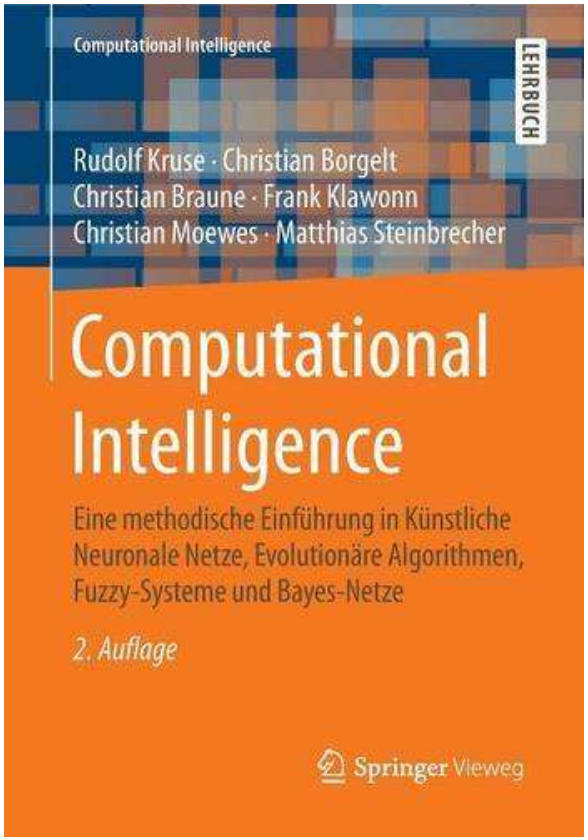
Artificial Neural Networks and Deep Learning

Christian Borgelt

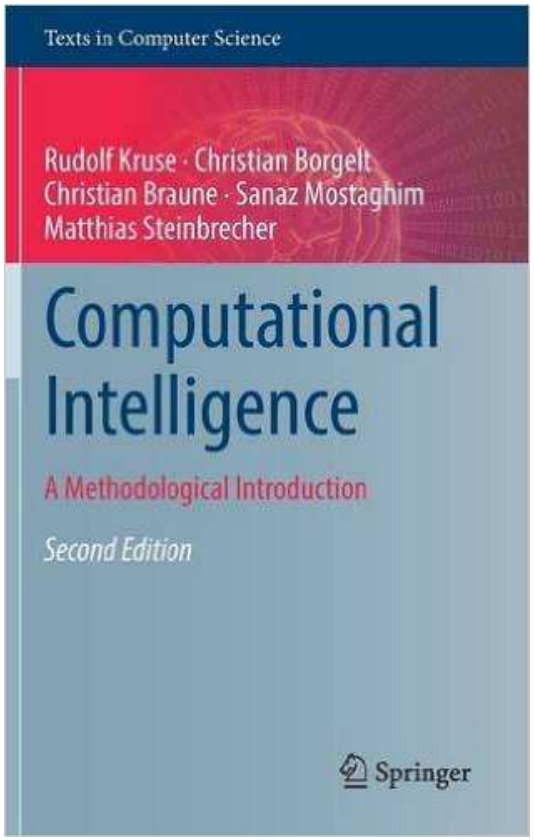
School of Computer Science
University of Konstanz
Universitätsstraße 10, 78457 Konstanz, Germany

christian.borgelt@uni-konstanz.de
christian@borgelt.net
http://www.borgelt.net/

Textbooks



Textbook, 2nd ed.
Springer-Verlag
Heidelberg, DE 2015
(in German)



Textbook, 2nd ed.
Springer-Verlag
Heidelberg, DE 2016
(in English)

This lecture follows the first parts of these books fairly closely, which treat artificial neural networks.

Contents

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

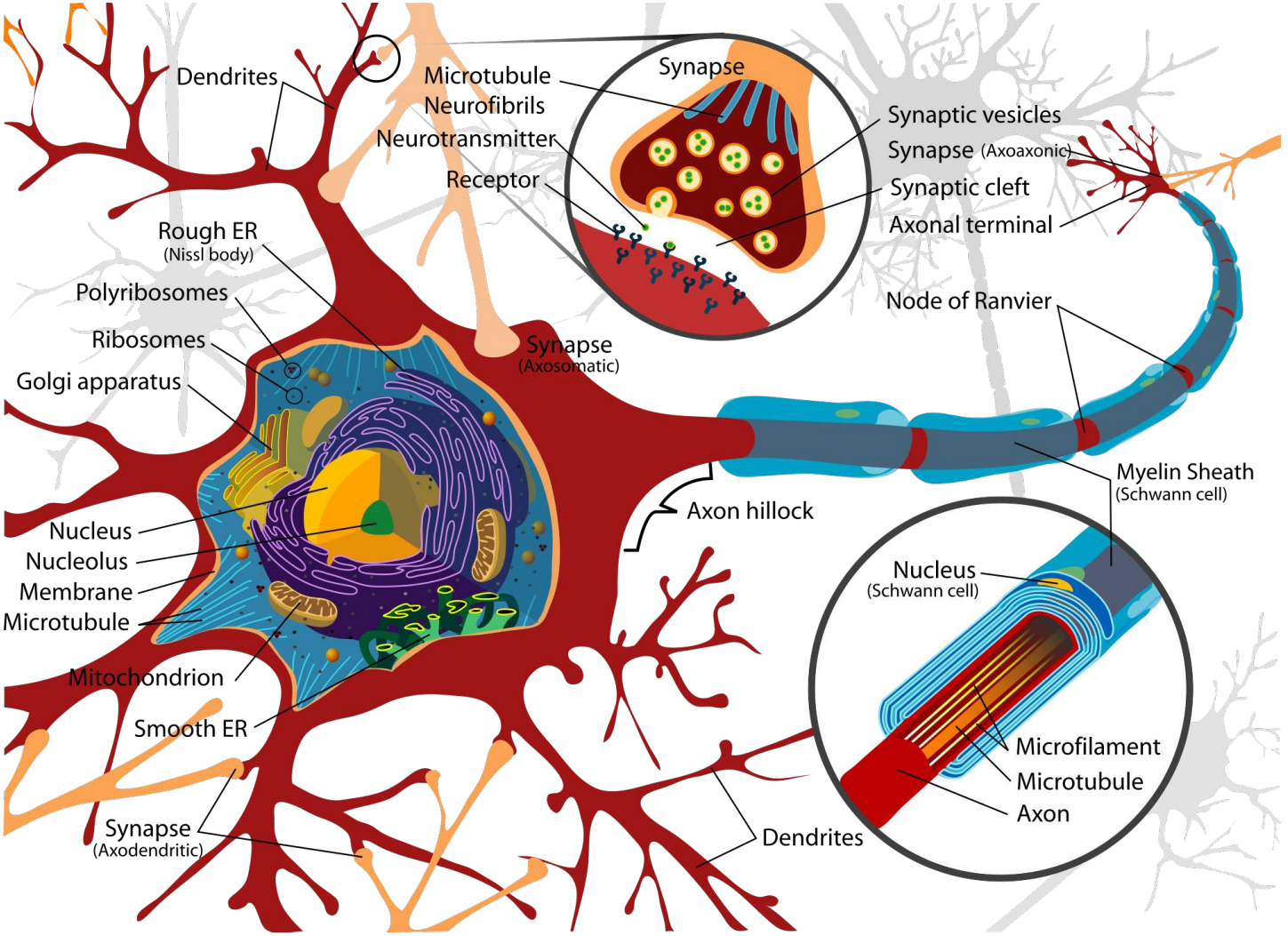
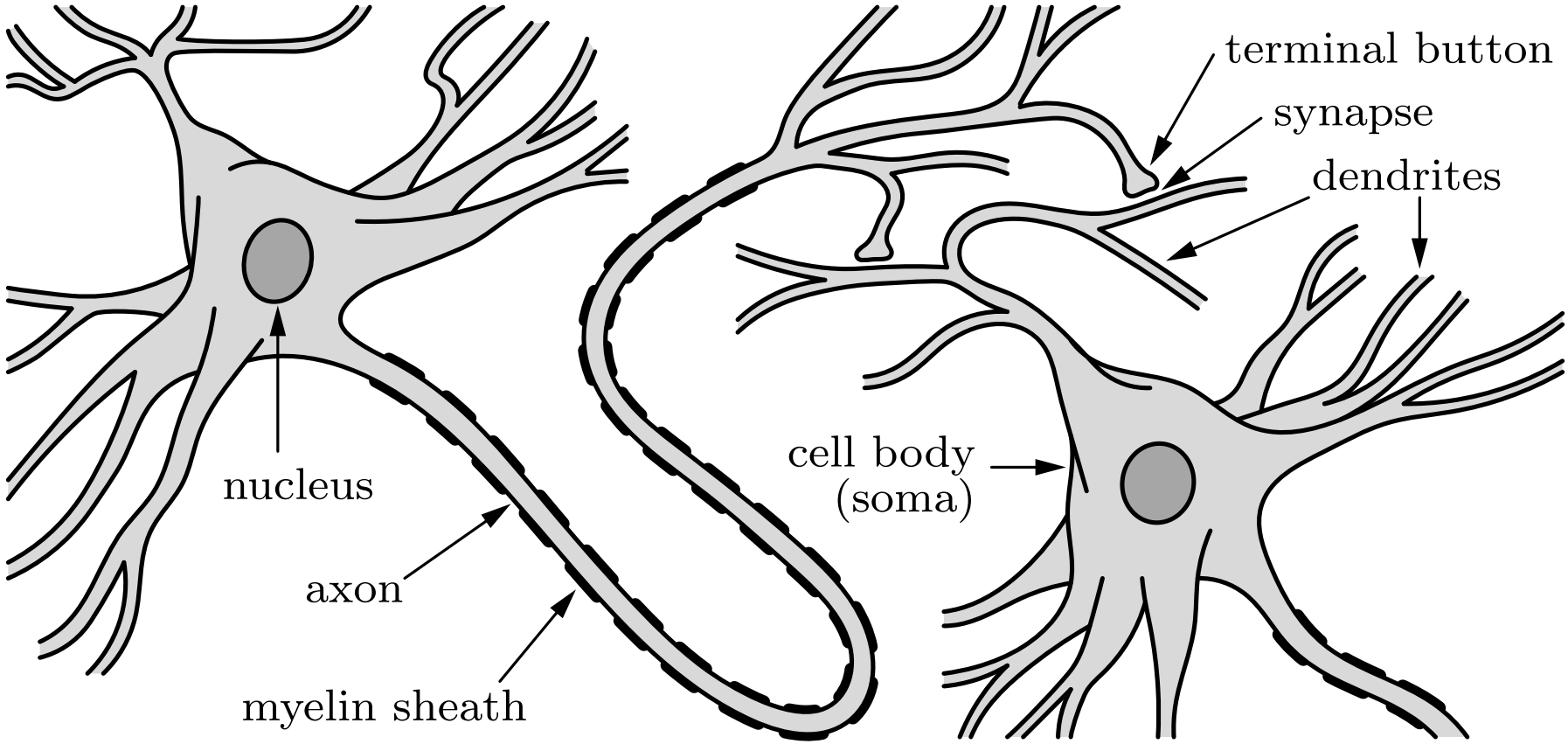


Diagram of a typical myelinated vertebrate motoneuron (source: Wikipedia, Ruiz-Villarreal 2007), showing the main parts involved in its signaling activity like the *dendrites*, the *axon*, and the *synapses*.

Biological Background

Structure of a prototypical biological neuron (simplified)



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.

(Personal) Computers versus the Human Brain

	Personal Computer	**Human Brain**
processing units	1 CPU, 2–10 cores 10^{10} transistors 1–2 graphics cards/GPUs, 10^3 cores/shaders 10^{10} transistors	10^{11} neurons
storage capacity	10^{10} bytes main memory (RAM) 10^{12} bytes external memory	10^{11} neurons 10^{14} synapses
processing speed	10^{-9} seconds 10^9 operations per second	$> 10^{-3}$ seconds < 1000 per second
bandwidth	10^{12} bits/second	10^{14} bits/second
neural updates	10^6 per second	10^{14} per second

(Personal) Computers versus the Human Brain

- The processing/switching time of a neuron is relatively large ($> 10^{-3}$ seconds), but updates are computed in parallel.
- A serial simulation on a computer takes several hundred clock cycles per update.

Advantages of Neural Networks:

- High processing speed due to massive parallelism.
- Fault Tolerance:
  Remain functional even if (larger) parts of a network get damaged.
- "Graceful Degradation":
  gradual degradation of performance if an increasing number of neurons fail.
- Well suited for inductive learning
  (learning from examples, generalization from instances).

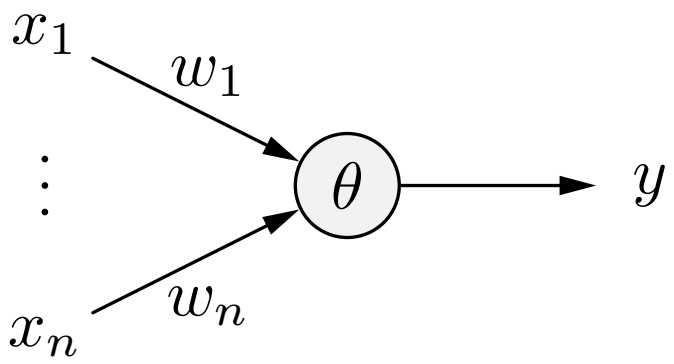
It appears to be reasonable to try to mimic or to recreate these advantages by constructing **artificial neural networks**.

Threshold Logic Units

Threshold Logic Units

A **Threshold Logic Unit (TLU)** is a processing unit for numbers with n inputs $x_1, \ldots, x_n$ 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

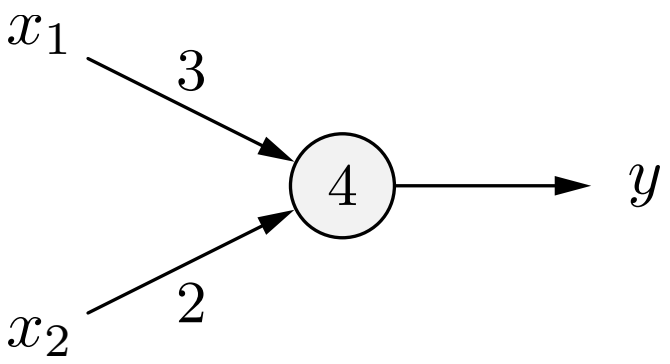
$$y = \begin{cases} 1, & \text{if} \quad \sum_{i=1}^{n} w_i x_i \geq \theta, \\ 0, & \text{otherwise.} \end{cases}$$



TLUs mimic the thresholding behavior of biological neurons in a (very) simple fashion.

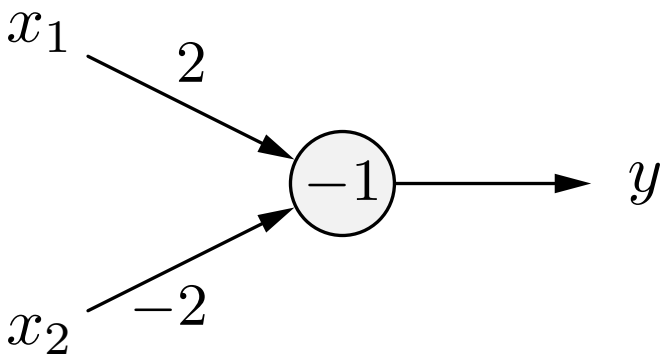
Threshold Logic Units: Examples

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

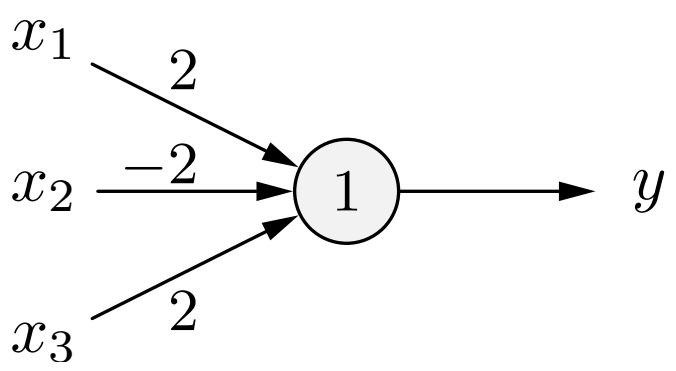
Threshold logic unit for the implication $x_2 \rightarrow x_1$.



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

Rough Intuition:

- Positive weights are analogous to excitatory synapses.
- Negative weights are analogous to inhibitory synapses.

Review of line representations

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

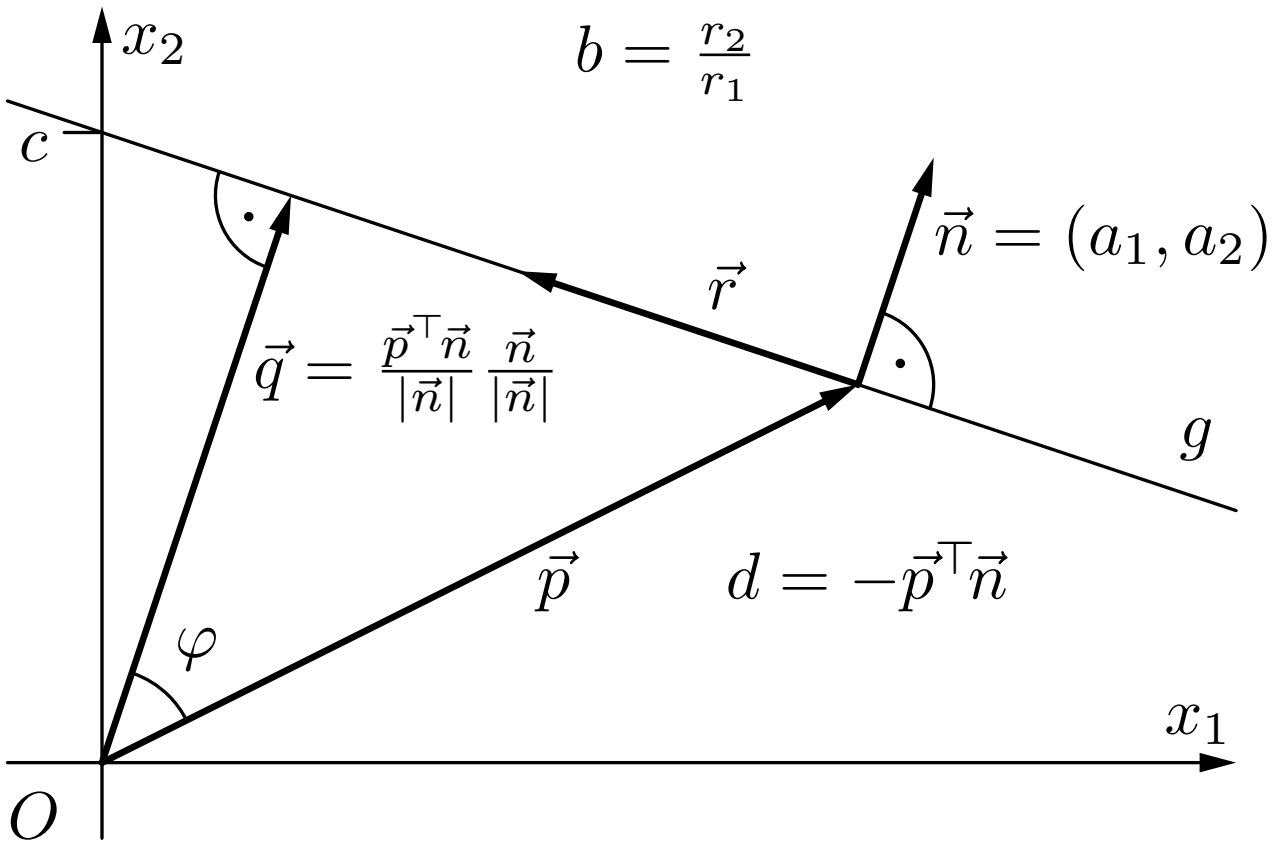
$$
\begin{array}{lcl}
\text{Explicit Form:} & g & \equiv \quad x_2 = bx_1 + c \\
\text{Implicit Form:} & g & \equiv \quad a_1 x_1 + a_2 x_2 + d = 0 \\
\text{Point-Direction Form:} & g & \equiv \quad \vec{x} = \vec{p} + k\vec{r} \\
\text{Normal Form:} & g & \equiv \quad (\vec{x} - \vec{p})^\top \vec{n} = 0
\end{array}
$$

with the parameters:

$$
\begin{array}{ll}
b: & \text{Gradient of the line} \\
c: & \text{Section of the } x_2 \text{ axis (intercept)} \\
\vec{p}: & \text{Vector of a point of the line (base vector)} \\
\vec{r}: & \text{Direction vector of the line} \\
\vec{n}: & \text{Normal vector of the line}
\end{array}
$$

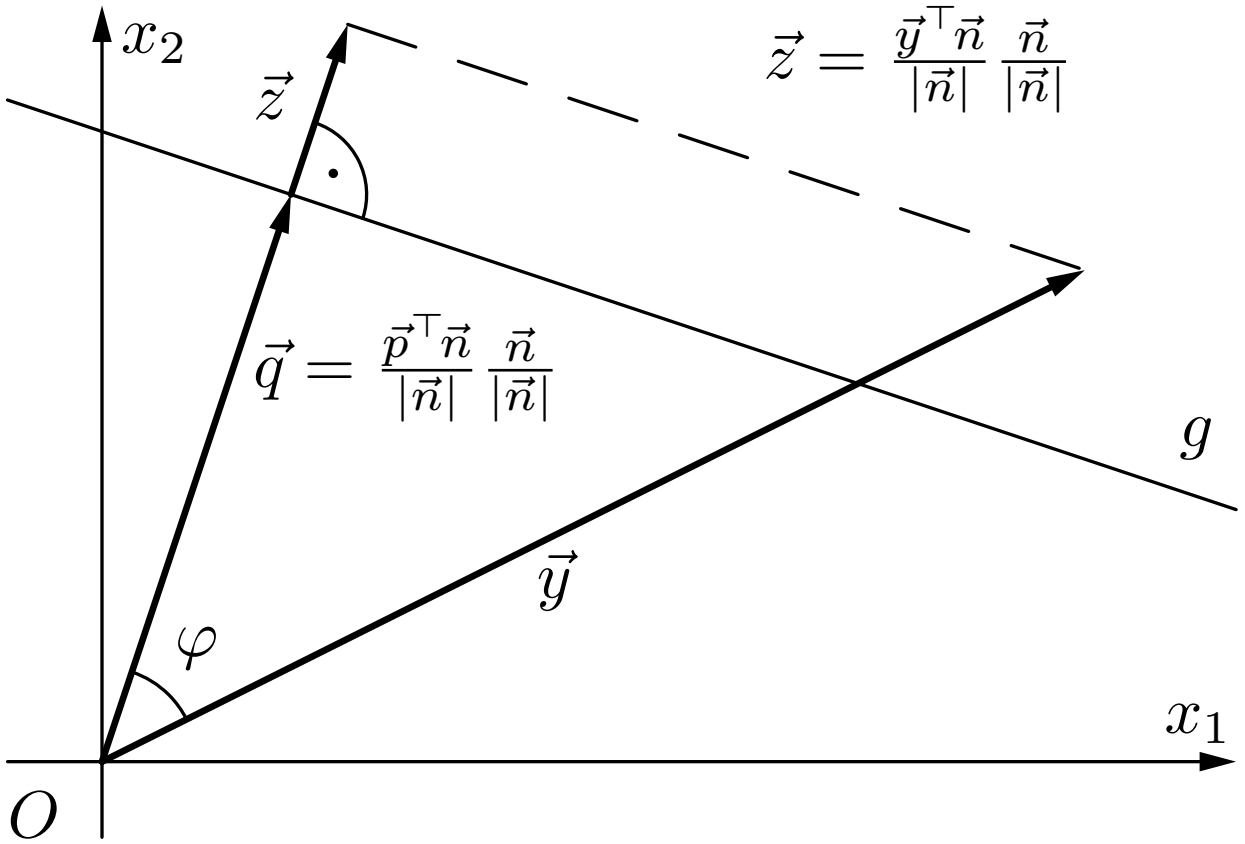
Threshold Logic Units: Geometric Interpretation

A straight line and its defining parameters:



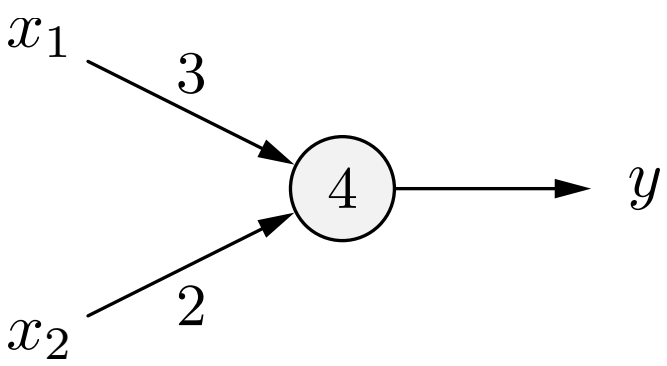
Threshold Logic Units: Geometric Interpretation

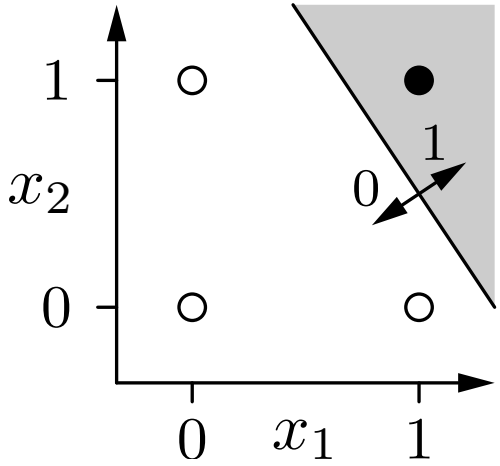
How to determine the side on which a point $\vec{y}$ lies:



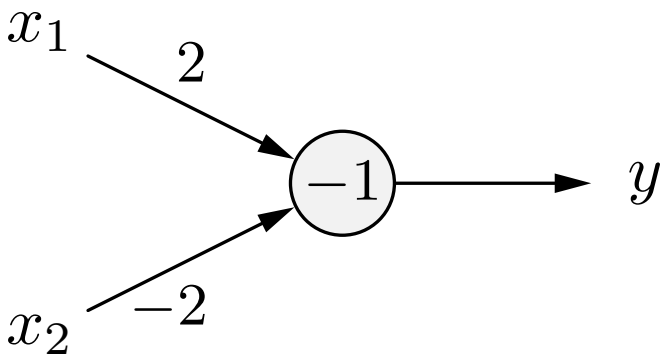
Threshold Logic Units: Geometric Interpretation

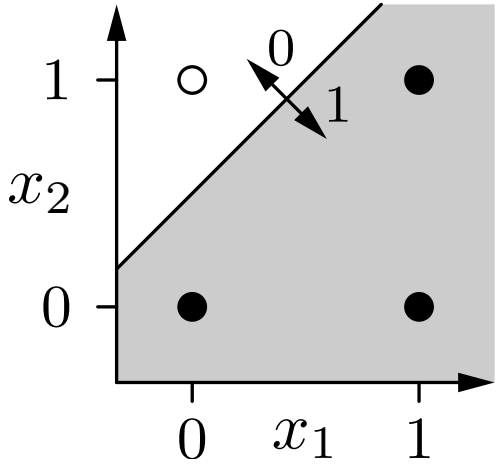
Threshold logic unit for $x_1 \wedge x_2$.





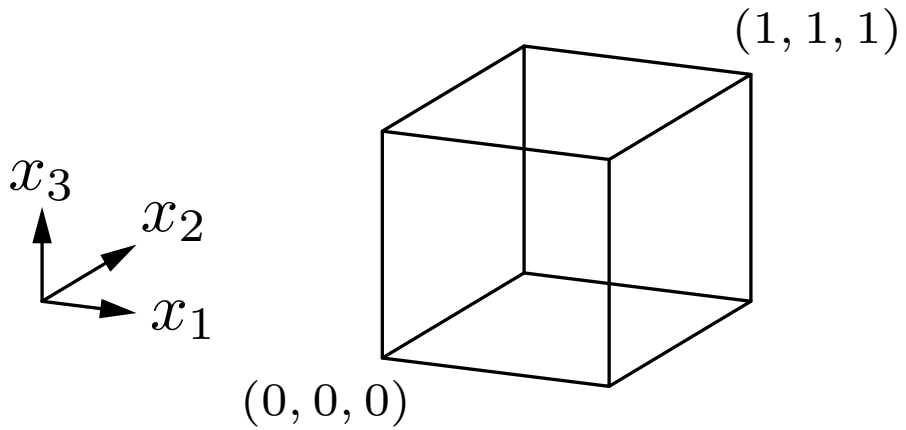
Threshold logic unit for $x_2 \rightarrow x_1$.



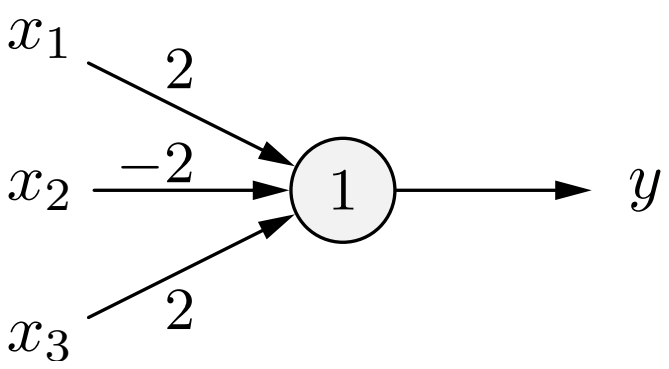


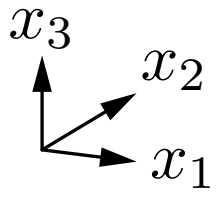
Threshold Logic Units: Geometric Interpretation

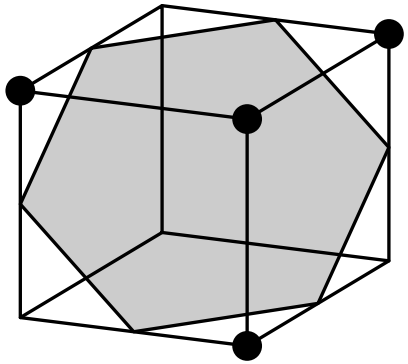
Visualization of 3-dimensional Boolean functions:



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



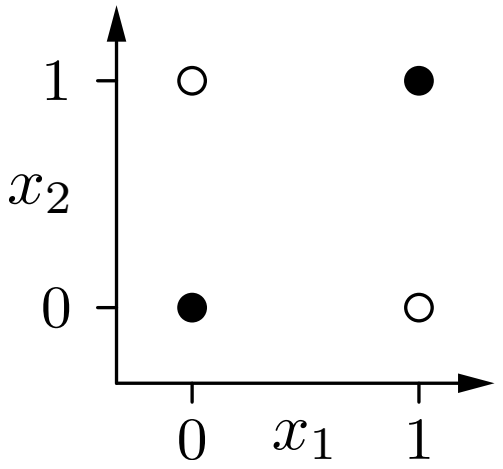




Threshold Logic Units: Limitations

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*:

$$\begin{aligned}
&\text{since } (0,0) \mapsto 1: && 0 && \geq \theta, && (1)\\
&\text{since } (1,0) \mapsto 0: && w_1 && < \theta, && (2)\\
&\text{since } (0,1) \mapsto 0: && w_2 && < \theta, && (3)\\
&\text{since } (1,1) \mapsto 1: && w_1 + w_2 && \geq \theta. && (4)
\end{aligned}$$

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

Linear Separability

Definition: Two sets of points in a Euclidean space are called **linearly separable**, iff there exists at least one point, line, plane or hyperplane (depending on the dimension of the Euclidean space), such that all points of the one set lie on one side and all points of the other set lie on the other side of this point, line, plane or hyperplane (or on it). That is, the point sets can be separated by a **linear decision function**. Formally: Two sets $X, Y \subset \mathbb{R}^m$ are linearly separable iff $\vec{w} \in \mathbb{R}^m$ and $\theta \in \mathbb{R}$ exist such that

$$\forall \vec{x} \in X: \quad \vec{w}^\top \vec{x} < \theta \qquad \text{and} \qquad \forall \vec{y} \in Y: \quad \vec{w}^\top \vec{y} \geq \theta.$$

- **Boolean functions** define two points sets, namely the set of points that are mapped to the function value 0 and the set of points that are mapped to 1.
  $\Rightarrow$ The term "linearly separable" can be transferred to Boolean functions.
- As we have seen, **conjunction** and **implication** are **linearly separable** (as are **disjunction**, NAND, NOR etc.).
- The **biimplication** is **not linearly separable** (and neither is the **exclusive or** (XOR)).

Linear Separability

Definition: A set of points in a Euclidean space is called **convex** if it is non-empty and connected (that is, if it is a *region*) and for every pair of points in it every point on the straight line segment connecting the points of the pair is also in the set.

Definition: The **convex hull** of a set of points X in a Euclidean space is the smallest convex set of points that contains X. Alternatively, the **convex hull** of a set of points X is the intersection of all convex sets that contain X.

Theorem: Two sets of points in a Euclidean space are **linearly separable** if and only if their convex hulls are disjoint (that is, have no point in common).

- For the biimplication problem, the convex hulls are the diagonal line segments.
- They share their intersection point and are thus not disjoint.
- Therefore the biimplication is not linearly separable.

Threshold Logic Units: Limitations

Total number and number of linearly separable Boolean functions
(On-Line Encyclopedia of Integer Sequences, `oeis.org`, A001146 and A000609):

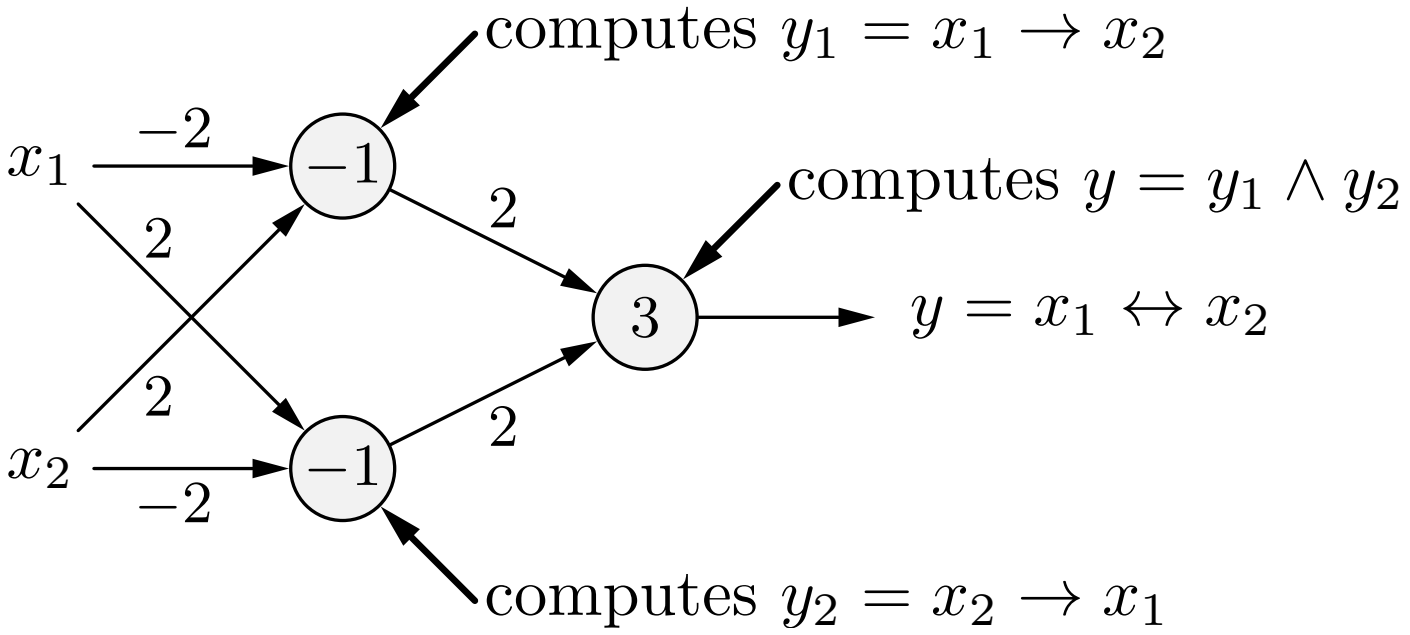
inputs	Boolean functions	linearly separable functions
1	4	4
2	16	14
3	256	104
4	65,536	1,882
5	4,294,967,296	94,572
6	18,446,744,073,709,551,616	15,028,134
n	$2^{(2^n)}$	no general formula known

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

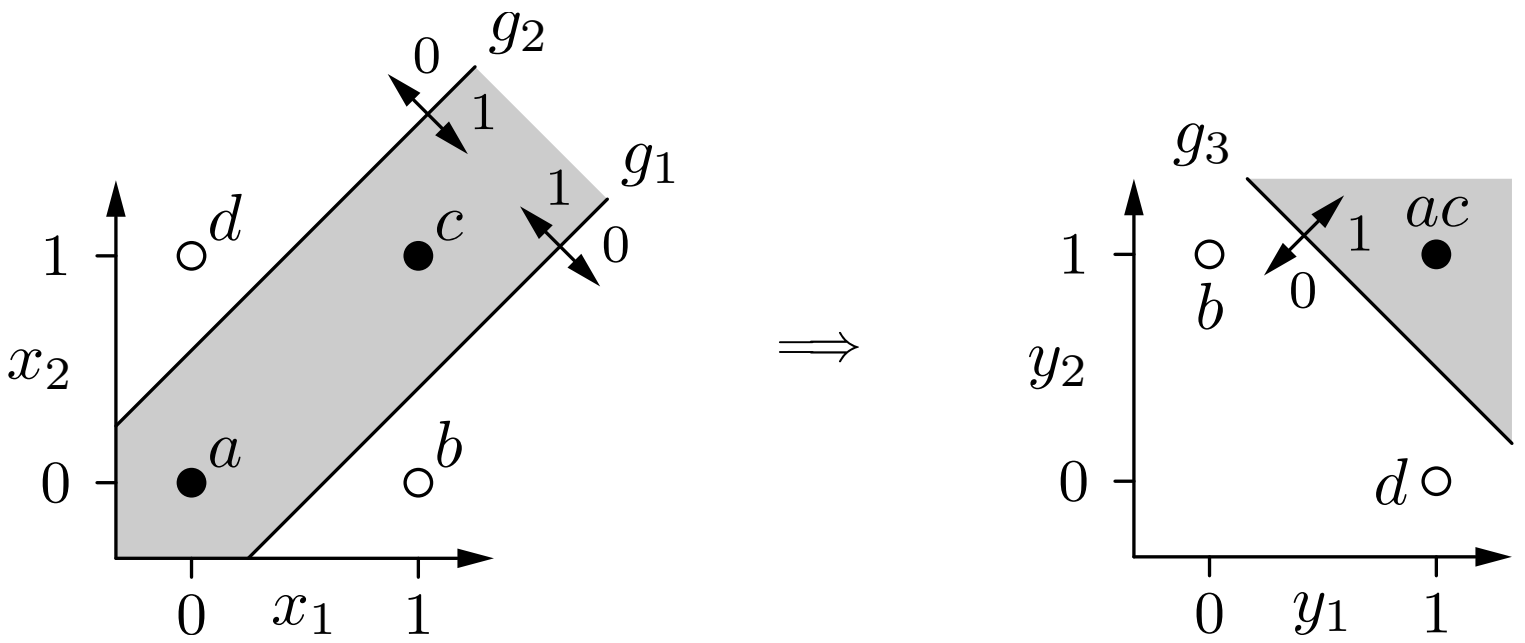
Networks of Threshold Logic Units

Solving the biimplication problem with a network.

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



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

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

(i) Represent the given function $f(x_1, \ldots, x_n)$ in disjunctive normal form. That is, determine $D_f = C_1 \vee \ldots \vee C_m$, where all C_j are conjunctions of n literals, that is, $C_j = l_{j1} \wedge \ldots \wedge l_{jn}$ with $l_{ji} = x_i$ (positive literal) or $l_{ji} = \neg x_i$ (negative literal).

(ii) Create a neuron for each conjunction C_j of the disjunctive normal form (having n inputs — one input for each variable), where

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

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

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

Remark: weights are set to ± 2 instead of ± 1 in order to ensure integer thresholds.

Representing Arbitrary Boolean Functions

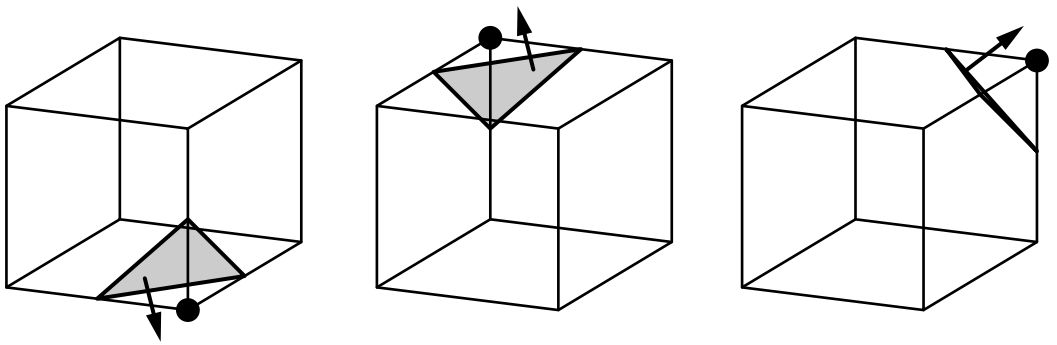
Example:
ternary Boolean function:

x_1	x_2	x_3	y	C_j
0	0	0	0	
1	0	0	1	$x_1 \wedge \overline{x_2} \wedge \overline{x_3}$
0	1	0	0	
1	1	0	0	
0	0	1	0	
1	0	1	0	
0	1	1	1	$\overline{x_1} \wedge x_2 \wedge x_3$
1	1	1	1	$x_1 \wedge x_2 \wedge x_3$

$$D_f = C_1 \vee C_2 \vee C_3$$

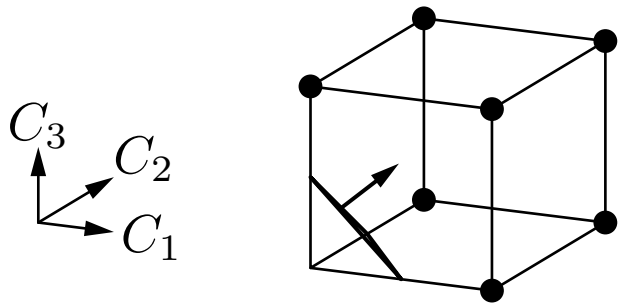
One conjunction for each row where the output y is 1 with literals according to input values.

First layer (conjunctions):



$C_1 = x_1 \wedge \overline{x_2} \wedge \overline{x_3}$ $\quad$ $C_2 = \overline{x_1} \wedge x_2 \wedge x_3$ $\quad$ $C_3 = x_1 \wedge x_2 \wedge x_3$

Second layer (disjunction):



$$D_f = C_1 \vee C_2 \vee C_3$$

Representing Arbitrary Boolean Functions

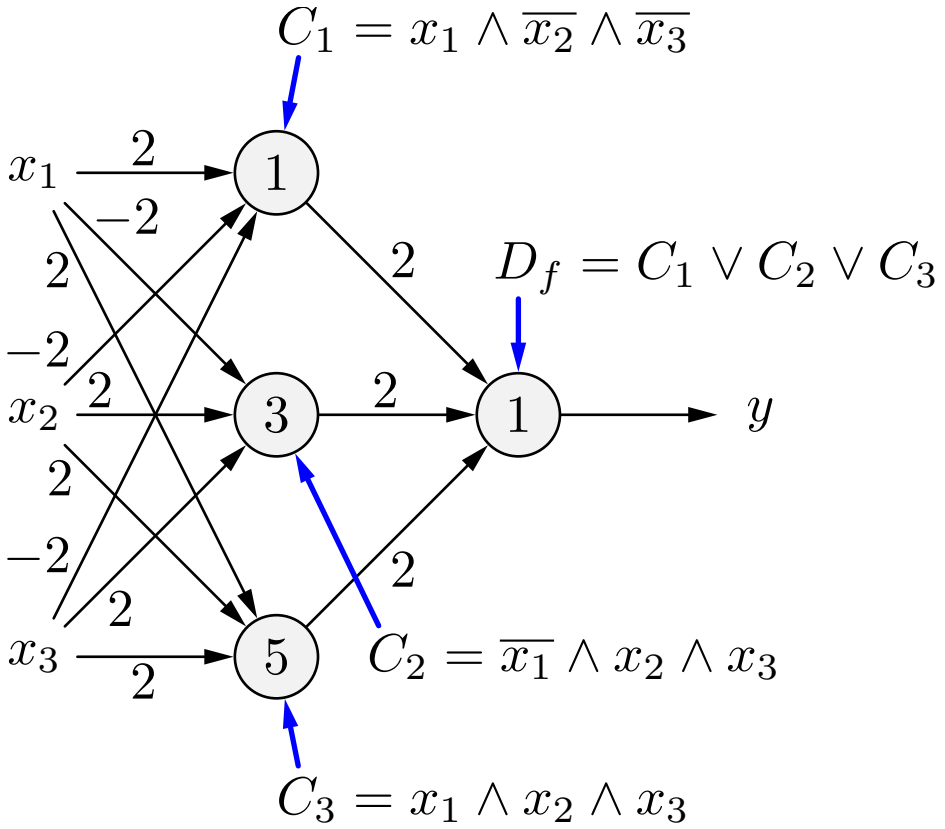
Example:
ternary Boolean function:

x_1	x_2	x_3	y	C_j
0	0	0	0	
1	0	0	1	$x_1 \wedge \overline{x_2} \wedge \overline{x_3}$
0	1	0	0	
1	1	0	0	
0	0	1	0	
1	0	1	0	
0	1	1	1	$\overline{x_1} \wedge x_2 \wedge x_3$
1	1	1	1	$x_1 \wedge x_2 \wedge x_3$

$D_f = C_1 \vee C_2 \vee C_3$

One conjunction for each row where the output y is 1 with literals according to input value.

Resulting network of threshold logic units:

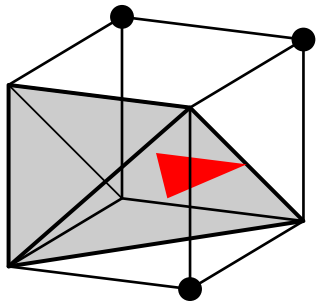


Reminder: Convex Hull Theorem

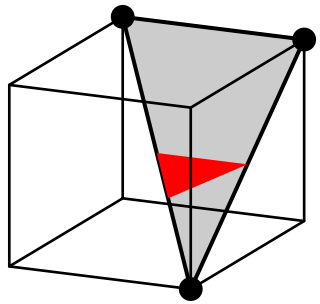
Theorem: Two sets of points in a Euclidean space are **linearly separable** if and only if their convex hulls are disjoint (that is, have no point in common).

Example function on the preceding slide:

$$y = f(x_1, x_2, x_3) = (x_1 \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_3)$$



Convex hull of points with $y = 0$



Convex hull of points with $y = 1$

- The convex hulls of the two point sets are not disjoint (red: intersection).
- Therefore the function $y = f(x_1, x_2, x_3)$ is not linearly separable.

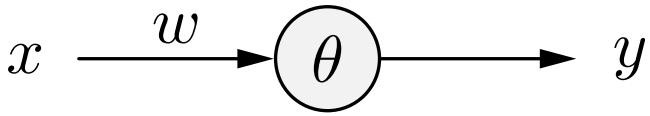
Training Threshold Logic Units

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: $e = e(w_1, \ldots, w_n, \theta)$.
  - Adapt weights and threshold so that the error becomes smaller.
  - Iterate adaptation until the error vanishes.

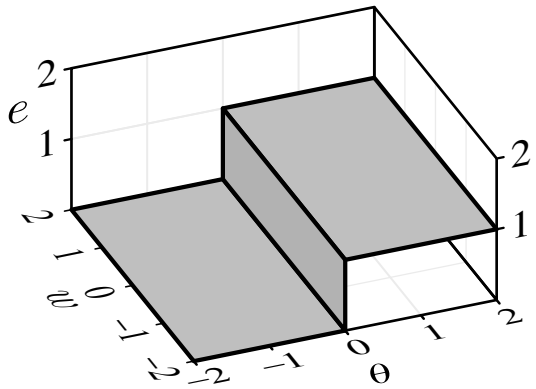
Training Threshold Logic Units

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

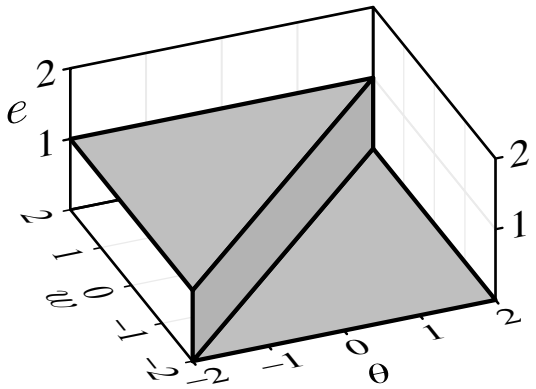


x	y
0	1
1	0

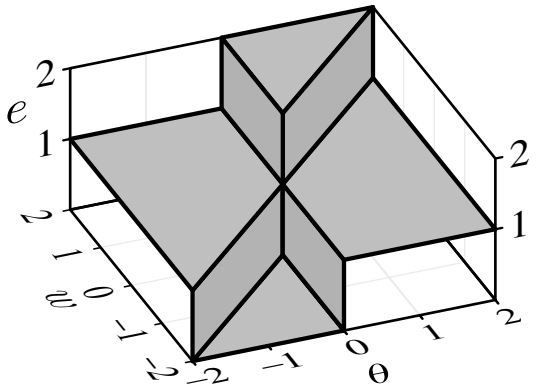
Output error as a function of weight and threshold.



error for $x = 0$



error for $x = 1$

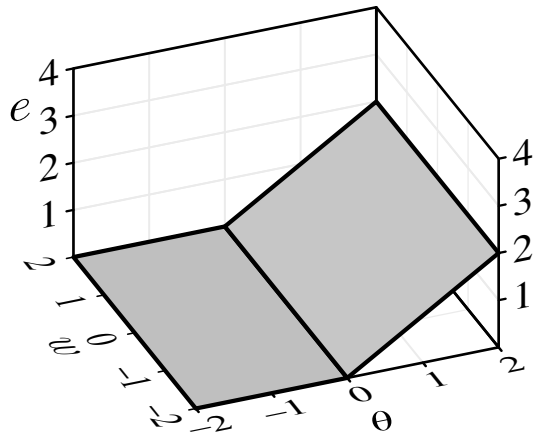


sum of errors

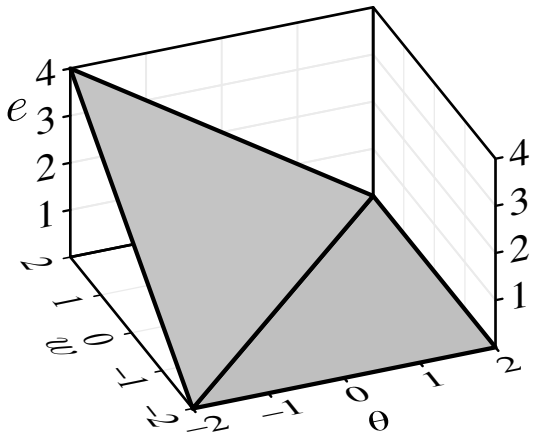
Training Threshold Logic Units

- 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
  (that is, consider "how wrong" the relation of weighted sum and threshold is).

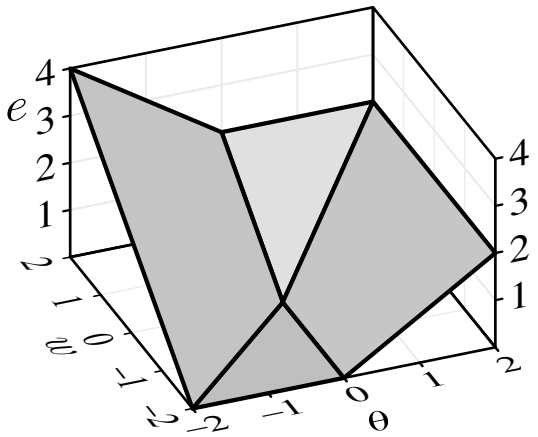
Modified output error as a function of weight and threshold.



error for $x = 0$



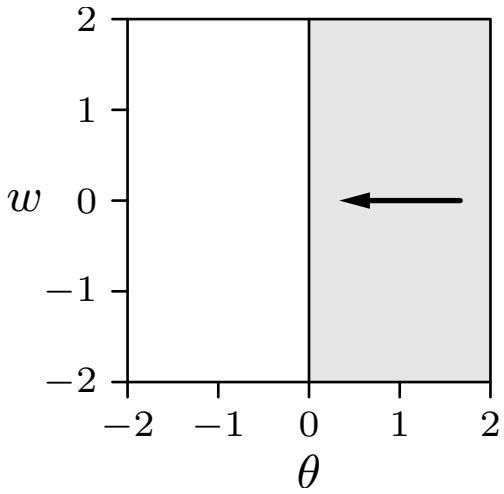
error for $x = 1$



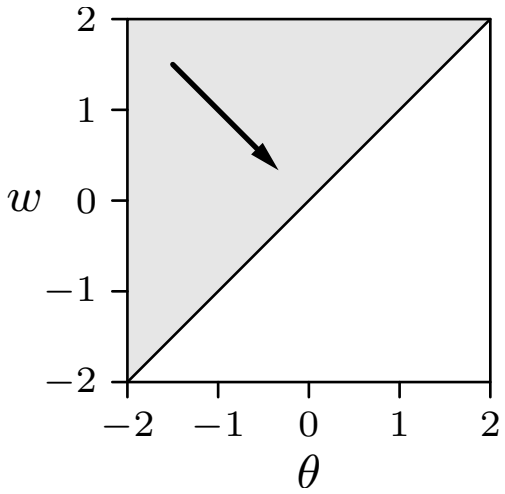
sum of errors

Training Threshold Logic Units

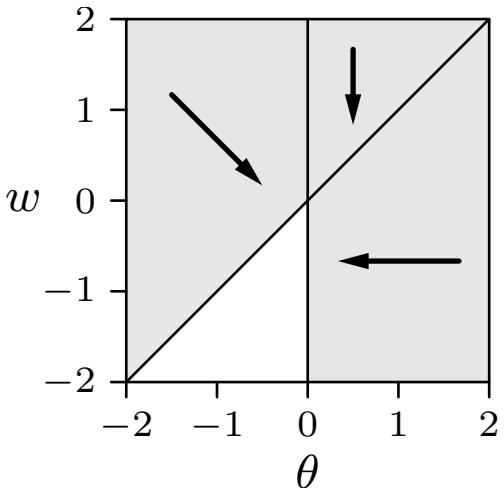
Schemata of resulting directions of parameter changes.



changes for $x = 0$



changes for $x = 1$



sum of changes

- Start at a random point.
- Iteratively adapt parameters
  according to the direction corresponding to the current point.
- Stop if the error vanishes.

Training Threshold Logic Units: Delta Rule

Formal Training Rule: Let $\vec{x} = (x_1, \ldots, x_n)^\top$ 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 $\vec{w} = (w_1, \ldots, w_n)^\top$ are adapted as follows in order to reduce the error:

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

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**, that is, after a traversal of all training patterns.

Training Threshold Logic Units: Delta Rule

```
procedure online_training (var w⃗, var θ, L, η);
var y, e;                                  (* output, sum of errors *)
begin
  repeat                                   (* training loop *)
    e := 0;                                (* initialize the error sum *)
    for all (x⃗, o) ∈ L do begin            (* traverse the patterns *)
      if (w⃗ᵀ x⃗ ≥ θ) then y := 1;           (* compute the output *)
                    else y := 0;           (* of the threshold logic unit *)
      if (y ≠ o) then begin                (* if the output is wrong *)
        θ  := θ  − η(o − y);               (* adapt the threshold *)
        w⃗ := w⃗ + η(o − y)x⃗;               (* and the weights *)
        e  := e  +  |o − y|;               (* sum the errors *)
      end;
    end;
  until (e ≤ 0);                           (* repeat the computations *)
end;                                       (* until the error vanishes *)
```

Training Threshold Logic Units: Delta Rule

```
procedure batch_training (var w⃗, var θ, L, η);
var y, e, θ_c, w⃗_c;                          (* output, sum of errors, sums of changes *)
begin
  repeat                                     (* training loop *)
    e := 0; θ_c := 0; w⃗_c := 0⃗;              (* initializations *)
    for all (x⃗, o) ∈ L do begin              (* traverse the patterns *)
      if (w⃗^⊤ x⃗ ≥ θ) then y := 1;            (* compute the output *)
                     else  y := 0;           (* of the threshold logic unit *)
      if (y ≠ o) then begin                  (* if the output is wrong *)
        θ_c  := θ_c  − η(o − y);              (* sum the changes of the *)
        w⃗_c := w⃗_c + η(o − y)x⃗;             (* threshold and the weights *)
        e    := e    +  |o − y|;             (* sum the errors *)
      end;
    end;
    θ  := θ  + θ_c;                          (* adapt the threshold *)
    w⃗ := w⃗ + w⃗_c;                          (* and the weights *)
  until (e ≤ 0);                             (* repeat the computations *)
end;                                         (* until the error vanishes *)
```

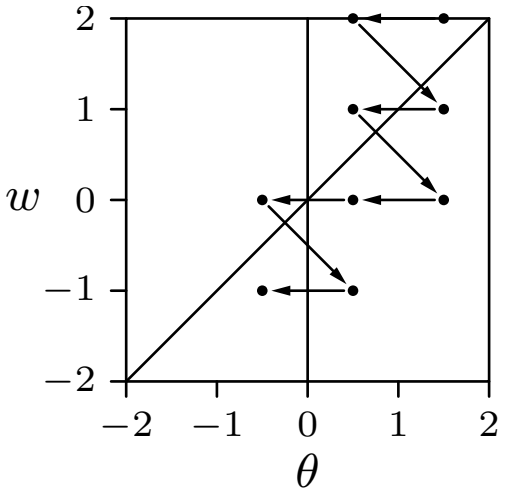
Training Threshold Logic Units: Online

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

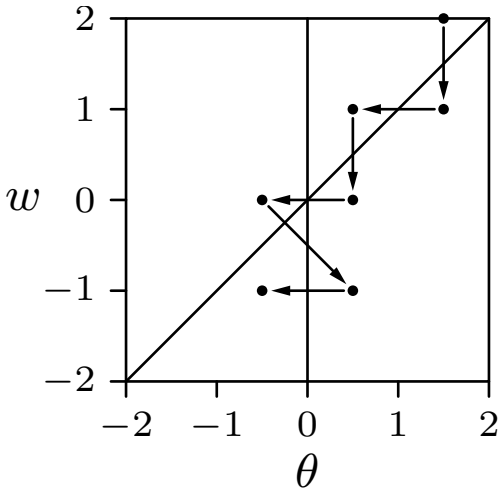
Training Threshold Logic Units: Batch

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	0	0.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	0.5	1
3	0	1	−0.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	0.5	0
4	0	1	−0.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	−0.5	0
5	0	1	0.5	1	0	0	0		
	1	0	0.5	1	−1	1	−1	0.5	−1
6	0	1	−0.5	0	1	−1	0		
	1	0	−1.5	0	0	0	0	−0.5	−1
7	0	1	0.5	1	0	0	0		
	1	0	−0.5	0	0	0	0	−0.5	−1

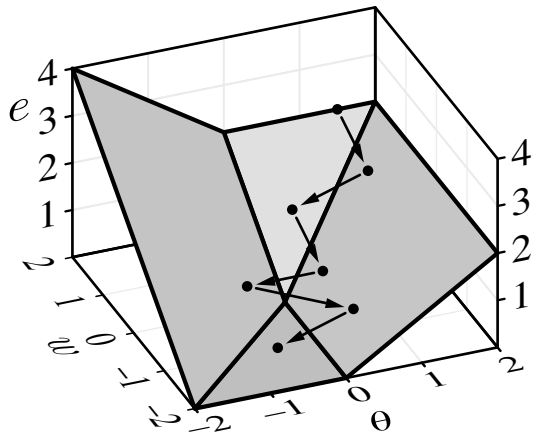
Example training procedure: Online and batch training.



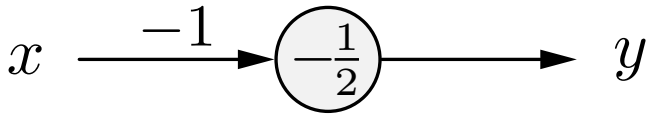
Online Training

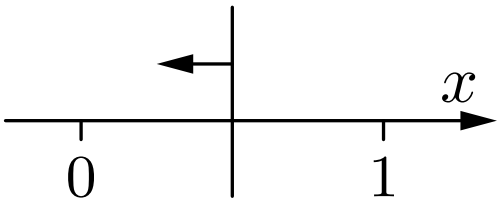


Batch Training



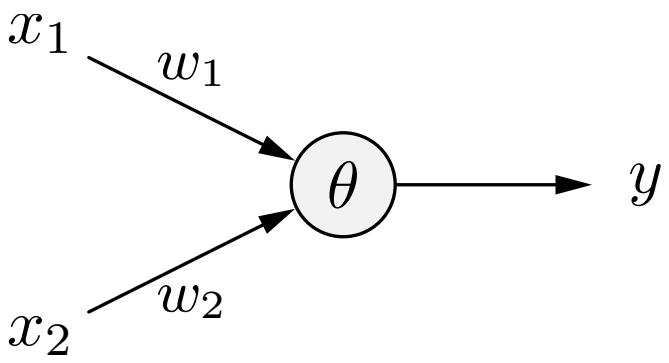
Batch Training



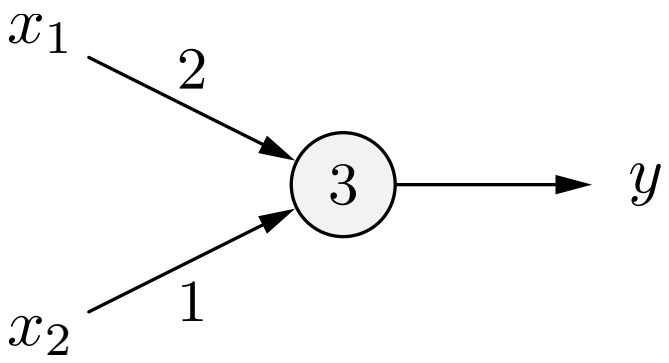


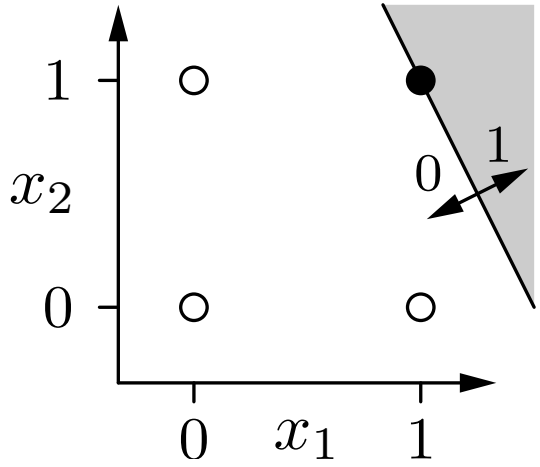
Training Threshold Logic Units: Conjunction

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 $L = \{(\vec{x}_1, o_1), \ldots (\vec{x}_m, o_m)\}$ 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, that is, if $\vec{w} \in \mathbb{R}^n$ and $\theta \in \mathbb{R}$ exist such that

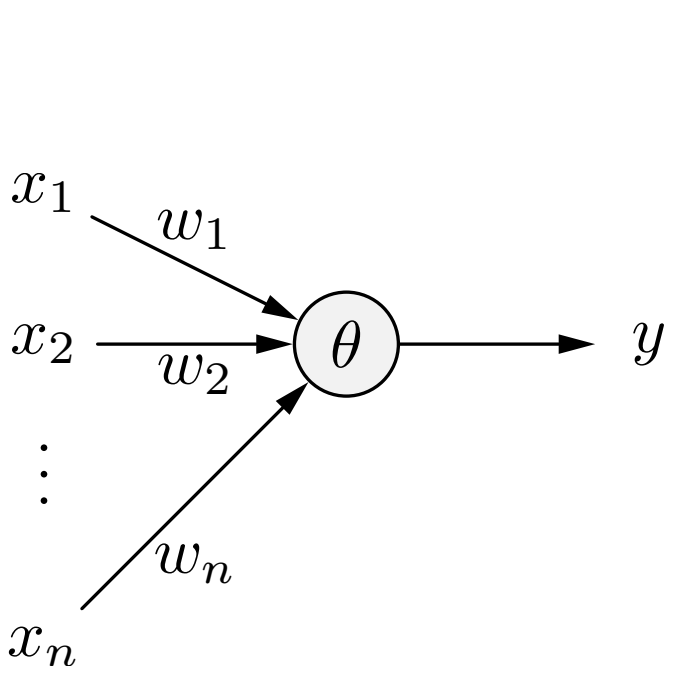
$$\begin{aligned} \forall (\vec{x}, 0) \in L_0 : \quad & \vec{w}^\top \vec{x} < \theta \qquad \text{and} \\ \forall (\vec{x}, 1) \in L_1 : \quad & \vec{w}^\top \vec{x} \geq \theta, \end{aligned}$$

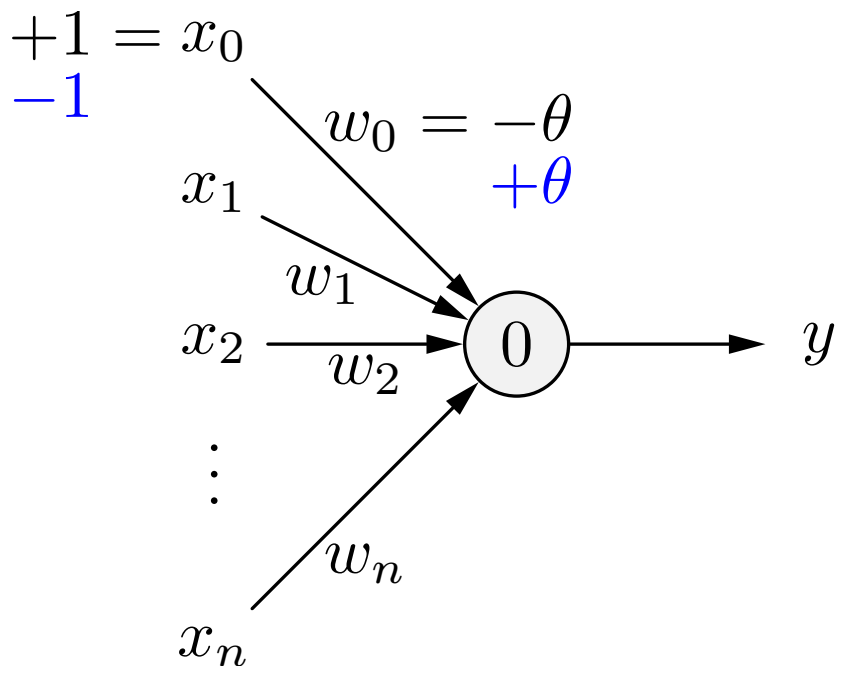
then online as well as batch training terminate.

