Artificial Neural Networks

From Perceptron to Feed Forward Neural Networks

Matteo Matteucci

matteo.matteucci@polimi.it

Department of Electronics, Information, and Bioengineering

Politecnico di Milano

Why should we care about Neural Networks?

Everyday computer systems are good at:

- Doing precisely what the programmer programs they to do
- Doing arithmetic very fast

But we whould like them to:

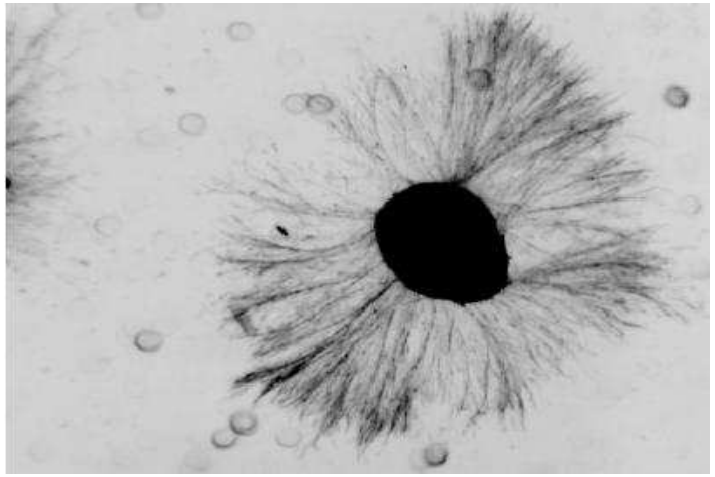
- Interact with noisy data or data from the environment
- Be massively parallel and fault tolerant
- Adapt to circumstances

We should look for a computational model other than
Von Neumann Machine!

The Biological Neuron

In the brain we have:

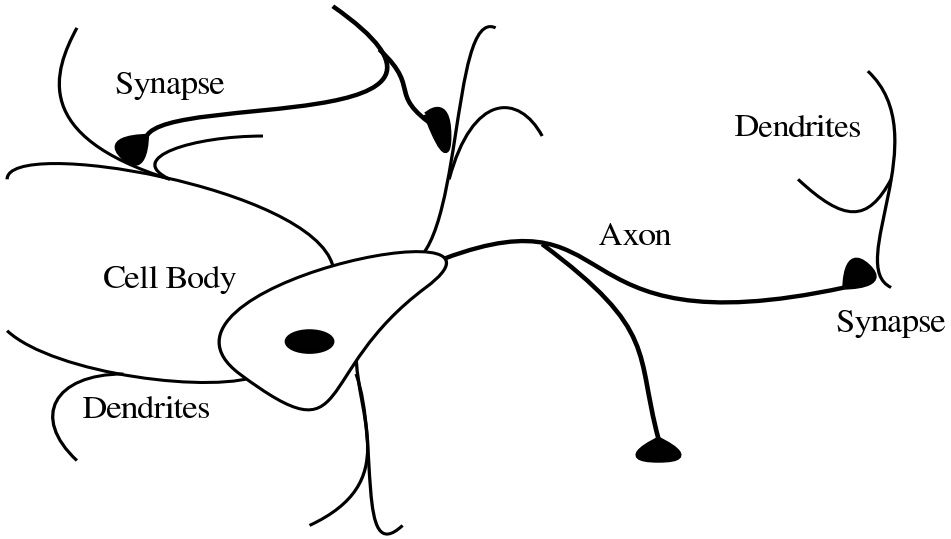
- 10^{11} neurons
- 10^{4} synapses per neurons



The computational model of the brain is:

- Distributed among simple units called neurons
- Intrinsically parallel
- Redundant and thus fault tolerant

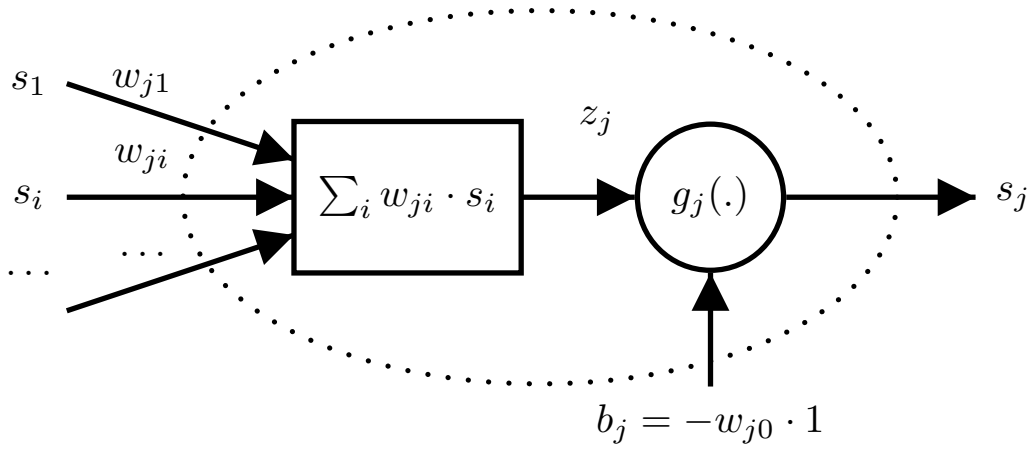
Computation in Biological Neurons



Information is transmitted through chemical mechanisms:

