Neural Networks

doc. RNDr. Iveta Mrázová, CSc.

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE AND MATHEMATICAL LOGIC

FACULTY OF MATHEMATICS AND PHYSICS, CHARLES UNIVERSITY IN PRAGUE

Neural Networks:

Multi-layered Neural Networks

doc. RNDr. Iveta Mrázová, CSc.

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE AND MATHEMATICAL LOGIC

FACULTY OF MATHEMATICS AND PHYSICS, CHARLES UNIVERSITY IN PRAGUE

Neural Networks:

Contents:

- **Perceptron and Linear Separability**
- **Multi-layered Neural Networks**
- **Multi-layered Neural Networks: analysis of their properties**

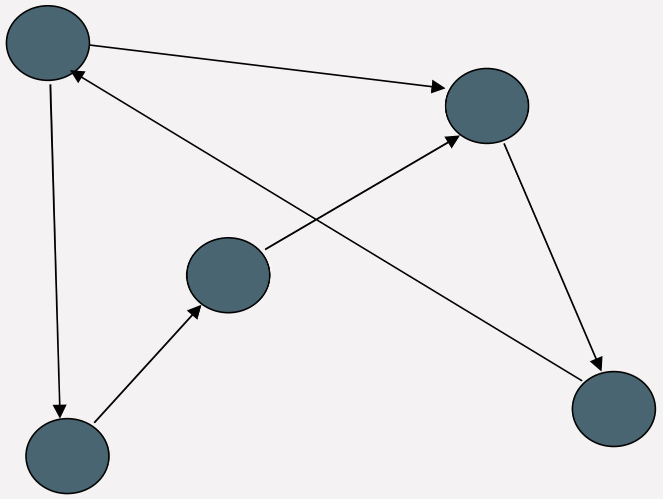
Contents:

Multi-layered neural networks (1)

D **A neural network** is a 6-tuple $M = (N, C, I, O, w, t)$, where:

- N is a finite non-empty set of neurons,
- $C \subseteq N \times N$ is a non-empty set of oriented interconnections among neurons
- $I \subseteq N$ is a non-empty set of input neurons
- $O \subseteq N$ is a non-empty set of output neurons
- $w: C \rightarrow R$ is a weight function
- $t: N \rightarrow R$ is a threshold function

  (R is the set of all real numbers)
- (N, C) is called the inter-connection graph of M



Multi-layered neural networks (2)

D **A Back-Propagation network** (BP-network) B is a neural network with a directed acyclic inter-connection graph. Its set of neurons consists of a sequence of $l+2$ pairwise disjunctive non-empty subsets called layers.

- The first layer called **the input layer** is the set of all input neurons of B, these neurons have no predecessors in the inter-connection graph; their input value x equals their output value.
- The last layer called **the output layer** is the set of all output neurons of B; these neurons are those having no successors in the inter-connection graph.
- All other neurons called hidden neurons are grouped in the remaining l **hidden layers**.

Neural Networks:

Contents:

Contents:

Back-propagation training algorithm (1)

The aim: find such a set of weights that ensure that for each input vector, the output vector produced by the network is the same as (or sufficiently close to) the desired output vector

The actual or desired output values of the hidden neurons are not specified by the task.

- For a fixed, finite training set, the objective function represents the total error between the desired and actual outputs of all the output neurons in the BP-network taken for all the training patterns.

Back-propagation training algorithm (2)

The Error Function

- corresponds to the difference between the actual and desired network output:

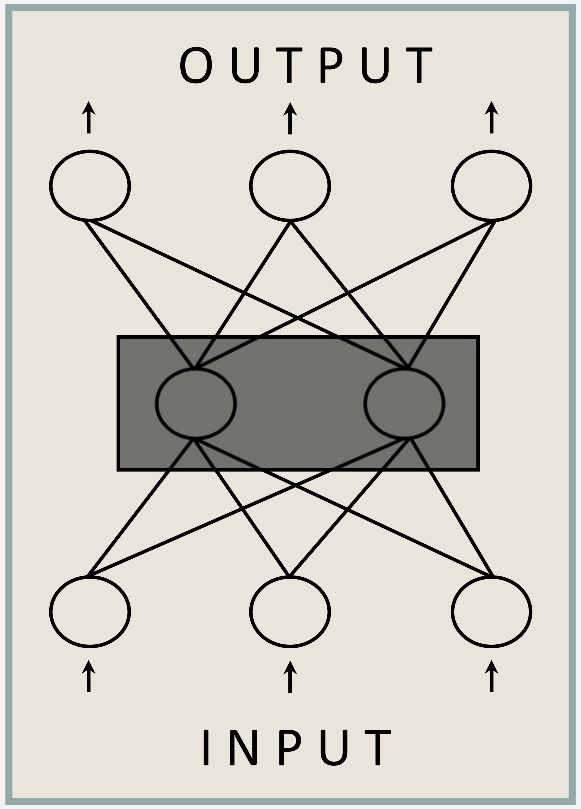
desired output

$$E = \frac{1}{2} \sum_{p} \sum_{j} \left(y_{j,p} - d_{j,p} \right)^2$$

patterns output neurons actual output

- during training, this difference should be minimized on the given training set $\Longrightarrow$ **the back-propagation training algorithm**

Multi-layered neural networks
(BP-networks)



- produce the actual output for the presented input pattern
- compare the actual and desired outputs
- adjust the weights and thresholds
  - against the gradient of the error function
  - from the output layer towards the input layer

BP-networks: adjustment rules (1)

Synaptic weights are adjusted against the gradient:

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

$\Delta_E w_{ij}(t)$ ……. the change of w_{ij} to minimize E

$$\Delta_E w_{ij} = -\frac{\overset{\text{error at network output}}{\partial E}}{\partial w_{ij}} = -\frac{\partial E}{\underset{\text{actual output}}{\partial y_j}} \frac{\partial y_j}{\partial \xi_j} \frac{\overset{\text{potential of the neuron } j}{\partial \xi_j}}{\underset{\text{connection weight}}{\partial w_{ij}}}$$

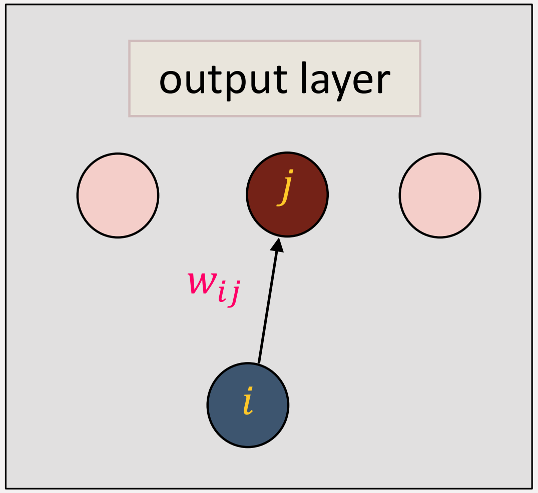
BP-networks: adjustment rules (2)

Weight adjustment in the output layer:

$$\Delta_E w_{ij} \cong -\frac{\partial E}{\partial w_{ij}} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j}\frac{\partial \xi_j}{\partial w_{ij}} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j}\frac{\partial}{\partial w_{ij}}\sum_{i'} w_{i'j} y_{i'} =$$

$$= -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j} y_i = -\frac{\partial E}{\partial y_j} f'(\xi_j)\, y_i =$$

$$= -(y_j - d_j)\, f'(\xi_j)\, y_i = \delta_j\, y_i$$



BP-networks: adjustment rules (3)

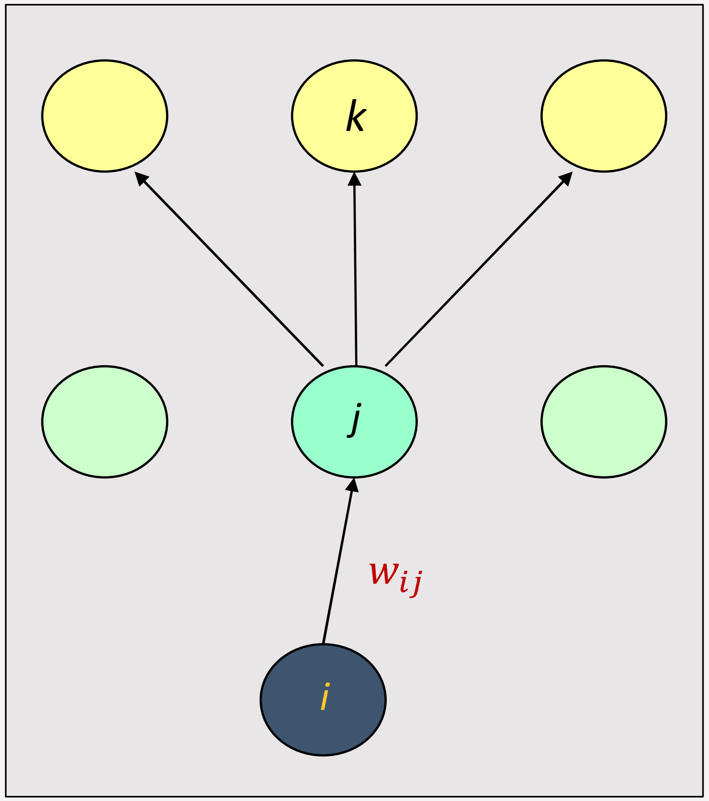
Weight adjustment in hidden layers:

$$\Delta_E w_{ij} \cong -\frac{\partial E}{\partial w_{ij}} = -\left(\sum_k \frac{\partial E}{\partial \xi_k}\frac{\partial \xi_k}{\partial y_j}\right)\frac{\partial y_j}{\partial \xi_j}\, y_i =$$

$$= -\left(\sum_k \frac{\partial E}{\partial \xi_k}\frac{\partial}{\partial y_j}\sum_{j'} w_{j'k} y_{j'}\right)\frac{\partial y_j}{\partial \xi_j}\, y_i =$$

$$= -\left(\sum_k \frac{\partial E}{\partial \xi_k} w_{jk}\right)\frac{\partial y_j}{\partial \xi_j}\, y_i =$$

$$= \left(\sum_k \delta_k w_{jk}\right) f'(\xi_j)\, y_i = \delta_j\, y_i$$



BP-networks: adjustment rules (4)

- The derivative of the sigmoidal transfer function is:

$$f'(\xi_j) = \lambda\, y_j\, (1 - y_j)$$

- Weight adjustment according to:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \delta_j y_i + \alpha_m \left(w_{ij}(t) - w_{ij}(t-1) \right)$$

where:

$$\delta_j = \begin{cases} (d_j - y_j)\lambda y_j (1 - y_j) & \text{for an output neuron} \\ \left(\sum_k \delta_k w_{jk}\right)\lambda y_j (1 - y_j) & \text{for a hidden neuron} \end{cases}$$

Back-propagation training algorithm (1)

Step 1: Initialize the weights to small random values

Step 2: Present a new training pattern in the form of:

[*input* $\vec{x}$, *desired output* $\vec{d}$]

Step 3: Calculate actual output in each layer, the activity of the neurons is given by:

$$y_j = f(\xi_j) = \frac{1}{1+e^{-\lambda \xi_j}}, \quad \text{where} \quad \xi_j = \sum_i y_i w_{ij}$$

The activities expressed in this way form the input of the following layer.

