University of Osnabrück, Institute of Cognitive Science

Machine Learning

7 – Neural Networks

SS 2018

Gunther Heidemann

Overview

1. Biological motivation
2. Types of learning
3. Hebbian learning
4. Perceptron
5. Multilayer Perceptron

For the part on Hebbian learning, see, e.g.,

John Hertz, Anders Krogh, Richard G. Palmer:
Introduction to the Theory of Neural Computation, Addison-Wesley

Computer vs. brain

Computers:

- Fast sequential processing.
- High effort for parallelization to just a few processors.
- Symbolic processing without errors.
- No redundancy.
- Due to lack of software, very bad at pattern recognition, learning, robot control etc.
- Progress in AI fields where symbolic formalism is available, e.g., data base management or games.
- Adaptivity is implemented by software, not hardware.

Computer vs. brain

Brain:

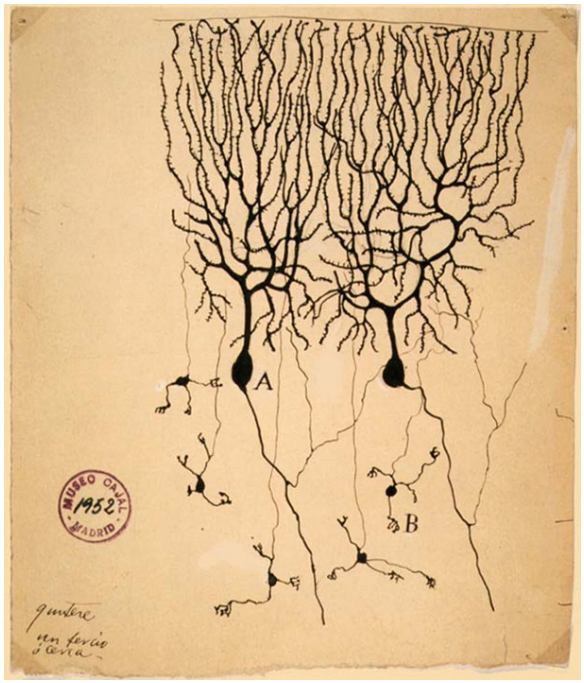
- Highly parallel: ~ 10^{10} neurons.
- 10^4-10^5 connections per neuron.
- Slow: Neuron switching time ~1 msec.
- Scene recognition in ~0.1 sec → only ~100 sequential steps, but highly parallel.
- High redundancy: Death of a neuron is compensated by others.
- High plasticity of hardware.
- Very good at tasks such as pattern recognition, robot control etc.
- Surprisingly bad at symbolic computation → high bias for the above tasks!

Idea of Neuroinformatics: Mimic neural principles by software !

Neurons

What we learn from neurobiology:

- Neurons have many inputs but only one output.
- Firing rate of neurons codes its activity.
- Activity is increased when the neuron gets input from other neurons via axons.
- The input from the axons is weighted differently by the synaptic weights.
- A neuron has a nonlinear activation function: Below a threshold, it will fire little, then there is an increase of its firing with the input, then saturation is reached.
- Best modeled by a sigmoid activation function.



Purkinje cells (A) and granule cells (B) from pigeon cerebellum [Ramon y Cajal, 1899]

Formal neurons

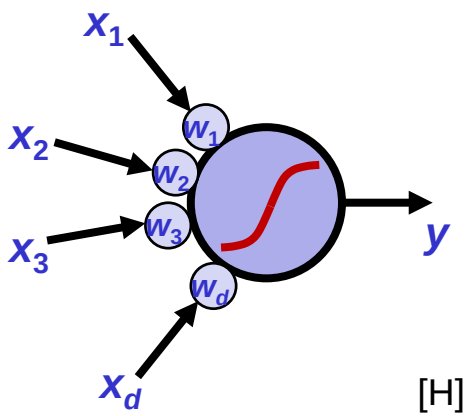
The basic neuron model used in Neuroinformatics is a very simple abstraction of the neurobiological findings:

- The activity communicated via an axon is represented as a single real number x, which can be interpreted as the spiking frequency. Phase is neglected.
- For a neuron with d inputs, we have $\vec{x} \in \mathbb{R}^d$.
- Each input x_i is weighted by a number w_i.
- The weighted inputs are summed up to obtain the activation s:

$$s = \sum_{i=1\ldots d} w_i x_i = \vec{w}\,\vec{x}, \qquad \vec{w} \in \mathbb{R}^d.$$

- The output y of the neuron is a sigmoid function (*activation function*) of the activation s:

$$y = y(s).$$



[H]

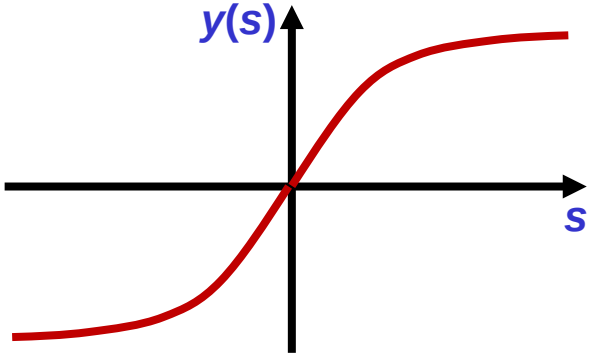
Activation functions

Some frequently used sigmoid activation functions:

Fermi function:

$$\sigma(s) = 1 \,/\, (1 + \exp(-a(s - s_0)), \quad a>0,$$

$$\sigma'(s) = a\, \sigma(s)\, (1 - \sigma(s)) \text{ (efficiency!).}$$



tanh:

$$\sigma(s) = \tanh(a(s - s_0)), \qquad a>0,$$

$$\sigma'(s) = a\, (1 - \sigma^2(s)) \qquad \text{(efficiency!).}$$

Grossberg's function:

$$\sigma(s) = \Theta(s - s_0) \cdot (s - s_0) \,/\, (s - s_0 + a), \quad a>0.$$

Most simple activation function is thresholding:

$$\sigma(s) = \Theta(s - s_0)$$

Types of learning

- Unsupervised learning:
  - No teacher, examples are unlabeled.
  - Effect of learning is coded in the learning rules.
- Supervised learning:
  - Teacher: Examples are labeled, e.g., with class information. Labeling may comprise many attribute values.
  - Learning is directed at mapping the input part of the examples (e.g., a stimulus) to the label as an output (e.g., a class).
- Reinforcement learning:
  - "Weak teacher": Agent tries to reach a goal and the teacher tells whether the goal has been reached.
  - The agent has to find the way by itself.
  - Example: Tennis ball reaches goal or not but there is no teacher to demonstrate the movement of the hand.

Types of learning

More realistic learning becomes popular:

Weakly supervised or semi-supervised learning.

- Pre-structuring of data by unsupervised learning.
- Use labeling information sparingly after unsupervised learning.

Classical conditioning

Ivan Pavlov, 1927.

Also: Pavlovian conditioning, respondent conditioning.

Experiment:

A dog is presented with food (unconditioned stimulus *US*) preceded by the sound of steps of the person who brings the food (conditioned stimulus *CS*, later replaced by ringing a bell).

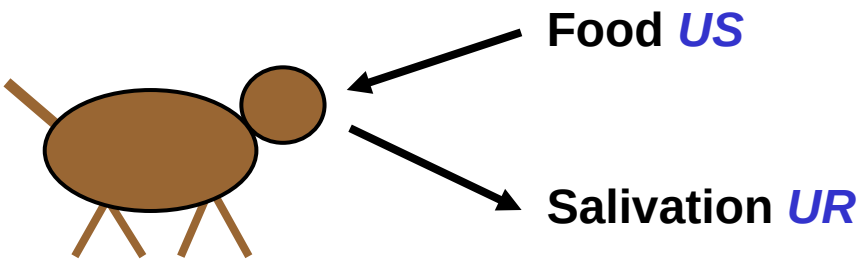
Initially, the dog reacts by salivation (unconditioned response *UR*) only to the food *US*.

After repeated presentations of *US* paired with *CS*, the dog reacts by salivation (conditioned response *CR*) already to the sound of steps / bell *CS*.

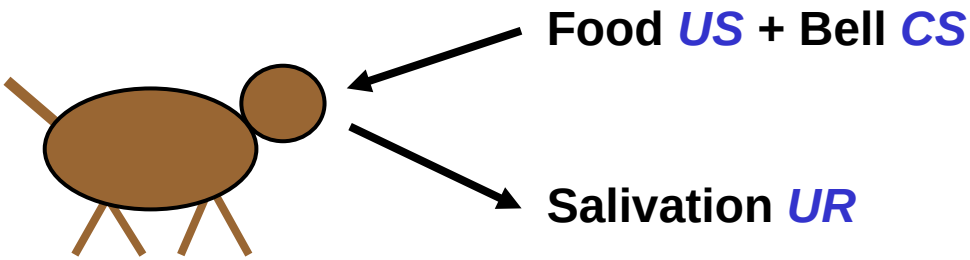
Thus the dog has learned the association of *US* and *CS* (to be more precise, it has observed the correlation and has inferred causality).

Classical conditioning

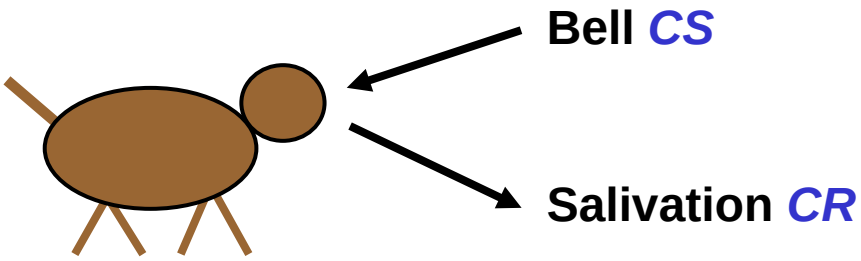
Natural reaction:



Learning phase:



After learning:



Hebbian learning

Donald Hebb, 1949:

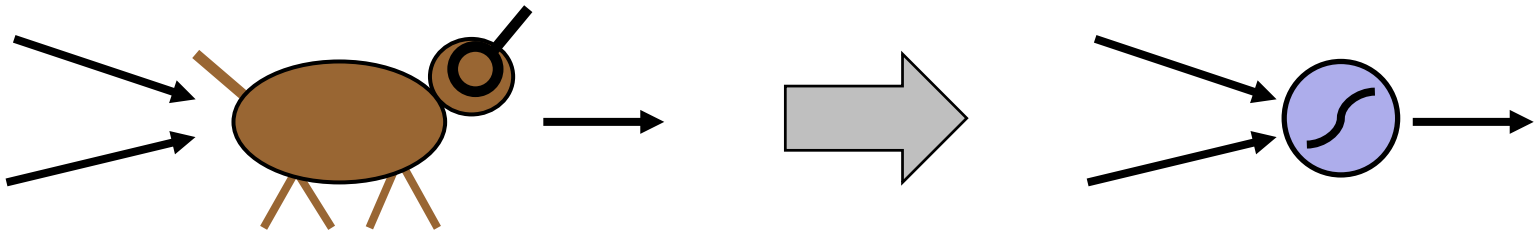
First idea for a basic mechanism to describe learning on the level of neurons by synaptic plasticity that can be easily formalized as an adaptation rule.

Principle: A synaptic weight of cell *B* is increased when the arriving axon from neuron *A* takes part in causing *B* to fire.

"Cells that fire together, wire together"

Thus, the correlation of the firing of *A* and *B* leads to increasing the weight.

This is a simple way to learn an association between different stimuli from cells *A* and *A'* arriving at *B*.



[H]

Hebbian learning

A neuron receives input signals x_i from other neurons.

To each input channel a weight w_i is assigned.

Hebb's rule (also: *cell assembly theory*) proposes to increase a weight w_i according to the product of the activity of the neuron and the input

$$\Delta w_i = \varepsilon \cdot y(\vec{x} \cdot \vec{w})\, x_i$$

with stepsize ε.

Classical conditioning requires two inputs (for *US* and *CS*):

$$\Delta w_1 = \varepsilon \cdot y(x_1 \cdot w_1 + x_2 \cdot w_2)\, x_1,$$

$$\Delta w_2 = \varepsilon \cdot y(x_1 \cdot w_1 + x_2 \cdot w_2)\, x_2,$$

where x_1 is the *US* and x_2 the *CS*.

