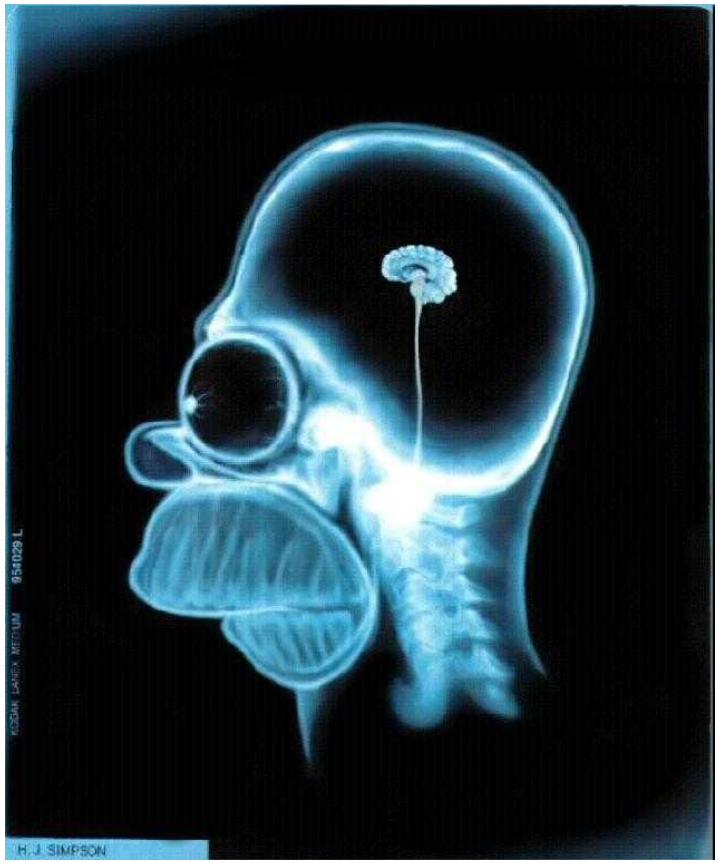
Pattern Recognition

Neural Networks

Outline

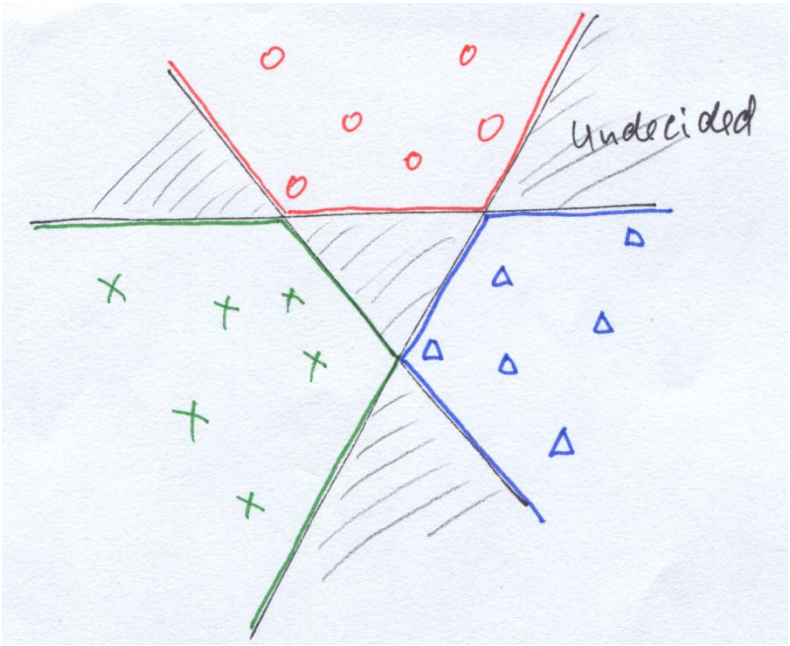


1. Fisher Classifier, Multi-Class Perceptron
2. Feed-Forward Networks, Learning, Error Back-Propagation
3. Hopfield Networks, complex output

Many classes

Before: two classes – a mapping $\mathbb{R}^n \rightarrow \{0, 1\}$
Now: many classes – a mapping $\mathbb{R}^n \rightarrow \{1 \ldots K\}$

How to generalize ? How to learn ?
Two simple (straightforward) approaches:

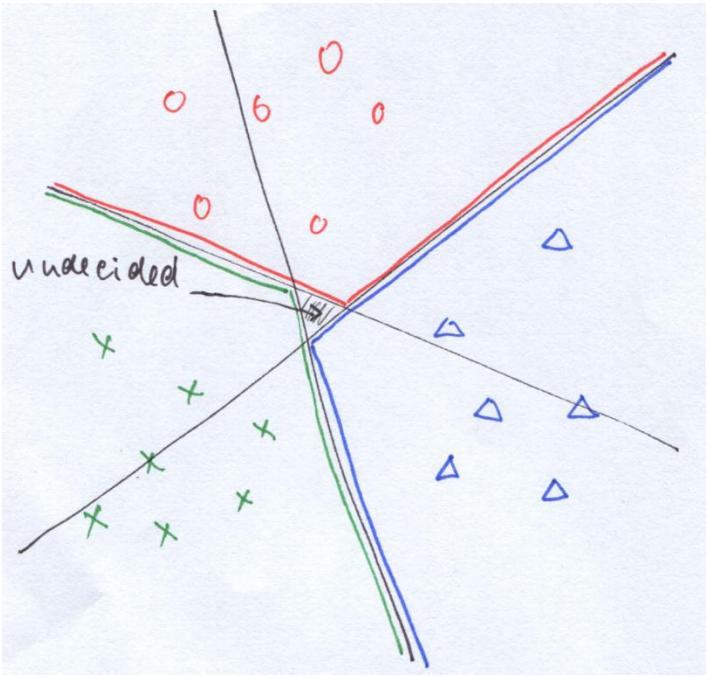


The first one: “one vs. all” – there is one binary classifier per class, that separates this class from all others.