- 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 (oscillation, repeated computation of same non-solving $\vec{w}$ and θ).

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

Formal Training Rule (with threshold turned into a weight):
Let $\vec{x} = (x_0 = 1, x_1, \ldots, x_n)^\top$ be an (extended) 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 (extended) weight vector $\vec{w} = (w_0 = -\theta, w_1, \ldots, w_n)^\top$ is adapted as follows in order to reduce the error:

$$\forall i \in \{0, \ldots, n\}: \quad w_i^{(\text{new})} = w_i^{(\text{old})} + \Delta w_i \qquad \text{with} \qquad \Delta w_i = \eta(o - y)x_i,$$

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].

- Note that with extended input and weight vectors, there is only one update rule (no distinction of threshold and weights).
- Note also that the (extended) input vector may be $\vec{x} = (x_0 = -1, x_1, \ldots, x_n)^\top$ and the corresponding (extended) weight vector $\vec{w} = (w_0 = +\theta, w_1, \ldots, w_n)^\top$.

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 easy and 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(s),
  - the problem can usually be solved with several different functions
    computed by the neurons of the first layer(s) (non-unique solution).
- When this situation became clear,
  neural networks were first seen as a "research dead end".

General (Artificial) Neural Networks

Basic graph theoretic notions

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

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

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

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

are called the **predecessors** of the vertex u
and the vertices of the set

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

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

General Neural Networks

General definition of a neural network

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

The set U of vertices is partitioned into

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

It is

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

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

General Neural Networks

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

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

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

- the **external input** ext_u.

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

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

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

General Neural Networks

Types of (artificial) neural networks:

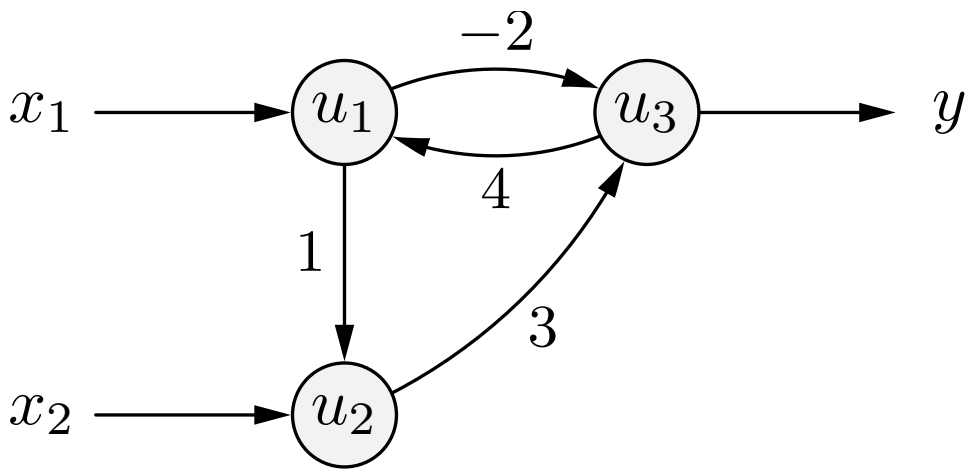
- If the graph of a neural network is **acyclic**, it is called a **feed-forward network**.
- If the graph of a neural network contains **cycles** (backward connections), it is called a **recurrent network**.

Representation of the connection weights as a matrix:

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

General Neural Networks: Example

A simple recurrent neural network

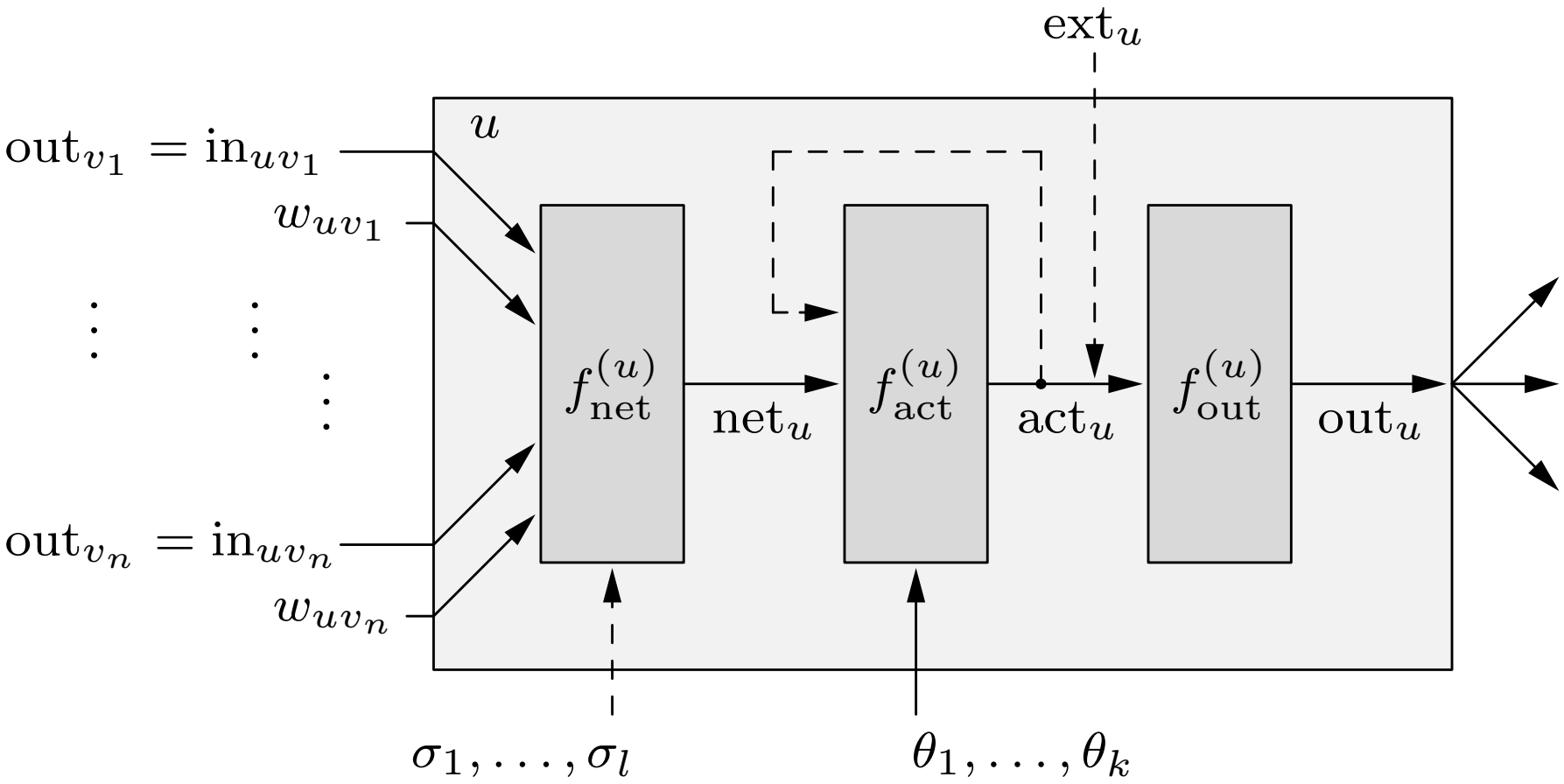


Weight matrix of this network

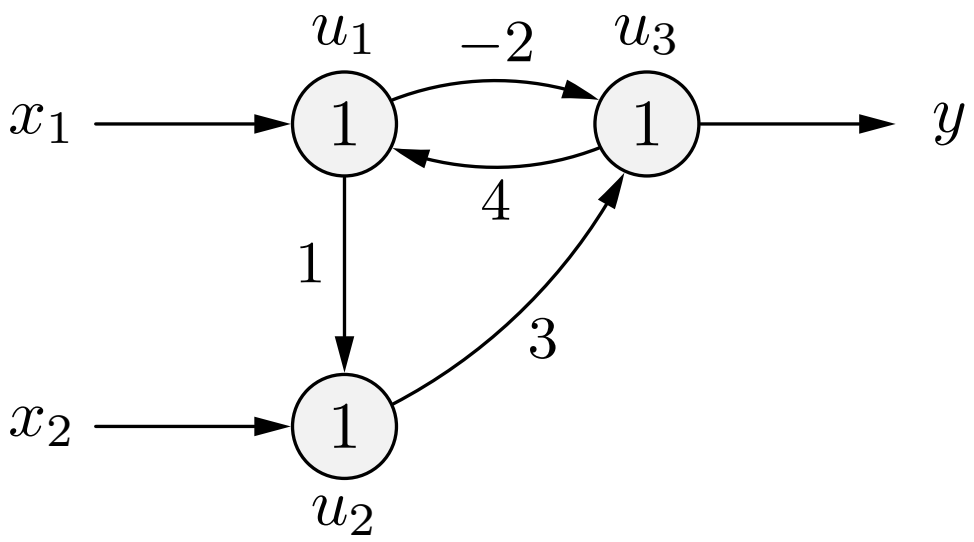
$$\begin{array}{c} \begin{array}{ccc} u_1 & u_2 & u_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 4 \\ 1 & 0 & 0 \\ -2 & 3 & 0 \end{pmatrix} \begin{array}{c} u_1 \\ u_2 \\ u_3 \end{array} \end{array}$$

Structure of a Generalized Neuron

A generalized neuron is a simple numeric processor



General Neural Networks: Example



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

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

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

General Neural Networks: Example

Updating the activations of the neurons

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

- Order in which the neurons are updated:
  $u_3, u_1, u_2, u_3, u_1, u_2, u_3, \ldots$
- Input phase: activations/outputs in the initial state.
- Work phase: activations/outputs of the next neuron to update (bold) are computed from the outputs of the other neurons and the weights/threshold.
- 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**	net_{u_3}	$= -2$	< 1
	1	**1**	0	net_{u_2}	$= 1$	≥ 1
	0	1	0	net_{u_1}	$= 0$	< 1
	0	1	**1**	net_{u_3}	$= 3$	≥ 1
	0	**0**	1	net_{u_2}	$= 0$	< 1
	1	0	1	net_{u_1}	$= 4$	≥ 1
	1	0	**0**	net_{u_3}	$= -2$	< 1

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

General Neural Networks: Training

Definition of learning tasks for a neural network

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

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

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

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

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

Solving a fixed learning task: Error definition

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

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

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

General Neural Networks: Training

Definition of learning tasks for a neural network

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

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

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

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

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)

Normalization of the input vectors

- Compute expected value and (corrected) standard deviation for each input:

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

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

$$\text{ext}_{u_k}^{(l)(\text{new})} = \frac{\text{ext}_{u_k}^{(l)(\text{old})} - \mu_k}{\sigma_k}$$

- Such a normalization avoids unit and scaling problems.
  It is also known as **z-scaling** or **z-score standardization**.

Multi-layer Perceptrons (MLPs)

Multi-layer 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) $U_{\text{hidden}} = U_{\text{hidden}}^{(1)} \cup \cdots \cup U_{\text{hidden}}^{(r-2)}$,

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

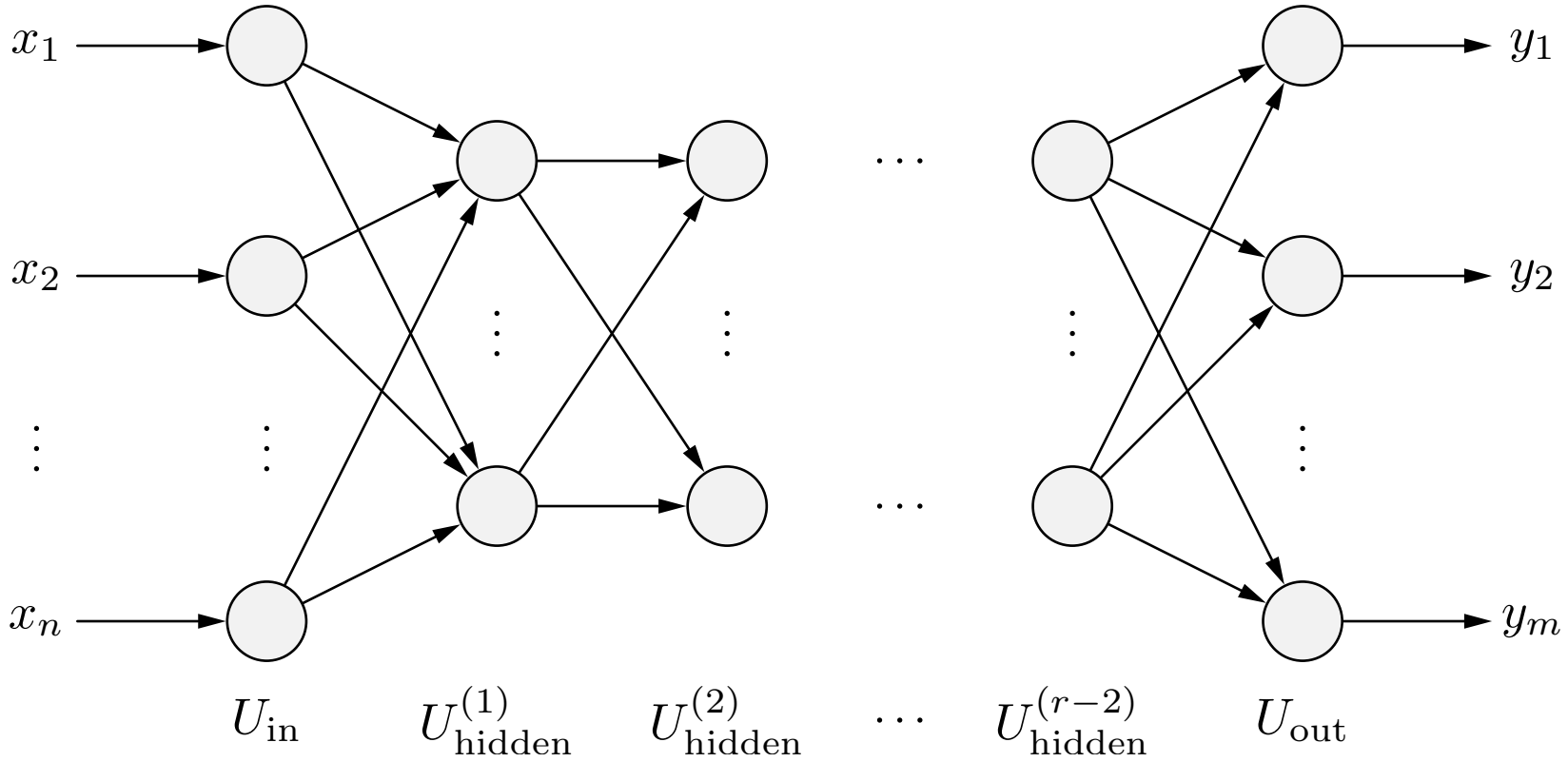
(iii) $C \subseteq \left(U_{\text{in}} \times U_{\text{hidden}}^{(1)}\right) \cup \left(\bigcup_{i=1}^{r-3} U_{\text{hidden}}^{(i)} \times U_{\text{hidden}}^{(i+1)}\right) \cup \left(U_{\text{hidden}}^{(r-2)} \times U_{\text{out}}\right)$

or, if there are no hidden neurons ($r = 2, U_{\text{hidden}} = \emptyset$),

$C \subseteq U_{\text{in}} \times U_{\text{out}}$.

- Feed-forward network with strictly layered structure.

General structure of a multi-layer perceptron



Multi-layer Perceptrons

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

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

- The activation function of each hidden neuron is a so-called **sigmoid function**, that is, a monotonically increasing function

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

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

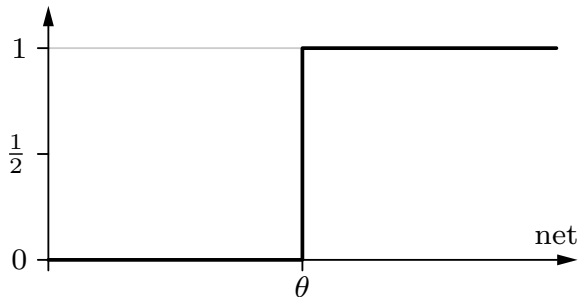
$$f_{\text{act}}(\text{net}, \theta) = \alpha\, \text{net} - \theta.$$

Only the step function is a neurobiologically plausible activation function.

Sigmoid Activation Functions

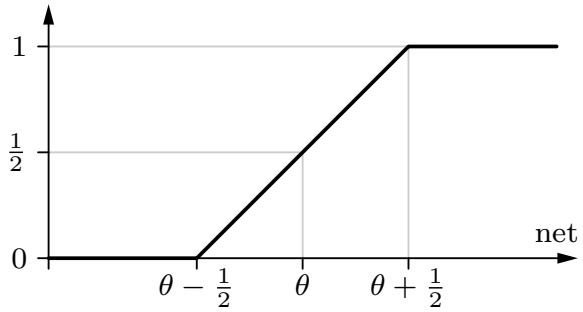
step function:

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



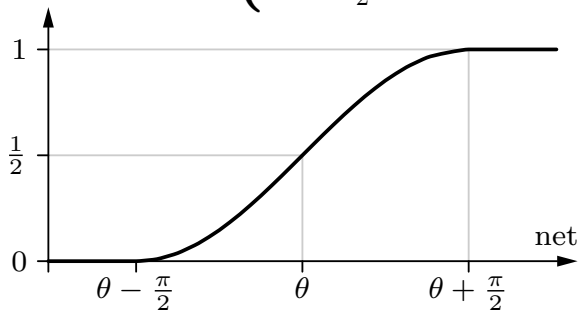
semi-linear function:

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



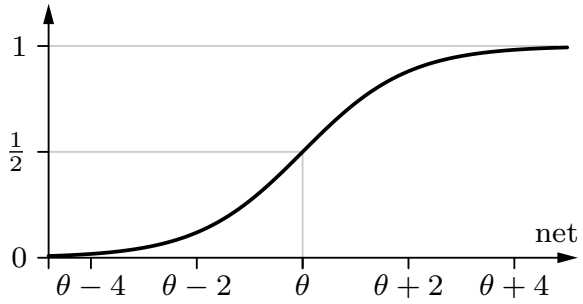
sine until saturation:

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



logistic function:

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

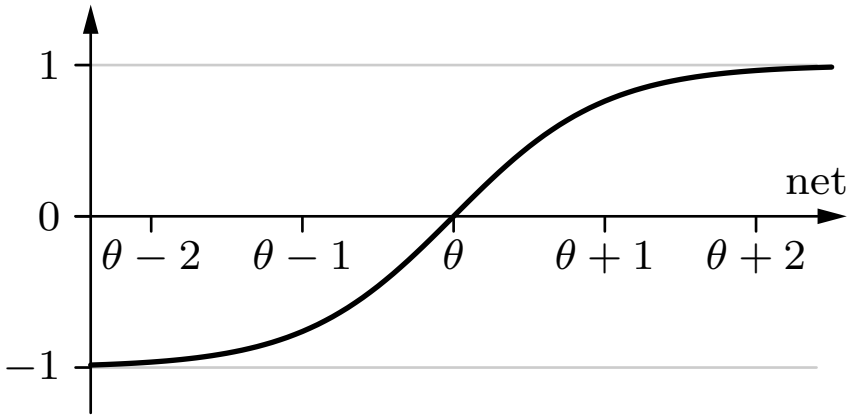


Sigmoid Activation Functions

- All sigmoid functions on the previous slide are **unipolar**, that is, they range from 0 to 1.
- Sometimes **bipolar** sigmoid functions are used (ranging from -1 to $+1$), like the hyperbolic tangent (*tangens hyperbolicus*).

hyperbolic tangent:

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



Multi-layer Perceptrons: Weight Matrices

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

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

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

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

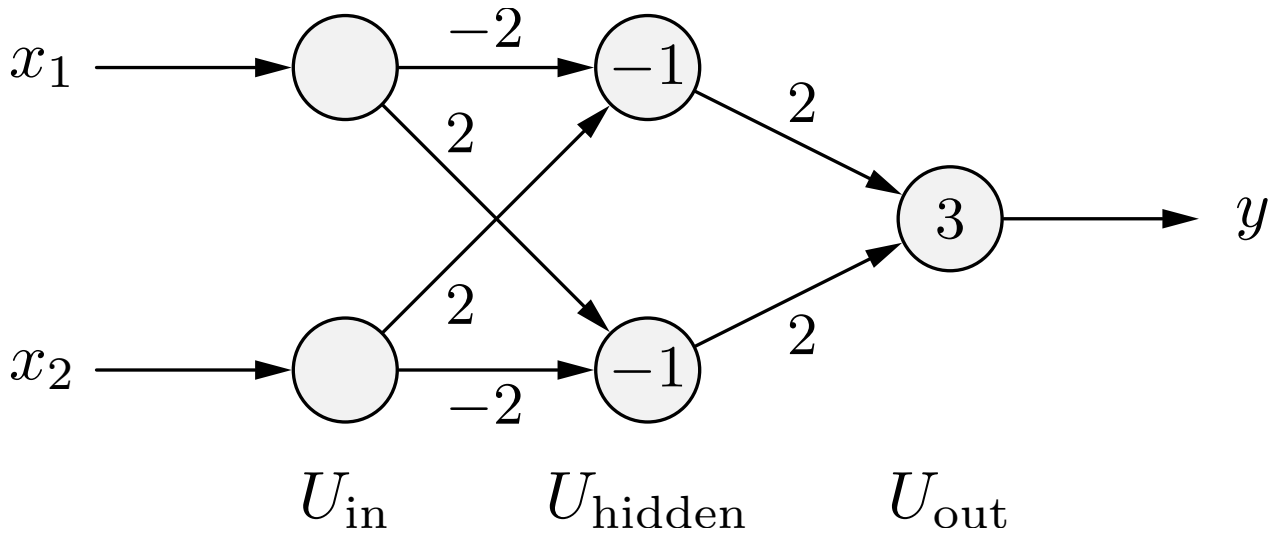
Advantage: The computation of the network input can be written as

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

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

Multi-layer Perceptrons: Biimplication

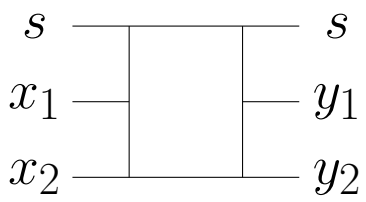
Solving the biimplication problem with a multi-layer perceptron.

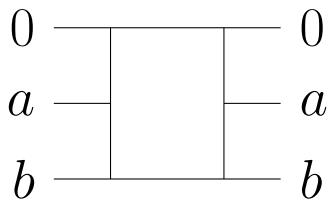


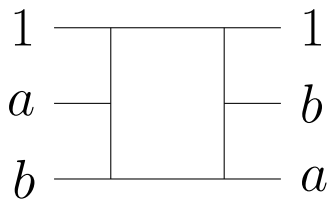
Note the additional input neurons compared to the TLU solution.

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

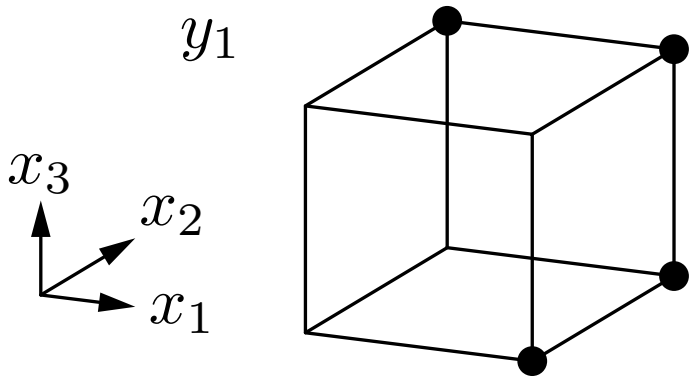
Multi-layer Perceptrons: Fredkin Gate

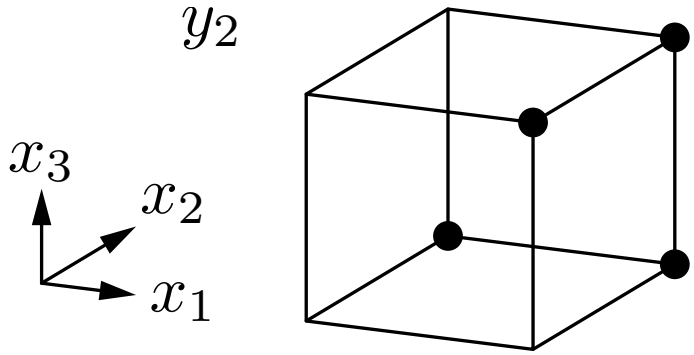






s	0	0	0	0	1	1	1	1
x_1	0	0	1	1	0	0	1	1
x_2	0	1	0	1	0	1	0	1
y_1	0	0	1	1	0	1	0	1
y_2	0	1	0	1	0	0	1	1





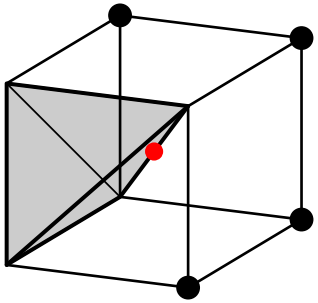
Multi-layer Perceptrons: Fredkin Gate

- The **Fredkin gate** (after Edward Fredkin *1934) or **controlled swap gate** (CSWAP) is a computational circuit that is used in **conservative logic** and **reversible computing**.
- Conservative logic is a model of computation that explicitly reflects the physical properties of computation, like the reversibility of the dynamical laws and the conservation of certain quantities (e.g. energy) [Fredkin and Toffoli 1982].
- The Fredkin gate is **reversible** in the sense that the inputs can be computed as functions of the outputs in the same way in which the outputs can be computed as functions of the inputs (no information loss, no entropy gain).
- The Fredkin gate is **universal** in the sense that all Boolean functions can be computed using only Fredkin gates.
- Note that both outputs, y_1 and y_2 are **not linearly separable**, because the convex hull of the points mapped to 0 and the convex hull of the points mapped to 1 share the point in the center of the cube.

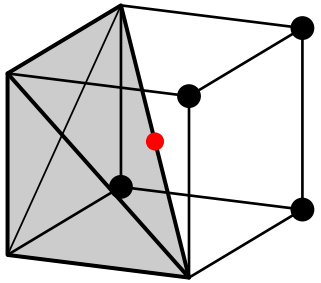
Reminder: Convex Hull Theorem

Theorem: Two sets of points in a Euclidean space are **linearly separable** if and only if their convex hulls are disjoint (that is, have no point in common).

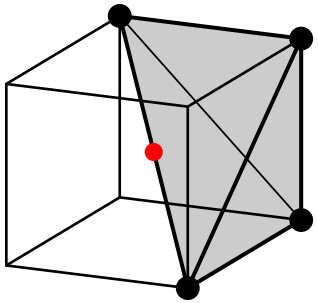
Both outputs y_1 and y_2 of a Fredkin gate are not linearly separable:



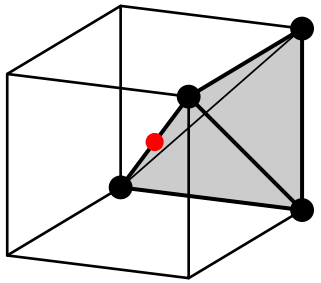
Convex hull of points with $y_1 = 0$



Convex hull of points with $y_2 = 0$



Convex hull of points with $y_1 = 1$

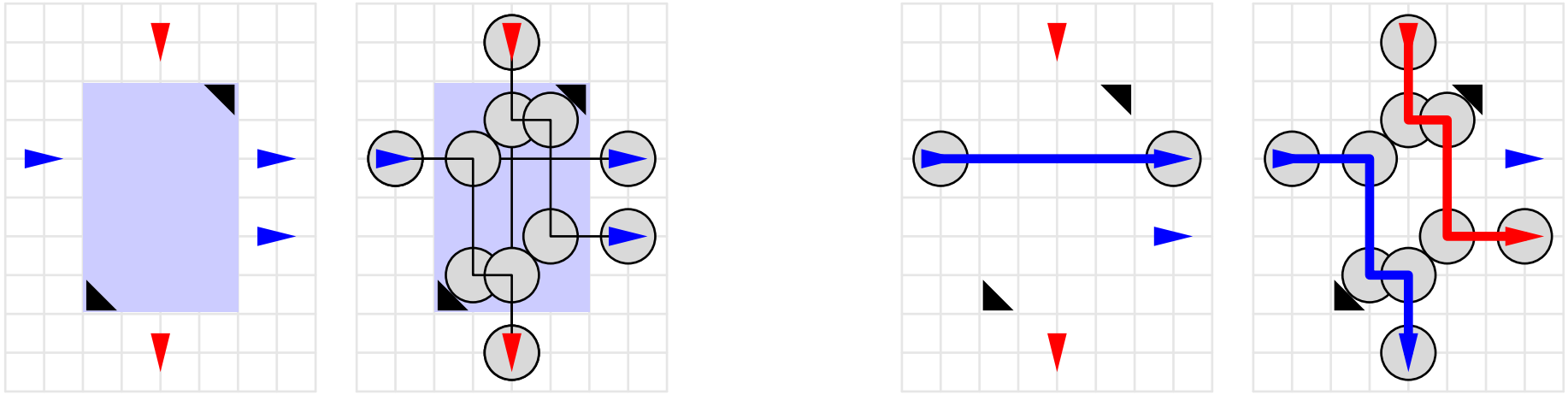


Convex hull of points with $y_2 = 1$

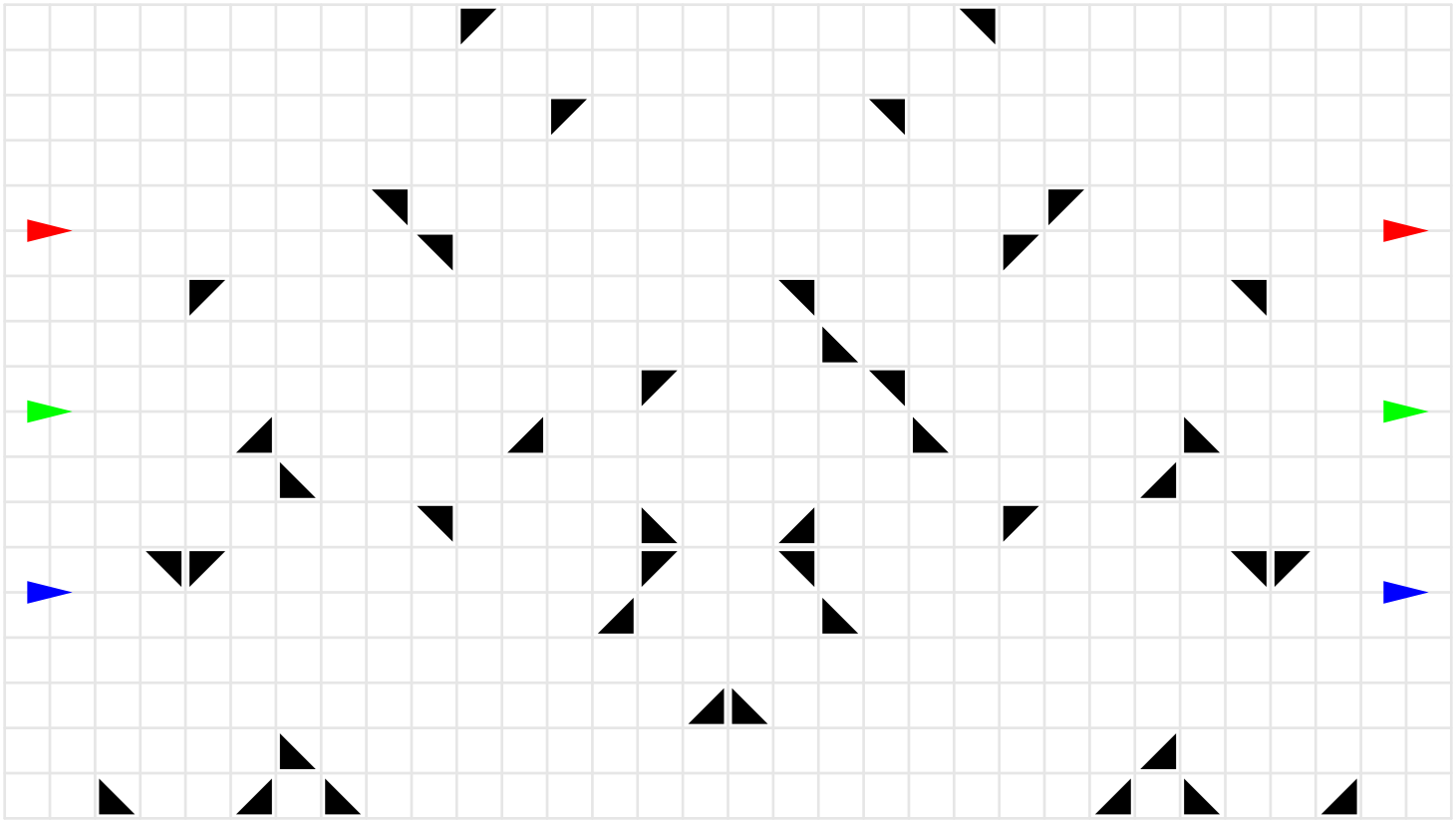
Excursion: Billiard Ball Computer

- A **billiard-ball computer** (a.k.a. **conservative logic circuit**) is an idealized model of a reversible mechanical computer that simulates the computations of circuits by the movements of spherical billiard balls in a frictionless environment [Fredkin and Toffoli 1982].

Switch (control output location of an input ball by another ball;
equivalent to an AND gate in its second/lower output):

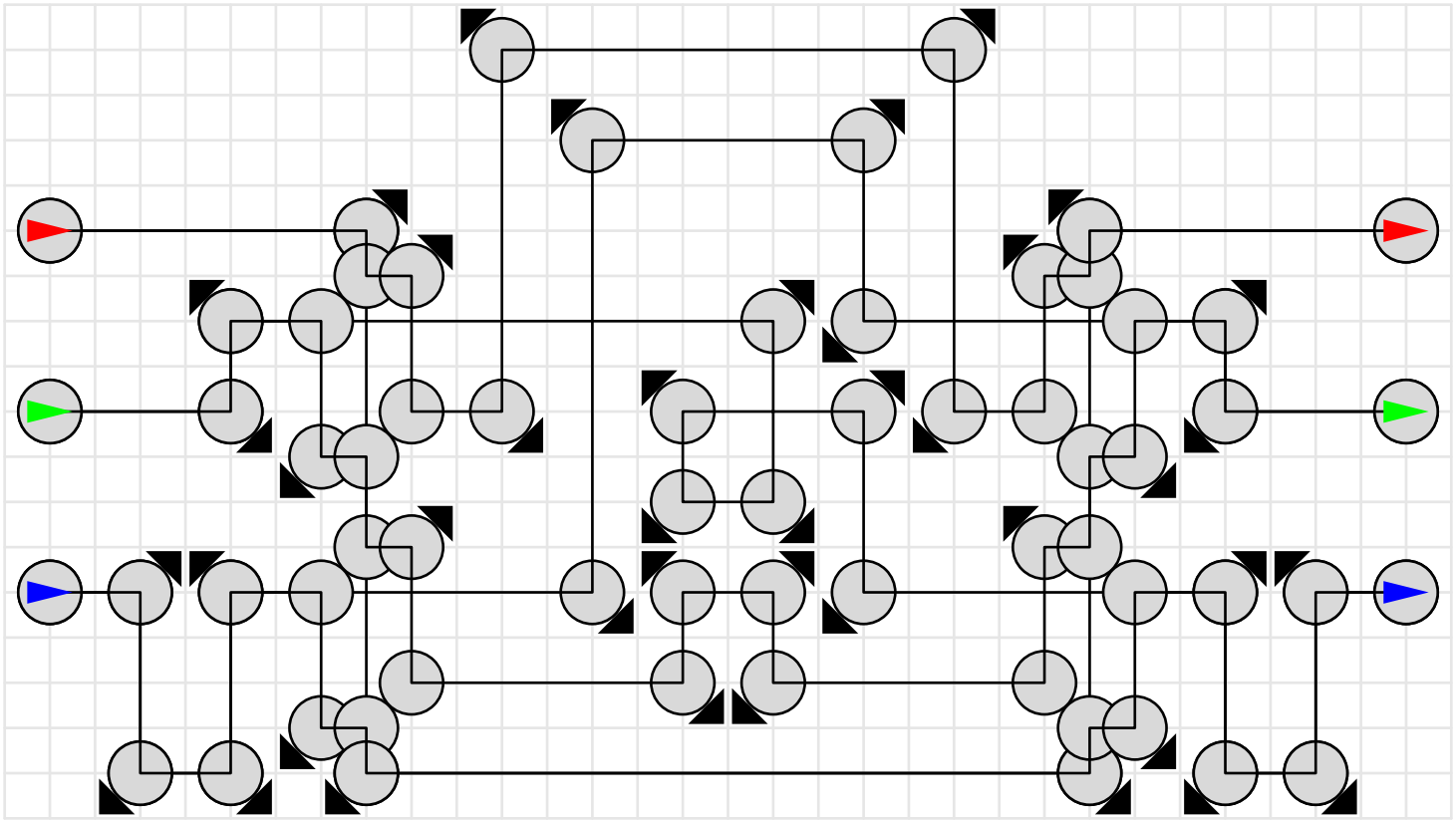


Excursion: The (Negated) Fredkin Gate



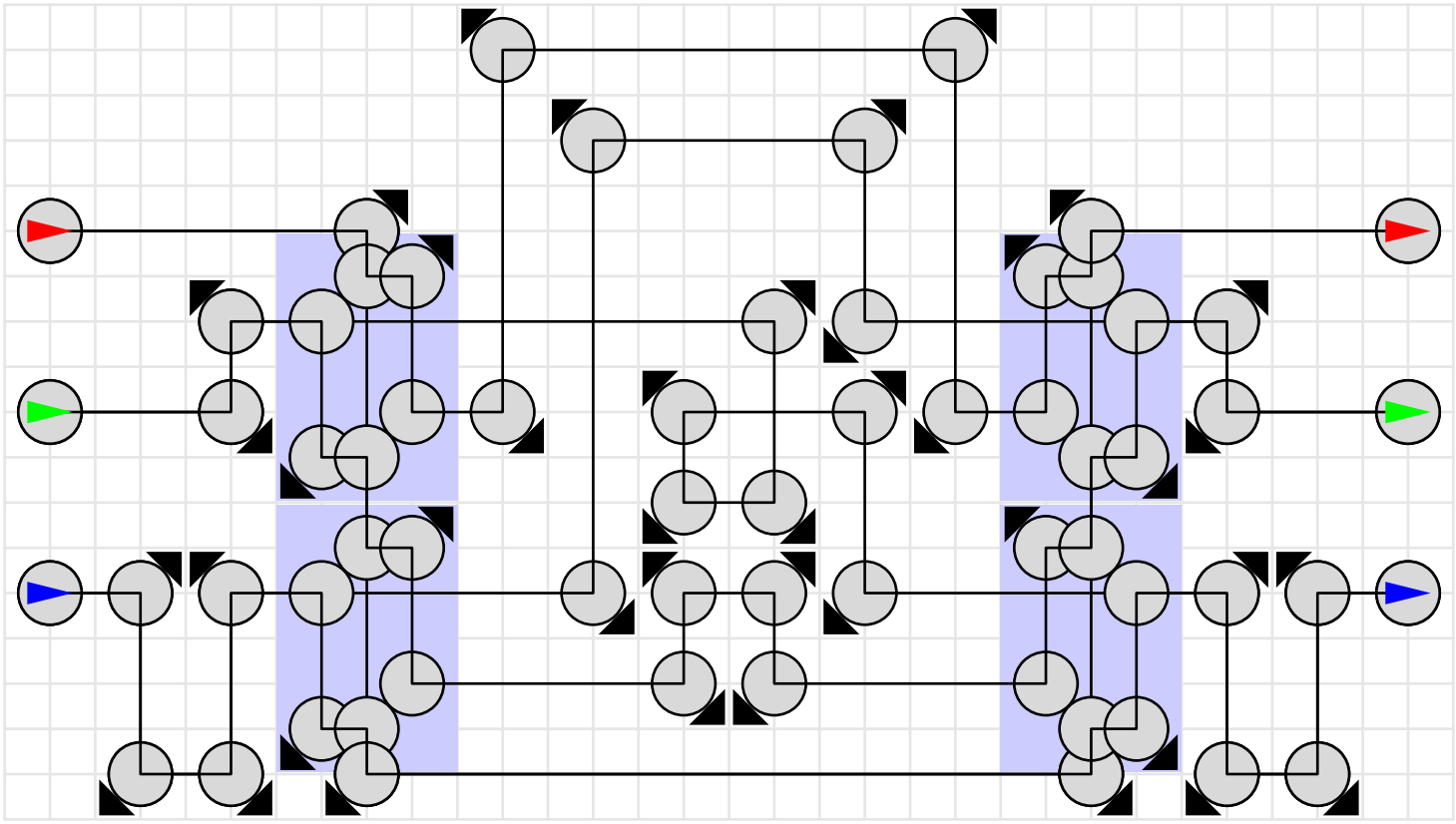
- Only reflectors (black) and inputs/outputs (color) shown.
- Meaning of switch variable is negated compared to value table on previous slide: 0 switches inputs, 1 passes them through.

Excursion: The (Negated) Fredkin Gate



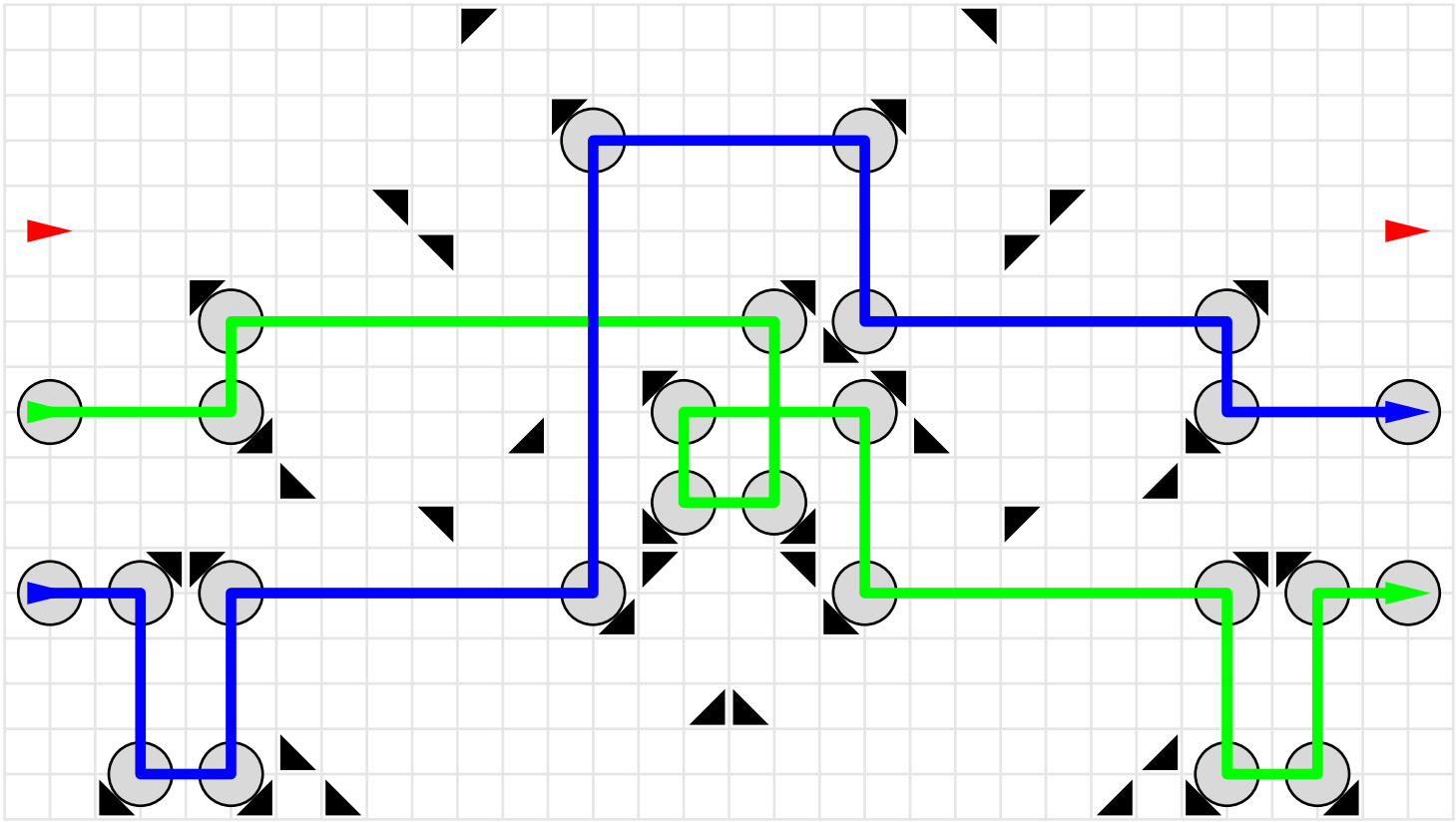
- All possible paths of billiard-balls through this gate are shown.
  Inputs/outputs are in color; red: switch, green: x_1/y_1, blue: x_2/y_2.
- Billiard-balls are shown in places where their trajectories may change direction.

Excursion: The (Negated) Fredkin Gate



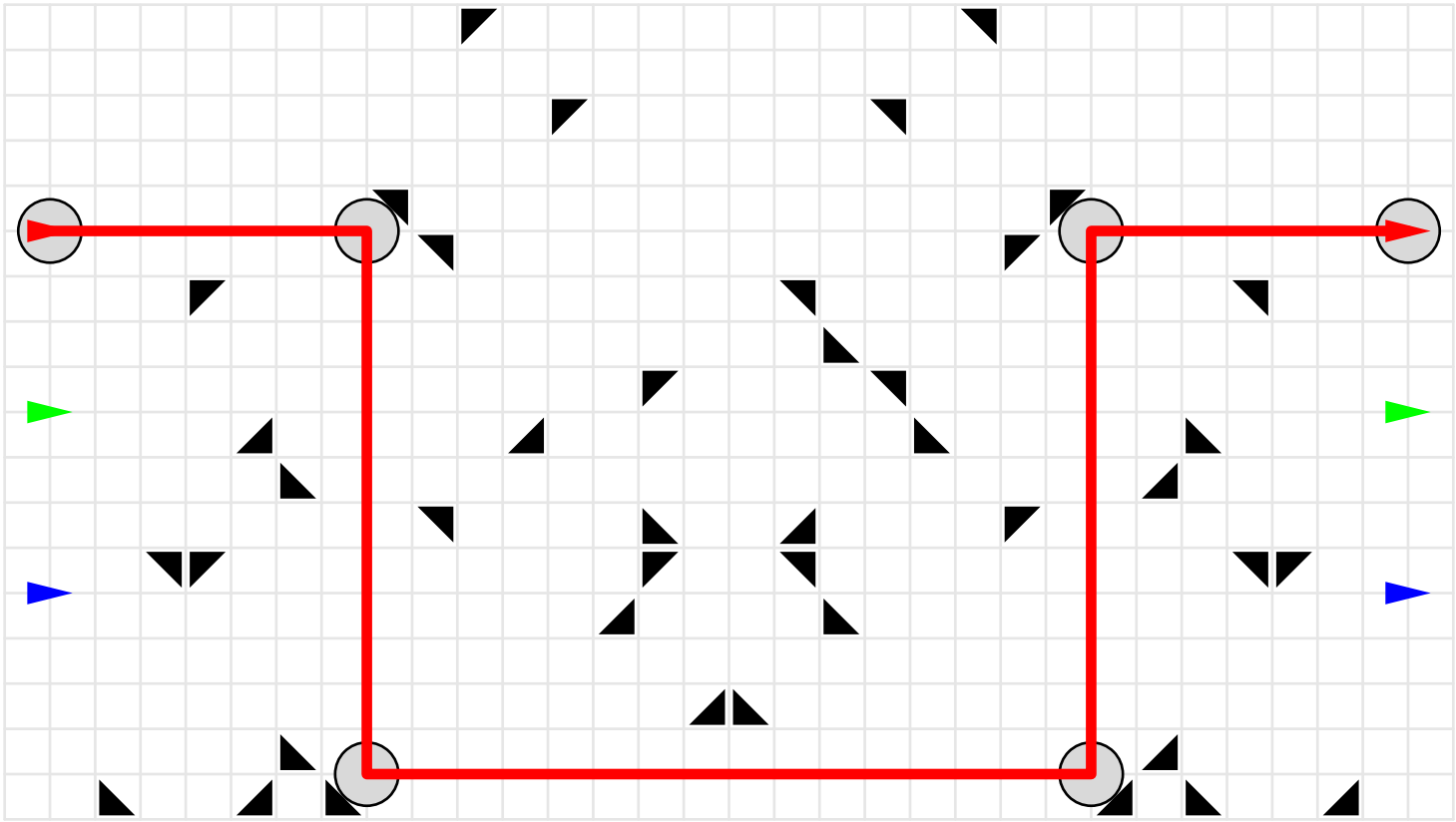
- This Fredkin gate implementation contains four switches (shown in light blue).
- The other parts serve the purpose to equate the travel times of the balls through the gate.

Excursion: The (Negated) Fredkin Gate



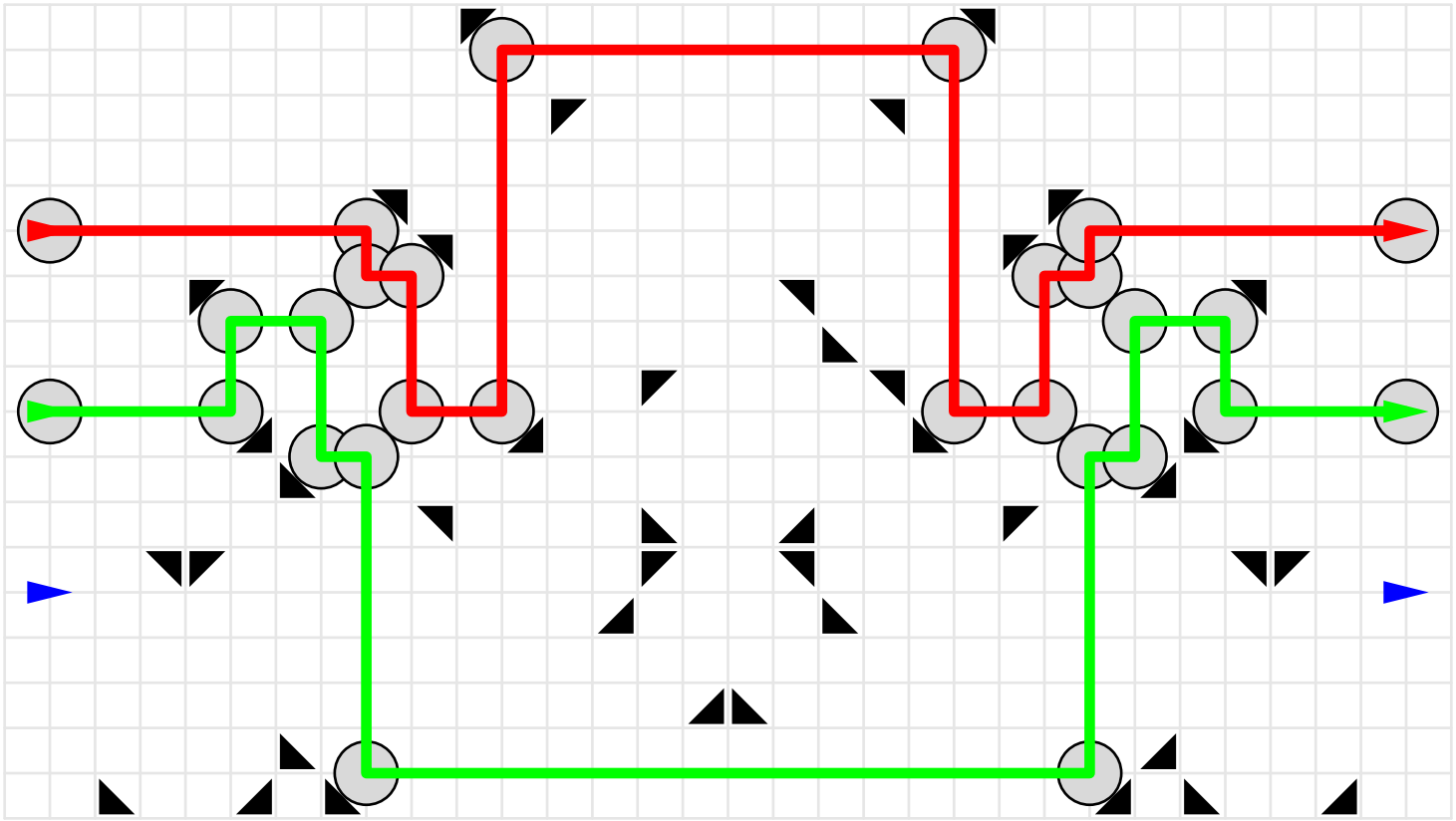
- Reminder: in this implementation the meaning of the switch variable is negated.
- Switch is zero (no ball enters at red arrow):
  blue and green balls switch to the other output.

Excursion: The (Negated) Fredkin Gate



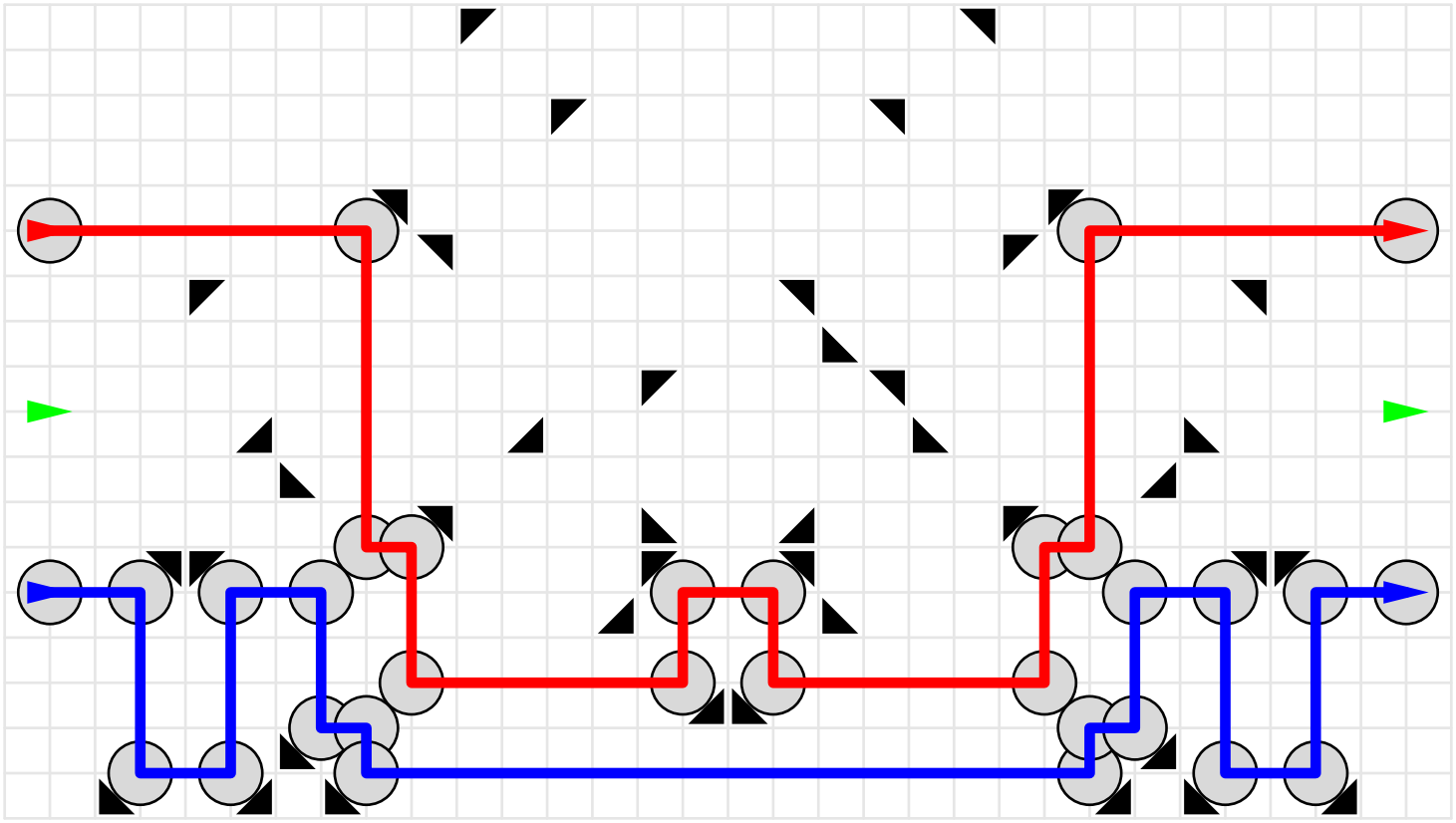
- If a ball is entered at the switch input, it is passed through.
- Since there are no other inputs, the other outputs remain 0 (no ball in, no ball out).

Excursion: The (Negated) Fredkin Gate



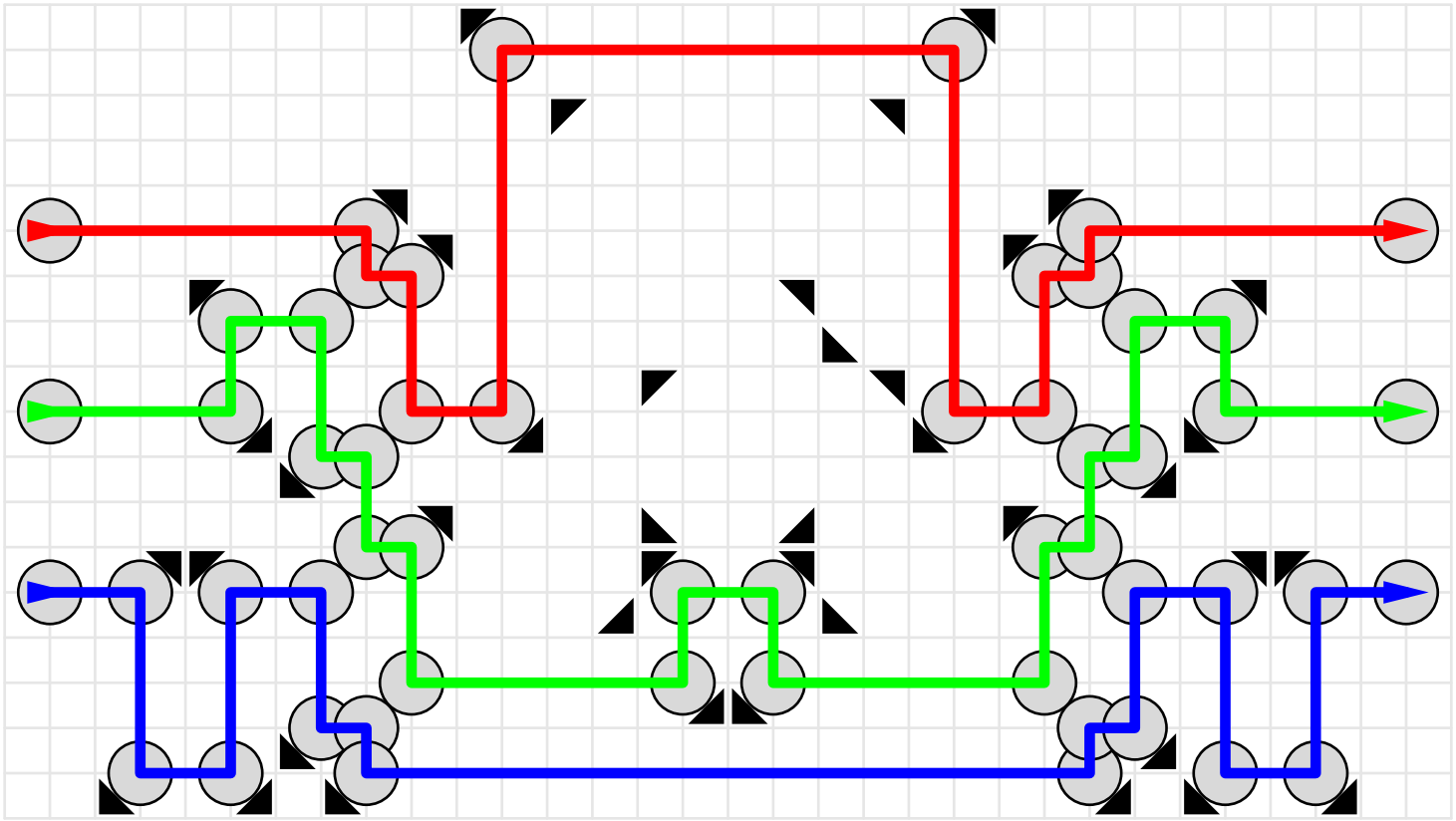
- A ball is entered at the switch input and passed through.
- A ball entered at the first input x_1 is also passed through to output y_1. (Note the symmetry of the trajectories.)

Excursion: The (Negated) Fredkin Gate



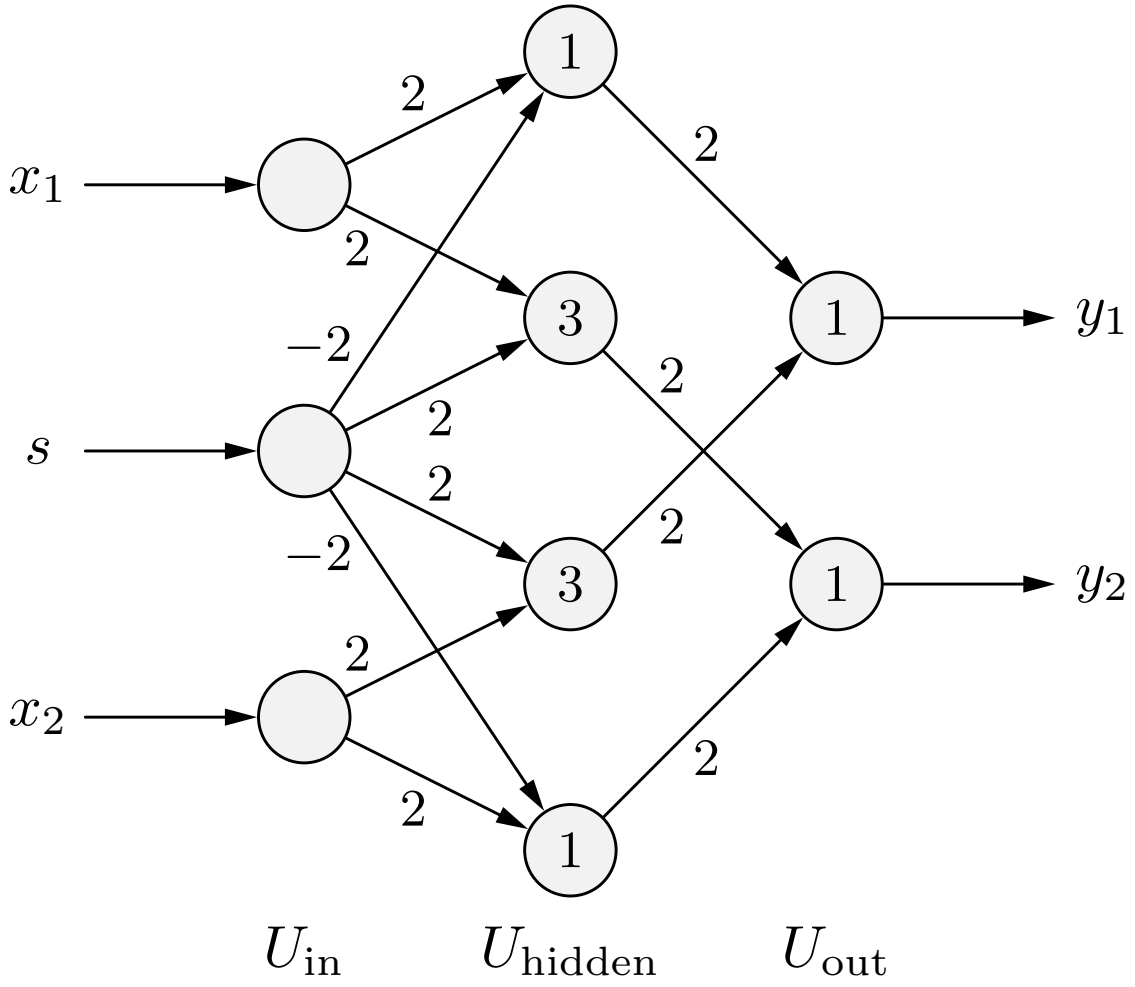
- A ball is entered at the switch input and passed through.
- A ball entered at the second input x_2 is also passed through to output y_2. (Note the symmetry of the trajectories.)

