Variations on Backpropagation

Backprop Variations

- Heuristic Modifications
  - Momentum
  - Variable Learning Rate
  - Quickprop

- Classical Optimization
  - Conjugate Gradient
  - Newton's Method
  - Levenberg-Marquardt Method

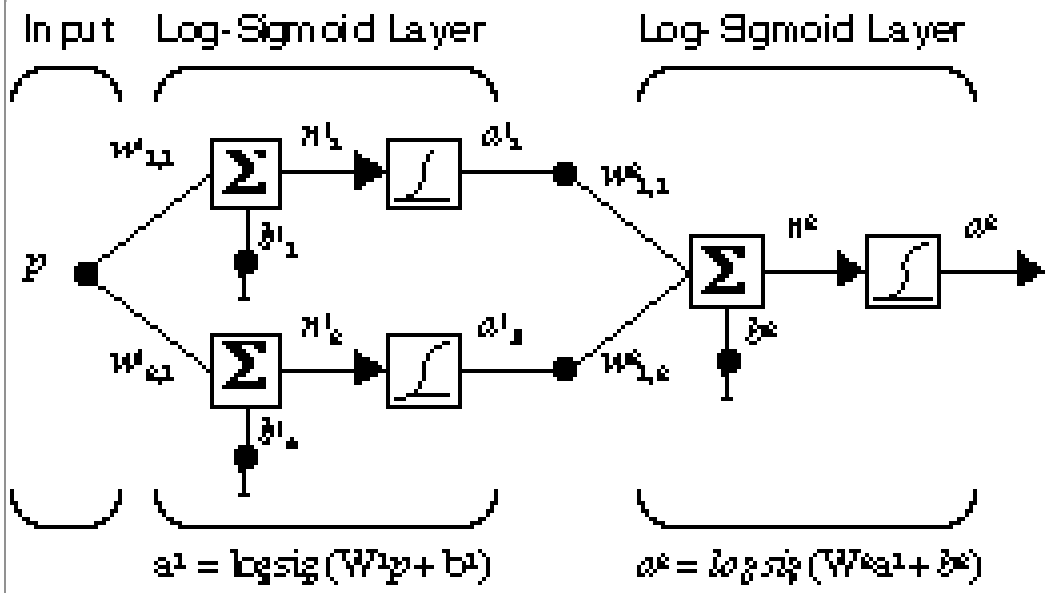
Common Theme

- 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.

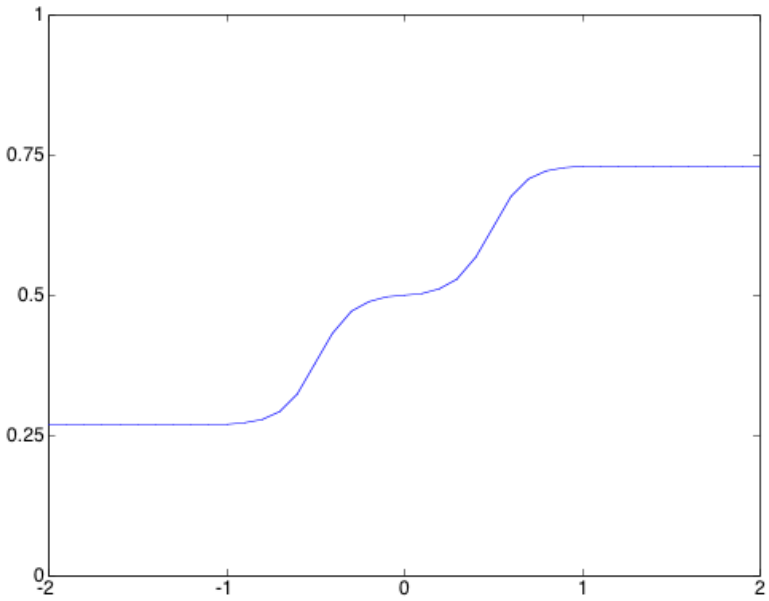
- These techniques try to speed up the gradient descent algorithm.

Error Surface Example

Network Architecture



Nominal Function

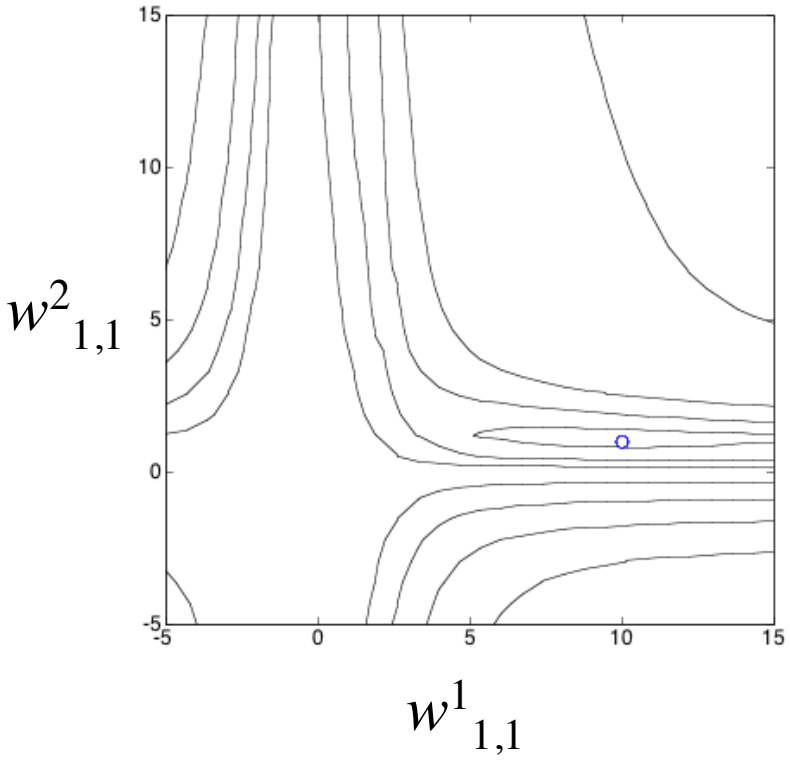


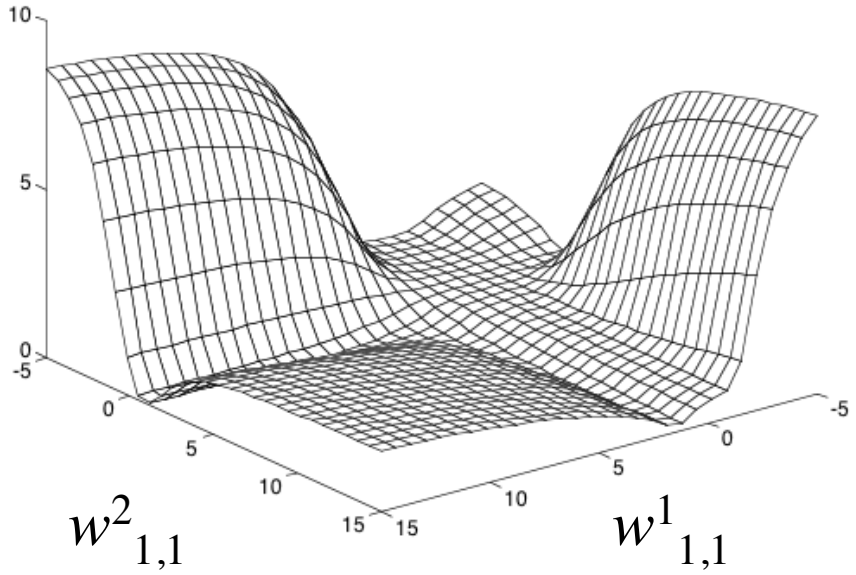
Parameter Values

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

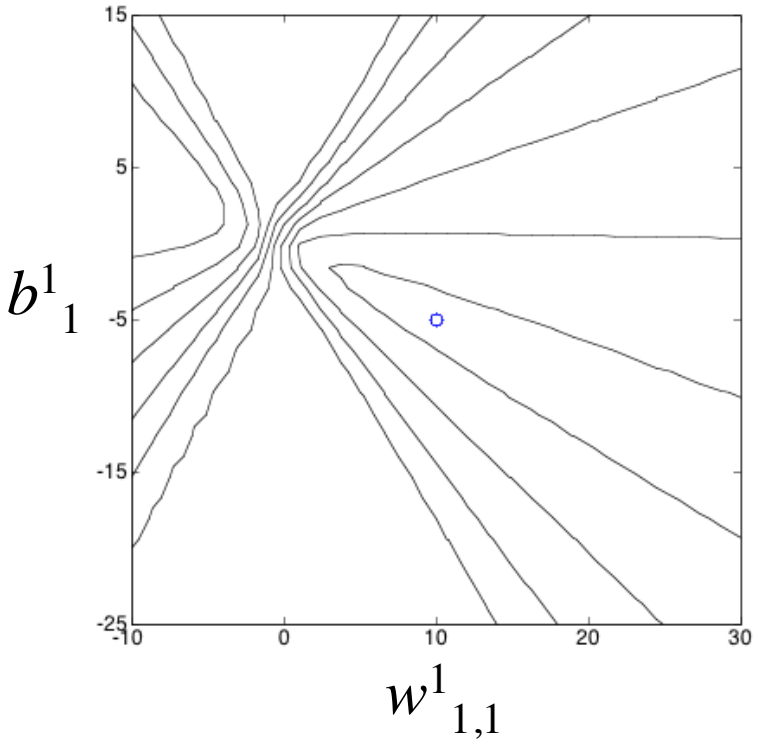
$$w^2_{1,1} = 1 \qquad w^2_{1,2} = 1 \qquad b^2 = -1$$