The classification is ambiguous in some areas.

Many classes

Another one:
"pairwise classifiers" – there is a classifier for each class pair



Less ambiguous, better separable.

However:

$K(K-1)/2$ binary classifiers instead of K in the previous case.

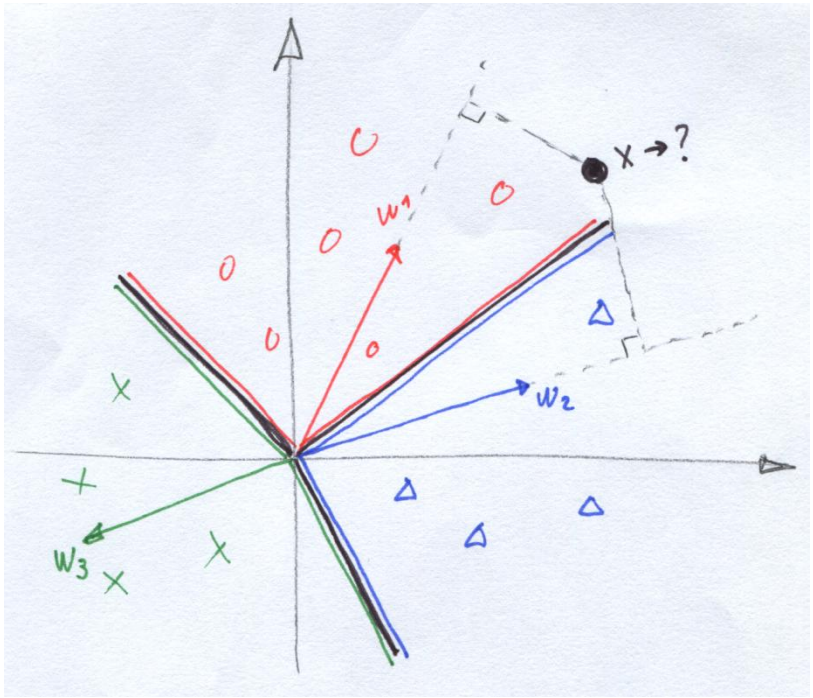
The goal:

- no ambiguities,
- K parameter vectors

Fisher Classifier

Idea: in the binary case the output y is the more likely to be "1" the greater is the scalar product $\langle x, w \rangle$ → generalization:

$$y = \arg\max_k \langle x, w_k \rangle$$



Geometric interpretation (let w_k be normalized)

Consider projections of an input vector x onto vectors w_k

The input space is partitioned into the set of convex cones.

Fisher Classifier

Given: training set $\left((x^1, k^1) \dots (x^l, k^l)\right),\ x^l \in \mathbb{R}^n,\ k^l \in \mathbb{R}$

To be learned: weighting vectors

The task is to choose w_k so that

$$y^l = \arg\max_k \langle x^l, w_k \rangle \quad \forall l$$

It can be equivalently written as

$$\langle x^l, w_{y^l} \rangle > \langle x^l, w_k \rangle \quad \forall l, k \neq y^l$$

– a system of linear inequalities, but a “heterogenic” one.

The trick – transformation of the input/parameter space.

Fisher Classifier

Example for three classes: Consider e.g. a training example $(x, 1)$, it leads to the following inequalities:

$$\langle x, w_1 \rangle > \langle x, w_2 \rangle$$
$$\langle x, w_1 \rangle > \langle x, w_3 \rangle$$

Let us define the new parameter vector as

$$\tilde{w} = (w_{11}, \ldots w_{1n}, w_{21}, \ldots w_{2n}, w_{31}, \ldots w_{3n})$$

i.e. we "concatenate" all w_k to a single vector.

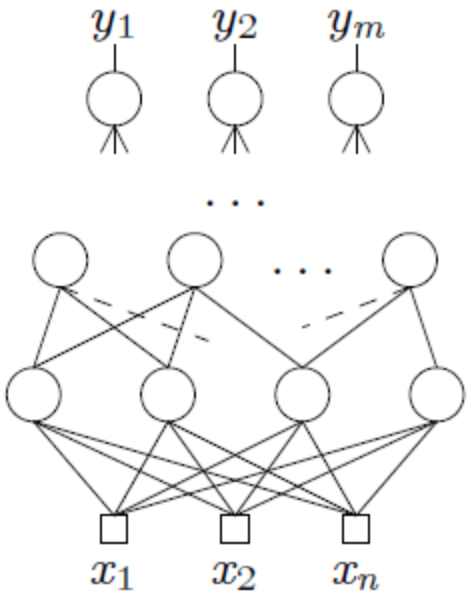
For each inequality (see example above) we introduce a "data point":

$$\tilde{x} = (x_1, \ldots, x_n, -x_1, \ldots, -x_n, \quad 0, \ldots, \quad 0)$$
$$\tilde{x} = (x_1, \ldots, x_n, \quad 0, \ldots, \quad 0, -x_1, \ldots, -x_n)$$

→ all inequalities are written in form of a scalar product $\langle \tilde{x}, \tilde{w} \rangle > 0$

Solution by the Perceptron Algorithm.