Back-propagation training algorithm (2)

Step 4: Weight adjustment starts at the output layer and proceeds back towards the input layer according to:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha\delta_j y_i + \alpha_m \left(w_{ij}(t) - w_{ij}(t-1)\right)$$

$$\delta_j = \begin{cases} (d_j - y_j)\lambda y_j(1-y_j) & \text{for an output neuron} \\ \left(\sum_k \delta_k w_{jk}\right)\lambda y_j(1-y_j) & \text{for a hidden neuron} \end{cases}$$

$w_{ij}(t)$ .... weight from neuron $\boldsymbol{i}$ to neuron $\boldsymbol{j}$ in time $\boldsymbol{t}$
α , α_m .... learning rate, resp. moment ($\boldsymbol{0 \leq \alpha,\ \alpha_m \leq 1}$)
ξ_j, resp. δ_j ...... potential, resp. local error on neuron $\boldsymbol{j}$
k .... index for the neurons from the layer above the neuron $\boldsymbol{j}$
$\boldsymbol{\lambda}$ .... slope of the transfer function

Step 5: Repeat by going to Step 2

An alternative example:

the sample multi-class labels are one hot binary vectors

The SOFTMAX transfer function is used for the output neurons (indexed by j'):

(all the desired output values are either 0 or 1; when using one-hot encoding, there is just one positive class (for the neuron j), all the other ones are negative)

$$y_j = \frac{e^{\xi_j}}{\sum_{j'} e^{\xi_{j'}}} \text{ , then: } \frac{\partial y_j}{\partial \xi_j} = \frac{\partial}{\partial \xi_j}\left(\frac{e^{\xi_j}}{\sum_{j'} e^{\xi_{j'}}}\right) = \frac{\left(e^{\xi_j}\right)' \sum_{j'} e^{\xi_{j'}} - e^{\xi_j}\left(\sum_{j'} e^{\xi_{j'}}\right)'}{\left(\sum_{j'} e^{\xi_{j'}}\right)^2} =$$

$$= \frac{e^{\xi_j} \sum_{j'} e^{\xi_{j'}}}{\left(\sum_{j'} e^{\xi_{j'}}\right)^2} - \frac{e^{\xi_j} e^{\xi_j}}{\left(\sum_{j'} e^{\xi_{j'}}\right)^2} = y_j\left(1 - y_j\right) \text{ for the derivative according to } \xi_j$$

and:
$$\frac{\partial y_j}{\partial \xi_k} = \frac{\partial}{\partial \xi_k}\left(\frac{e^{\xi_j}}{\sum_{j'} e^{\xi_{j'}}}\right) = \frac{\left(e^{\xi_j}\right)' \sum_{j'} e^{\xi_{j'}} - e^{\xi_j}\left(\sum_{j'} e^{\xi_{j'}}\right)'}{\left(\sum_{j'} e^{\xi_{j'}}\right)^2} = \frac{0 \,.\, \sum_{j'} e^{\xi_{j'}}}{\left(\sum_{j'} e^{\xi_{j'}}\right)^2} - \frac{e^{\xi_j} e^{\xi_k}}{\left(\sum_{j'} e^{\xi_{j'}}\right)^2} =$$

$$= 0 - \frac{e^{\xi_j} e^{\xi_k}}{\left(\sum_{j'} e^{\xi_{j'}}\right)^2} = -\, y_j\, y_k \text{ for the derivative according to } \xi_k \text{ with } k \neq j$$

An alternative example:

the sample multi-class labels are one hot binary vectors

Cross entropy loss function (~ negative log-likelihood)

$$L = -\sum_{j'} d_{j'} \log y_{j'}\,, \quad \text{then}$$

$$\frac{\partial L}{\partial \xi_j} = \frac{\partial}{\partial \xi_j}\left(-\sum_{j'} d_{j'} \log y_{j'}\right) = -\sum_{j'} d_{j'} \frac{\partial \log y_{j'}}{\partial y_{j'}} \frac{\partial y_{j'}}{\partial \xi_j} =$$

$$= -d_j \frac{1}{y_j} y_j \left(1 - y_j\right) - \sum_{j' \neq j} d_{j'} \frac{1}{y_{j'}} \left(-y_{j'} y_j\right) =$$

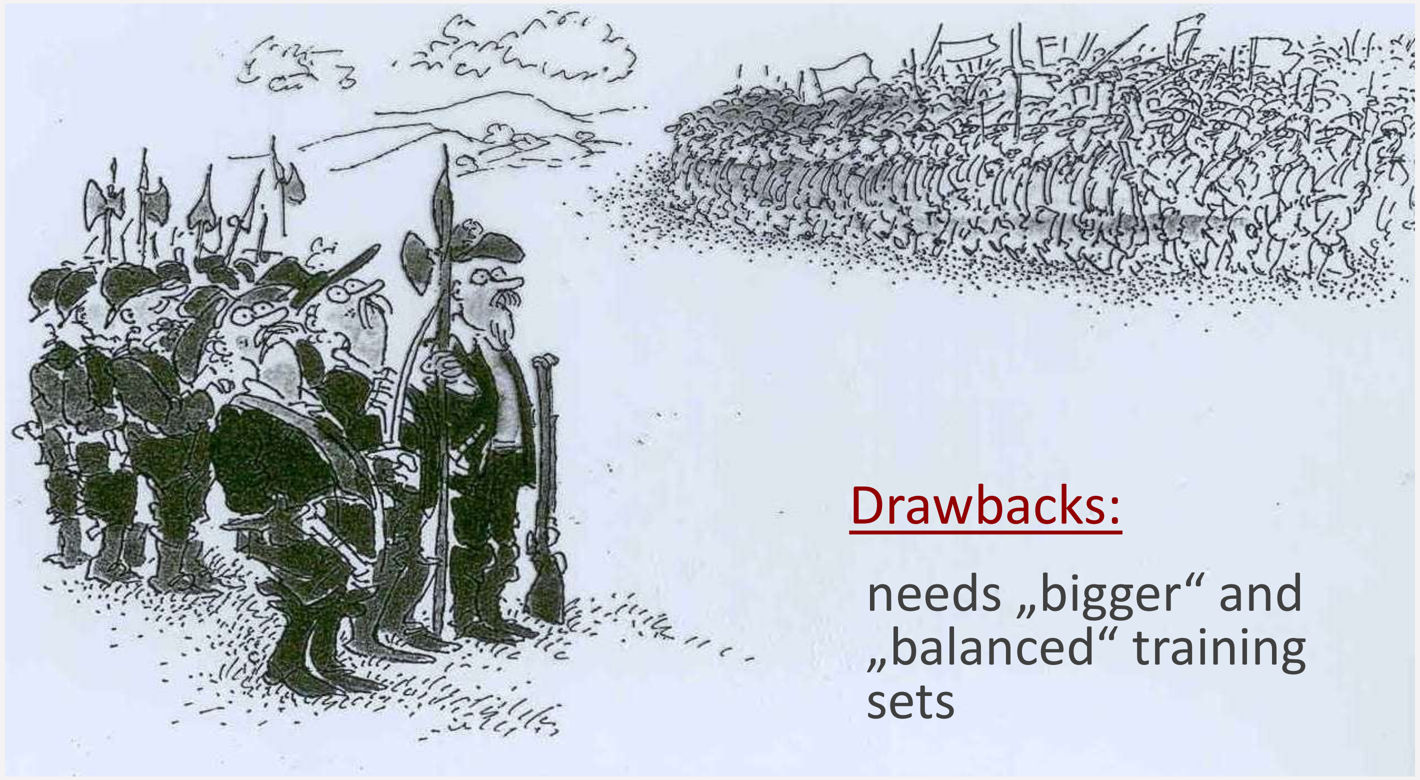
$$= -d_j \left(1 - y_j\right) + \sum_{j' \neq j} d_{j'}\, y_j = -d_j + y_j \sum_{j'} d_{j'}$$

Altogether, we obtain: $\frac{\partial L}{\partial \xi_j} = y_j \sum_{j'} d_{j'} - d_j = y_j - d_j$

BP-networks: analysis of the model

- Simple training algorithm
- A very often used approach
- Relatively good results
- Drawbacks:
  - Internal knowledge representation – „black box“
  - the number of neurons and generalization capabilities
    - pruning and retraining
  - error function (knowledge of the desired outputs)
    - „bigger“ and „balanced“ training sets
    - assessment of network outputs during recall

BP-networks: analysis of the model



Drawbacks:

needs „bigger“ and „balanced“ training sets

Neural Networks:

Contents:

Contents:

Back-propagation training algorithm: speeding-up the training process (1)

- The standard back-propagation training algorithm is rather slow
  - → a malicious selection of network parameters can make it even slower
- For artificial neural networks, the learning problem is NP-complete in the worst case
  - → computational complexity grows exponentially with the number of the variables
  - → despite of that the standard back-propagation performs often better than many „fast learning algorithms“
    - especially when the task achieves a realistic level of complexity and the size of the training set goes beyond a critical threshold

Back-propagation training algorithm: speeding-up the training process (2)

Algorithms speeding-up the training process:

- **Keeping a fixed network topology**
- **Modular networks**
  - considerable improvement of network approximation abilities
- **Adjustment of both the parameters** (weights, thresholds, etc.) **and the network topology**

Back-propagation training algorithm: initial weight selection (1)

- The weights should be uniformly distributed over the interval $[-a, +a]$
- Zero mean value
  - leads to an expected zero value of the total input to each node in the network (potential)
- The derivative of the sigmoidal transfer function is reached its maximum for zero (~ 0.25)
  - larger values of the backpropagated errors
  - more significant weight updates when training starts

Back-propagation training algorithm: initial weight selection (2)

Problem:

- **Too small weights paralyze learning**
  - The error backpropagated from the output layer to hidden layers is too small
- **Too large weights lead to saturation of neurons and slow learning** (in flat zones of the error function)

→ **Learning then stops at a suboptimal local minimum**

× the right choice of initial weights can significantly reduce the risk of getting stuck in a local minimum

Back-propagation training algorithm: initial weight selection (3)

Reduce the danger of local minima:

~ initialize the weights with small random values

Motivation:

- **Small weight values**
  - Large weight values impact saturation of hidden neurons (too active or too passive for all training patterns) → such neurons are incapable of further training (the derivative of the transfer function – sigmoid – is almost zero)
- **Random weight values**
  - The goal is to „break the symmetry“ → hidden neurons should specialize in the recognition of different features

Back-propagation training algorithm: initial weight selection (4)

IDEA:

- **The potential of a hidden neuron is given by:**

$$\xi = w_0 + w_1 x_1 + \cdots + w_n x_n$$

w_0 is the threshold

x_i … the activity of the i-th neuron from the preceding layer

w_i …the weight from the i-th neuron from the preceding layer

- **Expected value of the potential for hidden neurons:**

$$\mathrm{E}\{\xi_j\} = \mathrm{E}\left\{\sum_{i=0}^{n} w_{ij} x_i\right\} = \sum_{i=0}^{n} \mathrm{E}\{w_{ij}\}\ \mathrm{E}\{x_i\} = 0$$

  - the weights are independent of the input patterns
  - the weights are random variables with zero mean

Back-propagation training algorithm: initial weight selection (5)

IDEA - continue:

- **The variance of the potential ξ is given by:**

$$\sigma_\xi^2 = \mathrm{E}\left\{(\xi_j)^2\right\} - \underbrace{\mathrm{E}^2\{(\xi_j)\}}_{=0} = \mathrm{E}\left\{\left(\sum_{i=0}^{n} w_{ij}\ x_i\right)^2\right\} - 0 =$$

$$= \sum_{i,k=0}^{n} \mathrm{E}\{(w_{ij} w_{kj}\ x_i\ x_k)\} =$$

mutual independence for all ***j***

$$= \sum_{i=0}^{n} \mathrm{E}\left\{(w_{ij})^2\right\}\ \mathrm{E}\{(x_i)^2\}$$

Back-propagation training algorithm: initial weight selection (6)

IDEA - continue:

- **Further, we assume** that the training patterns are normalized and from the interval $[0,1]$. Then:

$$\mathrm{E}\{\,(x_i)^2\} \;=\; \int_0^1 x_i^2\;\mathrm{d}x \;=\; \frac{x^3}{3}\bigg|_0^1 = \frac{1}{3}$$

- Assumed that the weights of the hidden neurons are also random variables with a zero mean and uniformly distributed in the interval $\langle -a, a\rangle$, then:

$$\mathrm{E}\left\{\left(w_{ij}\right)^2\right\} = \int_{-a}^{a} w_{ij}^2 \cdot \frac{1}{2a}\mathrm{d}w_{ij} \;=\frac{w_{ij}^3}{6a}\bigg|_{-a}^{a} = \frac{a^2}{3}$$

- N … number of weights leading to the considered neuron $(= n + 1)$

Back-propagation training algorithm: initial weight selection (7)

IDEA - continue:

- **Standard deviation will thus correspond to:**

$$A = \sigma_\xi = \sqrt{N}\frac{a}{3} \qquad \left(\rightarrow \quad a = A\frac{3}{\sqrt{N}}\right)$$

