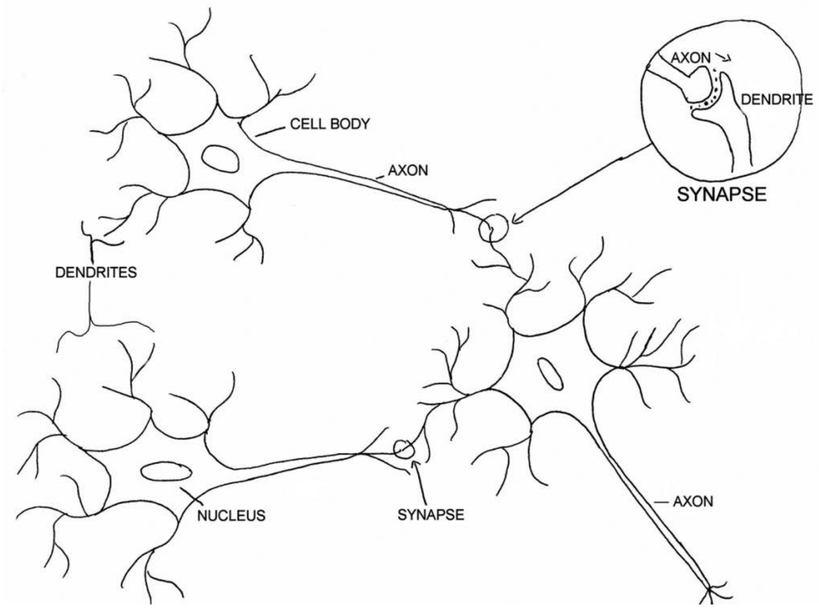
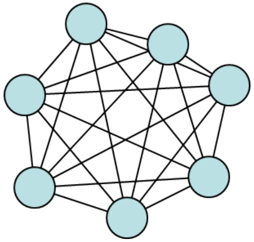
Biological Neural Networks



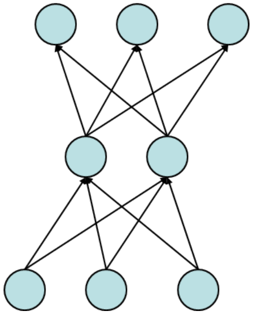
Topologies

Common Topologies of Artificial Neural Networks:



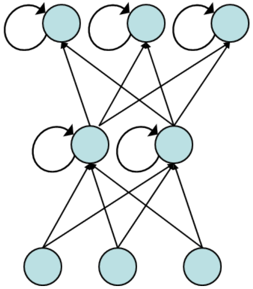
Fully Connected

Example:
Associative NN



Feed Forward

Example:
Autoassociative NN
Multilayer Perceptron



Recurrent

Example:
Recurrent NN

A Neuron

McCulloch-Pitts model of a neuron (1943)

Aim of the McCulloch-Pitts model: neurobiological modelling and simulation to understand very elementary functions of neurons and the brain.

A Neuron

McCulloch-Pitts model of a neuron (1943)

Aim of the McCulloch-Pitts model: neurobiological modelling and simulation to understand very elementary functions of neurons and the brain.

- A neuron is a binary switch, being either active or inactive.

A Neuron

McCulloch-Pitts model of a neuron (1943)

Aim of the McCulloch-Pitts model: neurobiological modelling and simulation to understand very elementary functions of neurons and the brain.

- A neuron is a binary switch, being either active or inactive.
- Each neuron has a fixed threshold value.

A Neuron

McCulloch-Pitts model of a neuron (1943)

Aim of the McCulloch-Pitts model: neurobiological modelling and simulation to understand very elementary functions of neurons and the brain.

- A neuron is a binary switch, being either active or inactive.
- Each neuron has a fixed threshold value.
- A neuron receives input signals from excitatory (positive) synapses (connections to other neuron).

A Neuron

McCulloch-Pitts model of a neuron (1943)

Aim of the McCulloch-Pitts model: neurobiological modelling and simulation to understand very elementary functions of neurons and the brain.

- A neuron is a binary switch, being either active or inactive.
- Each neuron has a fixed threshold value.
- A neuron receives input signals from excitatory (positive) synapses (connections to other neuron).
- A neuron receives input signals from inhibitory (negative) synapses (connections to other neuron).

A Neuron

McCulloch-Pitts model of a neuron (1943)

Aim of the McCulloch-Pitts model: neurobiological modelling and simulation to understand very elementary functions of neurons and the brain.

- A neuron is a binary switch, being either active or inactive.
- Each neuron has a fixed threshold value.
- A neuron receives input signals from excitatory (positive) synapses (connections to other neuron).
- A neuron receives input signals from inhibitory (negative) synapses (connections to other neuron).
- The inputs of a neuron are accumulated (integrated) for a certain time. When the threshold value of the neuron is exceeded, the neuron becomes active and sends signals to its neighbouring neurons via its synapses.

The simple perceptron

The Perceptron (Rosenblatt 1958)

The perceptron was introduced by Frank Rosenblatt for modelling pattern recognition abilities in 1958.
Aim: Automatic learning of the weights and the threshold to classify objects shown on the retina correctly.