Feed-Forward Networks



i – Nummer der Schicht
$i = 0$ – Input Schicht
$i = i_{max}$ – Output Schicht
j – Nummer des Neurons
y_i – Output der i-ten Schicht (Vektor)
y_{ij} – eine Komponente davon
$y_0 = x$ – Input Signal
w_i – Gewichtsmatrix, $i = 1 \dots i_{max}$
$w_{ijj'}$ – Gewicht, das (i, j) mit $(i-1, j')$ verbindet
b_{ij} – Schwellwerte

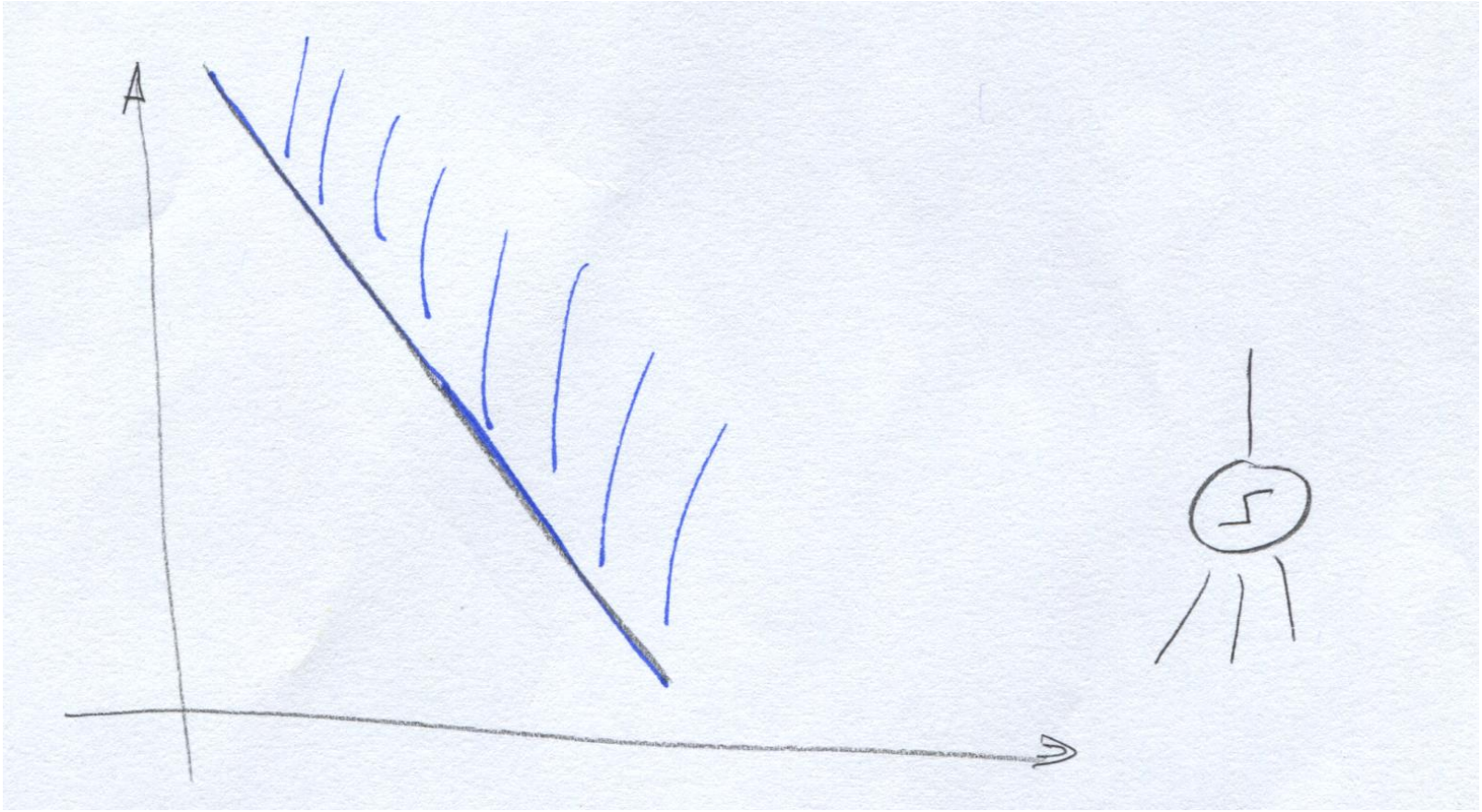
$$y_{ij} = f\left(\sum_{j'} w_{ijj'}\, y_{i-1\, j'} - b_{ij}\right)$$

Special case: $m = 1$, Step-neurons – a mapping $\mathbb{R}^n \to \{0, 1\}$

Which mappings can be modeled ?

