrix houseInputs consists of 506 column vectors of 13 real estate variables for 506 different houses. The target matrix houseTargets consists of the corresponding 506 relative valuations.

The next step is to create the network. The following call to feedforwardnet creates a two-layer network with 10 neurons in the hidden layer. (During the configuration step, the number of neurons in the output layer is set to one, which is the number of elements in each vector of targets.)

net = feedforwardnet;
net = configure(net,houseInputs,houseTargets);

Using Matlab Nnet Toolbox for Backpropagation

Example of setting network parameters and samples

```matlab
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net=newff(p,t,3,{},'trainbfg');
net.divideFcn = '';
net.performFcn = 'msereg';
net.performParam.ratio = 0.5;
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```
Matlab Nnet Toolbox Functions Handout

<table>
<thead>
<tr>
<th>Function name</th>
<th>Type</th>
<th>Description</th>
<th>No. of Parameters</th>
<th>Special Parameters</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>trainlm</td>
<td>Gradient descent</td>
<td>Full gradient descent</td>
<td>9</td>
<td>t, max_fad, max_grad</td>
<td></td>
</tr>
<tr>
<td>trainbptnet</td>
<td>Gradient descent</td>
<td>Momentum</td>
<td>10</td>
<td>t, leq</td>
<td></td>
</tr>
<tr>
<td>trainrp</td>
<td>Gradient descent</td>
<td>Resiliency learning rule</td>
<td>10</td>
<td>t, leq, t1, t2, t3, t4, max_decay</td>
<td></td>
</tr>
<tr>
<td>trainscg</td>
<td>Conjugate gradient</td>
<td>Powell-Broyden update</td>
<td>10</td>
<td>t, gnd, max_fad, max_grad, t1, t2, t3, t4, max_decay</td>
<td></td>
</tr>
<tr>
<td>traincg</td>
<td>Conjugate gradient</td>
<td>Hooke-Jeeves update</td>
<td>19</td>
<td>t, gnd, max_fad, max_grad, t1, t2, t3, t4, max_decay, x0, t0, t1, t2, t3, t4, max_decay</td>
<td></td>
</tr>
<tr>
<td>trainbr</td>
<td>Conjugate gradient</td>
<td>Powell-Broyden update</td>
<td>10</td>
<td>t, gnd, max_fad, max_grad, t1, t2, t3, t4, max_decay</td>
<td></td>
</tr>
<tr>
<td>trainoss</td>
<td>Gradient descent</td>
<td>One step descent</td>
<td>9</td>
<td>t, max</td>
<td></td>
</tr>
<tr>
<td>traingam</td>
<td>Gradient descent</td>
<td>Levenberg-Marquardt</td>
<td>20</td>
<td>t, gnd, max_fad, max_grad, t1, t2, t3, t4, max_decay, x0, t0, t1, t2, t3, t4, max_decay</td>
<td></td>
</tr>
<tr>
<td>trainmg</td>
<td>Gradient descent</td>
<td>Levenberg-Marquardt</td>
<td>13</td>
<td>t, gnd, max_fad, max_grad, t1, t2, t3, t4, max_decay, x0, t0, t1, t2, t3, t4, max_decay</td>
<td></td>
</tr>
<tr>
<td>trainscg</td>
<td>Conjugate gradient</td>
<td>Scall conjugate gradient</td>
<td>10</td>
<td>t, gnd, max_fad, max_grad, t1, t2, t3, t4, max_decay</td>
<td></td>
</tr>
<tr>
<td>trainbr</td>
<td>Conjugate gradient</td>
<td>Scall conjugate gradient</td>
<td>10</td>
<td>t, gnd, max_fad, max_grad, t1, t2, t3, t4, max_decay</td>
<td></td>
</tr>
</tbody>
</table>

Matlab Backpropagation Variants
Speed and Memory Comparisons


<table>
<thead>
<tr>
<th>Acronym</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM</td>
<td>trainlm - Levenberg-Marquardt</td>
</tr>
<tr>
<td>BFG</td>
