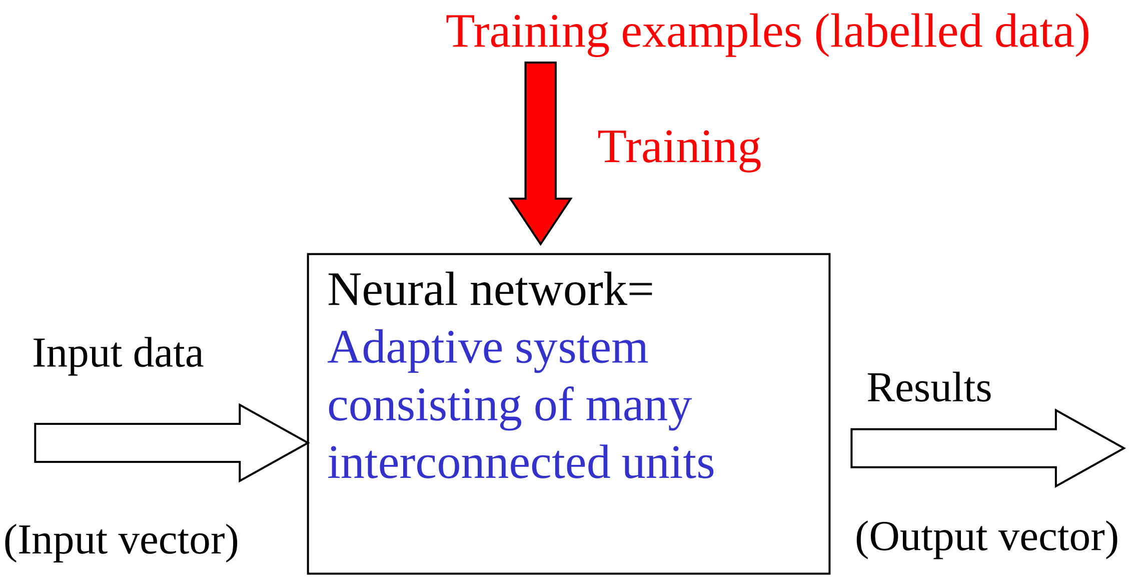
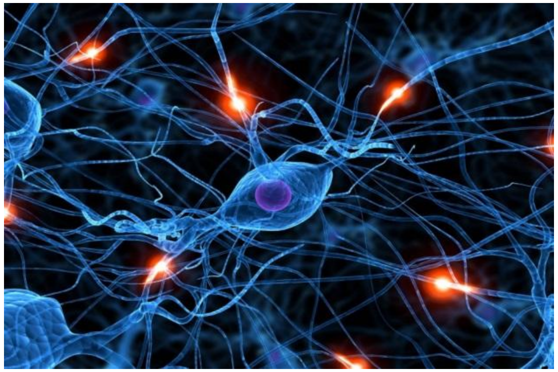
Artificial Neural Networks

- Feedforward Neural Networks
- Recurrent Neural Networks

Artificial Neural Networks

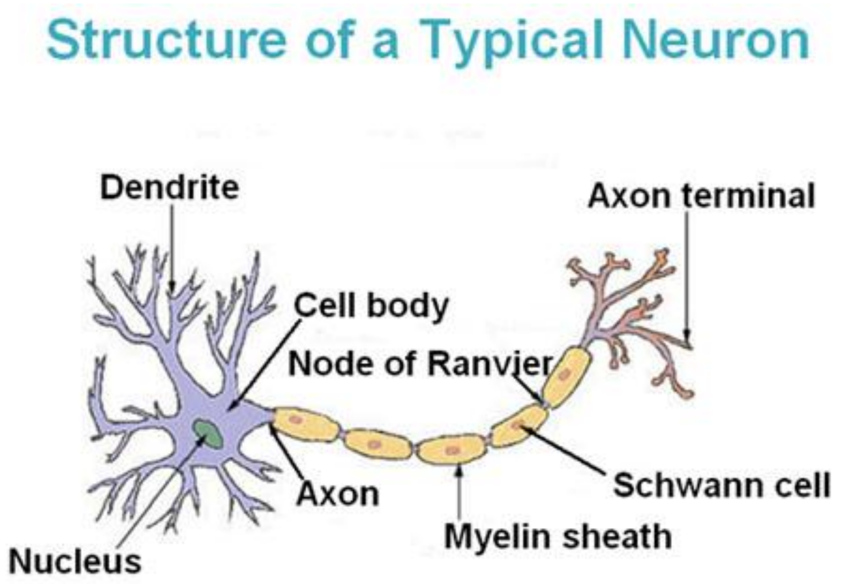
Artificial Neural Networks (ANNs) are black-box adaptive systems which extract models from data through a training process

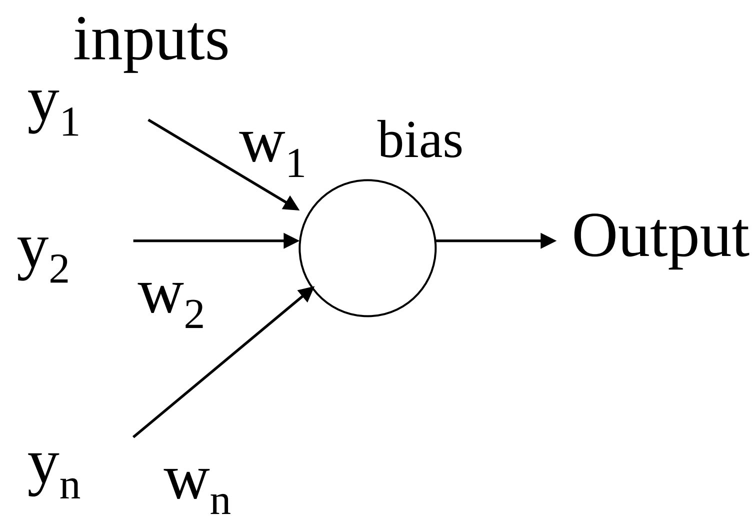




- ANNs are inspired by the brain structure and functioning
- They are very simplified models of the brain

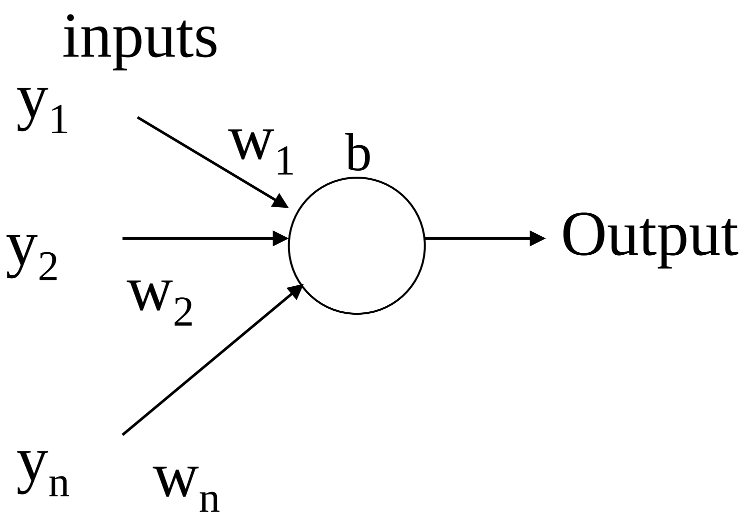
Artificial Neural Networks





w_1,w_2, ...: numerical weights associated to the connections (synaptic weights)

Artificial Neural Networks



w_1,w_2, ...: numerical weights associated to the connections (synaptic weights)

ANN = set of interconnected functional units (neurons)

Functional unit: simplified computational model of the biological neuron (several inputs, one output, an aggregation and an activation function)

Notation:

input signals: $y_1,y_2,\ldots,y_n$

synaptic weights: $w_1,w_2,\ldots,w_n$

activation threshold: b (sau w_0)

output: y

Rmk: All values are real

Artificial Neural Networks

Components of an ANN

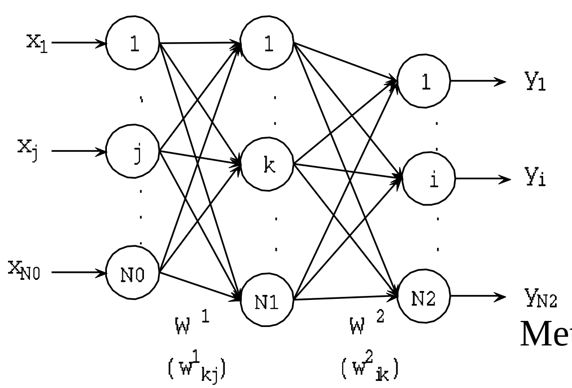
- Architecture:
  - Topology (how are placed the functional units) and connectivity (how are interconnected the functional units)
  - Defined by an oriented graph
- Functioning:
  - How the output signal is computed starting from the input signals
- Training:
  - Estimate the network parameters by using the training set

Artificial Neural Networks

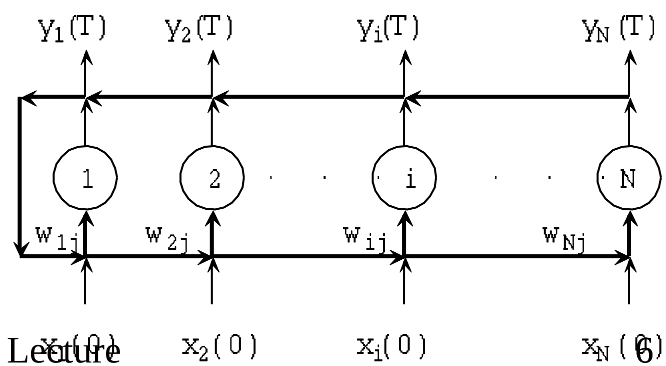
Architectures

- Feedforward
  - The graph does not contain cycles (usually the units are placed on layers)
  - The output vector can be computed directly from the input vector
- Recurrent:
  - The graph contains cycles
  - The output vector is obtained through an iterative process (simulation of a dynamical system)

Feed-forward network



Recurrent network (fully connected)



Artificial Neural Networks

Training:

- Supervised
  - The training examples contain the correct answer.
  - Aim: estimate the parameters which minimizes the error (difference between actual output and correct answers)

- Unsupervised
  - The training set contains only input data
  - Aim: estimate the parameters such that the model captures the statistical properties of the training data

Artificial Neural Networks

Applications:

- Classification/ Recognition problems
- Regression/ Prediction problems
- Clustering problems
- Association problems

Classification problems

Example 1: identifying the type of an iris flower

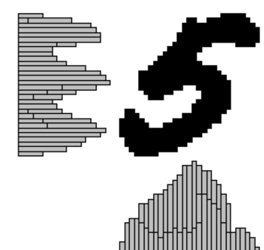
- Attributes: sepal/petal lengths, sepal/petal width
- Classes: Iris setosa, Iris versicolor, Iris virginica

Example 2: handwritten character recognition

- Attributes: various statistical and geometrical characteristics of the corresponding image
- Classes: set of characters to be recognized

⇒ Classification = find the relationship between some vectors with attribute values and classes labels

(Du Trier et al; Feature extraction methods for character Recognition. A Survey. Pattern Recognition, 1996)

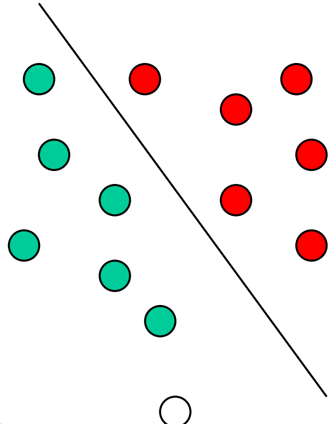


Classification problems

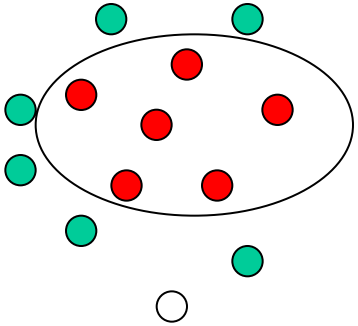
Classification:

– Problem: identify the class to which a given data (described by a set of attributes) belongs

– Prior knowledge: examples of data belonging to each class



Simple example:
linearly separable case



A more difficult example:
nonlinearly separable case

Approximation problems

- Estimation of a hous price knowing:
  - Total surface
  - Number of rooms
  - Size of the back yard
  - Location

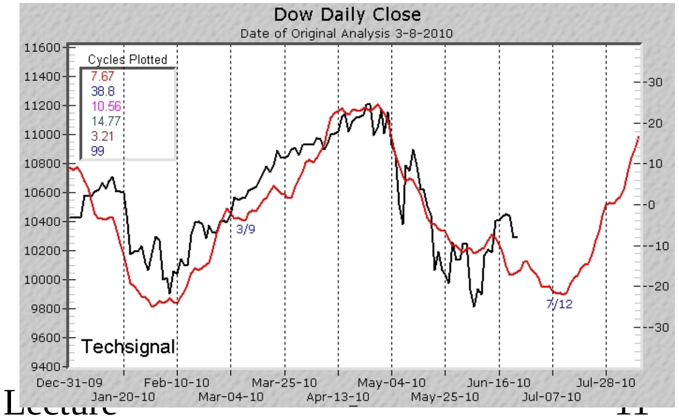
  => approximation problem = find a numerical relationship between some output and input value(s)

- Estimating the amount of resources required by a software application or the number of users of a web service or a stock price knowing historical values

  => prediction problem=

find a relationship between future values

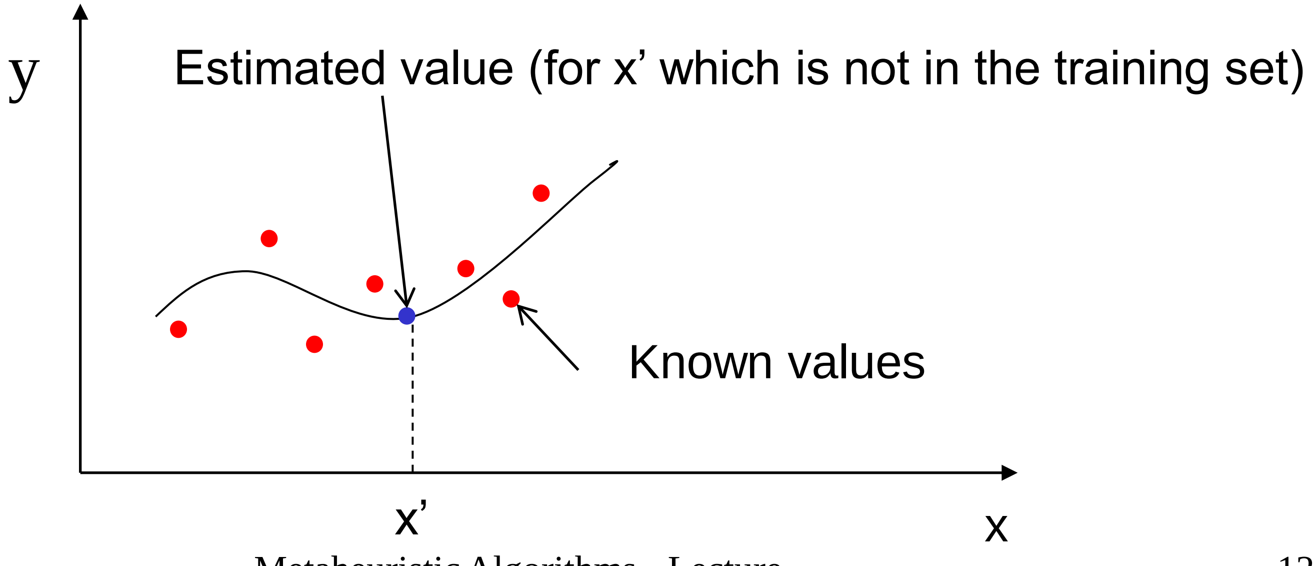
and previous values



Approximation problems

Regression (fitting, prediction):

- Problem: estimate the value of a characteristic depending on the values of some predicting characteristics
- Prior knowledge: pairs of corresponding values (training set)



Approximation problems

All approximation (mapping) problems can be stated as follows:

Starting from a set of data (X_i,Y_i), X_i in R^N and Y_i din R^M find a function $F:R^N \rightarrow R^M$ which minimizes the distance between the data and the corresponding points on its graph: $||Y_i-F(X_i)||^2$

Questions:

- What structure (shape) should have F ?
- How can we find the parameters defining the properties of F ?

