

Introduction to Neural Networks

Christian Borgelt

Intelligent Data Analysis and Graphical Models Research Unit
European Center for Soft Computing
c/ Gonzalo Gutiérrez Quirós s/n, 33600 Mieres, Spain

christian.borgelt@softcomputing.es
<http://www.borgelt.net/>

Contents

- **Introduction**
Motivation, Biological Background
- **Threshold Logic Units**
Definition, Geometric Interpretation, Limitations, Networks of TLUs, Training
- **General Neural Networks**
Structure, Operation, Training
- **Multilayer Perceptrons**
Definition, Function Approximation, Gradient Descent, Backpropagation, Variants, Sensitivity Analysis
- **Radial Basis Function Networks**
Definition, Function Approximation, Initialization, Training, Generalized Version
- **Self-Organizing Maps**
Definition, Learning Vector Quantization, Neighborhood of Output Neurons
- **Hopfield Networks**
Definition, Convergence, Associative Memory, Solving Optimization Problems
- **Recurrent Neural Networks**
Differential Equations, Vector Networks, Backpropagation through Time

Motivation: Why (Artificial) Neural Networks?

- **(Neuro-)Biology / (Neuro-)Physiology / Psychology:**
 - Exploit similarity to real (biological) neural networks.
 - Build models to understand nerve and brain operation by simulation.
- **Computer Science / Engineering / Economics**
 - Mimic certain cognitive capabilities of human beings.
 - Solve learning/adaptation, prediction, and optimization problems.
- **Physics / Chemistry**
 - Use neural network models to describe physical phenomena.
 - Special case: spin glasses (alloys of magnetic and non-magnetic metals).

Motivation: Why Neural Networks in AI?

Physical-Symbol System Hypothesis [Newell and Simon 1976]

A physical-symbol system has the necessary and sufficient means for general intelligent action.

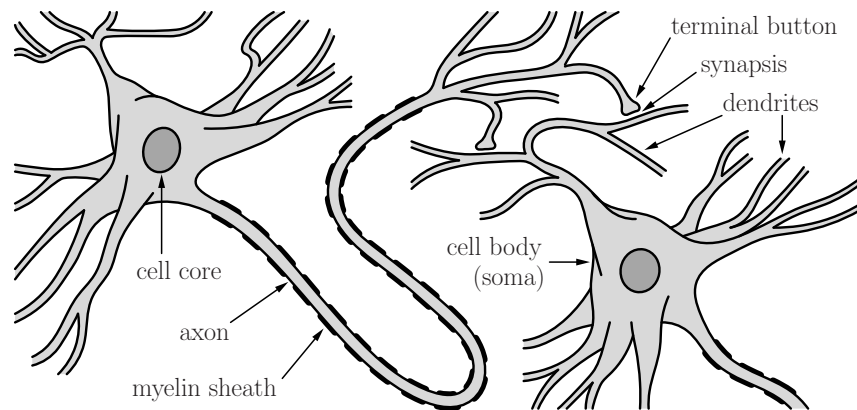
Neural networks process simple signals, not symbols.

So why study neural networks in Artificial Intelligence?

- Symbol-based representations work well for inference tasks, but are fairly bad for perception tasks.
- Symbol-based expert systems tend to get slower with growing knowledge, human experts tend to get faster.
- Neural networks allow for highly parallel information processing.
- There are several successful applications in industry and finance.

Biological Background

Structure of a prototypical biological neuron



Biological Background

(Very) simplified description of neural information processing

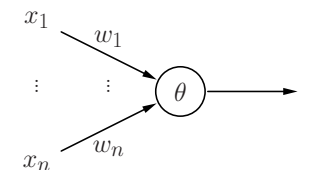
- Axon terminal releases chemicals, called **neurotransmitters**.
- These act on the membrane of the receptor dendrite to change its polarization. (The inside is usually 70mV more negative than the outside.)
- Decrease in potential difference: **excitatory** synapse
Increase in potential difference: **inhibitory** synapse
- If there is enough net excitatory input, the axon is depolarized.
- The resulting **action potential** travels along the axon. (Speed depends on the degree to which the axon is covered with myelin.)
- When the action potential reaches the terminal buttons, it triggers the release of neurotransmitters.

Threshold Logic Units

Threshold Logic Units

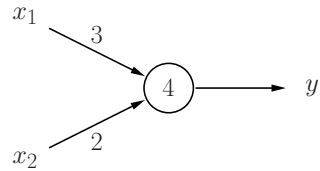
A **Threshold Logic Unit (TLU)** is a processing unit for numbers with n inputs x_1, \dots, x_n and one output y . The unit has a **threshold** θ and each input x_i is associated with a **weight** w_i . A threshold logic unit computes the function

$$y = \begin{cases} 1, & \text{if } \vec{x}\vec{w} = \sum_{i=1}^n w_i x_i \geq \theta, \\ 0, & \text{otherwise.} \end{cases}$$



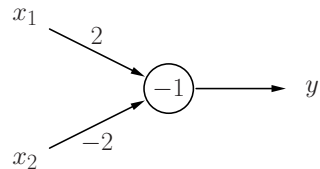
Threshold Logic Units: Examples

Threshold logic unit for the conjunction $x_1 \wedge x_2$.



x_1	x_2	$3x_1 + 2x_2$	y
0	0	0	0
1	0	3	0
0	1	2	0
1	1	5	1

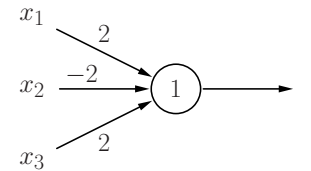
Threshold logic unit for the implication $x_2 \rightarrow x_1$.



x_1	x_2	$2x_1 - 2x_2$	y
0	0	0	1
1	0	2	1
0	1	-2	0
1	1	0	1

Threshold Logic Units: Examples

Threshold logic unit for $(x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_3)$.



x_1	x_2	x_3	$\sum_i w_i x_i$	y
0	0	0	0	0
1	0	0	2	1
0	1	0	-2	0
1	1	0	0	0
0	0	1	2	1
1	0	1	4	1
0	1	1	0	0
1	1	1	2	1

Threshold Logic Units: Geometric Interpretation

Review of line representations

Straight lines are usually represented in one of the following forms:

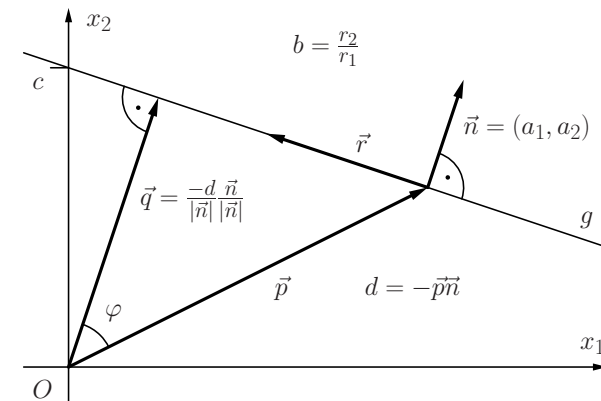
Explicit Form: $g \equiv x_2 = bx_1 + c$
 Implicit Form: $g \equiv a_1x_1 + a_2x_2 + d = 0$
 Point-Direction Form: $g \equiv \vec{x} = \vec{p} + k\vec{r}$
 Normal Form: $g \equiv (\vec{x} - \vec{p})\vec{n} = 0$

with the parameters:

- b : Gradient of the line
- c : Section of the x_2 axis
- \vec{p} : Vector of a point of the line (base vector)
- \vec{r} : Direction vector of the line
- \vec{n} : Normal vector of the line

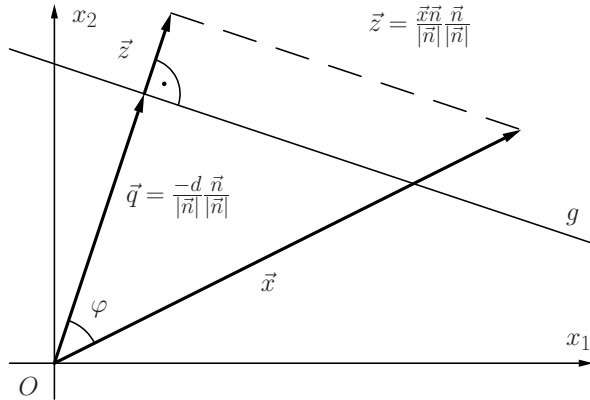
Threshold Logic Units: Geometric Interpretation

A straight line and its defining parameters.



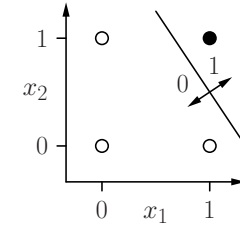
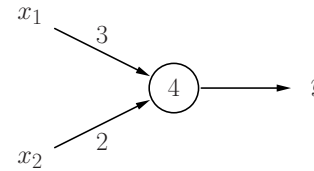
Threshold Logic Units: Geometric Interpretation

How to determine the side on which a point \vec{x} lies.

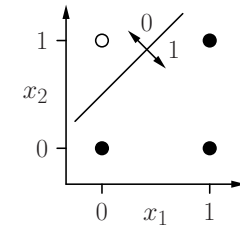
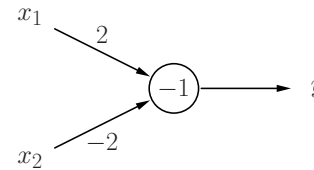


Threshold Logic Units: Geometric Interpretation

Threshold logic unit for $x_1 \wedge x_2$.

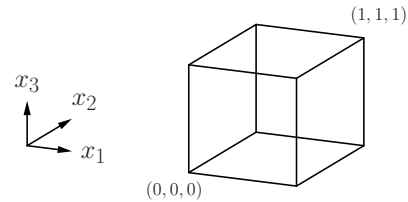


A threshold logic unit for $x_2 \rightarrow x_1$.

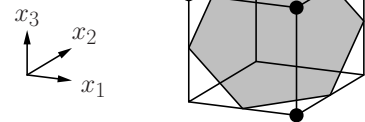
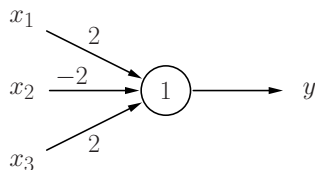


Threshold Logic Units: Geometric Interpretation

Visualization of 3-dimensional Boolean functions:



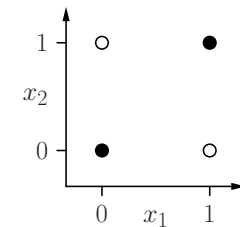
Threshold logic unit for $(x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_3)$.



Threshold Logic Units: Limitations

The biimplication problem $x_1 \leftrightarrow x_2$: There is no separating line.

x_1	x_2	y
0	0	1
1	0	0
0	1	0
1	1	1



Formal proof by *reductio ad absurdum*:

$$\text{since } (0,0) \mapsto 1: \quad 0 \geq \theta, \quad (1)$$

$$\text{since } (1,0) \mapsto 0: \quad w_1 < \theta, \quad (2)$$

$$\text{since } (0,1) \mapsto 0: \quad w_2 < \theta, \quad (3)$$

$$\text{since } (1,1) \mapsto 1: \quad w_1 + w_2 \geq \theta. \quad (4)$$

(2) and (3): $w_1 + w_2 < 2\theta$. With (4): $2\theta > \theta$, or $\theta > 0$. Contradiction to (1).

Threshold Logic Units: Limitations

Total number and number of linearly separable Boolean functions.
 ([Widner 1960] as cited in [Zell 1994])

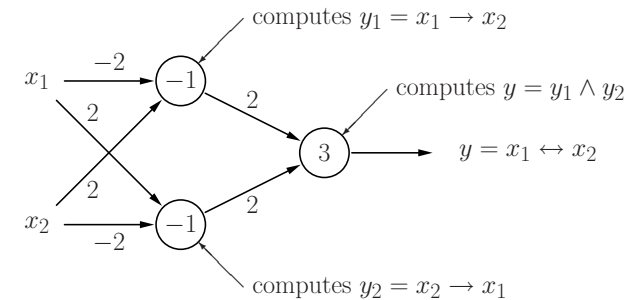
inputs	Boolean functions	linearly separable functions
1	4	4
2	16	14
3	256	104
4	65536	1774
5	$4.3 \cdot 10^9$	94572
6	$1.8 \cdot 10^{19}$	$5.0 \cdot 10^6$

- For many inputs a threshold logic unit can compute almost no functions.
- Networks of threshold logic units are needed to overcome the limitations.

Networks of Threshold Logic Units

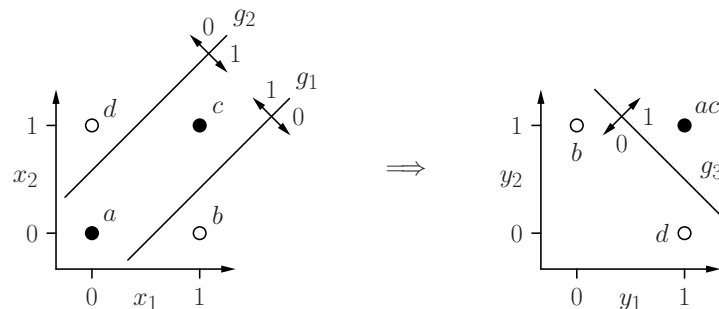
Solving the biimplication problem with a network.

Idea: logical decomposition $x_1 \leftrightarrow x_2 \equiv (x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_1)$



Networks of Threshold Logic Units

Solving the biimplication problem: Geometric interpretation



- The first layer computes new Boolean coordinates for the points.
- After the coordinate transformation the problem is linearly separable.

Representing Arbitrary Boolean Functions

Let $y = f(x_1, \dots, x_n)$ be a Boolean function of n variables.

- Represent $f(x_1, \dots, x_n)$ in disjunctive normal form. That is, determine $D_f = K_1 \vee \dots \vee K_m$, where all K_j are conjunctions of n literals, i.e., $K_j = l_{j1} \wedge \dots \wedge l_{jn}$ with $l_{ji} = x_i$ (positive literal) or $l_{ji} = \neg x_i$ (negative literal).
- Create a neuron for each conjunction K_j of the disjunctive normal form (having n inputs — one input for each variable), where

$$w_{ji} = \begin{cases} 2, & \text{if } l_{ji} = x_i, \\ -2, & \text{if } l_{ji} = \neg x_i, \end{cases} \quad \text{and} \quad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^n w_{ji}.$$

- Create an output neuron (having m inputs — one input for each neuron that was created in step (ii)), where

$$w_{(n+1)k} = 2, \quad k = 1, \dots, m, \quad \text{and} \quad \theta_{n+1} = 1.$$

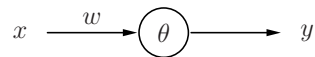
Training Threshold Logic Units

Training Threshold Logic Units

- Geometric interpretation provides a way to construct threshold logic units with 2 and 3 inputs, but:
 - Not an automatic method (human visualization needed).
 - Not feasible for more than 3 inputs.
- **General idea of automatic training:**
 - Start with random values for weights and threshold.
 - Determine the error of the output for a set of training patterns.
 - Error is a function of the weights and the threshold: $e = e(w_1, \dots, w_n, \theta)$.
 - Adapt weights and threshold so that the error gets smaller.
 - Iterate adaptation until the error vanishes.

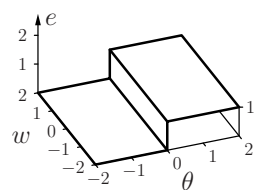
Training Threshold Logic Units

Single input threshold logic unit for the negation $\neg x$.

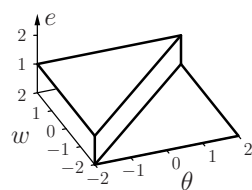


x	y
0	1
1	0

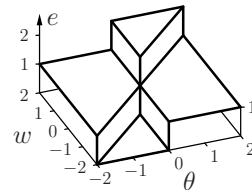
Output error as a function of weight and threshold.



error for $x = 0$



error for $x = 1$

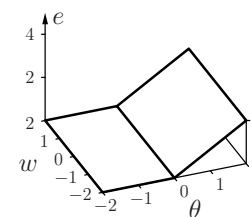


sum of errors

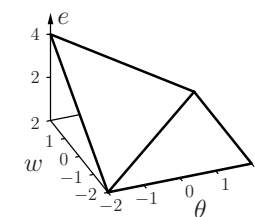
Training Threshold Logic Units

- The error function cannot be used directly, because it consists of plateaus.
- Solution: If the computed output is wrong, take into account, how far the weighted sum is from the threshold.

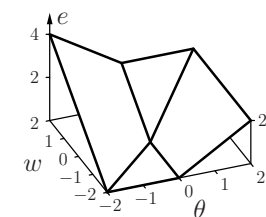
Modified output error as a function of weight and threshold.



error for $x = 0$



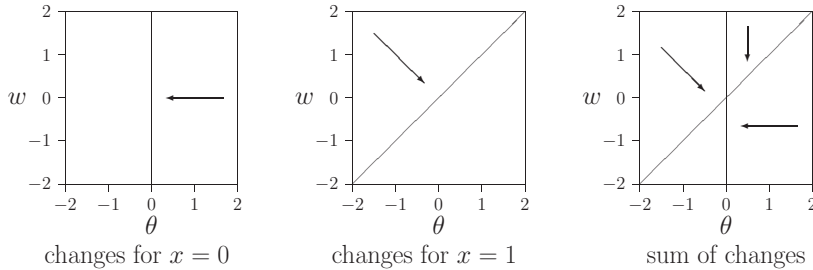
error for $x = 1$



sum of errors

Training Threshold Logic Units

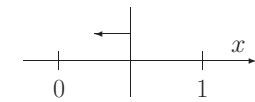
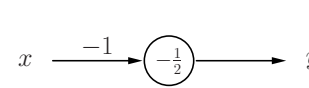
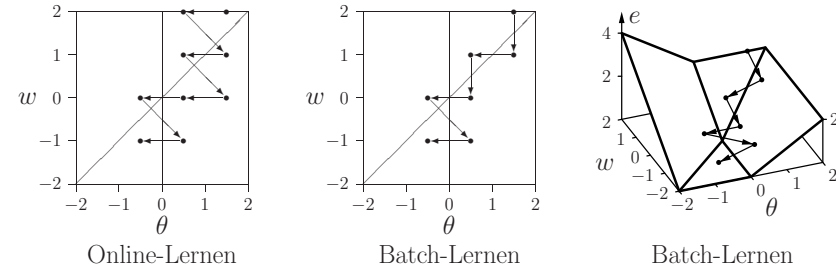
Schemata of resulting directions of parameter changes.



- Start at random point.
- Iteratively adapt parameters according to the direction corresponding to the current point.

Training Threshold Logic Units

Example training procedure: Online and batch training.