- Dendrites collect input charges from synapses
  - Inhibitory synapses with different weight
  - Excitatory synapses with different weight
- Axon transmit accumulated charges through synapses
  - Once the charge is above a threshold the neuron **fires**

The Artificial Neuron



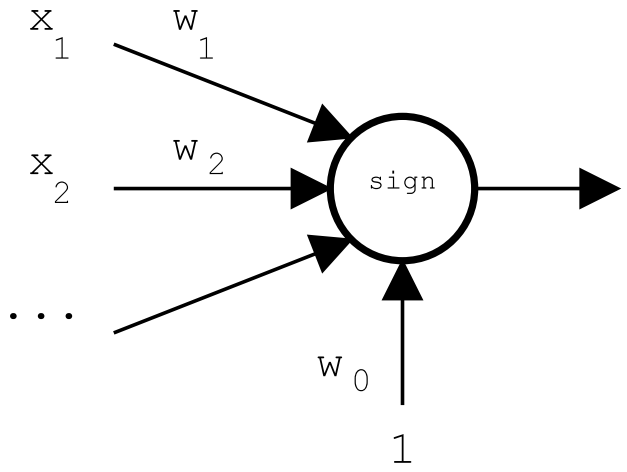
In this simple model of neuron j we can identify:

- The synaptic weights or simply weights w_{ji}
- The activation value $z_j = \sum_i w_{ji} s_i$
- The activation threshold or bias $b_j \doteq -w_{j0} \cdot 1$
- The activation function $g_j(.)$

The final output of neuron j is: $s_j = g_j(\sum_{i=1}^{I} w_{ji} s_i - b_j) = g_j(\sum_{i=0}^{I} w_{ji} s_i)$

The Perceptron

The first model of neuron proposed was the Perceptron:



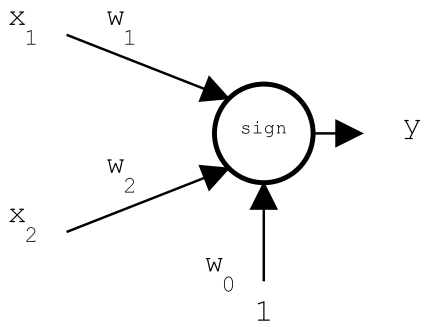
Sign function: $g(z_j) = \begin{cases} 1 & \text{if } z_j > 0 \\ 0 & \text{if } z_j = 0 \\ -1 & \text{if } z_j < 0 \end{cases}$

with $z_j = \sum_{i=0}^{I} w_i x_i$

What can I do with such a simple model?

The Perceptron as a Logic Operator

Perceptron as a logic AND



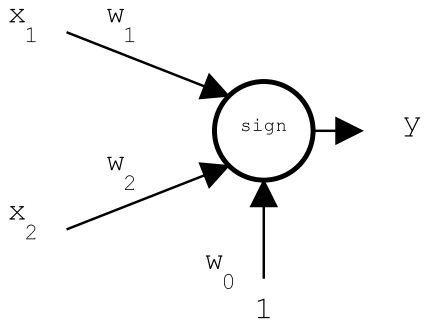
$w_1 = 3/2$

$w_2 = 1$

$w_0 = -2$

x_1	x_2	y
0	0	-1
0	1	-1
1	0	-1
1	1	1

Perceptron as a logic OR



$w_1 = 1$

$w_2 = 1$

$w_0 = -1/2$

x_1	x_2	y
0	0	-1
0	1	1
1	0	1
1	1	1

Hebb Learning Rule

Weights were learned using the Hebbian learning rule:

"The strength of a synapse increase according to the simultaneous activation of the relative input and the desired target"

(Hebb 1949)

Hebbian learning can be summarized by the following rule:

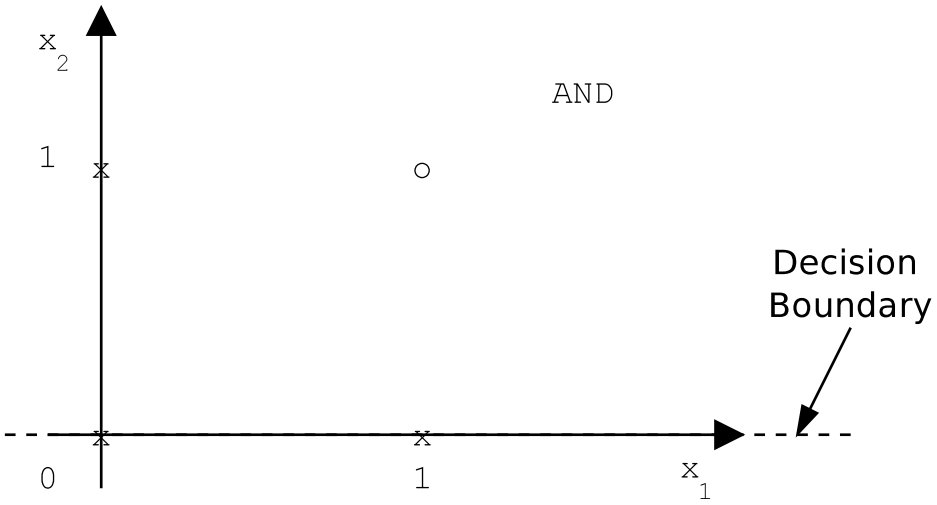
$$\begin{aligned} w_i^{k+1} &= w_i^k + \Delta w_i \\ \Delta w_i &= \eta \cdot t \cdot x_i \end{aligned}$$

Where we have

- η: learning rate
- x_i: the i^{th} perceptron input
- t: the desired output

Hebbian Learning of the AND Operator (0)

Suppose we start from a random initialization of $w_1 = 0, w_2 = 1$ and $w_0 = 0$ using a learning rate $\eta = 1/2$ we get:



Hebbian Learning of the AND Operator (...)

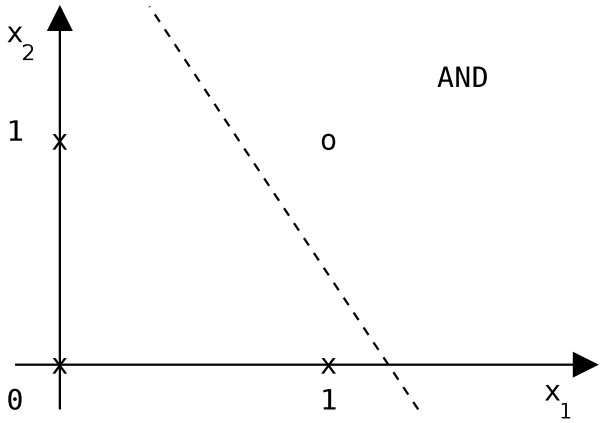
. . . let's skip some epochs ;) . . .

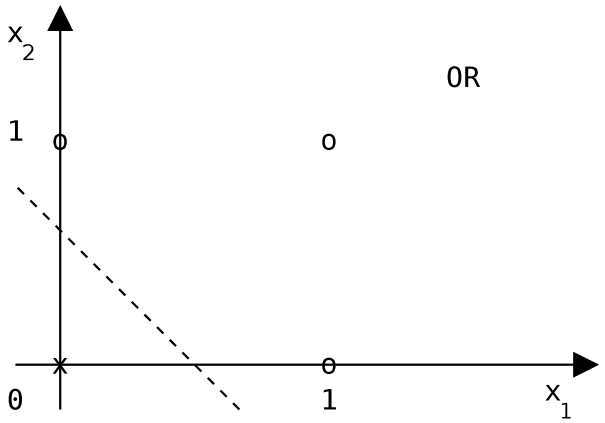
How does it work?

We can compute the decision boundary for the Perceptron:

$$\begin{aligned} 0 &= x_1 \cdot w_1 + x_2 \cdot w_2 + w_0 \\ x_2 \cdot w_2 &= -x_1 \cdot w_1 - w_0 \\ x_2 &= -\frac{w_1}{w_2} \cdot x_1 - \frac{w_0}{w_2} \end{aligned}$$

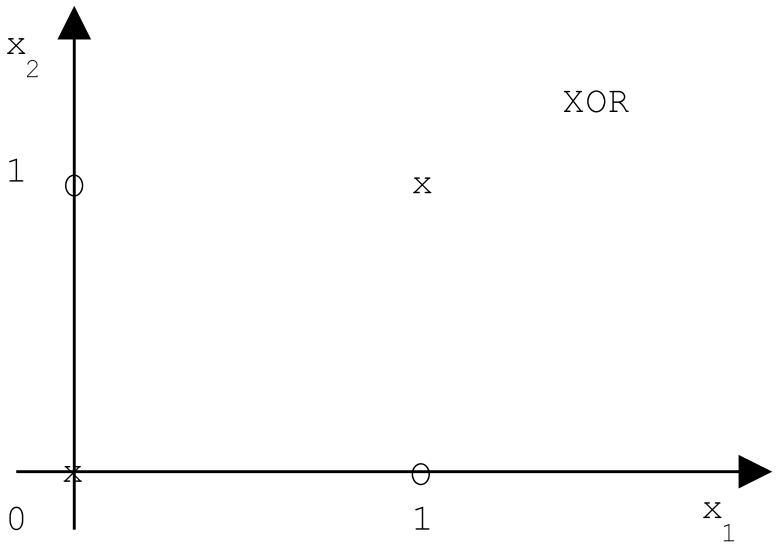
The decision boundary is a line:





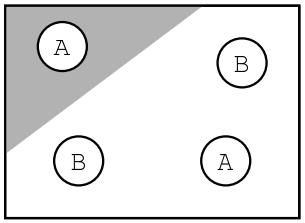
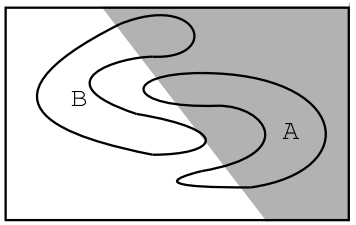
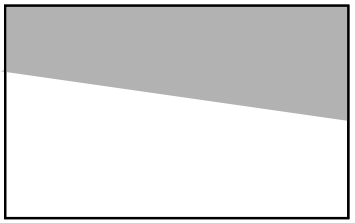
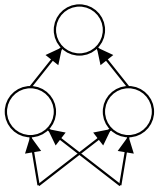
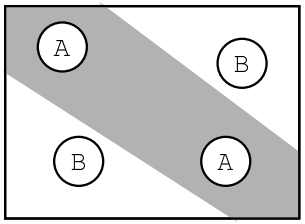
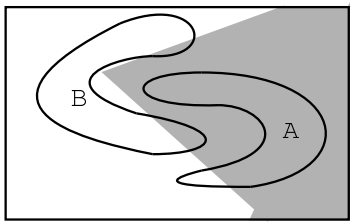
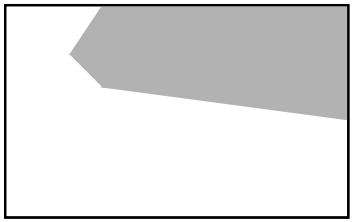
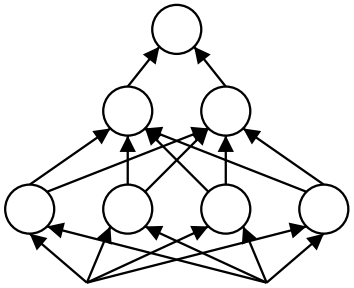
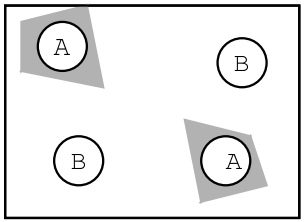
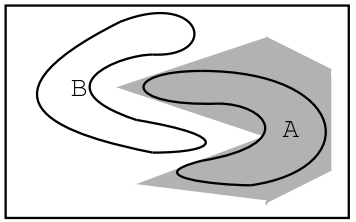
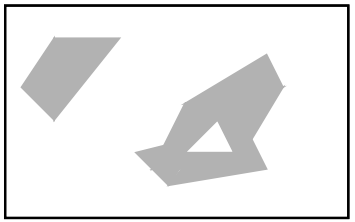
The XOR Problem

Great, but what if we have a non linearly separable problem?
(i.e., The XOR problem, Minsky '69)



Wait until the '80s and you'll see!

Topology and Complexity

Topology	Type of Decision Region	XOR Problem	Classes with Meshed Regions	Most General Region Shapes
	Half bounded by hyperplanes			
	Convex Open or Clesed Regions			
	Arbitrary Regions (Complexity limited by the number of nodes)			