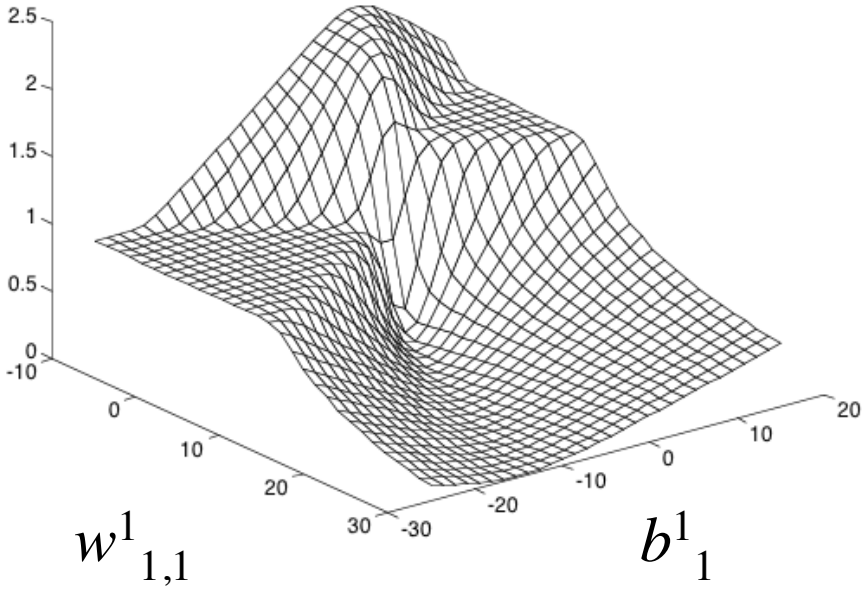
MSE vs. $w^1_{1,1}$ and $w^2_{1,1}$



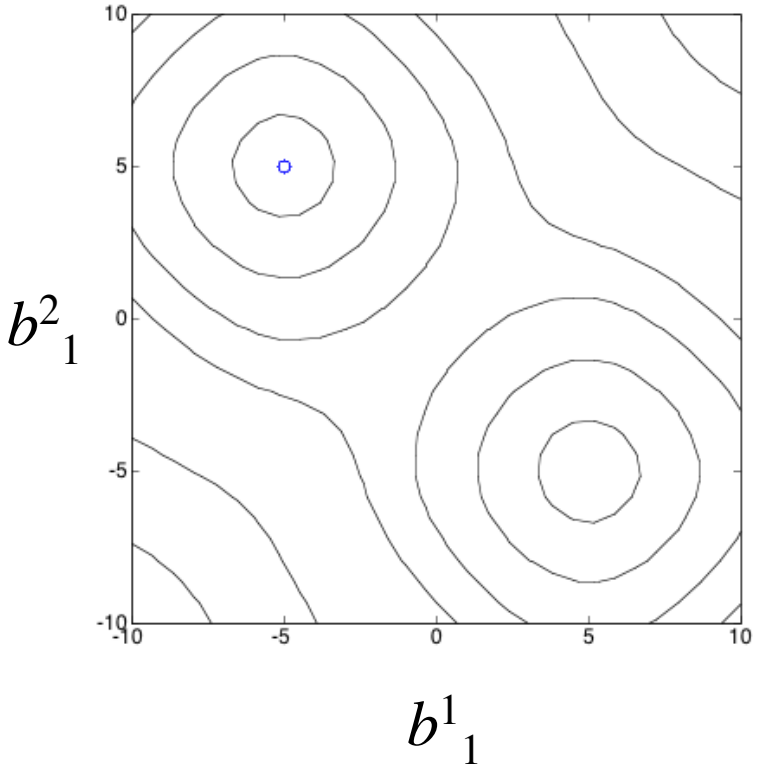


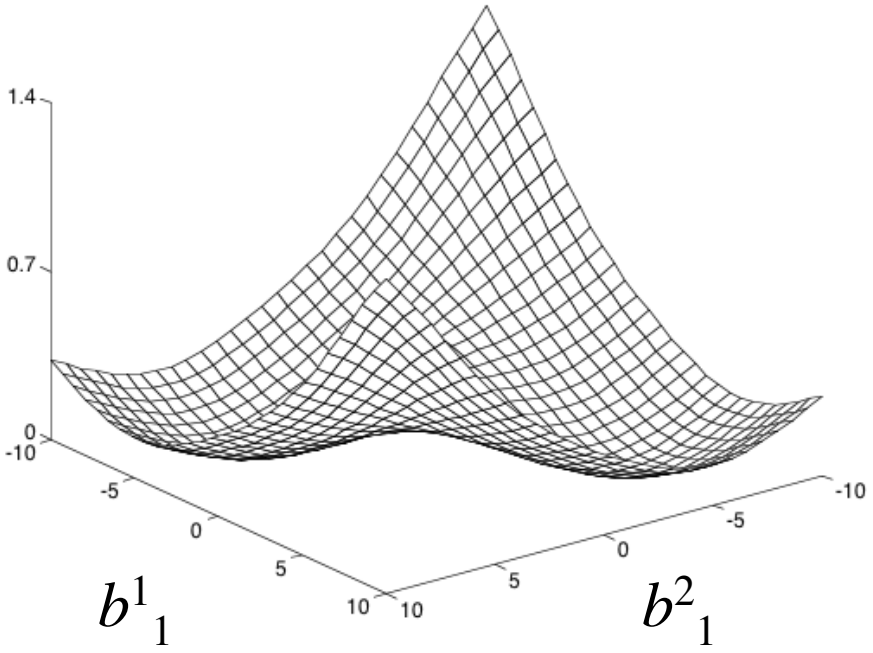
Squared Error vs. $w^1_{1,1}$ and b^1_1



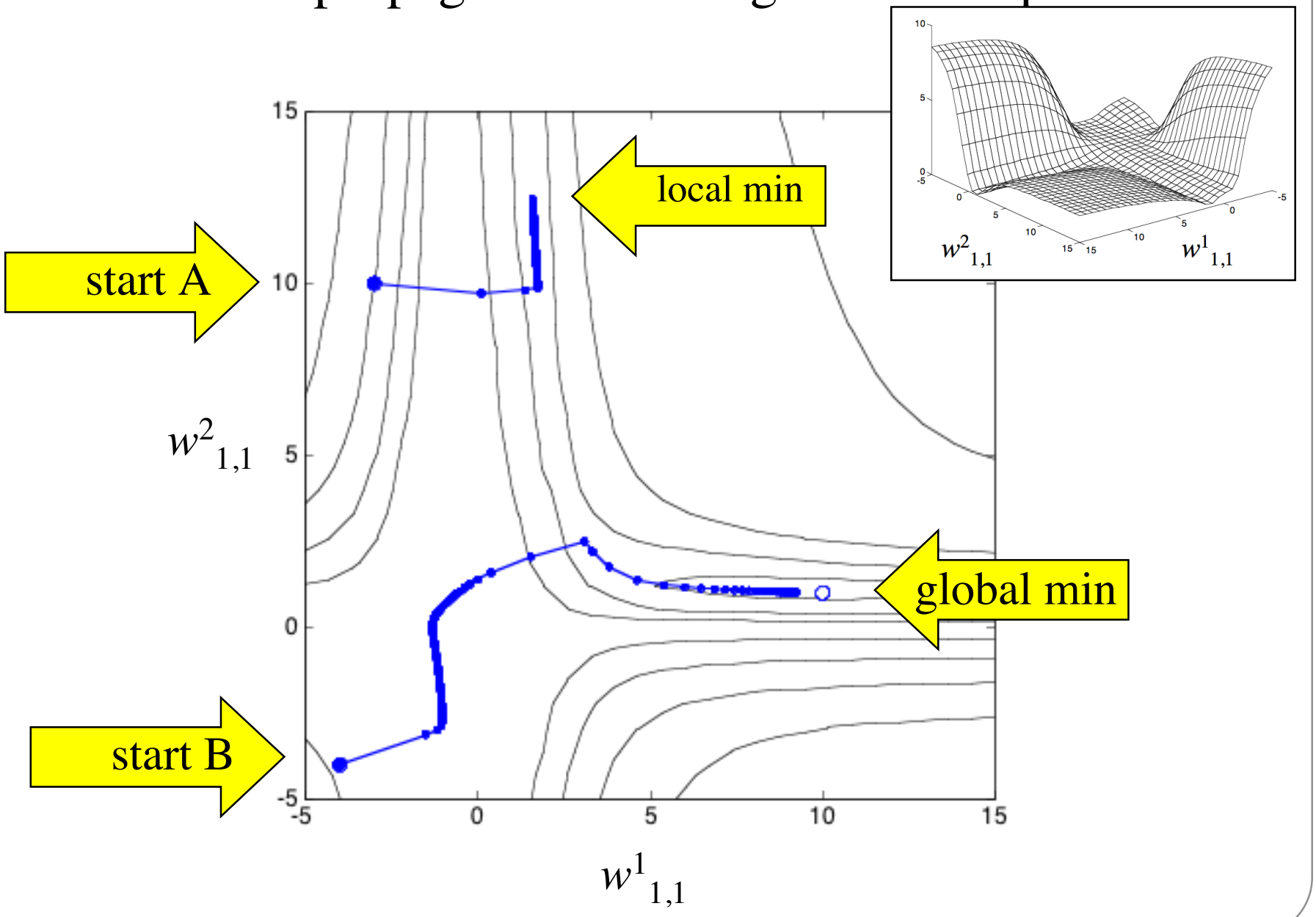


Squared Error vs. b^1_1 and b^1_2

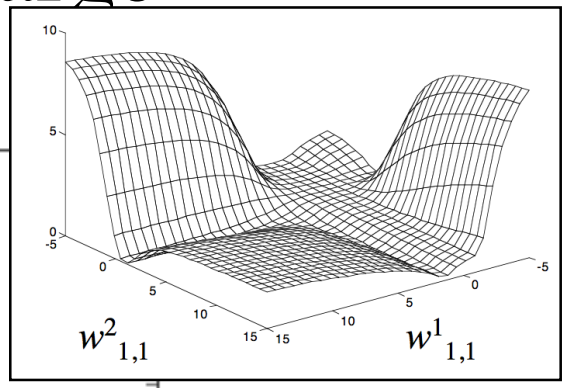




Backpropagation Convergence Example

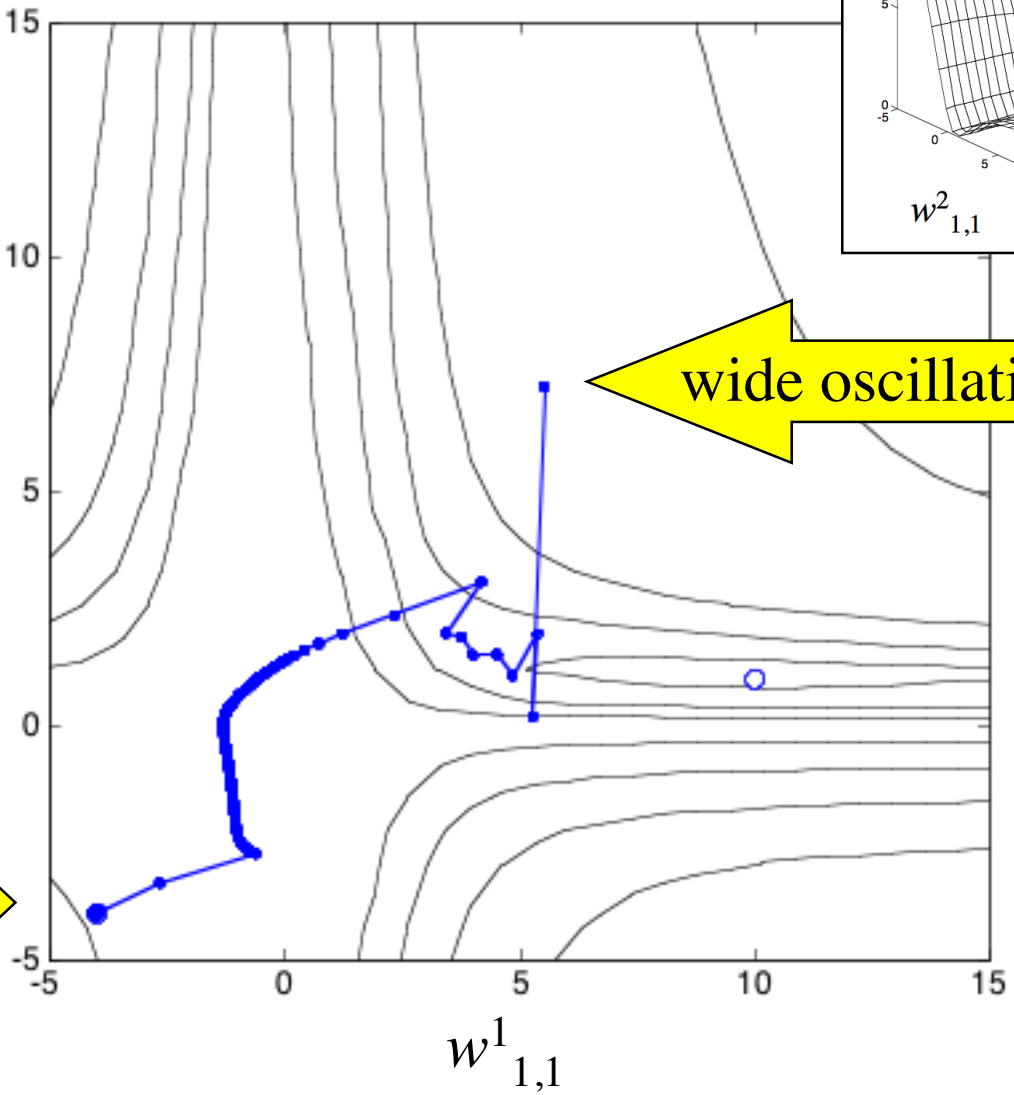


Learning Rate Too Large



wide oscillations ("rattling")

start B



Momentum Backpropagation

Steepest Descent Backpropagation (SDBP)

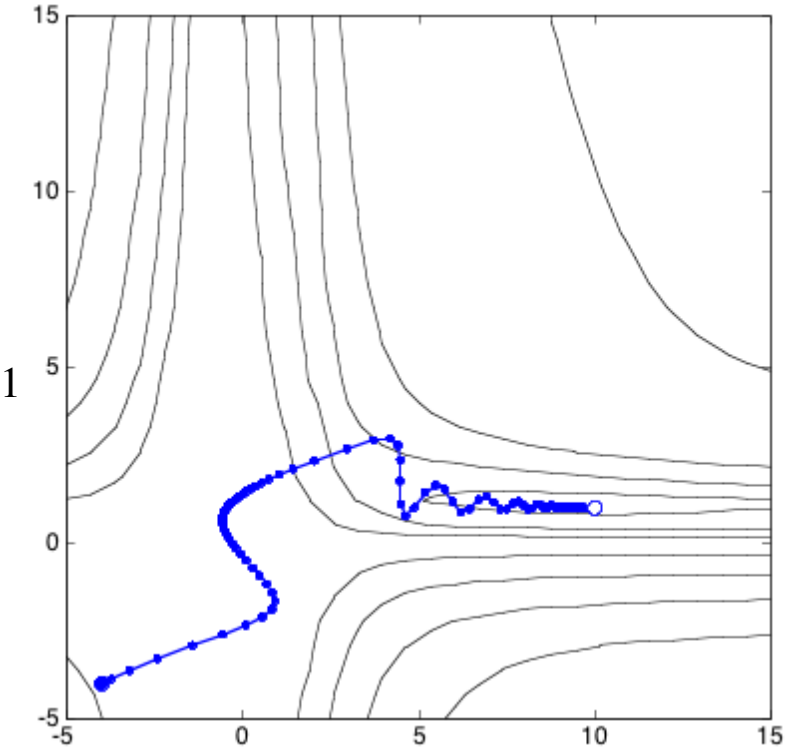
$$\Delta \mathbf{W}^m(k) = -\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

$$\Delta \mathbf{b}^m(k) = -\alpha \mathbf{s}^m$$

Momentum Backpropagation (MOBP)

$$\Delta \mathbf{W}^m(k) = \gamma \Delta \mathbf{W}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

$$\Delta \mathbf{b}^m(k) = \gamma \Delta \mathbf{b}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m$$



$\gamma = 0.8$

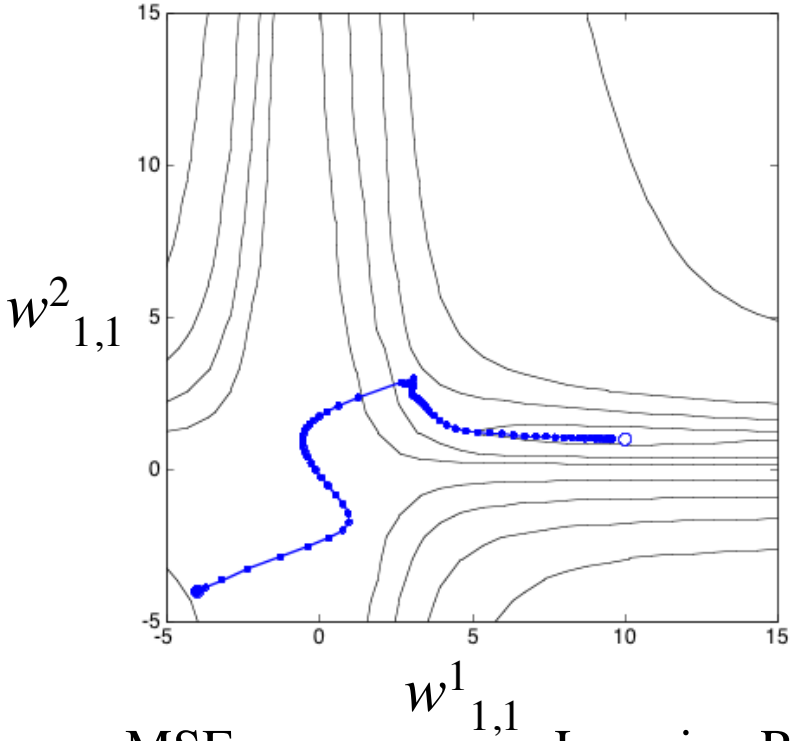
γ = Momentum

Variable Learning Rate (VLBP)

α = Learning rate, γ = Momentum are both variable

- There are three additional parameters: ζ, ρ both between 0 and 1 and $\eta>1$.
- Batch mode: weight updates are at the end of epoch.
- If **MSE increases by less than** ζ, then the weight update is accepted, but α and γ are unchanged.
- If **MSE increases by more than** ζ after a weight update, then:
  - the weight update is discarded,
  - α **is multiplied** by some factor ($0 < \rho < 1$), and
  - γ is set to zero.
- If **MSE decreases** after a weight update, then:
  - the weight update is accepted and
  - α **is multiplied** by factor $\eta>1$.
  - If γ has been previously set to zero, it is reset to its original value.

