Chapter 3:

General Neural Networks

General Neural Networks

Basic graph theoretic notions

A (directed) **graph** is a pair $G = (V, E)$ consisting of a (finite) set V of **nodes** or **vertices** and a (finite) set $E \subseteq V \times V$ of **edges**.

We call an edge $e = (u, v) \in E$ **directed** from node u to node v.

Let $G = (V, E)$ be a (directed) graph and $u \in V$ a node. Then the nodes of the set

$$\text{pred}(u) = \{v \in V \mid (v, u) \in E\}$$

are called the **predecessors** of the node u
and the nodes of the set

$$\text{succ}(u) = \{v \in V \mid (u, v) \in E\}$$

are called the **successors** of the node u.

General Neural Networks

General definition of a neural network

An (artificial) **neural network** is a (directed) graph $G = (U, C)$, whose nodes $u \in U$ are called **neurons** or **units** and whose edges $c \in C$ are called **connections**.

The set U of nodes is partitioned into

- the set U_{in} of **input neurons**,
- the set U_{out} of **output neurons**, and
- the set U_{hidden} of **hidden neurons**.

It is

$$U = U_{\text{in}} \cup U_{\text{out}} \cup U_{\text{hidden}},$$

$$U_{\text{in}} \neq \emptyset, \qquad U_{\text{out}} \neq \emptyset, \qquad U_{\text{hidden}} \cap (U_{\text{in}} \cup U_{\text{out}}) = \emptyset.$$

General Neural Networks

Each connection $(v, u) \in C$ possesses a **weight** w_{uv} and each neuron $u \in U$ possesses three (real-valued) state variables:

- the **network input** net_u,
- the **activation** act_u, and
- the **output** out_u.

Each input neuron $u \in U_{\mathrm{in}}$ also possesses a fourth (real-valued) state variable,

- the **external input** ex_u.

Furthermore, each neuron $u \in U$ possesses three functions:

- the **network input function** $f_{\mathrm{net}}^{(u)} : \mathbb{R}^{2|\mathrm{pred}(u)|+\kappa_1(u)} \to \mathbb{R}$,
- the **activation function** $f_{\mathrm{act}}^{(u)} : \mathbb{R}^{\kappa_2(u)} \to \mathbb{R}$, and
- the **output function** $f_{\mathrm{out}}^{(u)} : \mathbb{R} \to \mathbb{R}$,

which are used to compute the values of the state variables.

General Neural Networks

Types of (artificial) neural networks

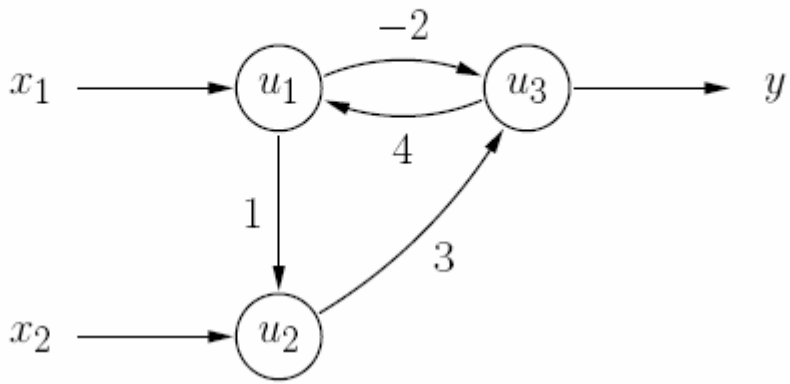
- If the graph of a neural network is **acyclic**, it is called a **feed-forward network**.
- If the graph of a neural network contains **cycles** (backward connections), it is called a **recurrent network**.

Representation of the connection weights by a matrix

$$\begin{array}{c} \begin{array}{cccc} u_1 & u_2 & \dots & u_r \end{array} \\ \begin{pmatrix} w_{u_1u_1} & w_{u_1u_2} & \dots & w_{u_1u_r} \\ w_{u_2u_1} & w_{u_2u_2} & & w_{u_2u_r} \\ \vdots & & & \vdots \\ w_{u_ru_1} & w_{u_ru_2} & \dots & w_{u_ru_r} \end{pmatrix} \end{array} \begin{array}{c} u_1 \\ u_2 \\ \vdots \\ u_r \end{array}$$

General Neural Networks: Example

A simple recurrent neural network

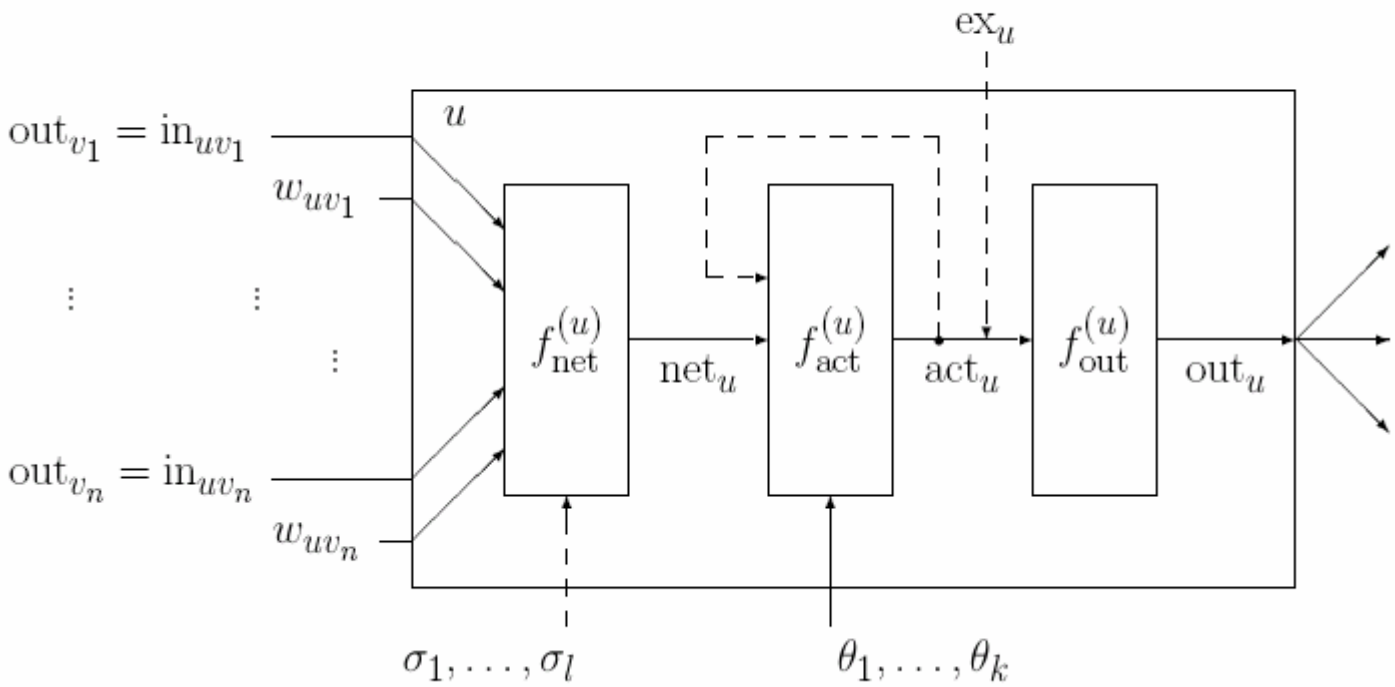


Weight matrix of this network

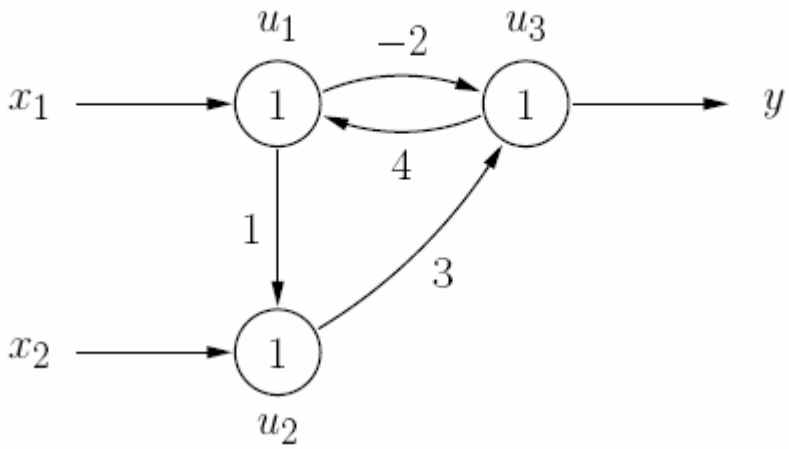
$$\begin{array}{c} \\ \begin{pmatrix} 0 & 0 & 4 \\ 1 & 0 & 0 \\ -2 & 3 & 0 \end{pmatrix} \end{array} \begin{array}{c} u_1 \quad u_2 \quad u_3 \\ \\ \begin{matrix} u_1 \\ u_2 \\ u_3 \end{matrix} \end{array}$$

Structure of a Generalized Neuron

A generalized neuron is a simple numeric processor



General Neural Networks: Example



$$f_{\text{net}}^{(u)}(\vec{w}_u, \vec{\text{in}}_u) = \sum_{v \in \text{pred}(u)} w_{uv} \text{in}_{uv} = \sum_{v \in \text{pred}(u)} w_{uv} \, \text{out}_v$$

$$f_{\text{act}}^{(u)}(\text{net}_u, \theta) = \begin{cases} 1, & \text{if} \quad \text{net}_u \geq \theta, \\ 0, & \text{otherwise.} \end{cases}$$

$$f_{\text{out}}^{(u)}(\text{act}_u) = \text{act}_u$$

General Neural Networks: Example

Updating the activations of the neurons

	u_1	u_2	u_3	
input phase	**1**	**0**	**0**	
work phase	1	0	**0**	$\text{net}_{u_3} = -2$
	0	0	0	$\text{net}_{u_1} = 0$
	0	**0**	0	$\text{net}_{u_2} = 0$
	0	0	**0**	$\text{net}_{u_3} = 0$
	0	0	0	$\text{net}_{u_1} = 0$

- Order in which the neurons are updated:
  $u_3, u_1, u_2, u_3, u_1, u_2, u_3, \ldots$
- A stable state with a unique output is reached.

General Neural Networks: Example

Updating the activations of the neurons

	u_1	u_2	u_3	
input phase	**1**	**0**	**0**	
work phase	1	0	**0**	$\text{net}_{u_3} = -2$
	1	**1**	0	$\text{net}_{u_2} = 1$
	0	1	0	$\text{net}_{u_1} = 0$
	0	1	**1**	$\text{net}_{u_3} = 3$
	0	**0**	1	$\text{net}_{u_2} = 0$
	1	0	1	$\text{net}_{u_1} = 4$
	1	0	**0**	$\text{net}_{u_3} = -2$

- Order in which the neurons are updated:
  $u_3, u_2, u_1, u_3, u_2, u_1, u_3, \ldots$
- No stable state is reached (oscillation of output).

General Neural Networks: Training

Definition of learning tasks for a neural network

A **fixed learning task** L_{fixed} for a neural network with

- n input neurons, i.e. $U_{\text{in}} = \{u_1, \ldots, u_n\}$, and
- m output neurons, i.e. $U_{\text{out}} = \{v_1, \ldots, v_m\}$,

is a set of **training patterns** $l = (\vec{\imath}^{(l)}, \vec{o}^{(l)})$, each consisting of

- an **input vector** $\vec{\imath}^{(l)} = (\text{ex}_{u_1}^{(l)}, \ldots, \text{ex}_{u_n}^{(l)})$ and
- an **output vector** $\vec{o}^{(l)} = (o_{v_1}^{(l)}, \ldots, o_{v_m}^{(l)})$.

A fixed learning task is solved, if for all training patterns $l \in L_{\text{fixed}}$ the neural network computes from the external inputs contained in the input vector $\vec{\imath}^{(l)}$ of a training pattern l the outputs contained in the corresponding output vector $\vec{o}^{(l)}$.

