Multi Layer Perceptrons

Prof. Dr. Martin Riedmiller
AG Maschinelles Lernen
Albert-Ludwigs-Universität Freiburg

riedmiller@informatik.uni-freiburg.de

Outline

- multi layer perceptrons (MLP)
- learning MLPs
- function minimization: gradient descend & related methods

Neural networks

- single neurons are not able to solve complex tasks (e.g. restricted to linear calculations)
- creating networks by hand is too expensive; we want to learn from data
- nonlinear features also are usually difficult to design by hand

Neural networks

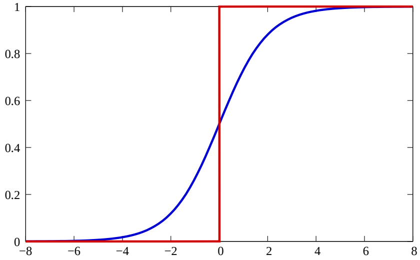
- single neurons are not able to solve complex tasks (e.g. restricted to linear calculations)
- creating networks by hand is too expensive; we want to learn from data
- nonlinear features also are usually difficult to design by hand

- we want to have a generic model that can adapt to some training data
- basic idea: multi layer perceptron (Werbos 1974, Rumelhart, McClelland, Hinton 1986), also named feed forward networks

Neurons in a multi layer perceptron

- standard perceptrons calculate a discontinuous function:

$$\vec{x} \mapsto f_{step}(w_0 + \langle \vec{w}, \vec{x} \rangle)$$



Neurons in a multi layer perceptron

- standard perceptrons calculate a discontinuous function:

$$\vec{x} \mapsto f_{step}(w_0 + \langle \vec{w}, \vec{x} \rangle)$$

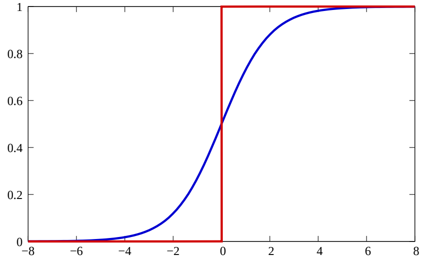
- due to technical reasons, neurons in MLPs calculate a smoothed variant of this:

$$\vec{x} \mapsto f_{log}(w_0 + \langle \vec{w}, \vec{x} \rangle)$$

  with

$$f_{log}(z) = \frac{1}{1 + e^{-z}}$$

  f_{log} is called logistic function



Neurons in a multi layer perceptron

- standard perceptrons calculate a discontinuous function:

$$\vec{x} \mapsto f_{step}(w_0 + \langle \vec{w}, \vec{x} \rangle)$$

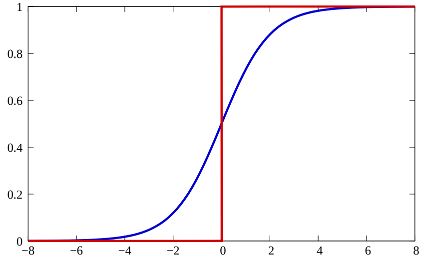
- due to technical reasons, neurons in MLPs calculate a smoothed variant of this:

$$\vec{x} \mapsto f_{log}(w_0 + \langle \vec{w}, \vec{x} \rangle)$$

with

$$f_{log}(z) = \frac{1}{1 + e^{-z}}$$

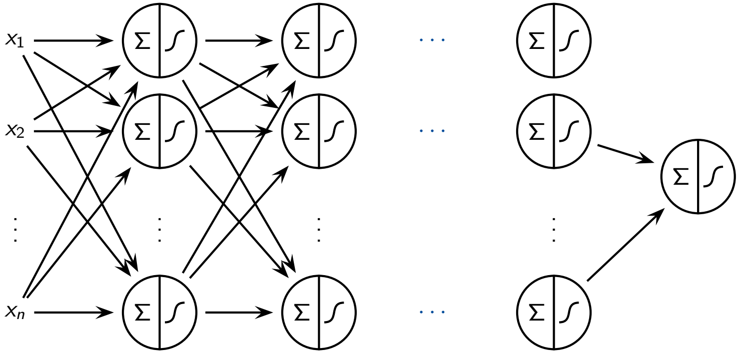
f_{log} is called logistic function



- properties:
  - monotonically increasing
  - $\lim_{z\to\infty} = 1$
  - $\lim_{z\to-\infty} = 0$
  - $f_{log}(z) = 1 - f_{log}(-z)$
  - continuous, differentiable

Multi layer perceptrons

- A multi layer perceptrons (MLP) is a finite acyclic graph. The nodes are neurons with logistic activation.



- neurons of i-th layer serve as input features for neurons of $i+1$th layer
- very complex functions can be calculated combining many neurons

Multi layer perceptrons (cont.)

- multi layer perceptrons, more formally:
  A MLP is a finite directed acyclic graph.
  - nodes that are no target of any connection are called input neurons. A MLP that should be applied to input patterns of dimension n must have n input neurons, one for each dimension. Input neurons are typically enumerated as neuron 1, neuron 2, neuron 3, ...
  - nodes that are no source of any connection are called output neurons. A MLP can have more than one output neuron. The number of output neurons depends on the way the target values (desired values) of the training patterns are described.
  - all nodes that are neither input neurons nor output neurons are called hidden neurons.
  - since the graph is acyclic, all neurons can be organized in layers, with the set of input layers being the first layer.

Multi layer perceptrons (cont.)

- connections that hop over several layers are called shortcut
- most MLPs have a connection structure with connections from all neurons of one layer to all neurons of the next layer without shortcuts
- all neurons are enumerated
- $Succ(i)$ is the set of all neurons j for which a connection $i \rightarrow j$ exists
- $Pred(i)$ is the set of all neurons j for which a connection $j \rightarrow i$ exists
- all connections are weighted with a real number. The weight of the connection $i \rightarrow j$ is named w_{ji}
- all hidden and output neurons have a bias weight. The bias weight of neuron i is named w_{i0}

Multi layer perceptrons (cont.)

- variables for calculation:
  - hidden and output neurons have some variable net_i ("network input")
  - all neurons have some variable a_i ("activation" / "output")

Multi layer perceptrons (cont.)

- variables for calculation:
  - hidden and output neurons have some variable net_i ("network input")
  - all neurons have some variable a_i ("activation"/"output")
- applying a pattern $\vec{x} = (x_1, \ldots, x_n)^T$ to the MLP:
  - for each input neuron the respective element of the input pattern is presented, i.e. $a_i \leftarrow x_i$

Multi layer perceptrons (cont.)

- variables for calculation:
  - hidden and output neurons have some variable net_i ("network input")
  - all neurons have some variable a_i ("activation" / "output")
- applying a pattern $\vec{x} = (x_1, \dots, x_n)^T$ to the MLP:
  - for each input neuron the respective element of the input pattern is presented, i.e. $a_i \leftarrow x_i$
  - for all hidden and output neurons i:
    after the values a_j have been calculated for all predecessors $j \in Pred(i)$, calculate net_i and a_i as:

$$net_i \leftarrow w_{i0} + \sum_{j \in Pred(i)} (w_{ij} a_j)$$

$$a_i \leftarrow f_{log}(net_i)$$

Multi layer perceptrons (cont.)

- variables for calculation:
  - hidden and output neurons have some variable net_i ("network input")
  - all neurons have some variable a_i ("activation" / "output")
- applying a pattern $\vec{x} = (x_1, \ldots, x_n)^T$ to the MLP:
  - for each input neuron the respective element of the input pattern is presented, i.e. $a_i \leftarrow x_i$
  - for all hidden and output neurons i:
    after the values a_j have been calculated for all predecessors $j \in Pred(i)$, calculate net_i and a_i as:

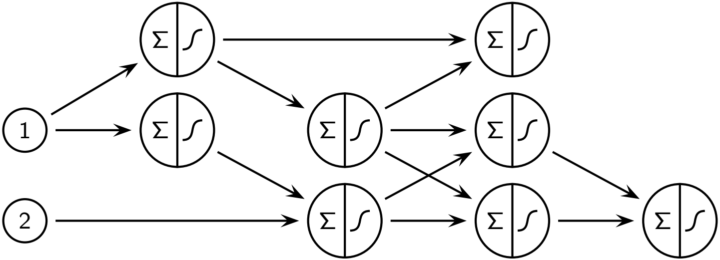
$$net_i \leftarrow w_{i0} + \sum_{j \in Pred(i)} (w_{ij} a_j)$$

$$a_i \leftarrow f_{log}(net_i)$$

  - the network output is given by the a_i of the output neurons

Multi layer perceptrons (cont.)

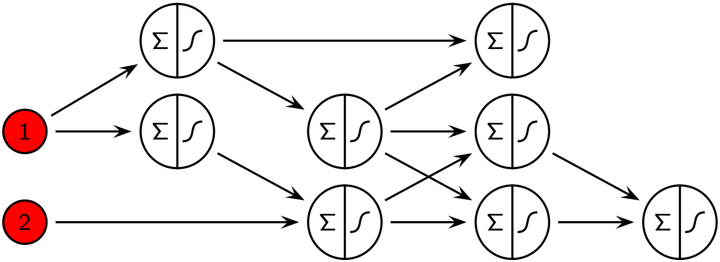
- illustration:



- apply pattern $\vec{x} = (x_1, x_2)^T$

Multi layer perceptrons (cont.)

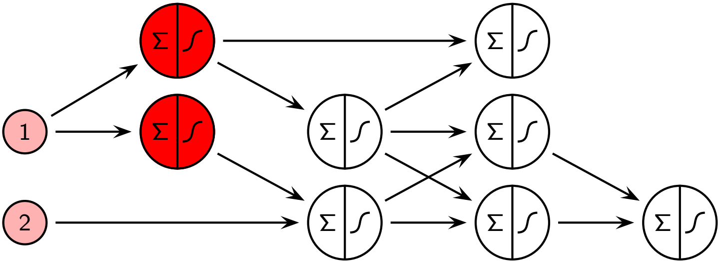
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - calculate activation of input neurons: $a_i \leftarrow x_i$

Multi layer perceptrons (cont.)

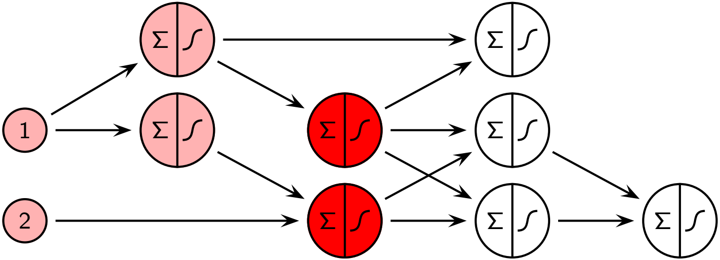
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - calculate activation of input neurons: $a_i \leftarrow x_i$
  - propagate forward the activations:

Multi layer perceptrons (cont.)