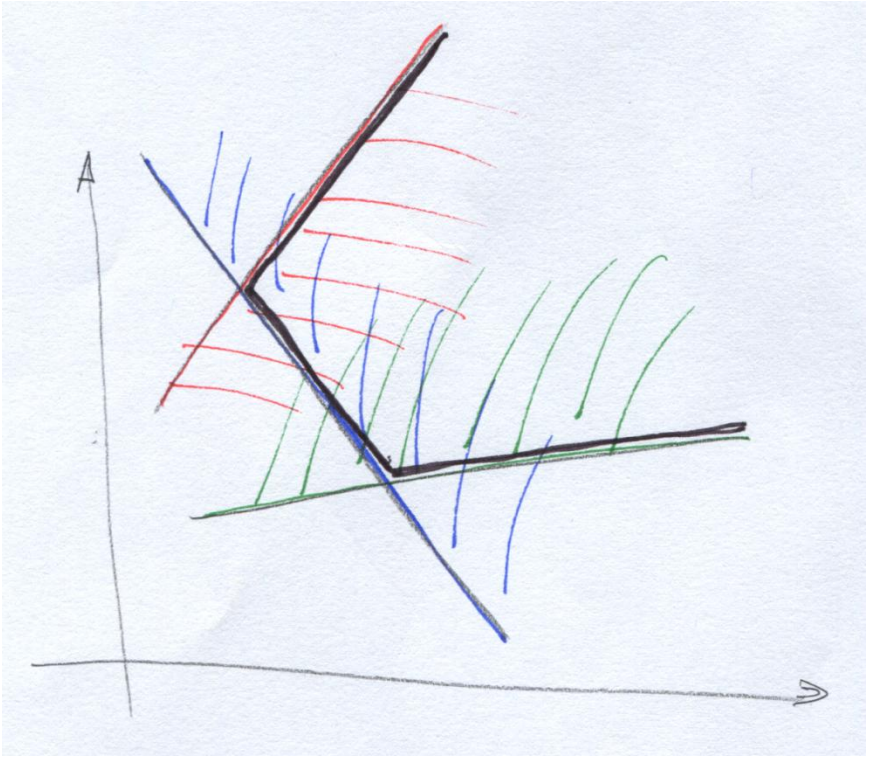
Feed-Forward Networks

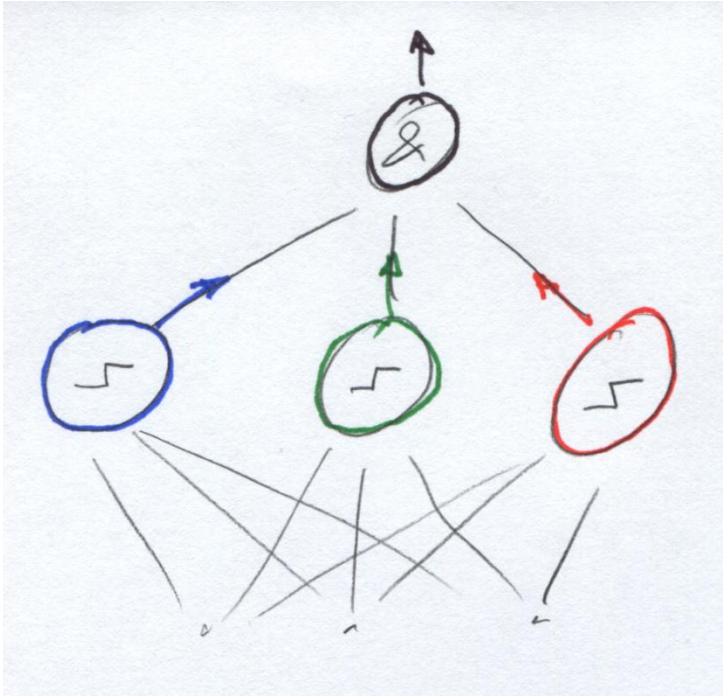
One level – single step-neuron – linear classifier