- **Neuron potential should be a random variable with the standard deviation** A (that is moreover independent of the number of weights leading to this neuron);
- **Select initial weights (roughly) from the interval:**

$$\left[-\frac{3}{\sqrt{N}} \cdot A, \frac{3}{\sqrt{N}} \cdot A\right]$$

  - especially for $A = 1$ large gradient (i.e., quick learning)

Neural Networks:

Contents:

- **Perceptron and Linear Separability**
- **Multi-layered Neural Networks**
- **Multi-layered Neural Networks: analysis of their properties**

Contents:

Back-propagation training algorithm: speeding-up the training process (3)

- first-order methods work with steepest-descent directions
- modifications to the basic form of steepest-descent:
  - need to reduce step sizes with algorithm progression
    - learning rate decay
  - need a way of avoiding local optima
    - add momentum term
  - need to address widely varying slopes with respect to different weight parameters

Back-propagation training algorithm: speeding-up the training process (4)

Learning rate decay:

- initial learning rates should be high but decrease over time
- the two most common decay functions are *exponential decay* and *inverse decay*
- the learning rate α_t can be expressed in terms of the initial decay rate α_0 and time *t* as follows:

$$\alpha_t = \exp(-k \cdot t) \qquad \text{exponential decay}$$

$$\alpha_t = \frac{\alpha_0}{1+k \cdot t} \qquad \text{inverse decay}$$

the parameter *k* controls the rate of the decay

Neural Networks:

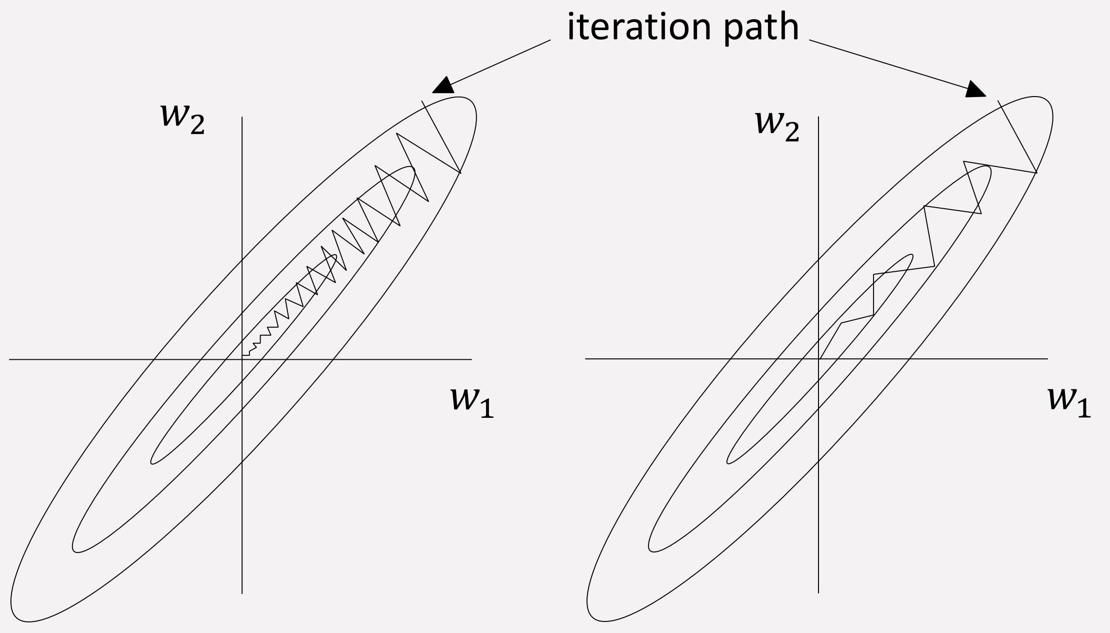
Contents:

- **Perceptron and Linear Separability**
- **Multi-layered Neural Networks**
- **Multi-layered Neural Networks: analysis of their properties**

Contents:

Back-propagation training algorithm with momentum (1)

Minimization of the error function with the gradient method



without the momentum term

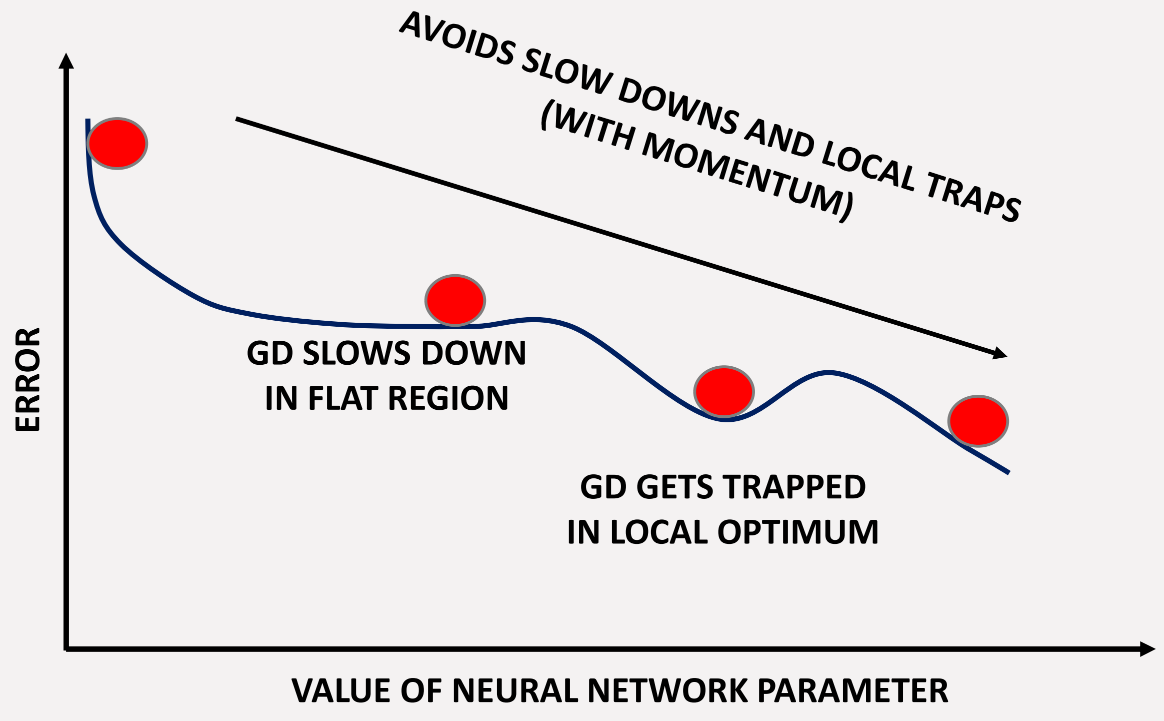
with the momentum term

Back-propagation training algorithm with momentum (2)

- When the minimum of the error function lies in a „narrow valley“ for the given task, following the gradient direction can lead to wide oscillations of the search process
- Solution: **introduce a momentum term**
  - a weighted average of the current gradient and the previous correction direction is computed at each step

    → **Inertia ~ could help to avoid excessive oscillations in „narrow valleys of the error function“**

Back-propagation training algorithm with momentum (3)



AVOIDS SLOW DOWNS AND LOCAL TRAPS (WITH MOMENTUM)
ERROR
GD SLOWS DOWN IN FLAT REGION
GD GETS TRAPPED IN LOCAL OPTIMUM
VALUE OF NEURAL NETWORK PARAMETER

- **The idea of a marble rolling down the hill:**
- Use a *friction parameter* $\alpha_m \in (0, 1)$ to gain speed in the direction of movement:

$$\Delta w(i+1) = \alpha_m \, \Delta w(i) - \alpha \frac{\partial E}{\partial w}$$

$$w(i+1) = w(i) + \Delta w(i+1)$$

Back-propagation training algorithm with momentum (4)

- For a network with n different weights $w_1, \dots, w_n$ the correction of w_k at time $i+1$ is given by:

$$\Delta\, w_k(i+1) = -\,\alpha\,\frac{\partial E}{\partial w_k(i)} + \alpha_m \Delta w_k(i) =$$

$$= -\,\alpha\,\frac{\partial E}{\partial w_k(i)} + \alpha_m\big(w_k(i) - w_k(i-1)\big)$$

where: $\boldsymbol{\alpha}$ ….. learning rate

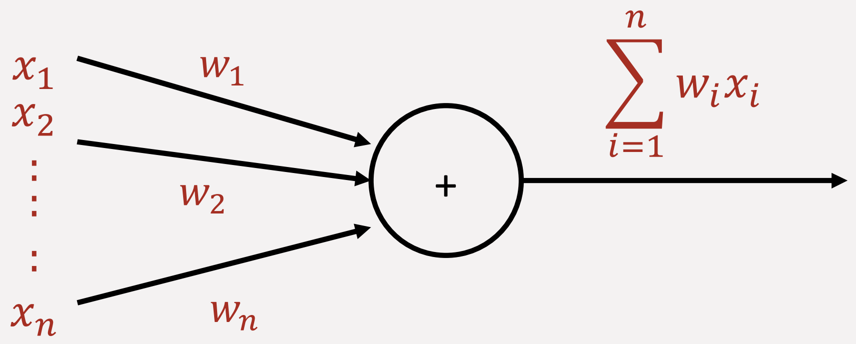
$\boldsymbol{\alpha_m}$ … momentum rate

Back-propagation training algorithm with momentum (5)

- In order to accelerate convergence to the minimum of the error function:
  - Increase the learning rate up to an optimum value α, that still guarantees convergence of the training process
  - Introduction of the momentum rate allows to attenuate the oscillations that might occur during training
- Optimal values for α and α_m highly depend on the character of the respective learning task

Back-propagation training algorithm with momentum (6)

EXAMPLE: Linear transfer function, p patterns



X....... matrix $(p \times n)$; $\vec{d}$....vector

$$X = \begin{pmatrix} x_1^{(1)} & \cdots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(p)} & \cdots & x_n^{(p)} \end{pmatrix}; \quad \vec{d} = \begin{pmatrix} d^{(1)} \\ \vdots \\ d^{(p)} \end{pmatrix}$$

→ MINIMIZATION OF E:

$$E = \|X\vec{w} - \vec{d}\|^2 = (X\vec{w} - \vec{d})^T (X\vec{w} - \vec{d}) = \vec{w}^T (X^T X)\vec{w} - 2\vec{d}^T X\vec{w} + \vec{d}^T \vec{d}$$

Back-propagation training algorithm with momentum (7)

- E is a quadratic function

  → the minimum can be found using gradient descent

  Interpretation: E has the form of a paraboloid in the $n-$dimensional space; its shape is determined by the eigenvalues of the correlation matrix $X^T X$

  → Gradient descent is most effective when the principal axes are all of the same length

  → When the axes are of very different sizes, the gradient direction can lead to oscillations

Back-propagation training algorithm
with momentum (8)

- Excessive oscillations can be prevented by choosing a small value for α and a larger value for the momentum parameter α_m

× **too small values** of α

  $\rightarrow$ the danger of local minima

× **too big values** of α

  $\rightarrow$ the danger of oscillations

Back-propagation training algorithm with momentum (9)

In the nonlinear case, the gradient of the error function is almost zero in the regions far from local minima – possibility of oscillations

→ in such a case, larger learning rates could help to return back to „convex“ regions of the error function

Solution:

- Nesterov momentum
- Adaptive learning rates
- Pre-processing of the training set
  - decorrelation on input patterns (PCA, …)

Back-propagation training algorithm with momentum (10)

Nesterov Momentum:

~ a modification of the traditional momentum method:

the gradients are computed at a point that would be reached after executing the α_m −discounted version of the previous step

- Computes the gradient at a point reached using the momentum portion of the previous update:

$$\Delta\, w(i+1) = \alpha_m \Delta w(i) - \alpha \frac{\partial E\big(w + \alpha_m \Delta\, w(i)\big)}{\partial w} \;\; ; \;\; w(i+1) = w(i) + \Delta\, w(i+1)$$

- Slows down as the marble reaches near bottom of the hill
- Should be used rather with mini-batch stochastic gradient descent (SGD)

Neural Networks:

Contents:

- **Perceptron and Linear Separability**
- **Multi-layered Neural Networks**
- **Multi-layered Neural Networks: analysis of their properties**

Contents:

Back-propagation training algorithm: strategies using adaptive learning rates (1)

