Neuronale Netzwerke

VL Algorithmisches Lernen, Teil 2b

Norman Hendrich

University of Hamburg
MIN Faculty, Dept. of Informatics
Vogt-Kölln-Str. 30, D-22527 Hamburg
hendrich@informatik.uni-hamburg.de

27/05/2009

Terminplanung: Part 2

- 27/05/2009 perceptron, backpropagation
- 28/05/2009 associative memory, self-organizing maps

- 03-04/06/2009 holidays
- 10-11/06/2009 dimensionality reduction

Outline

Remember: Eight major aspects of a PDP model

- a *set of processing units*
- the current *state of activation*
- an *output function* for each unit
- a *pattern of connectivity* between units
- a *propagation rule* for propagatin patterns of activities through the network of connectivities
- an *activation rule* for combining the inputs impinging on a unit
- a *learning rule* whereby patterns of connectivity are modified by experience
- the *environment* within the system must operate

(Rumelhard, McClelland, et.al. 1986)

Remember: McCulloch-Pitts model

- binary-valued threshold neurons with unweighted connections
- McCulloch-Pitts networks can compute any Boolean function
- but very similar to standard logic-gates

- network must be designed for a given computation
- no free parameters for adaptation
- learning only by modification of the network topology

- today, we will study weighted networks

The *classical* Perceptron

- proposed by Rosenblatt (1958)
- threshold elements with real-valued weights
- originally, assuming stochastic connections
- and a special learning algorithm
- a model for pattern recognition problems
- and the human visual processing

- very popular in the 1960s
- comprehensive critical analysis by Minsky and Papert (1969)
- more details and proofs: Rojas, chapter 3

The Perceptron

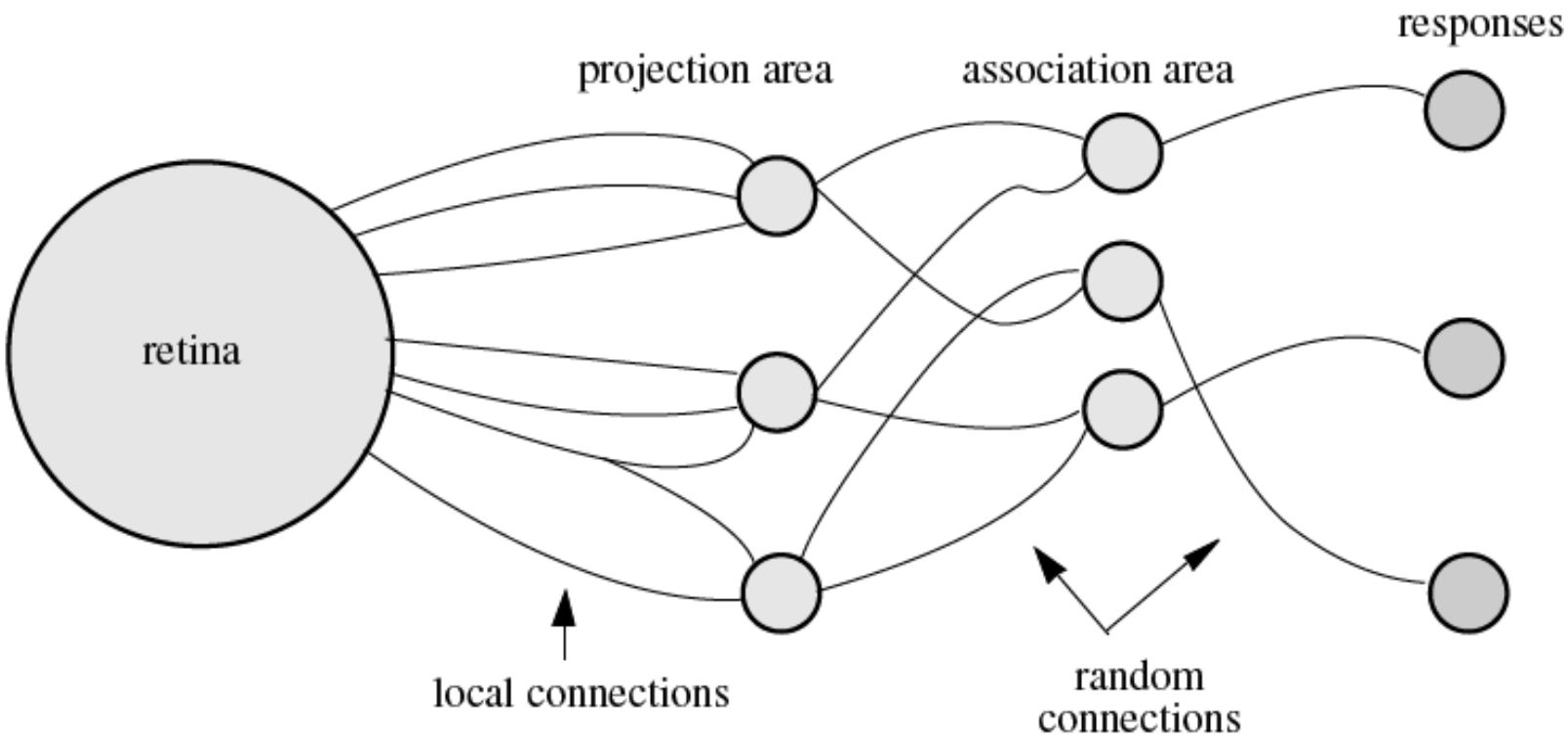
a *binary classifier* that maps a real-valued vector x of inputs to a binary output value $f(x)$

- binary output function with threshold θ
- similar to the McCulloch-Pitts neuron
- but weighted real-valued inputs

  $x_i(t+1) = 1$ if $(\sum_j w_{ij} \cdot x_j(t) \geq \theta)$, 0 else

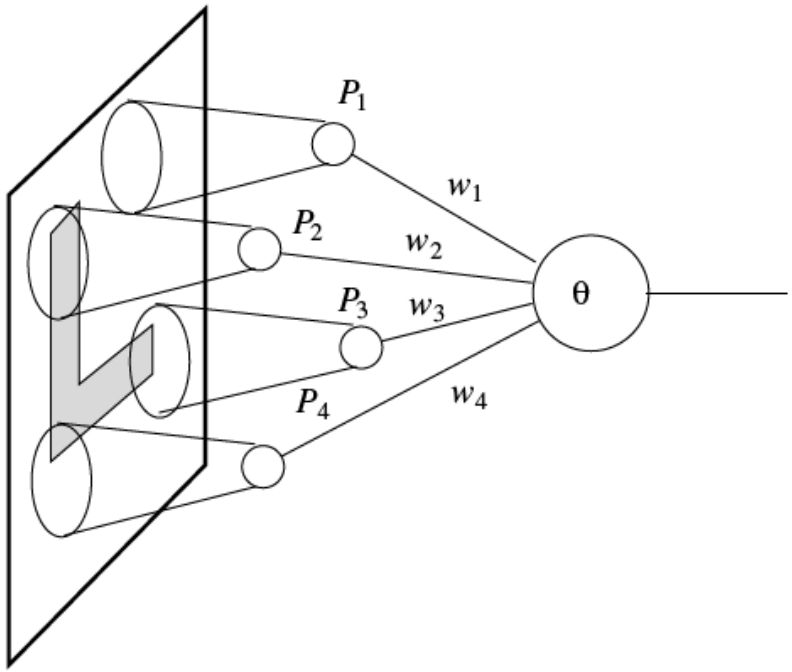
- what are the computational limits of this model?

Perceptron motivation



- abstract model of the retina and local preprocessing
- the association area: feature detection (perceptrons)
- active outputs indicate matched patterns

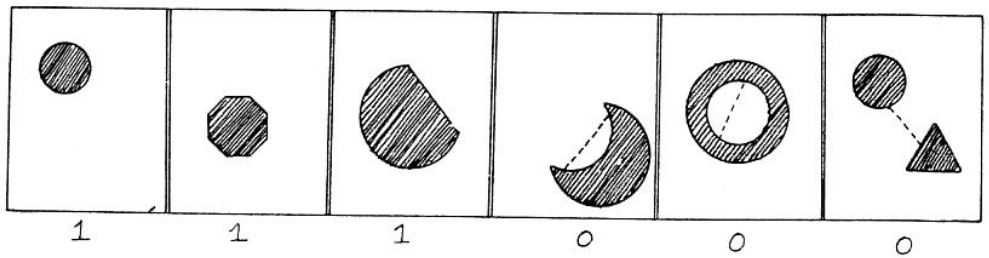
Predicate functions



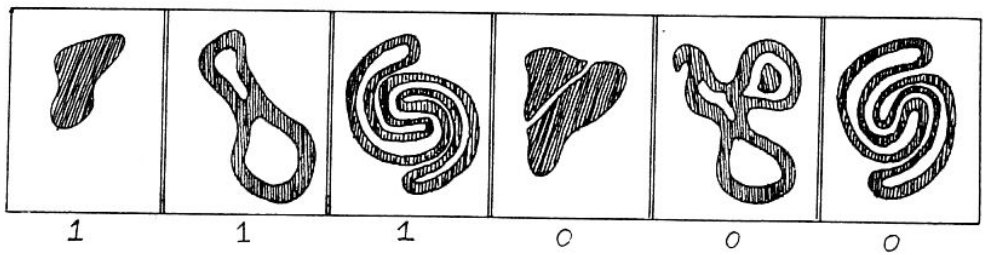
- arbitrary *predicate functions* P, possibly non-linear and complex
- but with limited *receptive fields*
- perceptron applies linear weighting and threshold θ

Example predicates

$$\psi_{\text{CONVEX}}(X) = \begin{cases} 1 \text{ if } X \text{ is a convex figure,} \\ 0 \text{ if } X \text{ is not a convex figure;} \end{cases}$$



$$\psi_{\text{CONNECTED}}(X) = \begin{cases} 1 \text{ if } X \text{ is a connected figure,} \\ 0 \text{ otherwise.} \end{cases}$$



(Minsky and Papert 1969)

Example task: detection of connected figures?

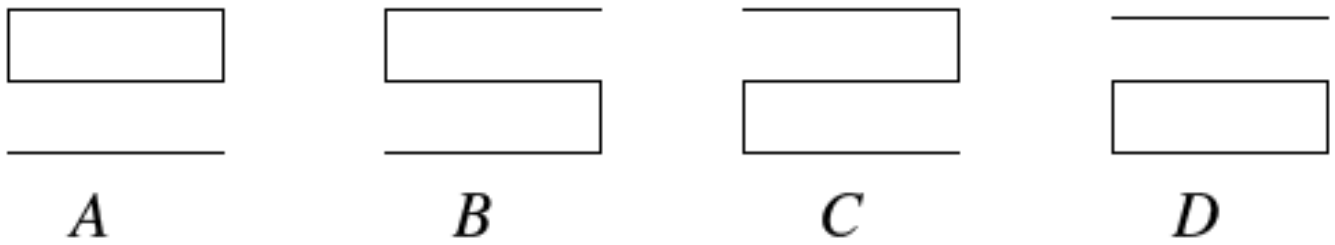
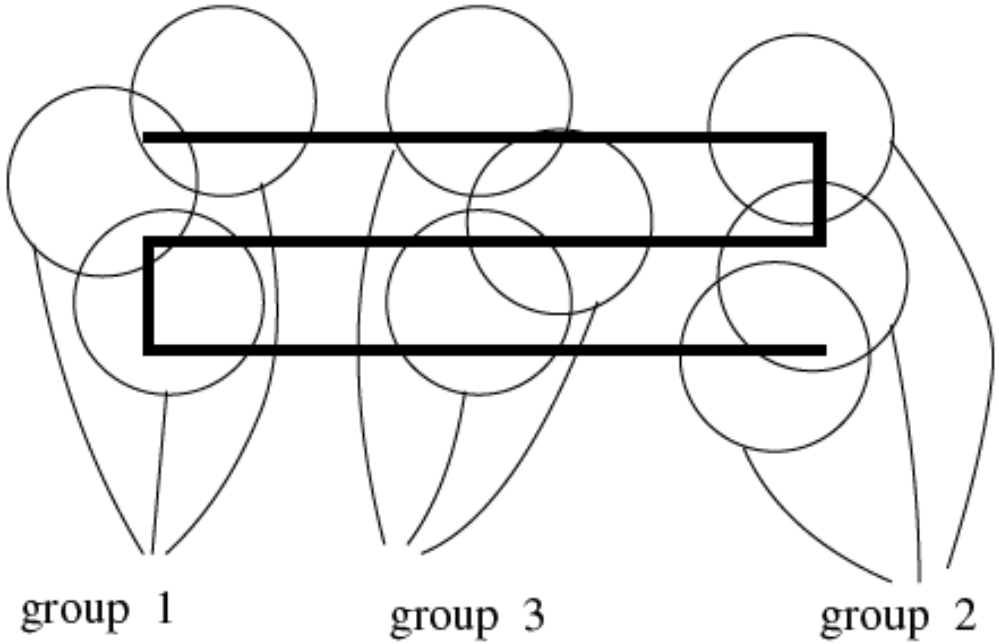


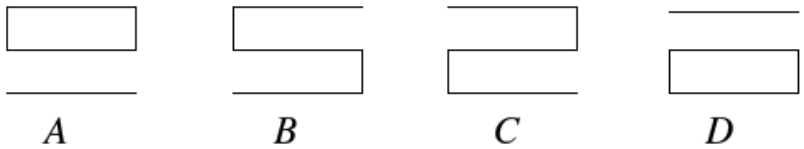
Figure 5.1

Example task: detection of connected figures?



- three receptive fields, combined by a predicate function each
- group outputs fed to a single perceptron
- can the perceptron detect *connectedness* of the line?

Can we detect *connected* figures?



Proposition: No diameter limited perceptron can decide whether a geometric figure is connected or not.

- assume the perceptron can decide this
- construct demo patterns A B C D
- scale until figure fills the left/middle/right receptive fields

- assume that figures A, B, C are classified correctly
- contradiction: figure D is classified incorrectly

(Minsky and Papert, 1969)

Proof

All predicates are connected to a threshold element through weighted edges which we denote by the letter w with an index. The threshold element decides whether a figure is connected or not by performing the computation

$$S = \sum_{P_i \in \text{group 1}} w_{1i} P_i + \sum_{P_i \in \text{group 2}} w_{2i} P_i + \sum_{P_i \in \text{group 3}} w_{3i} P_i - \theta \geq 0.$$

If S is positive the figure is recognized as connected, as is the case, for example, in Figure 3.3.

If the disconnected pattern A is analyzed, then we should have $S < 0$. Pattern A can be transformed into pattern B without affecting the output of the predicates of group 3, which do not recognize the difference since their receptive fields do not cover the sides of the figures. The predicates of group 2 adjust their outputs by $\Delta_2 S$ so that now

$$S + \Delta_2 S \geq 0 \Rightarrow \Delta_2 S \geq -S.$$

If pattern A is transformed into pattern C, the predicates of group 1 adjust their outputs so that the threshold element receives a net excitation, i.e.,

$$S + \Delta_1 S \geq 0 \Rightarrow \Delta_1 S \geq -S.$$

However, if pattern A is transformed into pattern D, the predicates of group 1 cannot distinguish this case from the one for figure C and the predicates of group 2 cannot distinguish this case from the one for figure B. Since the predicates of group 3 do not change their output we have

$\Delta S = \Delta_2 S + \Delta_1 S \geq -2S, \quad S + \Delta S \geq -S \geq 0$, so D connected

Logical functions with Perceptrons

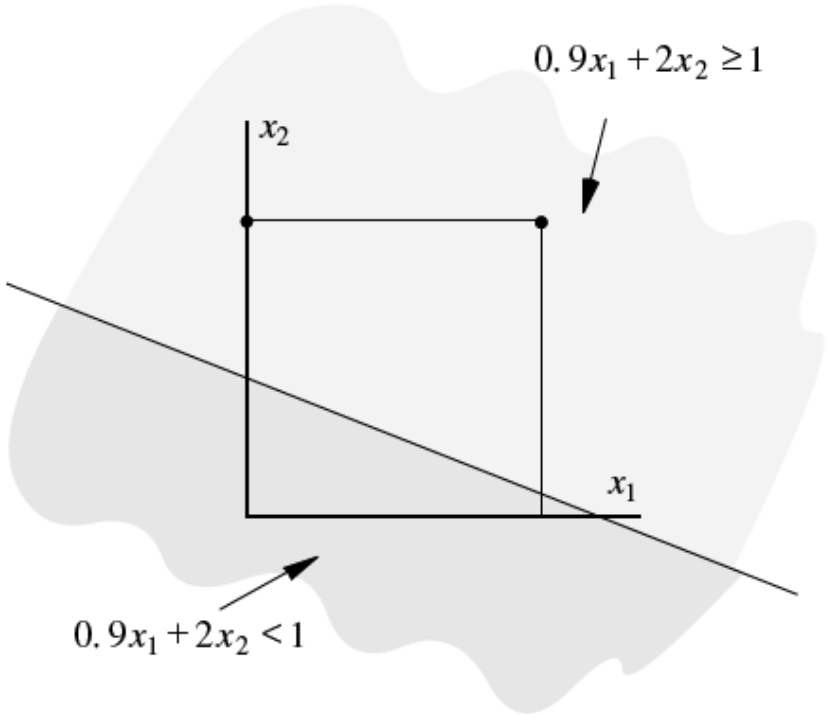
- single McCulloch-Pitts neurons implement monotonous functions
- McCulloch-Pitts networks can implement any Boolean function

- what can a single Perceptron do?

