Knowledge Engineering
Lecture Notes on Natural Computation

Matteo Matteucci

matteucci@elet.polimi.it

Department of Electronics and Information

Politecnico di Milano

Neural Networks
– Introduction –

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
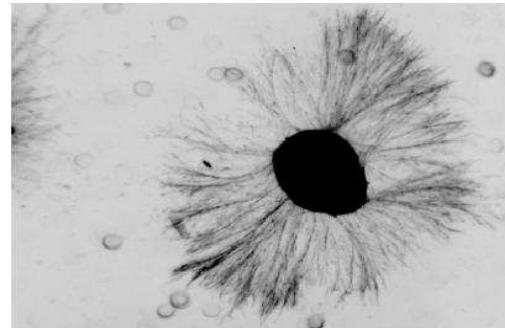
The computational model of the brain is:

- <u>Distributed</u> among simple units called neurons
- Intrinsically <u>parallel</u>
- Redundant and thus <u>fault tolerant</u>

Computation in Biological Neurons
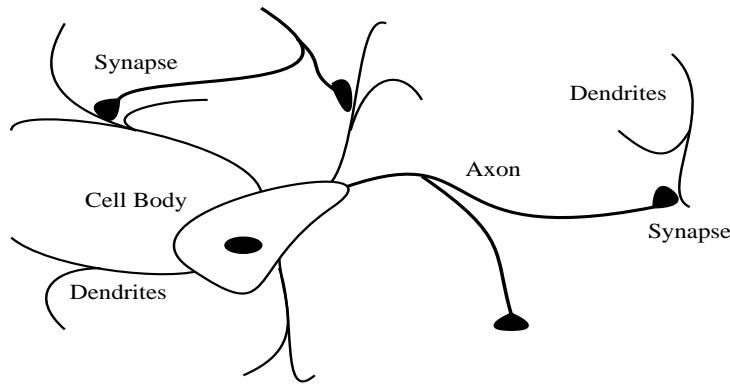
Information is transmitted through chemical mechanisms:

- 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
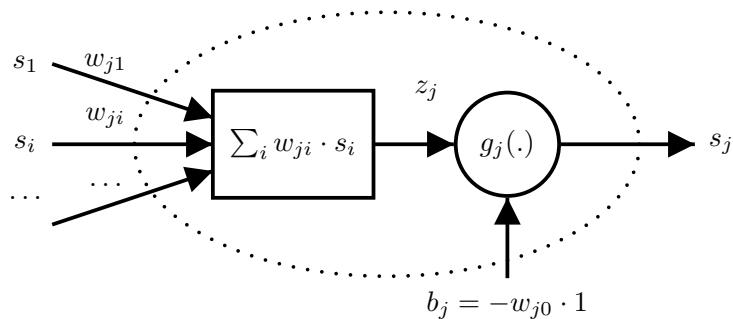
In this simple model of neuron j we can identify:

- 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: $s_j = g_j(\sum_{i=1}^{N} w_{ji} s_i - b_j) = g_j(\sum_{i=0}^{N} w_{ji} s_i)$

Common Activation Functions

Sign function: $g(z_j) = \begin{cases} 1 & \text{if } z_j > 0 \\ 0 & \text{if } z_j = 0 \\ -1 & \text{if } z_j < 0 \end{cases}$
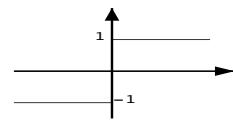
Linear function: $g(z_j) = z_j$
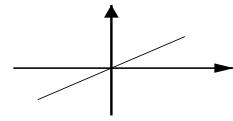
Sigmoid function: $g(z_j) = \dfrac{1}{1 + e^{-z_j}}$
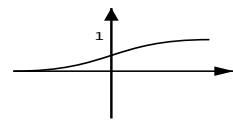
Hyperbolic tangent: $g(z_j) = \dfrac{e^{z_j} - e^{-z_j}}{e^{z_j} + e^{-z_j}}$
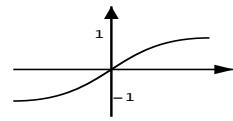
The Perceptron

The first model of neuron proposed was the Perceptron:
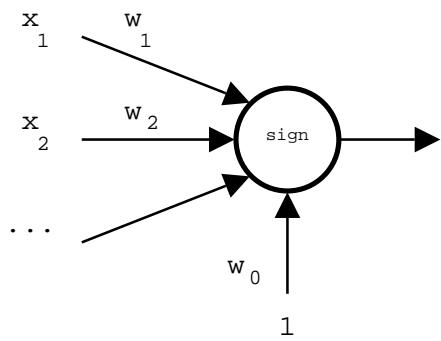
Sign function: $g(z_j) = \begin{cases} 1 & \text{if } z_j > 0 \\ 0 & \text{if } z_j = 0 \\ -1 & \text{if } z_j < 0 \end{cases}$

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

What can I do with such a simple model?

The Perceptron as a Logic Operator

Perceptron as a logic AND

$$w_1 = 3/2$$

$$w_2 = 1$$

$$w_0 = -2$$
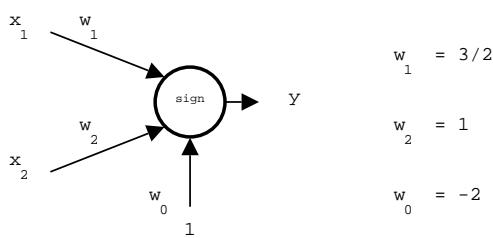
x_1	x_2	y
0	0	-1
0	1	-1
1	0	-1
1	1	1

Perceptron as a logic OR

$$w_1 = 1$$

$$w_2 = 1$$
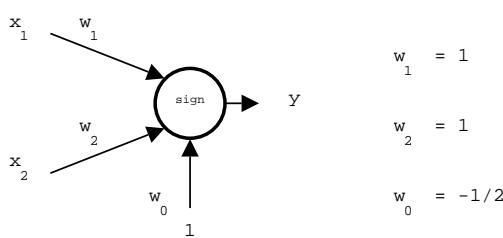
$$w_0 = -1/2$$

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

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:
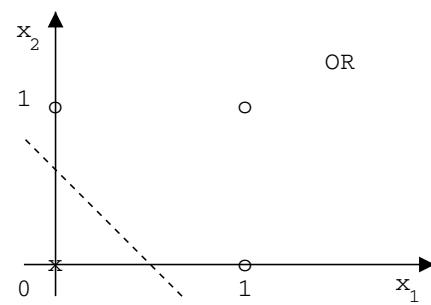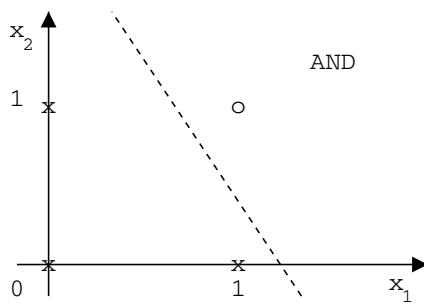
Hebb Learning Rule

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:

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

Where we have

- η: learning rate
- x_i: the ith 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:
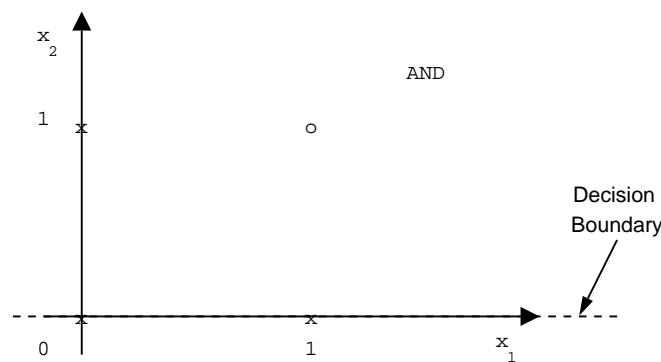
Hebbian Learning of the AND Operator (1)

Epoch 1

- Record 1:
  - $w_1 = 0 + 0 = 0$
  - $w_2 = 1 + 0 = 1$
  - $w_0 = 0 - 1/2 = -1/2$
- Record 2:
  - $w_1 = 0 + 0 = 0$
  - $w_2 = 1 - 1/2 = 1/2$
  - $w_0 = -1/2 - 1/2 = -1$
- Record 3: Ok
- Record 4:
  - $w_1 = 0 + 1/2 = 1/2$
  - $w_2 = 1/2 + 1/2 = 1$
  - $w_0 = -1 + 1/2 = -1/2$

x_1	x_2	x_0	y
0	0	1	-1
0	1	1	-1
1	0	1	-1
1	1	1	1
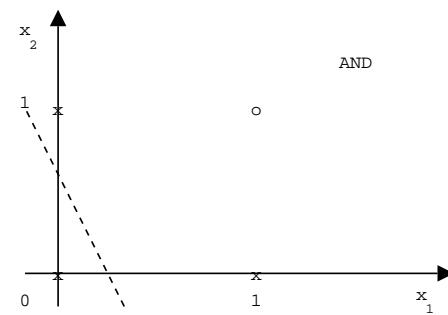
Hebbian Learning of the AND Operator (2)