General Neural Networks: Training

Solving a fixed learning task: Error definition

- Measure how well a neural network solves a given fixed learning task.
- Compute differences between desired and actual outputs.
- Do not sum differences directly in order to avoid errors canceling each other.
- Square has favorable properties for deriving the adaptation rules.

$$e = \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{v \in U_{\text{out}}} e_v = \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} e_v^{(l)},$$

$$\text{where} \qquad e_v^{(l)} = \left(o_v^{(l)} - \text{out}_v^{(l)} \right)^2$$

General Neural Networks: Training

Definition of learning tasks for a neural network

A **free learning task** L_{free} for a neural network with

- n input neurons, i.e. $U_{\text{in}} = \{u_1, \ldots, u_n\}$,

is a set of **training patterns** $l = (\vec{\imath}^{(l)})$, each consisting of

- an **input vector** $\vec{\imath}^{(l)} = (\text{ex}_{u_1}^{(l)}, \ldots, \text{ex}_{u_n}^{(l)})$.

Properties:

- There is no desired output for the training patterns.
- Outputs can be chosen freely by the training method.
- Solution idea: **Similar inputs should lead to similar outputs.** (clustering of input vectors)

General Neural Networks: Preprocessing

Normalization of the input vectors

- Compute expected value and standard deviation for each input:

$$\mu_k = \frac{1}{|L|} \sum_{l \in L} \mathrm{ex}_{u_k}^{(l)} \qquad \text{and} \qquad \sigma_k = \sqrt{\frac{1}{|L|} \sum_{l \in L} \left(\mathrm{ex}_{u_k}^{(l)} - \mu_k \right)^2},$$

- Normalize the input vectors to expected value 0 and standard deviation 1:

$$\mathrm{ex}_{u_k}^{(l)(\mathrm{neu})} = \frac{\mathrm{ex}_{u_k}^{(l)(\mathrm{alt})} - \mu_k}{\sigma_k}$$

- Avoids unit and scaling problems.

Chapter 4:

Multilayer Perceptrons

Multilayer Perceptrons

An **r-layered perceptron** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions:

(i) $U_{\text{in}} \cap U_{\text{out}} = \emptyset$,

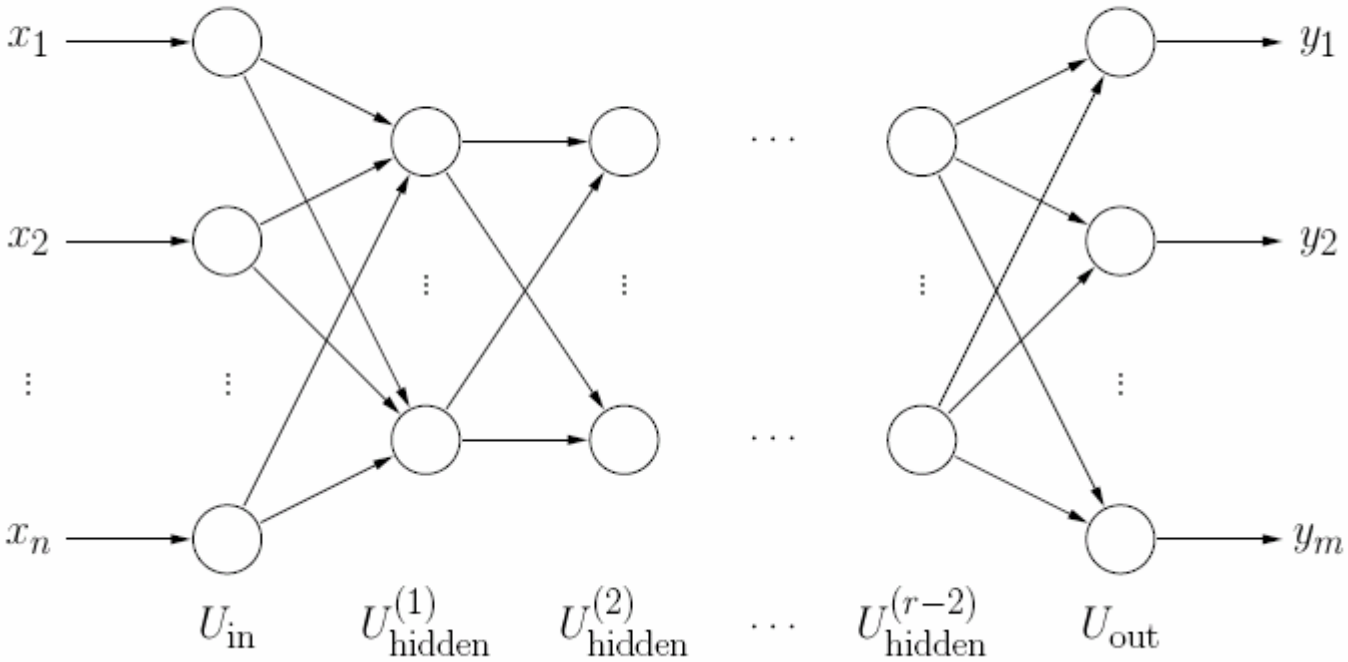
(ii) $U_{\text{hidden}} = U_{\text{hidden}}^{(1)} \cup \cdots \cup U_{\text{hidden}}^{(r-2)}$,
$\forall 1 \leq i < j \leq r-2: \quad U_{\text{hidden}}^{(i)} \cap U_{\text{hidden}}^{(j)} = \emptyset$,

(iii) $C \subseteq \left(U_{\text{in}} \times U_{\text{hidden}}^{(1)}\right) \cup \left(\bigcup_{i=1}^{r-3} U_{\text{hidden}}^{(i)} \times U_{\text{hidden}}^{(i+1)}\right) \cup \left(U_{\text{hidden}}^{(r-2)} \times U_{\text{out}}\right)$
or, if there are no hidden neurons ($r = 2, U_{\text{hidden}} = \emptyset$),
$C \subseteq U_{\text{in}} \times U_{\text{out}}$.

- Feed-forward network with strictly layered structure.

Multilayer Perceptrons

General structure of a multilayer perceptron



Multilayer Perceptrons

- The network input function of each hidden neuron and of each output neuron is the **weighted sum** of its inputs, i.e.

$$\forall u \in U_{\text{hidden}} \cup U_{\text{out}} : \qquad f_{\text{net}}^{(u)}(\vec{w}_u, \vec{\text{in}}_u) = \vec{w}_u \vec{\text{in}}_u = \sum_{v \in \text{pred}(u)} w_{uv} \, \text{out}_v \, .$$

- The activation function of each hidden neuron is a so-called **sigmoid function**, i.e. a monotonously increasing function

$$f : \mathbb{R} \to [0, 1] \quad \text{with} \quad \lim_{x \to -\infty} f(x) = 0 \quad \text{and} \quad \lim_{x \to \infty} f(x) = 1.$$

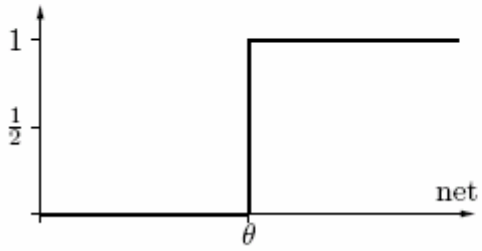
- The activation function of each output neuron is either also a sigmoid function or a **linear function**, i.e.

$$f_{\text{act}}(\text{net}, \theta) = \alpha \, \text{net} - \theta.$$

Sigmoid Activation Functions

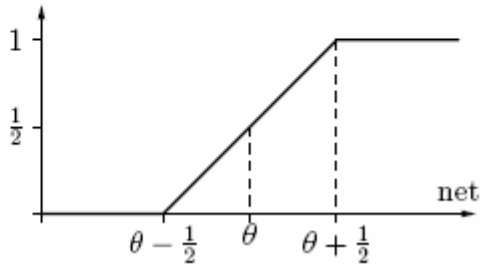
step function:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{if net} \geq \theta, \\ 0, & \text{otherwise.} \end{cases}$$



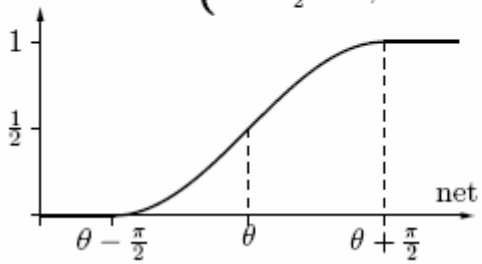
semi-linear function:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{if net} > \theta + \frac{1}{2}, \\ 0, & \text{if net} < \theta - \frac{1}{2}, \\ (\text{net} - \theta) + \frac{1}{2}, & \text{otherwise.} \end{cases}$$



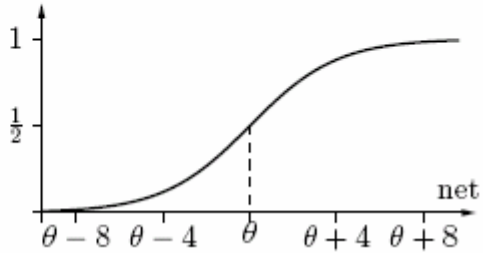
sine until saturation:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{if net} > \theta + \frac{\pi}{2}, \\ 0, & \text{if net} < \theta - \frac{\pi}{2}, \\ \frac{\sin(\text{net} - \theta) + 1}{2}, & \text{otherwise.} \end{cases}$$



logistic function:

$$f_{\text{act}}(\text{net}, \theta) = \frac{1}{1 + e^{-(\text{net} - \theta)}}$$

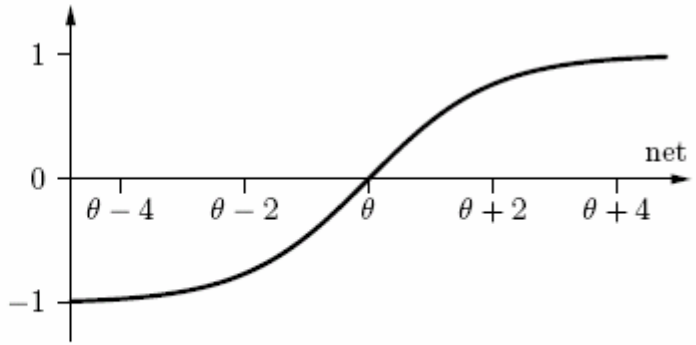


Sigmoid Activation Functions

- All sigmoid functions on the previous slide are **unipolar**, i.e., they range from 0 to 1.
- Sometimes **bipolar** sigmoid functions are used, like the *tangens hyperbolicus*.

tangens hyperbolicus:

$$f_{\text{act}}(\text{net}, \theta) = \tanh(\text{net} - \theta) = \frac{2}{1 + e^{-2(\text{net} - \theta)}} - 1$$



Multilayer Perceptrons: Weight Matrices

Let $U_1 = \{v_1, \ldots, v_m\}$ and $U_2 = \{u_1, \ldots, u_n\}$ be the neurons of two consecutive layers of a multilayer perceptron.

Their connection weights are represented by an $n \times m$ matrix

$$\mathbf{W} = \begin{pmatrix} w_{u_1 v_1} & w_{u_1 v_2} & \ldots & w_{u_1 v_m} \\ w_{u_2 v_1} & w_{u_2 v_2} & \ldots & w_{u_2 v_m} \\ \vdots & \vdots & & \vdots \\ w_{u_n v_1} & w_{u_n v_2} & \ldots & w_{u_n v_m} \end{pmatrix},$$

where $w_{u_i v_j} = 0$ if there is no connection from neuron v_j to neuron u_i.