Hebbian learning

Classical conditioning for a neuron with threshold activation function $\Theta(.)$, threshold ½ and learning rate $\varepsilon = 0{,}2$:

$$\Delta w_1 = \varepsilon \cdot \Theta(x_1 \cdot w_1 + x_2 \cdot w_2 - ½)\, x_1,$$

$$\Delta w_2 = \varepsilon \cdot \Theta(x_1 \cdot w_1 + x_2 \cdot w_2 - ½)\, x_2$$

Presence / absence of a stimulus *i*: $x_i = 1 / 0$.

Initial values are $w_1 = 1$ (*US* always causes response y=1) and $w_2 = 0$ (*CS* initially causes no response, y=0) .

x_1		1		0		1		1		0		1		0
x_2		0		1		1		1		1		1		1
y		1		0		1		1		0		1		1
w_1	1		1,2		1,2		1,4		1,6		1,6		1,8	
w_2	0		0		0		0,2		0,4		0,4		0,6	

US→*UR* *CS*→0 Learning *CS*→0 Learning *CS*→*CR*

time

Hebb's rule: Limit weight growth

Since weights never decrease, they become arbitrarily large:

$$\Delta\vec{w} = \varepsilon \cdot y(\vec{x} \cdot \vec{w})\, \vec{x}.$$

Several solutions:

- Decay term: $\Delta\vec{w} = \varepsilon \cdot y(\vec{x} \cdot \vec{w})\, \vec{x} - \lambda \cdot \vec{w}.$
- Dynamic normalization to $|\vec{w}| = 1$:

$$\Delta\vec{w} = \varepsilon \cdot y(\vec{x} \cdot \vec{w})\, \vec{x} - \lambda \cdot \vec{w} \cdot (\vec{w}^2 - 1).$$

- Explicit normalization to $|\vec{w}| = 1$:

$$\vec{w}^{new} = (\vec{w}^{old} + \Delta\vec{w}) \,/\, |\vec{w}^{old} + \Delta\vec{w}|.$$

- Oja's rule (Erkkii Oja, 1982) uses weight decay $\sim y^2$:

$$\Delta\vec{w} = \varepsilon \cdot y(\vec{x} \cdot \vec{w})\, (\vec{x} - y(\vec{x} \cdot \vec{w}) \cdot \vec{w}).$$

A stimulus that once has been learned but is presented no more will be forgotten since other stimuli become dominant.

Effect of Hebb's rule for a pair of neurons

Consider the connection w_{ij^*} of neuron i to a particular neuron j^*:

$$y_i = y_i\left(w_{ij^*} \cdot y_{j^*} + \Sigma_{j,\, j \neq j^*} w_{ij} \cdot y_j\right)$$

(i is also connected to other neurons $j \neq j^*$.)

Hebbian learning affects the selected weight w_{ij^*} as

$$\Delta w_{ij^*} = \varepsilon \cdot y_i \cdot x_{j^*} - \lambda \cdot w_{ij^*} ,$$

where the input $(\vec{x})_{j^*}$ is the output of neuron j^*:

$$\Delta w_{ij^*} = \varepsilon \cdot y_i \cdot y_{j^*} - \lambda \cdot w_{ij^*} .$$

After repeated presentation of samples of the same set, equilibrium is reached. Averaged over time (steps) t, w_{ij^*} does not change any more:

$$<\Delta w_{ij^*}>_t = <\varepsilon \cdot y_i \cdot y_{j^*} - \lambda \cdot w_{ij^*}>_t = 0,$$

$$\Rightarrow \quad <w_{ij^*}>_t = (\varepsilon/\lambda) <y_i \cdot y_{j^*}>_t ,$$

i.e., w_{ij^*} is proportional to the **correlation** of the two neurons activities.

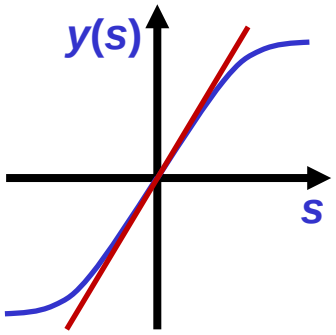
Effect of Hebb's rule on the weight vector

From a data set $D = \{\vec{x}_1, \vec{x}_2, \ldots\}$, $\vec{x}_i \in \mathbb{R}^d$ we randomly draw samples and train the weights $\vec{w} \in \mathbb{R}^d$ using Hebb's rule:

$$\Delta\vec{w} = \varepsilon \cdot y(\vec{x}^\top \cdot \vec{w})\, \vec{x}.$$

Assume the data are such that the activation remains within the linear range of the activation function y, so it can be approximated by a linear function:

$$\Delta\vec{w} = \varepsilon \cdot (\vec{x}^\top \cdot \vec{w}) \cdot \vec{x} = \varepsilon \cdot \vec{x} \cdot (\vec{x}^\top \cdot \vec{w}).$$



Average over many learning steps:

$$\langle \Delta\vec{w} \rangle = \varepsilon \cdot \langle (\vec{x} \cdot \vec{x}^\top) \cdot \vec{w} \rangle$$

For small variance:

$$\langle \Delta\vec{w} \rangle \approx \varepsilon \cdot \langle (\vec{x} \cdot \vec{x}^\top) \rangle \cdot \vec{w}$$

$$= \varepsilon \cdot C \cdot \vec{w}.$$

Effect of Hebb's rule on the weight vector

To understand the development of $\vec{w}$, divide

$$<\Delta\vec{w}> \quad = \quad \varepsilon \cdot C \cdot \vec{w}$$

by the "duration" Δt of an adaptation step

$$<\Delta\vec{w}> / \Delta t \quad = \quad \varepsilon / \Delta t \cdot C \cdot \vec{w},$$

which yields the differential equation

$$d\vec{w} / dt \quad \approx \quad \varepsilon / \Delta t \cdot C \cdot \vec{w},$$

$$= \quad \alpha \cdot C \cdot \vec{w} \qquad \text{with} \quad \alpha := \varepsilon / \Delta t.$$

The differential equation shows the "temporal" development of $\vec{w}$ (t is the step counter). The solution is

$$\vec{w}(t) \quad = \quad \exp(\alpha \cdot C \cdot t) \cdot \vec{w}(0) \; .$$

Effect of Hebb's rule on the weight vector

To understand the development of the solution

$$\vec{w}(t) = \exp(\alpha \cdot C \cdot t) \cdot \vec{w}(0),$$

we represent $\vec{w}(t)$ by an expansion after the eigenvectors of C:

$$\vec{w}(t) = \sum_{i=1\ldots d} a_i(t)\, \vec{v}_i,$$

$$C\vec{v}_i = \lambda_i \vec{v}_i \qquad \text{with} \qquad \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d > 0.$$

The time dependence is now in the coefficients $a_i(t)$.

With the expansion, the solution reads

$$\vec{w}(t) = \exp(\alpha \cdot C \cdot t) \cdot \sum_{i=1\ldots d} a_i(0)\, \vec{v}_i$$

$$= \sum_{i=1\ldots d} a_i(0) \exp(\alpha \cdot C \cdot t)\, \vec{v}_i$$

$$= \sum_{i=1\ldots d} a_i(0) \exp(\alpha \cdot \lambda_i \cdot t)\, \vec{v}_i,$$

since $f(C)\vec{v}_i = f(\lambda_i)\vec{v}_i$ for an expandable function f because $C^n\vec{v}_i = \lambda_i^n \vec{v}_i$.

Effect of Hebb's rule on the weight vector

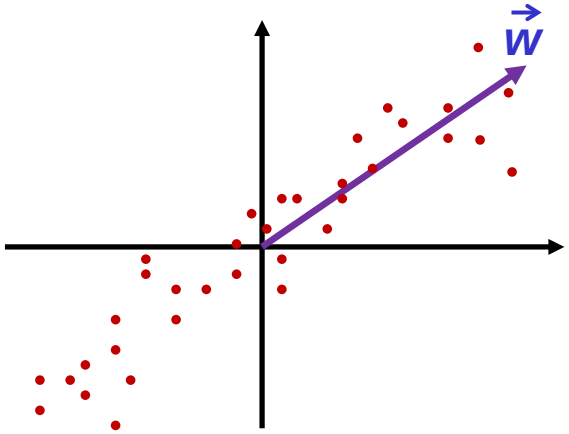
$$\vec{w}(t) = \sum_{i=1\ldots d} a_i(0)\exp(\alpha\cdot\lambda_i\cdot t)\,\vec{v}_i$$

$$= \exp(\alpha\cdot\lambda_1\cdot t)\left(a_1(0)\vec{v}_1 + \sum_{i=2\ldots d} a_i(0)\exp(-\alpha\cdot(\lambda_1-\lambda_i)\cdot t)\,\vec{v}_i\right)$$

For $t\to\infty$ we get $\exp(-\alpha\cdot(\lambda_1-\lambda_i)\cdot t)\to 0$, so the sum vanishes and

$$\lim_{t\to\infty} \vec{w}(t) \,/\, |\vec{w}(t)| = \vec{v}_1,$$

i.e., the weight vector converges to the eigenvector of C with the largest eigenvalue.



direction of largest variance

[H]

Effect of Hebb's rule on the weight vector

Compare to winner-takes-all rule:

WTA: $\Delta\vec{w}_{kl} = \varepsilon\,(\vec{x} - \vec{w}_{kl})$ with $\vec{w}_{kl}$ as best match.

Hebb: $\Delta\vec{w} = \varepsilon \cdot y(\vec{x}\cdot\vec{w})\,\vec{x}$.

WTA is local, i.e., a weight vector is affected only by stimuli in its neighborhood.

A Hebb-trained neuron is affected by all stimuli.

E.g., for $|\vec{w}|$ small, a "close" input ($|\vec{x}|$ small) has *less* effect than the same input stretched by $c>1$: $c\cdot\vec{x}$.

This is why $\vec{w}$ directs itself according to data variance.

Habituation

What happens if we modify Hebb's rule to an "Anti-Hebb rule"

$$\Delta\vec{w} = -\varepsilon \cdot y(\vec{x}^{\top} \cdot \vec{w}) \cdot \vec{x} \quad ?$$

To understand the effect, consider a stationary input $\vec{x}$ and apply the linearized version of the anti Hebb rule:

$$\Delta\vec{w} = -\varepsilon \cdot (\vec{x}^{\top} \cdot \vec{w}) \cdot \vec{x}$$

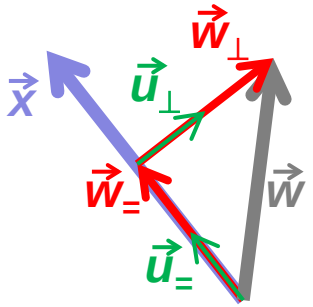
$\vec{w}$ has a component $\vec{w}_{=}$ parallel to $\vec{x}$ and a component $\vec{w}_{\perp}$ perpendicular to $\vec{x}$:

$$\vec{w} = \vec{w}_{=} + \vec{w}_{\perp}.$$

We define unit vectors in the directions $\vec{w}_{=}$ and $\vec{w}_{\perp}$:

$$\vec{u}_{=} = \vec{w}_{=} / |\vec{w}_{=}|,$$

$$\vec{u}_{\perp} = \vec{w}_{\perp} / |\vec{w}_{\perp}|.$$



[H]

Habituation

Decompose $\vec{w}$ into components along $\vec{u}_=$ and $\vec{u}_\perp$:

$$\vec{w} = a_= \vec{u}_= + a_\perp \vec{u}_\perp .$$

Observe the development of both components in separation under the influence of the ever repeated stimulus $\vec{x}$:

$$\Delta\vec{w} = \Delta a_= \vec{u}_= + \Delta a_\perp \vec{u}_\perp .$$

Note $\vec{u}_=$ and $\vec{u}_\perp$ do not change by definition as $\vec{x}$ is constant !

The change of $\vec{w}$ is represented by the coefficients.

With the learning rule we get

$$\Delta\vec{w} = \Delta a_= \vec{u}_= + \Delta a_\perp \vec{u}_\perp = -\varepsilon \cdot (\vec{x}^{\,\mathsf{T}} \cdot \vec{w}) \cdot \vec{x}$$

$$= -\varepsilon \cdot \left(\vec{x}^{\,\mathsf{T}} \cdot (a_= \vec{u}_= + a_\perp \vec{u}_\perp)\right) \cdot \vec{x}$$

$$= -\varepsilon \cdot \left(a_= \underbrace{\vec{x}^{\,\mathsf{T}} \cdot \vec{u}_=}_{|\vec{x}|} + a_\perp \underbrace{\vec{x}^{\,\mathsf{T}} \cdot \vec{u}_\perp}_{0}\right) \cdot \underbrace{\vec{x}}_{|\vec{x}|\vec{u}_=}$$

$$= -\varepsilon \cdot a_= \cdot |\vec{x}|^2 \cdot \vec{u}_= .$$

Habituation

$$\Rightarrow \quad \Delta a_{=} = -\varepsilon \cdot a_{=} \cdot |\vec{x}|^2,$$

$$\Delta a_{\perp} = 0.$$

The $\vec{w}$-component orthogonal to $\vec{x}$ does not change at all!