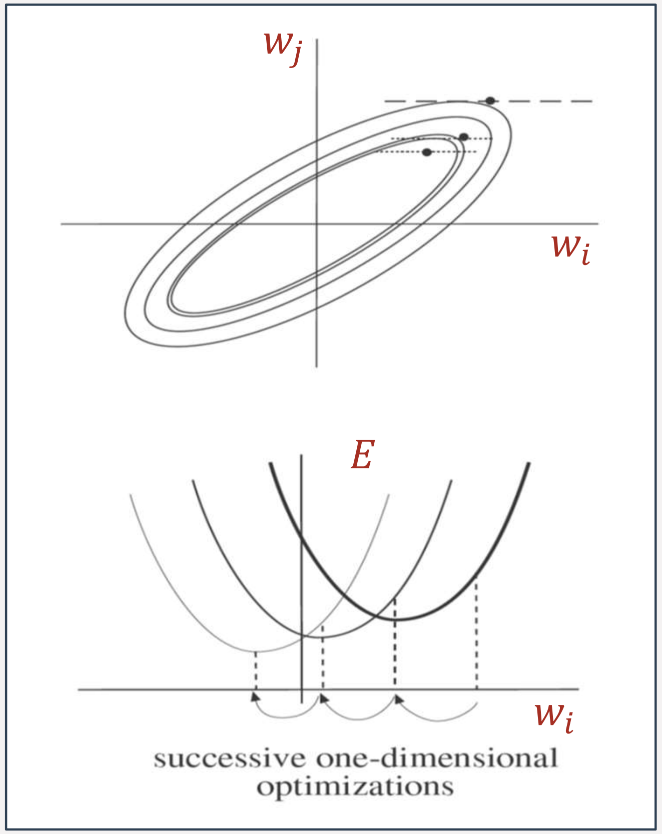
1. **Adaptive learning rates:**

   a local parameter α_i for each weight w_i

   weight adjustment: $\Delta w_i = - \alpha_i \frac{\partial E}{\partial w_i}$

- **Variants of the algorithm:**
  - Silva & Almeida
  - Delta-bar-delta
  - Super SAB

The algorithm of Silva & Almeida (1)



successive one-dimensional optimizations

Assumed: the network has n weights

- Quadratic error function:

$$c_1^2 w_1^2 + c_2^2 w_2^2 + \ldots + c_n^2 w_n^2 + \sum_{i \neq j} d_{ij} w_i w_j + C$$

- The step in the i-th direction minimizes

$$c_i^2 w_i^2 + k_1 w_i + k_2$$

k_1, k_2 are constants, which depend on the values of the „frozen“ variables at the current iteration point (c_i determine the curvature of the parabola)

The algorithm of Silva & Almeida (2)

The heuristic:

- ACCELERATE, if in two successive iterations, the sign of the partial derivative has not changed
- DECELERATE, if the sign changes

$\nabla_i E^{(k)}$ … partial derivative of the error function with respect to the weight w_i at the k-th iteration

$\alpha_i^{(0)}$ . . . . initial learning rates $(i = 1, \dots , n)$ initialized to a small positive value

The algorithm of Silva & Almeida (3)

- In the k-th iteration the value of the learning rate for the next step is recomputed for each weight by:

$$\alpha_i^{(k+1)} = \begin{cases} \alpha_i^{(k)} u, & \text{if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} > 0 \\ \alpha_i^{(k)} d, & \text{if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0 \end{cases}$$

- The constants u and d are set by hand with $u > 1$ and $d < 1$
- Weight adjustment: $\Delta^{(k)} w_i = -\alpha_i^{(k)} \nabla_i E^{(k)}$

The algorithm of Silva & Almeida (4)

Problems:

- The learning rates grow and decrease exponentially with regard to u and d

  $\rightarrow$ Problems can occur if too many acceleration steps are performed successively

The algorithm Delta-Bar-Delta

- Acceleration is done with more caution than deceleration (especially from small initial weights)
- k –th iteration: $\alpha_i^{(k+1)} = \begin{cases} \alpha_i^{(k)} + u, & \text{if } \nabla_i E^{(k)} \cdot \bar{\delta}_i^{(k-1)} > 0 \\ \alpha_i^{(k)} \cdot d, & \text{if } \nabla_i E^{(k)} \cdot \bar{\delta}_i^{(k-1)} < 0 \\ \alpha_i^{(k)} & \text{else} \end{cases}$

  u, d … fixed pre-set constants

  $\bar{\delta}_i^{(k)} = (1-\Phi)\ \nabla_i E^{(k)} + \Phi\ \bar{\delta}_i^{(k-1)}$, where Φ is a constant
- **Weight updates without momentum:** $\Delta^{(k)} w_i = -\alpha_i^{(k)}\ \nabla_i E^{(k)}$

What are the domains for u, d, Φ?

Algorithm Super SAB

- **SAB ~** <u>S</u>elf-<u>A</u>dapting <u>B</u>ack-propagation
- **Adaptive acceleration strategy** for the back-propagation training algorithm
  - Of order quicker that the original back-propagation algorithm
  - Relatively stable
  - Robust against the choice of initial parameters
- **Uses momentum:**
  - Accelerates convergence in flat areas of the weight space
  - In steep areas of the weight space, the momentum term curbs oscillations caused by changed signs of the gradient

Super SAB – the training algorithm (1)

α^{+} ……….. multiplicative constant to increase the learning rates ($\alpha^{+} = 1.05$)
α^{-} ……….. multiplicative constant to decrease the learning rates ($\alpha^{-} = 2$)
α_{START} …. initial value for the parameter α_{ij} $\forall j$, ($\alpha_{START} = 1.2$)
α_{m} ……….. momentum ($\alpha_{m} = 0.3$)

Step 1: set all α_{ij} to the initial value α_{START}

Step 2: perform Step(t) of back-propagation *with momentum*

Step 3: if the derivative (according to w_{ij}) did not change its sign, increase the learning rates ($\forall w_{ij}$): $\alpha_{ij}(t+1) = \alpha^{+} \cdot \alpha_{ij}(t)$

Super SAB – the training algorithm (2)

Step 4: if the derivative (according to w_{ij}) changed its sign:

- annul the previous weight change (that caused the change in the sign of the gradient): $\Delta w_{ij}(t+1) = -\Delta w_{ij}(t)$
- use smaller learning rates: $\alpha_{ij}(t+1) = \alpha_{ij}(t)/\alpha^{-}$
- and set: $\Delta w_{ij}(t+1) = 0$

  (the change from the previous step will be thus not considered in the next training step)

Step 5: go to Step 2

AdaGrad – Adaptive Gradients (1)

~ *Aggregate* squared magnitudes of the i-th partial derivative $\partial E / \partial w_i$ in the form of A_i

- The square root of A_i is proportional to the root-mean-square slope ∂E
  - the absolute value of A_i will increase over time:

$$A_i\,(t+1) = A_i(t) + \left(\frac{\partial E}{\partial w_i}\right)^2; \qquad \forall i$$

  - The update for the i-th parameter w_i is as follows:

$$w_i\,(t+1) = w_i(t) - \frac{\alpha}{\sqrt{A_i}}\frac{\partial E}{\partial w_i}; \qquad \forall i$$

  - Use $\sqrt{A_i + \varepsilon}$ in the denominator to avoid ill-conditioning; ε is a small positive number, e.g., 10^{-8}

AdaGrad – Adaptive Gradients (2)

Intuition:

- Scaling the derivative inversely with $\sqrt{A_i}$ (the square root of the aggregated squared gradient) encourages faster *relative* movements along gently sloping directions
  - Absolute movements tend to slow down prematurely due to the aggregated values of the entire history of partial derivatives
  - Scaling parameters use stale values that can increase inaccuracy

RMSProp: Root Mean Squared Propagation (1)

- uses *exponential smoothing* of the aggregated values using the parameter $\rho \in (0, 1)$ in the relative estimations of the gradients
  - Absolute magnitudes of scaling factors A_i do not grow with time.
  - Problem of staleness is ameliorated

$$A_i\,(t+1) = \rho\, A_i(t) + (1-\rho)\left(\frac{\partial E}{\partial w_i}\right)^2; \quad \forall i$$

$$w_i\,(t+1) = w_i(t) - \frac{\alpha}{\sqrt{A_i}}\,\frac{\partial E}{\partial w_i}; \quad \forall i$$

  - Use $\sqrt{A_i + \varepsilon}$ in the denominator to avoid ill-conditioning; ε is a small positive number, e.g., 10^{-8}

RMSProp: Root Mean Squared Propagation (2)

- Possibility to *combine RMSProp with Nesterov Momentum*:

$$\Delta w_i\,(t+1) = \alpha_m\,\Delta w_i(t) - \frac{\alpha}{\sqrt{A_i}}\ \frac{\partial E(w_i + \alpha_m \Delta w_i(t))}{\partial w_i}$$

$$w_i(t+1) = w_i(t) + \Delta w_i(t+1)\ ; \qquad \forall i$$

- Maintenance of A_i is done with shifted gradients as well

$$A_i(t+1) = \rho\, A_i(t) + (1-\rho)\left(\frac{\partial E(w_i + \alpha_m \Delta w_i(t))}{\partial w_i}\right)^2 ; \quad \forall i$$

AdaDelta and Adam

- Both methods derive intuition from RMSProp
  - AdaDelta keeps track of an exponentially smoothed value of the *incremental changes* of weights Δw_i in previous iterations to decide parameter-specific learning rates
  - Adam keeps track of *exponentially smoothed gradients from previous iterations* (in addition to normalizing like RMSProp)
    - Adam is an extremely popular method

Adam – Adaptive Moment Estimation (1)

- Efficient first-order stochastic optimization method
- Combines the advantages of:
  - **AdaGrad** – works well with sparse gradients
  - **RMSProp** – works well in non-stationary settings
- **The main idea:**
  - Maintain exponential moving averages for the gradient and its square
  - Update the parameters proportionally to $\frac{\boldsymbol{average\ gradient}}{\sqrt{\boldsymbol{average\ squared\ gradient}}}$
- **Properties:** scale-invariance, bounded norm, bias correction

Adam – weight update algorithm (2)

Initialization: $A_i(0) = 0;\ N_i(0) = 0;\ \forall i$

Iteratively adjust the weights: for $t = 0, \dots, T-1$, set $\forall i$:

- $N_i(t+1) = \beta_1 N_i(t) + (1-\beta_1)\frac{\partial E}{\partial w_i}$ 1st moment (gradient) estimate
- $A_i(t+1) = \beta_2 A_i(t) + (1-\beta_2)\left(\frac{\partial E}{\partial w_i}\right)^2$ 2nd moment (squared gradient) estimate
- $\widetilde{N}_i(t+1) = N_i(t+1)/\left(1-(\beta_1)^{(t+1)}\right)$ 1st moment bias correction
- $\tilde{A}_i(t+1) = A_i(t+1)/\left(1-(\beta_2)^{(t+1)}\right)$ 2nd moment bias correction
- $w_i(t+1) = w_i(t) - \alpha\,\frac{\widetilde{N}_i(t+1)}{\sqrt{\tilde{A}_i(t+1)+\varepsilon}}$

Return the weight matrix $\boldsymbol{W}(T)$.