Feed-Forward Networks

Two levels, “&”-neuron as the output – intersection of half-spaces

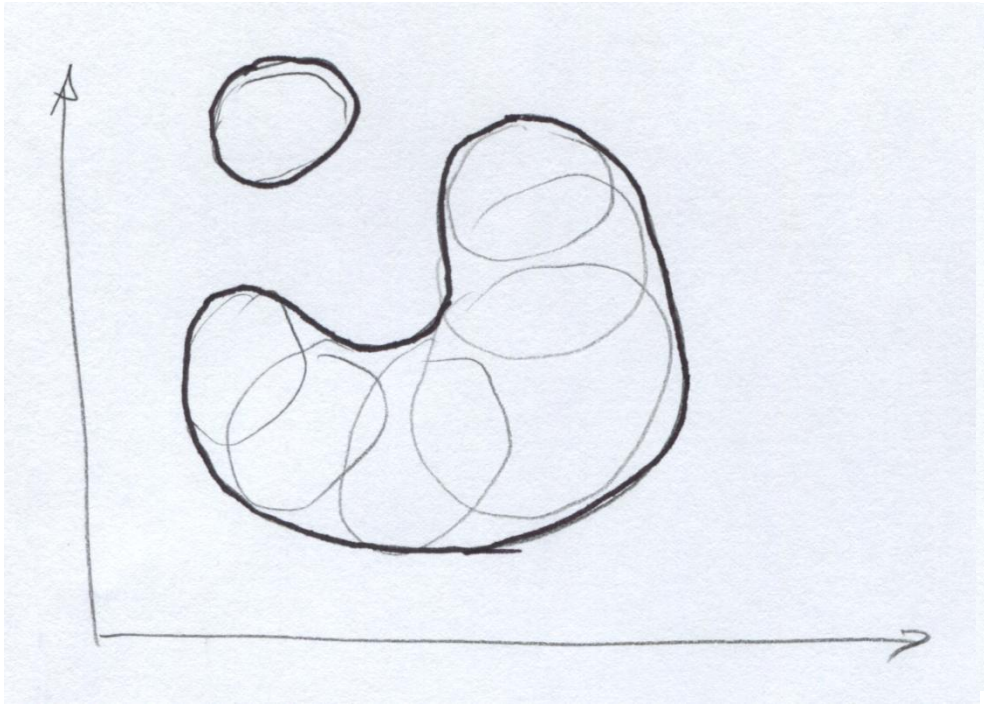


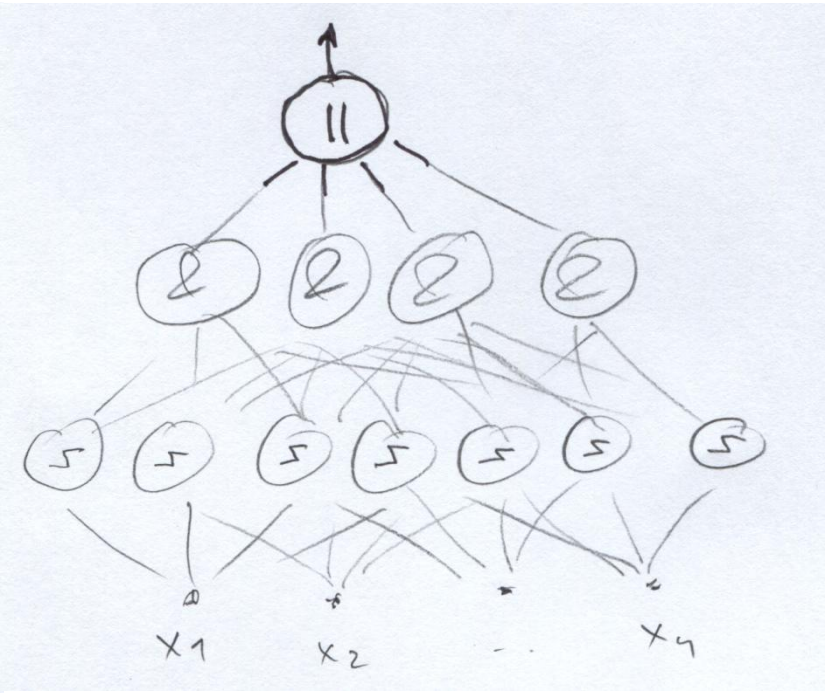


If the number of neurons is not limited, all convex subspaces can be implemented with an arbitrary precision.

Feed-Forward Networks

Three levels– all possible mappings $\mathbb{R}^n \rightarrow \{0, 1\}$ as union of convex subspaces:

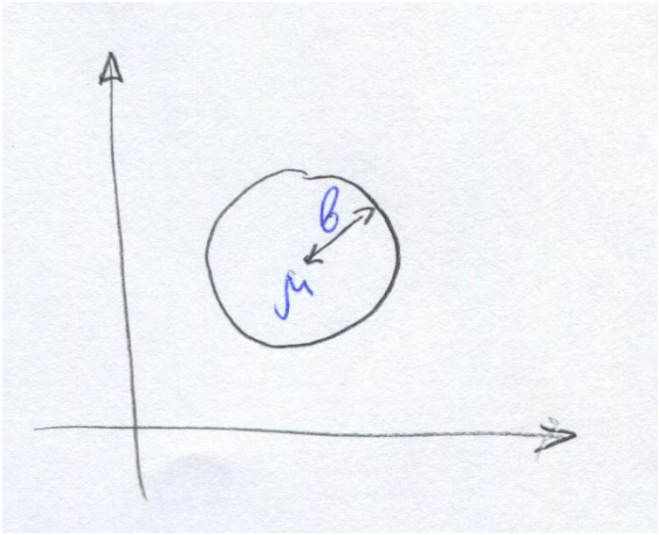




Three levels (sometimes even less) are enough to implement all possible mappings !!!

Radial Basis Functions

Another type of neurons,
corresponding classifier – “inside/outside a ball”



$$y = f(\|x - \mu\| - b)$$

The usage of RBF-neurons “replaces” a level in FF-networks.

With infinitely many RBF-neurons arbitrary mappings with only one intermediate level are possible.

Error Back-Propagation

Learning task:

Given: training data $\big((x^1, k^1) \dots (x^l, k^l)\big)$, $x^l \in \mathbb{R}^n$, $k^l \in \mathbb{R}$
Find: all weights and biases of the net.

Error Back-Propagation is a **gradient descent** method for Feed-Forward-Networks with Sigmoid-neurons

First, we need an objective (error to be minimized)

$$F(w, b) = \sum_l \big(k^l - y(x^l; w, b)\big)^2$$

Now: derive, build the gradient and go.