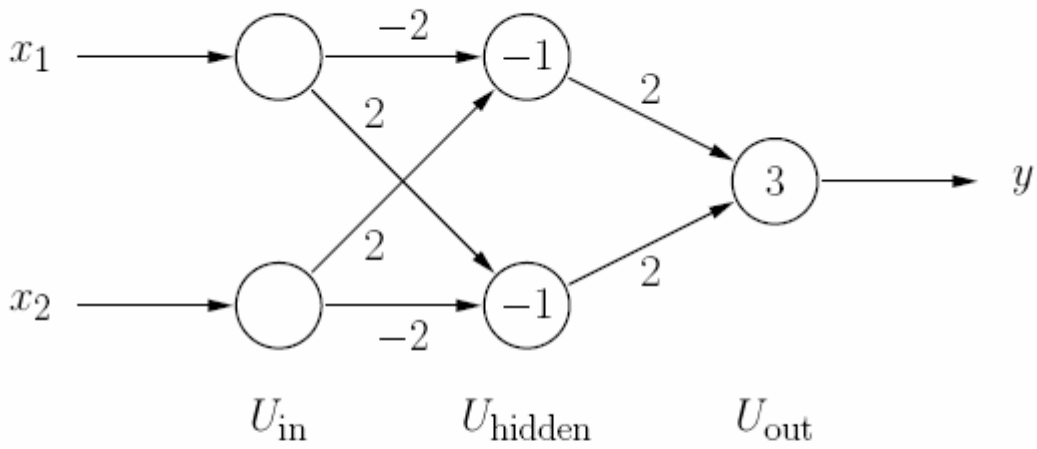
Advantage: The computation of the network input can be written as

$$\vec{\text{net}}_{U_2} = \mathbf{W} \cdot \vec{\text{in}}_{U_2} = \mathbf{W} \cdot \vec{\text{out}}_{U_1}$$

where $\vec{\text{net}}_{U_2} = (\text{net}_{u_1}, \ldots, \text{net}_{u_n})^T$ and $\vec{\text{in}}_{U_2} = \vec{\text{out}}_{U_1} = (\text{out}_{v_1}, \ldots, \text{out}_{v_m})^T$.

Multilayer Perceptrons: Biimplication

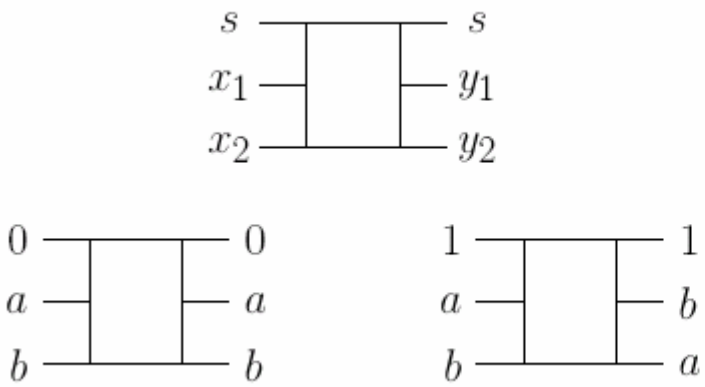
Solving the biimplication problem with a multilayer perceptron.



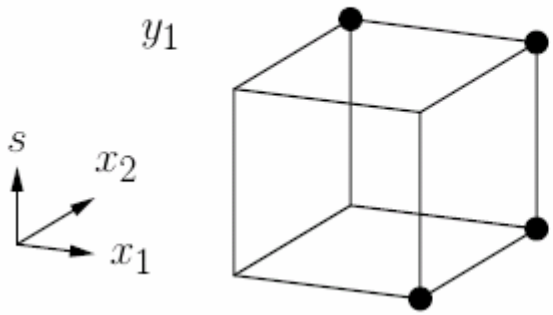
Note the additional input neurons compared to the TLU solution.

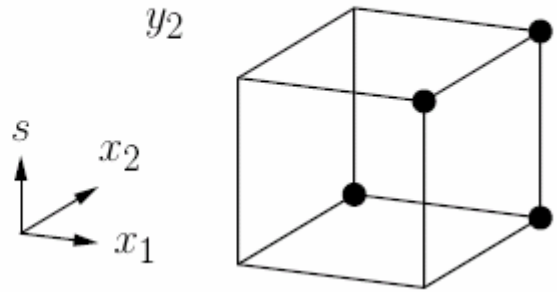
$$\mathbf{W}_1 = \begin{pmatrix} -2 & 2 \\ 2 & -2 \end{pmatrix} \quad \text{and} \quad \mathbf{W}_2 = \begin{pmatrix} 2 & 2 \end{pmatrix}$$

Multilayer Perceptrons: Fredkin Gate

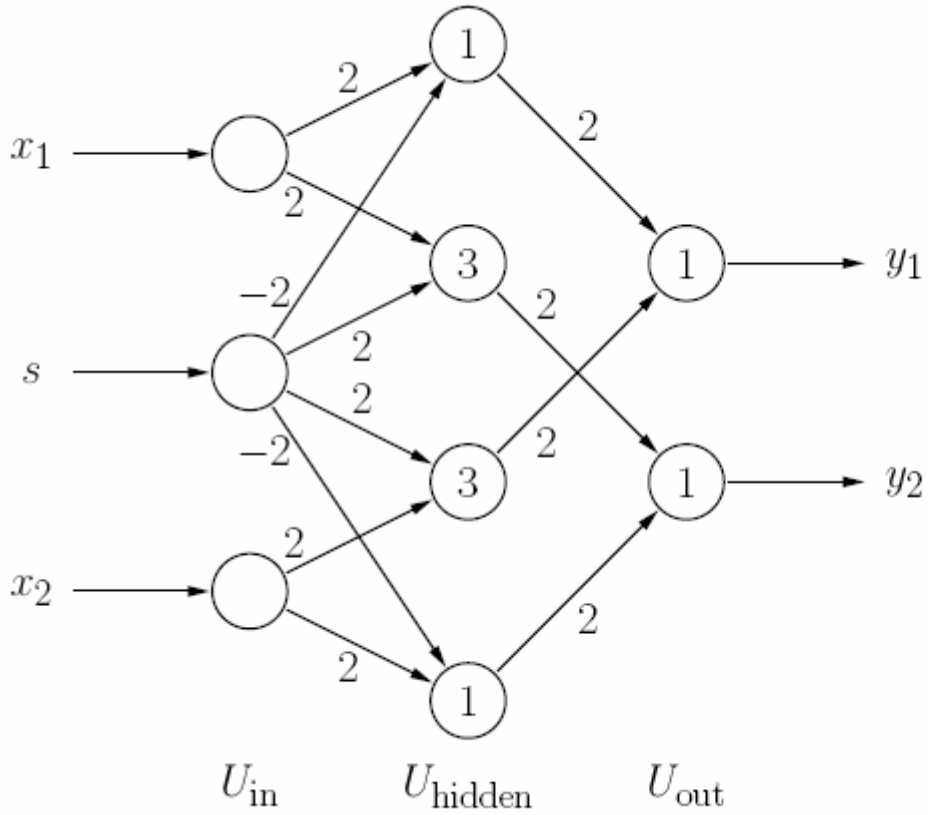


s	0	0	0	0	1	1	1	1
x_1	0	0	1	1	0	0	1	1
x_2	0	1	0	1	0	1	0	1
y_1	0	0	1	1	0	1	0	1
y_2	0	1	0	1	0	0	1	1





Multilayer Perceptrons: Fredkin Gate



$$\mathbf{W}_1 = \begin{pmatrix} 2 & -2 & 0 \\ 2 & 2 & 0 \\ 0 & 2 & 2 \\ 0 & -2 & 2 \end{pmatrix}$$

$$\mathbf{W}_2 = \begin{pmatrix} 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \end{pmatrix}$$

Why Non-linear Activation Functions?

With weight matrices we have for two consecutive layers U_1 and U_2

$$\vec{\text{net}}_{U_2} = \mathbf{W} \cdot \vec{\text{in}}_{U_2} = \mathbf{W} \cdot \vec{\text{out}}_{U_1}.$$

If the activation functions are linear, i.e.,

$$f_{\text{act}}(\text{net}, \theta) = \alpha \, \text{net} - \theta.$$

the activations of the neurons in the layer U_2 can be computed as

$$\vec{\text{act}}_{U_2} = \mathbf{D}_{\text{act}} \cdot \vec{\text{net}}_{U_2} - \vec{\theta},$$

where

- $\vec{\text{act}}_{U_2} = (\text{act}_{u_1}, \ldots, \text{act}_{u_n})^T$ is the activation vector,
- $\mathbf{D}_{\text{act}}$ is an $n \times n$ diagonal matrix of the factors α_{u_i}, $i = 1, \ldots, n$, and
- $\vec{\theta} = (\theta_{u_1}, \ldots, \theta_{u_n})^T$ is a bias vector.

Why Non-linear Activation Functions?

If the output function is also linear, it is analogously

$$\vec{\text{out}}_{U_2} = \mathbf{D}_{\text{out}} \cdot \vec{\text{act}}_{U_2} - \vec{\xi},$$

where

- $\vec{\text{out}}_{U_2} = (\text{out}_{u_1}, \ldots, \text{out}_{u_n})^T$ is the output vector,
- $\mathbf{D}_{\text{out}}$ is again an $n \times n$ diagonal matrix of factors, and
- $\vec{\xi} = (\xi_{u_1}, \ldots, \xi_{u_n})^T$ a bias vector.

Combining these computations we get

$$\vec{\text{out}}_{U_2} = \mathbf{D}_{\text{out}} \cdot \left(\mathbf{D}_{\text{act}} \cdot \left(\mathbf{W} \cdot \vec{\text{out}}_{U_1}\right) - \vec{\theta}\right) - \vec{\xi}$$

and thus

$$\vec{\text{out}}_{U_2} = \mathbf{A}_{12} \cdot \vec{\text{out}}_{U_1} + \vec{b}_{12}$$

with an $n \times m$ matrix $\mathbf{A}_{12}$ and an n-dimensional vector $\vec{b}_{12}$.

Why Non-linear Activation Functions?

Therefore we have

$$\vec{\text{out}}_{U_2} = \mathbf{A}_{12} \cdot \vec{\text{out}}_{U_1} + \vec{b}_{12}$$

and

$$\vec{\text{out}}_{U_3} = \mathbf{A}_{23} \cdot \vec{\text{out}}_{U_2} + \vec{b}_{23}$$

for the computations of two consecutive layers U_2 and U_3.

These two computations can be combined into

$$\vec{\text{out}}_{U_3} = \mathbf{A}_{13} \cdot \vec{\text{out}}_{U_1} + \vec{b}_{13},$$

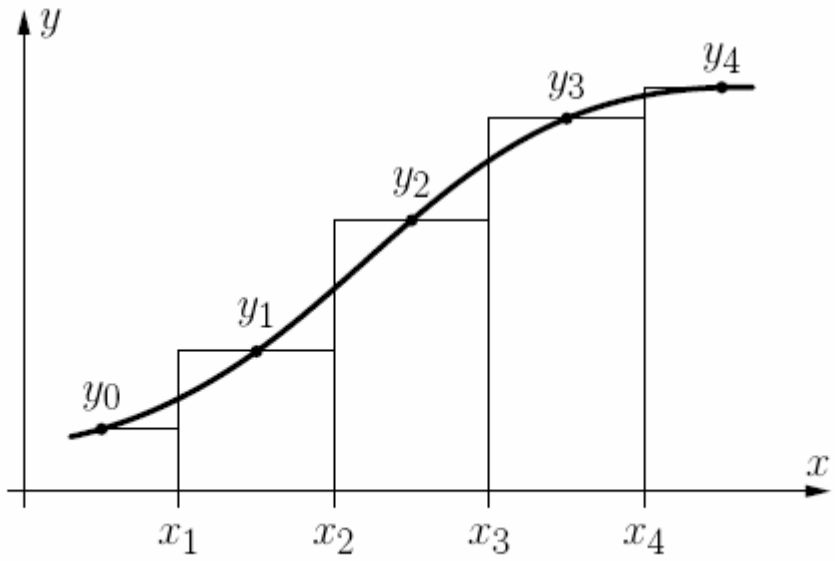
where $\mathbf{A}_{13} = \mathbf{A}_{23} \cdot \mathbf{A}_{12}$ and $\vec{b}_{13} = \mathbf{A}_{23} \cdot \vec{b}_{12} + \vec{b}_{23}$.