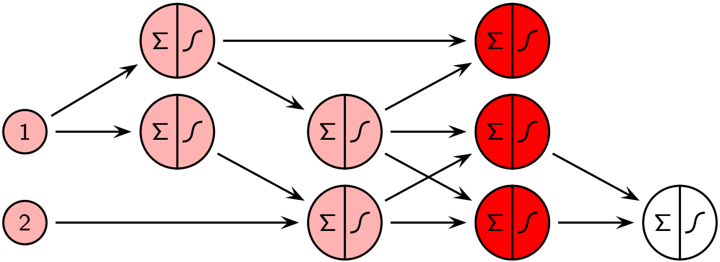
- illustration:



- apply pattern $\vec{x} = (x_1, x_2)^T$
- calculate activation of input neurons: $a_i \leftarrow x_i$
- propagate forward the activations: step

Multi layer perceptrons (cont.)

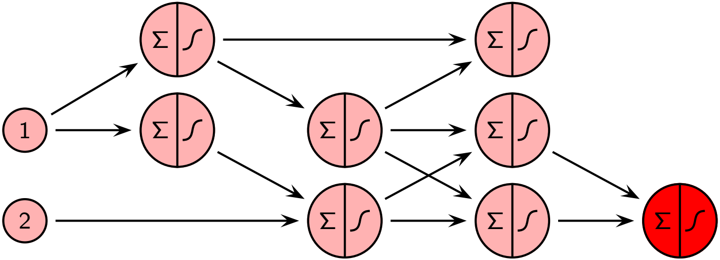
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - calculate activation of input neurons: $a_i \leftarrow x_i$
  - propagate forward the activations: step by

Multi layer perceptrons (cont.)

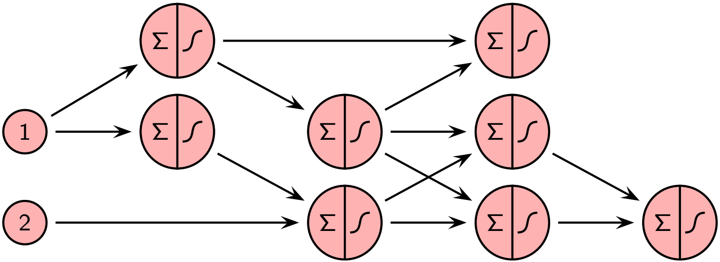
- illustration:



- apply pattern $\vec{x} = (x_1, x_2)^T$
- calculate activation of input neurons: $a_i \leftarrow x_i$
- propagate forward the activations: step by step

Multi layer perceptrons (cont.)

- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - calculate activation of input neurons: $a_i \leftarrow x_i$
  - propagate forward the activations: step by step
  - read the network output from both output neurons

Multi layer perceptrons (cont.)

- algorithm (forward pass):

Require: pattern $\vec{x}$, MLP, enumeration of all neurons in topological order
Ensure: calculate output of MLP

for all input neurons i **do**
  set $a_i \leftarrow x_i$
end for
for all hidden and output neurons i in topological order **do**
  set $net_i \leftarrow w_{i0} + \sum_{j \in Pred(i)} w_{ij} a_j$
  set $a_i \leftarrow f_{log}(net_i)$
end for
for all output neurons i **do**
  assemble a_i in output vector $\vec{y}$
end for
return $\vec{y}$

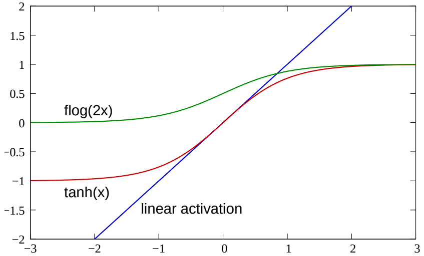
Multi layer perceptrons (cont.)

- variant:
  Neurons with logistic activation can only output values between 0 and 1. To enable output in a wider range of real number variants are used:
  - neurons with tanh activation function:

    $$a_i = \tanh(net_i) = \frac{e_i^{net} - e^{-net_i}}{e_i^{net} + e^{-net_i}}$$

  - neurons with linear activation:

    $$a_i = net_i$$



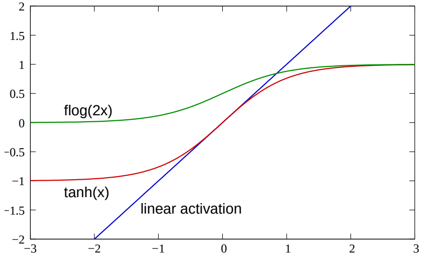
Multi layer perceptrons (cont.)

- variant:
  Neurons with logistic activation can only output values between 0 and 1. To enable output in a wider range of real number variants are used:
  - neurons with tanh activation function:

$$a_i = \tanh(net_i) = \frac{e_i^{net} - e^{-net_i}}{e_i^{net} + e^{-net_i}}$$

  - neurons with linear activation:

$$a_i = net_i$$



- the calculation of the network output is similar to the case of logistic activation except the relationship between net_i and a_i is different.
- the activation function is a local property of each neuron.

Multi layer perceptrons (cont.)

- typical network topologies:
  - for regression: output neurons with linear activation
  - for classification: output neurons with logistic/tanh activation
  - all hidden neurons with logistic activation
  - layered layout:
    input layer – first hidden layer – second hidden layer – ... – output layer
    with connection from each neuron in layer i with each neuron in layer $i+1$,
    no shortcut connections

Multi layer perceptrons (cont.)

- typical network topologies:
  - for regression: output neurons with linear activation
  - for classification: output neurons with logistic/tanh activation
  - all hidden neurons with logistic activation
  - layered layout:
    input layer – first hidden layer – second hidden layer – ... – output layer
    with connection from each neuron in layer i with each neuron in layer $i+1$,
    no shortcut connections
- Lemma:
  Any boolean function can be realized by a MLP with one hidden layer.
  Any bounded continuous function can be approximated with arbitrary precision by a MLP with one hidden layer.
  Proof: was given by Cybenko (1989). Idea: partition input space in small cells

MLP Training

- given training data: $\mathcal{D} = \{(\vec{x}^{(1)}, \vec{d}^{(1)}), \ldots, (\vec{x}^{(p)}, \vec{d}^{(p)})\}$ where $\vec{d}^{(i)}$ is the desired output (real number for regression, class label 0 or 1 for classification)
- given topology of a MLP
- task: adapt weights of the MLP

MLP Training (cont.)

- idea: minimize an error term

$$E(\vec{w}; \mathcal{D}) = \frac{1}{2} \sum_{i=1}^{p} ||y(\vec{x}^{(i)}; \vec{w}) - \vec{d}^{(i)}||^2$$

with $y(\vec{x}; \vec{w})$: network output for input pattern $\vec{x}$ and weight vector $\vec{w}$, $||\vec{u}||^2$ squared length of vector $\vec{u}$: $||\vec{u}||^2 = \sum_{j=1}^{dim(\vec{u})} (u_j)^2$

MLP Training (cont.)

- idea: minimize an error term

$$E(\vec{w};\mathcal{D}) = \frac{1}{2}\sum_{i=1}^{p} ||y(\vec{x}^{(i)};\vec{w}) - \vec{d}^{(i)}||^2$$

with $y(\vec{x};\vec{w})$: network output for input pattern $\vec{x}$ and weight vector $\vec{w}$, $||\vec{u}||^2$ squared length of vector $\vec{u}$: $||\vec{u}||^2 = \sum_{j=1}^{dim(\vec{u})}(u_j)^2$

- learning means: calculating weights for which the error becomes minimal

$$\underset{\vec{w}}{minimize}\ E(\vec{w};\mathcal{D})$$

MLP Training (cont.)

- idea: minimize an error term

$$E(\vec{w}; \mathcal{D}) = \frac{1}{2} \sum_{i=1}^{p} ||y(\vec{x}^{(i)}; \vec{w}) - \vec{d}^{(i)}||^2$$

  with $y(\vec{x}; \vec{w})$: network output for input pattern $\vec{x}$ and weight vector $\vec{w}$, $||\vec{u}||^2$ squared length of vector $\vec{u}$: $||\vec{u}||^2 = \sum_{j=1}^{dim(\vec{u})} (u_j)^2$
- learning means: calculating weights for which the error becomes minimal

$$\underset{\vec{w}}{minimize}\ E(\vec{w}; \mathcal{D})$$

- interpret E just as a mathematical function depending on $\vec{w}$ and forget about its semantics, then we are faced with a problem of mathematical optimization

Optimization theory

- discusses mathematical problems of the form:

$$\underset{\vec{u}}{minimize}\ f(\vec{u})$$

$\vec{u}$ can be any vector of suitable size. But which one solves this task and how can we calculate it?

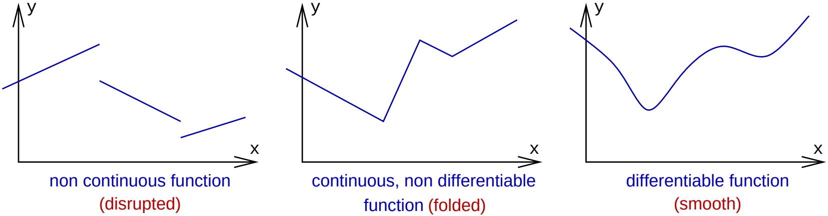
Optimization theory

- discusses mathematical problems of the form:

$$\underset{\vec{u}}{minimize}\ f(\vec{u})$$

$\vec{u}$ can be any vector of suitable size. But which one solves this task and how can we calculate it?

- some simplifications:
  here we consider only functions f which are continuous and differentiable



Optimization theory (cont.)

- A global minimum $\vec{u}^*$ is a point so that:

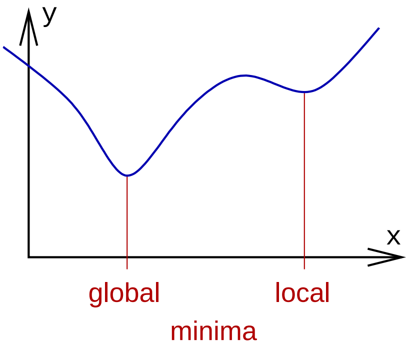
  $$f(\vec{u}^*) \leq f(\vec{u})$$

  for all $\vec{u}$.

- A local minimum $\vec{u}^+$ is a point so that exist $r > 0$ with

  $$f(\vec{u}^+) \leq f(\vec{u})$$

  for all points $\vec{u}$ with $||\vec{u} - \vec{u}^+|| < r$



Optimization theory (cont.)

- analytical way to find a minimum:
  For a local minimum $\vec{u}^+$, the gradient of f becomes zero:

$$\frac{\partial f}{\partial u_i}(\vec{u}^+) = 0 \quad \text{for all } i$$

  Hence, calculating all partial derivatives and looking for zeros is a good idea (c.f. linear regression)

Optimization theory (cont.)