Example



MSE Increase Threshold

$\zeta = 4\%$

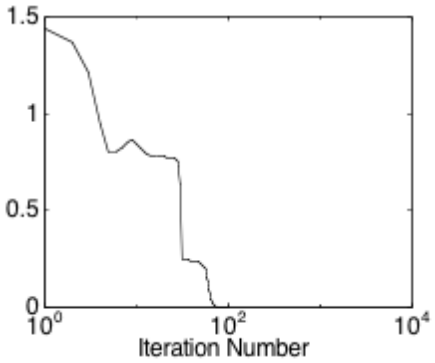
Learning Rate Increase Factor

$\eta = 1.05$

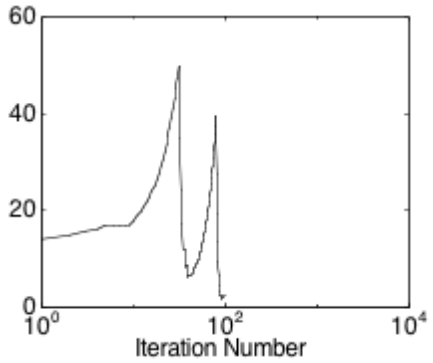
Learning Rate Decrease Factor

$\rho = 0.7$

MSE



Learning Rate

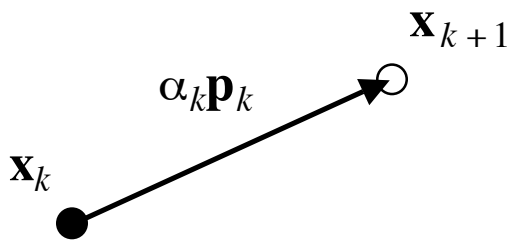


Toward Conjugate Gradient Optimization

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

or

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



$\mathbf{p}_k$ - Search Direction

α_k - Learning Rate

Steepest Descent (Gradient Descent)

Choose the next step so that the function decreases:

$$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$$

For small changes in $\mathbf{x}$ we can approximate $F(\mathbf{x})$:

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k$$

where

$$\mathbf{g}_k \equiv \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k}$$

If we want the function to decrease:

$$\mathbf{g}_k^T \Delta\mathbf{x}_k = \alpha_k \mathbf{g}_k^T \mathbf{p}_k < 0$$

(learning rate * gradient * direction)

We can maximize the decrease by choosing:

$$\mathbf{p}_k = -\mathbf{g}_k$$

(direction = neg. gradient)

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

Example for an Analytic Function

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

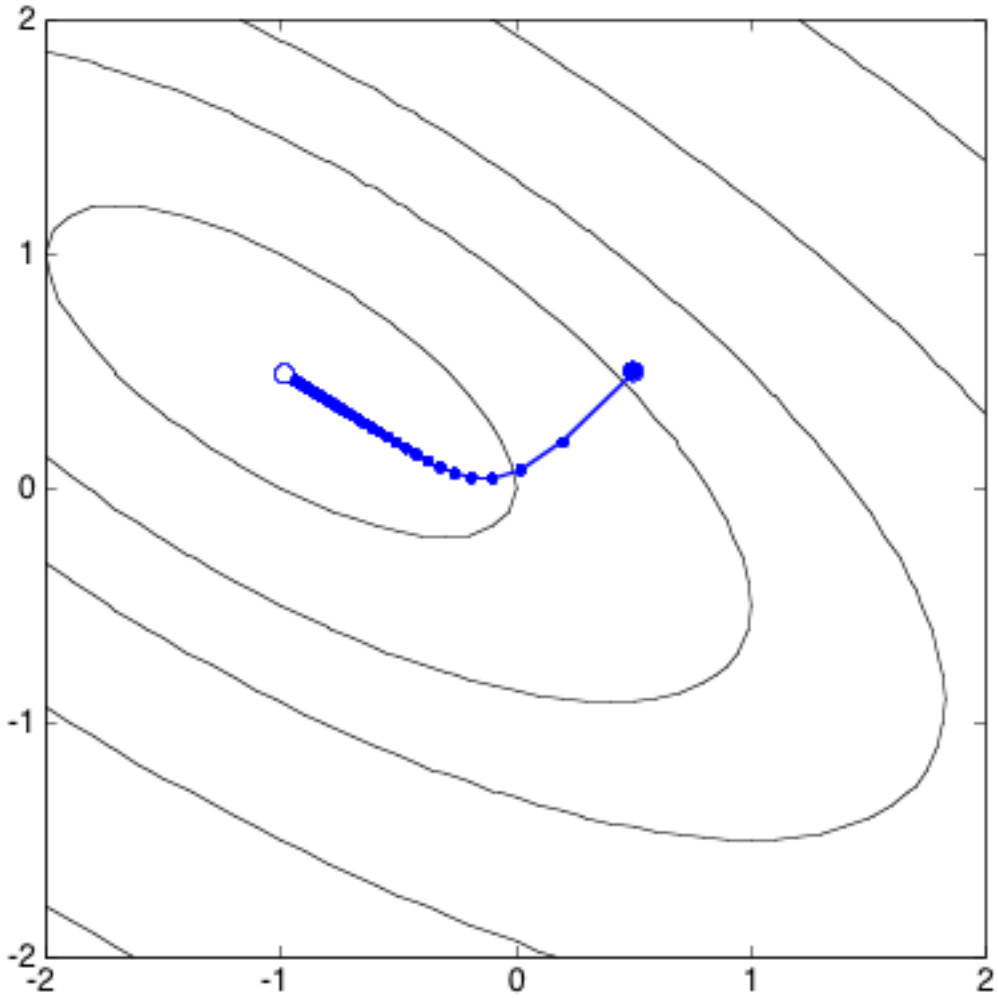
$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \qquad \alpha = 0.1$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \qquad \mathbf{g}_0 = \nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.1 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix}$$

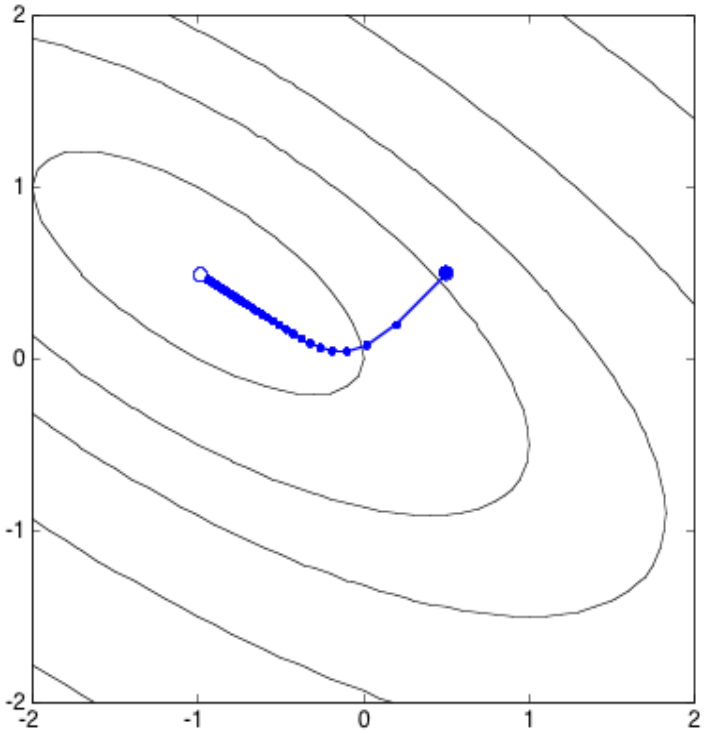
$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix} - 0.1 \begin{bmatrix} 1.8 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.08 \end{bmatrix}$$

Steepest Descent Plot

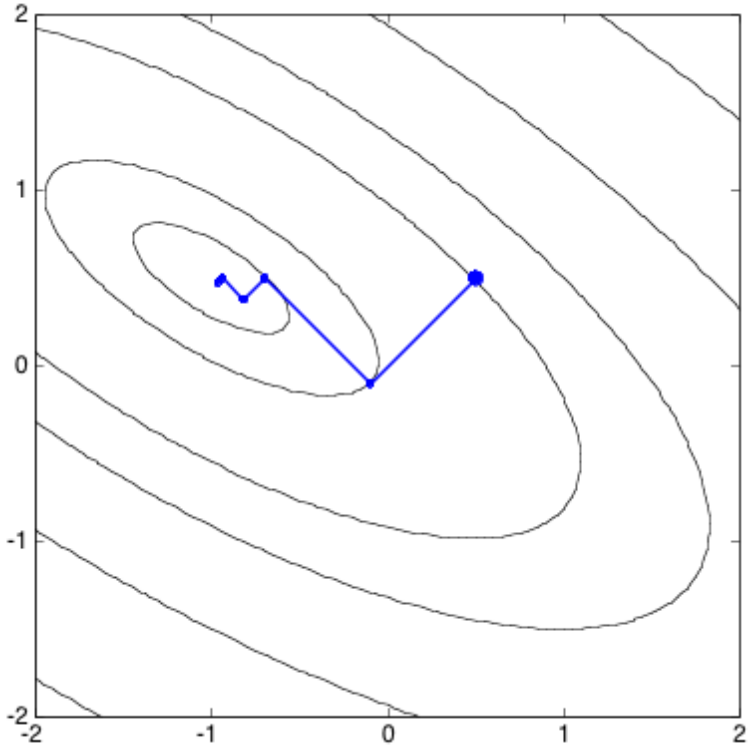


Note:
Lots of
small steps
toward end

Steepest Descent



Conjugate Gradient



CG is a **batch-mode** algorithm

Gradient is based on all samples.

CG Accelerates by Minimizing MSE Function ***Along a Line***

$\boldsymbol{p}_k$ is **some chosen line direction**, e.g. $-\boldsymbol{g}_k$ (negative gradient)

Compute learning rate α_k to minimize $F(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$

Derivative wrt α_k of Taylor expansion:

$$\frac{d}{d\alpha_k}(F(\mathbf{x}_k + \alpha_k \mathbf{p}_k)) = \nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k + \alpha_k \mathbf{p}_k^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k$$

Set derivative to 0 and **solve for** α_k:

$$\alpha_k = -\frac{\nabla F(\mathbf{x})^T \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k} \mathbf{p}_k} = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}_k \mathbf{p}_k}$$

is value where derivative is 0

where

where $\mathbf{A}_k \equiv \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}_k}$ (Hessian)

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

Analytic Example for Illustration Purposes

Quadratic function

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x}$$

Starting Point

$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

Gradient

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

Initial Direction

$$\mathbf{p}_0 = -\mathbf{g}_0 = -\nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} -3 \\ -3 \end{bmatrix}$$

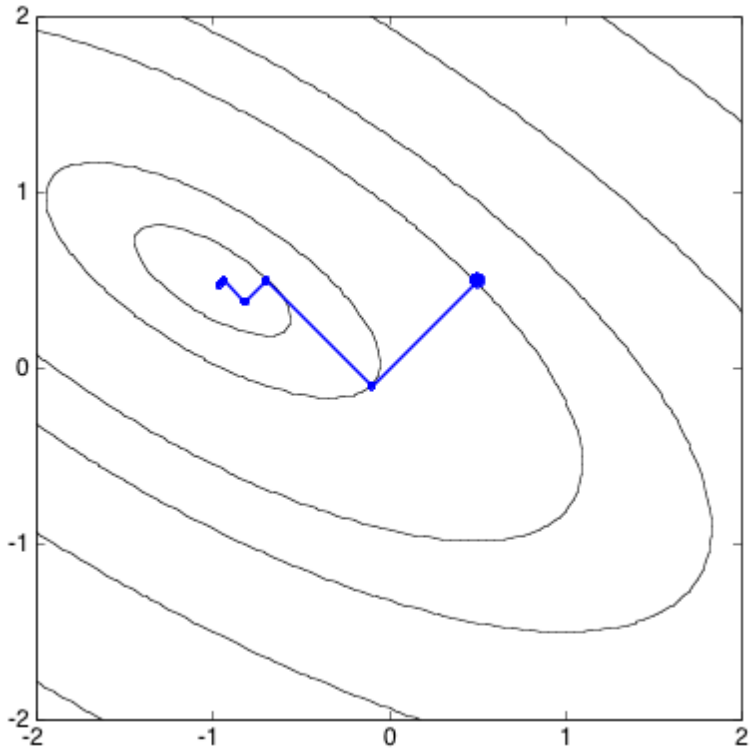
Solve

$$\alpha_0 = -\frac{\begin{bmatrix} 3 & 3 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix}}{\begin{bmatrix} -3 & -3 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix}} = 0.2$$

New Point

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha_0 \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.2 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix}$$

Successive Line Minimizations with different directions



How to choose directions?

Conjugate Vectors

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

A set of vectors $\mathbf{p}_i$ is mutually *conjugate* with respect to a positive definite Hessian matrix **A** provided

$$\mathbf{p}_k^T\mathbf{A}\mathbf{p}_j = 0 \qquad k \neq j$$

Conjugate is like "orthogonal with respect to A".

One set of conjugate vectors consists of the eigenvectors of **A**.

$$\mathbf{z}_k^T\mathbf{A}\mathbf{z}_j = \lambda_j\mathbf{z}_k^T\mathbf{z}_j = 0 \qquad k \neq j$$

(The eigenvectors of symmetric matrices are orthogonal.)

For *Quadratic* Functions

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}$$

$$\nabla^2 F(\mathbf{x}) = \mathbf{A}$$

The change in the gradient at iteration k is

$$\Delta \mathbf{g}_k = \mathbf{g}_{k+1} - \mathbf{g}_k = (\mathbf{A}\mathbf{x}_{k+1} + \mathbf{d}) - (\mathbf{A}\mathbf{x}_k + \mathbf{d}) = \mathbf{A}\Delta\mathbf{x}_k$$

where

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

The conjugacy conditions can then be rewritten

$$\alpha_k \mathbf{p}_k^T \mathbf{A}\mathbf{p}_j = \Delta \mathbf{x}_k^T \mathbf{A}\mathbf{p}_j = \Delta \mathbf{g}_k^T \mathbf{p}_j = 0 \qquad k \neq j$$

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:

$$\mathbf{p}_0 = -\mathbf{g}_0$$

Choose subsequent search directions to be *conjugate*:

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

where β_k is chosen according to *one of these formulae*:

$$\beta_k = \frac{\Delta\mathbf{g}_{k-1}^T \mathbf{g}_k}{\Delta\mathbf{g}_{k-1}^T \mathbf{p}_{k-1}} \quad \textbf{or} \quad \beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \quad \textbf{or} \quad \beta_k = \frac{\Delta\mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

Hestnes & Steiffel | Fletcher-Reeves | Polak & Ribiere

Conjugate Gradient algorithm

- 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.

$$\alpha_k = -\frac{\nabla F(\mathbf{x})^T \Big|_{\mathbf{x}=\mathbf{x}_k} \mathbf{p}_k}{\mathbf{p}_k^T \nabla^2 F(\mathbf{x}) \Big|_{\mathbf{x}=\mathbf{x}_k} \mathbf{p}_k} = -\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}_k \mathbf{p}_k}$$

(e.g. for quadratic functions only.)