The simple perceptron

The Perceptron (Rosenblatt 1958)

The perceptron was introduced by Frank Rosenblatt for modelling pattern recognition abilities in 1958.
Aim: Automatic learning of the weights and the threshold to classify objects shown on the retina correctly.

- A simplified retina is equipped with receptors (input neurons) that are activated by an optical stimulus.

The simple perceptron

The Perceptron (Rosenblatt 1958)

The perceptron was introduced by Frank Rosenblatt for modelling pattern recognition abilities in 1958.
Aim: Automatic learning of the weights and the threshold to classify objects shown on the retina correctly.

- A simplified retina is equipped with receptors (input neurons) that are activated by an optical stimulus.
- The stimulus is passed on to an output neuron via a weighted connection (synapse).

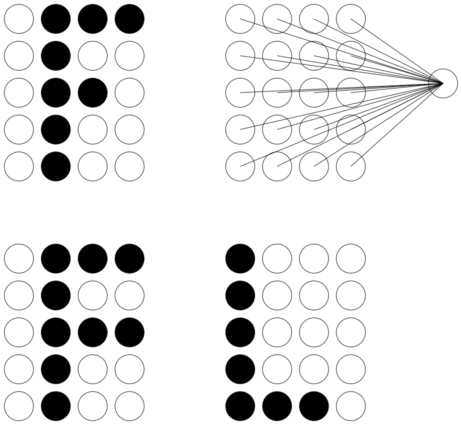
The simple perceptron

The Perceptron (Rosenblatt 1958)

The perceptron was introduced by Frank Rosenblatt for modelling pattern recognition abilities in 1958.
Aim: Automatic learning of the weights and the threshold to classify objects shown on the retina correctly.

- A simplified retina is equipped with receptors (input neurons) that are activated by an optical stimulus.
- The stimulus is passed on to an output neuron via a weighted connection (synapse).
- When the threshold θ of the output neuron is exceeded, the output is 1, otherwise 0.

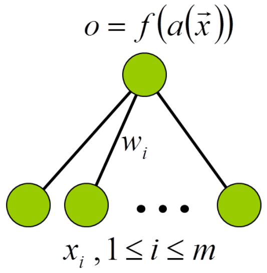
The simple perceptron



A perceptron for identifying the letter F.
Two positive and one negative example.

The simple perceptron

Schematic model of a perceptron:



$$a(\vec{x}) = \sum_{i=1}^{m} w_i \cdot x_i \qquad o = f(a) = \begin{cases} 1 & \text{if } a > \theta \\ 0 & \text{otherwise} \end{cases}$$

- (Numerical) input attributes A_i, output class (0 or 1).
- Classifier for two-class problem.
- For multiclass problems use one perceptron per class.

Perceptron learning algorithm

- Initialise the weights and the threshold value randomly.
- For each data object in the training data set, check whether the perceptron predicts the correct class.
- If the perceptron predicts the wrong class, adjust the weights and threshold value.
- Repeat this until no changes occur.

The delta rule

Whenever the perceptrons makes a wrong classification:
change weights and threshold in "'correct direction".

- If the desired output is 1 and the perceptron's output is 0, the threshold is not exceeded, although it should be. Therefore, lower the threshold and adjust the weights depending on the sign and magnitude of the inputs.
- If the desired output is 0 and the perceptron's output is 0, the threshold is exceeded, although it should not be. Therefore, increase the threshold and adjust the weights depending on the sign and magnitude of the inputs.

The delta rule

The **delta rule** recommends to adjust the weights and the threshold value according to:

$$\begin{aligned} w_i^{\text{new}} &= w_i^{\text{old}} + \Delta w_i \\ \theta^{\text{new}} &= \theta^{\text{old}} + \Delta \theta \end{aligned}$$

- w_i: A weight of the perceptron
- θ: The threshold value of the output neuron

The delta rule

The **delta rule** recommends to adjust the weights and the threshold value according to:

$$\begin{aligned} w_i^{\text{new}} &= w_i^{\text{old}} + \Delta w_i \\ \theta^{\text{new}} &= \theta^{\text{old}} + \Delta\theta \end{aligned}$$

- w_i: A weight of the perceptron
- θ: The threshold value of the output neuron
- $(a_1, \ldots, a_n)$: An input
- t: desired output for input $(a_1, \ldots, a_n)$
- t_p: The perceptron's output for input $(a_1, \ldots, a_n)$
- $\sigma > 0$: Learning rate

The delta rule

$$\Delta w_i = \begin{cases} 0 & \text{if } t_p = t \\ +\sigma a_i & \text{if } t_p = 0 \text{ and } t = 1 \\ -\sigma a_i & \text{if } t_p = 1 \text{ and } t = 0 \end{cases}$$

$$\Delta \theta = \begin{cases} 0 & \text{if } t_p = t \\ -\sigma & \text{if } t_p = 0 \text{ and } t = 1 \\ +\sigma & \text{if } t_p = 1 \text{ and } t = 0 \end{cases}$$

- w_i: A weight of the perceptron
- θ: The threshold value of the output neuron
- $(a_1, \dots, a_n)$: An input
- t: desired output for input $(a_1, \dots, a_n)$
- t_p: The perceptron's output for input $(a_1, \dots, a_n)$
- $\sigma > 0$: Learning rate