Training Threshold Logic Units: Delta Rule

Formal Training Rule: Let $\vec{x} = (x_1, \dots, x_n)$ be an input vector of a threshold logic unit, o the desired output for this input vector and y the actual output of the threshold logic unit. If $y \neq o$, then the threshold θ and the weight vector $\vec{w} = (w_1, \dots, w_n)$ are adapted as follows in order to reduce the error:

$$\theta^{(\text{new})} = \theta^{(\text{old})} + \Delta\theta \quad \text{with} \quad \Delta\theta = -\eta(o - y),$$

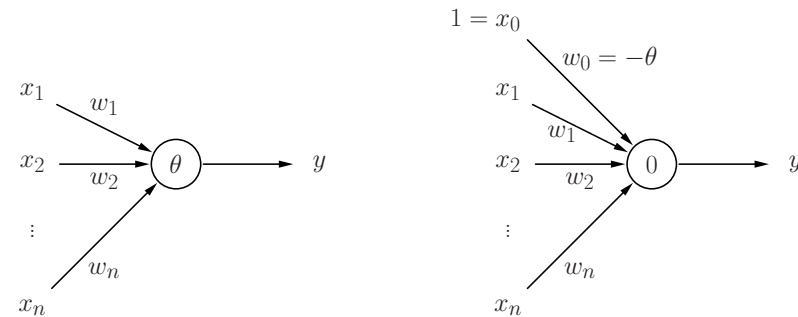
$$\forall i \in \{1, \dots, n\}: w_i^{(\text{new})} = w_i^{(\text{old})} + \Delta w_i \quad \text{with} \quad \Delta w_i = \eta(o - y)x_i,$$

where η is a parameter that is called **learning rate**. It determines the severity of the weight changes. This procedure is called **Delta Rule** or **Widrow–Hoff Procedure** [Widrow and Hoff 1960].

- **Online Training:** Adapt parameters after each training pattern.
- **Batch Training:** Adapt parameters only at the end of each **epoch**, i.e. after a traversal of all training patterns.

Training Threshold Logic Units: Delta Rule

Turning the threshold value into a weight:



$$\sum_{i=1}^n w_i x_i \geq \theta$$

$$\sum_{i=1}^n w_i x_i - \theta \geq 0$$

Training Threshold Logic Units: Delta Rule

```

procedure online_training (var  $\vec{w}$ , var  $\theta$ ,  $L$ ,  $\eta$ );
var  $y$ ,  $e$ ;                                (* output, sum of errors *)
begin
  repeat
     $e := 0$ ;                                (* initialize the error sum *)
    for all  $(\vec{x}, o) \in L$  do begin        (* traverse the patterns *)
      if  $(\vec{w}\vec{x} \geq \theta)$  then  $y := 1$ ;    (* compute the output *)
      else  $y := 0$ ;                          (* of the threshold logic unit *)
      if  $(y \neq o)$  then begin              (* if the output is wrong *)
         $\theta := \theta - \eta(o - y)$ ;        (* adapt the threshold *)
         $\vec{w} := \vec{w} + \eta(o - y)\vec{x}$ ;      (* and the weights *)
         $e := e + |o - y|$ ;                  (* sum the errors *)
      end;
    end;
  until  $(e \leq 0)$ ;                          (* repeat the computations *)
end;                                         (* until the error vanishes *)

```

Training Threshold Logic Units: Delta Rule

```

procedure batch_training (var  $\vec{w}$ , var  $\theta$ ,  $L$ ,  $\eta$ );
var  $y$ ,  $e$ ,                                (* output, sum of errors *)
     $\theta_c$ ,  $\vec{w}_c$ ;                        (* summed changes *)
begin
  repeat
     $e := 0$ ;  $\theta_c := 0$ ;  $\vec{w}_c := \vec{0}$ ;      (* initializations *)
    for all  $(\vec{x}, o) \in L$  do begin        (* traverse the patterns *)
      if  $(\vec{w}\vec{x} \geq \theta)$  then  $y := 1$ ;    (* compute the output *)
      else  $y := 0$ ;                          (* of the threshold logic unit *)
      if  $(y \neq o)$  then begin              (* if the output is wrong *)
         $\theta_c := \theta_c - \eta(o - y)$ ;    (* sum the changes of the *)
         $\vec{w}_c := \vec{w}_c + \eta(o - y)\vec{x}$ ;  (* threshold and the weights *)
         $e := e + |o - y|$ ;                  (* sum the errors *)
      end;
    end;
     $\theta := \theta + \theta_c$ ;              (* adapt the threshold *)
     $\vec{w} := \vec{w} + \vec{w}_c$ ;                  (* and the weights *)
  until  $(e \leq 0)$ ;                          (* repeat the computations *)
end;                                         (* until the error vanishes *)

```

Training Threshold Logic Units: Online

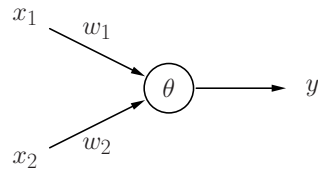
epoch	x	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	-1.5	0	1	-1	0	0.5	2
	1	0	1.5	1	-1	1	-1	1.5	1
2	0	1	-1.5	0	1	-1	0	0.5	1
	1	0	0.5	1	-1	1	-1	1.5	0
3	0	1	-1.5	0	1	-1	0	0.5	0
	1	0	0.5	0	0	0	0	0.5	0
4	0	1	-0.5	0	1	-1	0	-0.5	0
	1	0	0.5	1	-1	1	-1	0.5	-1
5	0	1	-0.5	0	1	-1	0	-0.5	-1
	1	0	-0.5	0	0	0	0	-0.5	-1
6	0	1	0.5	1	0	0	0	-0.5	-1
	1	0	-0.5	0	0	0	0	-0.5	-1

Training Threshold Logic Units: Batch

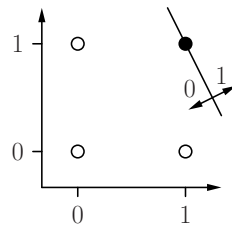
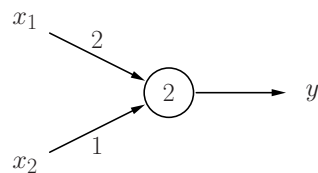
epoch	x	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	-1.5	0	1	-1	0	1.5	1
	1	0	0.5	1	-1	1	-1		
2	0	1	-1.5	0	1	-1	0	0.5	1
	1	0	-0.5	0	0	0	0		
3	0	1	-0.5	0	1	-1	0	0.5	0
	1	0	0.5	1	-1	1	-1		
4	0	1	-0.5	0	1	-1	0	-0.5	0
	1	0	-0.5	0	0	0	0		
5	0	1	0.5	1	0	0	0	0.5	-1
	1	0	0.5	1	-1	1	-1		
6	0	1	-0.5	0	1	-1	0	-0.5	-1
	1	0	-1.5	0	0	0	0		
7	0	1	0.5	1	0	0	0	-0.5	-1
	1	0	-0.5	0	0	0	0		

Training Threshold Logic Units: Conjunction

Threshold logic unit with two inputs for the conjunction.



x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1



Training Threshold Logic Units: Conjunction

epoch	x_1	x_2	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw_1	Δw_2	θ	w_1	w_2
										0	0	0
1	0	0	0	0	1	-1	1	0	0	1	0	0
	0	1	0	-1	0	0	0	0	0	1	0	0
	1	0	0	-1	0	0	0	0	0	1	0	0
	1	1	1	-1	0	1	-1	1	1	0	1	1
2	0	0	0	0	1	-1	1	0	0	1	1	1
	0	1	0	0	1	-1	1	0	-1	2	1	0
	1	0	0	-1	0	0	0	0	0	2	1	0
	1	1	1	-1	0	1	-1	1	1	1	2	1
3	0	0	0	-1	0	0	0	0	0	1	2	1
	0	1	0	0	1	-1	1	0	-1	2	2	0
	1	0	0	0	1	-1	1	-1	0	3	1	0
	1	1	1	-2	0	1	-1	1	1	2	2	1
4	0	0	0	-2	0	0	0	0	0	2	2	1
	0	1	0	-1	0	0	0	0	0	2	2	1
	1	0	0	0	1	-1	1	-1	0	3	1	1
	1	1	1	-1	0	1	-1	1	1	2	2	2
5	0	0	0	-2	0	0	0	0	0	2	2	2
	0	1	0	0	1	-1	1	0	-1	3	2	1
	1	0	0	-1	0	0	0	0	0	3	2	1
	1	1	1	0	1	0	0	0	0	3	2	1
6	0	0	0	-3	0	0	0	0	0	3	2	1
	0	1	0	-2	0	0	0	0	0	3	2	1
	1	0	0	-1	0	0	0	0	0	3	2	1
	1	1	1	0	1	0	0	0	0	3	2	1

Training Threshold Logic Units: Biimplication

epoch	x_1	x_2	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw_1	Δw_2	θ	w_1	w_2
										0	0	0
1	0	0	1	0	1	0	0	0	0	0	0	0
	0	1	0	0	1	-1	1	0	-1	1	0	-1
	1	0	0	-1	0	0	0	0	0	1	0	-1
	1	1	1	-2	0	1	-1	1	1	0	1	0
2	0	0	1	0	1	0	0	0	0	0	1	0
	0	1	0	0	1	-1	1	0	-1	1	1	-1
	1	0	0	0	1	-1	1	-1	0	2	0	-1
	1	1	1	-3	0	1	-1	1	1	1	1	0
3	0	0	1	0	1	0	0	0	0	0	1	0
	0	1	0	0	1	-1	1	0	-1	1	1	-1
	1	0	0	0	1	-1	1	-1	0	2	0	-1
	1	1	1	-3	0	1	-1	1	1	1	1	0

Training Threshold Logic Units: Convergence

Convergence Theorem: Let $L = \{(\vec{x}_1, o_1), \dots, (\vec{x}_m, o_m)\}$ be a set of training patterns, each consisting of an input vector $\vec{x}_i \in \mathbb{R}^n$ and a desired output $o_i \in \{0, 1\}$. Furthermore, let $L_0 = \{(\vec{x}, o) \in L \mid o = 0\}$ and $L_1 = \{(\vec{x}, o) \in L \mid o = 1\}$. If L_0 and L_1 are linearly separable, i.e., if $\vec{w} \in \mathbb{R}^n$ and $\theta \in \mathbb{R}$ exist, such that

$$\begin{aligned} \forall (\vec{x}, 0) \in L_0 : \quad \vec{w}\vec{x} < \theta \quad \text{and} \\ \forall (\vec{x}, 1) \in L_1 : \quad \vec{w}\vec{x} \geq \theta, \end{aligned}$$

then online as well as batch training terminate.

- The algorithms terminate only when the error vanishes.
- Therefore the resulting threshold and weights must solve the problem.
- For not linearly separable problems the algorithms do not terminate.

Training Networks of Threshold Logic Units

- Single threshold logic units have strong limitations:
They can only compute linearly separable functions.
- Networks of threshold logic units can compute arbitrary Boolean functions.
- Training single threshold logic units with the delta rule is fast and guaranteed to find a solution if one exists.
- Networks of threshold logic units cannot be trained, because
 - there are no desired values for the neurons of the first layer,
 - the problem can usually be solved with different functions computed by the neurons of the first layer.
- When this situation became clear, neural networks were seen as a “research dead end”.

General (Artificial) 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

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

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

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 $u \in U_{\text{in}}$ 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** $f_{\text{net}}^{(u)} : \mathbb{R}^{2|\text{pred}(u)|+\kappa_1(u)} \rightarrow \mathbb{R}$,
- the **activation function** $f_{\text{act}}^{(u)} : \mathbb{R}^{\kappa_2(u)} \rightarrow \mathbb{R}$, and
- the **output function** $f_{\text{out}}^{(u)} : \mathbb{R} \rightarrow \mathbb{R}$,

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

General Neural Networks

Types of (artificial) neural networks

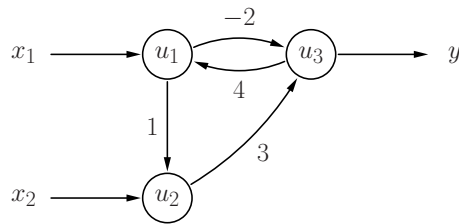
- If the graph of a neural network is **acyclic**, it is called a **feed-forward network**.
- If the graph of a neural network contains **cycles** (backward connections), it is called a **recurrent network**.

Representation of the connection weights by a matrix

$$\begin{pmatrix} w_{u_1 u_1} & w_{u_1 u_2} & \dots & w_{u_1 u_r} \\ w_{u_2 u_1} & w_{u_2 u_2} & & w_{u_2 u_r} \\ \vdots & & & \vdots \\ w_{u_r u_1} & w_{u_r u_2} & \dots & w_{u_r u_r} \end{pmatrix} \begin{matrix} u_1 \\ u_2 \\ \vdots \\ u_r \end{matrix}$$

General Neural Networks: Example

A simple recurrent neural network

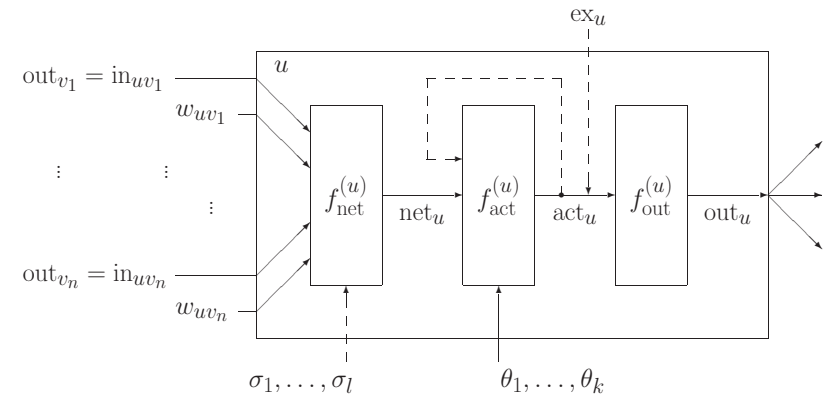


Weight matrix of this network

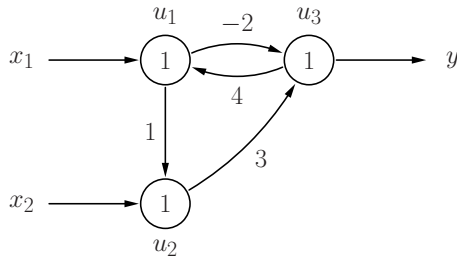
$$\begin{pmatrix} 0 & 0 & 4 \\ 1 & 0 & 0 \\ -2 & 3 & 0 \end{pmatrix} \begin{matrix} u_1 \\ u_2 \\ u_3 \end{matrix}$$

Structure of a Generalized Neuron

A generalized neuron is a simple numeric processor



General Neural Networks: Example



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

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

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

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:
 $u_3, u_1, u_2, u_3, u_1, u_2, u_3, \dots$
- 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:
 $u_3, u_2, u_1, u_3, u_2, u_1, u_3, \dots$
- 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. $U_{\text{in}} = \{u_1, \dots, u_n\}$, and
- m output neurons, i.e. $U_{\text{out}} = \{v_1, \dots, v_m\}$,

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

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

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 $\vec{v}^{(l)}$ 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.

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

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

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. $U_{\text{in}} = \{u_1, \dots, u_n\}$,

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

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

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:

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

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

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

- Avoids unit and scaling problems.

Multilayer Perceptrons (MLPs)

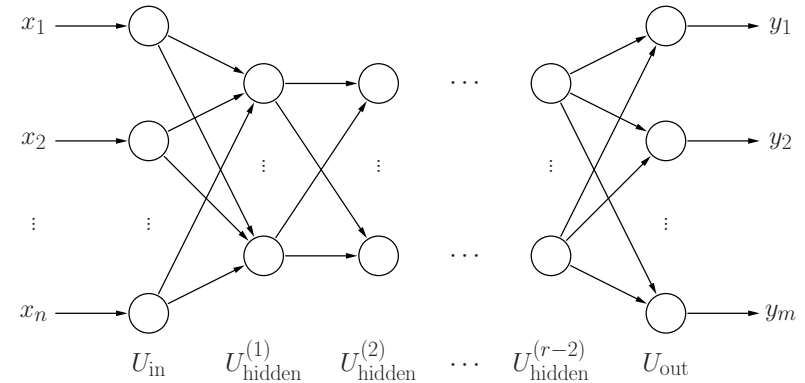
Multilayer Perceptrons

An **r layer 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) $U_{\text{hidden}} = U_{\text{hidden}}^{(1)} \cup \dots \cup U_{\text{hidden}}^{(r-2)}$,
 $\forall 1 \leq i < j \leq r-2: U_{\text{hidden}}^{(i)} \cap U_{\text{hidden}}^{(j)} = \emptyset$,
 - (iii) $C \subseteq (U_{\text{in}} \times U_{\text{hidden}}^{(1)}) \cup (U_{\text{hidden}}^{(1)} \times U_{\text{hidden}}^{(2)}) \cup \dots \cup (U_{\text{hidden}}^{(r-2)} \times U_{\text{out}})$
 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.

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

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

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

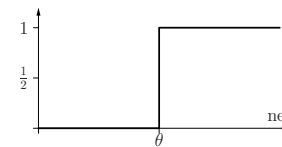
- The activation function of each output neuron is either also a sigmoid function or a **linear function**, i.e.

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

Sigmoid Activation Functions

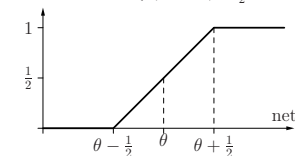
step function:

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



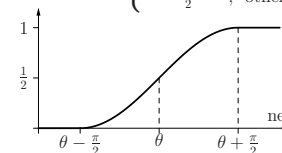
semi-linear function:

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



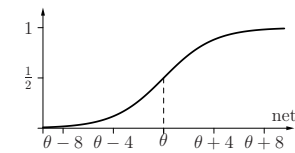
sine until saturation:

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



logistic function:

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

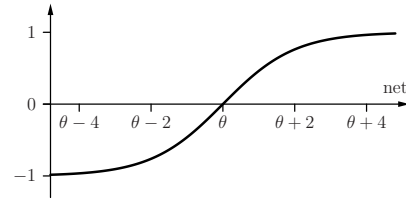


Sigmoid Activation Functions

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

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

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

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

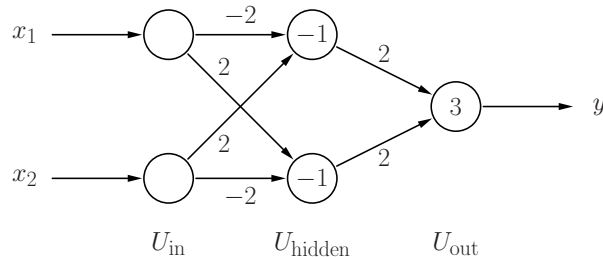
Advantage: The computation of the network input can be written as

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

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

Multilayer Perceptrons: Biimplication

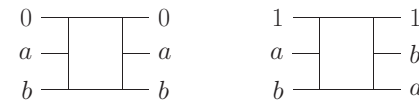
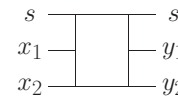
Solving the biimplication problem with a multilayer perceptron.