Approximation problems

Can be such a problem be solved by using neural networks ?

Yes, at least in theory, the neural networks are proven "universal approximators" [Hornik, 1985]:

" Any continuous function can be approximated by a feedforward neural network having at least one hidden layer. The accuracy of the approximation depends on the number of hidden units."

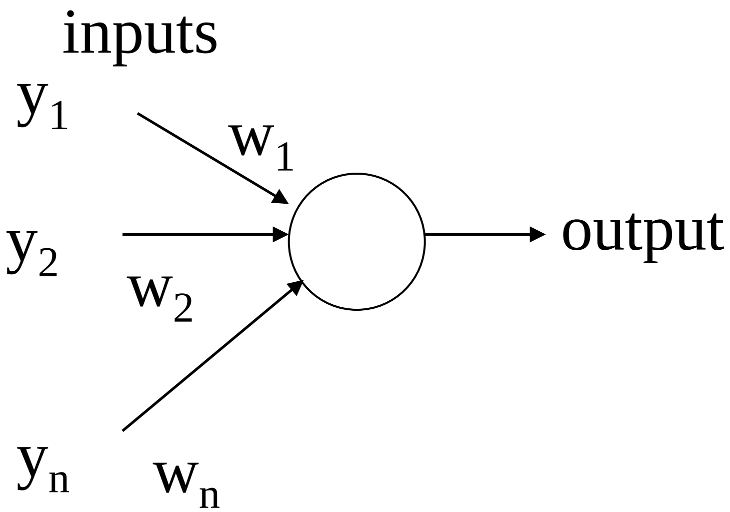
- The shape of the function is influenced by the architecture of the network and by the properties of the activation functions.
- The function parameters are in fact the weights corresponding to the connections between neurons.

Neural Networks Design

Steps to follow in designing a neural network:

- Choose the architecture: number of layers, number of units on each layer, activation functions, interconnection style
- Train the network: compute the values of the weights using the training set and a learning algorithm.
- Validate/test the network: analyze the network behavior for data which do not belong to the training set.

Functional units (neurons)



Weights assigned to the connections

Functional unit: several inputs, one output

Notations:

- input signals: y1,y2,…,yn
- synaptic weights: w1,w2,…,wn
  (they model the synaptic permeability)
- threshold (bias): b (or theta)

(it models the activation threshold of the neuron)

- Output: y

- All these values are usually real numbers

Functional units (neurons)

Output signal generation:

- The input signals are "combined" by using the connection weights and the threshold
  - The obtained value corresponds to the local potential of the neuron
  - This "combination" is obtained by applying a so-called aggregation function
- The output signal is constructed by applying an activation function
  - It corresponds to the pulse signals propagated along the axon

Input signals ($y_1,\ldots,y_n$) $\rightarrow$ Neuron's state (u) $\rightarrow$ Output signal (y)

Aggregation function | Activation function

Functional units (neurons)

Aggregation functions:

Weighted sum

$$u=\sum_{j=1}^{n} w_j y_j - w_0$$

Euclidean distance

$$u=\sqrt{\sum_{j=1}^{n}(w_j - y_j)^2}$$

$$u=\prod_{j=1}^{n} y_j^{w_j}$$

Multiplicative neuron

$$u=\sum_{j=1}^{n} w_j y_j + \sum_{i,j=1}^{n} w_{ij} y_i y_j + \ldots$$

High order connections

Remark: in the case of the weighted sum the threshold can be interpreted as a synaptic weight which corresponds to a virtual unit which always produces the value -1

$$u=\sum_{j=0}^{n} w_j y_j$$

Functional units (neurons)

Activation functions:

$$f(u) = \mathrm{sgn}(u) = \begin{cases} -1 & u \le 0 \\ 1 & u > 0 \end{cases}$$

signum

$$f(u) = H(u) = \begin{cases} 0 & u \le 0 \\ 1 & u > 0 \end{cases}$$

Heaviside

$$f(u) = \begin{cases} -1 & u < -1 \\ u & -1 \le u \le 1 \\ 1 & u > 1 \end{cases}$$

Saturated linear

$$f(u) = u$$

linear

$$f(u) = \max\{0, u\}$$

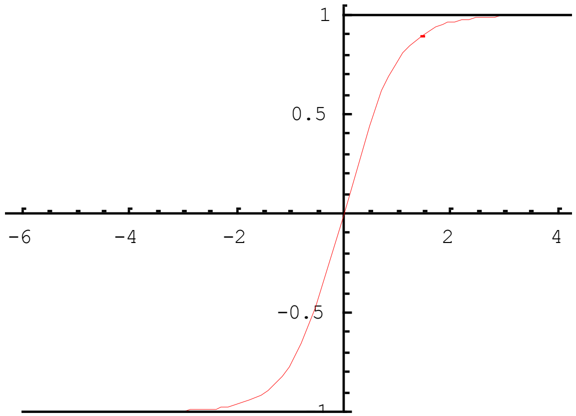
Rectified linear – used in deep networks

Functional units (neurons)

Sigmoidal aggregation functions

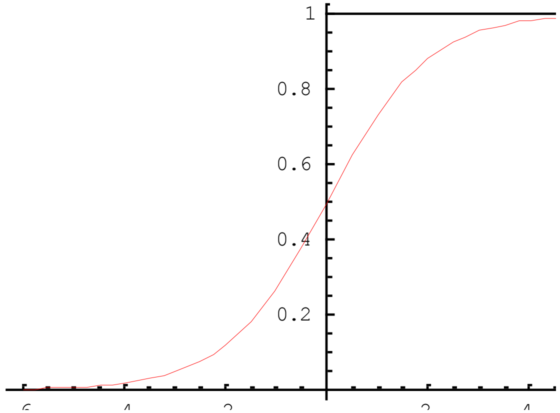
(Hyperbolic tangent)

$$f(u) = \tanh(u) = \frac{\exp(2u) - 1}{\exp(2u) + 1}$$



$$f(u) = \frac{1}{1 + \exp(-u)}$$

(Logistic)

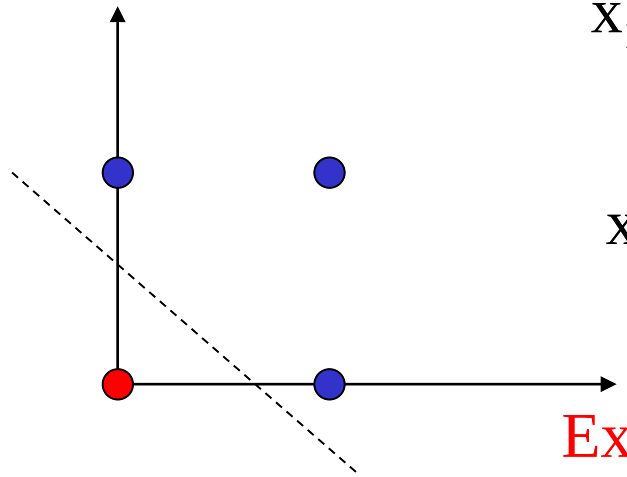


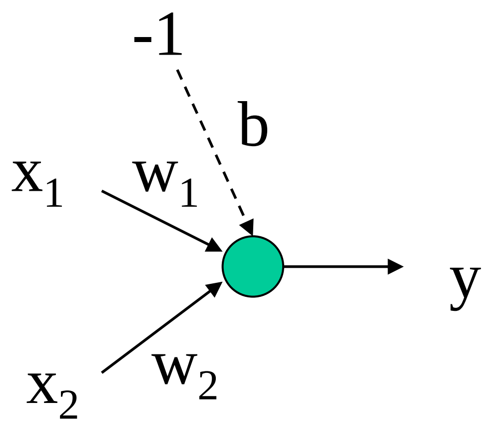
Functional units (neurons)

- What can do a single neuron ?
- It can solve simple problems (linearly separable problems)

OR

	0	1
0	0	1
1	1	1



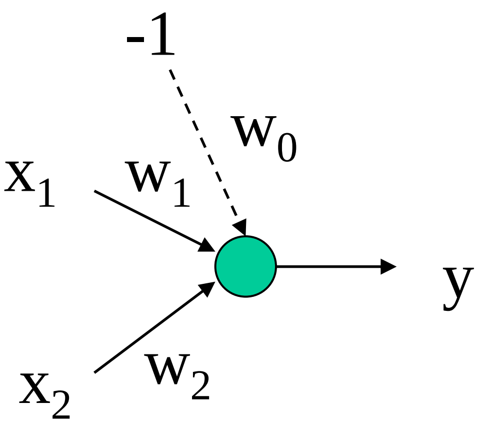


$$y=H(w_1x_1+w_2x_2-b)$$

Ex: $w_1=w_2=1,\ w_0=0.5$

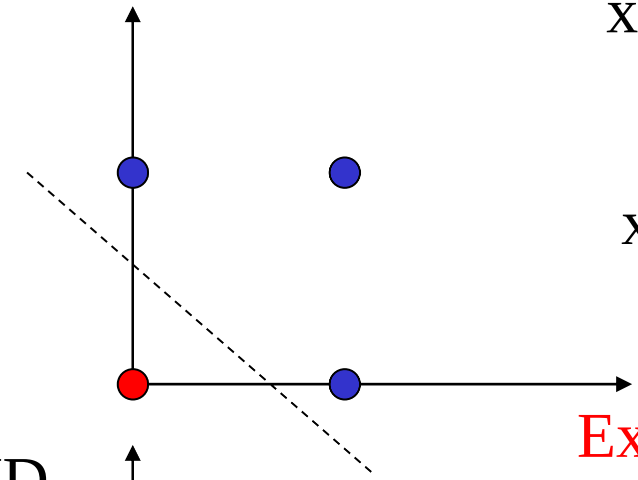
Functional units (neurons)

- What can do a single neuron ?
- It can solve simple problems (linearly separable problems)



OR

	0	1
0	0	1
1	1	1

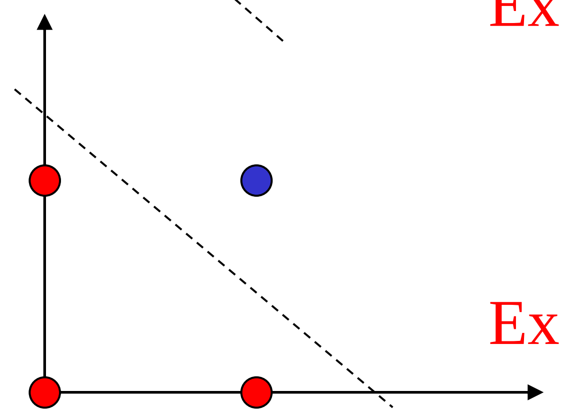


$$y=H(w_1x_1+w_2x_2-w_0)$$

Ex: $w_1=w_2=1,\ w_0=0.5$

AND

	0	1
0	0	0
1	0	1

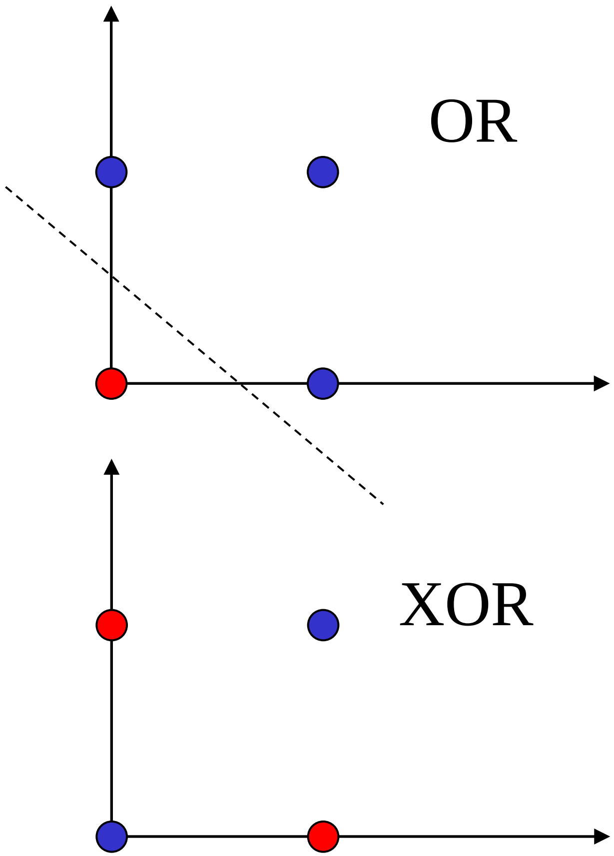


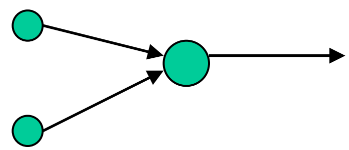
$$y=H(w_1x_1+w_2x_2-w_0)$$

Ex: $w_1=w_2=1,\ w_0=1.5$

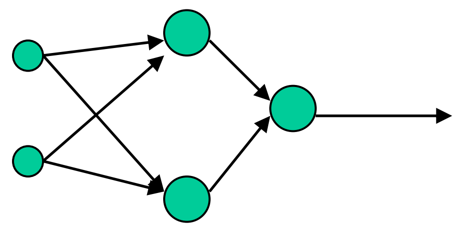
Functional units (neurons)