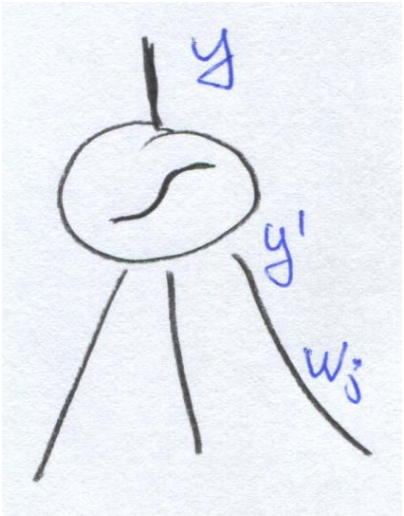
Error Back-Propagation

We start from a single neuron and just one example (x, k) .
Remember:

$$F(w, b) = (k - y)^2$$

$$y = \frac{1}{1 + \exp(-y')}$$

$$y' = \langle x, w \rangle = \sum_j x_j w_j$$



Derivation according to the chain-rule:

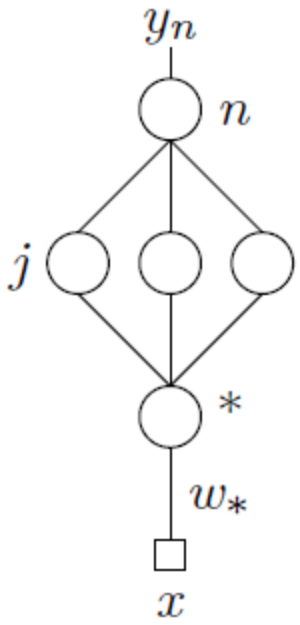
$$\frac{\partial F(w, b)}{\partial w_j} = \frac{\partial F}{\partial y} \cdot \frac{\partial y}{\partial y'} \cdot \frac{\partial y'}{\partial w_j} =$$

$$= (y - k) \cdot \frac{\exp(-y')}{\left(1 + \exp(-y')\right)^2} \cdot x_j = \delta \cdot d(y') \cdot x_j$$

Error Back-Propagation

The “problem”: for intermediate neurons the errors are not known !

Now a bit more complex:



$$\frac{\partial F}{\partial w_*} = \frac{\partial F}{\partial y_n} \cdot \frac{\partial y_n}{\partial y'_n} \cdot \left[\sum_j \frac{\partial y'_n}{\partial y_j} \cdot \frac{\partial y_j}{\partial y'_j} \cdot \frac{\partial y'_j}{\partial y_*} \right] \cdot \frac{\partial y_*}{\partial y'_*} \cdot \frac{\partial y'_*}{\partial w_*} =$$

$$\left[\sum_j \delta_n \cdot d(y'_n) \cdot w_{nj} \cdot d(y'_j) \cdot w_{j*} \right] \cdot d(y'_*) \cdot x =$$

$$\left[\sum_j \delta_j \cdot d(y'_j) \cdot w_{j*} \right] \cdot d(y'_*) \cdot x =$$

with:

$$\delta_j = \delta_n \cdot d(y'_n) \cdot w_{nj}$$

$$\delta_* = \left[\sum_j \delta_j \cdot d(y'_j) \cdot w_{j*} \right]$$

Error Back-Propagation

In general: compute "errors" δ at the i-th level from all δ-s at the i+1-th level – propagate the error.

The Algorithm (for just one example (x, k)):

1. Forward: compute all y and y' (apply the network), compute the output error $\delta_n = y_n - k$;
2. Backward: compute errors in the intermediate levels:

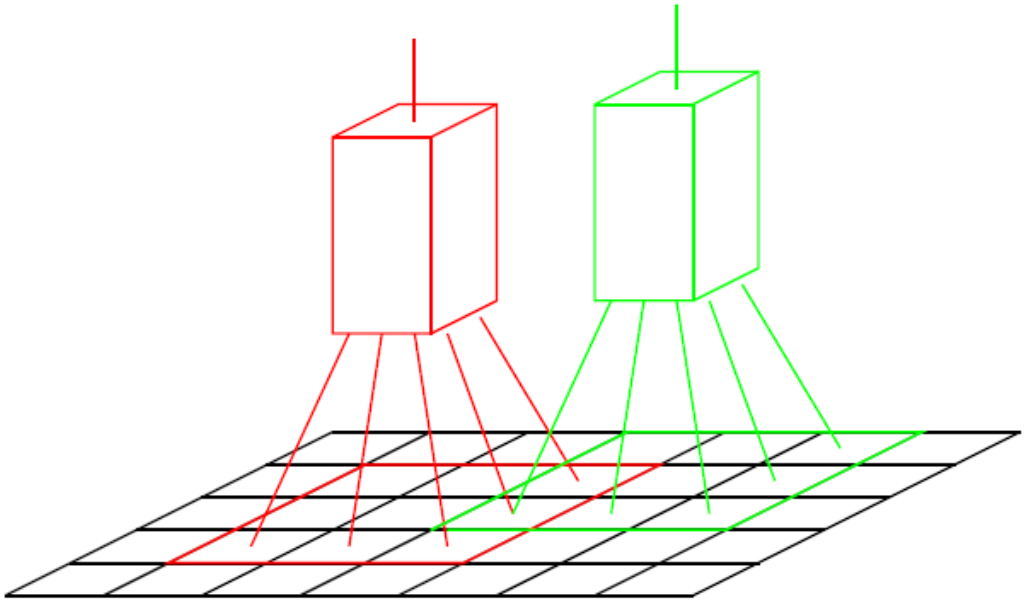
$$\delta_{ij} = \sum_{j'} \delta_{i+1\ j'} \cdot d(y_{i+1\ j'}) \cdot w_{i+1\ j'j}$$