- analytical way to find a minimum:
  For a local minimum $\vec{u}^+$, the gradient of f becomes zero:

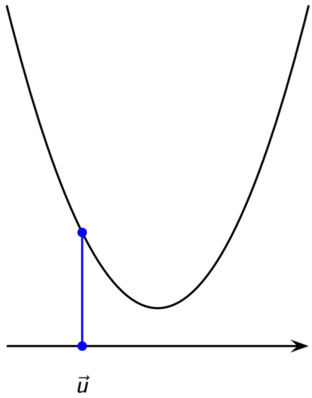
  $$\frac{\partial f}{\partial u_i}(\vec{u}^+) = 0 \quad \text{for all } i$$

  Hence, calculating all partial derivatives and looking for zeros is a good idea (c.f. linear regression)

  but: there are also other points for which $\frac{\partial f}{\partial u_i} = 0$, and resolving these equations is often not possible

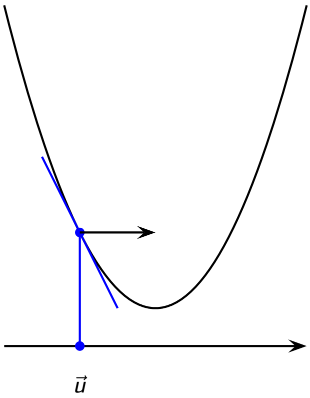
Optimization theory (cont.)

- numerical way to find a minimum, searching:
  assume we are starting at a point $\vec{u}$.
  Which is the best direction to search for a point $\vec{v}$ with $f(\vec{v}) < f(\vec{u})$?



Optimization theory (cont.)

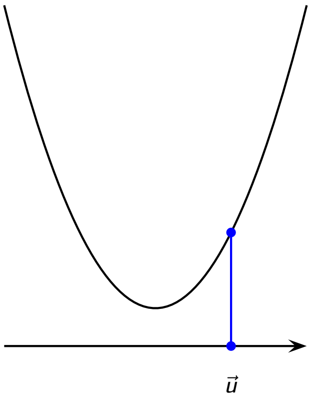
- numerical way to find a minimum, searching:
  assume we are starting at a point $\vec{u}$.
  Which is the best direction to search for a point $\vec{v}$ with $f(\vec{v}) < f(\vec{u})$?



slope is negative (descending), go right!

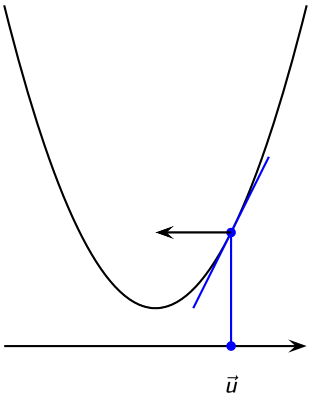
Optimization theory (cont.)

- numerical way to find a minimum, searching:
  assume we are starting at a point $\vec{u}$.
  Which is the best direction to search for a point $\vec{v}$ with $f(\vec{v}) < f(\vec{u})$?



Optimization theory (cont.)

- numerical way to find a minimum, searching:
  assume we are starting at a point $\vec{u}$.
  Which is the best direction to search for a point $\vec{v}$ with $f(\vec{v}) < f(\vec{u})$?

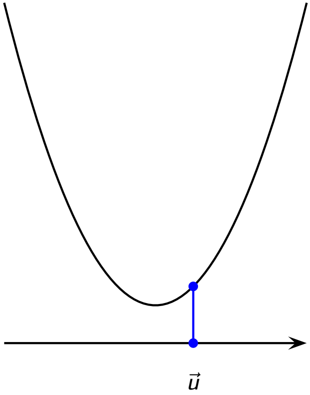


slope is positive (ascending), go left!

Optimization theory (cont.)

- numerical way to find a minimum, searching:
  assume we are starting at a point $\vec{u}$.
  Which is the best direction to search for a point $\vec{v}$ with $f(\vec{v}) < f(\vec{u})$?

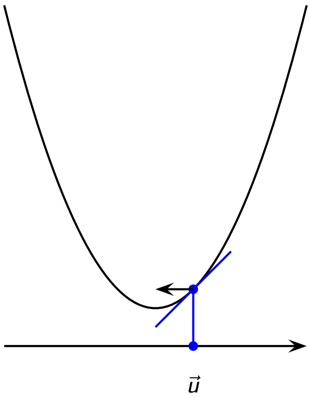
  Which is the best stepwidth?



Optimization theory (cont.)

- numerical way to find a minimum, searching:
  assume we are starting at a point $\vec{u}$.
  Which is the best direction to search for a point $\vec{v}$ with $f(\vec{v}) < f(\vec{u})$?

  Which is the best stepwidth?

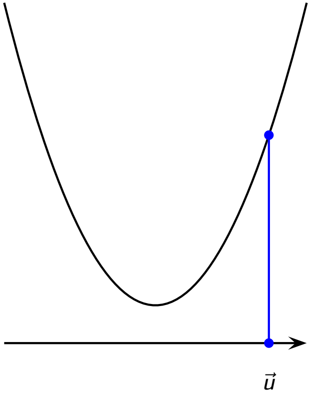


slope is small, small step!

Optimization theory (cont.)

- numerical way to find a minimum, searching:
  assume we are starting at a point $\vec{u}$.
  Which is the best direction to search for a point $\vec{v}$ with $f(\vec{v}) < f(\vec{u})$?

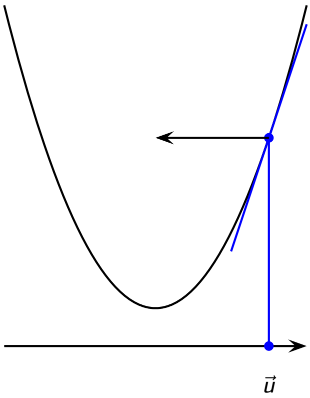
  Which is the best stepwidth?



Optimization theory (cont.)

- numerical way to find a minimum, searching:
  assume we are starting at a point $\vec{u}$.
  Which is the best direction to search for a point $\vec{v}$ with $f(\vec{v}) < f(\vec{u})$?

  Which is the best stepwidth?



slope is large, large step!

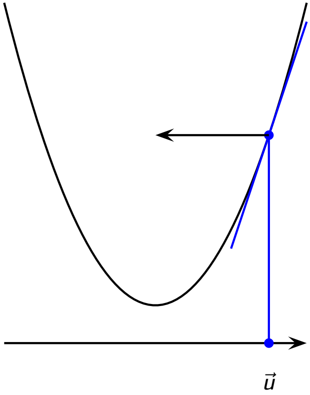
Optimization theory (cont.)

- numerical way to find a minimum, searching:
  assume we are starting at a point $\vec{u}$.
  Which is the best direction to search for a point $\vec{v}$ with $f(\vec{v}) < f(\vec{u})$?

  Which is the best stepwidth?
- general principle:

$$v_i \leftarrow u_i - \epsilon \frac{\partial f}{\partial u_i}$$

  $\epsilon > 0$ is called learning rate



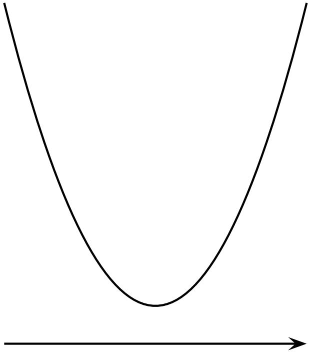
Gradient descent

- Gradient descent approach:

  Require: mathematical function f, learning rate $\epsilon > 0$
  Ensure: returned vector is close to a local minimum of f
  1: choose an initial point $\vec{u}$
  2: **while** $||gradf(\vec{u})||$ not close to 0 **do**
  3: $\quad \vec{u} \leftarrow \vec{u} - \epsilon \cdot gradf(\vec{u})$
  4: **end while**
  5: **return** $\vec{u}$
- open questions:
  - how to choose initial $\vec{u}$
  - how to choose ϵ
  - does this algorithm really converge?

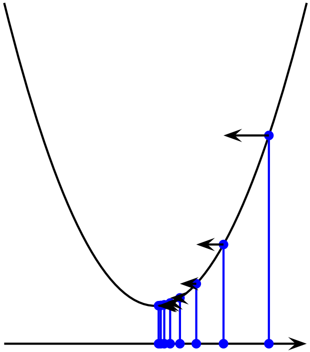
Gradient descent (cont.)

- choice of ϵ



Gradient descent (cont.)

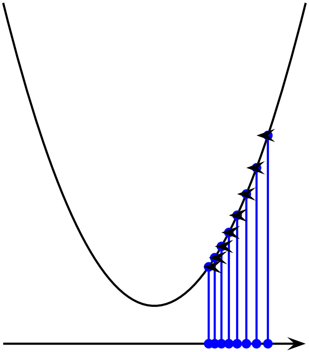
- choice of ϵ
  1. case small ϵ: convergence



Gradient descent (cont.)

- choice of ϵ

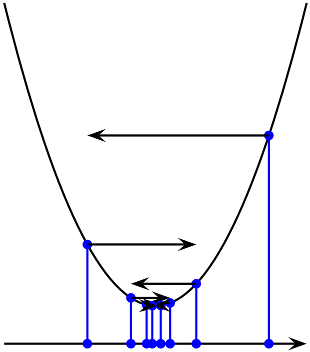
  2. case very small ϵ: convergence, but it may take very long



Gradient descent (cont.)

- choice of ϵ

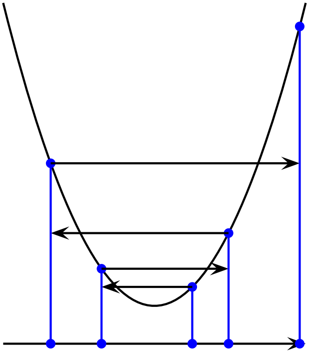
  3. case medium size ϵ: convergence



Gradient descent (cont.)

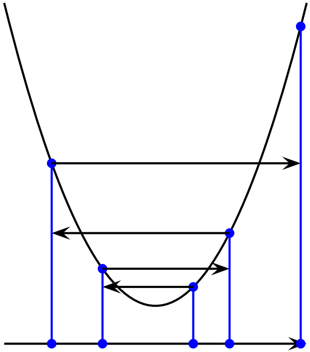
- choice of ϵ

  4. case large ϵ: divergence



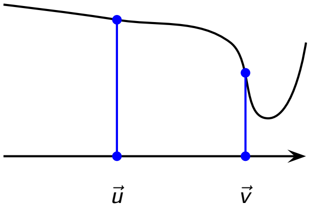
Gradient descent (cont.)

- choice of ϵ
  - is crucial. Only small ϵ guarantee convergence.
  - for small ϵ, learning may take very long
  - depends on the scaling of f: an optimal learning rate for f may lead to divergence for $2 \cdot f$



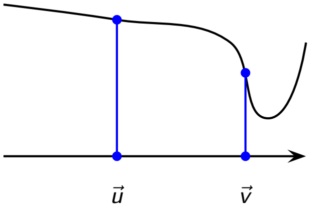
Gradient descent (cont.)