Geometric interpretation:

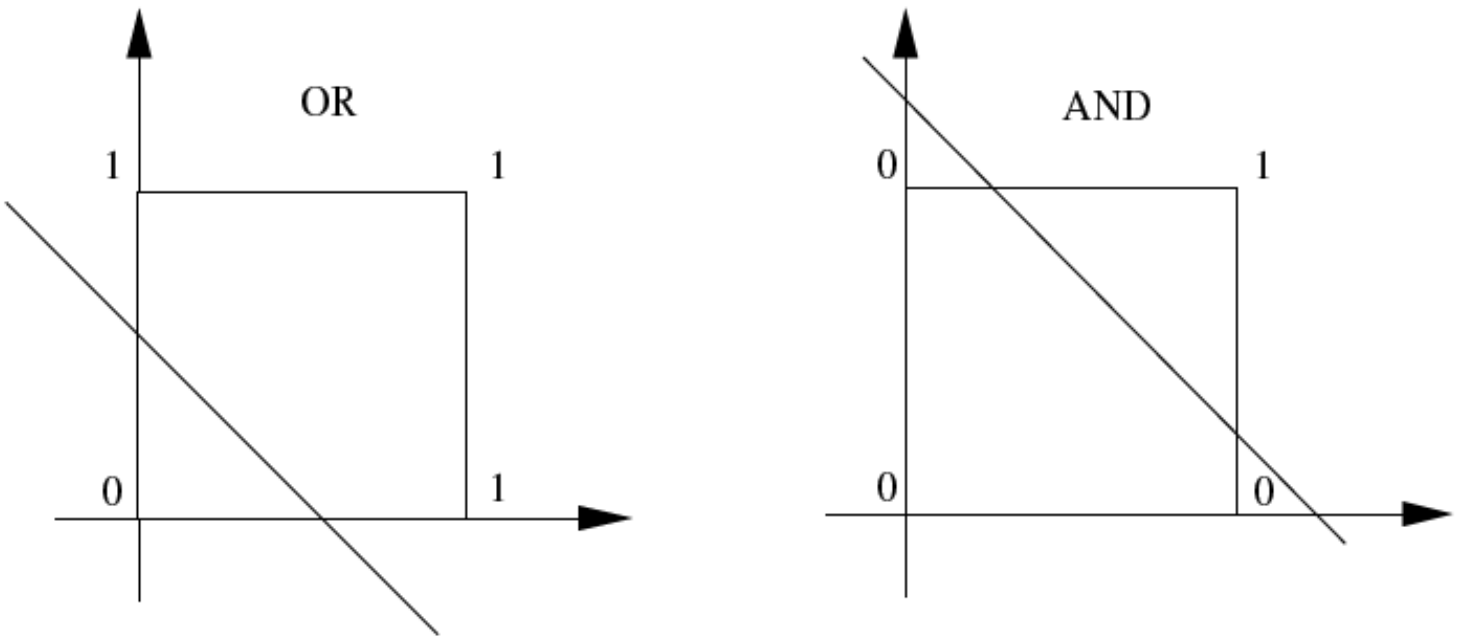
- Perceptron implements a thresholded decision
- it *separates* its input space into two *half-spaces*
- for points in one half the result is 0, and in the other half 1

Geometric interpretation



- basic visualization in the 2D-case: dimensions x_1 and x_2
- a line separating the upper and lower half of the plane

Example separations: OR and AND functions



- Perceptron computes separable functions
- one line/plane/hyperplane separates the 0- and the 1-values

Absolute linear separability

Definition 3. *Two sets A and B of points in an n-dimensional space are called absolutely linearly separable if $n+1$ real numbers $w_1, \ldots, w_{n+1}$ exist such that every point $(x_1, x_2, \ldots, x_n) \in A$ satisfies $\sum_{i=1}^{n} w_i x_i > w_{n+1}$ and every point $(x_1, x_2, \ldots, x_n) \in B$ satisfies $\sum_{i=1}^{n} w_i x_i < w_{n+1}$*

If a perceptron with threshold zero can linearly separate two finite sets of input vectors, then only a small adjustment to its weights is needed to obtain an absolute linear separation. This is a direct corollary of the following proposition.

Proposition 7. *Two finite sets of points, A and B, in n-dimensional space which are linearly separable are also absolutely linearly separable.*

Open half space

Definition 4. *The open (closed) positive half-space associated with the n-dimensional weight vector $\mathbf{w}$ is the set of all points $\mathbf{x} \in \mathbb{R}^n$ for which $\mathbf{w} \cdot \mathbf{x} > 0$ $(\mathbf{w} \cdot \mathbf{x} \geq 0)$. The open (closed) negative half-space associated with $\mathbf{w}$ is the set of all points $\mathbf{x} \in \mathbb{R}^n$ for which $\mathbf{w} \cdot \mathbf{x} < 0$ $(\mathbf{w} \cdot \mathbf{x} \leq \mathbf{0})$.*

Example separations: the 16 functions of 2 variables

x_1	x_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

- 2^{2^n} different Boolean with n variables
- table shows all possible functions for $n = 2$
- 14 functions can be computed with a single Perceptron
- but f_6 and f_9 cannot - the XOR and XNOR

Proof for the XOR problem

- XOR function cannot be separated with a single Perceptron

- must fulfil the following inequalities:
- $x_1 = 0$, $x_2 = 0$: $w_1 x_1 + w_2 x_2 = 0 \implies 0 < \theta$
- $x_1 = 1$, $x_2 = 0$: $w_1 x_1 + w_2 x_2 = w_1 \implies w_1 \geq \theta$
- $x_1 = 0$, $x_2 = 1$: $w_1 x_1 + w_2 x_2 = w_2 \implies w_2 \geq \theta$
- $x_1 = 1$, $x_2 = 1$: $w_1 x_1 + w_2 x_2 = w_1 + w_2 \implies w_1 + w_2 < \theta$

- contradiction: $w_1 \geq \theta$ and $w_2 \geq \theta$ and positive, but $w_1 + w_2 < \theta$.

Error function in weight space

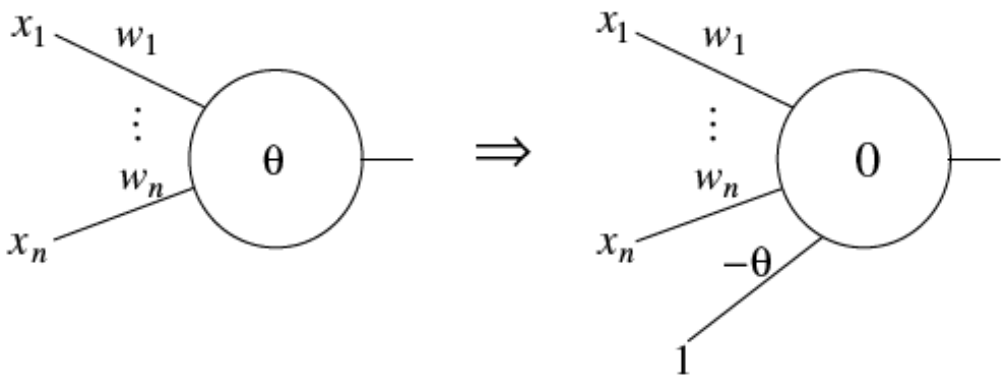
- learning should automatically find the weights
- to separate the given input patterns
- with $f_w(x) = 1$ for class A and $f_w(x) = 0$ for class B.

Define the *error function* as the number of false classifications, obtained using the current weight vector w:

$$E(w) = \sum_{x \in A} (1 - f_w(x)) + \sum_{x \in B} f_w(x)$$

- learning should minimize this error function
- at the global minimum, $E(w) = 0$, the problem is solved

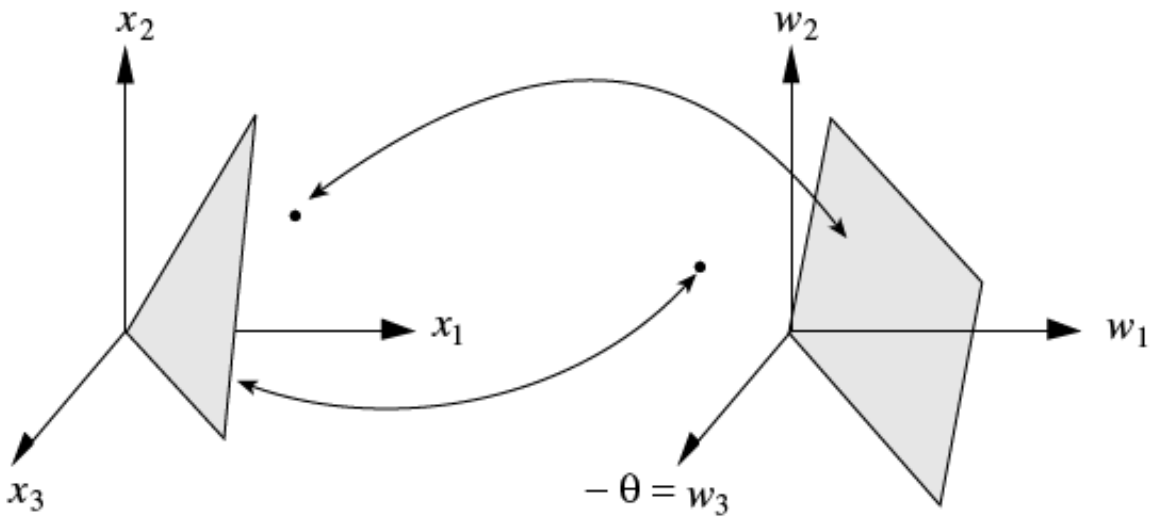
Extended weight vector



- proofs and implementation are often easier if threshold $\theta = 0$ (so that we can omit it)

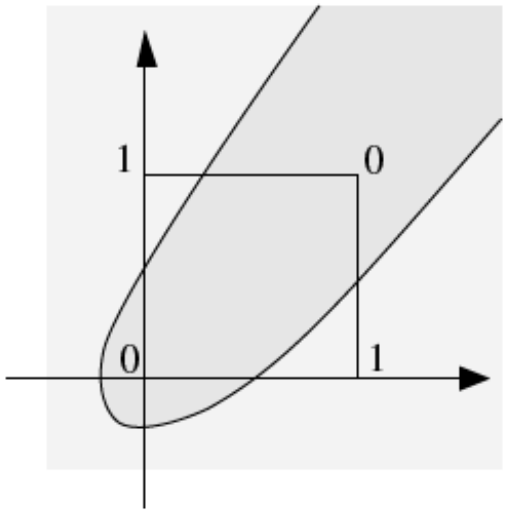
- introduce an extra weight, $w_{n+1} = -\theta$
- and an extra constant input, $x_{n+1} = 1$
- the so-called *extended weight vector* representation

The dual formulation



- the weighting calculation is $\sum_j w_j x_j \geq 0$
- symmetric in w and x
- classification uses inputs x as variables
- dual-problem: use w as the variables

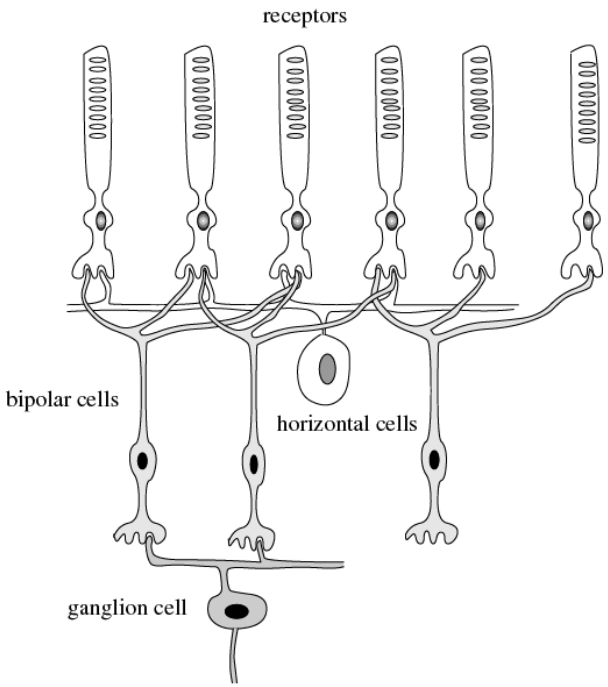
General decision curves



- original Perceptron allows complex *predicates*
- what happens with non-linear decision curves/surfaces?

- example shows a valid solution of the XOR function
- we will come back to this question later (*kernel-trick*)

Feature detection: structure of the retina



- retina known to use (non-linear) pre-processing
- reconsider the perceptron for visual pattern detection

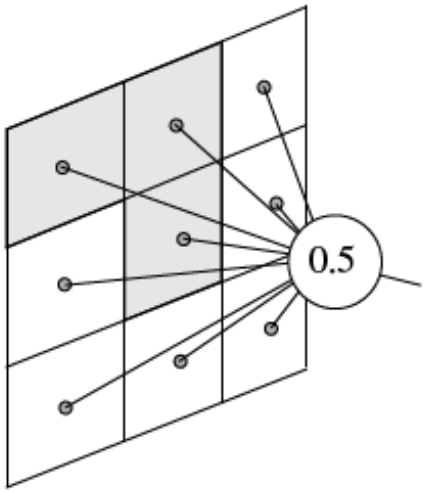
Low-level convolution operators

-1	-1	-1
-1	8	-1
-1	-1	-1

- compare: traditional image-processing / computer-vision
- basic convolution operators for low-level feature detection
- for example, Laplace 3x3 edge-detection operator

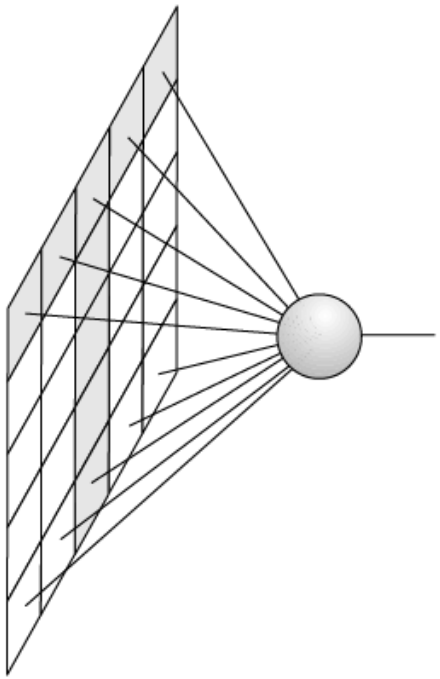
- separate neurons can compute those operators in parallel

Perceptron as an edge-detector

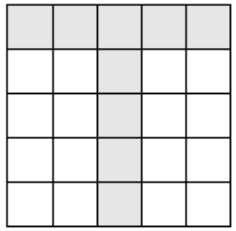


- assume binary or gray-scale images
- perceptron connected to input pixels works as feature detector
- e.g., 3x3 Laplace operator
- use one perceptron centered at every pixel
- massively parallel computation

Feature detection



pattern

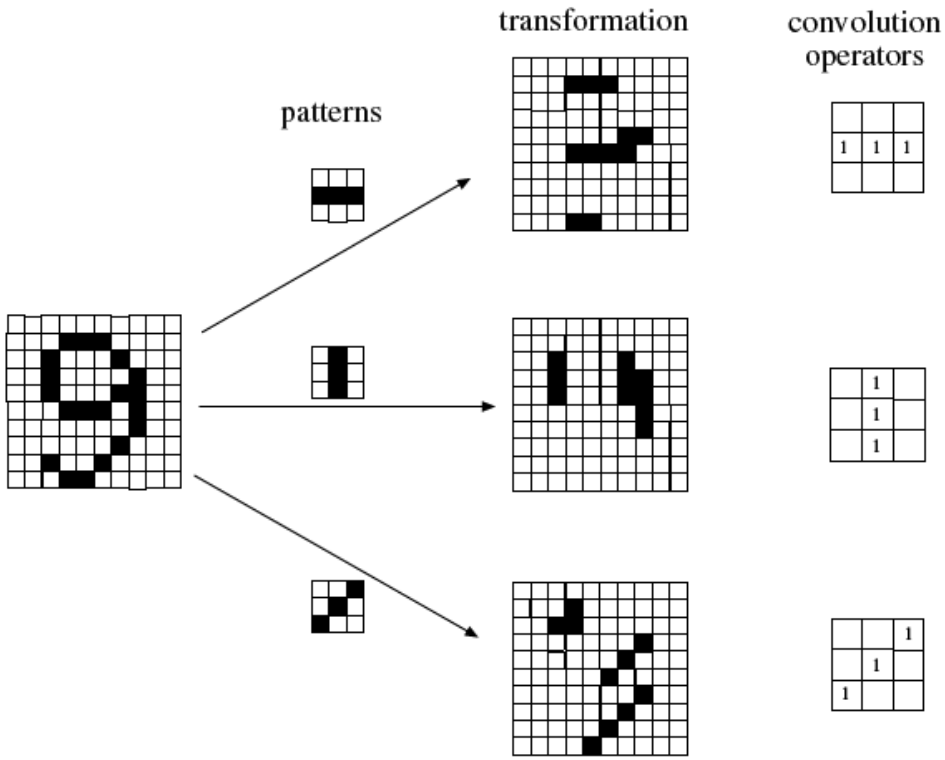


weights

1	1	1	1	1
−1	−1	1	−1	−1
−1	−1	1	−1	−1
−1	−1	1	−1	−1
−1	−1	1	−1	−1

- classification directly on the raw-pixels?
- not invariant against translation, rotation, etc.