To understand how the $\vec{w}$-component parallel to $\vec{x}$ evolves we transform the learning rule again to a differential equation:

$$da_{=}/dt \approx \Delta a_{=}/\Delta t = -\varepsilon \cdot a_{=} \cdot |\vec{x}|^2$$

for small ε and $\Delta t = 1$.

A solution is

$$a_{=}(t) = a_{=}(0) \cdot \exp(-\varepsilon |\vec{x}|^2 t),$$

so $a_{=}(t)$ decreases exponentially!

Only the $\vec{w}$-component orthogonal to $\vec{x}$ remains.

Habituation

The anti Hebb rule implements **habituation**:

- As $\vec{w}$ becomes orthogonal to a repeated stimulus $\vec{x}$, this stimulus no longer leads to activation of the neuron since $\vec{x} \cdot \vec{w} = 0$.
- $\vec{w}$ filters out new stimuli.
- If several stimuli $\vec{x}_1, \vec{x}_2, \ldots \vec{x}_n$, $n<d$, are presented repeatedly, only the $\vec{w}$-component orthogonal to $\text{span}\{\vec{x}_1, \vec{x}_2, \ldots \vec{x}_n\}$ remains.
- So only stimuli $\vec{x} \in V$ with $V \perp \text{span}\{\vec{x}_1, \vec{x}_2, \ldots \vec{x}_n\}$ can “pass” the filter, i.e., activate the neuron.

Extracting more PCs using Hebb's rule

So far: A single Hebb-neuron extracts $\vec{v}_1$.

To extract more principal components from D, there are two ways:

1. Successive application of single Hebb neurons:
   - Extract $\vec{v}_1$.
   - Project D to the space orthogonal to $\vec{v}_1$:

$$\vec{x}_i' = \vec{x}_i - (\vec{x}_i \cdot \vec{v}_1)\, \vec{v}_1.$$

   - Extract $\vec{v}_2$ by the same method etc.
2. Method proposed by Sanger, 1989:
   - Chain of laterally coupled neurons, each trained by Hebb's rule.
   - Only the first neuron of the chain gets the unfiltered input. The second gets the input "minus" the direction of the first weight vector, the third "minus" the direction of the first and second weight vector etc.

Extracting more PCs using Hebb's rule

Extract PCs ordered by EVs by a chain of neurons (Sanger, 1989):

$$\Delta\vec{w}_i = \varepsilon \cdot y_i \cdot (\vec{x} - \Sigma_{k=1\ldots i}\, y_k \vec{w}_k),$$

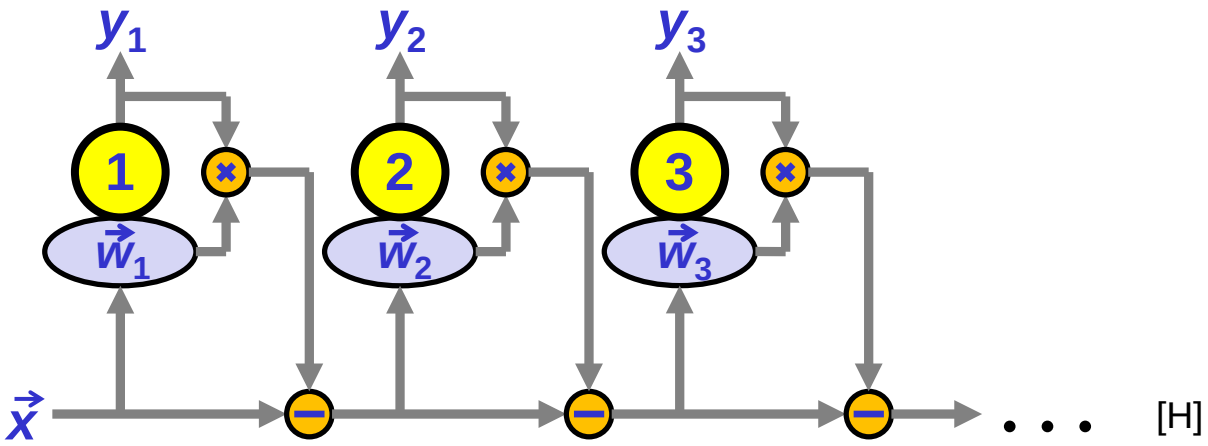
with linear activation $y_i = \vec{w}_i \cdot \vec{x}$,

where *i* is the number of the neuron.

For a single neuron $\Delta\vec{w}_1 = \varepsilon \cdot y_1 \cdot (\vec{x} - y_1\vec{w}_1)$,

it reduces to Oja's rule. "Lateral interaction" provides each of the following neurons with the input filtered by its predecessors, e.g.,

$$\Delta\vec{w}_2 = \varepsilon \cdot y_2 \cdot (\vec{x} - y_1\vec{w}_1 - y_2\vec{w}_2).$$



[H]

Extracting more PCs using Hebb's rule

Properties of the two methods:

1. Successively building the chain from Hebb neurons trained in isolation:
   - Each neuron gets only "correct" input when trained, i.e., the other PCs are already filtered out.
   - So the single neuron needs less steps.
2. Training the entire chain simultaneously after Sanger's rule:
   - Each step requires training of the complete chain.
   - As long as the neurons in the beginning of the chain are not sufficiently trained, the later neurons receive input where the PCs with larger eigenvalues are not filtered out correctly.
   - But there is no need to detect the moment when training of a neuron is sufficient since the following neurons are trained all the time.

Summary on Hebbian learning

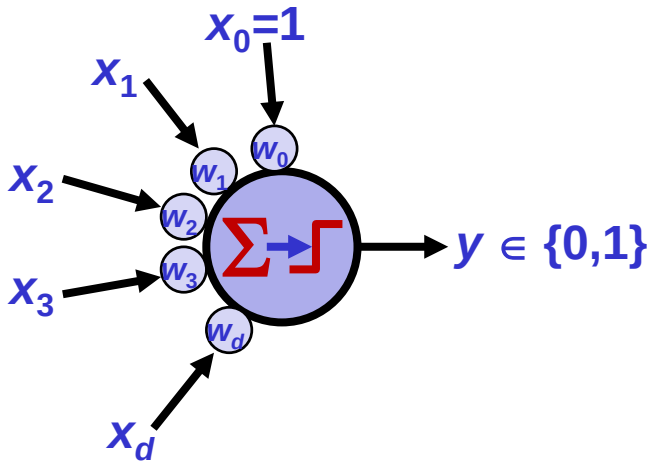
- Unsupervised learning by a simple rule to modify the weights of a single neuron.
- Motivation: Classical conditioning.
- Hebb-neuron learns association of stimuli from correlation.
- Weight vector directs itself in the direction of the principal component of the data distribution with largest eigenvalue.
- Anti-Hebb rule implements habituation, weight vector becomes orthogonal to all repeated stimuli.
- Thus new stimuli can be filtered out.
- PCA is possible by a chain of Hebb neurons, where earlier neurons filter PCs of large eigenvalues out of the input for the training of the later neurons.
- Chain can be trained either successively or all at once using a combination of the Hebb rule with a lateral anti Hebb rule.

The Perceptron

Perceptron

One of the most prominent neural networks for supervised learning [Rosenblatt, 1958].

Practical use is limited, but the perceptron is the building block of the multilayer perceptron.



[H]

Input: $D = \{\vec{x}^1, \vec{x}^2, \ldots\}, \vec{x}^i \in \mathbb{R}^d$

Activation: $s = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n$.

Output: $y = \begin{cases} 1 & \text{if } s \geq 0 \\ 0 & \text{else} \end{cases}$

Perceptron

w_0 acts as a threshold.

To simplify notation, add a constant component to the input vector:

$$\vec{x} = \{1, x_1, x_2, \ldots, x_d\}^T \in \mathbb{R}^{d+1}.$$

With

$$\vec{w} = \{w_0, w_1, w_2, \ldots, w_d\}^T$$

the activation is

$$s = \vec{x} \cdot \vec{w},$$

and the output

$$y = \Theta(\vec{x} \cdot \vec{w}).$$

Perceptron training rule

The perceptron is trained iteratively using a labeled data set $D = \{(\vec{x}^n, t^n)\}$, $\vec{x}^n \in \mathbb{R}^d$, where t^n is the target value (i.e., the label, class, etc. provided by the teacher) for input $\vec{x}^n$.

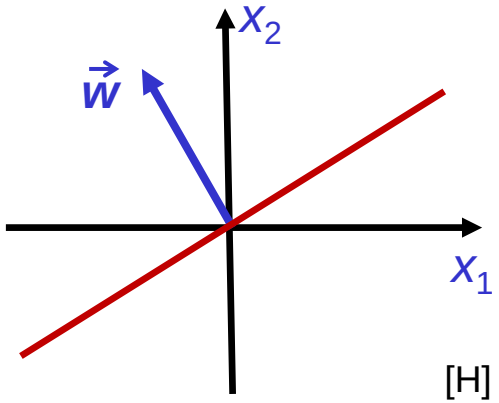
Perceptron training rule:

$$\Delta\vec{w} = \varepsilon\,(t - y)\,\vec{x},$$

ε is a small learning rate. Convergence can be shown if ε is sufficiently small and if the task is solvable by a perceptron.

The perceptron can represent the hypotheses space $\{\vec{w} \mid \vec{w} \in \mathbb{R}^{d+1}\}$.

The "decision surface" represented by a perceptron is the $d-1$ dimensional hyperplane orthogonal to $\vec{w}$. Data on the "positive side" of the hyperplane with $\vec{x}\cdot\vec{w}>0$ get $y=1$, the rest 0.

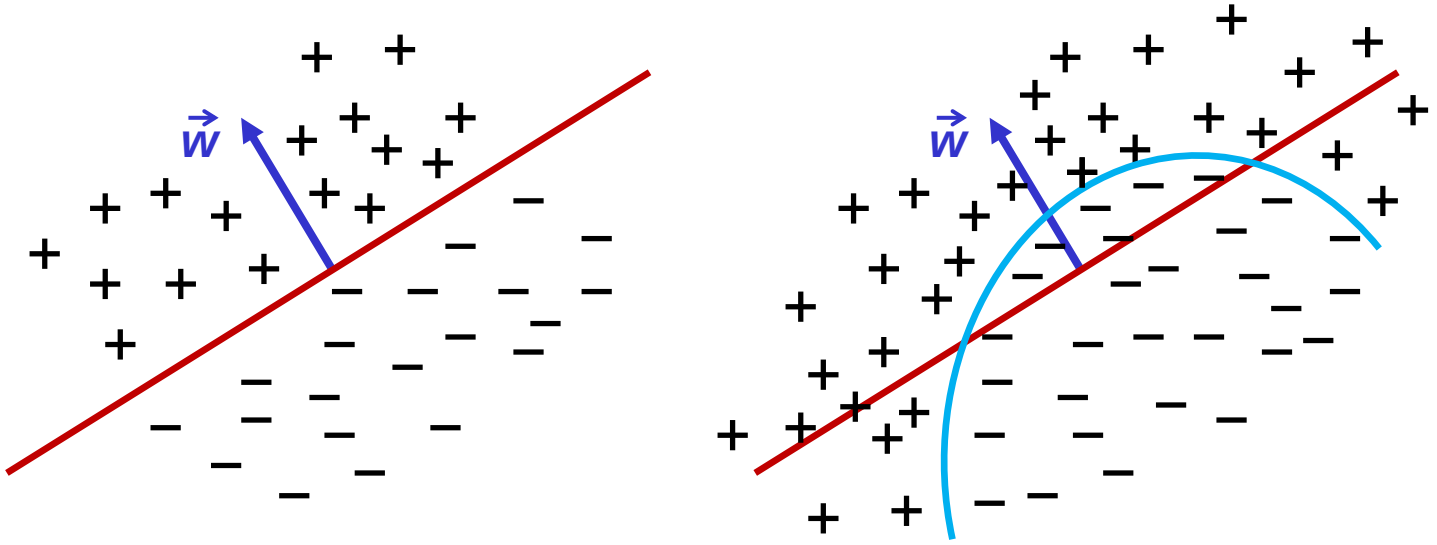


[H]

Perceptron: Class separability

Question: Under which conditions can a perceptron solve a classification task?

The data must be *linearly separable*, i.e., the two classes must be on the two sides of a hyperplane.