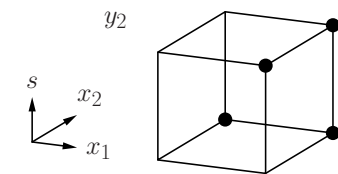
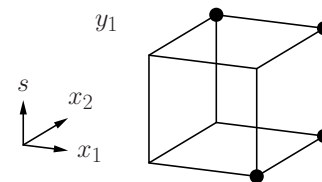
Note the additional input neurons compared to the TLU solution.

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

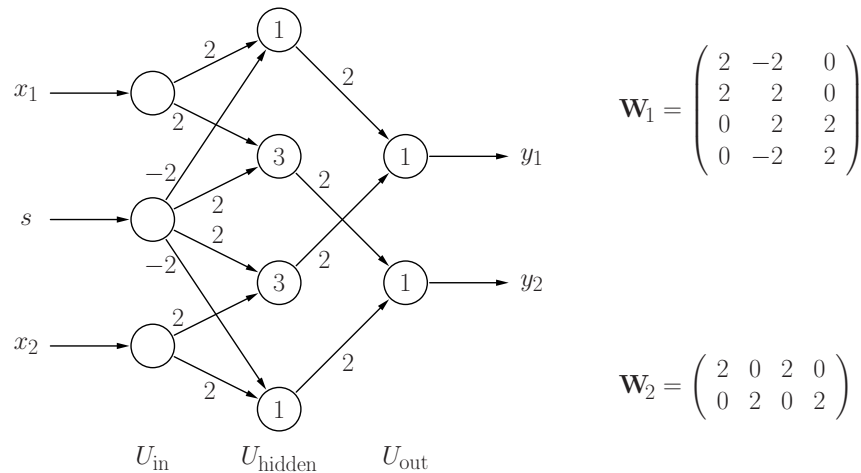
Multilayer Perceptrons: Fredkin Gate



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



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

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

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

Combining these computations we get

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

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

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

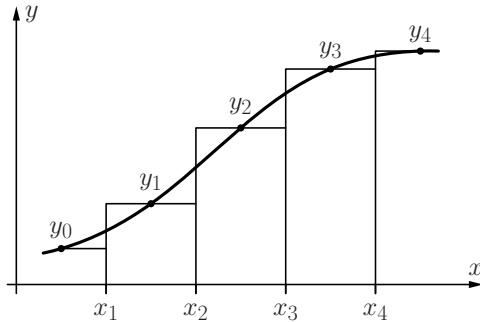
where $\mathbf{A}_{13} = \mathbf{A}_{23} \cdot \mathbf{A}_{12}$ and $\vec{b}_{13} = \mathbf{A}_{23} \cdot \vec{b}_{12} + \vec{b}_{23}$.

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

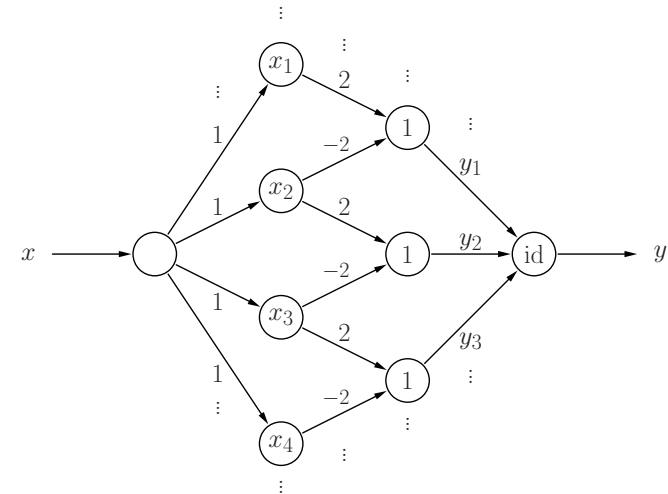
Multilayer Perceptrons: Function Approximation

General idea of function approximation

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



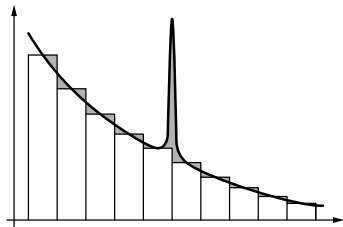
Multilayer Perceptrons: Function Approximation



Multilayer Perceptrons: Function Approximation

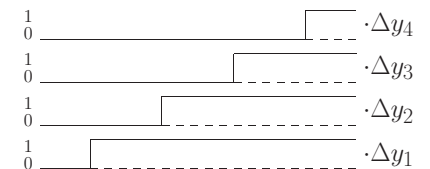
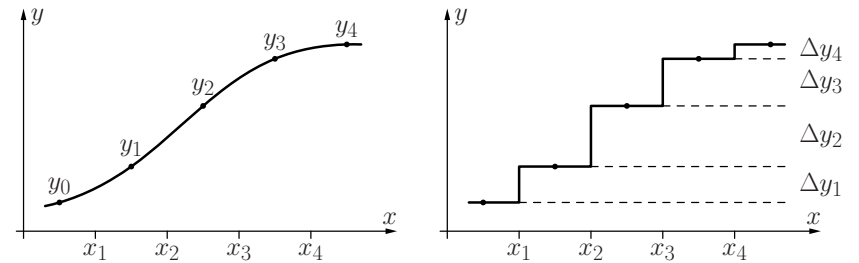
Theorem: Any Riemann-integrable function can be approximated with arbitrary accuracy by a four-layer perceptron.

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

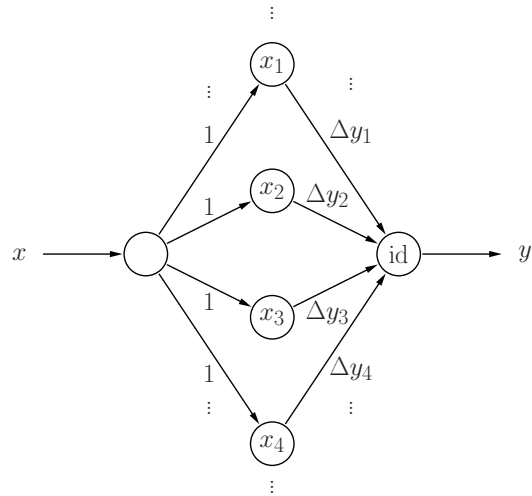


- More sophisticated mathematical examination allows a stronger assertion: With a three-layer perceptron any continuous function can be approximated with arbitrary accuracy (error: maximum function value difference).

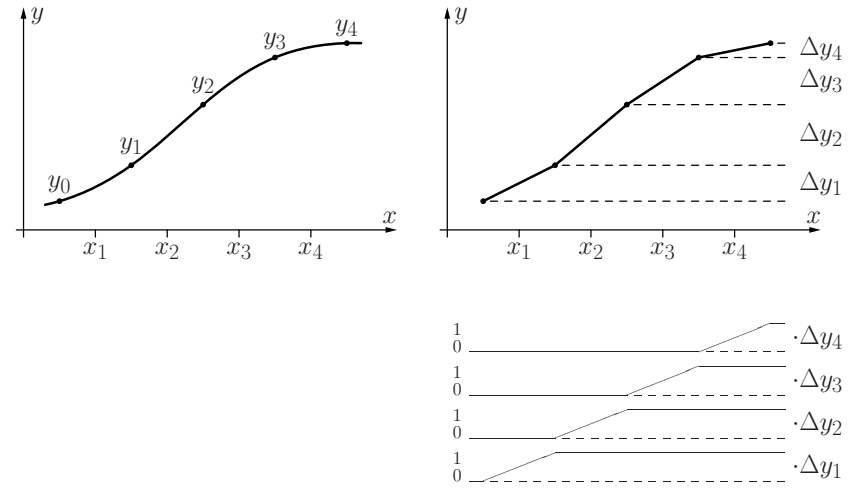
Multilayer Perceptrons: Function Approximation



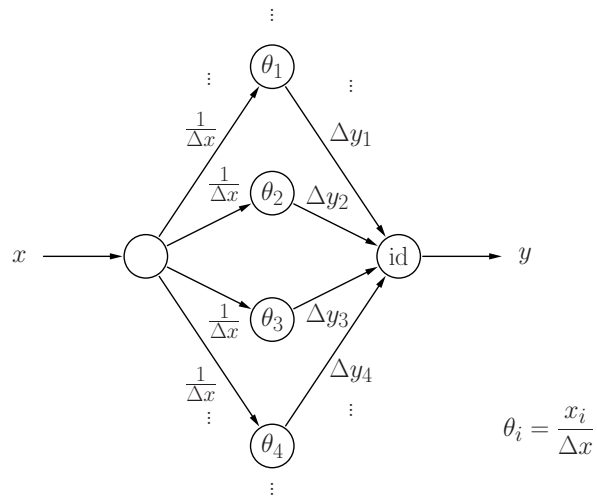
Multilayer Perceptrons: Function Approximation



Multilayer Perceptrons: Function Approximation



Multilayer Perceptrons: Function Approximation



Mathematical Background: Regression

Mathematical Background: Linear Regression

Training neural networks is closely related to regression

- Given:
- A dataset $((x_1, y_1), \dots, (x_n, y_n))$ 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.

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

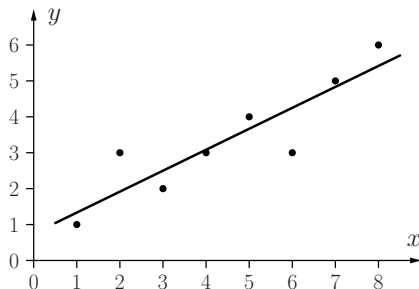
$$\left(\sum_{i=1}^n x_i \right) a + \left(\sum_{i=1}^n x_i^2 \right) b = \sum_{i=1}^n x_i y_i.$$

- 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

$$y = p(x) = a_0 + a_1x + \dots + a_mx^m$$

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

$$F(a_0, a_1, \dots, a_m) = \sum_{i=1}^n (p(x_i) - y_i)^2 = \sum_{i=1}^n (a_0 + a_1x_i + \dots + a_mx_i^m - y_i)^2$$

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

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

Multilinear Regression

- $\nabla_{\vec{a}} F(\vec{a})$ may easily be computed by remembering that the differential operator

$$\nabla_{\vec{a}} = \left(\frac{\partial}{\partial a_0}, \dots, \frac{\partial}{\partial a_m} \right)$$

behaves formally like a vector that is “multiplied” to the sum of squared errors.

- Alternatively, one may write out the differentiation componentwise.

With the former method we obtain for the derivative:

$$\begin{aligned} \nabla_{\vec{a}} (\mathbf{X}\vec{a} - \vec{y})^\top (\mathbf{X}\vec{a} - \vec{y}) &= (\nabla_{\vec{a}} (\mathbf{X}\vec{a} - \vec{y}))^\top (\mathbf{X}\vec{a} - \vec{y}) + ((\mathbf{X}\vec{a} - \vec{y})^\top (\nabla_{\vec{a}} (\mathbf{X}\vec{a} - \vec{y})))^\top \\ &= (\nabla_{\vec{a}} (\mathbf{X}\vec{a} - \vec{y}))^\top (\mathbf{X}\vec{a} - \vec{y}) + (\nabla_{\vec{a}} (\mathbf{X}\vec{a} - \vec{y}))^\top (\mathbf{X}\vec{a} - \vec{y}) \\ &= 2\mathbf{X}^\top (\mathbf{X}\vec{a} - \vec{y}) \\ &= 2\mathbf{X}^\top \mathbf{X}\vec{a} - 2\mathbf{X}^\top \vec{y} = \vec{0} \end{aligned}$$

Multilinear Regression

Necessary condition for a minimum therefore:

$$\begin{aligned} \nabla_{\vec{a}} F(\vec{a}) &= \nabla_{\vec{a}} (\mathbf{X}\vec{a} - \vec{y})^\top (\mathbf{X}\vec{a} - \vec{y}) \\ &= 2\mathbf{X}^\top \mathbf{X}\vec{a} - 2\mathbf{X}^\top \vec{y} \stackrel{!}{=} \vec{0} \end{aligned}$$

As a consequence we get the system of **normal equations**:

$$\mathbf{X}^\top \mathbf{X}\vec{a} = \mathbf{X}^\top \vec{y}$$

This system has a solution if $\mathbf{X}^\top \mathbf{X}$ is not singular. Then we have

$$\vec{a} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \vec{y}.$$

$(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ is called the (Moore-Penrose-) **Pseudoinverse** of the matrix \mathbf{X} .

With the matrix-vector representation of the regression problem an extension to **multipolynomial regression** is straightforward: Simply add the desired products of powers to the matrix \mathbf{X} .

Mathematical Background: Logistic Regression

Generalization to non-polynomial functions

Simple example: $y = ax^b$

Idea: Find transformation to linear/polynomial case.

Transformation for example: $\ln y = \ln a + b \cdot \ln x$.

Special case: **logistic function**

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

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

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

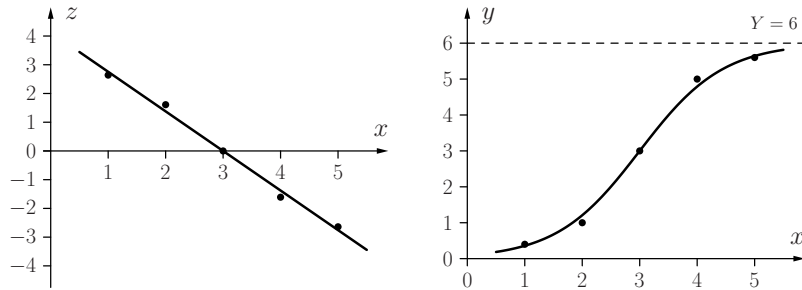
The transformed data points are

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

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**.

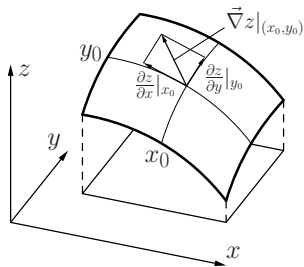


Illustration of the gradient of a real-valued function $z = f(x, y)$ at a point (x_0, y_0) .

It is $\vec{\nabla}z|_{(x_0, y_0)} = \left(\frac{\partial z}{\partial x}|_{x_0, y_0}, \frac{\partial z}{\partial y}|_{x_0, y_0} \right)$.

Gradient Descent: Formal Approach

General Idea: Approach the minimum of the 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 the direction of the step:

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

Exploit the sum over the 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:

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

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

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

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

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

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

Gradient Descent: Formal Approach

Therefore we have

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

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

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

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

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

Therefore we have for the gradient

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

and thus for the weight change

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

Gradient Descent: Formal Approach

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

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

Consider special case with

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

The first assumption yields

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

Gradient Descent: Formal Approach

For a logistic activation function we have

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

and therefore

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

The resulting weight change is therefore

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

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 $\text{succ}(u) = \{s \in U \mid (u, s) \in C\} = \{s_1, \dots, s_m\} \subseteq U_{k+1}$, 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

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

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

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

The result is the recursive equation (error backpropagation)

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

Error Backpropagation

The resulting formula for the weight change is

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

Consider again the special case with

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

The resulting formula for the weight change is then

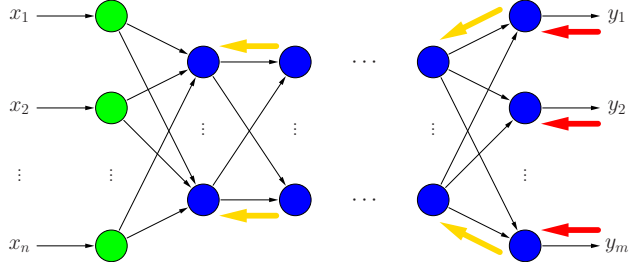
$$\Delta \vec{w}_u^{(l)} = \eta \left(\sum_{s \in \text{succ}(u)} \delta_s^{(l)} w_{su} \right) \text{out}_u^{(l)} (1 - \text{out}_u^{(l)}) \vec{\text{in}}_u^{(l)}.$$

Error Backpropagation: Cookbook Recipe

$$\forall u \in U_{\text{in}} : \text{out}_u^{(l)} = \text{ex}_u^{(l)}$$

forward propagation:

$$\forall u \in U_{\text{hidden}} \cup U_{\text{out}} : \text{out}_u^{(l)} = \left(1 + \exp\left(-\sum_{p \in \text{pred}(u)} w_{up} \text{out}_p^{(l)}\right)\right)^{-1}$$



- logistic activation function
- implicit bias value

error factor:

backward propagation:

$$\forall u \in U_{\text{hidden}} : \delta_u^{(l)} = \left(\sum_{s \in \text{succ}(u)} \delta_s^{(l)} w_{su}\right) \lambda_u^{(l)}$$

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

activation derivative:

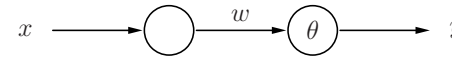
$$\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right)$$

weight change:

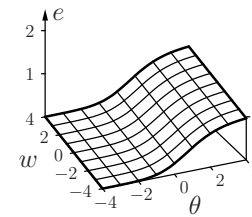
$$\Delta w_{up}^{(l)} = \eta \delta_u^{(l)} \text{out}_p^{(l)}$$

Gradient Descent: Examples

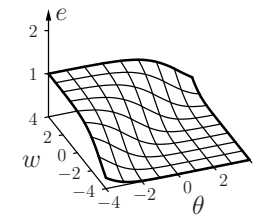
Gradient descent training for the negation $\neg x$



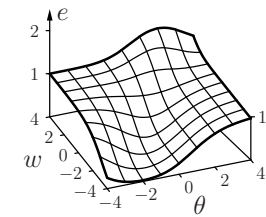
x	y
0	1
1	0



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

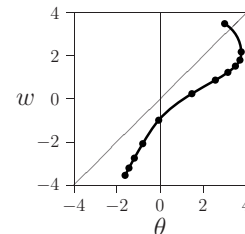
Online Training

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

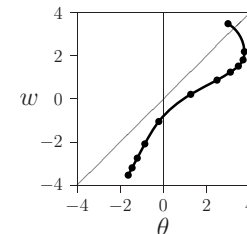
Batch Training

Gradient Descent: Examples

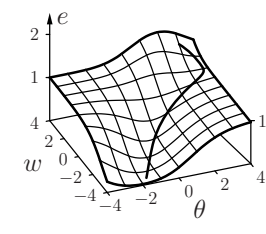
Visualization of gradient descent for the negation $\neg x$



Online Training



Batch Training



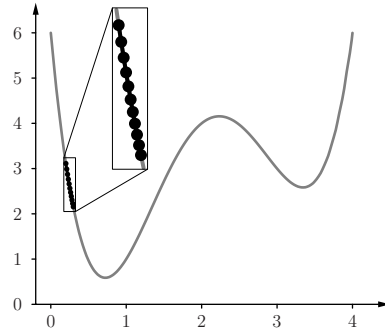
Batch Training

- 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		

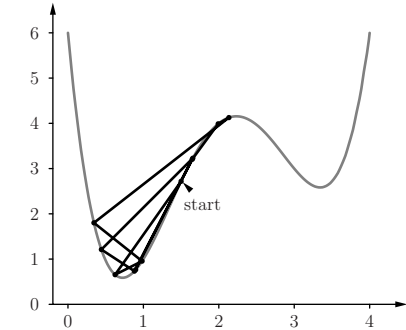


Gradient descent with initial value 0.2 and learning rate 0.001.