<td>traiBFG - BFGS Quasi-Newton</td>
</tr>
<tr>
<td>RP</td>
<td>trainrp - Resiliency Backpropagation</td>
</tr>
<tr>
<td>SCG</td>
<td>trainscg - Scall Conjugate Gradient</td>
</tr>
<tr>
<td>CGB</td>
<td>traincgb - Conjugate Gradient with Powell/Beale Restarts</td>
</tr>
<tr>
<td>CGF</td>
<td>traincgm - Fletcher-Powell Conjugate Gradient</td>
</tr>
<tr>
<td>CGP</td>
<td>traincgp - Polak-Ribiere Conjugate Gradient</td>
</tr>
<tr>
<td>OSS</td>
<td>trainoss - One-Step Secant</td>
</tr>
<tr>
<td>GDX</td>
<td>traingdx - Variable Learning Rate Backpropagation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Title</th>
<th>Problem Type</th>
<th>Network Structure</th>
<th>Error Goal</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN</td>
<td>Function Approx.</td>
<td>1-5-1</td>
<td>0.002</td>
<td>Sun Sparc 2</td>
</tr>
<tr>
<td>PARITY</td>
<td>Pattern Recog.</td>
<td>5-10-10-1</td>
<td>0.001</td>
<td>Sun Sparc 2</td>
</tr>
<tr>
<td>ENGINE</td>
<td>Function Approx.</td>
<td>2-30-2</td>
<td>0.005</td>
<td>Sun Enterprise 4000</td>
</tr>
<tr>
<td>CANCER</td>
<td>Pattern Recog.</td>
<td>9-5-5-2</td>
<td>0.012</td>
<td>Sun Sparc 2</td>
</tr>
<tr>
<td>CHOLESTEROL</td>
<td>Function Approx.</td>
<td>21-15-3</td>
<td>0.027</td>
<td>Sun Sparc 20</td>
</tr>
<tr>
<td>DIABETES</td>
<td>Pattern Recog.</td>
<td>8-15-15-2</td>
<td>0.05</td>
<td>Sun Sparc 20</td>
</tr>
</tbody>
</table>
Comparisons for 3-bit parity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mean Time (s)</th>
<th>Ratio</th>
<th>Min. Time (s)</th>
<th>Max. Time (s)</th>
<th>Std. (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RP</td>
<td>3.73</td>
<td>1.00</td>
<td>2.35</td>
<td>6.89</td>
<td>1.26</td>
</tr>
<tr>
<td>SCG</td>
<td>4.09</td>
<td>1.10</td>
<td>2.36</td>
<td>7.48</td>
<td>1.56</td>
</tr>
<tr>
<td>CGP</td>
<td>5.13</td>
<td>1.38</td>
<td>3.50</td>
<td>8.73</td>
<td>1.05</td>
</tr>
<tr>
<td>CGB</td>
<td>5.30</td>
<td>1.42</td>
<td>3.91</td>
<td>11.59</td>
<td>1.35</td>
</tr>
<tr>
<td>CGF</td>
<td>6.62</td>
<td>1.77</td>
<td>3.96</td>
<td>28.05</td>
<td>4.32</td>
</tr>
<tr>
<td>OSS</td>
<td>8.00</td>
<td>2.14</td>
<td>5.06</td>
<td>14.41</td>
<td>1.92</td>
</tr>
<tr>
<td>LM</td>
<td>13.07</td>
<td>3.50</td>
<td>6.48</td>
<td>23.78</td>
<td>4.96</td>
</tr>
<tr>
<td>BFG</td>
<td>19.68</td>
<td>5.28</td>
<td>14.19</td>
<td>26.64</td>
<td>2.85</td>
</tr>
<tr>
<td>GDX</td>
<td>27.07</td>
<td>7.26</td>
<td>25.21</td>
<td>28.52</td>
<td>0.86</td>
</tr>
</tbody>
</table>

The network used for this problem is a 3-10-10-1 network with tanh neurons in each layer. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.001. The fastest algorithm for this problem is the resilient backpropagation algorithm, although the conjugate gradient algorithms (in particular, the scaled conjugate gradient algorithm) are almost as fast. Notice that the LM algorithm does not perform well on this problem. In general, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems. The LM algorithm is designed for least squares problems that are approximately linear. Since the output neurons in pattern recognition problems will generally be saturated, we will not be operating in the linear region.