Hyper-parameters:
$\alpha > 0$ … learning rates (typical choice: 0.001)
$\beta_1;\ 0 \le \beta_1 < 1$ … 1st moment decay rate (typical choice: 0.9)
$\beta_2;\ 0 \le \beta_2 < 1$ … 2nd moment decay rate (typical choice: 0.999)
$\varepsilon > 0$ … numerical term (typical choice: 10^{-8})

Adam – weight update algorithm (3)

- Adam's step at iteration t (we assume $\varepsilon = 0$):

$$\Delta\, w_i(t) = -\alpha\, \frac{\tilde{N}_i(t+1)}{\sqrt{\tilde{A}_i(t+1)}}\ ;\ \ \forall i$$

- **Properties:**
  - **Scale-invariance:**

$$E(w_i)\ \rightarrow\ c\ \cdot\ E(w_i)\quad ==>\quad \tilde{N}_i(t)\ \rightarrow\ c\ \cdot \tilde{N}_i(t)\quad \wedge\quad \tilde{A}_i\,(t)\ \rightarrow\ c^2 \cdot\ \tilde{A}_i(t)$$

$==>\ \Delta\, w_i(t)$ does not change

  - **Bounded norm:**

$$\|\Delta\, w_i(t)\|_\infty \leq \begin{cases} \alpha \cdot (1-\beta_1)/\sqrt{1-\beta_2}\ , & (1-\beta_1) > \sqrt{1-\beta_2} \\ \alpha\ , & \text{otherwise} \end{cases}$$

Adam – bias correction (4)

- When considering the initialization $N_i(0) = 0 \ \ (\forall\, i)$, we get:

$$N_i(t+1) = \beta_1 N_i(t) + (1-\beta_1)\left(\frac{\partial E}{\partial w_i}\right)_t = \sum_{\tau=0}^{t}(1-\beta_1)(\beta_1)^{(t-\tau)} \cdot \left(\frac{\partial E}{\partial w_i}\right)_\tau$$

- As $\sum_{\tau=0}^{t}(1-\beta_1)(\beta_1)^{(t-\tau)} = 1-(\beta_1)^{(t+1)}$, we shall divide $N_i(t+1)$ by $1-(\beta_1)^{(t+1)}$ to obtain an unbiased estimate:

$$\tilde{N}_i(t+1) = N_i(t+1)/\left(1-(\beta_1)^{(t+1)}\right)$$

- Use an analogous argument to derive the bias correction for $A_i(t+1)$.

$\uparrow \uparrow \uparrow$ předchozí file

Neural Networks:

Contents:

- **Perceptron and Linear Separability**
- **Multi-layered Neural Networks**
- **Multi-layered Neural Networks: analysis of their properties**

Contents:

- Perceptron and Linear Separability
- Multi-layered Neural Networks
  - Back-Propagation Training Algorithm
  - Strategies to Speed-up the Training Process
    - Initial Weight Selection
    - First-order Methods
      - Learning Rate Decay
      - Training with Momentum
      - Adaptive Learning Rates
    - Second-order Algorithms
      - Quickprop
      - Levenberg-Marquardt Algorithm
      - Conjugate Gradient Methods
    - Relaxation Methods
    - Other Techniques

Back-propagation training algorithm: strategies speeding-up the training process (2)

- <u>**Second-order algorithms:**</u>
  - Consider more information about the shape of the error function than gradient $\rightarrow$ curvature of the error function
  - Second-order methods use a quadratic approximation of the error function E

    $\vec{w} = (w_1, \dots, w_n)$...... weight vector of the network

    $E(\vec{w})$ ....error function

$\rightarrow$ **The Taylor series approximating the error function E:**

$$E\left(\vec{w} + \vec{h}\right) \approx E(\vec{w}) + \nabla E(\vec{w})^T \vec{h} + \frac{1}{2} \vec{h}^T \nabla^2 E(\vec{w}) \, \vec{h}$$

$$E(\vec{w}+\vec{h}) \approx E(\vec{w}) + \nabla E(\vec{w})^T \vec{h} + \frac{1}{2}\vec{h}^T \nabla^2 E(\vec{w})\, \vec{h}$$

Second-order algorithms (2)

$\nabla E(\vec{w})$...differential

$\nabla^2 E(\vec{w})$ .... Hessian matrix $(n \times n)$ of second-order partial derivatives:

$$\nabla E(\vec{w}) = \begin{pmatrix} \dfrac{\partial E(\vec{w})}{\partial w_1} \\ \dfrac{\partial E(\vec{w})}{\partial w_2} \\ \vdots \\ \dfrac{\partial E(\vec{w})}{\partial w_n} \end{pmatrix}$$

$$\nabla^2 E(\vec{w}) = \begin{pmatrix} \dfrac{\partial^2 E(\vec{w})}{\partial w_1^2} & \dfrac{\partial^2 E(\vec{w})}{\partial w_1 \partial w_2} & \cdots & \dfrac{\partial^2 E(\vec{w})}{\partial w_1 \partial w_n} \\ \dfrac{\partial^2 E(\vec{w})}{\partial w_2 \partial w_1} & \dfrac{\partial^2 E(\vec{w})}{\partial w_2^2} & \cdots & \dfrac{\partial^2 E(\vec{w})}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 E(\vec{w})}{\partial w_n \partial w_1} & \dfrac{\partial^2 E(\vec{w})}{\partial w_n \partial w_2} & \cdots & \dfrac{\partial^2 E(\vec{w})}{\partial w_n^2} \end{pmatrix}$$

$$E(\vec{w}+\vec{h}) \approx E(\vec{w}) + \nabla E(\vec{w})^T\vec{h} + \frac{1}{2}\vec{h}^T\nabla^2 E(\vec{w})\,\vec{h}$$

Second-order algorithms (3)

→ Gradient of the error function (by differentiating $E(\vec{w}+\vec{h})$ w.r.t. $\vec{h}$):

$$\nabla E(\vec{w}+\vec{h})^T \approx \nabla E(\vec{w})^T + \vec{h}^T\nabla^2 E(\vec{w})$$

→ Gradient equal to zero (looking for the minimum of E):

$$\vec{h} = -\left(\nabla^2 E(\vec{w})\right)^{-1} \nabla E(\vec{w})$$

==> Newton´s methods:

- **Work iteratively**
- **Weight adjustment in the k-th iteration according to:**

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(\nabla^2 E(\vec{w})\right)^{-1} \nabla E(\vec{w})$$

- **Quick convergence**
- × **A problem might represent the inverse Hessian matrix**

Second-order algorithms (4)

- Pseudo-Newton methods:
  - Work with a „simplified form“ of the Hessian matrix
  - Only the diagonal elements are computed: $\left(\frac{\partial^2 E(\vec{w})}{\partial w_i^2}\right)$
  - The non-diagonal elements are all set to zero
  - Weight adjustment according to: $w_i^{(k+1)} = w_i^{(k)} - \frac{\nabla_i E(\vec{w})}{\frac{\partial^2 E(\vec{w})}{\partial w_i^2}}$

Second-order algorithms (5)

- **Pseudo-Newton methods:**
  - No matrix inversion necessary
  - Limited computational effort involved in finding the required second partial derivatives
  - Work well when the error function has a quadratic form, otherwise problems might occur since a small second-order partial derivative can lead to extremely large corrections
  - **Variants of Newton´s method:**
    - Quickprop
    - Levenberg-Marquardt algorithm

Neural Networks:

Contents:

Contents:

The algorithm Quickprop (1)

- Takes into account also second-order information
  - × Only one-dimensional minimization steps are taken
  - → Information about the curvature of the error function in the update direction is obtained from the current and past partial derivative of the error function
- Independent optimization steps for each weight
- A quadratic one-dimensional approximation of the error function is used

The algorithm Quickprop (2)

- Weight adjustment in the k-th iteration according to:

$$\vec{w}_i^{(k+1)} = \vec{w}_i^{(k)} + \Delta^{(k)} w_i$$, where

$$\Delta^{(k)} w_i = \Delta^{(k-1)} w_i \cdot \frac{\nabla_i E^{(k)}}{\nabla_i E^{(k-1)} - \nabla_i E^{(k)}}$$

Assumed: the error function has been computed at steps $(k-1)$ and k using the weight difference $\Delta^{(k-1)} w_i$ obtained from a previous Quickprop or standard gradient descent step

The algorithm Quickprop (3)

- Weight adjustment rules can be written as:

$$\Delta^{(k)} w_i = - \frac{\nabla_i E^{(k)}}{\dfrac{\nabla_i E^{(k)} - \nabla_i E^{(k-1)}}{\Delta^{(k-1)} w_i}}$$

- The denominator is just a discrete approximation to the second-order derivative $\partial^2 E(\vec{w}) / \partial w_i^2$

- Quickprop ~ discrete pseudo-Newton method, that uses the so-called „**SECANT STEP**“

Neural Networks:

Contents:

Contents:

Levenberg-Marquardt algorithm (1)

- Quicker around the minimum of the error function
- A combination of a gradient and Newton´s method
  - Gradient only update rule: $\vec{w}^{(k+1)} = \vec{w}^{(k)} - \alpha \, \nabla E^{(k)}$
  - Second-order update rule: $\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(\nabla^2 E^{(k)}\right)^{-1} \nabla E^{(k)}$
  - Levenberg – blends them together:

$$\begin{aligned} \vec{w}^{(k+1)} &= \vec{w}^{(k)} - \left(\nabla^2 E^{(k)} + \lambda I\right)^{-1} \nabla E^{(k)} \\ &= \vec{w}^{(k)} - \left(H + \lambda I\right)^{-1} \nabla E^{(k)} \end{aligned}$$

$$E(\vec{w}) = \frac{1}{2}\sum_{j=1}^{m} e_j(\vec{w}) = \frac{1}{2}\sum_{j=1}^{m}\left(y_j - d_j\right)^2; \quad \nabla E(\vec{w}) = \partial E / \partial \vec{w}$$

Levenberg-Marquardt algorithm (2)

- Hessian matrix can be ***approximated***
  - for a single output $g_i = \frac{\partial e}{\partial w_i} = 2(y-d)\frac{\partial y}{\partial w_i}$

$$\frac{\partial^2 e}{\partial w_i \, \partial w_j} = 2\left[\frac{\partial y}{\partial w_i}\frac{\partial y}{\partial w_j} + \cancel{(y-d)\frac{\partial^2 y}{\partial w_i \partial w_j}}\right]$$

  - thus instead of H in $\vec{w}^{(k+1)} = \vec{w}^{(k)} - (H + \lambda I)^{-1}\, \nabla E^{(k)}$

    we use $\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(J^{(k)^T} \cdot J^{(k)} + \lambda I\right)^{-1}\, \nabla E^{(k)}$

    where $J^{(k)} = \left(\frac{\partial y_1^{(k)}}{\partial \vec{w}^{(k)}}, \dots, \frac{\partial y_m^{(k)}}{\partial \vec{w}^{(k)}}\right)^T$

Levenberg-Marquardt algorithm (3)

$$g_i = \frac{\partial e}{\partial w_i} = 2\,(y-d)\,\frac{\partial y}{\partial w_i}; \qquad \frac{\partial^2 e}{\partial \mathrm{w_i}\,\partial \mathrm{w_j}} = 2\left[\frac{\partial y}{\partial w_i}\frac{\partial y}{\partial w_j} + \cancel{(y-d)\frac{\partial^2 y}{\partial w_i\,\partial w_j}}\right]$$

- Hence instead of H in $\quad \vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(H + \lambda I\right)^{-1} \nabla E^{(k)}$

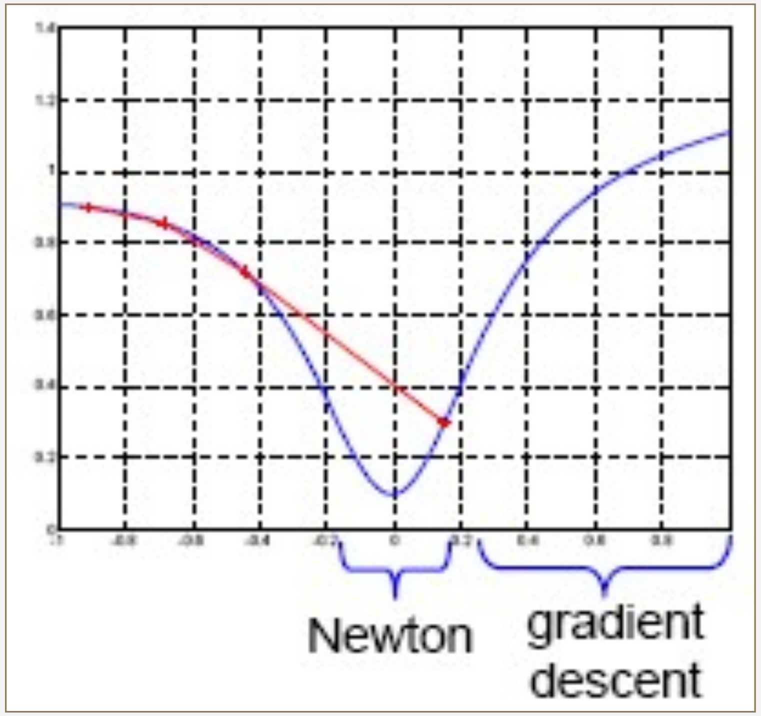
  it is used $\quad \vec{w}^{(k+1)} = \vec{w}^{(k)} - \left({J^{(k)}}^T \cdot J^{(k)} + \lambda I\right)^{-1} \nabla E^{(k)}$

  where

$$J^{(k)} = \begin{pmatrix} \frac{\partial y_1^{(k)}}{\partial w_1^{(k)}} & \frac{\partial y_1^{(k)}}{\partial w_2^{(k)}} & \cdots & \frac{\partial y_1^{(k)}}{\partial w_n^{(k)}} \\ \frac{\partial y_2^{(k)}}{\partial w_1^{(k)}} & \frac{\partial y_2^{(k)}}{\partial w_2^{(k)}} & \cdots & \frac{\partial y_2^{(k)}}{\partial w_n^{(k)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m^{(k)}}{\partial w_1^{(k)}} & \frac{\partial y_m^{(k)}}{\partial w_2^{(k)}} & \cdots & \frac{\partial y_m^{(k)}}{\partial w_n^{(k)}} \end{pmatrix}$$

Levenberg-Marquardt algorithm (4)

- Away from the minimum, in regions of negative curvature, the Gauss-Newton approximation is not very good
- In such regions, a simple steepest-descent step is probably the best plan
- The Levenberg-Marquardt method is a mechanism for varying between steepest-descent and Gauss-Newton steps depending on how good is the approximation J^TJ locally



Levenberg-Marquardt algorithm (5)

- Weight adjustment according to:

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left({J^{(k)}}^T \cdot J^{(k)} + \lambda I\right)^{-1} \nabla E^{(k)}$$

- When λ is small, the step approximates the second order Gauss-Newton method
- When λ is large, steepest-descent steps are taken.

