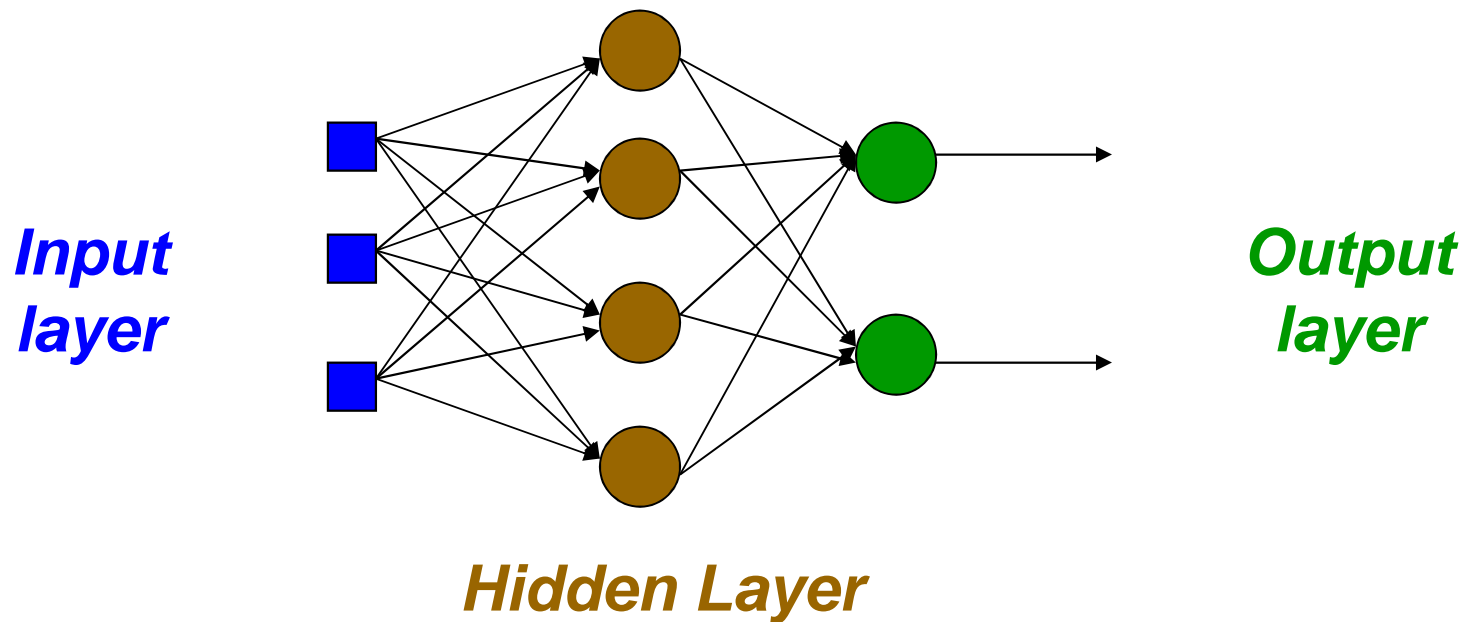
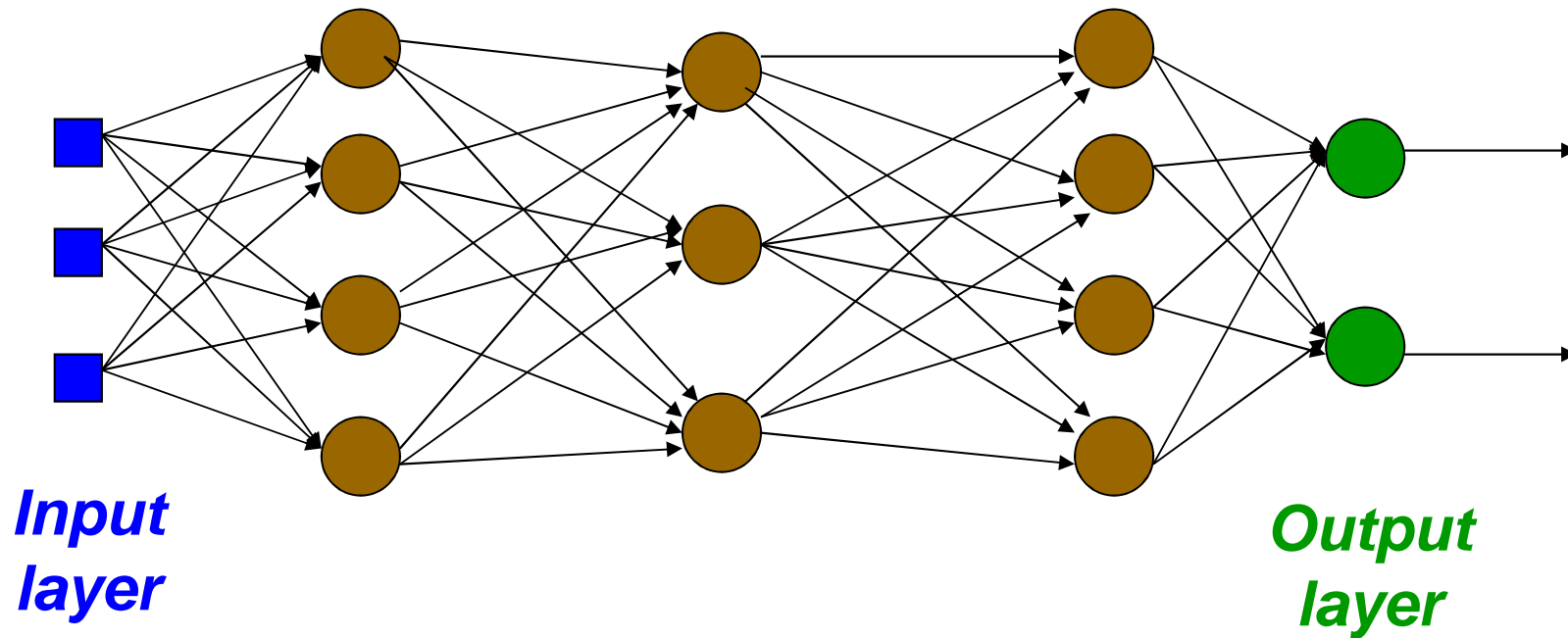


Multi-layer Perceptron (MLP) and Backpropagation

- Single perceptrons = linear decision boundary
- The XOR problem cannot be solved by a single perceptron (*proof?*)
- MLPs overcome the limitation of single-layer perceptrons
- How to train them ?