Epoch 2

- Record 1: OK
- Record 2:
  - $w_1 = 1/2 + 0 = 1/2$
  - $w_2 = 1 - 1/2 = 1/2$
  - $w_0 = -1/2 - 1/2 = -1$
- Record 3: Ok
- Record 4:
  - $w_1 = 1/2 + 1/2 = 1$
  - $w_2 = 1/2 + 1/2 = 1$
  - $w_0 = -1 + 1/2 = -1/2$

x_1	x_2	x_0	y
0	0	1	-1
0	1	1	-1
1	0	1	-1
1	1	1	1
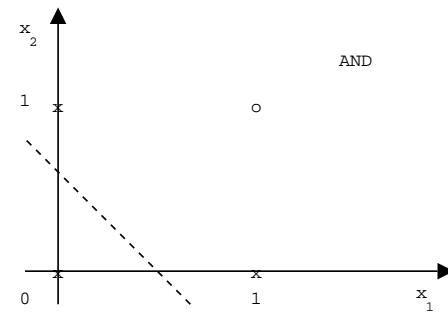
Hebbian Learning of the AND Operator (3)

Epoch 3

- Record 1: Ok
- Record 2:
  - $w_1 = 1 + 0 = 1$
  - $w_2 = 1 - 1/2 = 1/2$
  - $w_0 = -1/2 - 1/2 = -1$
- Record 3:
  - $w_1 = 1 - 1/2 = 1/2$
  - $w_2 = 1/2 - 0 = 1/2$
  - $w_0 = -1 - 1/2 = -3/2$
- Record 4:
  - $w_1 = 1/2 + 1/2 = 1$
  - $w_2 = 1/2 + 1/2 = 1$
  - $w_0 = -3/2 + 1/2 = -1$

x_1	x_2	x_0	y
0	0	1	-1
0	1	1	-1
1	0	1	-1
1	1	1	1
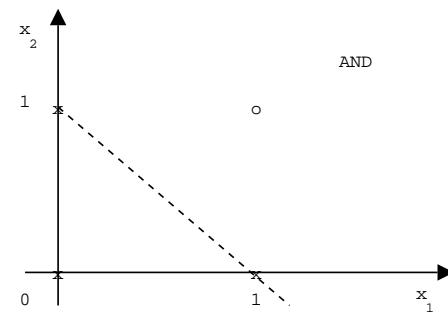
Hebbian Learning of the AND Operator (...)

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

Hebbian Learning of the AND Operator (7)

x_1	x_2	x_0	y
0	0	1	-1
0	1	1	-1
1	0	1	-1
1	1	1	1

Epoch 8

- Record 1: Ok
- Record 2:
  - $w_1 = 3/2 - 0 = 3/2$
  - $w_2 = 3/2 - 1/2 = 1$
  - $w_0 = -3/2 - 1/2 = -2$
- Record 3: OK
- Record 4: OK
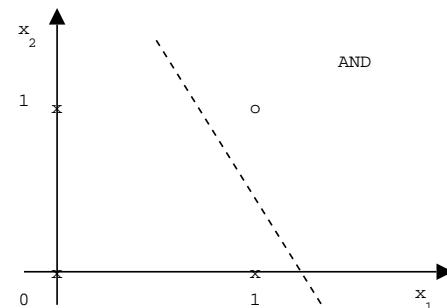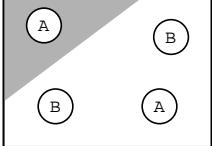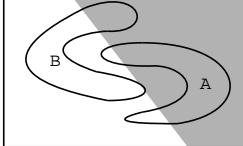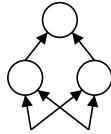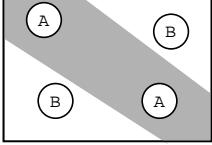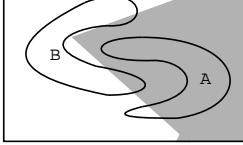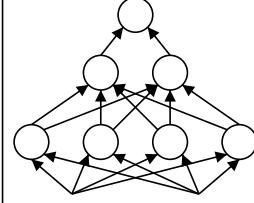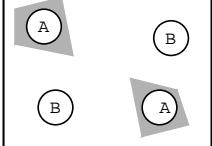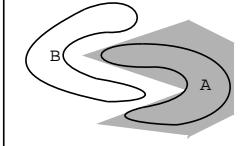
The XOR Problem

Great, but what if we have a non linearly separable problem?
(i.e., The XOR problem, Minsky '69)
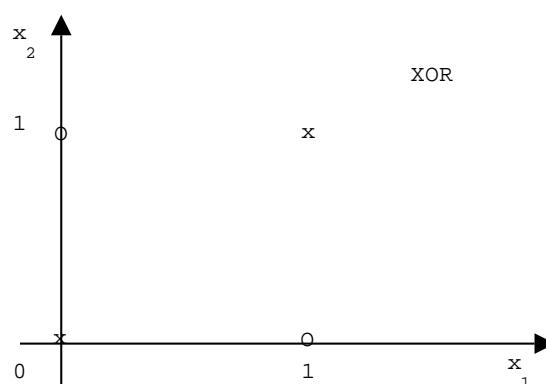
Wait until the '80s and you'll see!

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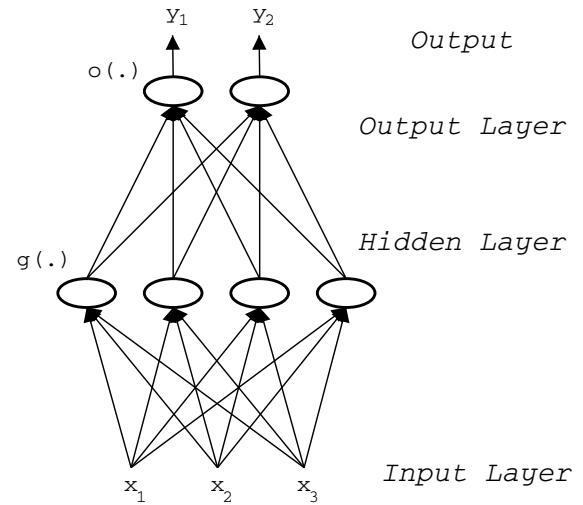
An artificial neural network is a **non-linear model** characterized by the number of neurons, their topology, activation functions, and the values of synaptic weights and biases.

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:

$$\mathbf{w}^{n+1} = \mathbf{w}^n + \Delta\mathbf{w}$$
$$\Delta\mathbf{w} = -\eta \cdot \frac{\partial E}{\partial \mathbf{w}}$$

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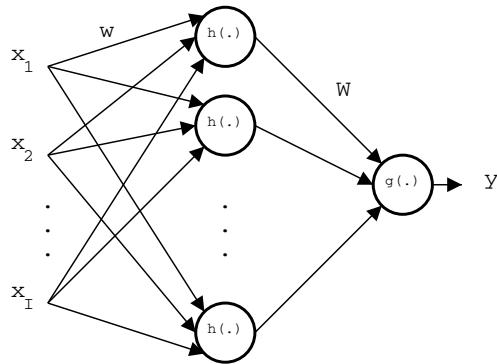
$$y = g(\sum_j^J W_j \cdot h(\sum_i^I w_{ji} \cdot x_i))$$

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

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 jth 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

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

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

Artificial Neural Network Demo

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

Learning in Multi Layer Perceptrons: Regression (I)
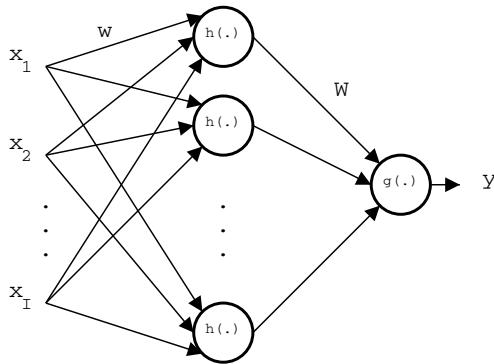
$$y = g(\sum_{j}^{J} W_j \cdot h(\sum_{i}^{I} w_{ji} \cdot x_i))$$