Excursion: The (Negated) Fredkin Gate



- A ball is entered at the switch input and passed through.
- Ball entered at both inputs x_1 and x_2 and are also passed through to the outputs y_1 and y_2. (Note the symmetry of the trajectories.)

Multi-layer Perceptrons: Fredkin Gate



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

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

Why Non-linear Activation Functions?

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

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

If the activation functions are linear, that is,

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

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

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

where

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

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

Combining these computations we get

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

and thus

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

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

Why Non-linear Activation Functions?

Therefore we have

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

and

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

for the computations of two consecutive layers U_2 and U_3.

These two computations can be combined into

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

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

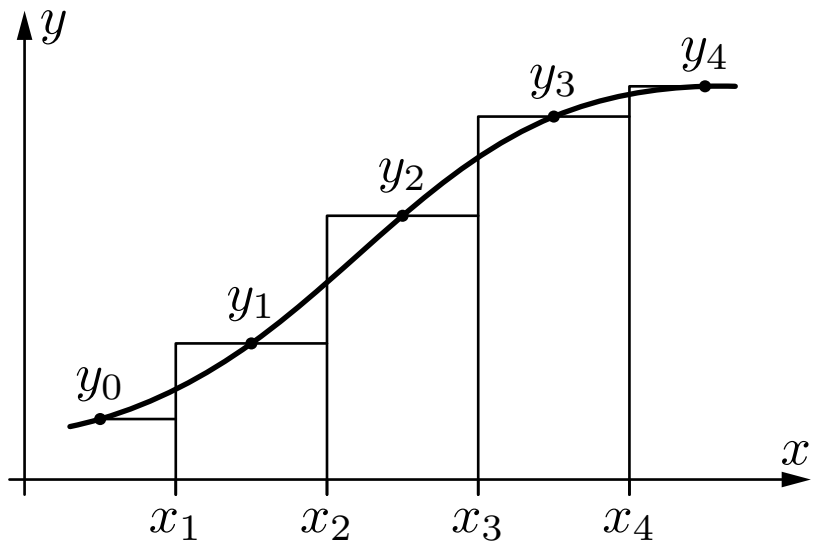
Result: With linear activation and output functions any multi-layer perceptron can be reduced to a two-layer perceptron.

Multi-layer Perceptrons: Function Approximation

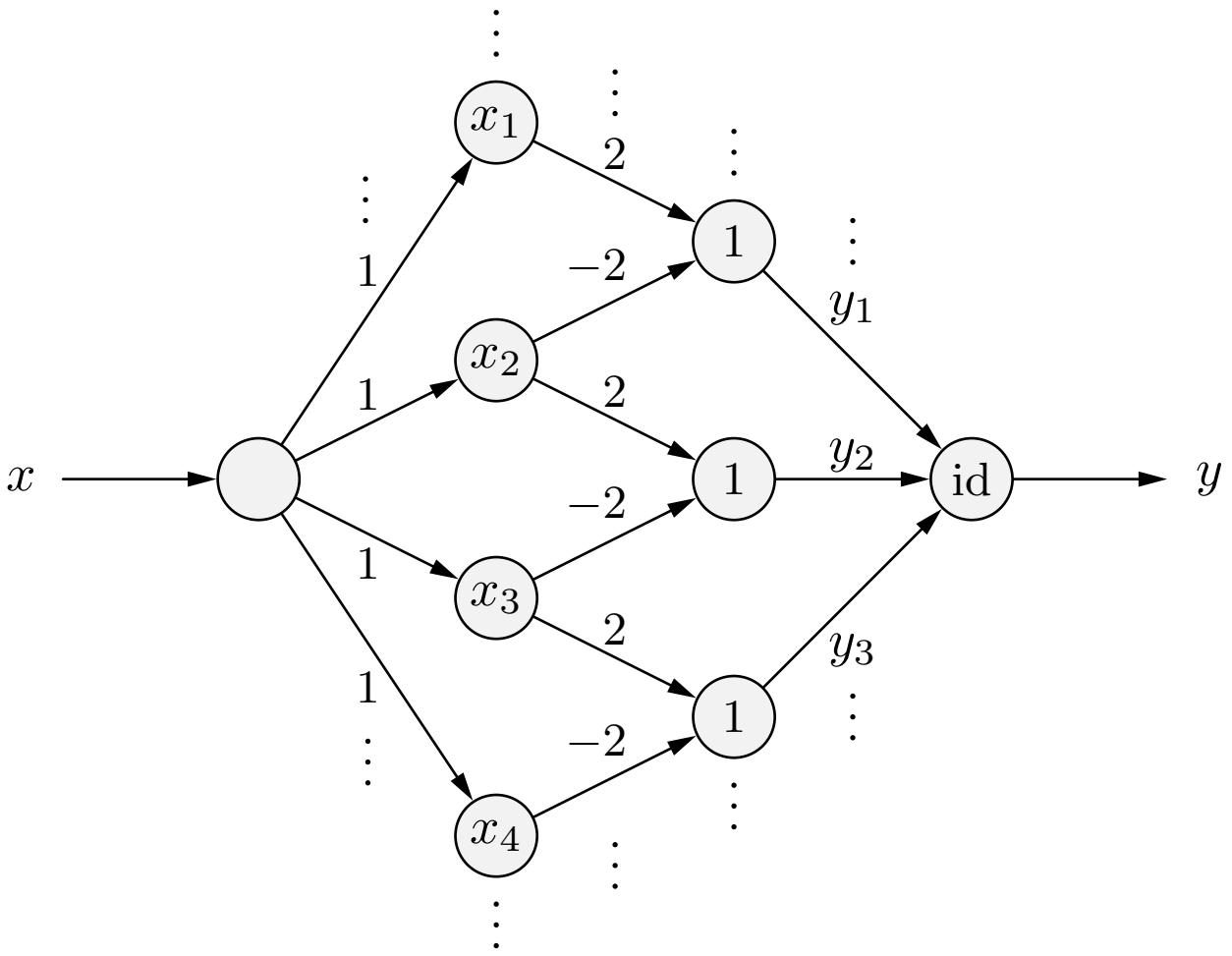
- Up to now: representing and learning Boolean functions $f : \{0,1\}^n \to \{0,1\}$.
- Now: representing and learning real-valued functions $f : \mathbb{R}^n \to \mathbb{R}$.

General idea of function approximation:

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



Multi-layer Perceptrons: Function Approximation

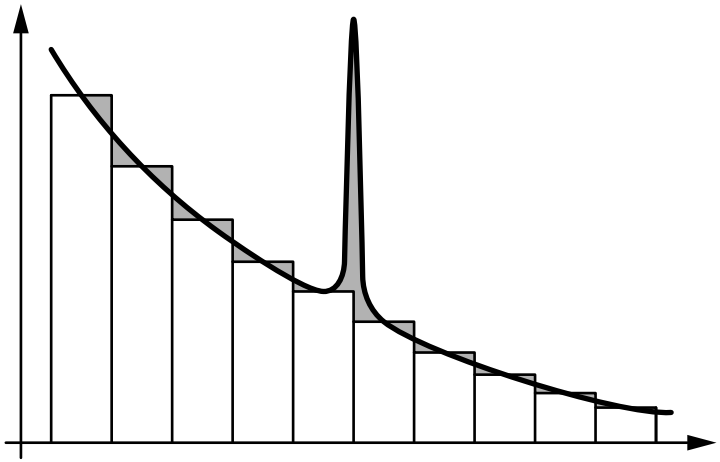


A neural network that computes the step function shown on the preceding slide.
According to the input value only one step is active at any time.
The output neuron has the identity as its activation and output functions.

Multi-layer 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).

Multi-layer Perceptrons as Universal Approximators

Universal Approximation Theorem [Hornik 1991]:

Let $\varphi(\cdot)$ be a continuous, bounded and nonconstant function,
let X denote an arbitrary compact subset of $\mathbb{R}^m$, and
let $C(X)$ denote the space of continuous functions on X.

Given any function $f \in C(X)$ and $\varepsilon > 0$, there exists an integer N, real constants $v_i, \theta_i \in \mathbb{R}$ and real vectors $\vec{w}_i \in \mathbb{R}^m$, $i = 1, \ldots, N$, such that we may define

$$F(\vec{x}) = \sum_{i=1}^{N} v_i \, \varphi\Big(\vec{w}_i^{\top} \vec{x} - \theta_i\Big)$$

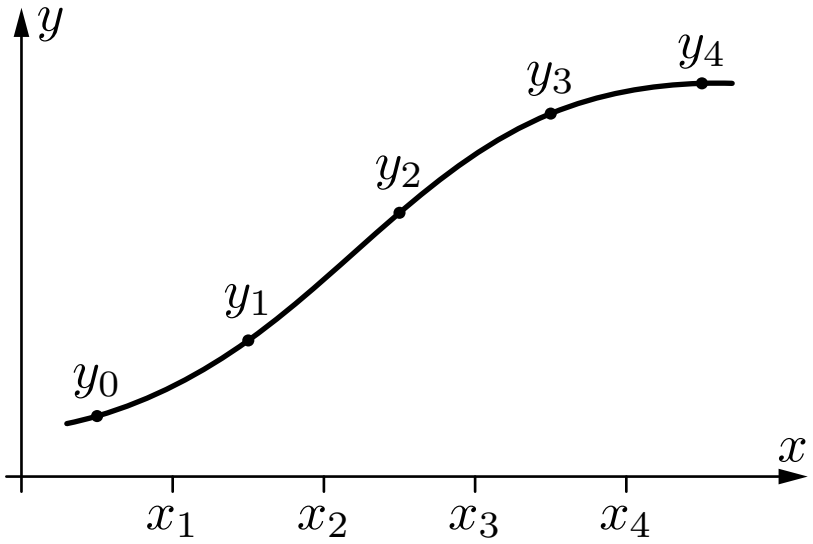
as an approximate realization of the function f where f is independent of φ. That is,

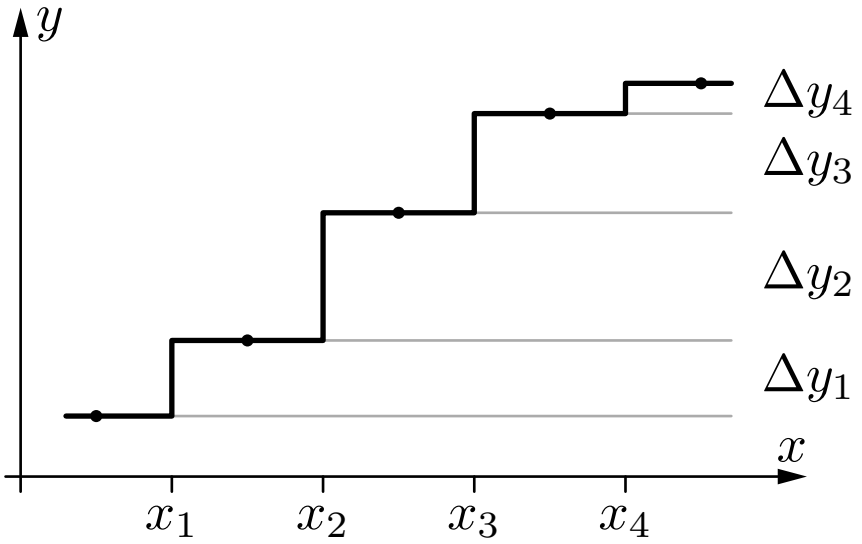
$$|F(\vec{x}) - f(\vec{x})| < \varepsilon$$

for all $\vec{x} \in X$. In other words, functions of the form $F(\vec{x})$ are dense in $C(X)$.

Note that it is *not* the shape of the activation function, but the layered structure of the feedforward network that renders multi-layer perceptrons universal approximators.

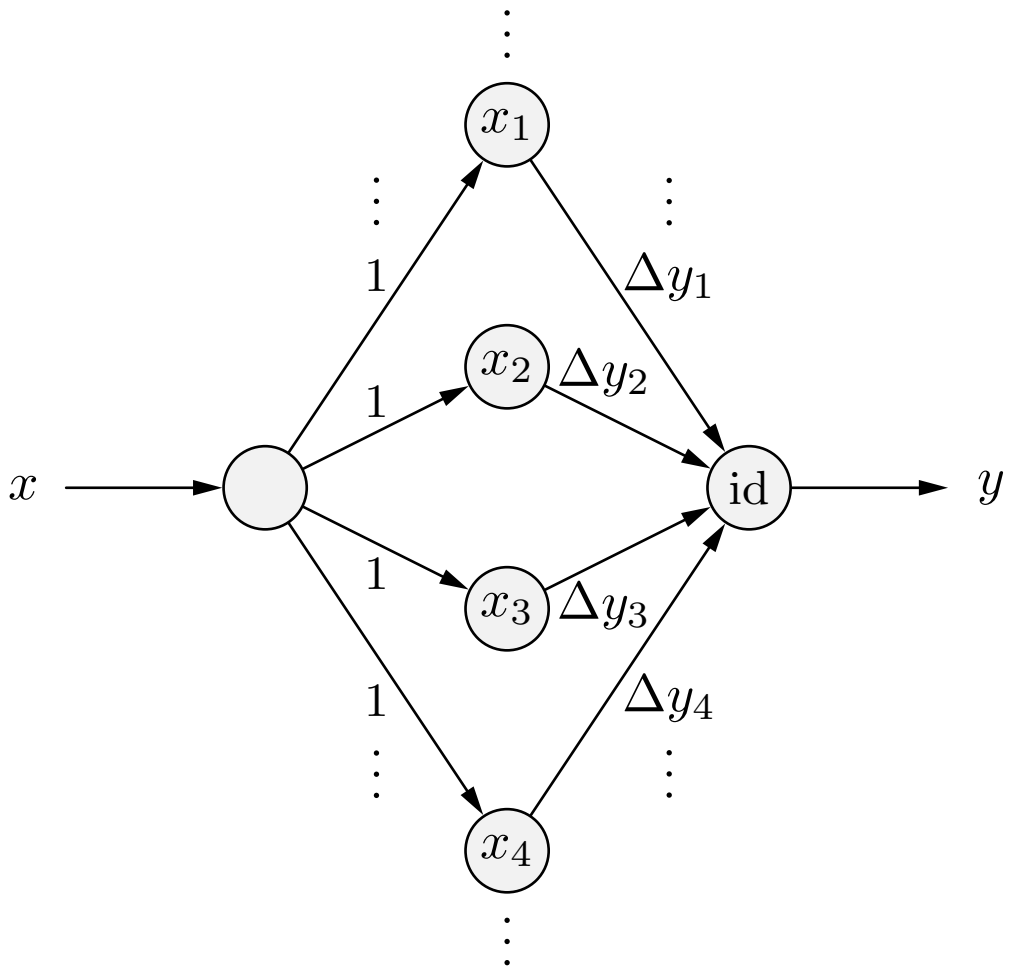
Multi-layer Perceptrons: Function Approximation





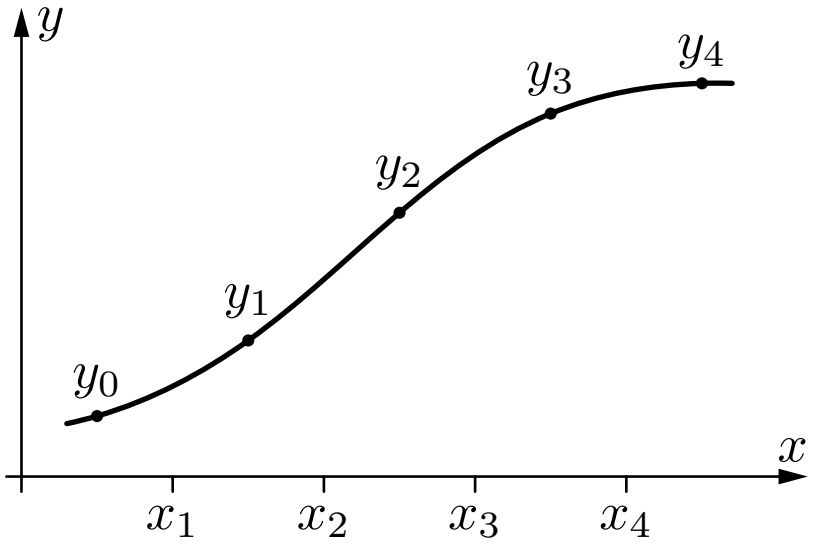
By using relative step heights one layer can be saved.

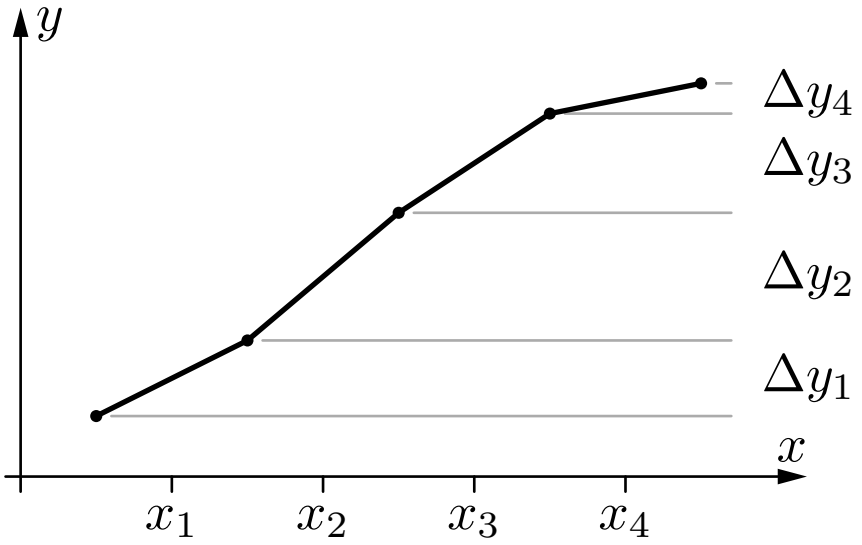
Multi-layer Perceptrons: Function Approximation



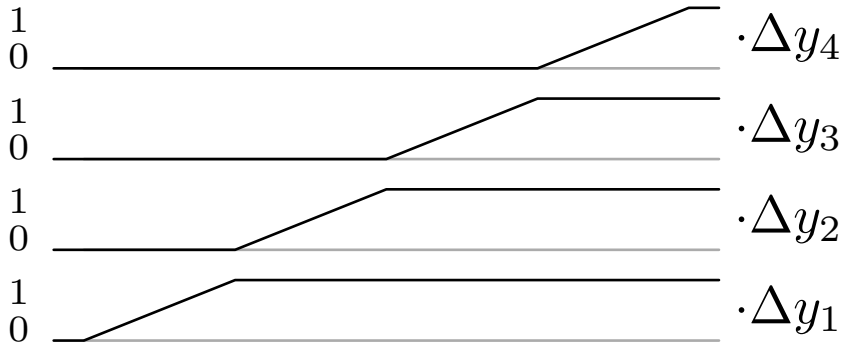
A neural network that computes the step function shown on the preceding slide. The output neuron has the identity as its activation and output functions.

Multi-layer Perceptrons: Function Approximation

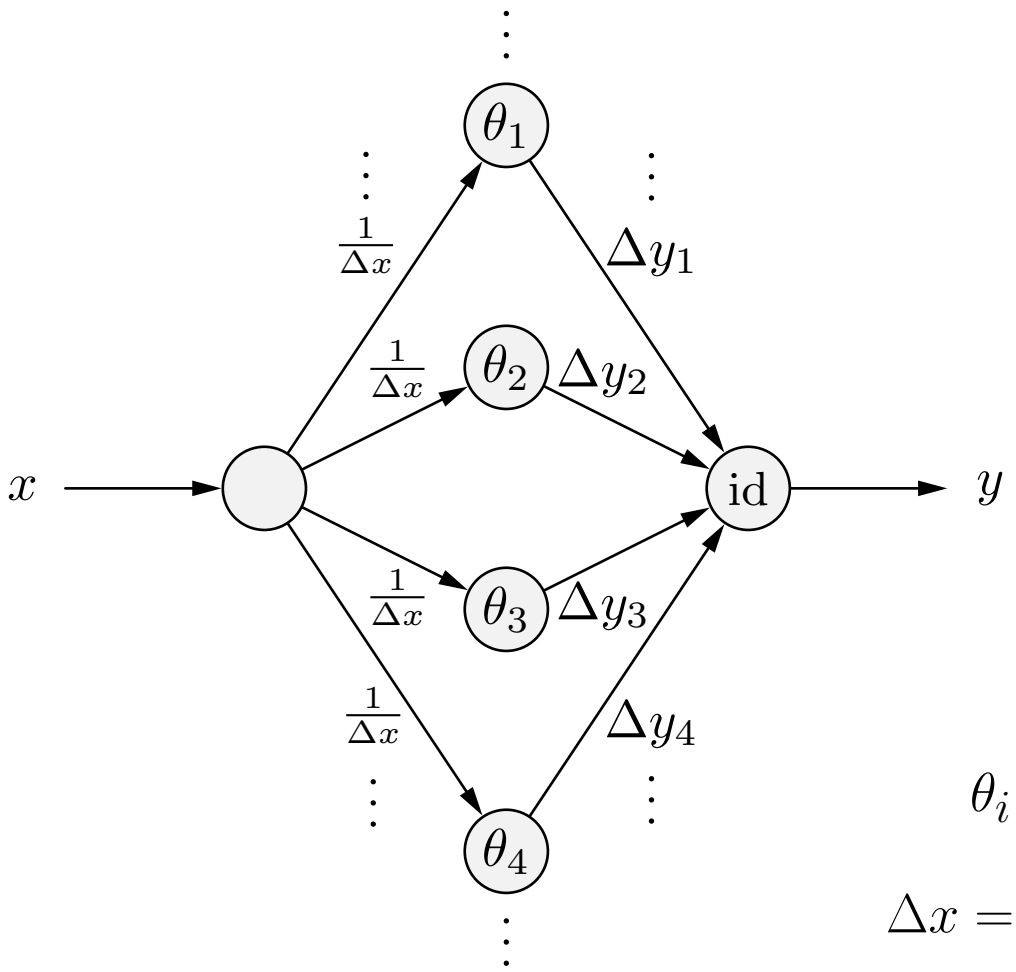




By using semi-linear functions the approximation can be improved.



Multi-layer Perceptrons: Function Approximation



$$\theta_i = \frac{x_i}{\Delta x}$$

$$\Delta x = x_{i+1} - x_i$$

A neural network that computes the step function shown on the preceding slide. The output neuron has the identity as its activation and output functions.

Mathematical Background: Regression

Mathematical Background: Linear Regression

Training neural networks is closely related to regression.

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

Approach: Minimize the sum of squared errors, that is,

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

Necessary conditions for a minimum
(a.k.a. Fermat's theorem, after Pierre de Fermat, 1601–1665):

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

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

Mathematical Background: Linear Regression

Result of necessary conditions: System of so-called **normal equations**, that is,

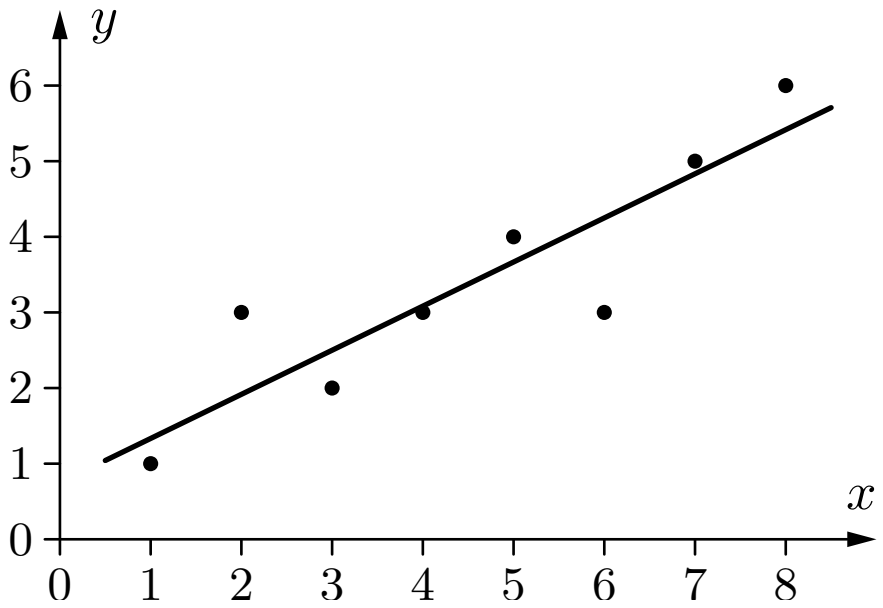
$$\begin{aligned} na + \left(\sum_{i=1}^{n} x_i\right) b &= \sum_{i=1}^{n} y_i, \\ \left(\sum_{i=1}^{n} x_i\right) a + \left(\sum_{i=1}^{n} x_i^2\right) b &= \sum_{i=1}^{n} x_i y_i. \end{aligned}$$

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

Linear Regression: Example

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

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



Mathematical Background: Polynomial Regression

Generalization to polynomials

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

Approach: Minimize the sum of squared errors, that is,

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

Necessary conditions for a minimum: All partial derivatives vanish, that is,

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

Mathematical Background: Polynomial Regression

System of normal equations for polynomials

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

- $m + 1$ linear equations for $m + 1$ unknowns $a_0, \ldots, a_m$.
- System can be solved with standard methods from linear algebra.
- Solution is unique unless the points lie exactly on a polynomial of lower degree.

Generalization to more than one argument

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

Approach: Minimize the sum of squared errors, that is,

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

Necessary conditions for a minimum: All partial derivatives vanish, that is,

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

Mathematical Background: Multilinear Regression

System of normal equations for several arguments

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

- 3 linear equations for 3 unknowns a, b, and c.
- System can be solved with standard methods from linear algebra.
- Solution is unique unless all data points lie on a straight line.

Multilinear Regression

General multilinear case:

$$y = f(x_1, \ldots, x_m) = a_0 + \sum_{k=1}^{m} a_k x_k$$

Approach: Minimize the sum of squared errors, that is,

$$F(\vec{a}) = (\mathbf{X}\vec{a} - \vec{y})^\top (\mathbf{X}\vec{a} - \vec{y}),$$

where

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \ldots & x_{m1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{1n} & \ldots & x_{mn} \end{pmatrix}, \qquad \vec{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \qquad \text{and} \qquad \vec{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix}$$

Necessary conditions for a minimum:

$$\vec{\nabla}_{\vec{a}}\, F(\vec{a}) = \vec{\nabla}_{\vec{a}}\, (\mathbf{X}\vec{a} - \vec{y})^\top (\mathbf{X}\vec{a} - \vec{y}) = \vec{0}$$

Multilinear Regression

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

$$\vec{\nabla}_{\vec{a}} = \left(\frac{\partial}{\partial a_0}, \ldots, \frac{\partial}{\partial a_m} \right)$$

  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:

$$\begin{aligned}
\vec{\nabla}_{\vec{a}} F(\vec{a}) &= \vec{\nabla}_{\vec{a}} \left((\mathbf{X}\vec{a} - \vec{y})^\top (\mathbf{X}\vec{a} - \vec{y}) \right) \\
&= \left(\vec{\nabla}_{\vec{a}} \left(\mathbf{X}\vec{a} - \vec{y} \right) \right)^\top (\mathbf{X}\vec{a} - \vec{y}) + ((\mathbf{X}\vec{a} - \vec{y})^\top \left(\vec{\nabla}_{\vec{a}} \left(\mathbf{X}\vec{a} - \vec{y} \right) \right))^\top \\
&= \left(\vec{\nabla}_{\vec{a}} \left(\mathbf{X}\vec{a} - \vec{y} \right) \right)^\top (\mathbf{X}\vec{a} - \vec{y}) + \left(\vec{\nabla}_{\vec{a}} \left(\mathbf{X}\vec{a} - \vec{y} \right) \right)^\top (\mathbf{X}\vec{a} - \vec{y}) \\
&= 2\mathbf{X}^\top (\mathbf{X}\vec{a} - \vec{y}) \\
&= 2\mathbf{X}^\top \mathbf{X}\vec{a} - 2\mathbf{X}^\top \vec{y} \quad = \quad \vec{0}
\end{aligned}$$

Multilinear Regression

Necessary condition for a minimum therefore:

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

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

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

This system has a solution unless $\mathbf{X}^\top \mathbf{X}$ is singular. If it is regular, 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 straighforward:
Simply add the desired products of powers to the matrix $\mathbf{X}$.

Mathematical Background: Logistic Regression

Generalization to non-polynomial functions

$$\text{Simple example:} \qquad y = ax^b$$

Idea: Find transformation to linear/polynomial case.

$$\text{Transformation for the above example:} \qquad \ln y = \ln a + b \cdot \ln x.$$

Special case: **logistic function**

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

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

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

Logistic Regression: Example

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

Transform the data with

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

The transformed data points are

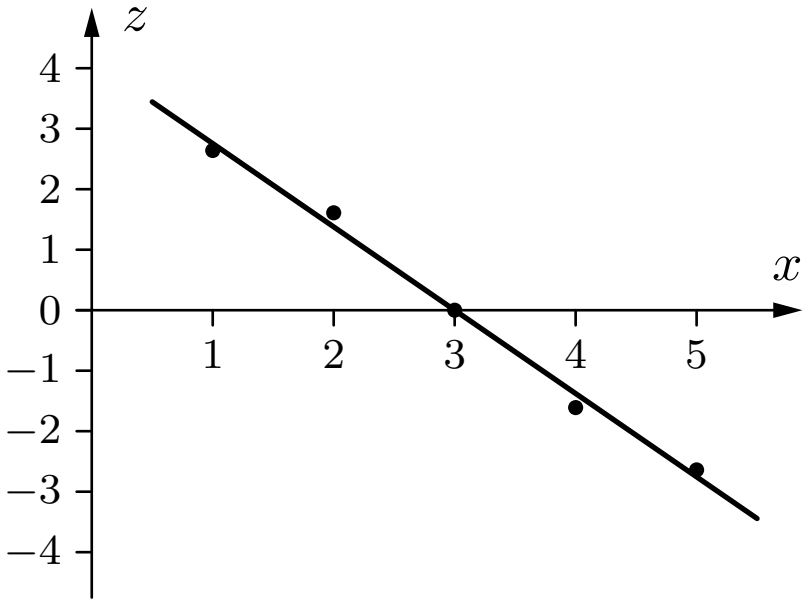
x	1	2	3	4	5
z	2.64	1.61	0.00	-1.61	-2.64

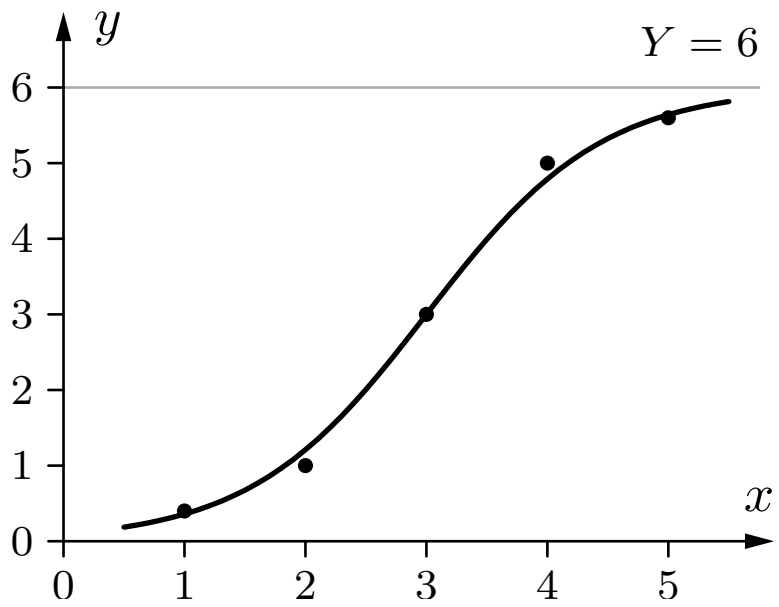
The resulting regression line and therefore the desired function are

$$z \approx -1.3775x + 4.133 \qquad \text{and} \qquad y \approx \frac{6}{1 + e^{-1.3775x+4.133}}.$$

Attention: Note that the error is minimized only in the transformed space! Therefore the function in the original space may not be optimal!

Logistic Regression: Example





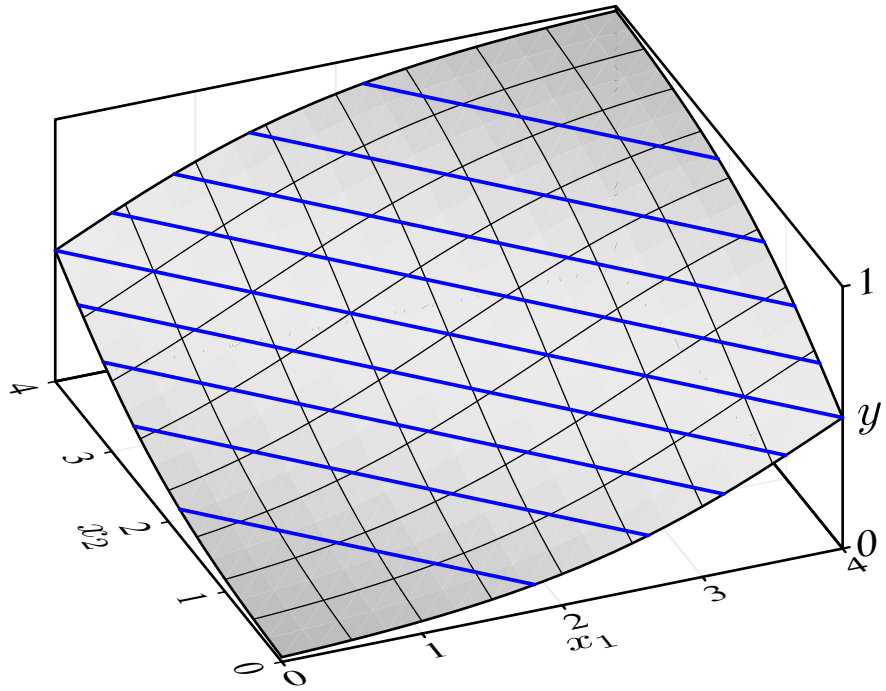
The logistic regression function can be computed by a single neuron with

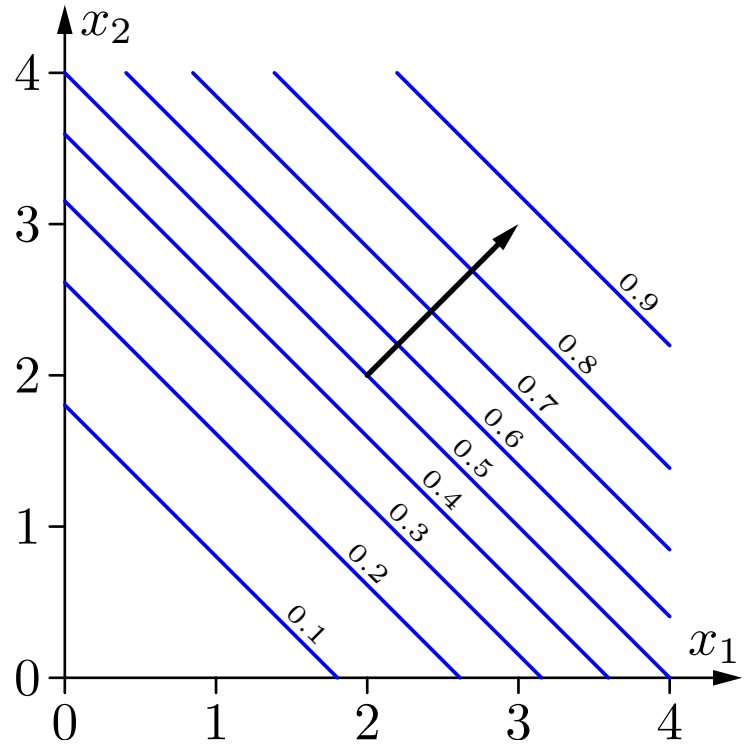
- network input function $f_{\text{net}}(x) \equiv wx$ with $w \approx -1.3775$,
- activation function $f_{\text{act}}(\text{net}, \theta) \equiv \left(1 + e^{-(\text{net} - \theta)}\right)^{-1}$ with $\theta \approx 4.133$ and
- output function $f_{\text{out}}(\text{act}) \equiv 6\,\text{act}$.

Logistic Function: Two-dimensional Example

Example logistic function for two arguments x_1 and x_2:

$$y = \frac{1}{1 + \exp(4 - x_1 - x_2)} = \frac{1}{1 + \exp\Big(4 - (1,1)^\top (x_1, x_2)\Big)}$$





The blue lines have show where the logistic function has a certain value in $\{0.1, \ldots, 0.9\}$.

Logistic Regression: Two Class Problems

- Let C be a class attrib., $\mathrm{dom}(C) = \{c_1, c_2\}$, and $\vec{X}$ an m-dim. random vector.
  Let $P(C = c_1 \mid \vec{X} = \vec{x}) = p(\vec{x})$ and $P(C = c_2 \mid \vec{X} = \vec{x}) = 1 - p(\vec{x})$.
- **Given:** A set of data points $\mathbf{X} = \{\vec{x}_1, \ldots, \vec{x}_n\}$ (realizations of $\vec{X}$), each of which belongs to one of the two classes c_1 and c_2.
- **Desired:** A simple description of the function $p(\vec{x})$.
- **Approach:** Describe p by a logistic function:

$$p(\vec{x}) \quad = \quad \frac{1}{1 + e^{a_0 + \vec{a}\vec{x}}} \quad = \quad \frac{1}{1 + \exp\left(a_0 + \sum_{i=1}^{m} a_i x_i\right)}$$

  Apply logit transformation to $p(x)$:

$$\ln\left(\frac{1 - p(\vec{x})}{p(\vec{x})}\right) \quad = \quad a_0 + \vec{a}\vec{x} \quad = \quad a_0 + \sum_{i=1}^{m} a_i x_i$$

  The values $p(\vec{x}_i)$ may be obtained by kernel estimation (see next slide).

Logistic Regression: Kernel Estimation

- **Idea:** Define an "influence function" (kernel), which describes how strongly a data point influences the probability estimate for neighboring points.
- Common choice for the kernel function: **Gaussian function**

$$K(\vec{x}, \vec{y}) \quad = \quad \frac{1}{(2\pi\sigma^2)^{\frac{m}{2}}} \exp\left(-\frac{(\vec{x}-\vec{y})^\top(\vec{x}-\vec{y})}{2\sigma^2}\right)$$

- Kernel estimate of probability density given a data set $\mathbf{X} = \{\vec{x}_1, \ldots, \vec{x}_n\}$:

$$\hat{f}(\vec{x}) \quad = \quad \frac{1}{n}\sum_{i=1}^{n} K(\vec{x}, \vec{x}_i).$$

- Kernel estimation applied to a two class problem:

$$\hat{p}(\vec{x}) \quad = \quad \frac{\sum_{i=1}^{n} c(\vec{x}_i) K(\vec{x}, \vec{x}_i)}{\sum_{i=1}^{n} K(\vec{x}, \vec{x}_i)}.$$

  It is $c(\vec{x}_i) = 1$ if x_i belongs to class c_1 and $c(\vec{x}_i) = 0$ otherwise.

Training Multi-layer Perceptrons

Training Multi-layer Perceptrons: Gradient Descent

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

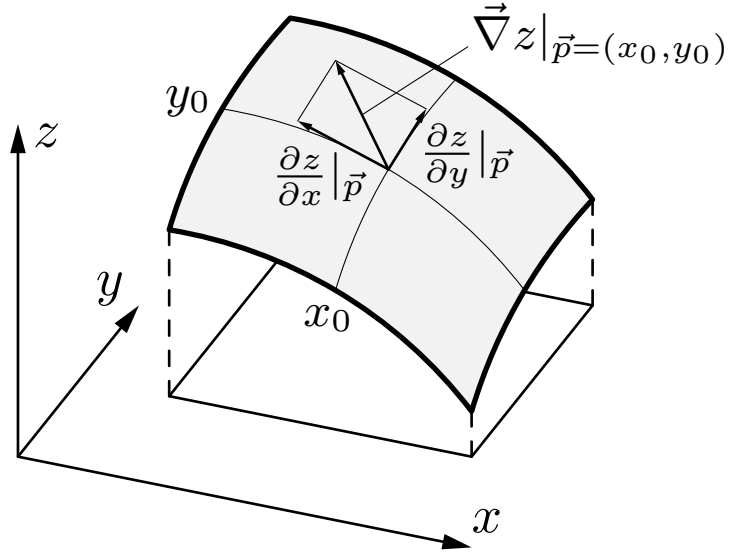


Illustration of the gradient of a real-valued function $z = f(x, y)$ at a point (x_0, y_0).
It is $\vec{\nabla} z|_{(x_0,y_0)} = \left(\frac{\partial z}{\partial x}|_{x_0}, \frac{\partial z}{\partial y}|_{y_0}\right)$. ($\vec{\nabla}$ is a differential operator called "nabla" or "del".)

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
(here and in the following: extended weight vector $\vec{w}_u = (-\theta_u, w_{up_1}, \ldots, w_{up_n})$):

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

Exploit 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:

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

Since $\operatorname{net}_u^{(l)} = \vec{w}_u^\top \vec{\operatorname{in}}_u^{(l)}$ (note: extended input vector $\vec{\operatorname{in}}_u^{(l)} = \left(1, \operatorname{in}_{p_1 u}^{(l)}, \ldots, \operatorname{in}_{p_n u}^{(l)}\right)$), we have for the second factor

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

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

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

that is, the sum of the errors over all output neurons.

Gradient Descent: Formal Approach

Therefore we have

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

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

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

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

Gradient Descent: Formal Approach

Distinguish two cases:

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

In the first case we have

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

Therefore we have for the gradient

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

and thus for the weight change

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

Gradient Descent: Formal Approach

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

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

Consider the special case with

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

The first assumption yields

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

Gradient Descent: Formal Approach

For a logistic activation function we have

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

and therefore

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

The resulting weight change is therefore

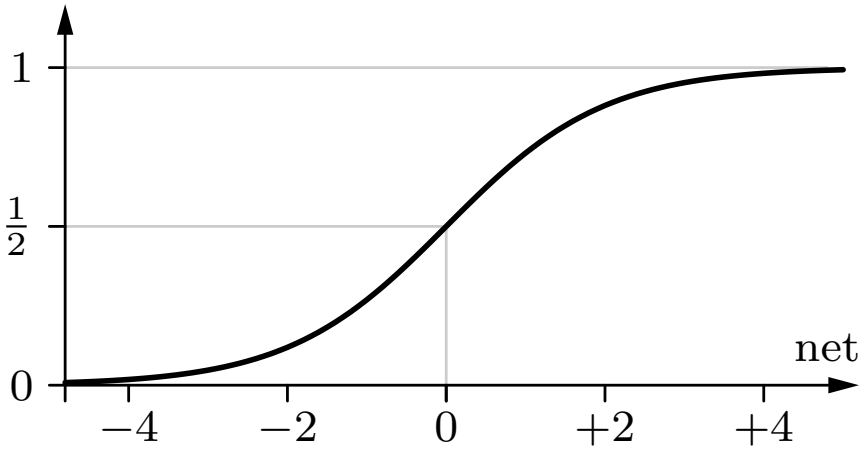
$$\Delta\vec{w}_u^{(l)} = \eta\left(o_u^{(l)}-\text{out}_u^{(l)}\right)\text{out}_u^{(l)}\left(1-\text{out}_u^{(l)}\right)\,\vec{\text{in}}_u^{(l)},$$

which makes the computations very simple.

Gradient Descent: Formal Approach

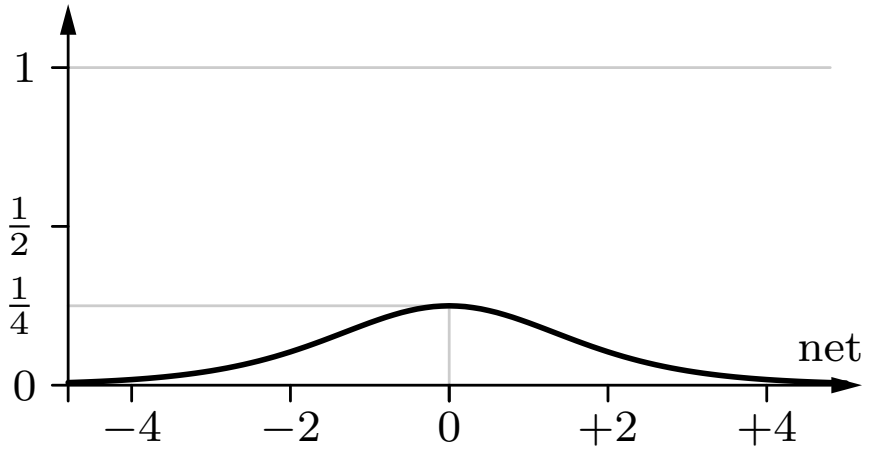
logistic activation function:

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



derivative of logistic function:

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

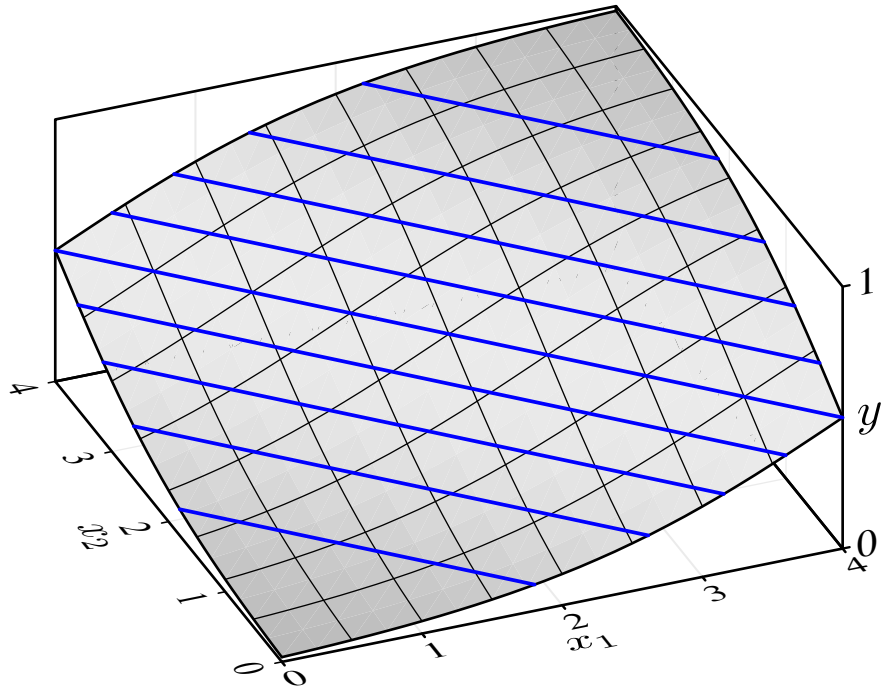


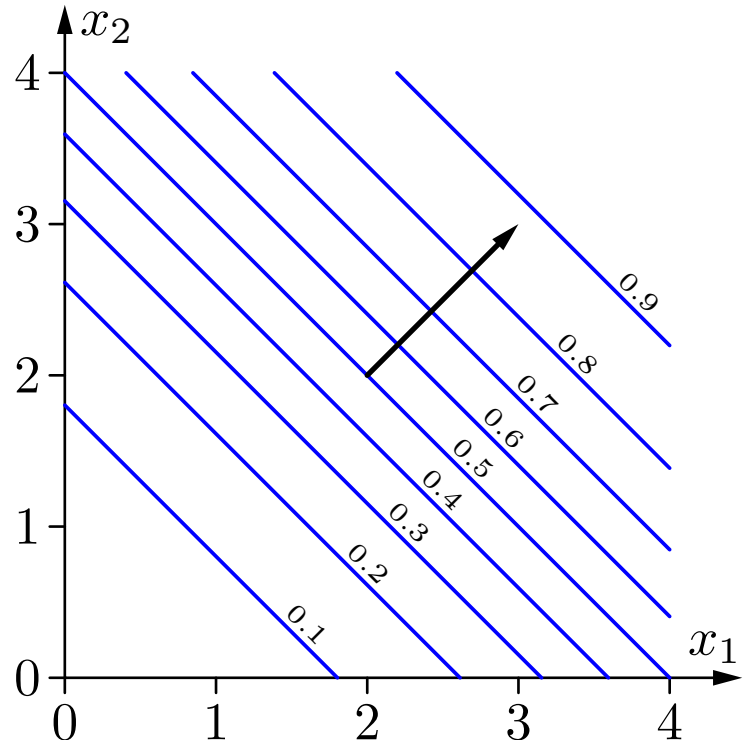
- If a logistic activation function is used (shown on left), the weight changes are proportional to $\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right)$ (shown on right; see preceding slide).
- Weight changes are largest, and thus the training speed highest, in the vicinity of $\text{net}_u^{(l)} = 0$. Far away from $\text{net}_u^{(l)} = 0$, the gradient becomes (very) small ("saturation regions") and thus training (very) slow.

Reminder: Two-dimensional Logistic Function

Example logistic function for two arguments x_1 and x_2:

$$y = \frac{1}{1 + \exp(4 - x_1 - x_2)} = \frac{1}{1 + \exp\Big(4 - (1,1)^\top (x_1, x_2)\Big)}$$





The blue lines have show where the logistic function has a certain value in $\{0.1, \ldots, 0.9\}$.

Error Backpropagation

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

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

We apply the chain rule to obtain

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

Exchanging the sums yields

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

Error Backpropagation

Consider the network input

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

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

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

The result is the recursive equation (error backpropagation)

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

Error Backpropagation

The resulting formula for the weight change is

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

Consider again the special case with

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

The resulting formula for the weight change is then

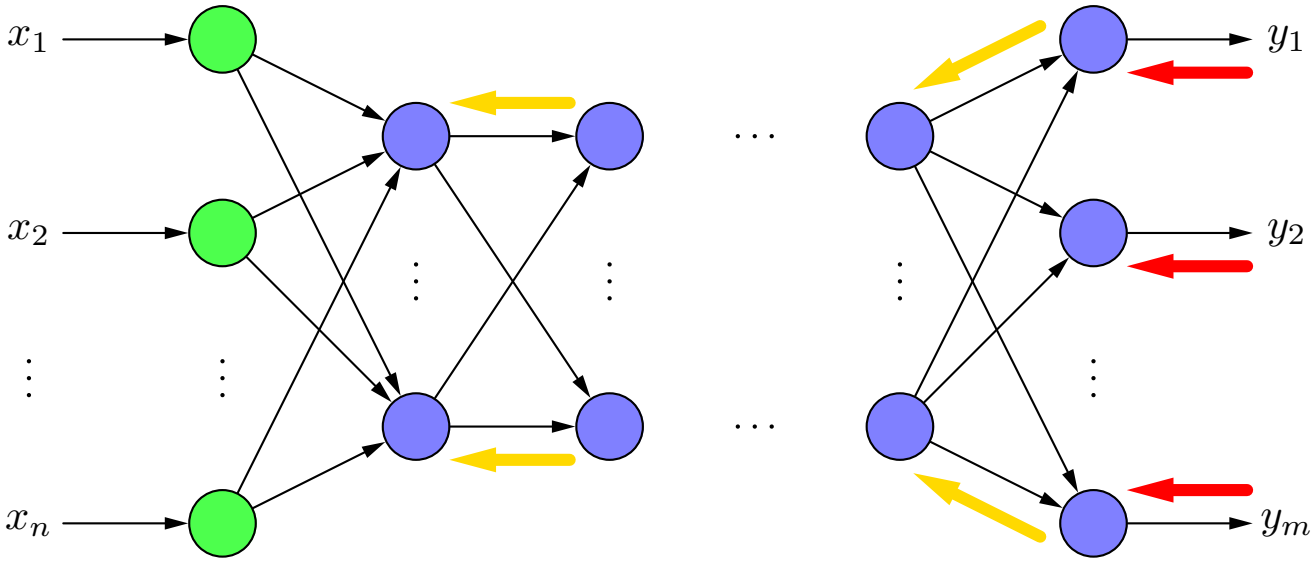
$$\Delta\vec{w}_u^{(l)} = \eta \left(\sum_{s \in \text{succ}(u)} \delta_s^{(l)} w_{su} \right) \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right) \vec{\text{in}}_u^{(l)}.$$

Error Backpropagation: Cookbook Recipe

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

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

backward propagation:

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

error factor:

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

activation derivative:

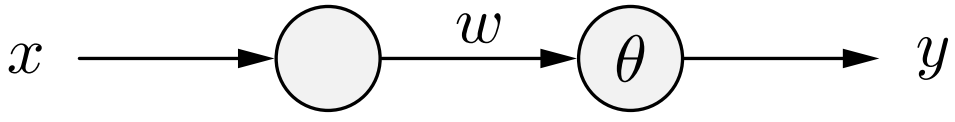
$\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right)$

weight change:

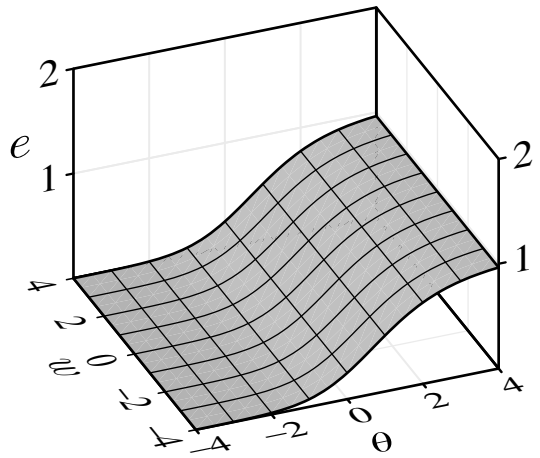
$\Delta w_{up}^{(l)} = \eta\, \delta_u^{(l)}\, \text{out}_p^{(l)}$

Gradient Descent: Examples

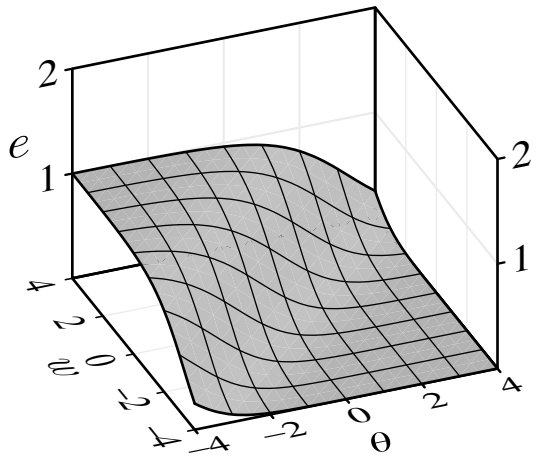
Gradient descent training for the negation $\neg x$



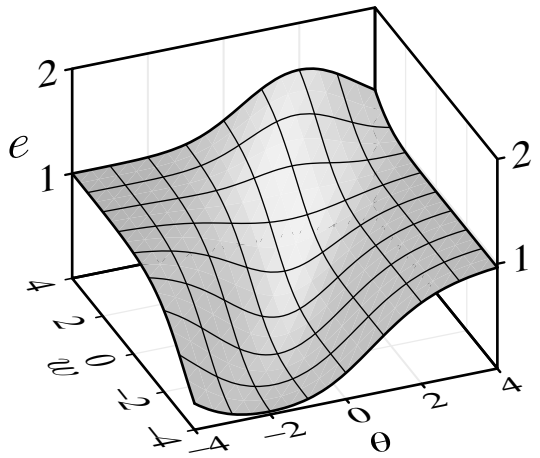
x	y
0	1
1	0



error for $x = 0$



error for $x = 1$



sum of errors

Note: error for $x = 0$ and $x = 1$ is effectively the squared logistic activation function!

Gradient Descent: Examples

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

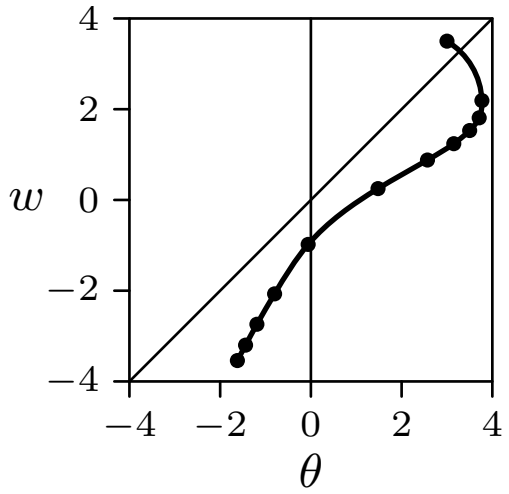
Online Training

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

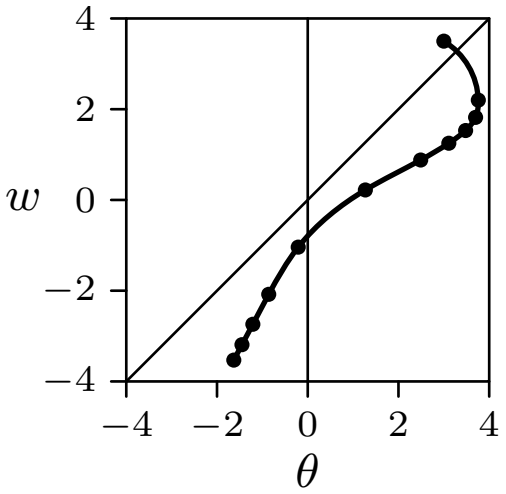
Batch Training

Gradient Descent: Examples

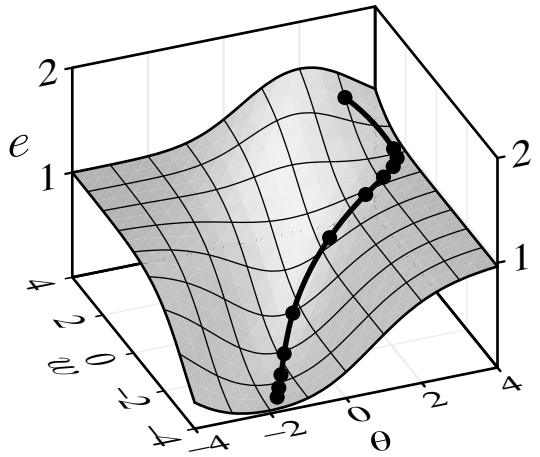
Visualization of gradient descent for the negation $\neg x$



Online Training



Batch Training



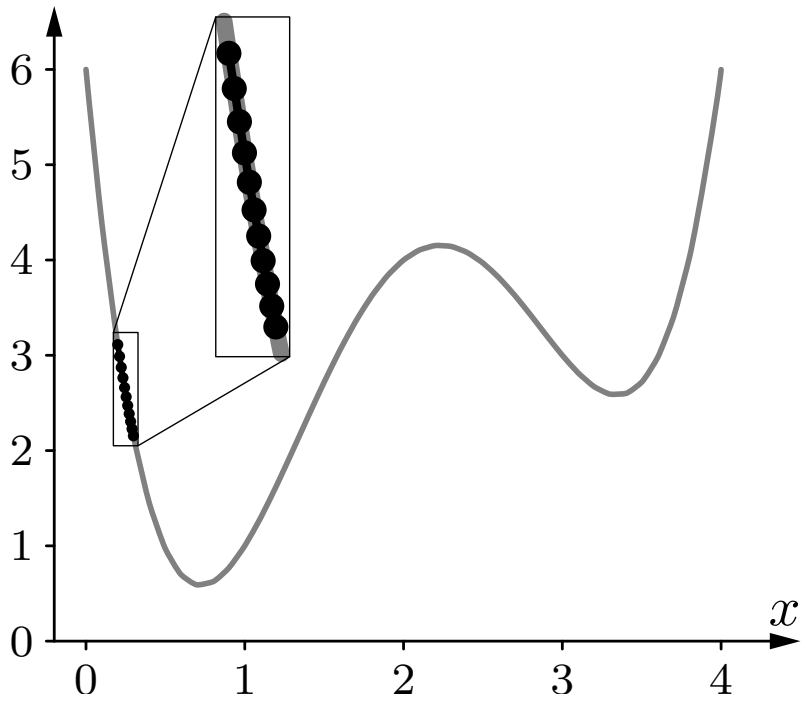
Batch Training

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

Gradient Descent: Examples

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

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

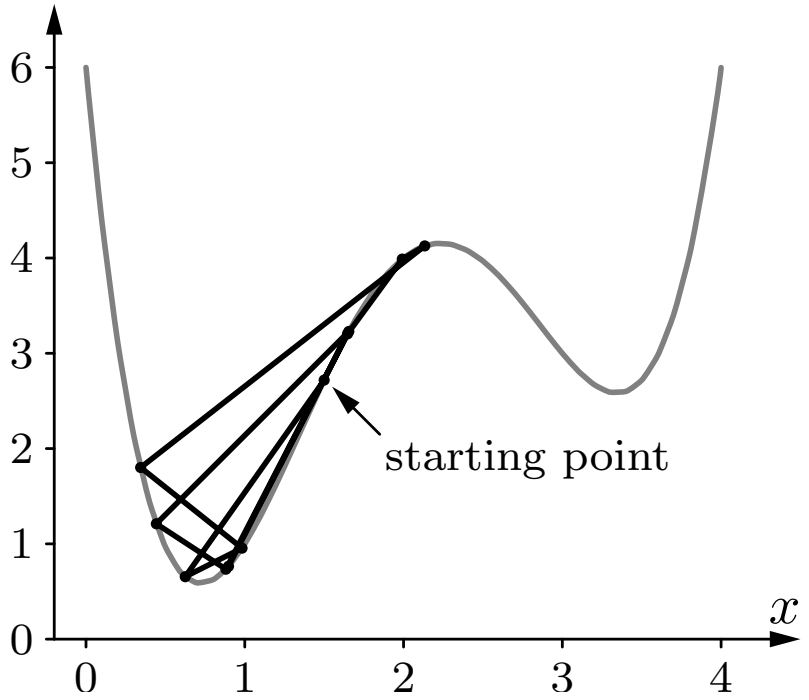


Gradient descent with initial value 0.2 and learning rate 0.001.

Gradient Descent: Examples

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

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

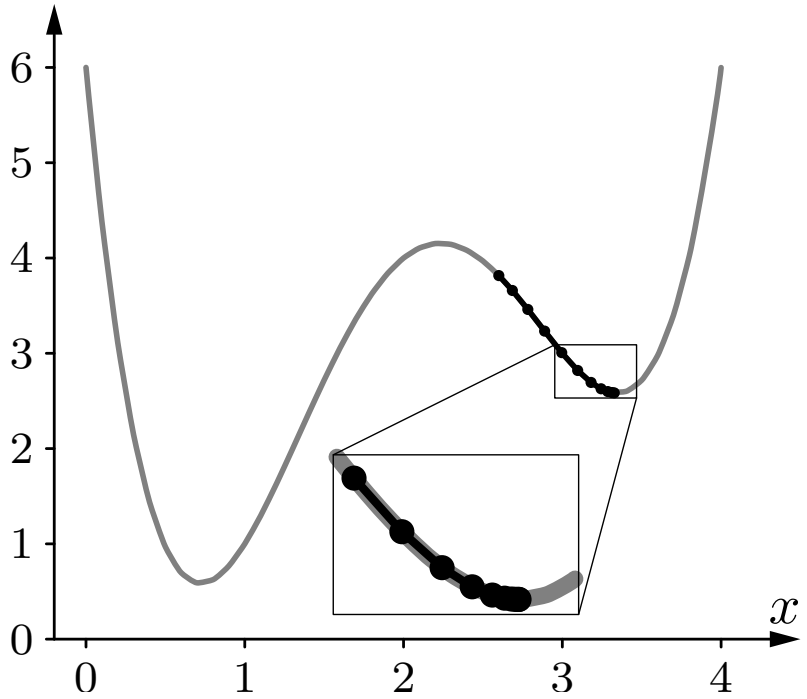


Gradient descent with initial value 1.5 and learning rate 0.25.

Gradient Descent: Examples

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

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



Gradient descent with initial value 2.6 and learning rate 0.05.

Gradient Descent: Variants

Weight update rule:

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

Standard backpropagation:

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

Manhattan training:

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

Fixed step width (grid), only sign of gradient (direction) is evaluated.

Momentum term:

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

Part of previous change is added, may lead to accelerated training ($\beta \in [0.5, 0.95]$).

Gradient Descent: Variants

Self-adaptive error backpropagation:

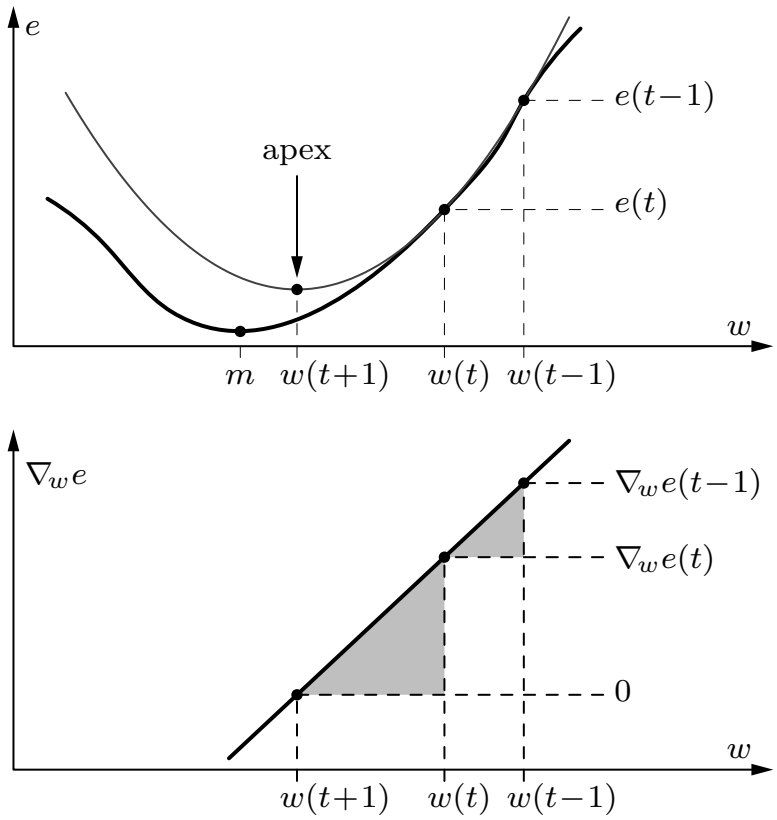
$$\eta_w(t) = \begin{cases} c^- \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) < 0, \\ c^+ \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \cdot \nabla_w e(t-1) > 0 \\ & \wedge \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \eta_w(t-1), & \text{otherwise.} \end{cases}$$

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 $c^+ \in [1.05, 1.2]$.