Comparisons for Cancer Dataset

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mean Time (s)</th>
<th>Ratio</th>
<th>Min. Time (s)</th>
<th>Max. Time (s)</th>
<th>Std. (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGB</td>
<td>80.27</td>
<td>1.00</td>
<td>55.07</td>
<td>102.31</td>
<td>13.17</td>
</tr>
<tr>
<td>RP</td>
<td>83.41</td>
<td>1.04</td>
<td>59.51</td>
<td>109.39</td>
<td>13.44</td>
</tr>
<tr>
<td>SCG</td>
<td>86.58</td>
<td>1.08</td>
<td>41.21</td>
<td>112.19</td>
<td>18.25</td>
</tr>
<tr>
<td>CGP</td>
<td>87.30</td>
<td>1.09</td>
<td>56.35</td>
<td>116.37</td>
<td>18.03</td>
</tr>
<tr>
<td>CGB</td>
<td>110.06</td>
<td>1.37</td>
<td>63.33</td>
<td>171.63</td>
<td>30.13</td>
</tr>
<tr>
<td>LM</td>
<td>110.33</td>
<td>1.37</td>
<td>58.94</td>
<td>201.07</td>
<td>38.20</td>
</tr>
<tr>
<td>BFG</td>
<td>209.60</td>
<td>2.61</td>
<td>118.92</td>
<td>318.18</td>
<td>58.44</td>
</tr>
<tr>
<td>GDX</td>
<td>313.22</td>
<td>3.90</td>
<td>106.48</td>
<td>446.43</td>
<td>75.44</td>
</tr>
<tr>
<td>OSS</td>
<td>463.87</td>
<td>5.78</td>
<td>250.62</td>
<td>599.99</td>
<td>97.35</td>
</tr>
</tbody>
</table>

The fourth benchmark problem is a realistic pattern recognition (or nonlinear discriminant analysis) problem. The network used for this problem is a 9-5-5-2 network with tanh neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.012. A few runs failed to converge for some of the algorithms, so only the top 75% of the runs from each algorithm were used to obtain the statistics.

The conjugate gradient algorithms and resilient backpropagation all provide fast convergence, and the LM algorithm is also reasonably fast. As we mentioned with the parity data set, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems.
Comparisons for Engine Dataset

The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem is a 2-30-2 network with tanh neurons in the hidden layer and linear neurons in the output layer. Each entry in the table represents 30 different trials (10 trials for RP and GDX because of time constraints), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.005. The fastest algorithm for this problem is the LM algorithm, although the BFGS quasi-Newton algorithm and the conjugate gradient algorithms (the scaled conjugate gradient algorithm in particular) are almost as fast. Although this is a function approximation problem, the LM algorithm is not as clearly superior as it was on the SIN data set. In this case, the number of weights and biases in the network is much larger than the one used on the SIN problem (152 versus. 16), and the advantages of the LM algorithm decrease as the number of network parameters increases.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mean Time (s)</th>
<th>Ratio</th>
<th>Min. Time (s)</th>
<th>Max. Time (s)</th>
<th>Std. (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM</td>
<td>18.45</td>
<td>1.00</td>
<td>12.01</td>
<td>30.03</td>
<td>4.27</td>
</tr>
<tr>
<td>BFG</td>
<td>27.12</td>
<td>1.47</td>
<td>16.42</td>
<td>47.36</td>
<td>5.95</td>
</tr>
<tr>
<td>SCG</td>
<td>36.02</td>
<td>1.95</td>
<td>19.39</td>
<td>52.45</td>
<td>7.78</td>
</tr>
<tr>
<td>CFG</td>
<td>37.93</td>
<td>2.06</td>
<td>18.89</td>
<td>50.34</td>
<td>6.12</td>
</tr>
<tr>
<td>CGB</td>
<td>39.93</td>
<td>2.16</td>
<td>23.33</td>
<td>55.62</td>
<td>7.50</td>
</tr>
<tr>
<td>CGP</td>
<td>44.20</td>
<td>2.40</td>
<td>24.99</td>
<td>71.55</td>
<td>9.89</td>
</tr>
<tr>
<td>OSS</td>
<td>48.71</td>
<td>2.64</td>
<td>23.51</td>
<td>80.90</td>
<td>12.33</td>
</tr>
<tr>
<td>RP</td>
<td>65.91</td>
<td>3.57</td>
<td>31.83</td>
<td>134.31</td>
<td>34.24</td>
</tr>
<tr>
<td>GDX</td>
<td>188.50</td>
<td>10.22</td>
<td>81.59</td>
<td>279.90</td>
<td>66.67</td>
</tr>
</tbody>
</table>

The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem is a 2-30-2 network with tanh neurons in the hidden layer and linear neurons in the output layer. Each entry in the table represents 30 different trials (10 trials for RP and GDX because of time constraints), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.005. The fastest algorithm for this problem is the LM algorithm, although the BFGS quasi-Newton algorithm and the conjugate gradient algorithms (the scaled conjugate gradient algorithm in particular) are almost as fast. Although this is a function approximation problem, the LM algorithm is not as clearly superior as it was on the SIN data set. In this case, the number of weights and biases in the network is much larger than the one used on the SIN problem (152 versus. 16), and the advantages of the LM algorithm decrease as the number of network parameters increases.