Levenberg-Marquardt algorithm (6)

The training algorithm:

1. Set $\lambda = 0.001$ (say)
2. Compute new weights: $\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(J^{(k)^T} \cdot J^{(k)} + \lambda I\right)^{-1} \nabla E^{(k)}$
3. If the error increases, retract the step (i.e., reset the weights to their previous values) and increase λ ($\times\ 10$ say). Then try an update again by going to 2.
4. Otherwise, accept the step done for weight adjustment (i.e., keep the weights at their new values) and decrease λ ($\times\ 0.1$ say). Then go to 2.

Neural Networks:

Contents:

- **Perceptron and Linear Separability**
- **Multi-layered Neural Networks**
- **Multi-layered Neural Networks: analysis of their properties**

Contents:

Conjugate gradient methods

Useful notions:

- Definition (orthogonality): Given that $\vec{u}\,, \vec{v} \in R^d$, then $\vec{u}$ and $\vec{v}$ are said to be *mutually orthogonal* if $(\vec{u}\,, \vec{v}) = \vec{u}^T \vec{v} = 0$ (where $(\vec{u}\,, \vec{v})$ denotes the scalar product).
- Definition (conjugacy): Given that $\vec{u}\,, \vec{v} \in R^d$, then $\vec{u}$ and $\vec{v}$ are said to be *mutually conjugate with respect to a symmetric positive definite matrix* A if $\vec{u}$ and $A\vec{v}$ are mutually orthogonal, i.e.,
$$\vec{u}^T A\vec{v} = (\vec{u}, A\vec{v}) = 0.$$
- Note: If two vectors are mutually conjugate with respect to the identity matrix, that is $A = 1$, then they are mutually orthogonal.

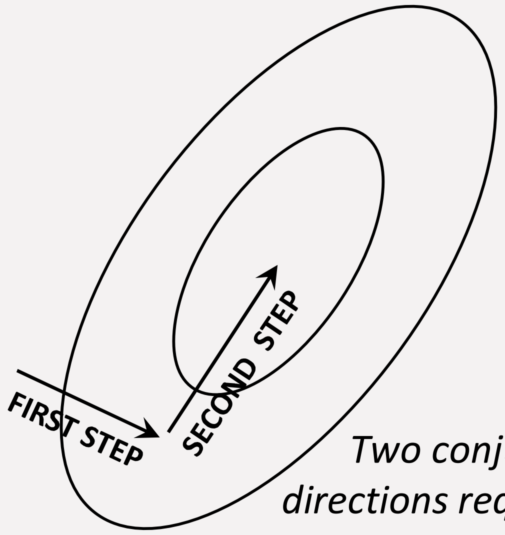
Conjugate gradients on 2D quadratic

Line search is used to determine the optimum step size by searching over different step sizes

- optimum step is taken along each direction and never undone by subsequent steps

  => d linearly independent steps are necessary to find the optimum of a d-dimensional function.

Mutual conjugacy of $\vec{q}_i$ and $\vec{q}_j$ with respect to a symmetric positive definite matrix H if $\vec{q}_i$ and $H\vec{q}_j$ are mutually orthogonal: $\vec{q}_i^T H \vec{q}_j = 0$.



Two conjugate directions required to reach optimality

- If we select any set of (not necessarily orthogonal) vectors $\vec{q}^{(0)}, \ldots, \vec{q}^{(d-1)}$, satisfying the mutual conjugacy condition, then movement along any of these directions does not disturb the projected gradient along other directions.

Conjugate gradient methods

- For quadratic functions, get to the optimum in d steps (instead of a single Newton step), where d is the number of parameters

- Use optimal step-sizes to get the best point along each direction:

  $\vec{w}^{(k+1)} = \vec{w}^{(k)} + \alpha^{(k)}\left(\vec{q}^{(k)}\right)$; $\alpha^{(k)}$ is computed by means of line search

- Conjugate direction: The gradient of the error function at *any* point of an update direction is always orthogonal to previous update directions; start with $\vec{q}^{(0)} = -\nabla E\left(\vec{w}^{(0)}\right)$:

  $$\vec{q}^{(k+1)} = -\nabla E\left(\vec{w}^{(k+1)}\right) + \left(\frac{\left(\vec{q}^{(k)}\right)^T H\left[\nabla E\left(\vec{w}^{(k+1)}\right)\right]}{\left(\vec{q}^{(k)}\right)^T H\,\vec{q}^{(k)}}\right)\vec{q}^{(k)}$$ and increment k by 1

- For non-quadratic functions, approximate the error function with the Taylor expansion and perform $\ll d$ of the above steps. Then repeat.

Efficiently computing projection of Hessian

- The update requires the computation of the projection of the Hessian rather than the inversion of the Hessian:

$$\vec{q}^{(k+1)} = -\nabla E\left(\vec{w}^{(k+1)}\right) + \left(\frac{\left(\vec{q}^{(k)}\right)^T H\left[\nabla E\left(\vec{w}^{(k+1)}\right)\right]}{\left(\vec{q}^{(k)}\right)^T H\,\vec{q}^{(k)}}\right)\vec{q}^{(k)}$$

- Easy to perform numerically, e.g., in the ***Scaled Conjugate Gradient algorithm (SCG)*** using:

$$H\,\vec{q}^{(k)} = E''\left(\vec{w}^{(k)}\right)\vec{q}^{(k)} \approx \frac{\nabla E\left(\vec{w}^{(k)}+\alpha^{(k)}\left(\vec{q}^{(k)}\right)\right)-\nabla E\left(\vec{w}^{(k)}\right)}{\alpha^{(k)}}$$

Neural Networks:

Contents:

- **Perceptron and Linear Separability**
- **Multi-layered Neural Networks**
- **Multi-layered Neural Networks: analysis of their properties**

Contents:

Back-propagation training algorithm: speeding-up the training process (3)

Relaxation methods ~ weight perturbation:

- Discrete approximation to the gradient is made at each iteration by comparing the errors for the initial weights $E(\vec{w})$ and for the altered weights $\vec{w}'$, $E(\vec{w}')$ - a small perturbation β was added to the weight w_i
- Weight adjustment by: $\Delta w_i = -\alpha \dfrac{E(\vec{w}') - E(\vec{w})}{\beta}$
- This adjustment is repeated iteratively, randomly selecting the weight to be updated

Relaxation methods (2)

<u>An alternative providing a faster convergence:</u>

- Perturbation of the output of the i-th neuron o_i by Δo_i
- The difference $E - E'$ in the error function is computed
- If the difference is positive (> 0), the new error E' could be achieved with the output $o_i + \Delta o_i$ for the i-th neuron
- In the case of the sigmoidal transfer function, the desired potential of the neuron i can be determined as: $\sum_{k=1}^{m} w'_k \, x_k = s^{-1}(o_i + \Delta \, o_i)$

(for $y = s(\xi) = \frac{1}{1+\mathrm{e}^{-\xi}}$ the potential is $\xi = s^{-1}(y) = \ln \frac{y}{1-y}$)

Relaxation methods (3)

- If the previous potential was $\sum_{k=1}^{m} w_k\, x_k$, then the new weights are given by:

$$w'_k = w_k \cdot \frac{s^{-1}\,(\,o_i \;+\; \Delta\, o_i\,)}{\sum_{k=1}^{m}\, w_k\, x_k}$$

- Weights are updated in proportion to their size: $\frac{w'_k}{\xi'} = \frac{w_k}{\xi}$

  this can be avoided by means of stochastic factors or node perturbation

  can be alternated with weight perturbation

Neural Networks:

Contents:

Contents:

Back-propagation training algorithm: speeding-up the training process (4)

Other strategies relevant to the training of feed-forward networks:

- Vanishing and exploding gradient problems
- Batch normalization
- Regularization
- Dropout

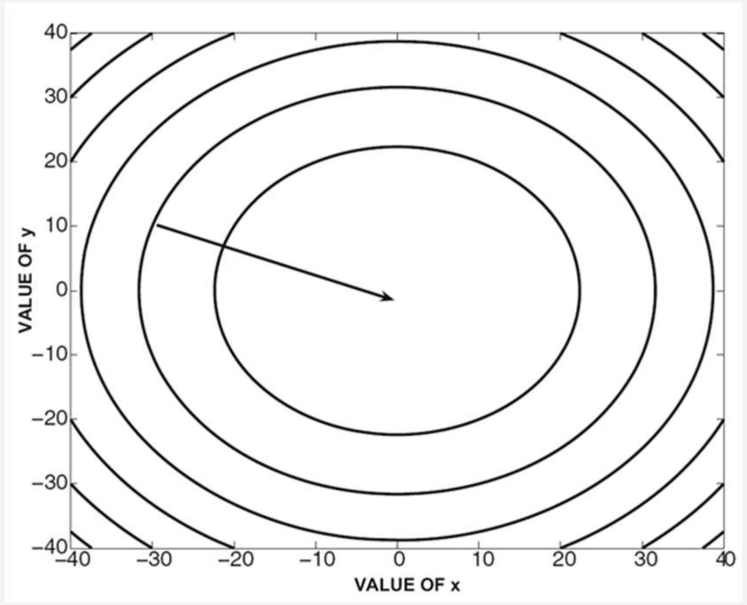
Vanishing and exploding gradient problems (1)

The Effect of Varying Slopes in Gradient Descent:

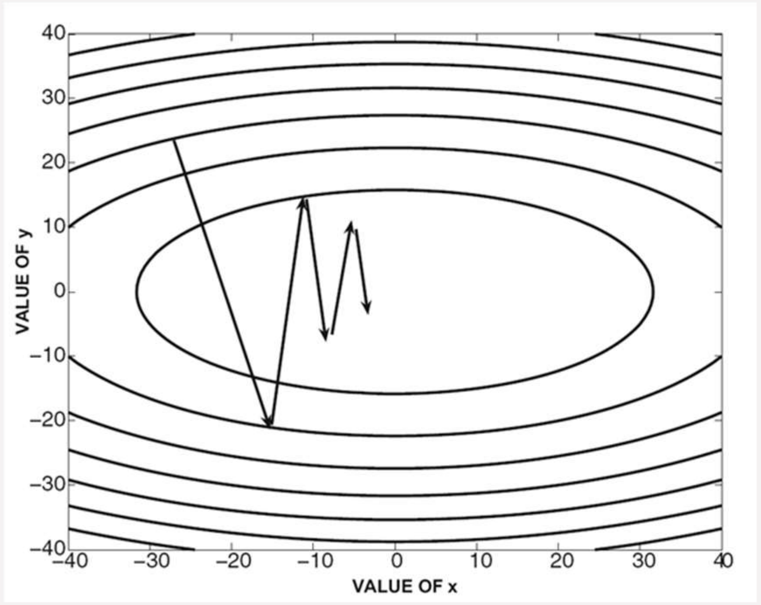
- Neural network learning is a *multivariable* optimization problem
- Different weights have different magnitudes of partial derivatives
- Widely varying magnitudes of partial derivatives affect the learning
- Gradient descent works best when the different weights have derivatives of similar magnitude.
    - *The path of the steepest descent is, in most loss functions, only an instantan-eous direction of the best movement and is not the correct direction of descent in the longer term.*