Neural Networks
– Feedforward Topology –

Multi-Layer Perceptrons and Artificial Neural Networks

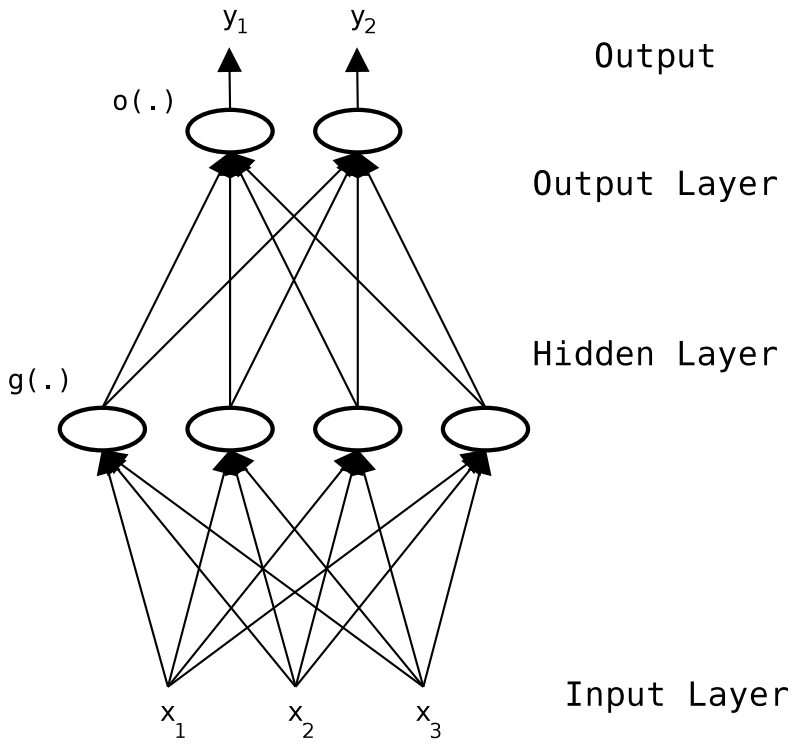
Artificial Neural Network: A set of neurons connected according to a topology

Layer: Neurons at the same distance from the input neurons

Input Layer: Layer of neurons that receives as input the data to process

Output Layer: Layer of neurons that gives the final result of the network

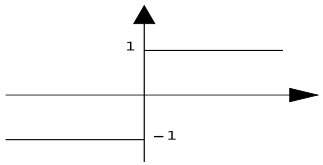
Hidden Layer: Layer of neurons that process data from other neurons to be processed from other neurons



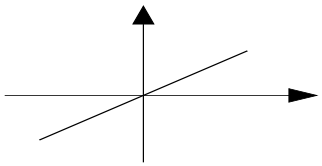
An artificial neural network is a **non-linear model** characterized by the number of neurons, their topology, activation functions, and the values of synaptic weights and biases.

Common Activation Functions

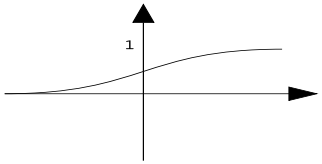
Sign function: $g(z_j) = \begin{cases} 1 & \text{if } z_j > 0 \\ 0 & \text{if } z_j = 0 \\ -1 & \text{if } z_j < 0 \end{cases}$



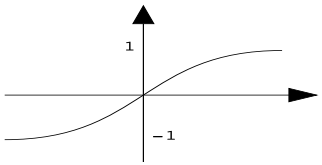
Linear function: $g(z_j) = z_j$



Sigmoid function: $g(z_j) = \dfrac{1}{1 + e^{-z_j}}$



Hyperbolic tangent: $g(z_j) = \dfrac{e^{z_j} - e^{-z_j}}{e^{z_j} + e^{-z_j}}$



Learning in Multi Layer Perceptrons: The Idea

Use Gradient Descend to iteratively minimize the network error (it turns out that this was rediscovered many times and named in a different ways):

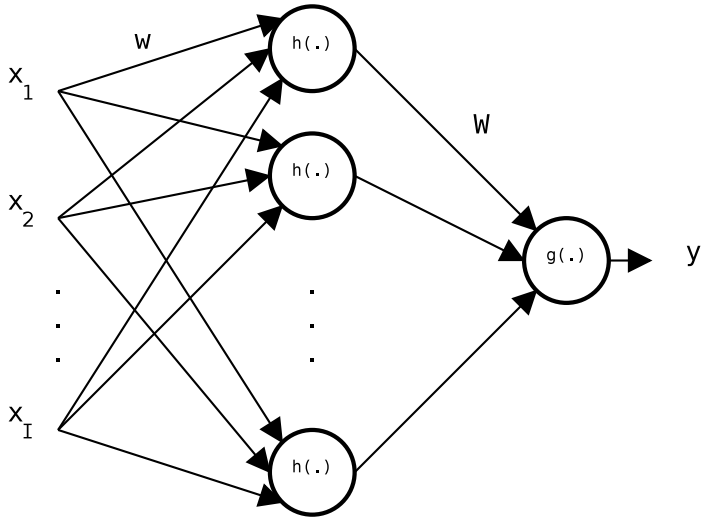
- Delta Rule
- Widrow Hoff Rule
- Adaline Rule
- Backpropagation

Learning can thus be summarized by the following rule:

$$\begin{aligned} \mathbf{w}^{k+1} &= \mathbf{w}^{k} + \Delta\mathbf{w} \\ \Delta\mathbf{w} &= -\eta \cdot \frac{\partial E}{\partial \mathbf{w}} \end{aligned}$$

- η: learning rate
- $\frac{\partial E}{\partial \mathbf{w}}$: gradient of the error function w.r.t. the weights