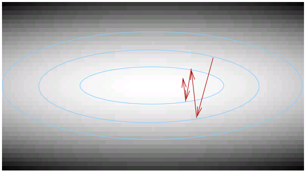
- some more problems with gradient descent:
  - flat spots and steep valleys: need larger ϵ in $\vec{u}$ to jump over the uninteresting flat area but need smaller ϵ in $\vec{v}$ to meet the minimum



Gradient descent (cont.)

- some more problems with gradient descent:
  - flat spots and steep valleys:
    need larger ϵ in $\vec{u}$ to jump over the uninteresting flat area but need smaller ϵ in $\vec{v}$ to meet the minimum
  - zig-zagging:
    in higher dimensions: ϵ is not appropriate for all dimensions





Gradient descent (cont.)

- conclusion:
  pure gradient descent is a nice theoretical framework but of limited power in practice. Finding the right ϵ is annoying. Approaching the minimum is time consuming.

Gradient descent (cont.)

- conclusion:
  pure gradient descent is a nice theoretical framework but of limited power in practice. Finding the right ϵ is annoying. Approaching the minimum is time consuming.
- heuristics to overcome problems of gradient descent:
  - gradient descent with momentum
  - individual lerning rates for each dimension
  - adaptive learning rates
  - decoupling steplength from partial derivates

Gradient descent (cont.)

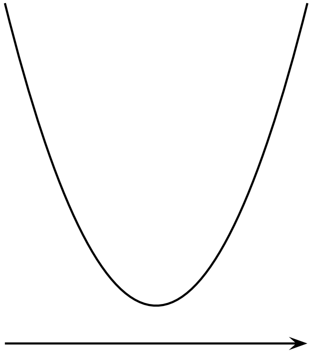
- gradient descent with momentum
  idea: make updates smoother by carrying forward the latest update.

  choose an initial point $\vec{u}$
  set $\vec{\Delta} \leftarrow \vec{0}$ (stepwidth)
  while $||grad f(\vec{u})||$ not close to 0 **do**
  $\quad \vec{\Delta} \leftarrow -\epsilon \cdot grad f(\vec{u}) + \mu\vec{\Delta}$
  $\quad \vec{u} \leftarrow \vec{u} + \vec{\Delta}$
  end while
  return $\vec{u}$

  $\mu \geq 0, \mu < 1$ is an additional parameter that has to be adjusted by hand.
  For $\mu = 0$ we get vanilla gradient descent.

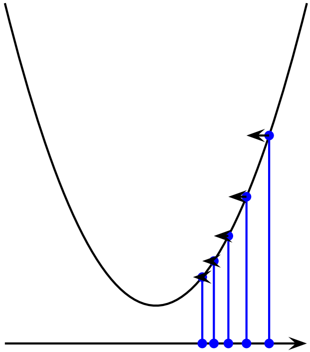
Gradient descent (cont.)

- advantages of momentum:
  - smoothes zig-zagging
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantage:
  - additional parameter μ
  - may cause additional zig-zagging



Gradient descent (cont.)

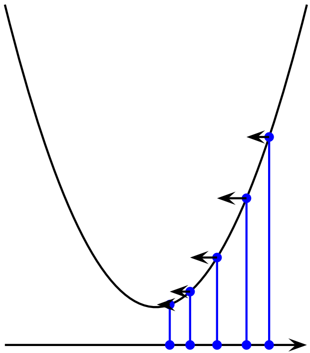
- advantages of momentum:
  - smoothes zig-zagging
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantage:
  - additional parameter μ
  - may cause additional zig-zagging



vanilla gradient descent

Gradient descent (cont.)

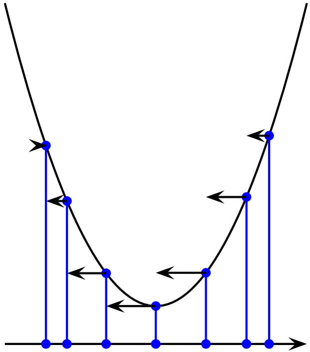
- advantages of momentum:
  - smoothes zig-zagging
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantage:
  - additional parameter μ
  - may cause additional zig-zagging



gradient descent with momentum

Gradient descent (cont.)

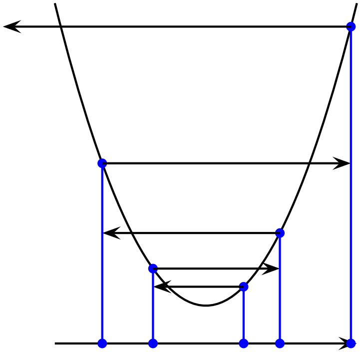
- advantages of momentum:
  - smoothes zig-zagging
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantage:
  - additional parameter μ
  - may cause additional zig-zagging



gradient descent with strong momentum

Gradient descent (cont.)

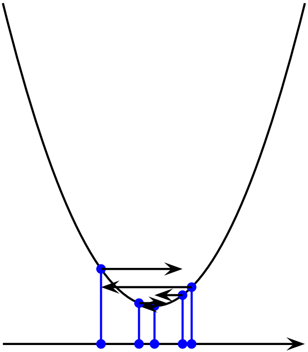
- advantages of momentum:
  - smoothes zig-zagging
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantage:
  - additional parameter μ
  - may cause additional zig-zagging



vanilla gradient descent

Gradient descent (cont.)

- advantages of momentum:
  - smoothes zig-zagging
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantage:
  - additional parameter μ
  - may cause additional zig-zagging



gradient descent with momentum

Gradient descent (cont.)

- adaptive learning rate
  idea:
  - make learning rate individual for each dimension and adaptive
  - if signs of partial derivative change, reduce learning rate
  - if signs of partial derivative don't change, increase learning rate
- algorithm: Super-SAB (Tollenare 1990)

Gradient descent (cont.)

choose an initial point $\vec{u}$
set initial learning rate $\vec{\epsilon}$
set former gradient $\vec{\gamma} \leftarrow \vec{0}$
while $||gradf(\vec{u})||$ not close to 0 **do**
  calculate gradient $\vec{g} \leftarrow gradf(\vec{u})$
  for all dimensions i **do**

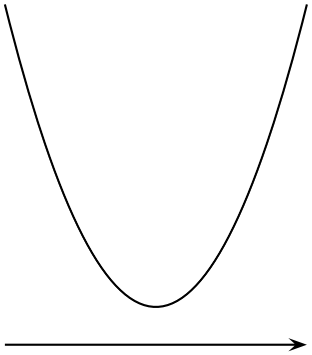
$$\epsilon_i \leftarrow \begin{cases} \eta^+ \epsilon_i & \text{if } g_i \cdot \gamma_i > 0 \\ \eta^- \epsilon_i & \text{if } g_i \cdot \gamma_i < 0 \\ \epsilon_i & \text{otherwise} \end{cases}$$

    $u_i \leftarrow u_i - \epsilon_i g_i$
  end for
  $\vec{\gamma} \leftarrow \vec{g}$
end while
return $\vec{u}$

$\eta^+ \geq 1, \eta^- \leq 1$ are additional parameters that have to be adjusted by hand. For $\eta^+ = \eta^- = 1$ we get vanilla gradient descent.

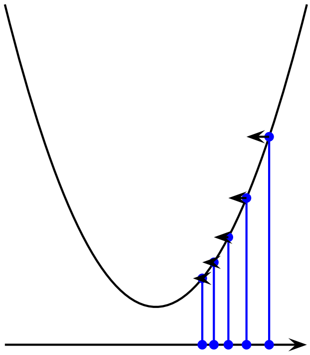
Gradient descent (cont.)

- advantages of Super-SAB and related approaches:
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantages:
  - steplength still depends on partial derivatives



Gradient descent (cont.)

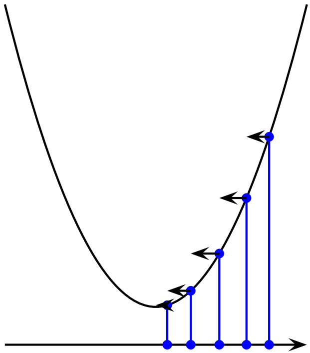
- advantages of Super-SAB and related approaches:
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantages:
  - steplength still depends on partial derivatives



vanilla gradient descent

Gradient descent (cont.)

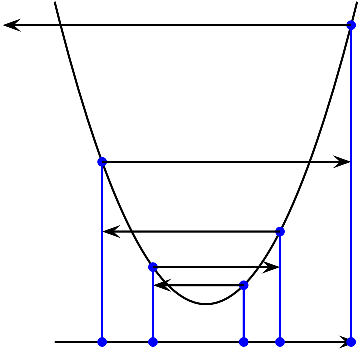
- advantages of Super-SAB and related approaches:
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantages:
  - steplength still depends on partial derivatives



SuperSAB

Gradient descent (cont.)

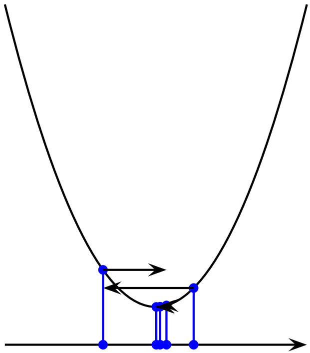
- advantages of Super-SAB and related approaches:
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantages:
  - steplength still depends on partial derivatives



vanilla gradient descent

Gradient descent (cont.)

- advantages of Super-SAB and related approaches:
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
- disadavantages:
  - steplength still depends on partial derivatives



SuperSAB

Gradient descent (cont.)

- make steplength independent of partial derivatives
  idea:
  - use explicit steplength parameters, one for each dimension
  - if signs of partial derivative change, reduce steplength
  - if signs of partial derivative don't change, increase steplegth
- algorithm: RProp (Riedmiller&Braun, 1993)

Gradient descent (cont.)

choose an initial point $\vec{u}$
set initial steplength $\vec{\Delta}$
set former gradient $\vec{\gamma} \leftarrow \vec{0}$
while $||gradf(\vec{u})||$ not close to 0 **do**
    calculate gradient $\vec{g} \leftarrow gradf(\vec{u})$
    for all dimensions i **do**

$$\Delta_i \leftarrow \begin{cases} \eta^+ \Delta_i & \text{if } g_i \cdot \gamma_i > 0 \\ \eta^- \Delta_i & \text{if } g_i \cdot \gamma_i < 0 \\ \Delta_i & \text{otherwise} \end{cases}$$

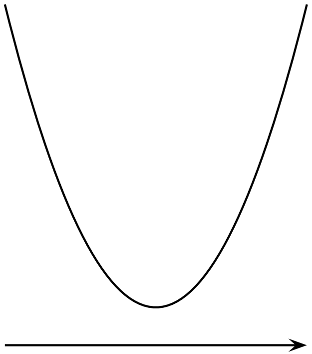
$$u_i \leftarrow \begin{cases} u_i + \Delta_i & \text{if } g_i < 0 \\ u_i - \Delta_i & \text{if } g_i > 0 \\ u_i & \text{otherwise} \end{cases}$$

    end for
    $\vec{\gamma} \leftarrow \vec{g}$
end while
return $\vec{u}$

$\eta^+ \geq 1, \eta^- \leq 1$ are additional parameters that have to be adjusted by hand. For MLPs, good heuristics exist for parameter settings: $\eta^+ = 1.2$, $\eta^- = 0.5$, initial $\Delta_i = 0.1$

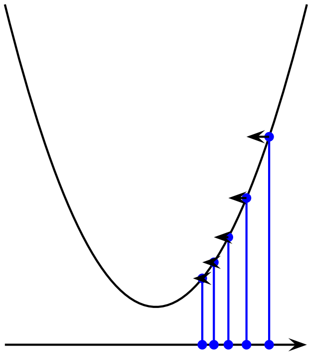
Gradient descent (cont.)