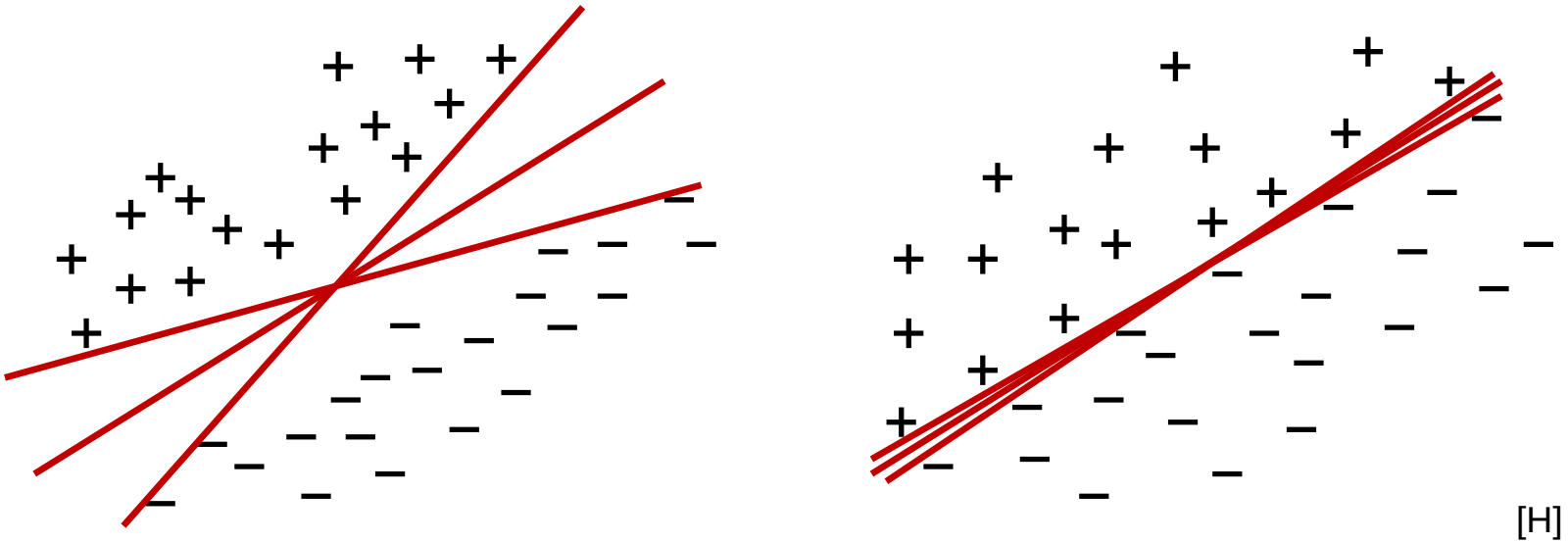
[H]

Linear separatrix exists

Nonlinear separatrix required

Perceptron: Convergence speed

Provided classes are linearly separable, there are still easy and difficult separation tasks:



[H]

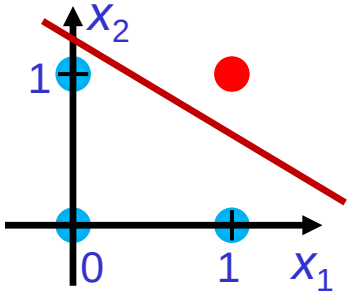
Large hypothesis space

Small hypothesis space

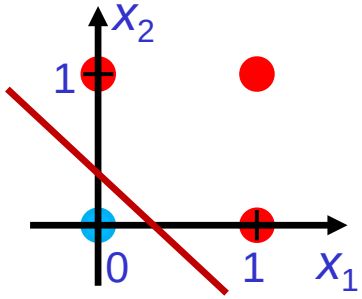
A small hypothesis space requires more training steps until convergence is achieved.

Perceptron: Logic operations

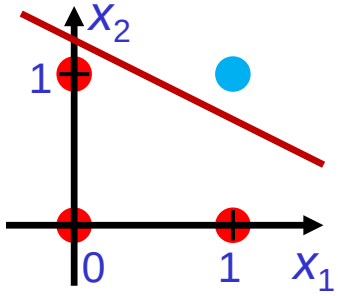
A perceptron with two boolean inputs (and the 1-component) can implement AND, OR, NAND and NOR but **not XOR**:



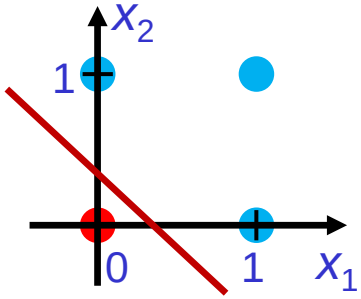
AND



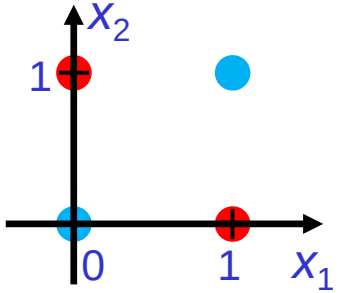
OR



NAND



NOR

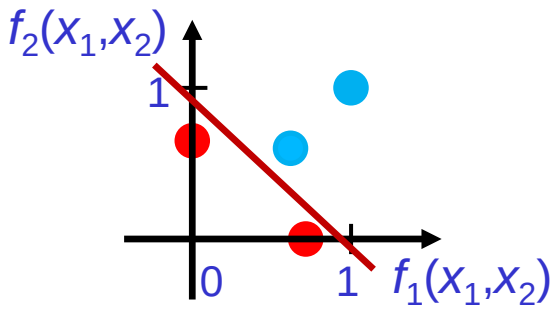


XOR

[H]

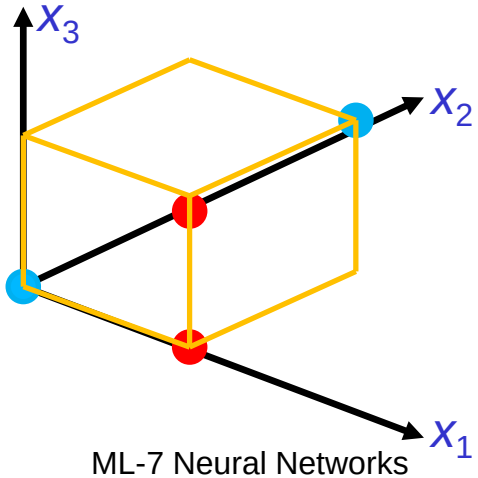
Perceptron: XOR

The XOR-problem can be solved by "distortion" of the input space to obtain a "feature space", i.e., a nonlinear transform $\vec{f}(\vec{x})$ of the inputs:



[H]

Alternatively, $x_3 = x_1 \cdot x_2$ can be added as another input channel:



[H]

Perceptron: Learning rule from error function

The learning rule can be derived from minimizing the mean square error:

$$E[\vec{w}] = \tfrac{1}{2} \sum_{i=1\ldots|D|} \left(t^i - y(\vec{x}^i)\right)^2 .$$

Gradient descent:

$$\begin{aligned}
\Delta\vec{w} &= -\varepsilon\, \partial E / \partial \vec{w} \\
&= -\varepsilon\, \partial / \partial \vec{w}\, \tfrac{1}{2} \sum_{i=1\ldots|D|} \left(t^i - y(\vec{x}^i)\right)^2 \\
&= -\varepsilon\, \tfrac{1}{2} \sum_{i=1\ldots|D|} \partial / \partial \vec{w} \left(t^i - y(\vec{x}^i)\right)^2 \\
&= -\varepsilon\, \tfrac{1}{2} \sum_{i=1\ldots|D|} 2\left(t^i - y(\vec{x}^i)\right) \partial / \partial \vec{w} \left(t^i - y(\vec{x}^i)\right) \\
&= -\varepsilon \sum_{i=1\ldots|D|} \left(t^i - y(\vec{x}^i)\right) \partial / \partial \vec{w} \left(t^i - \vec{w}\cdot\vec{x}^i\right) \\
&= -\varepsilon \sum_{i=1\ldots|D|} \left(t^i - y(\vec{x}^i)\right) \left(-\vec{x}^i\right).
\end{aligned}$$

Stochastic approximation:

$$\Delta\vec{w} = \varepsilon \left(t^i - y(\vec{x}^i)\right) \vec{x}^i .$$

Perceptron: Batch mode and incremental mode

The stochastic approximation is also called *incremental mode* (in contrast to *batch mode*):

Batch mode:

$$\Delta\vec{w} = \varepsilon \sum_{i=1\ldots|D|} (t^i - y(\vec{x}^i))\, \vec{x}^i.$$

Gradient descent with respect to the entire training set D.

Incremental mode:

$$\Delta\vec{w} = \varepsilon\, (t^i - y(\vec{x}^i))\, \vec{x}^i.$$

Gradient descent with respect to only one example $(\vec{x}^i, t^i)$ at a time.

For sufficiently small ε, incremental gradient descent can be shown to approximate batch gradient descent arbitrarily close.

Perceptron: Summary

- Simple supervised learning rule for a single neuron.
- Learning rule can be derived from mean square error function.
- Convergence proven.
- Large margin between linearly separable classes leaves many possible weight vectors (large hypothesis space), so the task is easier and convergence is faster.
- Can only solve linearly separable tasks.
- Most famous example for nonlinear two-class separation is XOR.
- XOR can not be solved by perceptron except with highly problem specific add-ons.
- This led to a temporary drop in the interest in ANN !

Milestones in the history of ANN

- Warren McCulloch & Walter Pitts, 1943: Simple neuron model for logic operations.
- Donald Hebb, 1949: Associative learning rule.
- Frank Rosenblatt, 1958: Convergence of perceptron.
- Marvin Minsky & Seymour Papert 1969: Perceptron can not represent XOR (and other problems).

→ little interest in ANN up to …

- … John Hopfield, 1985: Hopfield-network, a recurrent network that can solve optimization tasks such as travelling salesman.
- David Rumelhart, Geoffrey Hinton & Ronald Williams 1986: Backpropagation algorithm for training a multilayer perceptron that solve XOR (but Paul Werbos proposed backprop. in 1974).
- 1982: Self-organizing maps (Teuvo Kohonen)
- Since 2006: Deep ANN

Multilayer Perceptron

Multilayer perceptron

Also: MLP; backpropagation network (after the training algorithm).

Idea:

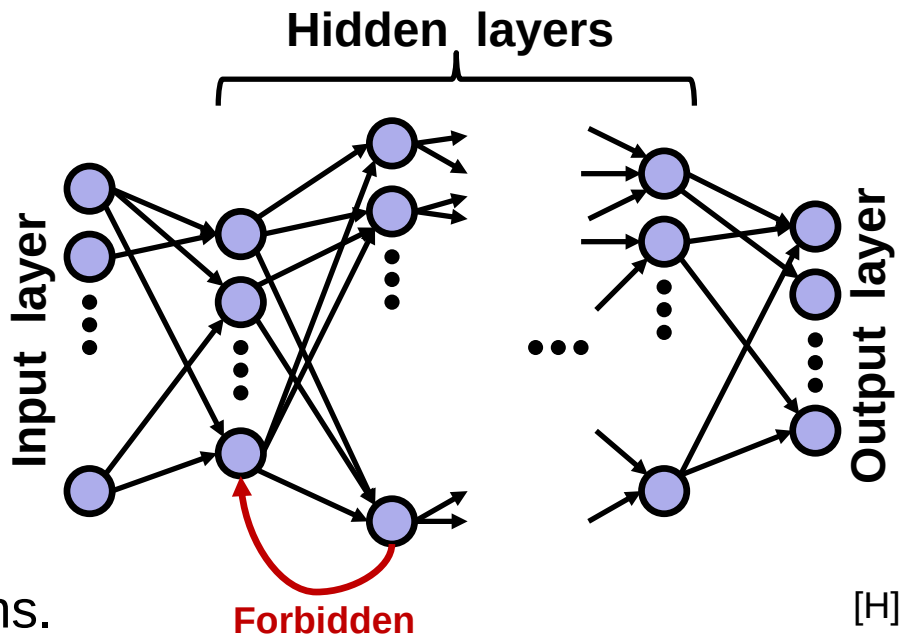
- Solve nonlinear separation tasks
  - with nonlinear separatrices
  - with classes covering disjoint areas

  by combining several perceptrons.
- In addition, generalize to several outputs.

Multilayer perceptron

MLP architecture:

- Input layer with one neuron for each component of the input vector $\vec{x}$. The input "neurons" do nothing but represent the input.
- At least one hidden layer.
- Hidden layers may have different numbers of neurons.