Example

Learning of the logical operator AND.
Training data:

$$\{((0,0),0),\ ((0,1),0),\ ((1,0),0),\ ((1,1),1)\}$$

Learning rate: $\sigma = 1$
Initialisation: $w_1 = w_2 = \theta = 0$

	a	t	t_p	Δw_1	Δw_2	$\Delta\theta$	w_1^{new}	w_2^{new}	θ
1. Epoch	0 0	0	0	0	0	0	0	0	0
	0 1	0	0	0	0	0	0	0	0
	1 0	0	0	0	0	0	0	0	0
	1 1	1	0	1	1	-1	1	1	-1

Example

	a	t	t_p	Δw_1	Δw_2	$\Delta\theta$	w_1^{new}	w_2^{new}	θ
2. Epoch	0 0	0	1	0	0	1	1	1	0
	0 1	0	1	0	-1	1	1	0	1
	1 0	0	0	0	0	0	1	0	1
	1 1	1	0	1	1	-1	2	1	0
3. Epoch	0 0	0	0	0	0	0	2	1	0
	0 1	0	1	0	-1	1	2	0	1
	1 0	0	1	-1	0	1	1	0	2
	1 1	1	0	1	1	-1	2	1	1

Example

	a	t	t_p	Δw_1	Δw_2	$\Delta\theta$	w_1^{new}	w_2^{new}	θ
4. Epoch	0 0	0	0	0	0	0	2	1	1
	0 1	0	0	0	0	0	2	1	1
	1 0	0	1	-1	0	1	1	1	2
	1 1	1	0	1	1	-1	2	2	1
5. Epoch	0 0	0	0	0	0	0	2	2	1
	0 1	0	1	0	-1	1	2	1	2
	1 0	0	0	0	0	0	2	1	2
	1 1	1	1	0	0	0	2	1	2

Example

	a	t	t_p	Δw_1	Δw_2	$\Delta\theta$	w_1^{new}	w_2^{new}	θ
6. Epoch	0 0	0	0	0	0	0	2	1	2
	0 1	0	0	0	0	0	2	1	2
	1 0	0	0	0	0	0	2	1	2
	1 1	1	1	0	0	0	2	1	2

Perceptron convergence theorem

Perceptron Convergence

If, for a given data set with two classes, there exists a perceptron that can classify all patterns correctly, then the delta rule will adjust the weights and the threshold after a finite number of steps in such way that all patterns are classified correctly.

Which kind of classification problems can be solved by a perceptron?

Linear separability

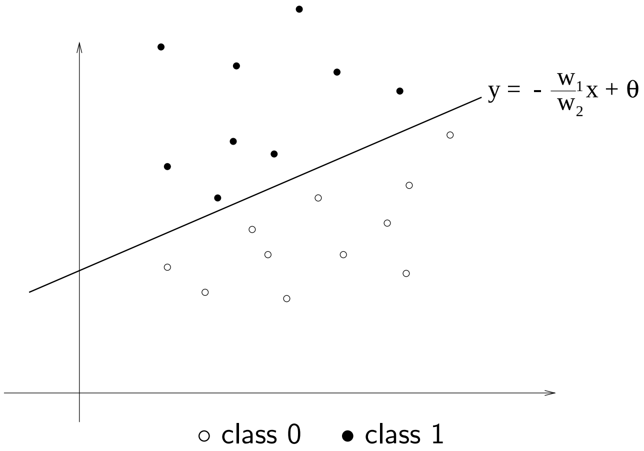
Consider a perceptron with two input neurons. Let t_p be the output of the perceptron for input (a_1, a_2). Then

$$\begin{aligned} t_p = 1 \quad &\iff \quad w_1 \cdot a_1 + w_2 \cdot a_2 > \theta \\ &\iff \quad a_2 > -\frac{w_1}{w_2} a_1 + \frac{\theta}{w_2}. \end{aligned}$$

The perceptron's output is 1 if and only if the input pattern (a_1, a_2) is above the line

$$y \;=\; -\frac{w_1}{w_2} x + \frac{\theta}{w_2}.$$

Linear separability



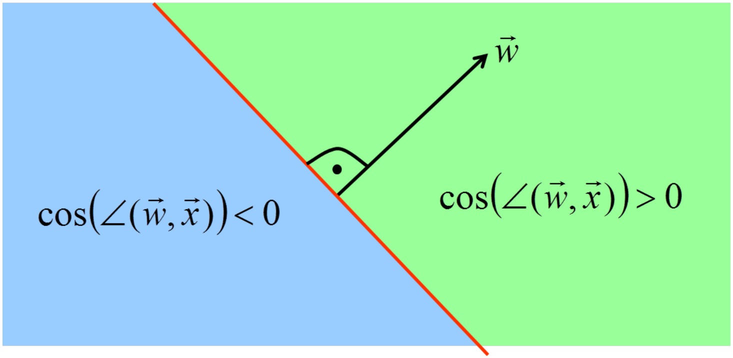
The parameters w_1, w_2, θ determine the line. All input patterns above this line are assigned to class 1, the input patterns below the line to class 0.