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		

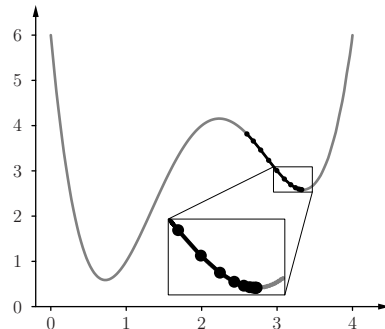


Gradient descent with initial value 1.5 and learning rate 0.25.

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:

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

Momentum term:

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

Gradient Descent: Variants

Self-adaptive error backpropagation:

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

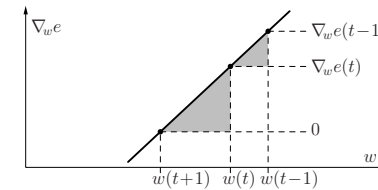
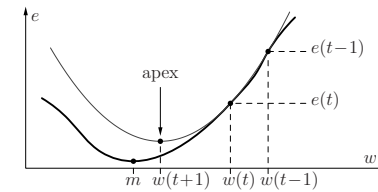
Resilient error backpropagation:

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

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

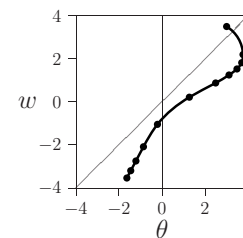
epoch	θ	w	error
0	3.00	3.50	1.295
20	3.76	2.20	0.985
40	3.70	1.82	0.970
60	3.48	1.53	0.957
80	3.11	1.25	0.934
100	2.49	0.88	0.880
120	1.27	0.22	0.676
140	-0.21	-1.04	0.292
160	-0.86	-2.08	0.140
180	-1.21	-2.74	0.084
200	-1.45	-3.19	0.058
220	-1.63	-3.53	0.044

without momentum term

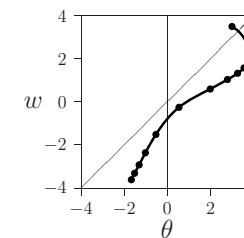
epoch	θ	w	error
0	3.00	3.50	1.295
10	3.80	2.19	0.984
20	3.75	1.84	0.971
30	3.56	1.58	0.960
40	3.26	1.33	0.943
50	2.79	1.04	0.910
60	1.99	0.60	0.814
70	0.54	-0.25	0.497
80	-0.53	-1.51	0.211
90	-1.02	-2.36	0.113
100	-1.31	-2.92	0.073
110	-1.52	-3.31	0.053
120	-1.67	-3.61	0.041

with momentum term

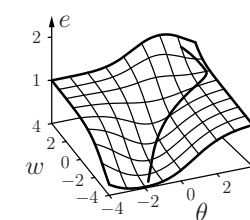
Gradient Descent: Examples



without momentum term



with momentum term



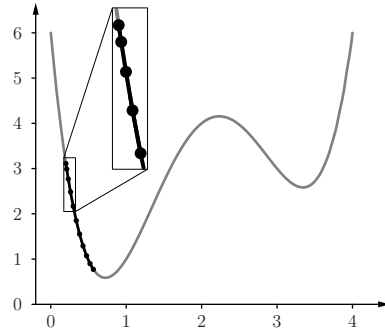
with momentum term

- 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		

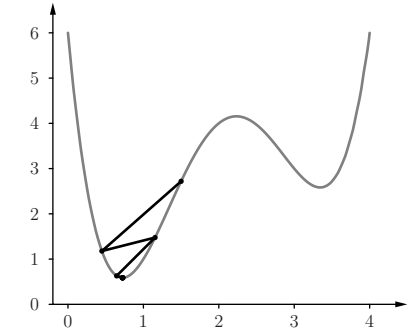


gradient descent with momentum term ($\beta = 0.9$)

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 ($c^+ = 1.2, c^- = 0.5$).

Other Extensions of Error Backpropagation

Flat Spot Elimination:

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

- Eliminates slow learning in saturation region of logistic function.
- Counteracts the decay of the error signals over the layers.

Weight Decay:

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

- Helps to improve the robustness of the training results.
- Can be derived from an extended error function penalizing large weights:

$$e^* = e + \frac{\xi}{2} \sum_{u \in U_{\text{out}} \cup U_{\text{hidden}}} \left(\theta_u^2 + \sum_{p \in \text{pred}(u)} w_{up}^2 \right).$$

Sensitivity Analysis

Sensitivity Analysis

Question: How important are different inputs to the network?

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

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

Formal derivation: Apply chain rule.

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

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:

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

This leads to the recursion formula

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

However, for the first hidden layer we get

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

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.

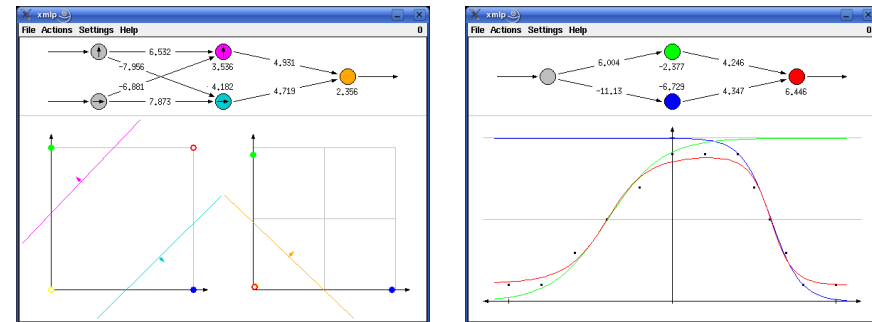
The recursion formula is in this case

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

and the anchor of the recursion is

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

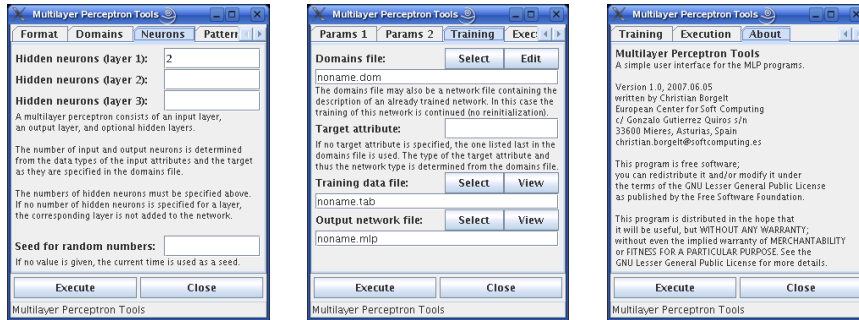
Demonstration Software: xmlp/wmlp



Demonstration of multilayer perceptron training:

- Visualization of the training process
- Bimplication and Exclusive Or, two continuous functions
- <http://www.borgelt.net/mlpd.html>

Multilayer Perceptron Software: mlp/mlpgui



Software for training general multilayer perceptrons:

- Command line version written in C, fast training
- Graphical user interface in Java, easy to use
- <http://www.borgelt.net/mlp.html>, <http://www.borgelt.net/mlpgui.html>

Radial Basis Function Networks

Radial Basis Function Networks

A **radial basis function network (RBFN)** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions

- $U_{\text{in}} \cap U_{\text{out}} = \emptyset$,
- $C = (U_{\text{in}} \times U_{\text{hidden}}) \cup C'$, $C' \subseteq (U_{\text{hidden}} \times U_{\text{out}})$

The network input function of each hidden neuron is a **distance function** of the input vector and the weight vector, i.e.

$$\forall u \in U_{\text{hidden}} : f_{\text{net}}^{(u)}(\vec{w}_u, \vec{m}_u) = d(\vec{w}_u, \vec{m}_u),$$

where $d : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}_0^+$ is a function satisfying $\forall \vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^n$:

- $d(\vec{x}, \vec{y}) = 0 \Leftrightarrow \vec{x} = \vec{y}$,
- $d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x})$ (symmetry),
- $d(\vec{x}, \vec{z}) \leq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z})$ (triangle inequality).

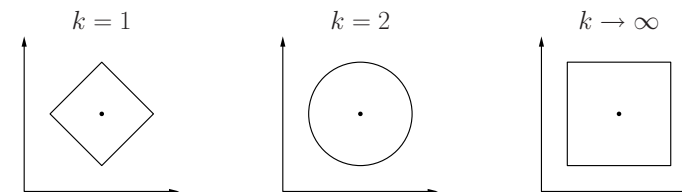
Distance Functions

Illustration of distance functions

$$d_k(\vec{x}, \vec{y}) = \left(\sum_{i=1}^n (x_i - y_i)^k \right)^{\frac{1}{k}}$$

Well-known special cases from this family are:

- $k = 1$: Manhattan or city block distance,
- $k = 2$: Euclidean distance,
- $k \rightarrow \infty$: maximum distance, i.e. $d_{\infty}(\vec{x}, \vec{y}) = \max_{i=1}^n |x_i - y_i|$.



Radial Basis Function Networks

The network input function of the output neurons is the weighted sum of their inputs, i.e.

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

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

$$f : \mathbb{R}_0^+ \rightarrow [0, 1] \text{ with } f(0) = 1 \text{ and } \lim_{x \rightarrow \infty} f(x) = 0.$$

The activation function of each output neuron is a linear function, namely

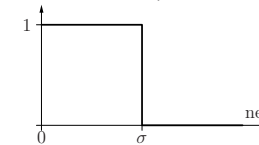
$$f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \text{net}_u - \theta_u.$$

(The linear activation function is important for the initialization.)

Radial Activation Functions

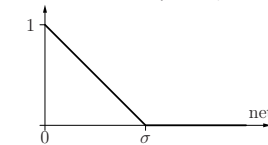
rectangle function:

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



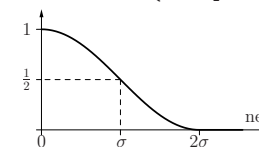
triangle function:

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



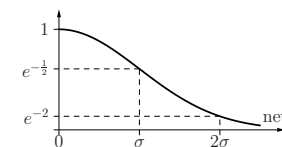
cosine until zero:

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



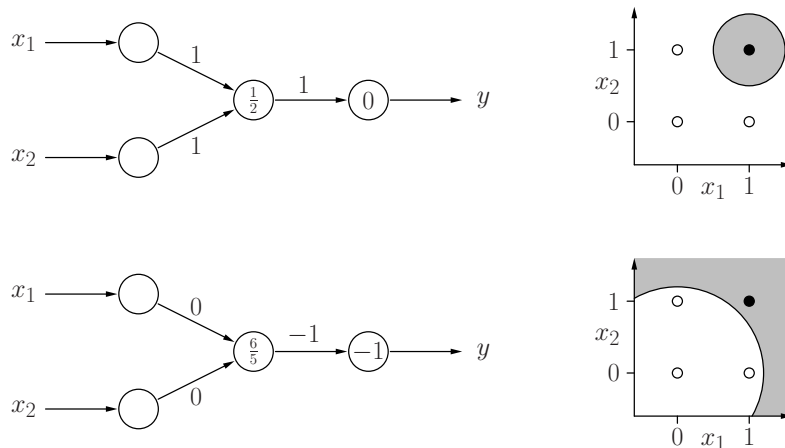
Gaussian function:

$$f_{\text{act}}(\text{net}, \sigma) = e^{-\frac{\text{net}^2}{2\sigma^2}}$$



Radial Basis Function Networks: Examples

Radial basis function networks for the conjunction $x_1 \wedge x_2$

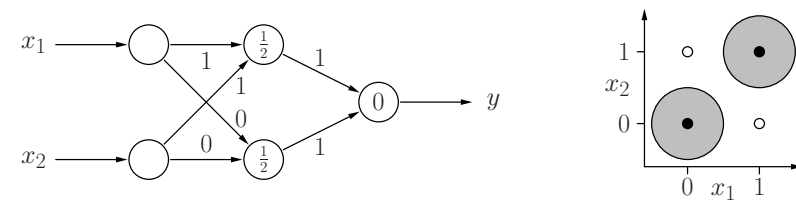


Radial Basis Function Networks: Examples

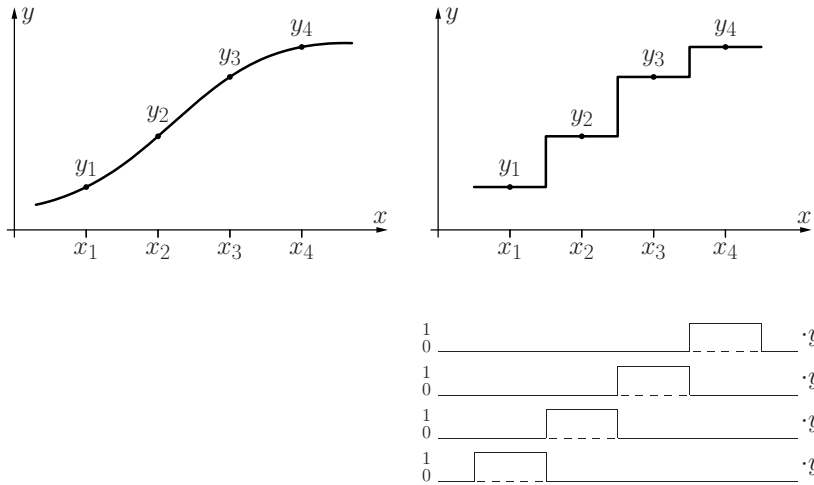
Radial basis function networks for the bimplication $x_1 \leftrightarrow x_2$

Idea: logical decomposition

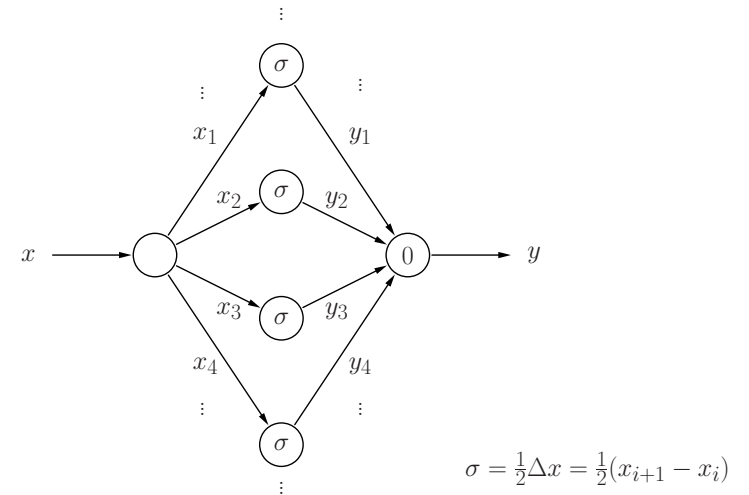
$$x_1 \leftrightarrow x_2 \equiv (x_1 \wedge x_2) \vee \neg(x_1 \vee x_2)$$



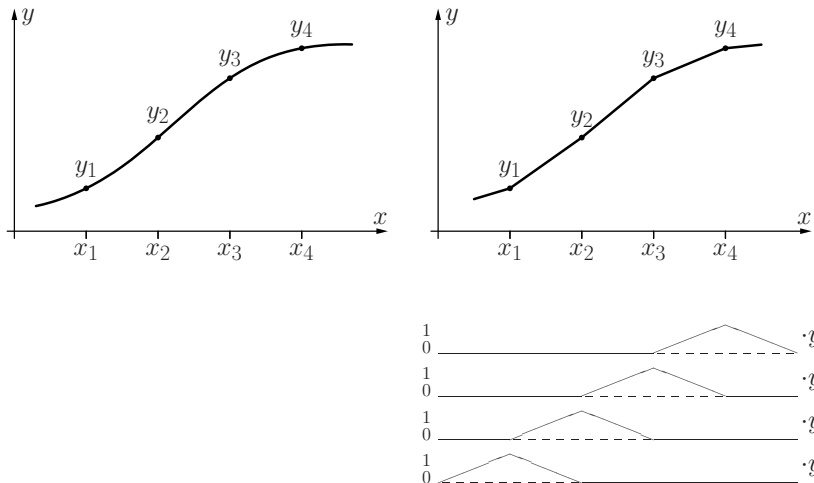
Radial Basis Function Networks: Function Approximation



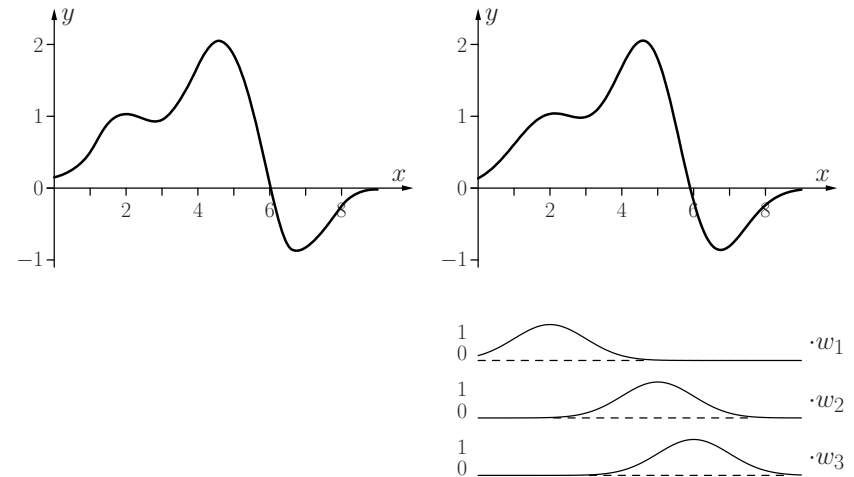
Radial Basis Function Networks: Function Approximation



Radial Basis Function Networks: Function Approximation

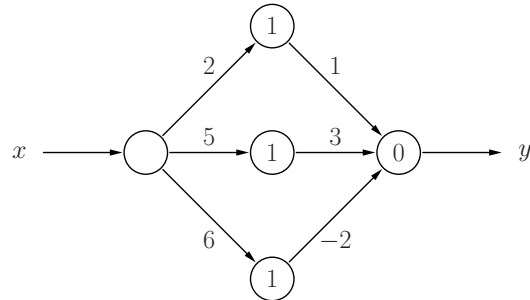


Radial Basis Function Networks: Function Approximation



Radial Basis Function Networks: Function Approximation

Radial basis function network for a sum of three Gaussian functions



Training Radial Basis Function Networks

Radial Basis Function Networks: Initialization

Let $L_{\text{fixed}} = \{l_1, \dots, l_m\}$ be a fixed learning task, consisting of m training patterns $l = (\vec{v}^{(l)}, \vec{o}^{(l)})$.

