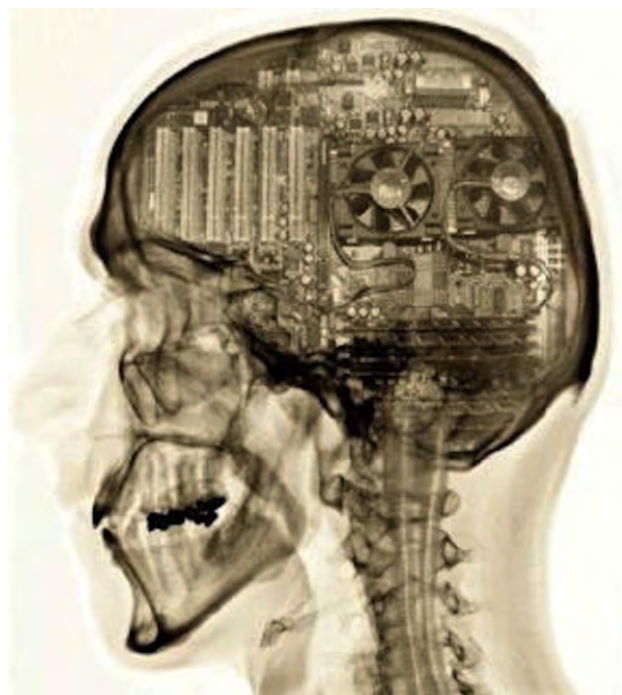




# **Machine Learning And Intelligent Systems**

## **LECTURE SLIDES**

### **PART 2**



**Prof. Bernard Merialdo**  
Fall 2017

**Data ScienceDepartment**  
**EURECOM**



## Content Part 2

---

Neural Network History	219
The Perceptron	225
Multi-Layer Perceptrons MLP	258
MLP Training	282
Deep Learning Networks	325
Associative Memory Networks	337
Self Organizing Maps	374
Simulated Annealing	403
Metropolis Algorithm	421
The Travelling Salesman Problem	440

## Neural Networks

## Neural Network History

---

- ◆ Perceptrons: 1960's
  - Developed for vision applications
  - Capable of automatic training
  - Limited capability
- ◆ Multi-Layer Perceptrons: 1980's
  - New training algorithm: back-propagation
  - Lots of applications
  - Many variants (Hopfield, Boltzman...)
  - Mimic human brain
- ◆ Deep Networks: 2010's
  - Outperforms current state-of-the-art systems on pattern recognition tasks

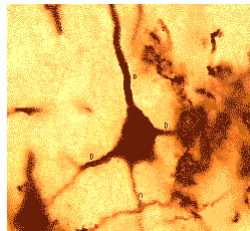
## Human Brain

---

- ◆ 100 billion neurons
- ◆ 1.3 kg



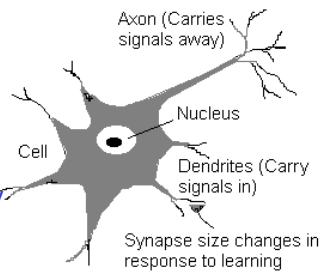
- ◆ neuron:
  - dendrites (input)
  - cell (processing)
    - 0.1 mm
  - axon (output)
    - 1 mm - 1 m
  - synapse (connection)



## Neurones

- ◆ Many-inputs / one-output
  - output can be *excited* or *not*
  - incoming signals determine if the neuron shall excite ("fire")
  - output subject to attenuation in the *synapses* (junctions)

- ◆ Hebb's Rule:
  - If an input of a neuron is repeatedly and persistently causing the neuron to fire, a metabolic change happens in the synapse of that particular input to reduce its resistance

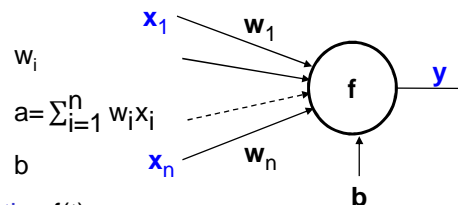


## Artificial Neuron

- ◆ A neuron is a processing unit
- ◆ It receives input signals ( $x_1, x_2, \dots, x_n$ )
- ◆ It produces an output signal  $y$

$$y = f\left(\sum_{i=1}^n w_i x_i - b\right)$$

- Weights
- Activation
- Threshold (or bias)
- Transfer (activation) function  $f(t)$



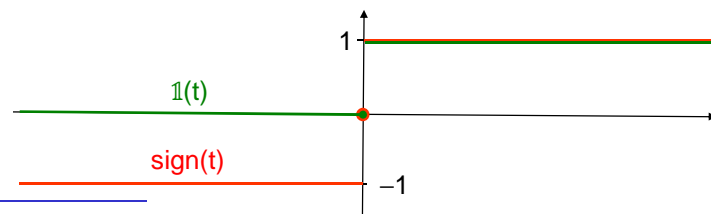
## Neuron Behaviour

- Typically  $f(t)=\text{sign}(t)$  or  $f(t)=\mathbb{1}(t)$

$$\bullet \quad \text{sign}(t) = \begin{cases} +1 & \text{if } t > 0 \\ 0 & \text{if } t = 0 \\ -1 & \text{if } t < 0 \end{cases} \quad \mathbb{1}(t) = \begin{cases} +1 & \text{if } t > 0 \\ 0 & \text{if } t = 0 \\ 0 & \text{if } t < 0 \end{cases}$$

- If activation is:

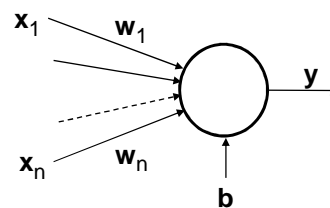
- Low:  $\sum_{i=1}^n w_i x_i < b$       High:  $\sum_{i=1}^n w_i x_i > b$
- neuron is not activated      neuron is activated
- output  $y$  is -1 (or 0)      output  $y$  is +1