Linear separability in hyper spaces

$$a\left(\vec{x}\right) = \theta + \sum_{i=1}^{m} w_i \cdot x_i = \theta + \vec{w}^T \cdot \vec{x} = \theta + |\vec{x}||\vec{w}| \cos\left(\angle(\vec{w}, \vec{x})\right)$$

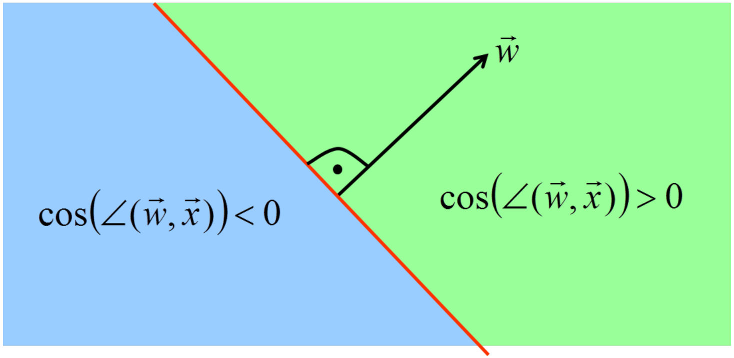
Linear separability in hyper spaces

$$a(\vec{x}) = \theta + \sum_{i=1}^{m} w_i \cdot x_i = \theta + \vec{w}^T \cdot \vec{x} = \theta + |\vec{x}||\vec{w}| \cos(\angle(\vec{w}, \vec{x}))$$


$\vec{w}$
$\cos(\angle(\vec{w}, \vec{x})) < 0$
$\cos(\angle(\vec{w}, \vec{x})) > 0$

Linear separability in hyper spaces

$$a(\vec{x}) = \theta + \sum_{i=1}^{m} w_i \cdot x_i = \theta + \vec{w}^T \cdot \vec{x} = \theta + |\vec{x}||\vec{w}| \cos(\angle(\vec{w}, \vec{x}))$$


$\vec{w}$
$\cos(\angle(\vec{w},\vec{x})) < 0$
$\cos(\angle(\vec{w},\vec{x})) > 0$

$\Rightarrow$ Perceptrons represent hyperplanes in feature space.

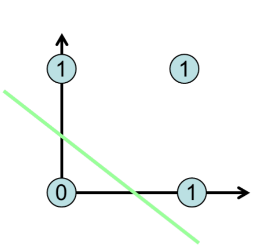
Linear separability

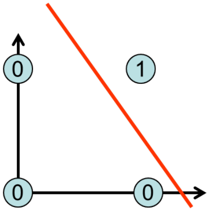
Linear Separability

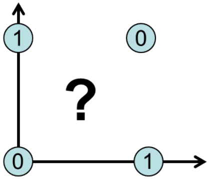
A perceptron with n input neurons can classify all examples from a data set with n input variables and two classes correctly, if there exists a hyperplane separating the two classes.
Such classification problems are called **linearly separable**.

Linear separability

Example: The exclusive OR (XOR) defines a classification task which is not linearly separable.

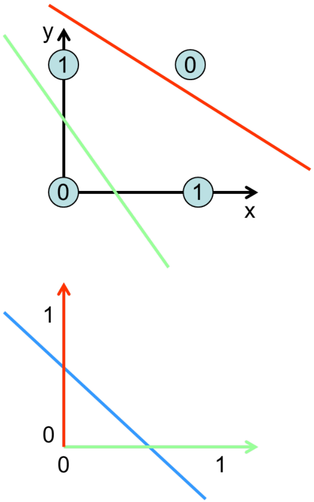


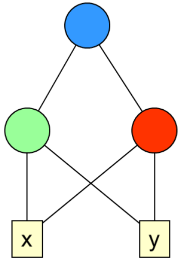




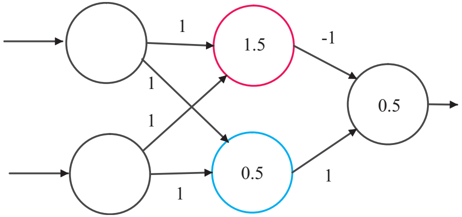
Linear separability

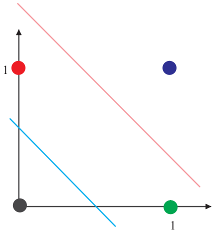
The exclusive OR (XOR) can be modeled using a second layer.



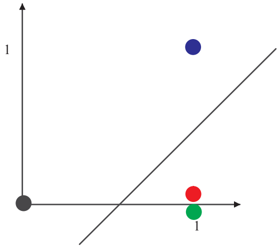


XOR with hidden layer





A perceptron with a hidden layer of neurons: The hidden layer carries out a transformation. The output neuron can solve the linearly separable problem in the transformed space.



Learning algorithm?

Problem: How to adjust the weights (and thresholds) for the neurons the hidden layer?

Learning algorithm?

Problem: How to adjust the weights (and thresholds) for the neurons the hidden layer?

Solution: Multilayer perceptrons with gradient descent

Learning algorithm?

Problem: How to adjust the weights (and thresholds) for the neurons the hidden layer?