Result: With linear activation and output functions any multilayer perceptron can be reduced to a two-layer perceptron.

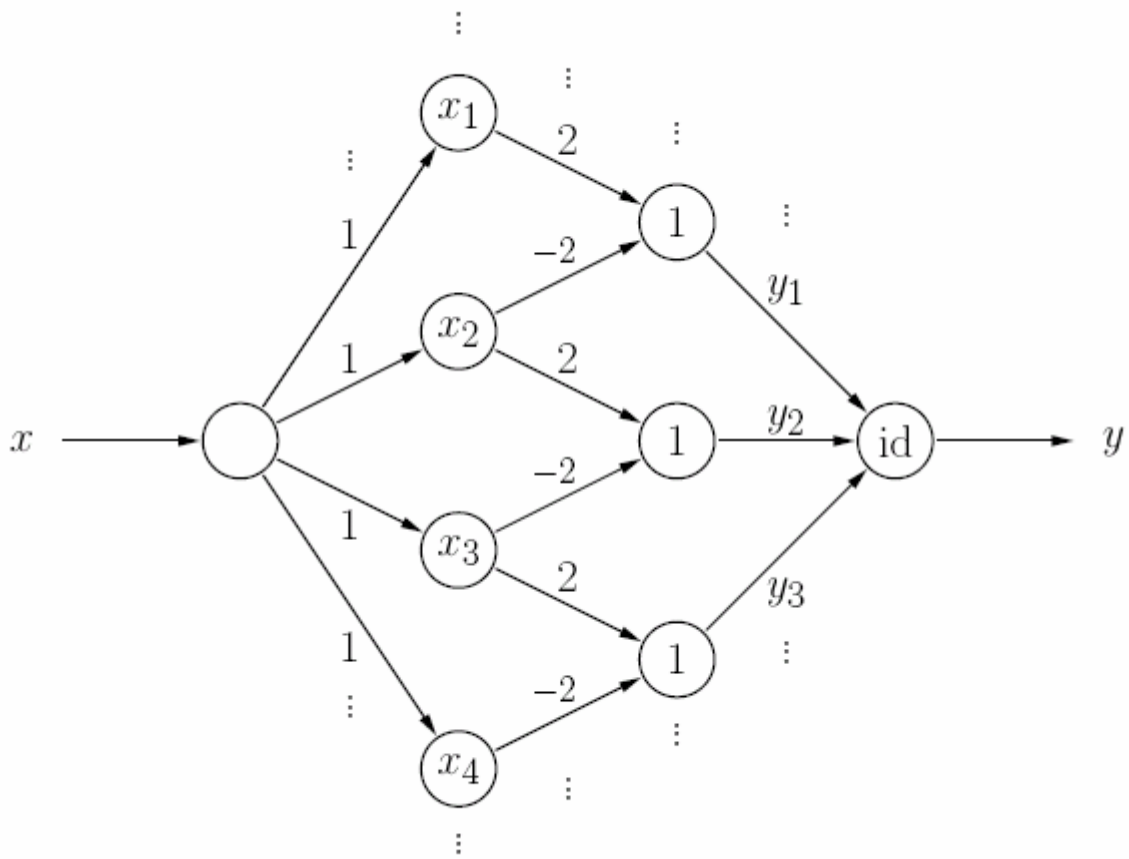
Multilayer Perceptrons: Function Approximation

General idea of function approximation

- Approximate a given function by a step function.
- Construct a neural network that computes the step function.



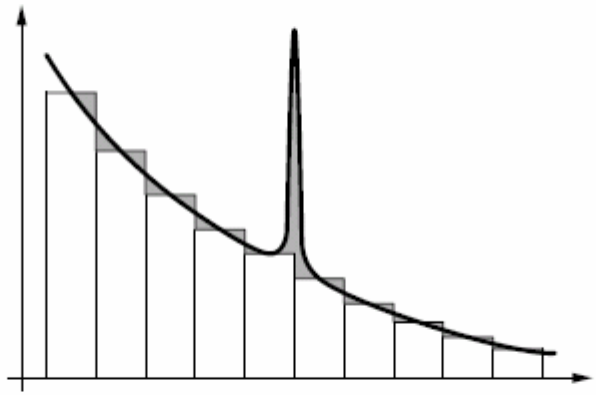
Multilayer Perceptrons: Function Approximation



Multilayer Perceptrons: Function Approximation

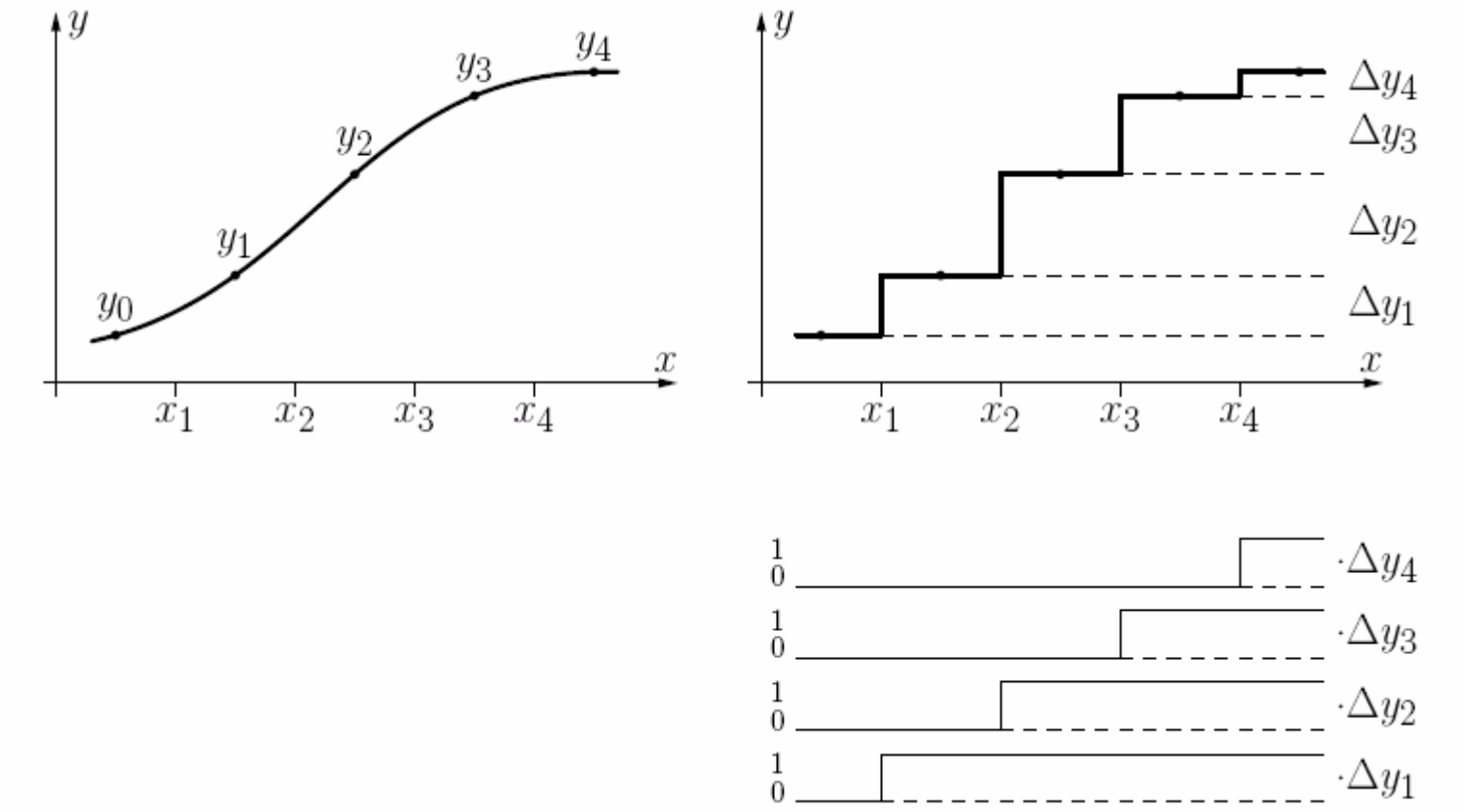
Theorem: Any Riemann-integrable function can be approximated with arbitrary accuracy by a four-layer perceptron.

- But: Error is measured as the **area** between the functions.

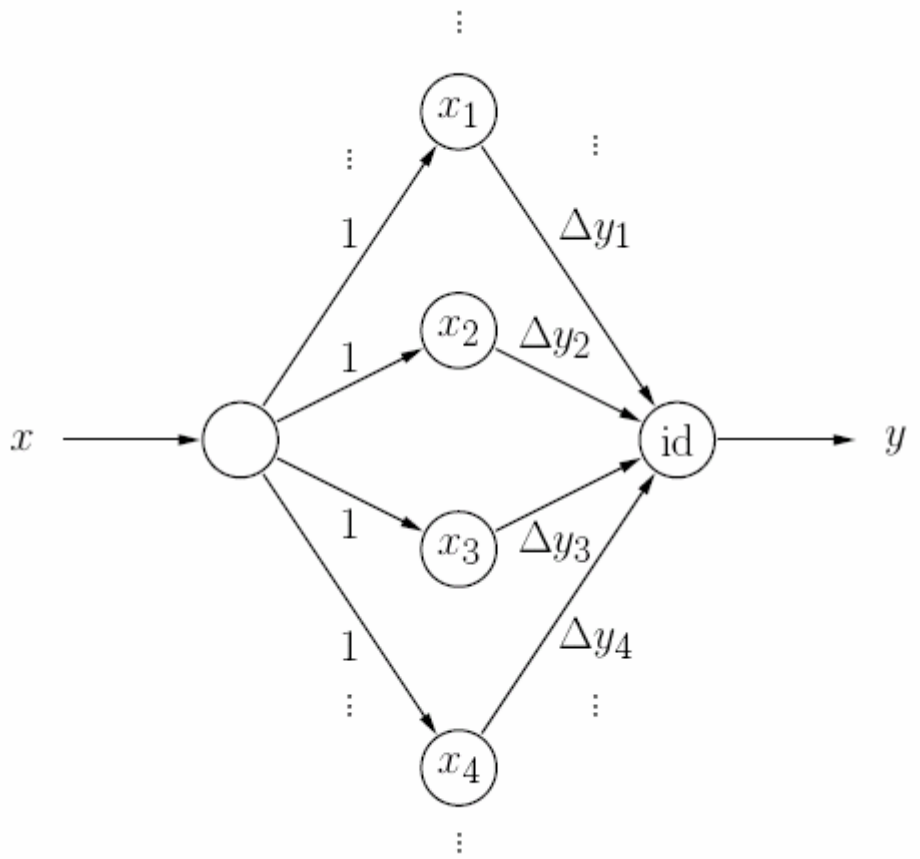


- More sophisticated mathematical examination allows a stronger assertion: With a three-layer perceptron any continuous function can be approximated with arbitrary accuracy (error: maximum function value difference).