- Select the next search direction using

β_k is from previous slide

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

- 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.

Previous Example, with Follow-On

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x} \qquad \mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \qquad \mathbf{p}_0 = -\mathbf{g}_0 = -\nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} -3 \\ -3 \end{bmatrix}$$

$$\alpha_0 = -\frac{\begin{bmatrix} 3 & 3 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix}}{\begin{bmatrix} -3 & -3 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \end{bmatrix}} = 0.2 \qquad \mathbf{x}_1 = \mathbf{x}_0 - \alpha_0 \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.2 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix}$$

Follow-On

$$\mathbf{g}_1 = \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_1} = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}\begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.6 \end{bmatrix}$$

$$\beta_1 = \frac{\mathbf{g}_1^T \mathbf{g}_1}{\mathbf{g}_0^T \mathbf{g}_0} = \frac{\begin{bmatrix} 0.6 & -0.6 \end{bmatrix}\begin{bmatrix} 0.6 \\ -0.6 \end{bmatrix}}{\begin{bmatrix} 3 & 3 \end{bmatrix}\begin{bmatrix} 3 \\ 3 \end{bmatrix}} = \frac{0.72}{18} = 0.04$$

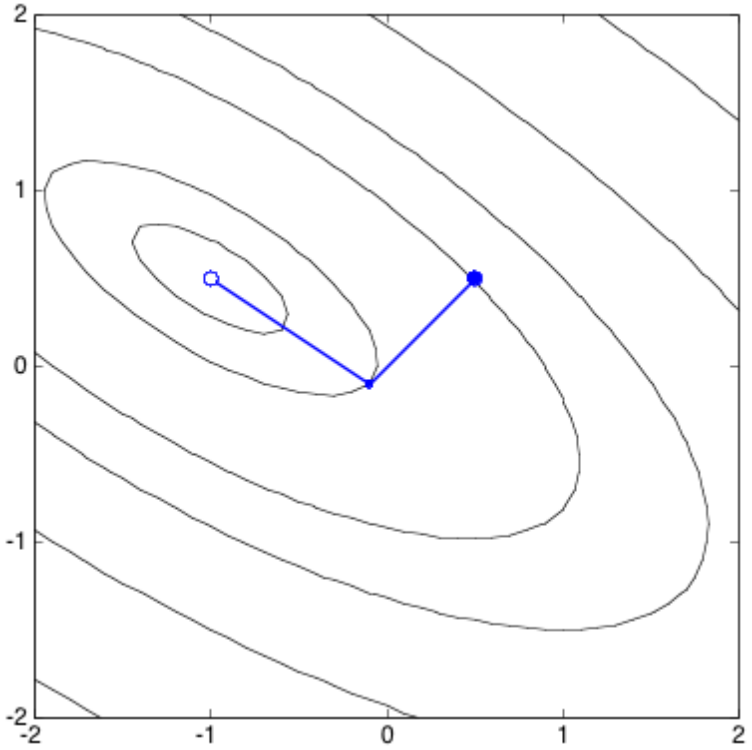
$$\mathbf{p}_1 = -\mathbf{g}_1 + \beta_1 \mathbf{p}_0 = \begin{bmatrix} -0.6 \\ 0.6 \end{bmatrix} + 0.04\begin{bmatrix} -3 \\ -3 \end{bmatrix} = \begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix}$$

$$\alpha_1 = -\frac{\begin{bmatrix} 0.6 & -0.6 \end{bmatrix}\begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix}}{\begin{bmatrix} -0.72 & 0.48 \end{bmatrix}\begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}\begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix}} = -\frac{-0.72}{0.576} = 1.25$$

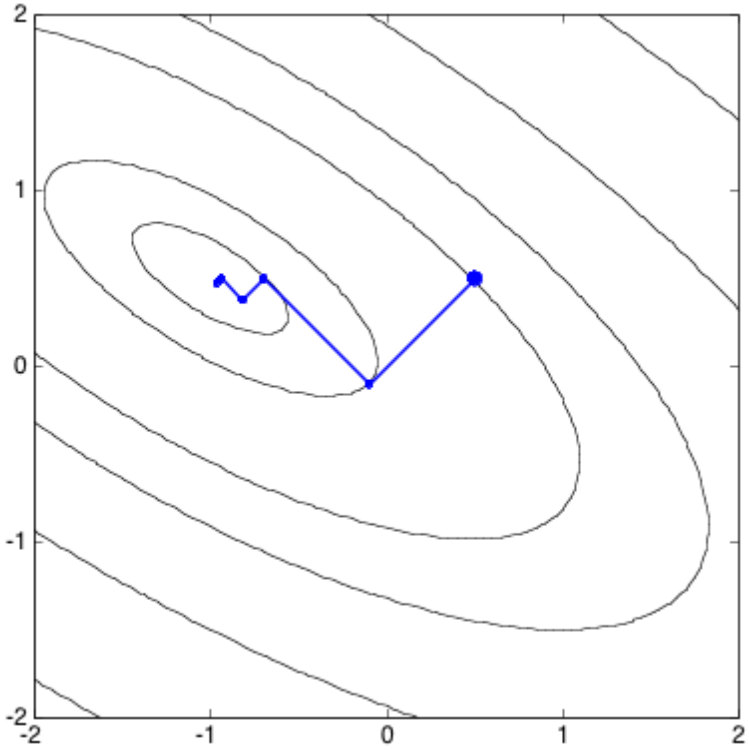
Conjugate Gradient vs. Steepest Descent Plots

$$\mathbf{x}_2 = \mathbf{x}_1 + \alpha_1 \mathbf{p}_1 = \begin{bmatrix} -0.1 \\ -0.1 \end{bmatrix} + 1.25 \begin{bmatrix} -0.72 \\ 0.48 \end{bmatrix} = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix}$$

Conjugate Gradient



Steepest Descent



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

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 srchFcn 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.

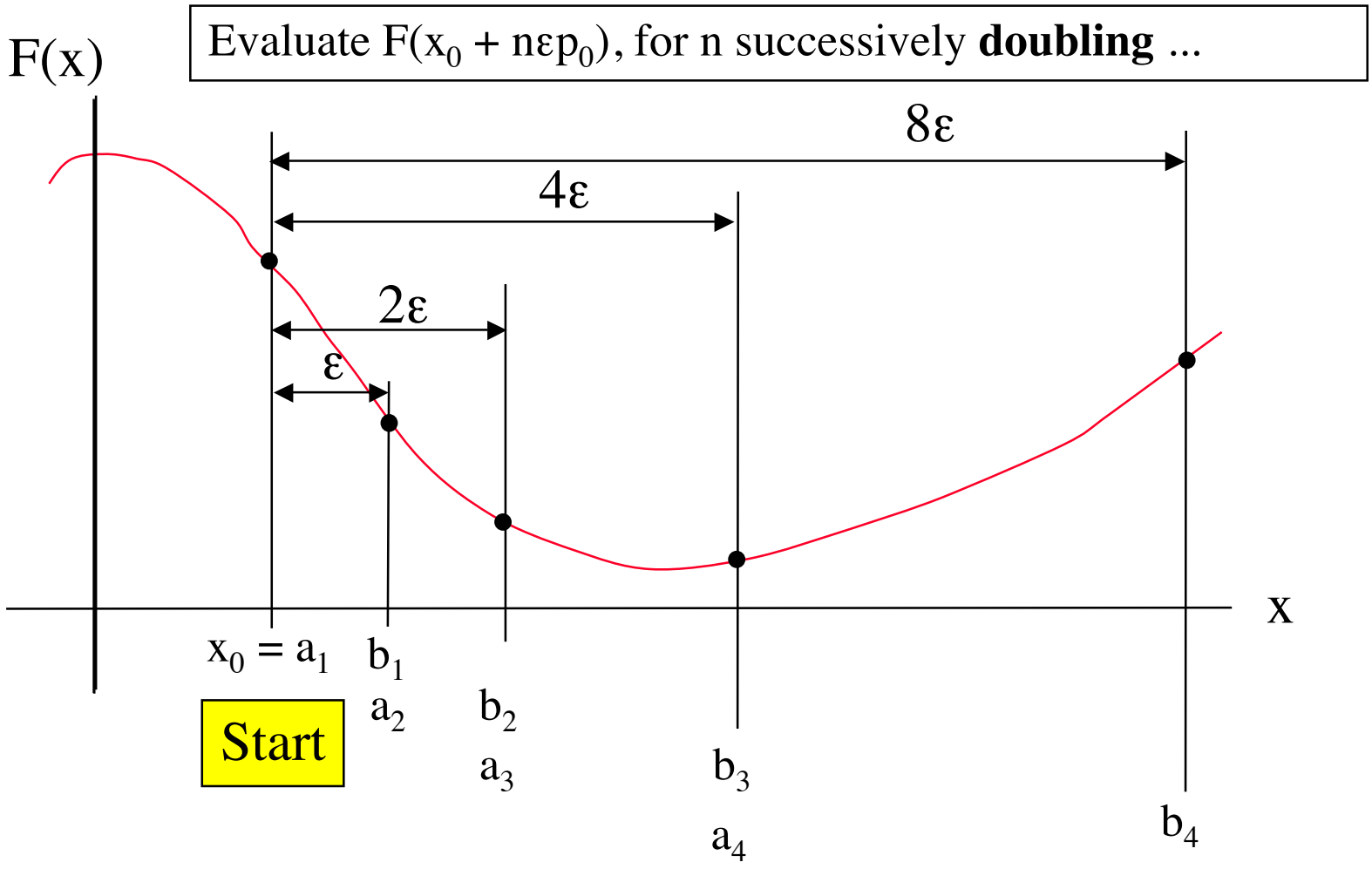
Available Searches

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

Example Numerical Line 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

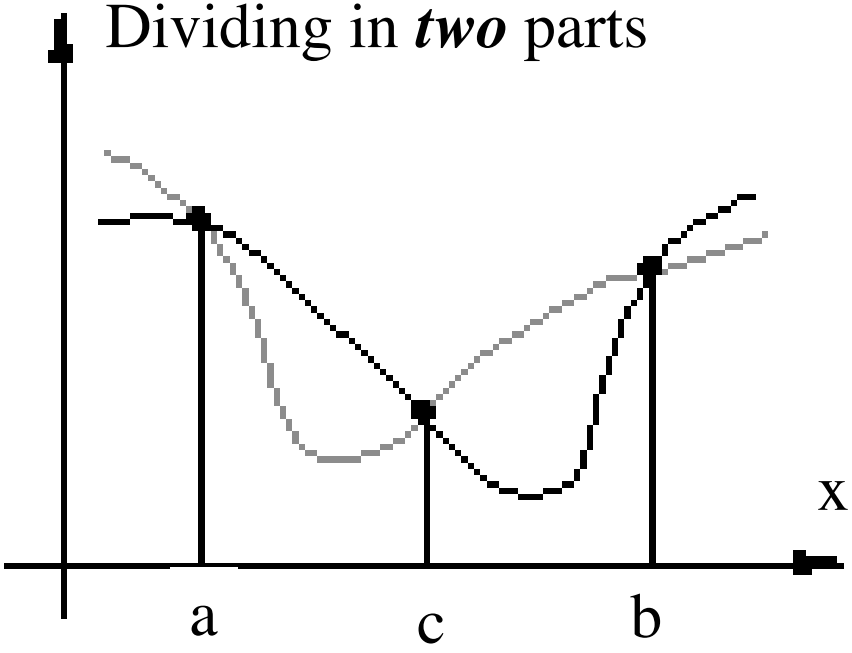


Stop when two successive increases occur. **Minimum is bracketed. Proceed to Part 2.**

Part 2: Interval Reduction: Divide&Conquer

What doesn't help:

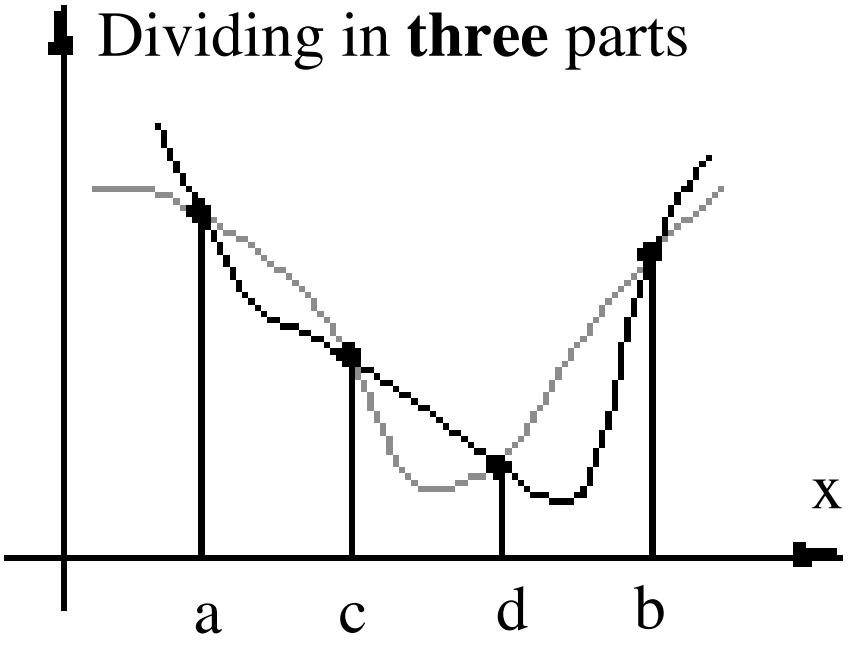
Dividing in ***two*** parts



Bracketing interval **not reduced**.