Solution: Multilayer perceptrons with gradient descent

Does not work with binary (non-differentiable) threshold function as activation function for the neurons.

Learning algorithm?

Problem: How to adjust the weights (and thresholds) for the neurons the hidden layer?

Solution: Multilayer perceptrons with gradient descent

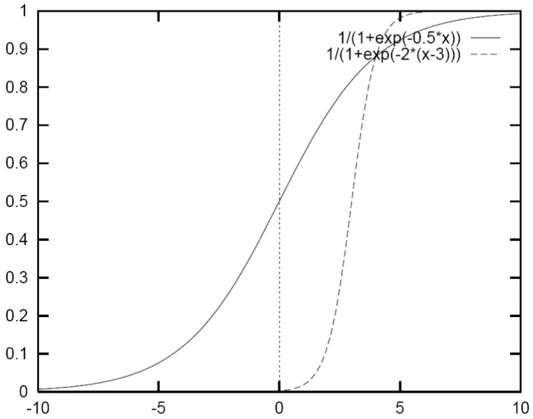
Does not work with binary (non-differentiable) threshold function as activation function for the neurons.

Must be replaced by a differentiable function.

Sigmoidal activation function

sigmoidal functions with bias θ and steepness $\alpha > 0$:

$$o_u(\mathsf{a}_u) = \frac{1}{1 + \exp(-\alpha(\mathsf{a}_u - \theta))}$$



Multilayer perceptrons

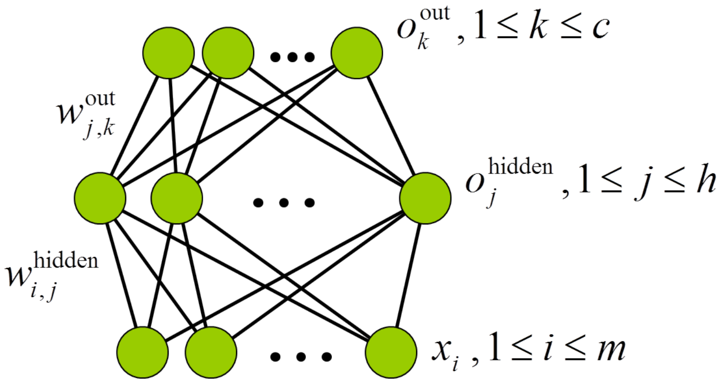
A multilayer perceptron is a neural network with

- an input layer,
- one or more hidden layers, and
- an output layer.

- Connections exist only between neurons from one layer to the next layer.
- Activation functions for neurons are usually sigmoidal functions.

Multilayer perceptrons

A two-layer Multilayer Perceptron:


$o_k^{\text{out}}, 1 \le k \le c$
$w_{j,k}^{\text{out}}$
$o_j^{\text{hidden}}, 1 \le j \le h$
$w_{i,j}^{\text{hidden}}$
$x_i, 1 \le i \le m$

$$o_k^{\text{out}} = f\left(\sum_{j=1}^{h} w_{j,k}^{\text{out}} \cdot f\left(\sum_{i=1}^{m} w_{i,j}^{\text{hidden}} \cdot x_i\right)\right)$$

Error function

For the gradient descent we need an error function.
Here: mean squared error (MSE).

$$E(\vec{w}) = \frac{1}{2} \sum_{\vec{x} \in T} \sum_{k=1}^{c} \left(f\left(o_k(\vec{x}) \right) - y_k(\vec{x}) \right)^2$$

- c: the number of output neurons
- $y_k(\vec{x})$: target output of output neuron k for input $\vec{x} \in T$
- $o_k(\vec{x})$: output (activation) of output neuron k for input $\vec{x}$

Learning rule

Adjust the weights based on a gradient descent technique, i.e. proportional to the gradient of the gradient of the error function.

$$\vec{w}(t+1) = \vec{w}(t) + \Delta\vec{w}(t)$$

with

$$\Delta\vec{w}(t) = -\eta\nabla(E(\vec{w}(t)))$$

with $\eta > 0$ a non-zero learning rate and

$$\Delta\vec{w}(t) = -\eta\nabla(E(\vec{w}(t))) = -\eta\left(\frac{\partial E(\vec{w}(t))}{\partial w_1}, \cdots, \frac{\partial E(\vec{w}(t))}{\partial w_m}\right)$$

so we really only need to determine

$$\Delta w_{u,v} \ = \ -\eta\frac{\partial E}{\partial w_{u,v}}$$

for each weight.