- advantages of Rprop
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
  - independent of gradient length



Gradient descent (cont.)

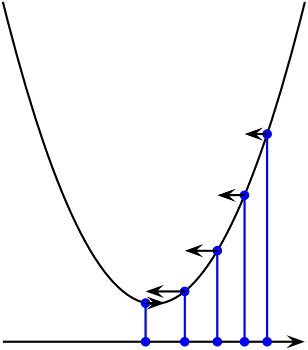
- advantages of Rprop
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
  - independent of gradient length



vanilla gradient descent

Gradient descent (cont.)

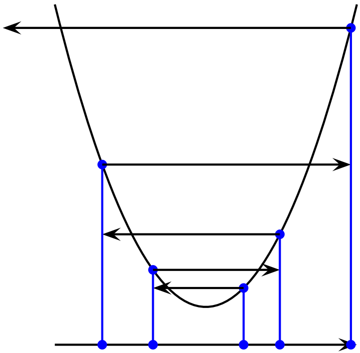
- advantages of Rprop
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
  - independent of gradient length



Rprop

Gradient descent (cont.)

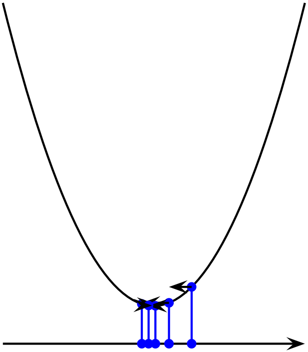
- advantages of Rprop
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
  - independent of gradient length



vanilla gradient descent

Gradient descent (cont.)

- advantages of Rprop
  - decouples learning rates of different dimensions
  - accelerates learning at flat spots
  - slows down when signs of partial derivatives change
  - independent of gradient length



Rprop

Beyond gradient descent

- Newton
- Quickprop
- line search

Beyond gradient descent (cont.)

- Newton's method:

  approximate f by a second-order Taylor polynomial:

  $$f(\vec{u}+\vec{\Delta}) \approx f(\vec{u}) + grad f(\vec{u}) \cdot \vec{\Delta} + \frac{1}{2}\vec{\Delta}^T H(\vec{u})\vec{\Delta}$$

  with $H(\vec{u})$ the Hessian of f at $\vec{u}$, the matrix of second order partial derivatives.

Beyond gradient descent (cont.)

- Newton's method:

  approximate f by a second-order Taylor polynomial:

  $$f(\vec{u}+\vec{\Delta}) \approx f(\vec{u}) + grad\, f(\vec{u}) \cdot \vec{\Delta} + \frac{1}{2}\vec{\Delta}^T H(\vec{u})\vec{\Delta}$$

  with $H(\vec{u})$ the Hessian of f at $\vec{u}$, the matrix of second order partial derivatives.

  Zeroing the gradient of approximation with respect to $\vec{\Delta}$:

  $$\vec{0} \approx (grad\, f(\vec{u}))^T + H(\vec{u})\vec{\Delta}$$

  Hence:

  $$\vec{\Delta} \approx -(H(\vec{u}))^{-1}(grad\, f(\vec{u}))^T$$

Beyond gradient descent (cont.)

- Newton's method:

  approximate f by a second-order Taylor polynomial:

  $$f(\vec{u}+\vec{\Delta}) \approx f(\vec{u}) + \mathit{grad} f(\vec{u}) \cdot \vec{\Delta} + \frac{1}{2}\vec{\Delta}^T H(\vec{u})\vec{\Delta}$$

  with $H(\vec{u})$ the Hessian of f at $\vec{u}$, the matrix of second order partial derivatives.

  Zeroing the gradient of approximation with respect to $\vec{\Delta}$:

  $$\vec{0} \approx (\mathit{grad} f(\vec{u}))^T + H(\vec{u})\vec{\Delta}$$

  Hence:

  $$\vec{\Delta} \approx -(H(\vec{u}))^{-1}(\mathit{grad} f(\vec{u}))^T$$

- advantages: no learning rate, no parameters, quick convergence
- disadvantages: calculation of H and H^{-1} very time consuming in high dimensional spaces

Beyond gradient descent (cont.)

- Quickprop (Fahlmann, 1988)
  - like Newton's method, but replaces H by a diagonal matrix containing only the diagonal entries of H.
  - hence, calculating the inverse is simplified
  - replaces second order derivatives by approximations (difference ratios)

Beyond gradient descent (cont.)

- Quickprop (Fahlmann, 1988)
  - like Newton's method, but replaces H by a diagonal matrix containing only the diagonal entries of H.
  - hence, calculating the inverse is simplified
  - replaces second order derivatives by approximations (difference ratios)
- update rule:

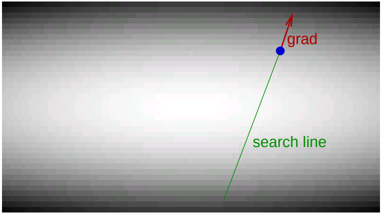
$$\triangle w_i^t := \frac{-g_i^t}{g_i^t - g_i^{t-1}} (w_i^t - w_i^{t-1})$$

  where $g_i^t = grad\, f$ at time t.
- advantages: no learning rate, no parameters, quick convergence in many cases
- disadvantages: sometimes unstable

Beyond gradient descent (cont.)

- line search algorithms:

  two nested loops:
  - outer loop: determine serach direction from gradient
  - inner loop: determine minimizing point on the line defined by current search position and search direction
- inner loop can be realized by any minimization algorithm for one-dimensional tasks
- advantage: inner loop algorithm may be more complex algorithm, e.g. Newton

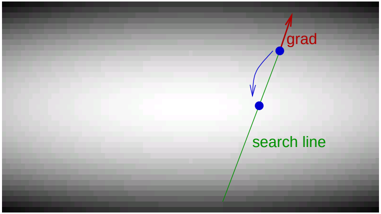


- problem: time consuming for high-dimensional spaces

Beyond gradient descent (cont.)

- line search algorithms:

  two nested loops:
  - outer loop: determine serach direction from gradient
  - inner loop: determine minimizing point on the line defined by current search position and search direction
- inner loop can be realized by any minimization algorithm for one-dimensional tasks
- advantage: inner loop algorithm may be more complex algorithm, e.g. Newton

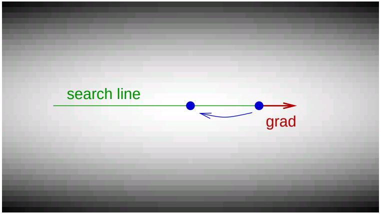


- problem: time consuming for high-dimensional spaces

Beyond gradient descent (cont.)

- line search algorithms:

  two nested loops:
  - outer loop: determine serach direction from gradient
  - inner loop: determine minimizing point on the line defined by current search position and search direction
- inner loop can be realized by any minimization algorithm for one-dimensional tasks
- advantage: inner loop algorithm may be more complex algorithm, e.g. Newton



- problem: time consuming for high-dimensional spaces

Summary: optimization theory

- several algorithms to solve problems of the form:

$$\underset{\vec{u}}{minimize}\ f(\vec{u})$$

- gradient descent gives the main idea
- Rprop plays major role in context of MLPs
- dozens of variants and alternatives exist

Back to MLP Training

- training an MLP means solving:

$$\underset{\vec{w}}{minimize}\ E(\vec{w};\mathcal{D})$$

for given network topology and training data $\mathcal{D}$

$$E(\vec{w};\mathcal{D}) = \frac{1}{2}\sum_{i=1}^{p} ||y(\vec{x}^{(i)};\vec{w}) - \vec{d}^{(i)}||^2$$

Back to MLP Training

- training an MLP means solving:

$$\underset{\vec{w}}{minimize}\ E(\vec{w}; \mathcal{D})$$

for given network topology and training data $\mathcal{D}$

$$E(\vec{w}; \mathcal{D}) = \frac{1}{2} \sum_{i=1}^{p} ||y(\vec{x}^{(i)}; \vec{w}) - \vec{d}^{(i)}||^2$$

- optimization theory offers algorithms to solve task of this kind

open question: how can we calculate derivatives of E?

Calculating partial derivatives

- the calculation of the network output of a MLP is done step-by-step: neuron i uses the output of neurons $j \in Pred(i)$ as arguments, calculates some output which serves as argument for all neurons $j \in Succ(i)$.
- apply the chain rule!

Calculating partial derivatives (cont.)

- the error term

$$E(\vec{w};\mathcal{D}) = \sum_{i=1}^{p} \left(\frac{1}{2}||y(\vec{x}^{(i)};\vec{w}) - \vec{d}^{(i)}||^2\right)$$

introducing $e(\vec{w};\vec{x},\vec{d}) = \frac{1}{2}||y(\vec{x};\vec{w}) - \vec{d}||^2$ we can write:

Calculating partial derivatives (cont.)

- the error term

$$E(\vec{w};\mathcal{D}) = \sum_{i=1}^{p} \left(\frac{1}{2}||y(\vec{x}^{(i)};\vec{w}) - \vec{d}^{(i)}||^2\right)$$

introducing $e(\vec{w};\vec{x},\vec{d}) = \frac{1}{2}||y(\vec{x};\vec{w}) - \vec{d}||^2$ we can write:

$$E(\vec{w};\mathcal{D}) = \sum_{i=1}^{p} e(\vec{w};\vec{x}^{(i)},\vec{d}^{(i)})$$

applying the rule for sums:

Calculating partial derivatives (cont.)