[H]

- Number of output neurons is dimensionality of the output vector $\vec{y}$.
- Feed forward architecture: Only connections from layer k to layer i with $k<i$.
- Generalization to a directed graph: Connections skipping layers are allowed. This case is not considered here since such connections can be represented by additional linear neurons.

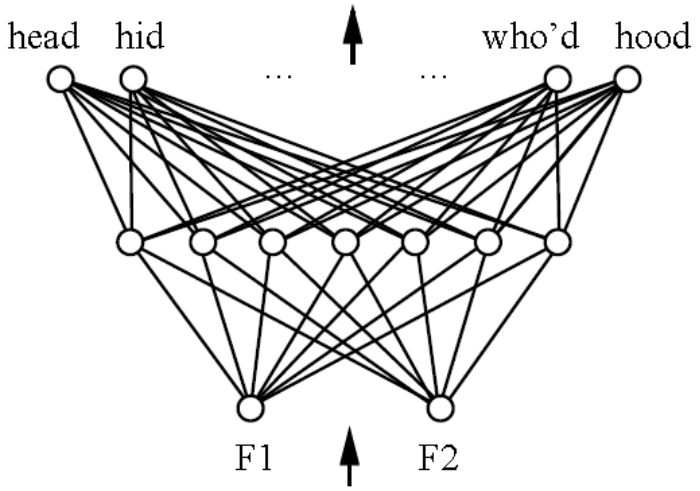
Multilayer perceptron

Example:

Task: Classify 10 vowel sounds spoken into microphone in a particular context (h_d).

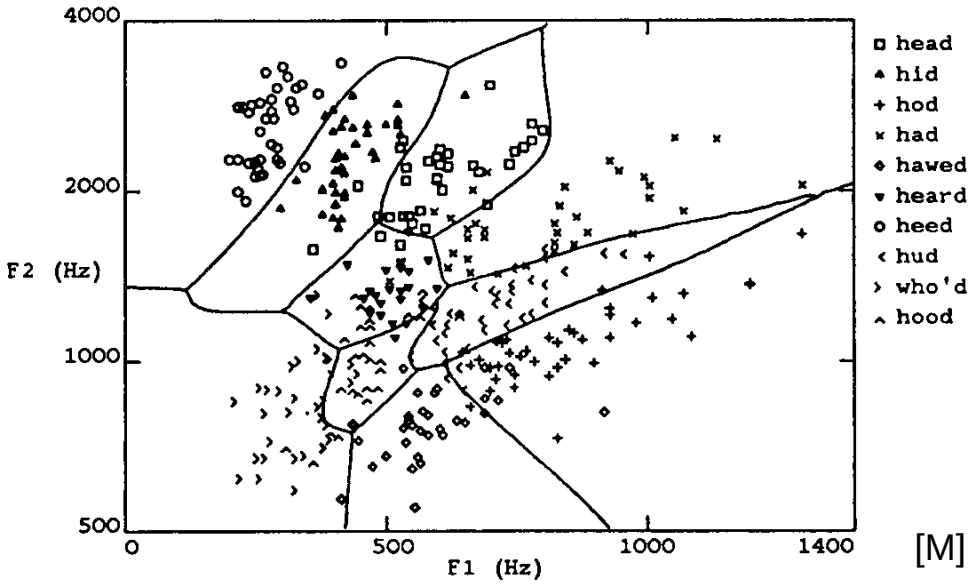
Input: 2 spectral features F1 and F2.

Output: 10 outputs, one for each sound.



Above: Network architecture

Below: Decision boundaries represented by the net in the 2d feature space.

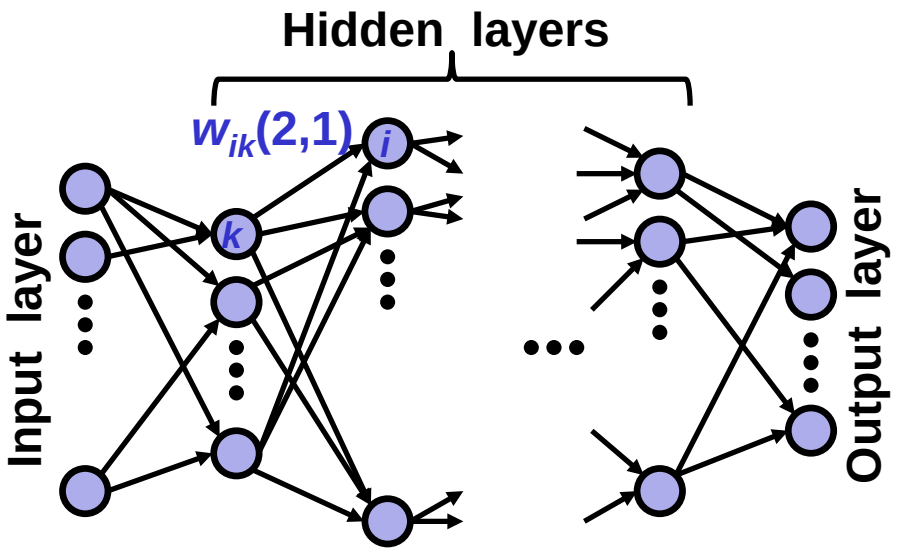


[Haung and Lippmann, 1988]

[M]

Multilayer perceptron

- $\vec{x} \in \mathbb{R}^{d_{in}}$, $\vec{y} \in \mathbb{R}^{d_{out}}$
- Layer 0: Input layer.
- L_H: # hidden layers.
- Layer L_H+1: Output layer.
- Neurons have numbers 1…$N(i)$ in layer i.
- $o_i(k)$: Output of neuron i in layer k.
- $x_i = o_i(0)$.
- $y_i = o_i(L_H+1)$.



[H]

Layer numbers:

0 1 2 … L_H L_H+1

- $w_{ik}(m,n)$: Weight from neuron k in layer n to neuron i in layer m.

Multilayer perceptron

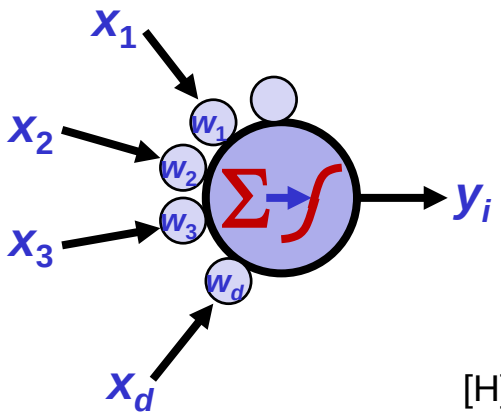
Neuron:

Linear activation function would not do, since a succession of linear transforms (represented by the layers) can be replaced by a single linear transform:

$$\vec{y} = \underbrace{M_1 M_2 M_3 \dots M_n}_{M} \vec{x}, \quad M_1 \in \mathbb{R}^{d_{out} \times a}, \quad M_n \in \mathbb{R}^{b \times d_{in}}, \quad a,b \in \mathbb{N},$$
$$= M \vec{x}, \quad M \in \mathbb{R}^{d_{out} \times d_{in}}.$$

→ Use sigmoid activation function σ :

- Step function enforces a “decision”.
- Soft step required because backpropagation algorithm needs differentiability.
- *Squashing*: Maps the incoming information to a much smaller range



[H]

Error function and learning rule

Error function is the same as for a simple perceptron:

$$E[\{w\}] \quad = \quad \tfrac{1}{2} \sum_{i=1\ldots|D|} (\vec{t}^{\,i} - \vec{y}(\vec{x}^{\,i}))^2.$$

Minimization by gradient descent:

$$\Delta w_{ik}(m,n) \quad = \quad -\varepsilon \, \partial E / \partial w_{ik}(m,n).$$

The problem is to compute the derivatives to w:

$$\partial E / \partial w_{ik}(m,n) \quad = \quad \tfrac{1}{2} \sum_{i=1\ldots|D|} 2(\vec{t}^{\,i} - \vec{y}(\vec{x}^{\,i})) \, \partial I / \partial w_{ik}(m,n) \, (\vec{t}^{\,i} - \vec{y}(\vec{x}^{\,i}))$$

$$= \quad -\sum_{i=1\ldots|D|} (\vec{t}^{\,i} - \vec{y}(\vec{x}^{\,i})) \, \partial I / \partial w_{ik}(m,n) \, \vec{y}(\vec{x}^{\,i}).$$

Note $\partial I / \partial w_{ik}(m,n) \, \vec{y}(\vec{x}^{\,i})$ requires all derivatives on the path from $w_{ik}(m,n)$ to the output layer!

The error-backpropagation algorithm provides a scheme for the efficient computation of all required derivatives and weight adaptations.

Backpropagation algorithm

Given: Labeled training samples $D = \{(\vec{x}^n, \vec{t}^n)\}$, $\vec{x}^n \in \mathbb{R}^{d_{in}}$, $\vec{t}^n \in \mathbb{R}^{d_{out}}$.

Algorithm (idea):

1. Set small random intial values for all weights $w_{ik}(m,n)$.
2. Interate the following steps until a stop criterion is met for samples $(\vec{x}, \vec{t})$, randomly chosen from D:
   1. Propagate input $\vec{x}$ **forward** through the net, i.e., compute from layer 1 to layer L_H+1 the neuron outputs to obtain the output $\vec{y}$.
   2. Compute the error $t_i - y_i(\vec{x})$ between the actual output and the desired target output for each output component i.
   3. Propagate the errors **backward** through the net (from layer L_H+1 to layer 1) to find which weights are "responsible".
   4. Update the weights.

Steps 3 and 4 are described in the following.

Backpropagation algorithm

Denote the *error signal* as $\delta_i(k)$ for neuron i in layer k.

It has a role similar to the error at the outputs $t_i - y_i(\vec{x})$.

Output layer:

For $i=1 \ldots N(L_H+1)$:

$$\delta_i(L_H+1) = \sigma'(y_i) \cdot (t_i - y_i(\vec{x})).$$

Hidden layers:

For $k=L_H \ldots 1$:

For $i=1 \ldots N(k)$: $\delta_i(k) = \sigma'(o_i(k)) \cdot \sum_{j=1 \ldots N(k+1)} w_{ji}(k+1, k)\, \delta_j(k+1)$.

Adaptation: $\Delta w_{ji}(k+1,k) = \varepsilon\, \delta_j(k+1)\, o_i(k)$.

Backpropagation algorithm

Notation in [M]:

- No explicit numbering of layers.
- Uses Fermi function $\sigma(y) = 1 / (1+\exp(-y))$

  with $d\sigma/dy = \sigma(y) \cdot (1 - \sigma(y))$.

  Therefore, $o \cdot (1-o)$ appears in place of $\sigma'(o)$.
- In the adaption rule x_{ji}, denoting the output of neuron i fed to neuron j, is used in place of $o_i(k)$.

Backpropagation algorithm

- A weight of neuron *j* is adapted in proportion to
  - the activation of the neuron in the previous layer to which it is connected,
  - the weighted errors it causes at the outputs.
- The complex scheme is necessary since target values are available only for the outputs, not the neurons in the hidden layers.
- The backpropagation algorithm performs a stochastic approximation to gradient descent for the error function *E* (based on one sample at a time).
- Minimization
  - is computationally expensive (many steps, many units),
  - suffers from numerous local minima,
  - is difficult to terminate: Achieve good minimization without overfitting.

Backpropagation: Local minima

Two (of many) ways to avoid local minima:

- Repeat training with different random initial weights in the hope minimization is caught in different basins of attraction.
- Annealing: Add noise, e.g., every n learning steps:

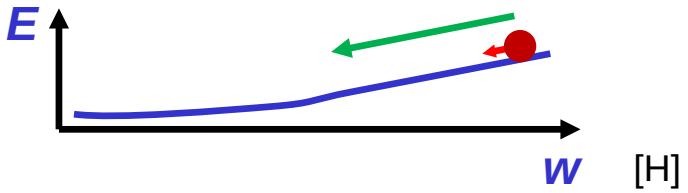
$$w_{ji}(k+1,k) \leftarrow w_{ji}(k+1,k) + T \cdot \zeta_{ji}(k+1,k),$$

where ζ is a Gaussian random variable with $<\zeta> = 0$ and $\zeta_{ji}(k+1,k)$ are pairwise uncorrelated. T is the "temperature", denoting the amount of noise added. T is gradually decreased during training.