Learning rule: Output Neurons

For each weight in the output layer (let's assume a perceptron here):

$$\Delta w_k^{\text{out}}(t) = -\eta \frac{\partial E(\vec{w}^{\text{out}}(t))}{\partial w_k^{\text{out}}}$$

and

$$\frac{\partial E(\vec{w}^{\text{out}}(t))}{\partial w_k^{\text{out}}} = \frac{\partial \frac{1}{2} \sum_{j=1}^{|T|} \left(f\left(a(\vec{x}_j)\right) - y_j\right)^2}{\partial w_i}$$

$$= \frac{1}{2} \sum_{j=1}^{|T|} \frac{\partial \left(f\left(a(\vec{x}_j)\right) - y_j\right)^2}{\partial w_k^{\text{out}}} = \sum_{j=1}^{|T|} \left(f\left(a(\vec{x}_j)\right) - y_j\right) \frac{\partial f\left(a(\vec{x}_j)\right)}{\partial w_k^{\text{out}}}$$

Learning rule: Output Neurons

cont.

$$\cdots = \sum_{j=1}^{|T|} \left(f\left(a(\vec{x}_j)\right) - y_j(\vec{x}_j)\right) \frac{\partial f\left(\sum_{k'=0}^{m} w_{k'}^{\text{out}} o^{\text{hidden}} k'(\vec{x}_j)\right)}{\partial w_k^{\text{out}}}$$

$$= \sum_{j=1}^{|T|} \left(f\left(a(\vec{x}_j)\right) - y_j(\vec{x}_j)\right) f'\left(a(\vec{x}_j)\right) o_k^{\text{hidden}}(\vec{x}_j)$$

$$= \sum_{j=1}^{|T|} \left(o_j(\vec{x}_j) - y_j(\vec{x}_j)\right) f'\left(a(\vec{x}_j)\right) o_k^{\text{hidden}}(\vec{x}_j)$$

Learning rule: Output Neurons

We can now define δ_i as:

$$\delta_j = (o_j - y_j) f'\left(a(\vec{x}_j)\right)$$

which results in the compact

Update Equation

$$\Delta w_{i,j} = -\eta \cdot \delta_j \cdot x_{j,i}$$

Note that if we do use the sigmoid function:

$$f'(a) = \left(\frac{1}{1+e^{-a}}\right)' = -\frac{-e^{-a}}{(1+e^{-a})^2} = \frac{1+e^{-a}}{(1+e^{-a})^2} - \frac{1}{(1+e^{-a})^2}$$

$$= \frac{1}{1+e^{-a}}\left(1 - \frac{1}{1+e^{-a}}\right) = f(a)\left(1 - f(a)\right)$$

Learning rule: Hidden Neurons

Now where do we get the target signal from for the hidden neurons?
Same approach: just apply gradient descent!

$$\Delta w_{i,j}^{\mathsf{hidden}} = -\eta \frac{\partial E(\vec{w})}{\partial w_{i,j}^{\mathsf{hidden}}} \quad , \quad 1 \leq i \leq m \wedge 1 \leq j \leq h$$

Learning rule: Hidden Neurons

$$\Delta w_{i,j}^{\text{hidden}} = -\eta \frac{\partial \frac{1}{2} \sum_{\vec{x}\in T} \sum_{k=1}^{c} \left(f\left(a_k^{\text{out}}(\vec{x})\right) - y_k(\vec{x})\right)^2}{\partial w_{i,j}^{\text{hidden}}}$$

$$= -\frac{\eta}{2} \sum_{\vec{x}\in T} \sum_{k=1}^{c} \frac{\partial \left(f\left(a_k^{\text{out}}(\vec{x})\right) - y_k(\vec{x})\right)^2}{\partial w_{i,j}^{\text{hidden}}}$$

$$= -\frac{\eta}{2} \sum_{\vec{x}\in T} \sum_{k=1}^{c} 2\left(f\left(a_k^{\text{out}}(\vec{x})\right) - y_k(\vec{x})\right)$$

$$\cdots \frac{\partial \left(f\left(\sum_{j'=1}^{h} w_{j',k}^{\text{out}} f\left(\sum_{i'=1}^{m} w_{i',j'}^{\text{hidden}} \cdot x_{i'}\right)\right) - y_k(\vec{x})\right)}{\partial w_{i,j}^{\text{hidden}}}$$

Learning rule: Hidden Neurons

cont.

$$= -\eta \sum_{\vec{x} \in T} \sum_{k=1}^{c} \underbrace{\left(f\left(a_k^{\mathsf{out}}(\vec{x})\right) - y_k(\vec{x}) \right) f'\left(\sum_{j'=1}^{h} w_{j',k}^{\mathsf{out}} f\left(\sum_{i'=1}^{m} w_{i',j'}^{\mathsf{hidden}} \cdot x_{i'} \right)\right)}_{=\delta_k^{\mathsf{out}}}$$

$$\dots \frac{\partial \sum_{j'=1}^{h} w_{j',k}^{\mathsf{out}} f\left(\sum_{i'=1}^{m} w_{i',j'}^{\mathsf{hidden}} \cdot x_{i'} \right)}{\partial w_{i,j}^{\mathsf{hidden}}}$$

$$= -\eta \sum_{\vec{x} \in T} \sum_{k=1}^{c} \delta_k^{\mathsf{out}} \frac{\partial \sum_{j'=1}^{h} w_{j',k}^{\mathsf{out}} f\left(\sum_{i'=1}^{m} w_{i',j'}^{\mathsf{hidden}} \cdot x_{i'} \right)}{\partial w_{i,j}^{\mathsf{hidden}}}$$

Learning rule: Hidden Neurons

cont.