- the error term

$$E(\vec{w};\mathcal{D}) = \sum_{i=1}^{p} \left(\frac{1}{2}||y(\vec{x}^{(i)};\vec{w}) - \vec{d}^{(i)}||^2\right)$$

introducing $e(\vec{w};\vec{x},\vec{d}) = \frac{1}{2}||y(\vec{x};\vec{w}) - \vec{d}||^2$ we can write:

$$E(\vec{w};\mathcal{D}) = \sum_{i=1}^{p} e(\vec{w};\vec{x}^{(i)},\vec{d}^{(i)})$$

applying the rule for sums:

$$\frac{\partial E(\vec{w};\mathcal{D})}{\partial w_{kl}} = \sum_{i=1}^{p} \frac{\partial e(\vec{w};\vec{x}^{(i)},\vec{d}^{(i)})}{\partial w_{kl}}$$

we can calculate the derivatives for each training pattern indiviudally and sum up

Calculating partial derivatives (cont.)

- individual error terms for a pattern $\vec{x}, \vec{d}$
  simplifications in notation:
  - omitting dependencies from $\vec{x}$ and $\vec{d}$
  - $y(\vec{w}) = (y_1, \ldots, y_m)^T$ network output (when applying input pattern $\vec{x}$)

Calculating partial derivatives (cont.)

- individual error term:

$$e(\vec{w}) = \frac{1}{2}||y(\vec{x};\vec{w}) - \vec{d}||^2 = \frac{1}{2}\sum_{j=1}^{m}(y_j - d_j)^2$$

by direct calculation:

$$\frac{\partial e}{\partial y_j} = (y_j - d_j)$$

y_j is the activation of a certain output neuron, say a_i

Hence:

$$\frac{\partial e}{\partial a_i} = \frac{\partial e}{\partial y_j} = (a_i - d_j)$$

Calculating partial derivatives (cont.)

- calculations within a neuron i

  assume we already know $\frac{\partial e}{\partial a_i}$

  observation: e depends indirectly from a_i and a_i depends on net_i
  $\Rightarrow$ apply chain rule

$$\frac{\partial e}{\partial net_i} = \frac{\partial e}{\partial a_i} \cdot \frac{\partial a_i}{\partial net_i}$$

  what is $\frac{\partial a_i}{\partial net_i}$?

Calculating partial derivatives (cont.)

- $\frac{\partial a_i}{\partial net_i}$

  a_i is calculated like: $a_i = f_{act}(net_i)$ (f_{act} activation function)
  Hence:

$$\frac{\partial a_i}{\partial net_i} = \frac{\partial f_{act}(net_i)}{\partial net_i}$$

Calculating partial derivatives (cont.)

- $\frac{\partial a_i}{\partial net_i}$

  a_i is calculated like: $a_i = f_{act}(net_i)$ (f_{act} activation function)
  Hence:

$$\frac{\partial a_i}{\partial net_i} = \frac{\partial f_{act}(net_i)}{\partial net_i}$$

  - linear activation: $f_{act}(net_i) = net_i$
    $\Rightarrow \quad \frac{\partial f_{act}(net_i)}{\partial net_i} = 1$

Calculating partial derivatives (cont.)

- $\frac{\partial a_i}{\partial net_i}$

  a_i is calculated like: $a_i = f_{act}(net_i)$ (f_{act} activation function)
  Hence:

  $$\frac{\partial a_i}{\partial net_i} = \frac{\partial f_{act}(net_i)}{\partial net_i}$$

  - linear activation: $f_{act}(net_i) = net_i$
    $\Rightarrow \quad \frac{\partial f_{act}(net_i)}{\partial net_i} = 1$
  - logistic activation: $f_{act}(net_i) = \frac{1}{1+e^{-net_i}}$
    $\Rightarrow \quad \frac{\partial f_{act}(net_i)}{\partial net_i} = \frac{e^{-net_i}}{(1+e^{-net_i})^2} = f_{log}(net_i) \cdot (1 - f_{log}(net_i))$

Calculating partial derivatives (cont.)

- $\frac{\partial a_i}{\partial net_i}$

  a_i is calculated like: $a_i = f_{act}(net_i)$ (f_{act} activation function)
  Hence:

$$\frac{\partial a_i}{\partial net_i} = \frac{\partial f_{act}(net_i)}{\partial net_i}$$

  - linear activation: $f_{act}(net_i) = net_i$
    $\Rightarrow \quad \frac{\partial f_{act}(net_i)}{\partial net_i} = 1$
  - logistic activation: $f_{act}(net_i) = \frac{1}{1+e^{-net_i}}$
    $\Rightarrow \quad \frac{\partial f_{act}(net_i)}{\partial net_i} = \frac{e^{-net_i}}{(1+e^{-net_i})^2} = f_{log}(net_i) \cdot (1 - f_{log}(net_i))$
  - tanh activation: $f_{act}(net_i) = \tanh(net_i)$
    $\Rightarrow \quad \frac{\partial f_{act}(net_i)}{\partial net_i} = 1 - (\tanh(net_i))^2$

Calculating partial derivatives (cont.)

- from neuron to neuron

  assume we already know $\frac{\partial e}{\partial net_j}$ for all $j \in Succ(i)$

  observation: e depends indirectly from net_j of successor neurons and net_j depends on a_i $\Rightarrow$ apply chain rule

Calculating partial derivatives (cont.)

- from neuron to neuron

  assume we already know $\frac{\partial e}{\partial net_j}$ for all $j \in Succ(i)$

  observation: e depends indirectly from net_j of successor neurons and net_j depends on a_i $\Rightarrow$ apply chain rule

$$\frac{\partial e}{\partial a_i} = \sum_{j \in Succ(i)} \left(\frac{\partial e}{\partial net_j} \cdot \frac{\partial net_j}{\partial a_i}\right)$$

Calculating partial derivatives (cont.)

- from neuron to neuron

  assume we already know $\frac{\partial e}{\partial net_j}$ for all $j \in Succ(i)$

  observation: e depends indirectly from net_j of successor neurons and net_j depends on a_i $\Rightarrow$ apply chain rule

$$\frac{\partial e}{\partial a_i} = \sum_{j \in Succ(i)} \left(\frac{\partial e}{\partial net_j} \cdot \frac{\partial net_j}{\partial a_i}\right)$$

  and:

$$net_j = w_{ji} a_i + \ldots$$

  hence:

$$\frac{\partial net_j}{\partial a_i} = w_{ji}$$

Calculating partial derivatives (cont.)

- the weights

  assume we already know $\frac{\partial e}{\partial net_i}$ for neuron i and neuron j is predecessor of i

  observation: e depends indirectly from net_i and net_i depends on w_{ij}
  $\Rightarrow$ apply chain rule

Calculating partial derivatives (cont.)

- the weights

  assume we already know $\frac{\partial e}{\partial net_i}$ for neuron i and neuron j is predecessor of i

  observation: e depends indirectly from net_i and net_i depends on w_{ij}
  $\Rightarrow$ apply chain rule

$$\frac{\partial e}{\partial w_{ij}} = \frac{\partial e}{\partial net_i} \cdot \frac{\partial net_i}{\partial w_{ij}}$$

Calculating partial derivatives (cont.)

- the weights

  assume we already know $\frac{\partial e}{\partial net_i}$ for neuron i and neuron j is predecessor of i

  observation: e depends indirectly from net_i and net_i depends on w_{ij}
  $\Rightarrow$ apply chain rule

$$\frac{\partial e}{\partial w_{ij}} = \frac{\partial e}{\partial net_i} \cdot \frac{\partial net_i}{\partial w_{ij}}$$

  and:

$$net_i = w_{ij} a_j + \ldots$$

  hence:

$$\frac{\partial net_i}{\partial w_{ij}} = a_j$$

Calculating partial derivatives (cont.)

- bias weights

  assume we already know $\frac{\partial e}{\partial net_i}$ for neuron i

  observation: e depends indirectly from net_i and net_i depends on w_{i0}
  $\Rightarrow$ apply chain rule

Calculating partial derivatives (cont.)

- bias weights

  assume we already know $\frac{\partial e}{\partial net_i}$ for neuron i

  observation: e depends indirectly from net_i and net_i depends on w_{i0}
  $\Rightarrow$ apply chain rule

$$\frac{\partial e}{\partial w_{i0}} = \frac{\partial e}{\partial net_i} \cdot \frac{\partial net_i}{\partial w_{i0}}$$

Calculating partial derivatives (cont.)

- bias weights

  assume we already know $\frac{\partial e}{\partial net_i}$ for neuron i

  observation: e depends indirectly from net_i and net_i depends on w_{i0}
  $\Rightarrow$ apply chain rule

$$\frac{\partial e}{\partial w_{i0}} = \frac{\partial e}{\partial net_i} \cdot \frac{\partial net_i}{\partial w_{i0}}$$

  and:

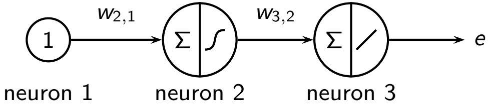
$$net_i = w_{i0} + ...$$

  hence:

$$\frac{\partial net_i}{\partial w_{i0}} = 1$$

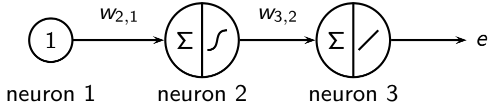
Calculating partial derivatives (cont.)

- a simple example:



Calculating partial derivatives (cont.)

- a simple example:



$$\frac{\partial e}{\partial a_3} = a_3 - d_1$$

$$\frac{\partial e}{\partial net_3} = \frac{\partial e}{\partial a_3} \cdot \frac{\partial a_3}{\partial net_3} = \frac{\partial e}{\partial a_3} \cdot 1$$

$$\frac{\partial e}{\partial a_2} = \sum_{j \in Succ(2)} \left(\frac{\partial e}{\partial net_j} \cdot \frac{\partial net_j}{\partial a_2} \right) = \frac{\partial e}{\partial net_3} \cdot w_{3,2}$$

$$\frac{\partial e}{\partial net_2} = \frac{\partial e}{\partial a_2} \cdot \frac{\partial a_2}{\partial net_2} = \frac{\partial e}{\partial a_2} \cdot a_2(1 - a_2)$$

$$\frac{\partial e}{\partial w_{3,2}} = \frac{\partial e}{\partial net_3} \cdot \frac{\partial net_3}{\partial w_{3,2}} = \frac{\partial e}{\partial net_3} \cdot a_2$$

$$\frac{\partial e}{\partial w_{2,1}} = \frac{\partial e}{\partial net_2} \cdot \frac{\partial net_2}{\partial w_{2,1}} = \frac{\partial e}{\partial net_2} \cdot a_1$$

$$\frac{\partial e}{\partial w_{3,0}} = \frac{\partial e}{\partial net_3} \cdot \frac{\partial net_3}{\partial w_{3,0}} = \frac{\partial e}{\partial net_3} \cdot 1$$

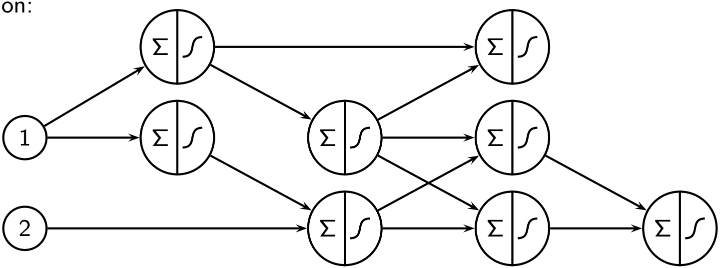
$$\frac{\partial e}{\partial w_{2,0}} = \frac{\partial e}{\partial net_2} \cdot \frac{\partial net_2}{\partial w_{2,0}} = \frac{\partial e}{\partial net_2} \cdot 1$$

Calculating partial derivatives (cont.)