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

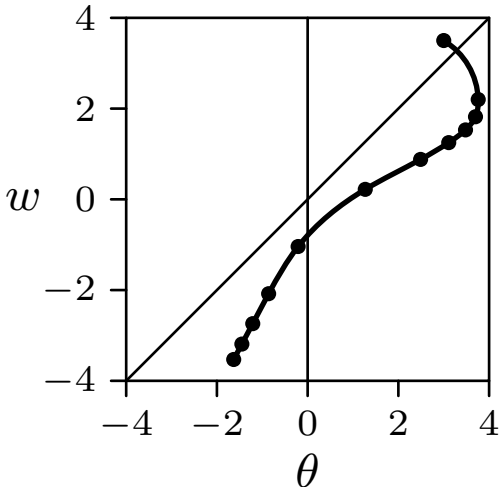
epoch	θ	w	error
0	3.00	3.50	1.295
20	3.76	2.20	0.985
40	3.70	1.82	0.970
60	3.48	1.53	0.957
80	3.11	1.25	0.934
100	2.49	0.88	0.880
120	1.27	0.22	0.676
140	−0.21	−1.04	0.292
160	−0.86	−2.08	0.140
180	−1.21	−2.74	0.084
200	−1.45	−3.19	0.058
220	−1.63	−3.53	0.044

without momentum term

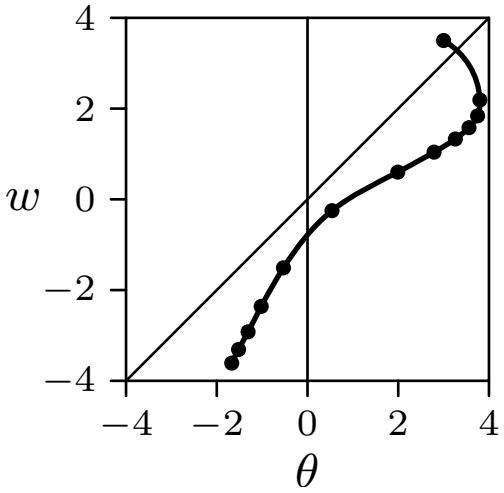
epoch	θ	w	error
0	3.00	3.50	1.295
10	3.80	2.19	0.984
20	3.75	1.84	0.971
30	3.56	1.58	0.960
40	3.26	1.33	0.943
50	2.79	1.04	0.910
60	1.99	0.60	0.814
70	0.54	−0.25	0.497
80	−0.53	−1.51	0.211
90	−1.02	−2.36	0.113
100	−1.31	−2.92	0.073
110	−1.52	−3.31	0.053
120	−1.67	−3.61	0.041

with momentum term ($\beta = 0.9$)

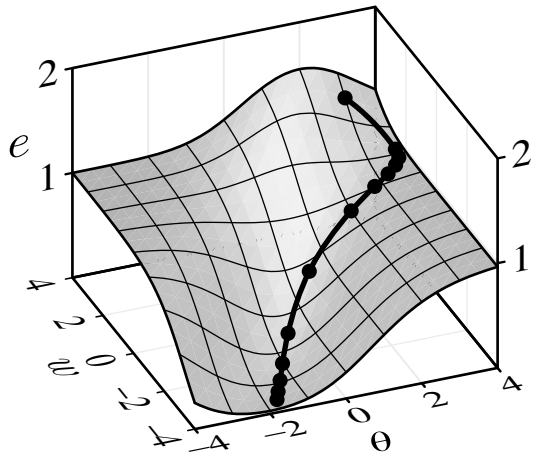
Gradient Descent: Examples



without momentum term



with momentum term



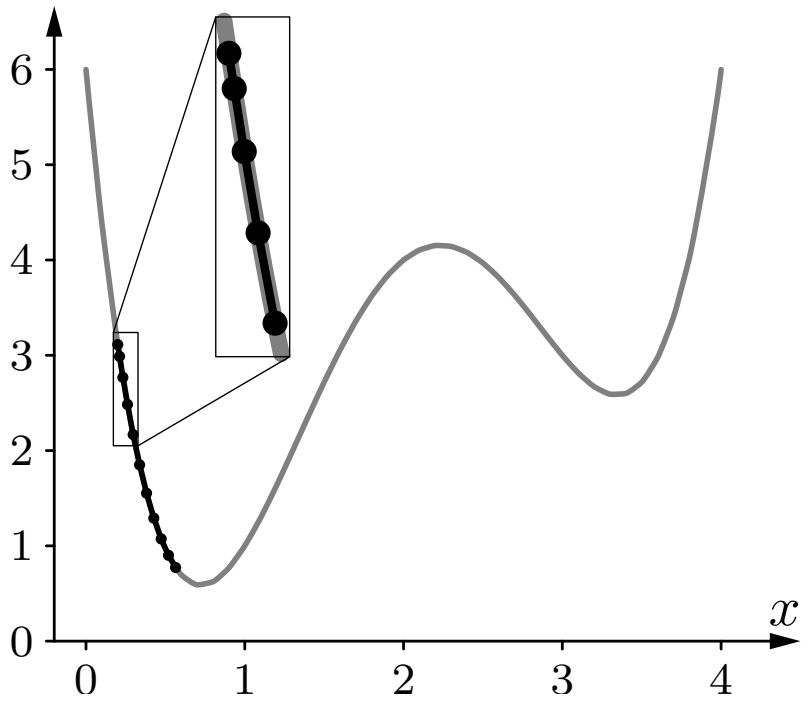
with momentum term

- Dots show position every 20 (without momentum term) or every 10 epochs (with momentum term).
- Learning with a momentum term ($\beta = 0.9$) 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		

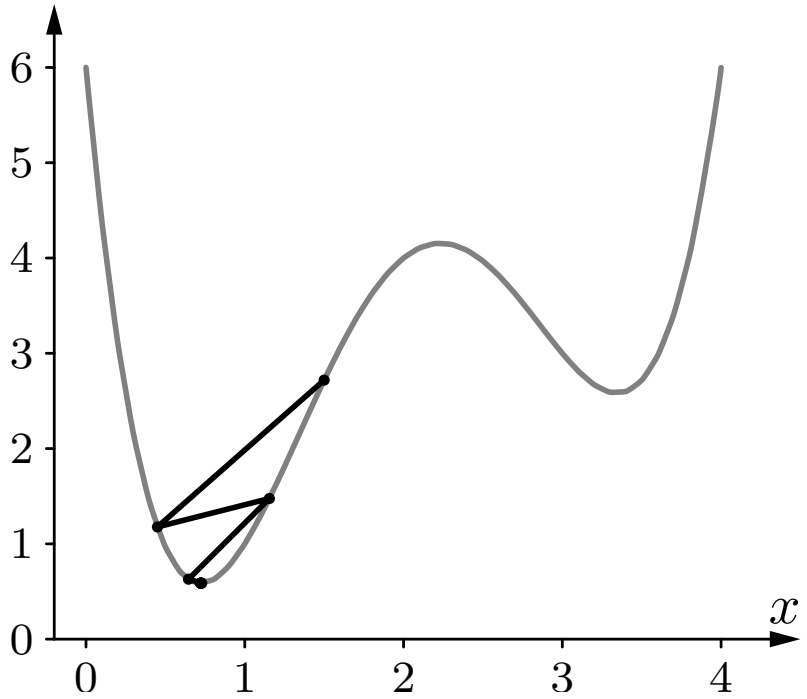


Gradient descent with initial value 0.2, learning rate 0.001 and momentum term $\beta = 0.9$.

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 initial value 1.5, inital learning rate 0.25,
and 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 ($\zeta \approx 0.1$).
- 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 ($\xi \leq 10^{-3}$).
- Can be derived from an extended error function penalizing large weights:

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

Number of Hidden Neurons

- Note that the approximation theorem only states that there *exists* a number of hidden neurons and weight vectors $\vec{v}$ and $\vec{w}_i$ and thresholds θ_i, but not how they are to be chosen for a given ε of approximation accuracy.
- For a single hidden layer the following **rule of thumb** is popular:

  number of hidden neurons = (number of inputs + number of outputs) / 2
- Better, though computationally expensive approach:
  - Randomly split the given data into two subsets of (about) equal size, the **training data** and the **validation data**.
  - Train multi-layer perceptrons with different numbers of hidden neurons on the training data and evaluate them on the validation data.
  - Repeat the random split of the data and training/evaluation many times and average the results over the same number of hidden neurons. Choose the number of hidden neurons with the best average error.
  - Train a final multi-layer perceptron on the whole data set.

Number of Hidden Neurons

Principle of training data/validation data approach:

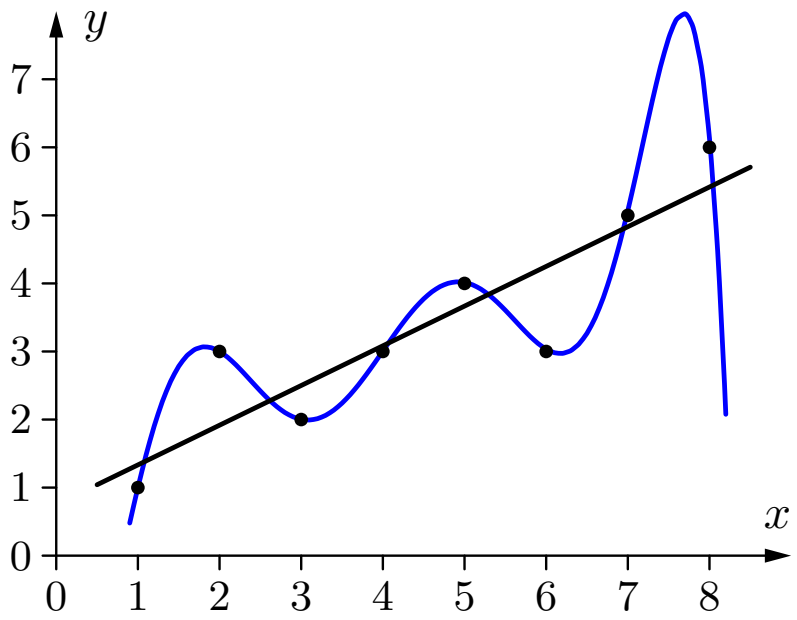
- **Underfitting:** If the number of neurons in the hidden layer is too small, the multi-layer perceptron may not be able to capture the structure of the relationship between inputs and outputs precisely enough due to a lack of parameters.
- **Overfitting:** With a larger number of hidden neurons a multi-layer perceptron may adapt not only to the regular dependence between inputs and outputs, but also to the accidental specifics (errors and deviations) of the training data set.
- Overfitting will usually lead to the effect that the error a multi-layer perceptron yields on the validation data will be (possibly considerably) greater than the error it yields on the training data.

  The reason is that the validation data set is likely distorted in a different fashion than the training data, since the errors and deviations are random.
- Minimizing the error on the validation data by properly choosing the number of hidden neurons prevents both under- and overfitting.

Number of Hidden Neurons: Avoid Overfitting

- Objective: select the model that best fits the data,
  taking the model complexity into account.

  The more complex the model, the better it usually fits the data.



black line:
regression line
(2 free parameters)

blue curve:
7th order regression polynomial
(8 free parameters)

- The blue curve fits the data points perfectly, **but it is not a good model**.

Number of Hidden Neurons: Cross Validation

- The described method of iteratively splitting the data into training and validation data may be referred to as **cross validation**.
- However, this term is more often used for the following specific procedure:
  - The given data set is split into n parts or subsets (also called folds) of about equal size (so-called **n-fold cross validation**).
  - If the output is nominal (also sometimes called symbolic or categorical), this split is done in such a way that the relative frequency of the output values in the subsets/folds represent as well as possible the relative frequencies of these values in the data set as a whole. This is also called **stratification** (derived from the Latin *stratum*: layer, level, tier).
  - Out of these n data subsets (or folds) n pairs of training and validation data set are formed by using one fold as a validation data set while the remaining $n - 1$ folds are combined into a training data set.

Number of Hidden Neurons: Cross Validation

- The advantage of the cross validation method is that one random split of the data yields n different pairs of training and validation data set.
- An obvious disadvantage is that (except for $n = 2$) the size of the training and the test data set are considerably different, which makes the results on the validation data statistically less reliable.
- It is therefore only recommended for sufficiently large data sets or sufficiently small n, so that the validation data sets are of sufficient size.
- Repeating the split (either with $n = 2$ or greater n) has the advantage that one obtains many more training and validation data sets, leading to more reliable statistics (here: for the number of hidden neurons).
- The described approaches fall into the category of **resampling methods**.
- Other well-known statistical resampling methods are **bootstrap**, **jackknife**, **subsampling** and **permutation test**.

Avoiding Overfitting: Alternatives

- An alternative way to prevent overfitting is the following approach:
  - During training the performance of the multi-layer perceptron is evaluated after each epoch (or every few epochs) on a **validation data set**.
  - While the error on the training data set should always decrease with each epoch, the error on the validation data set should, after decreasing initially as well, increase again as soon as overfitting sets in.
  - At this moment training is terminated and either the current state or (if available) the state of the multi-layer perceptron, for which the error on the validation data reached a minimum, is reported as the training result.
- Furthermore a stopping criterion may be derived from the shape of the error curve on the training data over the training epochs, or the network is trained only for a fixed, relatively small number of epochs (also known as **early stopping**).
- Disadvantage: these methods stop the training of a complex network early enough, rather than adjust the complexity of the network to the "correct" level.

Sensitivity Analysis

Sensitivity Analysis

Problem of Multi-Layer Perceptrons:

- The knowledge that is learned by a neural network
  is encoded in matrices/vectors of real-valued numbers and
  is therefore often difficult to understand or to extract.
- Geometric interpretation or other forms of intuitive understanding are possible
  only for very simple networks, but fail for complex practical problems.
- Thus a neural network is often effectively a *black box*,
  that computes its output, though in a mathematically precise way,
  in a way that is very difficult to understand and to interpret.

Idea of Sensitivity Analysis:

- Try to find out to which inputs the output(s) react(s) most sensitively.
- This may also give hints which inputs are not needed and may be discarded.

Sensitivity Analysis

Question: How important are different inputs to the network?

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

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

Formal derivation: Apply chain rule.

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{ext}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} \frac{\partial \operatorname{out}_u}{\partial \operatorname{ext}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} \frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} \frac{\partial \operatorname{out}_u}{\partial \operatorname{ext}_u}.$$

Simplification: Assume that the output function is the identity.

$$\frac{\partial \operatorname{out}_u}{\partial \operatorname{ext}_u} = 1.$$

Sensitivity Analysis

For the second factor we get the general result:

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

This leads to the recursion formula

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

However, for the first hidden layer we get

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

This formula marks the start of the recursion.

Sensitivity Analysis

Consider as usual the special case with

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

The recursion formula is in this case

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

and the anchor of the recursion is

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

Sensitivity Analysis

Attention: Use weight decay to stabilize the training results!

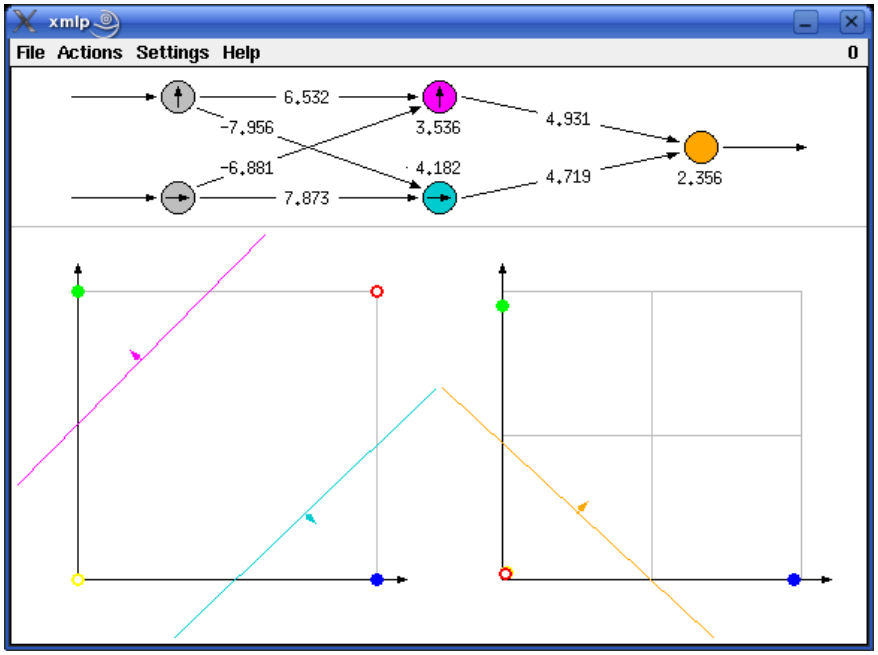
$$\Delta w(t) = -\frac{\eta}{2}\nabla_w e(t) - \xi\, w(t),$$

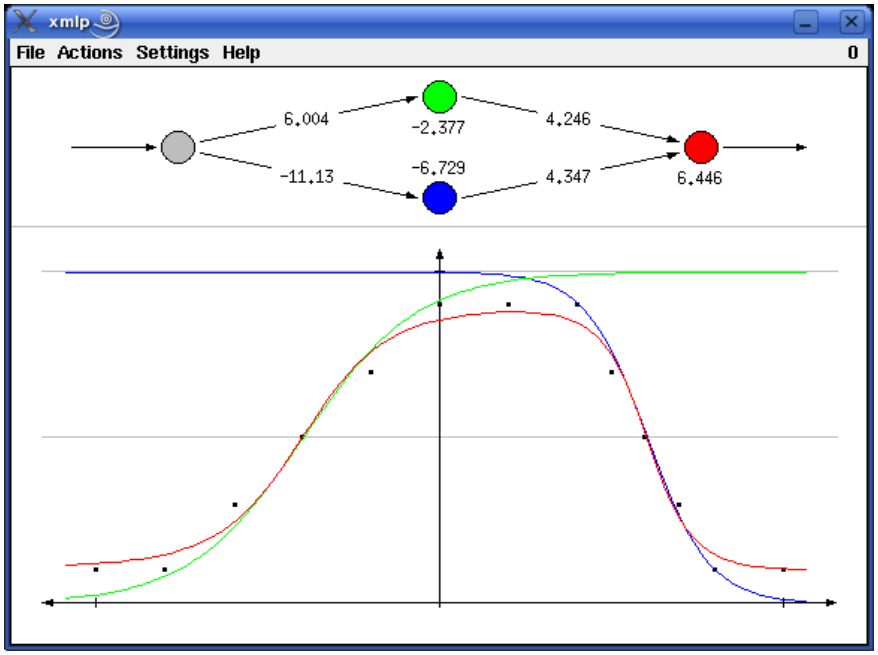
- Without weight decay the results of a sensitivity analysis can depend (strongly) on the (random) initialization of the network.

Example: Iris data, 1 hidden layer, 3 hidden neurons, 1000 epochs

attribute	$\xi = 0$				$\xi = 0.0001$			
sepal length	0.0216	0.0399	0.0232	0.0515	0.0367	0.0325	0.0351	0.0395
sepal width	0.0423	0.0341	0.0460	0.0447	0.0385	0.0376	0.0421	0.0425
petal length	0.1789	0.2569	0.1974	0.2805	0.2048	0.1928	0.1838	0.1861
petal width	0.2017	0.1356	0.2198	0.1325	0.2020	0.1962	0.1750	0.1743

Demonstration Software: xmlp/wmlp

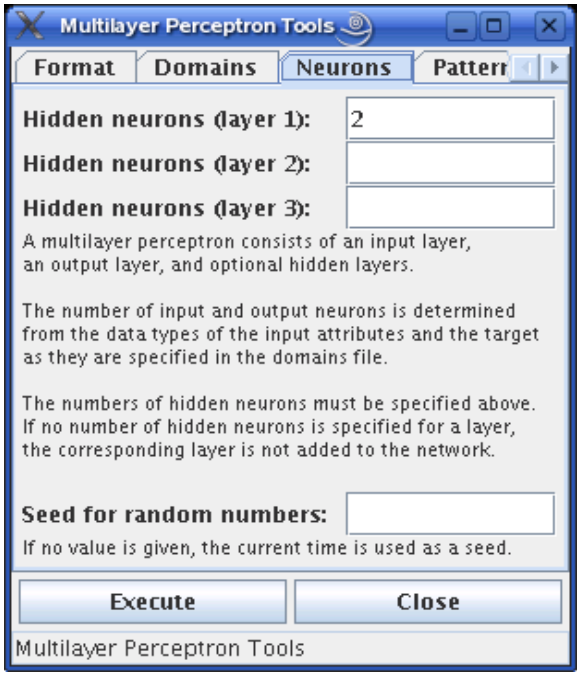


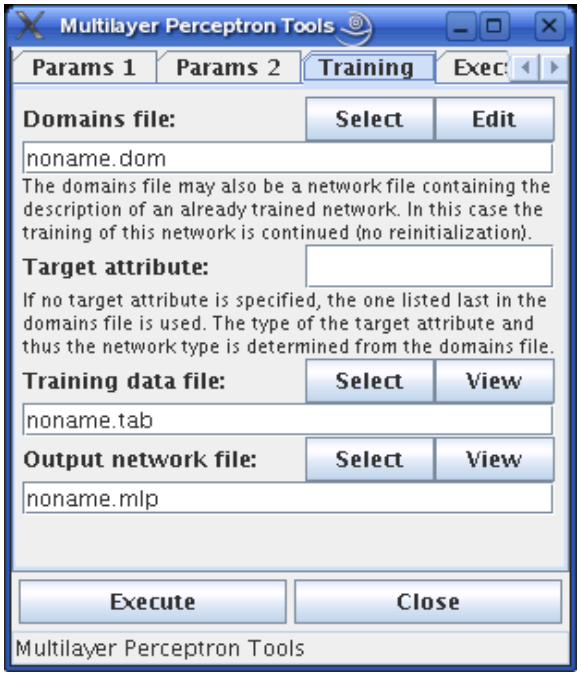


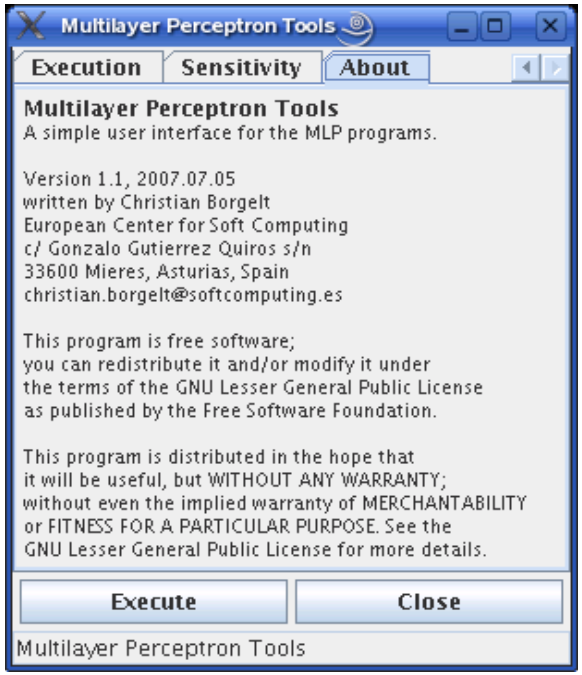
Demonstration of multi-layer perceptron training:

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

Multi-Layer Perceptron Software: mlp/mlpgui







Software for training general multi-layer 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`

Deep Learning

(Multi-Layer Perceptrons with Many Layers)

Deep Learning: Motivation

- Reminder: **Universal Approximation Theorem**
  Any continous function on an arbitrary compact subspace of $\mathbb{R}^n$
  can be approximated arbitrarily well with a three-layer perceptron.
- This theorem is often cited as (allegedly!) meaning that
  - we can confine ourselves to multi-layer perceptrons with only one hidden layer,
  - there is no real need to look at multi-layer perceptrons with more hidden layers.
- **However:** The theorem says nothing about the number of hidden neurons
  that may be needed to achieve a desired approximation accuracy.
- Depending on the function to approximate,
  a very large number of neurons may be necessary.
- Allowing for more hidden layers may enable us
  to achieve the same approximation quality
  with a significantly lower number of neurons.

Deep Learning: Motivation

- Very simple and commonly used example is the **n-bit parity function**:
  Output is 1 if an even number of inputs is 1; output is 0 otherwise.
- Can easily be represented by a multi-layer perceptron with only one hidden layer.
  (See reminder of TLU algorithm on next slide; adapt to MLPs.)
- However, the solution has $\mathbf{2^{n-1}}$ **hidden neurons**:
  disjunctive normal form is a disjunction of 2^{n-1} conjunctions,
  which represent the 2^{n-1} input combinations with an even number of set bits.
- Number of hidden neurons **grows exponentially** with the number of inputs.
- However, if more hidden layers are admissible, **linear growth** is possible:
  - Start with a *biimplication* of two inputs.
  - Continue with a chain of *exclusive or*s, each of which adds another input.
  - Such a network needs $n + 3(n-1) = 4n - 3$ neurons in total
    (n input neurons, $\mathbf{3(n-1)-1}$ **hidden neurons**, 1 output neuron)

Reminder: Representing Arbitrary Boolean Functions

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

(i) Represent the given function $f(x_1, \ldots, x_n)$ in disjunctive normal form. That is, determine where all K_j are conjunctions of n literals, that is, $K_j = l_{j1} \wedge \ldots \wedge l_{jn}$ with $l_{ji} = x_i$ (positive literal) or $l_{ji} = \neg x_i$ (negative literal).

(ii) Create a neuron for each conjunction K_j of the disjunctive normal form (having n inputs — one input for each variable), where

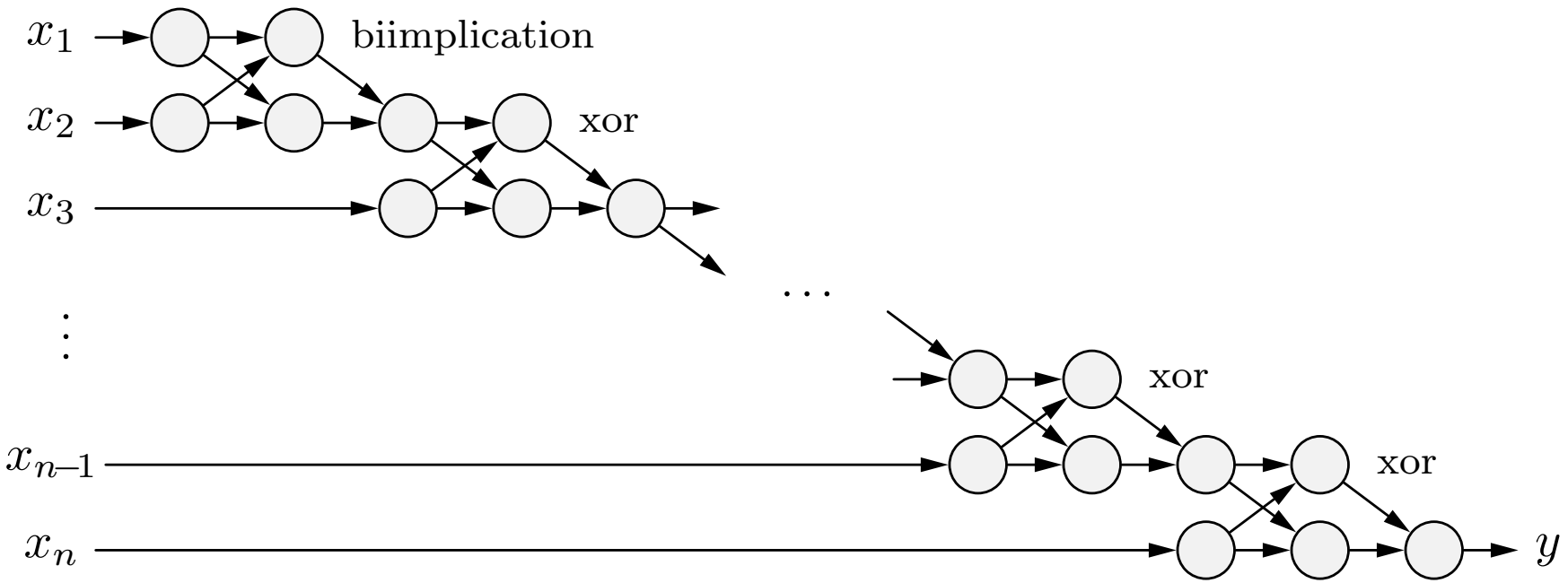
$$w_{ji} = \begin{cases} 2, & \text{if } l_{ji} = x_i, \\ -2, & \text{if } l_{ji} = \neg x_i, \end{cases} \qquad \text{and} \qquad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{ji}.$$

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

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

Remark: weights are set to ± 2 instead of ± 1 in order to ensure integer thresholds.

Deep Learning: n-bit Parity Function



- Implementation of the n-bit parity function with a chain of one *biimplication* and $n-2$ *exclusive or* sub-networks.
- Note that the structure is not strictly layered, but could be completed.
  If this is done, the number of neurons increases to $n(n+1)-1$.

Deep Learning: Motivation

- Similar to the situation w.r.t. linearly separable functions:
  - Only few n-ary Boolean functions are linearly separable (see next slide).
  - Only few n-ary Boolean functions need few hidden neurons in a single layer.
- An n-ary Boolean function has k_0 input combinations with an output of 0 and k_1 input combinations with an output of 1 (clearly $k_0 + k_1 = 2^n$).
- We need at most $2^{\min\{k_0,k_1\}}$ hidden neurons if we choose disjunctive normal form for $k_1 \leq k_0$ and conjunctive normal form for $k_1 > k_0$.
- As there are $\binom{2^n}{k_0}$ possible functions with k_0 input combinations mapped to 0 and k_1 mapped to 1, many functions require a substantial number of hidden neurons.
- Although the number of neurons may be reduced with minimization methods like the Quine–McCluskey algorithm [Quine 1952, 1955; McCluskey 1956], the fundamental problem remains.

Reminder: Limitations of Threshold Logic Units

Total number and number of linearly separable Boolean functions
(On-Line Encyclopedia of Integer Sequences, `oeis.org`, A001146 and A000609):

inputs	Boolean functions	linearly separable functions
1	4	4
2	16	14
3	256	104
4	65,536	1,882
5	4,294,967,296	94,572
6	18,446,744,073,709,551,616	15,028,134
n	$2^{(2^n)}$	no general formula known

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

Deep Learning: Motivation

- In practice the problem is mitigated considerably by the simple fact that **training data sets are necessarily limited in size**.
- Complete training data for an n-ary Boolean function has 2^n training examples. Data sets for practical problems usually contain much fewer sample cases.

  This leads to many input configurations for which no desired output is given; freedom to assign outputs to them allows for a simpler representation.
- Nevertheless, using more than one hidden layer promises in many cases to reduce the number of needed neurons.
- This is the focus of the area of **deep learning**.
  (**Depth** means the length of the longest path in the network graph.)
- For multilayer perceptrons (longest path: number of hidden layers plus one), deep learning starts with more than one hidden layer.
- For 10 or more hidden layers, one sometimes speaks of **very deep learning**.

Deep Learning: Main Problems

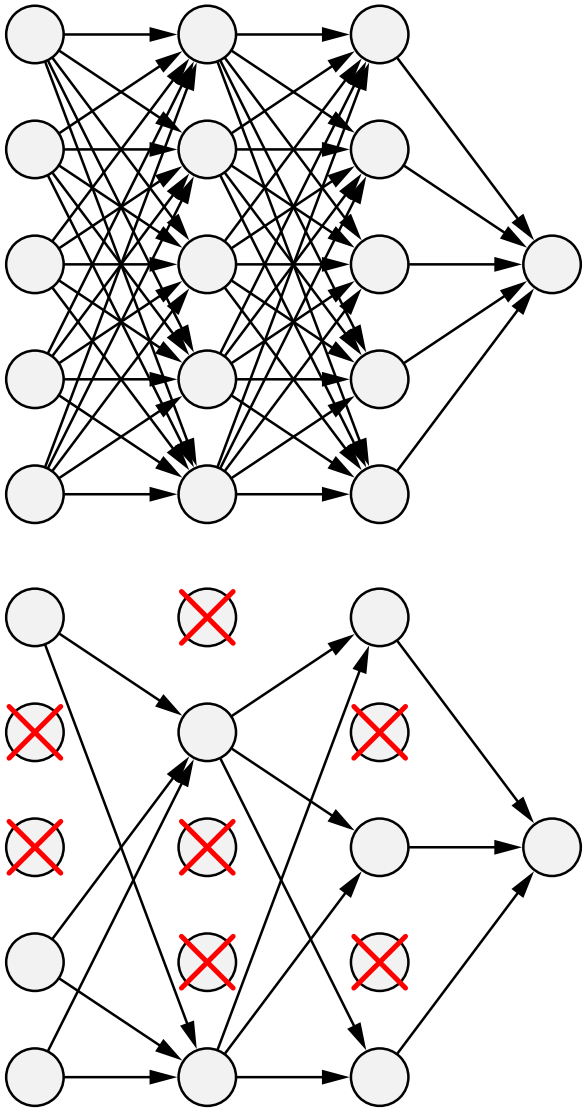
- Deep learning multilayer perceptrons suffer from two main problems:

  overfitting and **vanishing gradient**.

- Overfitting results mainly from the increased number of adaptable parameters.
- **Weight decay** prevents large weights and thus an overly precise adaptation.
- **Sparsity constraints** help to avoid overfitting:
  - there is only a restricted number of neurons in the hidden layers or
  - only few of the neurons in the hidden layers should be active (on average).

    May be achieved by adding a regularization term to the error function (compares the observed number of active neurons with desired number and pushes adaptations into a direction that tries to match these numbers).
- Furthermore, a training method called **dropout training** may be applied: some units are randomly omitted from the input/hidden layers during training.

Deep Learning: Dropout



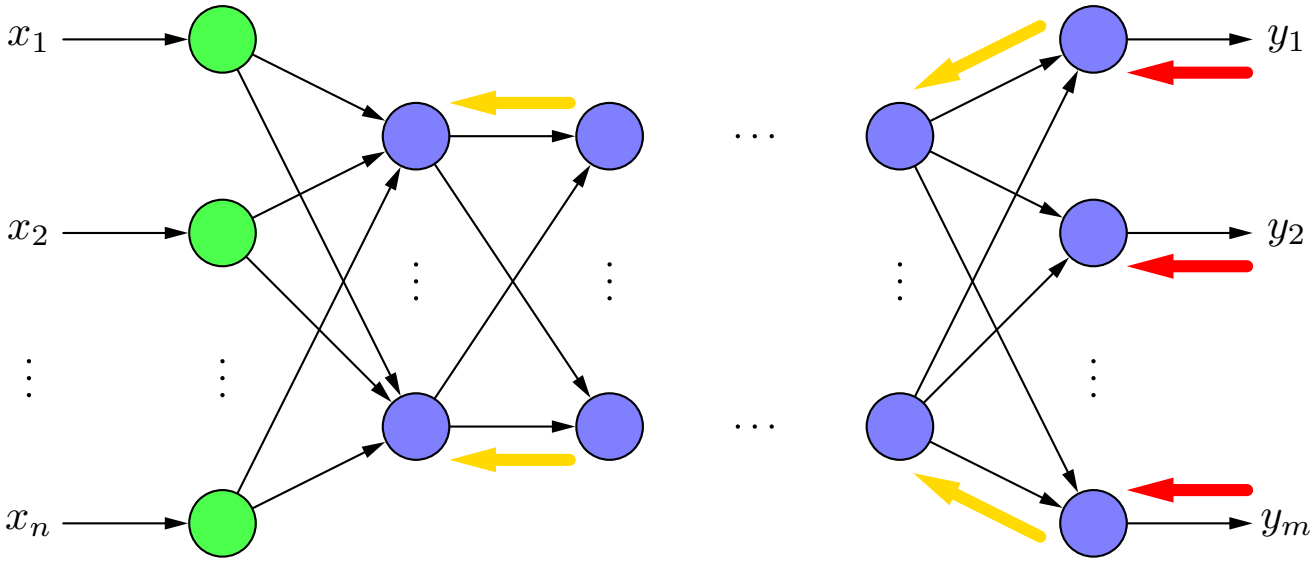
- Desired characteristic:
  Robustness against neuron failures.
- Approach during training:
  - Use only $p\%$ of the neurons
    (e.g. $p = 50$: half of the neurons).
  - Choose dropout neurons randomly.
- Approach during execution:
  - Use all of the neurons.
  - Multiply all weights by $p\%$.
- Result of this approach:
  - More robust representation.
  - Better generalization.

Reminder: Cookbook Recipe for Error Backpropagation

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

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

backward propagation:

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

error factor:

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

activation derivative:

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

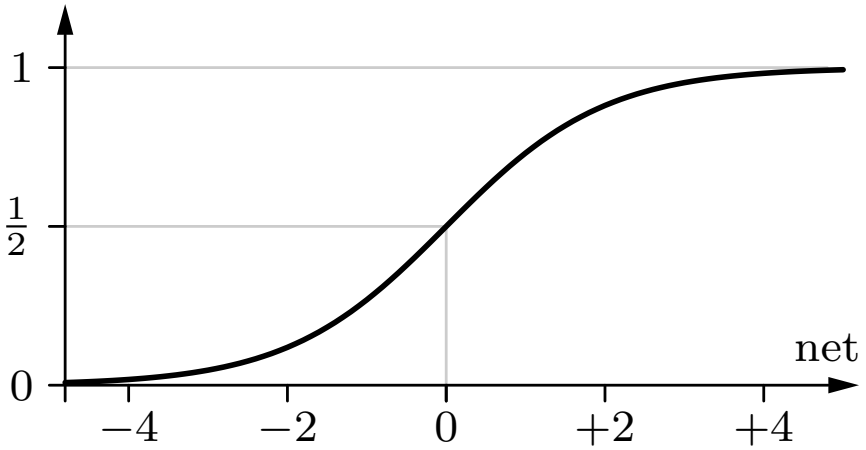
weight change:

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

Deep Learning: Vanishing Gradient

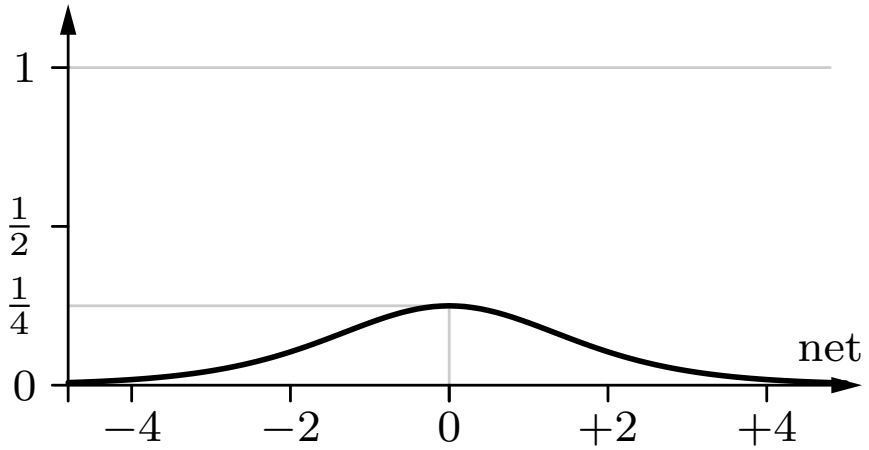
logistic activation function:

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



derivative of logistic function:

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



- If a logistic activation function is used (shown on left), the weight changes are proportional to $\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right)$ (shown on right; see recipe).
- This factor is also propagated back, but cannot be larger than $\frac{1}{4}$ (see right).

$\Rightarrow$ The gradient tends to vanish if many layers are backpropagated through. Learning in the early hidden layers can become very slow [Hochreiter 1991].

Deep Learning: Vanishing Gradient

- In principle, a small gradient may be counteracted by a large weight.

$$\delta_u^{(l)} = \Big(\sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} w_{su} \Big) \lambda_u^{(l)}.$$

- However, usually a large weight, since it also enters the activation function, drives the activation function to its **saturation regions**.

- Thus, (the absolute value of) the derivative factor is usually the smaller, the larger (the absolute value of) the weights.

- Furthermore, the connection weights are commonly initialized to a random value in the range from -1 to $+1$.

  $\Rightarrow$ **Initial training steps are particularly affected:**
  both the gradient as well as the weights provide a factor less than 1.

- Theoretically, there can be exceptions to this description.
  However, they are rare in practice and one usually observes a vanishing gradient.

Deep Learning: Vanishing Gradient

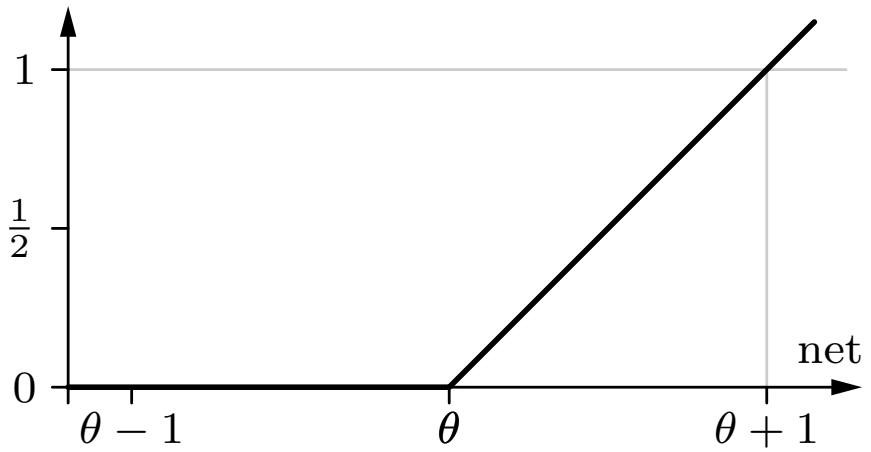
Alternative way to understand the vanishing gradient effect:

- The logistic activation function is a **contracting function**:
  for any two arguments x and y, $x \neq y$, we have $|f_{\text{act}}(x) - f_{\text{act}}(y)| < |x - y|$.
  (Obvious from the fact that its derivative is always < 1; actually $\leq \frac{1}{4}$.)
- If several logistic functions are chained, these contractions combine
  and yield an even stronger contraction of the input range.
- As a consequence, a rather large change of the input values
  will produce only a rather small change in the output values,
  and the more so, the more logistic functions are chained together.
- Therefore the function that maps the inputs of a multilayer perceptron to its outputs usually becomes the flatter the more layers the multilayer perceptron has.
- Consequently the gradient in the first hidden layer
  (were the inputs are processed) becomes the smaller.

Deep Learning: Different Activation Functions

rectified maximum/ramp function:

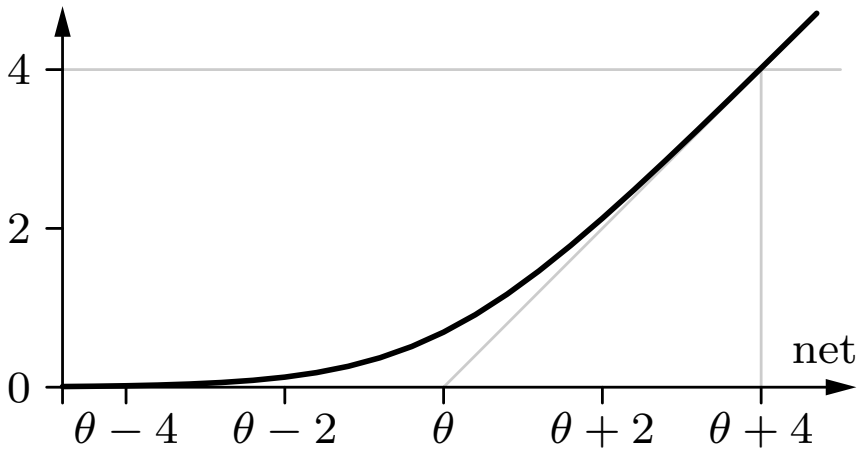
$f_{\text{act}}(\text{net}, \theta) = \max\{0, \text{net} - \theta\}$



softplus function:

Note the scale!

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



- The vanishing gradient problem may be battled with other activation functions.
- These activation functions yield so-called **rectified linear units (ReLUs)**.
- **Rectified maximum/ramp function**

  Advantages: simple computation, simple derivative, zeros simplify learning
  Disadvantages: no learning $\leq \theta$, rather inelegant, not continously differentiable

Deep Learning: AlphaGo

- **AlphaGo** is a computer program
  developed by Alphabet Inc.'s Google
  DeepMind to play the board game Go.
- It uses a combination of machine learning and tree search techniques,
  combined with extensive training, both from human and computer play.
- AlphaGo uses Monte Carlo tree search,
  guided by a "value network" and a "policy network",
  both of which are implemented using **deep neural networks**.
- A limited amount of game-specific feature detection is applied to the input
  before it is sent to the neural networks.
- The neural networks were bootstrapped from human gameplay experience.
  Later AlphaGo was set up to play against other instances of itself,
  using reinforcement learning to improve its play.

source: Wikipedia

Deep Learning: AlphaGo

Match against **Fan Hui**
(Elo 3016 on 01-01-2016,
#512 world ranking list),
best European player
at the time of the match

AlphaGo wins 5 : 0

Match against **Lee Sedol**
(Elo 3546 on 01-01-2016,
#1 world ranking list 2007–2010,
#3 at the time of the match)

AlphaGo wins 4 : 1

`www.goratings.org`

AlphaGo	threads	CPUs	GPUs	Elo
Async.	1	48	8	2203
Async.	2	48	8	2393
Async.	4	48	8	2564
Async.	8	48	8	2665
Async.	16	48	8	2778
Async.	32	48	8	2867
Async.	40	48	1	2181
Async.	40	48	2	2738
Async.	40	48	4	2850
Async.	40	48	8	2890
Distrib.	12	428	64	2937
Distrib.	24	764	112	3079
Distrib.	40	1202	176	3140
Distrib.	64	1920	280	3168

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

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

(ii) $C = (U_{\text{in}} \times U_{\text{hidden}}) \cup C', \quad C' \subseteq (U_{\text{hidden}} \times U_{\text{out}})$

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

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

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

$$\begin{aligned}
&(i) && d(\vec{x}, \vec{y}) = 0 \quad \Leftrightarrow \quad \vec{x} = \vec{y}, && \\
&(ii) && d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x}) && \text{(symmetry)}, \\
&(iii) && d(\vec{x}, \vec{z}) \leq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z}) && \text{(triangle inequality)}.
\end{aligned}$$

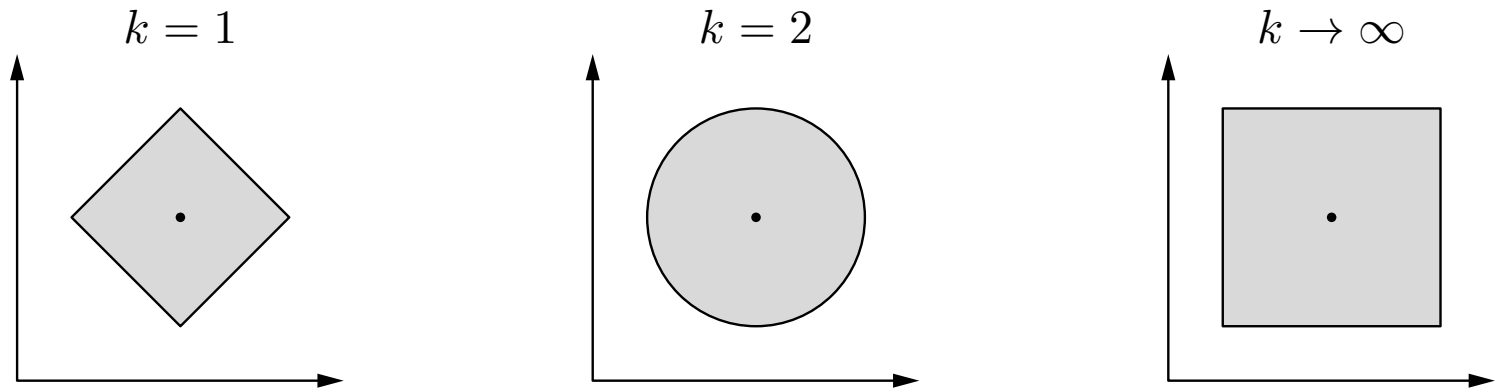
Distance Functions

Illustration of distance functions: Minkowski Family

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

Well-known special cases from this family are:

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



Radial Basis Function Networks

The network input function of the output neurons is the weighted sum of their inputs:

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

The activation function of each hidden neuron is a so-called **radial function**, that is, a monotonically decreasing function

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

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

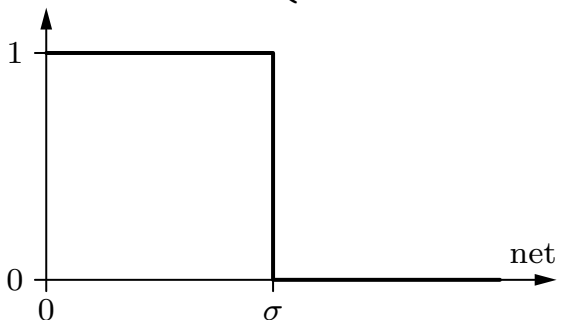
$$f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \text{net}_u - \theta_u.$$

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

Radial Activation Functions

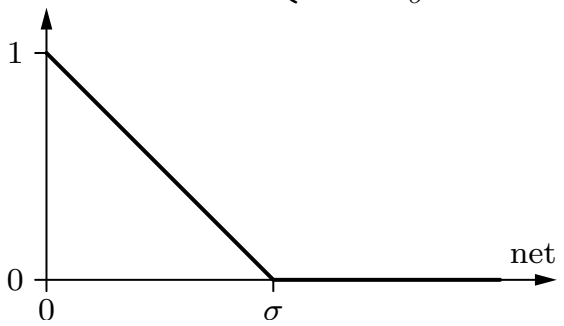
rectangle function:

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



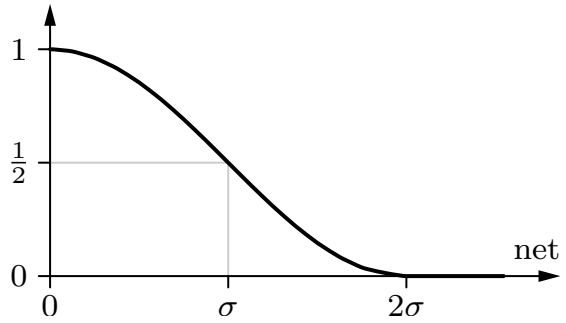
triangle function:

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



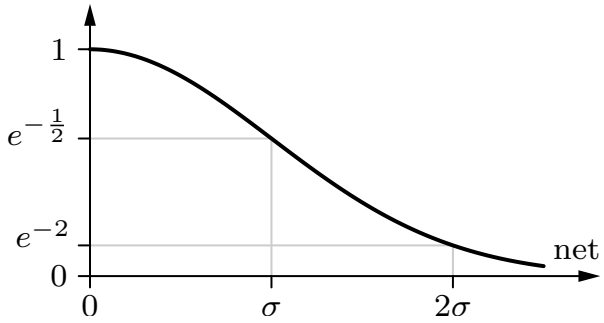
cosine until zero:

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



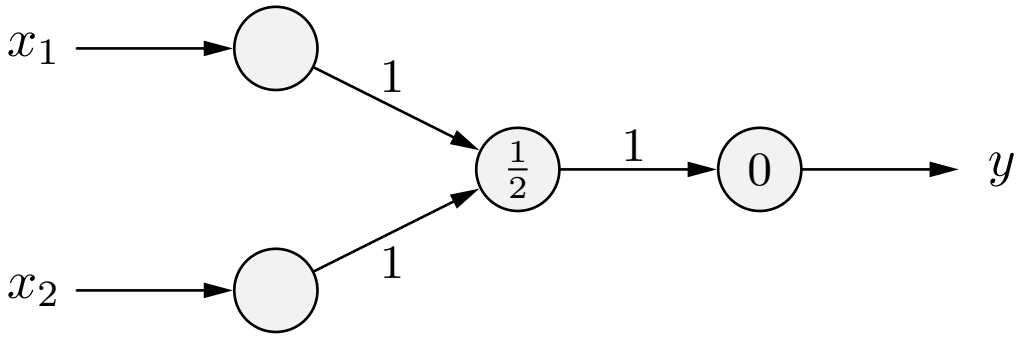
Gaussian function:

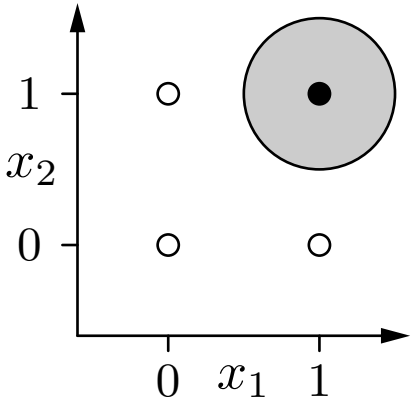
$$f_{\text{act}}(\text{net}, \sigma) = e^{-\frac{\text{net}^2}{2\sigma^2}}$$

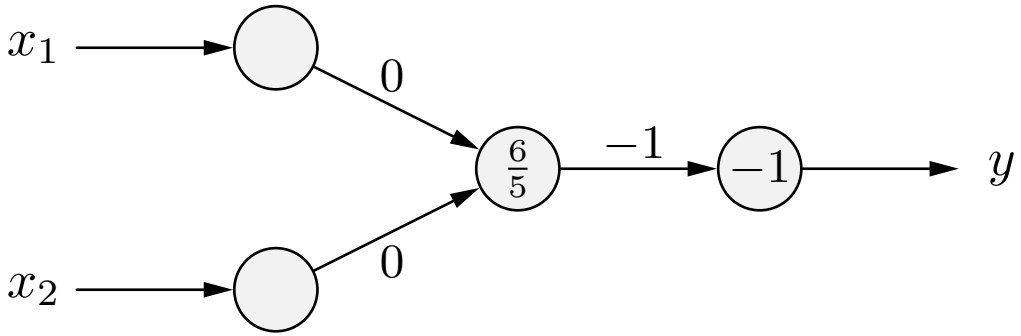


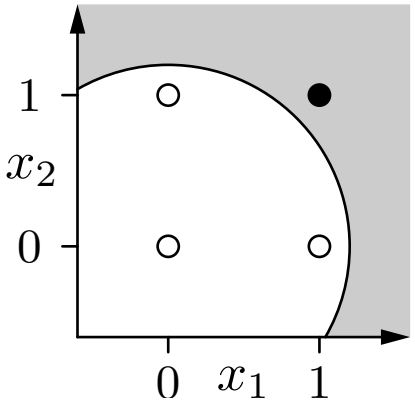
Radial Basis Function Networks: Examples

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







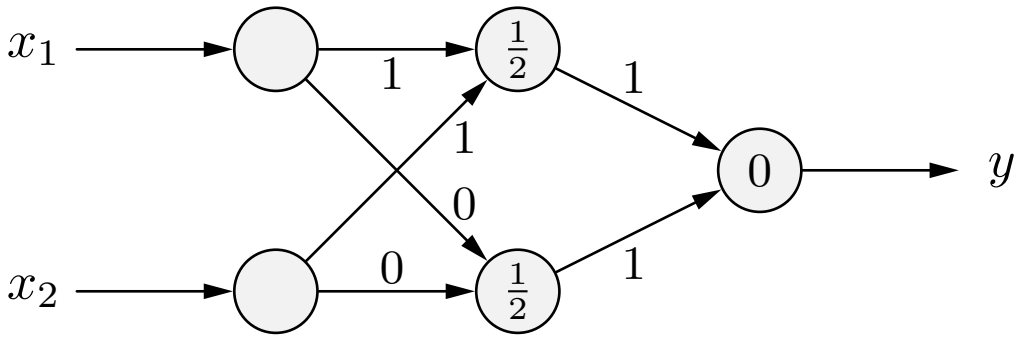


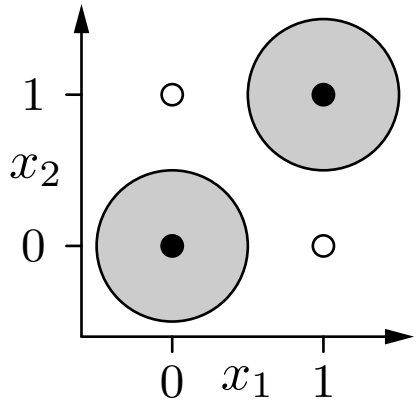
Radial Basis Function Networks: Examples

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

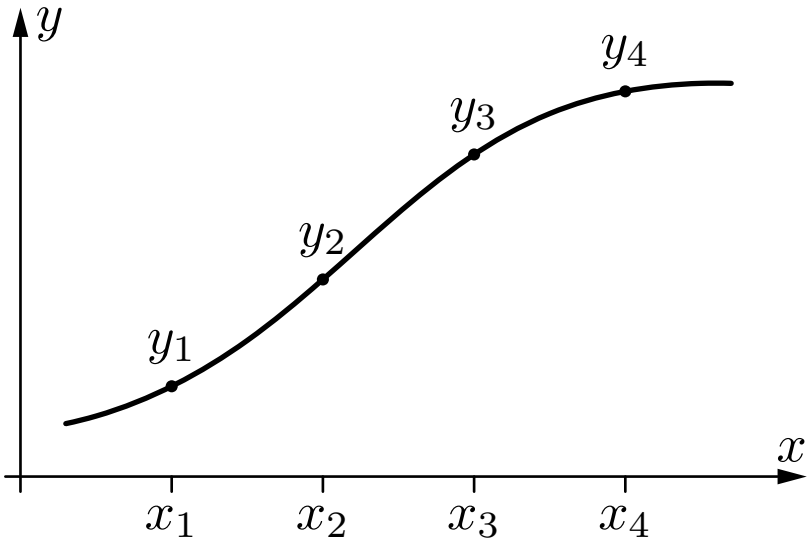
Idea: logical decomposition

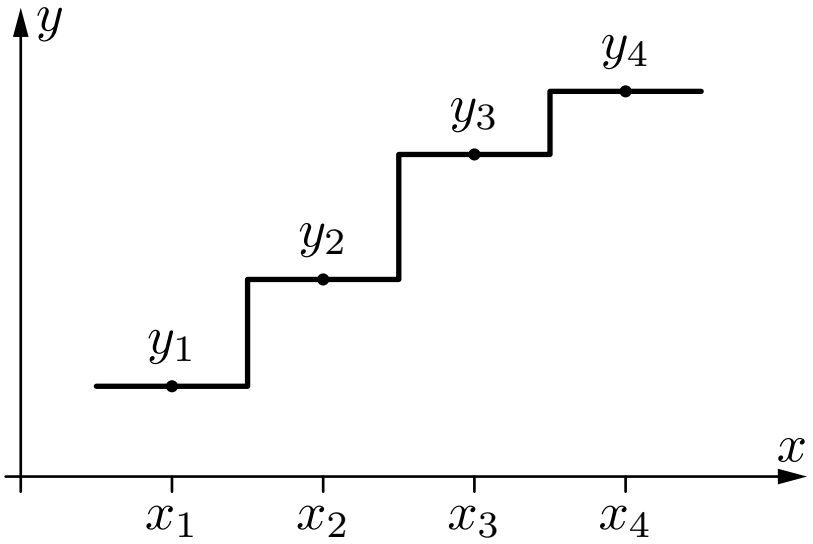
$$x_1 \leftrightarrow x_2 \quad \equiv \quad (x_1 \wedge x_2) \vee \neg(x_1 \vee x_2)$$



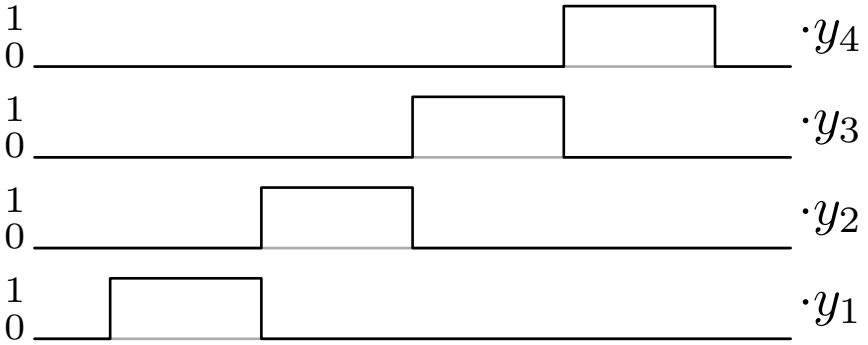


Radial Basis Function Networks: Function Approximation

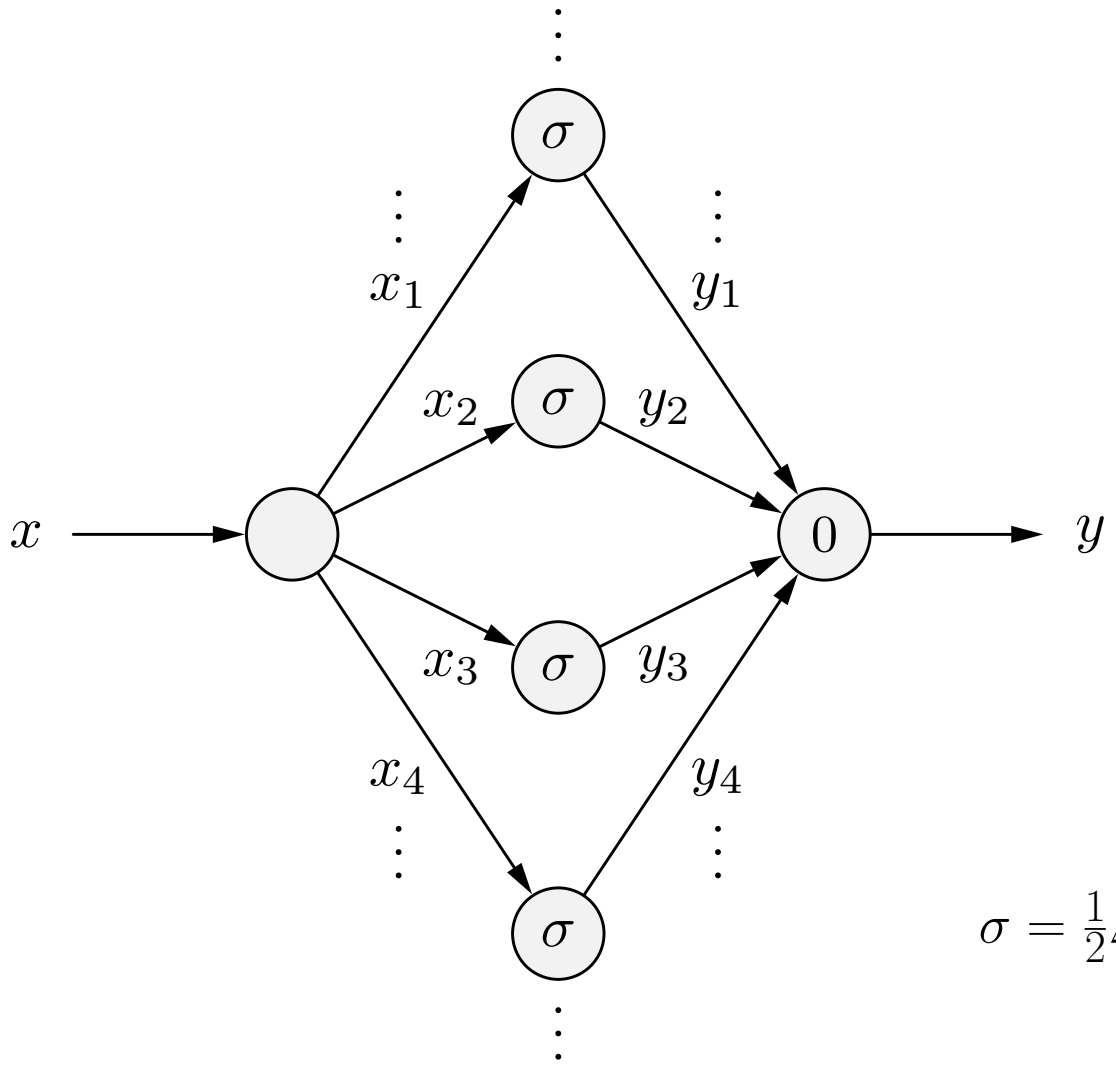




Approximation of a function by rectangular pulses, each of which can be represented by a neuron of an radial basis function network.