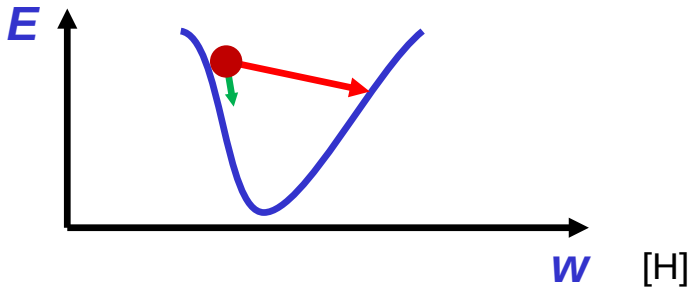
Annealing improves minimization but requires more learning steps.

Backpropagation: Step size adaptation

Increase ε in flat regions:



Decrease ε in steep terrain:



Step size adaptation:

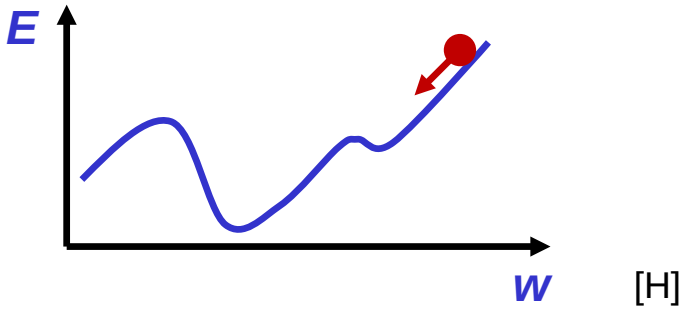
$$\varepsilon(t) = \begin{cases} \varepsilon^+ \cdot \varepsilon(t-1), & \varepsilon^+ > 1 \quad \text{if } \Delta E < 0 \\ \varepsilon^- \cdot \varepsilon(t-1), & \varepsilon^- < 1 \quad \text{if } \Delta E > 0 \end{cases}$$

with, e.g., $\varepsilon^+ = 1{,}1$ and $\varepsilon^- = 0{,}5$.

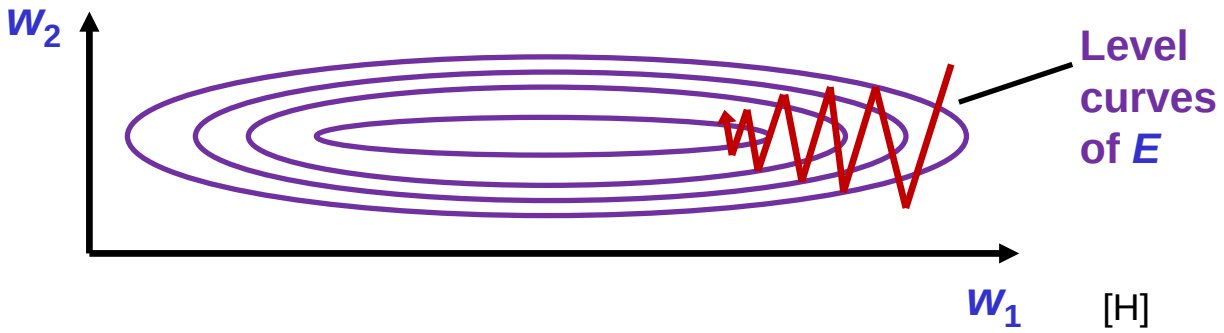
Backpropagation: Momentum

Problems:

- Minimization stops at minor minima because of too small ε:



- In valleys of E oscillation may occur due to too large ε:



Both problems are solved by adding "**momentum**"

$$\Delta w_{ji}(k+1,k)(t) \quad = \quad \varepsilon\, \delta_j(k+1)\, o_i(k) \;+\; \alpha\, \Delta w_{ji}(k+1,k)(t-1),$$

where t is a step counter. So direction of step $t-1$ is kept to some degree (controlled by $\alpha>0$). Momentum avoids abrupt changes of direction and increases effective step size in flat regions.

Backpropagation: Weight decay

Overly large weights are problematic since they make neurons too sensitive the input (leads to “binary” activations).

Weight decay: An additional quadratic regularization term in the error function avoids too large weights:

$$E[\{w\}] \quad = \quad \tfrac{1}{2} \sum_{i=1\ldots|D|} \left(t^i - y(x^i)\right)^2 \; + \; \beta I 2 \sum_{w \in \{w\}} w^2 \; .$$

It leads to linear weight decay in the learning rule:

$$\Delta w_{ji}(k+1,k) \quad = \quad \varepsilon \, \delta_j(k+1) \, o_i(k) \; - \; \beta \, w_{ji}(k+1,k).$$

MLP architecture

Two issues:

- How many layers?
- How many nodes in each layer?

Funahashi, 1989:

An arbitrary bounded continuous mapping $\vec{x} \rightarrow \vec{t}$, $\vec{x} \in \mathbb{R}^{d_{in}}$, $\vec{t} \in \mathbb{R}^{d_{out}}$ can be approximated with arbitrary precision with one hidden layer with sigmoid activation functions and a layer with linear output functions. The latter is necessary to map the range of σ to arbitrary values.

Cybenko 1988:

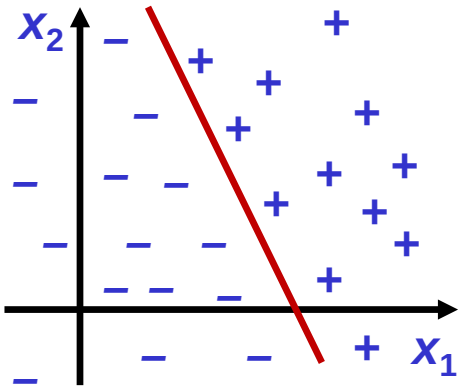
Add another sigmoid hidden layer, so *any* function can be approximated with arbitrary precision.

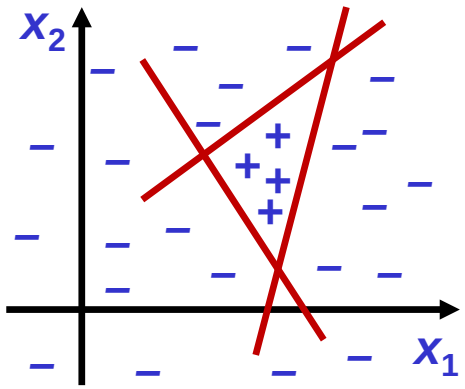
But: The hidden layer may be very large and is possibly not the most efficient representation of the mapping.

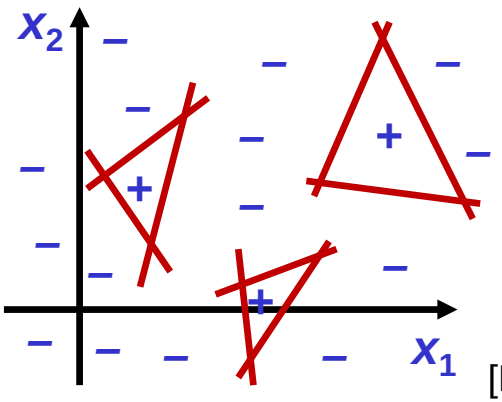
MLP architecture

Understand why an MLP can approximate arbitrary functions using two hidden layers with sigmoid activation functions and one output layer with sigmoid activation functions:

1. layer defines hyperplanes, dividing input space into half-planes.
2. layer defines ANDs over hyperplanes and thus convex areas.
3. layer defines ORs over convex areas.







[H]

MLP architecture

Two issues:

- How many layers?
- How many nodes in each layer?

→ No restrictions on numbers of nodes in hidden layers. But:

- The intrinsic dimensionality of the input will be reduced to the minimum of the numbers of nodes in the hidden layers.
- More nodes in a hidden layer than in the input layer may facilitate better representation but can not “invent” additional information.
- Hidden layers may serve to discover features in the input data, e.g., find that there are more input dimensions than necessary.

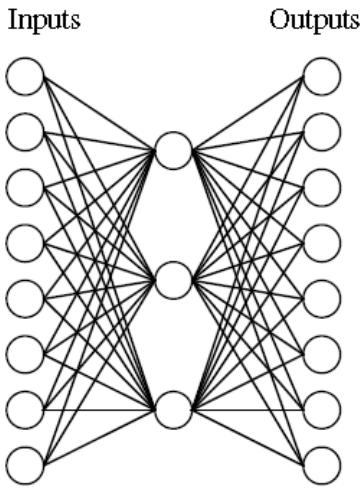
MLP: Encoder

Example: Encoder

Input: Binary strings of length 2^n with a single 1.

Output = Input.

One hidden layer with n units.



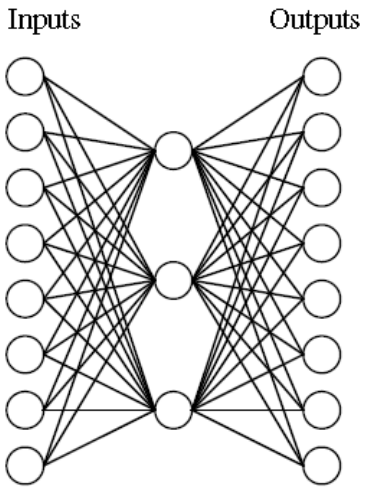
[M]

Obviously, the input *can* be coded with n units (binary numbers) !

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

[M]

MLP: Encoder



[M]

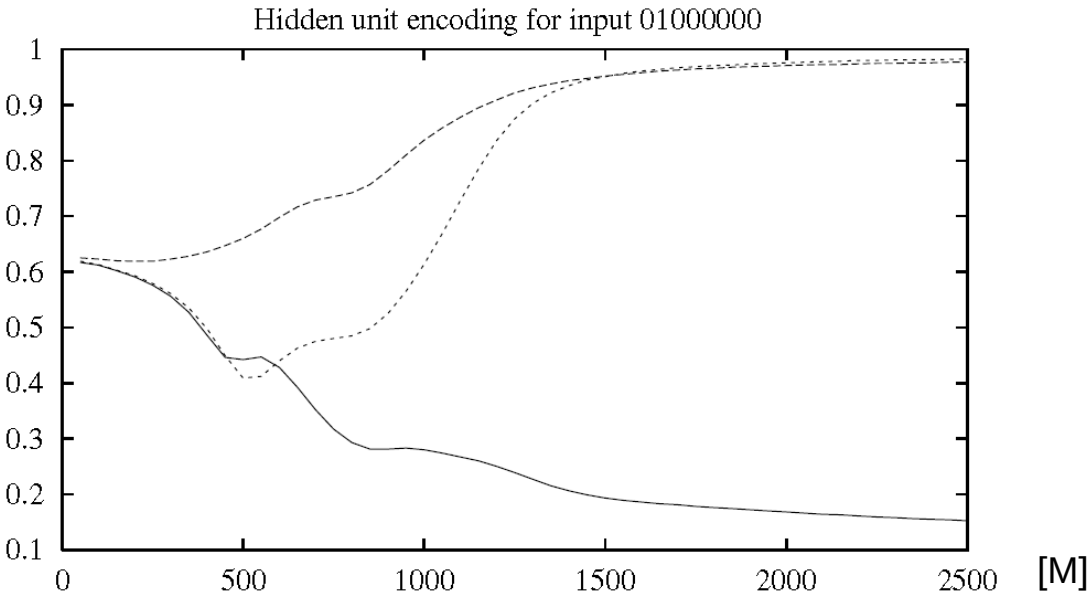
The encoder architecture implements a bottle neck, enforcing a more efficient representation of the input:

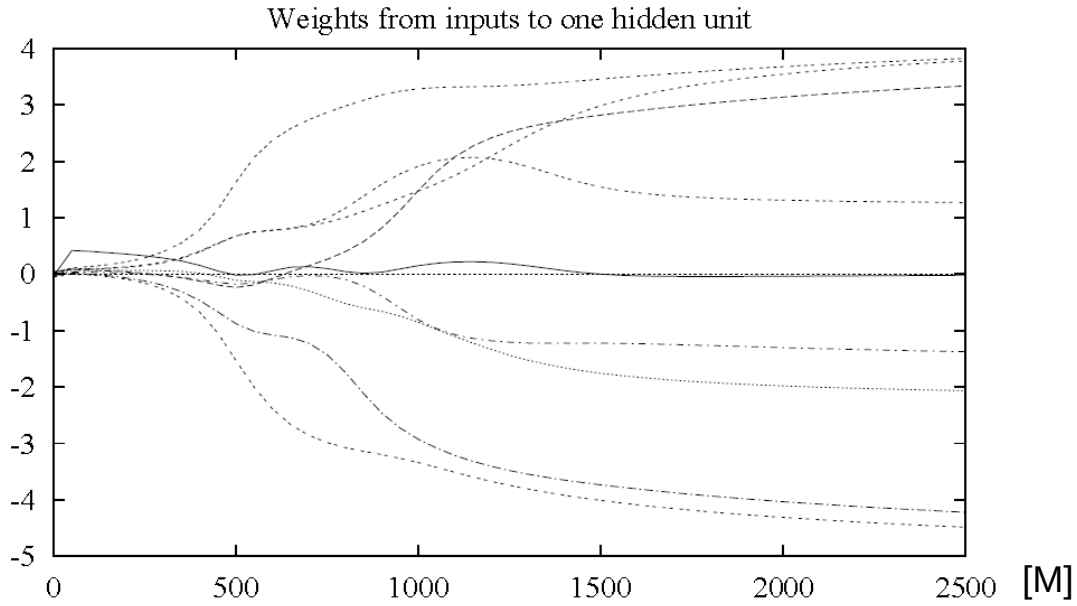
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