What does:

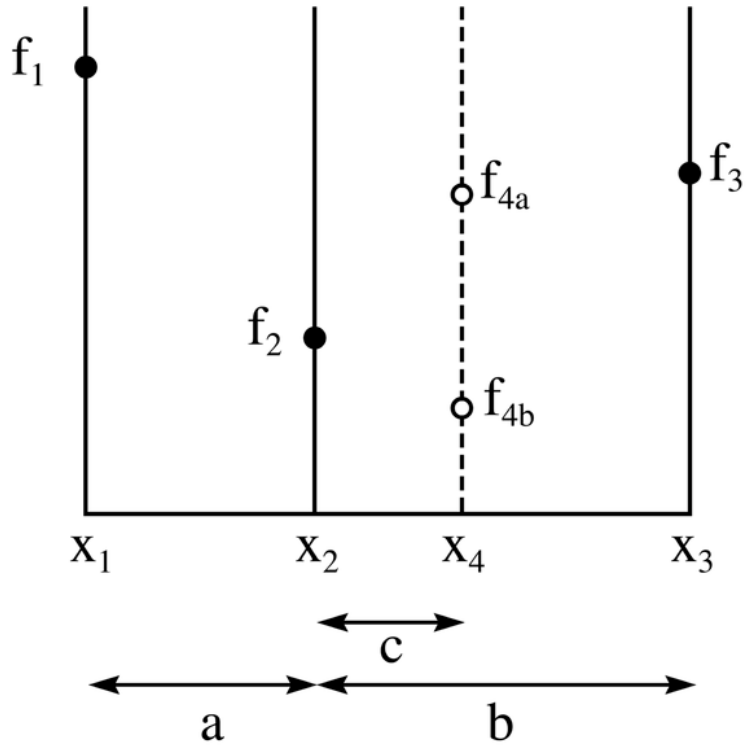
Dividing in **three** parts



$c > d \Rightarrow$ minimum is between c and b.

$d > c \Rightarrow$ minimum is between a and d.

Golden Section Search Logic

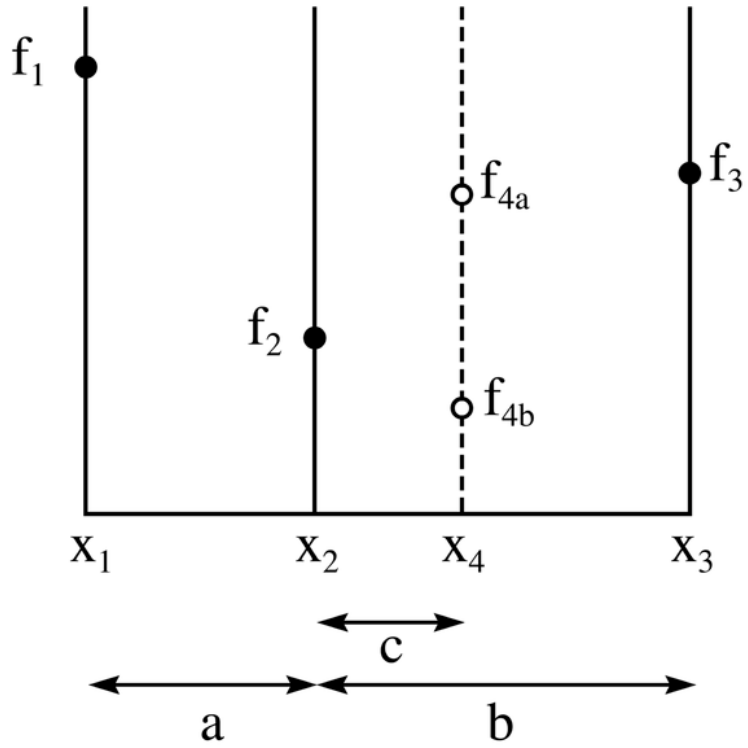


From the diagram above, it is seen that the **new search interval** will be either between x_1 and x_4 with a length of a+c , or between x_2 and x_3 with a length of b .

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.

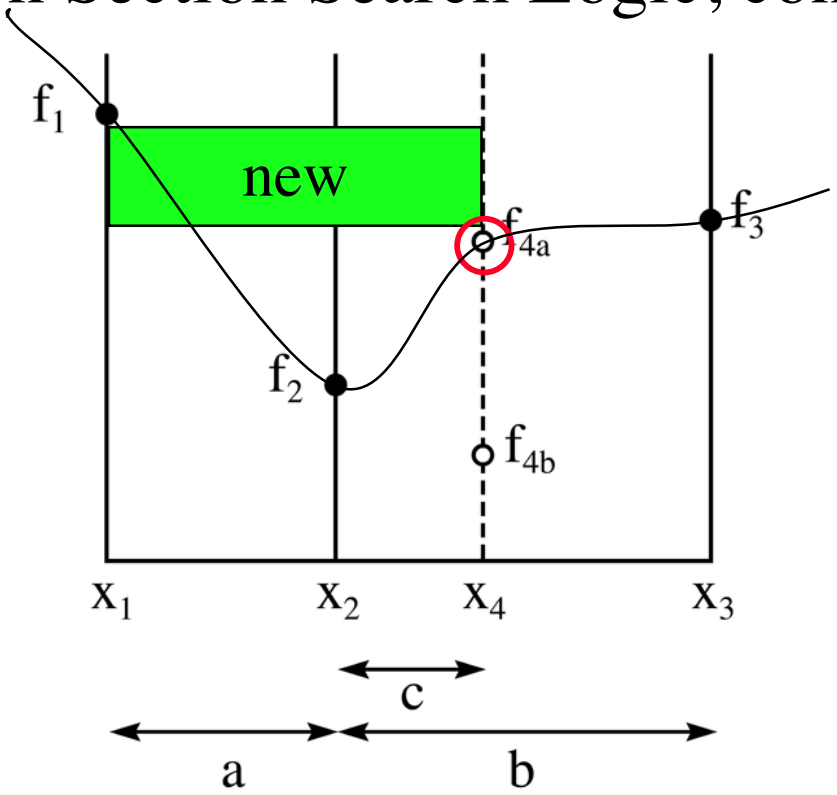
To ensure that b = a+c, the algorithm should choose $\mathbf{x_4 = x_1 - x_2 + x_3}$.

Golden Section Search Logic, continued



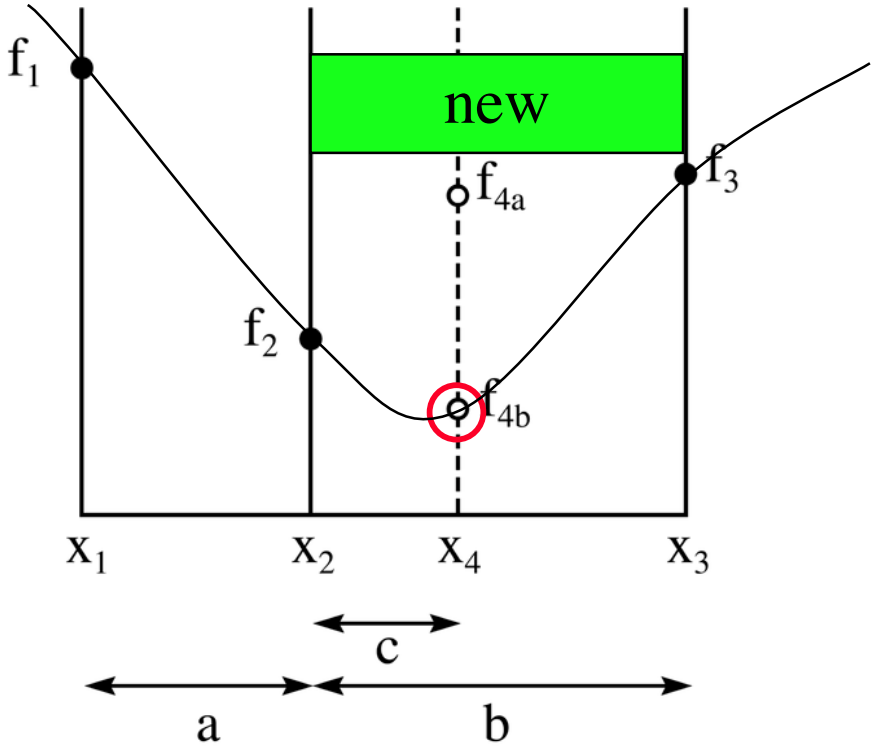
However there still remains the question of where x_2 should be placed in relation to x_1 and x_3. The golden section search **chooses the spacing between these points in such a way that these points have the same proportion of spacing as the subsequent triple x_1, x_2, x_4 or x_2, x_4, x_4**. By maintaining 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, and x_4 then we want **c/a = a/b**.

Golden Section Search Logic, continued

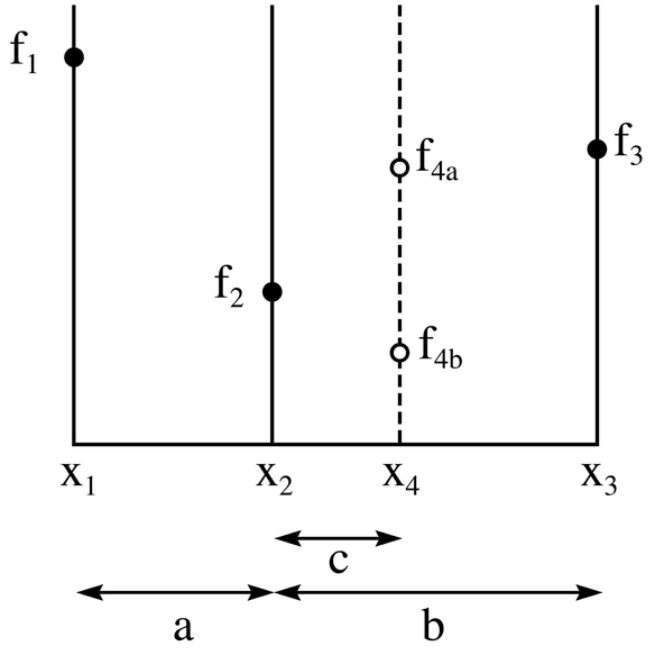


However, if $f(x_4)$ is f_{4b} and our new triplet of points is x_2, x_4, and x_3 then we want $c/(b-c) = a/b$.

Golden Section Search Logic, concluded

c/a = a/b

c/(b-c) = a/b



Eliminating *c* from these two simultaneous equations yields:

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

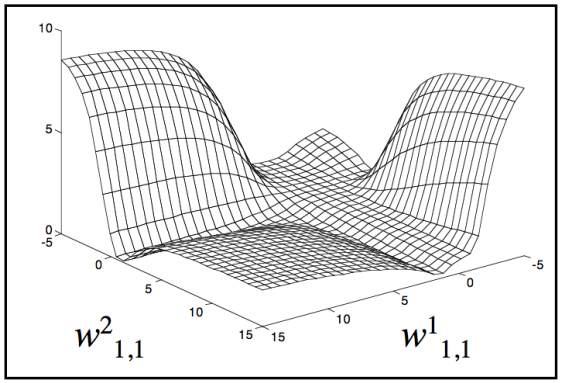
or

$$\frac{b}{a} = \varphi$$

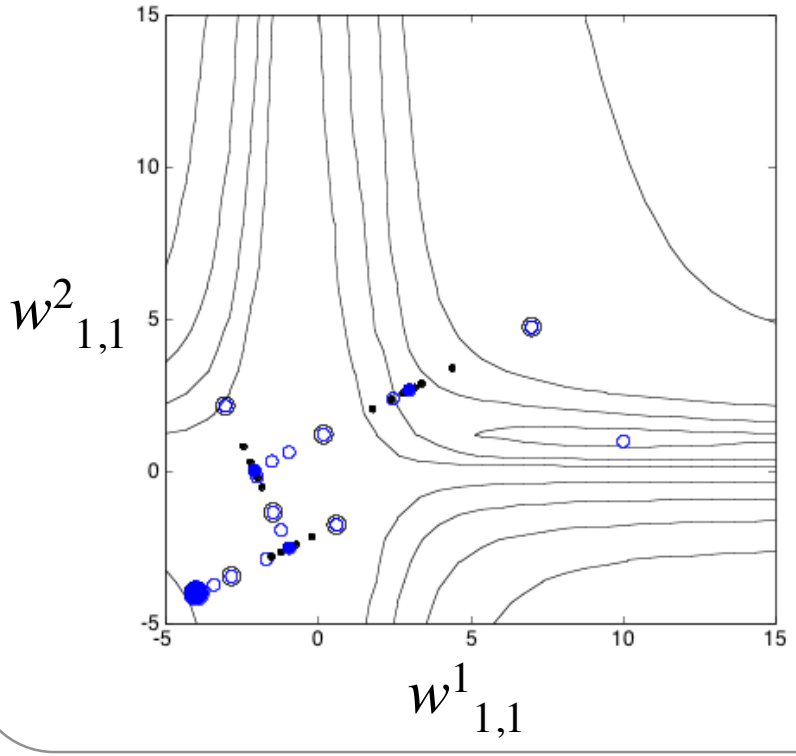
where ϕ is the golden ratio:

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.618033988\ldots$$

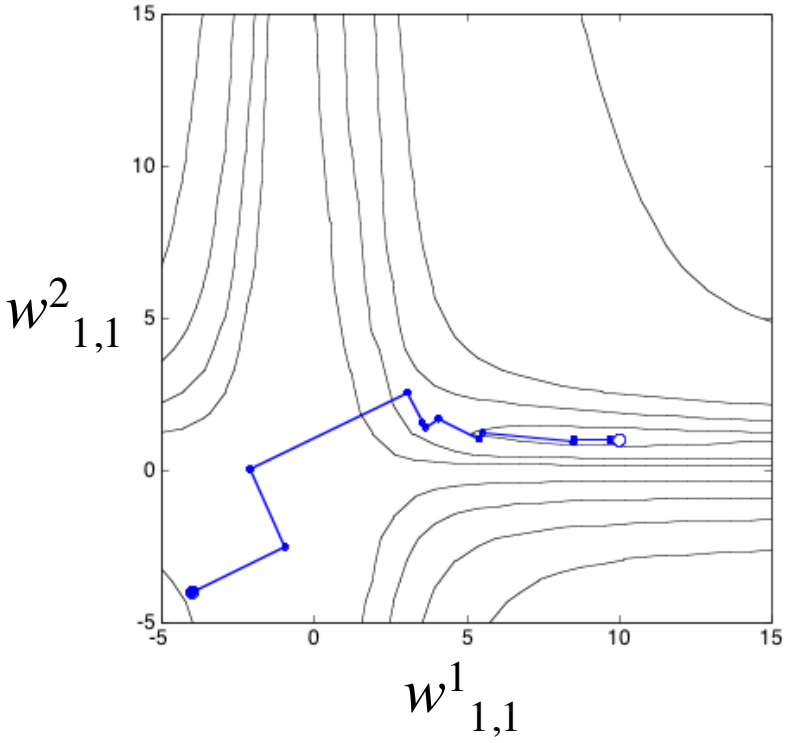
Conjugate Gradient BP (CGBP) Demo



Intermediate Steps
(showing line searches)