Other Comparisons

Summary:
One of LM, RP, or CG is usually the best.
GDX (simple backprop with variable learning rate) is almost always the worst.

Levenberg-Marquardt Method
(blinks Newton’s method with
steepest descent)

Also a batch method
A very fast method for training
(but storage intensive)

The algorithm was first published in 1944 by Kenneth Levenberg, while working at
the Frankford Army Arsenal. It was rediscovered by Donald Marquardt who worked
as a statistician at DuPont and independently by Girard, Wynn and Morrison.

Matrix Form
The gradient can be written in matrix form as a
matrix-vector product with v as the vector
of errors on a per-sample basis as a function of weights x.

\[ \nabla F(x) = 2J^T(x)v(x) \]

where J is the Jacobian matrix of first derivatives of errors
based on weights x, for each sample:

\[ J(x) = \begin{bmatrix}
\frac{\partial v_1(x)}{\partial x_1} & \frac{\partial v_1(x)}{\partial x_2} & \cdots & \frac{\partial v_1(x)}{\partial x_n} \\
\frac{\partial v_2(x)}{\partial x_1} & \frac{\partial v_2(x)}{\partial x_2} & \cdots & \frac{\partial v_2(x)}{\partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial v_N(x)}{\partial x_1} & \frac{\partial v_N(x)}{\partial x_2} & \cdots & \frac{\partial v_N(x)}{\partial x_n}
\end{bmatrix} \]

v are the derivative values for each weight (col)
and row (sample)
Express the Hessian of F in terms of the Jacobian:

From \( \nabla F(x) = 2J^T(x)v(x) \)

Hessian entries:

\[
[\nabla^2 F(x)]_{k,j} = \frac{\partial^2 F(x)}{\partial x_k \partial x_j} = 2 \sum_{i=1}^{N} \left[ \frac{\partial v_i(x) \partial v_i(x)}{\partial x_k} \frac{\partial v_i(x)}{\partial x_j} + v_i(x) \frac{\partial^2 v_i(x)}{\partial x_k \partial x_j} \right]
\]

Hessian in matrix form:

\[
\nabla^2 F(x) = 2J^T(x)J(x) + 2S(x)
\]

(outer product)

where \( S(x) = \sum_{i=1}^{N} v_i(x)v_i(x)^T \)

---

**Gauss-Newton Method**

*Approximate* the Hessian matrix by *neglecting* \( S \):

\[
\nabla^2 F(x) \approx 2J^T(x)J(x) \quad \text{(outer product)}
\]

Newton’s method \( \Delta x_k = -A_k^{-1}g_k \)

then becomes:

\[
x_{k+1} = x_k - \left[2J^T(x_k)J(x_k)\right]^{-1}2J^T(x_k)v(x_k)
\]

= \[
x_k - \left[J^T(x_k)J(x_k)\right]^{-1}J^T(x_k)v(x_k)
\]

(Unlike Newton's method, the Gauss–Newton algorithm can *only* be used to minimize a sum of squared function values, but it has the advantage that second derivatives, which can be difficult to compute, are not required.)