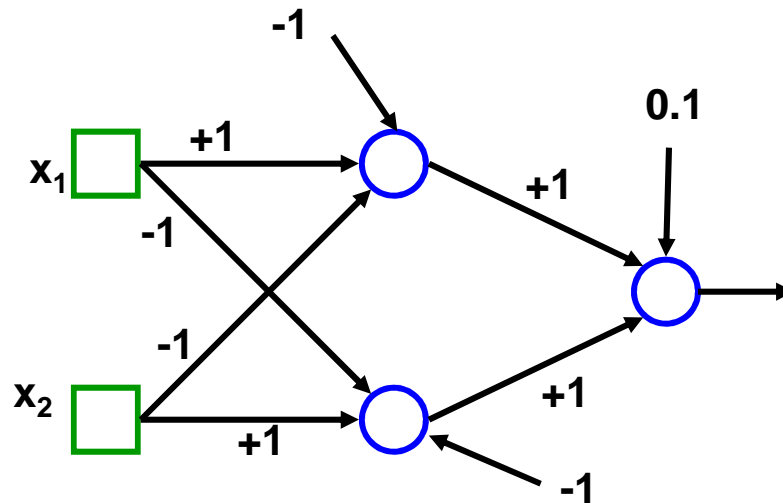
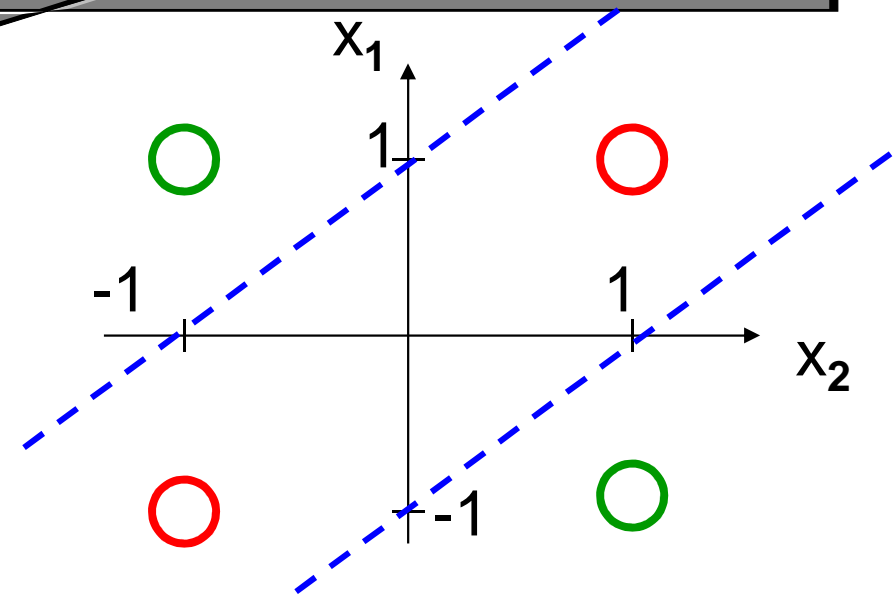
A 3:4:3:4:2 network



Hidden Layer 1 Hidden Layer 2 Hidden Layer 3

A solution for the XOR problem

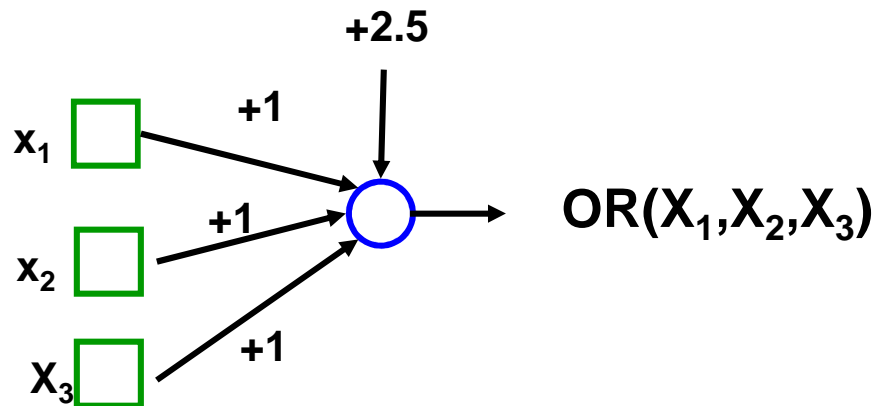
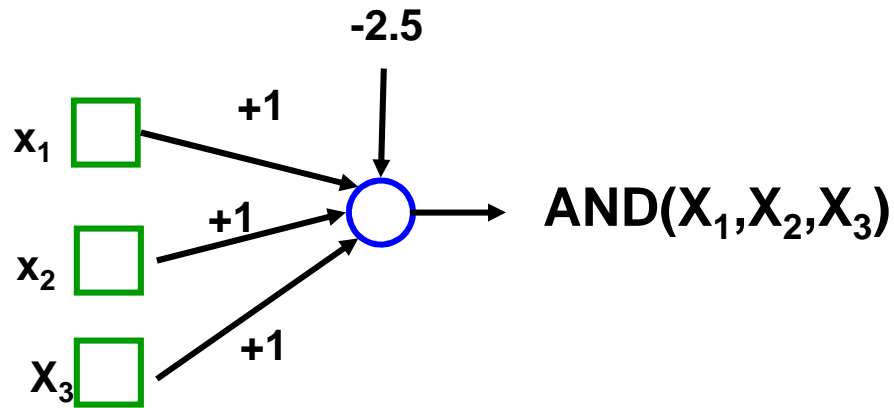
x_1	x_2	$x_1 \text{ xor } x_2$
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1



$$\phi(v) = \begin{cases} 1 & \text{if } v > 0 \\ -1 & \text{if } v \leq 0 \end{cases}$$

ϕ is the sign function.

Generalized AND and OR

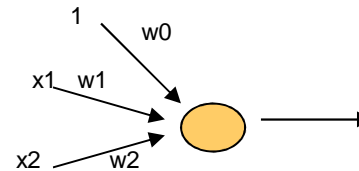


Generalized AND (OR) can be computed by a single perceptron!

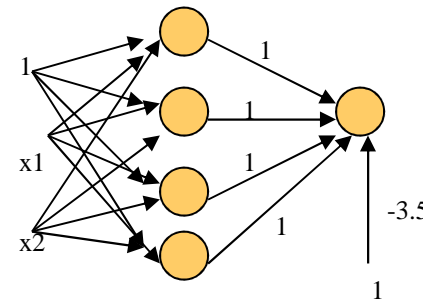
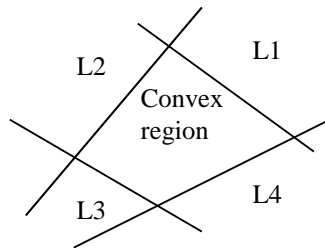
Types of decision regions

$$w_0 + w_1x_1 + w_2x_2 > 0$$

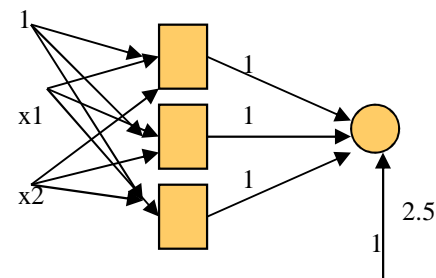
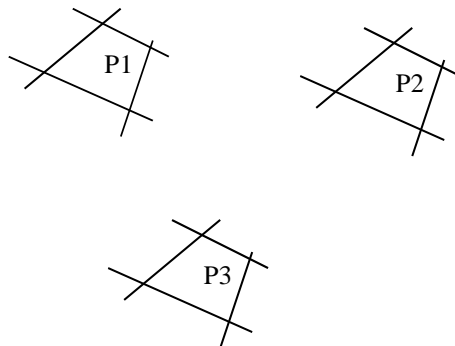
$$w_0 + w_1x_1 + w_2x_2 < 0$$