## Threshold Simplification

- Standard definition

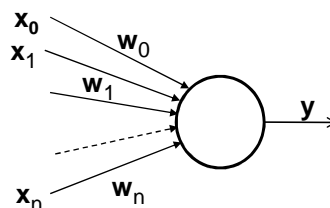
$$y = f\left(\sum_{i=1}^n w_i x_i - b\right)$$



- Simplified formulation by adding one constant input:

- $w_0 = b$  and  $x_0 = -1$
- $w_0 = -b$  and  $x_0 = 1$

$$y = f\left(\sum_{i=0}^n w_i x_i\right)$$



## The Perceptron

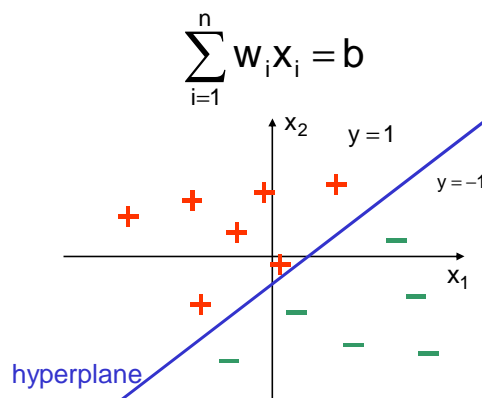
- ◆ A perceptron is a single neuron network where

$$y = \text{sign}\left(\sum_{i=1}^n w_i x_i - b\right)$$

- Its behaviour is defined by its weights and the value of the threshold  $b$ .
- It is possible to estimate automatically the parameters based on a set of training examples.

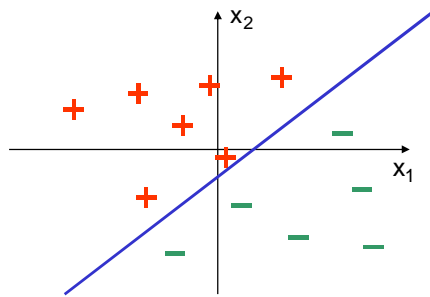
## Perceptron Representation

- ◆ The output is the characteristic function of a half-space, which is limited by the hyperplane

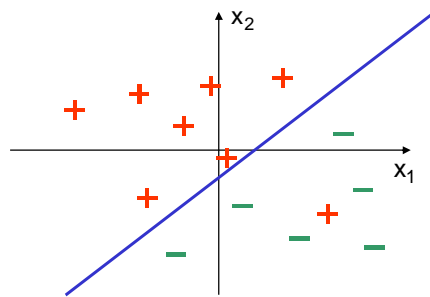


## Linear Separability

- ◆ Given two sets of points, is there a perceptron which classifies them ?



YES



NO

- ◆ True only if sets are linearly separable

## Perceptron Training

- ◆ Given a training set:
  - T Training examples
    - For each example, coordinates + class  
 $(x_1^j, x_2^j, \dots, x_n^j, t^j)$      $j = 1, 2, \dots, T$ ,     $t^j = \pm 1$
- ◆ We want to find the values of weights  $w_i$  such that:

$$t^j = \text{sign}\left(\sum_{i=0}^n w_i x_i^j\right) \quad \text{for all } j = 1, 2, \dots, T$$



## Perceptron Training Algorithm

---

- ◆ Iterative algorithm:

- ◆ Initialize weights to random values  $w^{(0)}_i$

- ◆ For each training vector  $j$ :

- Compute the output:

$$y^{j(k)} = \text{sign}\left(\sum_{i=0}^n w_i^{(k)} x_i^j\right)$$

- If  $t^j \neq y^j$  then update the weights:

$$w_i^{(k+1)} = w_i^{(k)} + (t^j - y^{j(k)}) x_i^j$$

- ◆ Continue until no update happens over the whole training set

## Perceptron Training Algorithm

---

- ◆ What is the effect of the update formula:

- if  $y^{j(k)} = t^j$ , no change

- if  $y^{j(k)} \neq t^j$ :  $w_i^{(k+1)} = w_i^{(k)} + (t^j - y^{j(k)}) x_i^j$

- if  $y^{j(k)} = 1, t^j = -1$ :  $w_i^{(k+1)} = w_i^{(k)} - 2 x_i^j$

$$\sum_i w_i^{(k+1)} x_i^j = \sum_i w_i^{(k)} x_i^j - 2 \sum_i (x_i^j)^2 \quad \text{Activation decreases}$$

- if  $y^{j(k)} = -1, t^j = 1$ :  $w_i^{(k+1)} = w_i^{(k)} + 2 x_i^j$

$$\sum_i w_i^{(k+1)} x_i^j = \sum_i w_i^{(k)} x_i^j + 2 \sum_i (x_i^j)^2 \quad \text{Activation increases}$$

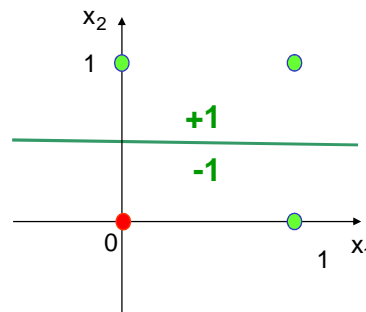
Activation is always modified in the direction of the true class

## Perceptron Example

- ♦ OR function
- ♦ training data:

$x_1$	$x_2$	$t$
0	0	-1
1	0	1
0	1	1
1	1	1

- ♦ initial perceptron
  - $w_1 = 0, w_2 = 1, b = 0.5$



(remember that  $b=w_0$  with  $x_0=-1$ )