Simple radial basis function network:

One hidden neuron v_k , $k = 1, \dots, m$, for each training pattern:

$$\forall k \in \{1, \dots, m\} : \quad \vec{w}_{v_k} = \vec{v}^{(l_k)}.$$

If the activation function is the Gaussian function, the radii σ_k are chosen heuristically

$$\forall k \in \{1, \dots, m\} : \quad \sigma_k = \frac{d_{\max}}{\sqrt{2m}},$$

where

$$d_{\max} = \max_{l_j, l_k \in L_{\text{fixed}}} d(\vec{v}^{(l_j)}, \vec{v}^{(l_k)}).$$

Radial Basis Function Networks: Initialization

Initializing the connections from the hidden to the output neurons

$$\forall u : \sum_{k=1}^m w_{uv_m} \text{out}_{v_m}^{(l)} - \theta_u = o_u^{(l)} \quad \text{or abbreviated} \quad \mathbf{A} \cdot \vec{w}_u = \vec{o}_u,$$

where $\vec{o}_u = (o_u^{(l_1)}, \dots, o_u^{(l_m)})^\top$ is the vector of desired outputs, $\theta_u = 0$, and

$$\mathbf{A} = \begin{pmatrix} \text{out}_{v_1}^{(l_1)} & \text{out}_{v_2}^{(l_1)} & \dots & \text{out}_{v_m}^{(l_1)} \\ \text{out}_{v_1}^{(l_2)} & \text{out}_{v_2}^{(l_2)} & \dots & \text{out}_{v_m}^{(l_2)} \\ \vdots & \vdots & \ddots & \vdots \\ \text{out}_{v_1}^{(l_m)} & \text{out}_{v_2}^{(l_m)} & \dots & \text{out}_{v_m}^{(l_m)} \end{pmatrix}.$$

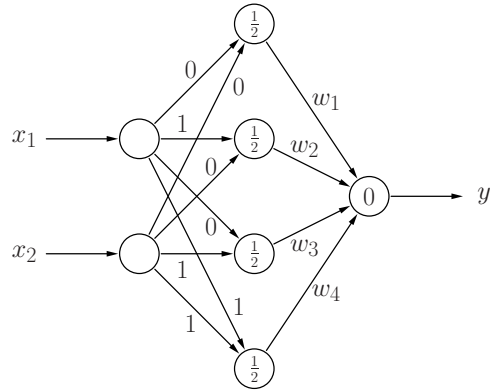
This is a linear equation system, that can be solved by inverting the matrix \mathbf{A} :

$$\vec{w}_u = \mathbf{A}^{-1} \cdot \vec{o}_u.$$

RBFN Initialization: Example

Simple radial basis function network for the biimplication $x_1 \leftrightarrow x_2$

x_1	x_2	y
0	0	1
1	0	0
0	1	0
1	1	1



RBFN Initialization: Example

Simple radial basis function network for the biimplication $x_1 \leftrightarrow x_2$

$$\mathbf{A} = \begin{pmatrix} 1 & e^{-2} & e^{-2} & e^{-4} \\ e^{-2} & 1 & e^{-4} & e^{-2} \\ e^{-2} & e^{-4} & 1 & e^{-2} \\ e^{-4} & e^{-2} & e^{-2} & 1 \end{pmatrix} \quad \mathbf{A}^{-1} = \begin{pmatrix} \frac{a}{D} & \frac{b}{D} & \frac{b}{D} & \frac{c}{D} \\ \frac{b}{D} & \frac{a}{D} & \frac{c}{D} & \frac{b}{D} \\ \frac{b}{D} & \frac{c}{D} & \frac{a}{D} & \frac{b}{D} \\ \frac{c}{D} & \frac{b}{D} & \frac{b}{D} & \frac{a}{D} \end{pmatrix}$$

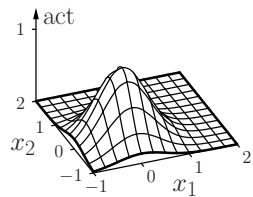
where

$$\begin{aligned} D &= 1 - 4e^{-4} + 6e^{-8} - 4e^{-12} + e^{-16} \approx 0.9287 \\ a &= 1 - 2e^{-4} + e^{-8} \approx 0.9637 \\ b &= -e^{-2} + 2e^{-6} - e^{-10} \approx -0.1304 \\ c &= e^{-4} - 2e^{-8} + e^{-12} \approx 0.0177 \end{aligned}$$

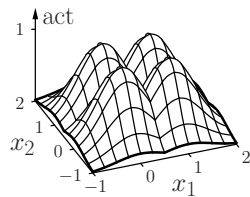
$$\vec{w}_u = \mathbf{A}^{-1} \cdot \vec{o}_u = \frac{1}{D} \begin{pmatrix} a + c \\ 2b \\ 2b \\ a + c \end{pmatrix} \approx \begin{pmatrix} 1.0567 \\ -0.2809 \\ -0.2809 \\ 1.0567 \end{pmatrix}$$

RBFN Initialization: Example

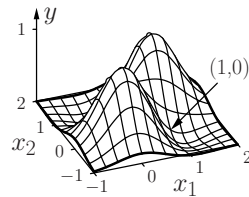
Simple radial basis function network for the biimplication $x_1 \leftrightarrow x_2$



single basis function



all basis functions



output

- Initialization leads already to a perfect solution of the learning task.
- Subsequent training is not necessary.

Radial Basis Function Networks: Initialization

Normal radial basis function networks:

Select subset of k training patterns as centers.

$$\mathbf{A} = \begin{pmatrix} 1 & \text{out}_{v_1}^{(l_1)} & \text{out}_{v_2}^{(l_1)} & \dots & \text{out}_{v_k}^{(l_1)} \\ 1 & \text{out}_{v_1}^{(l_2)} & \text{out}_{v_2}^{(l_2)} & \dots & \text{out}_{v_k}^{(l_2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \text{out}_{v_1}^{(l_m)} & \text{out}_{v_2}^{(l_m)} & \dots & \text{out}_{v_k}^{(l_m)} \end{pmatrix} \quad \mathbf{A} \cdot \vec{w}_u = \vec{o}_u$$

Compute (Moore–Penrose) pseudo inverse:

$$\mathbf{A}^+ = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top.$$

The weights can then be computed by

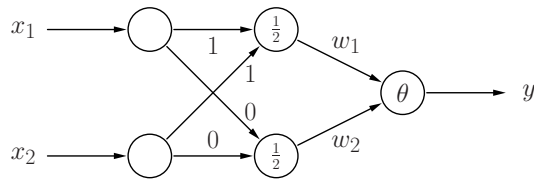
$$\vec{w}_u = \mathbf{A}^+ \cdot \vec{o}_u = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \cdot \vec{o}_u$$

RBFN Initialization: Example

Normal radial basis function network for the biimplication $x_1 \leftrightarrow x_2$

Select two training patterns:

- $l_1 = (\vec{z}^{(l_1)}, \vec{\sigma}^{(l_1)}) = ((0, 0), (1))$
- $l_4 = (\vec{z}^{(l_4)}, \vec{\sigma}^{(l_4)}) = ((1, 1), (1))$



RBFN Initialization: Example

Normal radial basis function network for the biimplication $x_1 \leftrightarrow x_2$

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & e^{-4} \\ 1 & e^{-2} & e^{-2} \\ 1 & e^{-2} & e^{-2} \\ 1 & e^{-4} & 1 \end{pmatrix} \quad \mathbf{A}^+ = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top = \begin{pmatrix} a & b & b & a \\ c & d & d & e \\ e & d & d & c \end{pmatrix}$$

where

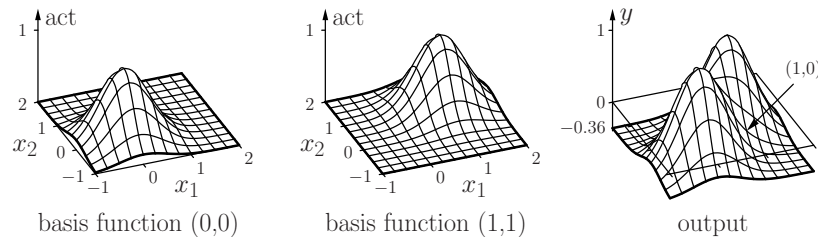
$$\begin{aligned} a &\approx -0.1810, & b &\approx 0.6810, \\ c &\approx 1.1781, & d &\approx -0.6688, & e &\approx 0.1594. \end{aligned}$$

Resulting weights:

$$\vec{w}_u = \begin{pmatrix} -\theta \\ w_1 \\ w_2 \end{pmatrix} = \mathbf{A}^+ \cdot \vec{\sigma}_u \approx \begin{pmatrix} -0.3620 \\ 1.3375 \\ 1.3375 \end{pmatrix}.$$

RBFN Initialization: Example

Normal radial basis function network for the biimplication $x_1 \leftrightarrow x_2$



- Initialization leads already to a perfect solution of the learning task.
- This is an accident, because the linear equation system is not over-determined, due to linearly dependent equations.

Radial Basis Function Networks: Initialization

Finding appropriate centers for the radial basis functions

One approach: **k-means clustering**

- Select randomly k training patterns as centers.
- Assign to each center those training patterns that are closest to it.
- Compute new centers as the center of gravity of the assigned training patterns
- Repeat previous two steps until convergence, i.e., until the centers do not change anymore.
- Use resulting centers for the weight vectors of the hidden neurons.

Alternative approach: **learning vector quantization**

Radial Basis Function Networks: Training

Training radial basis function networks:

Derivation of update rules is analogous to that of multilayer perceptrons.

Weights from the hidden to the output neurons.

Gradient:

$$\vec{\nabla}_{\vec{w}_u} e_u^{(l)} = \frac{\partial e_u^{(l)}}{\partial \vec{w}_u} = -2(o_u^{(l)} - \text{out}_u^{(l)}) \vec{\text{in}}_u^{(l)},$$

Weight update rule:

$$\Delta \vec{w}_u^{(l)} = -\frac{\eta_3}{2} \vec{\nabla}_{\vec{w}_u} e_u^{(l)} = \eta_3 (o_u^{(l)} - \text{out}_u^{(l)}) \vec{\text{in}}_u^{(l)}$$

(Two more learning rates are needed for the center coordinates and the radii.)

Radial Basis Function Networks: Training

Training radial basis function networks:

Center coordinates (weights from the input to the hidden neurons).

Gradient:

$$\vec{\nabla}_{\vec{w}_v} e^{(l)} = \frac{\partial e^{(l)}}{\partial \vec{w}_v} = -2 \sum_{s \in \text{succ}(v)} (o_s^{(l)} - \text{out}_s^{(l)}) w_{sv} \frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial \text{net}_v^{(l)}}{\partial \vec{w}_v}$$

Weight update rule:

$$\Delta \vec{w}_v^{(l)} = -\frac{\eta_1}{2} \vec{\nabla}_{\vec{w}_v} e^{(l)} = \eta_1 \sum_{s \in \text{succ}(v)} (o_s^{(l)} - \text{out}_s^{(l)}) w_{sv} \frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial \text{net}_v^{(l)}}{\partial \vec{w}_v}$$

Radial Basis Function Networks: Training

Training radial basis function networks:

Center coordinates (weights from the input to the hidden neurons).

Special case: **Euclidean distance**

$$\frac{\partial \text{net}_v^{(l)}}{\partial \vec{w}_v} = \left(\sum_{i=1}^n (w_{vp_i} - \text{out}_{p_i}^{(l)})^2 \right)^{-\frac{1}{2}} (\vec{w}_v - \vec{\text{in}}_v^{(l)}).$$

Special case: **Gaussian activation function**

$$\frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_v^{(l)}} = \frac{\partial f_{\text{act}}(\text{net}_v^{(l)}, \sigma_v)}{\partial \text{net}_v^{(l)}} = \frac{\partial}{\partial \text{net}_v^{(l)}} e^{-\frac{(\text{net}_v^{(l)})^2}{2\sigma_v^2}} = -\frac{\text{net}_v^{(l)}}{\sigma_v^2} e^{-\frac{(\text{net}_v^{(l)})^2}{2\sigma_v^2}}.$$

Radial Basis Function Networks: Training

Training radial basis function networks:

Radii of radial basis functions.

Gradient:

$$\frac{\partial e^{(l)}}{\partial \sigma_v} = -2 \sum_{s \in \text{succ}(v)} (o_s^{(l)} - \text{out}_s^{(l)}) w_{sv} \frac{\partial \text{out}_v^{(l)}}{\partial \sigma_v}.$$

Weight update rule:

$$\Delta \sigma_v^{(l)} = -\frac{\eta_2}{2} \frac{\partial e^{(l)}}{\partial \sigma_v} = \eta_2 \sum_{s \in \text{succ}(v)} (o_s^{(l)} - \text{out}_s^{(l)}) w_{sv} \frac{\partial \text{out}_v^{(l)}}{\partial \sigma_v}.$$

Special case: **Gaussian activation function**

$$\frac{\partial \text{out}_v^{(l)}}{\partial \sigma_v} = \frac{\partial}{\partial \sigma_v} e^{-\frac{(\text{net}_v^{(l)})^2}{2\sigma_v^2}} = \frac{(\text{net}_v^{(l)})^2}{\sigma_v^3} e^{-\frac{(\text{net}_v^{(l)})^2}{2\sigma_v^2}}.$$

Radial Basis Function Networks: Generalization

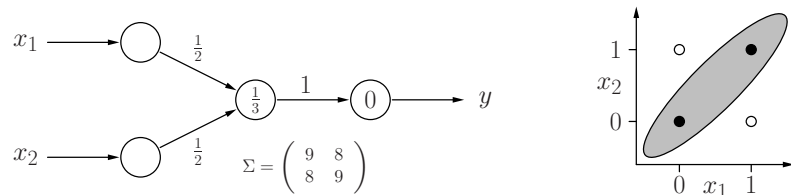
Generalization of the distance function

Idea: Use anisotropic distance function.

Example: **Mahalanobis distance**

$$d(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^\top \Sigma^{-1} (\vec{x} - \vec{y})}$$

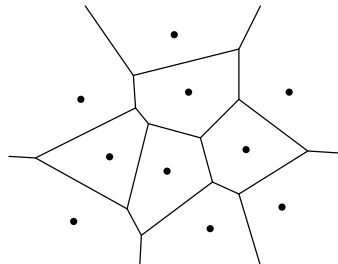
Example: **biimplication**



Learning Vector Quantization

Vector Quantization

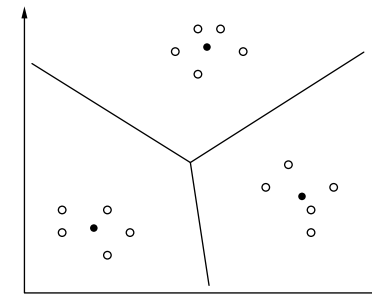
Voronoi diagram of a vector quantization



- Dots represent vectors that are used for quantizing the area.
- Lines are the boundaries of the regions of points that are closest to the enclosed vector.

Learning Vector Quantization

Finding clusters in a given set of data points



- Data points are represented by empty circles (○).
- Cluster centers are represented by full circles (●).

Learning Vector Quantization Networks

A **learning vector quantization network (LVQ)** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions

- (i) $U_{\text{in}} \cap U_{\text{out}} = \emptyset$, $U_{\text{hidden}} = \emptyset$
- (ii) $C = U_{\text{in}} \times U_{\text{out}}$

The network input function of each output neuron is a **distance function** of the input vector and the weight vector, i.e.

$$\forall u \in U_{\text{out}} : f_{\text{net}}^{(u)}(\vec{w}_u, \vec{in}_u) = d(\vec{w}_u, \vec{in}_u),$$

where $d : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}_0^+$ is a function satisfying $\forall \vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^n$:

- (i) $d(\vec{x}, \vec{y}) = 0 \Leftrightarrow \vec{x} = \vec{y}$,
- (ii) $d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x})$ (symmetry),
- (iii) $d(\vec{x}, \vec{z}) \leq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z})$ (triangle inequality).

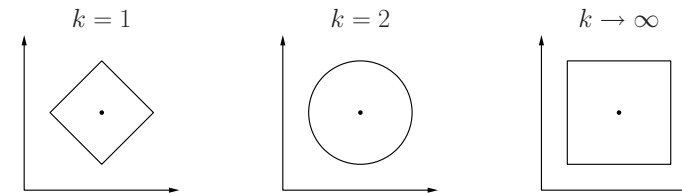
Distance Functions

Illustration of distance functions

$$d_k(\vec{x}, \vec{y}) = \left(\sum_{i=1}^n (x_i - y_i)^k \right)^{\frac{1}{k}}$$

Well-known special cases from this family are:

- $k = 1$: Manhattan or city block distance,
- $k = 2$: Euclidean distance,
- $k \rightarrow \infty$: maximum distance, i.e. $d_{\infty}(\vec{x}, \vec{y}) = \max_{i=1}^n |x_i - y_i|$.



Learning Vector Quantization

The activation function of each output neuron is a so-called **radial function**, i.e. a monotonously decreasing function

$$f : \mathbb{R}_0^+ \rightarrow [0, \infty] \text{ with } f(0) = 1 \text{ and } \lim_{x \rightarrow \infty} f(x) = 0.$$

Sometimes the range of values is restricted to the interval $[0, 1]$. However, due to the special output function this restriction is irrelevant.

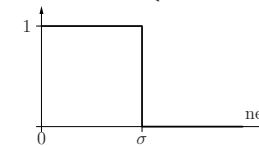
The output function of each output neuron is not a simple function of the activation of the neuron. Rather it takes into account the activations of all output neurons:

$$f_{\text{out}}^{(u)}(\text{act}_u) = \begin{cases} 1, & \text{if } \text{act}_u = \max_{v \in U_{\text{out}}} \text{act}_v, \\ 0, & \text{otherwise.} \end{cases}$$

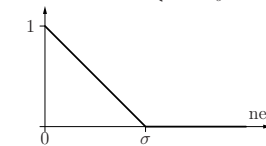
If more than one unit has the maximal activation, one is selected at random to have an output of 1, all others are set to output 0: **winner-takes-all principle**.