$$= -\eta \sum_{\vec{x}\in T} \sum_{k=1}^{c} \delta_k^{\text{out}} w_{j,k}^{\text{out}} \frac{\partial f\left(\sum_{i'=1}^{m} w_{i',j}^{\text{hidden}} \cdot x_{i'}\right)}{\partial w_{i,j}^{\text{hidden}}}$$

$$= -\eta \sum_{\vec{x}\in T} \sum_{k=1}^{c} \delta_k^{\text{out}} w_{j,k}^{\text{out}} f'\left(\sum_{i'=1}^{m} w_{i',j}^{\text{hidden}} \cdot x_{i'}\right) \cdot x_i$$

$$= -\eta \sum_{\vec{x}\in T} \underbrace{\sum_{k=1}^{c} \delta_k^{\text{out}} w_{j,k}^{\text{out}} f'\left(a_j^{\text{hidden}}\right)}_{=:\delta_j^{\text{hidden}}} \cdot x_i$$

$$= \sum_{\vec{x}\in T} -\eta \cdot \delta_j^{\text{hidden}} \cdot x_i$$

Haven't we seen an equation of this form before?

Backpropagation

Error Backpropagation or generalised delta rule:

$$\Delta w_{i,j}^{\text{out}} = -\eta \cdot \delta_j^{\text{out}} \cdot o_i^{\text{hidden}}$$

with

$$\delta_j^{\text{out}} = (o_j - y_j) f'\left(a_j^{\text{out}})\right)$$

and

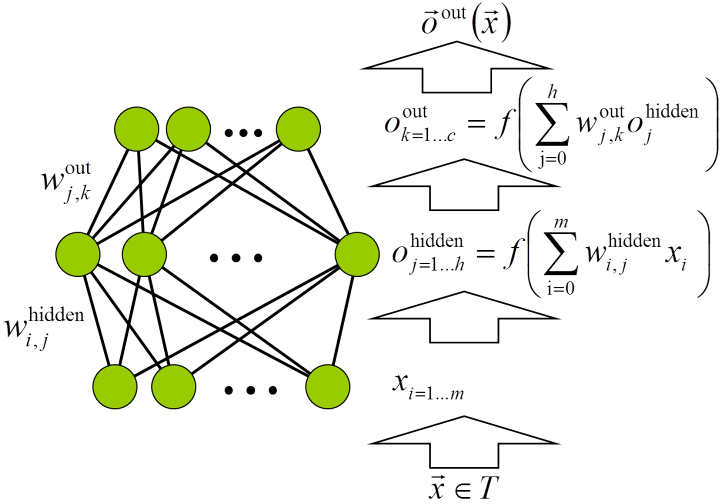
$$\Delta w_{i,j}^{\text{hidden}} = -\eta \cdot \delta_j^{\text{hidden}} \cdot x_i$$

with

$$\delta_j^{\text{hidden}} = \sum_{k=1}^{c} \delta_k^{\text{out}} w_{j,k}^{\text{out}} f'\left(a_j^{\text{hidden}}\right)$$

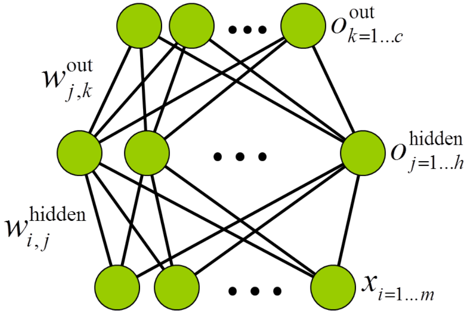
$\Rightarrow$ Recursive equation for updating the weights: Update the weights to the neuron in the output layer first and then go back layer by layer and update the corresponding weights.

Error Backpropagation - Forward Phase



$\vec{o}^{\,\text{out}}(\vec{x})$
$o^{\text{out}}_{k=1\ldots c} = f\left(\sum_{j=0}^{h} w^{\text{out}}_{j,k}\, o^{\text{hidden}}_{j}\right)$
$o^{\text{hidden}}_{j=1\ldots h} = f\left(\sum_{i=0}^{m} w^{\text{hidden}}_{i,j}\, x_i\right)$
$x_{i=1\ldots m}$
$\vec{x} \in T$
$w^{\text{out}}_{j,k}$
$w^{\text{hidden}}_{i,j}$

Error Backpropagation - Backward Phase



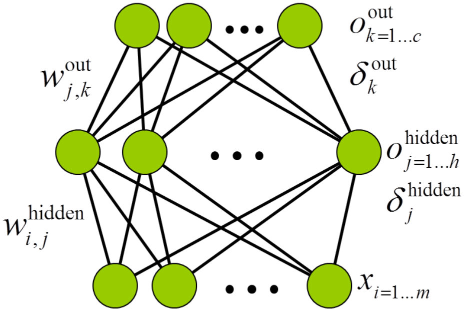
$o^{\text{out}}_{k=1\ldots c}$
$w^{\text{out}}_{j,k}$
$o^{\text{hidden}}_{j=1\ldots h}$
$w^{\text{hidden}}_{i,j}$
$x_{i=1\ldots m}$

$$\vec{y}(\vec{x})$$

$$\delta^{\text{out}}_{k=1\ldots c} = \left(o^{\text{out}}_k - y_k(\vec{x})\right) \cdot o'^{\text{out}}_k$$

$$\delta^{\text{hidden}}_{j=1\ldots h} = \sum_{k=1}^{c} \delta^{\text{out}}_k \cdot w^{\text{out}}_{j,k} \cdot o'^{\text{hidden}}_j$$