## Perceptron Example

- $w_1 = 0, w_2 = 1, b = 0.5$
- ♦ Iteration 1, example 1

$x_1$	$x_2$	$t$	activation	threshold	output
0	0	-1	0	0.5	-1

## Perceptron Example

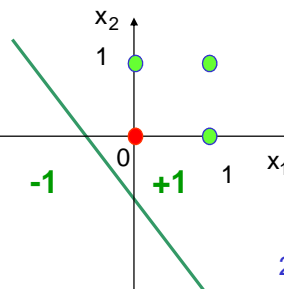
- $w_1 = 0, w_2 = 1, b = 0.5$

### ♦ Iteration 1, example 2

$x_1$	$x_2$	$t$	activation	threshold	output
1	0	1	0	0.5	-1

### ♦ Update

- $w_1 = 0 + (1+1) * 1 = 2$
- $w_2 = 1 + (1+1) * 0 = 1$
- $b = 0.5 + (1+1) * (-1) = -1.5$



## Perceptron Example

- $w_1 = 2, w_2 = 1, b = -1.5$

### ♦ Iteration 1, example 3

$x_1$	$x_2$	$t$	activation	threshold	output
0	1	1	1	-1.5	1

### ♦ Iteration 1, example 4

$x_1$	$x_2$	$t$	activation	threshold	output
1	1	1	3	-1.5	1

## Perceptron Example

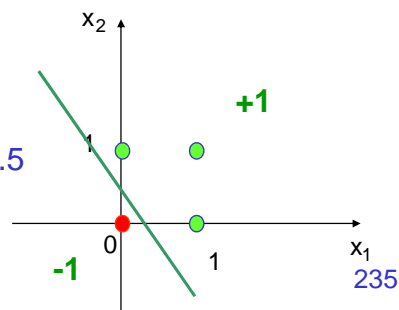
- $w_1 = 2, w_2 = 1, b = -1.5$

### ♦ Iteration 2, example 1

$x_1$	$x_2$	$t$	activation	threshold	output
0	0	-1	0	-1.5	1

### ♦ Update

- $w_1 = 2 + (-1-1) * 0 = 2$
- $w_2 = 1 + (-1-1) * 0 = 1$
- $b = -1.5 + (-1-1) * (-1) = 0.5$



## Perceptron Training Theorem

### ♦ Theorem:

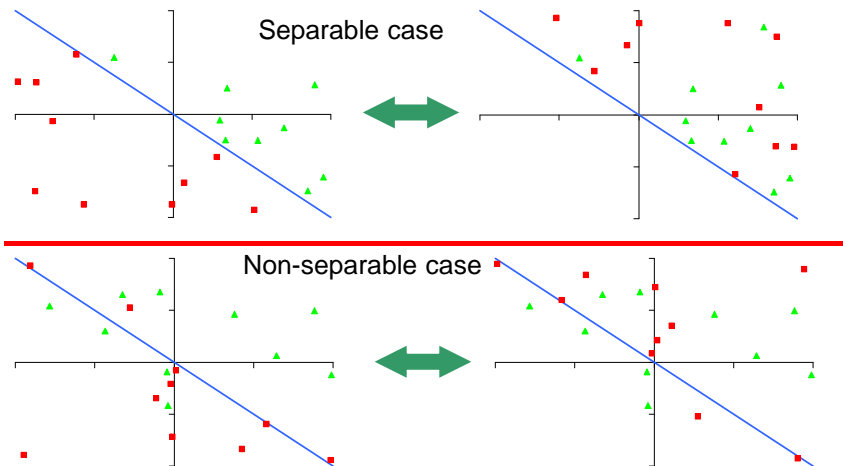
if the two sets (positive and negative examples) are linearly separable, then the perceptron training algorithm will stop after a finite number of steps.

### ♦ Proof:

- First note that  $\sum w_i x_i \leq 0 \Leftrightarrow \sum w_i (-x_i) \geq 0$   
we can replace  $(x_1^j, \dots, x_n^j, -1) \rightarrow (-x_1^j, \dots, -x_n^j, +1)$   
so, we may assume  $t^j > 0 \quad \forall j$

- Same update formula:  $w_i^{(k+1)} = w_i^{(k)} + (t^j - y^j) x_i^j$
- So we need to find  $w_i$  such that  $\sum w_i x_i^j > 0 \quad \forall j$

## Problem Symetrization



## Proof (1/3)

- ◆ Let  $\bar{w}$  be a solution
  - It exists, since the sets are separable
- ◆ Let  $a = \min_j \sum_{i=0}^n \bar{w}_i x_i^j > 0$      $A = \max_j \sum_{i=0}^n (x_i^j)^2 > 0$
- ◆ and:
 
$$\cos(w, \bar{w}) = \frac{\sum w_i \bar{w}_i}{\sqrt{\sum w_i^2} \sqrt{\sum \bar{w}_i^2}} \leq 1$$
- ◆ If  $w_i^{(k)}$  is changed:
 
$$w_i^{(k)} \rightarrow w_i^{(k+1)} = w_i^{(k)} + (t^j - y^j) x_i^j = w_i^{(k)} + 2x_i^j$$

(because  $t=+1$ )