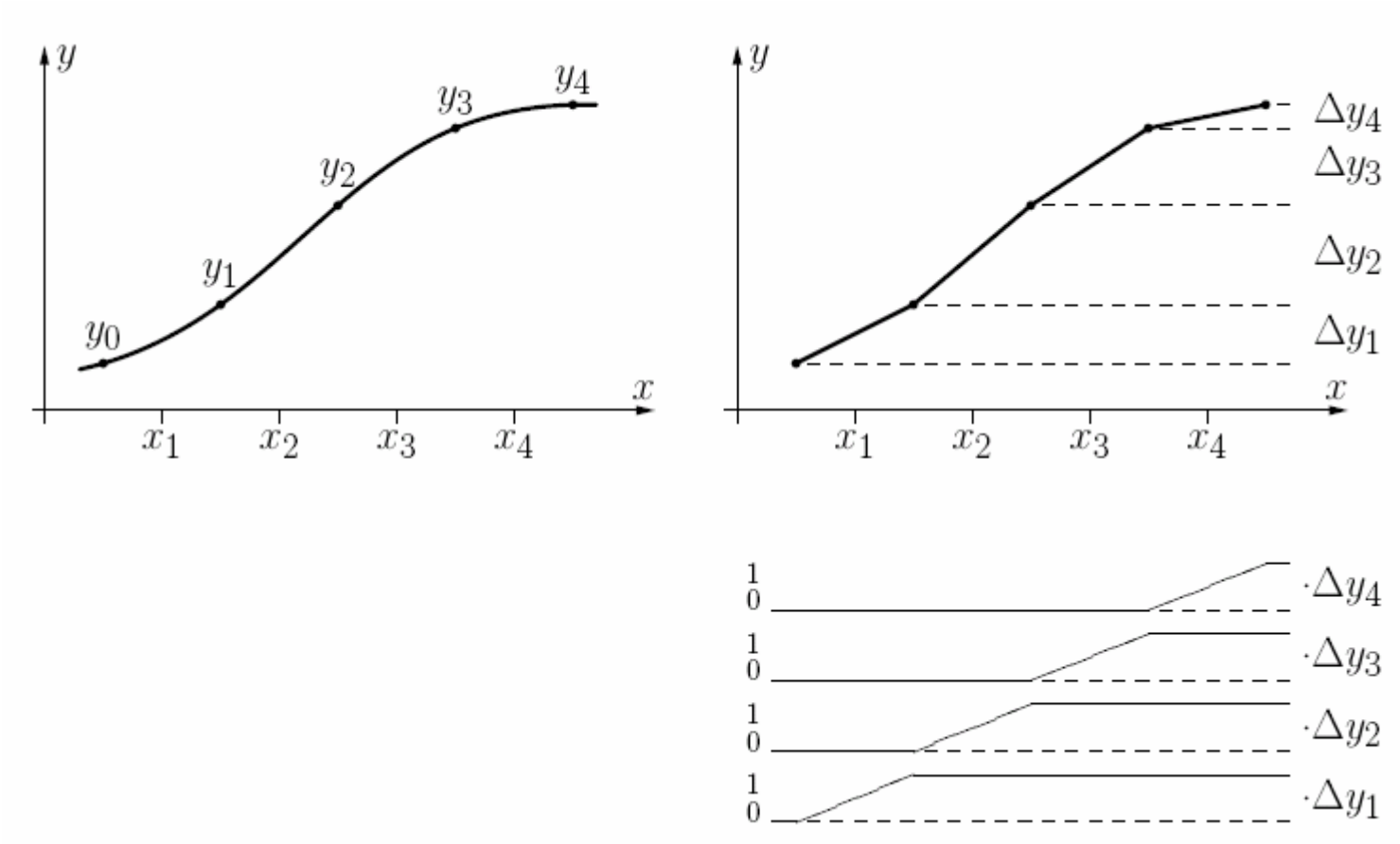
Multilayer Perceptrons: Function Approximation



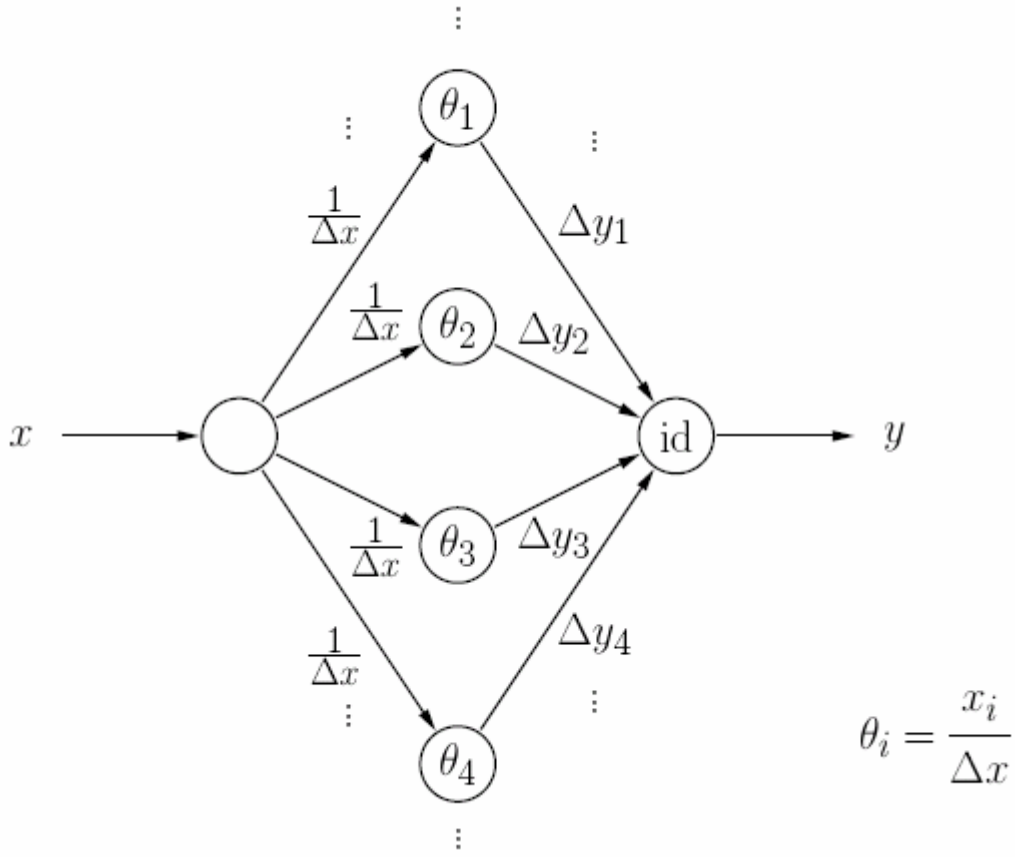
Multilayer Perceptrons: Function Approximation



Multilayer Perceptrons: Function Approximation



Multilayer Perceptrons: Function Approximation



Mathematical Background: Linear Regression

Training neural networks is closely related to regression

Given:
- A dataset $((x_1, y_1), \ldots, (x_n.y_n))$ of n data tuples and
- a hypothesis about the functional relationship, e.g. $y = g(x) = a + bx$.

Approach: Minimize the sum of squared errors, i.e.

$$F(a, b) = \sum_{i=1}^{n} (g(x_i) - y_i)^2 = \sum_{i=1}^{n} (a + bx_i - y_i)^2.$$

Necessary conditions for a minimum:

$$\frac{\partial F}{\partial a} = \sum_{i=1}^{n} 2(a + bx_i - y_i) = 0 \quad \text{and}$$

$$\frac{\partial F}{\partial b} = \sum_{i=1}^{n} 2(a + bx_i - y_i)x_i = 0$$

Mathematical Background: Linear Regression

Result of necessary conditions: System of so-called **normal equations**, i.e.

$$na + \left(\sum_{i=1}^{n} x_i\right) b = \sum_{i=1}^{n} y_i,$$

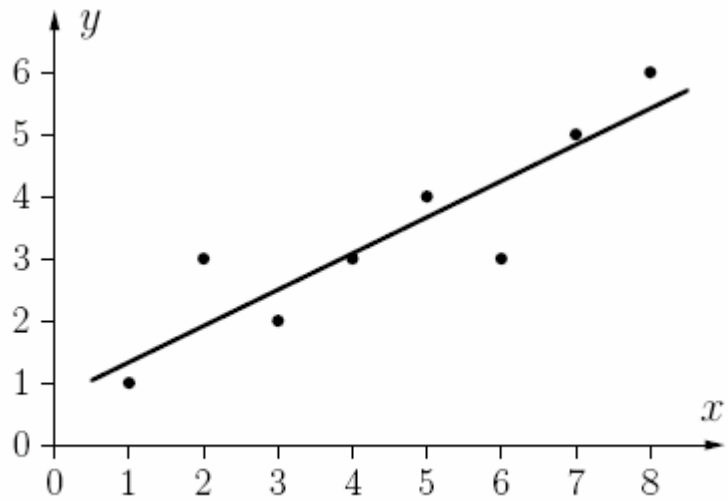
$$\left(\sum_{i=1}^{n} x_i\right) a + \left(\sum_{i=1}^{n} x_i^2\right) b = \sum_{i=1}^{n} x_i y_i.$$

- Two linear equations for two unknowns a and b.
- System can be solved with standard methods from linear algebra.
- Solution is unique unless all x-values are identical.
- The resulting line is called a **regression line**.

Linear Regression: Example

x	1	2	3	4	5	6	7	8
y	1	3	2	3	4	3	5	6

$$y = \frac{3}{4} + \frac{7}{12}x.$$



Mathematical Background: Polynomial Regression

Generalization to polynomials

$$y = p(x) = a_0 + a_1 x + \ldots + a_m x^m$$

Approach: Minimize the sum of squared errors, i.e.

$$F(a_0, a_1, \ldots, a_m) = \sum_{i=1}^{n} (p(x_i) - y_i)^2 = \sum_{i=1}^{n} (a_0 + a_1 x_i + \ldots + a_m x_i^m - y_i)^2$$

Necessary conditions for a minimum: All partial derivatives vanish, i.e.

$$\frac{\partial F}{\partial a_0} = 0, \quad \frac{\partial F}{\partial a_1} = 0, \quad \ldots \quad , \frac{\partial F}{\partial a_m} = 0.$$

Mathematical Background: Polynomial Regression

System of normal equations for polynomials

$$
\begin{aligned}
n a_0 \;+\; \left(\sum_{i=1}^n x_i\right) a_1 \;+\; \dots \;+\; \left(\sum_{i=1}^n x_i^m\right) a_m &= \sum_{i=1}^n y_i \\
\left(\sum_{i=1}^n x_i\right) a_0 \;+\; \left(\sum_{i=1}^n x_i^2\right) a_1 \;+\; \dots \;+\; \left(\sum_{i=1}^n x_i^{m+1}\right) a_m &= \sum_{i=1}^n x_i y_i \\
&\vdots \\
\left(\sum_{i=1}^n x_i^m\right) a_0 \;+\; \left(\sum_{i=1}^n x_i^{m+1}\right) a_1 \;+\; \dots \;+\; \left(\sum_{i=1}^n x_i^{2m}\right) a_m &= \sum_{i=1}^n x_i^m y_i,
\end{aligned}
$$

- $m+1$ linear equations for $m+1$ unknowns $a_0, \dots, a_m$.
- System can be solved with standard methods from linear algebra.
- Solution is unique unless all x-values are identical.

Mathematical Background: Multilinear Regression

Generalization to more than one argument

$$z = f(x, y) = a + bx + cy$$

Approach: Minimize the sum of squared errors, i.e.

$$F(a, b, c) = \sum_{i=1}^{n} (f(x_i, y_i) - z_i)^2 = \sum_{i=1}^{n} (a + bx_i + cy_i - z_i)^2$$

Necessary conditions for a minimum: All partial derivatives vanish, i.e.

$$\begin{aligned}
\frac{\partial F}{\partial a} &= \sum_{i=1}^{n} 2(a + bx_i + cy_i - z_i) &&= 0, \\
\frac{\partial F}{\partial b} &= \sum_{i=1}^{n} 2(a + bx_i + cy_i - z_i)x_i &&= 0, \\
\frac{\partial F}{\partial c} &= \sum_{i=1}^{n} 2(a + bx_i + cy_i - z_i)y_i &&= 0.
\end{aligned}$$

Mathematical Background: Multilinear Regression

System of normal equations for several arguments

$$
\begin{aligned}
na &+ \left(\sum_{i=1}^{n} x_i\right) b + \left(\sum_{i=1}^{n} y_i\right) c = \sum_{i=1}^{n} z_i \\
\left(\sum_{i=1}^{n} x_i\right) a &+ \left(\sum_{i=1}^{n} x_i^2\right) b + \left(\sum_{i=1}^{n} x_i y_i\right) c = \sum_{i=1}^{n} z_i x_i \\
\left(\sum_{i=1}^{n} y_i\right) a &+ \left(\sum_{i=1}^{n} x_i y_i\right) b + \left(\sum_{i=1}^{n} y_i^2\right) c = \sum_{i=1}^{n} z_i y_i
\end{aligned}
$$

- 3 linear equations for 3 unknowns a, b, and c.
- System can be solved with standard methods from linear algebra.
- Solution is unique unless all x- or all y-values are identical.