- calculating the partial derivatives:
  - starting at the output neurons
  - neuron by neuron, go from output to input
  - finally calculate the partial derivatives with respect to the weights
- Backpropagation

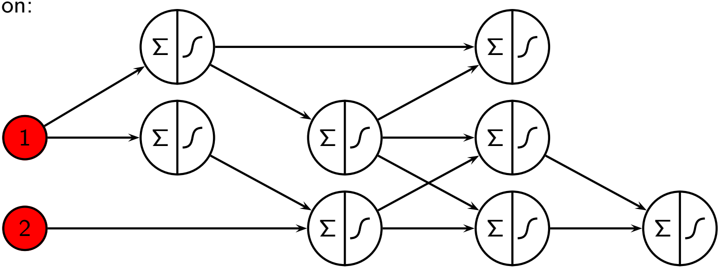
Calculating partial derivatives (cont.)

- illustration:



Calculating partial derivatives (cont.)

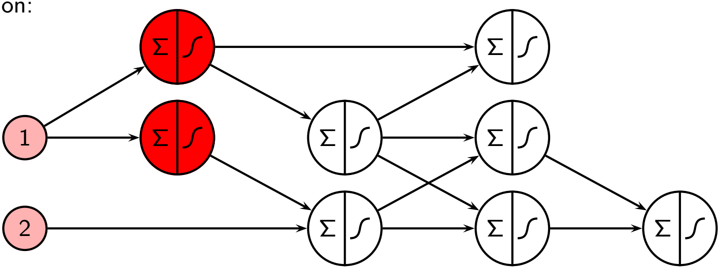
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$

Calculating partial derivatives (cont.)

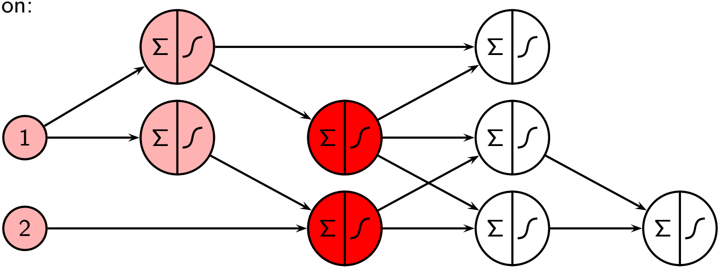
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations:

Calculating partial derivatives (cont.)

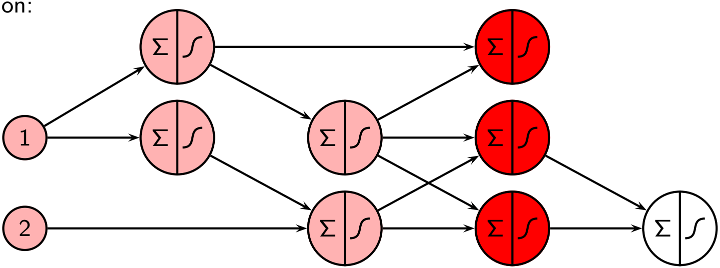
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations: step

Calculating partial derivatives (cont.)

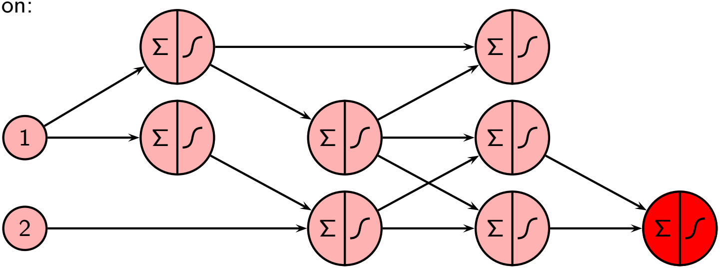
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations: step by

Calculating partial derivatives (cont.)

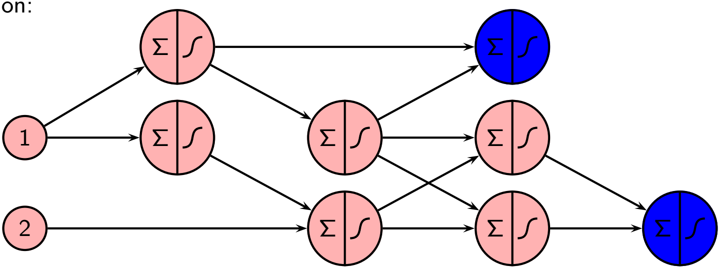
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations: step by step

Calculating partial derivatives (cont.)

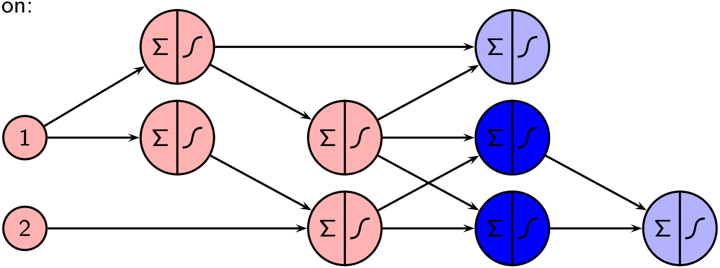
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations: step by step
  - calculate error, $\frac{\partial e}{\partial a_i}$, and $\frac{\partial e}{\partial net_i}$ for output neurons

Calculating partial derivatives (cont.)

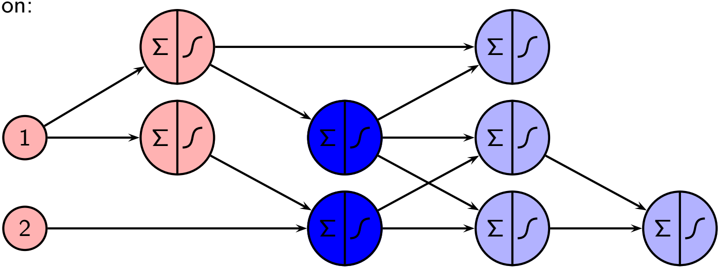
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations: step by step
  - calculate error, $\frac{\partial e}{\partial a_i}$, and $\frac{\partial e}{\partial net_i}$ for output neurons
  - propagate backward error: step

Calculating partial derivatives (cont.)

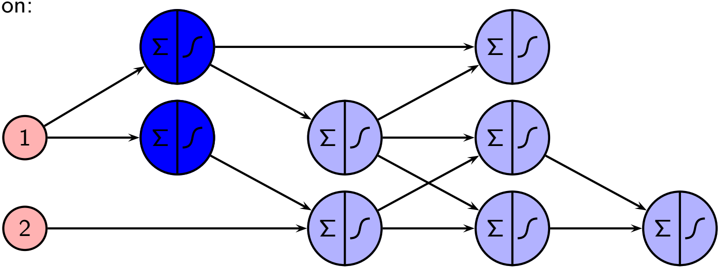
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations: step by step
  - calculate error, $\frac{\partial e}{\partial a_i}$, and $\frac{\partial e}{\partial net_i}$ for output neurons
  - propagate backward error: step by

Calculating partial derivatives (cont.)

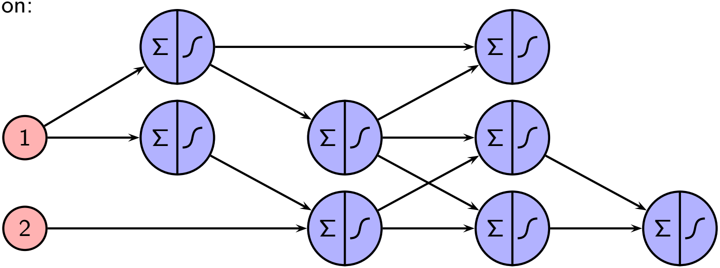
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations: step by step
  - calculate error, $\frac{\partial e}{\partial a_i}$, and $\frac{\partial e}{\partial net_i}$ for output neurons
  - propagate backward error: step by step

Calculating partial derivatives (cont.)

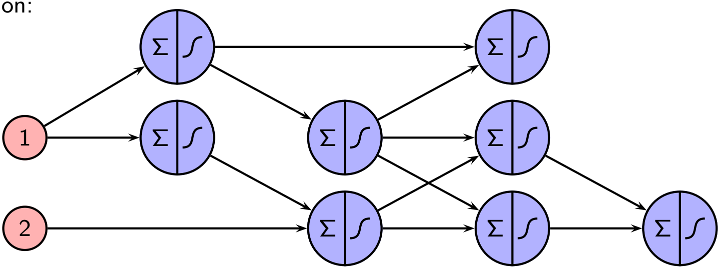
- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations: step by step
  - calculate error, $\frac{\partial e}{\partial a_i}$, and $\frac{\partial e}{\partial net_i}$ for output neurons
  - propagate backward error: step by step
  - calculate $\frac{\partial e}{\partial w_{ji}}$

Calculating partial derivatives (cont.)

- illustration:



  - apply pattern $\vec{x} = (x_1, x_2)^T$
  - propagate forward the activations: step by step
  - calculate error, $\frac{\partial e}{\partial a_i}$, and $\frac{\partial e}{\partial net_i}$ for output neurons
  - propagate backward error: step by step
  - calculate $\frac{\partial e}{\partial w_{ji}}$
  - repeat for all patterns and sum up

Back to MLP Training

- bringing together building blocks of MLP learning:
  - we can calculate $\frac{\partial E}{\partial w_{ij}}$
  - we have discussed methods to minimize a differentiable mathematical function

Back to MLP Training

- bringing together building blocks of MLP learning:
  - we can calculate $\frac{\partial E}{\partial w_{ij}}$
  - we have discussed methods to minimize a differentiable mathematical function
- combining them yields a learning algorithm for MLPs:
  - (standard) backpropagation = gradient descent combined with calculating $\frac{\partial E}{\partial w_{ij}}$ for MLPs
  - backpropagation with momentum = gradient descent with moment combined with calculating $\frac{\partial E}{\partial w_{ij}}$ for MLPs
  - Quickprop
  - Rprop
  - ...

Back to MLP Training (cont.)