Learning in Multi Layer Perceptrons: An Example (I)



$$y = g(\sum_j^J W_j \cdot h(\sum_i^I w_{ji} \cdot x_i))$$

$$E = \sum_n^N (t_n - y_n)^2$$

Goal: approximate a target function t given a finite set of N observation

We define some useful variables:

- $a_j = \sum_i^I w_{ji} \cdot x_i$ (activation value)
- $b_j = h(a_j)$ (output of j^{th} hidden neuron)
- $A = \sum_j^J W_j b_j$

Learning in Multi Layer Perceptrons: An Example (II)

Given $E = \sum_n^N (t_n - y_n)^2 = \sum_n^N (t_n - g(A))^2$

$$\begin{aligned}
\frac{\partial E}{\partial W_j} &= \sum_n^N 2(t - g(A)) \cdot \frac{\partial}{\partial W_j}(t - g(A)) \\
&= \sum_n^N 2(t - g(A)) \cdot (-g'(A)) \cdot \frac{\partial}{\partial W_j} A \\
&= \sum_n^N 2(t - g(A)) \cdot (-g'(A)) \cdot b_j
\end{aligned}$$

We obtain the Backpropagation update rule for W_js

$$W_j^{k+1} = W_j^k + 2\eta \sum_n^N (t - g(A)) \cdot g'(A) \cdot b_j$$

Learning in Multi Layer Perceptrons: An Example (III)

$$\begin{aligned}
\frac{\partial E}{\partial w_{ji}} &= \sum_n^N 2(t - g(A)) \cdot (-g'(A)) \cdot \frac{\partial}{\partial w_{ji}} A \\
&= \sum_n^N 2(t - g(A)) \cdot (-g'(A)) \cdot W_j \cdot \frac{\partial}{\partial w_{ji}} b_j \\
&= \sum_n^N 2(t - g(A)) \cdot (-g'(A)) \cdot W_j \cdot h'(a_j) \frac{\partial}{\partial w_{ji}} a_j \\
&= \sum_n^N 2(t - g(A)) \cdot (-g'(A)) \cdot W_j \cdot h'(a_j) \cdot x_i
\end{aligned}$$

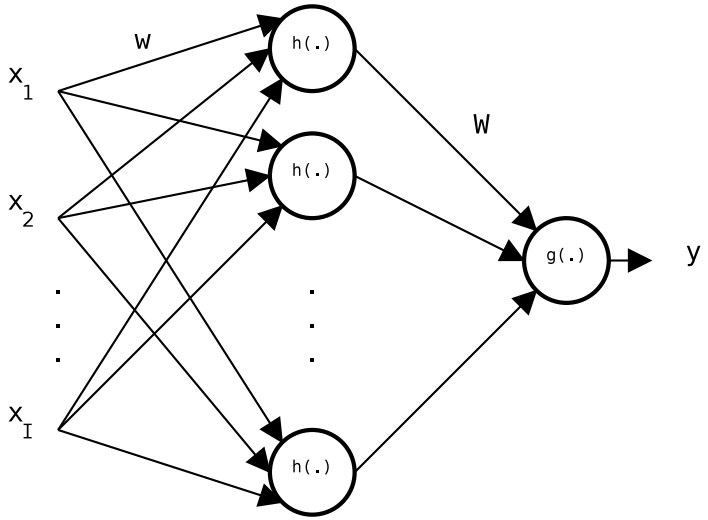
We obtain the Backpropagation update rule for w_{ji}s

$$w_{ji}^{k+1} = w_{ji}^k + 2\eta \sum_n^N (t - g(A)) \cdot g'(A) \cdot W_j \cdot h'(a_j) \cdot x_i$$

Artificial Neural Network Demo

Stolen from:
http://www.obitko.com/tutorials/neural-network-prediction/function-prediction.html

Learning in Multi Layer Perceptrons: Regression (I)



$$y = g(\sum_{j}^{J} W_j \cdot h(\sum_{i}^{I} w_{ji} \cdot x_i))$$

Goal: approximate a target function t given a finite set of N observation

$$t_n = y_n + \epsilon_n \quad \epsilon \sim N(0, \sigma^2) \quad \rightarrow \quad t_n \sim N(y_n, \sigma^2)$$

Thus we have a sample $\mathbf{t}_1, \mathbf{t}_2, \ldots \mathbf{t}_N$ of i.i.d. observations from $N(\mathbf{y}, \mathbf{\Sigma})$

Learning in Artificial Neural Networks $\Rightarrow$ Maximum Likelihood Estimation

Learning in Multi Layer Perceptrons: Regression (II)