[M]

MLP: Encoder

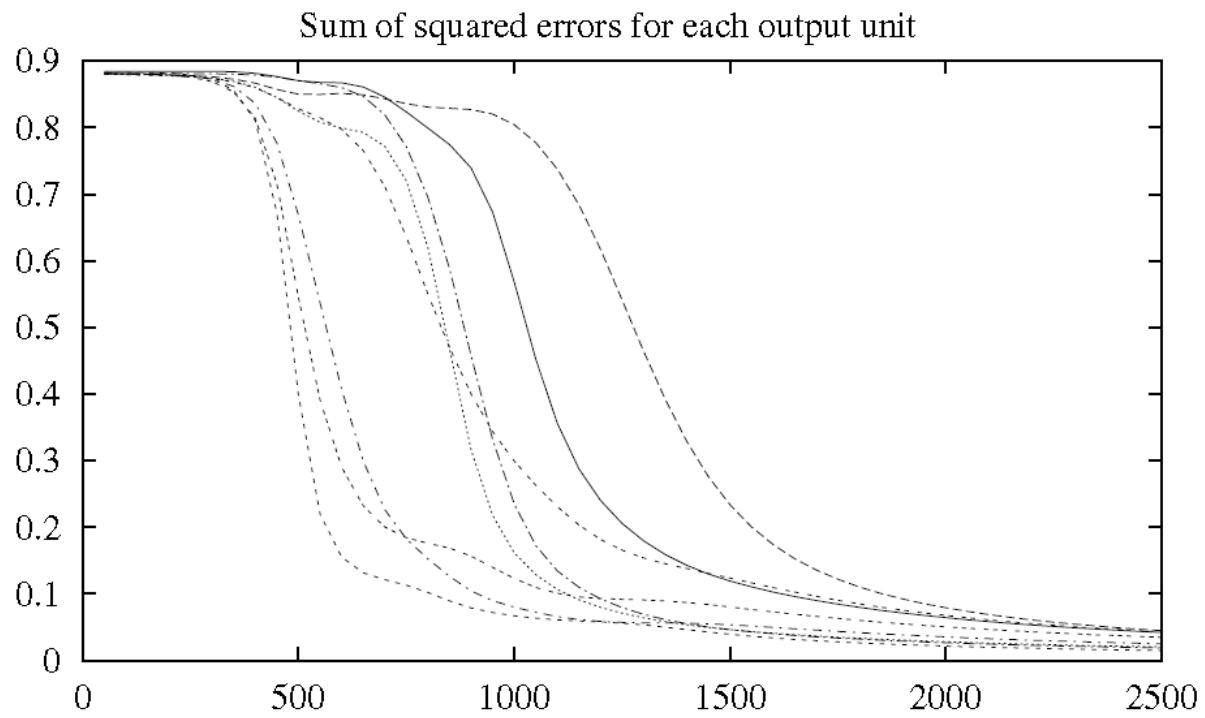
Training of the 8-3-8 network:





MLP: Encoder

Training of the 8-3-8 network:

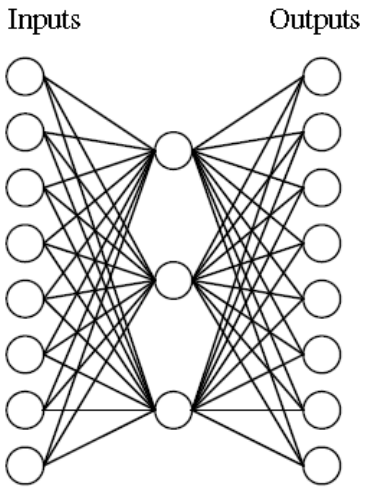


MLP: Encoder

General result by Baldi & Hornik, 1989 (qualitatively):

For an arbitrary data distribution of real valued vectors, the *n* neurons of the hidden layer (the bottle neck) converges to span the subspace of the input space which is spanned by the *n* principal components with largest eigenvalues.

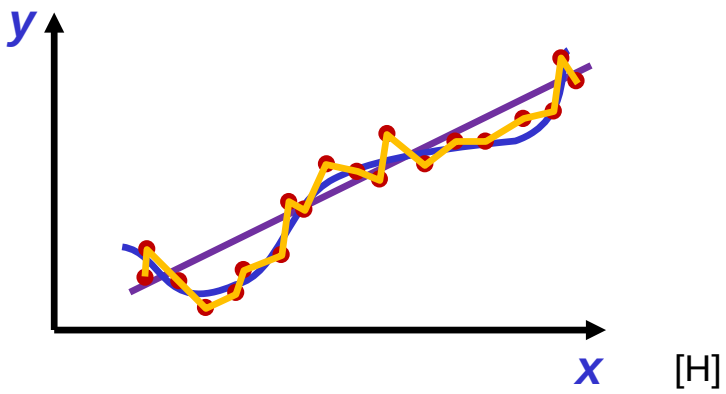
Note the weight vectors of the hidden neurons are not necessarily identical to the PCs, but span the same subspace.



[M]

Generalization and overfitting

- Hypothesis space: *n*-dimensional space for *n* weights.
- Hypothesis space is "well organized" insofar as *E* is a differentiable function over the hypotheses.
- Inductive bias: Given by the network architecture and the gradient descent search. Basically aims at smooth interpolation.
- Depending on the network size, overfitting is possible.



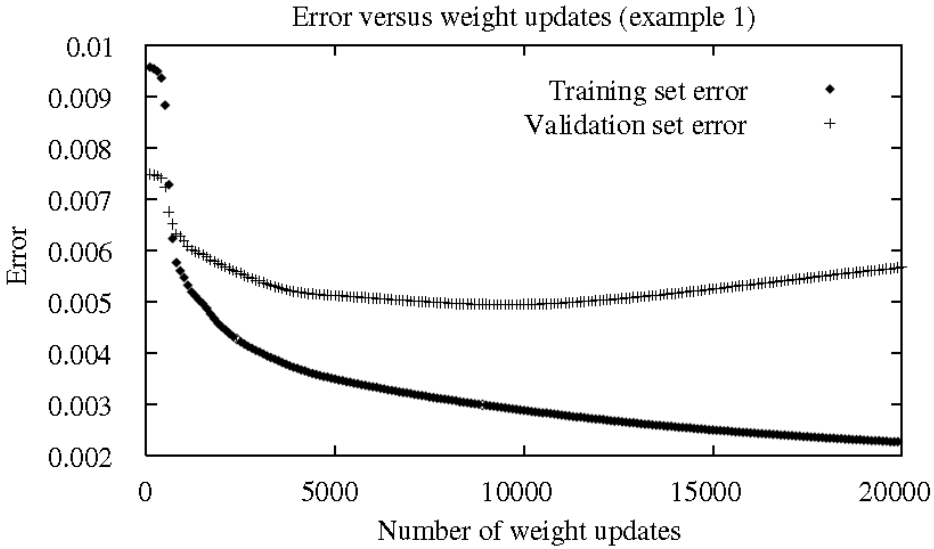
Linear fit

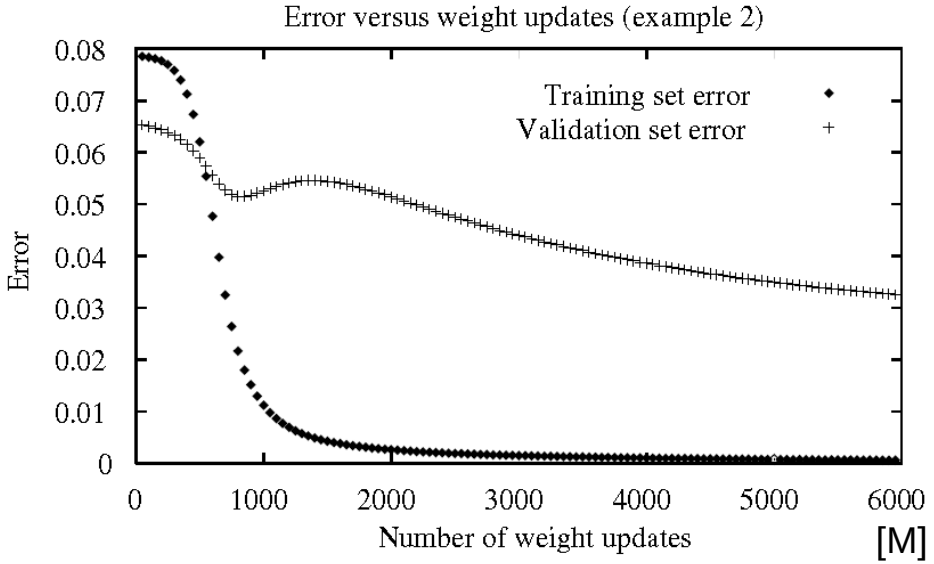
More complex fit

Overfit

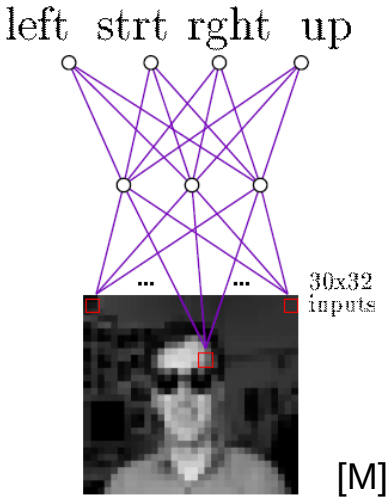
Generalization and overfitting

- Error on the training data set decreases continuously.
- The error on a validation data set usually increases after a certain number of training steps.
- Minimization must be repeated for different architectures and to obtain different local minima for a particular architecture.
- The best architecture / the best local minimum has the smallest error on the validation data set.
- Weight decay helps to avoid overfitting.





Historical Example: Face pose recognition



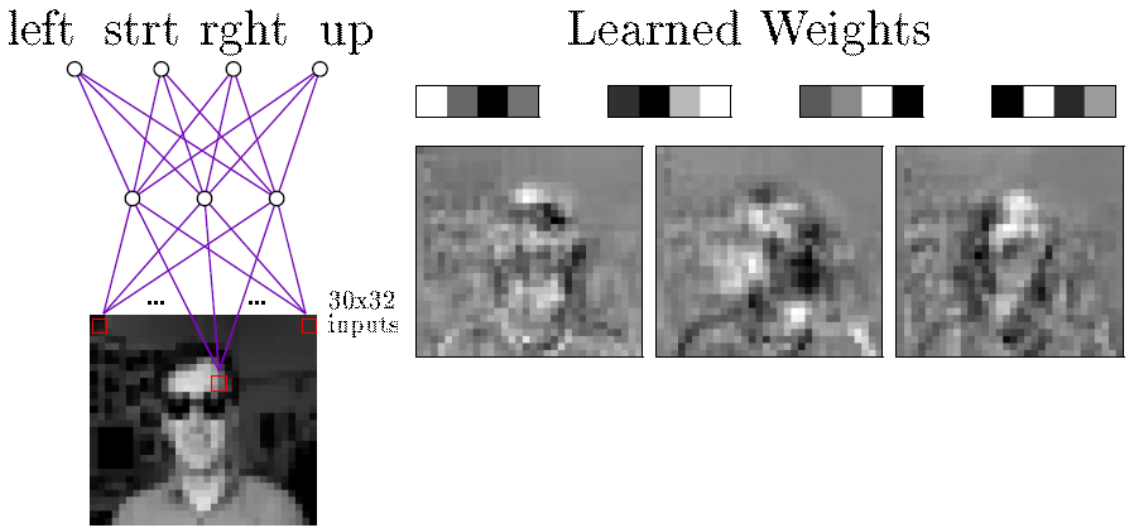
[M]

- Task: Recognize face pose.
- Input: 30x32 gray value image (passport-like setting), gray values scaled to 0–1.
- Output: *Left – straight – right – up*.
- Architecture: 960 x 3 x 4.
  - One input for each pixel.
  - 3 hidden units proved to be sufficient. 30 hidden units yield only slightly increased performance but require much more training.
  - One output for each class.
- Training data: 260 images of 20 people.
- Batch training (without stochastic approximation).
- Parameters: $\varepsilon = 0{,}3$ momentum $\alpha = 0{,}3$.
- Accuracy on test set > 90%.