Representation of boolean functions: f:{0,1}^{2}->{0,1}





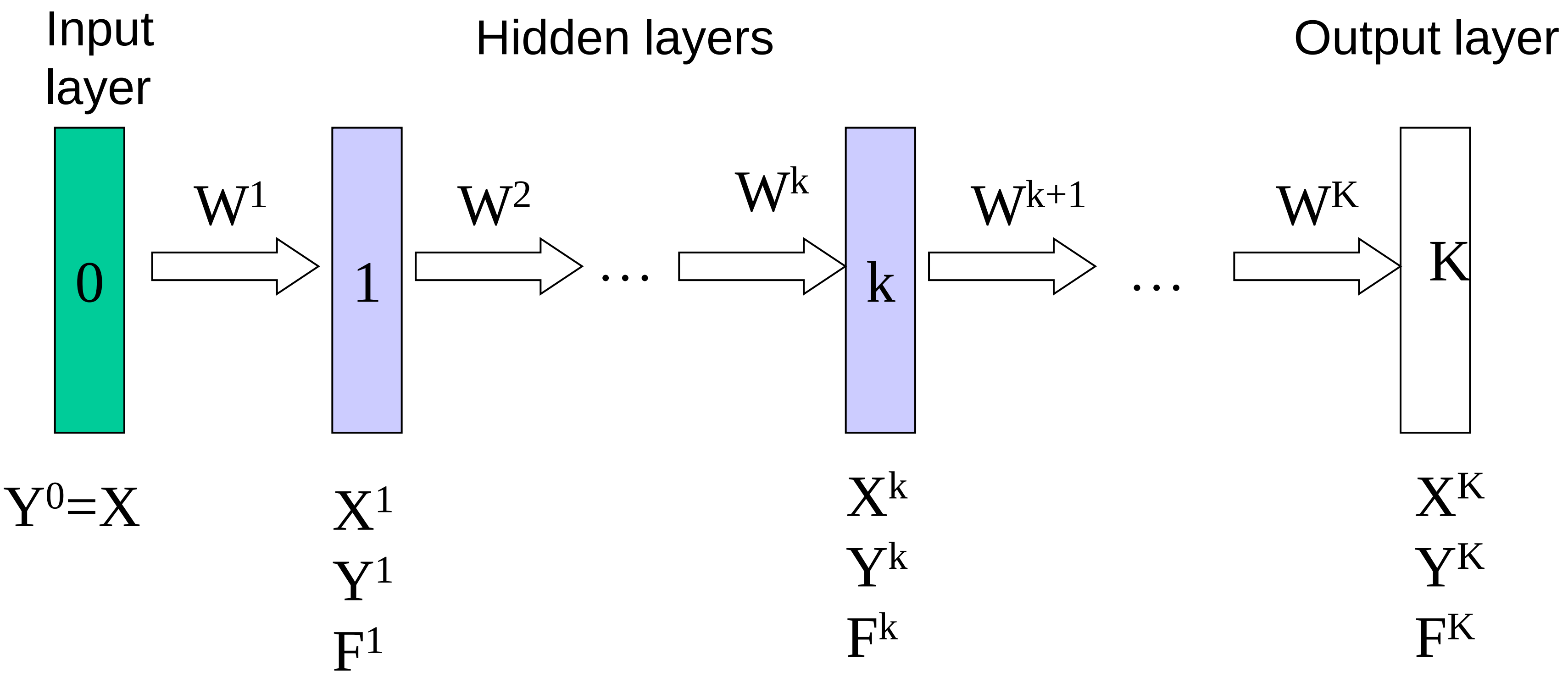
Linearly separable problem: one layer network



Nonlinearly separable problem: multilayer network

Architecture and notations

Feedforward network with K layers



X = input vector, Y= output vector, F=vectorial activation function

Functioning

Computation of the output vector

$$Y^K = F^K(W^K F^{K-1}(W^{K-1} \ldots F^1(W^1 X)))$$

$$Y^k = F^k(X^k) = F(W^k Y^{k-1})$$

FORWARD Algorithm (propagation of the input signal toward the output layer)

```
Y[0]:=X (X is the input signal)
FOR k:=1,K DO
    X[k]:=W[k]Y[k-1]
    Y[k]:=F(X[k])
ENDFOR
```

Rmk: Y[K] is the output of the network

A particular case

One hidden layer

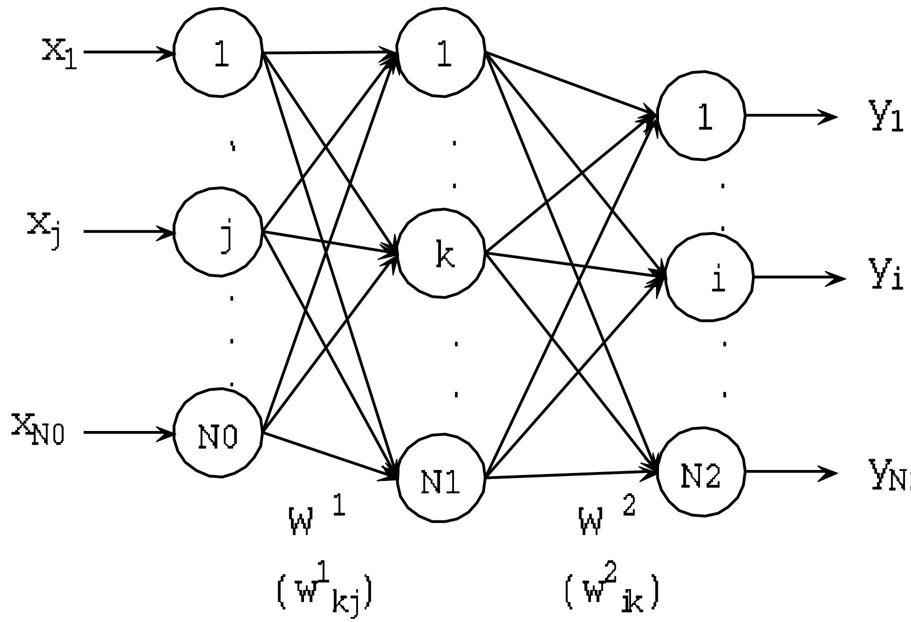
Adaptive parameters: W1, W2

$$y_i = f_2\left(\sum_{k=0}^{N1} w^{(2)}{}_{ik}\, f_1\left(\sum_{j=0}^{N0} w^{(1)}{}_{kj}\, x_j\right)\right)$$

A simpler notation:

$$w^{(2)}{}_{ik} = w_{ik};$$

$$w^{(1)}{}_{kj} = w_{kj}$$



Remark:

Traditionally only 1 or 2 hidden layers are used

Lately, architectures involving many hidden layers became more popular (Deep Neural Networks) – the are used mainly for image and language processing (http://deeplearning.net)

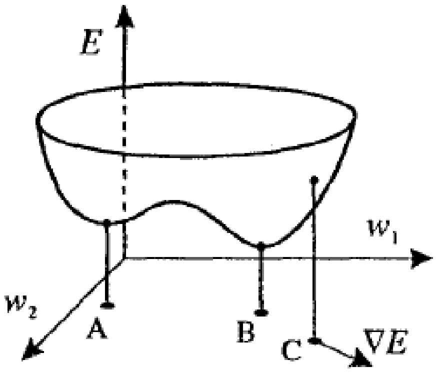
Learning process

Learning based on minimizing a error function

- Training set: $\{(x^1,d^1), \ldots, (x^L,d^L)\}$
- Error function (mean squared error):

$$E(W) = \frac{1}{2L}\sum_{l=1}^{L}\sum_{i=1}^{N2}\left(d_i^l - f_2\left(\sum_{k=0}^{N1} w_{ik} f_1\left(\sum_{j=0}^{N0} w_{kj} x_j\right)\right)\right)^2$$

- Aim of learning process: find W which minimizes the error function
- Minimization method: gradient method



Learning process

Gradient based adjustement

$$w_{ij}(t+1) = w_{ij}(t) - \eta \frac{\partial E(w(t))}{\partial w_{ij}}$$

Learning rate

$$E(W) = \frac{1}{2L} \sum_{l=1}^{L} \underbrace{\sum_{i=1}^{N2} \left(d_i^l - \underbrace{f_2\left(\underbrace{\sum_{k=0}^{N1} w_{ik} \underbrace{f_1\left(\underbrace{\sum_{j=0}^{N0} w_{kj} x_j}_{x_k} \right)}_{y_k}}_{x_i} \right)}_{y_i} \right)^2}_{E_l(W)}$$

Learning process

- Partial derivatives computation

$$E(W)=\frac{1}{2L}\sum_{l=1}^{L}\sum_{i=1}^{N2}\left(d_i^l-\underbrace{f_2\left(\underbrace{\sum_{k=0}^{N1}w_{ik}\,\underbrace{f_1\left(\underbrace{\sum_{j=0}^{N0}w_{kj}x_j}_{x_k}\right)}_{y_k}}_{x_i}\right)}_{y_i}\right)^2$$

$$\frac{\partial E_l(W)}{\partial w_{ik}}=-(d_i^l-y_i)\,f_2^{'}(x_i)\,y_k=-\delta_i^l\,y_k$$

$$\frac{\partial E_l(W)}{\partial w_{kj}}=-\sum_{i=1}^{N2}w_{ik}(d_i^l-y_i)\,f_2^{'}(x_i)\,f_1^{'}(x_k)\,x_j=-\left(f_1^{'}(x_k)\sum_{i=1}^{N2}w_{ik}\delta_i^l\right)x_j=-\delta_k^l\,x_j$$

$$E_l(W)=\frac{1}{2}\sum_{i=1}^{N2}\left(d_i^l-f_2\left(\sum_{k=0}^{N1}w_{ik}\,f_1\left(\sum_{j=0}^{N0}w_{kj}x_j\right)\right)\right)^2$$

Learning process

- Partial derivatives computation

$$\frac{\partial E_l(W)}{\partial w_{ik}} = -(d_i^l - y_i)\, f_2^{'}(x_i)\, y_k = -\delta_i^l y_k$$

$$\frac{\partial E_l(W)}{\partial w_{kj}} = -\sum_{i=1}^{N2} w_{ik}(d_i^l - y_i)\, f_2^{'}(x_i)\, f_1^{'}(x_k)\, x_j = -\left(f_1^{'}(x_k) \sum_{i=1}^{N2} w_{ik} \delta_i^l \right) x_j = -\delta_k^l x_j$$

$$E_l(W) = \frac{1}{2} \sum_{i=1}^{N2} \left(d_i^l - f_2\left(\sum_{k=0}^{N1} w_{ik}\, f_1\left(\sum_{j=0}^{N0} w_{kj} x_j \right)\right)\right)^2$$

Remark:

The derivatives of sigmoidal activation functions have particular properties:

Logistic: f'(x)=f(x)(1-f(x))=y(1-y)

Tanh: f'(x)=1-f^2(x)=1-y^2

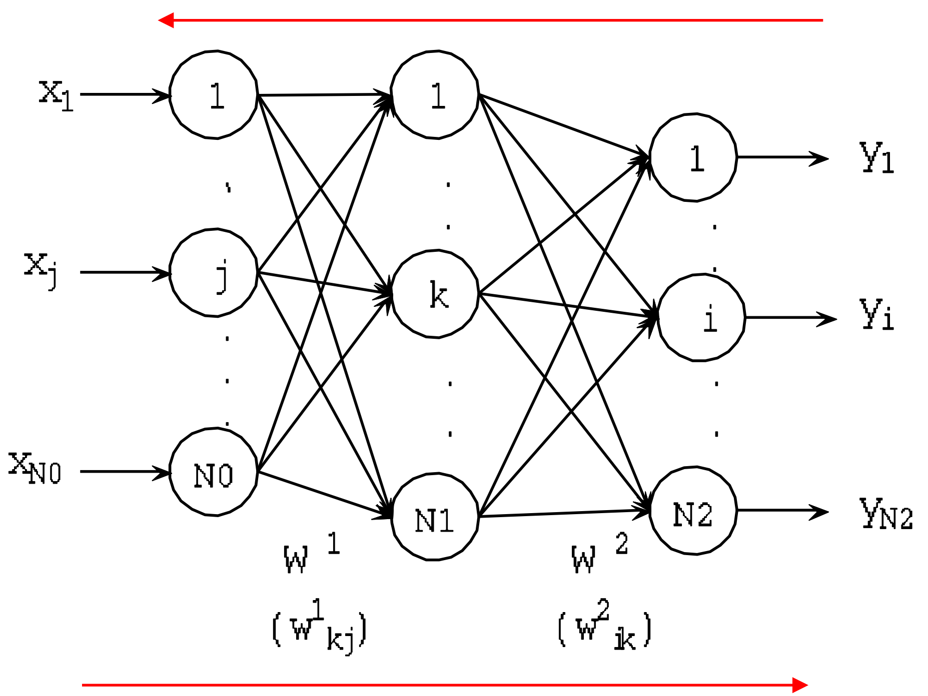
The BackPropagation Algorithm

Main idea:

For each example in the training set:

- compute the output signal
- compute the error corresponding to the output level
- propagate the error back into the network and store the corresponding delta values for each layer
- adjust each weight by using the error signal and input signal for each layer

Computation of the error signal (BACKWARD)



Computation of the output signal (FORWARD)

The BackPropagation Algorithm

General structure

Random initialization of weights

REPEAT

FOR l=1,L DO

FORWARD stage

BACKWARD stage

weights adjustement

ENDFOR

Error (re)computation

UNTIL <stopping condition>

epoch

Rmk.