### Proof (2/3)

$$\begin{aligned} \diamond \quad \sum w_i^{(k+1)} \bar{w}_i &= \sum [w_i^{(k)} + 2x_i^j] \bar{w}_i = \sum w_i^{(k)} \bar{w}_i + 2 \sum \bar{w}_i x_i^j \\ &\geq \sum w_i^{(k)} \bar{w}_i + 2a \end{aligned}$$

$$\text{So: } \sum w_i^{(n)} \bar{w}_i \geq \sum w_i^{(0)} \bar{w}_i + 2na$$

$$\diamond \quad \sum w_i^{(k+1)2} = \sum (w_i^{(k)} + 2x_i^j)^2 = \sum w_i^{(k)2} + \underbrace{4 \sum w_i^{(k)} x_i^j}_{\leq 0} + \underbrace{4 \sum x_i^{j2}}_{\leq A}$$

$$\text{So: } \sum w_i^{(n)2} \leq \sum w_i^{(0)2} + 4nA$$

◆ Finally:

$$\cos(w^{(n)}, \bar{w}) = \frac{\sum w_i^{(n)} \bar{w}_i}{\sqrt{\sum w_i^{(n)2}} \sqrt{\sum \bar{w}_i^2}} \geq \frac{\sum w_i^{(0)} \bar{w}_i + 2na}{\sqrt{\sum w_i^{(0)2} + 4nA} \sqrt{\sum \bar{w}_i^2}}$$

### Proof (3/3)

◆ We have:

$$\cos(w^{(n)}, \bar{w}) = \frac{\sum w_i^{(n)} \bar{w}_i}{\sqrt{\sum w_i^{(n)2}} \sqrt{\sum \bar{w}_i^2}} \geq \frac{\sum w_i^{(0)} \bar{w}_i + 2na}{\sqrt{\sum w_i^{(0)2} + 4nA} \sqrt{\sum \bar{w}_i^2}} \xrightarrow{n \rightarrow \infty} \infty$$

- ◆ So there may not exist an infinity of values of  $n$  for which the update occurs
- ◆ There exists some  $N$  such that there is no update for steps where  $n > N$
- ◆ After step  $N$ , all training data is correctly classified

## Non-Separable Case

- ◆ Assume now that examples are not separable
- ◆ Then no perfect perceptron exist
  - Perceptron Training is not applicable
- ◆ Does there exist a “good” perceptron ?
  - Define error as distance between the output and desired values:

$$E(w_1, w_2, \dots, w_n) = \frac{1}{2} \sum_j (t^j - y^j)^2$$

- A “good” perceptron is:

$$\min_{(w_1, w_2, \dots, w_n)} E(w_1, w_2, \dots, w_n)$$

## Remember Gradient Descent

- ◆  $f: \mathbb{R}^n \rightarrow \mathbb{R}$  differentiable

$$f(x_1, x_2, \dots, x_n) \quad df(x) = \left( \frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \dots, \frac{\partial f(x)}{\partial x_n} \right)$$

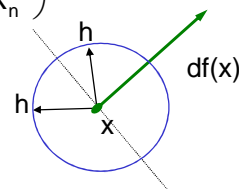
- ◆ If  $\|h\| = \varepsilon$  small:

$$f(x+h) \approx f(x) + df(x) \cdot h$$

- ◆ Minimum value for  $h = -\varepsilon \frac{df(x)}{\|df(x)\|}$

$$f(x+h) \approx f(x) - \varepsilon df(x) \cdot \frac{df(x)}{\|df(x)\|} = f(x) - \varepsilon \|df(x)\| < f(x)$$

- ◆ We can decrease  $f$  by moving in the opposite direction of the gradient



## Gradient Training

- ◆ Problem:  $y = f\left(\sum_i w_i x_i\right)$
- ◆ If  $f$  is a step function,  $f$  is not differentiable and gradient optimization cannot be applied
- ◆ Solution: replace  $f$  with linear function:  $f(t) = t$

$$E(w_1, w_2, \dots, w_n) = \frac{1}{2} \sum_j \left( t^j - \sum_i w_i x_i^j \right)^2$$

- Gradient:

$$\frac{\partial E}{\partial w_i} = - \sum_j \left( t^j - \sum_i w_i x_i^j \right) x_i^j$$

## Gradient Training

- ◆ Algorithm:
  - initialize  $w^{(0)}$   $\lambda > 0$  small
  - Iterate:

$$w_i^{(k+1)} = w_i^{(k)} - \lambda \frac{\partial E}{\partial w_i} = w_i^{(k)} + \lambda \sum_j \left( t^j - \sum_i w_i^{(k)} x_i^j \right) x_i^j$$

- ◆ Notes:
  - Since we chose  $f(t)=t$ , the update formula is

$$w_i^{(k+1)} = w_i^{(k)} + \lambda \sum_j (t^j - y^{j(k)}) x_i^j$$

- Compare with the Perceptron Training formula:

$$w_i^{(k+1)} = w_i^{(k)} + (t^j - y^{j(k)}) x_i^j$$



## Gradient Training

---

◆ Matrix form:

$W$  = vector of weights:  $W^T = (w_1, w_2, \dots, w_n)$

$X$  = vector of inputs:  $X^T = (x_1, x_2, \dots, x_n)$

$y$  = output:  $y = f(W^T X)$

$E(X)$  = output error for  $X$ , total error  $E = \sum_X E(X)$

Gradient:  $\nabla E(X)^T = \left( \frac{\partial E(X)}{\partial w_1}, \frac{\partial E(X)}{\partial w_2}, \dots, \frac{\partial E(X)}{\partial w_n} \right)$

Gradient training:

$$W \leftarrow W - \lambda \nabla E = W - \lambda \sum_X \nabla E(X)$$

## Stochastic Gradient Training

---

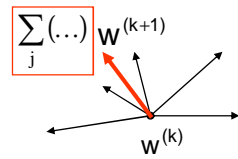
- ◆ Computing  $\sum_X \nabla E(X)$  requires to process all the training samples for one iteration
- ◆ This can be a huge computation for a single step of the training algorithm
- ◆ In Stochastic gradient, we split the training set into minibatches of size  $k$  ( $k=1$ , or  $10$ , or  $100\dots$ )
- ◆ We update:

$$W \leftarrow W - \lambda \sum_{X \in \text{minibatch}} \nabla E(X)$$

## Stochastic Gradient Training

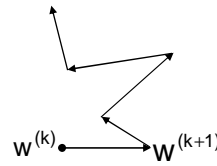
- ◆ Stochastic gradient allows for faster convergence with less computation