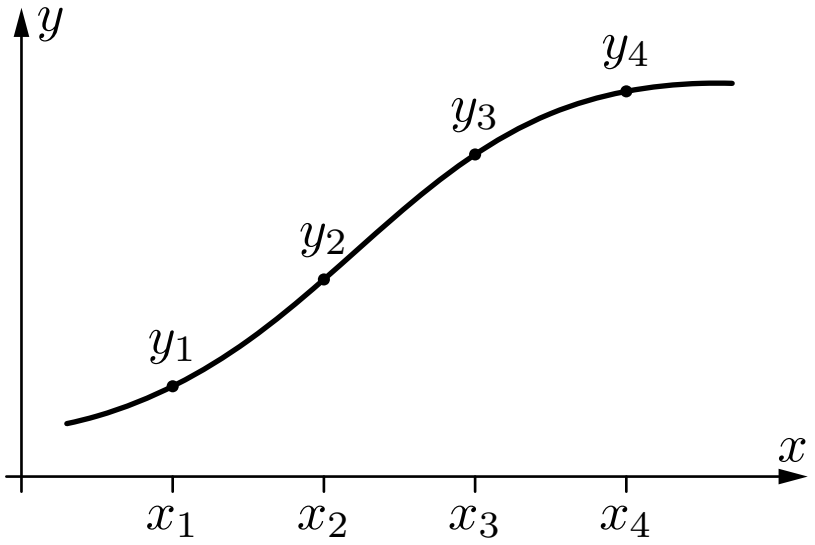
Radial Basis Function Networks: Function Approximation

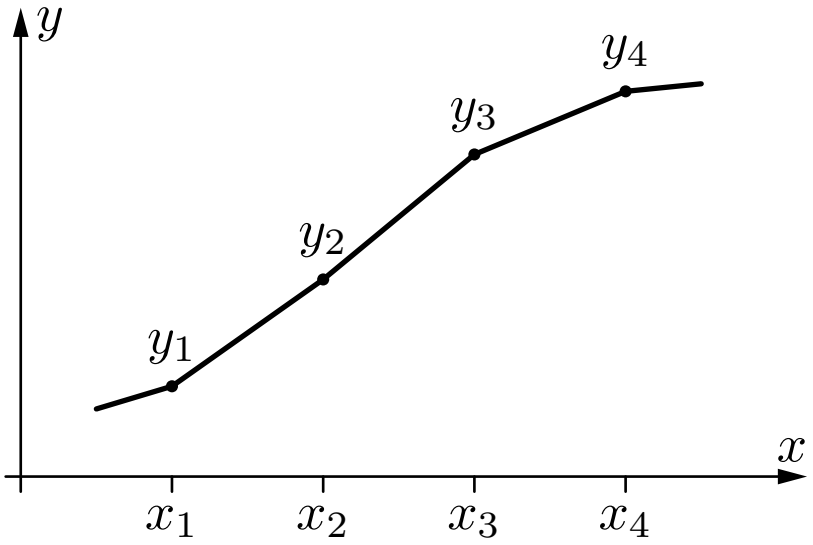


$$\sigma = \frac{1}{2}\Delta x = \frac{1}{2}(x_{i+1} - x_i)$$

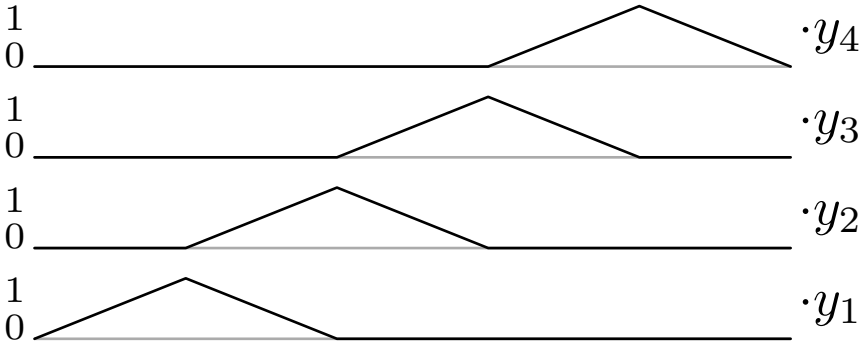
A radial basis function network that computes the step function on the preceding slide and the piecewise linear function on the next slide (depends on activation function).

Radial Basis Function Networks: Function Approximation

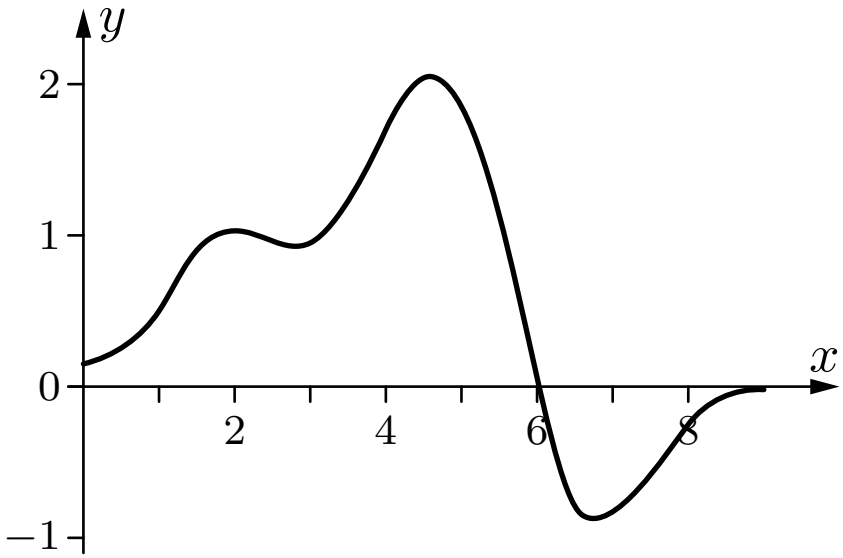


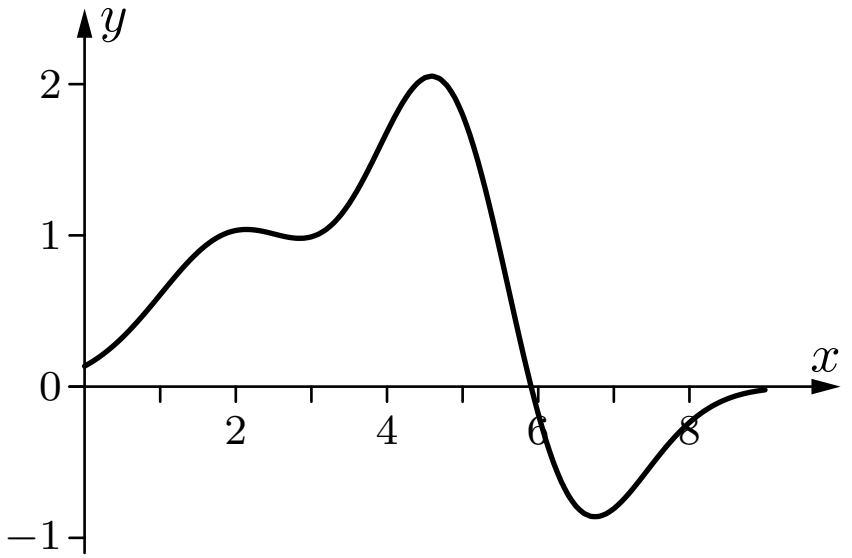


Approximation of a function by triangular pulses, each of which can be represented by a neuron of an radial basis function network.

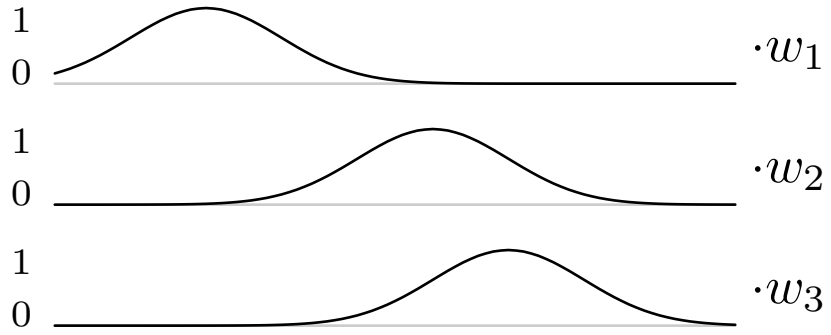


Radial Basis Function Networks: Function Approximation



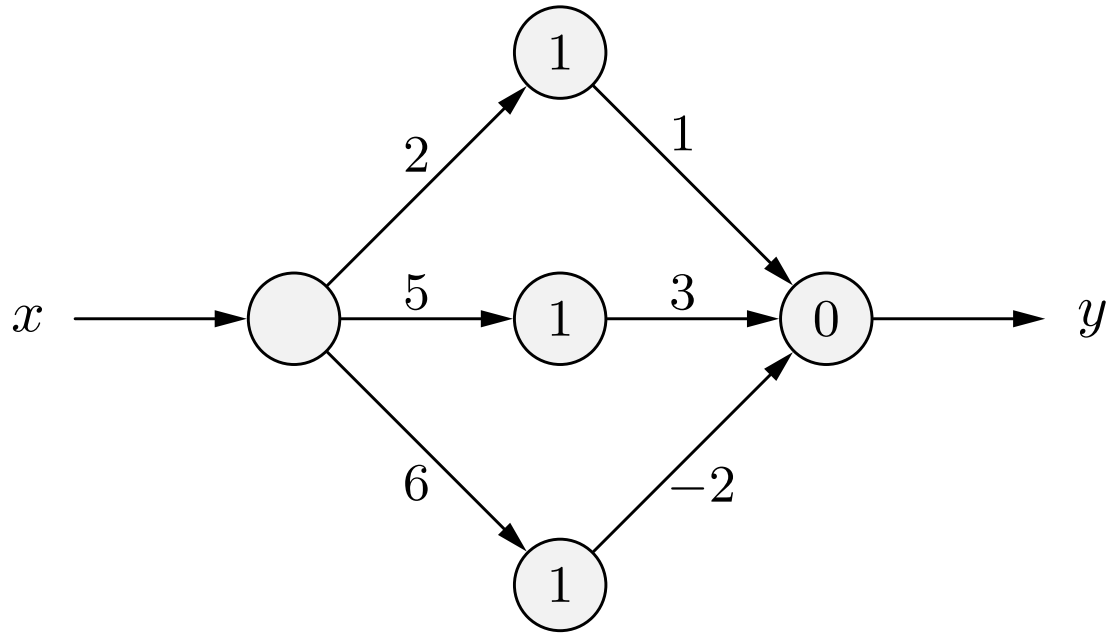


Approximation of a function by Gaussian functions with radius $\sigma = 1$. It is $w_1 = 1$, $w_2 = 3$ and $w_3 = -2$.



Radial Basis Function Networks: Function Approximation

Radial basis function network for a sum of three Gaussian functions



- The weights of the connections from the input neuron to the hidden neurons determine the locations of the Gaussian functions.
- The weights of the connections from the hidden neurons to the output neuron determine the height/direction (upward or downward) of the Gaussian functions.

Training Radial Basis Function Networks

Radial Basis Function Networks: Initialization

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

Simple radial basis function network:
One hidden neuron v_k, $k = 1, \ldots, m$, for each training pattern:

$$\forall k \in \{1, \ldots, m\}: \qquad \vec{w}_{v_k} = \vec{\imath}^{\,(l_k)}.$$

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

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

where

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

Radial Basis Function Networks: Initialization

Initializing the connections from the hidden to the output neurons

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

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

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

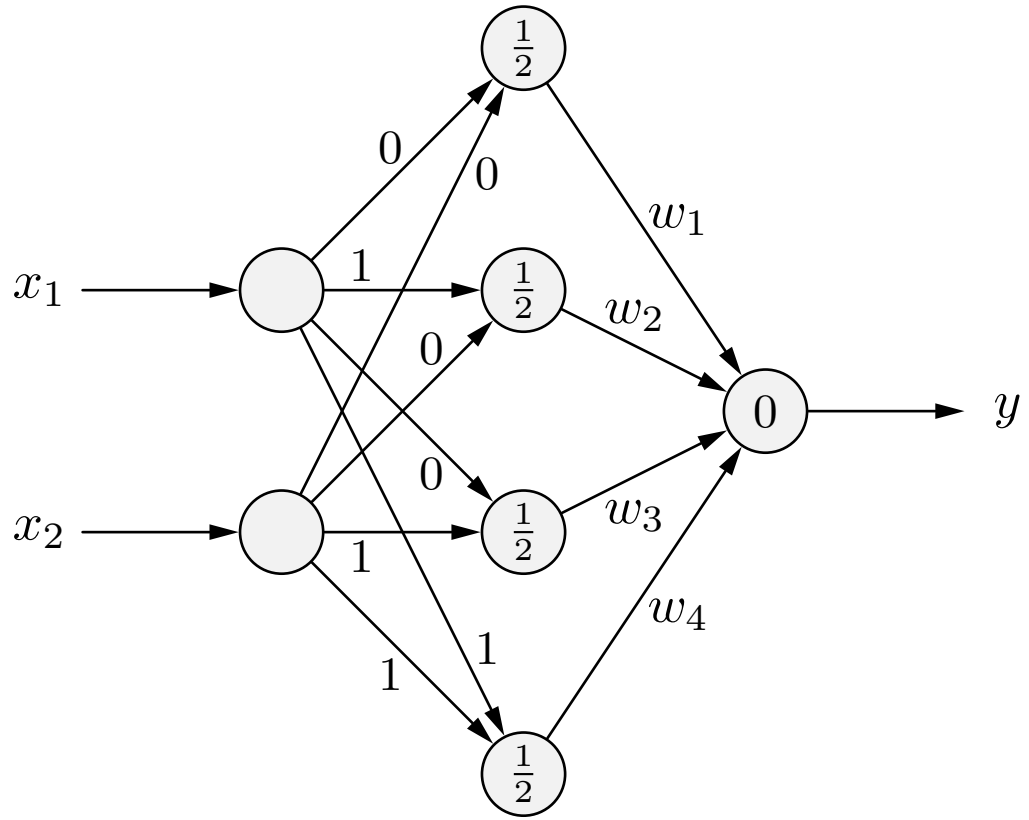
This is a linear equation system, that can be solved by inverting the matrix $\mathbf{A}$:

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

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 $x_1 \leftrightarrow x_2$

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

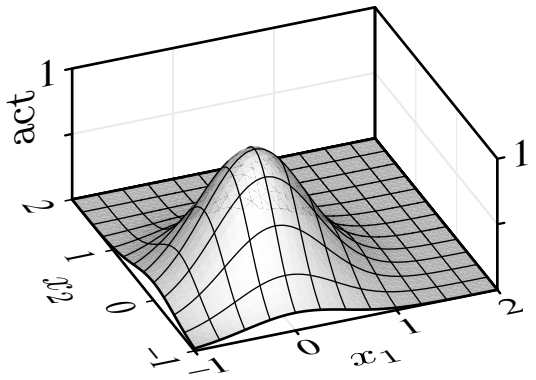
where

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

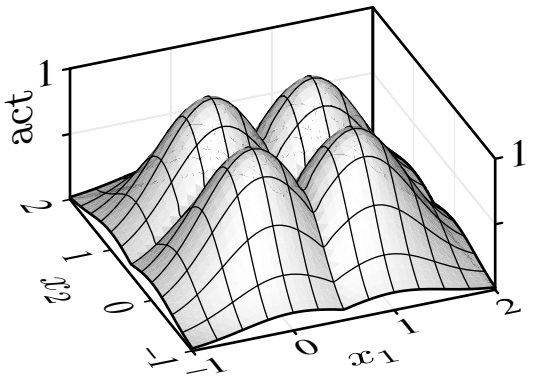
$$\vec{w}_u = \mathbf{A}^{-1} \cdot \vec{o}_u = \frac{1}{D} \begin{pmatrix} a+c \\ 2b \\ 2b \\ a+c \end{pmatrix} \approx \begin{pmatrix} 1.0567 \\ -0.2809 \\ -0.2809 \\ 1.0567 \end{pmatrix}$$

RBFN Initialization: Example

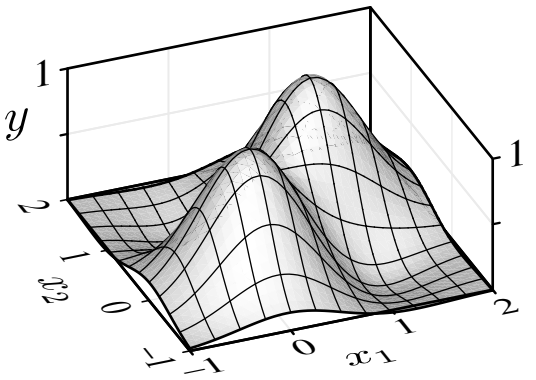
Simple radial basis function network for the biimplication $x_1 \leftrightarrow x_2$



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.

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

Compute (Moore–Penrose) pseudo inverse:

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

The weights can then be computed by

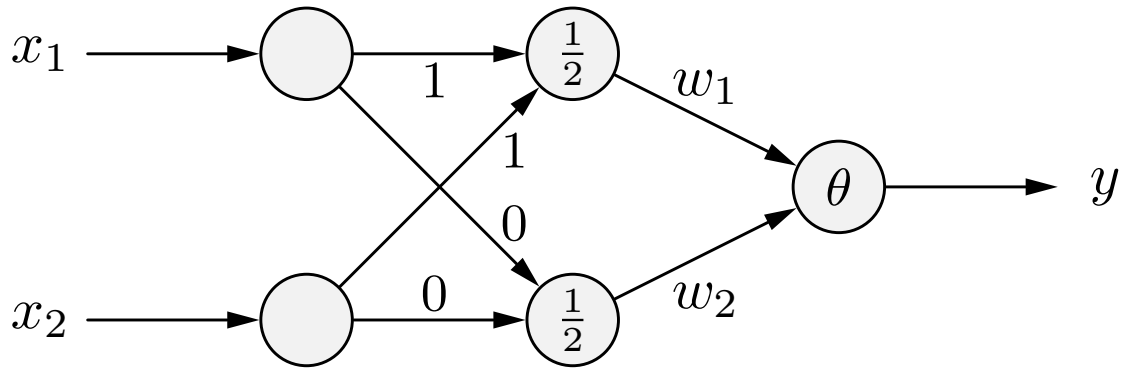
$$\vec{w}_u = \mathbf{A}^+ \cdot \vec{o}_u = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \cdot \vec{o}_u$$

RBFN Initialization: Example

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

Select two training patterns:

- $l_1 = (\vec{\imath}^{(l_1)}, \vec{o}^{(l_1)}) = ((0,0),(1))$
- $l_4 = (\vec{\imath}^{(l_4)}, \vec{o}^{(l_4)}) = ((1,1),(1))$



RBFN Initialization: Example

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

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

where

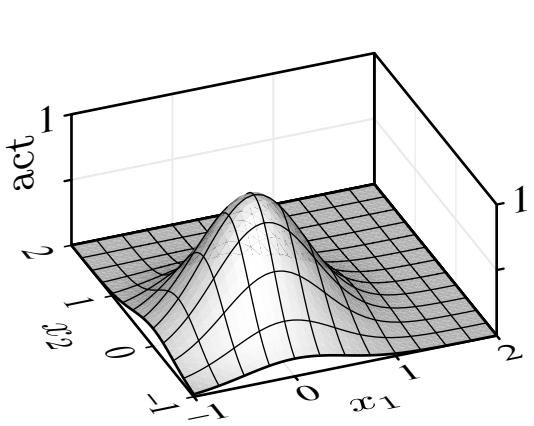
$$\begin{aligned} a &\approx -0.1810, & b &\approx 0.6810, \\ c &\approx 1.1781, & d &\approx -0.6688, & e &\approx 0.1594. \end{aligned}$$

Resulting weights:

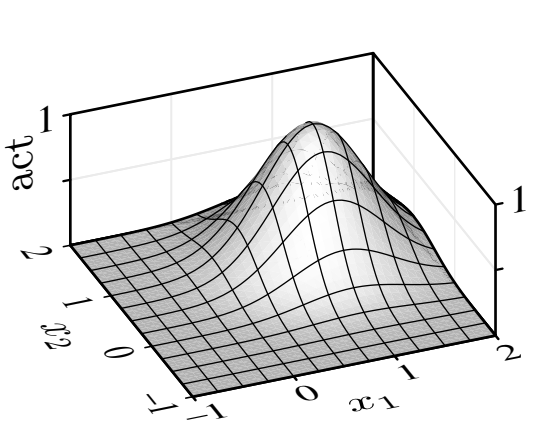
$$\vec{w}_u = \begin{pmatrix} -\theta \\ w_1 \\ w_2 \end{pmatrix} = \mathbf{A}^+ \cdot \vec{o}_u \approx \begin{pmatrix} -0.3620 \\ 1.3375 \\ 1.3375 \end{pmatrix}.$$

RBFN Initialization: Example

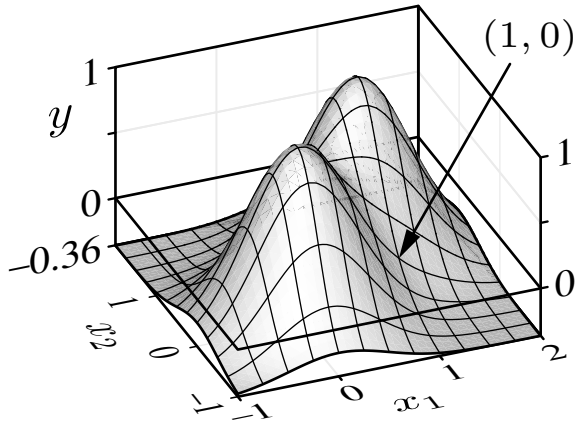
Normal radial basis function network for the biimplication $x_1 \leftrightarrow x_2$



basis function (0,0)



basis function (1,1)



output

- 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

How to choose the radial basis function centers?

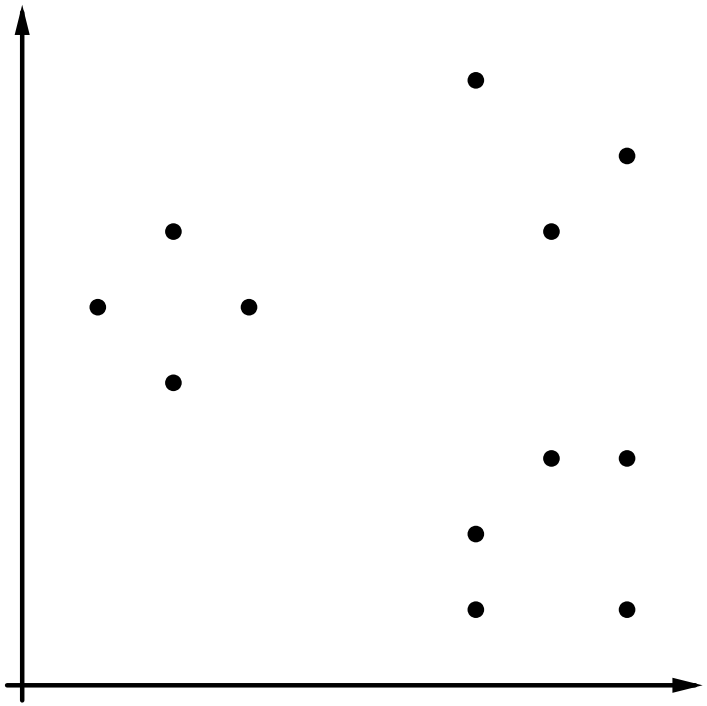
- Use **all data points** as centers for the radial basis functions.
  - Advantages: Only radius and output weights need to be determined; desired output values can be achieved exactly (unless there are inconsistencies).
  - Disadvantage: Often much too many radial basis functions; computing the weights to the output neuron via a pseudo-inverse can become infeasible.
- Use a **random subset** of data points as centers for the radial basis functions.
  - Advantages: Fast; only radius and output weights need to be determined.
  - Disadvantages: Performance depends heavily on the choice of data points.
- Use the **result of clustering** as centers for the radial basis functions, e.g.
  - c-means clustering (on the next slides)
  - Learning vector quantization (to be discussed later)

RBFN Initialization: c-means Clustering

- Choose a number c of clusters to be found (user input).
- Initialize the cluster centers randomly
  (for instance, by randomly selecting c data points).
- **Data point assignment**:
  Assign each data point to the cluster center that is closest to it
  (that is, closer than any other cluster center).
- **Cluster center update**:
  Compute new cluster centers as the mean vectors of the assigned data points.
  (Intuitively: center of gravity if each data point has unit weight.)
- Repeat these two steps (data point assignment and cluster center update)
  until the clusters centers do not change anymore.

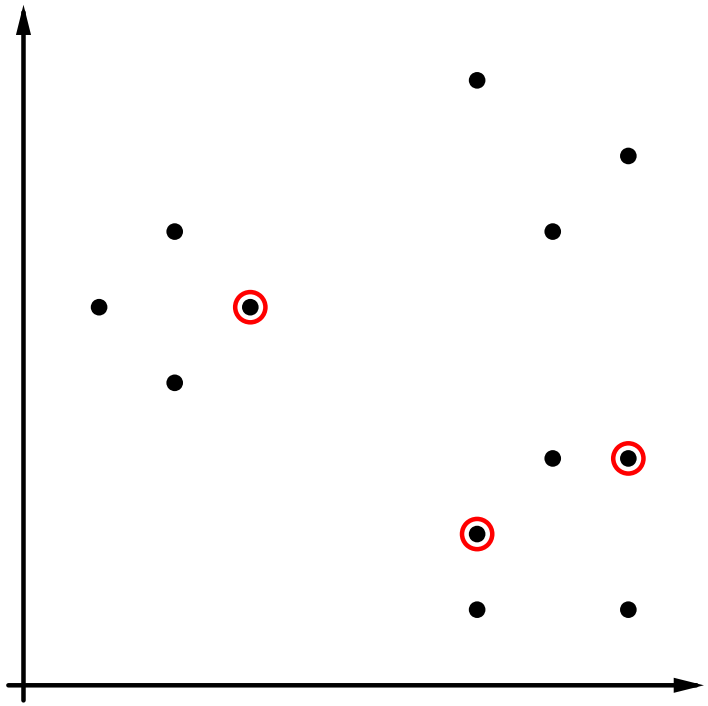
It can be shown that this scheme must converge,
that is, the update of the cluster centers cannot go on forever.

c-Means Clustering: Example



Data set to cluster.

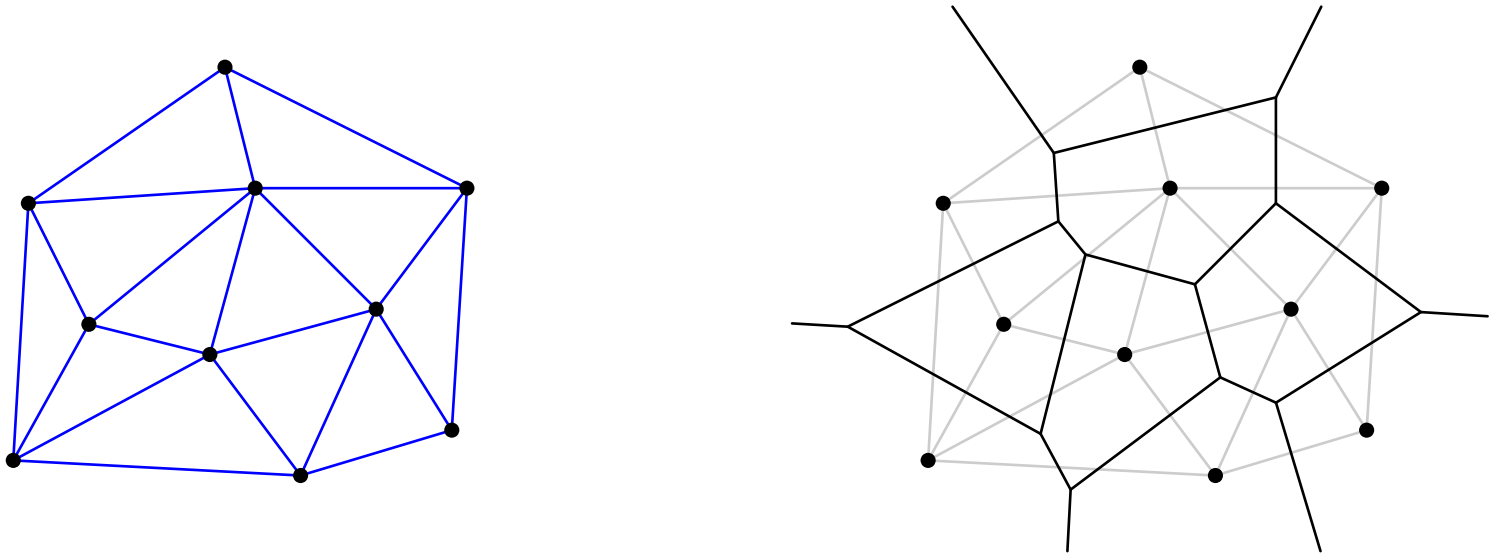
Choose $c = 3$ clusters.
(From visual inspection, can be difficult to determine in general.)



Initial position of cluster centers.

Randomly selected data points.
(Alternative methods include e.g. latin hypercube sampling)

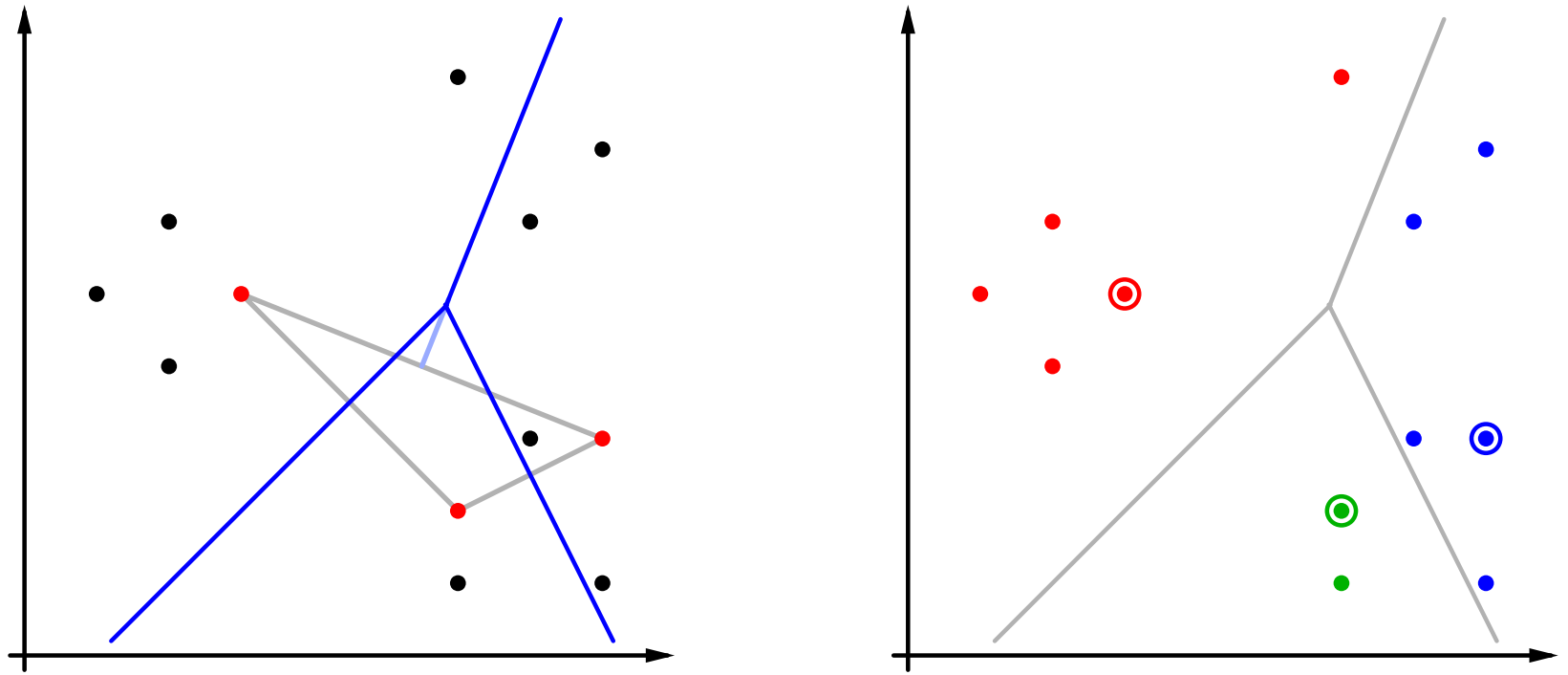
Delaunay Triangulations and Voronoi Diagrams



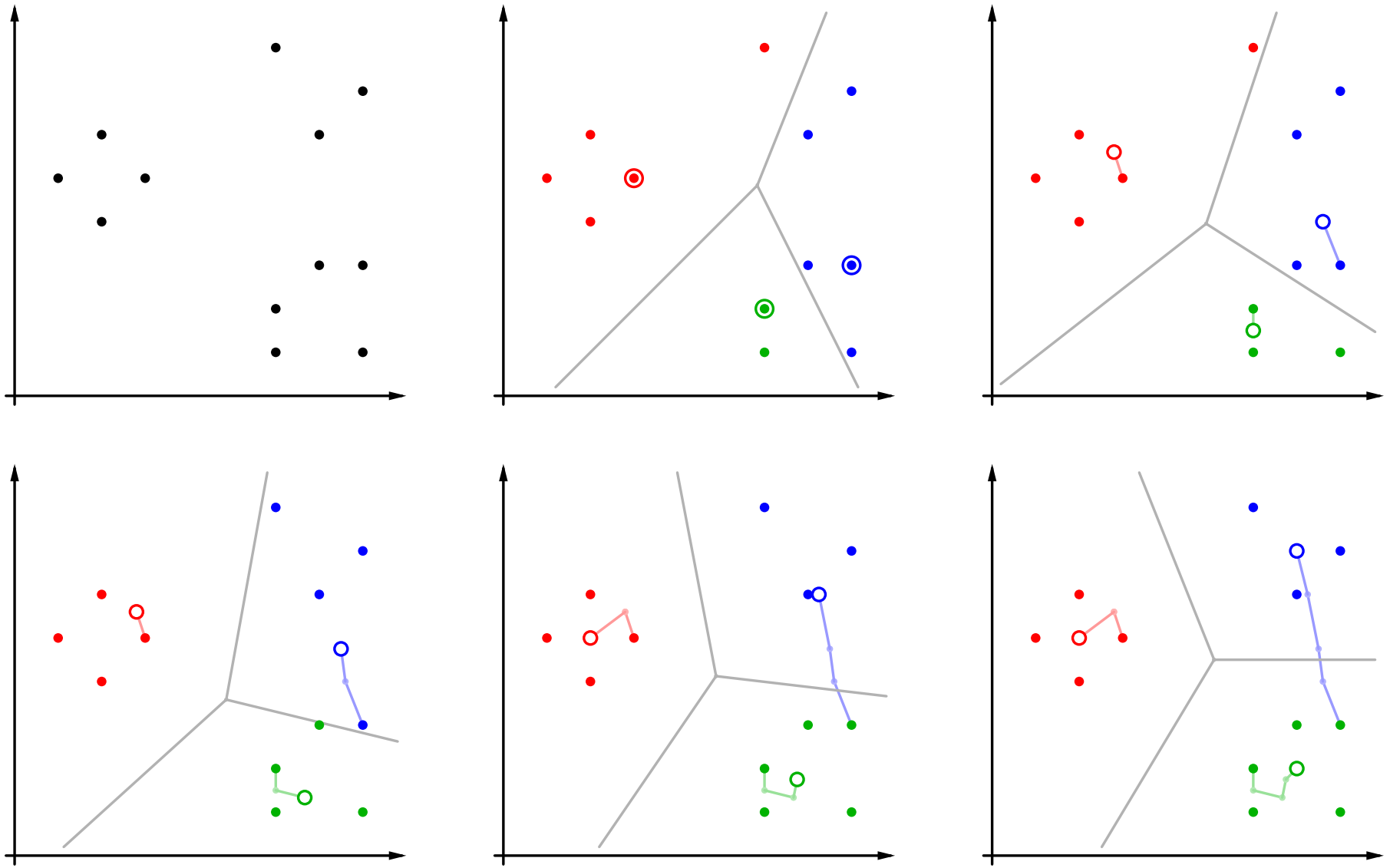
- Dots represent cluster centers.
- Left: **Delaunay Triangulation**
  The circle through the corners of a triangle does not contain another point.
- Right: **Voronoi Diagram / Tesselation**
  Midperpendiculars of the Delaunay triangulation: boundaries of the regions of points that are closest to the enclosed cluster center (Voronoi cells).

Delaunay Triangulations and Voronoi Diagrams

- **Delaunay Triangulation:** simple triangle (shown in gray on the left)
- **Voronoi Diagram:** midperpendiculars of the triangle's edges (shown in blue on the left, in gray on the right)



c-Means Clustering: Example



Radial Basis Function Networks: Training

Training radial basis function networks:
Derivation of update rules is analogous to that of multi-layer perceptrons.

Weights from the hidden to the output neurons.

Gradient:

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

Weight update rule:

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

Typical learning rate: $\eta_3 \approx 0.001$.

(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:

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

Weight update rule:

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

Typical learning rate: $\eta_1 \approx 0.02$.

Radial Basis Function Networks: Training

Training radial basis function networks:
Center coordinates (weights from the input to the hidden neurons).

Special case: **Euclidean distance**

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

Special case: **Gaussian activation function**

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

Radial Basis Function Networks: Training

Training radial basis function networks:
Radii of radial basis functions.

Gradient:

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

Weight update rule:

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

Typical learning rate: $\eta_2 \approx 0.01$.

Radial Basis Function Networks: Training

Training radial basis function networks:
Radii of radial basis functions.

Special case: **Gaussian activation function**

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

(The distance function is irrelevant for the radius update,
since it only enters the network input function.)

Radial Basis Function Networks: Generalization

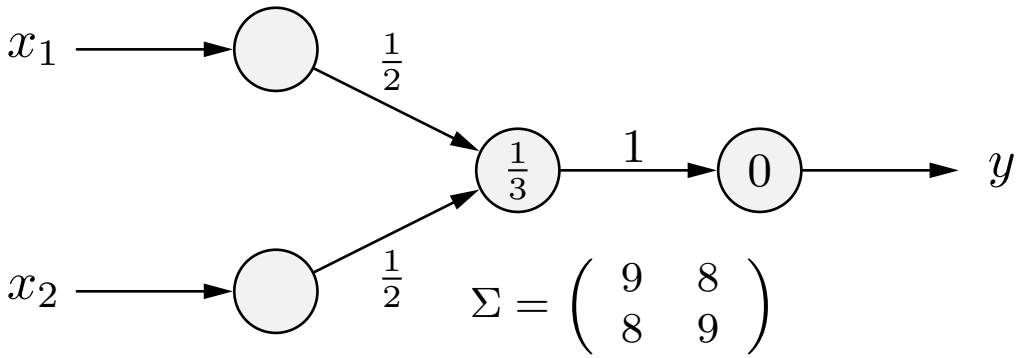
Generalization of the distance function

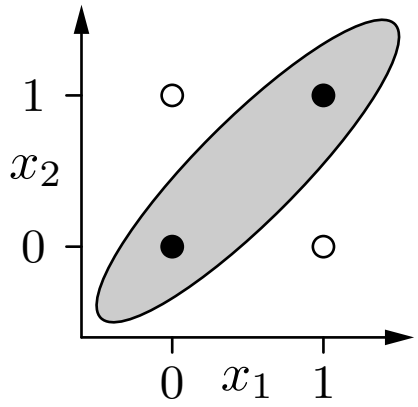
Idea: Use anisotropic (direction dependent) distance function.

Example: **Mahalanobis distance**

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

Example: **biimplication**





Application: Recognition of Handwritten Digits

- **Comparison of various classifiers:**
  - Nearest Neighbor (1NN)
  - Decision Tree (C4.5)
  - Multi-Layer Perceptron (MLP)
  - Learning Vector Quantization (LVQ)
  - Radial Basis Function Network (RBF)
  - Support Vector Machine (SVM)
- **Distinction of the number of RBF training phases:**
  - 1 phase: find output connection weights e.g. with pseudo-inverse.
  - 2 phase: find RBF centers e.g. with some clustering plus 1 phase.
  - 3 phase: 2 phase plus error backpropagation training.
- **Initialization of radial basis function centers:**
  - Random choice of data points
  - c-means Clustering
  - Learning Vector Quantization
  - Decision Tree (one RBF center per leaf)

Application: Recognition of Handwritten Digits

Classification results:

Classifier	Accuracy
Nearest Neighbor (1NN)	97.68%
Learning Vector Quantization (LVQ)	96.99%
Decision Tree (C4.5)	91.12%
2-Phase-RBF (data points)	95.24%
2-Phase-RBF (c-means)	96.94%
2-Phase-RBF (LVQ)	95.86%
2-Phase-RBF (C4.5)	92.72%
3-Phase-RBF (data points)	97.23%
3-Phase-RBF (c-means)	98.06%
3-Phase-RBF (LVQ)	98.49%
3-Phase-RBF (C4.5)	94.83%
Support Vector Machine (SVM)	98.76%
Multi-Layer Perceptron (MLP)	97.59%

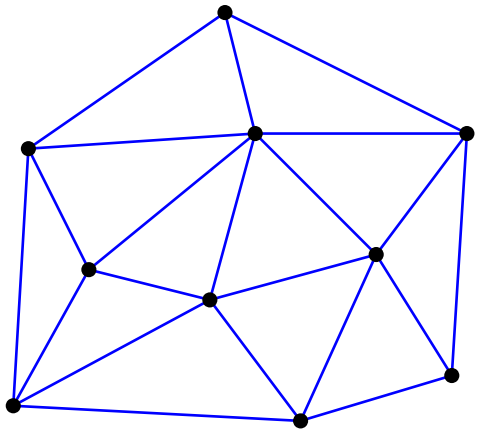
- LVQ: 200 vectors (20 per class)
  C4.5: 505 leaves
  c-means: 60 centers(?) (6 per class)
  SVM: 10 classifiers, $\approx$ 4200 vectors
  MLP: 1 hidden layer with 200 neurons
- Results are medians of three training/test runs.
- Error backpropagation improves RBF results.

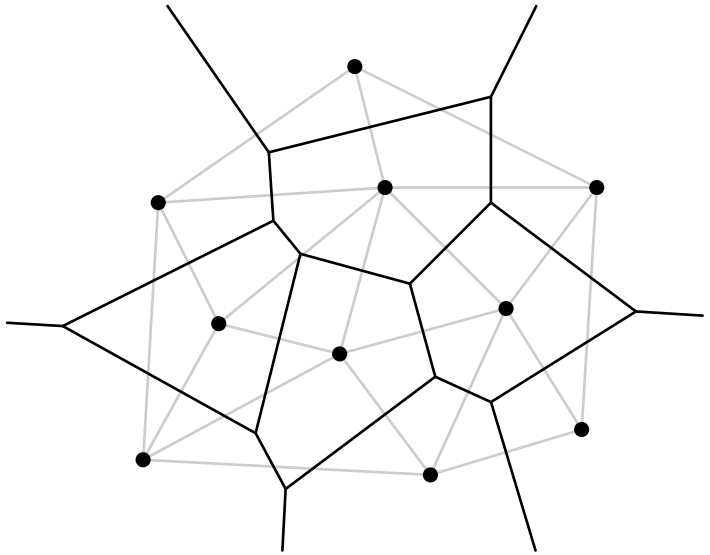
Learning Vector Quantization

Learning Vector Quantization

- Up to now: **fixed learning tasks**
  - The data consists of input/output pairs.
  - The objective is to produce desired output for given input.
  - This allows to describe training as error minimization.
- Now: **free learning tasks**
  - The data consists only of input values/vectors.
  - The objective is to produce similar output for similar input (clustering).
- **Learning Vector Quantization**
  - Find a suitable quantization (many-to-few mapping, often to a finite set) of the input space, e.g. a tesselation of a Euclidean space.
  - Training adapts the coordinates of so-called reference or codebook vectors, each of which defines a region in the input space.

Reminder: Delaunay Triangulations and Voronoi Diagrams

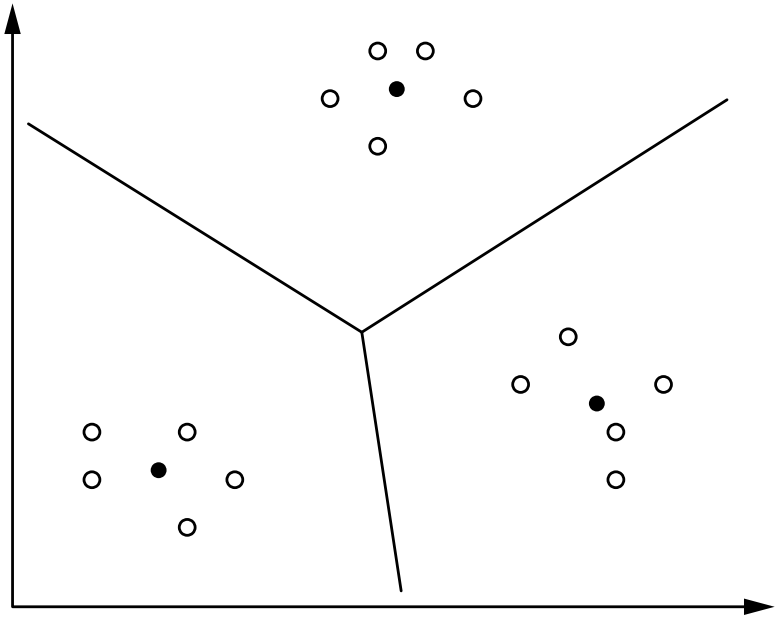




- Dots represent vectors that are used for quantizing the area.
- Left: **Delaunay Triangulation**
  (The circle through the corners of a triangle does not contain another point.)
- Right: **Voronoi Diagram / Tesselation**
  (Midperpendiculars of the Delaunay triangulation: boundaries of the regions of points that are closest to the enclosed cluster center (Voronoi cells)).

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, that is,

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

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

$$\begin{aligned}
&(i) && d(\vec{x}, \vec{y}) = 0 \quad \Leftrightarrow \quad \vec{x} = \vec{y}, && \\
&(ii) && d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x}) && \text{(symmetry)}, \\
&(iii) && d(\vec{x}, \vec{z}) \leq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z}) && \text{(triangle inequality)}.
\end{aligned}$$

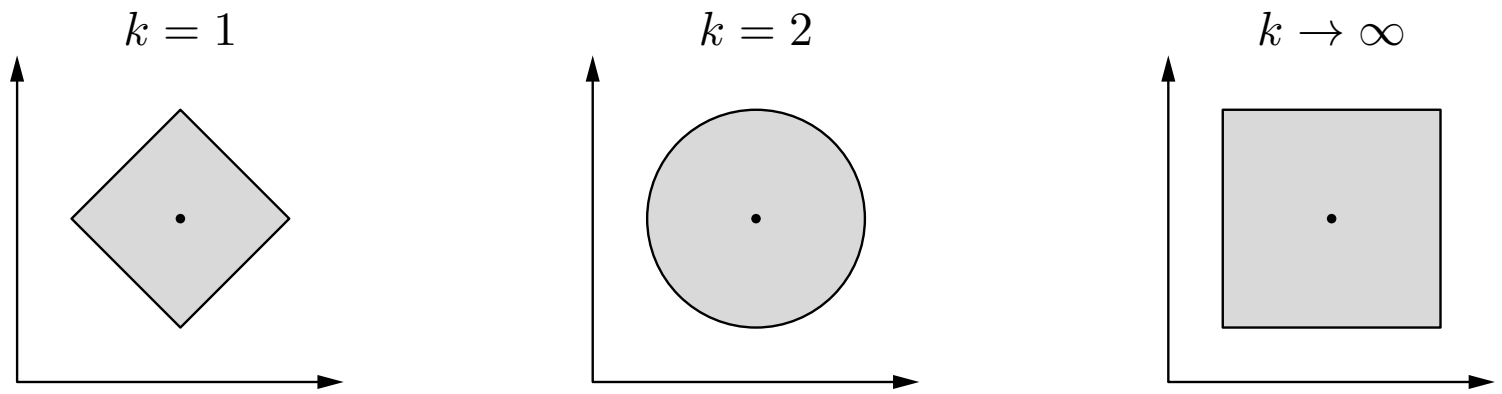
Reminder: Distance Functions

Illustration of distance functions: Minkowski family

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

Well-known special cases from this family are:

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



Learning Vector Quantization

The activation function of each output neuron is a so-called **radial function**, that is, a monotonically decreasing function

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

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:

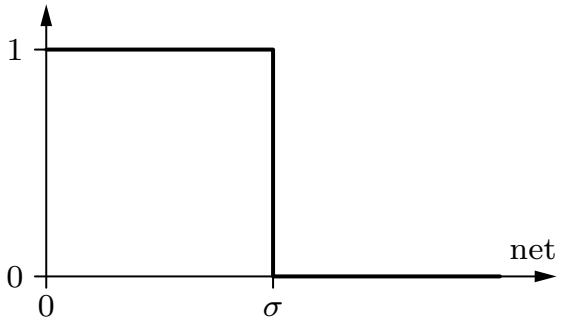
$$f_{\text{out}}^{(u)}(\text{act}_u) = \begin{cases} 1, & \text{if } \text{act}_u = \max\limits_{v \in U_{\text{out}}} \text{act}_v, \\ 0, & \text{otherwise.} \end{cases}$$

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

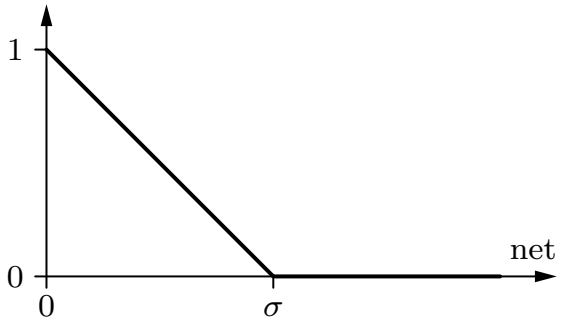
rectangle function:

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



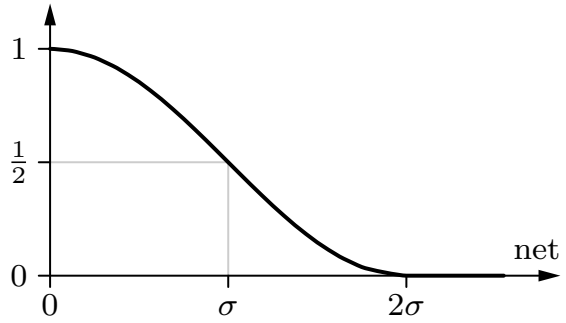
triangle function:

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



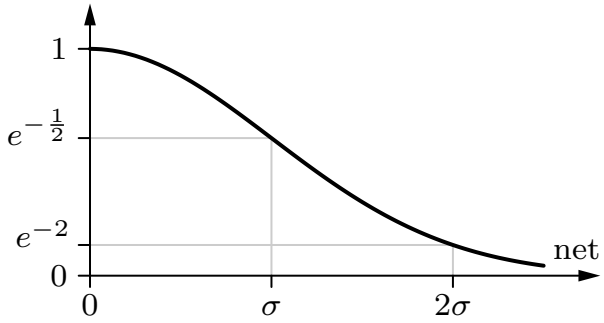
cosine until zero:

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



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)

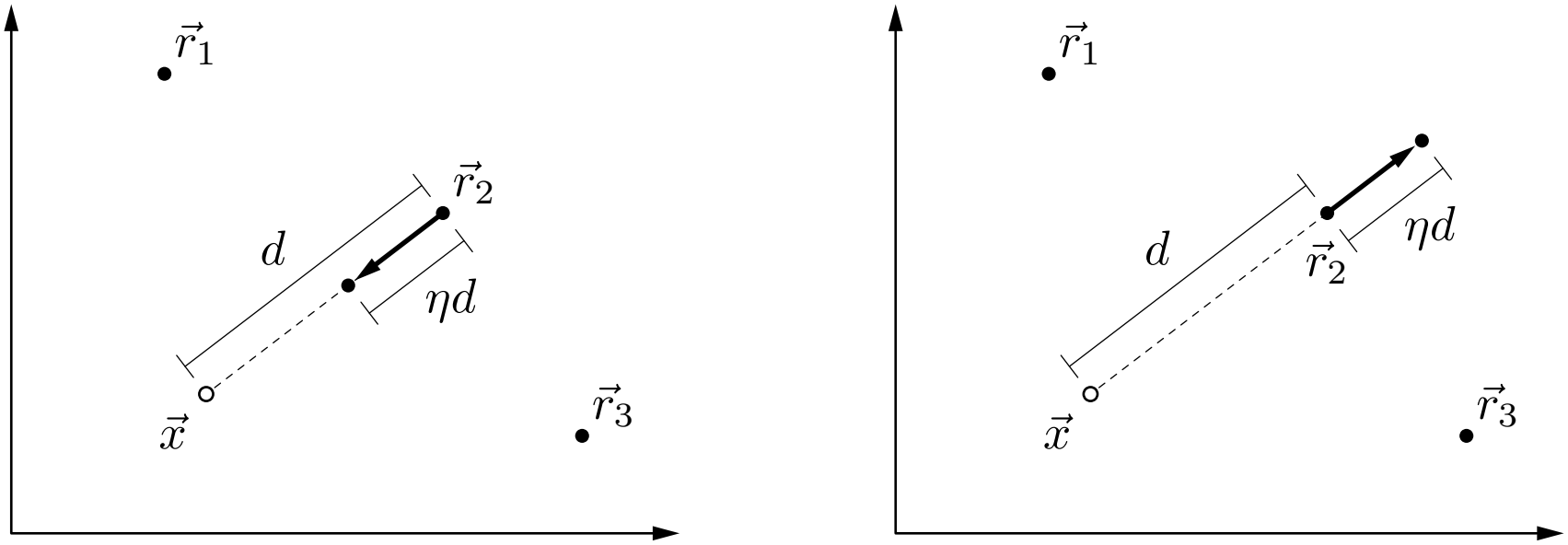
$$\vec{r}^{\,(\text{new})} = \vec{r}^{\,(\text{old})} + \eta(\vec{x} - \vec{r}^{\,(\text{old})}),$$

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

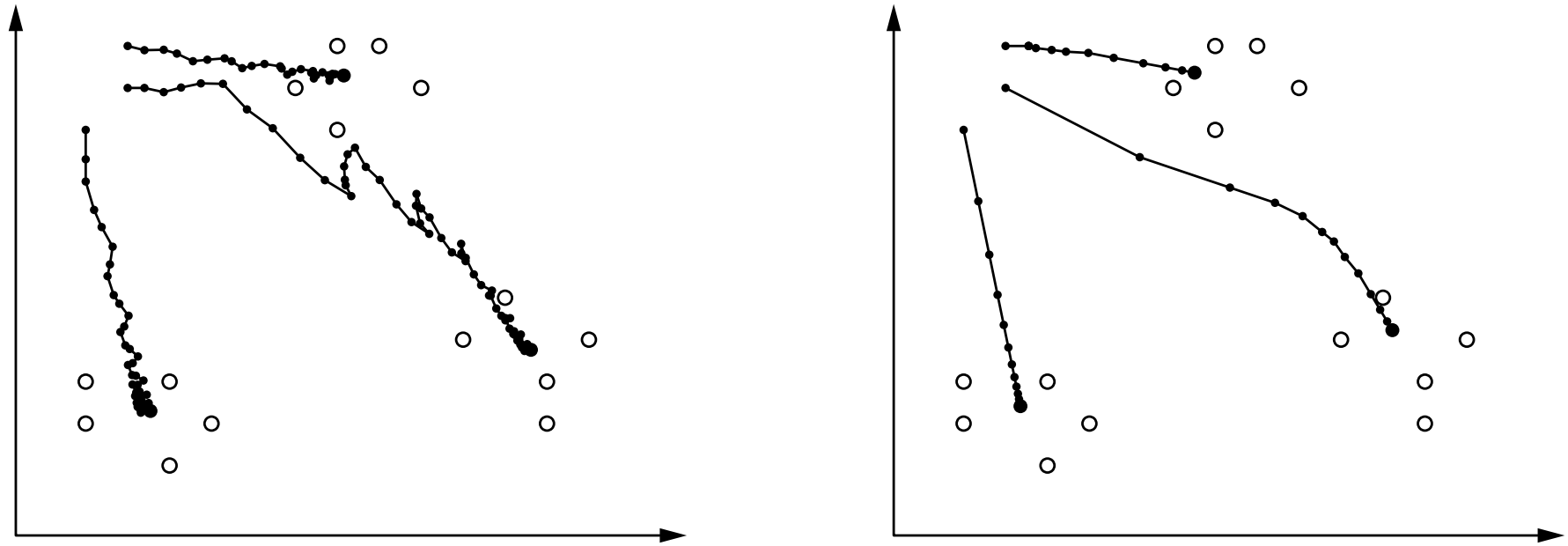
Adaptation of reference vectors / codebook vectors



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

Learning Vector Quantization: Example

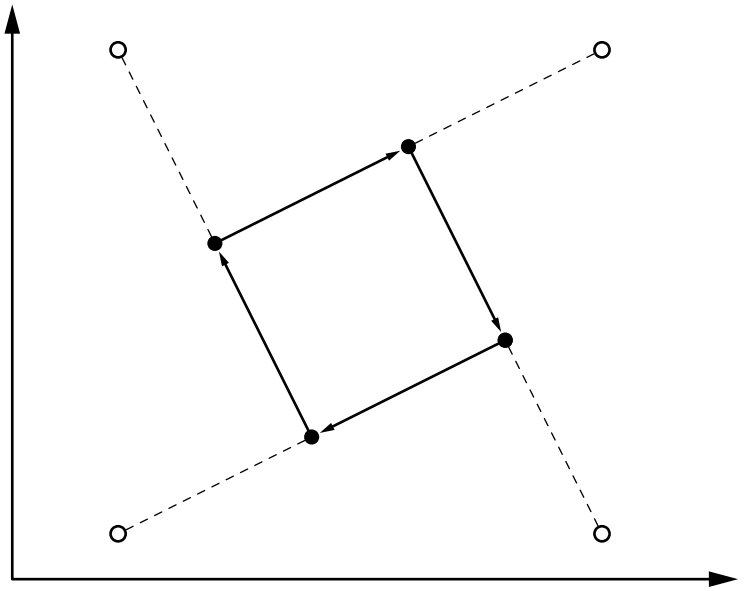
Adaptation of reference vectors / codebook vectors

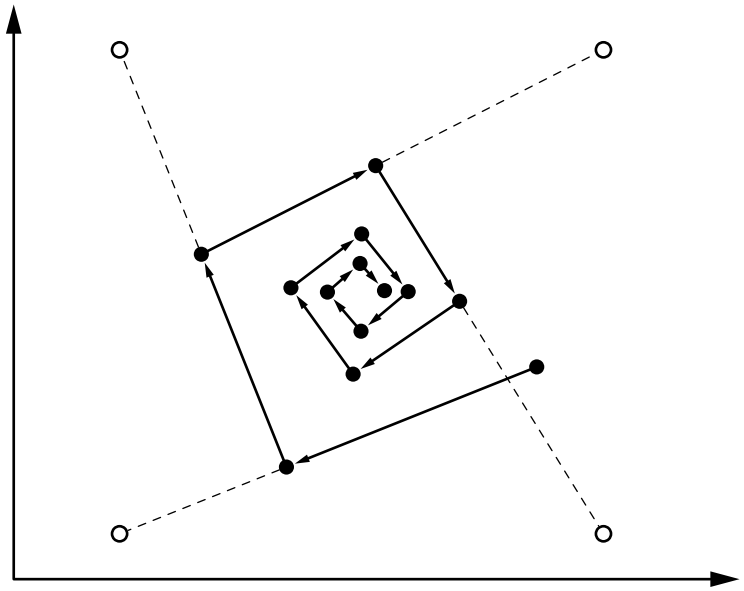


- Left: Online training with learning rate $\eta = 0.1$,
- Right: Batch training with learning rate $\eta = 0.05$.

Learning Vector Quantization: Learning Rate Decay

Problem: fixed learning rate can lead to oscillations





Solution: **time dependent learning rate**

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

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:

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

  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

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

  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 (approach descibed here: [Seo and Obermayer 2003]).
- **Assumption:** Given data was sampled from a mixture of normal distributions. Each reference vector describes one normal distribution.
- Closely related to clustering by estimating a **mixture of Gaussians**.
  - (Crisp or hard) learning vector quantization can be seen as an "online version" of c-means clustering.
  - Soft learning vector quantization can be seed as an "online version" of estimating a mixture of Gaussians (that is, of normal distributions). (In the following: brief review of the Expectation Maximization (EM) Algorithm for estimating a mixture of Gaussians.)
- Hardening soft learning vector quantization (by letting the "radii" of the Gaussians go to zero, see below) yields a version of (crisp or hard) learning vector quantization that works well without a window rule.

Expectation Maximization: Mixture of Gaussians

- **Assumption:** Data was generated by sampling a set of normal distributions. (The probability density is a mixture of Gaussian distributions.)
- **Formally:** We assume that the probability density can be described as

$$f_{\vec{X}}(\vec{x}; \mathbf{C}) = \sum_{y=1}^{c} f_{\vec{X},Y}(\vec{x}, y; \mathbf{C}) = \sum_{y=1}^{c} p_Y(y; \mathbf{C}) \cdot f_{\vec{X}|Y}(\vec{x}|y; \mathbf{C}).$$

$\mathbf{C}$	is the set of cluster parameters
$\vec{X}$	is a random vector that has the data space as its domain
Y	is a random variable that has the cluster indices as possible values (i.e., $\mathrm{dom}(\vec{X}) = \mathbb{R}^m$ and $\mathrm{dom}(Y) = \{1, \ldots, c\}$)
$p_Y(y; \mathbf{C})$	is the probability that a data point belongs to (is generated by) the y-th component of the mixture
$f_{\vec{X}\|Y}(\vec{x}\|y; \mathbf{C})$	is the conditional probability density function of a data point given the cluster (specified by the cluster index y)

Expectation Maximization

- **Basic idea:** Do a maximum likelihood estimation of the cluster parameters.
- **Problem:** The likelihood function,

$$L(\mathbf{X}; \mathbf{C}) = \prod_{j=1}^{n} f_{\vec{X}_j}(\vec{x}_j; \mathbf{C}) = \prod_{j=1}^{n} \sum_{y=1}^{c} p_Y(y; \mathbf{C}) \cdot f_{\vec{X}|Y}(\vec{x}_j|y; \mathbf{C}),$$

  is difficult to optimize, even if one takes the natural logarithm (cf. the maximum likelihood estimation of the parameters of a normal distribution), because

$$\ln L(\mathbf{X}; \mathbf{C}) = \sum_{j=1}^{n} \ln \sum_{y=1}^{c} p_Y(y; \mathbf{C}) \cdot f_{\vec{X}|Y}(\vec{x}_j|y; \mathbf{C})$$

  contains the natural logarithms of complex sums.
- **Approach:** Assume that there are "hidden" variables Y_j stating the clusters that generated the data points $\vec{x}_j$, so that the sums reduce to one term.
- **Problem:** Since the Y_j are hidden, we do not know their values.

Expectation Maximization

- **Formally:** Maximize the likelihood of the "completed" data set $(\mathbf{X}, \vec{y})$, where $\vec{y} = (y_1, \ldots, y_n)$ combines the values of the variables Y_j. That is,

$$L(\mathbf{X}, \vec{y}; \mathbf{C}) = \prod_{j=1}^{n} f_{\vec{X}_j, Y_j}(\vec{x}_j, y_j; \mathbf{C}) = \prod_{j=1}^{n} p_{Y_j}(y_j; \mathbf{C}) \cdot f_{\vec{X}_j|Y_j}(\vec{x}_j|y_j; \mathbf{C}).$$

- **Problem:** Since the Y_j are hidden, the values y_j are unknown (and thus the factors $p_{Y_j}(y_j; \mathbf{C})$ cannot be computed).

- **Approach to find a solution nevertheless:**
  - See the Y_j as random variables (the values y_j are not fixed) and consider a probability distribution over the possible values.
  - As a consequence $L(\mathbf{X}, \vec{y}; \mathbf{C})$ becomes a random variable, even for a fixed data set $\mathbf{X}$ and fixed cluster parameters $\mathbf{C}$.
  - Try to **maximize the expected value** of $L(\mathbf{X}, \vec{y}; \mathbf{C})$ or $\ln L(\mathbf{X}, \vec{y}; \mathbf{C})$ (hence the name **expectation maximization**).

Expectation Maximization

- **Formally:** Find the cluster parameters as

$$\hat{\mathbf{C}} = \underset{\mathbf{C}}{\operatorname{argmax}}\, E([\ln] L(\mathbf{X}, \vec{y}; \mathbf{C}) \mid \mathbf{X}; \mathbf{C}),$$

  that is, maximize the expected likelihood

$$E(L(\mathbf{X}, \vec{y}; \mathbf{C}) \mid \mathbf{X}; \mathbf{C}) = \sum_{\vec{y} \in \{1,\ldots,c\}^n} p_{\vec{Y}|\mathcal{X}}(\vec{y}\,|\mathbf{X}; \mathbf{C}) \cdot \prod_{j=1}^{n} f_{\vec{X}_j, Y_j}(\vec{x}_j, y_j; \mathbf{C})$$

  or, alternatively, maximize the expected log-likelihood

$$E(\ln L(\mathbf{X}, \vec{y}; \mathbf{C}) \mid \mathbf{X}; \mathbf{C}) = \sum_{\vec{y} \in \{1,\ldots,c\}^n} p_{\vec{Y}|\mathcal{X}}(\vec{y}\,|\mathbf{X}; \mathbf{C}) \cdot \sum_{j=1}^{n} \ln f_{\vec{X}_j, Y_j}(\vec{x}_j, y_j; \mathbf{C}).$$

- Unfortunately, these functionals are still difficult to optimize directly.
- **Solution:** Use the equation as an iterative scheme, fixing $\mathbf{C}$ in some terms (iteratively compute better approximations, similar to Heron's algorithm).