Levenberg-Marquardt

Gauss-Newton method approximates the Hessian by:
\[ H = J^T J \]

This matrix may be singular, but can be made invertible as follows:
\[ G = H + \mu I \quad \text{for some } \mu > 0 \]

If the eigenvalues and eigenvectors of \( H \) are:
\[ \{ \lambda_1, \lambda_2, \ldots, \lambda_n \}, \quad \{ z_1, z_2, \ldots, z_n \} \]

thus
\[ Gz_i = [H + \mu I]z_i = Hz_i + \mu z_i = \lambda_i z_i + \mu z_i = (\lambda_i + \mu)z_i \]

Thus use
\[ x_{k+1} = x_k - [J^T(x_k)J(x_k) + \mu_k I]^{-1} J^T(x_k) v(x_k) \]

Varying \( \mu_k \)

As \( \mu_k \to 0 \), LM approaches pure Gauss-Newton.
\[ x_{k+1} = x_k - [J^T(x_k)J(x_k)]^{-1} J^T(x_k) v(x_k) \]

Converges fast in the vicinity of a minimum, but may diverge in general.

As \( \mu_k \to \infty \), LM approaches pure Gradient Descent with small learning rate \( 1/\mu_k \).
\[ x_{k+1} = x_k - \frac{1}{\mu_k} J^T(x_k) v(x_k) = x_k - \frac{1}{2\mu_k} \nabla F(x) \]

Generally converges, but may do so slowly.
Adjustment of $\mu_k$ in L-M Method

• Begin with a small $\mu_k$ to approximate Gauss-Newton.

• If the step does not yield a smaller $F(x)$, then repeat the step with a larger $\mu_k$, until $F(x)$ is decreased.

• $F(x)$ must decrease eventually, as for sufficiently large $\mu_k$ we will be taking a very small step in the Steepest Descent direction.

Application of LM to Multilayer Networks

The MSE for the multilayer, multi-output, network is:

$$F(x) = \sum_{q=1}^{Q} (t_q - a_q)^T (t_q - a_q) = \sum_{q=1}^{Q} e_q^T e_q = \sum_{q=1}^{Q} \sum_{j=1}^{S \mu} (e_{j,q})^2 = \sum_{i=1}^{N} (v_i)^2$$

The error vector is:

$$v^T = \left[ v_1 \quad v_2 \quad \ldots \quad v_N \right] = \left[ e_{1,1} \quad e_{2,1} \quad \ldots \quad e_{i,q} \quad e_{1,2} \quad \ldots \quad e_{S \mu,1} \right]$$

The weight vector (across all layers) is:

$$x^T = \left[ x_1 \quad x_2 \quad \ldots \quad x_n \right] = \left[ w_{1,1} \quad w_{1,2} \quad \ldots \quad w_{S \mu,1} \quad b_{1} \quad \ldots \quad b_{S \mu,1} \quad b_{S \mu,2} \quad \ldots \quad b_{S \mu,S \mu} \right]$$

The dimensions of the two vectors are:

$$N = Q \times S \mu \quad n = S \mu (R + 1) + S \mu (S \mu' + 1) + \cdots + \frac{1}{2} S \mu (S \mu' + 1)$$
Steepest descent computes squared-error terms of the form:

For the Jacobian \( J(x) \), computing the rows

\[
J(x) = \begin{bmatrix}
\frac{\partial e_{1,1}}{\partial w_{1,1}} & \frac{\partial e_{1,1}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{1,1}}{\partial w_{S,1}} & \frac{\partial e_{1,1}}{\partial b_1} \\
\frac{\partial e_{2,1}}{\partial w_{1,1}} & \frac{\partial e_{2,1}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{2,1}}{\partial w_{S,1}} & \frac{\partial e_{2,1}}{\partial b_1} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\frac{\partial e_{l,1}}{\partial w_{1,1}} & \frac{\partial e_{l,1}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{l,1}}{\partial w_{S,1}} & \frac{\partial e_{l,1}}{\partial b_1} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\frac{\partial e_{s,1}}{\partial w_{1,1}} & \frac{\partial e_{s,1}}{\partial w_{1,2}} & \cdots & \frac{\partial e_{s,1}}{\partial w_{S,1}} & \frac{\partial e_{s,1}}{\partial b_1}
\end{bmatrix}
\]

where each row corresponds to an input sample and is repeated for every sample.