Vanishing and exploding gradient problems (2)

Example:



The loss function is a circular bowl $L = x^2 + y^2$



The loss function is an elliptical bowl $L = x^2 + 4y^2$

Vanishing and exploding gradient problems (3)

Revisiting feature normalization:

- Loss functions with varying sensitivity to different attributes cause bouncing
  - When features have very different magnitudes, gradient ratios of different weights are likely very different
- Feature normalization helps even out gradient ratios to some extent
  - Exact behavior depends on target variable and loss function

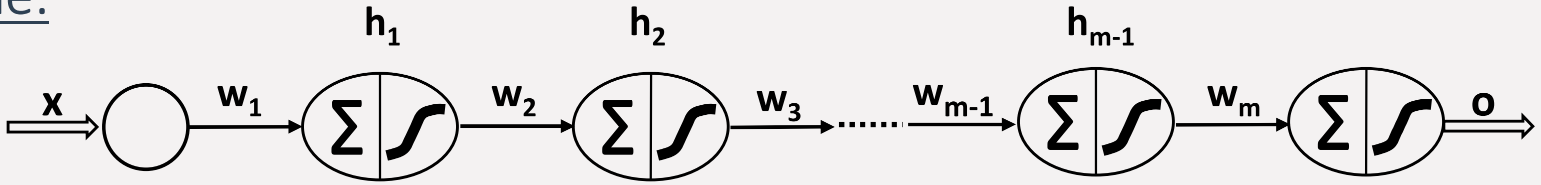
Vanishing and exploding gradient problems (4)

The problem:

- An extreme manifestation of varying sensitivity occurs in deep networks
- The weights/activation derivatives in different layers affect backpropagated gradient in a multiplicative way
  - This effect is magnified with increasing depth
  - The partial derivatives can either increase or decrease with depth

Vanishing and exploding gradient problems (5)

Example:



- A neural network with one node per layer
- Forward propagation multiplicatively depends on each weight and activation function evaluation
- Backpropagated partial derivative get multiplied by weights and activation function derivatives
- Unless the values are exactly one, the partial derivatives will either continuously increase (explode) or decrease (vanish)
- Hard to initialize weights exactly right

Vanishing and exploding gradient problems (6)

Propensity of the transfer function to vanishing gradients:

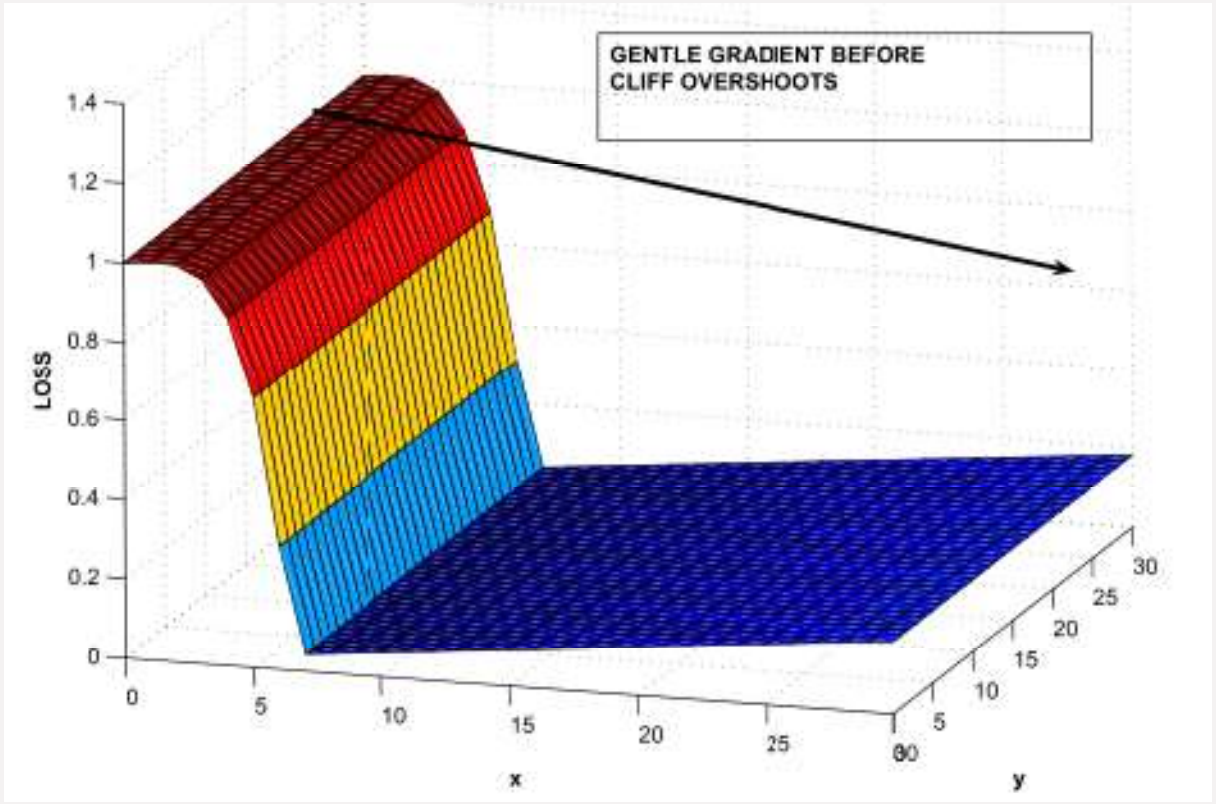
- Partial derivative of the sigmoid with output o is $o(1 - o)$
  - Its maximum value at $o = 0.5$ is equal to 0.25
  - For 10 layers, the transfer function alone will yield less than $0.25^{10} \approx 10^{-6}$
- At the extremes of the output values, the partial derivative is close to 0, which is called *saturation*
- The $\tanh$ transfer function with the partial derivative of $(1 - o^2)$ has the maximum value of 1 at $o = 0$, but saturation will still cause problems

Vanishing and exploding gradient problems (7)

Exploding gradients:

- Initializing the weights to very large values (to compensate for the transfer functions) can cause *exploding gradients*
- Exploding gradients can also occur when weights across different layers are shared (e.g., in recurrent neural networks)
  - The effect of a finite change in the weight is extremely unpredictable across different layers
  - A small *finite* change changes the loss negligibly, but a slightly larger value might change the loss drastically

Vanishing and exploding gradient problems (8)



Cliffs:

- Often occur with the exploding gradient problem

Vanishing and exploding gradient problems (9)

A partial fix to vanishing gradients:

- The ReLU transfer function has a linear activation for nonnegative potential values and otherwise sets its outputs to 0
- The ReLU transfer function has a partial derivative of 1 for nonnegative inputs

x However, it can have a partial derivative of 0 in some cases and never get updated weights

  - Such a neuron is permanently dead!

Vanishing and exploding gradient problems (10)

Leaky ReLU:

- For negative inputs, leaky ReLU can still propagate some gradient backwards
  - At the reduced rate of $\alpha < 1$ times the learning case for non-negative inputs:

$$\Phi(\xi) = \begin{cases} \alpha \cdot \xi & \xi \leq 0 \\ \xi & \text{otherwise} \end{cases}$$

- The value of α is a hyperparameter chosen by the user
- Gains with the leaky ReLU are not guaranteed

Vanishing and exploding gradient problems (11)

Maxout:

- The activation used is $\max\{\vec{w}_1 \cdot \vec{x},\ \vec{w}_2 \cdot \vec{x}\}$ with two coefficient vectors
- One can view the maxout transfer function as a generalization of the ReLU
    - ReLU is obtained by setting one of the coefficient vectors to 0
    - Leaky ReLU can also be simulated by setting the other coefficient vector to $\vec{w}_2 = \alpha\, \vec{w}_1$
- The main disadvantage is that it doubles the number of parameters

Vanishing and exploding gradient problems (12)

Gradient clipping for exploding gradients:

- Try to make the different components of the partial derivatives more even
  - ***Value-based clipping:*** All partial derivatives outside the ranges are set to range boundaries
  - ***Norm-based clipping:*** The entire gradient vector is normalized by the L_2-norm of the entire vector
- One can achieve a better conditioning of gradient values, so that the updates from mini-batch to mini-batch are similar
- Prevents anomalous gradient explosion during training

Vanishing and exploding gradient problems (13)

Other comments on vanishing and exploding gradients:

- The methods discussed above are only partial fixes
- Other options to fix the issue:
  - Initializations improved with pretraining
  - Second-order learning methods that make use of second-order derivatives (~ *curvature* of the loss function).

Batch normalization (1)

Mini-batch stochastic gradient descent:

- One can improve the accuracy of gradient computation by using a batch of instances
  - Instead of holding a vector of activations, we hold a matrix of activations in each layer
  - Matrix-to-matrix multiplications required for forward and backward propagation
  - Increases the memory requirements
- Typical sizes are powers of 2 like $32, 64, 128, 256$

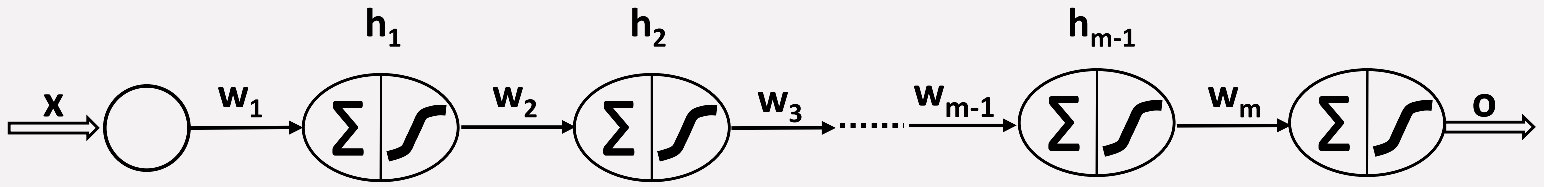
Batch normalization (2)

Why does mini-batching work?

- At early learning stages, the weight vectors are poor
  - The training data is highly redundant in terms of important patterns
  - Small batch sizes provide correct direction of gradient
- Later on, the gradient direction becomes less accurate
  - x But some amount of noise helps avoid overfitting anyway!
- **Performance on out-of-sample data does not deteriorate!**

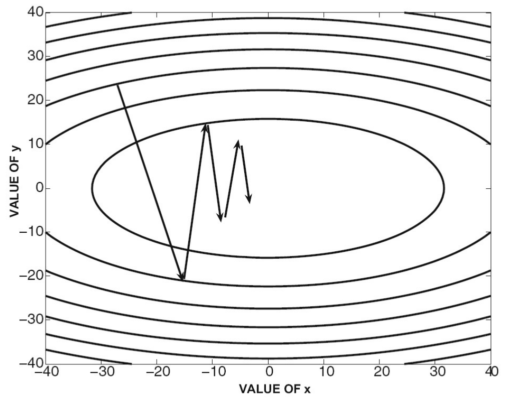
Batch normalization (3)

Revisiting the vanishing and exploding gradient problems:

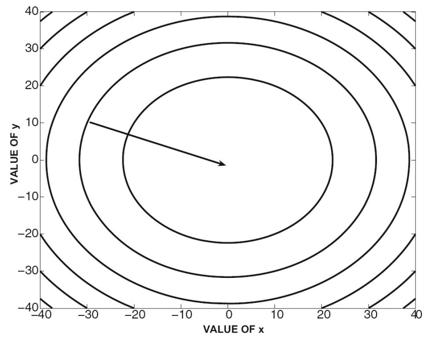


- Neural network with one node per layer
- Forward propagation multiplicatively depends on each weight and transfer function evaluation
- Backpropagated partial derivative gets multiplied by weights and transfer function derivatives
- Unless the values are exactly one, the partial derivatives will either continuously increase (explode) or decrease (vanish)
- Hard to initialize weights exactly right