Neocognitron network



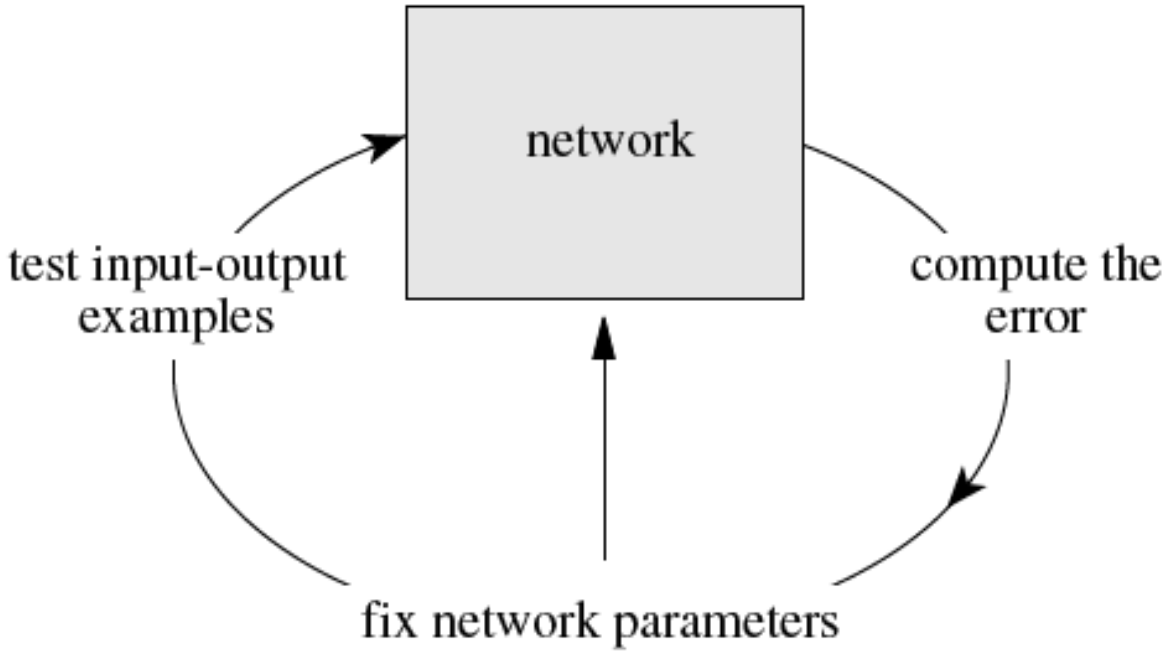
- use pre-processing to extract low-level features
- run classification on the features, not the raw pixels

Perceptron summary:

- binary neurons, internal state is $x = \sum_j w_{ij} x_j$
- threshold output function $o(x) = \frac{1}{2}(\mathrm{sgn}(x) - 1)$
- single-layer, fully connected to input neurons
- continous weights w_{ij}
- basic learning rule

- single Perceptron cannot solve XOR or connectedness
- non-linear predicate functions might help
- a model for low-level feature detection

Learning algorithms for neural networks

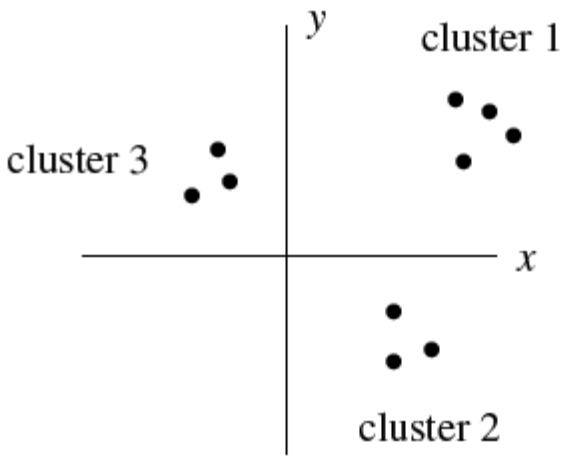


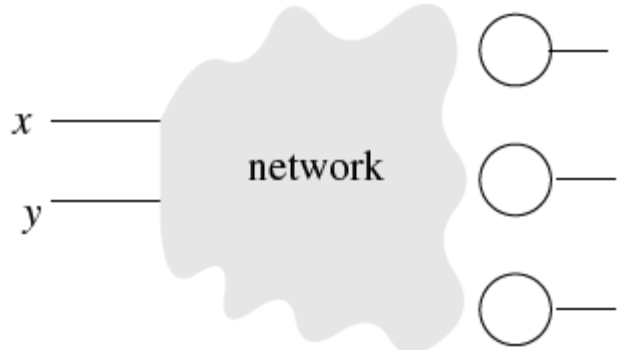
- network self-organizes to implement the desired behaviour
- correction-steps to improve the network error

Three models of learning

- *supervised learning* (corrective learning)
  - learning from *labelled examples*
  - provided by a knowledgable external supervisor
  - the correct outputs are known for all *training samples*
  - the type of learning usually assumed in NN studies
- *reinforcement learning*:
  - no labelled examples
  - environment provides a scalar feedback signal
  - combine *exploration* and *exploitation* to maximize the reward
- *unsupervised learning*:
  - no external feedback at all

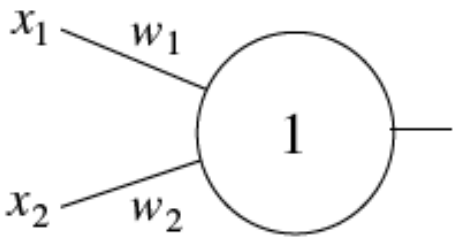
Classes of learning algorithms





- most general is *unsupervised learning*
- e.g., classify input-space clusters
- but we don't know how many or even which clusters

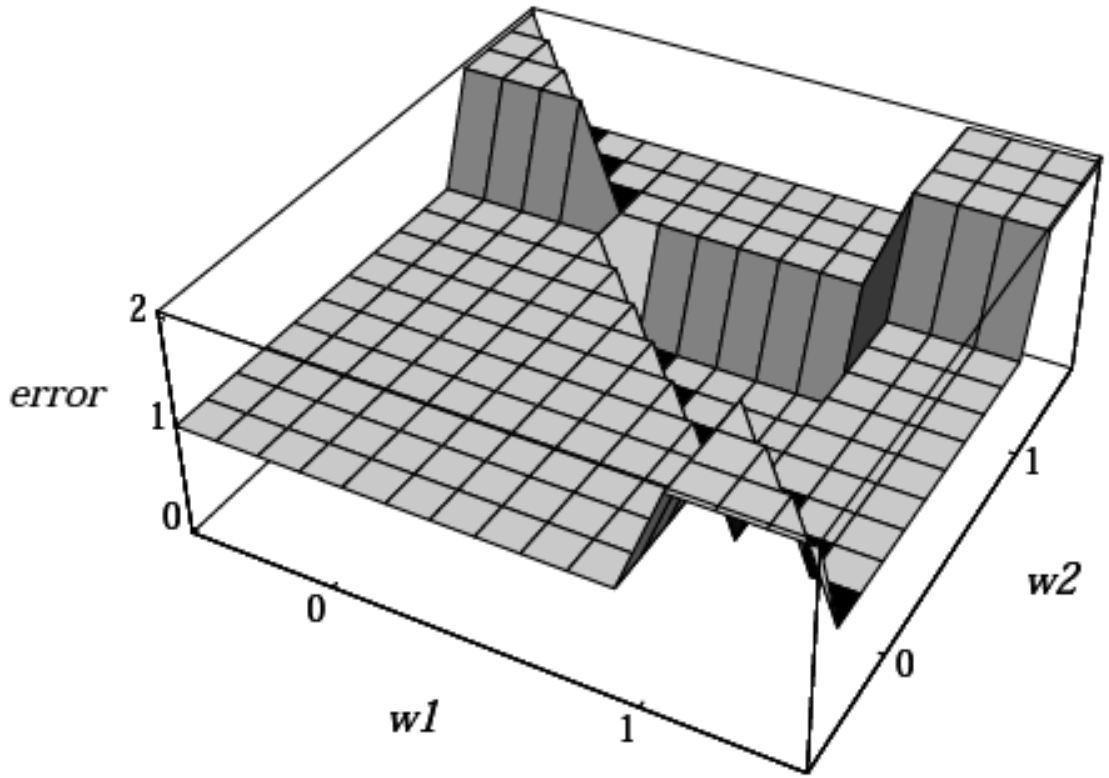
Perceptron learning example



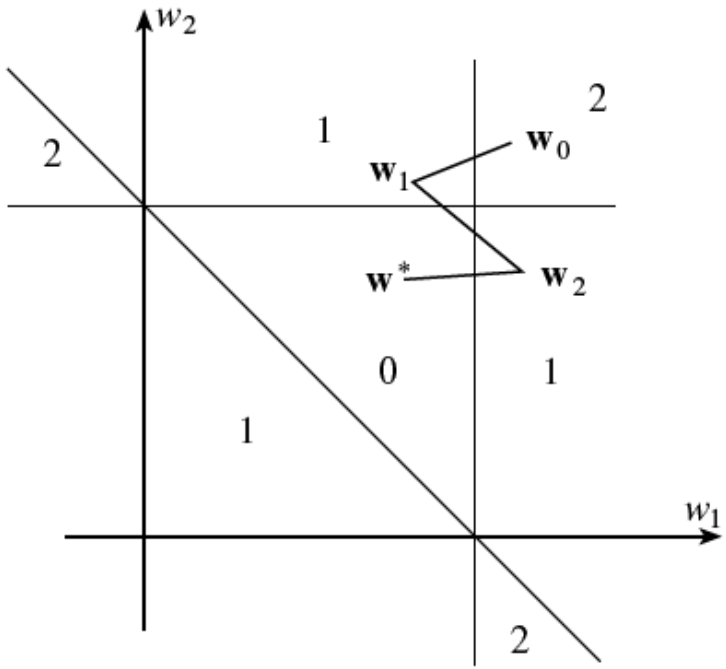
- assume a perceptron with constant threshold $\theta = 1$
- look for weights w_1 and w_2 to realize the logic AND function

- calculate and plot the error-function $E(w)$ for all weights
- learning algorithm should reach the solution 'triangle'

Error surface for the AND function

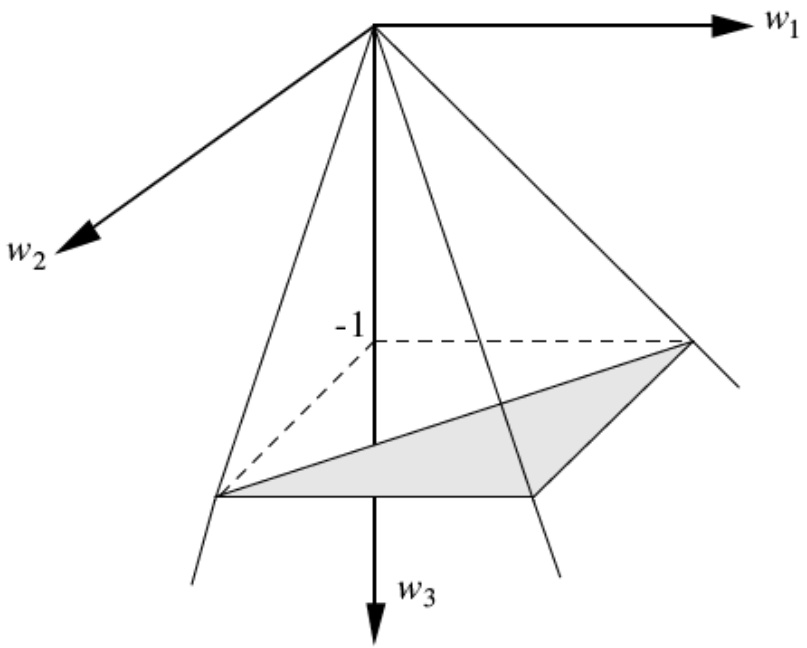


Error surface for the AND function, from above



- areas of the error-function in weight-space
- four steps of weight adjustments during learning
- final state w^* is a valid solution with error $E(w) = 0$

Solution polytope



- in general, solution area is a *polytope* in R^n
- at given threshold θ, a cut of the solution polytope

Perceptron training algorithm

- assume a given classification problem
- set of m training input patterns p_i and outputs $f(p_i)$
- try to minimize error-function $E(w)$ in weight-space

- standard algorithms starts with random weights
- pick a training pattern p_i randomly
- update the weights to improve classification for this pattern
- pick the next training pattern
- until $E(w) = 0$

Perceptron training algorithm

start: The weight vector $\mathbf{w}_0$ is generated randomly, set $t := 0$

test: A vector $\mathbf{x} \in P \cup N$ is selected randomly,

if $\mathbf{x} \in P$ and $\mathbf{w}_t \cdot \mathbf{x} > 0$ go to *test*,
if $\mathbf{x} \in P$ and $\mathbf{w}_t \cdot \mathbf{x} \leq 0$ go to *add*,
if $\mathbf{x} \in N$ and $\mathbf{w}_t \cdot \mathbf{x} < 0$ go to *test*,
if $\mathbf{x} \in N$ and $\mathbf{w}_t \cdot \mathbf{x} \geq 0$ go to *subtract*.

add: set $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{x}$ and $t := t + 1$, goto *test*

subtract: set $\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{x}$ and $t := t + 1$, goto *test*

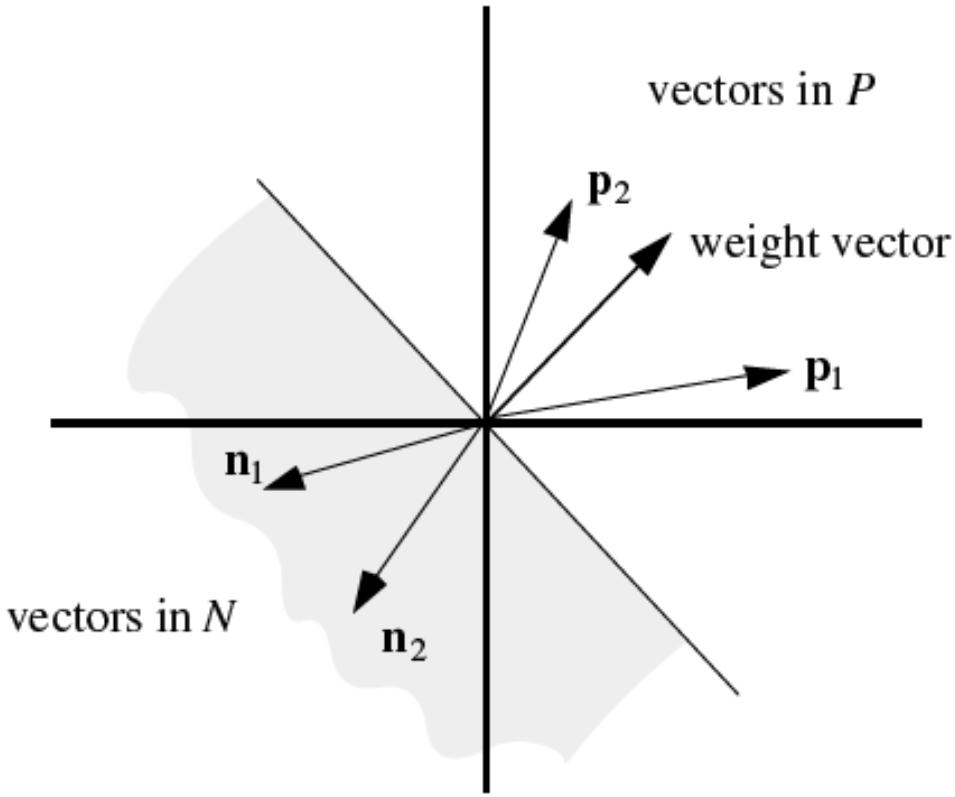
Perceptron training algorithm

- algorithm can stop once a first solution is found
- we will study a more robust variant later
  (the so-called *support-vector machine*)

- algorithm converges in finite number of steps
- iff the patterns are linearly separable
- proof: Rojas 4.2.2

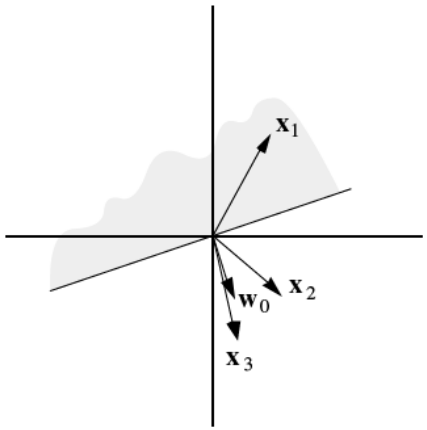
- no convergence if problem is not solvable
- learning a conflicting pattern can degrade network performance