- generic MLP learning algorithm:

  choose an initial weight vector $\vec{w}$
  intialize minimization approach
  while error did not converge **do**
      for all $(\vec{x}, \vec{d}) \in \mathcal{D}$ **do**
          apply $\vec{x}$ to network and calculate the network output
          calculate $\frac{\partial e(\vec{x})}{\partial w_{ij}}$ for all weights
      end for
      calculate $\frac{\partial E(\mathcal{D})}{\partial w_{ij}}$ for all weights suming over all training patterns
      perform one update step of the minimization approach
  end while

- learning by epoch: all training patterns are considered for one update step of function minimization

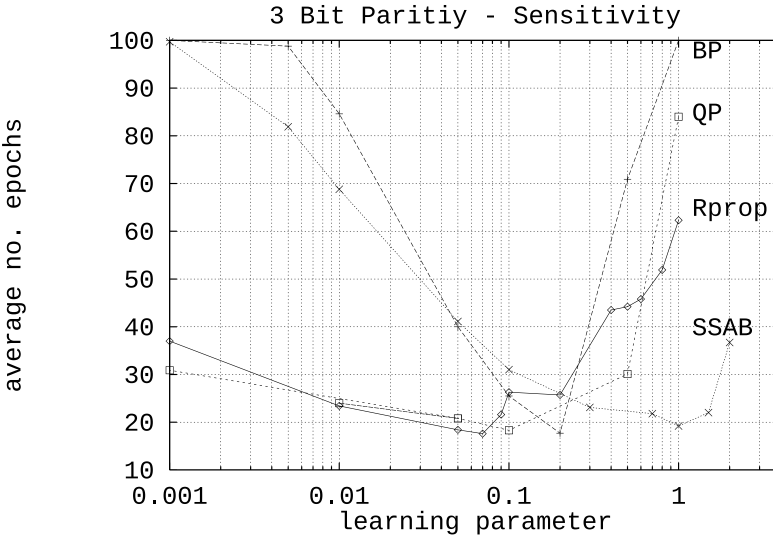
Back to MLP Training (cont.)

- generic MLP learning algorithm:

  choose an initial weight vector $\vec{w}$
  intialize minimization approach
  while error did not converge **do**
      for all $(\vec{x}, \vec{d}) \in \mathcal{D}$ **do**
          apply $\vec{x}$ to network and calculate the network output
          calculate $\frac{\partial e(\vec{x})}{\partial w_{ij}}$ for all weights
          perform one update step of the minimization approach
      end for
  end while

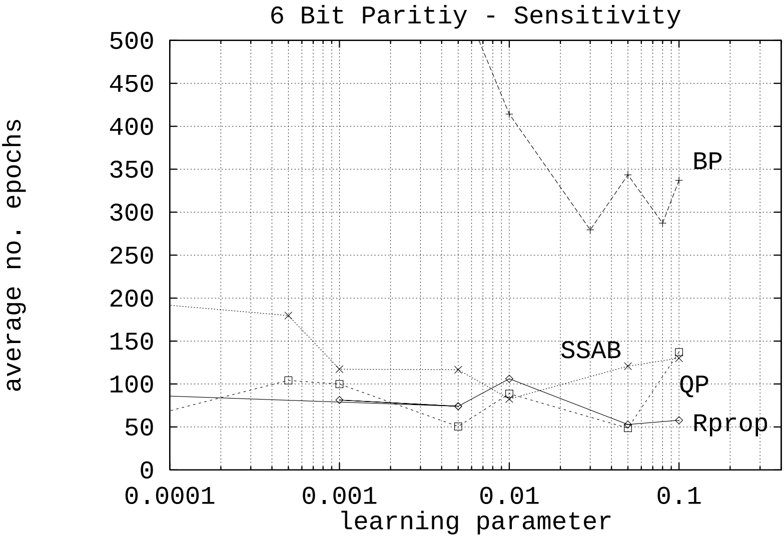
- learning by pattern: only one training patterns is considered for one update step of function minimization (only works with vanilla gradient descent!)

Lernverhalten und Parameterwahl - 3 Bit Parity



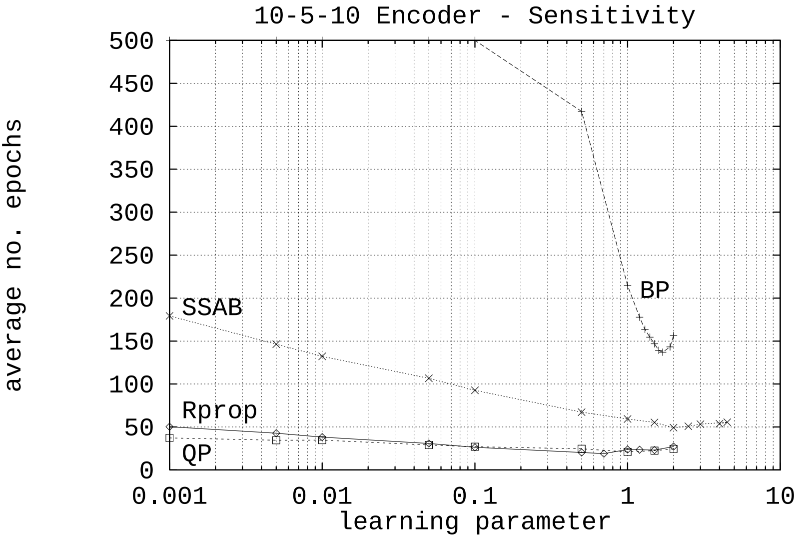
Quelle:

Lernverhalten und Parameterwahl - 6 Bit Parity



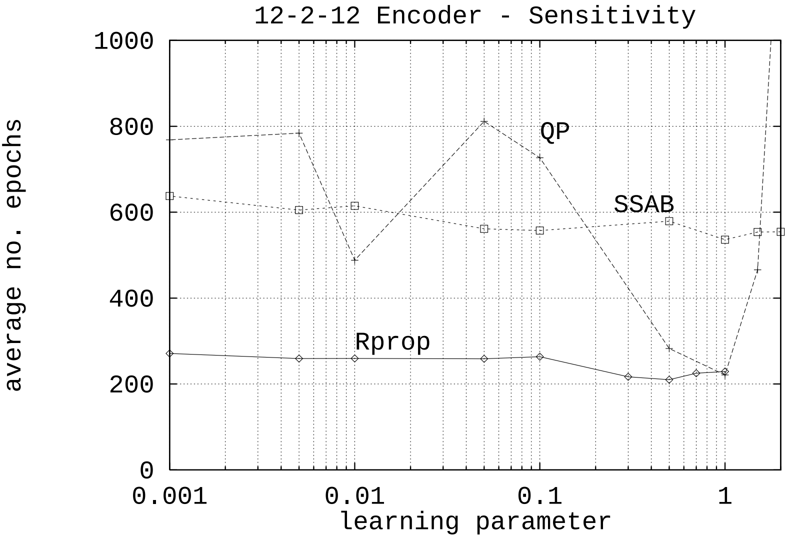
Quelle:

Lernverhalten und Parameterwahl - 10 Encoder



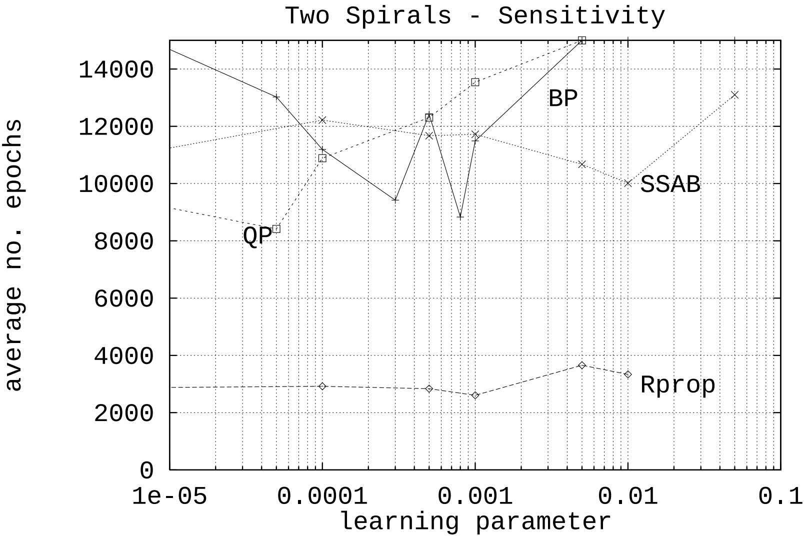
Quelle:

Lernverhalten und Parameterwahl - 12 Encoder



Quelle:

Lernverhalten und Parameterwahl - 'two sprials'



Quelle:

Real-world examples: sales rate prediction

- Bild-Zeitung is the most frequently sold newspaper in Germany, approx. 4.2 million copies per day
- it is sold in 110 000 sales outlets in Germany, differing in a lot of facets

Real-world examples: sales rate prediction

- Bild-Zeitung is the most frequently sold newspaper in Germany, approx. 4.2 million copies per day
- it is sold in 110 000 sales outlets in Germany, differing in a lot of facets
- problem: how many copies are sold in which sales outlet?

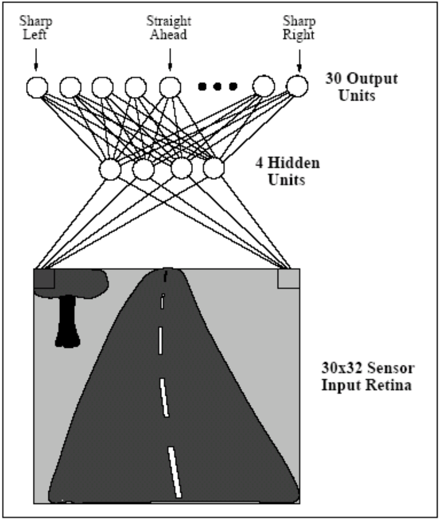
Real-world examples: sales rate prediction

- Bild-Zeitung is the most frequently sold newspaper in Germany, approx. 4.2 million copies per day
- it is sold in 110 000 sales outlets in Germany, differing in a lot of facets
- problem: how many copies are sold in which sales outlet?
- neural approach: train a neural network for each sales outlet, neural network predicts next week's sales rates
- system in use since mid of 1990s

Examples: Alvinn (Dean, Pommerleau, 1992)

- autonomous vehicle driven by a multi-layer perceptron
- input: raw camera image
- output: steering wheel angle
- generation of training data by a human driver
- drives up to 90 km/h
- 15 frames per second

Alvinn MLP structure



Alvinn Training aspects

- training data must be 'diverse'
- training data should be balanced (otherwise e.g. a bias towards steering left might exist)
- if human driver makes errors, the training data contains errors
- if human driver makes no errors, no information about how to do corrections is available
- generation of artificial training data by shifting and rotating images

Summary

- MLPs are broadly applicable ML models
- continuous features, continuos outputs
- suited for regression and classification
- learning is based on a general principle: gradient descent on an error function
- powerful learning algorithms exist
- likely to overfit $\Rightarrow$ regularisation methods