Network with a single node

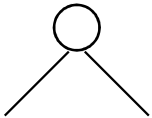
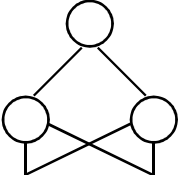
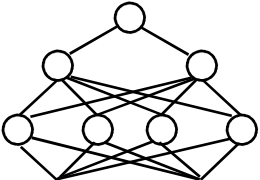


One-hidden layer network that realizes the convex region: each hidden node realizes one of the lines bounding the convex region



two-hidden layer network that realizes the union of three convex regions: each box represents a one hidden layer network realizing one convex region

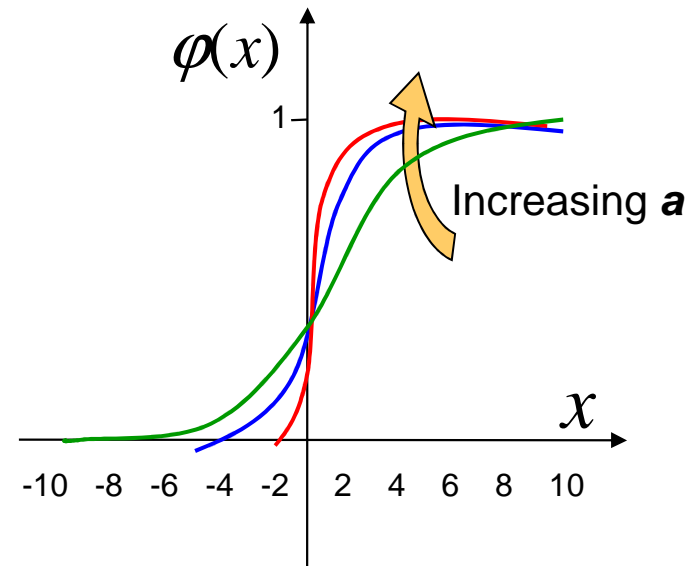
Different Non-Linearly Separable Problems

Structure	Types of Decision Regions	Exclusive-OR Problem	Classes with Meshed regions	Most General Region Shapes
Single-Layer 	Half Plane Bounded By Hyperplane			
Two-Layer 	Convex Open Or Closed Regions			
Three-Layer 	Arbitrary (Complexity Limited by No. of Nodes)			

How to train multi-layer networks?

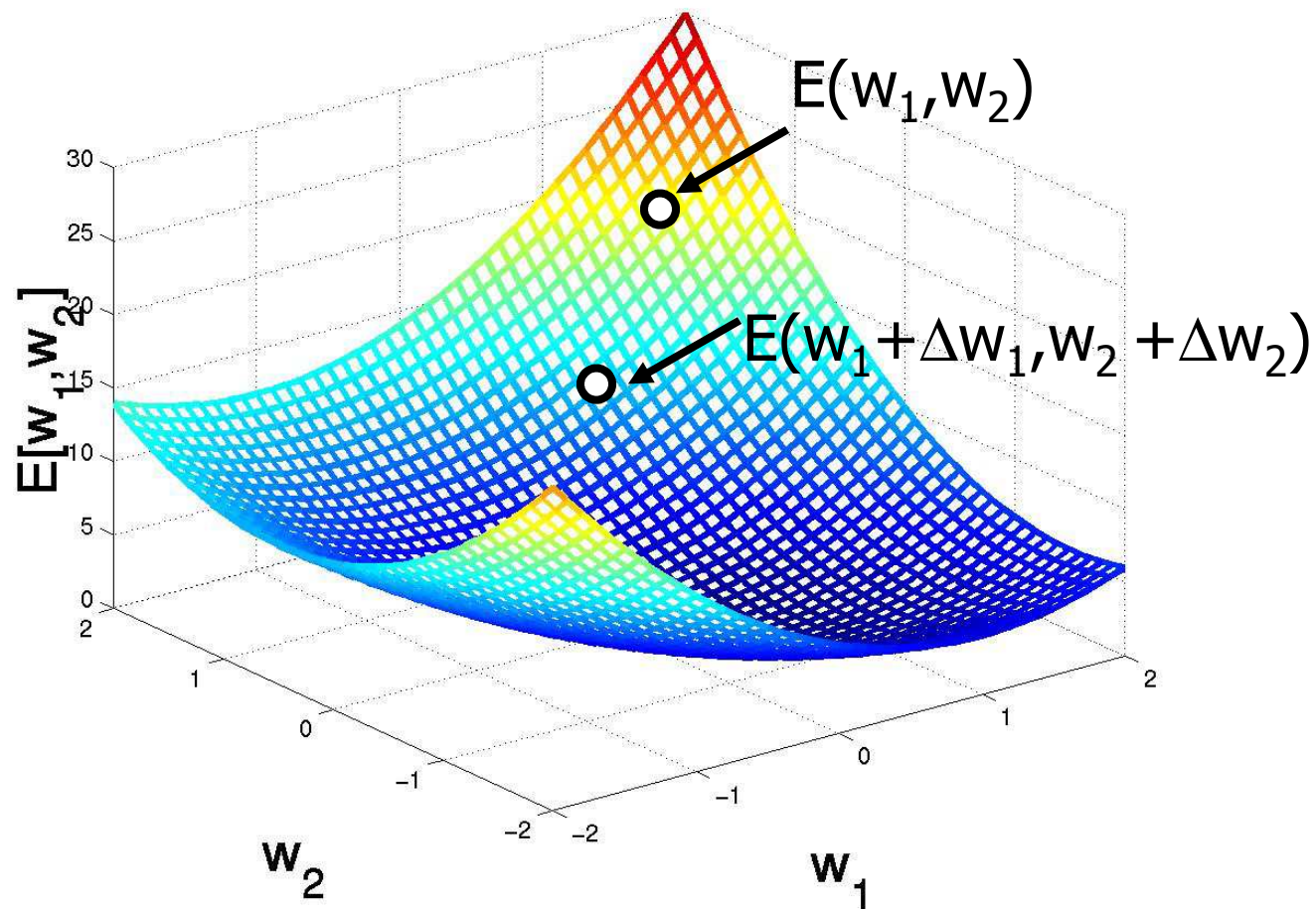
- Main Idea:
Replace the sign function by its “smooth approximation” and use the gradient descent algorithm to find weights that minimize the error (as with ADALINE)

$$\varphi(x) = \frac{1}{1+e^{-ax}} \text{ with } a > 0$$



- The function to be optimized is a complicated one, with many unknowns and nestings ...
- FORTUNATELY: the final update rules are simple !!!