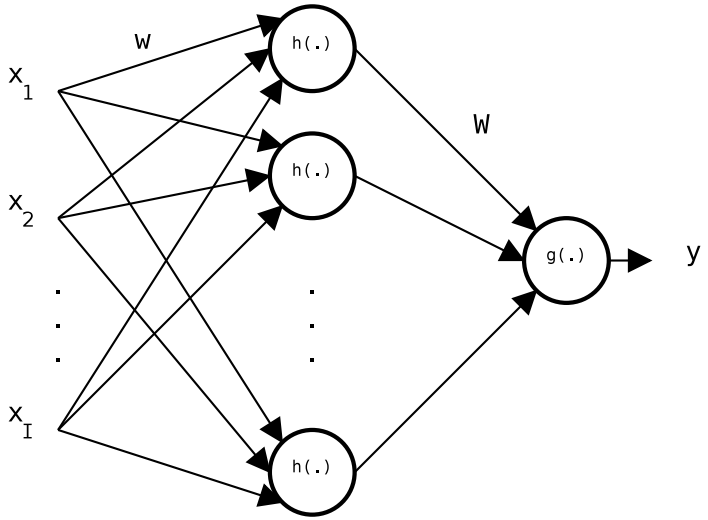
We have a sample $t_1, t_2, \ldots t_N$ of i.i.d. observations from $N(y, \sigma^2)$; define the Likelihood L of the sample as its probability (suppose $k = 1$)

$$L(\mathbf{w}) = \prod_n^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(t_n - y_n)^2}$$

Look for the set of weights that maximize it

$$\begin{aligned}
\arg\max_{\mathbf{w}} L(\mathbf{w}) &= \arg\max_{\mathbf{w}} \sum_n^N \left[\log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2}(t_n - y_n)^2 \right] \\
&= \arg\max_{\mathbf{w}} - \sum_n^N \frac{1}{2\sigma^2}(t_n - y_n)^2 \\
&= \arg\min_{\mathbf{w}} \sum_n^N (t_n - y_n)^2
\end{aligned}$$

Learning in Multi Layer Perceptrons: Classification (I)



$$y = g(\sum_j^J W_j \cdot h(\sum_i^I w_{ji} \cdot x_i))$$

Goal: separate two (or more) classes Ω_0, Ω_1 according to the posterior probability (this time $t = 1$ if $t \in \Omega_1$ and $t = 0$ if $t \in \Omega_0$)

$$p(t|\mathbf{x}) = y^t(1-y)^{1-t} \quad \rightarrow \quad t \sim Be(y)$$

Thus we have a sample $\mathbf{t}_1, \mathbf{t}_2, \ldots \mathbf{t}_N$ of i.i.d. observations from a Bernulli

Learning in Artificial Neural Networks $\Rightarrow$ Maximum Likelihood Estimation

Learning in Multi Layer Perceptrons: Classification (II)

We have a sample $t_1, t_2, \ldots t_N$ of i.i.d. observations from a Bernoulli distribution define the Likelihood L of the sample as its probability

$$L(\mathbf{w}) = \prod_n^N y_n^{t_n}(1-y_n)^{1-t_n}$$

Look for the set of weights that maximize it

$$\begin{aligned}\arg\max_{\mathbf{w}} L(\mathbf{w}) &= \arg\max_{\mathbf{w}} \sum_n^N t_n \log y_n + (1-t_n)\log(1-y_n) \\ &= \arg\min_{\mathbf{w}} -\sum_n^N t_n \log y_n + (1-t_n)\log(1-y_n)\end{aligned}$$

Note: this is called cross entropy and its minimization is equivalent to the minimization of the Kullback-Leibler divergence of the network output and the target distribution

Issues in Learning Artificial Neural Networks

- Improving Convergence
  - Gradient Descend with Momentum
  - Quasi Newton Methods
  - Conjugate Gradient Methods
  - ...
- Local Optima
  - Multiple Restarts
  - Randomized Algorithms
  - ...
- Generalization and Overfitting
  - Early Stopping
  - Weight Decay

Let's focus on generalization and overfitting!

Generalization & Overfitting

Generalization: the model we learnt on a training set is able to produce good results also on new unseen samples.

Overfitting: the model we learnt fits perfectly the training set, but it does not generalize on new samples (it just memorizes the training and its noise).

How do we evaluate generalization/overfitting?