- The weights adjustment depends on the learning rate
- The error computation needs the recomputation of the output signal for the new values of the weights
- The stopping condition depends on the value of the error and on the number of epochs
- This is a so-called serial (incremental) variant: the adjustment is applied separately for each example from the training set

The BackPropagation Algorithm

Details (serial variant)

$$w_{kj} := rand(-1,1), w_{ik} := rand(-1,1)$$

$$p := 0$$

REPEAT

FOR $l := 1, L$ DO

/* FORWARD Step */

$$x_k^l := \sum_{j=0}^{N0} w_{kj} x_j^l,\; y_k^l := f_1(x_k^l),\; x_i^l := \sum_{k=0}^{N1} w_{ik} y_k^l,\; y_i^l := f_2(x_i^l)$$

/* BACKWARD Step */

$$\delta_i^l := f_2^{'}(x_i^l)(d_i^l - y_i^l),\; \delta_k^l := f_1^{'}(x_k^l)\sum_{i=1}^{N2} w_{ik}\delta_i^l$$

/* Adjustement Step */

$$w_{kj} := w_{kj} + \eta\delta_k^l x_j^l,\;\; w_{ik} := w_{ik} + \eta\delta_i^l y_k^l$$

ENDFOR

The BackPropagation Algorithm

Details (serial variant)

/* Error computation */

$E := 0$

FOR $l := 1, L$ DO

/* FORWARD Step */

$$x_k^l := \sum_{j=0}^{N0} w_{kj} x_j^l,\ y_k^l := f_1(x_k^l),\ x_i^l := \sum_{k=0}^{N1} w_{ik} y_k^l,\ y_i^l := f_2(x_i^l)$$

/* Error summation */

$$E := E + \sum_{l=1}^{L} (d_i^l - y_i^l)^2$$

ENDFOR

$E := E/(2L)$

$p := p + 1$

UNTIL $p > p_{\max}$ OR $E < E^*$

E* denotes the expected training accuracy

p_{max} denots the maximal number of epochs

The BackPropagation Algorithm

Batch variant

Random initialization of weights

REPEAT

epoch

initialize the variables which will contain the adjustments

FOR l=1,L DO

FORWARD stage

BACKWARD stage

cumulate the adjustments

ENDFOR

Apply the cumulated adjustments

Error (re)computation

UNTIL <stopping condition>

Rmk.

- The incremental variant can be sensitive to the presentation order of the training examples
- The batch variant is not sensitive to this order and is more robust to the errors in the training examples
- It is the starting algorithm for more elaborated variants, e.g. momentum variant

The BackPropagation Algorithm

Details (batch variant) $w_{kj} := rand(-1,1), w_{ik} := rand(-1,1),\ i = 1..N2, k = 0..N1,\ j = 0..N0$

$p := 0$

REPEAT

$\boxed{\Delta^1_{kj} := 0, \Delta^2_{ik} := 0}$

FOR $l := 1, L$ DO

/* FORWARD step */

$$x_k^l := \sum_{j=0}^{N0} w_{kj} x_j^l,\ y_k^l := f_1(x_k^l),\ x_i^l := \sum_{k=0}^{N1} w_{ik} y_k^l,\ y_i^l := f_2(x_i^l)$$

/* BACKWARD step */

$$\delta_i^l := f_2'(x_i^l)(d_i^l - y_i^l),\ \delta_k^l := f_1'(x_k^l) \sum_{i=1}^{N2} w_{ik} \delta_i^l$$

/* Adjustment step */

$$\boxed{\Delta^1_{kj} := \Delta^1_{kj} + \eta \delta_k^l x_j^l,\ \ \Delta^2_{ik} := \Delta^2_{ik} + \eta \delta_i^l y_k^l}$$

ENDFOR

$$\boxed{w_{kj} := w_{kj} + \Delta^1_{kj},\ \ w_{ik} := w_{ik} + \Delta^2_{ik}}$$

The BackPropagation Algorithm

/* Error computation */

$E := 0$

FOR $l := 1, L$ DO

/* FORWARD Step */

$$x_k^l := \sum_{j=0}^{N0} w_{kj} x_j^l,\ y_k^l := f_1(x_k^l),\ x_i^l := \sum_{k=0}^{N1} w_{ik} y_k^l,\ y_i^l := f_2(x_i^l)$$

/* Error summation */

$$E := E + \sum_{l=1}^{L} (d_i^l - y_i^l)^2$$

ENDFOR

$E := E/(2L)$

$p := p + 1$

UNTIL $p > p_{\max}$ OR $E < E^*$

Variants

Different variants of BackPropagation can be designed by changing:

- Error function
- Minimization method
- Learning rate choice
- Weights initialization

Variants

Error function:

- MSE (mean squared error function) is appropriate in the case of approximation problems
- For classification problems a better error function is the cross-entropy error:
- Particular case: two classes (one output neuron):
  - d_l is from {0,1} (0 corresponds to class 0 and 1 corresponds to class 1)
  - y_l is from (0,1) and can be interpreted as the probability of class 1

$$CE(W) = -\sum_{l=1}^{L} (d_l \ln y_l + (1-d_l)\ln(1-y_l))$$

Rmk: the partial derivatives change, thus the adjustment terms will be different

Variants

Entropy based error:

- Different values of the partial derivatives
- In the case of logistic activation functions the error signal will be:

$$\delta_l = \left(\frac{d_l}{y_l} - \frac{1-d_l}{1-y_l}\right) f_2^{'}(x^{(2)}) = \frac{d_l(1-y_l) - y_l(1-d_l)}{y_l(1-y_l)} \cdot y_l(1-y_l)$$

$$= d_l(1-y_l) - y_l(1-d_l)$$

Variants

Minimization method:

- The gradient method is a simple but not very efficient method

- More sophisticated and faster methods can be used instead:
  - Conjugate gradient methods
  - Newton’s method and its variants

- Particularities of these methods:
  - Faster convergence (e.g. the conjugate gradient converges in n steps for a quadratic error function)
  - Needs the computation of the hessian matrix (matrix with second order derivatives) : second order methods

Variants

Example: Newton's method

$E : R^n \to R,\ w \in R^n$ is the vector of all weights

By Taylor's expansion in $w(p)$ (estimation corresponding to epoch p)

$$E(w) \cong E(w(p)) + (\nabla E(w(p)))^T (w - w(p)) + \frac{1}{2}(w - w(p))^T H(w(p))(w - w(p))$$

$$H(w(p))_{ij} = \frac{\partial E(w(p))}{\partial w_i \partial w_j}$$

By derivating the Taylor's expansion with respect to w the minimum will be the solution of :

$$H(w(p))w - H(w(p))w(p) + \nabla E(w(p)) = 0$$

Thus the new estimation of w is :

$$w(p+1) = w(p) - H^{-1}(w(p)) \cdot \nabla E(w(p))$$

Variants

Particular case: Levenberg-Marquardt

- This is the Newton method adapted for the case when the objective function is a sum of squares (as MSE is)

$$E(w) = \sum_{l=1}^{L} E_l(w), \;\; e(w) = (E_1(w), \ldots, E_L(w))^T$$

$$w(p+1) = w(p) - (J^T(w(p)) \cdot J(w(p)) + \mu_p I)^{-1} J^T(w(p)) e(w(p))$$

$$J(w) = \text{ jacobian of } e(w)$$

$$J_{ij}(w) = \frac{\partial E_i(w)}{\partial w_j}$$

Used in order to deal with singular matrices

Advantage:

- Does not need the computation of the hessian

Problems in BackPropagation

- Low convergence rate (the error decreases too slow)
- Oscillations (the error value oscillates instead of continuously decreasing)
- Local minima problem (the learning process is stuck in a local minima of the error function)
- Stagnation (the learning process stagnates even if it is not a local minima)
- Overtraining and limited generalization

Problems in BackPropagation

Problem 1: The error decreases too slow or the error value oscillates instead of continuously decreasing

Causes:

- Inappropriate value of the learning rate (too small values lead to slow convergence while too large values lead to oscillations)
  – Solution: adaptive learning rate

- Slow minimization method (the gradient method needs small learning rates in order to converge)

  Solutions:

  - heuristic modification of the standard BP (e.g. momentum)
  - other minimization methods (Newton, conjugate gradient)

Problems in BackPropagation

Adaptive learning rate:

- If the error is increasing then the learning rate should be decreased
- If the error significantly decreases then the learning rate can be increased
- In all other situations the learning rate is kept unchanged

$$E(p) > (1+\gamma)E(p-1) \Rightarrow \eta(p) = a\eta(p-1),\ 0 < a < 1$$
$$E(p) < (1-\gamma)E(p-1) \Rightarrow \eta(p) = b\eta(p-1),\ 1 < b < 2$$
$$(1-\gamma)E(p-1) \leq E(p) \leq (1+\gamma)E(p-1) \Rightarrow \eta(p) = \eta(p-1)$$

Example: $\gamma=0.05$

Problems in BackPropagation

Momentum variant:

- Increase the convergence speed by introducing some kind of "inertia" in the weights adjustment: the weight changes corresponding to the current epoch includes the adjustments from the previous epoch

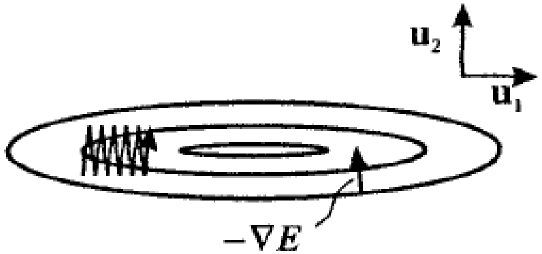
$$\Delta w_{ij}(p+1) = \eta(1-\alpha)\delta_i y_j + \alpha \Delta w_{ij}(p)$$

Momentum coefficient: α in [0.1,0.9]

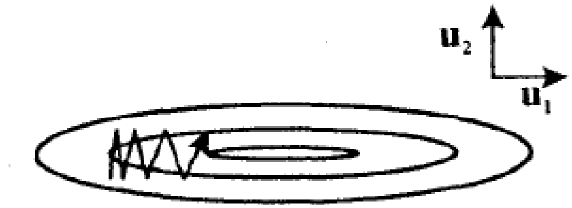
Problems in BackPropagation

Momentum variant:

- The effect of these enhancements is that flat spots of the error surface are traversed relatively rapidly with a few big steps, while the step size is decreased as the surface gets rougher. This implicit adaptation of the step size increases the learning speed significantly.



Simple gradient descent



Use of inertia term

Problems in BackPropagation

Problem 2: Local minima problem (the learning process is stuck in a local minima of the error function)

Cause: the gradient based methods are local optimization methods

Solutions:

- Restart the training process using other randomly initialized weights
- Introduce random perturbations into the values of weights:

$$w_{ij} := w_{ij} + \xi_{ij}, \quad \xi_{ij} = \text{random variables}$$

- Use a global optimization method

Problems in BackPropagation

Solution:

- Replacing the gradient method with a stochastic optimization method
- This means using a random perturbation instead of an adjustment based on the gradient computation
- Adjustment step:

$$\Delta_{ij} = \text{random values}$$

$$\text{IF } E(W+\Delta) < E(W) \text{ THEN accept the adjustment } (\mathrm{W} := \mathrm{W}+\Delta)$$

Rmk:

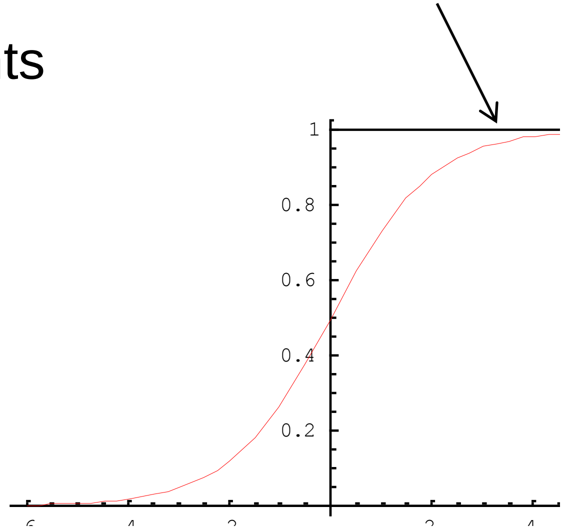
- The adjustments are usually based on normally distributed random variables
- If the adjustment does not lead to a decrease of the error then it is not accepted

Problems in BackPropagation

Problem 3: Stagnation (the learning process stagnates even if it is not a local minima)

Cause: the adjustments are too small because the arguments of the sigmoidal functions are too large

Very small derivates



Solutions:

- Penalize the large values of the weights (weights-decay)

- Use only the signs of derivatives not tl values

Problems in BackPropagation

Penalization of large values of the weights: add a regularization term to the error function

$$E_{(r)}(W) = E(W) + \lambda \sum_{i,j} w_{ij}^2$$

The adjustment will be:

$$\Delta_{ij}^{(r)} = \Delta_{ij} - 2\lambda w_{ij}$$

Problems in BackPropagation

Resilient BackPropagation (use only the sign of the derivative not its value)

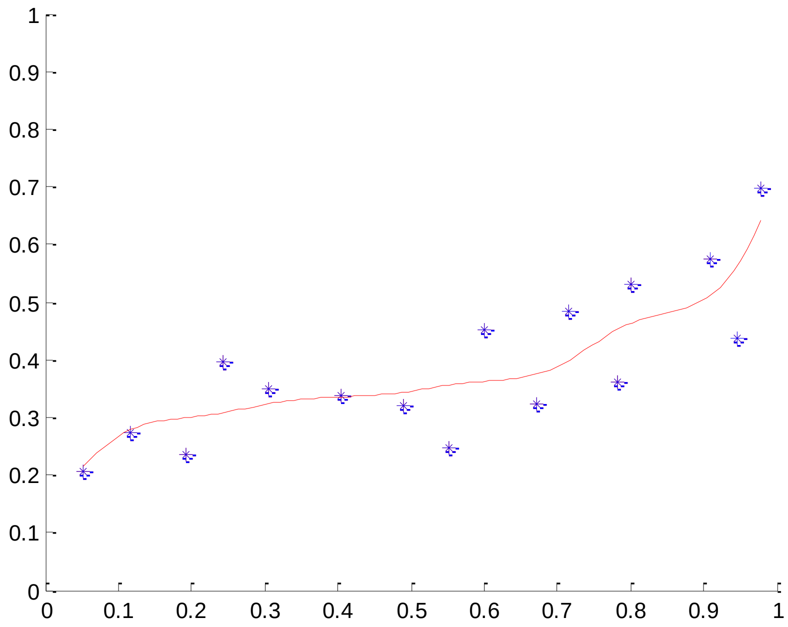
$$\Delta w_{ij}(p) = \begin{cases} -\Delta_{ij}(p) & \text{if } \dfrac{\partial E(W(p-1))}{\partial w_{ij}} > 0 \\ \Delta_{ij}(p) & \text{if } \dfrac{\partial E(W(p-1))}{\partial w_{ij}} < 0 \end{cases}$$

$$\Delta_{ij}(p) = \begin{cases} a\Delta_{ij}(p-1) & \text{if } \dfrac{\partial E(W(p-1))}{\partial w_{ij}} \cdot \dfrac{\partial E(W(p-2))}{\partial w_{ij}} > 0 \\ b\Delta_{ij}(p-1) & \text{if } \dfrac{\partial E(W(p-1))}{\partial w_{ij}} \cdot \dfrac{\partial E(W(p-2))}{\partial w_{ij}} < 0 \end{cases}$$