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

$$\arg\max_{\mathbf{w}} L(\mathbf{w}) = \arg\max_{\mathbf{w}} \sum_{n}^{N} \left[\log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2}(t_n - y_n)^2 \right]$$

$$= \arg\max_{\mathbf{w}} - \sum_{n}^{N} \frac{1}{2\sigma^2}(t_n - y_n)^2$$

$$= \arg\min_{\mathbf{w}} \sum_{n}^{N} (t_n - y_n)^2$$

Learning in Multi Layer Perceptrons: Classification (I)
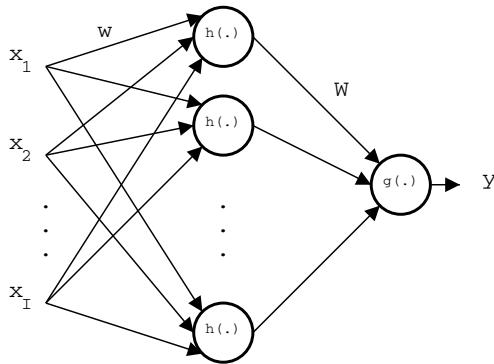
$$y = g\left(\sum_{j}^{J} W_j \cdot h\left(\sum_{i}^{I} w_{ji} \cdot x_i\right)\right)$$

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 $\mathbf{t}_1, \mathbf{t}_2, \dots \mathbf{t}_N$ 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 $t_1, t_2, \dots t_N$ 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 <u>cross entropy</u> 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

Overfitting: we have overfitting when the model we learnt fits the noise in the data, thus it does not generalize on new samples (it just memorizes the training set).

Can we measure generalization?

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?

- Use cross-validation techniques
- Use statistical bias on the model space

General rule: Use Occam's razor!
"Entia non sunt multiplicanda praeter necessitatem"

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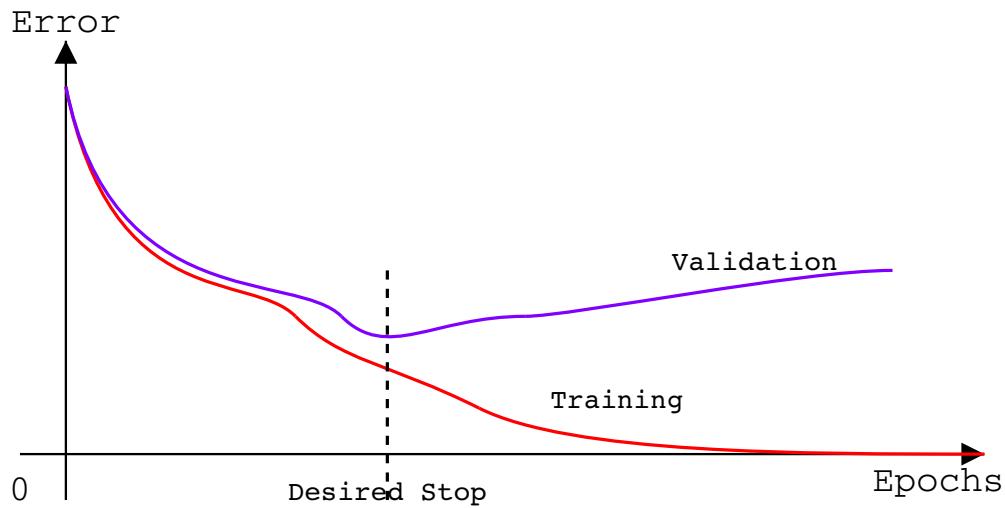
This is usually a good method to find out how many neurons we need in the hidden layer: compare different topologies w.r.t. the validation error.

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

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

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

Neural Networks
– Radial Basis Functions –

Radial Basis Approximation

Consider the following function approximation:

$$\hat{f}(\mathbf{x}) = w_0 + \sum_{u=1}^{U} w_u \cdot K_u(d(\mathbf{x}_u, \mathbf{x}))$$

Where we have:

- $\mathbf{x}_u$ is an instance from $\mathbf{X}$
- $K_u(d(\mathbf{x}_u, \mathbf{x}))$ is called <u>Kernel function</u> and it is defined so that it decrease as the distance $d(\mathbf{x}_u, \mathbf{x})$ increase

$\hat{f}(\mathbf{x})$ is a global approximation of $f(\mathbf{x})$ obtain from the local contributions of $\mathbf{x}_u$s and it is possible approximate any function with arbitrarily small error, provided a sufficiently large number U of radial basis functions.

Radial Basis Functions

Consider to use a Gaussian function as Kernel

$$K_u(d(\mathbf{x}_u, \mathbf{x})) = e^{-\frac{1}{2\sigma^2} d^2(\mathbf{x}_u, \mathbf{x})}.$$

We can rewrite the radial bases approximation as a two layer artificial neural network (called Radial Basis Function):
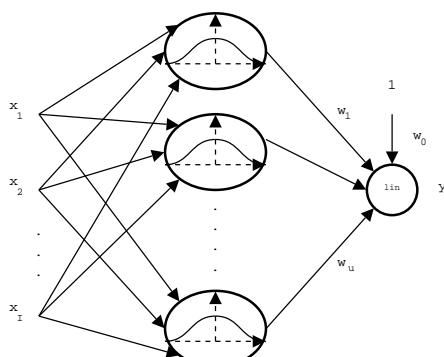
$$y = w_0 + \sum_{u=1}^{U} w_u \cdot e^{-\frac{1}{2\sigma^2} d^2(\mathbf{x}_u, \mathbf{x})}$$

Where Does $d(\mathbf{x}_u, \mathbf{x})$ Come From?

Consider the classical formula to compute the activation value for the jth hidden neuron as in feed-forward neural network (our u^{th} radial base)

$$z_j = \sum_i^I w_{ji} \cdot x_i$$

This can be written as the scalar (dot) product between the input vector $\vec{x}$ and the weigth vector of jth hidden unit $\mathbf{w}_j$:

$$z_j = \sum_i^I w_{ji} \cdot x_i = \sum_i^I x_i \cdot w_{ji} = \mathbf{x}.\mathbf{w}_j^T$$

Just rename j as u and $\mathbf{w}_j$ as $\mathbf{x}_u$ to get $d(\mathbf{x}_u, \mathbf{x}) = \mathbf{x}.\mathbf{x}_u^T$.

The **scalar product** is used as distance between
the input vector and the basis centers.

Learning with Radial Basis Functions

Radial Basis Function are typically trained in a two stage process:

1. The number U of hidden units is determined and each hidden unit is defined by choosing the values of $\mathbf{x}_u$ and σ_u^2 that define its kernel function $K_u(d(\mathbf{x}_u, \mathbf{x}))$

2. The weights w_u are trained to maximize the fitting of the network using the error function

$$E = \frac{1}{2} \sum_n^N (f(\mathbf{x}_n) - \hat{f}(\mathbf{x}_n))^2$$

Since the kernel functions are held fixed during the second stage, the
linear weights w_u can be trained very efficiently!

Linear Regression to Learn the Weighs (I)

According to the RBF model we can consider the network output $\mathbf{Y}$ as a linear combination of hidden neurons output $\mathbf{U}$:

$$\mathbf{Y} = \mathbf{U}\mathbf{w}$$

Learning is thus the solution of N linear equations in U unknowns that we can write using matrix notation:

$$
\begin{vmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{vmatrix}
=
\begin{vmatrix}
h(\sum_i^I w_{1i}x_i)_1 & h(\sum_i^I w_{2i}x_i)_1 & \cdots & h(\sum_i^I w_{Ui}x_i)_1 \\
h(\sum_i^I w_{1i}x_i)_2 & h(\sum_i^I w_{2i}x_i)_2 & \cdots & h(\sum_i^I w_{Ui}x_i)_2 \\
\vdots & \vdots & \vdots & \vdots \\
h(\sum_i^I w_{1i}x_i)_N & h(\sum_i^I w_{2i}x_i)_N & \cdots & h(\sum_i^I w_{Ui}x_i)_N
\end{vmatrix}
\cdot
\begin{vmatrix} w_1 \\ w_2 \\ \vdots \\ w_U \end{vmatrix}
$$

$$[N \times 1] = [N \times U] \cdot [U \times 1]$$

Linear Regression to Learn the Weighs (II)