3. Compute the gradient and go.

$$\frac{\partial F}{\partial w_{ijj'}} = \delta_{ij} \cdot d(y'_{ij}) \cdot y_{i-1\ j'}$$

For many examples – just sum them up.

Time Delay Neural Networks (TDNN)

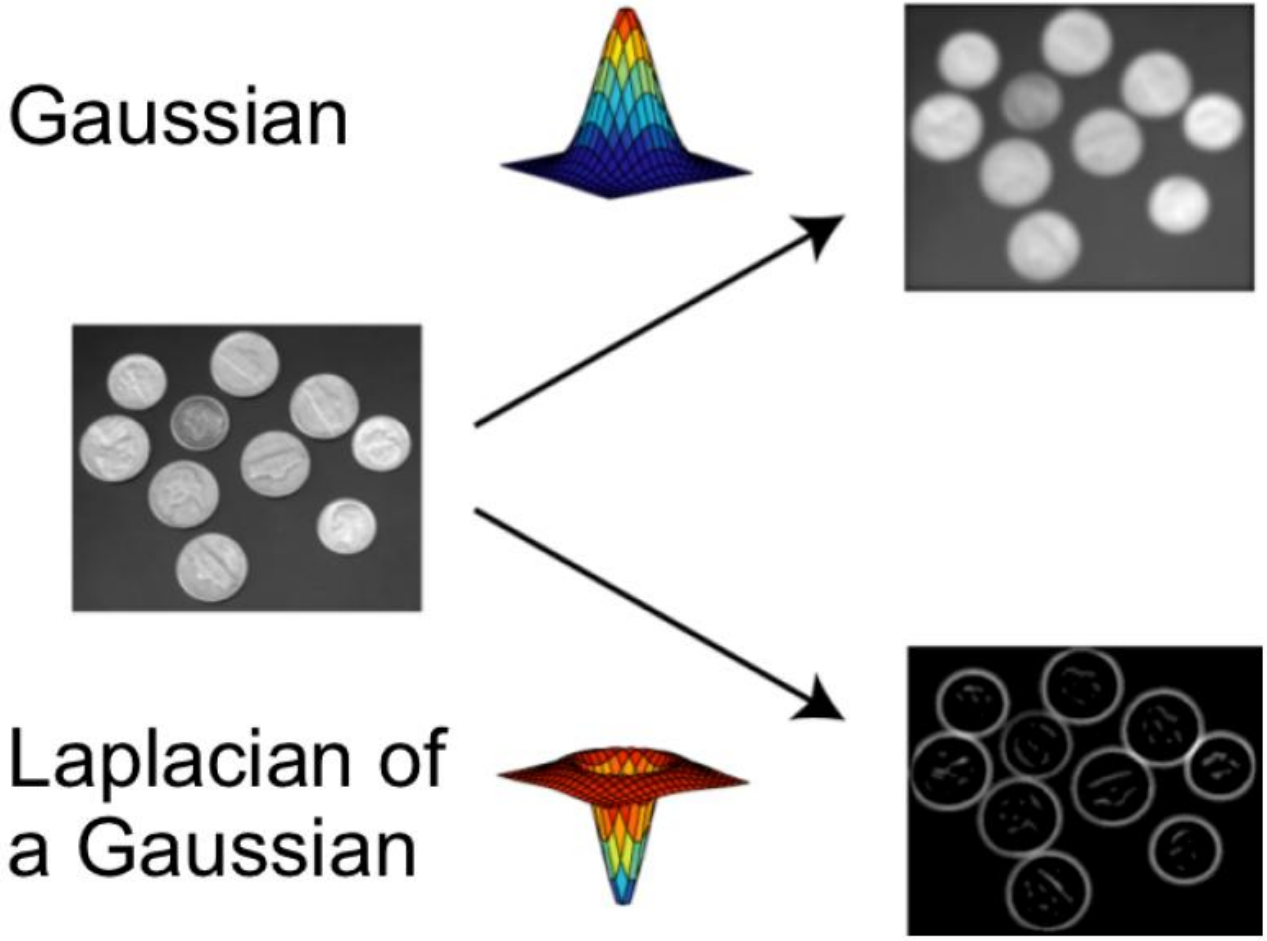


Feed-Forward network of a particular architecture.
Many equivalent “parts” (i.e. of the same structure with the same weights), but having different **Receptive Fields**. The output level of each part gives an information about the signal in the corresponding receptive field – computation of **local features**.

Problem: During the Error Back-Propagation the equivalence gets lost. Solution: average the gradients.