Complete Trajectory



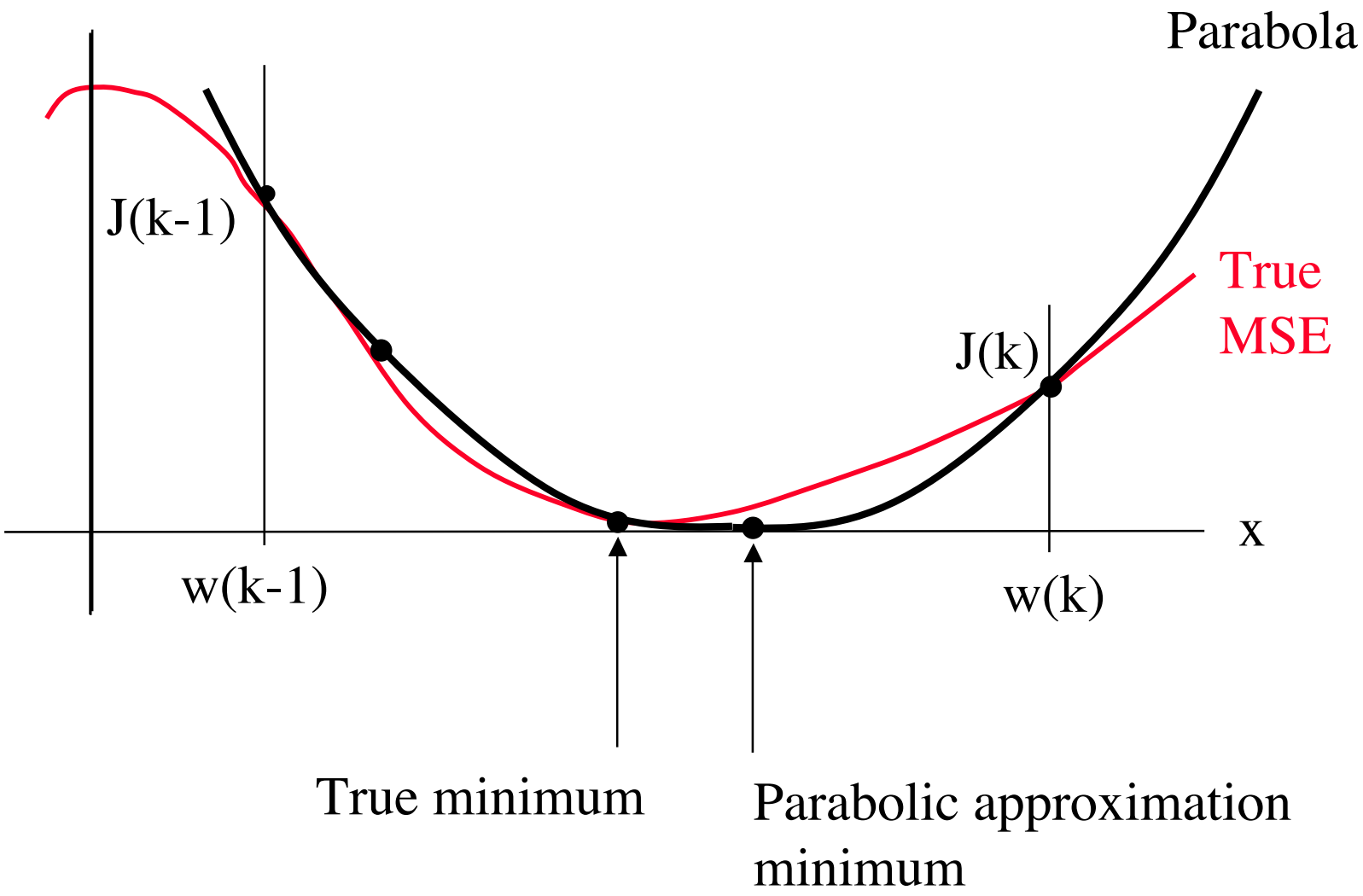
Methods Involving Solving Equations to get the Next Point

- 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 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 applicable when, between two steps, the **gradient** has decreased in magnitude and has **changed sign**.

Quickprop, step k, in 1 dimension



Quickprop

- Assume a parabola:

  $J(w) = aw^2 + bw + c$

- First derivative of J is a line:

  $\partial J/\partial w = 2aw + b$

  abbreviate $\partial J/\partial w$ 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:

  $2a = [J'(w(k))-J'(w(k-1))]/ \Delta w(k-1)$

  $b = J'(w(k)) - [(J'(w(k))-J'(w(k-1)))w(k) / \Delta w(k-1)]$

  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

  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 Waugh* 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. Ramasubramanian and A. Kannan

Department of Computer Science and Engineering
Anna University, Chennai 600025, India.
suryarams@cs.annauniv.edu, kannan@annauniv.edu

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

2000

Neural Processing Letters **12:** 159–169, 2000.

Globally Convergent Modification of the Quickprop Method

MICHAEL N. VRAHATIS[1,3], GEORGE D. MAGOULAS[2,3]*
and VASSILIS P. PLAGIANAKOS[1,3]
[1] *Department of Mathematics, University of Patras, GR-261.10 Patras, Greece.* [2] *Department of Informatics, University of Athens, GR-157.84 Athens, Greece.* [3] *University of Patras Artificial Intelligence Research Center–UPAIRC.*
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) =

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

- Use

$$w_i^{k+1} = w_i^k - \eta \left\{ \frac{|\partial_i E(w^k) - \partial_i E(w^{k-1})|}{|w_i^k - w_i^{k-1}|} \right\}^{-1} \partial_i E(w^k),$$

Examples (Modified Qprop)

Table II. Results for the texture classification problem, ($n = 244$)

Algorithm	μ_{IT}	μ_{FE}	Success
BP	15839	31678	960/1000
SDLS	13256	26517	965/1000
BPM	12422	24844	940/1000
ABP	560	1120	1000/1000
FR	1624	12674	250/1000
PR	140	810	990/1000
PR-FR	145	1005	996/1000
MQprop	406	1228	1000/1000

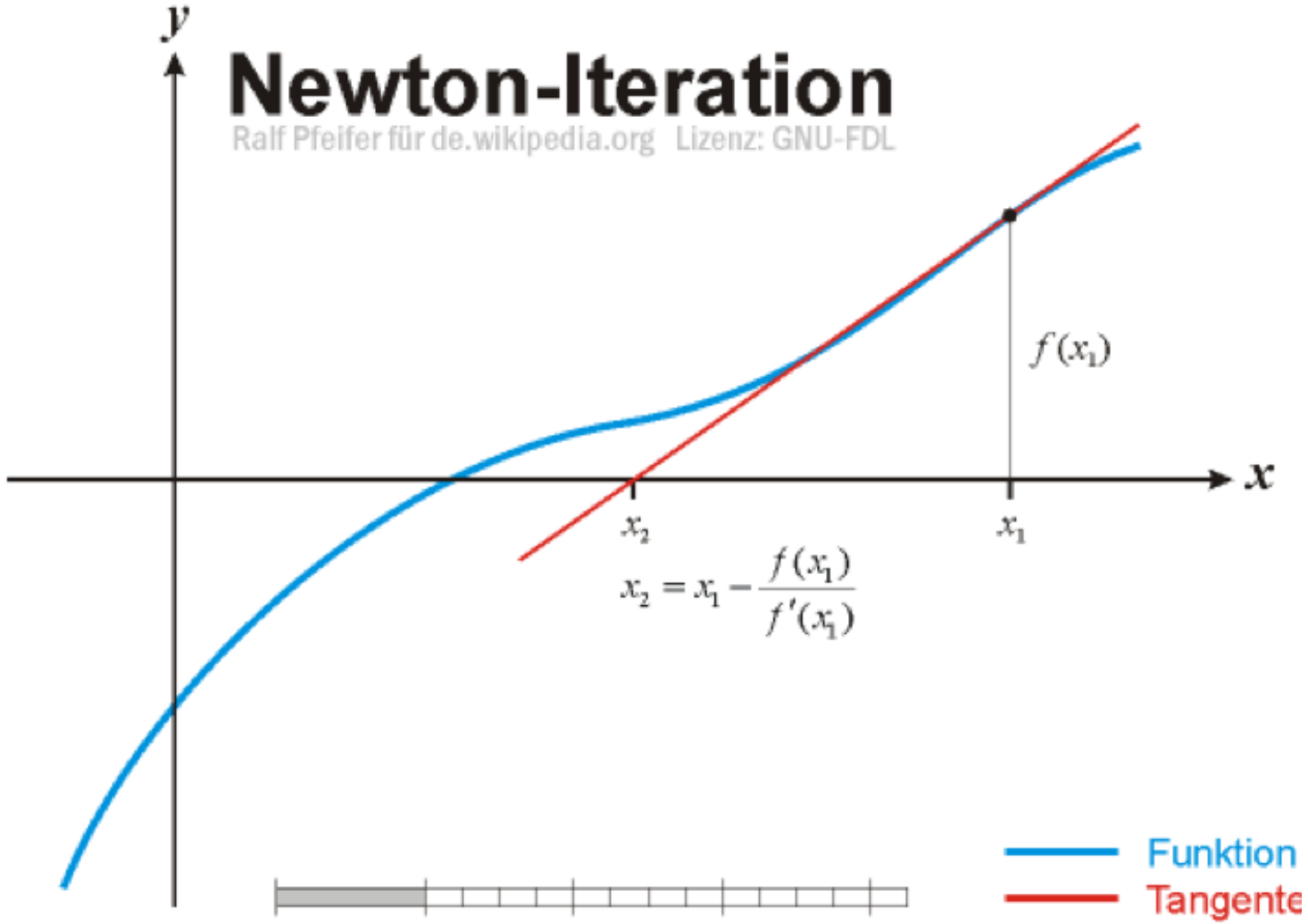
Table I. Results for the XOR problem, ($n = 9$).

Algorithm	μ_{IT}	μ_{FE}	Success
BP	549	1098	810/1000
SDLS	64	435	810/1000
BPM	803	1606	810/1000
ABP	157	314	810/1000
FR	84	282	130/1000
PR	21	169	380/1000
PR-FR	22	171	410/1000
MQprop	52	234	810/1000

Table III. Results for the numeric font learning problem, ($n = 460$).

Algorithm	μ_{IT}	μ_{FE}	Success
BP	14489	28978	660/1000
SDLS	12225	24454	990/1000
BPM	10142	20284	540/1000
ABP	1975	3950	910/1000
FR	620	3121	420/1000
PR	649	2124	960/1000
PR-FR	750	3473	1000/1000
MQprop	159	739	1000/1000

Newton's Method
for finding zero of a function



Newton's Method for finding minimum of a function

To find a zero:

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

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

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 $\mathbf{x}_k$

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k + \frac{1}{2}\Delta\mathbf{x}_k^T \mathbf{A}_k \Delta\mathbf{x}_k$$

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

$$\mathbf{g}_k + \mathbf{A}_k \Delta\mathbf{x}_k = \mathbf{0}$$

$$\Delta\mathbf{x}_k = -\mathbf{A}_k^{-1}\mathbf{g}_k$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k$$

Iteration formula

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

scalar analogue

Newton's Method for Locating a Minimum of MSE: Solving rather than searching

(Notation shift: x's are weights, not inputs)

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k$$

Hessian = vector 2nd derivative

$$\mathbf{A}_k \equiv \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k} \qquad \mathbf{g}_k \equiv \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k}$$

Gradient = vector 1st derivative

For the SSE function, ν_i = error in ith sample:

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

the j^{th} element of the gradient is

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

Example

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

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

$$\mathbf{g}_0 = \nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

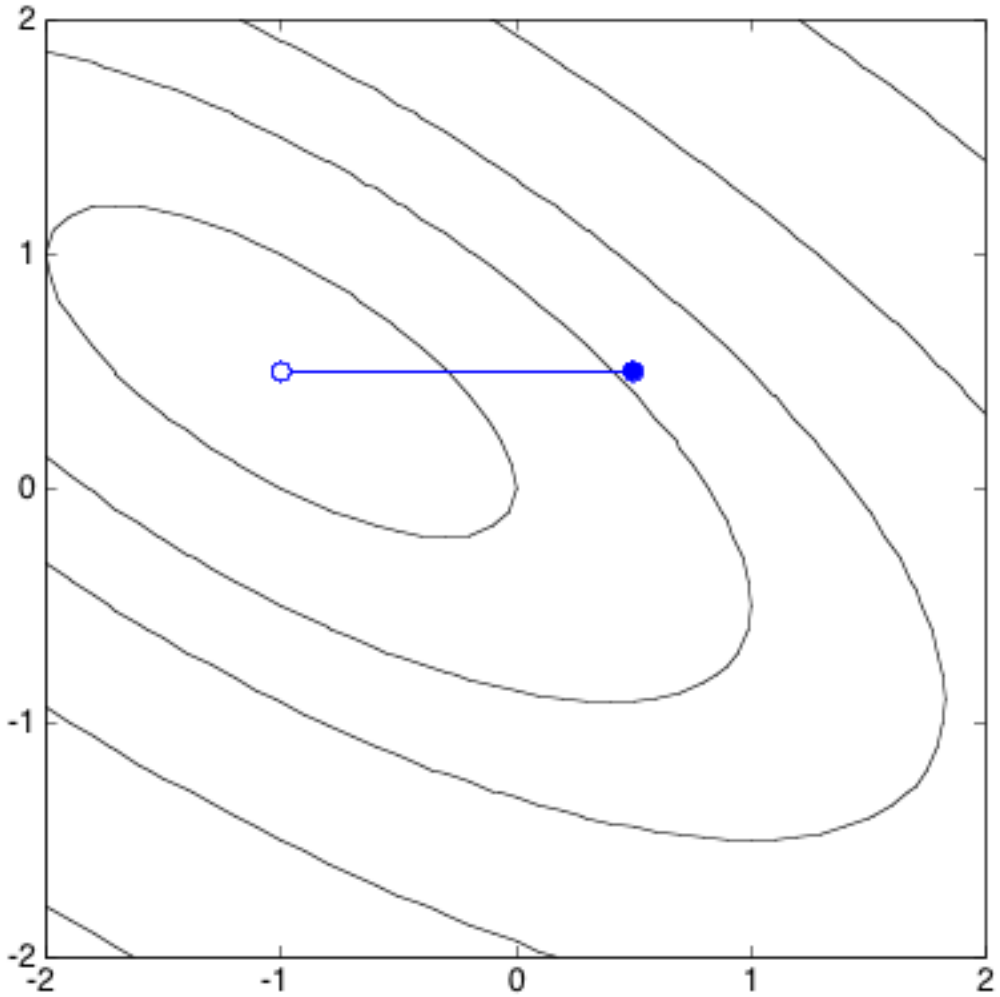
$$\mathbf{A} = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k$$

$$\mathbf{x}_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}^{-1} \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 1 & -0.5 \\ -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 1.5 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix}$$