Mathematical Background: Logistic Regression

Generalization to non-polynomial functions

$$y = ax^b$$

Idea: Find transformation to linear/polynomial case.

$$\ln y = \ln a + b \cdot \ln x.$$

Special case: **logistic function**

$$y = \frac{Y}{1 + e^{a+bx}} \quad \Leftrightarrow \quad \frac{1}{y} = \frac{1 + e^{a+bx}}{Y} \quad \Leftrightarrow \quad \frac{Y - y}{y} = e^{a+bx}.$$

Result: Apply so-called **Logit-Transformation**

$$\ln\left(\frac{Y - y}{y}\right) = a + bx.$$

Logistic Regression: Example

x	1	2	3	4	5
y	0.4	1.0	3.0	5.0	5.6

Transform the data with

$$z = \ln\left(\frac{Y - y}{y}\right), \qquad Y = 6.$$

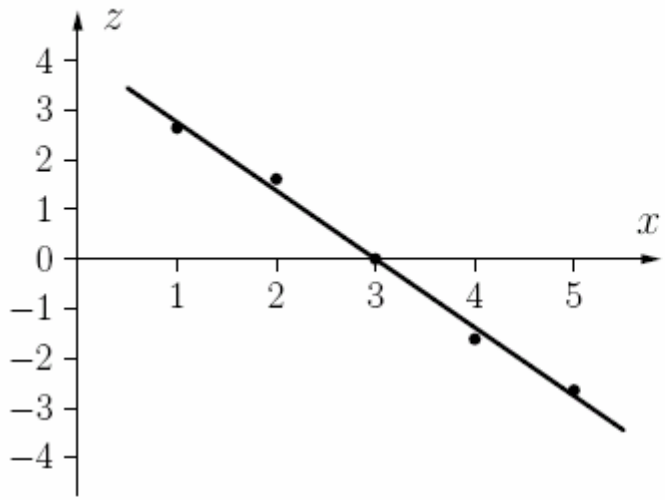
The transformed data points are

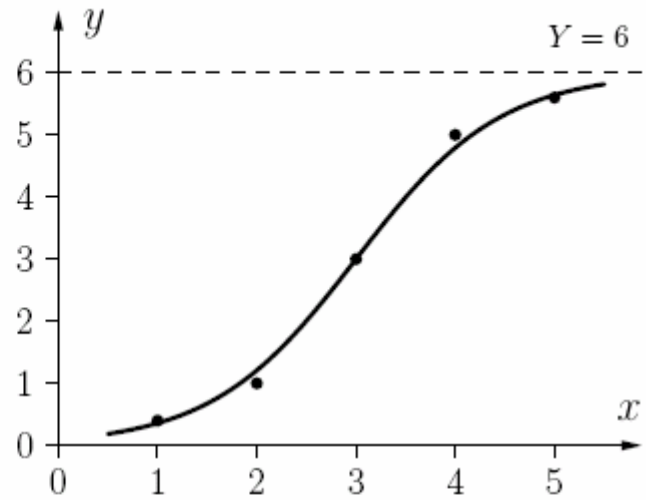
x	1	2	3	4	5
z	2.64	1.61	0.00	-1.61	-2.64

The resulting regression line is

$$z \approx -1.3775x + 4.133.$$

Logistic Regression: Example





The logistic regression function can be computed by a single neuron with

- network input function $f_{\text{net}}(x) \equiv wx$ with $w \approx -1.3775$,
- activation function $f_{\text{act}}(\text{net}, \theta) \equiv (1 + e^{-(\text{net} - \theta)})^{-1}$ with $\theta \approx 4.133$ and
- output function $f_{\text{out}}(\text{act}) \equiv 6\,\text{act}$.

Training Multilayer Perceptrons: Gradient Descent

- Problem of logistic regression: Works only for two-layer perceptrons.
- More general approach: **gradient descent**.
- Necessary condition: **differentiable activation and output functions**.

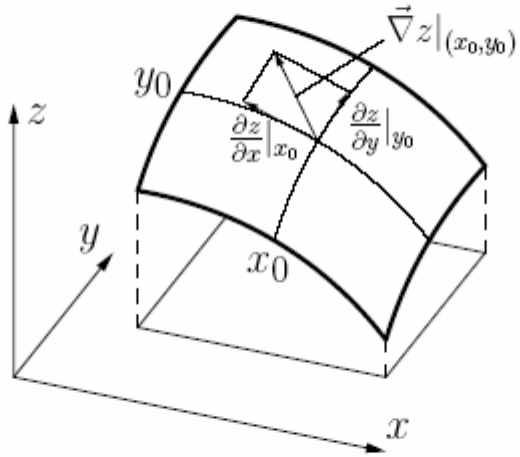


Illustration of the gradient of a real-valued function $z = f(x, y)$ at a point (x_0, y_0).
It is $\vec{\nabla} z|_{(x_0,y_0)} = \left(\frac{\partial z}{\partial x}|_{x_0}, \frac{\partial z}{\partial y}|_{y_0}\right)$.

Gradient Descent: Formal Approach

Idea of gradient descent: Approach minimum of error function in small steps.

Error function:

$$e = \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{v \in U_{\text{out}}} e_v = \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} e_v^{(l)},$$

Form gradient to determine direction of step:

$$\vec{\nabla}_{\vec{w}_u} e = \frac{\partial e}{\partial \vec{w}_u} = \left(-\frac{\partial e}{\partial \theta_u}, \frac{\partial e}{\partial w_{up_1}}, \ldots, \frac{\partial e}{\partial w_{up_n}} \right).$$

Exploit sum over training patterns:

$$\vec{\nabla}_{\vec{w}_u} e = \frac{\partial e}{\partial \vec{w}_u} = \frac{\partial}{\partial \vec{w}_u} \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{l \in L_{\text{fixed}}} \frac{\partial e^{(l)}}{\partial \vec{w}_u}.$$

Gradient Descent: Formal Approach

Single pattern error depends on weights only through the network input:

$$\vec{\nabla}_{\vec{w}_u} e^{(l)} = \frac{\partial e^{(l)}}{\partial \vec{w}_u} = \frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} \frac{\partial \operatorname{net}_u^{(l)}}{\partial \vec{w}_u}.$$

Since $\operatorname{net}_u^{(l)} = \vec{w}_u \vec{\operatorname{in}}_u^{(l)}$ we have for the second factor

$$\frac{\partial \operatorname{net}_u^{(l)}}{\partial \vec{w}_u} = \vec{\operatorname{in}}_u^{(l)}.$$

For the first factor we consider the error $e^{(l)}$ for the training pattern $l = (\vec{\imath}^{(l)}, \vec{o}^{(l)})$:

$$e^{(l)} = \sum_{v \in U_{\text{out}}} e_u^{(l)} = \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \operatorname{out}_v^{(l)} \right)^2,$$

i.e. the sum of the errors over all output neurons.

Gradient Descent: Formal Approach

Therefore we have

$$\frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} = \frac{\partial \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right)^2}{\partial \operatorname{net}_u^{(l)}} = \sum_{v \in U_{\text{out}}} \frac{\partial \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right)^2}{\partial \operatorname{net}_u^{(l)}}.$$

Since only the actual output $\operatorname{out}_v^{(l)}$ of an output neuron v depends on the network input $\operatorname{net}_u^{(l)}$ of the neuron u we are considering, it is

$$\frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} = -2 \underbrace{\sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right) \frac{\partial \operatorname{out}_v^{(l)}}{\partial \operatorname{net}_u^{(l)}}}_{\delta_u^{(l)}},$$

which also introduces the abbreviation $\delta_u^{(l)}$ for the important sum appearing here.

Gradient Descent: Formal Approach

Distinguish two cases:

- The neuron u is an **output neuron**.
- The neuron u is a **hidden neuron**.

In the first case we have

$$\forall u \in U_{\text{out}}: \qquad \delta_u^{(l)} = \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}}$$

Therefore we have for the gradient

$$\forall u \in U_{\text{out}}: \qquad \vec{\nabla}_{\vec{w}_u} e_u^{(l)} = \frac{\partial e_u^{(l)}}{\partial \vec{w}_u} = -2\left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \vec{\text{in}}_u^{(l)}$$

and thus for the weight change

$$\forall u \in U_{\text{out}}: \qquad \Delta\vec{w}_u^{(l)} = -\frac{\eta}{2}\vec{\nabla}_{\vec{w}_u} e_u^{(l)} = \eta\left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \vec{\text{in}}_u^{(l)}.$$

Gradient Descent: Formal Approach

Exact formulae depend on choice of activation and output function, since it is

$$\mathrm{out}_u^{(l)} = f_{\mathrm{out}}(\mathrm{act}_u^{(l)}) = f_{\mathrm{out}}(f_{\mathrm{act}}(\mathrm{net}_u^{(l)})).$$

Consider special case with

- output function is the identity,
- activation function is logistic, i.e. $f_{\mathrm{act}}(x) = \frac{1}{1+e^{-x}}$.

The first assumption yields

$$\frac{\partial \mathrm{out}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}} = \frac{\partial \mathrm{act}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}} = f'_{\mathrm{act}}(\mathrm{net}_u^{(l)}).$$

Gradient Descent: Formal Approach

For a logistic activation function we have

$$\begin{aligned} f'_{\text{act}}(x) &= \frac{\mathrm{d}}{\mathrm{d}x}(1+e^{-x})^{-1} = -(1+e^{-x})^{-2}(-e^{-x}) \\ &= \frac{1+e^{-x}-1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) \\ &= f_{\text{act}}(x)\cdot(1-f_{\text{act}}(x)), \end{aligned}$$

and therefore

$$f'_{\text{act}}(\,\text{net}_u^{(l)}) = f_{\text{act}}(\,\text{net}_u^{(l)})\cdot\left(1-f_{\text{act}}(\,\text{net}_u^{(l)})\right) = \text{out}_u^{(l)}\left(1-\text{out}_u^{(l)}\right).$$

The resulting weight change is therefore

$$\Delta\vec{w}_u^{(l)} = \eta\left(o_u^{(l)}-\text{out}_u^{(l)}\right)\text{out}_u^{(l)}\left(1-\text{out}_u^{(l)}\right)\vec{\text{in}}_u^{(l)},$$

which makes the computations very simple.

Error Backpropagation

Consider now: The neuron u is a **hidden neuron**, i.e. $u \in U_k$, $0 < k < r-1$.

The output $\text{out}_v^{(l)}$ of an output neuron v depends on the network input $\text{net}_u^{(l)}$ only indirectly through its successor neurons $\text{succ}(u) = \{s \in U \mid (u,s) \in C\} = \{s_1, \ldots, s_m\} \subseteq U_{k+1}$, namely through their network inputs $\text{net}_s^{(l)}$.