Gradient Descent



Weight Update Rule

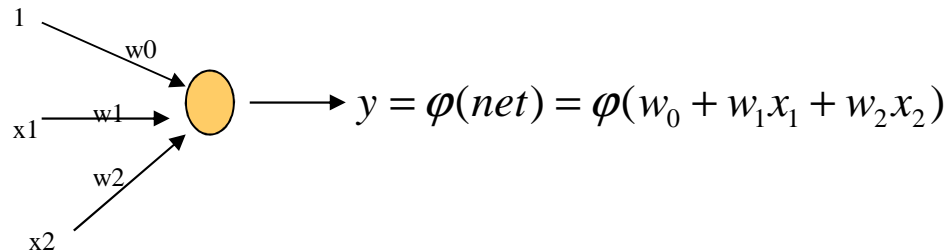
gradient descent method: “walk” in the direction yielding the maximum decrease of the network error E . This direction is the opposite of the gradient of E .

$$w_{ji} = w_{ji} + \Delta w_{ji}$$

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

Delta Rule: one node

- Derive the Delta rule for the following network



$$E(w_0, w_1, w_2) = \frac{1}{2}(y - d)^2 = \frac{1}{2}(\varphi(w_0 + w_1x_1 + w_2x_2) - d)^2$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \text{ for } i \text{ in } \{0,1,2\}$$

- We need to find $\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}$

Delta Rule: one node

$$\begin{aligned}\frac{\partial E(w_0, w_1, w_2)}{\partial w_1} &= \frac{1}{2} \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)^2}{\partial w_1} \\&= \frac{1}{2} 2(\varphi(w_0 + w_1 x_1 + w_2 x_2) - d) \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)}{\partial w_1} \\&= (\varphi(net) - d) \varphi'(net) \frac{\partial (w_0 + w_1 x_1 + w_2 x_2)}{\partial w_1} \\&= (\varphi(net) - d) \varphi'(net) x_1\end{aligned}$$

- From similar calculations we get:

$$\frac{\partial E(w_0, w_1, w_2)}{\partial w_2} = (\varphi(net) - d) \varphi'(net) x_2$$

- and $\frac{\partial E(w_0, w_1, w_2)}{\partial w_0} = (\varphi(net) - d) \varphi'(net)$

Delta Rule: one node

Thus the update rules for the weights are:

$$\Delta w_0 = \eta (d - \varphi(\text{net})) \varphi'(\text{net}) \cdot 1$$

$$\Delta w_1 = \eta (d - \varphi(\text{net})) \varphi'(\text{net}) x_1$$

$$\Delta w_2 = \underbrace{\eta (d - \varphi(\text{net}))}_{\text{delta}} \underbrace{\varphi'(\text{net}) x_2}_{\text{input}}$$

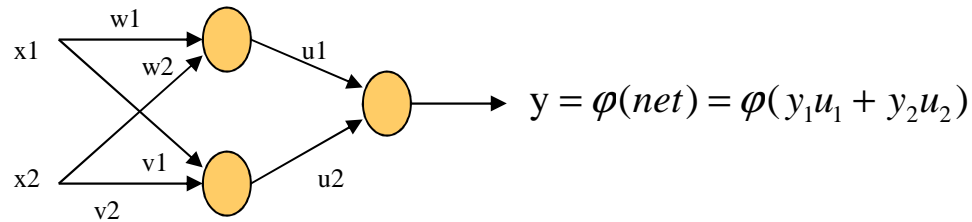
It's handy to know that for the logistic sigmoid function:

$\varphi(x) = 1/(1+\exp(-x))$ we have: $\varphi'(x) = \varphi(x)(1-\varphi(x)) = \text{output}(1-\text{output})$,

so once we know $\varphi(x)$ we also know $\varphi'(x)$!

Delta Rule: XOR-network

Consider the network (no biases):



$$\text{net}_1 = w_1 x_1 + w_2 x_2, \quad y_1 = \varphi(\text{net}_1)$$

$$\text{net}_2 = v_1 x_1 + v_2 x_2, \quad y_2 = \varphi(\text{net}_2)$$

$$\text{net} = y_1 u_1 + y_2 u_2, \quad y = \varphi(\text{net})$$

$$= \varphi(y_1 u_1 + y_2 u_2) =$$

$$= \varphi(\varphi(\text{net}_1) u_1 + \varphi(\text{net}_2) u_2) =$$

$$= \varphi(\varphi(w_1 x_1 + w_2 x_2) u_1 + \varphi(v_1 x_1 + v_2 x_2) u_2)$$

Derivation of the Delta Rule

$$\begin{aligned}\frac{\partial E}{\partial u_1} &= \frac{1}{2} \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)^2}{\partial u_1} = \frac{1}{2} 2(\varphi(y_1 u_1 + y_2 u_2) - d) \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)}{\partial u_1} \\ &= (y - d) \varphi'(net) \frac{\partial (y_1 u_1 + y_2 u_2)}{\partial u_1} = (y - d) \varphi'(net) y_1\end{aligned}$$

From similar calculations we get: $\frac{\partial E}{\partial u_2} = (y - d) \varphi'(net) y_2$

$$\begin{aligned}\frac{\partial E}{\partial w_1} &= \frac{1}{2} \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)^2}{\partial w_1} = \frac{1}{2} 2(\varphi(y_1 u_1 + y_2 u_2) - d) \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)}{\partial w_1} \\ &= (y - d) \varphi'(net) \frac{\partial (y_1 u_1 + y_2 u_2)}{\partial w_1}\end{aligned}$$

Derivation of the Delta Rule

$$\begin{aligned}\frac{\partial(y_1 u_1 + y_2 u_2)}{\partial w_1} &= \frac{\partial(y_1 u_1)}{\partial w_1} = u_1 \frac{\partial y_1}{\partial w_1} = u_1 \frac{\partial(\varphi(w_1 x_1 + w_2 x_2))}{\partial w_1} \\ &= u_1 \varphi'(net_1) \frac{\partial(w_1 x_1)}{\partial w_1} = u_1 \varphi'(net_1) x_1\end{aligned}$$

So we obtain:

$$\begin{aligned}\frac{\partial E}{\partial w_1} &= (y - d) \varphi'(net) \frac{\partial(\varphi(y_1 u_1 + y_2 u_2))}{\partial w_1} \\ &= (y - d) \varphi'(net) u_1 \varphi'(net_1) x_1\end{aligned}$$

From similar calculations we get:

$$\begin{aligned}\frac{\partial E}{\partial w_2} &= (y - d) \varphi'(net) u_1 \varphi'(net_1) x_2 \\ \frac{\partial E}{\partial v_1} &= (y - d) \varphi'(net) u_2 \varphi'(net_2) x_1, \quad \frac{\partial E}{\partial v_2} = (y - d) \varphi'(net) u_2 \varphi'(net_2) x_2\end{aligned}$$

delta

General case: Generalized Delta Rule

$\Delta w_{ji} = \eta \delta_j y_i$, where:

$$\delta_j = \begin{cases} \phi'(v_j)(d_j - y_j) & \text{IF } j \text{ output node} \\ \phi'(v_j) \sum_{\text{k of nextlayer}} \delta_k w_{kj} & \text{IF } j \text{ hidden node} \end{cases}$$

w_{ji} : weight from node j to node i (moving from output to input!)