Suppose we have no noise, exact model and well-conditioned $\mathbf{U}$ matrix we can obtain a closed form for the solution (note we cannot invert directly $\mathbf{U}$ since it could be not invertible):

$$
\begin{aligned}
\mathbf{Y} &= \mathbf{U}\mathbf{w} \\
\mathbf{U}'\mathbf{Y} &= \mathbf{U}'(\mathbf{U}\mathbf{w}) \\
\mathbf{U}'\mathbf{Y} &= (\mathbf{U}'\mathbf{U})\mathbf{w} \\
(\mathbf{U}'\mathbf{U})^{-1}\mathbf{U}'\mathbf{Y} &= (\mathbf{U}'\mathbf{U})^{-1}(\mathbf{U}'\mathbf{U})\mathbf{w} \\
(\mathbf{U}'\mathbf{U})^{-1}\mathbf{U}'\mathbf{Y} &= \mathbf{w}
\end{aligned}
$$

This is not iterative! It could be computationally intensive, but "years" of linear algebra have discovered efficient solutions for computing $(\mathbf{U}'\mathbf{U})^{-1}$ even with ill-conditioned matrixes ... check out for that on:

Numerical recipes in Fortran/C/C++

Radial Basis Function Demo

Stolen from:
http://lcn.epfl.ch/tutorial/english/rbf/html/index.html

Setting Up the Kernels

We can find different approaches for the choice of the kernel functions:

1. Allocate a Gaussian kernel function for each training example
   $< \mathbf{x}_n, f(\mathbf{x}_n) >$ centering it at $\mathbf{x}_n$ and assigning some width σ^2.

   - Global approximation from local interactions of data points
   - Good fitting of the training data
   - We could have problem with overfitting and performance

2. Use a relatively small U, place one neuron at the time, train the network and place a new neuron on the record with the higher prediction error.

   - Faster that using a kernel function for each data point
   - Improved generalization capabilities

Neural Networks
– Recurrent Architectures –

Neural Networks for Dynamical Systems

A new Task: <u>we want to predict the next day stock market average $y(t+1)$ based on the current day's economic indicators $\mathbf{x}(t)$</u>

We have mainly three approaches to do that with artificial neural networks:

- Standard Feedforward: we can do regression from $\mathbf{x}(t)$ to $y(t+1)$, but we cannot capture dependencies on earlier values of $\mathbf{x}$

- Feedforward with Delayed Input: we can replicate a finite number of earlier values of $\mathbf{x}$ and apply regression from $[\mathbf{x}(t)\mathbf{x}(t-1)\dots\mathbf{x}(t-q)]$, but we do not know how much is q

- Recurrent Artificial Neural Networks: we can add a new unit b to the hidden layer and a new input unit $c(t)$ to represent the value of b at time $(t-1)$. b thus can summarize information from earlier values of $\mathbf{x}$ arbitrarily distant in time.

Feedforward with Delayed Input
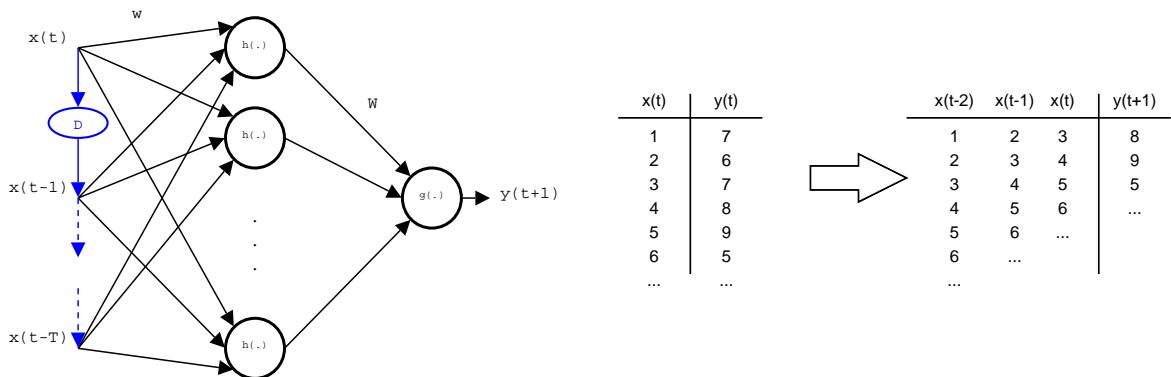
How can we learn the weights for this kind of network?

- Build-up a <u>delayed</u> dataset from the original one
- Use the standard backpropagation learning algorithm

$$y(t+1) = g(\sum_j^J W_j \cdot h(\sum_{i=0}^T w_{ji} \cdot x(t-i)))$$

Recurrent Artificial Neural Networks
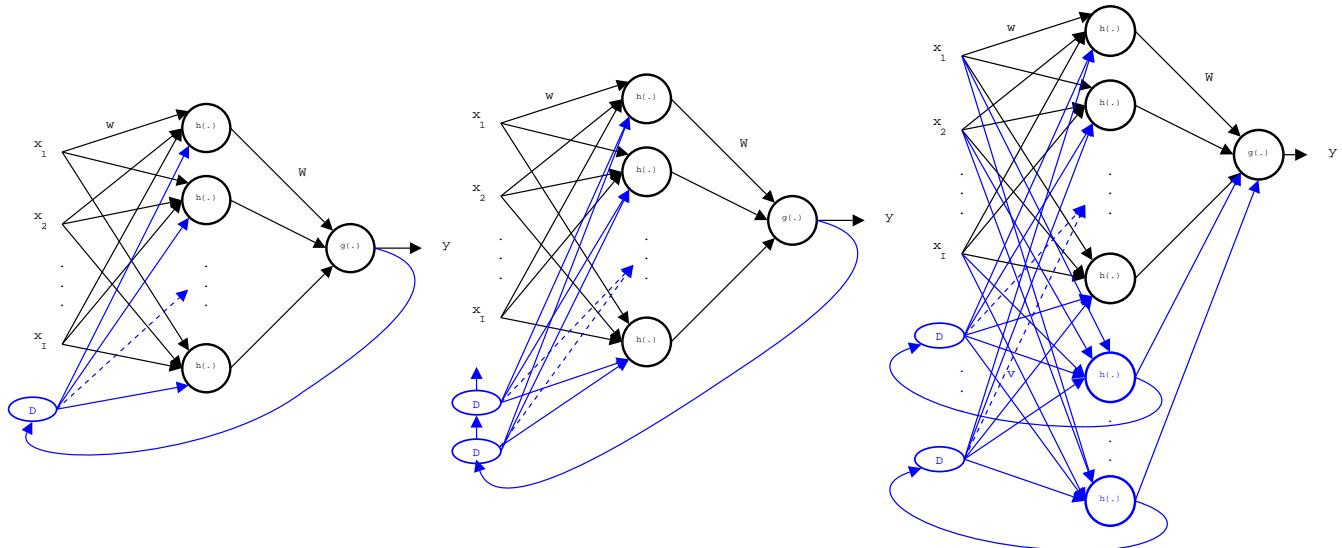
1. Recurr the output as input (network output → equilibrium)
2. Recurr the output (and its previous values) as input
3. Recurr the internal (hidden) state of the network as input

Finding Structure in Time (Elman Networks)
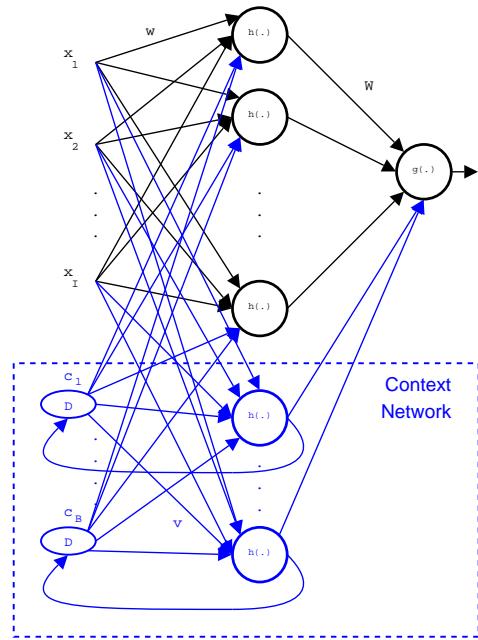
$$y^t = g(\sum_j^J W_j \cdot h(\sum_b^B v_{jb} \cdot c_b^{t-1} + \sum_i^I w_{ji} \cdot x_i^t) +$$

$$+ \sum_b^B W_b \cdot h(\sum_{b'}^B v_{bb'} \cdot c_{b'}^{t-1} + \sum_i^I v_{bi} \cdot x_i^t))$$

$$c_b^t = h(\sum_{b'}^B v_{bb'} \cdot c_{b'}^{t-1} + \sum_i^I v_{bi} \cdot x_i^t)$$

How can we learn the weights for this kind of networks?