Visualization in input and extended weight space

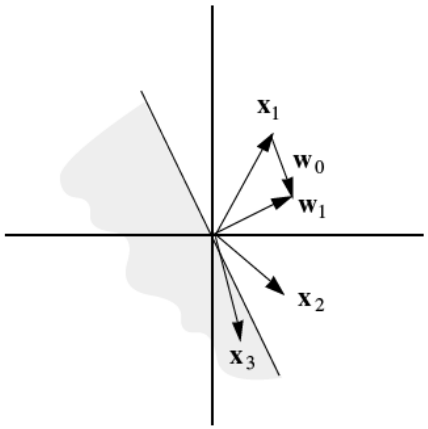


Behaviour of Perceptron learning

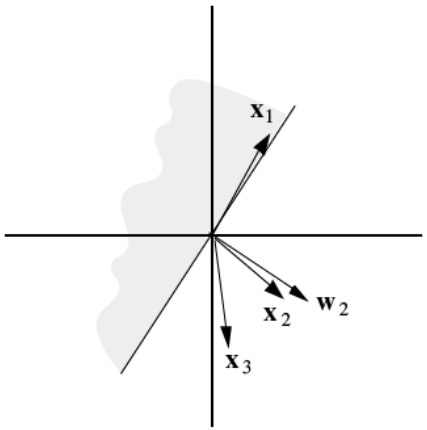
1) Initial configuration



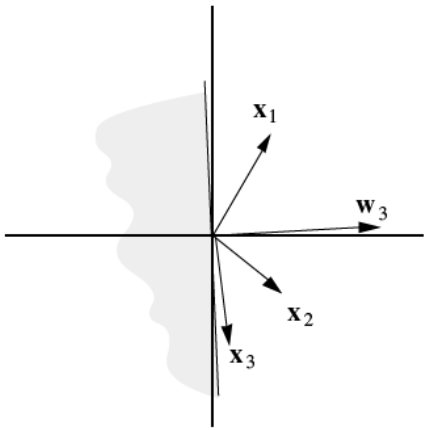
2) After correction with $\mathbf{x}_1$



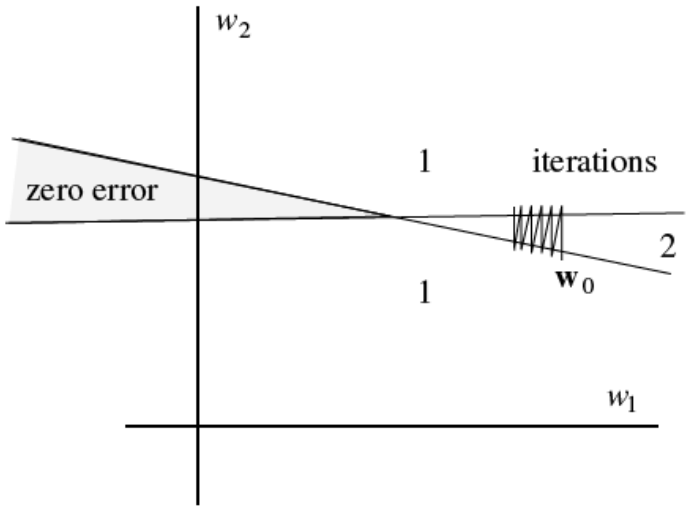
3) After correction with $\mathbf{x}_3$



4) After correction with $\mathbf{x}_1$



Worst-case behaviour



- search direction (in w space) according to last incorrect classified pattern
- but no information about the shape of the error function
- figure shows 'worst case' for Perceptron learning

Pocket training algorithm

- how to guarantee good behaviour if problem not separable?
- we want the best solution: least total error $E(w)$
- standard Perceptron algorithm won't converge

Pocket algorithm:

- keep a history of the learning process over a set of patterns
- keep previous best solution in your *pocket*
- restore previous best solution if $E(w)$ degrades

Pocket training algorithm

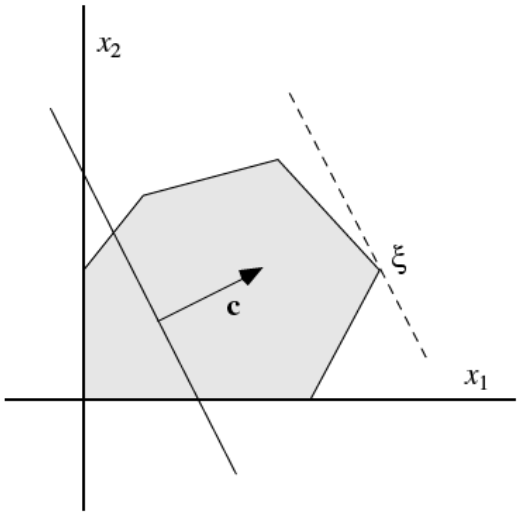
Algorithm 4.2.2 *Pocket algorithm*

start: Initialize the weight vector $\mathbf{w}$ randomly. Define a "stored" weight vector $\mathbf{w}_s = \mathbf{w}$. Set h_s, the history of $\mathbf{w}_s$, to zero.

iterate: Update $\mathbf{w}$ using a single iteration of the perceptron learning algorithm. Keep track of the number h of consecutively successfully tested vectors. If at any moment $h > h_s$, substitute $\mathbf{w}_s$ with $\mathbf{w}$ and h_s with h. Continue iterating.

- variant of the basic Perceptron learning
- keep previous best solution in your *pocket*
- restore previous solution if $E(w)$ degrades
- converges to the best possible solution

Linear programming



- solution area bounded by hyperplanes
- use *linear programming* (*simplex-algorithm*) to find optimal solution
- worst-case performance is exponential
- but usually, well-behaved

Multi-layer networks

- combine multiple threshold elements to larger networks
- can compute any Boolean function
- but we have no learning rule (yet)

Network architecture?

- 2-layer networks
- multi-layer feed-forward networks
- multi-layer networks with shortcut connections
- networks with feedback connections

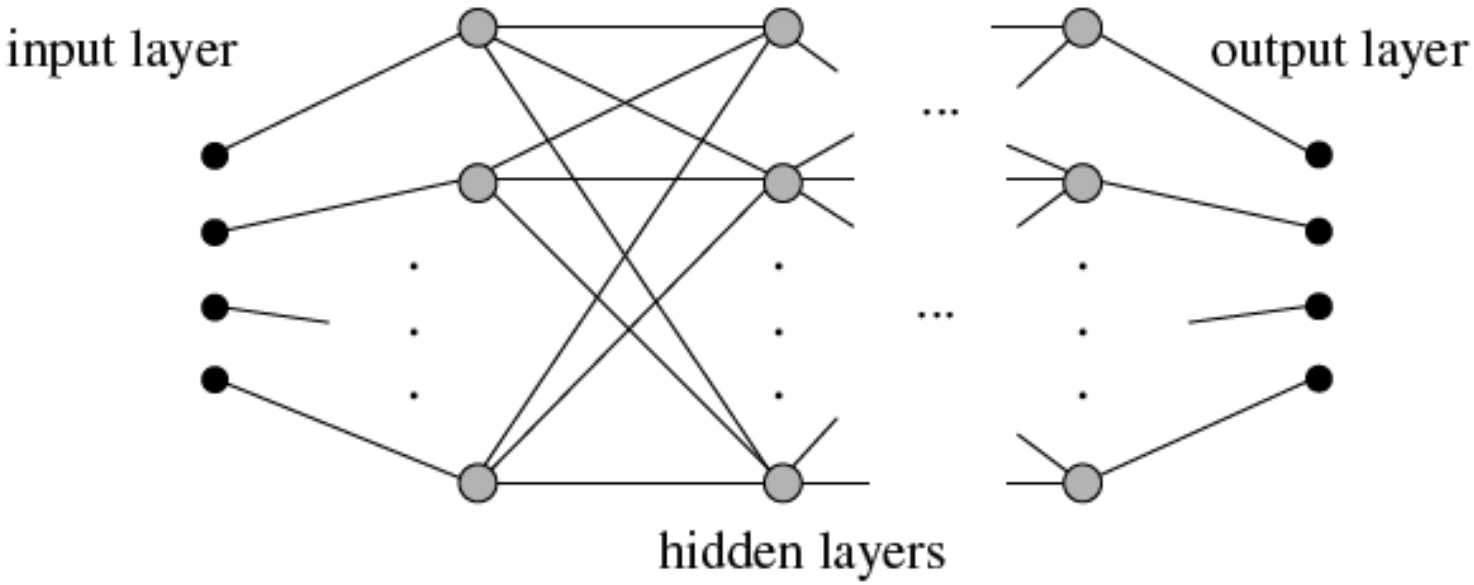
Network architecture

6.1.1 Network architecture

The networks we want to consider must be defined in a more precise way in terms of their *architecture*. The atomic elements of any architecture are the computing units and their interconnections. Each computing unit collects the information from n input lines with an *integration function* $\Psi : \mathbb{R}^n \to \mathbb{R}$. The total excitation computed in this way is then evaluated using an *activation function* $\Phi : \mathbb{R} \to \mathbb{R}$. In perceptrons the integration function is the sum of the inputs. The activation (also called output function) compares the sum with a threshold. Later we will generalize Φ to produce all values between 0 and 1. In the case of Ψ some functions other than addition can also be considered [454], [259]. In this case the networks can compute some difficult functions with fewer computing units.

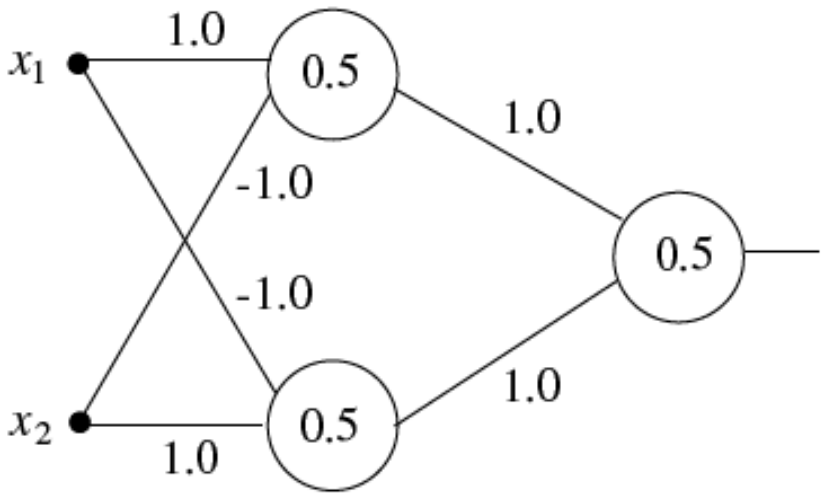
Definition 9. *A network architecture is a tuple (I, N, O, E) consisting of a set I of input sites, a set N of computing units, a set O of output sites and a set E of weighted directed edges. A directed edge is a tuple (u, v, w) whereby $u \in I \cup N$, $v \in N \cup O$ and $w \in \mathbb{R}$.*

General multi-layered feed-forward network



- input layer (remember: inputs only, no computation here)
- one or more layers of hidden neurons
- one layer of output neurons

The XOR in a 2-layer network



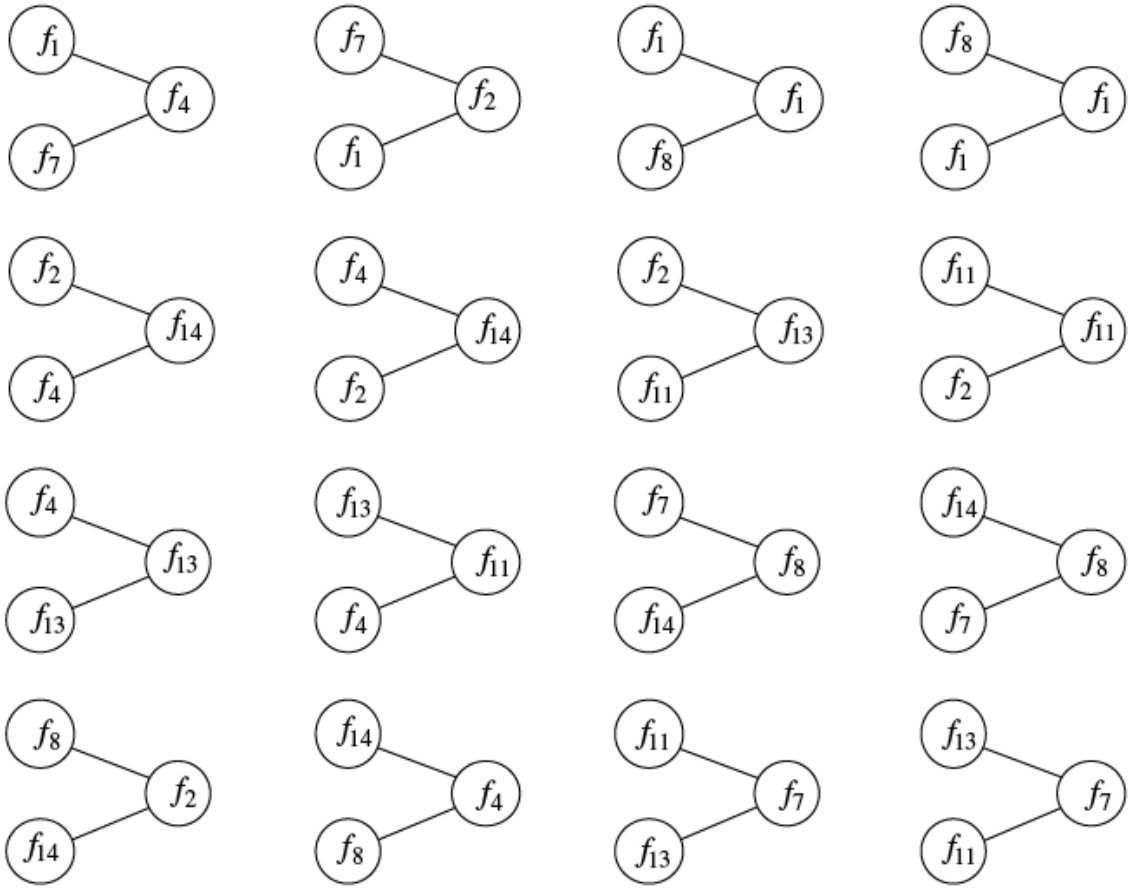
- a solution of the XOR function
- based on $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$

The 16 Boolean two-arg functions, again

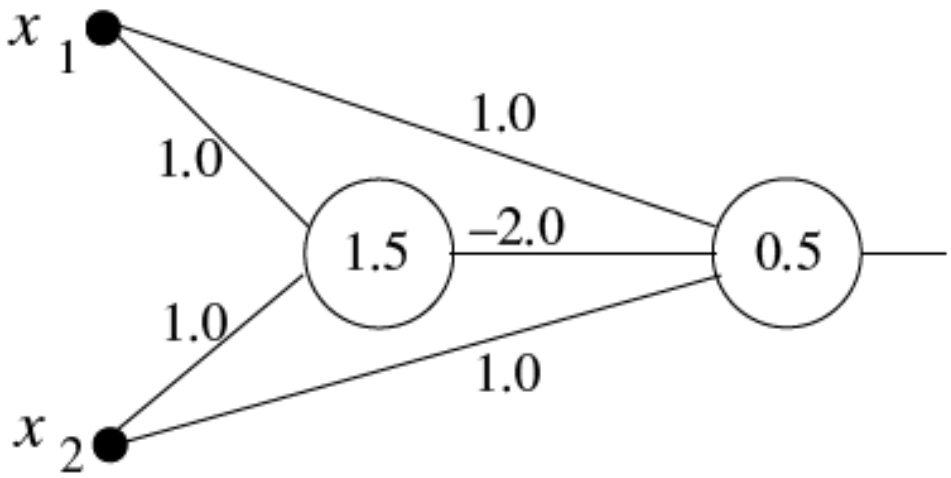
$$
\begin{aligned}
f_0(x_1, x_2) &= f_{0000}(x_1, x_2) = 0 \\
f_1(x_1, x_2) &= f_{0001}(x_1, x_2) = \neg(x_1 \vee x_2) \\
f_2(x_1, x_2) &= f_{0010}(x_1, x_2) = x_1 \wedge \neg x_2 \\
f_3(x_1, x_2) &= f_{0011}(x_1, x_2) = \neg x_2 \\
f_4(x_1, x_2) &= f_{0100}(x_1, x_2) = \neg x_1 \wedge x_2 \\
f_5(x_1, x_2) &= f_{0101}(x_1, x_2) = \neg x_1 \\
f_6(x_1, x_2) &= f_{0110}(x_1, x_2) = x_1 \oplus x_2 \\
f_7(x_1, x_2) &= f_{0111}(x_1, x_2) = \neg(x_1 \wedge x_2)
\end{aligned}
$$

$$
\begin{aligned}
f_8(x1, x_2) &= f_{1000}(x_1, x_2) = x_1 \wedge x_2 \\
f_9(x1, x_2) &= f_{1001}(x_1, x_2) = x_1 \equiv x_2 \\
f_{10}(x1, x_2) &= f_{1010}(x_1, x_2) = x_1 \\
f_{11}(x1, x_2) &= f_{1011}(x_1, x_2) = x_1 \vee \neg x_2 \\
f_{12}(x1, x_2) &= f_{1100}(x_1, x_2) = x_2 \\
f_{13}(x1, x_2) &= f_{1101}(x_1, x_2) = \neg x_1 \vee x_2 \\
f_{14}(x1, x_2) &= f_{1110}(x_1, x_2) = x_1 \vee x_2 \\
f_{15}(x1, x_2) &= f_{1111}(x_1, x_2) = 1
\end{aligned}
$$