### ◆ Normal gradient



$$w^{(k+1)} = w^{(k)} - \lambda \sum_{x \in T} \nabla E(X)$$

### Stochastic gradient



$$w^{(k+1)} = w^{(k)} - \lambda \sum_{x \in \text{minibatch}} \nabla E(X)$$

## Training Algorithms Comparison

### ◆ Perceptron training:

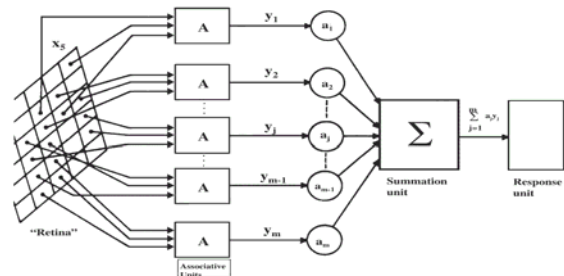
- no parameters,
- convergence in a finite number of iterations,
- BUT: works only for linearly separable problems.

### ◆ Gradient descent:

- 2 alternatives: Global vs Stochastic
- choice of parameter  $\lambda$
- converges at infinity
- works even for non linearly separable training sets

## Rosenblatt's Mark I Perceptron

- ◆ Early vision system (1957/58)
  - 20x20 sensors
  - 1 layer random association units
  - 1 perceptron layer
  - 8 outputs
- ◆ Able to automatically learn to recognize simple shapes

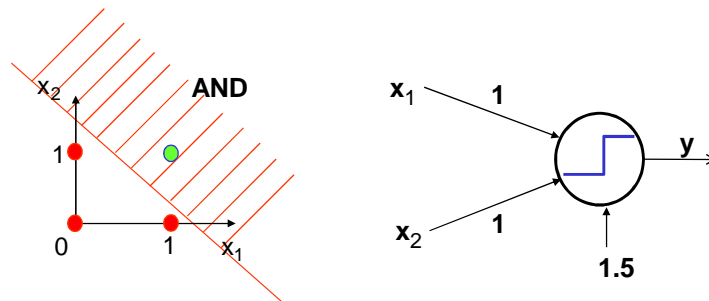


MALIS 2017

249

## Perceptron Limitations

- ◆ Perceptrons can generate AND

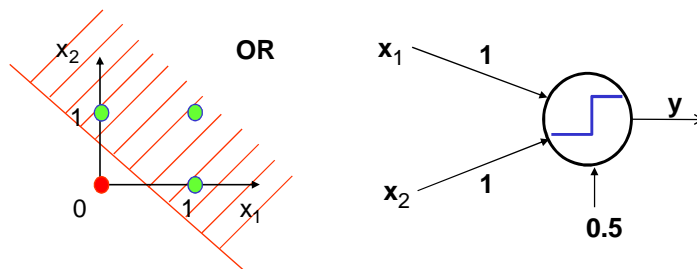


MALIS 2017

250

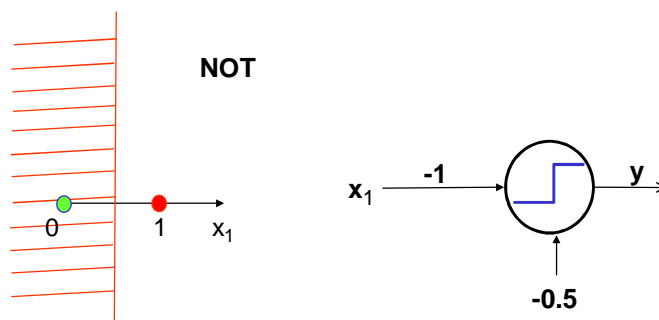
## Perceptron Limitations

- ◆ Perceptrons can generate OR



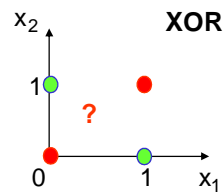
## Perceptron Limitations

- ◆ Perceptrons can generate NOT



## Perceptron Limitations

◆ Perceptrons cannot generate XOR:



$x_1$	$x_2$	XOR
0	0	0
1	0	1
0	1	1
1	1	0

## Perceptron Limitations

◆ Perceptrons cannot generate XOR:

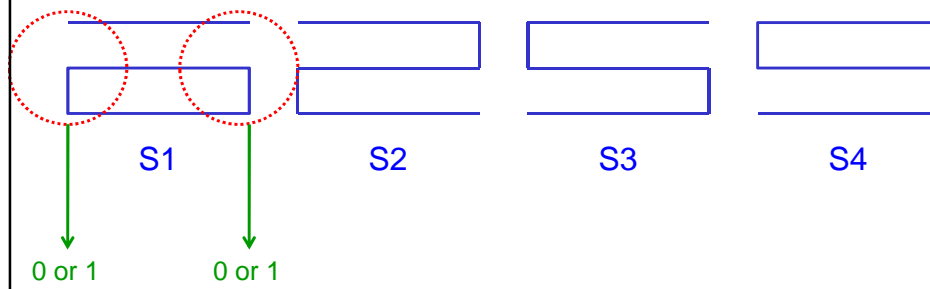
◆  $y = \text{sign}(w_1 x_1 + w_2 x_2 - b)$