Backpropagation Through Time (Elman Networks)

1. Perform "network unfolding"
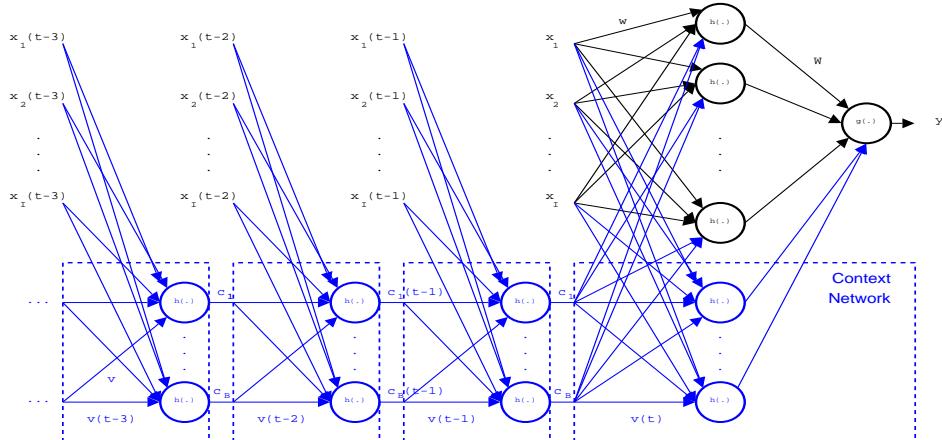
2. Approximate backpropagation by:
   - Use a finite number unwrapping steps U
   - Take the average of weigths over time: $v_{jb} = \frac{1}{U} \sum_u^U v_{jb}(t-u)$
   - Compute the sum of gradients over time: $\frac{\partial E}{\partial v_{jb}} = \sum_u^U \frac{\partial E}{\partial v_{jb}(t-u)}$

Neural Networks
– Neuro-Fuzzy Networks –

Neuro vs Fuzzy Modeling

Both Neural Networks and Fuzzy Rules are **non-linear** modeling paradigms robust with respect to **uncertanty** (noise) in the data. In theory, they are equivalent, yet in practice each has its own pros and cons:

- Neural networks
  - Knowledge automatically acquired from experience $\rightarrow$good :)
  - No explanation on what's going on in the black-box $\rightarrow$bad :(
  - No way to extract/integrate structural knowledge $\rightarrow$bad :(
- Fuzzy rules
  - Reasoning on imprecise information $\rightarrow$good :)
  - Clear understanding of the inference mechanism $\rightarrow$good :)
  - No way of learning experience directly from data $\rightarrow$bad :(

Can we get the best of the two worlds from a single (hybrid) system?
Can we extend neural network to extract fuzzy rules from data?

Neural Learning of Sugeno-Type Fuzzy Inference (I)

Suppose we have the the fuzzy system having rules in the form:

$$R_j : \text{ if } x_1 \text{ is } A_{j1} \wedge x_2 \text{ is } A_{j2} \wedge \ldots \wedge x_I \text{ is } A_{jI} \text{ then } y = z_j$$

Being A_{ji} fuzzy numbers of triangular form and z_j real numbers, the system can be represented as a non-linear mapping:

$$y = f(\mathbf{x}) = f(x_1, \ldots x_I)$$

- Define the firing level of jth rule by the Larsen's product operator (or any other t-norm for the `and` operator): $\alpha_j = \prod_{i=1}^{I} A_{ji}(x_i)$

- The output of the system is thus defined as: $y = \frac{\sum_j \alpha_j z_j}{\sum_j \alpha_j}$

  <u>Can we learn the shapes of fuzzy numbers A_{ji} and output z_j from experience $\mathcal{D} = \{(\mathbf{x}, t)_1, \ldots, (\mathbf{x}, t)_n, \ldots, (\mathbf{x}, t)_N\}$?</u>

Neural Learning of Sugeno-Type Fuzzy Inference (II)

To better understand how this is related to neural networks and learning consider a <u>simplified example</u>.

- Suppose we have a system formed by just two rule with two input and one output variable:

$$\begin{aligned} R_1 &: \quad \text{if } x_1 \text{ is } A_1 \wedge x_2 \text{ is } A_2 \text{ then } y = z_1 \\ R_2 &: \quad \text{if } x_1 \text{ is } A_2 \wedge x_2 \text{ is } A_1 \text{ then } y = z_2 \end{aligned}$$

  with fuzzy terms A_1 (`small`) and A_2 (`big`) having a sigmoidal membership function defined by: $A_k(x) = \frac{1}{1 + \exp(b_k(x - a_k))}$.

- Define now a three new special neurons ...
  1. $P(.)$: performing the Product of its input
  2. $S(.)$: performing the Sum of its input
  3. $R(.)$: performing the Ratio of its input

  Here it is your first **Fuzzy-Neural Network** ... o well on the next slide ;)

Neural Learning of Sugeno-Type Fuzzy Inference (III)
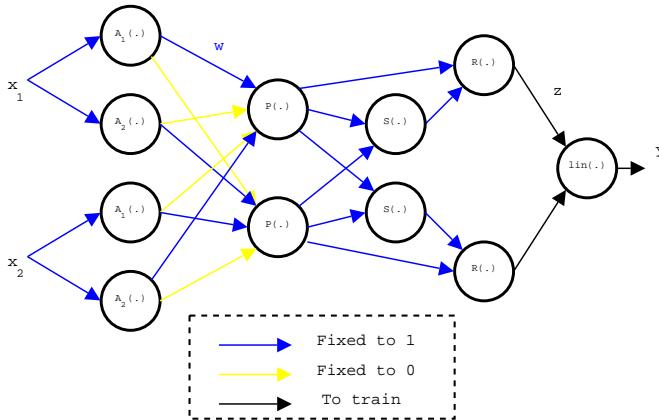
$$y = \sum_{j=1}^{2} \frac{z_j \cdot \alpha_j}{\alpha_1 + \alpha_2}$$

$$= \frac{z_1 \cdot \alpha_1 + z_2 \cdot \alpha_2}{\alpha_1 + \alpha_2}$$

Now you got the idea, all remains to do is learning z_j, a_k and b_k so let's:

1. Define your favourite Error measure over the training samples:

$$E = \tfrac{1}{2} \sum_{n}^{N} (y_n - t_n)^2$$

2. Run your fashioned steepest descend optimization package on the network ... or do it by hand!

Neural Learning of Sugeno-Type Fuzzy Inference (IV)

$$\frac{\partial E}{\partial z_j} = \frac{\partial}{\partial z_j} \frac{1}{2} \sum_{n}^{N} (y_n - t_n)^2 = \sum_{n}^{N} \left((y_n - t_n)\frac{\partial y_n}{\partial z_j}\right) = \sum_{n}^{N} \left((y_n - t_n)\frac{\alpha_j(\mathbf{x}_n)}{\sum_j \alpha_j(\mathbf{x}_n)}\right)$$

$$\frac{\partial E}{\partial a_i} = \frac{\partial}{\partial a_i} \frac{1}{2} \sum_{n}^{N} (y_n - t_n)^2 = \sum_{n}^{N} \left((y_n - t_n)\frac{\partial y_n}{\partial a_i}\right) = \ldots$$

$$\frac{\partial E}{\partial b_i} = \frac{\partial}{\partial b_i} \frac{1}{2} \sum_{n}^{N} (y_n - t_n)^2 = \sum_{n}^{N} \left((y_n - t_n)\frac{\partial y_n}{\partial b_i}\right) = \ldots$$

In 1993 R. Jang showed that fuzzy inference systems with simplified fuzzy rules (i.e., having crisp outputs) are universal approximators. The more fuzzy terms and rules are used, the closer is the output of the fuzzy system to the desired one.

Does this remind you about anything?