Plot of Newton's Method for a Quadratic

Solve for minimum in one step



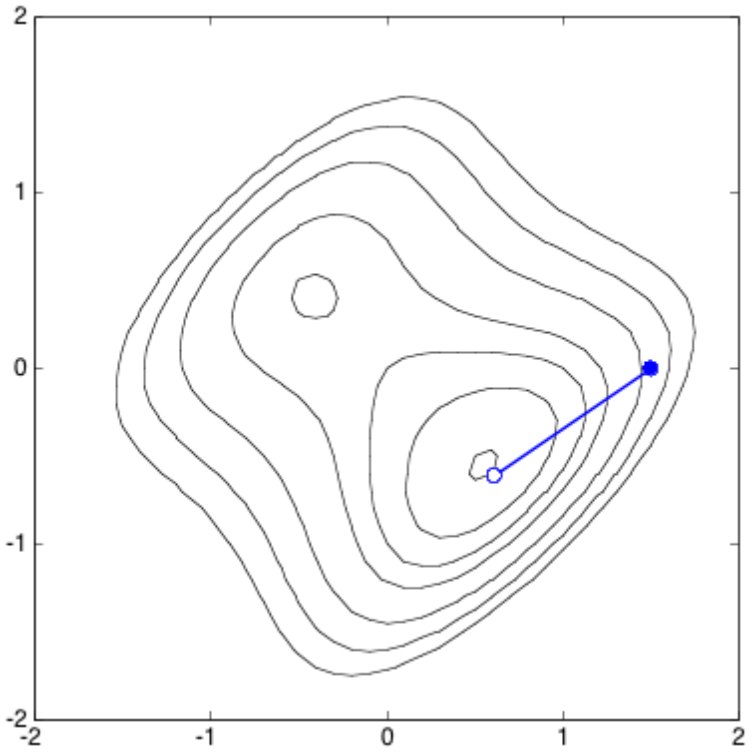
Non-Quadratic Case:

Use Quadratic Approximation Iteratively

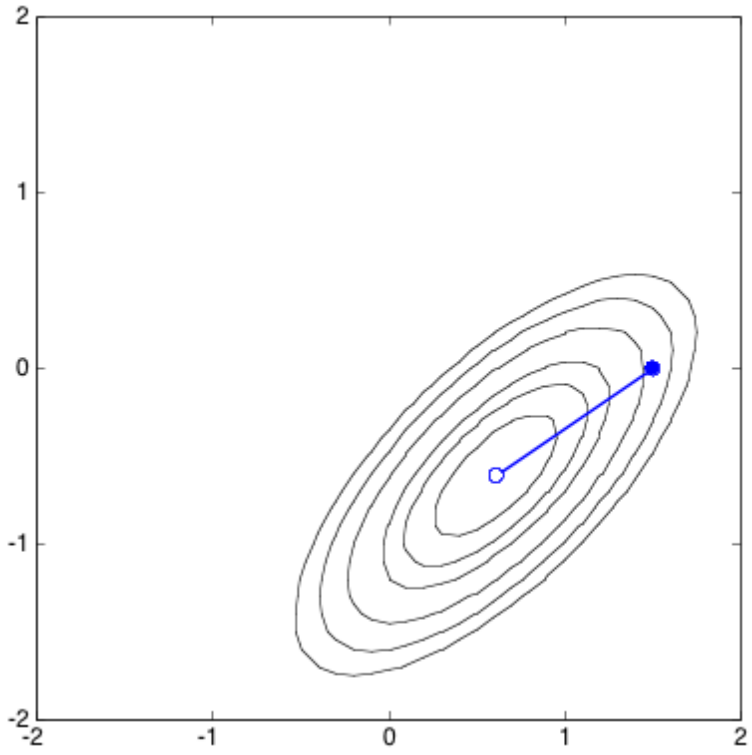
$$F(\mathbf{x}) = (x_2 - x_1)^4 + 8x_1x_2 - x_1 + x_2 + 3$$

Stationary Points: $\mathbf{x}^1 = \begin{bmatrix} -0.42 \\ 0.42 \end{bmatrix}$ $\mathbf{x}^2 = \begin{bmatrix} -0.13 \\ 0.13 \end{bmatrix}$ $\mathbf{x}^3 = \begin{bmatrix} 0.55 \\ -0.55 \end{bmatrix}$

$F(\mathbf{x})$

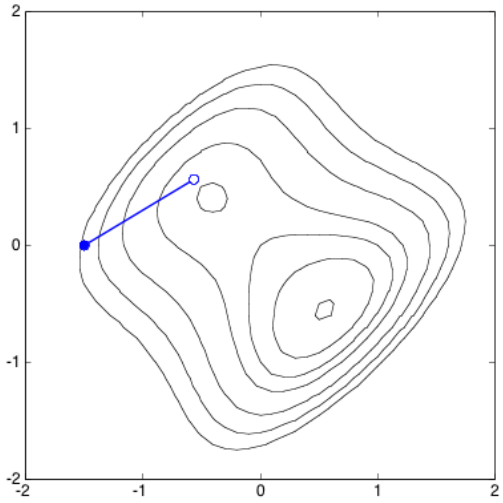


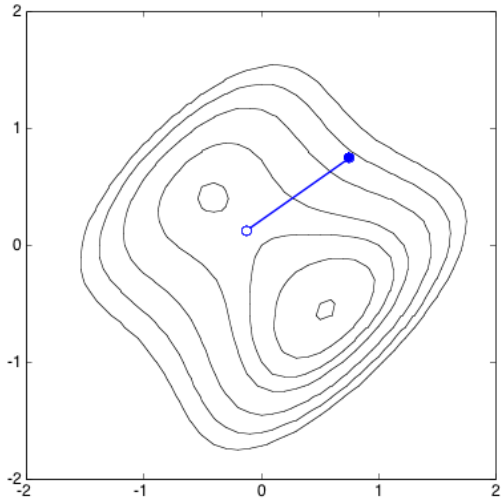
$F_2(\mathbf{x})$, **Quadratic Approximation**

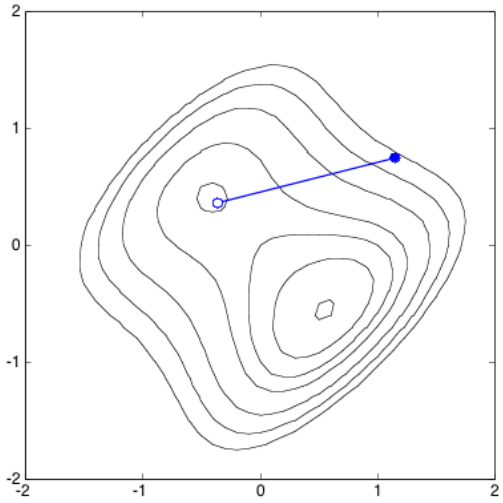


Different Initial Conditions and Quadratic Approximations to Each

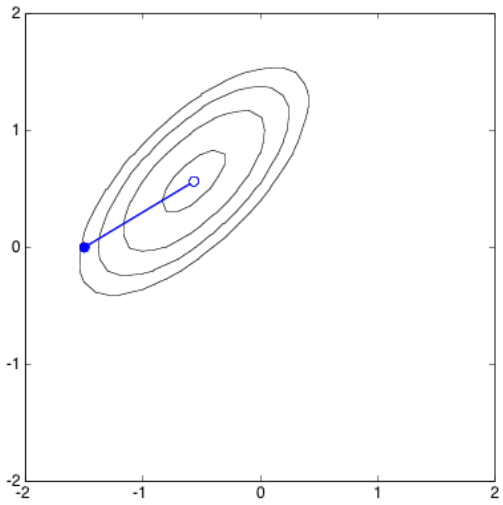
$F(\mathbf{x})$

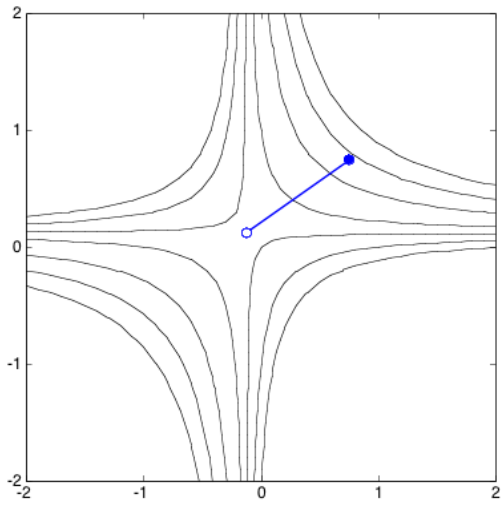


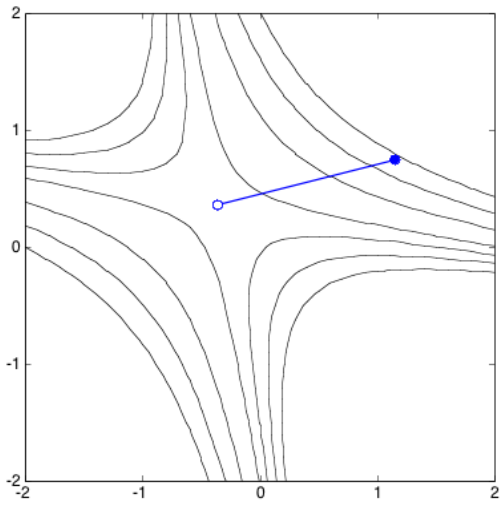




$F_2(\mathbf{x})$







Levenberg-Marquardt Method

(blends 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:

$$\nabla F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{v}(\mathbf{x})$$

where **J** is the Jacobian matrix first derivatives of errors based on weights:

samples

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

ν are the error values for **each sample**

weights

Express the Hessian of F in terms of the Jacobian:

Hessian

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

$$\nabla^2 \mathrm{F}(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\mathbf{S}(\mathbf{x})$$

(outer product)

where $\mathbf{S}(\mathbf{x}) = \sum_{i=1}^{N} v_i(\mathbf{x})\nabla^2 v_i(\mathbf{x})$

Gauss-Newton Method

Approximate the Hessian matrix by ***neglecting S***:

$$\nabla^2 F(\mathbf{x}) \cong 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) \qquad \text{(outer product)}$$

Newton's method $\Delta\mathbf{x}_k = -\mathbf{A}_k^{-1}\mathbf{g}_k$

then becomes:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \underbrace{[2\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}}\underbrace{2\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)}$$

$$= \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

(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 challenging to compute, are not required.)

Levenberg-Marquardt

As seen, Gauss-Newton method approximates the Hessian by:

$$\mathbf{H} = \mathbf{J}^T\mathbf{J}$$

This matrix may be **singular**, but can be made invertible as follows:

$$\mathbf{G} = \mathbf{H} + \mu\mathbf{I} \quad \text{for some } \mu > 0$$

If the eigenvalues and eigenvectors of $\mathbf{H}$ are:

$$\{\lambda_1, \lambda_2, \ldots, \lambda_n\} \qquad \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$$

thus

$$\mathbf{G}\mathbf{z}_i = [\mathbf{H} + \mu\mathbf{I}]\mathbf{z}_i = \mathbf{H}\mathbf{z}_i + \mu\mathbf{z}_i = \lambda_i\mathbf{z}_i + \mu\mathbf{z}_i = \overbrace{(\lambda_i + \mu)}^{\text{Eigenvalues of } \mathbf{G}}\mathbf{z}_i$$

Thus use $\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k\mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$

Varying μ_k

As $\mu_k \to 0$, LM approaches pure **Gauss-Newton**.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

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

As $\mu_k \to \infty$, LM approaches pure
Steepest Descent with small learning rate $(1/\mu_k)$.

$$\mathbf{x}_{k+1} \cong \mathbf{x}_k - \frac{1}{\mu_k}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) = \mathbf{x}_k - \frac{1}{2\mu_k}\nabla F(\mathbf{x})$$

Generally converges, but may do so slowly.

Adjustment of μ_k

- Begin with a small μ_k to approximate **Gauss-Newton**.

- If the step does not yield a smaller $F(\mathbf{x})$, then repeat the step with a **larger** μ_k, until $F(\mathbf{x})$ is decreased.

- $F(\mathbf{x})$ *must* decrease eventually, since for sufficiently large μ_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:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) = \sum_{q=1}^{Q} \mathbf{e}_q^T \mathbf{e}_q = \sum_{q=1}^{Q} \sum_{j=1}^{S^M} (e_{j,q})^2 = \sum_{i=1}^{N} (v_i)^2$$

The error vector is:

$$\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \cdots & v_N \end{bmatrix} = \begin{bmatrix} e_{1,1} & e_{2,1} & \cdots & e_{S^M,1} & e_{1,2} & \cdots & e_{S^M,Q} \end{bmatrix}$$

S^M = number of outputs,

Q = number of samples

The weight vector (across all layers) is:

$$\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} = \begin{bmatrix} w^1_{1,1} & w^1_{1,2} & \cdots & w^1_{S^1,R} & b^1_1 & \cdots & b^1_{S^1} & w^2_{1,1} & \cdots & b^M_{S^M} \end{bmatrix}$$

The dimensions of the two vectors are:

$$N = Q \times S^M \qquad n = S^1(R+1) + S^2(S^1+1) + \cdots + S^M(S^{M-1}+1)$$

Jacobian Matrix for Levenberg-Marquardt

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w^1_{1,1}} & \frac{\partial e_{1,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,1}}{\partial b^1_1} & \cdots \\ \frac{\partial e_{2,1}}{\partial w^1_{1,1}} & \frac{\partial e_{2,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{2,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{2,1}}{\partial b^1_1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \\ \frac{\partial e_{S^M,1}}{\partial w^1_{1,1}} & \frac{\partial e_{S^M,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{e_{S^M,1}}}{\partial w^1_{S^1,R}} & \frac{\partial e_{e_{S^M,1}}}{\partial b^1_1} & \cdots \\ \frac{\partial e_{1,2}}{\partial w^1_{1,1}} & \frac{\partial e_{1,2}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,2}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,2}}{\partial b^1_1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \end{bmatrix}$$

M rows (one per output) for every input sample

Repeated number-of-samples times

weights →

Computing the Jacobian

Steepest descent computes squared-error terms of the form:

$$\frac{\partial \hat{F}(\mathbf{x})}{\partial x_l} = \frac{\partial \mathbf{e}_q^T \mathbf{e}_q}{\partial x_l}$$

using the chain rule:

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m}$$

where the sensitivity

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m}$$

is computed using backpropagation.