Radial Activation Functions

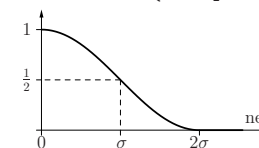
rectangle function:
 $f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{if } \text{net} > \sigma, \\ 1, & \text{otherwise.} \end{cases}$



triangle function:
 $f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{if } \text{net} > \sigma, \\ 1 - \frac{\text{net}}{\sigma}, & \text{otherwise.} \end{cases}$

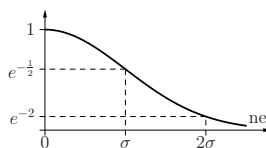


cosine until zero:
 $f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{if } \text{net} > 2\sigma, \\ \frac{\cos(\frac{\pi}{2\sigma} \text{net}) + 1}{2}, & \text{otherwise.} \end{cases}$



Gaussian function:

$$f_{\text{act}}(\text{net}, \sigma) = e^{-\frac{\text{net}^2}{2\sigma^2}}$$



Learning Vector Quantization

Adaptation of reference vectors / codebook vectors

- For each training pattern find the closest reference vector.
- Adapt only this reference vector (winner neuron).
- For classified data the class may be taken into account:
Each reference vector is assigned to a class.

Attraction rule (data point and reference vector have same class)

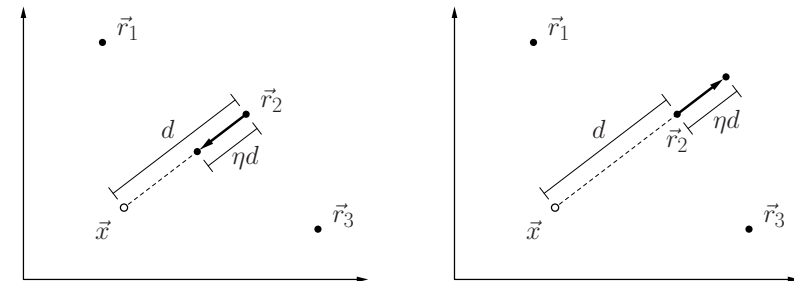
$$\vec{r}^{(\text{new})} = \vec{r}^{(\text{old})} + \eta(\vec{x} - \vec{r}^{(\text{old})}),$$

Repulsion rule (data point and reference vector have different class)

$$\vec{r}^{(\text{new})} = \vec{r}^{(\text{old})} - \eta(\vec{x} - \vec{r}^{(\text{old})}).$$

Learning Vector Quantization

Adaptation of reference vectors / codebook vectors



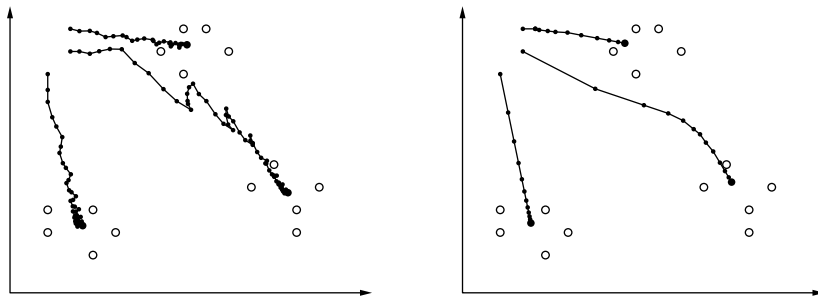
attraction rule

repulsion rule

- \vec{x} : data point, \vec{r}_i : reference vector
- $\eta = 0.4$ (learning rate)

Learning Vector Quantization: Example

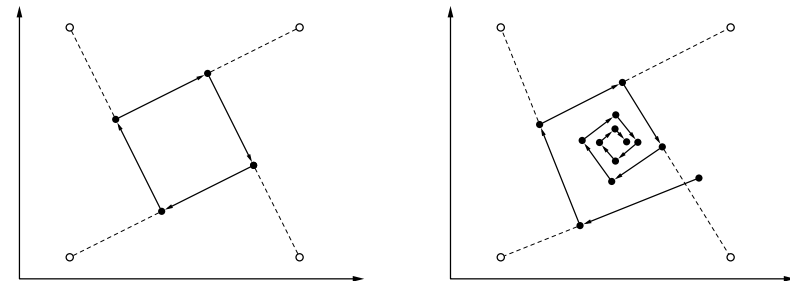
Adaptation of reference vectors / codebook vectors



- Left: Online training with learning rate $\eta = 0.1$,
- Right: Batch training with learning rate $\eta = 0.05$.

Learning Vector Quantization: Learning Rate Decay

Problem: fixed learning rate can lead to oscillations



Solution: time dependent learning rate

$$\eta(t) = \eta_0 \alpha^t, \quad 0 < \alpha < 1, \quad \text{or} \quad \eta(t) = \eta_0 t^\kappa, \quad \kappa > 0.$$

Learning Vector Quantization: Classified Data

Improved update rule for classified data

- **Idea:** Update not only the one reference vector that is closest to the data point (the winner neuron), but **update the two closest reference vectors**.
- Let \vec{x} be the currently processed data point and c its class. Let \vec{r}_j and \vec{r}_k be the two closest reference vectors and z_j and z_k their classes.
- Reference vectors are updated only if $z_j \neq z_k$ and either $c = z_j$ or $c = z_k$. (Without loss of generality we assume $c = z_j$.)
The **update rules** for the two closest reference vectors are:

$$\begin{aligned}\vec{r}_j^{(\text{new})} &= \vec{r}_j^{(\text{old})} + \eta(\vec{x} - \vec{r}_j^{(\text{old})}) & \text{and} \\ \vec{r}_k^{(\text{new})} &= \vec{r}_k^{(\text{old})} - \eta(\vec{x} - \vec{r}_k^{(\text{old})}),\end{aligned}$$

while all other reference vectors remain unchanged.

Learning Vector Quantization: Window Rule

- It was observed in practical tests that standard learning vector quantization may drive the reference vectors further and further apart.
- To counteract this undesired behavior a **window rule** was introduced: update only if the data point \vec{x} is close to the classification boundary.
- “Close to the boundary” is made formally precise by requiring

$$\min \left(\frac{d(\vec{x}, \vec{r}_j)}{d(\vec{x}, \vec{r}_k)}, \frac{d(\vec{x}, \vec{r}_k)}{d(\vec{x}, \vec{r}_j)} \right) > \theta, \quad \text{where} \quad \theta = \frac{1 - \xi}{1 + \xi}.$$

ξ is a parameter that has to be specified by a user.

- Intuitively, ξ describes the “width” of the window around the classification boundary, in which the data point has to lie in order to lead to an update.
- Using it prevents divergence, because the update ceases for a data point once the classification boundary has been moved far enough away.

Soft Learning Vector Quantization

- Idea:** Use soft assignments instead of winner-takes-all.
- Assumption:** Given data was sampled from a mixture of normal distributions. Each reference vector describes one normal distribution.
- Objective:** Maximize the log-likelihood ratio of the data, that is, maximize

$$\ln L_{\text{ratio}} = \sum_{j=1}^n \ln \sum_{\vec{r} \in R(c_j)} \exp \left(-\frac{(\vec{x}_j - \vec{r})^\top (\vec{x}_j - \vec{r})}{2\sigma^2} \right) - \sum_{j=1}^n \ln \sum_{\vec{r} \in Q(c_j)} \exp \left(-\frac{(\vec{x}_j - \vec{r})^\top (\vec{x}_j - \vec{r})}{2\sigma^2} \right).$$

Here σ is a parameter specifying the “size” of each normal distribution. $R(c)$ is the set of reference vectors assigned to class c and $Q(c)$ its complement.

Intuitively: at each data point the probability density for its class should be as large as possible while the density for all other classes should be as small as possible.

Soft Learning Vector Quantization

Update rule derived from a maximum log-likelihood approach:

$$\vec{r}_i^{(\text{new})} = \vec{r}_i^{(\text{old})} + \eta \cdot \begin{cases} u_{ij}^{\oplus} \cdot (\vec{x}_j - \vec{r}_i^{(\text{old})}), & \text{if } c_j = z_i, \\ -u_{ij}^{\ominus} \cdot (\vec{x}_j - \vec{r}_i^{(\text{old})}), & \text{if } c_j \neq z_i, \end{cases}$$

where z_i is the class associated with the reference vector \vec{r}_i and

$$\begin{aligned}u_{ij}^{\oplus} &= \frac{\exp \left(-\frac{1}{2\sigma^2} (\vec{x}_j - \vec{r}_i^{(\text{old})})^\top (\vec{x}_j - \vec{r}_i^{(\text{old})}) \right)}{\sum_{\vec{r} \in R(c_j)} \exp \left(-\frac{1}{2\sigma^2} (\vec{x}_j - \vec{r}^{(\text{old})})^\top (\vec{x}_j - \vec{r}^{(\text{old})}) \right)} & \text{and} \\ u_{ij}^{\ominus} &= \frac{\exp \left(-\frac{1}{2\sigma^2} (\vec{x}_j - \vec{r}_i^{(\text{old})})^\top (\vec{x}_j - \vec{r}_i^{(\text{old})}) \right)}{\sum_{\vec{r} \in Q(c_j)} \exp \left(-\frac{1}{2\sigma^2} (\vec{x}_j - \vec{r}^{(\text{old})})^\top (\vec{x}_j - \vec{r}^{(\text{old})}) \right)}.\end{aligned}$$

$R(c)$ is the set of reference vectors assigned to class c and $Q(c)$ its complement.

Hard Learning Vector Quantization

Idea: Derive a scheme with hard assignments from the soft version.

Approach: Let the size parameter σ of the Gaussian function go to zero.

The resulting update rule is in this case:

$$\vec{r}_i^{(\text{new})} = \vec{r}_i^{(\text{old})} + \eta \cdot \begin{cases} u_{ij}^{\oplus} \cdot (\vec{x}_j - \vec{r}_i^{(\text{old})}), & \text{if } c_j = z_i, \\ -u_{ij}^{\ominus} \cdot (\vec{x}_j - \vec{r}_i^{(\text{old})}), & \text{if } c_j \neq z_i, \end{cases}$$

where

$$u_{ij}^{\oplus} = \begin{cases} 1, & \text{if } \vec{r}_i = \operatorname{argmin}_{\vec{r} \in R(c_j)} d(\vec{x}_j, \vec{r}), \\ 0, & \text{otherwise,} \end{cases} \quad u_{ij}^{\ominus} = \begin{cases} 1, & \text{if } \vec{r}_i = \operatorname{argmin}_{\vec{r} \in Q(c_j)} d(\vec{x}_j, \vec{r}), \\ 0, & \text{otherwise.} \end{cases}$$

\vec{r}_i is closest vector of same class

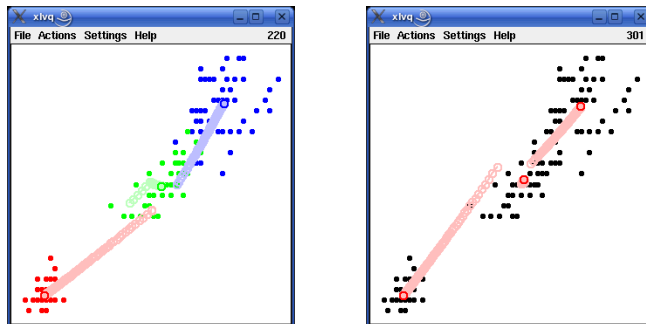
\vec{r}_i is closest vector of different class

This update rule is stable without a *window rule* restricting the update.

Learning Vector Quantization: Extensions

- **Frequency Sensitive Competitive Learning**
 - The distance to a reference vector is modified according to the number of data points that are assigned to this reference vector.
- **Fuzzy Learning Vector Quantization**
 - Exploits the close relationship to fuzzy clustering.
 - Can be seen as an online version of fuzzy clustering.
 - Leads to faster clustering.
- **Size and Shape Parameters**
 - Associate each reference vector with a cluster radius. Update this radius depending on how close the data points are.
 - Associate each reference vector with a covariance matrix. Update this matrix depending on the distribution of the data points.

Demonstration Software: xlvq/wlvq



Demonstration of learning vector quantization:

- Visualization of the training process
- Arbitrary datasets, but training only in two dimensions
- <http://www.borgelt.net/lvqd.html>

Self-Organizing Maps

Self-Organizing Maps

A **self-organizing map** or **Kohonen feature map** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions

- (i) $U_{\text{hidden}} = \emptyset, U_{\text{in}} \cap U_{\text{out}} = \emptyset,$
- (ii) $C = U_{\text{in}} \times U_{\text{out}}.$

The network input function of each output neuron is a **distance function** of input and weight vector. The activation function of each output neuron is a **radial function**, i.e. a monotonously decreasing function

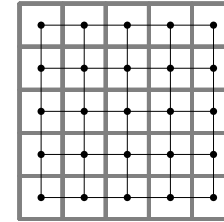
$$f : \mathbb{R}_0^+ \rightarrow [0, 1] \quad \text{with} \quad f(0) = 1 \quad \text{and} \quad \lim_{x \rightarrow \infty} f(x) = 0.$$

The output function of each output neuron is the identity.
The output is often discretized according to the “**winner takes all**” principle.
On the output neurons a **neighborhood relationship** is defined:

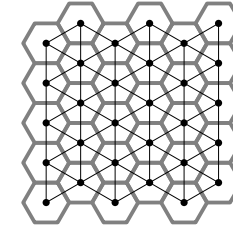
$$d_{\text{neurons}} : U_{\text{out}} \times U_{\text{out}} \rightarrow \mathbb{R}_0^+.$$

Self-Organizing Maps: Neighborhood

Neighborhood of the output neurons: neurons form a grid



quadratic grid



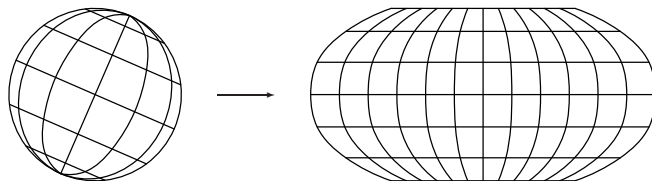
hexagonal grid

- Thin black lines: Indicate nearest neighbors of a neuron.
- Thick gray lines: Indicate regions assigned to a neuron for visualization.

Topology Preserving Mapping

Images of points close to each other in the original space should be close to each other in the image space.

Example: **Robinson projection** of the surface of a sphere



- Robinson projection is frequently used for world maps.

Self-Organizing Maps: Neighborhood

Find topology preserving mapping by respecting the neighborhood

Reference vector update rule:

$$\vec{r}_u^{(\text{new})} = \vec{r}_u^{(\text{old})} + \eta(t) \cdot f_{\text{nb}}(d_{\text{neurons}}(u, u_*), \varrho(t)) \cdot (\vec{x} - \vec{r}_u^{(\text{old})}),$$

- u_* is the winner neuron (reference vector closest to data point).
- The function f_{nb} is a radial function.

Time dependent learning rate

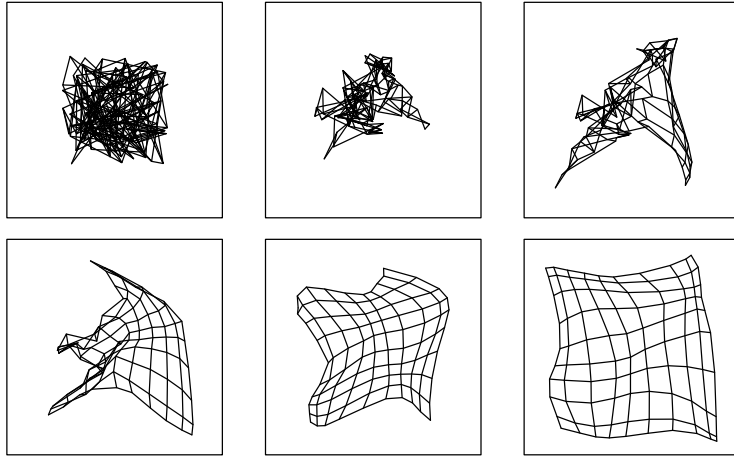
$$\eta(t) = \eta_0 \alpha_\eta^t, \quad 0 < \alpha_\eta < 1, \quad \text{or} \quad \eta(t) = \eta_0 t^{\kappa_\eta}, \quad \kappa_\eta > 0.$$

Time dependent neighborhood radius

$$\varrho(t) = \varrho_0 \alpha_\varrho^t, \quad 0 < \alpha_\varrho < 1, \quad \text{or} \quad \varrho(t) = \varrho_0 t^{\kappa_\varrho}, \quad \kappa_\varrho > 0.$$

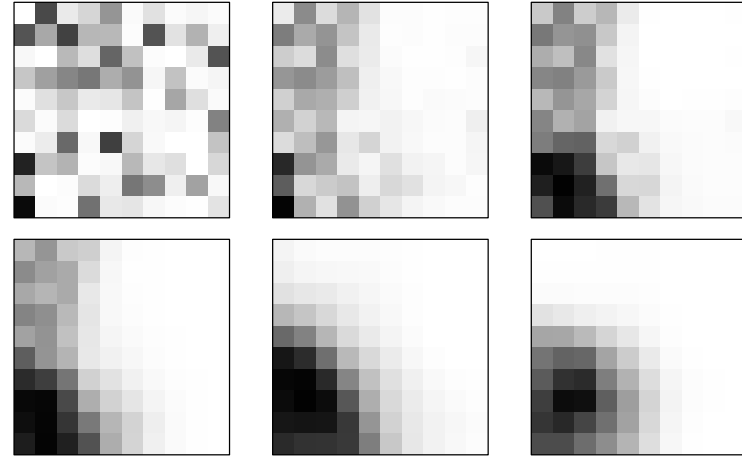
Self-Organizing Maps: Examples

Example: Unfolding of a two-dimensional self-organizing map.



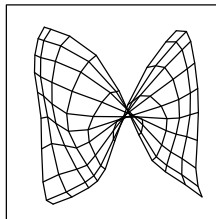
Self-Organizing Maps: Examples

Example: Unfolding of a two-dimensional self-organizing map.



Self-Organizing Maps: Examples

Example: Unfolding of a two-dimensional self-organizing map.

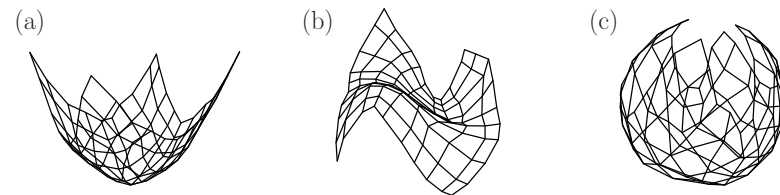


Training a self-organizing map may fail if

- the (initial) learning rate is chosen too small or
- or the (initial) neighbor is chosen too small.

Self-Organizing Maps: Examples

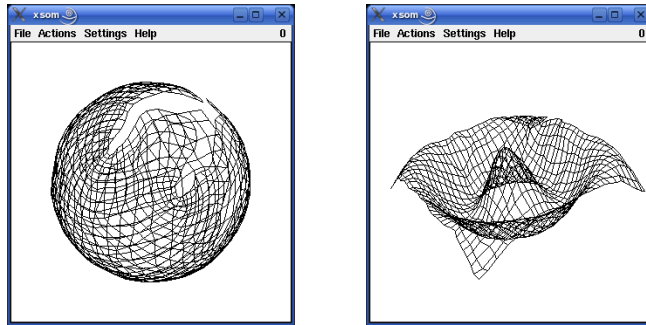
Example: Unfolding of a two-dimensional self-organizing map.