Excursion: Heron's Algorithm

- **Task:** Find the square root of a given number x, i.e., find $y = \sqrt{x}$.
- **Approach:** Rewrite the defining equation $y^2 = x$ as follows:

$$y^2 = x \quad \Leftrightarrow \quad 2y^2 = y^2 + x \quad \Leftrightarrow \quad y = \frac{1}{2y}(y^2 + x) \quad \Leftrightarrow \quad y = \frac{1}{2}\left(y + \frac{x}{y}\right).$$

- Use the resulting equation as an iteration formula, i.e., compute the sequence

$$y_{k+1} = \frac{1}{2}\left(y_k + \frac{x}{y_k}\right) \qquad \text{with} \qquad y_0 = 1.$$

- It can be shown that $0 \leq y_k - \sqrt{x} \leq y_{k-1} - y_n$ for $k \geq 2$.
  Therefore this iteration formula provides increasingly better approximations of the square root of x and thus is a safe and simple way to compute it.
  Ex.: $x = 2$: $y_0 = 1$, $y_1 = 1.5$, $y_2 \approx 1.41667$, $y_3 \approx 1.414216$, $y_4 \approx 1.414213$.
- Heron's algorithm converges very quickly and is often used in pocket calculators and microprocessors to implement the square root.

Expectation Maximization

- **Iterative scheme for expectation maximization:**

  Choose some initial set $\mathbf{C}_0$ of cluster parameters and then compute

$$\begin{aligned}
\mathbf{C}_{k+1} &= \operatorname*{argmax}_{\mathbf{C}} E(\ln L(\mathbf{X}, \vec{y}; \mathbf{C}) \mid \mathbf{X}; \mathbf{C}_k) \\
&= \operatorname*{argmax}_{\mathbf{C}} \sum_{\vec{y} \in \{1,\ldots,c\}^n} p_{\vec{Y}|\mathcal{X}}(\vec{y}\,|\mathbf{X}; \mathbf{C}_k) \sum_{j=1}^{n} \ln f_{\vec{X}_j, Y_j}(\vec{x}_j, y_j; \mathbf{C}) \\
&= \operatorname*{argmax}_{\mathbf{C}} \sum_{\vec{y} \in \{1,\ldots,c\}^n} \left(\prod_{l=1}^{n} p_{Y_l|\vec{X}_l}(y_l|\vec{x}_l; \mathbf{C}_k) \right) \sum_{j=1}^{n} \ln f_{\vec{X}_j, Y_j}(\vec{x}_j, y_j; \mathbf{C}) \\
&= \operatorname*{argmax}_{\mathbf{C}} \sum_{i=1}^{c} \sum_{j=1}^{n} p_{Y_j|\vec{X}_j}(i|\vec{x}_j; \mathbf{C}_k) \cdot \ln f_{\vec{X}_j, Y_j}(\vec{x}_j, i; \mathbf{C}).
\end{aligned}$$

- It can be shown that each EM iteration increases the likelihood of the data and that the algorithm converges to a local maximum of the likelihood function (i.e., EM is a safe way to maximize the likelihood function).

Expectation Maximization

Justification of the last step on the previous slide:

$$\sum_{\vec{y}\in\{1,\ldots,c\}^n} \left(\prod_{l=1}^{n} p_{Y_l|\vec{X}_l}(y_l|\vec{x}_l;\mathbf{C}_k)\right) \sum_{j=1}^{n} \ln f_{\vec{X}_j,Y_j}(\vec{x}_j,y_j;\mathbf{C})$$

$$= \sum_{y_1=1}^{c} \cdots \sum_{y_n=1}^{c} \left(\prod_{l=1}^{n} p_{Y_l|\vec{X}_l}(y_l|\vec{x}_l;\mathbf{C}_k)\right) \sum_{j=1}^{n}\sum_{i=1}^{c} \delta_{i,y_j} \ln f_{\vec{X}_j,Y_j}(\vec{x}_j,i;\mathbf{C})$$

$$= \sum_{i=1}^{c}\sum_{j=1}^{n} \ln f_{\vec{X}_j,Y_j}(\vec{x}_j,i;\mathbf{C}) \sum_{y_1=1}^{c} \cdots \sum_{y_n=1}^{c} \delta_{i,y_j} \prod_{l=1}^{n} p_{Y_l|\vec{X}_l}(y_l|\vec{x}_l;\mathbf{C}_k)$$

$$= \sum_{i=1}^{c}\sum_{j=1}^{n} p_{Y_j|\vec{X}_j}(i|\vec{x}_j;\mathbf{C}_k) \cdot \ln f_{\vec{X}_j,Y_j}(\vec{x}_j,i;\mathbf{C})$$

$$\underbrace{\sum_{y_1=1}^{c} \cdots \sum_{y_{j-1}=1}^{c} \sum_{y_{j+1}=1}^{c} \cdots \sum_{y_n=1}^{c} \prod_{l=1,l\neq j}^{n} p_{Y_l|\vec{X}_l}(y_l|\vec{x}_l;\mathbf{C}_k)}_{= \prod_{l=1,l\neq j}^{n} \sum_{y_l=1}^{c} p_{Y_l|\vec{X}_l}(y_l|\vec{x}_l;\mathbf{C}_k) \;=\; \prod_{l=1,l\neq j}^{n} 1 \;=\; 1} .$$

Expectation Maximization

- The probabilities $p_{Y_j|\vec{X}_j}(i|\vec{x}_j; \mathbf{C}_k)$ are computed as

$$p_{Y_j|\vec{X}_j}(i|\vec{x}_j; \mathbf{C}_k) = \frac{f_{\vec{X}_j,Y_j}(\vec{x}_j, i; \mathbf{C}_k)}{f_{\vec{X}_j}(\vec{x}_j; \mathbf{C}_k)} = \frac{f_{\vec{X}_j|Y_j}(\vec{x}_j|i; \mathbf{C}_k) \cdot p_{Y_j}(i; \mathbf{C}_k)}{\sum_{l=1}^{c} f_{\vec{X}_j|Y_j}(\vec{x}_j|l; \mathbf{C}_k) \cdot p_{Y_j}(l; \mathbf{C}_k)},$$

  that is, as the relative probability densities of the different clusters (as specified by the cluster parameters) at the location of the data points $\vec{x}_j$.

- The $p_{Y_j|\vec{X}_j}(i|\vec{x}_j; \mathbf{C}_k)$ are the posterior probabilities of the clusters given the data point $\vec{x}_j$ and a set of cluster parameters $\mathbf{C}_k$.

- They can be seen as **case weights** of a "completed" data set:
  - Split each data point $\vec{x}_j$ into c data points $(\vec{x}_j, i)$, $i = 1, \ldots, c$.
  - Distribute the unit weight of the data point $\vec{x}_j$ according to the above probabilities, i.e., assign to $(\vec{x}_j, i)$ the weight $p_{Y_j|\vec{X}_j}(i|\vec{x}_j; \mathbf{C}_k)$, $i = 1, \ldots, c$.

Expectation Maximization: Cookbook Recipe

Core Iteration Formula

$$\mathbf{C}_{k+1} = \underset{\mathbf{C}}{\operatorname{argmax}} \sum_{i=1}^{c} \sum_{j=1}^{n} p_{Y_j|\vec{X}_j}(i|\vec{x}_j; \mathbf{C}_k) \cdot \ln f_{\vec{X}_j, Y_j}(\vec{x}_j, i; \mathbf{C})$$

Expectation Step

- For all data points $\vec{x}_j$:
  Compute for each normal distribution the probability $p_{Y_j|\vec{X}_j}(i|\vec{x}_j; \mathbf{C}_k)$
  that the data point was generated from it
  (ratio of probability densities at the location of the data point).
  $\rightarrow$ "weight" of the data point for the estimation.

Maximization Step

- For all normal distributions:
  Estimate the parameters by standard maximum likelihood estimation
  using the probabilities ("weights") assigned to the data points
  w.r.t. the distribution in the expectation step.

Expectation Maximization: Mixture of Gaussians

Expectation Step: Use Bayes' rule to compute

$$p_{C|\vec{X}}(i|\vec{x};\mathbf{C}) = \frac{p_C(i;\mathbf{c}_i)\cdot f_{\vec{X}|C}(\vec{x}|i;\mathbf{c}_i)}{f_{\vec{X}}(\vec{x};\mathbf{C})} = \frac{p_C(i;\mathbf{c}_i)\cdot f_{\vec{X}|C}(\vec{x}|i;\mathbf{c}_i)}{\sum_{k=1}^{c} p_C(k;\mathbf{c}_k)\cdot f_{\vec{X}|C}(\vec{x}|k;\mathbf{c}_k)}.$$

$\rightarrow$ "weight" of the data point $\vec{x}$ for the estimation.

Maximization Step: Use maximum likelihood estimation to compute

$$\varrho_i^{(t+1)} = \frac{1}{n}\sum_{j=1}^{n} p_{C|\vec{X}_j}(i|\vec{x}_j;\mathbf{C}^{(t)}), \qquad \vec{\mu}_i^{(t+1)} = \frac{\sum_{j=1}^{n} p_{C|\vec{X}_j}(i|\vec{x}_j;\mathbf{C}^{(t)})\cdot\vec{x}_j}{\sum_{j=1}^{n} p_{C|\vec{X}_j}(i|\vec{x}_j;\mathbf{C}^{(t)})},$$

$$\text{and} \qquad \mathbf{\Sigma}_i^{(t+1)} = \frac{\sum_{j=1}^{n} p_{C|\vec{X}_j}(i|\vec{x}_j;\mathbf{C}^{(t)})\cdot\left(\vec{x}_j-\vec{\mu}_i^{(t+1)}\right)\left(\vec{x}_j-\vec{\mu}_i^{(t+1)}\right)^\top}{\sum_{j=1}^{n} p_{C|\vec{X}_j}(i|\vec{x}_j;\mathbf{C}^{(t)})}$$

Iterate until convergence (checked, e.g., by change of mean vector).

Expectation Maximization: Technical Problems

- If a fully general mixture of Gaussian distributions is used,
  the likelihood function is truly optimized if
  - all normal distributions except one are contracted to single data points and
  - the remaining normal distribution is the maximum likelihood estimate for the remaining data points.
- This undesired result is rare,
  because the algorithm gets stuck in a local optimum.
- Nevertheless it is recommended to take countermeasures,
  which consist mainly in reducing the degrees of freedom, like
  - Fix the determinants of the covariance matrices to equal values.
  - Use a diagonal instead of a general covariance matrix.
  - Use an isotropic variance instead of a covariance matrix.
  - Fix the prior probabilities of the clusters to equal values.

Soft Learning Vector Quantization

Idea: Use soft assignments instead of winner-takes-all (approach descibed here: [Seo and Obermayer 2003]).

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

$$u_{ij}^{\oplus} = \frac{\exp\left(-\frac{1}{2\sigma^2}(\vec{x}_j - \vec{r}_i^{\,(\text{old})})^\top (\vec{x}_j - \vec{r}_i^{\,(\text{old})})\right)}{\sum_{\vec{r} \in R(c_j)} \exp\left(-\frac{1}{2\sigma^2}(\vec{x}_j - \vec{r}^{\,(\text{old})})^\top (\vec{x}_j - \vec{r}^{\,(\text{old})})\right)} \quad \text{and}$$

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

$R(c)$ 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

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

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

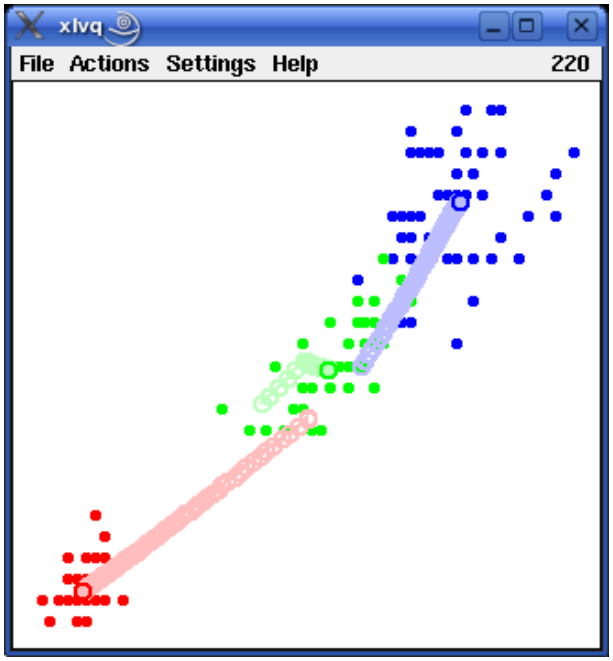
$\vec{r}_i$ is closest vector of different class

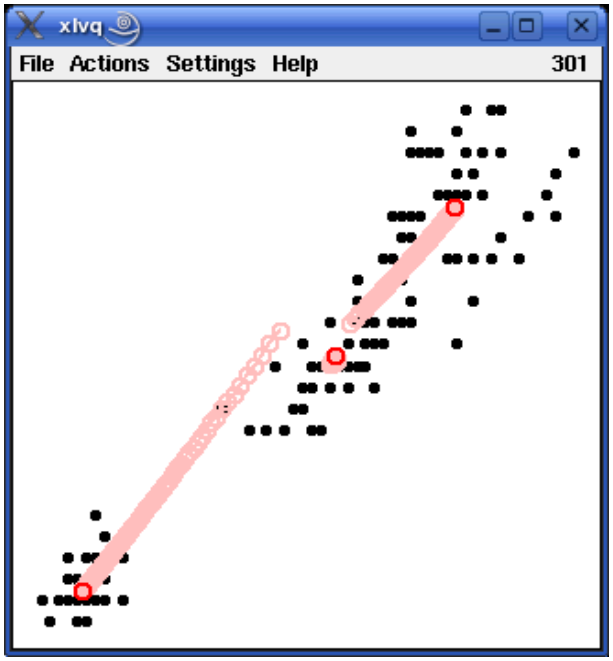
This update rule is stable without a *window rule* restricting the update.

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**, that is, a monotonically decreasing function

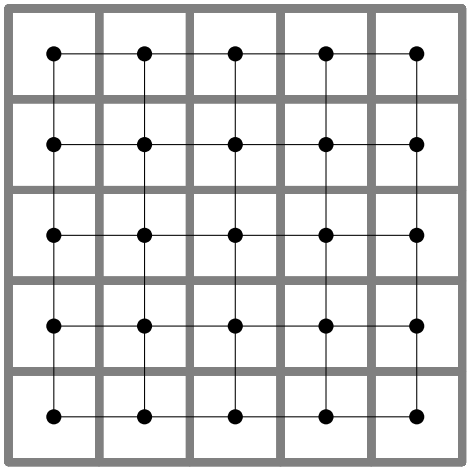
$$f : \mathbb{R}_0^+ \to [0, 1] \quad \text{with} \quad f(0) = 1 \quad \text{and} \quad \lim_{x \to \infty} f(x) = 0.$$

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:

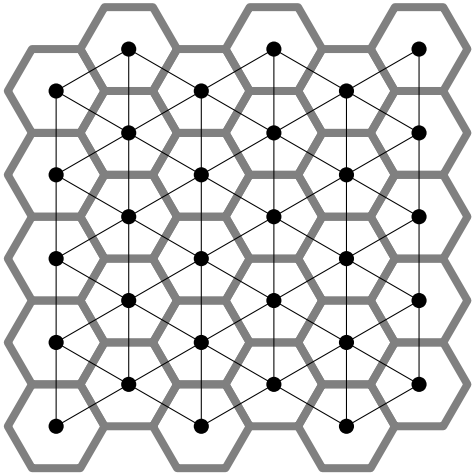
$$d_{\text{neurons}} : U_{\text{out}} \times U_{\text{out}} \to \mathbb{R}_0^+ .$$

Self-Organizing Maps: Neighborhood

Neighborhood of the output neurons: neurons form a grid



quadratic grid



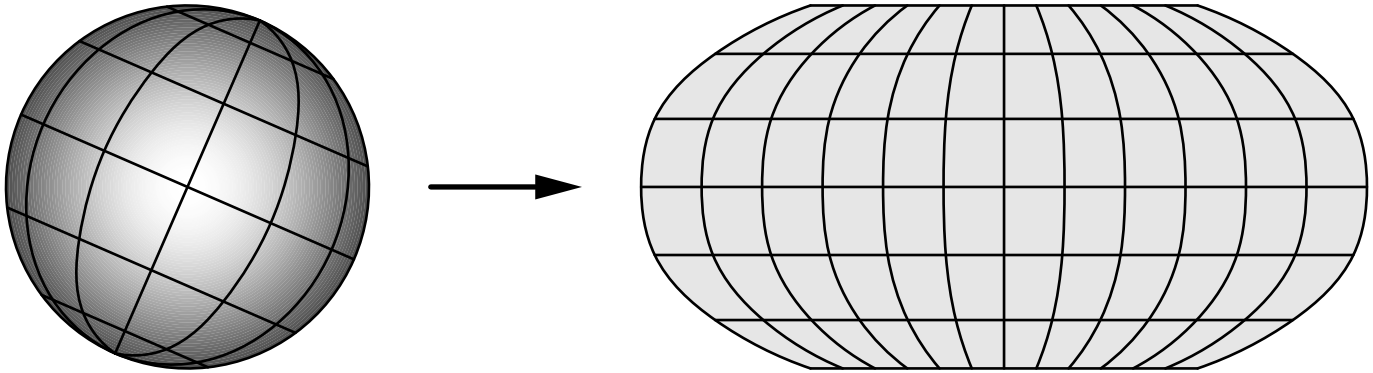
hexagonal grid

- Thin black lines: Indicate nearest neighbors of a neuron.
- Thick gray lines: Indicate regions assigned to a neuron for visualization.
- Usually two-dimensional grids are used to be able to draw the map easily.

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
(maps from 3 dimensions to 2 dimensions)



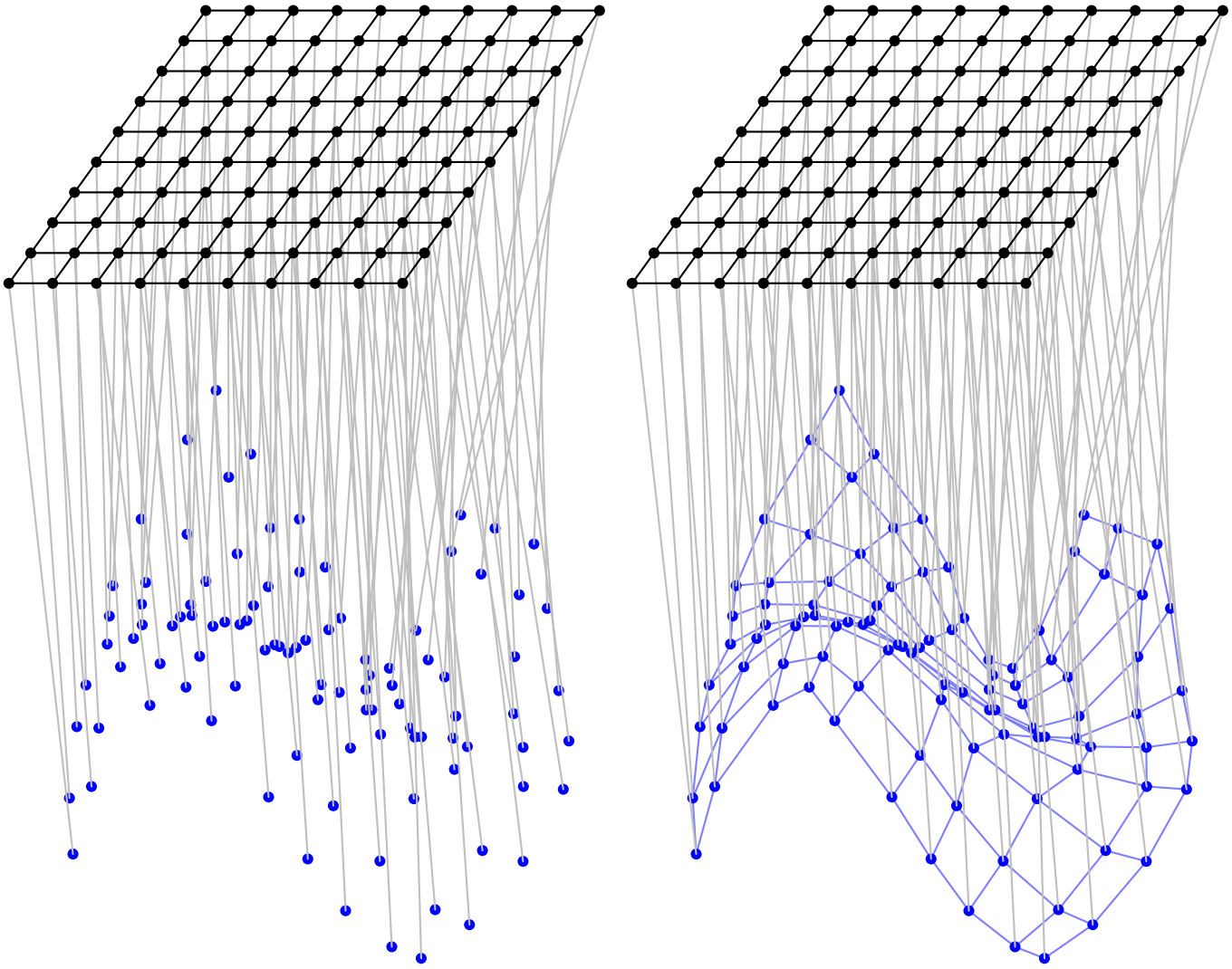
- Robinson projection is/was frequently used for world maps.
- The topology is preserved, although distances, angles, areas may be distorted.

Self-Organizing Maps: Topology Preserving Mapping

neuron space/grid
usually 2-dimensional
quadratic or
hexagonal grid

input/data space
usually high-dim.
(here: only 3-dim.)
blue: ref. vectors

Connections may
be drawn between
vectors corresponding
to adjacent neurons.



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 neighborhood function f_{nb} is a radial function.

Time dependent learning rate

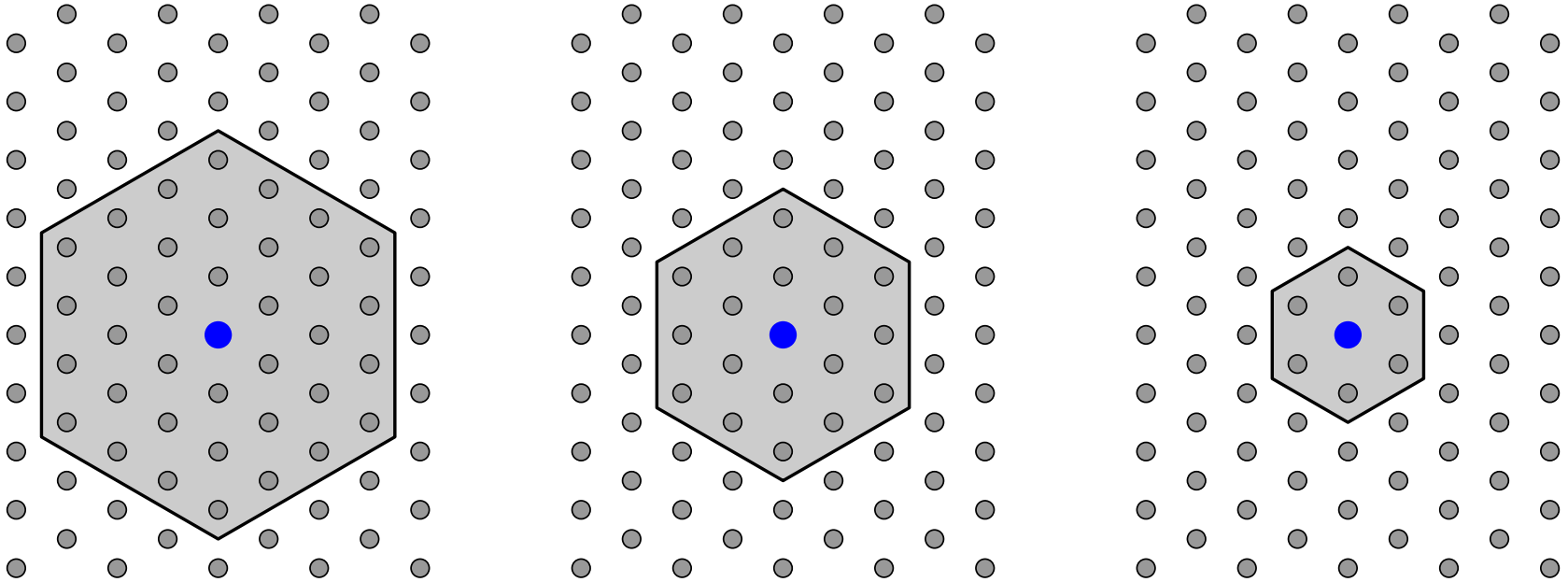
$$\eta(t) = \eta_0 \alpha_\eta^t, \quad 0 < \alpha_\eta < 1, \qquad \text{or} \qquad \eta(t) = \eta_0 t^{\kappa_\eta}, \quad \kappa_\eta < 0.$$

Time dependent neighborhood radius

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

Self-Organizing Maps: Neighborhood

The neighborhood size is reduced over time: (here: step function)



Note that a neighborhood function that is not a step function has a "soft" border and thus allows for a smooth reduction of the neighborhood size (larger changes of the reference vectors are restricted more and more to the close neighborhood).

Self-Organizing Maps: Training Procedure

- Initialize the weight vectors of the neurons of the self-organizing map, that is, place initial reference vectors in the input/data space.
- This may be done by randomly selecting training examples (provided there are fewer neurons that training examples, which is the usual case) or by sampling from some probability distribution on the data space.
- For the actual training, repeat the following steps:
  - Choose a training sample / data point (traverse the data points, possibly shuffling after each epoch).
  - Find the winner neuron with the distance function in the data space, that is, find the neuron with the closest reference vector.
  - Compute the time dependent radius and learning rate and adapt the corresponding neighbors of the winner neuron (severity of weight changes depend by neighborhood and learning rate).

Self-Organizing Maps: Examples

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

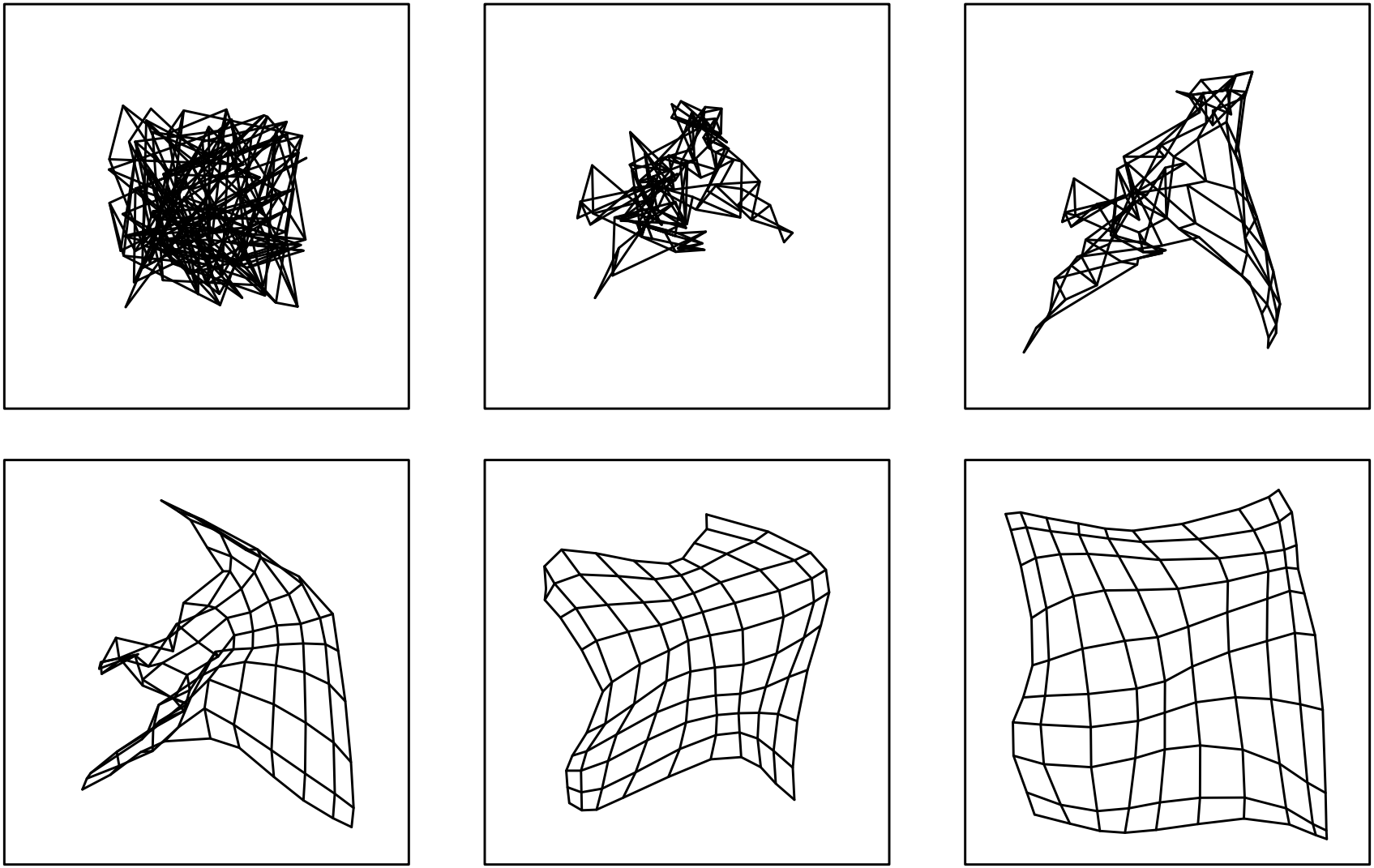
- Self-organizing map with 10×10 neurons (quadratic grid) that is trained with random points chosen uniformly from the square $[-1, 1] \times [-1, 1]$.
- Initialization with random reference vectors chosen uniformly from $[-0.5, 0.5] \times [-0.5, 0.5]$.
- Gaussian neighborhood function

$$f_{\text{nb}}(d_{\text{neurons}}(u, u_*), \varrho(t)) = \exp\left(-\frac{d^2_{\text{neurons}}(u,u_*)}{2\varrho(t)^2}\right).$$

- Time-dependent neighborhood radius $\varrho(t) = 2.5 \cdot t^{-0.1}$
- Time-dependent learning rate $\eta(t) = 0.6 \cdot t$.
- The next slides show the SOM state after 10, 20, 40, 80 and 160 training steps. In each training step one training sample is processed. Shading of the neuron grid shows neuron activations for $(-0.5, -0.5)$.

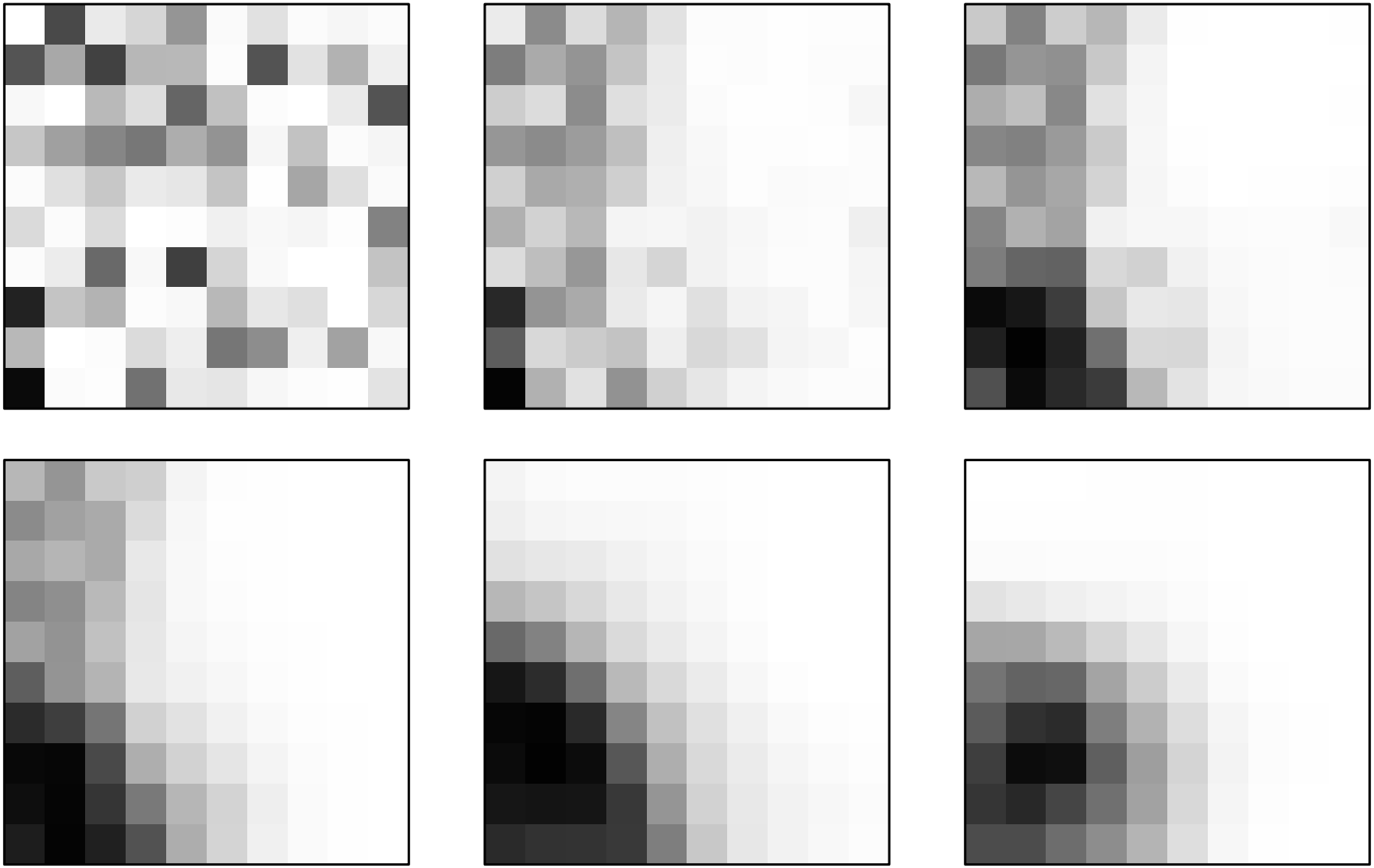
Self-Organizing Maps: Examples

Unfolding of a two-dimensional self-organizing map. (data space)



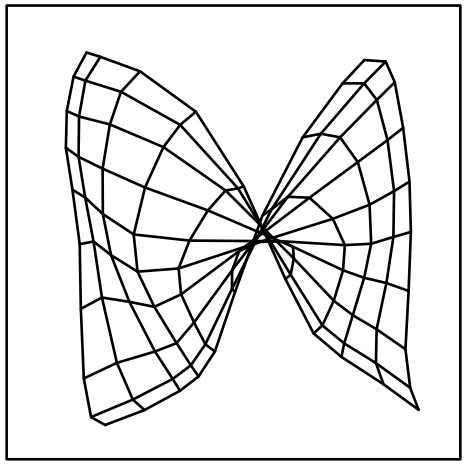
Self-Organizing Maps: Examples

Unfolding of a two-dimensional self-organizing map. (neuron grid)



Self-Organizing Maps: Examples

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

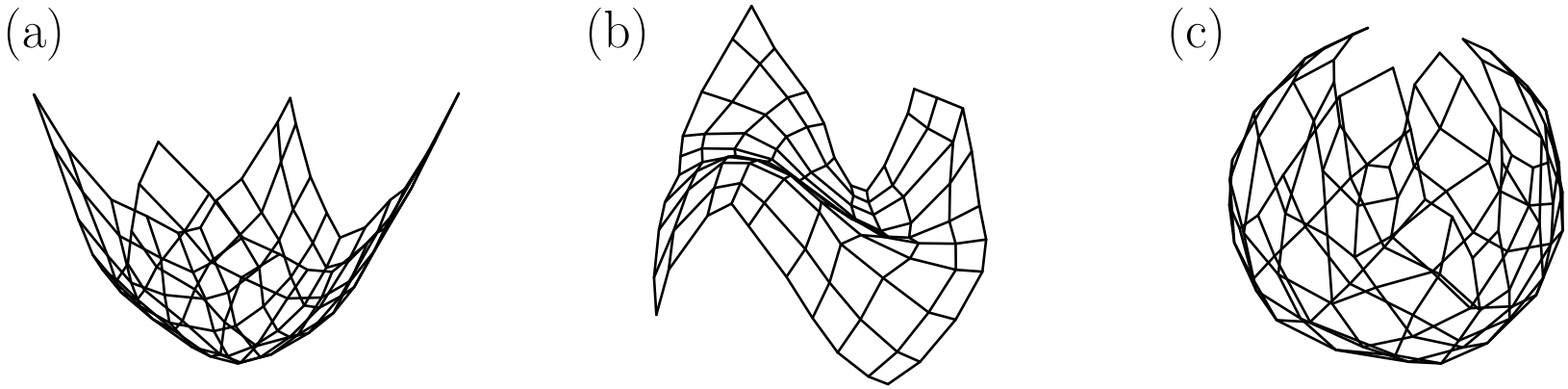


Training a self-organizing map may fail if

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

Self-Organizing Maps: Examples

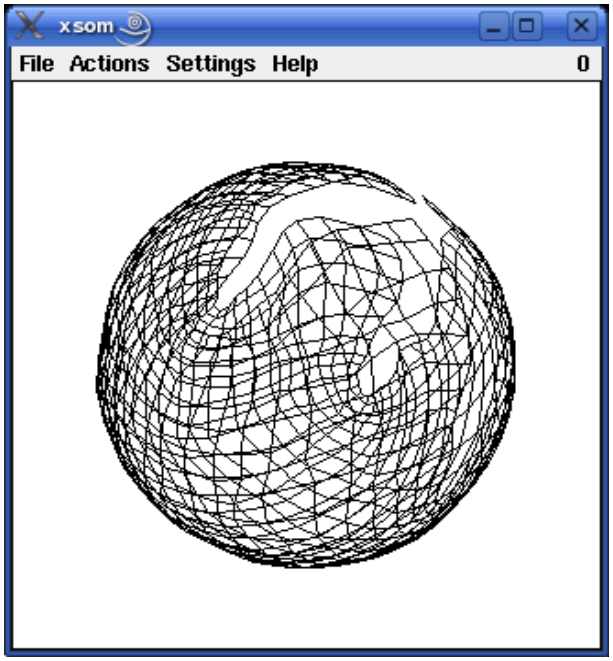
Example: Unfolding of a two-dimensional self-organizing map.

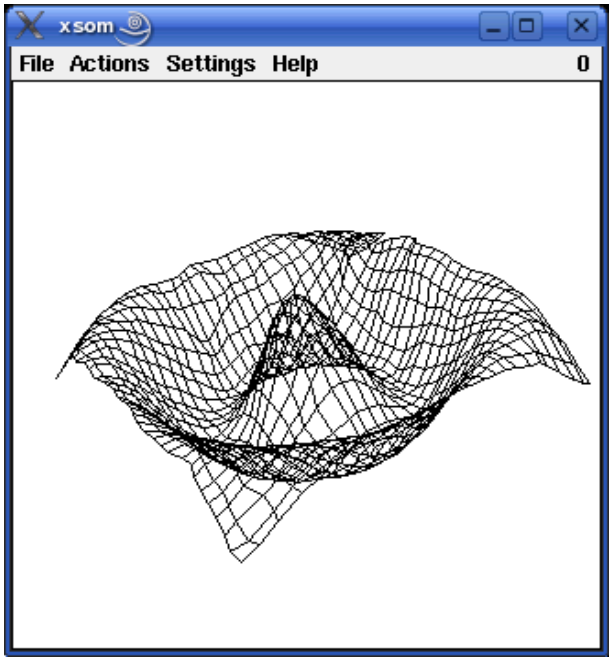


Self-organizing maps that have been trained with random points from
(a) a rotation parabola, (b) a simple cubic function, (c) the surface of a sphere.
Since the data points come from a two-dimensional subspace, training works well.

- In this case original space and image space have different dimensionality.
  (In the previous example they were both two-dimensional.)
- Self-organizing maps can be used for dimensionality reduction
  (in a quantized fashion, but interpolation may be used for smoothing).

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 and Boltzmann Machines

Hopfield Networks

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

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

(ii) $C = U \times U - \{(u, u) \mid u \in U\}$.

- 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, that is,

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

Hopfield Networks

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

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

The activation function of each neuron is a threshold function, that is,

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

The output function of each neuron is the identity, that is,

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

Alternative activation function

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

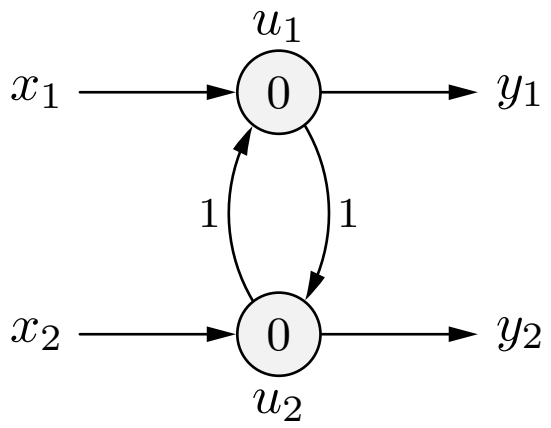
This activation function has advantages w.r.t. the physical interpretation of a Hopfield network.

General weight matrix of a Hopfield network

$$\mathbf{W} = \begin{pmatrix} 0 & w_{u_1u_2} & \ldots & w_{u_1u_n} \\ w_{u_1u_2} & 0 & \ldots & w_{u_2u_n} \\ \vdots & \vdots & & \vdots \\ w_{u_1u_n} & w_{u_1u_n} & \ldots & 0 \end{pmatrix}$$

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	$\mathbf{-1}$	$\mathbf{1}$
work phase	$\mathbf{1}$	$\mathbf{-1}$
	$\mathbf{-1}$	$\mathbf{1}$
	$\mathbf{1}$	$\mathbf{-1}$
	$\mathbf{-1}$	$\mathbf{1}$
	$\mathbf{1}$	$\mathbf{-1}$
	$\mathbf{-1}$	$\mathbf{1}$

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

Hopfield Networks: Examples

Sequential update of neuron activations

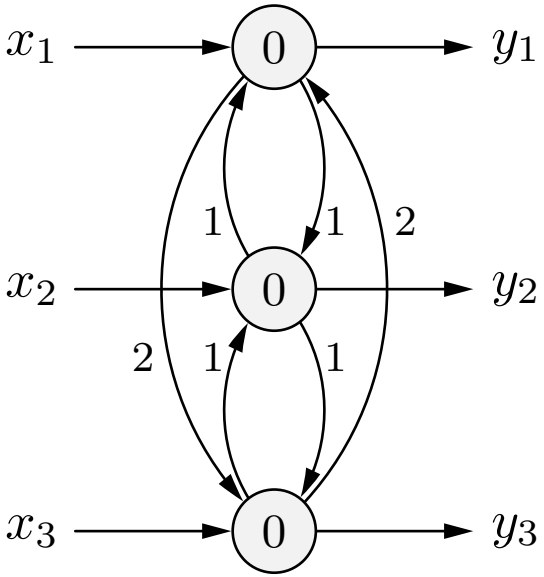
	u_1	u_2
input phase	**−1**	**1**
work phase	**1**	1
	1	**1**
	1	1
	1	**1**

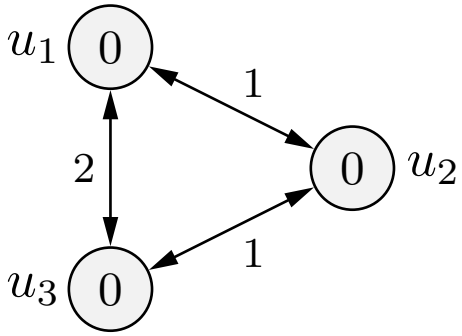
	u_1	u_2
input phase	**−1**	**1**
work phase	−1	**−1**
	−1	−1
	−1	**−1**
	−1	−1

- Update order u_1, u_2, u_1, u_2, ... (left) or u_2, u_1, u_2, u_1, ... (right)
- Regardless of the update order a stable state is reached.
- However, *which* state is reached depends on the update order.

Hopfield Networks: Examples

Simplified representation of a Hopfield network



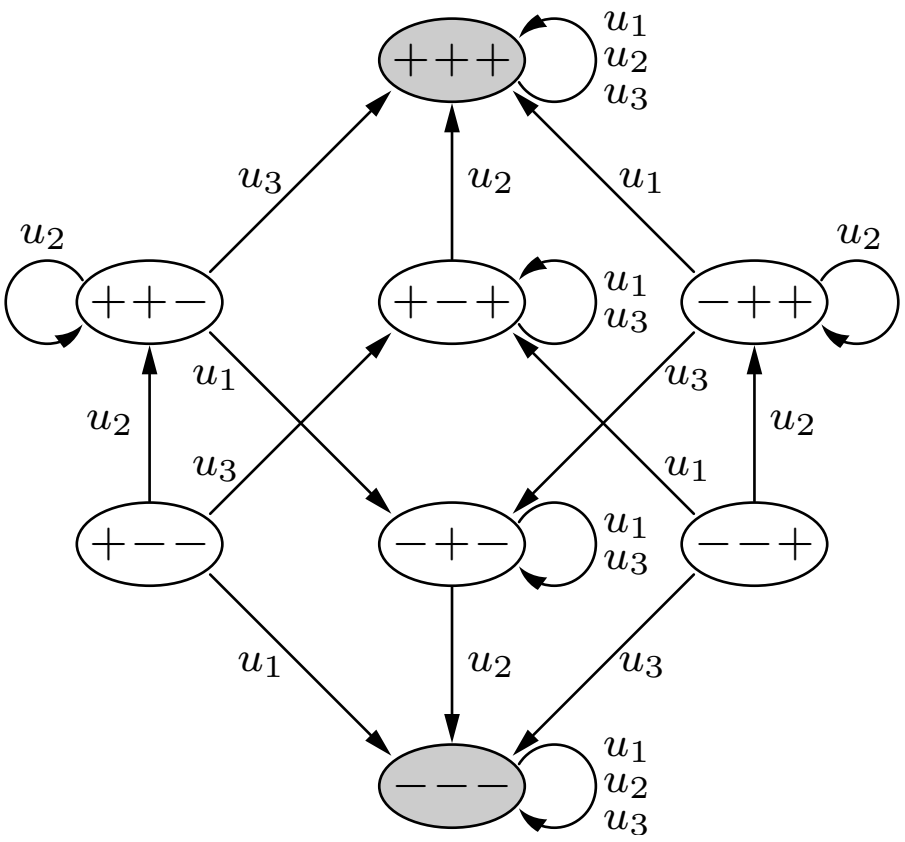


$$\mathbf{W} = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix}$$

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

Hopfield Networks: State Graph

Graph of activation states and transitions:
(for the Hopfield network shown on the preceding slide)



"+"/"−" encode the neuron activations:
"+" means +1 and "−" means −1.

Labels on arrows indicate the neurons, whose updates (activation changes) lead to the corresponding state transitions.

States shown in gray:
stable states, cannot be left again
States shown in white:
unstable states, may be left again.

Such a state graph captures all imaginable update orders.

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 $u_1, \ldots, u_n$ is defined as

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

Hopfield Networks: Convergence

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

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

- $\text{net}_u < \theta_u$: 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: $\Delta E \leq 0$.

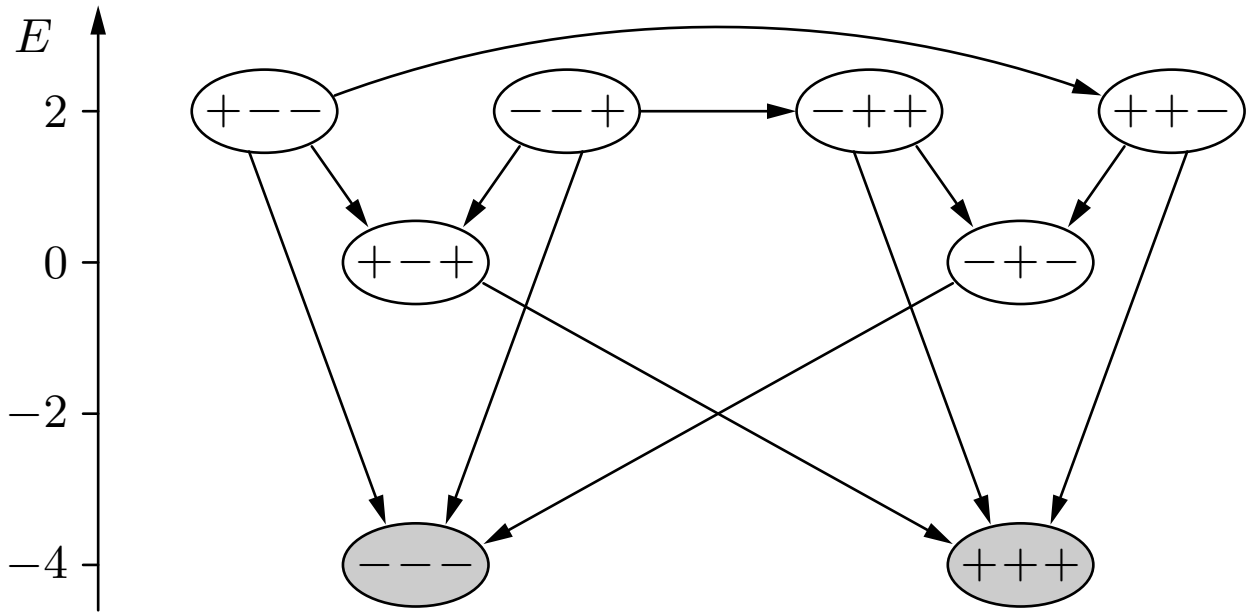
Hopfield Networks: Convergence

It takes at most $n \cdot 2^n$ update steps to reach convergence.

- Provided that the neurons are updated in an arbitrary, but fixed order, since this guarantees that the neurons are traversed cyclically, and therefore each neuron is updated every n steps.
- If in a traversal of all n neurons no activation changes: **a stable state has been reached**.
- If in a traversal of all n neurons at least one activation changes: **the previous state cannot be reached again**, because
  - either the new state has a smaller energy than the old (no way back: updates cannot increase the network energy)
  - or the number of +1 activations has increased (no way back: equal energy is possible only for $\text{net}_u \geq \theta_u$).
- The number of possible states of the Hopfield network is 2^n, at least one of which must be rendered unreachable in each traversal of the n neurons.

Hopfield Networks: Examples

Arrange states in state graph according to their energy

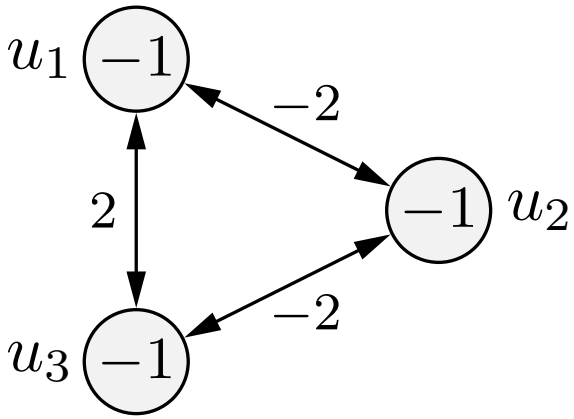


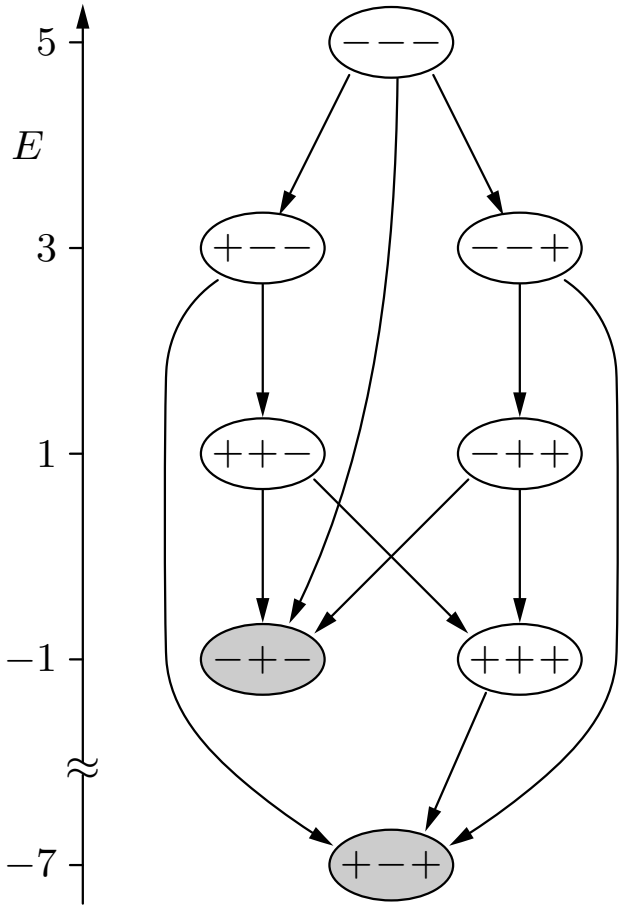
Energy function for example Hopfield network:

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

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

Idea: Use stable states to store patterns

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

Necessary and sufficient condition:

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

where

$$\begin{array}{rcl} S : \mathbb{R}^n & \to & \{-1, 1\}^n, \\ \vec{x} & \mapsto & \vec{y} \end{array}$$

with

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

Hopfield Networks: Associative Memory

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

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

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

Choose

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

where $\vec{x}\vec{x}^\top$ is the so-called **outer product** of $\vec{x}$ with itself.

With this matrix we have

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

$(*)$ holds, because vector/matrix multiplication is associative.

Hopfield Networks: Associative Memory

Hebbian learning rule [Hebb 1949]

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

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

- 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:

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

Storing several patterns

Choose

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

If the patterns are orthogonal, we have

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

and therefore

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

Storing several patterns

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

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.

With $\mathbf{W}\vec{x}_j = (n-m)\vec{x}_j$ 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),
since at most $m = n-1$ ortogonal patterns can be stored (so that $n-m > 0$).

Furthermore, the requirement that the patterns must be orthogonal
is a strong limitation of the usefulness of this result.

Hopfield Networks: Associative Memory

Non-orthogonal patterns:

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

- The “disturbance term” need not make it impossible to store the patterns.
- The states corresponding to the patterns $\vec{x}_j$ may still be stable, if the “disturbance term” is sufficiently small.
- For this term to be sufficiently small, the patterns must be “almost” orthogonal.
- The larger the number of patterns to be stored (that is, the smaller $n - m$), the smaller the “disturbance term” must be.
- The theoretically possible maximal capacity of a Hopfield network (that is, $m = n - 1$) is hardly ever reached in practice.

Associative Memory: Example

Example: Store patterns $\vec{x}_1 = (+1, +1, -1, -1)^\top$ and $\vec{x}_2 = (-1, +1, -1, +1)^\top$.

$$\mathbf{W} = \mathbf{W}_1 + \mathbf{W}_2 = \vec{x}_1 \vec{x}_1^\top + \vec{x}_2 \vec{x}_2^\top - 2\mathbf{E}$$

where

$$\mathbf{W}_1 = \begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{pmatrix}, \qquad \mathbf{W}_2 = \begin{pmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{pmatrix}.$$

The full weight matrix is:

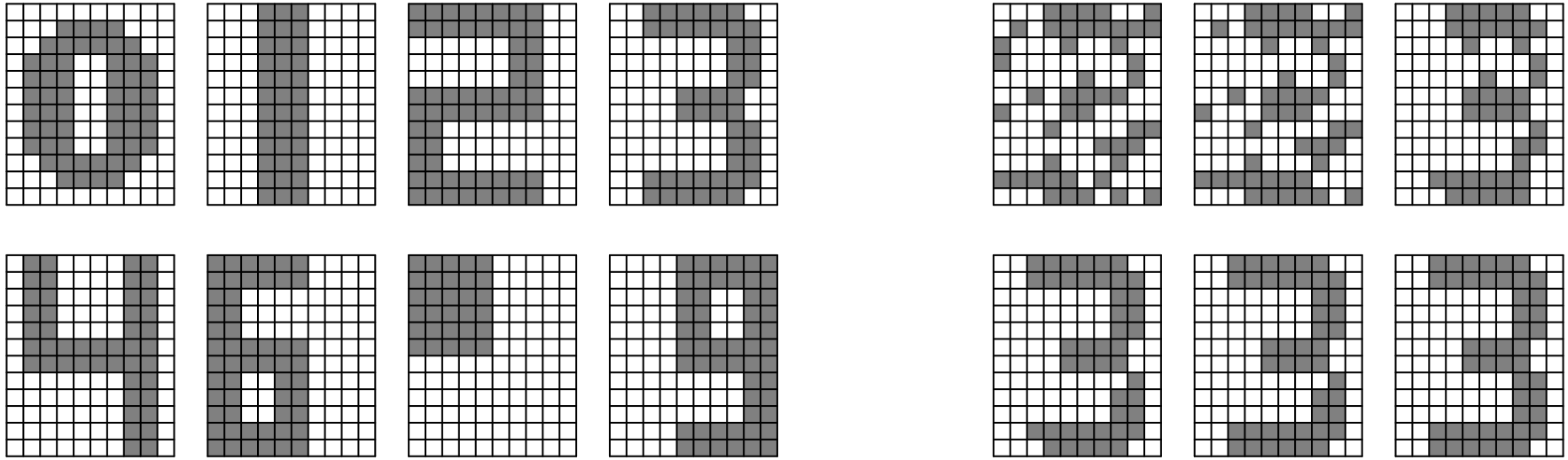
$$\mathbf{W} = \begin{pmatrix} 0 & 0 & 0 & -2 \\ 0 & 0 & -2 & 0 \\ 0 & -2 & 0 & 0 \\ -2 & 0 & 0 & 0 \end{pmatrix}.$$

Therefore it is

$$\mathbf{W}\vec{x}_1 = (+2, +2, -2, -2)^\top \qquad \text{and} \qquad \mathbf{W}\vec{x}_1 = (-2, +2, -2, +2)^\top.$$

Associative Memory: Examples

Example: Storing bit maps of numbers



- Left: Bit maps stored in a Hopfield network.
- Right: Reconstruction of a pattern from a random input.

Training a Hopfield network with the Delta rule

Necessary condition for pattern $\vec{x}$ being a stable state:

$$\begin{array}{llllll}
s(0 & + w_{u_1u_2}\,\mathrm{act}_{u_2}^{(p)} & + \ldots + w_{u_1u_n}\,\mathrm{act}_{u_n}^{(p)} & - \theta_{u_1}) = \mathrm{act}_{u_1}^{(p)}, \\
s(w_{u_2u_1}\,\mathrm{act}_{u_1}^{(p)} & + 0 & + \ldots + w_{u_2u_n}\,\mathrm{act}_{u_n}^{(p)} & - \theta_{u_2}) = \mathrm{act}_{u_2}^{(p)}, \\
\vdots & \vdots & \vdots & \vdots \quad \vdots \\
s(w_{u_nu_1}\,\mathrm{act}_{u_1}^{(p)} & + w_{u_nu_2}\,\mathrm{act}_{u_2}^{(p)} & + \ldots + 0 & - \theta_{u_n}) = \mathrm{act}_{u_n}^{(p)}.
\end{array}$$

with the standard threshold function

$$s(x) = \begin{cases} 1, & \text{if} \quad x \geq 0, \\ -1, & \text{otherwise.} \end{cases}$$

Training a Hopfield network with the Delta rule

Turn weight matrix into a weight vector:

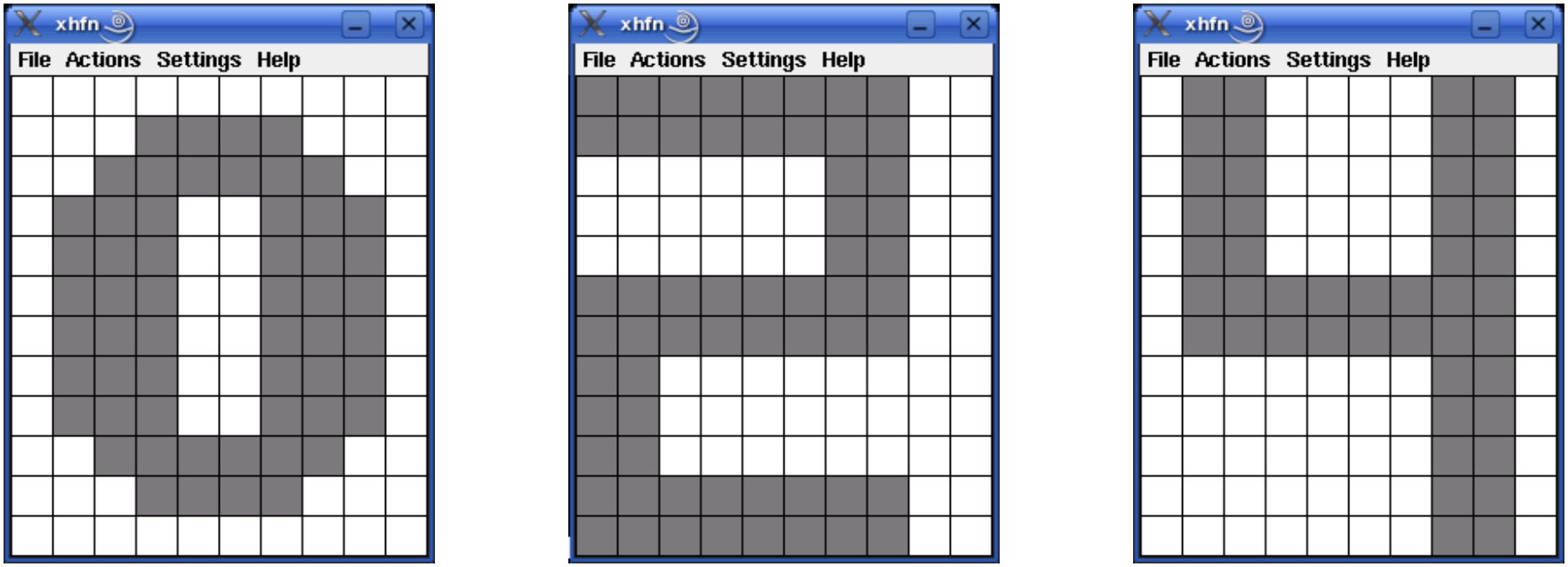
$$\begin{aligned}\vec{w} = (\quad & w_{u_1u_2}, & w_{u_1u_3}, & \ldots, & w_{u_1u_n}, \\ & & w_{u_2u_3}, & \ldots, & w_{u_2u_n}, \\ & & & \ddots & \vdots \\ & & & & w_{u_{n-1}u_n}, \\ & -\theta_{u_1}, & -\theta_{u_2}, & \ldots, & -\theta_{u_n} \qquad).\end{aligned}$$

Construct input vectors for a threshold logic unit

$$\vec{z}_2 = (\mathrm{act}_{u_1}^{(p)}, \underbrace{0, \ldots, 0,}_{n-2 \text{ zeros}} \mathrm{act}_{u_3}^{(p)}, \ldots, \mathrm{act}_{u_n}^{(p)}, \ldots 0, 1, \underbrace{0, \ldots, 0}_{n-2 \text{ zeros}}).$$

Apply Delta rule training / Widrow–Hoff procedure until convergence.

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

$$\begin{aligned} w^0_{uv} &= 2w^-_{uv} && \text{and} \\ \theta^0_u &= \theta^-_u + \sum_{v \in U - \{u\}} w^-_{uv} \end{aligned}$$

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

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

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

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

$i = 1, 2$, 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 $\theta_u = a\theta_u^{(1)} + b\theta_u^{(2)}$.

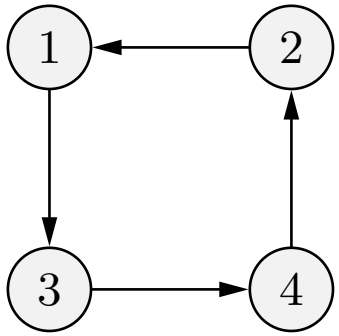
Proof: Just do the computations.

Idea: Additional conditions can be formalized separately and incorporated later.
(One energy function per condition, then apply combination lemma.)

Hopfield Networks: Solving Optimization Problems

Example: Traveling salesman problem

Idea: Represent tour by a matrix.



$$
\begin{array}{c}
\text{city} \\
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\left(\begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0
\end{array} \right)
\end{array}
\begin{array}{l}
\\
1. \\
2. \\
3. \\
4.
\end{array}
\quad \text{step}
$$

An element m_{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.

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:

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

where

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

Symmetric version of the energy function:

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

Hopfield Networks: Solving Optimization Problems

Additional conditions that have to be satisfied:

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

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

  that is, each column of the matrix contains exactly one 1.

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

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

  that is, 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:

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

Hopfield Networks: Solving Optimization Problems

Resulting energy function:

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

Second additional condition is handled in a completely analogous way:

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

Combining the energy functions:

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

Hopfield Networks: Solving Optimization Problems

From the resulting energy function we can read the weights

$$w_{(i_1,j_1)(i_2,j_2)} = \underbrace{-ad_{j_1j_2} \cdot (\delta_{(i_1 \bmod n)+1,i_2} + \delta_{i_1,(i_2 \bmod n)+1})}_{\text{from } E_1} \underbrace{-2b\delta_{j_1j_2}}_{\text{from } E_2} \underbrace{-2c\delta_{i_1i_2}}_{\text{from } E_3}$$

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, let alone an optimal one.

Hopfield Networks: Reasons for Failure

Hopfield network only rarely finds a tour, let alone an optimal one.

- One of the main problems is that the Hopfield network is unable to switch from a found tour to another with a lower total length.
- The reason is that transforming a matrix that represents a tour into another matrix that represents a different tour requires that at least four neurons (matrix elements) change their activations.
- However, each of these changes, if carried out individually, violates at least one of the constraints and thus increases the energy.
- Only all four changes together can result in a smaller energy, but cannot be executed together due to the asynchronous update.
- Therefore the normal activation updates can never change an already found tour into another, even if this requires only a marginal change of the tour.