We apply the chain rule to obtain

$$\delta_u^{(l)} = \sum_{v \in U_{\text{out}}} \sum_{s \in \text{succ}(u)} (o_v^{(l)} - \text{out}_v^{(l)}) \frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_s^{(l)}} \frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_u^{(l)}}.$$

Exchanging the sums yields

$$\delta_u^{(l)} = \sum_{s \in \text{succ}(u)} \left(\sum_{v \in U_{\text{out}}} (o_v^{(l)} - \text{out}_v^{(l)}) \frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_s^{(l)}} \right) \frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_u^{(l)}} = \sum_{s \in \text{succ}(u)} \delta_s^{(l)} \frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_u^{(l)}}.$$

Error Backpropagation

Consider the network input

$$\mathrm{net}_s^{(l)} = \vec{w}_s \vec{\mathrm{in}}_s^{(l)} = \left(\sum_{p \in \mathrm{pred}(s)} w_{sp} \, \mathrm{out}_p^{(l)} \right) - \theta_s,$$

where one element of $\vec{\mathrm{in}}_s^{(l)}$ is the output $\mathrm{out}_u^{(l)}$ of the neuron u. Therefore it is

$$\frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}} = \left(\sum_{p \in \mathrm{pred}(s)} w_{sp} \frac{\partial \mathrm{out}_p^{(l)}}{\partial \mathrm{net}_u^{(l)}} \right) - \frac{\partial \theta_s}{\partial \mathrm{net}_u^{(l)}} = w_{su} \frac{\partial \mathrm{out}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}},$$

The result is the recursive equation (error backpropagation)

$$\delta_u^{(l)} = \left(\sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} w_{su} \right) \frac{\partial \mathrm{out}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}}.$$

Error Backpropagation

The resulting formula for the weight change is

$$\Delta\vec{w}_u^{(l)} = -\frac{\eta}{2}\vec{\nabla}_{\vec{w}_u} e^{(l)} = \eta\,\delta_u^{(l)}\,\vec{\mathrm{in}}_u^{(l)} = \eta\left(\sum_{s\in\mathrm{succ}(u)} \delta_s^{(l)} w_{su}\right)\frac{\partial\,\mathrm{out}_u^{(l)}}{\partial\,\mathrm{net}_u^{(l)}}\,\vec{\mathrm{in}}_u^{(l)}.$$

Consider again the special case with

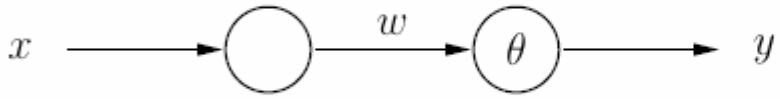
- output function is the identity,
- activation function is logistic.

The resulting formula for the weight change is then

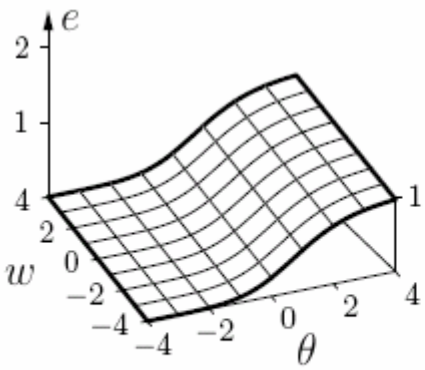
$$\Delta\vec{w}_u^{(l)} = \eta\left(\sum_{s\in\mathrm{succ}(u)} \delta_s^{(l)} w_{su}\right)\mathrm{out}_u^{(l)}\,(1-\mathrm{out}_u^{(l)})\,\vec{\mathrm{in}}_u^{(l)}.$$

Gradient Descent: Examples

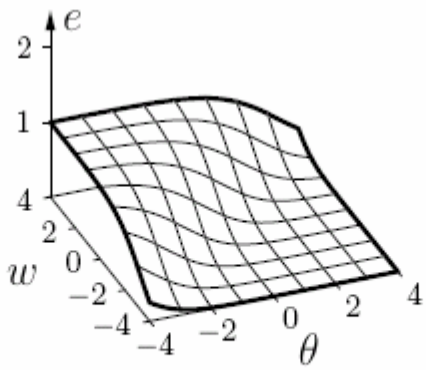
Gradient descent training for the negation $\neg x$



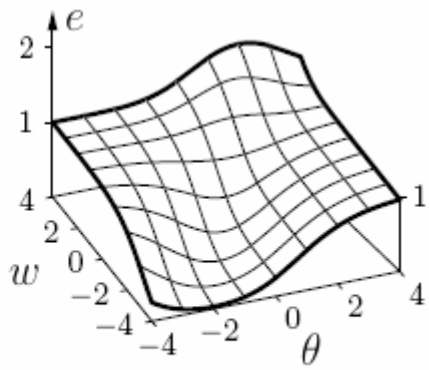
x	y
0	1
1	0



error for $x = 0$



error for $x = 1$



sum of errors

Gradient Descent: Examples

epoch	θ	w	error
0	3.00	3.50	1.307
20	3.77	2.19	0.986
40	3.71	1.81	0.970
60	3.50	1.53	0.958
80	3.15	1.24	0.937
100	2.57	0.88	0.890
120	1.48	0.25	0.725
140	−0.06	−0.98	0.331
160	−0.80	−2.07	0.149
180	−1.19	−2.74	0.087
200	−1.44	−3.20	0.059
220	−1.62	−3.54	0.044

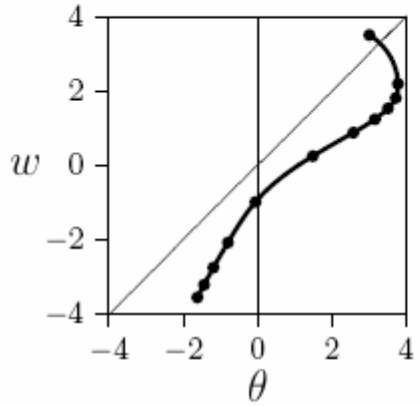
Online Training

epoch	θ	w	error
0	3.00	3.50	1.295
20	3.76	2.20	0.985
40	3.70	1.82	0.970
60	3.48	1.53	0.957
80	3.11	1.25	0.934
100	2.49	0.88	0.880
120	1.27	0.22	0.676
140	−0.21	−1.04	0.292
160	−0.86	−2.08	0.140
180	−1.21	−2.74	0.084
200	−1.45	−3.19	0.058
220	−1.63	−3.53	0.044

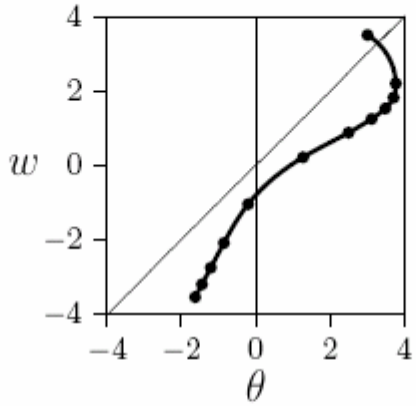
Batch Training

Gradient Descent: Examples

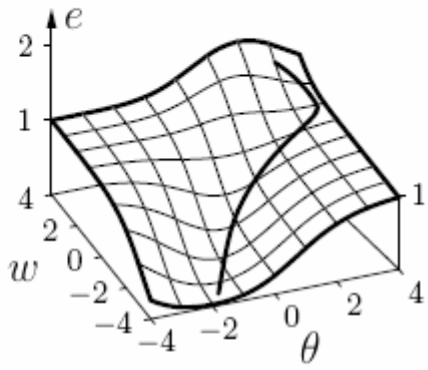
Visualization of gradient descent for the negation $\neg x$



Online Training



Batch Training



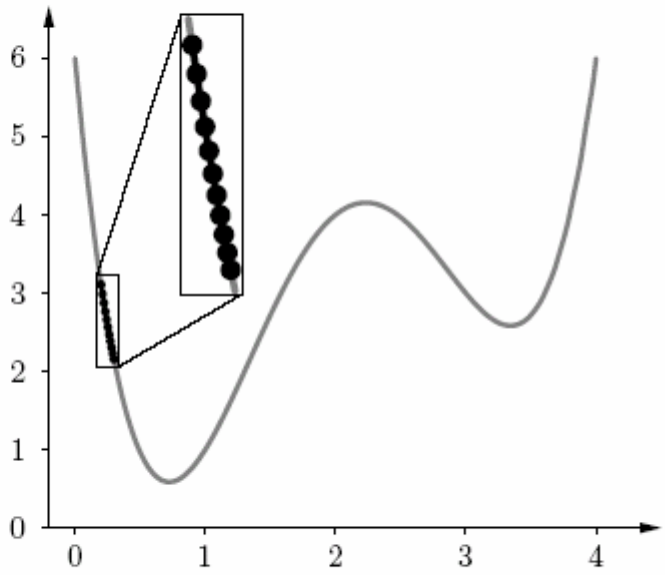
Batch Training

- Training is obviously successful.
- Error cannot vanish completely due to the properties of the logistic function.

Gradient Descent: Examples

Example function: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	0.200	3.112	−11.147	0.011
1	0.211	2.990	−10.811	0.011
2	0.222	2.874	−10.490	0.010
3	0.232	2.766	−10.182	0.010
4	0.243	2.664	−9.888	0.010
5	0.253	2.568	−9.606	0.010
6	0.262	2.477	−9.335	0.009
7	0.271	2.391	−9.075	0.009
8	0.281	2.309	−8.825	0.009
9	0.289	2.233	−8.585	0.009
10	0.298	2.160		

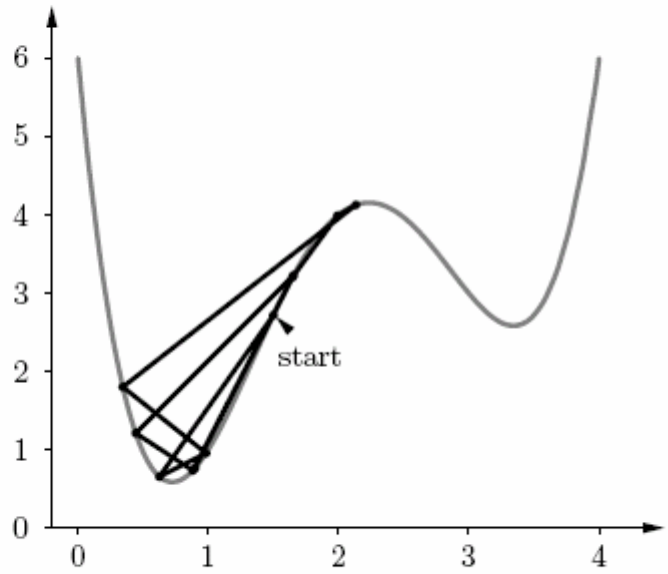


Gradient descent with initial value 0.2 and learning rate 0.001.

Gradient Descent: Examples

Example function: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	1.500	2.719	3.500	−0.875
1	0.625	0.655	−1.431	0.358
2	0.983	0.955	2.554	−0.639
3	0.344	1.801	−7.157	1.789
4	2.134	4.127	0.567	−0.142
5	1.992	3.989	1.380	−0.345
6	1.647	3.203	3.063	−0.766
7	0.881	0.734	1.753	−0.438
8	0.443	1.211	−4.851	1.213
9	1.656	3.231	3.029	−0.757
10	0.898	0.766		

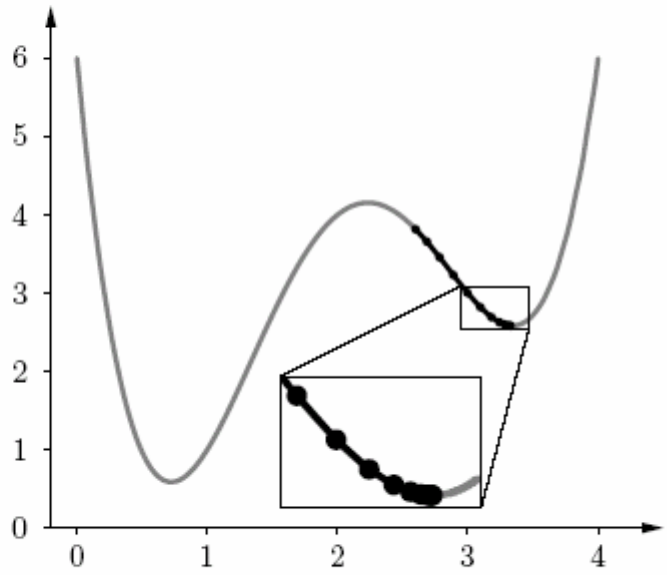


Gradient descent with initial value 1.5 and learning rate 0.25.