- $-b \leq 0$
  - $w_1 - b > 0$
  - $w_2 - b > 0$
  - $w_1 + w_2 - b \leq 0$
- $w_1 + w_2 - 2b > 0$   
 $w_1 + w_2 - 2b \leq 0$



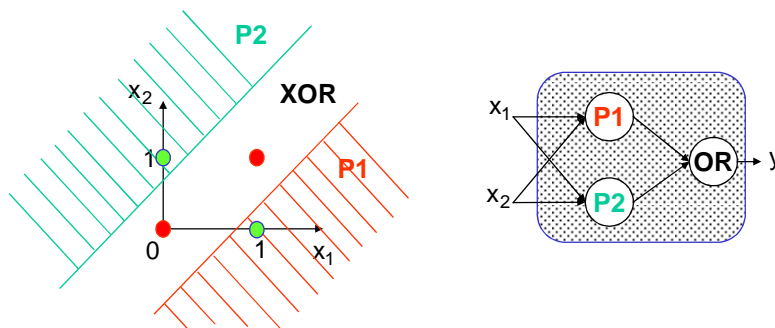
## Perceptron Limitations

- ◆ Perceptrons with bounded associative units cannot decide about shape connectedness



- ◆ This is equivalent to the XOR problem

## Perceptron Limitations

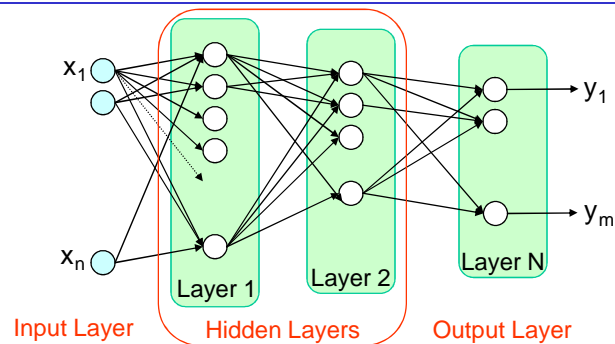


- ◆ Single Perceptrons cannot model XOR
- ◆ Combinations of perceptrons can model any boolean function (why?)

## Neural Networks

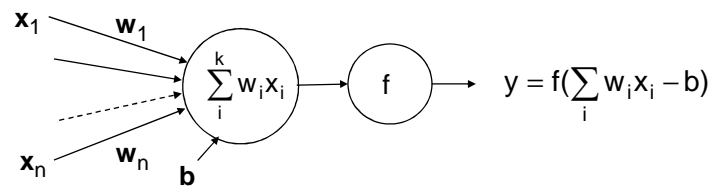
- ◆ Need for more complex networks of perceptrons (neurons): Neural Networks
- ◆ Problem: no training algorithm existed before 80'
- ◆ 80': Several types of networks:
  - Multi-Layer (also called Feed-Forward)
  - Hopfield (Boltzmann)
- ◆ Various training algorithms
- ◆ Networks can be fully connected or not

## Multi-Layer Perceptrons MLP

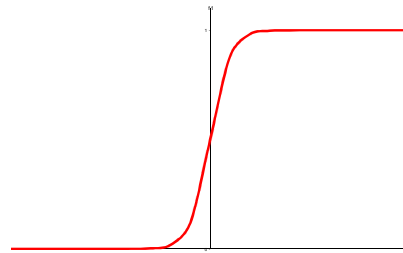


- outputs of neurons on layer  $i$  are connected to the inputs of neurons on layer  $i+1$
- most common form of Neural Net (feed-forward)

## Neurons



- ♦ differentiable activation function: sigmoid



$$f(t) = \frac{1}{1 + e^{-t}}$$

## Other activation functions

- ♦ hyperbolic tangent:

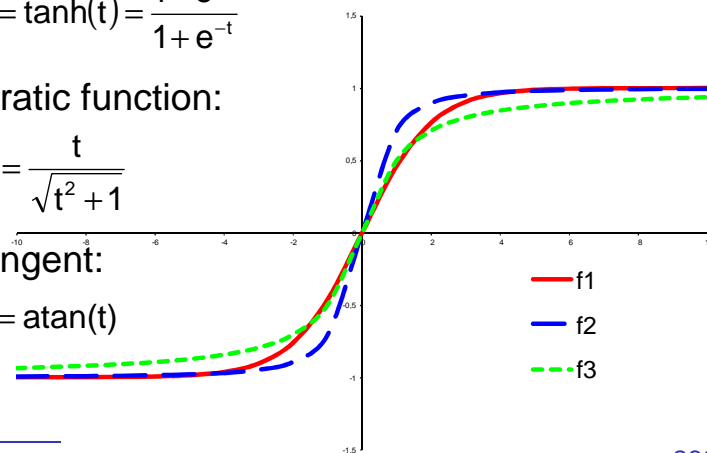
$$f_1(t) = \tanh(t) = \frac{1 - e^{-t}}{1 + e^{-t}}$$

- ♦ quadratic function:

$$f_2(t) = \frac{t}{\sqrt{t^2 + 1}}$$

- ♦ arctangent:

$$f_3(t) = \text{atan}(t)$$





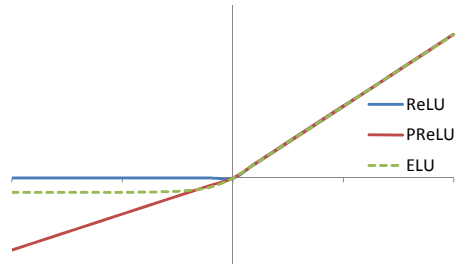
## Other activation functions

### ◆ ReLU (Rectified Linear Unit)

- $f(t) = \begin{cases} 0 & \text{for } t < 0 \\ t & \text{for } t \geq 0 \end{cases}$