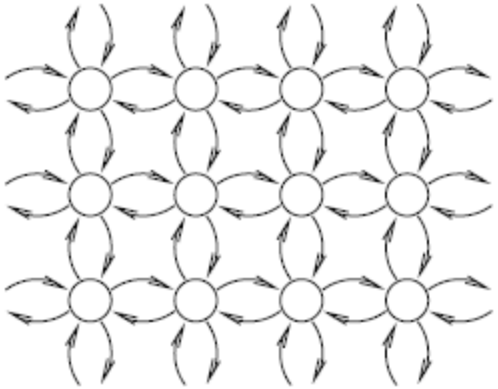
Convolutional Networks

Local features – convolutions with a set of predefined masks (see lectures “Image Processing”).



Hopfield Networks

There is a symmetric neighborhood relation (e.g. a grid).
The output of each neuron serves as inputs for the neighboring ones.



$$y_r = f\left(\sum_{r' \in N(r)} w_{rr'} \cdot y_{r'} + b_r \right)$$

with symmetric weights, i.e. $w_{rr'} = w_{r'r}$

A **network configuration** is a mapping $y : D \to \{0, 1\}$
A configuration is **stable** if "outputs y_r do not contradict"

The **Energy** of a configuration is

$$E(y) = \sum_{rr'} w_{rr'} \cdot y_r \cdot y_{r'} + \sum_r b_r \cdot y_r$$

Hopfield Networks

Network dynamic:

1. Start with an arbitrary configuration $y^{(0)}$,

2. Decide for each neuron whether it should be activated or not according to

$$y_r = f\left(\sum_{r' \in N(r)} w_{rr'} \cdot y_{r'} + b_r \right)$$

Do it **sequentially** for all neurons until convergence, i.e. apply the changes immediately.

In doing so the energy increases !!!

Attention!!! It does not work with the parallel dynamic (seminar).

Hopfield Networks

During the sequential dynamic the energy may only increase !

Proof:
Consider the energy "part" that depend on a particular neuron:

$$E_r(y) = \sum_{r' \in N(r)} w_{rr'} \cdot y_r \cdot y_{r'} + b_r \cdot y_r = y_r \cdot \left[\sum_{r' \in N(r)} w_{rr'} \cdot y_{r'} + b_r \right]$$

After the decision the energy difference is

$$E^{(t+1)}(y) - E^{(t)}(y) =$$
$$= \left(y_r^{(t+1)} - y_r^{(t)} \right) \cdot \left[\sum_{r' \in N(r)} w_{rr'} \cdot y_{r'} + b_r \right]$$

If $[\cdot] > 0$, the new output $y_r^{(t+1)}$ is set to 1 → energy grows.
If $[\cdot] < 0$, the new output $y_r^{(t+1)}$ is set to 0 → energy grows too.

Hopfield Networks

The network dynamic is the simplest method to find a configuration of the maximal energy (synonym – “Iterated Conditional Modes”).

The network dynamic is **not globally optimal**, it stops at a **stable** configuration, i.e. a **local** maxima of the Energy.

The **most stable** configuration – **global** maximum.

The task (find the global maximum) is NP-complete in general.

Polynomial solvable special cases:

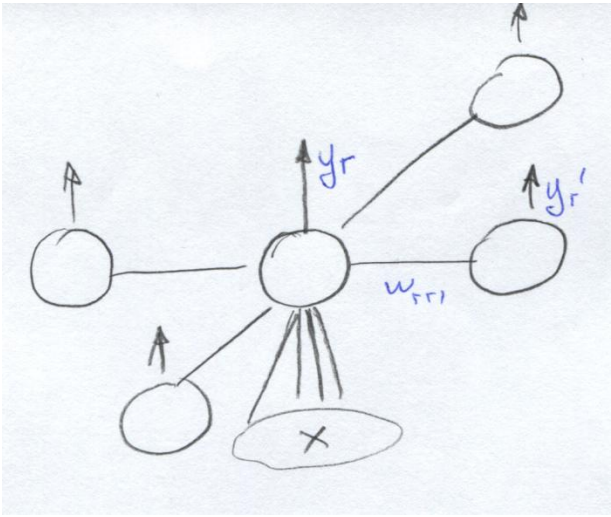
1. The neighborhood structure is simple – e.g. a tree
2. All weights $w_{rr'}$ are non-negative (supermodular energies).

Of course, nowadays there are many good approximations.

Hopfield Networks

Hopfield Network with external input x:

$$y_r = f\Big(\sum_{r' \in N(r)} w_{rr'} \cdot y_{r'} + b_r(x) \Big)$$



The energy is

$$E(y, x) = \sum_{rr'} w_{rr'} \cdot y_r \cdot y_{r'} + \sum_r b_r(x) \cdot y_r$$

Hopfield Networks implement mappings $X \rightarrow Y$ according to the principle of Energy maximum:

$$y = \arg\max_{y'} E(y', x)$$

Note: no single output but a configuration – **structured output**.

Hopfield Networks

Hopfield Networks model patterns – network configurations of the optimal energy.

Example:
Let y be a network configuration and $n(y)$ the number of "cracks" – pairs of neighboring neurons of different outputs.

Design a network (weights and biases for each neuron) so that the energy of a configuration is proportional to the number of cracks, i.e. $E(y) \sim -n(y)$.

Solution: $w_{rr'} = 2,\ b_r = -4$

Further examples at the seminar.