Computation of XOR with three units

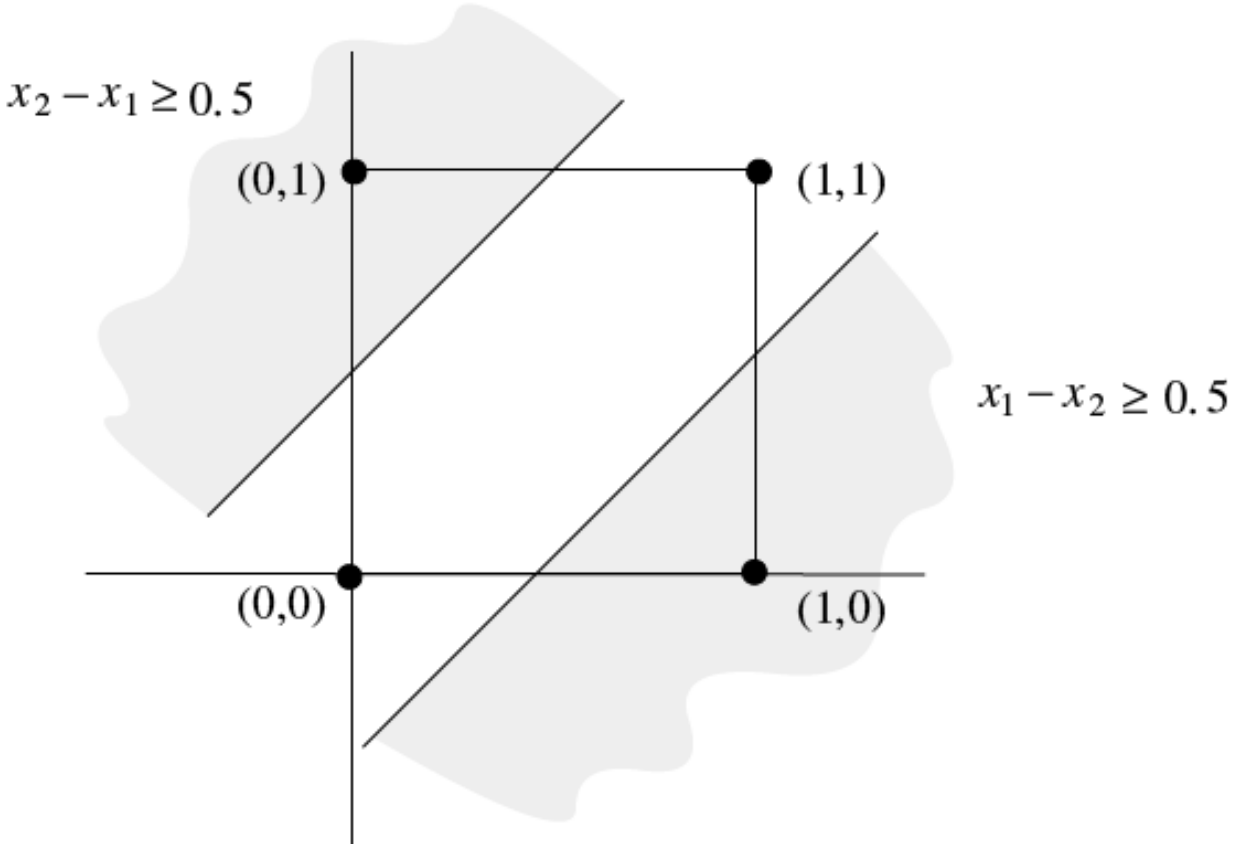


The XOR with two neurons



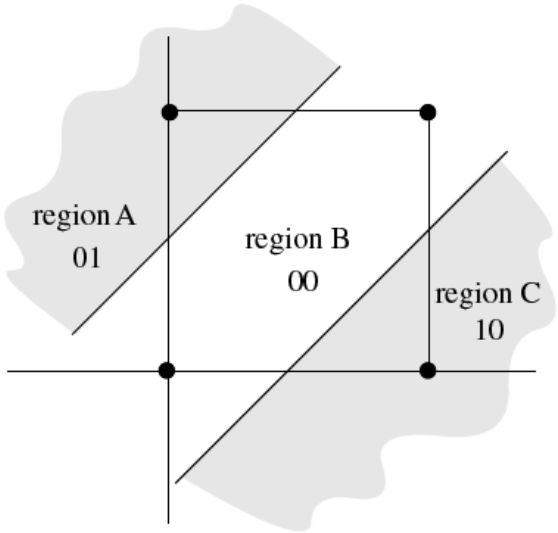
- network with *shortcut* connections
- only two neurons
- but learning is more complex

Geometric interpretation



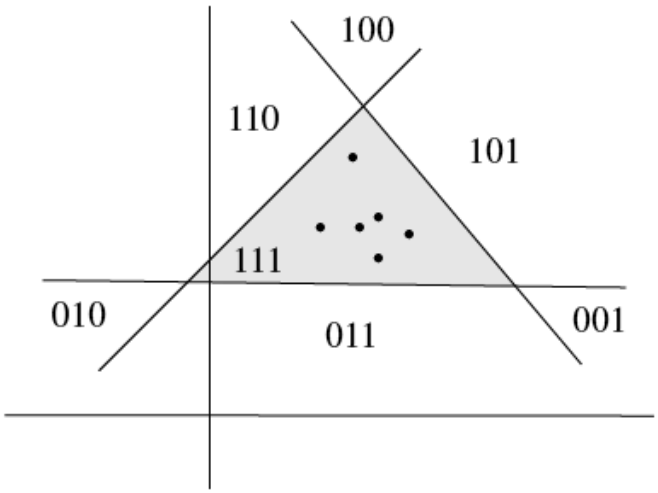
- every neuron in the hidden layer defines one separation
- output neuron combines the half-spaces

Geometric interpretation



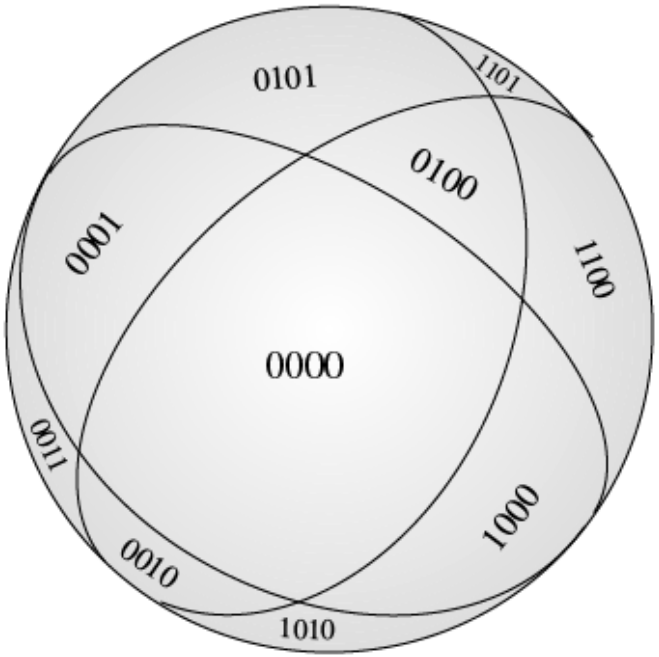
- output neuron must only learn 3 regions
- *feature space* has lower dimension that the input space

Polytopes



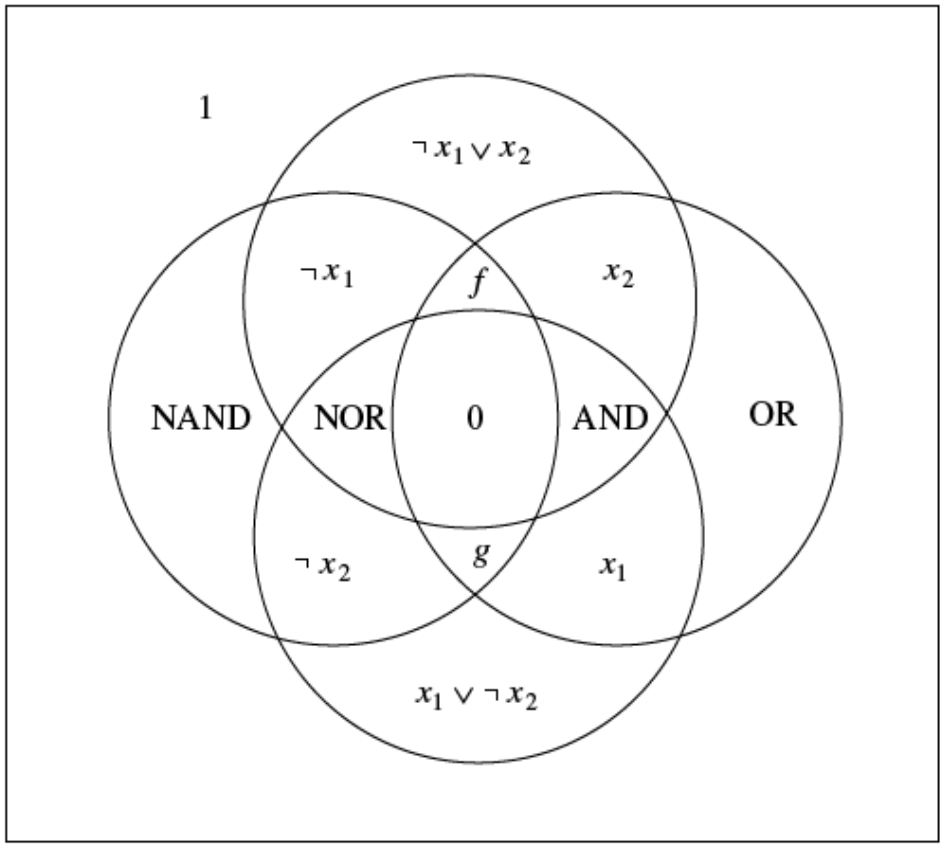
- multiple separations delimit a polytope

Counting regions in input and weight space



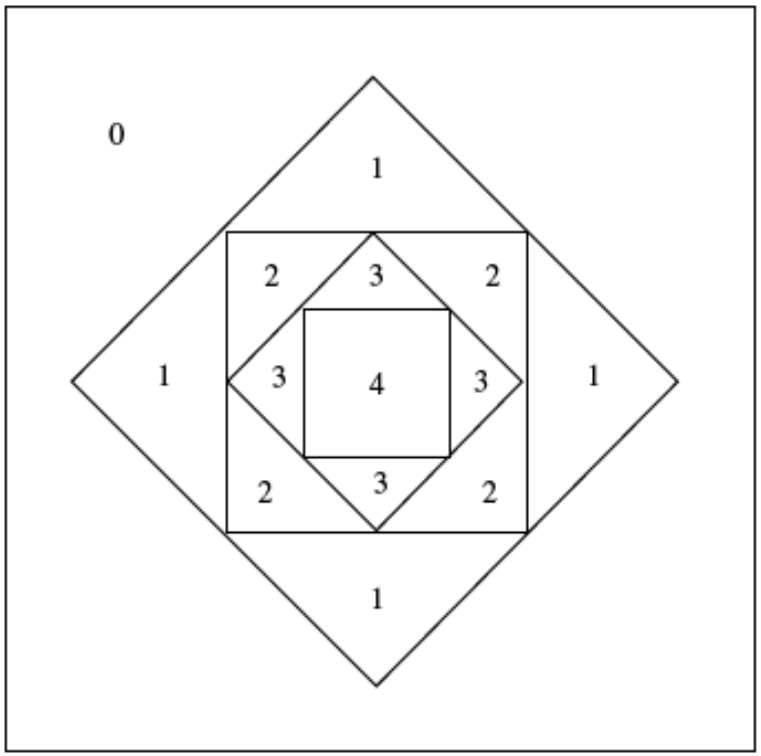
- how many regions can be delimited with n hyperplanes?

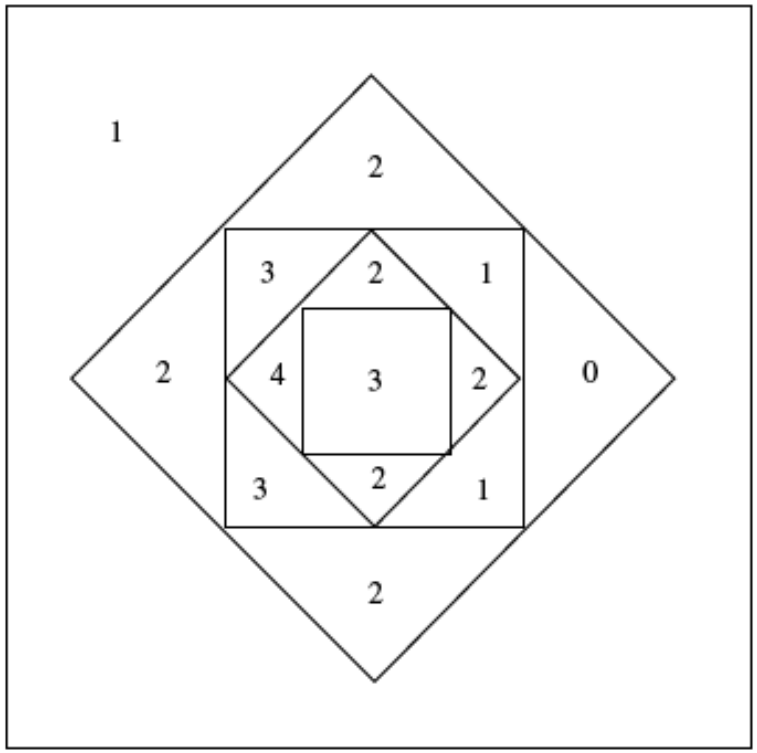
Visualization of the two-arg functions



$f = \neg x_1 \wedge x_2$ $g = x_1 \wedge \neg x_2$

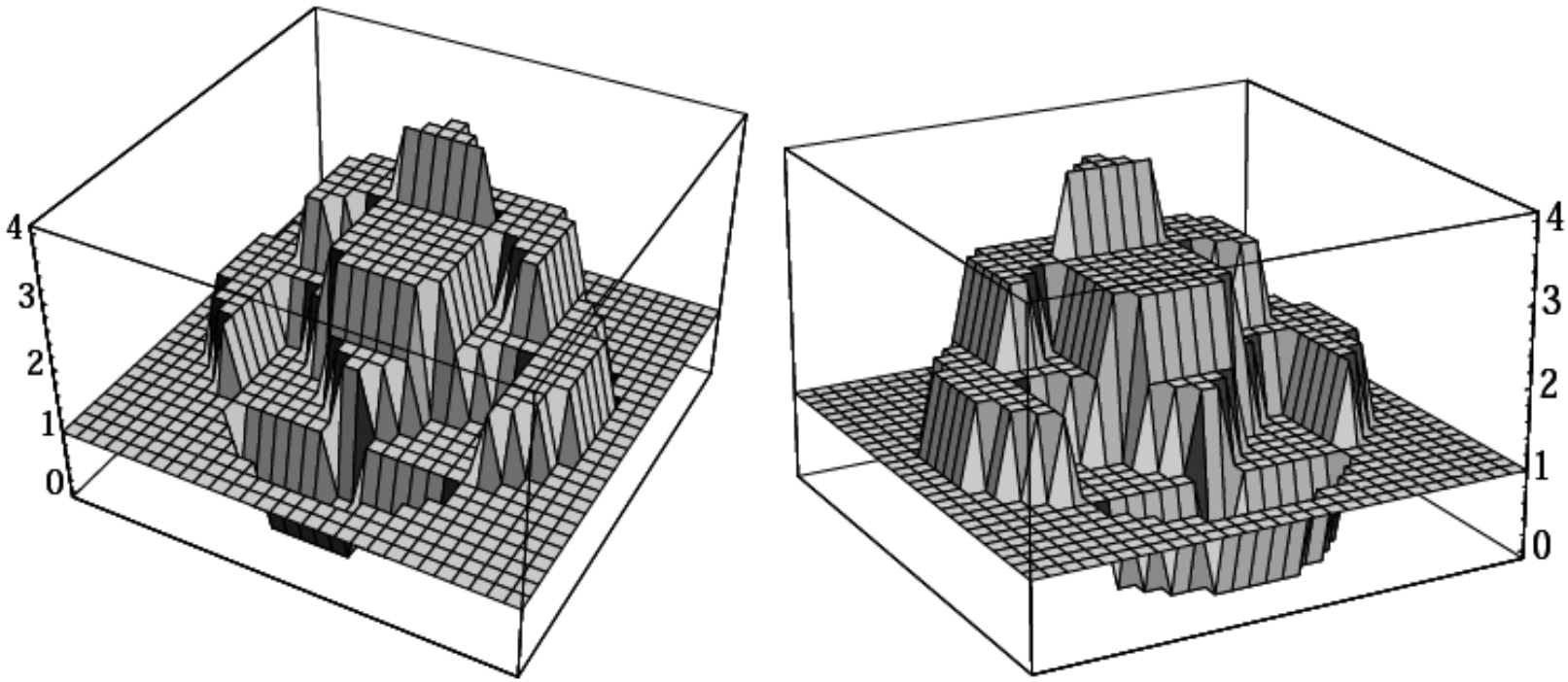
Example error functions





- f_{1111} (left) $x_1 \vee x_2$ (right)

Example error functions



- error-function for OR

Shatterings

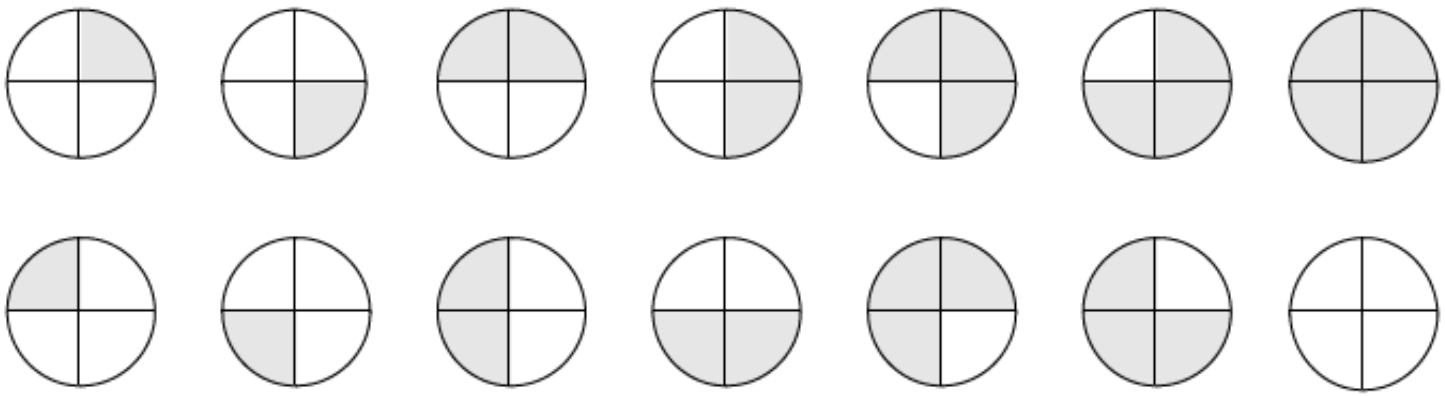
What is the maximum number of elements of an input space which can be classfied by our networks?

- single two-input Perceptron can classify three points
- but not four (see XOR)

- network with 2 hidden neurons: two dividing lines in feature space, 14 regions

- how many *concepts* can be classified by a network with n hidden units?