Adaptive Neuro-Fuzzy Inference System [ANFIS] (I)

Consider now a more complex example using the following fuzzy rules:

$$R_1 \quad : \quad \text{if } x_1 \text{ is } L_1 \wedge x_2 \text{ is } L_2 \wedge x_3 \text{ is } L_3 \text{ then } y = VB$$
$$R_2 \quad : \quad \text{if } x_1 \text{ is } H_1 \wedge x_2 \text{ is } H_2 \wedge x_3 \text{ is } L_3 \text{ then } y = B$$
$$R_3 \quad : \quad \text{if } x_1 \text{ is } H_1 \wedge x_2 \text{ is } H_2 \wedge x_3 \text{ is } H_3 \text{ then } y = S$$

Being x_1, x_2, x_3 the exchange rates USD vs DEM, USD vs SEK, and USD vs FIM, we obtain:

R_1 if the USD is weak against DEM, SEK, and FIM then our portfolio value is very big

R_2 if the USD is strong against DEM, SEK, and USD is weak against FIM then our portfolio value is big

R_3 if the USD is strong against DEM, SEK, and FIM then our portfolio value is small big

Note: this was a pre-Euro Swedish example :)

Adaptive Neuro-Fuzzy Inference System [ANFIS] (II)

Define the fuzzy sets for the input and output variables:

- LOW is $L_i(x) = \frac{1}{1+exp(b_i(x-c_i))}$

- HIGH is $H_i(x) = 1 - L_i(x) = \frac{1}{1+exp(-b_i(x-c_i))}$

- VERY_SMALL is $VS(x) = \frac{1}{1+exp(b_S(x-c_S-c_V))}$

- VERY_BIG is $VB(x) = 1 - VS(x) = 1 - \frac{1}{1+exp(b_S(x-c_S-c_V))}$

- SMALL is $S(x) = \frac{1}{1+exp(b_S(x-c_S))}$

- BIG is $B(x) = 1 - S(x) = 1 - \frac{1}{1+exp(-b_S(x-c_S))}$

Let's evaluate the daily portfolio value by Tsukamoto's reasoning:

1. The firing level is obtained by: $\alpha_j = L_1(x_1) \wedge L_2(x_2) \wedge L_3(x_3)$.

2. The output is derived: $z_j = \text{FuzzySet}^{-1}(\alpha_j)$

3. The overall output is composed: $y = \frac{\sum_j \alpha_j \cdot z_j}{\sum_j \alpha_j}$

# Adaptive Neuro-Fuzzy Inference System [ANFIS] (III)

Let's evaluate the first rule by Tsukamoto's reasoning:

1. The firing level is obtained by: $\alpha_1 = L_1(x_1) \wedge L_2(x_2) \wedge L_3(x_3)$.

2. The output is derived: $z_1 = VB^{-1}(\alpha_j) = c_S + c_V + \frac{1}{b_B} \ln \frac{1-\alpha_1}{\alpha_1}$

3. The overall output is composed: $y = \frac{\alpha_1 \cdot z_1 + \alpha_3 \cdot z_2 + \alpha_3 \cdot z_3}{\alpha_1 + \alpha_2 + \alpha_3}$
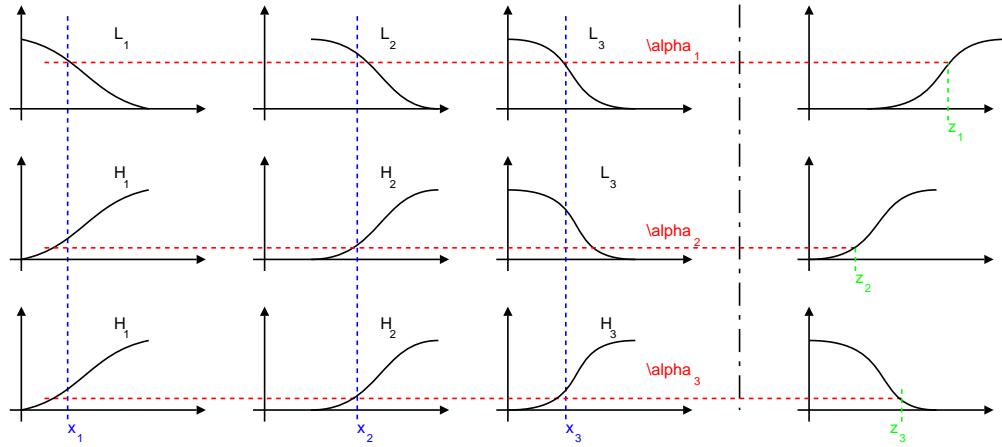

Let's put it in a neural network!

# Adaptive Neuro-Fuzzy Inference System [ANFIS] (IV)
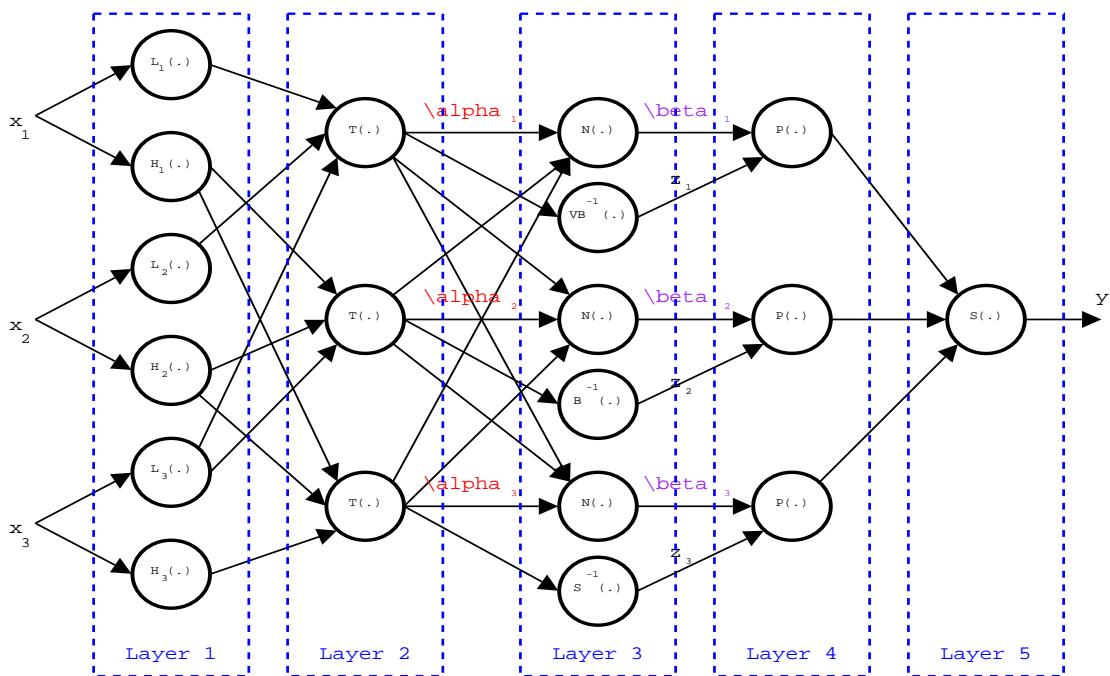
Layer 1: The output of the nodes is the degree to which the give input satisfies the linguistic label

# Adaptive Neuro-Fuzzy Inference System [ANFIS] (IV)
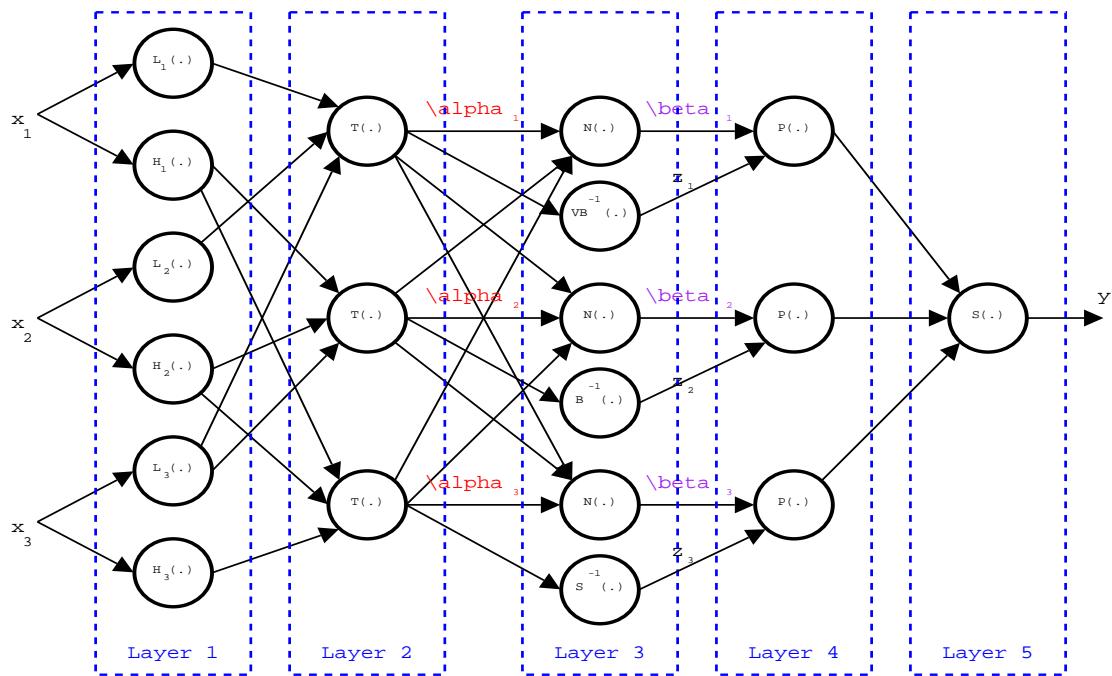
Layer 2: Each node computes the firing strength of the associated rule (rule nodes) using a t-norm operator