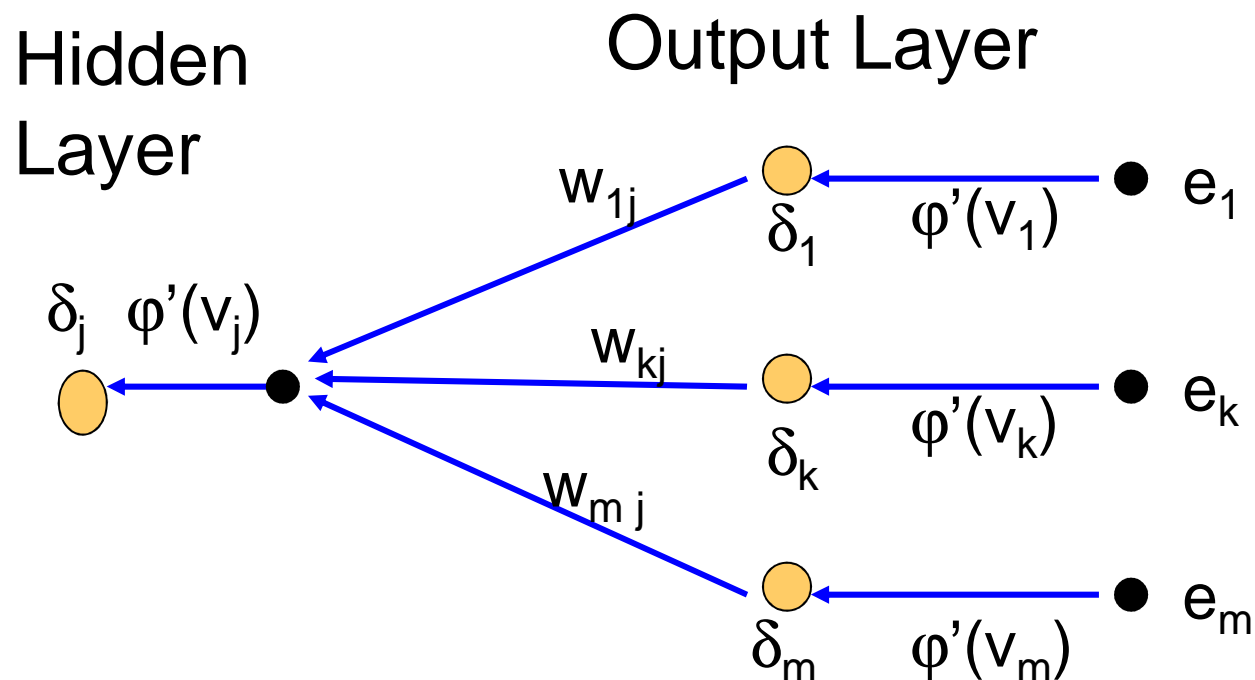
η : learning rate (constant)

y_j : output of node j

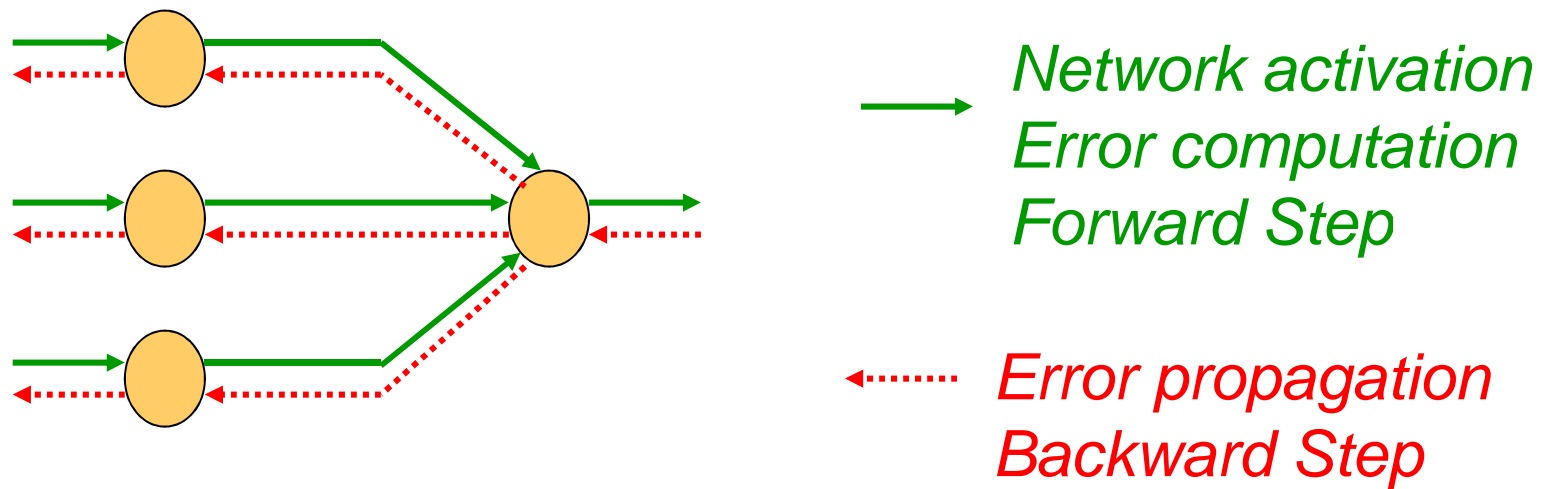
v_j : activation of node j

Error backpropagation

The flow-graph below illustrates how errors are back-propagated from m output nodes to the hidden neuron j



Backpropagation: two phases



Backpropagation algorithm

- The Backprop algorithm searches for weight values that **minimize the total error of the network**
- Backprop consists of the **repeated** application of the following two passes:
 - **Forward pass**: in this step the network is activated on one example and the **error** of each neuron of the output layer is **computed**.
 - **Backward pass**: in this step the network error is used for updating the weights. Starting at the output layer, **the error is propagated backwards through the network, layer by layer** with help of the generalized delta rule. Finally, all weights are updated.
- No guarantees of convergence
(when learning rate too big or too small)
- In case of convergence: local (or global) minimum
- In practice: try several starting configurations and learning rates.

Network training:

Two types of network training:

- **Incremental mode** (on-line, stochastic, or per-pattern)
Weights updated after each pattern is presented
- **Batch mode** (off-line or per -epoch)
Weights updated after all the patterns are presented

Advantages and disadvantages of different modes

Sequential mode:

- Less storage for each weighted connection
- Random order of presentation and updating per pattern means search of weight space is stochastic--reducing risk of local minima able to take advantage of any redundancy in training set (i.e.. same pattern occurs more than once in training set, esp. for large training sets)
- Simpler to implement

Batch mode:

- Faster learning than sequential mode

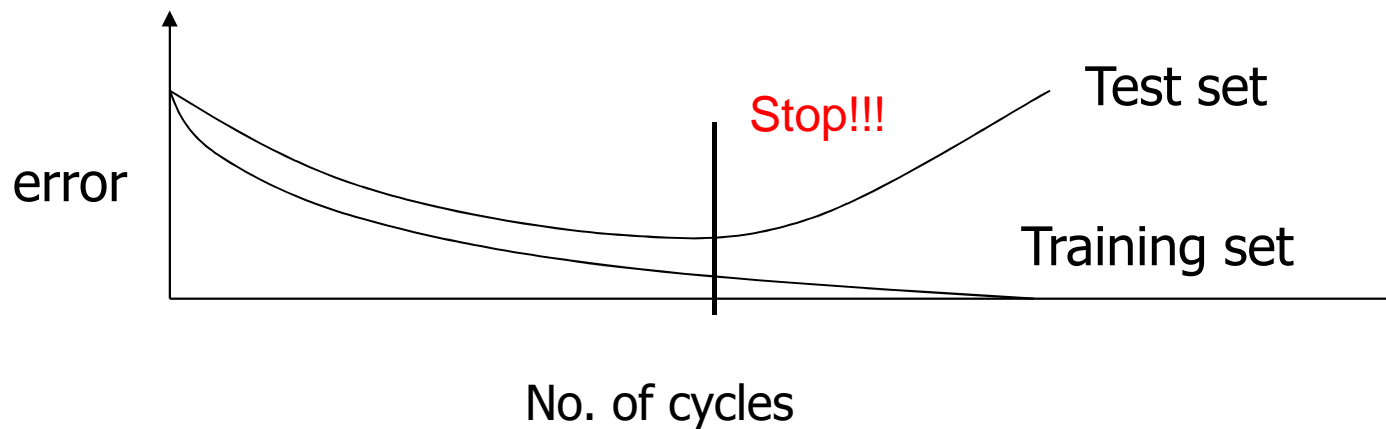
Stopping criteria

- **total mean squared error change**: Backprop is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small
- **generalization based criterion**:
After each epoch the NN is tested for generalization using a different set of examples (test set). If the generalization performance is adequate then stop. (**Early Stopping**)