Shatterings



- possible colorings of the input space for two dividing lines

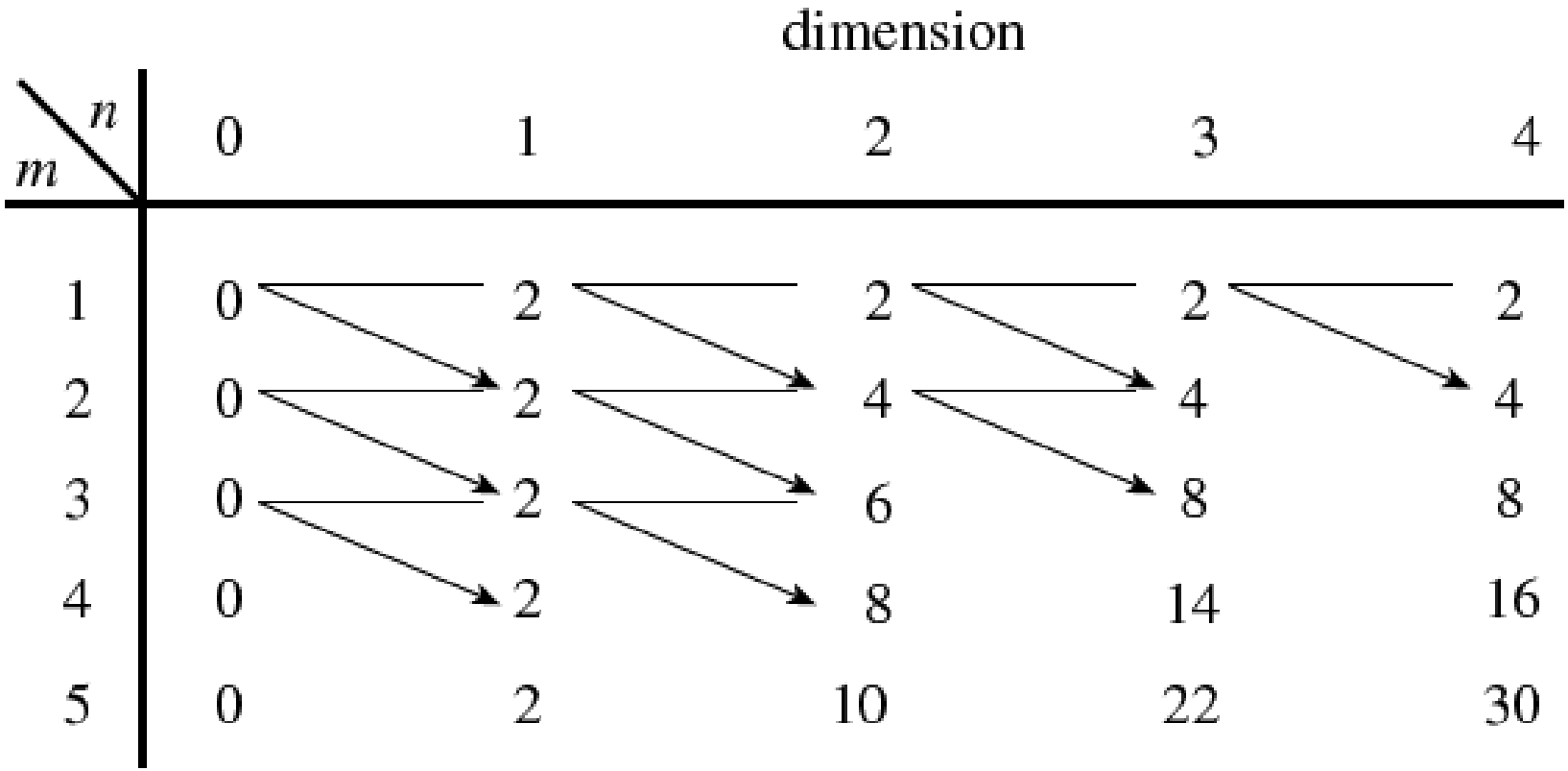
Number of regions in general

How many regions are defined by m cutting hyperplanes of dimension $n-1$ in an n-dimensional space?

- consider only hyperplanes going through the origin
- in 2D: $2m$ regions
- in 3D: each cut increases the number of regions by a factor of 2

- general case: recursive calculation

Recursive calculation of $R(m, n)$



Proposition 9. *Let $R(m,n)$ denote the number of different regions defined by m separating hyperplanes of dimension $n-1$, in general position, in an n-dimensional space. We set $R(1,n)=2$ for $n \geq 1$ and $R(m,0)=0, \forall m \geq 1$. For $n \geq 1$ and $m > 1$*

$$R(m,n) = R(m-1,n) + R(m-1,n-1).$$

Proof. The proof is by induction on m. When $m = 2$ and $n = 1$ the formula is valid. If $m = 2$ and $n \geq 2$ we know that $R(2,n) = 4$ and the formula is valid again:

$$R(2,n) = R(1,n) + R(1,n-1) = 2 + 2 = 4.$$

Number of Boolean and threshold functions of n inputs

Table 6.2. Comparison of the number of Boolean and threshold functions of n inputs and two different bounds

n	2^{2^n}	$T(2^n, n)$	$R(2^n, n)$	$\lfloor 2^{n^2+1}/n! \rfloor$
1	4	4	4	4
2	16	14	14	16
3	256	104	128	170
4	65,536	1,882	3,882	5,461
5	4.3×10^9	94,572	412,736	559,240

Number of threshold functions grows polynomially

First consequence. The number of threshold functions in an n-dimensional space grows polynomially whereas the number of possible Boolean functions grows exponentially. The number of Boolean functions definable on n Boolean inputs is 2^{2^n}. The number of threshold functions is a function of the form $2^{n(n-1)}$, since the 2^n input vectors define at most $R(2^n, n)$ regions in weight space, that is, a polynomial of degree $n-1$ on 2^n. The percentage of threshold functions in relation to the total number of logical functions goes to zero as n increases.

Learnability problems

Second consequence. In networks with two or more layers we also have learnability problems. Each unit in the first hidden layer separates input space into two halves. If the hidden layer contains m units and the input vector is of dimension n, the maximum number of classification regions is $R(m, n)$. If the number of input vectors is higher, it can happen that not enough classification regions are available to compute a given logical function. Unsolvable problems for all networks with a predetermined number of units can easily be fabricated by increasing the number of input lines into the network.

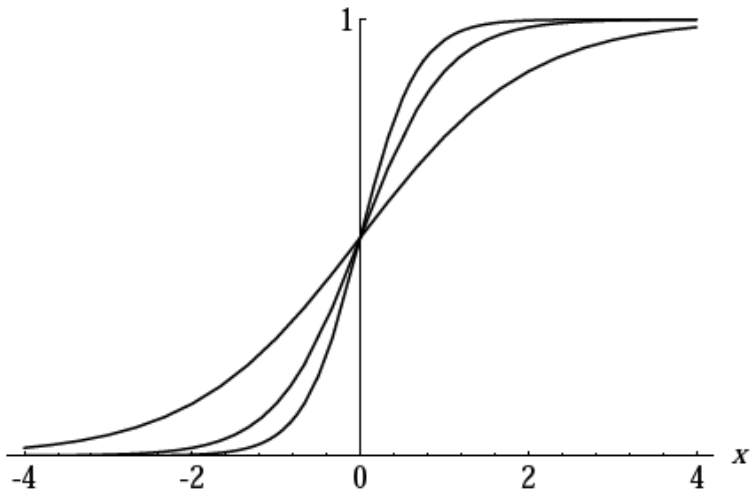
Learning in multi-layer networks

- we now need learning algorithms for our multi-layer networks
- minimization of the error-funtion

- gradient-descent algorithms look promising
- but error-functions of binary threshold neurons are flat

- use neurons with smooth output functions
- instead of the step-function
- accept small/large values (e.g. 0.1, 0.9) for classification

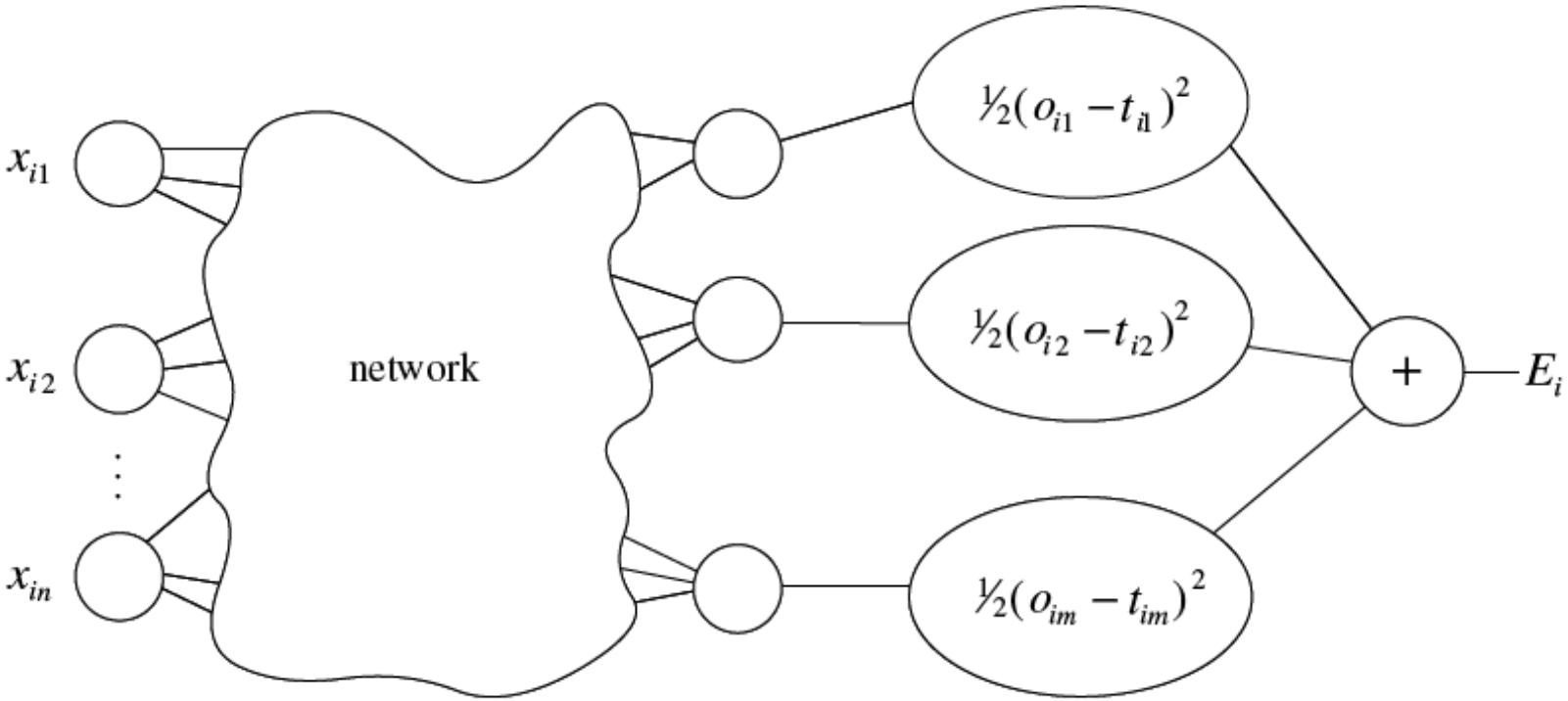
Sigmoid function



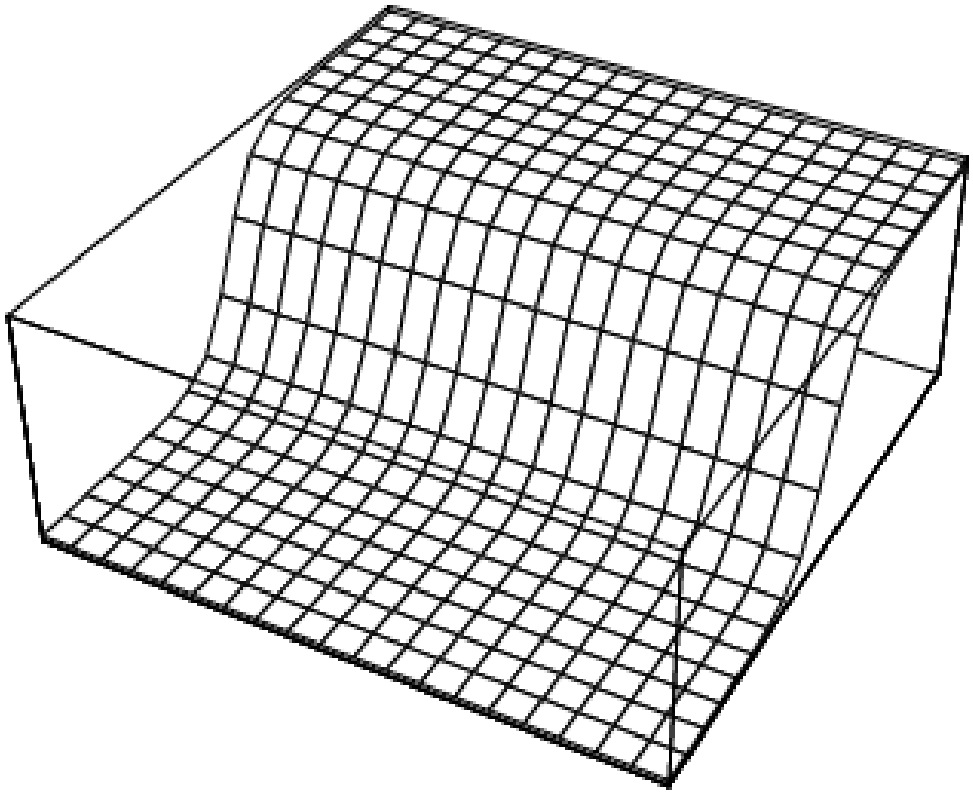
$$s_c(x) = \frac{1}{1 + e^{-cx}}$$

$$S(x) = 2s(x) - 1 = \frac{1 - e^{-x}}{1 + e^{-x}}$$

Generalized error function

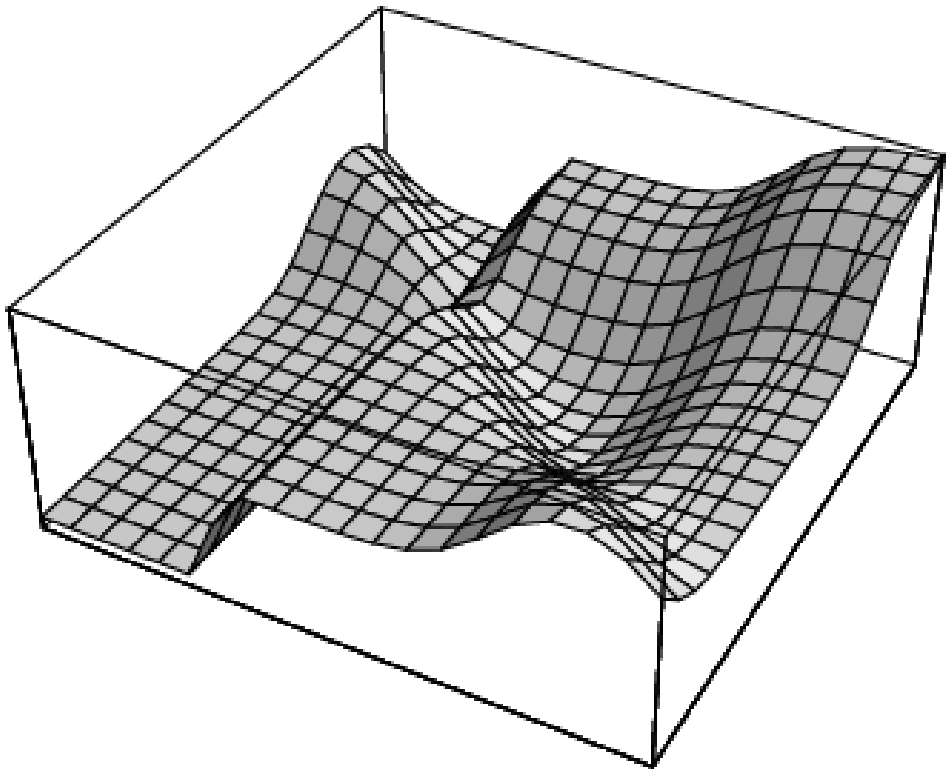


Error function with sigmoid transfer function



- smooth gradients instead of steps

Error function with local minima



- possibly, exponential number of local minima

Two sides of a computing unit

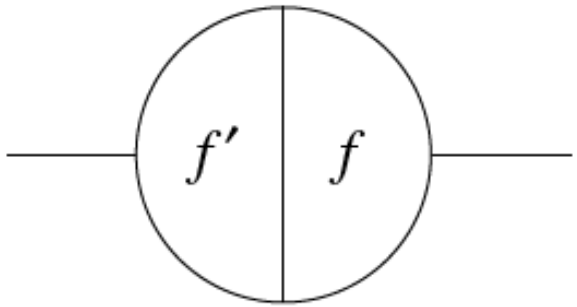


Fig. 7.7. The two sides of a computing unit

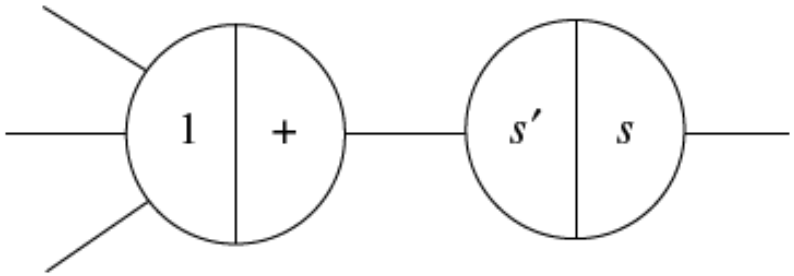


Fig. 7.8. Separation of integration and activation function

- our output function is differentiable
- calculate and use $f(x)$ and $f'(x)$

Backpropagation idea

- present a training pattern to the network
- compare the network output to the desired output, calculate the error in each output neuron
- for each neuron, calculate what the output should have been. Calculate a scaling factor required to reach the desired output.
- Adjust the weights of each neuron to lower the local error
- Assign "blame" for the local error to neurons at the previous level, giving greater responsibility to neurons connected by stronger weights
- repeat for every previous layer, using the "blame" as the local error

(Wikipedia)

Backpropagation pseudocode

Initialize the weights in the network (often randomly)
  Do
    For each example e in the training set
      O = neural-net-output(network, e) forward pass
      T = teacher output for e
      Calculate error $(T - O)$ at the output units
      ; backward pass
      Compute δw_i for all weights from hidden layer to output layer
      Compute δw_i for all weights from input layer to hidden layer
      Update the weights in the network
  Until all examples classified correctly or stopping criterion satisfied
  Return the network

(Wikipedia)

Backpropagation algorithm

Algorithm 7.2.1 *Backpropagation algorithm.*

Consider a network with a single real input x and network function F. The derivative $F'(x)$ is computed in two phases:

Feed-forward: the input x is fed into the network. The primitive functions at the nodes and their derivatives are evaluated at each node. The derivatives are stored.