# Adaptive Neuro-Fuzzy Inference System [ANFIS] (IV)
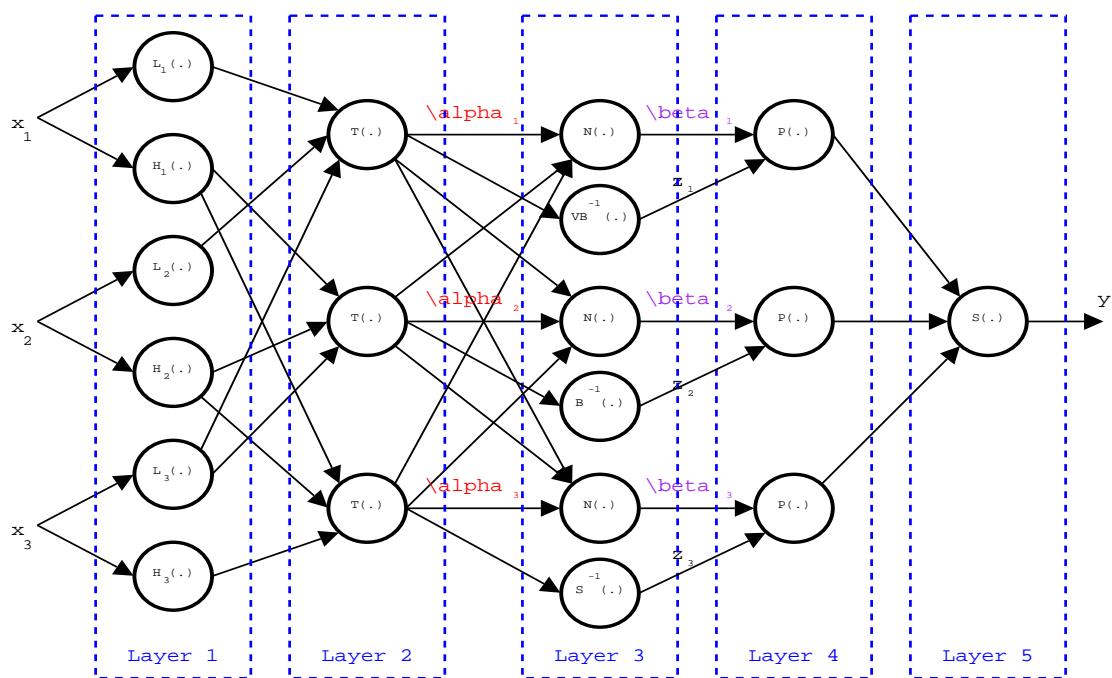
Layer 3: Nodes labeled $N(.)$ perform a normalization of firing level $\beta_j = \frac{\alpha_j}{\sum_j \alpha_j}$. Others compute rule output (i.e., $z_1 = VB^{-1}(\alpha_1)$)

Adaptive Neuro-Fuzzy Inference System [ANFIS] (IV)
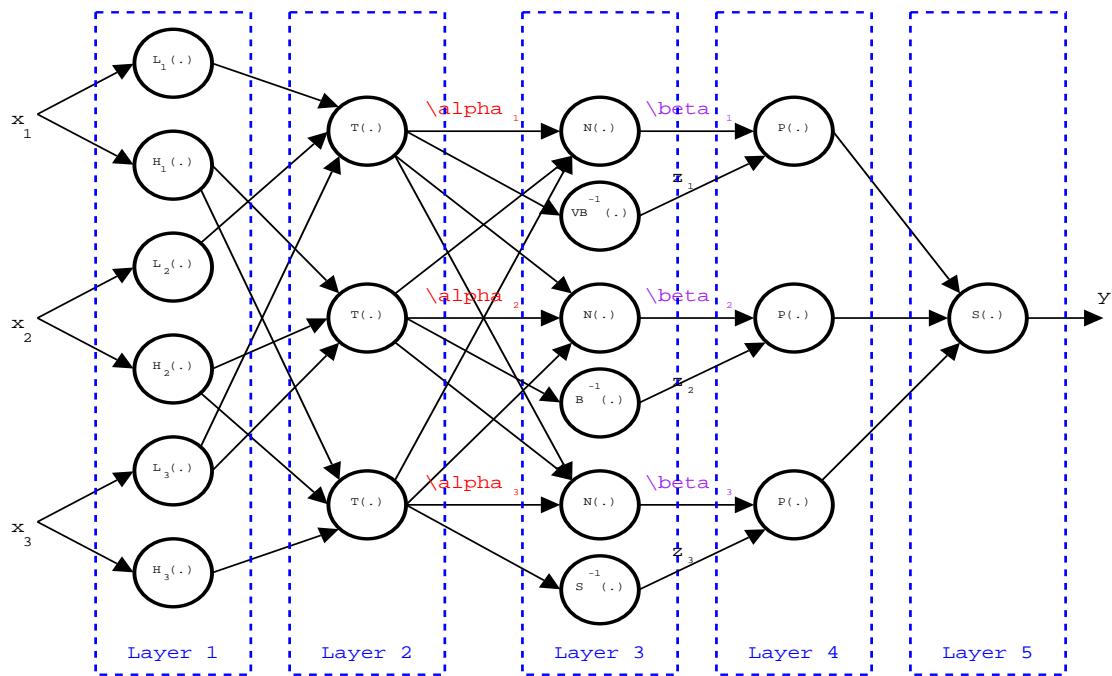
Layer 4: Node in this layer compute the product of the normalized strength β_j and the rule output z_j

Adaptive Neuro-Fuzzy Inference System [ANFIS] (IV)
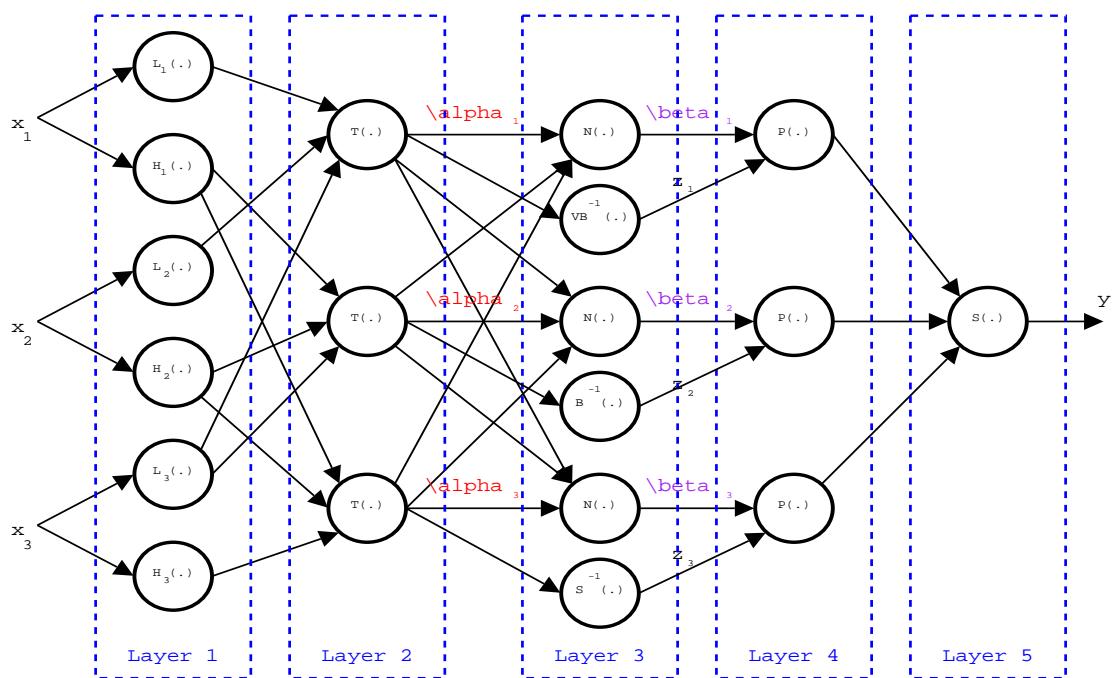
Layer 5: The single node in this layer computes the overall system output as the sum of all incoming signals $y = \sum_j z_j$