Self-organizing maps that have been trained with random points from
(a) a rotation parabola, (b) a simple cubic function, (c) the surface of a sphere.

- In this case original space and image space have different dimensionality.
- Self-organizing maps can be used for dimensionality reduction.

Demonstration Software: xsom/wsom



Demonstration of self-organizing map training:

- Visualization of the training process
- Two-dimensional areas and three-dimensional surfaces
- <http://www.borgelt.net/somd.html>

Hopfield Networks

Hopfield Networks

A **Hopfield network** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions:

- $U_{\text{hidden}} = \emptyset, U_{\text{in}} = U_{\text{out}} = U,$
- $C = U \times U - \{(u, u) \mid u \in U\}.$

- In a Hopfield network all neurons are input as well as output neurons.
- There are no hidden neurons.
- Each neuron receives input from all other neurons.
- A neuron is not connected to itself.

The connection weights are symmetric, i.e.

$$\forall u, v \in U, u \neq v : \quad w_{uv} = w_{vu}.$$

Hopfield Networks

The network input function of each neuron is the weighted sum of the outputs of all other neurons, i.e.

$$\forall u \in U : \quad f_{\text{net}}^{(u)}(\vec{w}_u, \vec{\text{in}}_u) = \vec{w}_u \vec{\text{in}}_u = \sum_{v \in U - \{u\}} w_{uv} \text{out}_v.$$

The activation function of each neuron is a threshold function, i.e.

$$\forall u \in U : \quad f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \begin{cases} 1, & \text{if } \text{net}_u \geq \theta, \\ -1, & \text{otherwise.} \end{cases}$$

The output function of each neuron is the identity, i.e.

$$\forall u \in U : \quad f_{\text{out}}^{(u)}(\text{act}_u) = \text{act}_u.$$

Hopfield Networks

Alternative activation function

$$\forall u \in U : f_{\text{act}}^{(u)}(\text{net}_u, \theta_u, \text{act}_u) = \begin{cases} 1, & \text{if } \text{net}_u > \theta, \\ -1, & \text{if } \text{net}_u < \theta, \\ \text{act}_u, & \text{if } \text{net}_u = \theta. \end{cases}$$

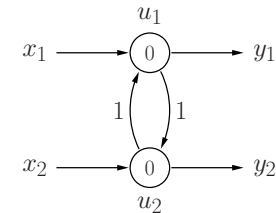
This activation function has advantages w.r.t. the physical interpretation of a Hopfield network.

General weight matrix of a Hopfield network

$$\mathbf{W} = \begin{pmatrix} 0 & w_{u_1 u_2} & \dots & w_{u_1 u_n} \\ w_{u_1 u_2} & 0 & \dots & w_{u_2 u_n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{u_1 u_n} & w_{u_1 u_n} & \dots & 0 \end{pmatrix}$$

Hopfield Networks: Examples

Very simple Hopfield network



$$\mathbf{W} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

The behavior of a Hopfield network can depend on the update order.

- Computations can oscillate if neurons are updated in parallel.
- Computations always converge if neurons are updated sequentially.

Hopfield Networks: Examples

Parallel update of neuron activations

	u_1	u_2
input phase	-1	1
work phase	1	-1
	-1	1
	1	-1
	-1	1
	1	-1
	-1	1

- The computations oscillate, no stable state is reached.
- Output depends on when the computations are terminated.

Hopfield Networks: Examples

Sequential update of neuron activations

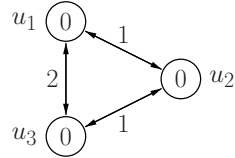
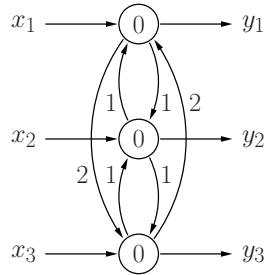
	u_1	u_2
input phase	-1	1
work phase	1	1
	1	1
	1	1
	1	1

	u_1	u_2
input phase	-1	1
work phase	-1	-1
	-1	-1
	-1	-1
	-1	-1

- Regardless of the update order a stable state is reached.
- Which state is reached depends on the update order.

Hopfield Networks: Examples

Simplified representation of a Hopfield network

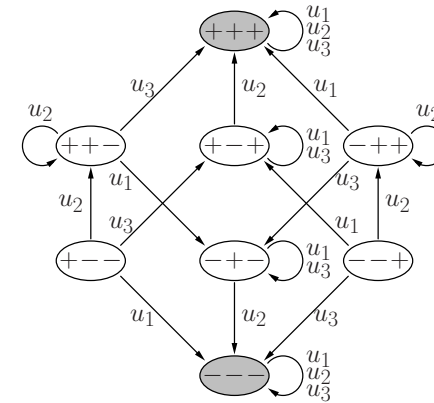


$$\mathbf{W} = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix}$$

- Symmetric connections between neurons are combined.
- Inputs and outputs are not explicitly represented.

Hopfield Networks: State Graph

Graph of activation states and transitions



Hopfield Networks: Convergence

Convergence Theorem: If the activations of the neurons of a Hopfield network are updated sequentially (asynchronously), then a stable state is reached in a finite number of steps.

If the neurons are traversed cyclically in an arbitrary, but fixed order, at most $n \cdot 2^n$ steps (updates of individual neurons) are needed, where n is the number of neurons of the Hopfield network.

The proof is carried out with the help of an **energy function**.

The energy function of a Hopfield network with n neurons u_1, \dots, u_n is

$$\begin{aligned} E &= -\frac{1}{2} \vec{\text{act}}^\top \mathbf{W} \vec{\text{act}} + \vec{\theta}^\top \vec{\text{act}} \\ &= -\frac{1}{2} \sum_{u,v \in U, u \neq v} w_{uv} \text{act}_u \text{act}_v + \sum_{u \in U} \theta_u \text{act}_u. \end{aligned}$$

Hopfield Networks: Convergence

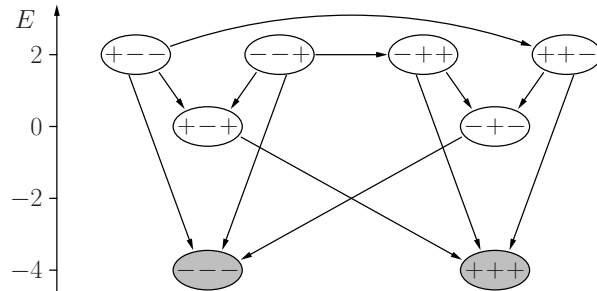
Consider the energy change resulting from an update that changes an activation:

$$\begin{aligned} \Delta E &= E^{(\text{new})} - E^{(\text{old})} = \left(- \sum_{v \in U - \{u\}} w_{uv} \text{act}_u^{(\text{new})} \text{act}_v + \theta_u \text{act}_u^{(\text{new})} \right) \\ &\quad - \left(- \sum_{v \in U - \{u\}} w_{uv} \text{act}_u^{(\text{old})} \text{act}_v + \theta_u \text{act}_u^{(\text{old})} \right) \\ &= \left(\text{act}_u^{(\text{old})} - \text{act}_u^{(\text{new})} \right) \underbrace{\left(\sum_{v \in U - \{u\}} w_{uv} \text{act}_v - \theta_u \right)}_{= \text{net}_u}. \end{aligned}$$

- $\text{net}_u < \theta_u$: Second factor is less than 0.
 $\text{act}_u^{(\text{new})} = -1$ and $\text{act}_u^{(\text{old})} = 1$, therefore first factor greater than 0.
Result: $\Delta E < 0$.
- $\text{net}_u \geq \theta_u$: Second factor greater than or equal to 0.
 $\text{act}_u^{(\text{new})} = 1$ and $\text{act}_u^{(\text{old})} = -1$, therefore first factor less than 0.
Result: $\Delta E \leq 0$.

Hopfield Networks: Examples

Arrange states in state graph according to their energy

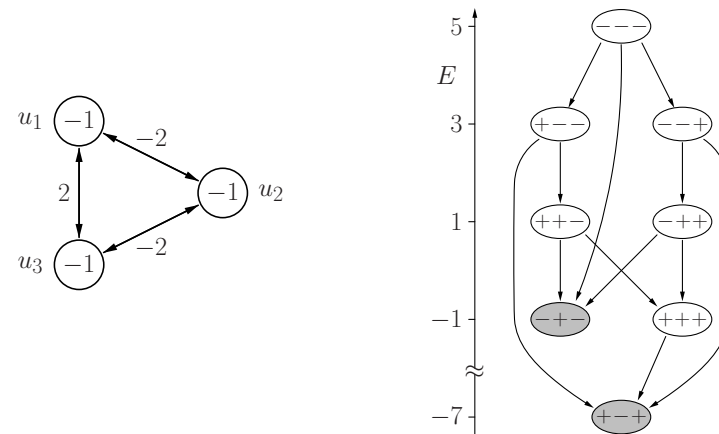


Energy function for example Hopfield network:

$$E = -\text{act}_{u_1} \text{act}_{u_2} - 2 \text{act}_{u_1} \text{act}_{u_3} - \text{act}_{u_2} \text{act}_{u_3} .$$

Hopfield Networks: Examples

The state graph need not be symmetric



Hopfield Networks: Physical Interpretation

Physical interpretation: Magnetism

A Hopfield network can be seen as a (microscopic) model of magnetism (so-called Ising model, [Ising 1925]).

physical	neural
atom	neuron
magnetic moment (spin)	activation state
strength of outer magnetic field	threshold value
magnetic coupling of the atoms	connection weights
Hamilton operator of the magnetic field	energy function

Hopfield Networks: Associative Memory

Idea: Use stable states to store patterns

First: Store only one pattern $\vec{x} = (\text{act}_{u_1}^{(l)}, \dots, \text{act}_{u_n}^{(l)})^\top \in \{-1, 1\}^n$, $n \geq 2$, i.e., find weights, so that pattern is a stable state.

Necessary and sufficient condition:

$$S(\mathbf{W}\vec{x} - \vec{\theta}) = \vec{x},$$

where

$$S: \mathbb{R}^n \rightarrow \{-1, 1\}^n, \\ \vec{x} \mapsto \vec{y}$$

with

$$\forall i \in \{1, \dots, n\}: y_i = \begin{cases} 1, & \text{if } x_i \geq 0, \\ -1, & \text{otherwise.} \end{cases}$$

Hopfield Networks: Associative Memory

If $\vec{\theta} = \vec{0}$ an appropriate matrix \mathbf{W} can easily be found. It suffices

$$\mathbf{W}\vec{x} = c\vec{x} \quad \text{with } c \in \mathbb{R}^+.$$

Algebraically: Find a matrix \mathbf{W} that has a positive eigenvalue w.r.t. \vec{x} .

Choose

$$\mathbf{W} = \vec{x}\vec{x}^T - \mathbf{E}$$

where $\vec{x}\vec{x}^T$ is the so-called **outer product**.

With this matrix we have

$$\begin{aligned} \mathbf{W}\vec{x} &= (\vec{x}\vec{x}^T)\vec{x} - \underbrace{\mathbf{E}\vec{x}}_{=\vec{x}} \stackrel{(*)}{=} \underbrace{\vec{x}(\vec{x}^T\vec{x})}_{=|\vec{x}|^2=n} - \vec{x} \\ &= n\vec{x} - \vec{x} = (n-1)\vec{x}. \end{aligned}$$

Hopfield Networks: Associative Memory

Hebbian learning rule [Hebb 1949]

Written in individual weights the computation of the weight matrix reads:

$$w_{uv} = \begin{cases} 0, & \text{if } u = v, \\ 1, & \text{if } u \neq v, \text{act}_u^{(p)} = \text{act}_v^{(p)}, \\ -1, & \text{otherwise.} \end{cases}$$

- Originally derived from a biological analogy.
- Strengthen connection between neurons that are active at the same time.

Note that this learning rule also stores the complement of the pattern:

$$\text{With } \mathbf{W}\vec{x} = (n-1)\vec{x} \quad \text{it is also } \mathbf{W}(-\vec{x}) = (n-1)(-\vec{x}).$$

Hopfield Networks: Associative Memory

Storing several patterns

Choose

$$\begin{aligned} \mathbf{W}\vec{x}_j &= \sum_{i=1}^m \mathbf{W}_i\vec{x}_j = \left(\sum_{i=1}^m (\vec{x}_i\vec{x}_i^T)\vec{x}_j \right) - m \underbrace{\mathbf{E}\vec{x}_j}_{=\vec{x}_j} \\ &= \left(\sum_{i=1}^m \vec{x}_i(\vec{x}_i^T\vec{x}_j) \right) - m\vec{x}_j \end{aligned}$$

If patterns are orthogonal, we have

$$\vec{x}_i^T\vec{x}_j = \begin{cases} 0, & \text{if } i \neq j, \\ n, & \text{if } i = j, \end{cases}$$

and therefore

$$\mathbf{W}\vec{x}_j = (n-m)\vec{x}_j.$$

Hopfield Networks: Associative Memory

Storing several patterns

Result: As long as $m < n$, \vec{x} is a stable state of the Hopfield network.

Note that the complements of the patterns are also stored.

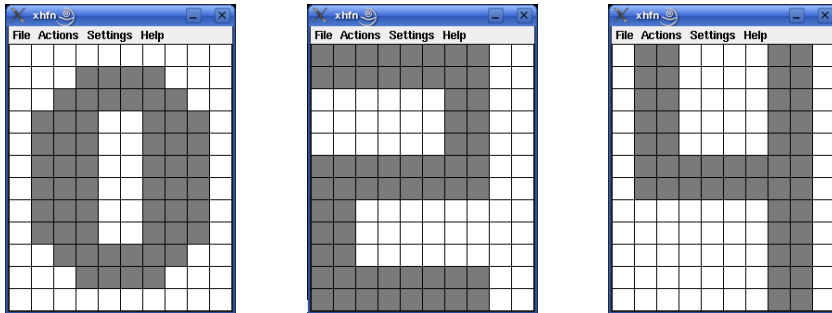
$$\text{With } \mathbf{W}\vec{x}_j = (n-m)\vec{x}_j \quad \text{it is also } \mathbf{W}(-\vec{x}_j) = (n-m)(-\vec{x}_j).$$

But: Capacity is very small compared to the number of possible states (2^n).

Non-orthogonal patterns:

$$\mathbf{W}\vec{x}_j = (n-m)\vec{x}_j + \underbrace{\sum_{\substack{i=1 \\ i \neq j}}^m \vec{x}_i(\vec{x}_i^T\vec{x}_j)}_{\text{"disturbance term"}}.$$

Demonstration Software: xhfn/whfn



Demonstration of Hopfield networks as associative memory:

- Visualization of the association/recognition process
- Two-dimensional networks of arbitrary size
- <http://www.borgelt.net/hfnd.html>

Hopfield Networks: Solving Optimization Problems

Use energy minimization to solve optimization problems

General procedure:

- Transform function to optimize into a function to minimize.
- Transform function into the form of an energy function of a Hopfield network.
- Read the weights and threshold values from the energy function.
- Construct the corresponding Hopfield network.
- Initialize Hopfield network randomly and update until convergence.
- Read solution from the stable state reached.
- Repeat several times and use best solution found.

Hopfield Networks: Activation Transformation

A Hopfield network may be defined either with activations -1 and 1 or with activations 0 and 1 . The networks can be transformed into each other.

From $\text{act}_u \in \{-1, 1\}$ to $\text{act}_u \in \{0, 1\}$:

$$\begin{aligned} w_{uv}^0 &= 2w_{uv}^- & \text{and} \\ \theta_u^0 &= \theta_u^- + \sum_{v \in U - \{u\}} w_{uv}^- \end{aligned}$$

From $\text{act}_u \in \{0, 1\}$ to $\text{act}_u \in \{-1, 1\}$:

$$\begin{aligned} w_{uv}^- &= \frac{1}{2}w_{uv}^0 & \text{and} \\ \theta_u^- &= \theta_u^0 - \frac{1}{2} \sum_{v \in U - \{u\}} w_{uv}^0. \end{aligned}$$

Hopfield Networks: Solving Optimization Problems

Combination lemma: Let two Hopfield networks on the same set U of neurons with weights $w_{uv}^{(i)}$, threshold values $\theta_u^{(i)}$ and energy functions

$$E_i = -\frac{1}{2} \sum_{u \in U} \sum_{v \in U - \{u\}} w_{uv}^{(i)} \text{act}_u \text{act}_v + \sum_{u \in U} \theta_u^{(i)} \text{act}_u,$$

$i = 1, 2$, be given. Furthermore let $a, b \in \mathbb{R}$. Then $E = aE_1 + bE_2$ is the energy function of the Hopfield network on the neurons in U that has the weights $w_{uv} = aw_{uv}^{(1)} + bw_{uv}^{(2)}$ and the threshold values $\theta_u = a\theta_u^{(1)} + b\theta_u^{(2)}$.

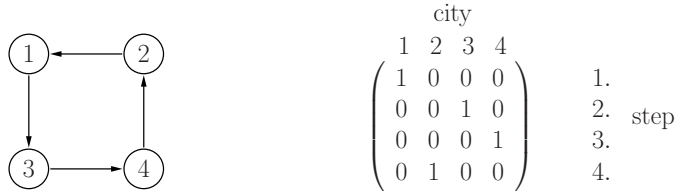
Proof: Just do the computations.

Idea: Additional conditions can be formalized separately and incorporated later.

Hopfield Networks: Solving Optimization Problems

Example: Traveling salesman problem

Idea: Represent tour by a matrix.



An element a_{ij} of the matrix is 1 if the i -th city is visited in the j -th step and 0 otherwise.

Each matrix element will be represented by a neuron.

Hopfield Networks: Solving Optimization Problems

Minimization of the tour length

$$E_1 = \sum_{j_1=1}^n \sum_{j_2=1}^n \sum_{i=1}^n d_{j_1 j_2} \cdot m_{i j_1} \cdot m_{(i \bmod n)+1, j_2}$$

Double summation over steps (index i) needed:

$$E_1 = \sum_{(i_1, j_1) \in \{1, \dots, n\}^2} \sum_{(i_2, j_2) \in \{1, \dots, n\}^2} d_{j_1 j_2} \cdot \delta_{(i_1 \bmod n)+1, i_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2},$$

where

$$\delta_{ab} = \begin{cases} 1, & \text{if } a = b, \\ 0, & \text{otherwise.} \end{cases}$$

Symmetric version of the energy function:

$$E_1 = -\frac{1}{2} \sum_{\substack{(i_1, j_1) \in \{1, \dots, n\}^2 \\ (i_2, j_2) \in \{1, \dots, n\}^2}} -d_{j_1 j_2} \cdot (\delta_{(i_1 \bmod n)+1, i_2} + \delta_{i_1, (i_2 \bmod n)+1}) \cdot m_{i_1 j_1} \cdot m_{i_2 j_2}$$

Hopfield Networks: Solving Optimization Problems

Additional conditions that have to be satisfied:

- Each city is visited on exactly one step of the tour:

$$\forall j \in \{1, \dots, n\} : \sum_{i=1}^n m_{ij} = 1,$$

i.e., each column of the matrix contains exactly one 1.