Hopfield Networks: Local Optima

- Results can be somewhat improved if instead of
  discrete Hopfield networks (activations in $\{-1, 1\}$ (or $\{0, 1\}$)) one uses
  continuous Hopfield networks (activations in $[-1, 1]$ (or $[0, 1]$)).
  However, the fundamental problem is not solved in this way.

- More generally, the reason for the difficulties that are encountered
  if an optimization problem is to be solved with a Hopfield network is:

  The update procedure may get stuck in a local optimum.

- The problem of local optima occurs also with many other optimization methods,
  for example, gradient descent, hill climbing, alternating optimization etc.

- Ideas to overcome this difficulty for other optimization methods
  may be transferred to Hopfield networks.

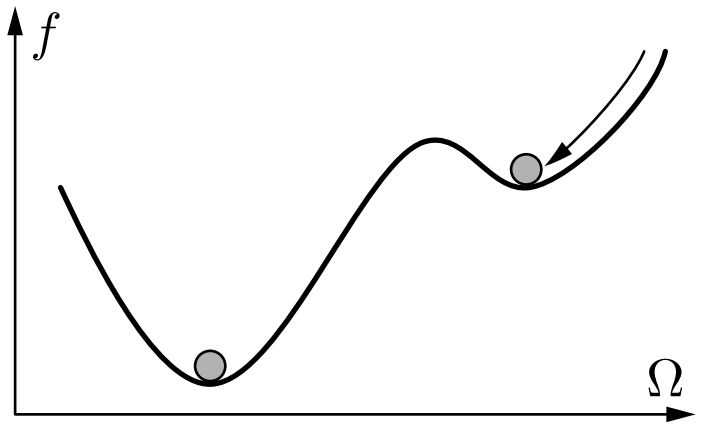
- One such method, which is very popular, is **simulated annealing**.

Simulated Annealing

May be seen as an extension of random or gradient descent that tries to avoid getting stuck.

Idea: transitions from higher to lower (local) minima should be more probable than *vice versa*.

[Metropolis *et al.* 1953; Kirkpatrik *et al.* 1983]



Principle of Simulated Annealing:

- Random variants of the current solution (candidate) are created.
- Better solution (candidates) are always accepted.
- Worse solution (candidates) are accepted with a probability that depends on
  - the quality difference between the new and the old solution (candidate) and
  - a temperature parameter that is decreased with time.

Simulated Annealing

- **Motivation:**
  - Physical minimization of the energy (more precisely: atom lattice energy) if a heated piece of metal is cooled slowly.
  - This process is called **annealing**.
  - It serves the purpose to make the metal easier to work or to machine by relieving tensions and correcting lattice malformations.
- **Alternative Motivation:**
  - A ball rolls around on an (irregularly) curved surface; minimization of the potential energy of the ball.
  - In the beginning the ball is endowed with a certain kinetic energy, which enables it to roll up some slopes of the surface.
  - In the course of time, friction reduces the kinetic energy of the ball, so that it finally comes to a rest in a valley of the surface.
- Attention: **There is no guarantee that the global optimum is found!**

Simulated Annealing: Procedure

1. Choose a (random) starting point $s_0 \in \Omega$ (Ω is the search space).
2. Choose a point $s' \in \Omega$ "in the vicinity" of s_i
   (for example, by a small random variation of s_i).
3. Set
$$s_{i+1} = \begin{cases} s' & \text{if } f(s') \geq f(s_i), \\ s' & \text{with probability } p = e^{-\frac{\Delta f}{kT}} \text{ and} \\ s_i & \text{with probability } 1 - p \text{ otherwise.} \end{cases}$$
$\Delta f = f(s_i) - f(s')$	quality difference of the solution (candidates)
$k = \Delta f_{\max}$	(estimation of the) range of quality values
T	temperature parameter (is (slowly) decreased over time)
4. Repeat steps 2 and 3 until some termination criterion is fulfilled.

- For (very) small T the method approaches a pure random descent.

Hopfield Networks: Simulated Annealing

Applying simulated annealing to Hopfield networks is very simple:

- All neuron activations are initialized randomly.
- The neurons of the Hopfield network are traversed repeatedly (for example, in some random order).
- For each neuron, it is determined whether an activation change leads to a reduction of the network energy or not.
- An activation change that reduces the network energy is always accepted (in the normal update process, only such changes occur).
- However, if an activation change increases the network energy, it is accepted with a certain probability (see preceding slide).
- Note that in this case we have simply

$$\Delta f = \Delta E = |\operatorname{net}_u - \theta_u|$$

Hopfield Networks: Summary

- Hopfield networks are **restricted recurrent neural networks** (full pairwise connections, symmetric connection weights).
- Synchronous update of the neurons may lead to oscillations, but **asynchronous update is guaranteed to reach a stable state** (asynchronous updates either reduce the energy of the network or increase the number of +1 activations).
- Hopfield networks can be used as **associative memory**, that is, as memory that is addressed by its contents, by using the stable states to store desired patterns.
- Hopfield networks can be used to **solve optimization problems**, if the function to optimize can be reformulated as an energy function (stable states are (local) minima of the energy function).
- Approaches like **simulated annealing** may be needed to prevent that the update gets stuck in a local optimum.

Boltzmann Machines

- Boltzmann machines are closely related to Hopfield networks.
- They differ from Hopfield networks mainly in how the neurons are updated.
- They also rely on the fact that one can define an **energy function** that assigns a numeric value (an *energy*) to each state of the network.
- With the help of this energy function a probability distribution over the states of the network is defined based on the **Boltzmann distribution** (also known as **Gibbs distribution**) of statistical mechanics, namely

$$P(\vec{s}) = \frac{1}{c}\, e^{-\frac{E(\vec{s})}{kT}}.$$

$\vec{s}$	describes the (discrete) state of the system,
c	is a normalization constant,
E	is the function that yields the energy of a state $\vec{s}$,
T	is the thermodynamic temperature of the system,
k	is Boltzmann's constant ($k \approx 1.38 \cdot 10^{-23}\, J/K$).

Boltzmann Machines

- For Boltzmann machines the product kT is often replaced by merely T, combining the temperature and Boltzmann's constant into a single parameter.
- The state $\vec{s}$ consists of the vector $\vec{\text{act}}$ of the neuron activations.
- The energy function of a Boltzmann machine is

$$E(\vec{\text{act}}) = -\frac{1}{2}\,\vec{\text{act}}^{\top}\mathbf{W}\vec{\text{act}} + \vec{\theta}^{\top}\vec{\text{act}},$$

  where $\mathbf{W}$: matrix of connection weights; $\vec{\theta}$: vector of threshold values.
- Consider the energy change resulting from the change of a single neuron u:

$$\Delta E_u = E_{\text{act}_u=1} - E_{\text{act}_u=0} = \sum_{v \in U-\{u\}} w_{uv}\,\text{act}_v - \theta_u$$

- Writing the energies in terms of the Boltzmann distribution yields

$$\Delta E_u = -kT\ln(P(\text{act}_u = 1) - (-kT\ln(P(\text{act}_u = 0)).$$

Boltzmann Machines

- Rewrite as
$$\begin{aligned}\frac{\Delta E_u}{kT} &= \ln(P(\text{act}_u = 1)) - \ln(P(\text{act}_u = 0)) \\ &= \ln(P(\text{act}_u = 1)) - \ln(1 - P(\text{act}_u = 1))\end{aligned}$$
(since obviously $P(\text{act}_u = 0) + P(\text{act}_u = 1) = 1$).

- Solving this equation for $P(\text{act}_u = 1)$ finally yields
$$P(\text{act}_u = 1) = \frac{1}{1 + e^{-\frac{\Delta E_u}{kT}}}.$$

- That is: the probability of a neuron being active is a logistic function of the (scaled) energy difference between its active and inactive state.

- Since the energy difference is closely related to the network input, namely as
$$\Delta E_u = \sum_{v \in U - \{u\}} w_{uv}\, \text{act}_v - \theta_u = \text{net}_u - \theta_u,$$
this formula suggests a stochastic update procedure for the network.

Boltzmann Machines: Update Procedure

- A neuron u is chosen (randomly), its network input, from it the energy difference ΔE_u and finally the probability of the neuron having activation 1 is computed. The neuron is set to activation 1 with this probability and to 0 otherwise.
- This update is repeated many times for randomly chosen neurons.
- Simulated annealing is carried out by slowly lowering the temperature T.
- This update process is a **Markov Chain Monte Carlo** (**MCMC**) procedure.
- After sufficiently many steps, the probability that the network is in a specific activation state depends only on the energy of that state.
  It is independent of the initial activation state the process was started with.
- This final situation is also referred to as **thermal equlibrium**.
- Therefore: Boltzmann machines are representations of and sampling mechanisms for the Boltzmann distributions defined by their weights and threshold values.

Boltzmann Machines: Training

- **Idea of Training:**
  Develop a training procedure with which the probability distribution represented by a Boltzmann machine via its energy function can be adapted to a given sample of data points, in order to obtain a **probabilistic model of the data**.
- This objective can only be achieved sufficiently well
  if the data points are actually a sample from a Boltzmann distribution.
  (Otherwise the model cannot, in principle, be made to fit the sample data well.)
- In order to mitigate this restriction to Boltzmann distributions,
  a deviation from the structure of Hopfield networks is introduced.
- The neurons are divided into
  - **visible neurons**, which receive the data points as input, and
  - **hidden neurons**, which are not fixed by the data points.

  (Reminder: Hopfield networks have only visible neurons.)

Boltzmann Machines: Training

- **Objective of Training:**
  Adapt the connection weights and threshold values in such a way that the true distribution underlying a given data sample is approximated well by the probability distribution represented by the Boltzmann machine on its visible neurons.

- **Natural Approach to Training:**
  - Choose a measure for the difference between two probability distributions.
  - Carry out a gradient descent in order to minimize this difference measure.

- Well-known measure: **Kullback–Leibler information divergence**.
  For two probability distributions p_1 and p_2 defined over the same sample space Ω:

$$KL(p_1, p_2) = \sum_{\omega \in \Omega} p_1(\omega) \ln \frac{p_1(\omega)}{p_2(\omega)}.$$

  Applied to Boltzmann machines: p_1 refers to the data sample,
  p_2 to the visible neurons of the Boltzmann machine.

Boltzmann Machines: Training

- In each training step the Boltzmann machine is ran twice (two "phases").
- **"Positive Phase"**: Visible neurons are fixed to a randomly chosen data point; only the hidden neurons are updated until thermal equilibrium is reached.
- **"Negative Phase"**: All units are updated until thermal equilibrium is reached.
- In the two phases statistics about individual neurons and pairs of neurons (both visible and hidden) being activated (simultaneously) are collected.
- Then update is performed according to the following two equations:

$$\Delta w_{uv} = \frac{1}{\eta}(p_{uv}^{+} - p_{uv}^{-}) \qquad \text{and} \qquad \Delta\theta_u = -\frac{1}{\eta}(p_u^{+} - p_u^{-}).$$

p_u	probability that neuron u is active,
p_{uv}	probability that neurons u and v are both active simultaneously,
$+\ -$	as upper indices indicate the phase referred to.

(All probabilities are estimated from observed relative frequencies.)

Boltzmann Machines: Training

- Intuitive explanation of the update rule:
  - If a neuron is more often active when a data sample is presented
    than when the network is allowed to run freely,
    the probability of the neuron being active is too low,
    so the threshold should be reduced.
  - If neurons are more often active together when a data sample is presented
    than when the network is allowed to run freely,
    the connection weight between them should be increased,
    so that they become more likely to be active together.
- This training method is very similar to the **Hebbian learning rule**.

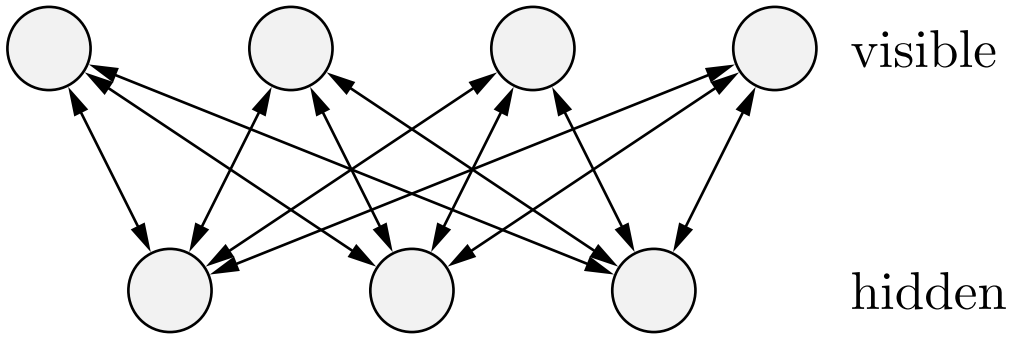
  Derived from a biological analogy it says:

  connections between two neurons that are synchronously active are strengthened

  ("cells that fire together, wire together").

Boltzmann Machines: Training

- Unfortunately, this procedure is impractical unless the networks are very small.
- The main reason is the fact that the larger the network, the more update steps need to be carried out in order to obtain sufficiently reliable statistics for the neuron activation (pairs) needed in the update formulas.
- Efficient training is possible for the **restricted Boltzmann machine**.
- Restriction consists in using a bipartite graph instead of a fully connected graph:
  - Vertices are split into two groups, the visible and the hidden neurons;
  - Connections only exist between neurons from different groups.



Restricted Boltzmann Machines

A **restricted Boltzmann Machine (RBM)** or **Harmonium** is a neural network with a graph $G = (U, C)$ that satisfies the following conditions:

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

(ii) $C = U_{\text{in}} \times U_{\text{hidden}}.$

- In a restricted Boltzmann machine, all input neurons are also output neurons and *vice versa* (all output neurons are also input neurons).
- There are hidden neurons, which are different from input and output neurons.
- Each input/output neuron receives input from all hidden neurons; each hidden neuron receives input from all input/output neurons.

The connection weights between input and hidden neurons are symmetric, that is,

$$\forall u \in U_{\text{in}}, v \in U_{\text{hidden}} : \qquad w_{uv} = w_{vu}.$$

Restricted Boltzmann Machines: Training

Due to the lack of connections within the visible units and within the hidden units, training can proceed by repeating the following three steps:

- Visible units are fixed to a randomly chosen data sample $\vec{x}$;
  hidden units are updated once and in parallel (result: $\vec{y}$).

  $\vec{x}\vec{y}^\top$ is called the **positive gradient** for the weight matrix.

- Hidden neurons are fixed to the computed vector $\vec{y}$;
  visible units are updated once and in parallel (result: $\vec{x}^*$).

  Visible neurons are fixed to "reconstruction" $\vec{x}^*$;
  hidden neurons are update once more (result: $\vec{y}^*$).

  $\vec{x}^*\vec{y}^{*\top}$ is called the **negative gradient** for the weight matrix.

- Connection weights are updated with difference of positive and negative gradient:

$$\Delta w_{uv} = \eta(\vec{x}_u\vec{y}_v^\top - \vec{x}_u^*\vec{y}_v^{*\top}) \qquad \text{where } \eta \text{ is a learning rate.}$$

Restriced Boltzmann Machines: Training and Deep Learning

- Many **improvements of this basic procedure** exist [Hinton 2010]:
  - use a momentum term for the training,
  - use actual probabilities instead of binary reconstructions,
  - use online-like training based on small batches etc.
- Restricted Boltzmann machines have also been used to build **deep networks** in a fashion similar to stacked auto-encoders for multi-layer perceptrons.
- Idea: train a restricted Boltzmann machine, then create a data set of hidden neuron activations by sampling from the trained Boltzmann machine, and build another restricted Boltzmann machine from the obtained data set.
- This procedure can be repeated several times and the resulting Boltzmann machines can then easily be stacked.
- The obtained stack is fine-tuned with a procedure similar to back-propagation [Hinton *et al.* 2006].

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**:

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

Exact analytical solution:

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

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

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

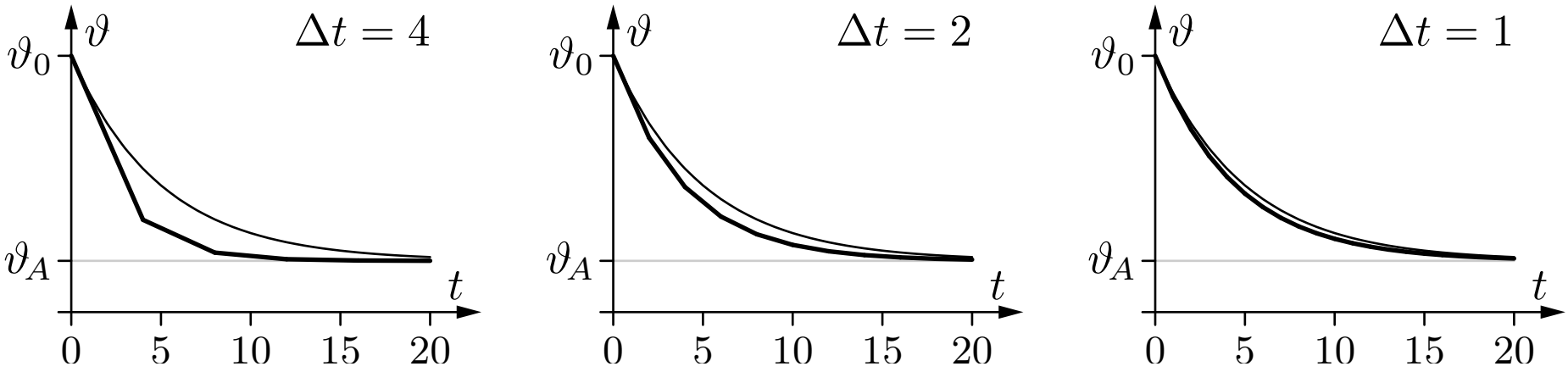
$$\vartheta_2 = \vartheta(t_2) = \vartheta(t_1) + \dot{\vartheta}(t_1)\Delta t = \vartheta_1 - k(\vartheta_1 - \vartheta_A)\Delta t.$$

General recursive formula:

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

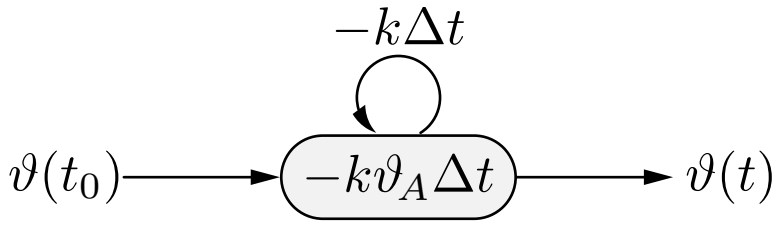
Recurrent Networks: Cooling Law

Euler–Cauchy polygonal 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**

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

with sufficiently small Δt. Then it is

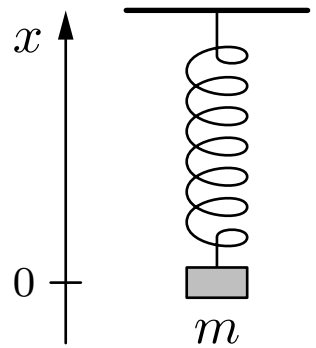
$$\vartheta(t+\Delta t) - \vartheta(t) = \Delta\vartheta(t) \approx -k(\vartheta(t) - \vartheta_A)\Delta t,$$

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

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:

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

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

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

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:

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

Approximate differential quotient by forward difference:

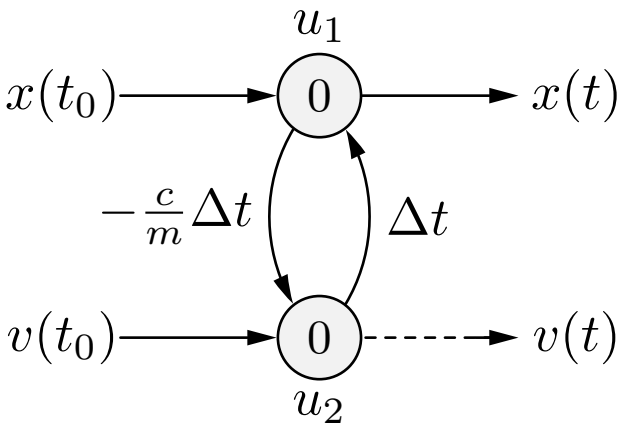
$$\frac{\Delta x}{\Delta t} = \frac{x(t+\Delta t) - x(t)}{\Delta t} = v \qquad \text{and} \qquad \frac{\Delta v}{\Delta t} = \frac{v(t+\Delta t) - v(t)}{\Delta t} = -\frac{c}{m}x$$

Resulting recursive equations:

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

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

Recurrent Networks: Mass on a Spring



Neuron u_1:

$$f_{\text{net}}^{(u_1)}(v, w_{u_1u_2}) = w_{u_1u_2}v = -\frac{c}{m}\Delta t\ v \qquad \text{and}$$

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

Neuron u_2:

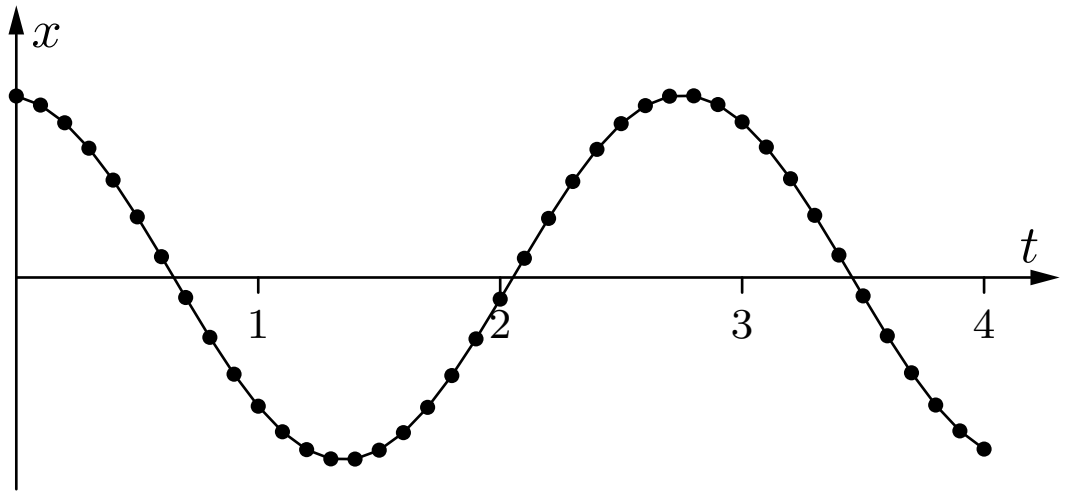
$$f_{\text{net}}^{(u_2)}(x, w_{u_2u_1}) = w_{u_2u_1}x = \Delta t\ x \qquad \text{and}$$

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

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:

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

Introduce $n-1$ intermediary quantities

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

to obtain the system

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

of n coupled first order differential equations.

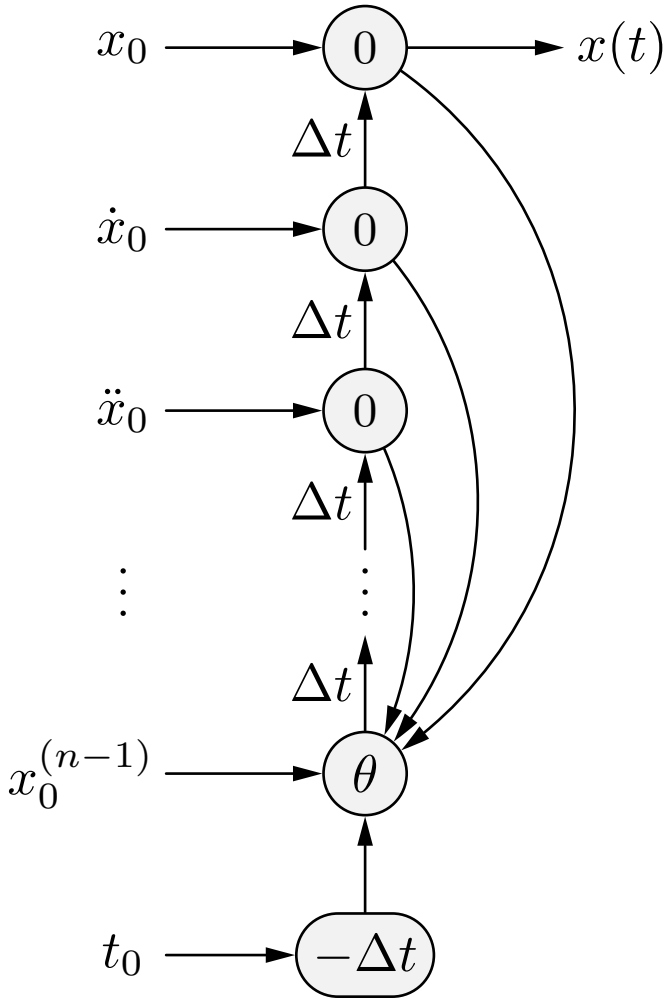
Recurrent Networks: Differential Equations

Replace differential quotient by forward distance to obtain the recursive equations

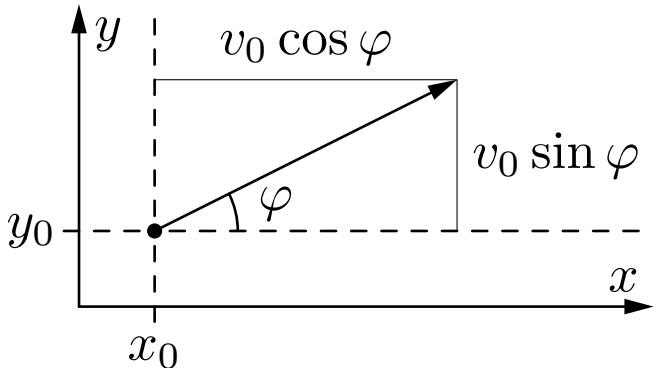
$$
\begin{aligned}
x(t_i) &= x(t_{i-1}) + \Delta t \cdot y_1(t_{i-1}), \\
y_1(t_i) &= y_1(t_{i-1}) + \Delta t \cdot y_2(t_{i-1}), \\
&\vdots \\
y_{n-2}(t_i) &= y_{n-2}(t_{i-1}) + \Delta t \cdot y_{n-3}(t_{i-1}), \\
y_{n-1}(t_i) &= y_{n-1}(t_{i-1}) + f(t_{i-1}, x(t_{i-1}), y_1(t_{i-1}), \ldots, y_{n-1}(t_{i-1}))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

- 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):

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

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

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

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

to reach the system of differential equations:

$$\dot{x} = v_x, \qquad \dot{v}_x = 0,$$

$$\dot{y} = v_y, \qquad \dot{v}_y = -g,$$

from which we get the system of recursive update formulae

$$x(t_i) = x(t_{i-1}) + \Delta t \; v_x(t_{i-1}), \qquad v_x(t_i) = v_x(t_{i-1}),$$

$$y(t_i) = y(t_{i-1}) + \Delta t \; v_y(t_{i-1}), \qquad v_y(t_i) = v_y(t_{i-1}) - \Delta t \; g.$$

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

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

This leads to the recursive update rules

$$\begin{aligned} \vec{r}(t_i) &= \vec{r}(t_{i-1}) + \Delta t\ \vec{v}(t_{i-1}), \\ \vec{v}(t_i) &= \vec{v}(t_{i-1}) - \Delta t\ g\vec{e}_y \end{aligned}$$

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

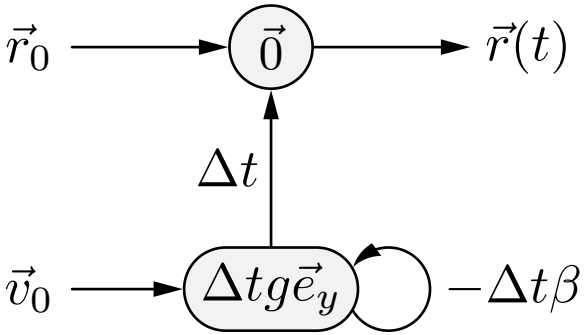
$$\dot{\vec{r}} = \vec{v}, \qquad\qquad \dot{\vec{v}} = -\beta\vec{v} - g\vec{e}_y,$$

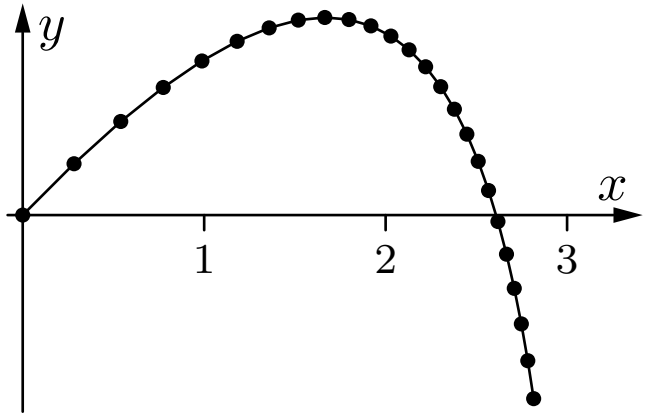
from which we obtain the recursive update formulae

$$\begin{aligned} \vec{r}(t_i) &= \vec{r}(t_{i-1}) + \Delta t\ \vec{v}(t_{i-1}), \\ \vec{v}(t_i) &= \vec{v}(t_{i-1}) - \Delta t\ \beta\ \vec{v}(t_{i-1}) - \Delta t\ g\vec{e}_y. \end{aligned}$$

Recurrent Networks: Diagonal Throw

Resulting recurrent neural network:





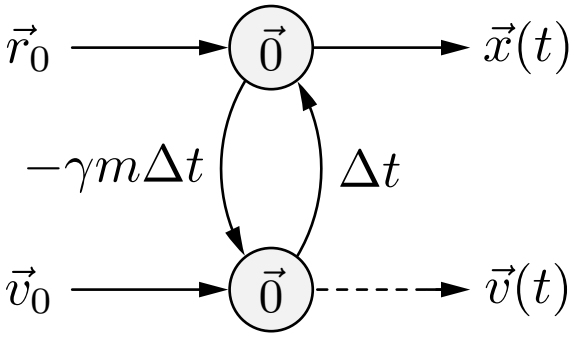
- There are no strange couplings as there would be in a non-vector network.
- Note the deviation from a parabola that is due to the friction.

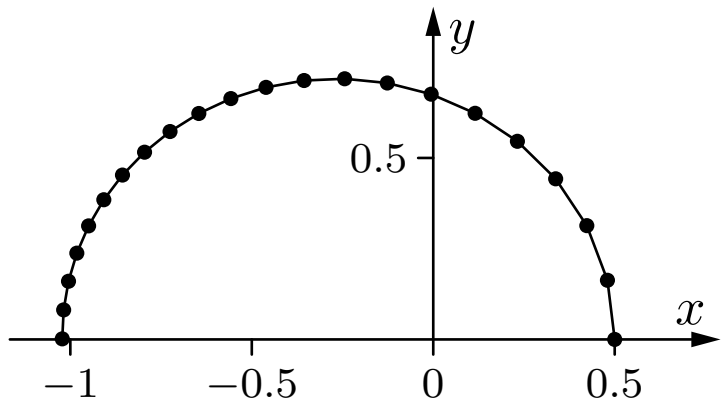
Recurrent Networks: Planet Orbit

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

Recursive update rules:

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





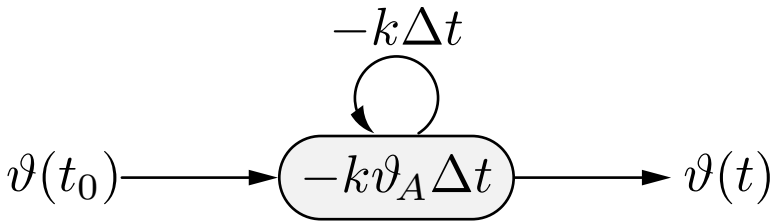
Recurrent Networks: Training

- All recurrent network computations shown up to now are possible only if the differential equation *and* its parameters are known.
- In practice one often knows only the form of the differential equation.
- If measurement data of the described system are available, one may try to find the system parameters by training a recurrent neural network.
- In principle, recurrent neural networks are trained like multi-layer perceptrons, that is, an output error is computed and propagated back through the network.
- However, a direct application of error backpropagation is problematic due to the backward connections / recurrence.
- **Solution:** The backward connections are eliminated by unfolding the network in time between two training patterns (*error backpropagation through time*).
- Technically: create one (pseudo-)neuron for each intermediate point in time.

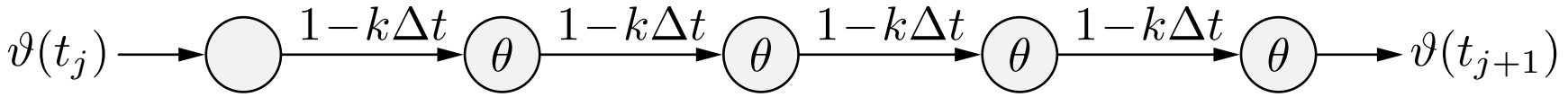
Recurrent Networks: Backpropagation through Time

Example: **Newton's cooling law**

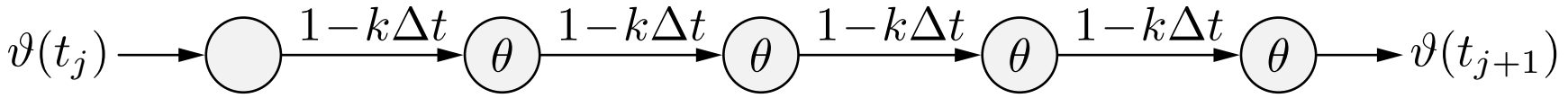
$$\frac{\mathrm{d}\vartheta}{\mathrm{d}t} = \dot{\vartheta} = -k(\vartheta - \vartheta_A)$$



- Assumption: we have measurements of the temperature of a body at different points in time. We also know the temperature ϑ_A of the environment.
- Objective: find the value of the cooling constant k.
- Initialization: like for an MLP, choose random thresholds and weights.
- The time between two consecutive measurement values is split into intervals. Thus the recurrence of the network is *unfolded in time*.
- For example, if there are 4 intervals between two consecutive measurement values ($t_{j+1} = t_j + 4\Delta t$), we obtain



Recurrent Networks: Backpropagation through Time



- For a given measurement $\vartheta(t_j)$, an approximation of $\vartheta(t_{j+1})$ is computed.
- By comparing this (approximate) output with the actual value $\vartheta(t_{j+1})$, an error signal is obtained that is backpropagated through the network and leads to changes of the thresholds and the connection weights.
- Therefore: training is standard backpropagation on the unfolded network.
- However: the above example network possesses only one free parameter, namely the cooling constant k, which is part of the weight and the threshold.
- Therefore: updates can be carried out only after the first neuron is reached.
- Generally, training recurrent networks is beneficial if the system of differential equations cannot be solved analytically.

Supplementary Topic:

Neuro-Fuzzy Systems

Brief Introduction to Fuzzy Theory

- **Classical Logic:** only two truth values *true* and *false*

  Classical Set Theory: either *is element of* or *is not element of*

- The bivalence of the classical theories is often inappropriate.

  Illustrative example: **Sorites Paradox** (greek. *sorites*: pile)

  - One billion grains of sand are a pile of sand. *(true)*
  - If a grain of sand is taken away from a pile of sand, the remainder is a pile of sand. *(true)*

  It follows:

  - 999 999 999 grains of sand are a pile of sand. *(true)*

  Repeated application of this inference finally yields

  - A single grain of sand is a pile of sand. *(false!)*

  At which number of grains of sand is the inference not truth-preserving?

Brief Introduction to Fuzzy Theory

- Obviously: There is no precise number of grains of sand,
  at which the inference to the next smaller number is false.
- Problem: **Terms of natural language are vague.**
  (e.g.. "pile of sand", "bald", "warm", "fast", "light", "high pressure")
- Note: **Vague terms are *inexact*, but nevertheless *not useless*.**
  - Even for vague terms there are situations or objects,
    to which they are *certainly applicable*, and others,
    to which they are *certainly not applicable*.
  - In between lies a so-called **penumbra** (lat. for *half shadow*) of situations, in which it is unclear whether the terms are applicable, or in which they are applicable only with certain restrictions ("small pile of sand").
  - Fuzzy theory tries to model this penumbra in a mathematical fashion
    ("soft transition" between *applicable* and *not applicable*).

Fuzzy Logic

- **Fuzzy Logic** is an extension of classical logic
  by values between *true* and *false*.
- **Truth values are any values from the real interval $[0, 1]$**,
  where $0 \mathrel{\widehat{=}}$ *false* and $1 \mathrel{\widehat{=}}$ *true*.
- Therefore necessary: **extension of the logical operators**
  - Negation classical: $\neg a$, fuzzy: $\sim a$ Fuzzy-Negation
  - Conjunction classical: $a \wedge b$, fuzzy: $\top(a, b)$ t-Norm
  - Disjunction classical: $a \vee b$, fuzzy: $\bot(a, b)$ t-Konorm
- **Basic principles of this extension:**
  - For the extreme values 0 and 1 the operations should behave exactly
    like their classical counterparts (border or corner conditions).
  - For the intermediate values the behavior should be monotone.
  - As far as possible, the laws of classical logic should be preserved.

Fuzzy Negations

A **fuzzy negation** is a function $\sim\colon [0,1] \to [0,1]$,
that satisfies the following conditions:

- $\sim 0 = 1 \quad$ and $\quad \sim 1 = 0 \qquad\qquad$ (boundary conditions)
- $\forall a, b \in [0,1]: \quad a \leq b \;\Rightarrow\; \sim a \geq \sim b \qquad$ (monotony)

If in the second condition the relations $<$ and $>$ hold instead of merely $\leq$ and $\geq$,
the fuzzy negation is called a *strict* negation.

Additional conditions that are sometimes required are:

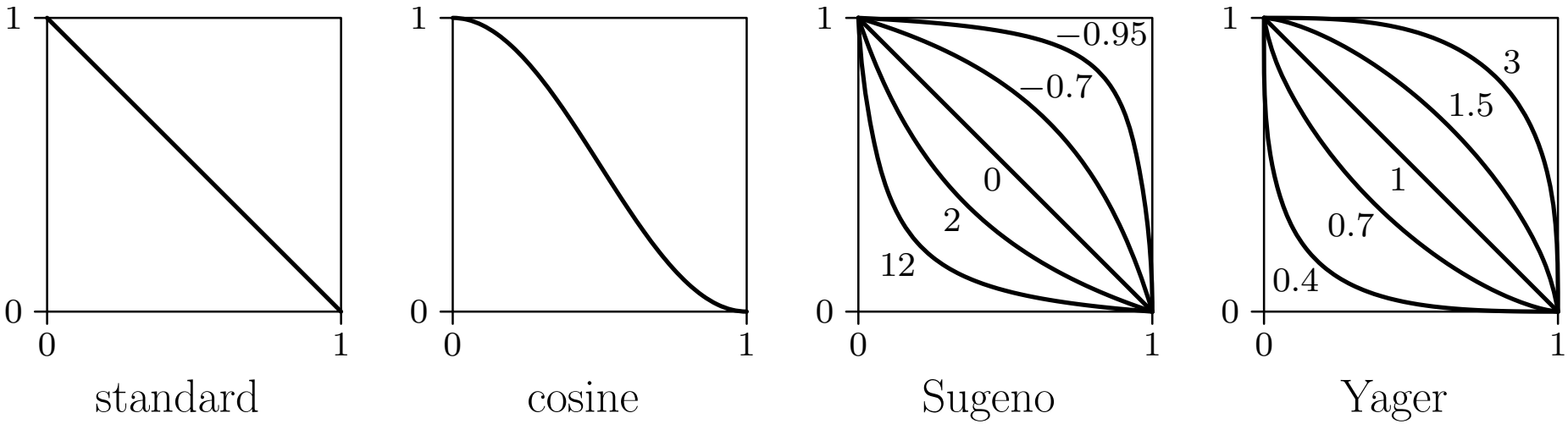
- $\sim$ is a continuous function.
- $\sim$ is *involutive*, that is, $\quad \forall a \in [0,1]: \quad \sim\sim a = a.$

Involutivity corresponds to the classical *law of identity* $\neg\neg a = a$.

The above conditions do not uniquely determine a fuzzy negation.

Fuzzy Negations

$$
\begin{array}{lllll}
\text{standard negation:} & \sim a & = & 1-a \\
\text{threshold negation:} & \sim(a;\theta) & = & \begin{cases} 1 & \text{if } x \leq \theta, \\ 0 & \text{otherwise.} \end{cases} \\
\text{cosine negation:} & \sim a & = & \frac{1}{2}(1+\cos \pi a) \\
\text{Sugeno negation:} & \sim(a;\lambda) & = & \dfrac{1-a}{1+\lambda a} \\
\text{Yager negation:} & \sim(a;\lambda) & = & (1-a^{\lambda})^{\frac{1}{\lambda}}
\end{array}
$$



t-Norms / Fuzzy Conjunctions

A **t-norm** or **fuzzy conjunction** is a function $\top : [0,1]^2 \to [0,1]$,
that satisfies the following conditions:

- $\forall a \in [0,1]: \quad \top(a,1) = a$ (boundary condition)
- $\forall a,b,c \in [0,1]: \quad b \leq c \Rightarrow \top(a,b) \leq \top(a,c)$ (monotony)
- $\forall a,b \in [0,1]: \quad \top(a,b) = \top(b,a)$ (commutativity)
- $\forall a,b,c \in [0,1]: \quad \top(a,\top(b,c)) = \top(\top(a,b),c)$ (associativity)

Additional conditions that are sometimes required are:

- $\top$ is a continuous function (continuity)
- $\forall a \in [0,1]: \quad \top(a,a) < a$ (sub-idempotency)
- $\forall a,b,c,d \in [0,1]: \quad a < b \wedge c < d \Rightarrow \top(a,b) < \top(c,d)$ (strict monotony)

The first two of these conditions (in addition to the top four)
define the sub-class of so-called *Archimedic t-norms*.

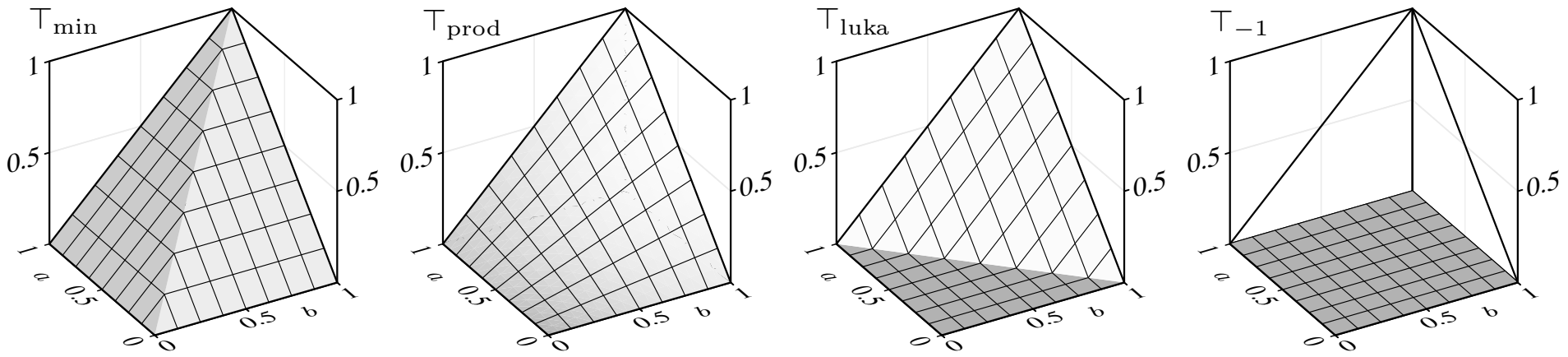
t-Norms / Fuzzy Conjunctions

standard conjunction: $\top_{\min}(a, b) = \min\{a, b\}$

algebraic product: $\top_{\text{prod}}(a, b) = a \cdot b$

Łukasiewicz: $\top_{\text{luka}}(a, b) = \max\{0, a + b - 1\}$

drastic product:
$$\top_{-1}(a, b) = \begin{cases} a & \text{if } b = 1, \\ b & \text{if } a = 1, \\ 0 & \text{otherwise.} \end{cases}$$



t-Conorms / Fuzzy Disjunctions

A **t-conorm** or **fuzzy disjunction** is a function $\perp : [0,1]^2 \to [0,1]$,
that satisfies the following conditions:

- $\forall a \in [0,1]: \quad \perp(a,0) = a$ (boundary condition)
- $\forall a,b,c \in [0,1]: \quad b \leq c \Rightarrow \perp(a,b) \leq \perp(a,c)$ (monotony)
- $\forall a,b \in [0,1]: \quad \perp(a,b) = \perp(b,a)$ (commutativity)
- $\forall a,b,c \in [0,1]: \quad \perp(a,\perp(b,c)) = \perp(\perp(a,b),c)$ (assoziativity)

Additional conditions that are sometimes required are:

- $\perp$ is a continuous function (continuity)
- $\forall a \in [0,1]: \quad \perp(a,a) > a$ (super-idempotency)
- $\forall a,b,c,d \in [0,1]: \quad a < b \wedge c < d \Rightarrow \perp(a,b) < \perp(c,d)$ (strict monotony)

The first two of these conditions (in addition to the top four)
define the sub-class of so-called *Archimedic t-conorms*.

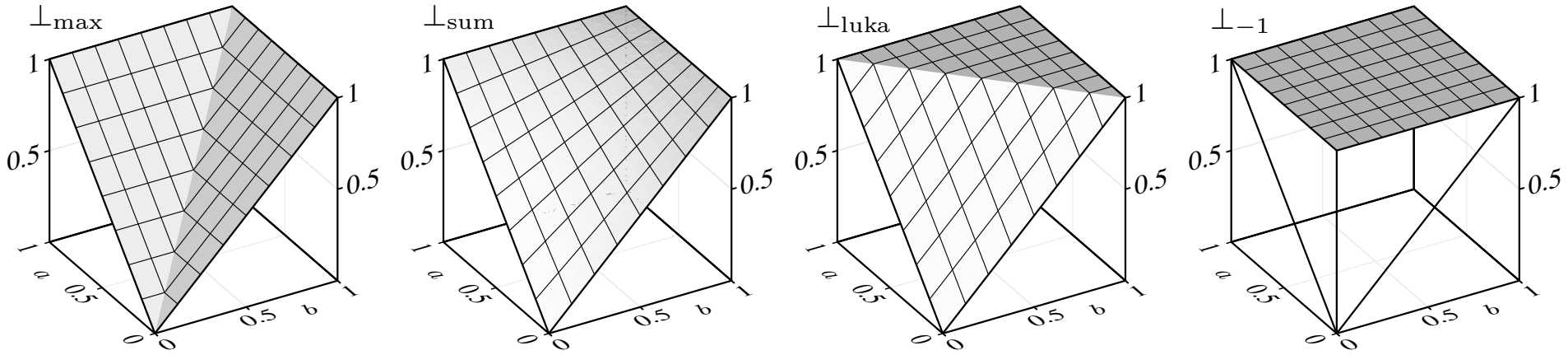
t-Conorms / Fuzzy Disjunctions

standard disjunction: $\perp_{\max}(a,b) = \max\{a,b\}$

algebraische sum: $\perp_{\text{sum}}(a,b) = a + b - a \cdot b$

Łukasiewicz: $\perp_{\text{luka}}(a,b) = \min\{1, a+b\}$

drastic sum:
$$\perp_{-1}(a,b) = \begin{cases} a & \text{if } b = 0, \\ b & \text{if } a = 0, \\ 1 & \text{otherwise.} \end{cases}$$



Interplay of the Fuzzy Operators

- It is $\forall a, b \in [0,1]: \quad \top_{-1}(a,b) \leq \top_{\text{luka}}(a,b) \leq \top_{\text{prod}}(a,b) \leq \top_{\min}(a,b)$.
  All other possible t-norms lie between $\top_{-1}$ and $\top_{\min}$ as well.
- It is $\forall a, b \in [0,1]: \quad \bot_{\max}(a,b) \leq \bot_{\text{sum}}(a,b) \leq \bot_{\text{luka}}(a,b) \leq \bot_{-1}(a,b)$.
  All other possible t-conorms lie between $\bot_{\max}$ and $\bot_{-1}$ as well.
- Note: Generally *neither* $\top(a, \sim a) = 0$ *nor* $\bot(a, \sim a) = 1$ holds.
- A set of operators $(\sim, \top, \bot)$ consisting of a fuzzy negation $\sim$, a t-norm $\top$, and a t-conorm $\bot$ is called a **dual triplet** if with these operators DeMorgan's laws hold, that is, if

$$\forall a, b \in [0,1]: \quad \sim\top(a,b) = \bot(\sim a, \sim b)$$

$$\forall a, b \in [0,1]: \quad \sim\bot(a,b) = \top(\sim a, \sim b)$$

- The most frequently used set of operators is the dual triplet $(\sim, \top_{\min}, \bot_{\max})$ with the standard negation $\sim a \equiv 1 - a$.

Fuzzy Set Theory

- Classical set theory is based on the notion "*is element of*" ($\in$).
  Alternatively the membership in a set can be described
  with the help of an *indicator function*:

  Let X be a set (base set). Then

  $$I_M : X \to \{0, 1\}, \qquad I_M(x) = \begin{cases} 1 & \text{if } x \in X, \\ 0 & \text{otherwise,} \end{cases}$$

  is called **indicator function** of the set M w.r.t. the base set X.

- In fuzzy set theory the indicator function of classical set theory
  is replaced by a *membership function*:

  Let X be a (classical/crisp) set. Then

  $$\mu_M : X \to [0, 1], \qquad \mu_M(x) \mathrel{\widehat{=}} \text{degree of membership of } x \text{ to } M,$$

  is called **membership function** of the **fuzzy set** M w.r.t. the *base set* X.

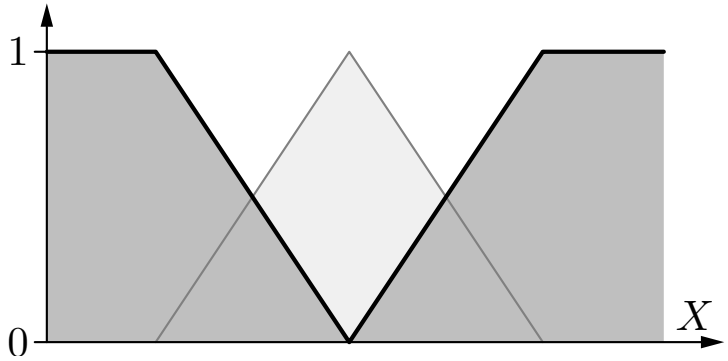
  Usually the fuzzy set is identified with its membership function.

Fuzzy Set Theory: Operations

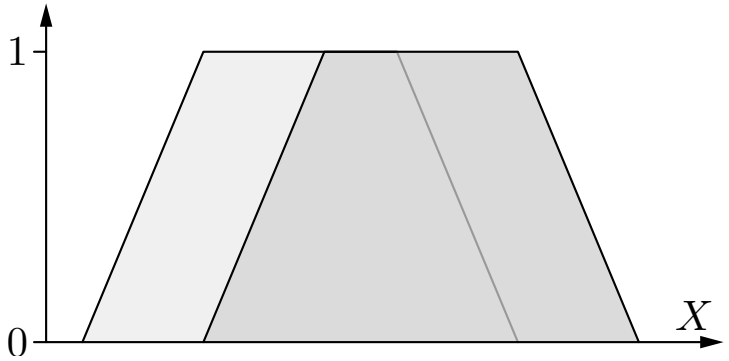
- In analogy to the transition from classical logic to fuzzy logic the transition from classical set theory to fuzzy set theory requires an **extension of the operators**.
- **Basic principle of the extension**:
  Draw on the logical definition of the operators.
  $\Rightarrow$ element-wise application of the logical operators.
- Let A and B be (fuzzy) sets w.r.t. the base set X.

complement	classical	$\overline{A} = \{x \in X \mid x \notin A\}$
	fuzzy	$\forall x \in X: \quad \mu_{\overline{A}}(x) = \sim \mu_A(x)$
intersection	classical	$A \cap B = \{x \in X \mid x \in A \wedge x \in B\}$
	fuzzy	$\forall x \in X: \quad \mu_{A \cap B}(x) = \top(\mu_A(x), \mu_B(x))$
union	classical	$A \cup B = \{x \in X \mid x \in A \vee x \in B\}$
	fuzzy	$\forall x \in X: \quad \mu_{A \cup B}(x) = \bot(\mu_A(x), \mu_B(x))$

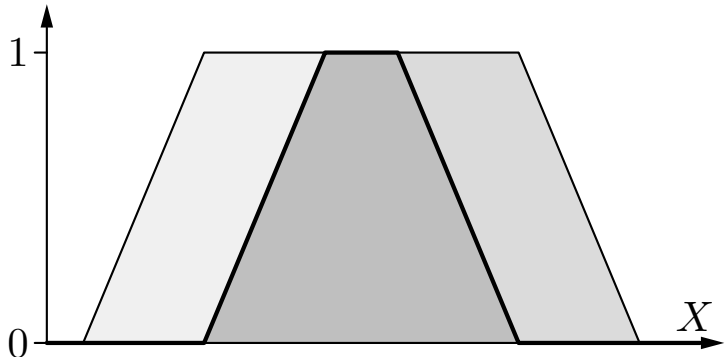
Fuzzy Set Operations: Examples



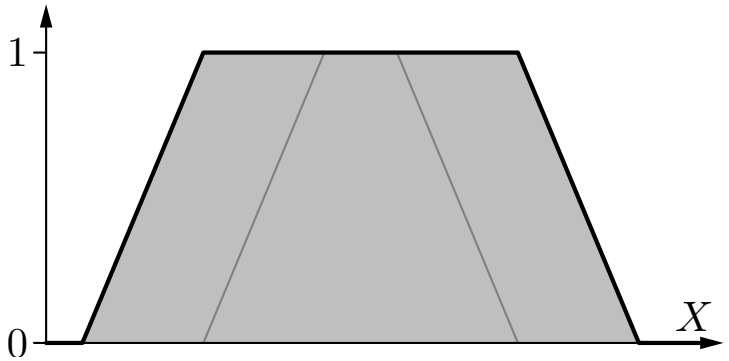
fuzzy complement



two fuzzy sets



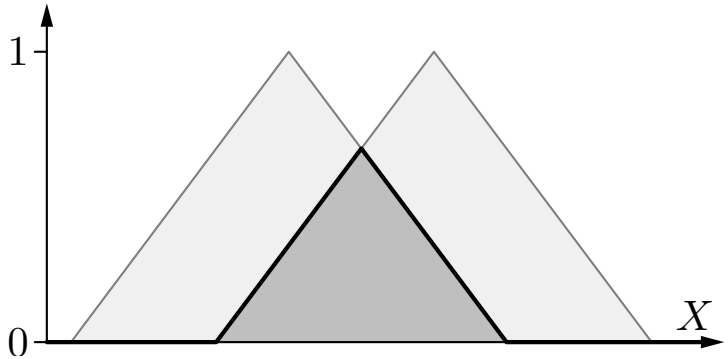
fuzzy intersection



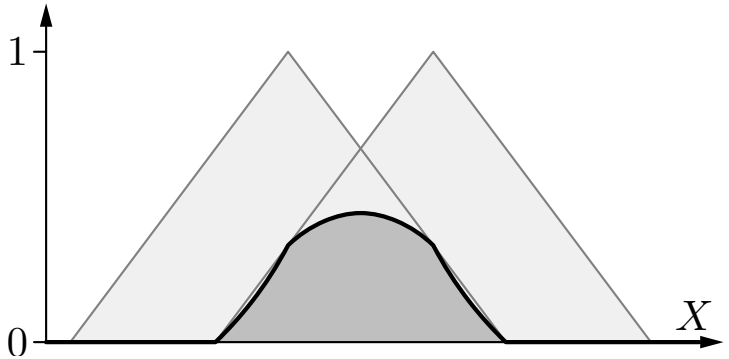
fuzzy union

- The fuzzy intersection shown on the left and the fuzzy union shown on the right are independent of the chosen t-norm or t-conorm.

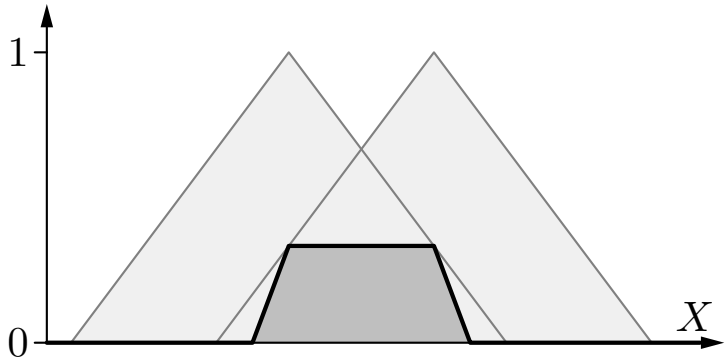
Fuzzy Intersection: Examples



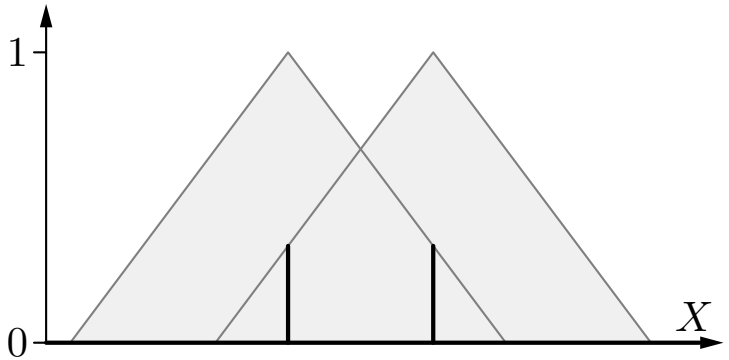
t-norm $\top_{\text{min}}$



t-norm $\top_{\text{prod}}$



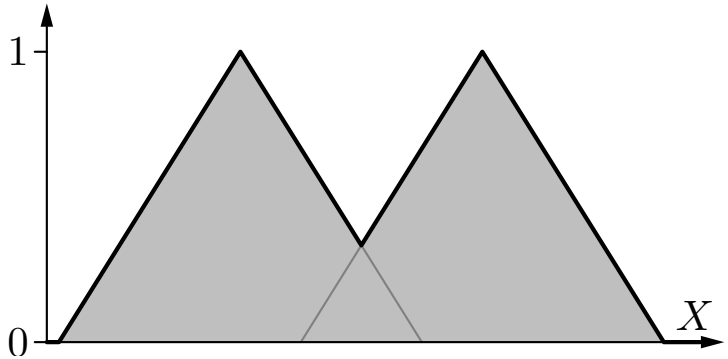
t-norm $\top_{\text{luka}}$



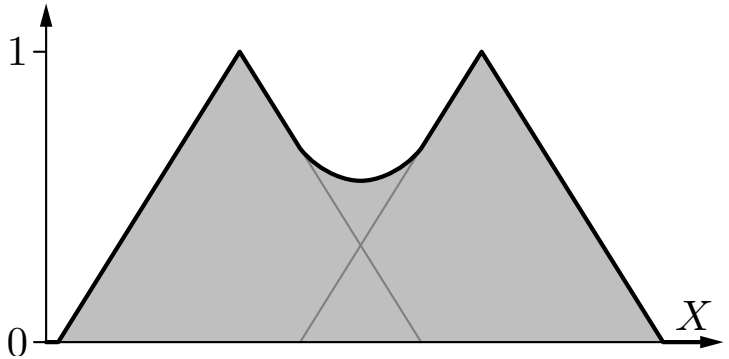
t-norm $\top_{-1}$

- Note that all fuzzy intersections lie between the one shown at the top right and the one shown at the right bottom.

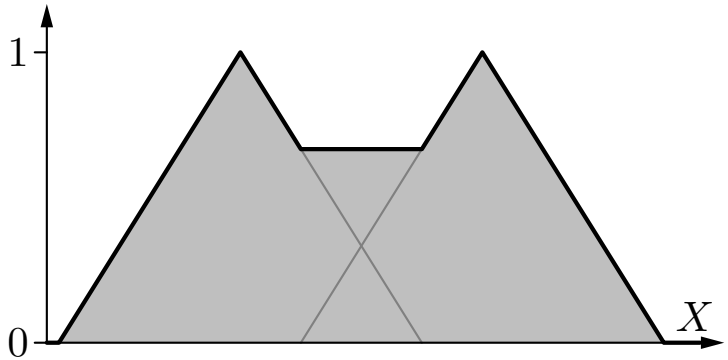
Fuzzy Union: Examples



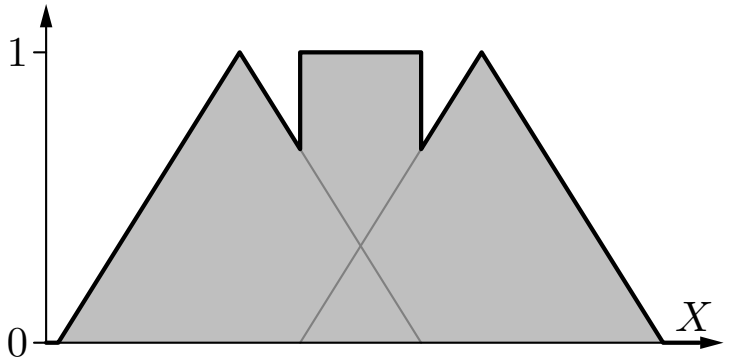
t-norm $\top_{\min}$



t-norm $\top_{\text{prod}}$



t-norm $\top_{\text{luka}}$



t-norm $\top_{-1}$

- Note that all fuzzy unions lie between the one shown at the top right and the one shown at the right bottom.

Fuzzy Partitions and Linguistic Variables

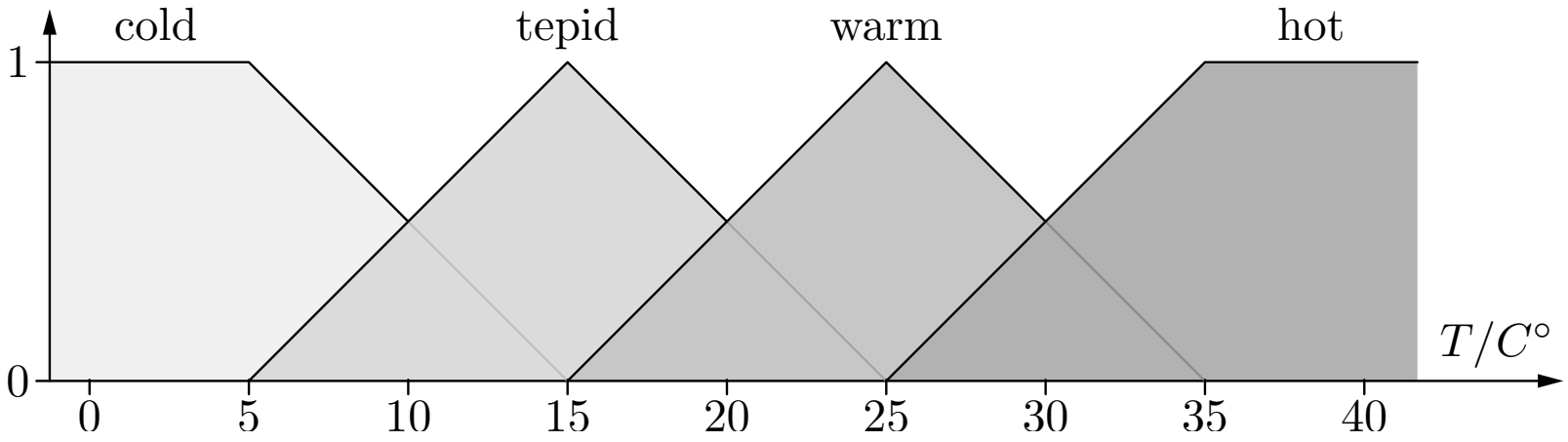
- In order to describe a domain by linguistic terms,
  it is fuzzy-partitioned with the help of fuzzy sets.

  To each fuzzy set of the partition a linguistic term is assigned.

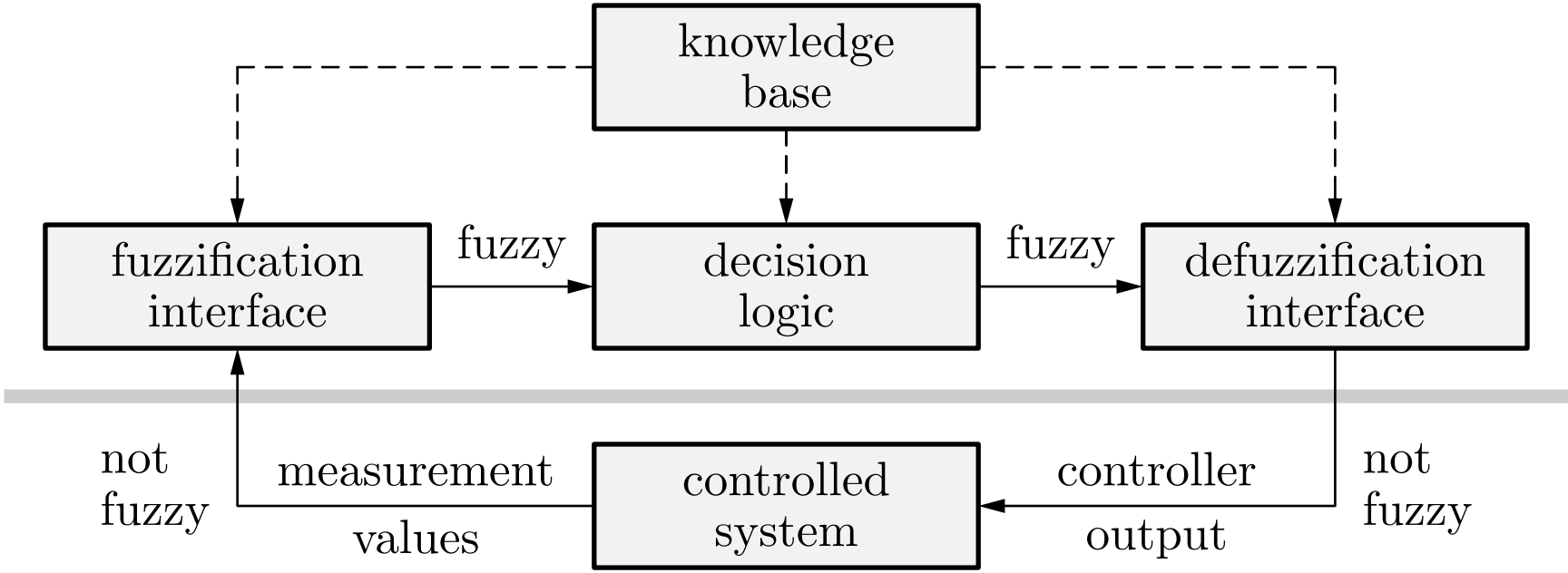
- Common condition: At each point the membership values of the fuzzy sets must sum to 1 *(partition of unity)*.