Adaptive Neuro-Fuzzy Inference System [ANFIS] (IV)
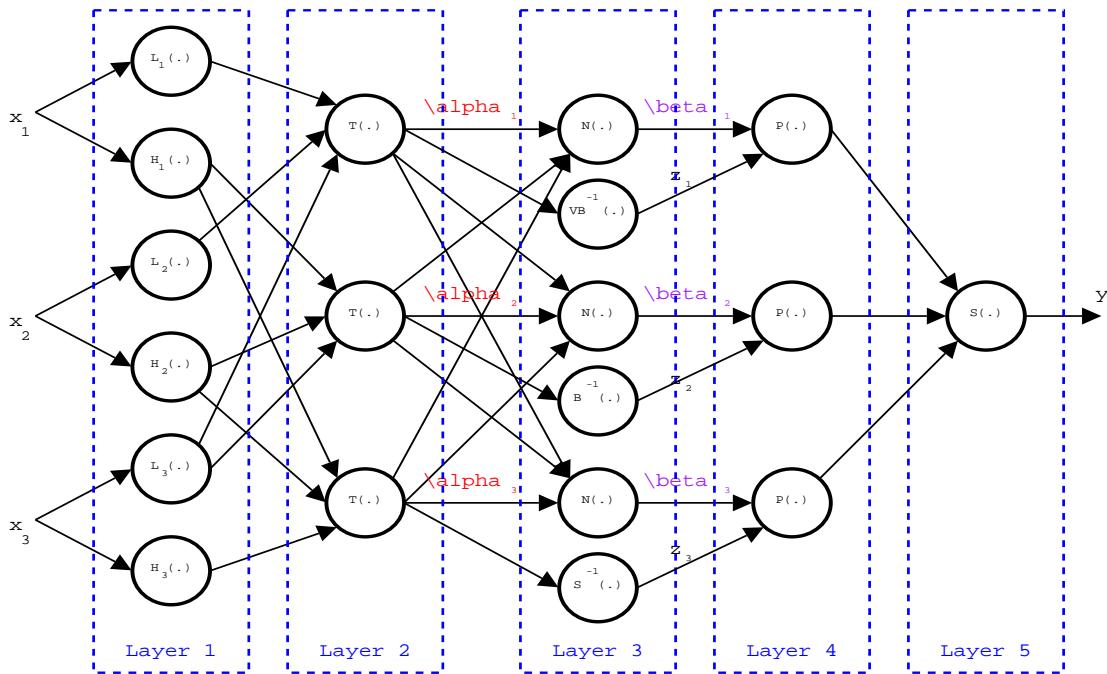
Now you know the trick! Plug you best backpropagation algorithm and learn your brand new Adaptive Neuro Fuzzy Inference System from data!

The Neuro-Fuzzy Trick Explained

Given a rule set:

$$R_j : \text{ if } x_1 \text{ is } A_{j1} \wedge x_2 \text{ is } A_{j2} \wedge \ldots \wedge x_I \text{ is } A_{jI} \text{ then } y = z_j$$

We can write a closed form to express its output as a non-linear mapping:

$$y = f(\mathbf{x}) = f(x_1, \ldots x_I)$$

If this form is <u>differentiable</u> then it is possible to minimize and error function (e.g., sum of squared errors) by gradient descend <u>as</u> with neural networks:

$$\theta^{t+1} = \theta^t - \eta \frac{\partial E(f, \mathcal{D})}{\partial \theta}$$

But only if we use differentiable t-norm, t-conorm, and inference!

Neural Networks
– *Self Organizing Maps* –

Self Organizing Features Maps: The Task

Kohonen Self Organizing Features Maps (a.k.a SOM) provide a way to represent multidimensional data in much lower dimensional spaces.

- They implement a data compression technique similar to <u>vector quantization</u>

- They store information in such a way that any topological relationships within the training set are maintained

Example: Mapping of colors from their three dimensional components (i.e., red, green and blue) into two dimensions.
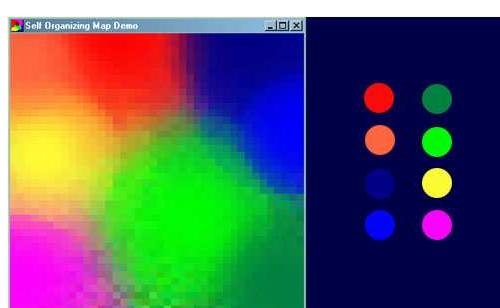
Self Organizing Features Maps: The Topology
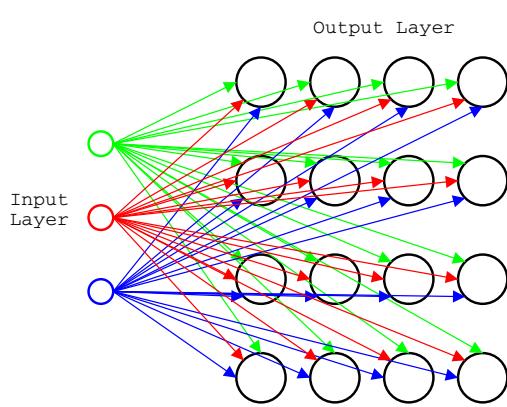
- The network is a lattice of 'nodes', each of which is fully connected to the input layer

- Each node has a specific topological position and contains a vector of weights of the same dimension as the input vectors

- There are no lateral connections between nodes within the lattice

A SOM does not need a target output to be specified; instead, where the node weights match the input vector, that area of the lattice is selectively optimized to more closely resemble the data vector

Self Organizing Features Maps: The Algorithm

Training occurs in several steps over many iterations:

1. Initialize each node's weights

2. Presented a random vector from the training set to the lattice

3. Examinate every node to calculate which one's weights are most like the input vector (the winning node is commonly known as the Best Matching Unit)

4. Calculate the radius of the neighborhood of the BMU (this is a value that starts large, typically set to the 'radius' of the lattice, but diminishes each time-step), any nodes found within this radius are deemed to be inside the BMU's neighborhood

5. Each neighboring node's weights are adjusted to make them more like the input vector. The closer a node is to the BMU, the more its weights get altered

6. Repeat step 2 for N iterations

Practical Learning of Self Organizing Features Maps

There are few things that have to be specified in the previous algorithm:

- Choosing the weights initialization

- We select the Best Matching Unit according to its the weight distance from the input vector:

$$||\mathbf{x} - \mathbf{w}_i|| = \sqrt{\sum_{k=1}^{p}(\mathbf{x}[k] - \mathbf{w}_i[k])^2}$$

- Select the neighborhood according to some decreasing function
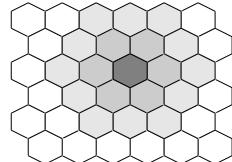
$$h_{ij} = e^{-\frac{(i-j)^2}{2\sigma^2}}$$

- Define the updating rule

$$\mathbf{w}_i(t+1) = \begin{cases} \mathbf{w}_i + \alpha(t)[\mathbf{x}(t) - \mathbf{w}_i(t)], & i \in N_i(t) \\ \mathbf{w}_i, & i \notin N_i(t) \end{cases}$$
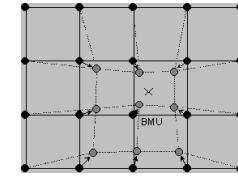
Self Organizing Feature Maps Demo

Stolen from:
http://www.ai-junkie.com

Material on Neural Networks

- <u>Machine Learning</u>, T. Mitchell, McGraw Hill, 1997
- <u>Neural Networks and Pattern Recognition</u>, C. Bishop, Oxford University Press, 1995
- <u>Reti Neuronali e Metordi Statistici</u>, a cura di Salvatore Ingrassia e Cristina Davino, Franco Angeli editore, 2002
- <u>Neural Networks FAQ</u>: ftp://ftp.sas.com/pub/neural/FAQ.html

  . . . and much much more on the net!