### ◆ Parametric ReLU

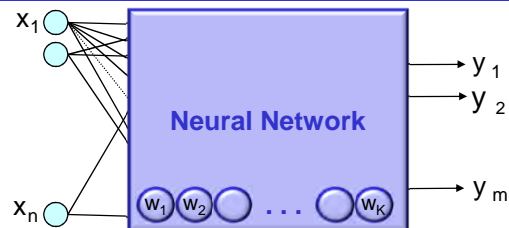
- $f(t) = \begin{cases} \alpha t & \text{for } t < 0 \\ t & \text{for } t \geq 0 \end{cases}$



### ◆ ELU (Exponential Linear Unit)

- $f(t) = \begin{cases} \alpha(e^t - 1) & \text{for } t < 0 \\ t & \text{for } t \geq 0 \end{cases}$

## NN Applications



### ◆ Application: set of examples

$(x_1^j, x_2^j, \dots, x_n^j, t_1^j, t_2^j, \dots, t_m^j) \quad j = 1, 2, \dots, T$

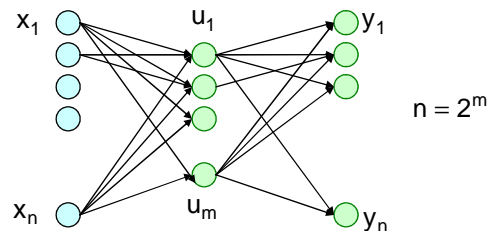
### ◆ Training: find the best values for $w_i$

If  $(x_1^j, x_2^j, \dots, x_n^j)$  as input,  $(y_1, y_2, \dots, y_m)$  close to  $(t_1^j, t_2^j, \dots, t_m^j)$

### ◆ Test: compute output for new input $x_1, x_2, \dots, x_n$

## NN Applications

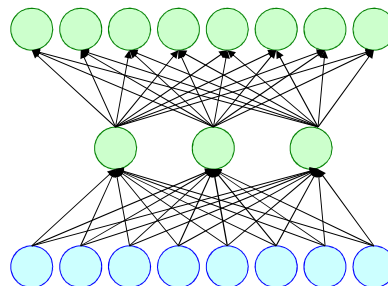
### ◆ Example: NN to invent binary representation



- $n$  samples: inputs are 0 except  $x_i=1$ , for  $i=1, 2, \dots, n$   
 $1=10000, 2=01000, 3=00100, 4=00010, \dots$
- desired output: identity  $y_i = x_i \quad \forall i$
- $u_i$  should code binary representation of input

## Can This Be Learned?

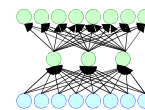
Input X		Output Y
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001



## Learned Hidden Layer Representation

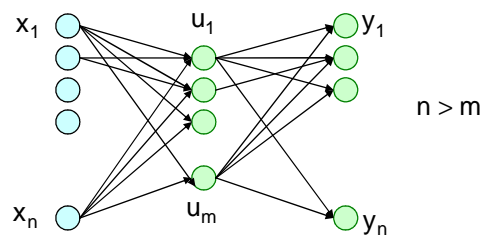
### ◆ Experimental result:

Input X		$u_1$ $u_2$ $u_3$		Output Y
10000000	→	.89 .04 .08	→	10000000
01000000	→	.01 .11 .88	→	01000000
00100000	→	.01 .97 .27	→	00100000
00010000	→	.99 .97 .71	→	00010000
00001000	→	.03 .05 .02	→	00001000
00000100	→	.22 .99 .99	→	00000100
00000010	→	.80 .01 .98	→	00000010
00000001	→	.60 .94 .01	→	00000001



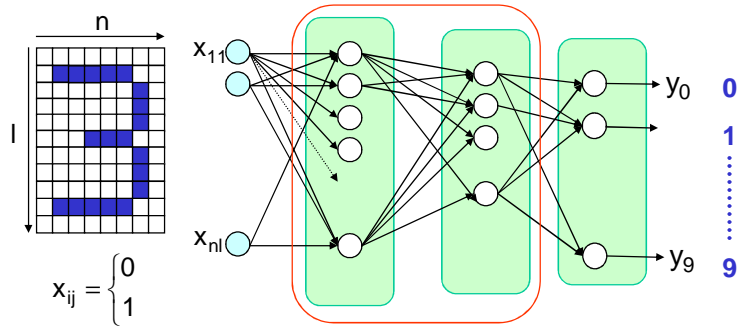
## NN Applications

### ◆ Encoder-decoder:



- desired output: identity  $y_i = x_i \quad \forall i$
- $u_i$  extracts relevant features of input samples
- use to reduce data dimensionality (aka **PCA**)

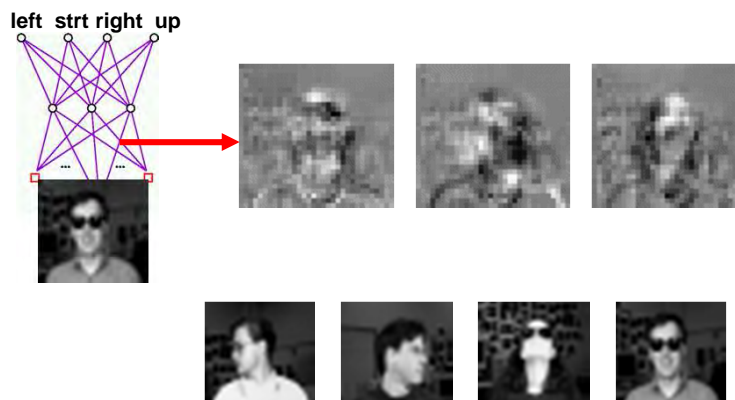
## NN Character Recognition



- Input: binary image pattern
- Output: character number
- Recognition:  $\operatorname{argmax}_{i=1 \dots m, y_i > \theta} y_i$