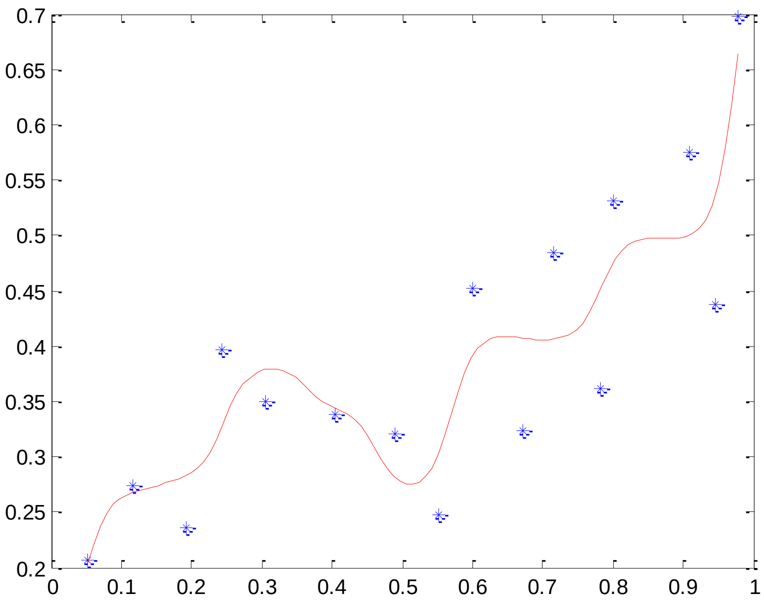
$$0 < b < 1 < a$$

Problems in BackPropagation

Problem 4: Overtraining and limited generalization ability



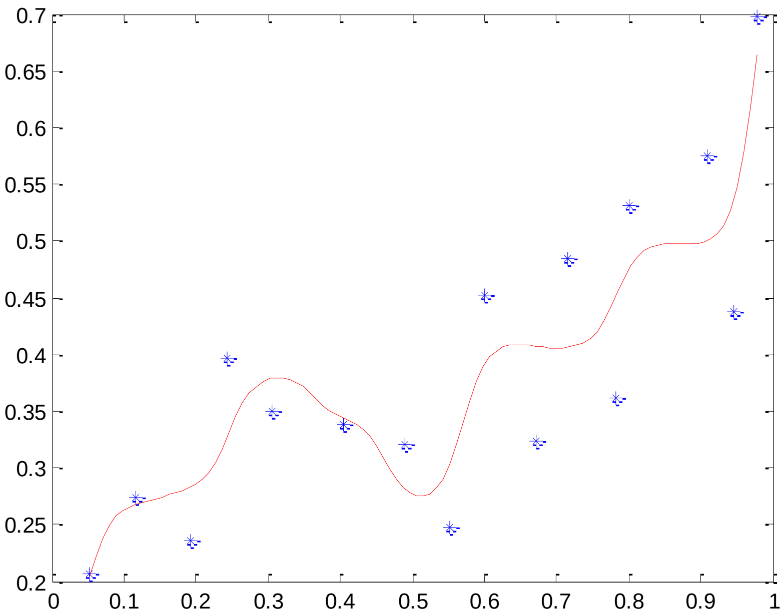
5 hidden units



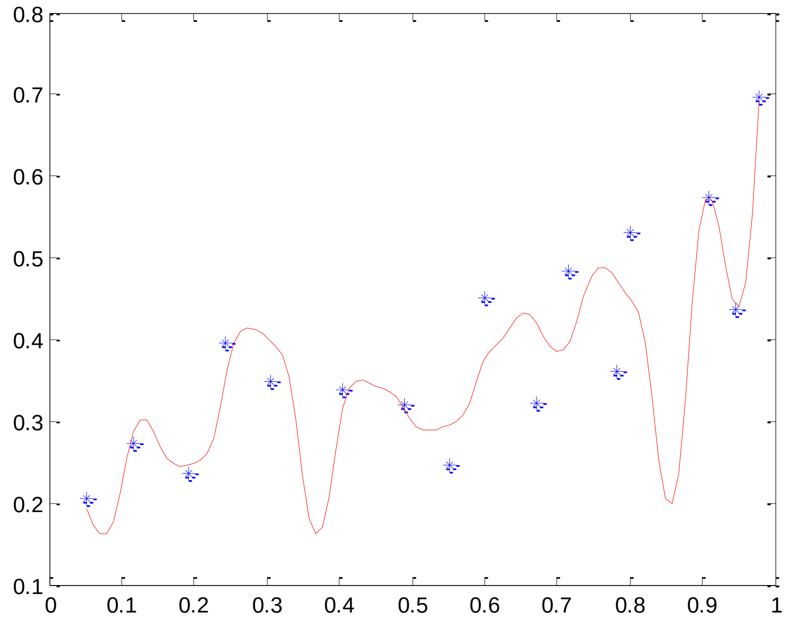
10 hidden units

Problems in BackPropagation

Problem 4: Overtraining and limited generalization ability



10 hidden units



20 hidden units

Problems in BackPropagation

Problem 4: Overtraining and limited generalization ability

Causes:

- Network architecture (e.g. number of hidden units)
  - A large number of hidden units can lead to overtraining (the network extracts not only the useful knowledge but also the noise in data)
- The size of the training set
  - Too few examples are not enough to train the network
- The number of epochs (accuracy on the training set)
  - Too many epochs could lead to overtraining

Solutions:

- Dynamic adaptation of the architecture
- Stopping criterion based on validation error; cross-validation

Problems in BackPropagation

Dynamic adaptation of the architectures:

- Incremental strategy:
  - Start with a small number of hidden neurons
  - If the learning does not progress new neurons are introduced

- Decremental strategy:
  - Start with a large number of hidden neurons
  - If there are neurons with small weights (small contribution to the output signal) they can be eliminated

Problems in BackPropagation

Stopping criterion based on validation error :

- Divide the learning set in m parts: (m-1) are for training and another one for validation
- Repeat the weights adjustment as long as the error on the validation subset is decreasing (the learning is stopped when the error on the validation subset start increasing)

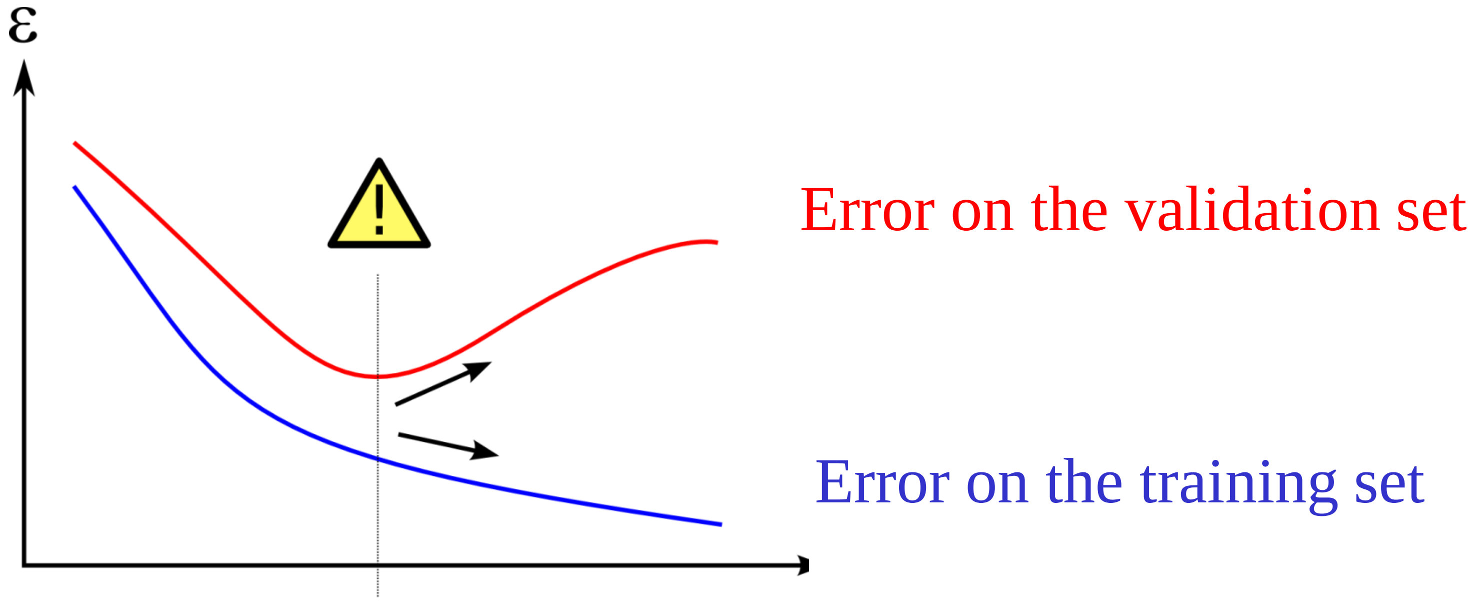
Cross-validation:

- Applies for m times the learning algorithm by successively changing the learning and validation sets

1: S=(S1,S2, ...,Sm)
2: S=(S1,S2, ...,Sm)
....
m: S=(S1,S2, ...,Sm)

Problems in BackPropagation

Stop the learning process when the error on the validation set start to increase (even if the error on the training set is still decreasing) :



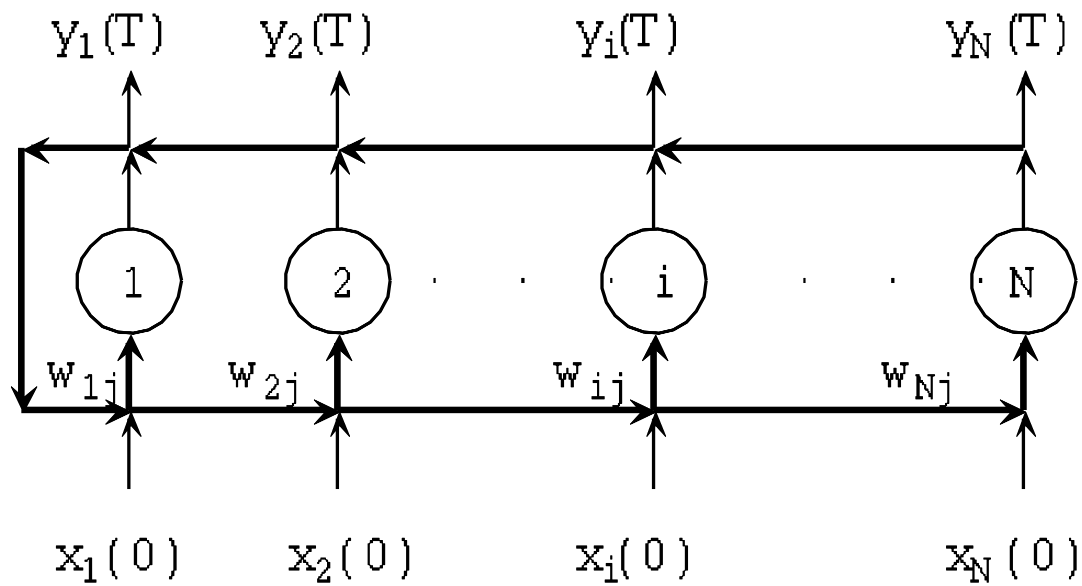
Recurrent neural networks

- Architectures
  - Fully recurrent networks
  - Partially recurrent networks
- Dynamics of recurrent networks
  - Continuous time dynamics
  - Discrete time dynamics
- Applications

Recurrent neural networks

- Architecture
  - Contains feedback connections
  - Depending on the density of feedback connections there are:
    - Fully recurrent networks (Hopfield model)
    - Partially recurrent networks:
      - With contextual units (Elman model, Jordan model)
      - Cellular networks (Chua-Yang model)
- Applications
  - Associative memories
  - Combinatorial optimization problems
  - Prediction
  - Image processing
  - Dynamical systems and chaotical phenomena modelling

Hopfield networks



Architecture:

N fully connected units

Activation function:

Signum/Heaviside

Logistica/Tanh

Parameters:

weight matrix

Notations: $x_i(t)$ – potential (state) of the neuron i at moment t

$y_i(t)=f(x_i(t))$ – the output signal generated by unit i at moment t

$I_i(t)$ – the input signal

w_{ij} – weight of connection between j and i

Hopfield networks

Functioning: - the output signal is generated by the evolution of a dynamical system

- Hopfield networks are equivalent to dynamical systems

Network state:

- the vector of neuron's state $X(t)=(x_1(t), \ldots, x_N(t))$

or

- output signals vector $Y(t)=(y_1(t),\ldots,y_N(t))$

Dynamics:

- Discrete time – recurrence relations (difference equations)
- Continuous time – differential equations

Hopfield networks

Discrete time functioning:

the network state corresponding to moment t+1 depends on the network state corresponding to moment t

Network's state: Y(t)

Variants:

- Asynchronous: only one neuron can change its state at a given time
- Synchronous: all neurons can simultaneously change their states

Network's answer: the stationary state of the network

Hopfield networks

Asynchronous variant:

$$y_{i*}(t+1) = f\left(\sum_{j=1}^{N} w_{i*j}\, y_j(t) + I_{i*}(t)\right)$$

$$y_i(t+1) = y_i(t), \quad i \neq i*$$

Choice of i*:

- systematic scan of {1,2,…,N}
- random (but such that during N steps each neuron changes its state just once)

Network simulation:

- choose an initial state (depending on the problem to be solved)
- compute the next state until the network reach a stationary state (the distance between two successive states is less than ε)

Hopfield networks

Synchronous variant:

$$y_i(t+1) = f\left(\sum_{j=1}^{N} w_{ij} y_j(t) + I_i(t)\right),\ i = \overline{1, N}$$

Either continuous or discrete activation functions can be used

Functioning:

Initial state

REPEAT

compute the new state starting from the current one

UNTIL < the difference between the current state and the previous one is small enough >

Hopfield networks

Continuous time functioning:

$$\frac{dx_i(t)}{dt} = -x_i(t) + \sum_{j=1}^{N} w_{ij} f(x_j(t)) + I_i(t), \; i = \overline{1, N}$$