Early Stopping

Training network for too many cycles may lead to **data overfitting** (or “overtraining”)

Early Stopping:
stop training as soon as the error on the test set increases



Early stopping and 3 sets:

- **Training set** – used for training
- **Test set** – used to decide when to stop training by monitoring the error on it
- **Validation set** – used for final estimation of the performance of the trained neural network
(using the test set for it is methodologically not correct)

In practice, the use of validation set is not so important!

Model Selection by Cross-validation

- **Too few hidden units** may prevent the network from learning adequately the data and learning the concept.
- **Too many hidden units** leads to overfitting.
- A **cross-validation scheme** can be used to determine an appropriate number of hidden units by using the optimal test error to select the model with optimal number of hidden layers and nodes.
- N-fold cross-validation:
 1. split your data into N parts (equal size);
 2. develop N networks on all combinations of (N-1) parts;
 3. test each network on the remaining parts (test sets);
 4. average the error over these test sets.

Expressive Power of MLP

Boolean functions:

- **Every boolean function** can be described by a network with a **single hidden layer**
- but it might require **exponential** (in the number of inputs) hidden **neurons**.

Continuous functions:

- **Any bounded continuous function** can be approximated with arbitrarily small error by a network with **two hidden layers**.
- **Stronger: Any bounded continuous function** can be approximated with arbitrarily small error by a network with **one hidden layer**.

Complexity of Learning

Blum, Rivest (1993):

“Training a 3-node neural network is NP-complete”

- “training” = “optimal training”
- 3-nodes, but N k-dimensional input patterns
- “NP-complete” =
“no polynomial-time algorithm exists”
- “in practice, the problem is not solvable”

Optimization Methods (Ch. 7)

- Variable Learning Rate algorithms
 - Generalized Delta Rule (with momentum)
 - Adaptive Gradient Descent
- Numerical Optimization Algorithms
 - Quickpropagation Algorithm
 - Line Search Method (Brent's Algorithm)
 - Conjugate Gradient Algorithms
- Applications

Generalized delta rule

- In classical backpropagation weights are updated in cycles which may lead to some “chaotic” directions of updates (from cycle to cycle or from pattern to pattern).
- Therefore it make sense to keep an extra term that takes care of the “general direction of changes”.
- This is achieved by introducing **a momentum term α** in the delta rule that takes into account previous updates:

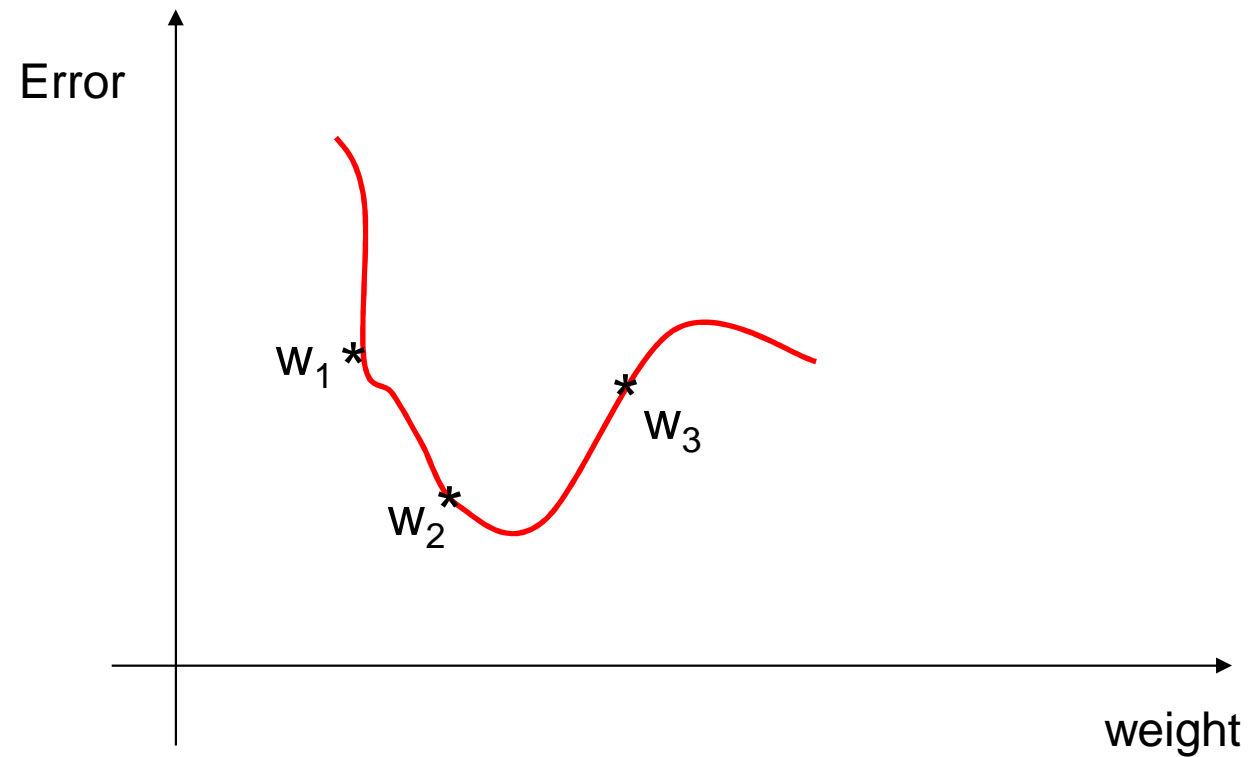
$$\Delta w_{ji}^{\text{new}} = \alpha \Delta w_{ji}^{\text{previous}} + \eta \delta_j y_i$$

Adaptive Gradient Descent

- "classical" backproagation is very slow !!!
 - if learning rate too small \Rightarrow convergence too slow
 - if learning rate too big \Rightarrow no convergence
- Idea: ***dynamically tune the value of the learning rate !***
- **Adaptive Gradient Descent (a bold-driver strategy):**