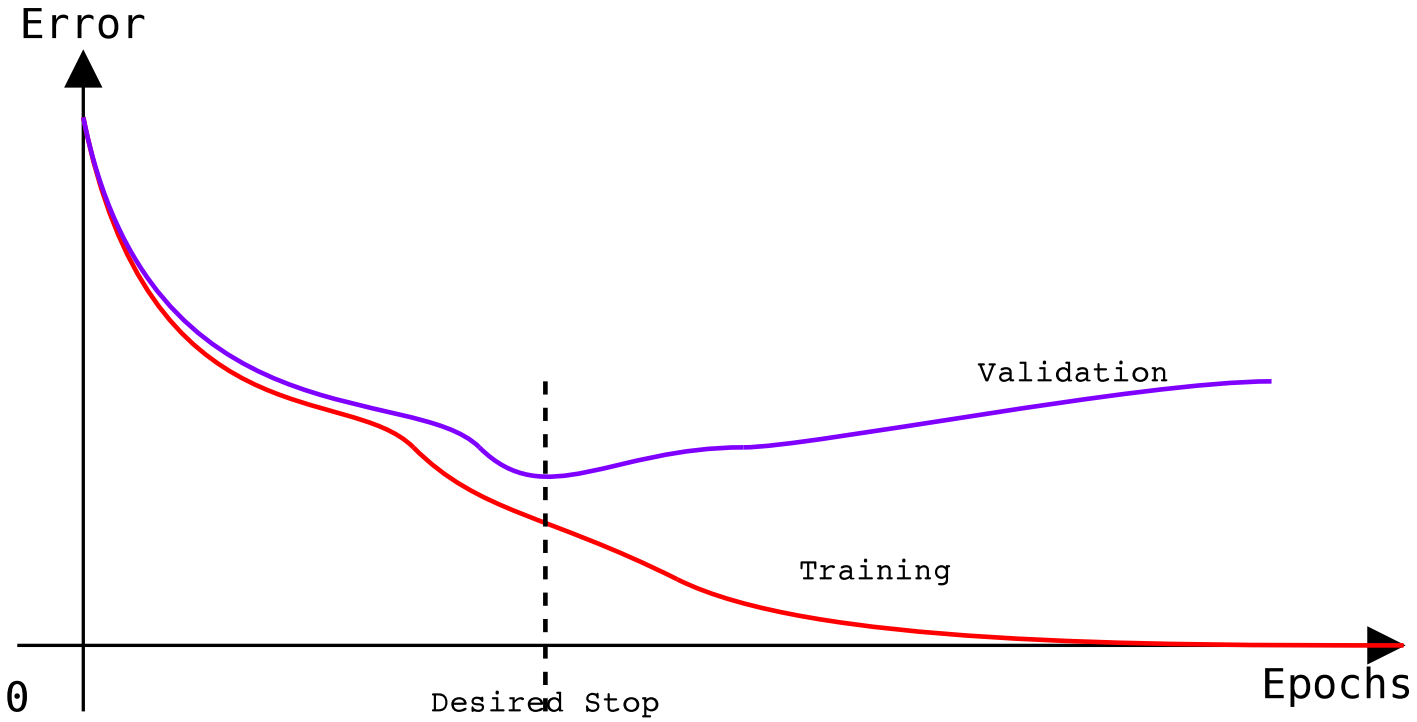
1. Hide some data before learning the model(Test Set)
2. Estimate how well the model predict on "new" data (Test Set Error)

Can we avoid overfitting in Neural Networks?

- Early Stopping: uses data cross-validation to prevent overfitting
- Weight Decay: uses a statistical bias on the model space
- ...

Improving Generalization: Early Stopping

We would like to stop the learning process (a.k.a. optimization routine) before the model starts to fit the noise in the data



This is usually a good method to find out how many neurons we need in the hidden layer: compare different topologies w.r.t. the validation error.

Improving Generalization: Weight Decay (I)

Up to now, we have used Maximum Likelihood Estimation for the weights:

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} P(\mathcal{D}|\mathbf{w})$$

If we use a Bayesian approach, we can use Maximum A-Posteriori:

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} P(\mathbf{w}|\mathcal{D}) = \arg\max_{\mathbf{w}} P(\mathcal{D}|\mathbf{w})P(\mathbf{w})$$

(We "just" need a prior distribution $P(\mathbf{w})$ for the network weights)

From theoretical consideration and empirical results we get:

- Use conjugate priors to get an "easy" posterior distribution
- Small weights improve network generalization capabilities

$$w \sim N(0, \sigma_w^2)$$

Improving Generalization: Weight Decay (II)

$$
\begin{aligned}
\hat{\mathbf{w}} &= \arg\max_{\mathbf{w}} P(\mathcal{D}|\mathbf{w})P(\mathbf{w}) \\
&= \arg\max_{\mathbf{w}} \prod_n^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(t_n - y_n)^2} \cdot \prod_m^M \frac{1}{\sqrt{2\pi}\sigma_w} e^{-\frac{1}{2\sigma_w^2}(0 - w_m)^2} \\
&= \arg\min_{\mathbf{w}} \sum_n^N \frac{1}{2\sigma^2}(t_n - y)^2 + \sum_m^M \frac{1}{2\sigma_w^2} w^2 \\
&= \arg\min_{\mathbf{w}} \sum_n^N (t_n - y)^2 + \gamma \sum_m^M w^2
\end{aligned}
$$

This way we penalize “network complexity” introducing a bias