Computing the Jacobian

Steepest descent computes squared-error terms of the form:

\[
\frac{\partial \hat{F}(x)}{\partial x_j} = \frac{\partial e_q}{\partial x_j}
\]

using the chain rule:

\[
\frac{\partial \hat{F}}{\partial w_{m,i,j}} = \frac{\partial \hat{F}}{\partial \mathbf{w}_m} \times \frac{\partial \mathbf{w}_m}{\partial w_{i,j}}
\]

where the sensitivity

\[
s_{i,j}^m = \frac{\partial F}{\partial n_{i,j}^m}
\]

is computed using backpropagation.

For the Jacobian we need to compute terms of the form:

\[
[J]_{h,i} = \frac{\partial y_h}{\partial x_j} = \frac{\partial e_{k,q}}{\partial x_l}
\]
Marquardt Sensitivity

Define the Marquardt sensitivity (q is the sample’s index):

\[ s_{i,h}^m = \frac{\partial \hat{e}_{i,q}}{\partial n_{i,q}^m} = \frac{\partial e_{i,q}}{\partial n_{i,q}^m} \]

\[ h = (q - 1)S^M + k \]

then compute the Jacobian as follows:

**weight**

\[ [J]_{h,t} = \frac{\partial \hat{e}_k}{\partial x_h} = \frac{\partial e_{k,q}}{\partial w_{i,j}^m} \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} \frac{\partial s_i}{\partial n_{i,q}^m} = s_i^m \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = s_i^m \frac{\partial n_{i,q}^m}{\partial w_{i,j}^m} = s_i^m \]

**bias**

\[ [J]_{h,t} = \frac{\partial \hat{e}_k}{\partial x_h} = \frac{\partial e_{k,q}}{\partial b_i^m} \frac{\partial n_{i,q}^m}{\partial b_i^m} \frac{\partial s_i}{\partial n_{i,q}^m} = s_i^m \frac{\partial n_{i,q}^m}{\partial b_i^m} = s_i^m \]

---

Computing the Sensitivities

**Initialization**

\[ \tilde{s}_{i,h}^M = \frac{\partial \hat{e}_{i,q}}{\partial n_{i,q}^M} = \frac{\partial e_{i,q}}{\partial n_{i,q}^M} \]

\[ \frac{\partial (A q - A q - k)}{\partial n_{i,q}^M} = \frac{\partial e_{i,q}}{\partial n_{i,q}^M} \]

\[ \tilde{s}_{i,h}^M = \begin{cases} -f^M_{(n_{i,q}^M)} & \text{for } i = k \\ 0 & \text{for } i \neq k \end{cases} \]

\[ \tilde{S}_q^M = -F^M(n_q^M) \]

**Backpropagation in Matrix Form**

\[ \tilde{S}_q^m = F^m(n_q^m)(W^{m+1})^T \tilde{S}_{q}^{m+1} \]

\[ \tilde{S}_q^m = [\tilde{S}_1^m \tilde{S}_2^m \cdots \tilde{S}_Q^m] \quad Q = \text{number of samples} \]
Levenberg-Marquardt Backpropagation Summary

1. Present all inputs to the network and compute the corresponding network outputs and the errors. Compute the sum of squared errors over all inputs.

2. Compute the Jacobian matrix: Calculate the sensitivities with the backpropagation algorithm, after initializing. Augment the individual matrices into the Marquardt sensitivities. Compute the elements of the Jacobian matrix using previous equations.

3. Invert the matrix in the following equation
   \[ x_{k+1} = x_k - \left[ J^T(x_k)J(x_k) + \mu_k I \right]^{-1} J^T(x_k)v(x_k) \]
   to obtain the weight updates.