Historical Example: Face pose recognition

Weights of the hidden units visualized as gray value images.

Bars above: Weights of the output layer.



Typical input images

[M]

http://www.cs.cmu.edu/~tom/faces.html

Summary MLP

- MLP is “the” ANN most well known and has most widespread use in industry.
- Very broad range of applications for function approximation, in particular, classification.
- Hidden layer representations may lead to new insight into the data.
- Training is difficult:
  - Suitable architecture must be found.
  - Suitable parameters must be found.
  - Local optima must be avoided.

Neural architectures

Neural architectures

Neural networks can be wired in highly complex ways.

ANN usually implement only special cases.

Given N neurons, let w_{ik} be the weight from neuron k to neuron i (i.e., i receives its input from k), $i,k \in [1,N]$.

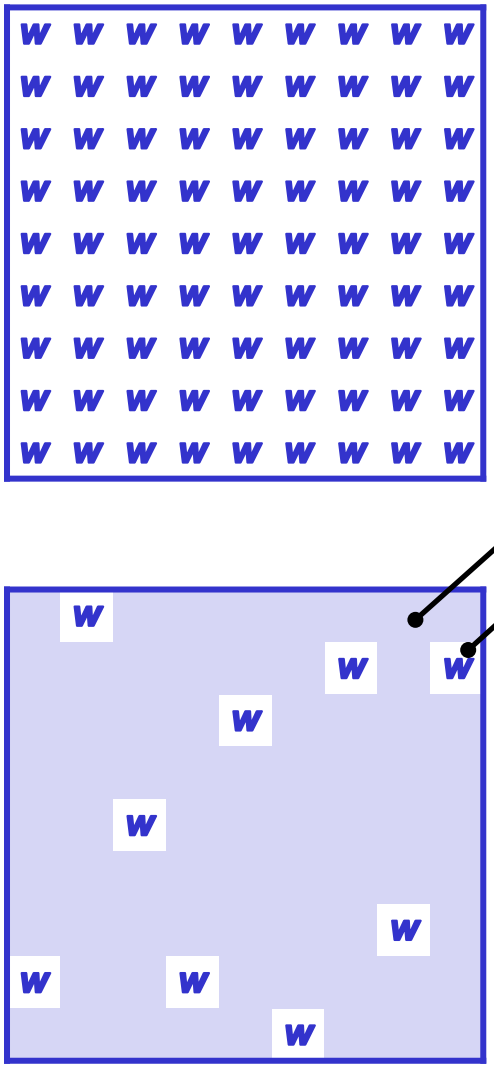
Note i and k are now numbers of neurons, not numbers of inputs on a particular neuron.

The NxN matrix w is called **connectivity matrix**.

It comprises the complete wiring information of the ANN, defining a directed graph.

There are different special cases of connectivity matrices.

Neural architectures



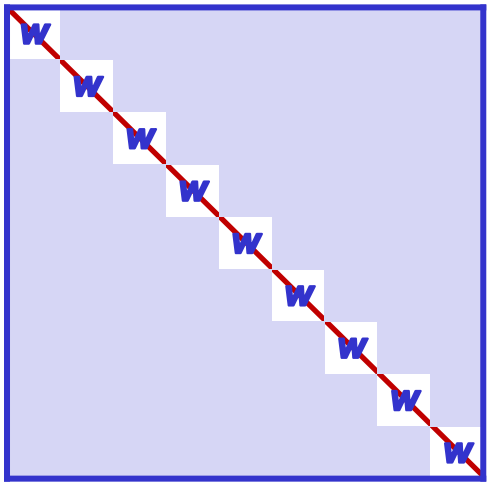
Fully connected net:

- Each neuron has connections to all other neurons.
- Highly complex dynamics.

Sparsely connected net:

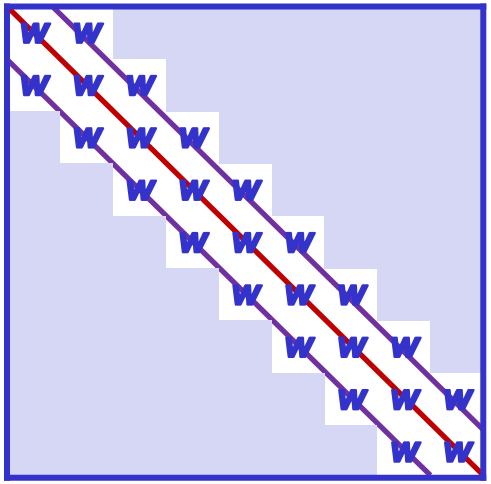
- Only few non-zero elements.
- Both forward and backward coupling.
- Some statements about dynamics are possible.

Neural architectures



Diagonal matrix:

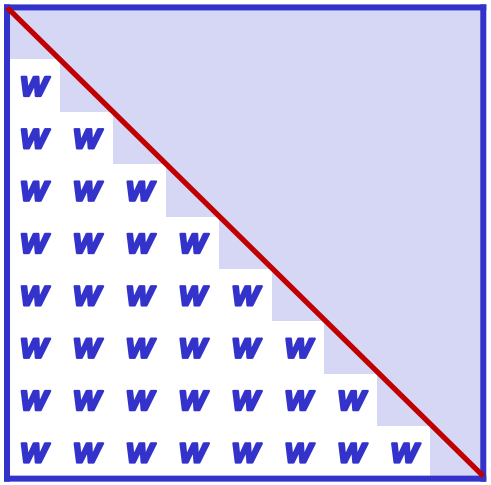
- $w_{ii} \neq 0$, $w_{ik} = 0$ for $i \neq k$.
- N isolated neurons without connections.
- Each neuron has only a connection to itself.

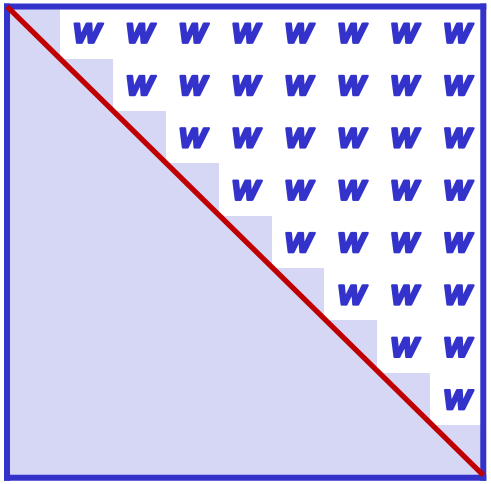


Tridiagonal matrix:

- Chain of N neurons.
- Each neuron has both forward and backward coupling to its neighbors.
- Only local couplings.

Neural architectures





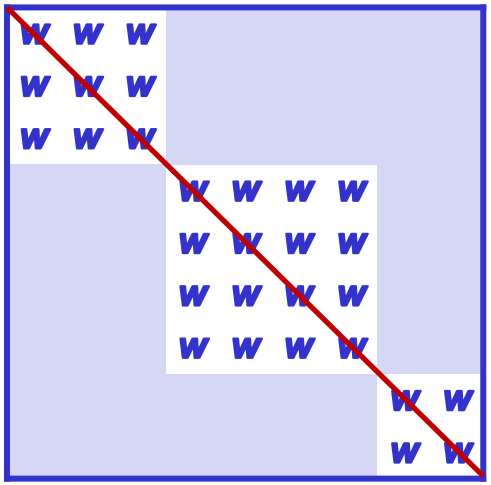
Lower triangular matrix:

- $w_{ik} = 0$ for $i \leq k$.
- Feedforward network (provided neuron index grows in “forward” direction, e.g., from input to output).
- Architecture type of the MLP.
- No recurrency.
- Diagonal is 0 → no self-coupling of neurons.

Upper triangular matrix (theoretical):

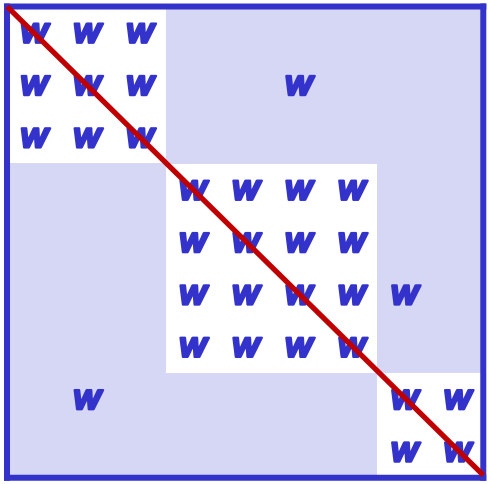
- Only backward coupling.
- Re-numbering → feedforward network

Neural architectures



Block diagonal matrix:

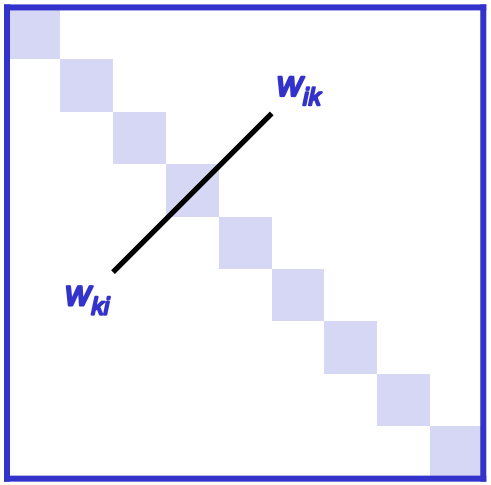
- Three separate networks.
- Each network is fully connected.



Block diagonal matrix, sparse rest of matrix:

- Three networks sparsely connected to each other.
- Forward connection from first to third net (in the sense of neuron numbers).
- Backward connections 3→2 and 2→1.

Neural architectures



Symmetric matrix:

- $W_{ik} = W_{ki}$.
- **Hopfield** network.
- Dynamics converges to a point attractor.

Recurrent networks

Feedforward network:

- Connectivity matrix has only zero weights in the upper triangle.
- Network is an acyclic directed graph.
- Example: MLP.
- Output is computed by updating (= computing the output) the neurons from the input side to the output side.

Recurrent network:

- Connectivity matrix has non-zero weights in lower **and** upper triangle.

→ There are cycles.

- It is impossible to find an order of neurons for updating as in a feedforward network (to compute the output of one neuron you need the output of another and vice versa).

Recurrent networks

- While feedforward networks are “timeless”, recurrent networks can be used only by defining dynamics.
- Recurrent networks are a dynamical system.
- The *state* of the system is the vector comprising all neuron activations. It varies over time.
- Two ways to define the dynamics:
  - Continuous time → description as a system of differential equations.
  - Discrete time steps:
    - Synchronous updating: All neurons within a single time step (provided the previous state).
    - Asynchronous updating: Select one neuron randomly, update.
- Note the above refers to abstract neurons as introduced earlier, not spiking neuron models.

Recurrent networks

- The phase space is the space in which all possible states of the network are represented (all possible activation vectors).
- The evolution of the system is the trajectory of the activation vector in phase space over time.
- There are different basic scenarios for the trajectory:
  - Converges to a point attractor.
  - Converges to a limit cycle.
  - Converges to a strange attractor (may be chaotic).
  - Chaos.

Remarks on neural representations

- The weights of a formal neuron can be thought of as its *receptive field*.
- This raises the general question how information is coded by receptive fields.
- Two alternatives:
  1. Specific neurons:
     - Small, localized receptive fields
     - Good separation of stimuli but bad generalization
     - “Grandmother cells”
  2. General neurons:
     - Large, diffuse receptive fields
     - Little specifity
     - Bad separation but good generalization
     - “Holistic code”, “distributed code”

Summary neural networks

- Application domains for ANN:
  - Input from real world:
    - High dimension,
    - noisy,
    - often real valued, in particular, raw sensor data.
  - Output:
    - May be vector,
    - form of target function is not known explicitly.
- ANN are well suited when
  - explicit modeling in infeasible,
  - resulting system needs not to be understood by humans.
- Architecture and training of neural networks requires expertise.

Image sources

[M] Online material available at www.cs.cmu.edu/~tom/mlbook.html for the textbook: Tom M. Mitchell: *Machine Learning*, McGraw-Hill

[H] Gunther Heidemann, 2012.