Batch normalization (4)



The loss function is an elliptical bowl $L = x^2 + 4y^2$



The loss function is a circular bowl $L = x^2 + y^2$

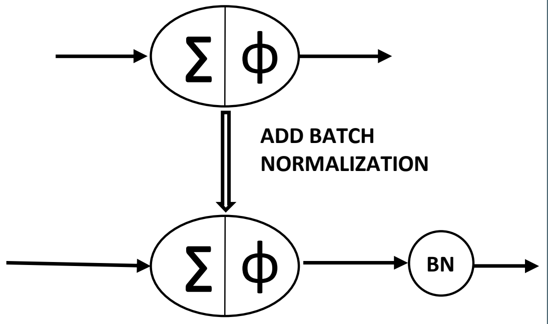
Revisiting the bowl:

- Varying scale of different parameters will cause bouncing
- Varying scale of features causes varying scale of parameters

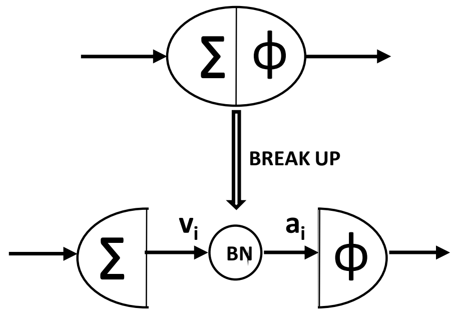
Batch normalization (5)

Input shift:

- One can view the input to each layer as shifting a data set of hidden activations during training
- A shifting input causes problems during training
  - Convergence becomes slower
  - The final result may not generalize well because of unstable inputs
- Batch normalization ensures (somewhat) more stable inputs to each layer



Post-activation normalization



Pre-activation normalization

Batch normalization (6)

Solution: batch normalization

- Add an additional layer that normalizes in a *batch-wise* fashion
- Additional learnable parameters to ensure that an optimal level of nonlinearity is used
- Pre-activation normalization more common than the post-activation one

Batch normalization (7)

<u>Batch normalization node:</u>

- The i-th neuron has two parameters β_i and γ_i that need to be learned
- Normalize its (pre-activation) potential values $a_i^{(r)}$ over a *batch* of m instances; ε is a small constant added for numerical stability
  - batch mean: $\mu_i = \frac{\sum_{r=1}^{m} \xi_i^{(r)}}{m} \quad \forall i$
  - batch variance: $\sigma_i^2 = \frac{\sum_{r=1}^{m} \left(\xi_i^{(r)} - \mu_i\right)^2}{m} + \varepsilon \quad \forall i$
  - normalize batch instances: $\hat{\xi}_i^{(r)} = \frac{\xi_i^{(r)} - \mu_i}{\sigma_i} \quad \forall\, i, r$
  - scale with learnable parameters: $a_i^{(r)} = \gamma_i \cdot \hat{\xi}_i^{(r)} + \beta_i \quad \forall\, i, r$
- Why do we need β_i and γ_i: most potentials will be near zero (near-linear regime)

Batch normalization (8)

Changes to backpropagation:

- We need to backpropagate through the newly added layer of normalization neurons
  - The BN node can be treated like any other node
- We want to optimize the parameters β_i and γ_i
  - The gradients with respect to these parameters have to be computed during backpropagation, too

Batch normalization (9)

Issues in inference:

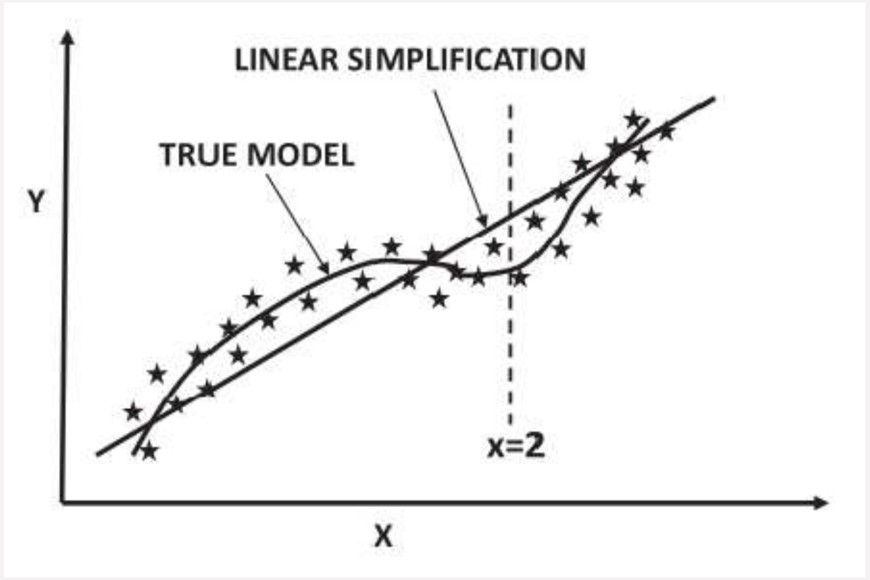
- The transformation parameters μ_i and σ_i depend on the batch
- How should one compute them during testing when a *single* test instance is available?
- The values of μ_i and σ_i are computed up front using the *entire* population (of training data), and then treated as constants during testing time
  - One can also maintain exponentially weighted averages during training
- The normalization is a simple linear transformation during inference

Batch normalization (10)

Batch normalization as a regularizer:

- Batch normalization also acts as a regularizer
- Same data points can cause somewhat different updates depending on which batch it is included in
- One can view this effect as a kind of noise added to the update process
- Regularization can be shown to be equivalent to adding a small amount of noise to the training data.
- This sort of regularization is relatively mild

Penalty-based regularization (1)



Revisiting example – predict y from x:

- **First impression:**

  A polynomial model such as

  $y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$

  is "better" than the linear model

  $y = w_0 + w_1 x$

x However, with less data, using the linear model is better

Penalty-based regularization (2)

Economy in parameters:

- A lower-order model has an economy in parameters
  - A linear model uses two parameters, whereas an order-four model uses five parameters
  - The economy in parameters discourages overfitting
- Choosing a neural network with fewer neurons per layer enforces economy

Penalty-based regularization (3)

Soft economy vs. hard economy:

- Fixing the architecture up front is an inflexible solution
- A softer solution uses a larger model but imposes a (tunable) penalty on the used parameters: $y = \sum_{i=0}^{d} w_i \, x^i$
- Loss function: $L = \sum_{(x,\hat{y})\in D} (\hat{y} - y)^2 + \underbrace{\lambda \cdot \sum_{i=0}^{d} w_i^2}_{L_2\text{–Regularization}}$
- The (tuned) value of λ decides the level of regularization
- Softer approach with a complex model performs better!

Penalty-based regularization (4)

Effect on updates:

- The effect of the learning rate α on the update is to multiply the parameter with $(1 - \alpha\,\lambda) \in (0,1)$:

$$w_i \leftarrow w_i(1 - \alpha\,\lambda) - \alpha\,\frac{\partial\, L}{\partial\, w_i}$$

  - **Interpretation:** *decay-based forgetting!*

- Unless a parameter is important, it will have a small absolute value
  - The model decides what is important!
  - Works better than inflexibly deciding upfront

Penalty-based regularization (5)

L_1-regularization:

- In L_1-regularization, an L_1-penalty is imposed on the loss function:

$$L = \sum_{(x,\hat{y})\in D} (\hat{y} - y)^2 + \lambda \cdot \sum_{i=0}^{d} \|w_i\|_1$$

- The update has a slightly different form:

$$w_i \leftarrow w_i - \alpha\, \lambda s_i - \alpha \frac{\partial\, L}{\partial\, w_i}$$

- The value of s_i is the partial derivative of $|w_i|$ w.r.t. w_i:

$$s_i = \begin{cases} -1 & w_i < 0 \\ +1 & w_i > 0 \end{cases}$$

Penalty-based regularization (6)

L_1- or L_2-Regularization?

- L_1-regularization leads to sparse parameter learning
- Zero values of w_i can be dropped
- Equivalent to dropping edges from a neural network
- L_2-regularization generally provides better performance.

Penalty-based regularization (7)

Connections to Noise Injection:

- *L_2-regularization with parameter λ is equivalent to adding Gaussian noise with variance λ to input*
  - **Intuition:** The negative effect of noise will be minimized with simpler models (smaller parameters)
  - Result is only true for single-layer networks (linear regression)
    - The main value of the result is in providing general intuition
    - Similar results can be shown for denoising autoencoders

Penalty-based regularization (8)

Penalizing Hidden Units:

- One can also penalize hidden units
- Applying L_1-penalty leads to sparse activations
- More common in unsupervised applications for *sparse feature learning*
- Straightforward modification of backpropagation
  - Penalty contributions from hidden units are picked up in the backward phase

Dropout (1)

<u>Feature Co-Adaptation:</u>

- The process of training a neural network often leads to a high level of dependence among features
- Different parts of the network train at different rates:
    - That causes some parts of the network to adapt to others
- This is referred to as *feature co-adaptation*
- Uninformative dependencies are sensitive to nuances of specific training data $\Rightarrow$ *overfitting*

Dropout (2)

One-Way Adaptation:

- Consider a single-hidden layer neural network
  - All weights into and out of half the hidden nodes are fixed to random values
  - Only the other half are updated during backpropagation
- Half the features will adapt to the other half (random features)
- Feature co-adaptation is natural in neural networks where the rates of training vary across their different parts over time
  - This is partially a manifestation of training inefficiency (over and above true synergy)

Dropout (3)

Why is Feature Co-Adaptation Bad?

- We want features working together only when essential for prediction
  - We do not want features adjusting to each other because of the inefficiencies in training
  - The final trained network, namely, does not generalize well to new test data
- We prefer *many* groups of minimally essential features for robust prediction $\Rightarrow$ better redundancies
- We do not want a *few* large and inefficiently created groups of co-adapted features

Dropout (4)

The Basic Dropout Training Procedure:

- For each training pattern do:
  - Sample each neuron in the network in each layer (except the output layer) with probability p
  - Keep only those weights for which both ends are included in the network
  - Perform forward propagation and backpropagation only on the sampled network
- Note that the weights are shared between different sampled networks

Dropout (5)

Basic Dropout Testing Procedures:

- **The first procedure:**
  - Performs repeated sampling (like during training) and averages the results
  - Geometric averaging is then used to assess probabilistic outputs for class i

    ~ an equivalent to averaging log-likelihood

$$p_i^{DROPOUT} = \sqrt[s_n]{p_i^{(1)} \cdot p_i^{(2)} \cdot \ldots \cdot p_i^{(s_n)}} = \left[\prod_{j=1}^{s_n} p_i^{(j)}\right]^{1/s_n}$$

s_n denotes the number of sampled networks

  - Normalization over all c classes: $p_i^{DROPOUT} \leftarrow \dfrac{p_i^{DROPOUT}}{\sum_{i=1}^{c} p_i^{DROPOUT}}$

Dropout (6)

Basic Dropout Testing Procedures:

- **The second procedure** with *weight scaling inference rule:*
  - Is more common
  - Multiplies the weight of each outgoing synapse of a sampled neuron i with its sampling probability p_i
  - Performs a single inference on the full network with down-scaled weights

Dropout (7)

Why Does Dropout Help?

- By dropping the neurons, we force the network to learn without the presence of some inputs (in each layer)
- Each of the sampled subnetworks is trained with a small subset of sampled data instances
- Will resist co-adaptation unless the features are truly synergistic
- Will create many (smaller) groups of self-sufficient predictors
- Many groups of self-sufficient predictors will have a model-averaging effect

Dropout (8)

The Regularization Perspective:

- One can view the dropping of a neuron to be the same process as adding masking noise
  - Noise is added to both input and hidden layers
- Adding noise is equivalent to regularization
- Forces the weights to become more spread out
  - Updates are distributed across the weights based on sampling

Dropout (9)

Practical Aspects of Dropout:

- Typical dropout rate (i.e., probability of exclusion) is somewhere between 20% to 50%
- Use rather a larger network with dropout to enable learning of independent representations
- Dropout is applied to both input layers and hidden layers
- Use large learning rates with decay and large momentum
- Impose a max-norm constraint on the size of network weights
  - The norm of the input weights to a neuron is upper bounded by a constant