4. Recompute the sum of squared errors with the new weights. If this new sum of squares is smaller than that computed in step 1, then divide \( \mu_k \) (mu) by \( \nu \) (nu), update the weights and go back to step 1. If the sum of squares is not reduced, then multiply \( \mu_k \) by \( \nu \) and go back to step 3.

Example LMBP 1 Step
LMBP Trajectory

Demos nnd12ms and nnd12m
A Few Examples of LM Training for NN Applications

Levenberg-Marquardt Algorithm for Karachi Stock Exchange Share Rates Forecasting

Syed Muhammad Aqil Burney, Tabheen Ahmed Jilani, Cemal Ardil

Quick and reliable diagnosis of stomach cancer by artificial neural network

Saeid Afsar1, Fahime Abdolrahmani2, Fereshteh vakili tanha2, Mahin Zohdi seifi2, Kobra Taheri2
1Department of biophysics and biochemistry, faculty of science, Tarbiat Modares University,
Tehran, Iran
2IPNU, Hamadan, Iran
s.programers@gmail.com

LM Drawback

• Because inversion of a relatively large matrix is required, the problem size is probably limited to a few hundred points.

• There is a parameter mem_reduc which is an integer factor for trainlm to use less memory, by breaking the training cycle into pieces.
Rprop (Resilient Backpropagation)

From Wikipedia, the free encyclopedia:

Rprop, short for resilient backpropagation, is ... a first-order optimization algorithm. This algorithm was created by Martin Riedmiller and Heinrich Braun in 1992.

Rprop takes into account only the sign of the partial derivative over all patterns (not the magnitude), and acts independently on each weight. For each weight, if there was a sign change of the partial derivative of the total error function compared to the last iteration, the update value for that weight is multiplied by a factor \( \eta^- \), where \( \eta^- < 1 \).

If the last iteration produced the same sign, the update value is multiplied by a factor of \( \eta^+ \), where \( \eta^+ > 1 \). The update values are calculated for each weight in the above manner, and finally each weight is changed by its own update value, in the opposite direction of that weight's partial derivative, so as to minimize the total error function.

\( \eta^+ \) is empirically set to 1.2 and \( \eta^- \) to 0.5.

Next to the cascade correlation algorithm and the Levenberg–Marquardt algorithm, Rprop is one of the fastest weight update mechanisms.

RPROP is a batch update algorithm.

Rprop Theory, 1994, part 1

The basic principle of Rprop is to eliminate the harmful influence of the size of the partial derivative on the weight step. As a consequence, only the sign of the derivative is considered to indicate the direction of the weight update. The size of the weight change is exclusively determined by a weight-specific, so-called 'update-value' \( \Delta_{ij}^{(t)} \):

\[
\Delta w_{ij}^{(t)} = \begin{cases} 
-\Delta_{ij}^{(t)}, & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\
+\Delta_{ij}^{(t)}, & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\
0, & \text{else} 
\end{cases}
\]  

(1)

where \( \frac{\partial E}{\partial w_{ij}}^{(t)} \) denotes the summed gradient information over all patterns of the pattern set ('batch learning').

It should be noted, that by replacing the \( \Delta_{ij}^{(t)} \) by a constant update-value \( \Delta \), equation (1) yields the so-called 'Manhattan'-update rule.
Rprop Implementation, 1994, part 2

The second step of Rprop learning is to determine the new update values $\Delta_{ij}(t)$. This is based on a sign-dependent adaptation process, similar to the learning-rate adaptation in [4], [5].