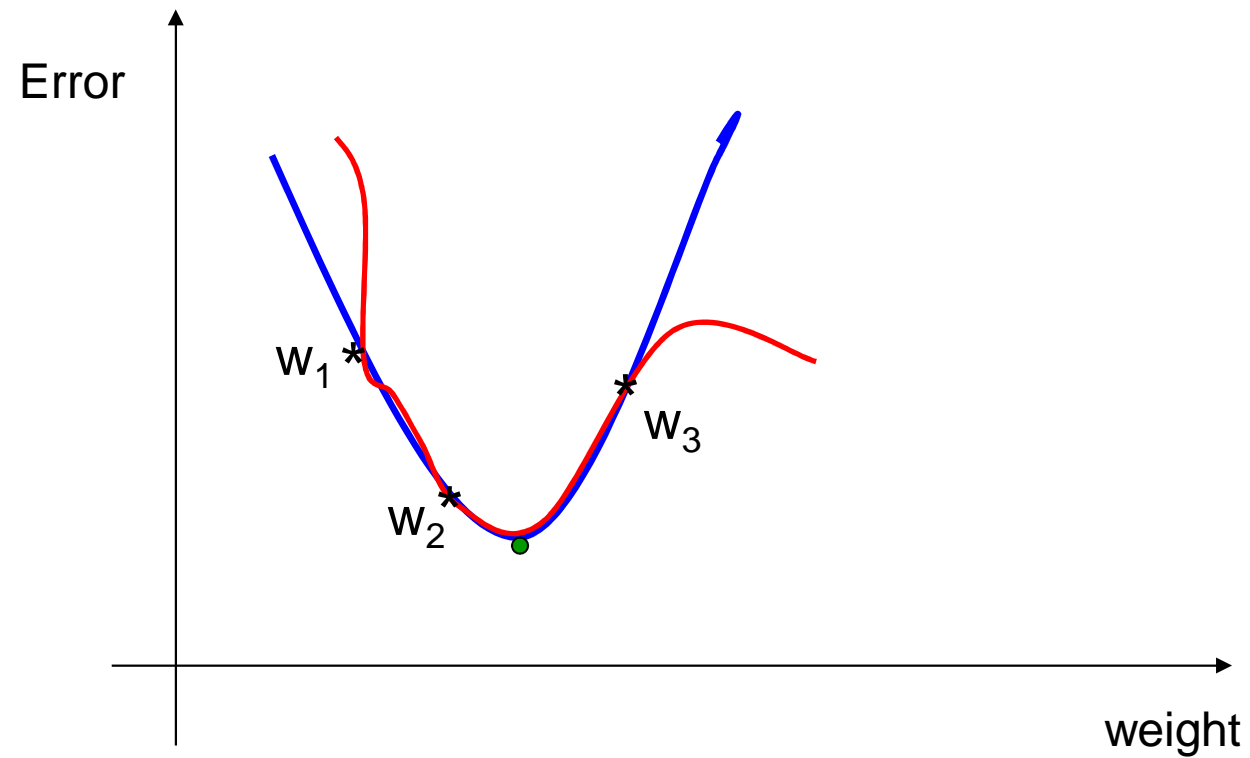
if new_error < old_error	\Rightarrow	increase learning rate $lr = 1.05 * lr;$
if new_error > 1.04 * old.error	\Rightarrow	ignore weight changes; decrease learning rate $lr = 0.7 * lr;$

Quickpropagation (Fahlman)



What should be the next value of w ?

Quickpropagation (Fahlman)



Quickpropagation (Fahlman)

- Main Idea:
 - 'locally' error function \approx paraboloid
 - finding minimum of a paraboloid is simple
 - However: It works only close to local minima !

Algorithm:

- run ordinary backpropagation
- when $E'(t)$ changes its sign or magnitude then apply the “clever” update rule to “the last 3 points”!

Line Search

Idea 1.

Finding minimum of a function of 1 variable is easy !!!
(Brent's Algorithm)

Idea 2. (When you don't have derivatives ...)

Just walk along x_1, x_2, \dots, x_n (selected in a random order, starting from a randomly chosen point) minimizing your function along these dimensions (as long as it takes...)

Idea 3. (When you do have derivatives ...)

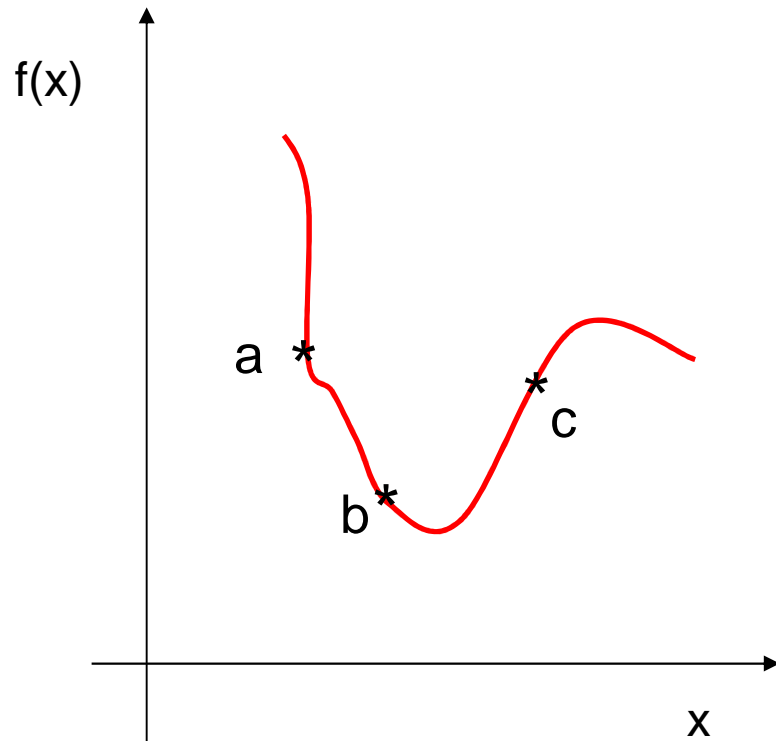
Once we know the direction of steepest descent we can "jump" to the minimum along this direction:

$error(t) = E(W_0 + t * W_{sd})$ - a function of 1 variable t !!!

W_0 - current point

W_{sd} - direction of steepest descent

Brent's Algorithm



A triplet $[a, b, c]$ is called a *bracket* when $f(a) > f(b)$ and $f(c) > f(b)$ (it "brackets" a minimum!)

By selecting d that splits ab or bc we get a new, smaller (!) bracket.

Heuristic 1: Always split the longer segment

Heuristic 2: Instead of using the "1:1" ratio use the *"golden ratio"* or *"parabolic estimate"*

Result: In practice, a minimum is located in 20-30 iterations
($[a, c]$ is shrinking exponentially fast!)

Conjugate Gradients

Main Idea:

Instead of following the "gradient descent directions" it is better to follow "conjugate directions".

Two directions \mathbf{w} and \mathbf{v} are conjugate with respect to a matrix \mathbf{H} if and only if $\mathbf{wHv}=\mathbf{0}$.

In our case:

\mathbf{H} is the matrix of second derivatives (Hessian) of the Error Function

Conjugate directions can be computed without exact knowledge of \mathbf{A} !!!

For a function of n variables, \mathbf{H} is of size n^2 ;

e.g., the "digit-recognition" network has $n=257*10$,
so \mathbf{H} would have **6.604.900** entries!

Conjugate Gradients Algorithm

Start at a randomly selected point x_0

Iterate ***m times*** the following ***n steps*** (n =number of dimensions):

Find d_1 = a direction of steepest descent => line search => optimal point x_1

Find d_2 = a new direction that is conjugate to d_1 at x_1 => line search => optimal point x_2

Find d_3 = a new direction that is conjugate to d_2 at x_2 => line search => optimal point x_3

....