- On each step of the tour exactly one city is visited:

$$\forall i \in \{1, \dots, n\} : \sum_{j=1}^n m_{ij} = 1,$$

i.e., each row of the matrix contains exactly one 1.

These conditions are incorporated by finding additional functions to optimize.

Hopfield Networks: Solving Optimization Problems

Formalization of first condition as a minimization problem:

$$\begin{aligned} E_2^* &= \sum_{j=1}^n \left(\left(\sum_{i=1}^n m_{ij} \right)^2 - 2 \sum_{i=1}^n m_{ij} + 1 \right) \\ &= \sum_{j=1}^n \left(\left(\sum_{i_1=1}^n m_{i_1 j} \right) \left(\sum_{i_2=1}^n m_{i_2 j} \right) - 2 \sum_{i=1}^n m_{ij} + 1 \right) \\ &= \sum_{j=1}^n \sum_{i_1=1}^n \sum_{i_2=1}^n m_{i_1 j} m_{i_2 j} - 2 \sum_{j=1}^n \sum_{i=1}^n m_{ij} + n. \end{aligned}$$

Double summation over cities (index i) needed:

$$E_2 = \sum_{(i_1, j_1) \in \{1, \dots, n\}^2} \sum_{(i_2, j_2) \in \{1, \dots, n\}^2} \delta_{j_1 j_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2} - 2 \sum_{(i, j) \in \{1, \dots, n\}^2} m_{ij}.$$

Hopfield Networks: Solving Optimization Problems

Resulting energy function:

$$E_2 = -\frac{1}{2} \sum_{\substack{(i_1, j_1) \in \{1, \dots, n\}^2 \\ (i_2, j_2) \in \{1, \dots, n\}^2}} -2\delta_{j_1 j_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2} + \sum_{(i, j) \in \{1, \dots, n\}^2} -2m_{ij}$$

Second additional condition is handled in a completely analogous way:

$$E_3 = -\frac{1}{2} \sum_{\substack{(i_1, j_1) \in \{1, \dots, n\}^2 \\ (i_2, j_2) \in \{1, \dots, n\}^2}} -2\delta_{i_1 i_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2} + \sum_{(i, j) \in \{1, \dots, n\}^2} -2m_{ij}$$

Combining the energy functions:

$$E = aE_1 + bE_2 + cE_3 \quad \text{where} \quad \frac{b}{a} = \frac{c}{a} > 2 \quad \max_{(j_1, j_2) \in \{1, \dots, n\}^2} d_{j_1 j_2}.$$

Hopfield Networks: Solving Optimization Problems

From the resulting energy function we can read the weights

$$w_{(i_1, j_1)(i_2, j_2)} = \underbrace{-ad_{j_1 j_2}}_{\text{from } E_1} \cdot (\underbrace{\delta_{(i_1 \bmod n)+1, i_2} + \delta_{i_1, (i_2 \bmod n)+1}}_{\text{from } E_2}) \underbrace{-2b\delta_{j_1 j_2}}_{\text{from } E_2} \underbrace{-2c\delta_{i_1 i_2}}_{\text{from } E_3}$$

and the threshold values:

$$\theta_{(i, j)} = \underbrace{0a}_{\text{from } E_1} \quad \underbrace{-2b}_{\text{from } E_2} \quad \underbrace{-2c}_{\text{from } E_3} = -2(b + c).$$

Problem: Random initialization and update until convergence not always leads to a matrix that represents a tour, leave alone an optimal one.

Recurrent Neural Networks

Recurrent Networks: Cooling Law

A body of temperature ϑ_0 that is placed into an environment with temperature ϑ_A .

The cooling/heating of the body can be described by **Newton's cooling law**:

$$\frac{d\vartheta}{dt} = \dot{\vartheta} = -k(\vartheta - \vartheta_A).$$

Exact analytical solution:

$$\vartheta(t) = \vartheta_A + (\vartheta_0 - \vartheta_A)e^{-k(t-t_0)}$$

Approximate solution with **Euler-Cauchy polygon courses**:

$$\vartheta_1 = \vartheta(t_1) = \vartheta(t_0) + \dot{\vartheta}(t_0)\Delta t = \vartheta_0 - k(\vartheta_0 - \vartheta_A)\Delta t.$$

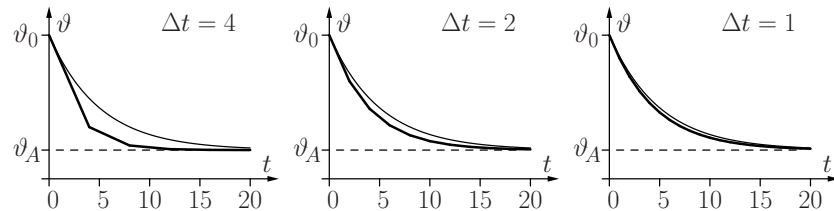
$$\vartheta_2 = \vartheta(t_2) = \vartheta(t_1) + \dot{\vartheta}(t_1)\Delta t = \vartheta_1 - k(\vartheta_1 - \vartheta_A)\Delta t.$$

General recursive formula:

$$\vartheta_i = \vartheta(t_i) = \vartheta(t_{i-1}) + \dot{\vartheta}(t_{i-1})\Delta t = \vartheta_{i-1} - k(\vartheta_{i-1} - \vartheta_A)\Delta t$$

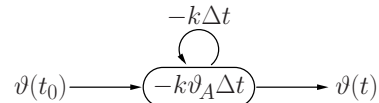
Recurrent Networks: Cooling Law

Euler–Cauchy polygon courses for different step widths:



The thin curve is the exact analytical solution.

Recurrent neural network:



Recurrent Networks: Cooling Law

More formal derivation of the recursive formula:

Replace differential quotient by **forward difference**

$$\frac{d\vartheta(t)}{dt} \approx \frac{\Delta\vartheta(t)}{\Delta t} = \frac{\vartheta(t + \Delta t) - \vartheta(t)}{\Delta t}$$

with sufficiently small Δt . Then it is

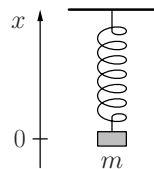
$$\vartheta(t + \Delta t) - \vartheta(t) = \Delta\vartheta(t) \approx -k(\vartheta(t) - \vartheta_A)\Delta t,$$

$$\vartheta(t + \Delta t) - \vartheta(t) = \Delta\vartheta(t) \approx -k\Delta t\vartheta(t) + k\vartheta_A\Delta t$$

and therefore

$$\vartheta_i \approx \vartheta_{i-1} - k\Delta t\vartheta_{i-1} + k\vartheta_A\Delta t.$$

Recurrent Networks: Mass on a Spring



Governing physical laws:

- **Hooke's law:** $F = c\Delta l = -cx$ (c is a spring dependent constant)
- **Newton's second law:** $F = ma = m\ddot{x}$ (force causes an acceleration)

Resulting differential equation:

$$m\ddot{x} = -cx \quad \text{or} \quad \ddot{x} = -\frac{c}{m}x.$$

Recurrent Networks: Mass on a Spring

General analytical solution of the differential equation:

$$x(t) = a \sin(\omega t) + b \cos(\omega t)$$

with the parameters

$$\omega = \sqrt{\frac{c}{m}}, \quad \begin{aligned} a &= x(t_0) \sin(\omega t_0) + v(t_0) \cos(\omega t_0), \\ b &= x(t_0) \cos(\omega t_0) - v(t_0) \sin(\omega t_0). \end{aligned}$$

With given initial values $x(t_0) = x_0$ and $v(t_0) = 0$ and the additional assumption $t_0 = 0$ we get the simple expression

$$x(t) = x_0 \cos\left(\sqrt{\frac{c}{m}} t\right).$$

Recurrent Networks: Mass on a Spring

Turn differential equation into two coupled equations:

$$\dot{x} = v \quad \text{and} \quad \dot{v} = -\frac{c}{m}x.$$

Approximate differential quotient by forward difference:

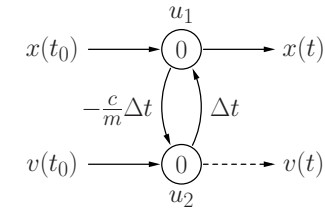
$$\frac{\Delta x}{\Delta t} = \frac{x(t + \Delta t) - x(t)}{\Delta t} = v \quad \text{and} \quad \frac{\Delta v}{\Delta t} = \frac{v(t + \Delta t) - v(t)}{\Delta t} = -\frac{c}{m}x$$

Resulting recursive equations:

$$x(t_i) = x(t_{i-1}) + \Delta x(t_{i-1}) = x(t_{i-1}) + \Delta t \cdot v(t_{i-1}) \quad \text{and}$$

$$v(t_i) = v(t_{i-1}) + \Delta v(t_{i-1}) = v(t_{i-1}) - \frac{c}{m}\Delta t \cdot x(t_{i-1}).$$

Recurrent Networks: Mass on a Spring



Neuron u_1 : $f_{\text{net}}^{(u_1)}(v, w_{u_1 u_2}) = w_{u_1 u_2} v = -\frac{c}{m}\Delta t v$ and

$$f_{\text{act}}^{(u_1)}(\text{act}_{u_1}, \text{net}_{u_1}, \theta_{u_1}) = \text{act}_{u_1} + \text{net}_{u_1} - \theta_{u_1},$$

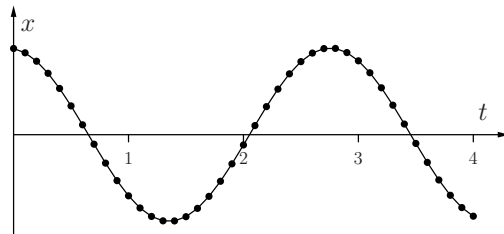
Neuron u_2 : $f_{\text{net}}^{(u_2)}(x, w_{u_2 u_1}) = w_{u_2 u_1} x = \Delta t x$ and

$$f_{\text{act}}^{(u_2)}(\text{act}_{u_2}, \text{net}_{u_2}, \theta_{u_2}) = \text{act}_{u_2} + \text{net}_{u_2} - \theta_{u_2}.$$

Recurrent Networks: Mass on a Spring

Some computation steps of the neural network:

t	v	x
0.0	0.0000	1.0000
0.1	-0.5000	0.9500
0.2	-0.9750	0.8525
0.3	-1.4012	0.7124
0.4	-1.7574	0.5366
0.5	-2.0258	0.3341
0.6	-2.1928	0.1148



- The resulting curve is close to the analytical solution.
- The approximation gets better with smaller step width.

Recurrent Networks: Differential Equations

General representation of explicit n -th order differential equation:

$$x^{(n)} = f(t, x, \dot{x}, \ddot{x}, \dots, x^{(n-1)})$$

Introduce $n - 1$ intermediary quantities

$$y_1 = \dot{x}, \quad y_2 = \ddot{x}, \quad \dots \quad y_{n-1} = x^{(n-1)}$$

to obtain the system

$$\begin{aligned} \dot{x} &= y_1, \\ \dot{y}_1 &= y_2, \\ &\vdots \\ \dot{y}_{n-2} &= y_{n-1}, \\ \dot{y}_{n-1} &= f(t, x, y_1, y_2, \dots, y_{n-1}) \end{aligned}$$

of n coupled first order differential equations.

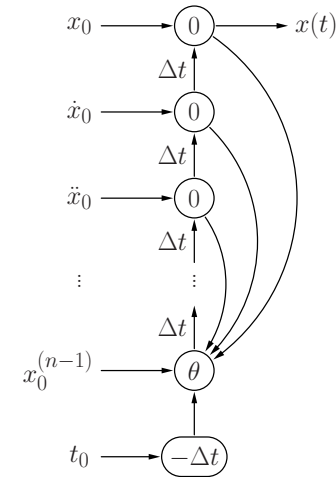
Recurrent Networks: Differential Equations

Replace differential quotient by forward distance to obtain the recursive equations

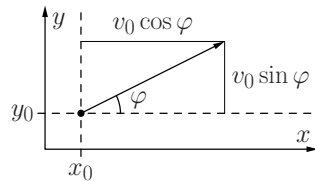
$$\begin{aligned} x(t_i) &= x(t_{i-1}) + \Delta t \cdot y_1(t_{i-1}), \\ y_1(t_i) &= y_1(t_{i-1}) + \Delta t \cdot y_2(t_{i-1}), \\ &\vdots \\ y_{n-2}(t_i) &= y_{n-2}(t_{i-1}) + \Delta t \cdot y_{n-3}(t_{i-1}), \\ y_{n-1}(t_i) &= y_{n-1}(t_{i-1}) + f(t_{i-1}, x(t_{i-1}), y_1(t_{i-1}), \dots, y_{n-1}(t_{i-1})) \end{aligned}$$

- Each of these equations describes the update of one neuron.
- The last neuron needs a special activation function.

Recurrent Networks: Differential Equations



Recurrent Networks: Diagonal Throw



Diagonal throw of a body.

Two differential equations (one for each coordinate):

$$\ddot{x} = 0 \quad \text{and} \quad \ddot{y} = -g,$$

where $g = 9.81 \text{ ms}^{-2}$.

Initial conditions $x(t_0) = x_0$, $y(t_0) = y_0$, $\dot{x}(t_0) = v_0 \cos \varphi$ and $\dot{y}(t_0) = v_0 \sin \varphi$.

Recurrent Networks: Diagonal Throw

Introduce intermediary quantities

$$v_x = \dot{x} \quad \text{and} \quad v_y = \dot{y}$$

to reach the system of differential equations:

$$\begin{aligned} \dot{x} &= v_x, & \dot{v}_x &= 0, \\ \dot{y} &= v_y, & \dot{v}_y &= -g, \end{aligned}$$

from which we get the system of recursive update formulae

$$\begin{aligned} x(t_i) &= x(t_{i-1}) + \Delta t v_x(t_{i-1}), & v_x(t_i) &= v_x(t_{i-1}), \\ y(t_i) &= y(t_{i-1}) + \Delta t v_y(t_{i-1}), & v_y(t_i) &= v_y(t_{i-1}) - \Delta t g. \end{aligned}$$

Recurrent Networks: Diagonal Throw

Better description: Use **vectors** as inputs and outputs

$$\ddot{\vec{r}} = -g\vec{e}_y,$$

where $\vec{e}_y = (0, 1)$.

Initial conditions are $\vec{r}(t_0) = \vec{r}_0 = (x_0, y_0)$ and $\dot{\vec{r}}(t_0) = \vec{v}_0 = (v_0 \cos \varphi, v_0 \sin \varphi)$.

Introduce one **vector-valued** intermediary quantity $\vec{v} = \dot{\vec{r}}$ to obtain

$$\dot{\vec{r}} = \vec{v}, \quad \dot{\vec{v}} = -g\vec{e}_y$$

This leads to the recursive update rules

$$\vec{r}(t_i) = \vec{r}(t_{i-1}) + \Delta t \vec{v}(t_{i-1}),$$

$$\vec{v}(t_i) = \vec{v}(t_{i-1}) - \Delta t g\vec{e}_y$$

Recurrent Networks: Diagonal Throw

Advantage of vector networks becomes obvious if friction is taken into account:

$$\vec{a} = -\beta\vec{v} = -\beta\dot{\vec{r}}$$

β is a constant that depends on the size and the shape of the body.

This leads to the differential equation

$$\ddot{\vec{r}} = -\beta\dot{\vec{r}} - g\vec{e}_y.$$

Introduce the intermediary quantity $\vec{v} = \dot{\vec{r}}$ to obtain

$$\dot{\vec{r}} = \vec{v}, \quad \dot{\vec{v}} = -\beta\vec{v} - g\vec{e}_y,$$

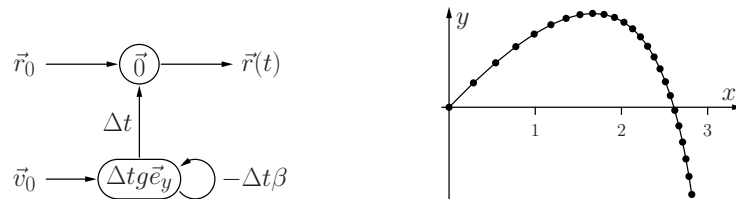
from which we obtain the recursive update formulae

$$\vec{r}(t_i) = \vec{r}(t_{i-1}) + \Delta t \vec{v}(t_{i-1}),$$

$$\vec{v}(t_i) = \vec{v}(t_{i-1}) - \Delta t \beta \vec{v}(t_{i-1}) - \Delta t g\vec{e}_y.$$

Recurrent Networks: Diagonal Throw

Resulting recurrent neural network:



- There are no strange couplings as there would be in a non-vector network.
- Note the deviation from a parabola that is due to the friction.

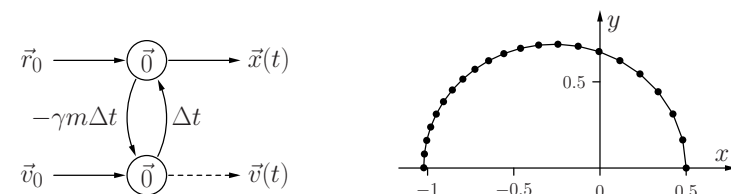
Recurrent Networks: Planet Orbit

$$\ddot{\vec{r}} = -\gamma m \frac{\vec{r}}{|\vec{r}|^3}, \quad \Rightarrow \quad \dot{\vec{r}} = \vec{v}, \quad \dot{\vec{v}} = -\gamma m \frac{\vec{r}}{|\vec{r}|^3}.$$

Recursive update rules:

$$\vec{r}(t_i) = \vec{r}(t_{i-1}) + \Delta t \vec{v}(t_{i-1}),$$

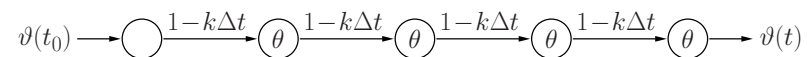
$$\vec{v}(t_i) = \vec{v}(t_{i-1}) - \Delta t \gamma m \frac{\vec{r}(t_{i-1})}{|\vec{r}(t_{i-1})|^3},$$



Recurrent Networks: Backpropagation through Time

Idea: Unfold the network between training patterns,
i.e., create one neuron for each point in time.

Example: **Newton's cooling law**



Unfolding into four steps. It is $\theta = -k\vartheta_A\Delta t$.

- Training is standard backpropagation on unfolded network.
- All updates refer to the same weight.
- updates are carried out after first neuron is reached.