$$\Delta_{ij}^{(t)} = \begin{cases} 
\eta^+ \cdot \Delta_{ij}^{(t-1)}, & \text{if } \frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) > 0 \\
\eta^- \cdot \Delta_{ij}^{(t-1)}, & \text{if } \frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) < 0 \\
\Delta_{ij}^{(t-1)}, & \text{else}
\end{cases} \quad (2)$$

where $0 < \eta^- < 1 < \eta^+$

Rprop Algorithm

\[
\forall i, j : \Delta_{ij}(t) = \Delta_{ij}
\]
\[
\forall i, j : \frac{\partial E}{\partial w_{ij}}(t-1) = 0
\]

Repeat

Compute Gradient $\frac{\partial E}{\partial w_{ij}}(t)$

For all weights and biases:

if $(\frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) > 0)$ then

$\Delta_{w_{ij}}(t) = \min(\Delta_{w_{ij}}(t-1) \cdot \eta^+ \cdot \Delta_{w_{ij}}(t))$

$w_{ij}(t+1) = w_{ij}(t) + \Delta_{w_{ij}}(t)$

else if $(\frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) < 0)$ then

$\Delta_{w_{ij}}(t) = \max(\Delta_{w_{ij}}(t-1) \cdot \eta^- \cdot \Delta_{w_{ij}}(t))$

$w_{ij}(t+1) = w_{ij}(t) + \Delta_{w_{ij}}(t)$

else if $(\frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) = 0)$ then

$\Delta_{w_{ij}}(t) = -\text{sign} \left( \frac{\partial E}{\partial w_{ij}}(t) \right) \cdot \Delta_{w_{ij}}(t)$

$w_{ij}(t+1) = w_{ij}(t) + \Delta_{w_{ij}}(t)$

\]

Until (converged)

\[
\text{sign}(x) = x > 0 \? 1 : x < 0 \? -1 : 0
\]
Rprop Parameters

The Rprop algorithm takes two parameters: the initial update-value $\Delta_0$ and a limit for the maximum step size, $\Delta_{\text{max}}$.

When learning starts, all update-values are set to an initial value $\Delta_0$. Since $\Delta_0$ directly determines the size of the first weight step, it should be chosen according to the initial values of the weights themselves, for example $\Delta_0 = 0.1$ (default setting). The choice of this value is rather uncritical, for it is adapted as learning proceeds.

In order to prevent the weights from becoming too large, the maximum weight-step determined by the size of the update-value, is limited. The upper bound is set by the second parameter of Rprop, $\Delta_{\text{max}}$. The default upper bound is set somewhat arbitrarily to $\Delta_{\text{max}} = 50.0$. Usually, convergence is rather insensitive to this parameter as well. Nevertheless, for some problems it can be advantageous to allow only very cautious (namely small) steps, in order to prevent the algorithm getting stuck too quickly in suboptimal local minima. The minimum step size is constantly fixed to $\Delta_{\min} = 1e^{-8}$.

Another Set of Comparisons

A 1-10-1 network was trained on a data set with 41 input/output pairs until a mean square error performance of 0.01 was obtained.

<table>
<thead>
<tr>
<th>Function</th>
<th>Technique</th>
<th>Time</th>
<th>Epochs</th>
<th>Mflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>traindx</td>
<td>Variable Learning Rate</td>
<td>57.71</td>
<td>980</td>
<td>2.50</td>
</tr>
<tr>
<td>trainrp</td>
<td><strong>Rprop</strong></td>
<td>12.95</td>
<td>185</td>
<td>0.56</td>
</tr>
<tr>
<td>trainscg</td>
<td>Scaled Conj. Grad.</td>
<td>16.06</td>
<td>106</td>
<td>0.70</td>
</tr>
<tr>
<td>traincgf</td>
<td>Fletcher-Powell CG</td>
<td>16.40</td>
<td>81</td>
<td>0.99</td>
</tr>
<tr>
<td>traincgp</td>
<td>Polak-Ribiére CG</td>
<td>19.16</td>
<td>89</td>
<td>0.75</td>
</tr>
<tr>
<td>traincgb</td>
<td>Powell-Beale CG</td>
<td>15.03</td>
<td>74</td>
<td>0.59</td>
</tr>
<tr>
<td>trainoss</td>
<td>One-Step-Secant</td>
<td>18.46</td>
<td>101</td>
<td>0.75</td>
</tr>
<tr>
<td>trainbfg</td>
<td>BFGS quasi-Newton</td>
<td>10.86</td>
<td>44</td>
<td>1.02</td>
</tr>
<tr>
<td>trainlm</td>
<td>Levenberg-Marquardt</td>
<td>1.87</td>
<td>6</td>
<td>0.46</td>
</tr>
</tbody>
</table>