Gradient Descent: Examples

Example function: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	2.600	3.816	−1.707	0.085
1	2.685	3.660	−1.947	0.097
2	2.783	3.461	−2.116	0.106
3	2.888	3.233	−2.153	0.108
4	2.996	3.008	−2.009	0.100
5	3.097	2.820	−1.688	0.084
6	3.181	2.695	−1.263	0.063
7	3.244	2.628	−0.845	0.042
8	3.286	2.599	−0.515	0.026
9	3.312	2.589	−0.293	0.015
10	3.327	2.585		



Gradient descent with initial value 2.6 and learning rate 0.05.

Gradient Descent: Variants

Weight update rule:

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

Standard backpropagation:

$$\Delta w(t) = -\frac{\eta}{2} \nabla_w e(t)$$

Manhattan training:

$$\Delta w(t) = -\eta \, \mathrm{sgn}(\nabla_w e(t)).$$

Momentum term:

$$\Delta w(t) = -\frac{\eta}{2} \nabla_w e(t) + \beta \, \Delta w(t-1),$$

Gradient Descent: Variants

Self-adaptive error backpropagation:

$$\eta_w(t) = \begin{cases} c^- \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) < 0, \\ c^+ \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) > 0 \\ & \wedge\ \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \eta_w(t-1), & \text{otherwise.} \end{cases}$$

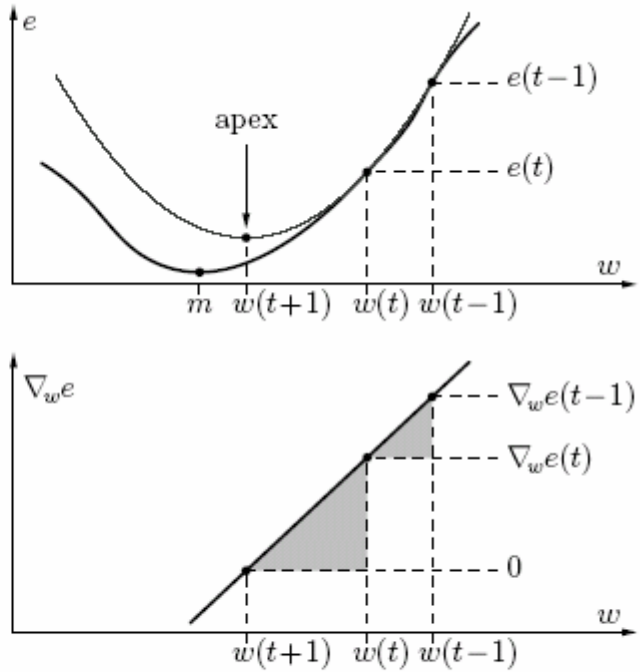
Resilient error backpropagation:

$$\Delta w(t) = \begin{cases} c^- \cdot \Delta w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) < 0, \\ c^+ \cdot \Delta w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) > 0 \\ & \wedge\ \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \Delta w(t-1), & \text{otherwise.} \end{cases}$$

Typical values: $c^- \in [0.5, 0.7]$ and $c^+ \in [1.05, 1.2]$.

Gradient Descent: Variants

Quickpropagation



The weight update rule can be derived from the triangles:

$$\Delta w(t) = \frac{\nabla_w e(t)}{\nabla_w e(t-1) - \nabla_w e(t)} \cdot \Delta w(t-1).$$

Gradient Descent: Examples

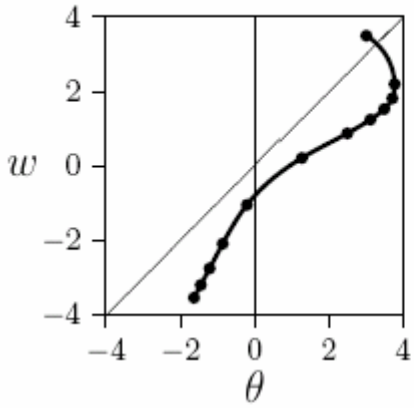
epoch	θ	w	error
0	3.00	3.50	1.295
20	3.76	2.20	0.985
40	3.70	1.82	0.970
60	3.48	1.53	0.957
80	3.11	1.25	0.934
100	2.49	0.88	0.880
120	1.27	0.22	0.676
140	−0.21	−1.04	0.292
160	−0.86	−2.08	0.140
180	−1.21	−2.74	0.084
200	−1.45	−3.19	0.058
220	−1.63	−3.53	0.044

without momentum term

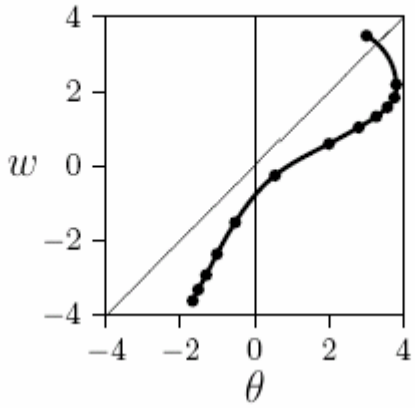
epoch	θ	w	error
0	3.00	3.50	1.295
10	3.80	2.19	0.984
20	3.75	1.84	0.971
30	3.56	1.58	0.960
40	3.26	1.33	0.943
50	2.79	1.04	0.910
60	1.99	0.60	0.814
70	0.54	−0.25	0.497
80	−0.53	−1.51	0.211
90	−1.02	−2.36	0.113
100	−1.31	−2.92	0.073
110	−1.52	−3.31	0.053
120	−1.67	−3.61	0.041

with momentum term

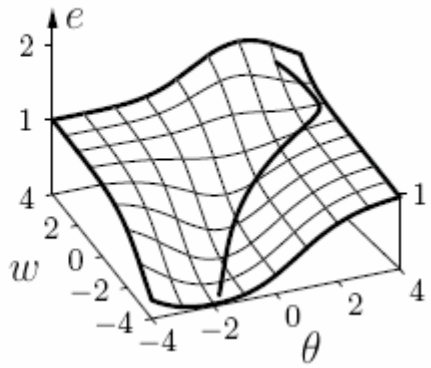
Gradient Descent: Examples



without momentum term



with momentum term



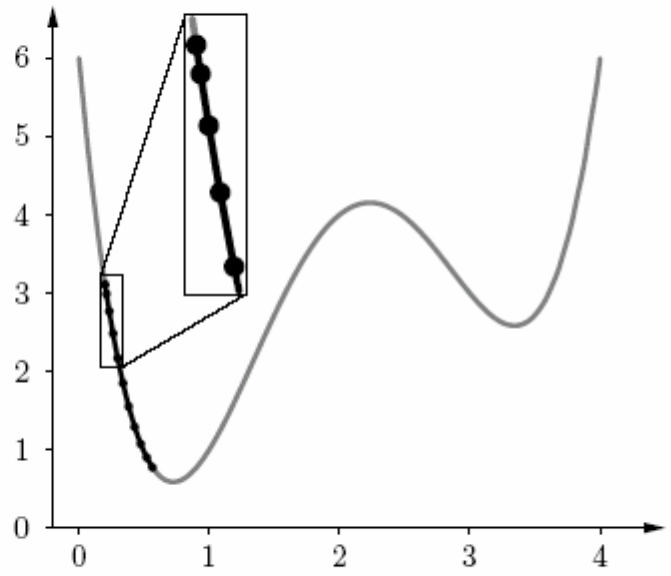
with momentum term

- Dots show position every 20 (without momentum term) or every 10 epochs (with momentum term).
- Learning with a momentum term is about twice as fast.

Gradient Descent: Examples

Example function: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	0.200	3.112	−11.147	0.011
1	0.211	2.990	−10.811	0.021
2	0.232	2.771	−10.196	0.029
3	0.261	2.488	−9.368	0.035
4	0.296	2.173	−8.397	0.040
5	0.337	1.856	−7.348	0.044
6	0.380	1.559	−6.277	0.046
7	0.426	1.298	−5.228	0.046
8	0.472	1.079	−4.235	0.046
9	0.518	0.907	−3.319	0.045
10	0.562	0.777		

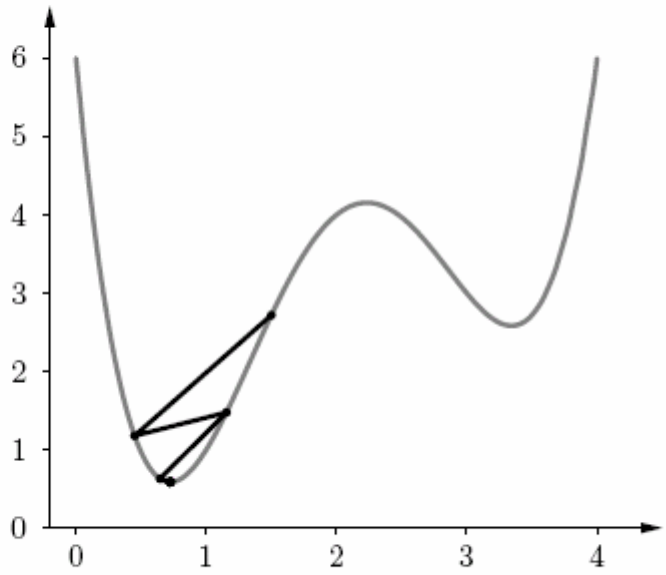


gradient descent with momentum term ($\beta = 0.9$)

Gradient Descent: Examples

Example function: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	1.500	2.719	3.500	−1.050
1	0.450	1.178	−4.699	0.705
2	1.155	1.476	3.396	−0.509
3	0.645	0.629	−1.110	0.083
4	0.729	0.587	0.072	−0.005
5	0.723	0.587	0.001	0.000
6	0.723	0.587	0.000	0.000
7	0.723	0.587	0.000	0.000
8	0.723	0.587	0.000	0.000
9	0.723	0.587	0.000	0.000
10	0.723	0.587		



Gradient descent with self-adapting learning rate ($c^+ = 1.2$, $c^- = 0.5$).

Sensitivity Analysis

Question: How important are different inputs to the network?

Idea: Determine change of output relative to change of input.

$$\forall u \in U_{\text{in}}: \qquad s(u) = \frac{1}{|L_{\text{fixed}}|} \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} \frac{\partial \text{out}_v^{(l)}}{\partial \text{ex}_u^{(l)}}.$$

Formal derivation: Apply chain rule.

$$\frac{\partial \text{out}_v}{\partial \text{ex}_u} = \frac{\partial \text{out}_v}{\partial \text{out}_u} \frac{\partial \text{out}_u}{\partial \text{ex}_u} = \frac{\partial \text{out}_v}{\partial \text{net}_v} \frac{\partial \text{net}_v}{\partial \text{out}_u} \frac{\partial \text{out}_u}{\partial \text{ex}_u}.$$

Simplification: Assume that the output function is the identity.

$$\frac{\partial \text{out}_u}{\partial \text{ex}_u} = 1.$$

Sensitivity Analysis

For the second factor we get the general result:

$$\frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} = \frac{\partial}{\partial \operatorname{out}_u} \sum_{p \in \operatorname{pred}(v)} w_{vp} \operatorname{out}_p = \sum_{p \in \operatorname{pred}(v)} w_{vp} \frac{\partial \operatorname{out}_p}{\partial \operatorname{out}_u}.$$

This leads to the recursion formula

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} \frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} \sum_{p \in \operatorname{pred}(v)} w_{vp} \frac{\partial \operatorname{out}_p}{\partial \operatorname{out}_u}.$$

However, for the first hidden layer we get

$$\frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} = w_{vu}, \qquad \text{therefore} \qquad \frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} w_{vu}.$$

This formula marks the start of the recursion.

Sensitivity Analysis

Consider as usual the special case with

- output function is the identity,
- activation function is logistic.

The recursion formula is in this case

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \operatorname{out}_v(1 - \operatorname{out}_v) \sum_{p \in \operatorname{pred}(v)} w_{vp} \frac{\partial \operatorname{out}_p}{\partial \operatorname{out}_u}$$

and the anchor of the recursion is

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \operatorname{out}_v(1 - \operatorname{out}_v) w_{vu}.$$