Backpropagation: the constant 1 is fed into the output unit and the network is run backwards. Incoming information to a node is added and the result is multiplied by the value stored in the left part of the unit. The result is transmitted to the left of the unit. The result collected at the input unit is the derivative of the network function with respect to x.

Four phases of the algorithm

After choosing the weights of the network randomly, the backpropagation algorithm is used to compute the necessary corrections. The algorithm can be decomposed in the following four steps:

i) Feed-forward computation

ii) Backpropagation to the output layer

iii) Backpropagation to the hidden layer

iv) Weight updates

The algorithm is stopped when the value of the error function has become sufficiently small.

Function composition

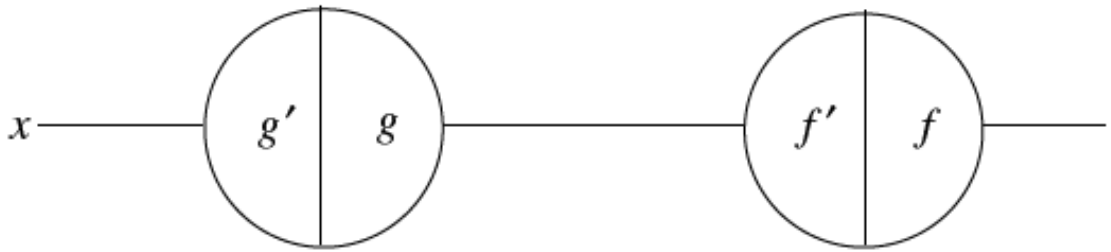


Fig. 7.9. Network for the composition of two functions

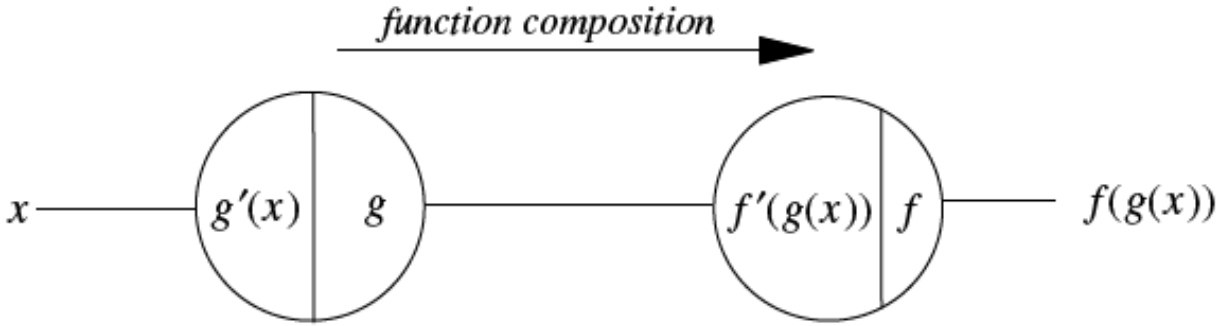
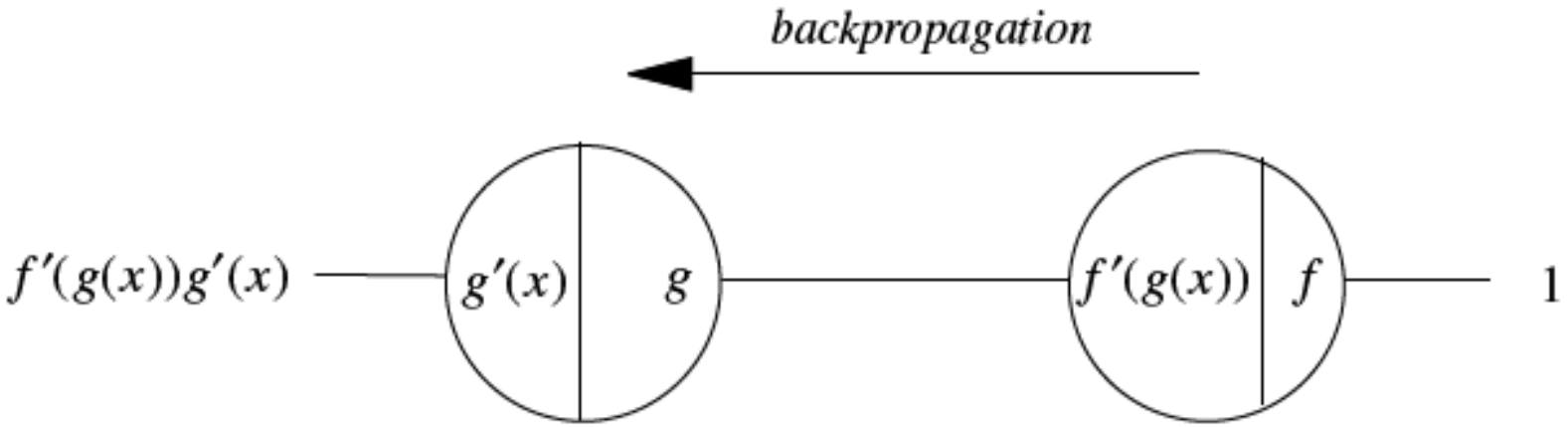
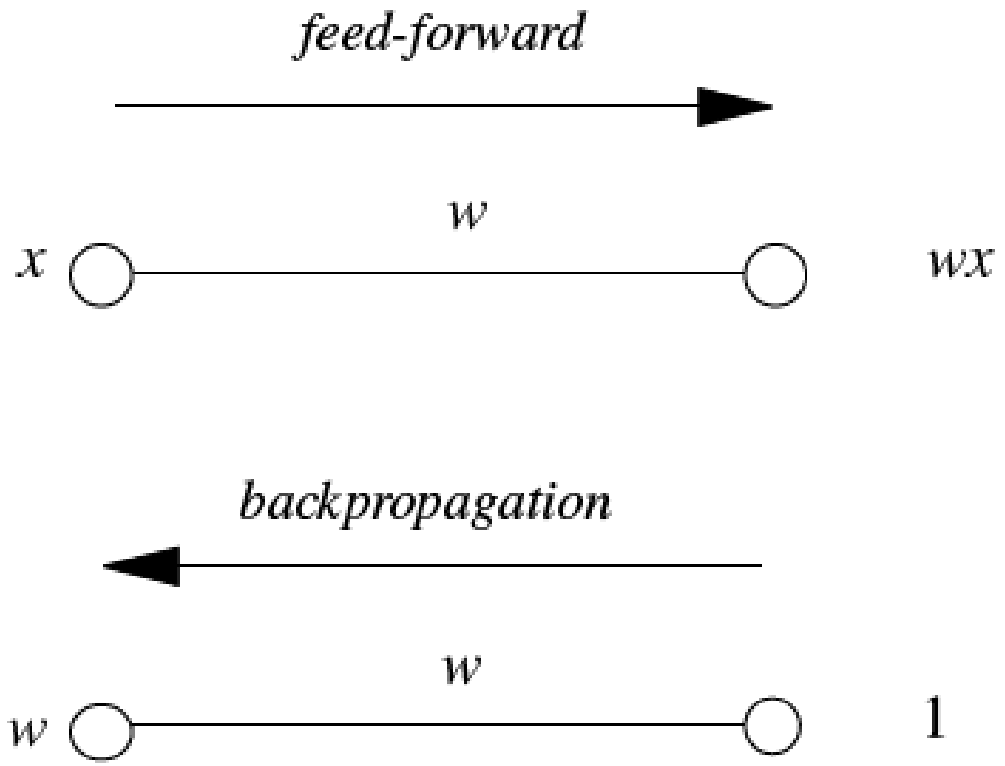