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

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial x_l}$$

Marquardt Sensitivity

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

$$\tilde{s}^m_{i,h} \equiv \frac{\partial v_h}{\partial n^m_{i,q}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \qquad h = (q-1)S^M + k$$

then compute the Jacobian as follows:

weight

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w^m_{i,j}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times a^{m-1}_{j,q}$$

bias

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b^m_i} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h}$$

Computing the Sensitivities

Initialization

$$\tilde{s}_{i,h}^{M} = \frac{\partial v_h}{\partial n_{i,q}^{M}} = \frac{\partial e_{k,q}}{\partial n_{i,q}^{M}} = \frac{\partial (t_{k,q} - a_{k,q}^{M})}{\partial n_{i,q}^{M}} = -\frac{\partial a_{k,q}^{M}}{\partial n_{i,q}^{M}}$$

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

$$\tilde{\mathbf{S}}_q^M = -\dot{\mathbf{F}}^M(\mathbf{n}_q^M)$$

Backpropagation

$$\tilde{\mathbf{S}}_q^m = \dot{\mathbf{F}}^m(\mathbf{n}_q^m)(\mathbf{W}^{m+1})^T \tilde{\mathbf{S}}_q^{m+1}$$

$$\tilde{\mathbf{S}}^m = \left[\tilde{\mathbf{S}}_1^m \middle| \tilde{\mathbf{S}}_2^m \middle| \cdots \middle| \tilde{\mathbf{S}}_Q^m\right] \qquad \text{Q = number of samples}$$

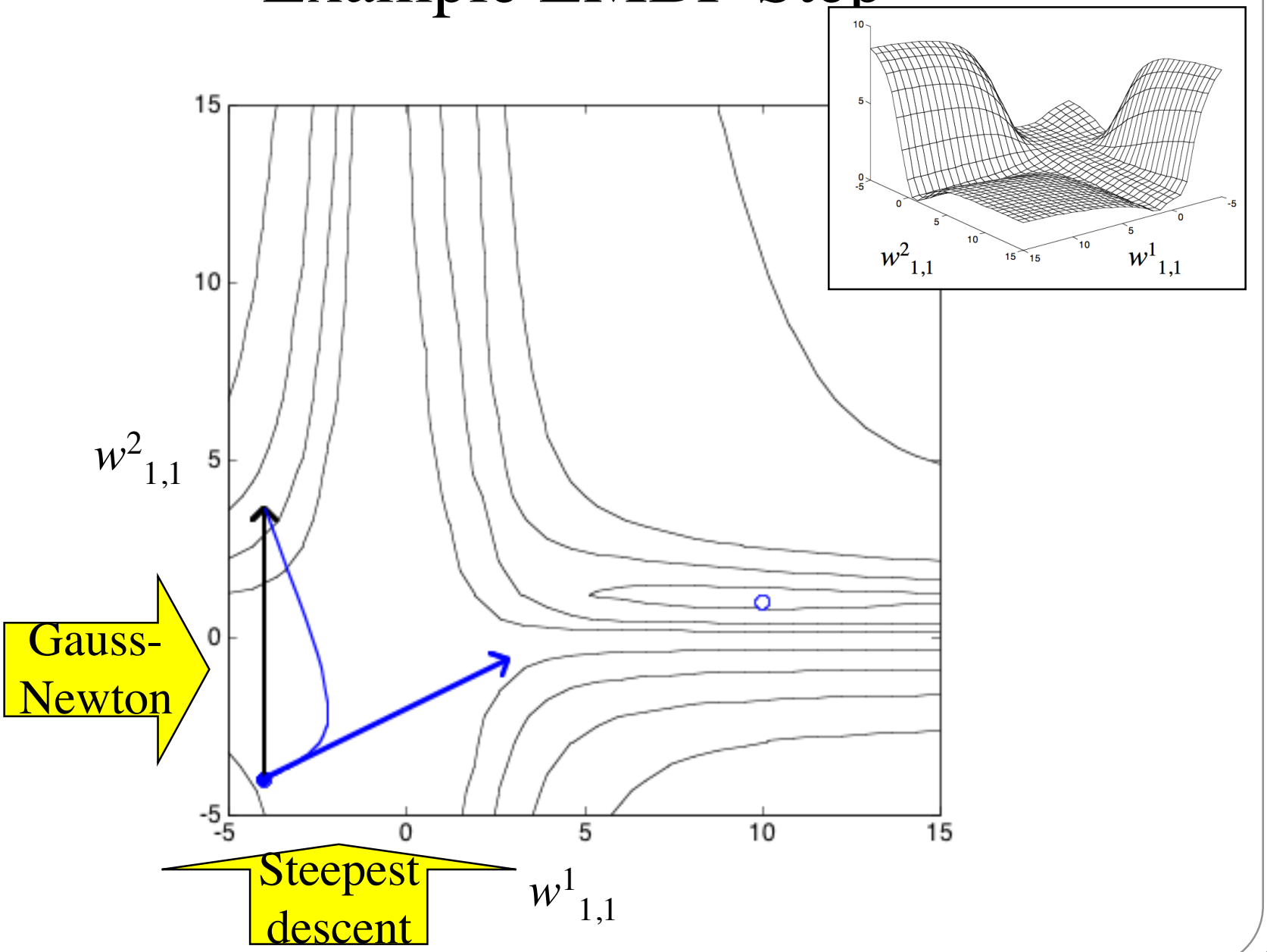
Levenberg-Marquardt Backpropagation Summary

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

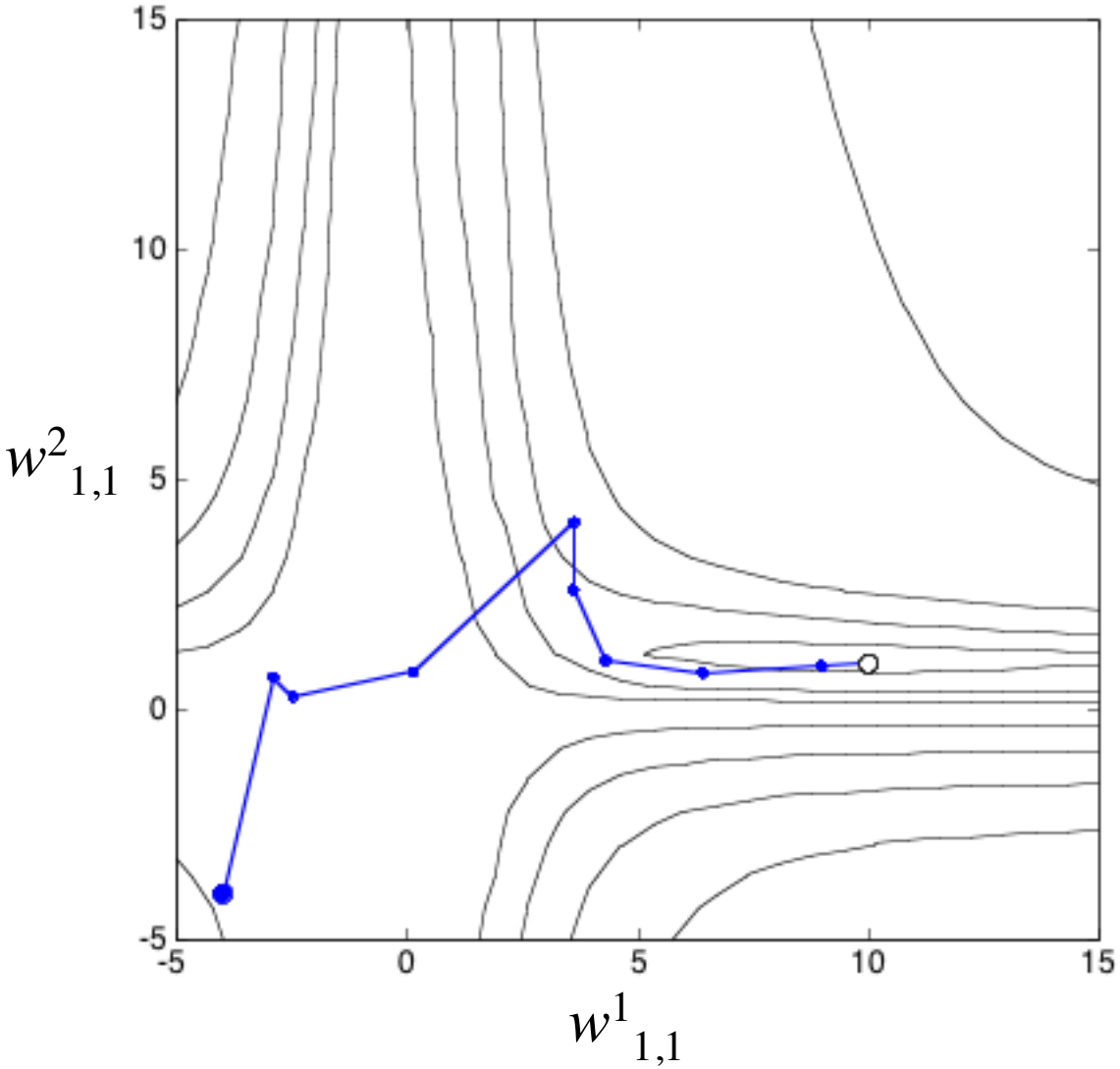
$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

   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 μ_k by υ, update the weights and go back to step 1. If the sum of squares is not reduced, then multiply μ_k by υ and go back to step 3.

Example LMBP Step



LMBP Trajectory



A Few Examples of LM Training for NN Applications

World Academy of Science, Engineering and Technology 3 2005

Levenberg-Marquardt Algorithm for Karachi Stock Exchange Share Rates Forecasting

Syed Muhammad Aqil Burney, Tahseen Ahmed Jilani, Cemal Ardil

Quick and reliable diagnosis of stomach cancer by artificial neural network

Saeid Afshar [1], Fahime Abdolrahmani[2], Fereshte vakili tanha[2], Mahin Zohdi seaf[2], Kobra Taheri[2]

[1]Department of biophysics and biochemistry, faculty of science, Tarbiat Modares University, Tehran, **Iran**

[2]PNU, 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.

Rprop (Resilient Backpropagation)

From Wikipedia, the free encyclopedia:

Rprop, short for resilient backpropagation, is a learning heuristic for supervised learning in feedforward artificial neural networks. This is a first-order optimization algorithm. This algorithm was created by Martin Riedmiller and Heinrich Braun in 1992.

Similarly to the Manhattan update rule, 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 minimise 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' $\Delta_{ij}^{(t)}$:

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

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 Δ, 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].

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

$$\text{where } 0 < \eta^- < 1 < \eta^+$$

Rprop Algorithm

$\forall i,j : \triangle_{ij}(t) = \triangle_0$

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

Repeat

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

  For all weights and biases{

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

      $\triangle_{ij}(t) =$ **minimum** $(\triangle_{ij}(t-1) * \eta^+, \triangle_{max})$

      $\triangle w_{ij}(t) = -$ **sign** $(\frac{\partial E}{\partial w_{ij}}(t)) * \triangle_{ij}(t)$

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

      $\frac{\partial E}{\partial w_{ij}}(t-1) = \frac{\partial E}{\partial w_{ij}}(t)$

    }

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

      $\triangle_{ij}(t) =$ **maximum** $(\triangle_{ij}(t-1) * \eta^-, \triangle_{min})$

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

    }

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

      $\triangle w_{ij}(t) = -$ **sign** $(\frac{\partial E}{\partial w_{ij}}(t)) * \triangle_{ij}(t)$

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

      $\frac{\partial E}{\partial w_{ij}}(t-1) = \frac{\partial E}{\partial w_{ij}}(t)$

    }

  }

Until (converged)

sign(x) = x > 0? +1 : x < 0? -1 : 0

Rprop Parameters

The Rprop algorithm takes two parameters: the initial update-value $\triangle_0$ and a limit for the maximum step size, $\triangle_{max}$.

When learning starts, all update-values are set to an initial value $\triangle_0$. Since $\triangle_0$ directly determines the size of the first weight step, it should be chosen according to the initial values of the weights themselves, for example $\triangle_0 = 0.1$ (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, $\triangle_{max}$. The default upper bound is set somewhat arbitrarily to $\triangle_{max} = 50.0$. 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 $\triangle_{min} = 1e^{-6}$.

Backprop Variations in Matlab NN Toolbox

Function name	Algorithm
trainb	Batch training with weight & bias learning rules
trainbfg	BFGS quasi-Newton backpropagation
trainbr	Bayesian regularization
trainc	Cyclical order incremental training w/learning functions
traincgb	Powell -Beale conjugate gradient backpropagation
traincgf	Fletcher-Powell conjugate gradient backpropagation
traincgp	Polak-Ribiere conjugate gradient backpropagation
traingd	Gradient descent backpropagation
traingdm	Gradient descent with momentum backpropagation
traingda	Gradient descent with adaptive lr backpropagation
traingdx	Gradient descent w/momentum & adaptive lr backpropagation
trainlm	Levenberg-Marquardt backpropagation
trainoss	One step secant backpropagation
trainr	Random order incremental training w/learning functions
trainrp	Resilient backpropagation (Rprop)
trains	Sequential order incremental training w/learning functions
trainscg	Scaled conjugate gradient backpropagation

from http://www.cs.ucr.edu/~vladimir/cs171/nn_summary.pdf

Speed Comparisons (Matlab Implementation)

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.

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