Network simulation: solve (numerically) the system of differential equations for a given initial state $x_i(0)$

Example: Explicit Euler method

$$\frac{x_i(t+h) - x_i(t)}{h} \cong -x_i(t) + \sum_{j=1}^{N} w_{ij} f(x_j(t)) + I_i(t), \; i = \overline{1, N}$$

$$x_i(t+h) \cong (1-h)x_i(t) + h(\sum_{j=1}^{N} w_{ij} f(x_j(t)) + I_i(t)), \; i = \overline{1, N}$$

Constant input signal :

$$x_i^{new} \cong (1-h)x_i^{old} + h(\sum_{j=1}^{N} w_{ij} f(x_j^{old}) + I_i), \; i = \overline{1, N}$$

Stability properties

Possible behaviours of a network:

- X(t) converged to a stationary state X* (fixed point of the network dynamics)
- X(t) oscillates between two or more states
- X(t) has a chaotic behavior or ||X(t)|| becomes too large

Useful behaviors:

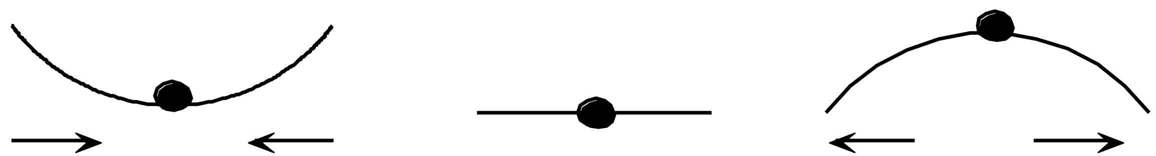
- The network converges to a stationary state
  - Many stationary states: associative memory
  - Unique stationary state: combinatorial optimization problems

- The network has a periodic behavior
  - Modelling of cycles

Obs. Most useful situation: the network converges to a stable stationary state

Stability properties

Illustration:



Asymptotic stable   Stable   Unstable

Formalization:

$$\frac{dX(t)}{dt} = F(X(t)), \quad X(0) = X_0$$

$$F(X^*) = 0$$

X* is asymptotic stable (wrt the initial conditions) if it is

stable

attractive

Stability properties

Stability:

X* is stable if for all $\varepsilon>0$ there exists $\delta(\varepsilon) > 0$ such that:

$$||X0-X^*||< \delta(\varepsilon) \quad \text{implies} \quad ||X(t;X0)-X^*||< \varepsilon$$

Attractive:

X* is attractive if there exists $\delta > 0$ such that:

$$||X0-X^*||< \delta \quad \text{implies} \quad X(t;X0) \to X^*$$

In order to study the asymptotic stability one can use the Lyapunov method.

Stability properties

Lyapunov function:

$$V : R^N \to R, \qquad \text{bounded}$$

$$\frac{dV(X(t))}{dt} < 0, \qquad t > 0$$

- If one can find a Lyapunov function for a system then its stationary solutions are asymptotically stable
- The Lyapunov function is similar to the energy function in physics (the physical systems naturally converges to the lowest energy state)
- The states for which the Lyapunov function is minimum are stable states
- Hopfield networks satisfying some properties have Lyapunov functions.

Stability properties

Stability result for continuous neural networks

If:

- the weight matrix is symmetrical ($w_{ij}=w_{ji}$)
- the activation function is strictly increasing ($f'(u)>0$)
- the input signal is constant ($I(t)=I$)

Then all stationary states of the network are asymptotically stable

Associated Lyapunov function:

$$V(x_1,\dots,x_N) = -\frac{1}{2}\sum_{i,j=1}^{N} w_{ij} f(x_i) f(x_j) - \sum_{i=1}^{N} f(x_i) I_i + \sum_{i=1}^{N} \int_0^{f(x_i)} f^{-1}(z)\,dz$$

Stability properties

Stability result for discrete neural networks (asynchronous case)

If:

- the weight matrix is symmetrical ($w_{ij}=w_{ji}$)
- the activation function is signum or Heaviside
- the input signal is constant ($I(t)=I$)

Then all stationary states of the network are asymptotically stable

Corresponding Lyapunov function

$$V(y_1,\ldots,y_N) = -\frac{1}{2}\sum_{i,j=1}^{N} w_{ij} y_i y_j - \sum_{i=1}^{N} y_i I_i$$

Stability properties

This result means that:

- All stationary states are stable

- Each stationary state has attached an attraction region (if the initial state of the network is in the attraction region of a given stationary state then the network will converge to that stationary state)

Remarks:

- This property is useful for associative memories

- For synchronous discrete dynamics this result is no more true, but the network converges toward either fixed points or cycles of period two

Associative memories

Memory = system to store and recall the information

Address-based memory:

- Localized storage: all components bytes of a value are stored together at a given address
- The information can be recalled based on the address

Associative memory:

- The information is distributed and the concept of address does not have sense
- The recall is based on the content (one starts from a clue which corresponds to a partial or noisy pattern)

Associative memories

Properties:

- Robustness

Implementation:

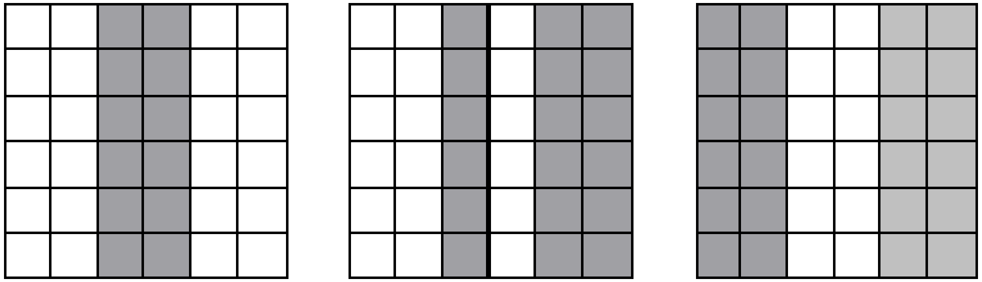
- Hardware:
  - Electrical circuits
  - Optical systems

- Software:
  - Hopfield networks simulators

Associative memories

Software simulations of associative memories:

- The information is binary: vectors having elements from {-1,1}
- Each component of the pattern vector corresponds to a unit in the networks



(a) (b) (c)

Example (a)

(-1,-1,1,1,-1,-1, -1,-1,1,1,-1,-1, -1,-1,1,1,-1,-1, -1,-1,1,1,-1,-1, -1,-1,1,1,-1,-1, -1,-1,1,1,-1,-1)

Associative memories

Associative memories design:

- Fully connected network with N signum units (N is the patterns size)

Patterns storage:

- Set the weights values (elements of matrix W) such that the patterns to be stored become fixed points (stationary states) of the network dynamics

Information recall:

- Initialize the state of the network with a clue (partial or noisy pattern) and let the network to evolve toward the corresponding stationary state.

Associative memories

Patterns to be stored: $\{X^1,\ldots,X^L\}$, X^l in $\{-1,1\}^N$

Methods:

- Hebb rule
- Pseudo-inverse rule (Diederich – Opper algorithm)

Hebb rule:

- It is based on the Hebb's principle: "the synaptic permeability of two neurons which are simultaneously activated is increased"

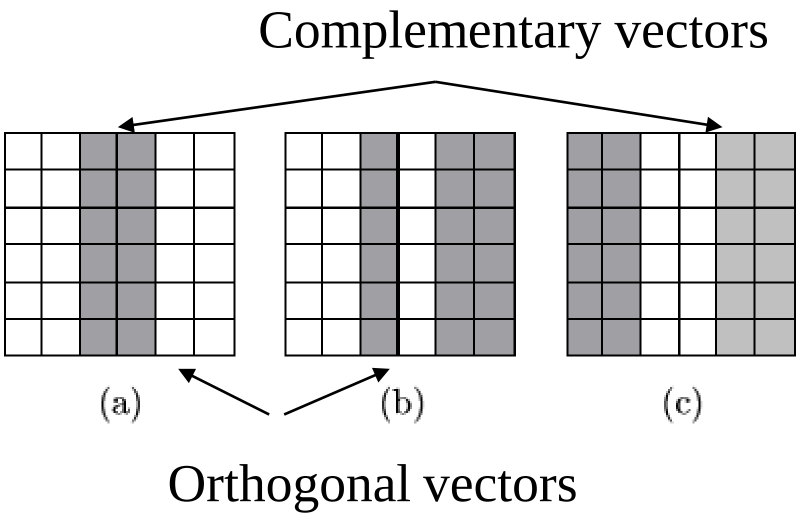
$$w_{ij} = \frac{1}{N}\sum_{l=1}^{L} x_i^l x_j^l$$

Associative memories

$$w_{ij} = \frac{1}{N}\sum_{l=1}^{L} x_i^l x_j^l$$

Properties of the Hebb's rule:

- If the vectors to be stored are orthogonal (statistically uncorrelated) then all of them become fixed points of the network dynamics
- Once the vector X is stored the vector –X is also stored
- An improved variant: the pseudo-inverse method



Associative memories

Pseudo-inverse method:

$$w_{ij} = \frac{1}{N} \sum_{l,k} x_i^l (Q^{-1})_{lk} x_j^l$$

$$Q_{lk} = \frac{1}{N} \sum_{i=1}^{N} x_i^l x_i^k$$

- If Q is invertible then all elements of $\{X^1,\ldots,X^L\}$ are fixed points of the network dynamics

- In order to avoid the costly operation of inversion one can use an iterative algorithm for weights adjustment

Associative memories

Diederich-Opper algorithm :

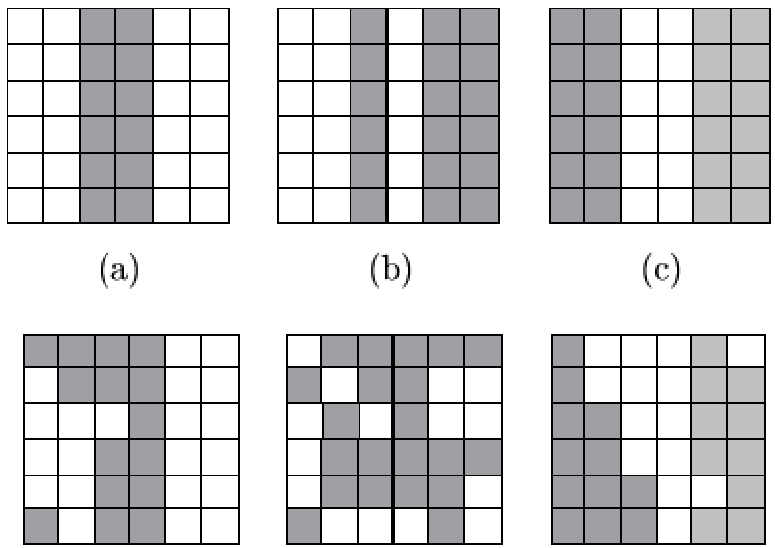
$t := 0$
Initialize W(0) using the Hebb rule
REPEAT
    FOR $l := 1, L$ DO
        $y_i^l := \sum_{j=1}^{N} w_{ij}(t) x_j^l,\ i = \overline{1, N}$
        $w_{ij}(t+1) := w_{ij}(t) + \frac{1}{N}(x_i^l - y_i^l) x_j^l,\ i = \overline{1, N},\ j = \overline{1, N}$
    $t := t + 1$
UNTIL $\|W(t) - W(t-1)\| < \epsilon$

Associative memories

Recall process:

- Initialize the network state with a starting clue
- Simulate the network until the stationary state is reached.

Stored patterns



Noisy patterns (starting clues)

Associative memories

Storage capacity:

- The number of patterns which can be stored and recalled (exactly or approximately)
- Exact recall: capacity=N/(4lnN)
- Approximate recall (prob(error)=0.005): capacity = 0.15*N

Spurious attractors:

- These are stationary states of the networks which were not explicitly stored but they are the result of the storage method.

Avoiding the spurious states

- Modifying the storage method
- Introducing random perturbations in the network's dynamics

Solving optimization problems

- First approach: Hopfield & Tank (1985)
  - They propose the use of a Hopfield model to solve the traveling salesman problem.
  - The basic idea is to design a network whose energy function is similar to the cost function of the problem (e.g. the tour length) and to let the network to naturally evolve toward the state of minimal energy; this state would represent the problem's solution.

Solving optimization problems

A constrained optimization problem:

find $(y_1,\ldots,y_N)$ satisfying:

it minimizes a cost function $C:R^N\text{->}R$

it satisfies some constraints as $R_k(y_1,\ldots,y_N)=0$ with

R_k nonnegative functions

Main steps:

- Transform the constrained optimization problem in an unconstrained optimization one (penalty method)
- Rewrite the cost function as a Lyapunov function
- Identify the values of the parameteres (W and I) starting from the Lyapunov function
- Simulate the network

Solving optimization problems

Step 1: Transform the constrained optimization problem in an unconstrained optimization one

$$C^*(y_1,\ldots,y_N) = aC(y_1,\ldots,y_N) + \sum_{k=1}^{r} b_k R_k(y_1,\ldots,y_N)$$

$$a, b_k > 0$$

The values of a and b are chosen such that they reflect the relative importance of the cost function and constraints

Solving optimization problems

Step 2: Reorganizing the cost function as a Lyapunov function

$$C(y_1,\ldots,y_N) = -\frac{1}{2}\sum_{i,j=1}^{N} w_{ij}^{obj} y_i y_j - \sum_{i=1}^{N} I_i^{obj} y_i$$

$$R_k(y_1,\ldots,y_N) = -\frac{1}{2}\sum_{i,j=1}^{N} w_{ij}^{k} y_i y_j - \sum_{i=1}^{N} I_i^{k} y_i, \quad k=\overline{1,r}$$

Remark: This approach works only for cost functions and constraints which are linear or quadratic

Solving optimization problems

Step 3: Identifying the network parameters:

$$w_{ij} = aw_{ij}^{obj} + \sum_{k=1}^{r} b_k w_{ij}^k, \quad i, j = \overline{1, N}$$

$$I_i = aI_i^{obj} + \sum_{k=1}^{r} b_k I_i^k, \quad i = \overline{1, N}$$

Solving optimization problems

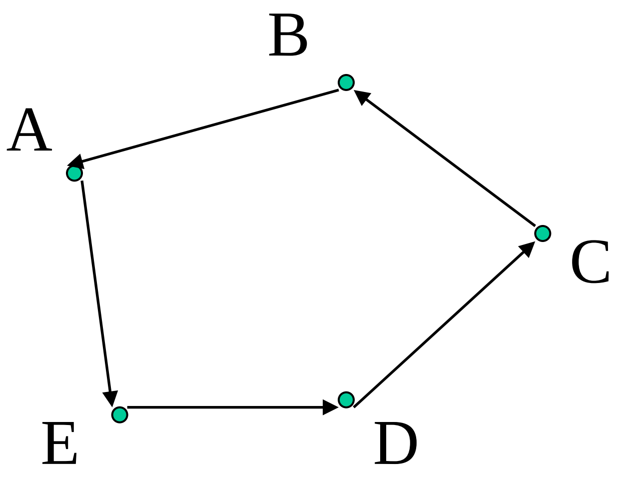
Designing a neural network for TSP (n towns):