Error Backpropagation - Weight Update


$o^{\text{out}}_{k=1\ldots c}$
δ^{out}_k
$w^{\text{out}}_{j,k}$
$o^{\text{hidden}}_{j=1\ldots h}$
δ^{hidden}_j
$w^{\text{hidden}}_{i,j}$
$x_{i=1\ldots m}$

$$\Delta w^{\text{out}}_{j,k} += -\eta \cdot \delta^{\text{out}}_k \cdot o^{\text{hidden}}_j$$

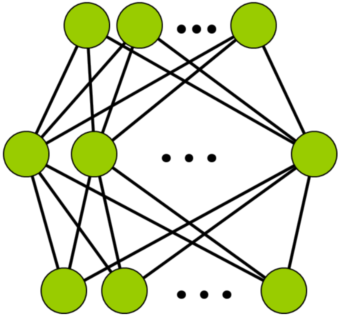
$$\Delta w^{\text{hidden}}_{i,j} += -\eta \cdot \delta^{\text{hidden}}_j \cdot x_i$$

Error Backpropagation - Batch Weight Update

$$w_{j,k}^{\text{out}}(t+1) = w_{j,k}^{\text{out}}(t) + \Delta w_{j,k}^{\text{out}}$$

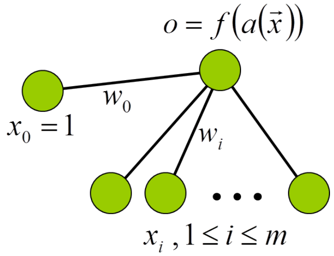
$$\Longrightarrow$$

$$w_{i,j}^{\text{hidden}}(t+1) = w_{i,j}^{\text{hidden}}(t) + \Delta w_{i,j}^{\text{hidden}}$$



Training the bias values

The bias values can be considered as special weights when an artificial input neuron with constant input 1 is introduced:



Backpropagation

- Backpropagation as a gradient descent technique can only find a local minimum. Training the networks with different random initialisations will usually lead to different results.
- The learning rate η defines the stepwidth of the gradient descent technique.
  - A very large η leads to skipping minima or oscillation.
  - A very small η leads to starving, i.e. slow convergence or even convergence before the (local) minimum is reached.

Backpropagation

- Introduce a momentum term: For the weight change, the previous weight change is taken into account:

$$\Delta_p W(u,v) \;=\; \eta \delta_v^{(p)} a_u^{(p)} + \beta \Delta_q W(u,v)$$

  $\Delta_p W(u,v)$ is the weight change in the previous step of the gradient descent algorithm.
  If weight is changed continuously in the same direction, the weight change increases, otherwise it decreases.
  Typical choices for η and β: $\eta = 0.2$, $\beta = 0.8$.

Backpropagation

- Usually, weights are not updated after a whole epoch, i.e. after all patterns have been presented once, but after the presentation of each input pattern.
- There is no general rule, how to choose the number of hidden layers and the size of the hidden layers. Small neural networks might not be flexible enough to fit the data. Large neural networks tend to overfitting.
- The steepness of the activation function is usually fixed and is not adjusted.
- The multilayer perceptron learns only in those regions where the activation function is not close to zero or one, otherwise the derivative is almost zero.

Other learning algorithms

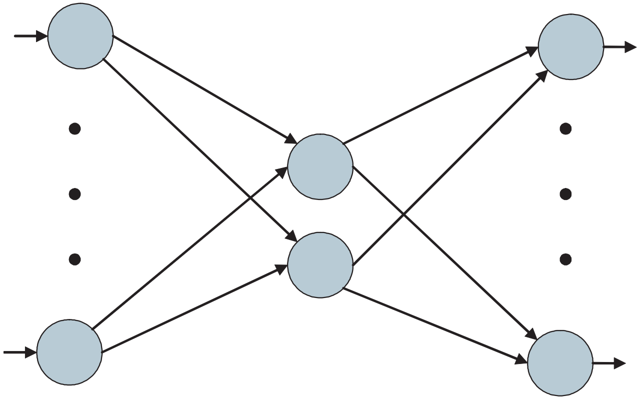
- Weight decay is sometimes included, pushing all weights in the direction of zero. Only those weights will “survive” that are really needed.
- Approximate the error function locally by a quadratic curve and find in each step the minimum of the quadratic curve.
- . . .

Nonlinear PCA

Dimension reduction with multilayer perceptrons:

- Input and output are identical, i.e. the neural network should learn the identity function. (Autoassociative network)
- Introduce a hidden layer with only two neurons (representing the two dimensions for the graphical representation of the dimension reduction), the bottleneck.
- Train the neural network with the data.
- After training, input the data into the network and use the outputs of the bottleneck neurons for the graphical representation.

Autoassociative bottleneck neural networks



Other Neural Network Topics

- (Hard/Soft) Competitive Learning
- Learning Vector Quantization
- Self Organizating Maps
- Radial (and other) Basis Function Networks

Many connections to Kernel Methods and Support Vector Machines...