Example: **fuzzy partition for temperatures**

We define a linguistic variable with the values *cold*, *tepid*, *warm* and *hot*.

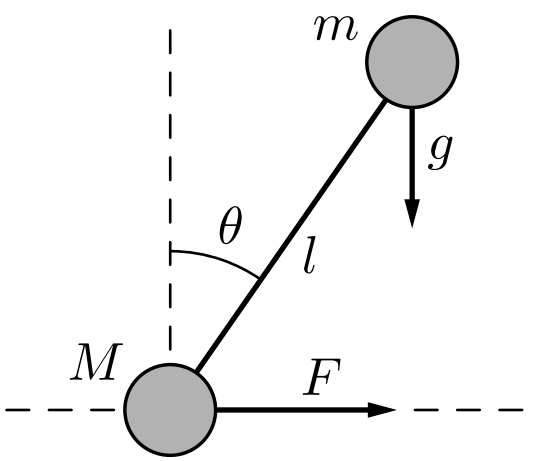


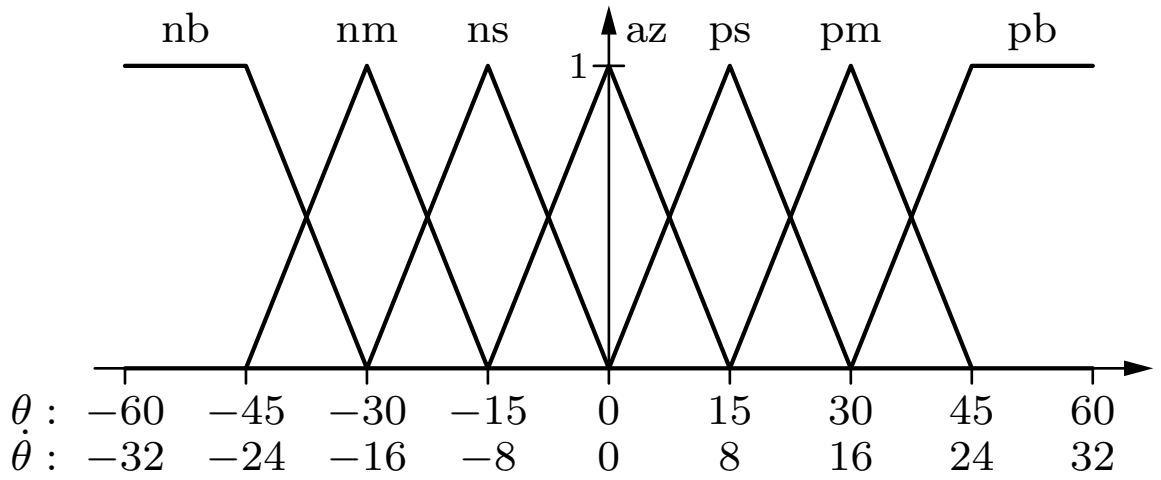
Architecture of a Fuzzy Controller



- The knowledge base contains the fuzzy rules of the controller as well as the fuzzy partitions of the domains of the variables.
- A fuzzy rule reads: **if** X_1 **is** $A_{i_1}^{(1)}$ **and** ... **and** X_n **is** $A_{i_n}^{(n)}$ **then** Y **is** B.
  $X_1, \ldots, X_n$ are the measurement values and Y is the controller output.
  $A_{i_k}^{(k)}$ and B are linguistic terms to which fuzzy sets are assigned.

Example Fuzzy Controller: Inverted Pendulum



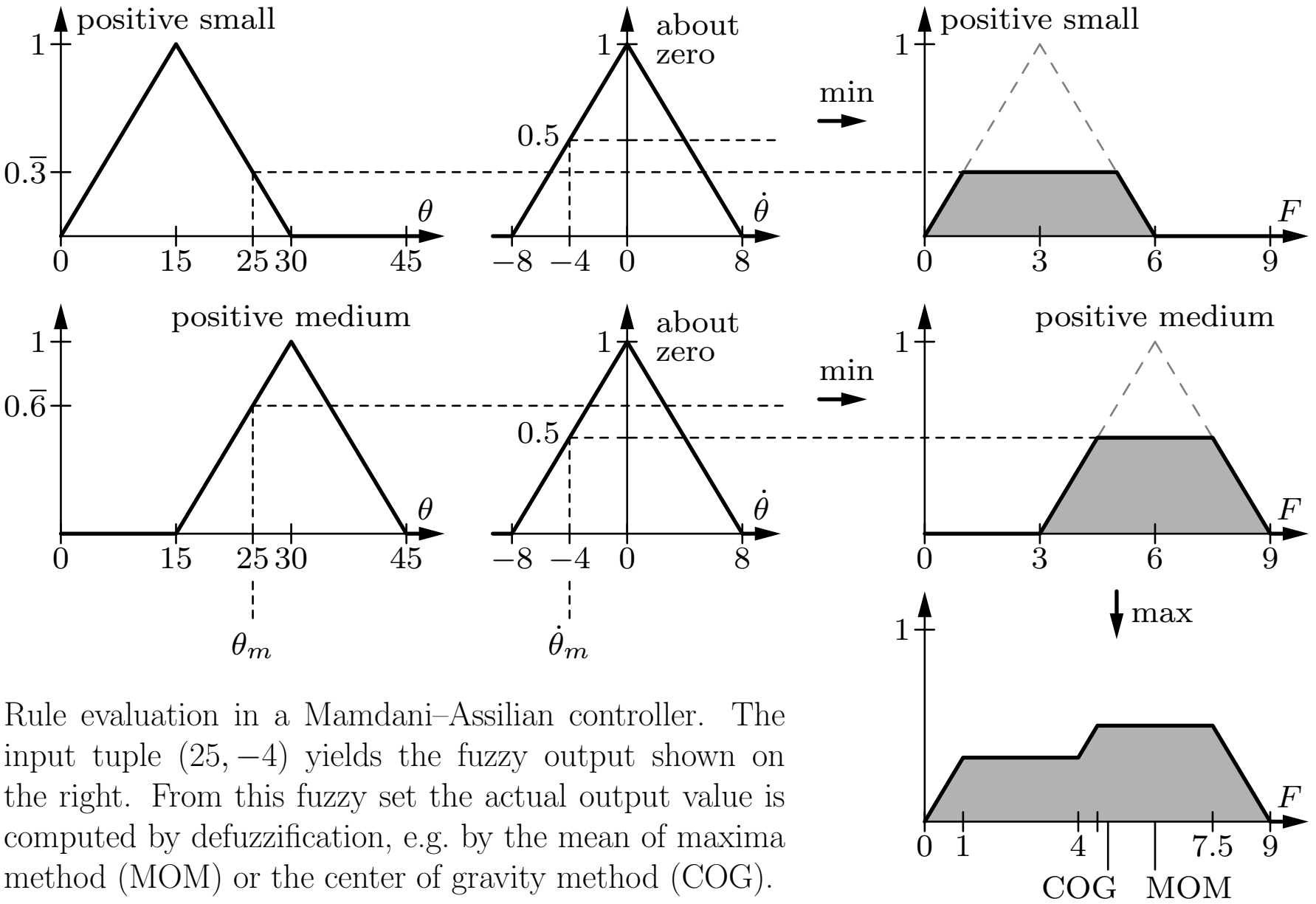


abbreviations

pb – positive big
pm – positive medium
ps – positive small
az – approximately zero
ns – negative small
nm – negative medium
nb – negative big

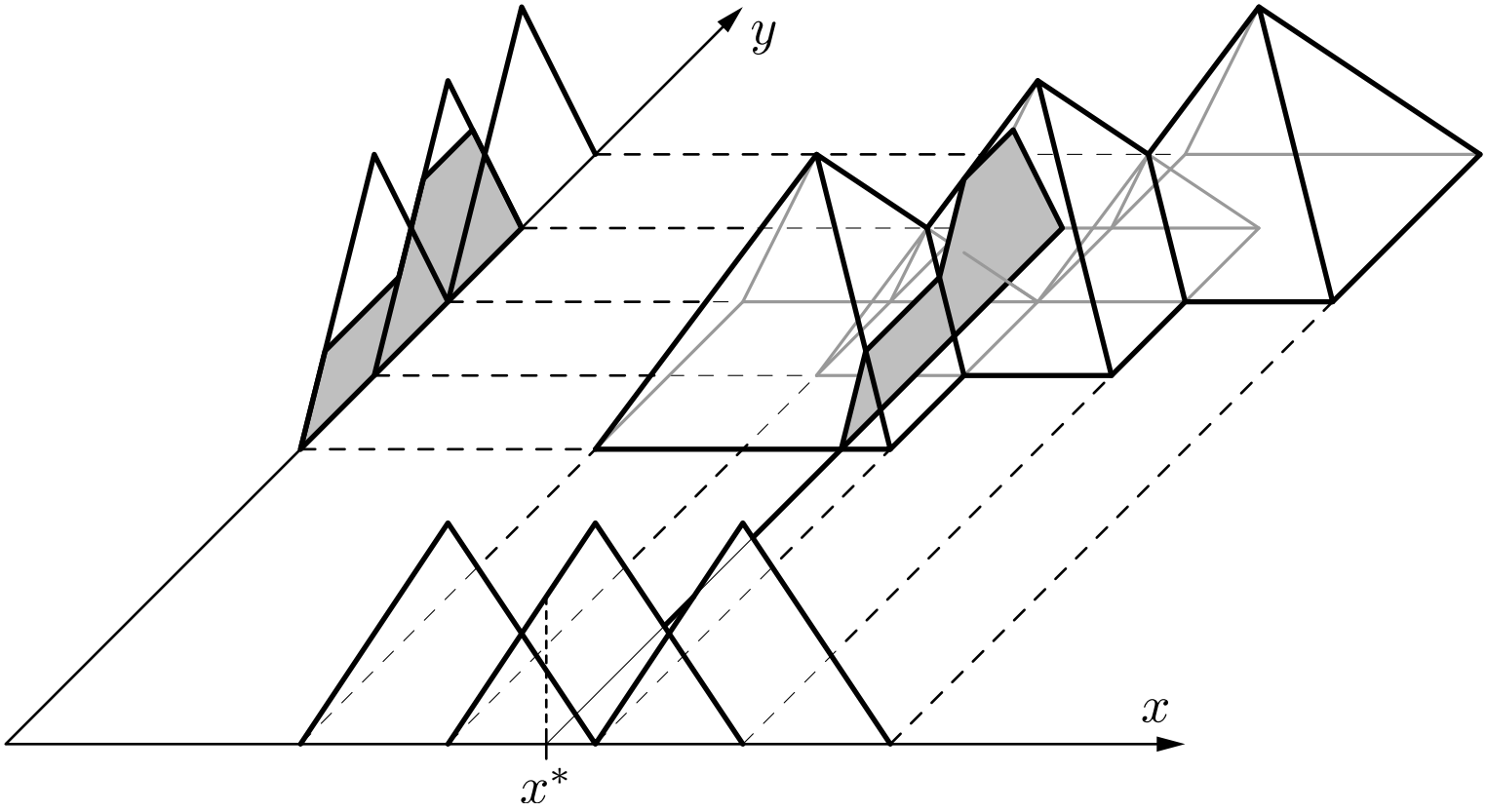
$\dot{\theta}\backslash\theta$	nb	nm	ns	az	ps	pm	pb
pb			**ps**	**pb**			
pm				**pm**			
ps	**nm**		**az**	**ps**			
az	**nb**	**nm**	**ns**	**az**	**ps**	**pm**	**pb**
ns				**ns**	**az**		**pm**
nm				**nm**			
nb				**nb**	**ns**		

Mamdani–Assilian Controller



Rule evaluation in a Mamdani–Assilian controller. The input tuple $(25, -4)$ yields the fuzzy output shown on the right. From this fuzzy set the actual output value is computed by defuzzification, e.g. by the mean of maxima method (MOM) or the center of gravity method (COG).

Mamdani–Assilian Controller



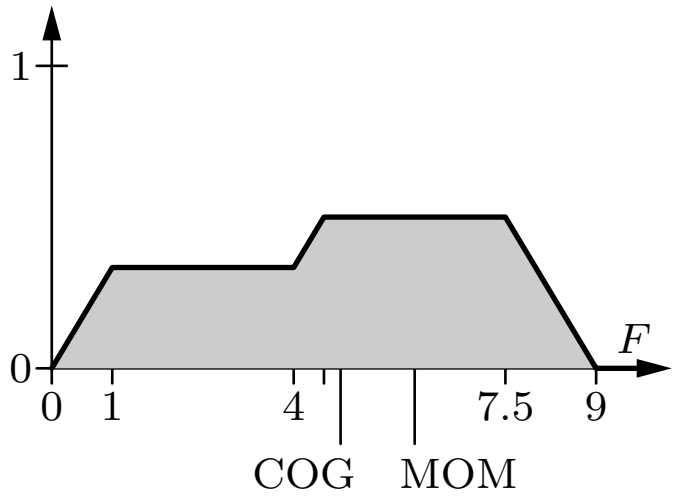
A fuzzy control system with one measurement and one control variable and three fuzzy rules. Each pyramid is specified by one fuzzy rule. The input value x^* leads to the fuzzy output shown in gray.

Defuzzification

The evaluation of the fuzzy rules yields an **output fuzzy set**.

The output fuzzy set has to be turned into a **crisp controller output**.

This process is called **defuzzification**.



The most important defuzzification methods are:

- **Center of Gravity** (COG)
  The center of gravity of the area under the output fuzzy set.
- **Center of Area** (COA)
  The point that divides the area under the output fuzzy set into equally large parts.
- **Mean of Maxima** (MOM)
  The arithmetic mean of the points with maximal degree of membership.

Takagi–Sugeno–Kang Controller (TSK Controller)

- The rules of a Takagi–Sugeno–Kang controller have the same kind of antecedent as the rules of a Mamdani–Assilian controller, but a different kind of consequent:

  R_i: **if** x_1 is $\mu_i^{(1)}$ **and** ... **and** x_n is $\mu_i^{(n)}$, **then** $y = f_i(x_1, \ldots, x_n)$.

  The consequent of a Takagi–Sugeno–Kang rule specifies a function of the inputs that is to be computed if the antecedent is satisfied.

- Let $\tilde{a}_i$ be the activation of the antecedent of the rule R_i, that is,

$$\tilde{a}_i(x_1, \ldots, x_n) = \top\left(\top\left(\ldots\top\left(\mu_i^{(1)}(x_1), \mu_i^{(2)}(x_2)\right), \ldots\right), \mu_i^{(n)}(x_n)\right).$$

- Then the output of a Takagi–Sugeno–Kang controller is computed as

$$y(x_1, \ldots, x_n) = \frac{\sum_{i=1}^{r} \tilde{a}_i \cdot f_i(x_1, \ldots, x_n)}{\sum_{i=1}^{r} \tilde{a}_i},$$

  that is, the controller output is a weighted average of the outputs of the individual rules, where the activations of the antecedents provide the weights.

Neuro-Fuzzy Systems

- **Disadvantages of Neural Networks:**
  - Training results are difficult to interpret (black box).

    The result of training an artificial neural network are matrices or vectors of real-valued numbers. Even though the computations are clearly defined, humans usually have trouble understanding what is going on.
  - It is difficult to specify and incorporate prior knowledge.

    Prior knowledge would have to be specified as the mentioned matrices or vectors of real-valued numbers, which are difficult to understand for humans.
- **Possible Remedy:**
  - Use a hybrid system, in which an artificial neural network is coupled with a rule-based system.

    The rule-based system can be interpreted and set up by a human.
  - One such approach are **neuro-fuzzy systems**.

Neuro-Fuzzy Systems

Neuro-fuzzy systems are commonly divided into cooperative and hybrid systems.

- **Cooperative Models:**
  - A neural network and a fuzzy controller work independently.
  - The neural network generates (offline) or optimizes (online) certain parameters.
- **Hybrid Models:**
  - Combine the structure of a neural network and a fuzzy controller.
  - A hybrid neuro-fuzzy controller can be interpreted as a neural network and can be implemented with the help of a neural network.
  - Advantages: integrated structure;
    no communication between two different models is needed;
    in principle, both offline and online training are possible,

Hybrid models are more accepted and more popular than cooperative models.

Neuro-Fuzzy Systems: Hybrid Methods

- Hybrid methods map fuzzy sets and fuzzy rules to a neural network structure.
- The activation $\tilde{a}_i$ of the antecedent of a Mamdani–Assilian rule

  R_i: **if** x_1 is $\mu_i^{(1)}$ **and** ... **and** x_n is $\mu_i^{(n)}$, **then** y is ν_i.

  or of a Takagi–Sugeno–Kang rule

  R_i: **if** x_1 is $\mu_i^{(1)}$ **and** ... **and** x_n is $\mu_i^{(n)}$, **then** $y = f_i(x_1, \ldots, x_n)$.

  is computed with a t-norm $\top$ (most commonly $\top_{\min}$).
- For given input values $x_1, \ldots, x_n$ the network structure has to compute:

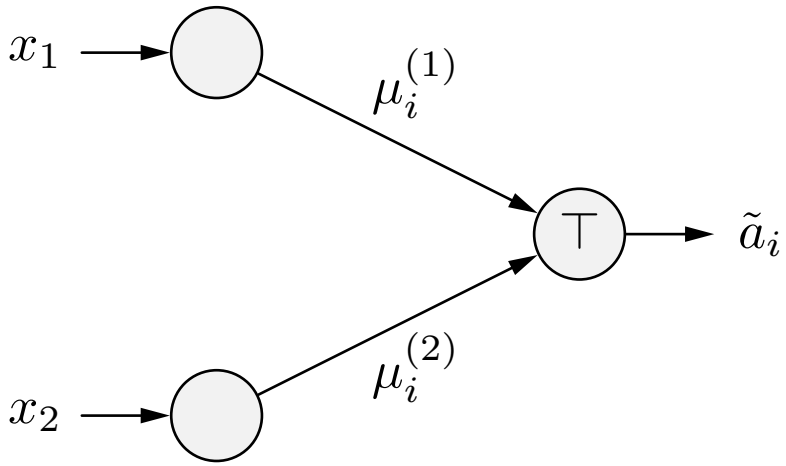
$$\tilde{a}_i(x_1, \ldots, x_n) = \top\left(\top\left(\ldots\top\left(\mu_i^{(1)}(x_1), \mu_i^{(2)}(x_2)\right), \ldots\right), \mu_i^{(n)}(x_n)\right).$$

  (Since t-norms are associative, it does not matter in which order the membership degrees are combined by successive pairwise applications of the t-norm.)

Computing the activation $\tilde{a}_i$ of the antecedent of a fuzzy rule:

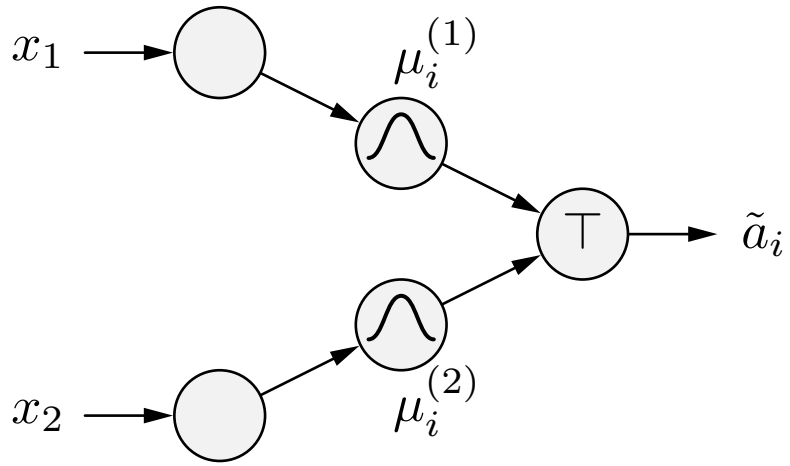
The fuzzy sets appearing in the antecedents of fuzzy rules can be modeled

as connection weights:



The neurons in the first hidden layer represent the rule antecedents and the connections from the input units represent the fuzzy sets of these antecedents.

as activation functions:



The neurons in the first hidden layer represent the fuzzy sets and the neurons in the second hidden layer represent the rule antecedents ($\top$ is a t-norm).

From Fuzzy Rule Base to Network Structure

If the fuzzy sets are represented as activation functions (right on previous slide), a fuzzy rule base is turned into a network structure as follows:

1. For every input variable x_i: create a neuron in the input layer.
2. For every output variable y_i: create a neuron in the output layer.
3. For every fuzzy set $\mu_i^{(j)}$: create a neuron in the first hidden layer and connect it to the input neuron corresponding to x_i.
4. For every fuzzy rule R_i: create a (rule) neuron in the second hidden layer and specify a t-norm for computing the rule (antecedent) activation.
5. Connect each rule neuron to the neurons that represent the fuzzy sets of the antecedent of its corresponding fuzzy rule R_i.
6. (This step depends on whether the controller is of the Mamdani–Assilian or of the Takagi–Sugeno–Kang type; see next slide.)

From Fuzzy Rule Base to Network Structure

Mamdani–Assilian controller:

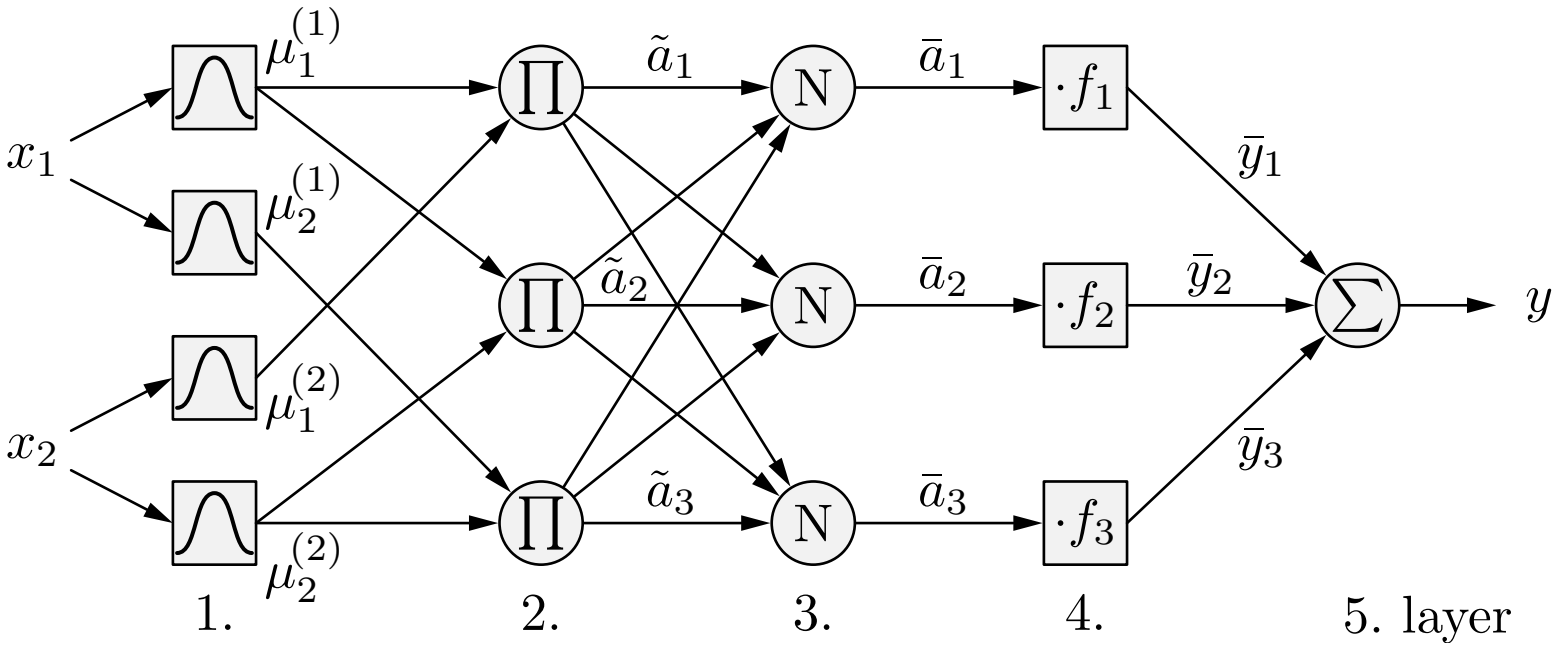
6. Connect each rule neuron to the output neuron corresponding to the consequent domain of its fuzzy rule. As connection weight choose the consequent fuzzy set of the fuzzy rule. Furthermore, a t-conorm for combining the outputs of the individual rules and a defuzzification method have to be integrated adequately into the output neurons (e.g. as network input and activation functions).

Takagi–Sugeno–Kang controller:

6. For each rule neuron, create a sibling neuron that computes the output function of the corresponding fuzzy rule and connect all input neurons to it (arguments of the consequent function). All rule neurons that refer to the same output domain as well as their sibling neurons are connected to the corresponding output neuron (in order to compute the weighted average of the rule outputs).

The resulting network structure can now be trained with procedures that are analogous to those of standard neural networks (e.g. error backpropagation).

Adaptive Network-based Fuzzy Inference Systems (ANFIS)

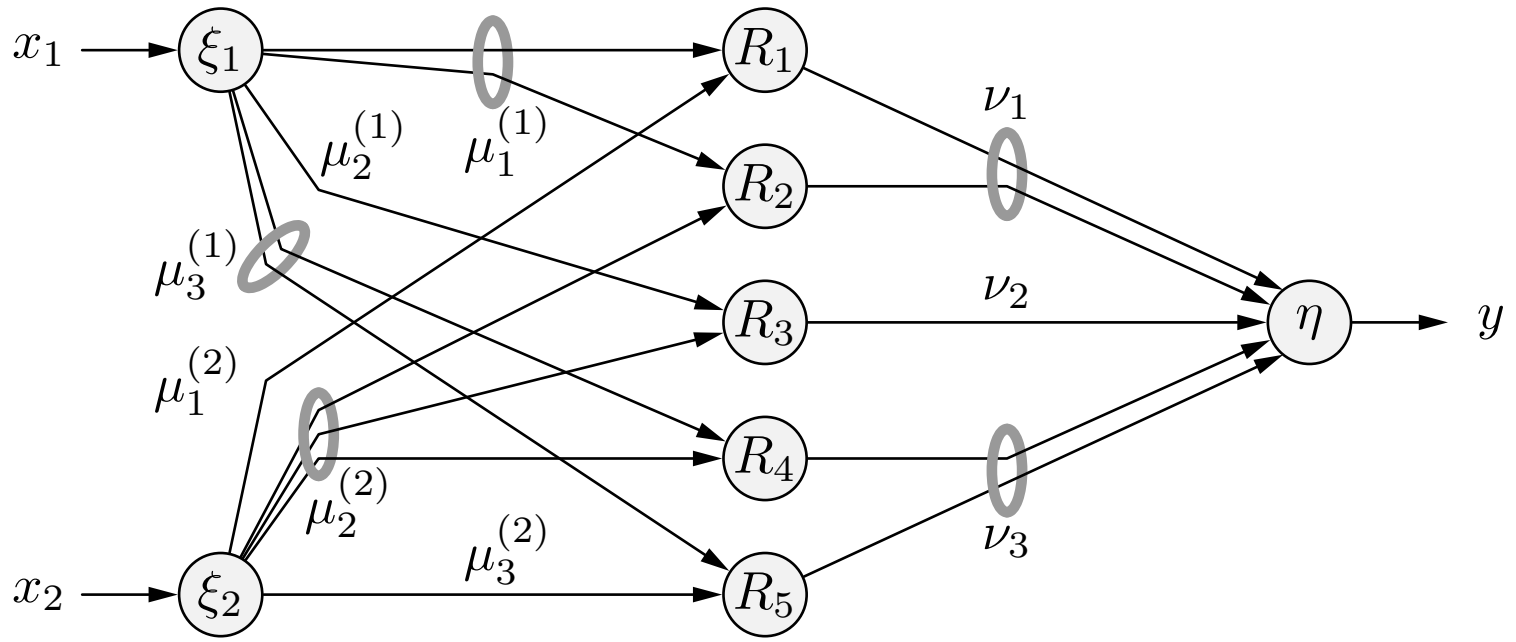


(Connections from inputs to output function neurons for f_1, f_2, f_3 not shown.)

This ANFIS network represents the fuzzy rule base (Takagi–Sugeno–Kang rules):

$$
\begin{aligned}
R_1\text{:}\quad & \textbf{if} \quad x_1 \text{ is } \mu_1^{(1)} \quad \textbf{and} \quad x_2 \text{ is } \mu_1^{(2)}, \quad \textbf{then} \quad y = f_1(x_1, x_2) \\
R_2\text{:}\quad & \textbf{if} \quad x_1 \text{ is } \mu_1^{(1)} \quad \textbf{and} \quad x_2 \text{ is } \mu_2^{(2)}, \quad \textbf{then} \quad y = f_2(x_1, x_2) \\
R_3\text{:}\quad & \textbf{if} \quad x_1 \text{ is } \mu_2^{(1)} \quad \textbf{and} \quad x_2 \text{ is } \mu_2^{(2)}, \quad \textbf{then} \quad y = f_3(x_1, x_2)
\end{aligned}
$$

Neuro-Fuzzy Control (NEFCON)



This NEFCON network represents the fuzzy rule base (Mamdani–Assilian rules):

R_1: **if** x_1 is $\mu_1^{(1)}$ **and** x_2 is $\mu_1^{(2)}$, **then** y is ν_1

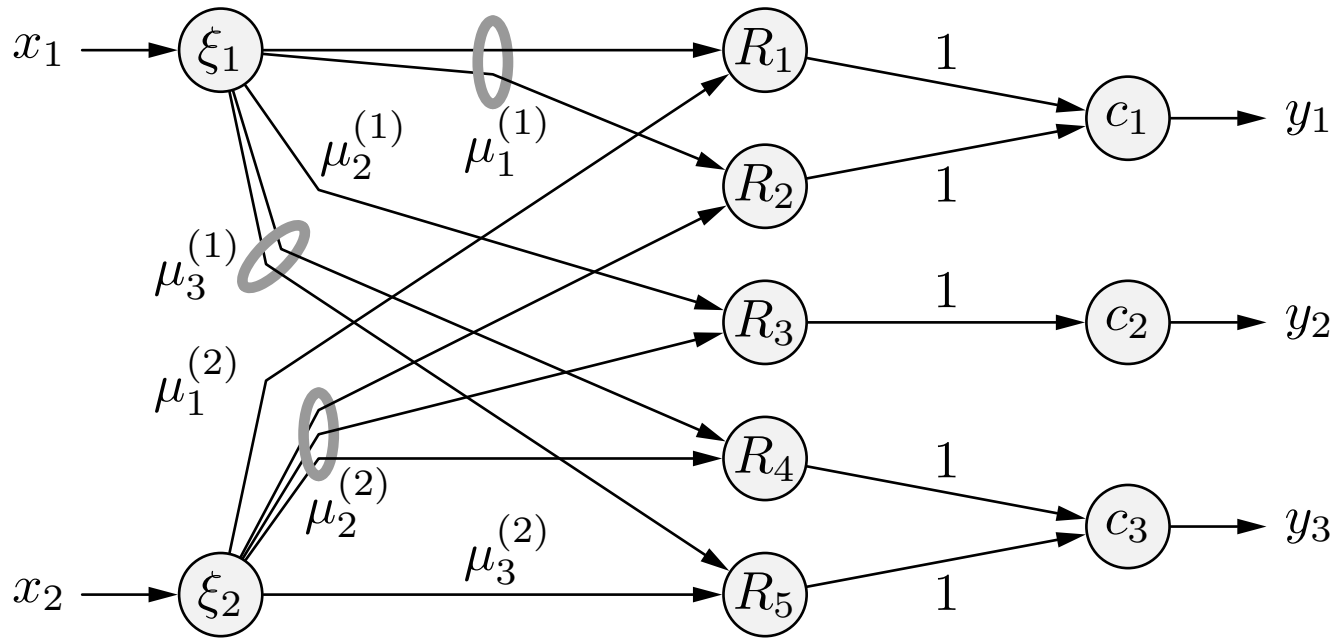
R_2: **if** x_1 is $\mu_1^{(1)}$ **and** x_2 is $\mu_2^{(2)}$, **then** y is ν_1

R_3: **if** x_1 is $\mu_2^{(1)}$ **and** x_2 is $\mu_2^{(2)}$, **then** y is ν_2

R_4: **if** x_1 is $\mu_3^{(1)}$ **and** x_2 is $\mu_2^{(2)}$, **then** y is ν_3

R_5: **if** x_1 is $\mu_3^{(1)}$ **and** x_2 is $\mu_3^{(2)}$, **then** y is ν_3

Neuro-Fuzzy Classification (NEFCLASS)



This NEFCLASS network represents fuzzy rules that predict classes:

R_1: **if** x_1 is $\mu_1^{(1)}$ **and** x_2 is $\mu_1^{(2)}$, **then** class c_1

R_2: **if** x_1 is $\mu_1^{(1)}$ **and** x_2 is $\mu_2^{(2)}$, **then** class c_1

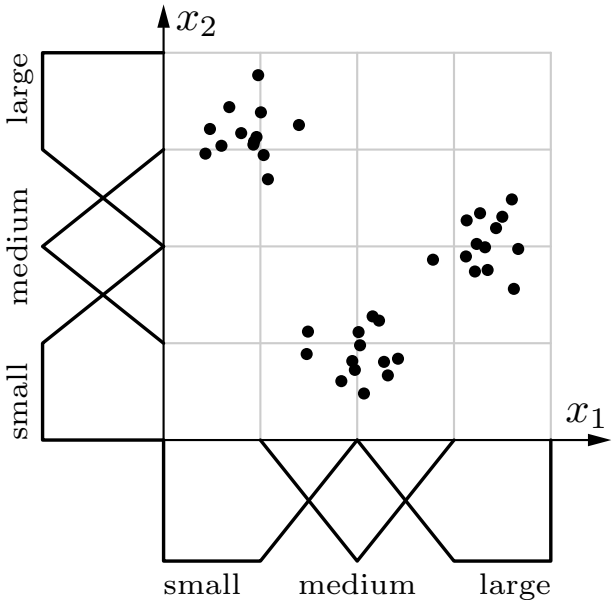
R_3: **if** x_1 is $\mu_2^{(1)}$ **and** x_2 is $\mu_2^{(2)}$, **then** class c_2

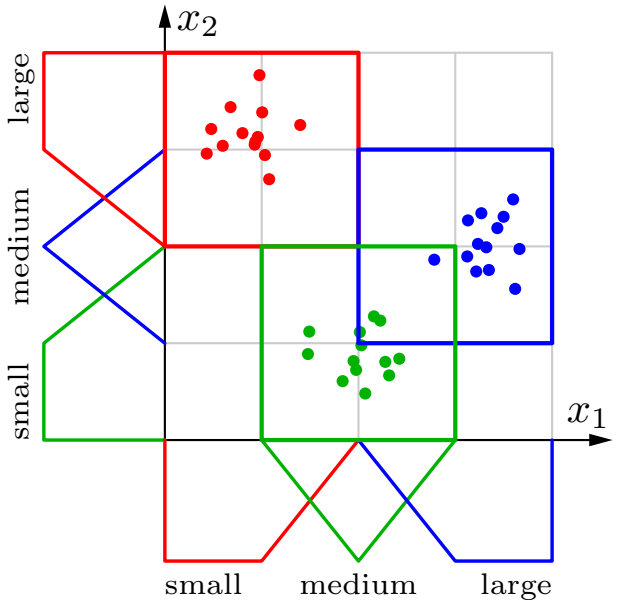
R_4: **if** x_1 is $\mu_3^{(1)}$ **and** x_2 is $\mu_2^{(2)}$, **then** class c_3

R_5: **if** x_1 is $\mu_3^{(1)}$ **and** x_2 is $\mu_3^{(2)}$, **then** class c_3

NEFCLASS: Initializing the Fuzzy Partitions

- NEFCLASS is based on modified Wang–Mendel procedure. [Nauck 1997]
- NEFCLASS first fuzzy partitions the domain of each variable,
  usually with a given number of equally sized triangular fuzzy sets;
  the boundary fuzzy sets are "shouldered" (membership 1 to the boundary).
- Based on the initial fuzzy partitions, the initial rule base is selected.





NEFCLASS: Initializing the Rule Base

$\mathcal{A} := \emptyset$; (∗ initialize the antecedent set ∗)
for each training pattern p **do begin** (∗ traverse the training patterns ∗)
  find rule antecedent A such that $A(p)$ is maximal;
  if $A \notin \mathcal{A}$ **then** (∗ if this is a new antecedent, ∗)
    $\mathcal{A} := \mathcal{A} \cup \{A\}$; (∗ add it to the antecedent set, ∗)
end (∗ that is, collect needed antecedents ∗)

$\mathcal{R} := \emptyset$; (∗ initialize the rule base ∗)
for each antecedent $A \in \mathcal{A}$ **do begin** (∗ traverse the antecedents ∗)
  find best consequent C for antecedent A; (∗ e.g. most frequent class in ∗)
  create rule base candidate $R = (A, C)$; (∗ training patterns assigned to A ∗)
  determine performance of R;
  $\mathcal{R} := \mathcal{R} \cup \{R\}$; (∗ collect the created rules ∗)
end

return rule base $\mathcal{R}$; (∗ return the created rule base ∗)

Fuzzy rule bases may also be created from prior knowledge or using
fuzzy cluster analysis, fuzzy decision trees, evolutionary algorithms etc.

NEFCLASS: Selecting the Rule Base

- In order to reduce/limit the number of initial rules, their performace is evaluated.
- The performance of a rule R_i is computed as

$$\text{Perf}(R_i) = \frac{1}{m} \sum_{j=1}^{m} e_{ij}\, \tilde{a}_i(\vec{x}_j)$$

  where m is the number of training patterns and e_{ij} is an error indicator,

$$e_{ij} = \begin{cases} +1 & \text{if class}(\vec{x}_j) = \text{cons}(R_i), \\ -1 & \text{otherwise.} \end{cases}$$

- Sort the rules in the initial rule base by their performance.
- Choose either the best r rules or the best r/c rules per class,
  where c is the number of classes.
- The number r of rules in the rule base is either provided by a user
  or is automatically determined in such a way that all patterns are covered.

NEFCLASS: Computing the Error Signal

- Let $o_k^{(l)}$ be the desired output and $\text{out}_k^{(l)}$ the actual output of the k-th output neuron (class c_k).
- Fuzzy error (k-th output/class):

$$e_k^{(l)} = 1 - \gamma\left(\varepsilon_k^{(l)}\right),$$

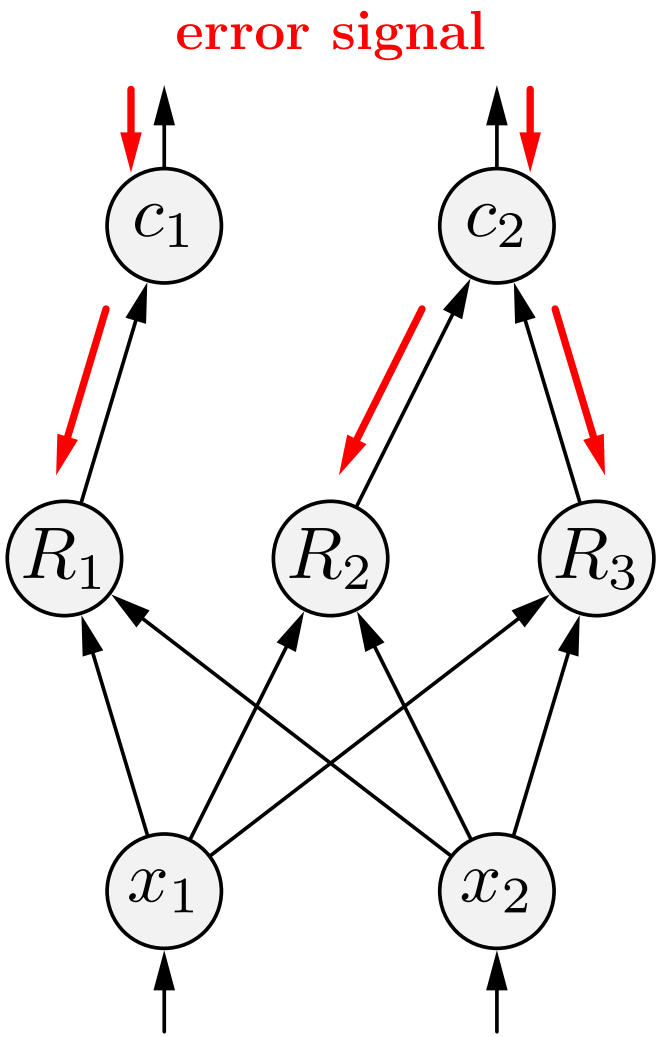
where $\varepsilon_k^{(l)} = o_k^{(l)} - \text{out}_k^{(l)}$ and

$$\gamma(z) = e^{-\beta z^2},$$

where $\beta > 0$ is a sensitivity parameter: larger β means larger error tolerance.

- Error signal (k-th output/class):

$$\delta_k^{(l)} = \text{sgn}\left(\varepsilon_k^{(l)}\right)\, e_k^{(l)}.$$



NEFCLASS: Computing the Error Signal

- Rule error signal (rule R_i for c_k):

$$\delta_{R_i}^{(l)} = \mathrm{out}_{R_i}^{(l)} \left(1 - \mathrm{out}_{R_i}^{(l)}\right) \delta_k^{(l)},$$

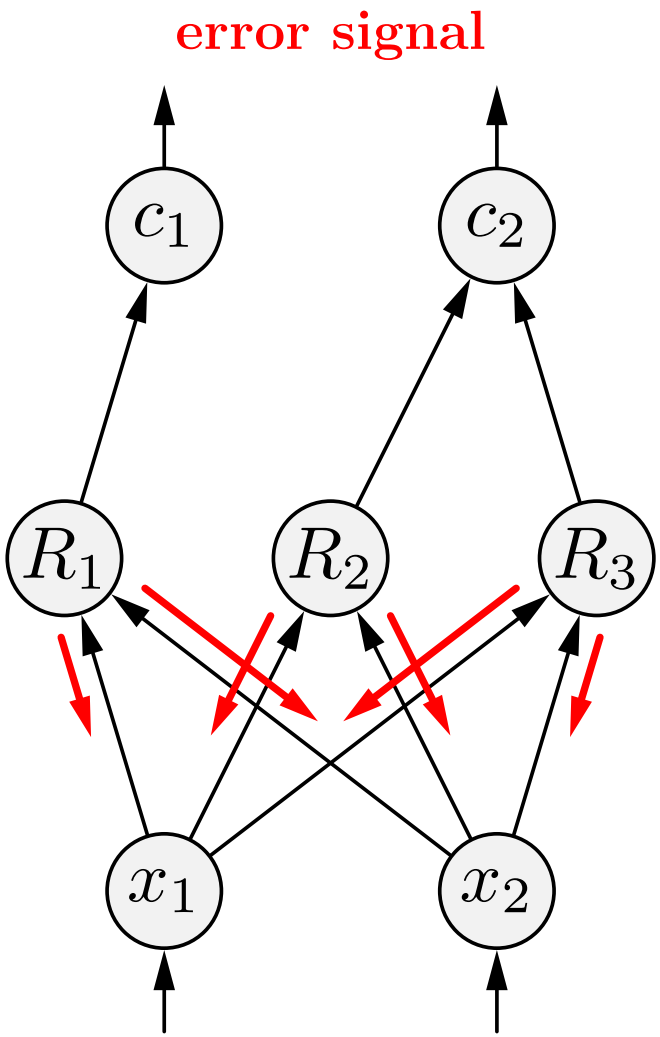
(the factor "out$(1 - \mathrm{out})$" is chosen in analogy to multi-layer perceptrons).

- Find input variable x_j such that

$$\mu_i^{(j)}\left(\imath_j^{(l)}\right) = \tilde{a}_i\left(\vec{\imath}^{(l)}\right) = \min_{\nu=1}^{d} \mu_i^{(\nu)}\left(\imath_\nu^{(l)}\right),$$

where d is the number of inputs (find antecedent term of R_i giving the smallest membership degree, which yields the rule activation).

- Adapt parameters of the fuzzy set $\mu_i^{(j)}$ (see next slide for details).



NEFCLASS: Training the Fuzzy Sets

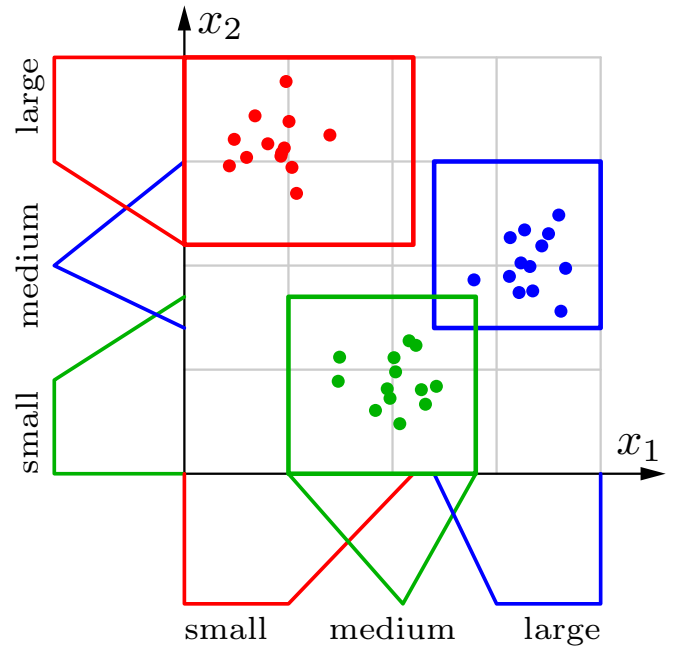
- Triangular fuzzy set as an example:

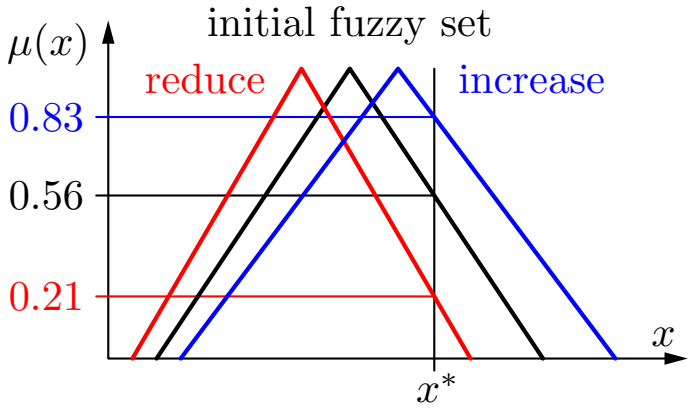
$$\mu_{a,b,c}(x) = \begin{cases} \frac{x-a}{b-a} & \text{if } x \in [a,b), \\ \frac{c-x}{c-b} & \text{if } x \in [b,c], \\ 0 & \text{otherwise.} \end{cases}$$

- Parameter changes (learning rate η):

$$\begin{aligned} \Delta b &= +\eta \cdot \delta_{R_i}^{(l)} \cdot (c-a) \cdot \operatorname{sgn}\left(\imath_j^{(l)} - b\right) \\ \Delta a &= -\eta \cdot \delta_{R_i}^{(l)} \cdot (c-a) + \Delta b \\ \Delta c &= +\eta \cdot \delta_{R_i}^{(l)} \cdot (c-a) + \Delta b \end{aligned}$$

- Heuristics: the fuzzy set to train is moved away from x^* (towards x^*) and its support is reduced (increased) in order to reduce (increase) the degree of membership of x^*.



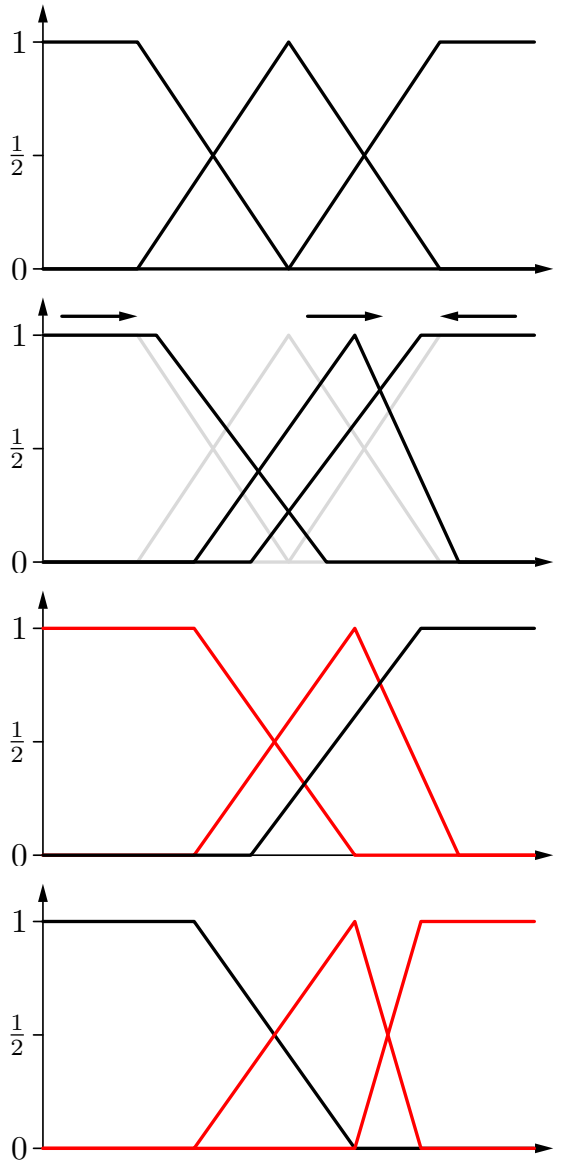


NEFCLASS: Restricted Training of Fuzzy Sets

When fuzzy sets of a fuzzy partition are trained, restrictions apply, so correction procedures are needed to

- ensure valid parameter values
- ensure non-empty intersections of neighboring fuzzy sets
- preserve relative positions
- preserve symmetry
- ensure partition of unity (membership degrees sum to 1 everywhere)

On the right: example of a correction of a fuzzy partition with three fuzzy sets.



NEFCLASS: Pruning the Rules

Objective: Remove variables, rules and fuzzy sets,
in order to improve interpretability and generalization ability.

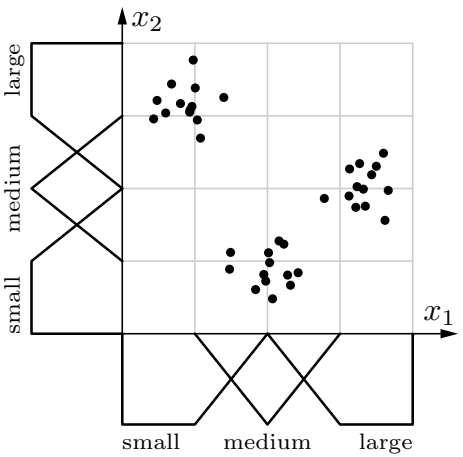
repeat
  select pruning method;
  repeat
    execute pruning step;
    train fuzzy sets;
    if no improvement
    then undo step;
  until there is no improvement;
until no further method;

1. Remove variables
   (correlation, information gain etc.)
2. Remove rules
   (performance of a rule)
3. Remove antecedent terms
   (satisfaction of a rule)
4. Remove fuzzy sets

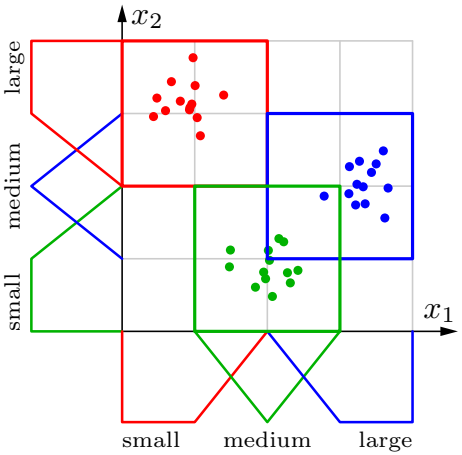
- After each pruning step the fuzzy sets need to be retrained,
  in order to obtain the optimal parameter values for the new structure.
- A pruning step that does not improve performance,
  the system is reverted to its state before the pruning step.

NEFCLASS: Full Procedure

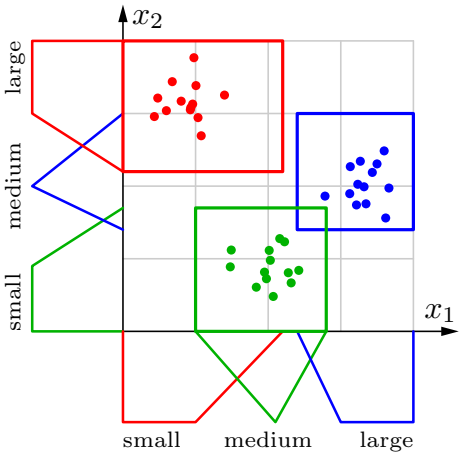
fuzzy partitions



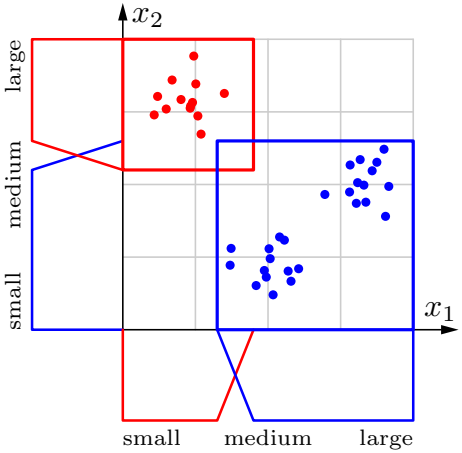
initial rule base

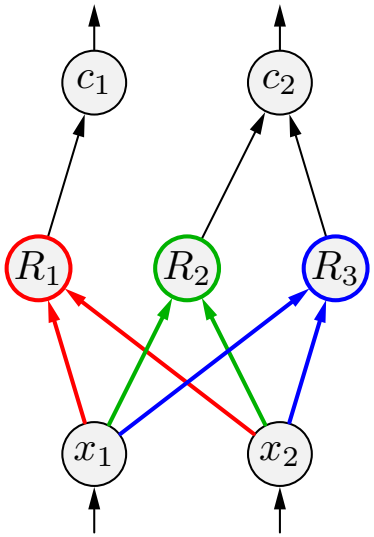


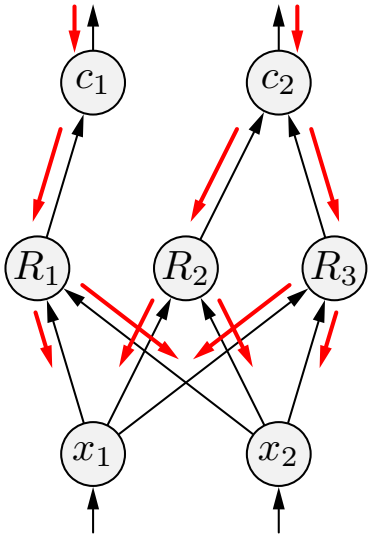
trained rule base

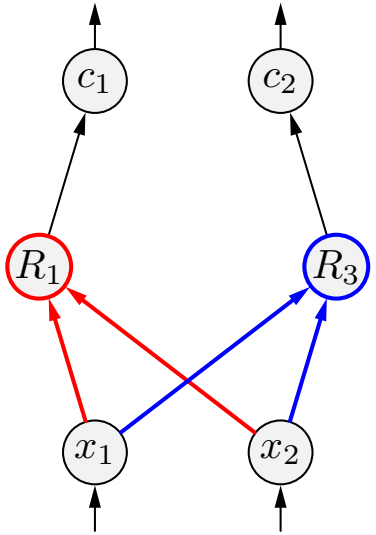


pruned rule base









Neuro-Fuzzy Systems in Finance

Stock Index Prediction (DAX) [Siekmann 1999]

- Prediction of the daily relative changes of the German stock index (Deutscher Aktienindex, DAX).
- Based on time series of stock indices and other quantities between 1986 and 1997.

Input Variables:

- DAX (Germany)
- Composite DAX (Germany)
- Dow Jones industrial index (USA)
- Nikkei index (Japan)
- Morgan–Stanley index Germany
- Morgan–Stanley index Europe
- German 3 month interest rate
- return Germany
- US treasure bonds
- price to income ratio
- exchange rate DM / US-$
- gold price

DAX Prediction: Example Rules

- trend rule: **if** DAX is decreasing **and** US-$ is decreasing
  then DAX prediction is decreasing
  with high certainty

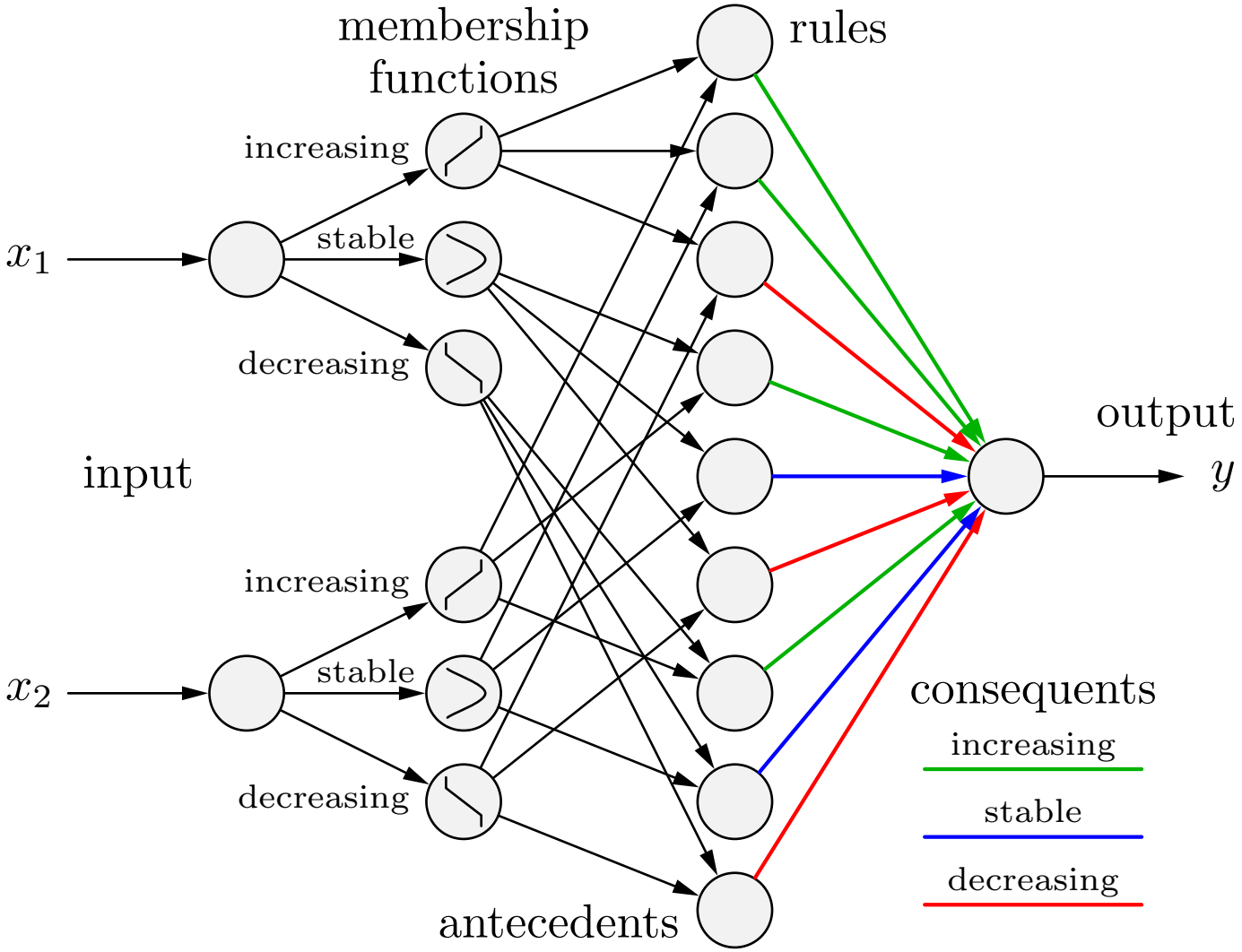
- turning point rule: **if** DAX is decreasing **and** US-$ is increasing
  then DAX prediction is increasing
  with low certainty

- delay rule: **if** DAX is stable **and** US-$ is decreasing
  then DAX prediction is decreasing
  with very high certainty

- general form: **if** x_1 is μ_1 **and** x_2 is μ_2 **and** $\ldots$ **and** x_n is μ_n
  then y is ν
  with certainty c

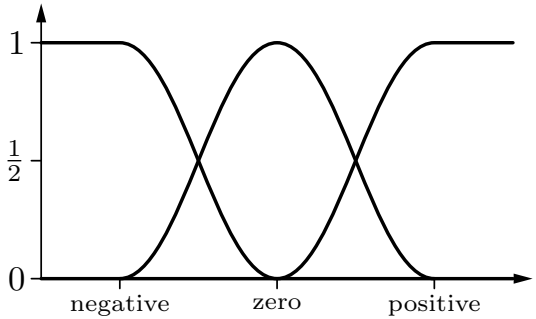
Initial rules may be provided by financial experts.

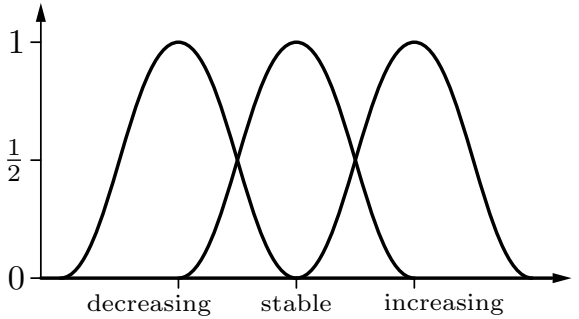
DAX Prediction: Architecture



DAX Prediction: From Rules to Neural Network

- Finding the membership values (evaluate membership functions):





- Evaluating the rules (computing the rule activation for r rules):

$$\forall j \in \{1, \ldots, r\}: \qquad \tilde{a}_j(x_1, \ldots, x_d) = \prod_{i=1}^{d} \mu_j^{(i)}(x_i).$$

- Accumulation of r rule activations, normalization:

$$y = \sum_{j=1}^{r} w_j \, \frac{c_j \, \tilde{a}_j(x_1, \ldots, x_d)}{\sum_{k=1}^{r} c_k \, \tilde{a}_k(x_1, \ldots, x_d)}, \qquad \text{where} \qquad \sum_{j=1}^{r} w_j = 1.$$

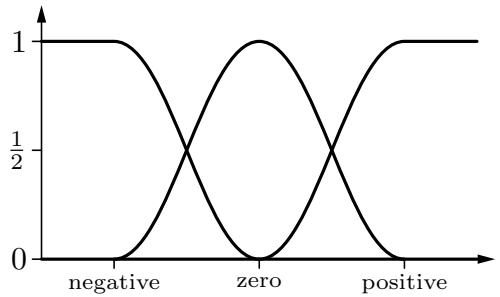
DAX Prediction: Training the Network

- Membership degrees of different inputs share their parameters, e.g.

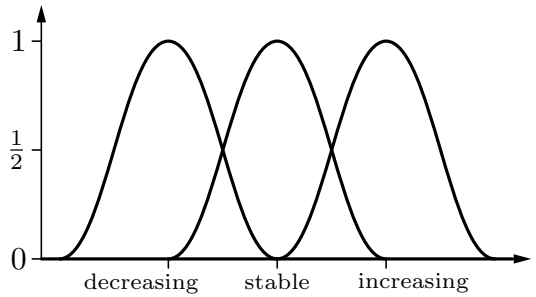
$$\mu_{\text{stable}}^{(\text{DAX})} = \mu_{\text{stable}}^{(\text{Composite DAX})}$$

Advantage: number of free parameters is reduced.

- Membership functions of the same input variable must not "pass each other", but must preserve their original order:



$\mu_{\text{negative}} < \mu_{\text{zero}} < \mu_{\text{positive}}$



$\mu_{\text{decreasing}} < \mu_{\text{stable}} < \mu_{\text{increasing}}$

Advantage: optimized rule base remains interpretable.

- The parameters of the fuzzy sets, the rule certainties, and the rule weights are optimized with a backpropagation approach.

- Pruning methods are employed to simplify the rules and the rule base.

Neuro-Fuzzy Systems: Summary

- Neuro-fuzzy systems can be useful for **discovering knowledge** in the form of rules and rule bases.
- The fact that they are **interpretable**, allows for plausibility checks and improves acceptance.
- Neuro-fuzzy systems exploit tolerances in the underlying system in order to find near-optimal solutions.
- Training procedures for neuro-fuzzy systems have to be able to cope with restrictions in order to preserve the semantics of the original model.
- **No (fully) automatic model generation.**
  $\Rightarrow$ A user has to work and interact with the system.
- Simple training methods support **exploratory data analysis**.