N=n*n neurons

The state of the neuron (i,j) is interpreted as follows:

1 - the town i is visited at time j

0 - otherwise



AEDCB

	1	2	3	4	5
A	1	0	0	0	0
B	0	0	0	0	1
C	0	0	0	1	0
D	0	0	1	0	0
E	0	1	0	0	0

Solving optimization problems

Constraints:

- at a given time only one town is visited (each column contains exactly one value equal to 1)
- each town is visited only once (each row contains exactly one value equal to 1)

	1	2	3	4	5
A	1	0	0	0	0
B	0	0	0	0	1
C	0	0	0	1	0
D	0	0	1	0	0
E	0	1	0	0	0

Cost function:

the tour length = sum of distances between towns visited at consecutive time moments

Solving optimization problems

Constraints and cost function:

$$\sum_{j=1}^{n}\left(\sum_{i=1}^{n} y_{ij} - 1\right)^2 = 0$$

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{n} y_{ij} - 1\right)^2 = 0$$

$$C(Y) = \sum_{i=1}^{n}\sum_{k=1,k\neq i}^{n}\sum_{j=1}^{n} c_{ik} y_{ij}(y_{k,j-1} + y_{k,j+1})$$

Cost function in the unconstrained case:

$$C^*(Y) = \frac{a}{2}\sum_{i=1}^{n}\sum_{k=1,k\neq i}^{n}\sum_{j=1}^{n} c_{ik} y_{ij}(y_{k,j-1} + y_{k,j+1}) +$$

$$\frac{b}{2}\left(\sum_{j=1}^{n}\left(\sum_{i=1}^{n} y_{ij} - 1\right)^2 + \sum_{i=1}^{n}\left(\sum_{j=1}^{n} y_{ij} - 1\right)^2\right)$$

Solving optimization problems

$$V(Y) = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} w_{ij,kl}\, y_{ij}\, y_{kl} - \sum_{i=1}^{n}\sum_{j=1}^{n} y_{ij} I_{ij}$$

$$C^*(Y) = \frac{a}{2}\sum_{i=1}^{n}\sum_{k=1,k\neq i}^{n}\sum_{j=1}^{n} c_{ik}\, y_{ij}\,(y_{k,j-1} + y_{k,j+1}) +$$

$$\frac{b}{2}\left(\sum_{j=1}^{n}\left(\sum_{i=1}^{n} y_{ij} - 1\right)^2 + \sum_{i=1}^{n}\left(\sum_{j=1}^{n} y_{ij} - 1\right)^2\right)$$

Identified parameters:

$$w_{ij,kl} = -a c_{ik}(\delta_{l,j-1} + \delta_{l,j+1}) - b(\delta_{ik} + \delta_{jl} + \delta_{ik}\delta_{jl})$$

$$w_{ij,ij} = 0$$

$$I_{ij} = 2b$$

Prediction in time series

- Time series = sequence of values measured at successive moments of time

- Examples:
  - Currency exchange rate evolution
  - Stock price evolution
  - Biological signals (EKG)

- Aim of time series analysis: predict the future value(s) in the series

Time series

The prediction (forecasting) is based on a model which describes the dependency between previous values and the next value in the series.

$$x(t+1) = F(x(t), x(t-1), \ldots, x(t-p+1), \varphi_1, \varphi_2, \ldots, \varphi_s)$$

Order of the model

Parameters corresponding to external factors

Time series

The model associated to a time series can be:

- Linear
- Nonlinear

- Deterministic
- Stochastic

Example: autoregressive model (AR(p))

$$x(t+1) = \sum_{i=0}^{p-1} \alpha_i x(t-i) + \epsilon$$

noise = random variable from N(0,1)

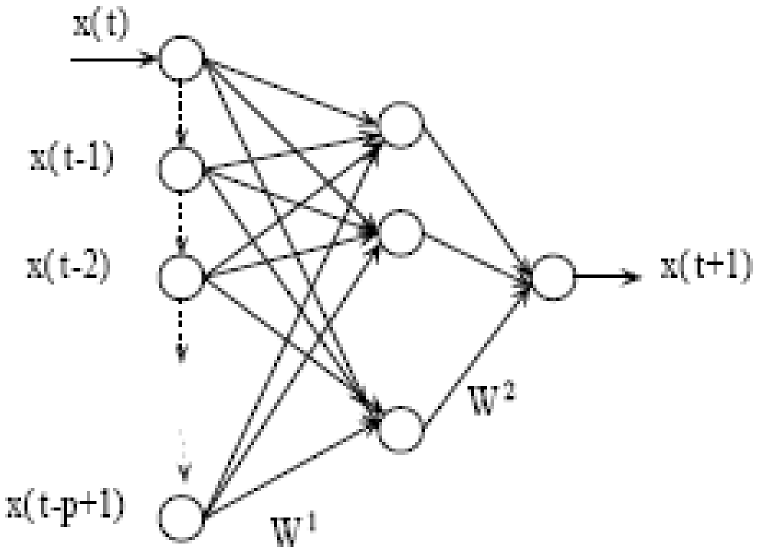
Time series

Neural networks. Variants:

- The order of the model is known
  - Feedforward neural network with delayed input layer (p input units)
- The order of the model is unknown
  - Network with contextual units (Elman network)

Networks with delayed input layer

Architecture:



Functioning:

$$y = \sum_{k=1}^{K} w_k^2 f\left(\sum_{j=0}^{p-1} w_{kj}^1 x(t-j)\right)$$

Networks with delayed input layer

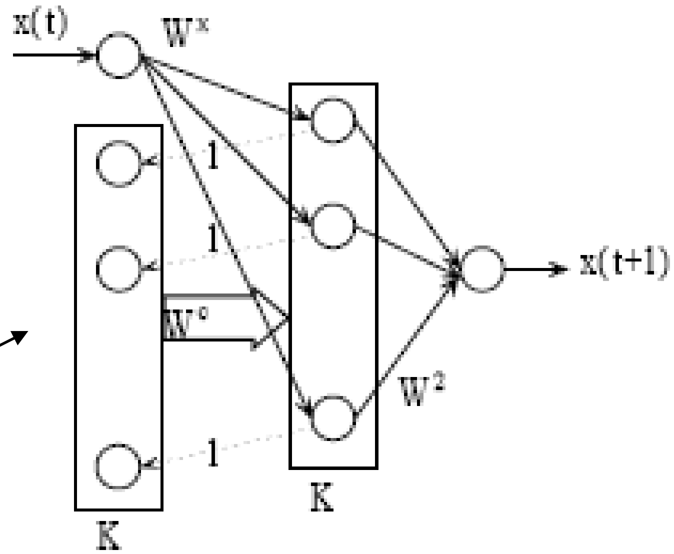
Training:

- Training set: $\{((x_l, x_{l-1}, \ldots, x_{l-p+1}), x_{l+1})\}_{l=1..L}$
- Training algorithm: BackPropagation
- Drawback: needs the knowledge of p

Elman network

Architecture:

Contextual units



Functioning:

$$y = \sum_{k=1}^{K} w_k^2 h_k(t)$$

$$h_k(t) = f\left(w_k^x x(t) + \sum_{j=1}^{K} w_{kj}^c h_j(t-1) \right) \qquad h_j(0) = 0.$$

Rmk: the contextual units contain copies of the outputs of the hidden layers corresponding to the previous moment

Elman network

Training

Training set : {(x(1),x(2)),(x(2),x(3)),…(x(t-1),x(t))}

Sets of weights:

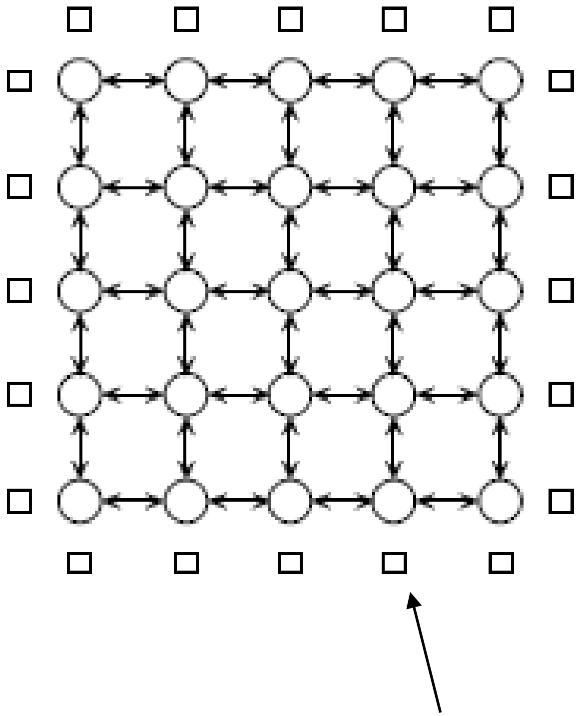
- Adaptive: W^x, W^c si W^2
- Fixed: the weights of the connections between the hidden and the contextual layers.

Training algorithm: BackPropagation

Cellular networks

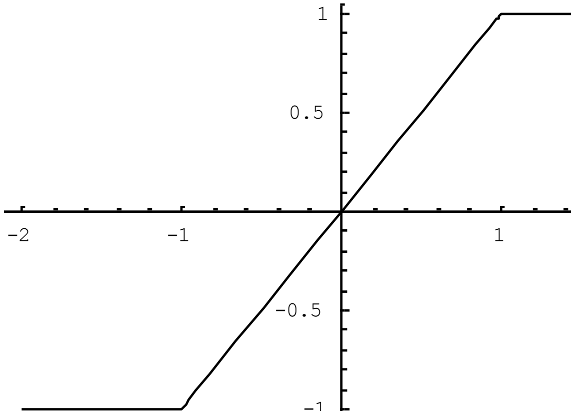
Architecture:

- All units have a double role: input and output units
- The units are placed in the nodes of a two dimensional grid
- Each unit is connected only with units from its neighborhood (the neighborhoods are defined as in the case of Kohonen’s networks)
- Each unit is identified through its position p=(i,j) in the grid



virtual cells (used to define the context for border cells)

Cellular networks



Activation function: ramp

$$f(u) = (|u+1| - |u-1|)/2$$

Notations:

$X_p(t)$ – state of unit p at time t

$Y_p(t)$ - output signal

$U_p(t)$ – control signal

$I_p(t)$ – input from the environment

a_{pq} – weight of connection between unit q and unit p

b_{pq} - influence of control signal U_q on unit p

Cellular networks

Functioning: Signal generated by other units; Control signal

$$\frac{dx_p(t)}{dt} = -x_p(t) + \sum_{q\in V(p)} a_{pq}y_q(t) + \sum_{q\in V(p)} b_{pq}u_q(t) + i_p(t), \quad p \in \mathcal{L}$$

Input signal

Remarks:

- The grid has a boundary of fictitious units (which usually generate signals equal to 0)
- Particular case: the weights of the connections between neighboring units do not depend on the positions of units

Example: if p=(i,j), q=(i-1,j), p'=(i',j'), q'=(i'-1,j') then

$$a_{pq}= a_{p'q'}=a_{-1,0}$$

Cellular networks

These networks are called cloning template cellular networks
Example:

$$V_1(i,j) = \{(i-1,j-1),(i-1,j),(i-1,j+1),(i,j-1),(i,j),(i,j+1),(i+1,j-1),(i+1,j),(i+1,j+1)\}$$

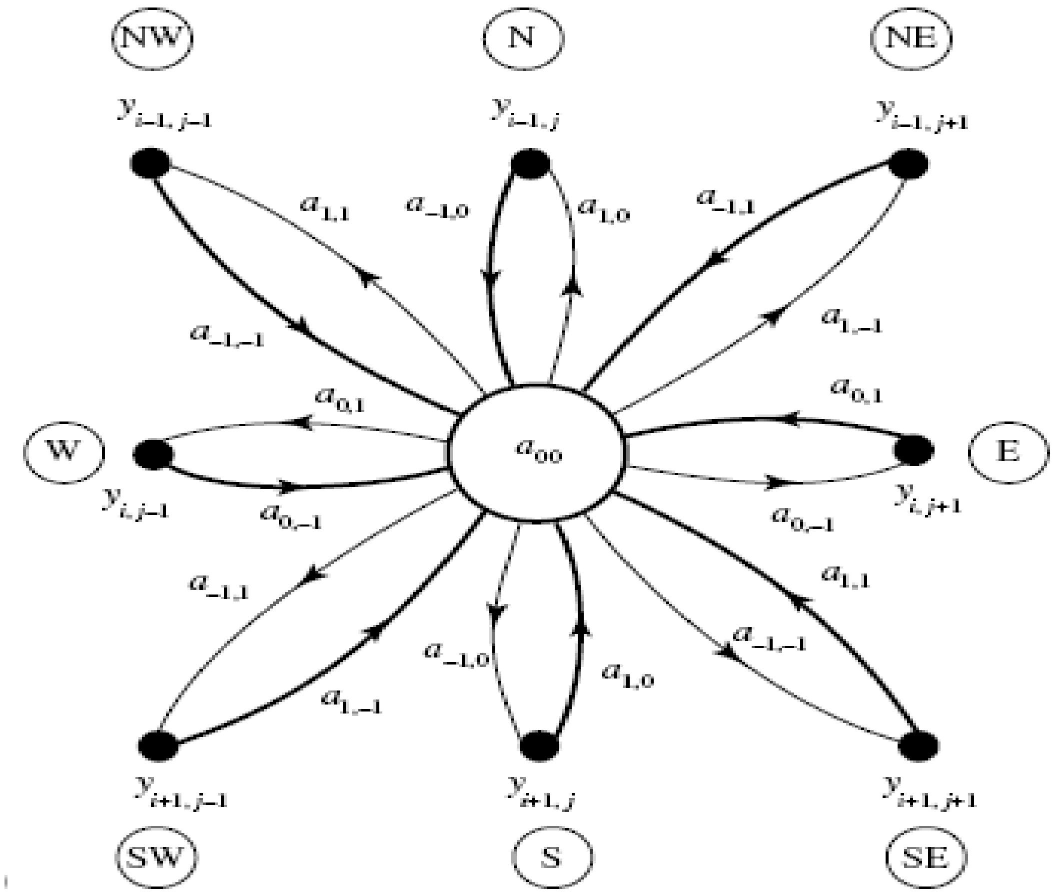
$$A = \begin{bmatrix} a_{-1,-1} & a_{-1,0} & a_{-1,1} \\ a_{0,-1} & a_{0,0} & a_{0,1} \\ a_{1,-1} & a_{1,0} & a_{1,1} \end{bmatrix} \qquad B = \begin{bmatrix} b_{-1,-1} & b_{-1,0} & b_{-1,1} \\ b_{0,-1} & b_{0,0} & b_{0,1} \\ b_{1,-1} & b_{1,0} & b_{1,1} \end{bmatrix}$$

$$\frac{dx_{i,j}(t)}{dt} = -x_{i,j}(t) + \sum_{(k,l)\in V^*} a_{k,l}y_{i+k,j+l}(t) + \sum_{(k,l)\in V^*} b_{k,l}u_{i+k,j+l}(t) + I, \qquad i,j \in \{1,\ldots,n\}$$

$$V^* = \{(-1,-1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1)\}$$

Cellular networks

Illustration of the cloning template elements



Cellular networks

Software simulation = equivalent to numerical solving of a differential system (initial value problem)