Find d_n = a new direction that is conjugate to d_{n-1} at x_{n-1} => line search => optimal point x_n

For a quadratic function of n variables the minimum is reached after a single iteration!

Optimization based on (approx.) of Hessian



-Conjugate Gradients (various formulas for finding c. directions):

- Hestens-Stiefel**
- Polak-Ribiere**
- Fletcher-Reeves**

-Scaled Conjugate Gradients (Møller) (eliminates the line-search steps)

-Newton method (explicit use of H and its inverse)

-Quasi-Newton (iterative approximation of the inverse of H)

-Levenberg-Marquardt (another way of approx. the inverse of H)

Conclusions

- **backpropagation** is important for historical reasons; in practice **almost useless** (too slow)
- **faster algorithms** for training FF-networks are based on some heuristics for dynamic updates of learning parameters or make use of second derivatives (or their approximations) of the error function; **speedup up to 300 times!**
- some of these algorithms are **not applicable** to big networks or data sets (**memory requirements**)
- some **"manual tuning"** of learning parameters is **always needed**

Example Application: Evolutionary Computing 2002 Challenge

Given a “black-box” function of 30 variables, find a minimum of f asking ***at most*** 1 million times for values of $f(x_1, \dots, x_{30})$.

Find (x_1, \dots, x_{30}) s.t. $f(x_1, \dots, x_{30})=0$

Final grade based on the result:

10 for a solution in $[0, 10^{-5}]$

9 for a solution in $[10^{-5}, 10^{-4}]$

8 for a solution in $[10^{-4}, 10^{-3}]$

...

And the function was: Fletcher-Powell function (m=30)

(11) *Fletcher-Powell function*: This is an m-variable function with $f_{\min}(c_1, c_2, \dots, c_m) = 0$ given as follows:

$$f(x) = \sum_{i=1}^m (A_i - B_i)^2,$$

where $A_i = \sum_{j=1}^m [u_{ij} \sin(c_j) + v_{ij} \cos(c_j)]$; $B_i = \sum_{j=1}^m [u_{ij} \sin(x_j) + v_{ij} \cos(x_j)]$; $u_{ij}, v_{ij} = rand[-100, 100]$

Our approach:



- Spend no more than one evening
- Use conventional optimization methods
- Use standard software (MATLAB+NETLAB)
- No thinking, no parameter tuning

Real objective:

Beat Evolutionary Methods !!!

Classical methods:

- Gradient descent (useless)
 - **Line search**
 - **Conjugate Gradients**
 - **Scaled Conjugate Gradients**
 - **Quasi-Newton**
- Levenberg-Marquard (not applicable)
- Nelder-Mead simplex direct search (???)

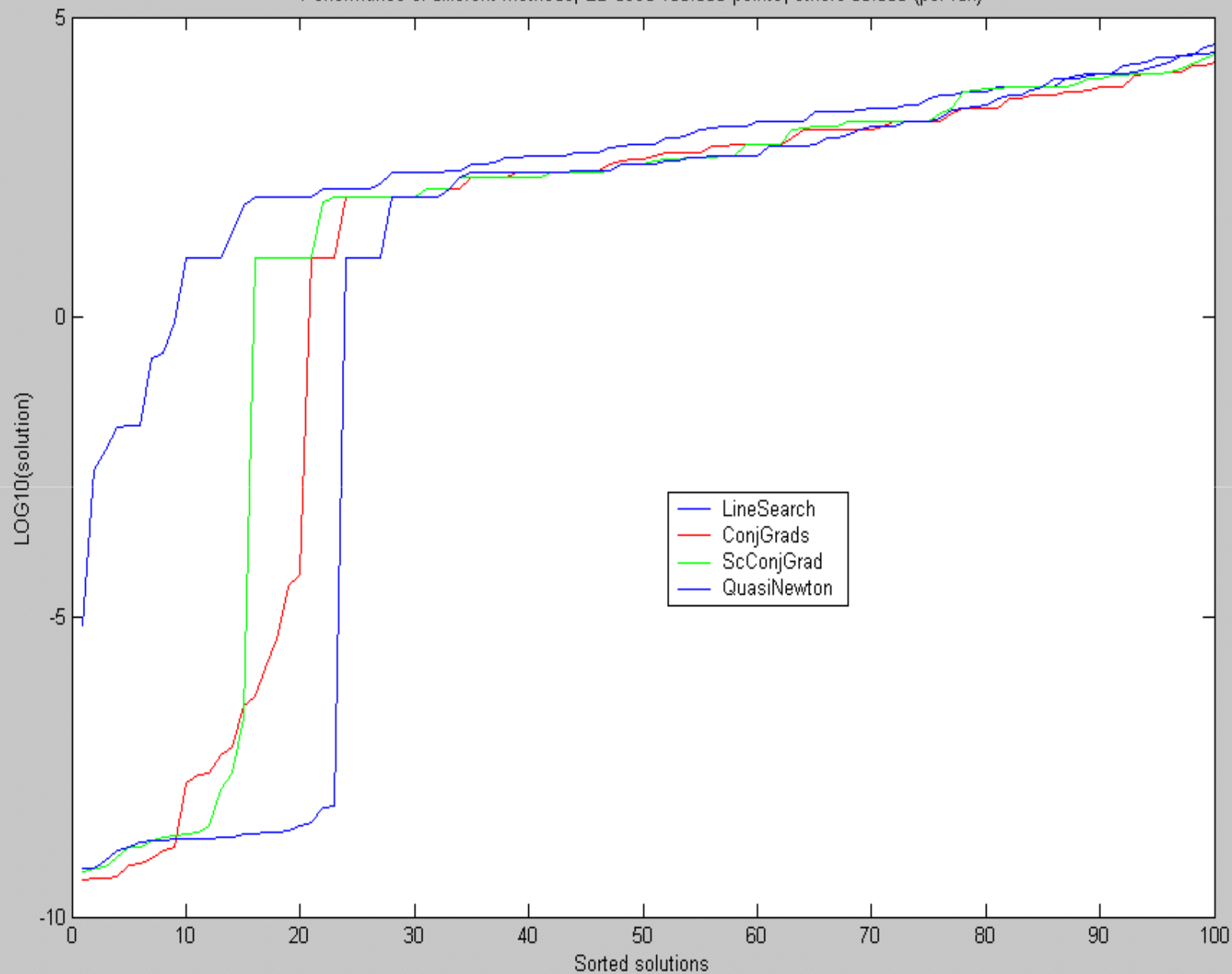
We need partial derivatives!!!

$$\frac{\partial f(x_1, \dots, x_i, \dots, x_{30})}{\partial x_i} \approx \frac{f(x_1, \dots, x_i + \varepsilon, \dots, x_{30}) - f(x_1, \dots, x_i, \dots, x_{30})}{\varepsilon}$$

Finding the gradient vector costs 30 evaluations of f!

But it's worth the effort!!!

Performance of different methods; LS used 150.000 points, others 30.000 (per run)



Results

- Line search sufficient for a “10” but extra work needed:
 - global minimum,
 - dynamic tolerance adjustment,
 - intelligent restart strategy.
- Methods based on second derivatives are 10-100 times more efficient than line search:
 - **100.000 points are enough to get 10!!!**
 - further improvements still possible (parameter tuning, restart strategy, etc.)