Fig. 7.10. Result of the feed-forward step

Backpropagation composition

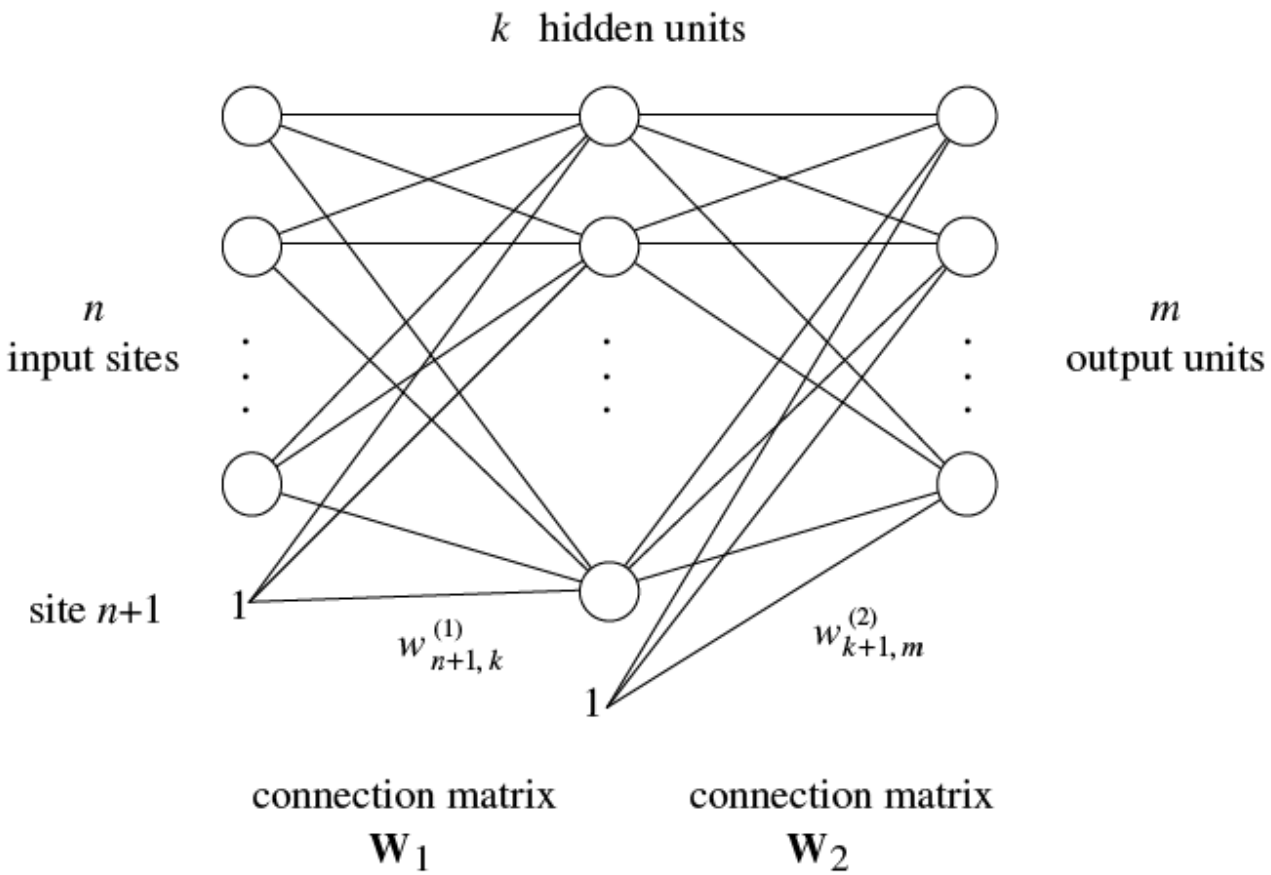


Schema

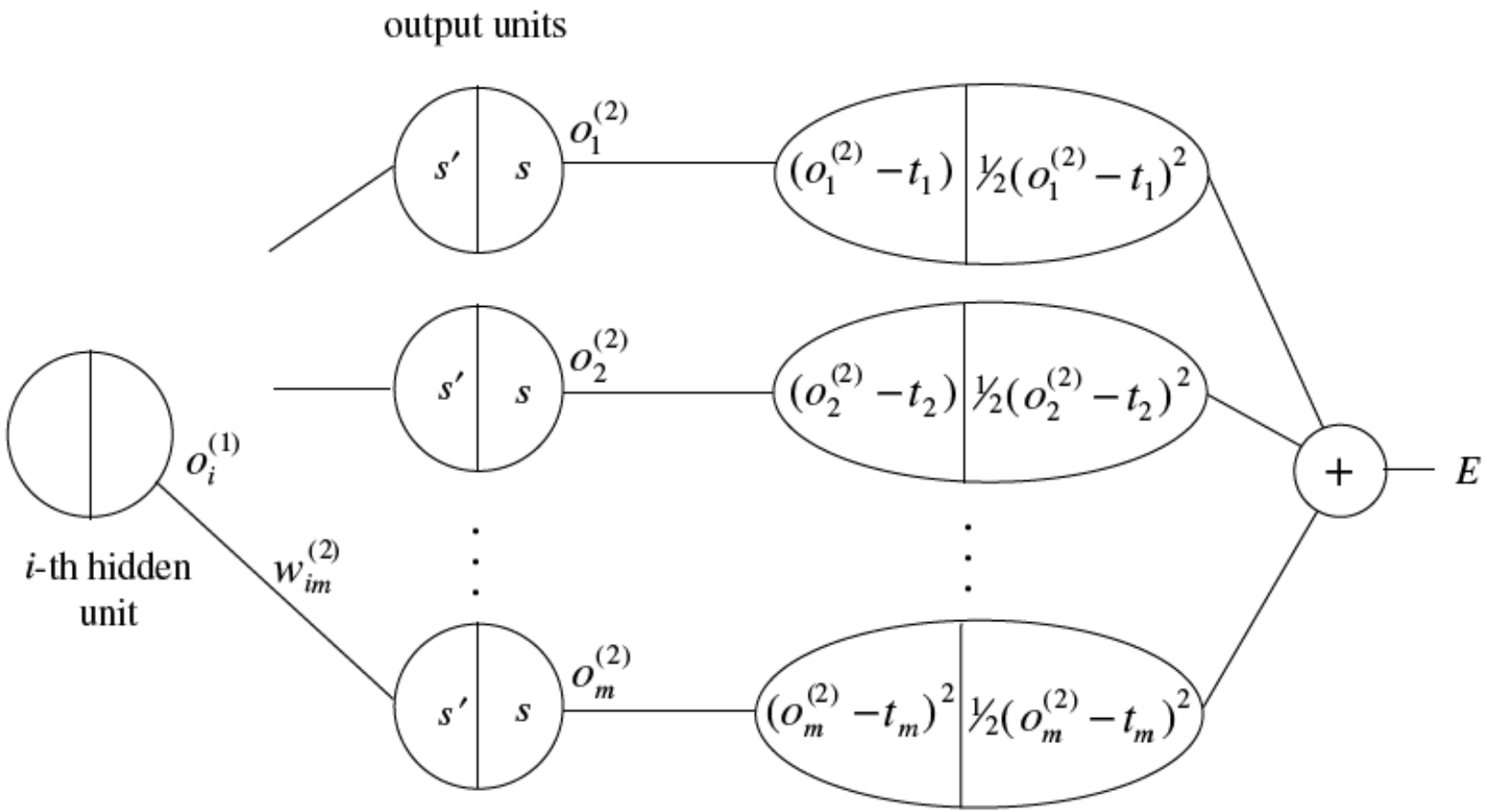


- in both directions, values weighted with w_{ij}

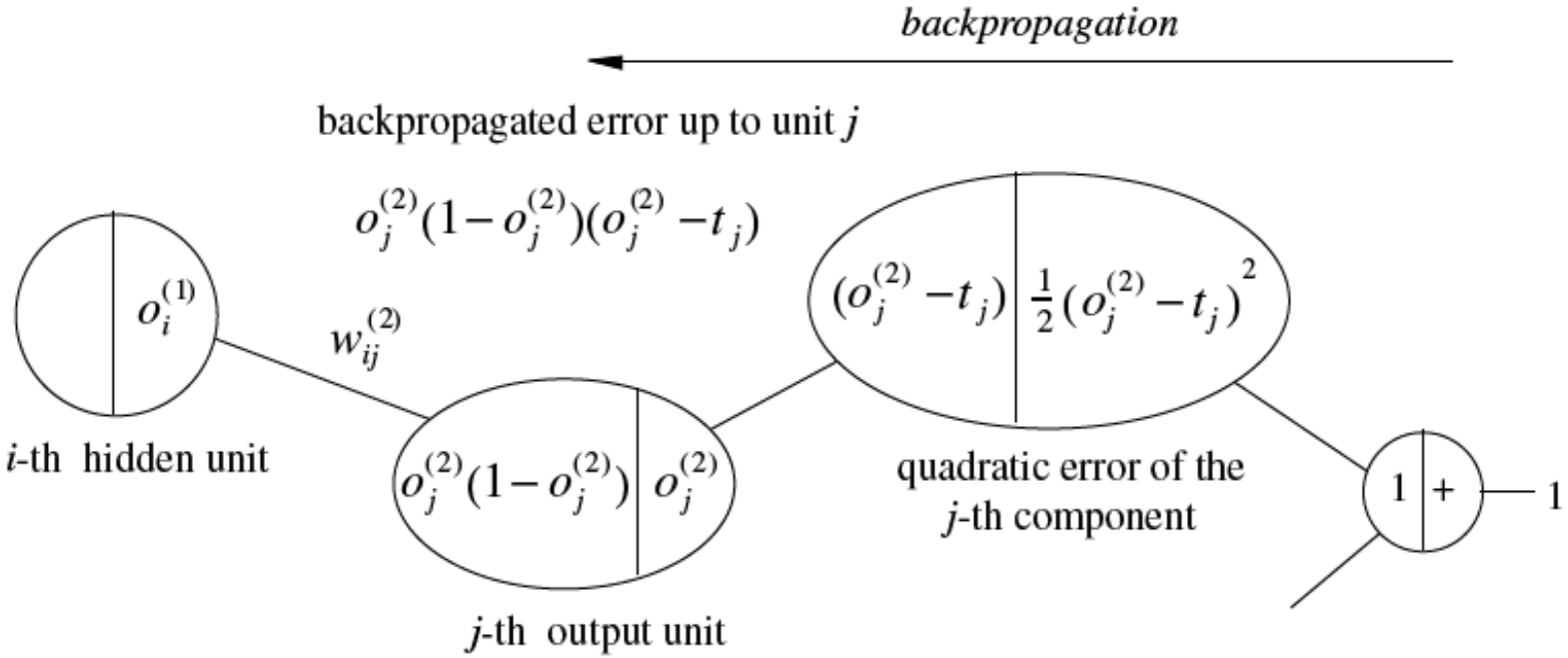
Connections for the 3-layer network



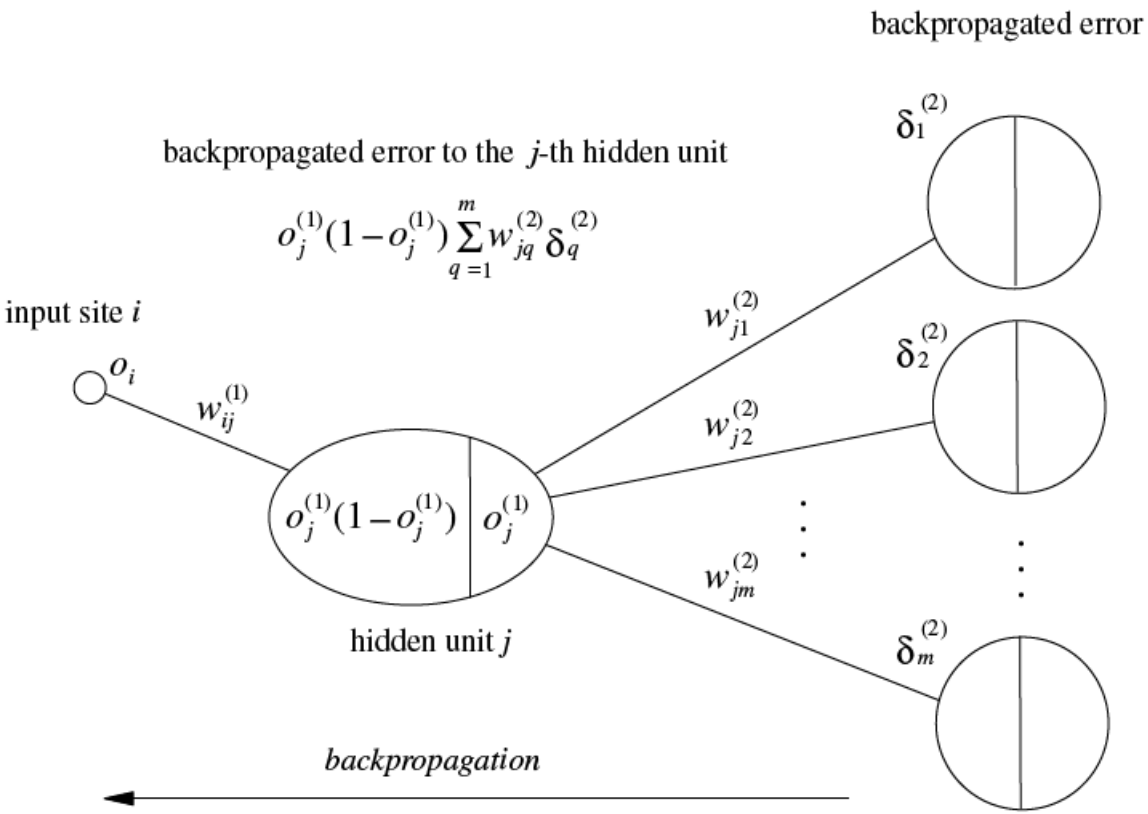
Calculation of the output layer



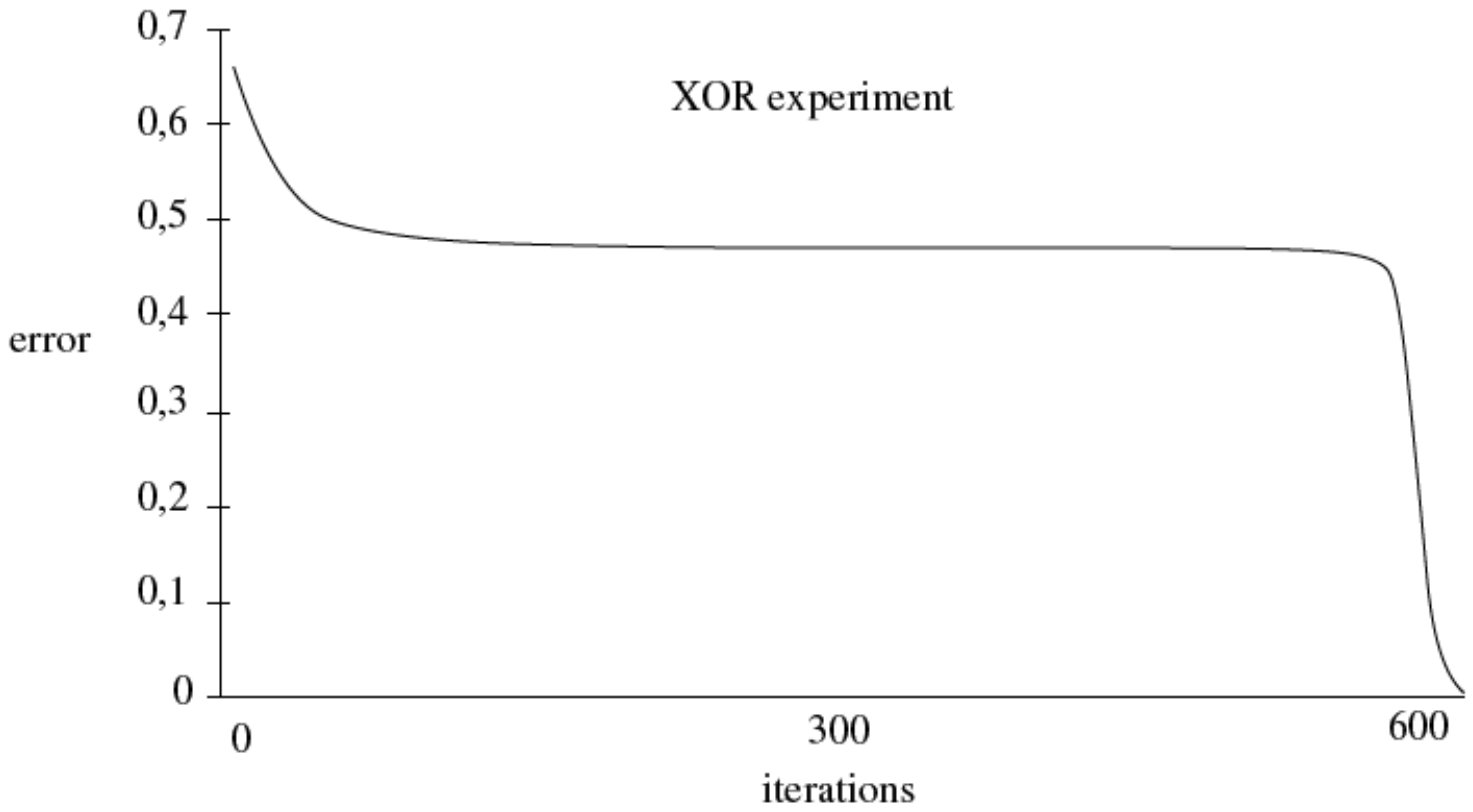
Backpropagation of the error



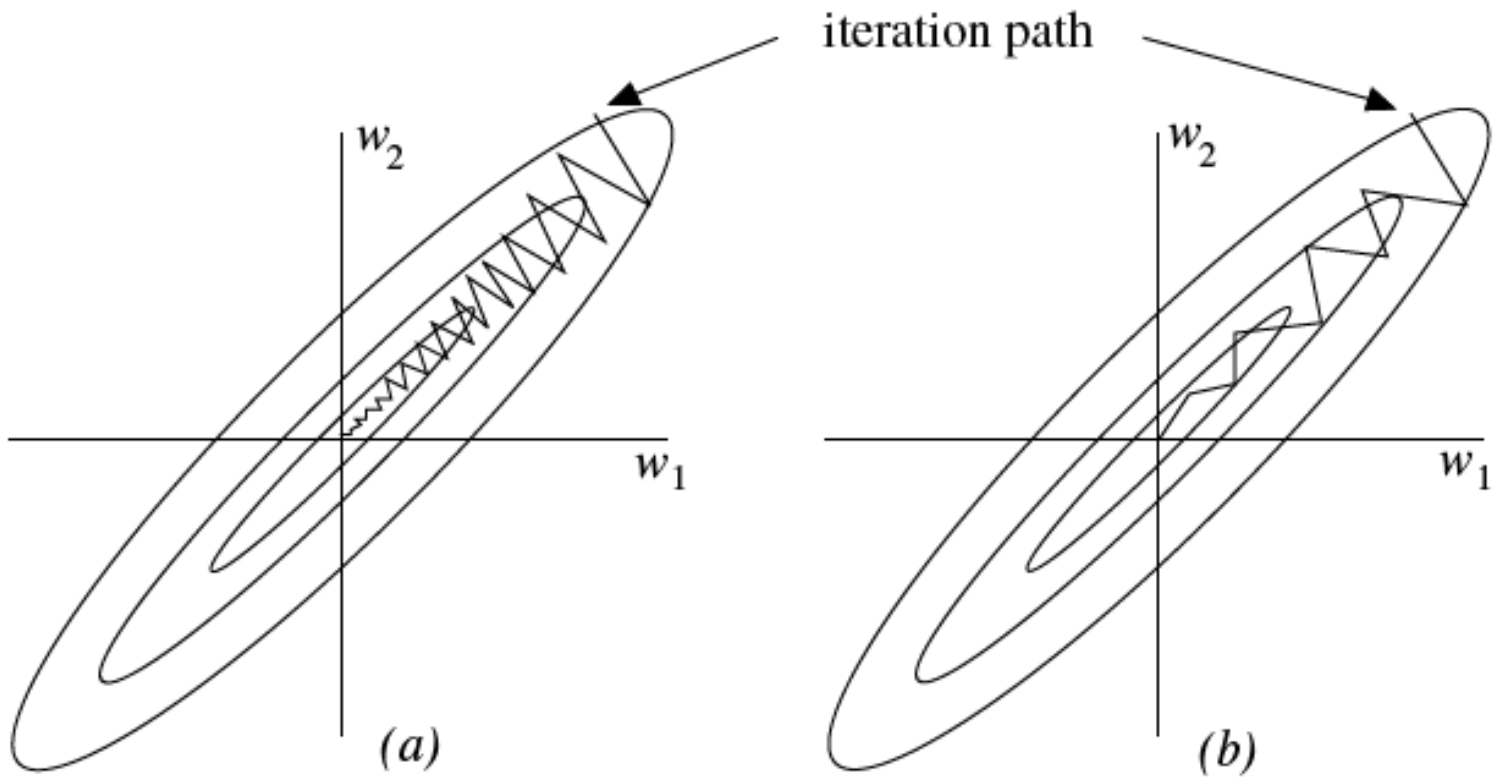
Calculation of hidden units



Time evolution of the error function



Backpropagation with momentum term



Typical backpropagation experiment

- find an interesting problem
- prepare a pattern set
  - normalize, pre-process if necessary
  - select training patterns
  - select validation patterns
- select the network architecture
  - number of layers and neurons per layer
  - select input and output encodings
  - select neuron activation and output functions
  - select the learning algorithm and parameters
- run the learning algorithm on the training patters
- check network performance on the validation patterns
- repeat until convergence

Online vs. offline learning

Online-learning:

- for each training pattern:
  - apply pattern to the input layer
  - propagate the pattern forward to the output neurons
  - calculate error
  - backpropagate errors to the hidden layers
- patterns selected randomly or in fixed order
- usual strategy for BP learning

Offline-learning (batch-learning):

- apply all training patterns to the network
- accumulate the error over all patterns
- one backpropagation step
- very slow when large training set

Applications of backpropagation

- NETtalk
- function approximation

NETtalk: speech synthesis

- text-to-speech synthesis?
- commercially available for many years
- most spoken languages based on a small set of fixed *phonemes*
- so, transform a string of input characters into (a string of) phonemes

- usually, built around hand-crafted *linguistic* rules
- and *transformation rules*
- plus *dictionaries* to handle exceptions from the rule

- can neural networks be taught to speak?

NETtalk

- English speech synthesis
- without linguistic knowledge or transformations

- 3-layer feed-forward network
- input is 7-character sliding window of text
- $7 \cdot 29$ input sites: 1-hot encoding of the 26 letters plus punctuation
- 80 hidden units
- 26 output units: 1-hot encoding of the phonemes
- network generates the phoneme corresponding to the middle of the seven input characters

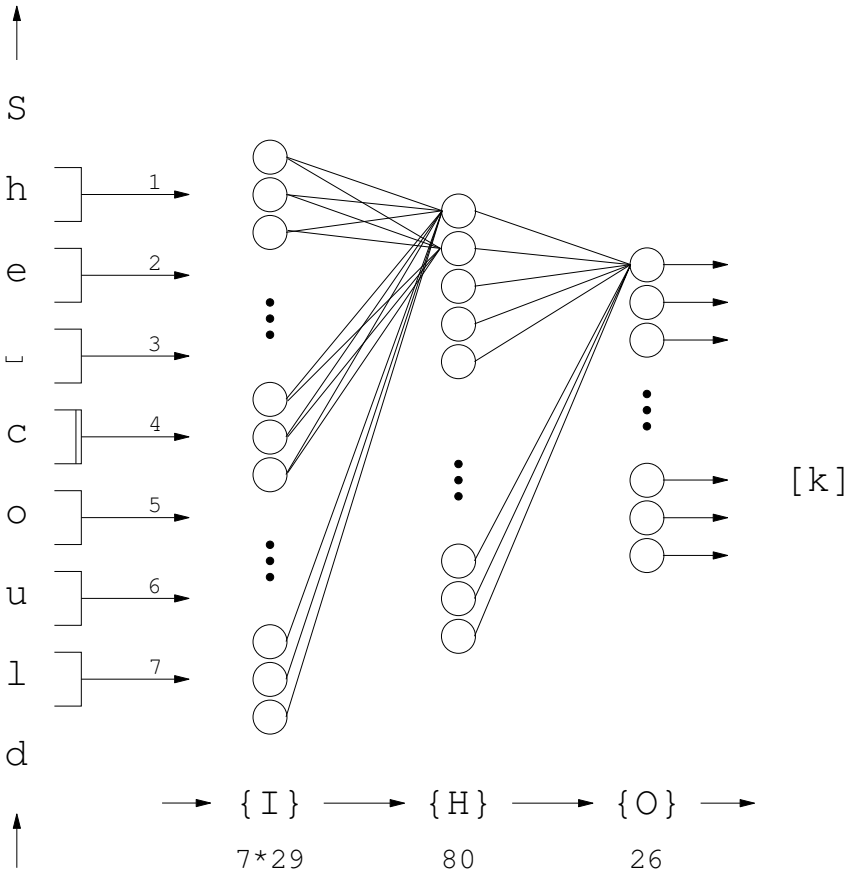
(Sejnowski and Rosenberg 1980)

NETtalk

- corpus of several hundred works together with their phonetic transcription used as training pattersn
- backpropagation learning used to set the network weights ($7 \cdot 29 \cdot 80 + 26 \cdot 80 = 18320$)
- the output neuron with the highest activation is used to select the output phoneme
- phoneme speech-synthesizer drivers the speaker

(Sejnowski and Rosenberg 1980)

NETtalk architecture



NETtalk demo

- initially, network errors similar to children's errors
- final network performance acceptable but not outstanding
- increasing the network size does not really help

- analysis of the final learning network is difficult
- evidence that some of the hidden units encode well-known linguistic rules
- damaging some of the weights produces specific deficiencies

- demo

Outlook

We will look at several PDP/NN architectures:

- McCulloch-Pitts model
- Perceptron
- Multilayer perceptron with backpropagation learning
- Recurrent netwoks: Hopfield-model and associative memory
- Self-organizing maps

- their units and interconnections
- their learning rules
- their environment model
- and possible applications