Explicit Euler method

$$x_p(t+1) = (1-h)x_p(t) + h\Big(\sum_{(k,l)\in V^*} a_{k,l}y_{i+k,j+l}(t) + \sum_{(k,l)\in V^*} b_{k,l}u_{i+k,j+l}(t) + I\Big), \qquad i,j \in \{1,\dots,n\}$$

Applications:

- Gray level image processing
- Each pixel corresponds to a unit of the network
- The gray level is encoded by using real values from [-1,1]

Cellular networks

Image processing:

- Depending on the choice of templates, of control signal (u), initial condition (x(0)), boundary conditions (z) different image processing tasks can be solved:
  - Edge detection in binary images
  - Gap filling in binary images
  - Noise elimination in binary images
  - Identification of horizontal/vertical line segments

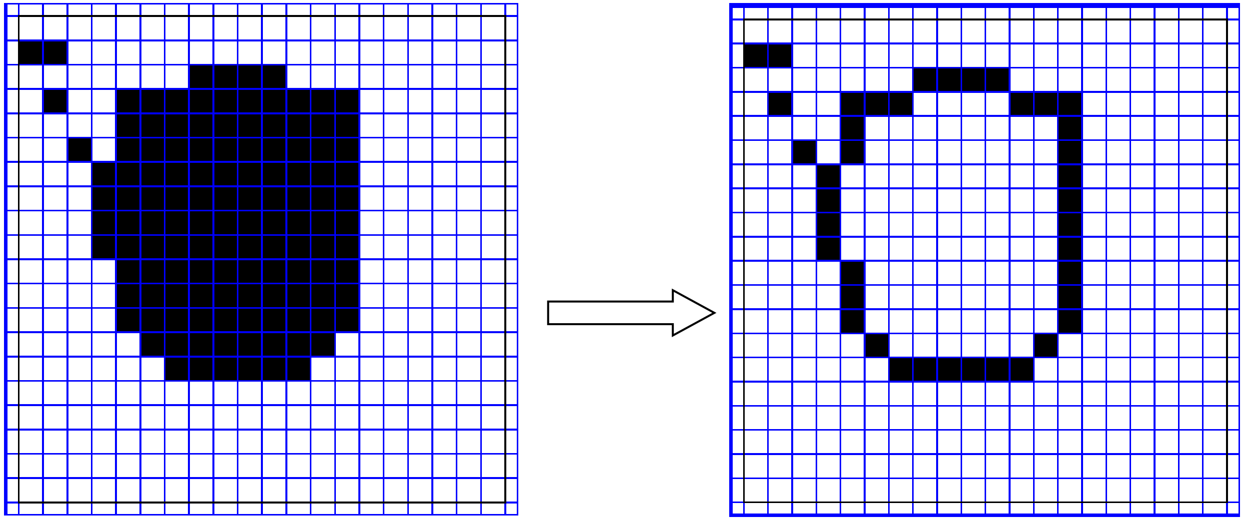
Cellular networks

Example 1: edge detection

z=-1, U=input image, h=0.1

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$I = -1, X(0) = U$$



http://www.isiweb.ee.ethz.ch/haenggi/CNN_web/CNNsim_adv.html

Cellular networks

Example 2: gap filling

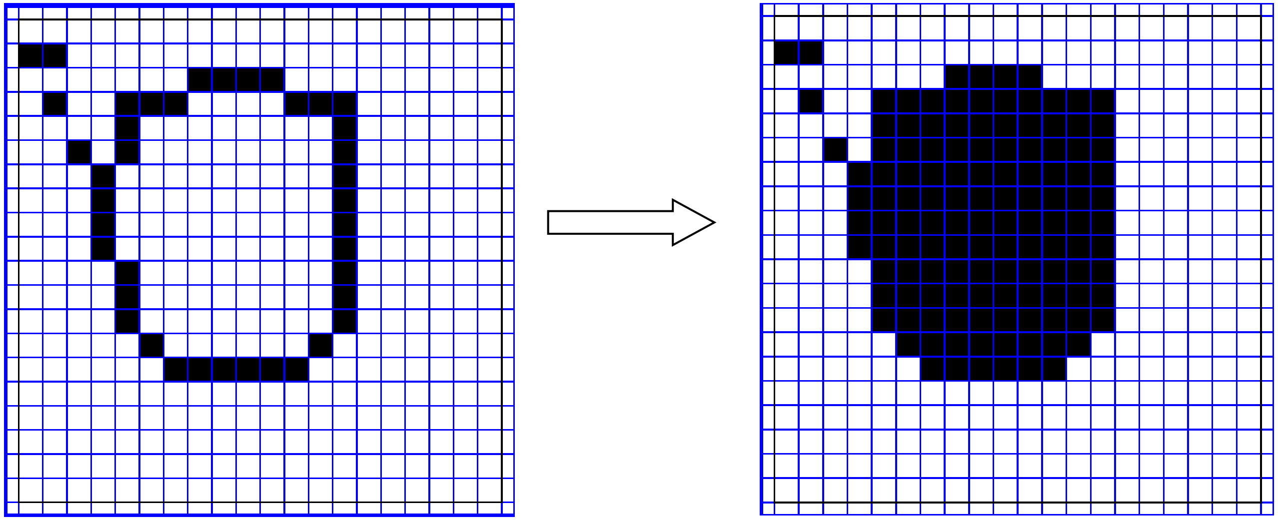
z=-1,

U=input image,

h=0.1

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1.5 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$I = 0.5, x_{ij}(0) = 1 \text{ (all pixels are 1)}$$



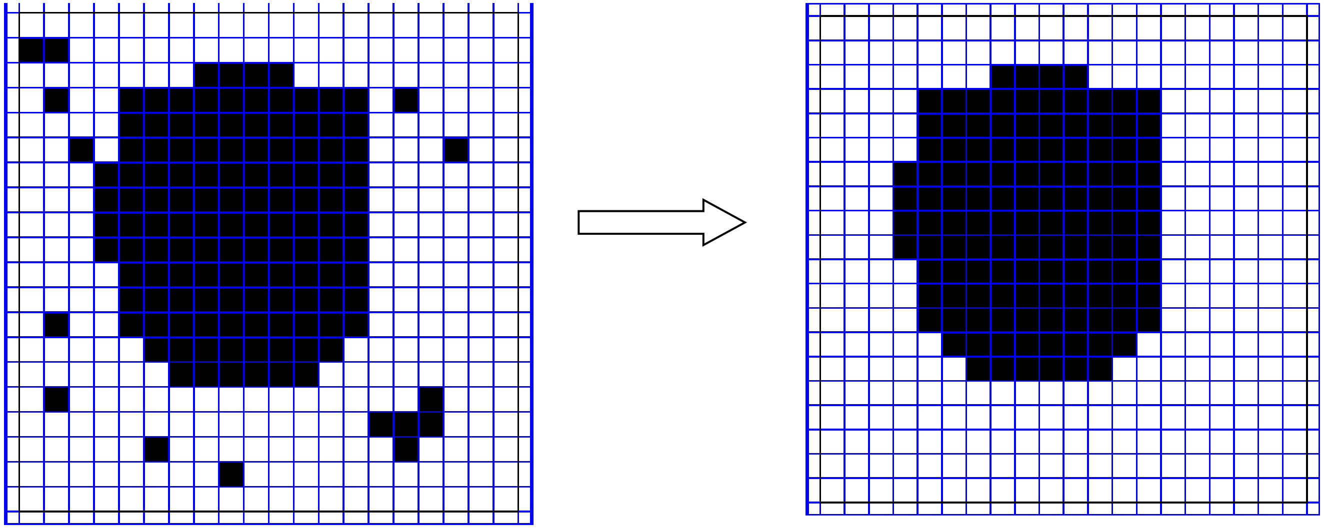
Cellular networks

Example 3: noise removing

z=-1, U=input image, h=0.1

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$I = 0, X(0) = U$$



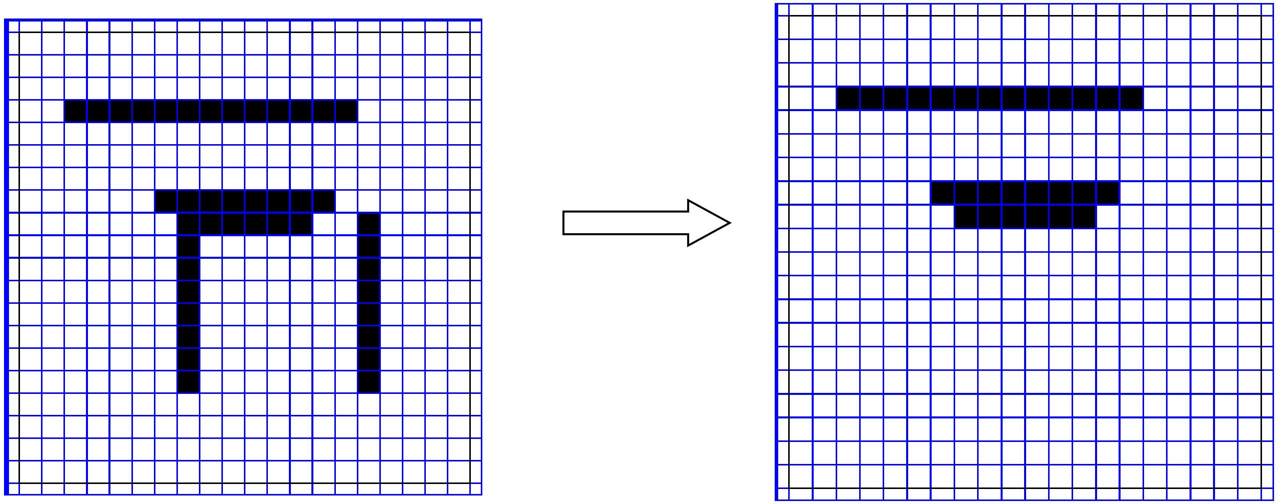
Cellular networks

Example 4: horizontal line detection

z=-1, U=input image, h=0.1

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$I = -1, X(0) = U$$



Other related models

Reservoir computing (www.reservoir-computing.org)

Particularities:

- These models use a set of hidden units (called reservoir) which are arbitrarly connected (their connection weights are randomly set; each of these units realize a nonlinear transformation of the signals received from the input units.

- The output values are obtained by a linear combination of the signals produced by the input units and by the reservoir units.

- Only the weights of connections toward the output units are trained

Other related models

Reservoir computing (www.reservoir-computing.org)

Variants:

- Temporal Recurrent Neural Network (Dominey 1995)
- Liquid State Machines (Natschläger, Maass and Markram 2002)
- Echo State Networks (Jaeger 2001)
- Decorrelation-Backpropagation Learning (Steil 2004)

Other related models

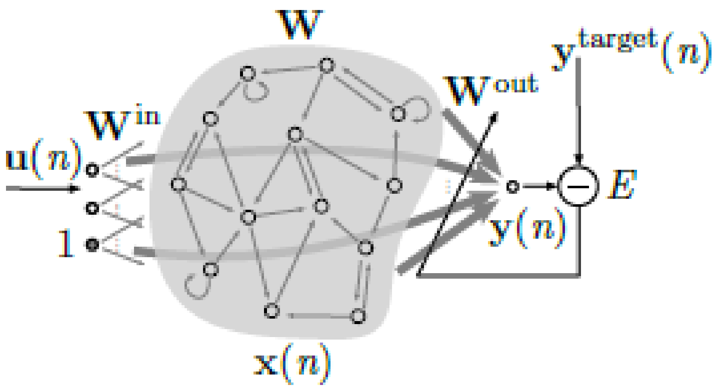
Echo State Networks:

U(t) = input vector

X(t) = reservoir state vector

Z(t)=[U(t);X(t)] = concatenated input and state vectors

Y(t) = output vector



M. Lukosevicius – Practical Guide to Applying Echo State Networks

$X(t)=(1-a)X(t-1)+a \tanh(W^{in}\, U(t)+W\, X(t-1))$

$Y(t)=W^{out}\, Z(t)$

W^{in}, W – random matrices (W is scaled such that the spectral radius has a predefined value);

W^{out} - set by training

Other related models

Applications of reservoir computing:

- Speech recognition
- Handwritten text recognition
- Robot control
- Financial data prediction
- Real time prediction of epilepsy seizures

Other related models

Deep learning (http://deeplearning.net/)

Particularities:

- Deep architecture = many layers (aim: hierarchical extraction of data features);

- Unsupervised training based on Restricted Boltzmann Machines) followed by a fine tuning of weights using a supervised training (e.g. Backpropagation)

Remarks:

- Boltzmann Machines = recurrent neural networks with binary stochastic units
- Restricted BM = recurrent neural networks with bidirectional connections only between the units belonging to different subsets of units (e.g. subsets: visible units, hidden units)
- There are feed-forward deep neural networks (e.g: Convolutional Neural Networks)

Other related models

Deep learning (http://deeplearning.net/)

Applications:

- Image classification, objects detection (e.g. Face recognition – Deep Face)
- Speech recognition (Google Brain, Siri)
- Semantic indexing (ex: word2vec) and automated translation
- Dream simulation (http://npcontemplation.blogspot.ca/2012/02/machine-that-can-dream.html)