## Face Orientation

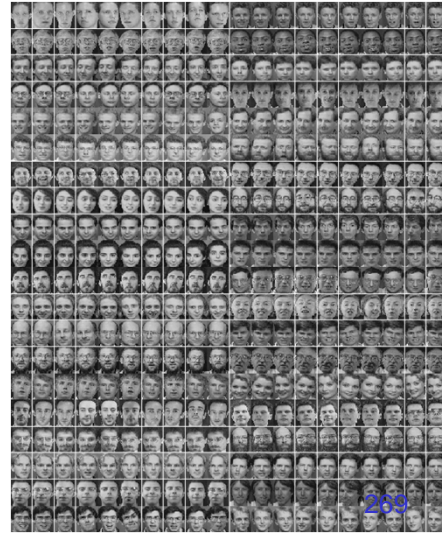
- ◆ Head pose: left, straight, right, up



Typical input images

## Face Recognition

- ◆ using face database

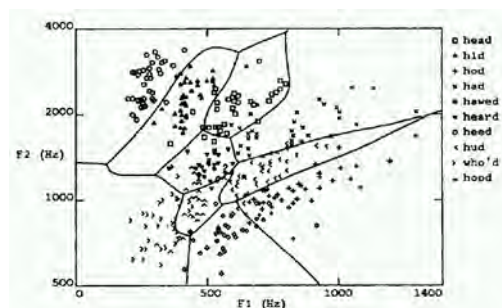
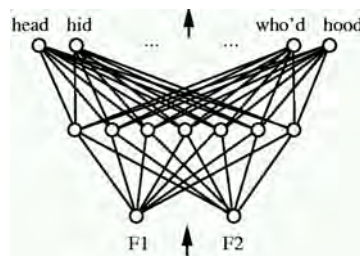


MALIS 2017

269

## Speech Recognition

- ◆ Vowel classification based on formants F1 and F2



[Haung/Lippman 1988]

MALIS 2017

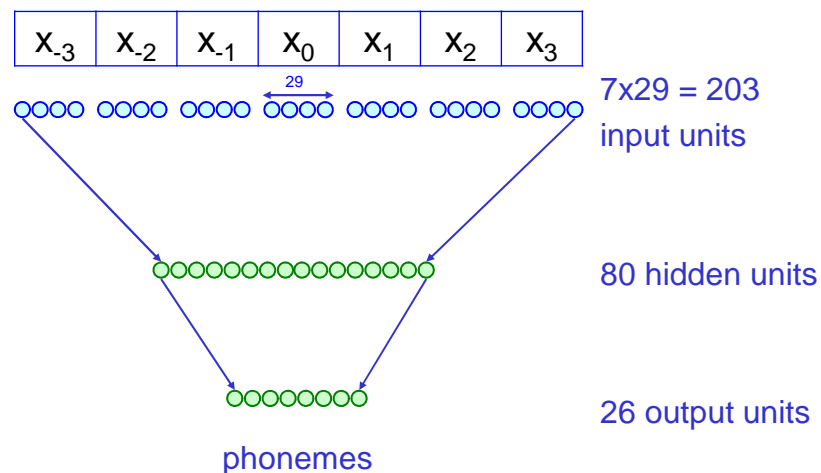
270

## NetTalk

### ◆ Talking Network: read sentences

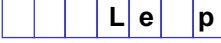
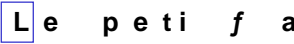
- input: character sequence
  - 26 characters + punctuation (total 29)
  - current character + context of 3 left-right
- output: phonetic parameters to speech synthesizer
  - ~ 30 phonemes
  - 26 characteristics (tongue, mouth position...)
  - 1 phoneme = 1 characteristics vector

## NetTalk: Structure



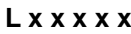


## NetTalk: Usage

### ◆ Training:

- Input: orthographic 
- Output: phoneme 
- Train weights

### ◆ Speaking:

- Input: sentence 
- Compute activations 
- Output: phoneme 

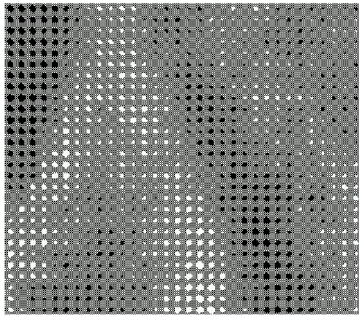
## ALVINN

- ◆ Autonomous Land Vehicle In a Neural Network
- ◆ CMU project 1998
- ◆ Purpose: drive standard vehicle on public highways
- ◆ Learning by observing human driving

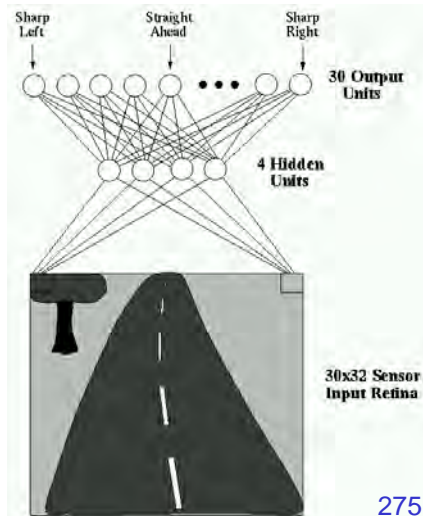


## ALVINN

- ◆ input:
  - 30x32 from video camera
- ◆ output: steering control



MALIS 2017



275

## ALVINN

- ◆ Learning: about 5 minutes of human driving
- ◆ Test: successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways



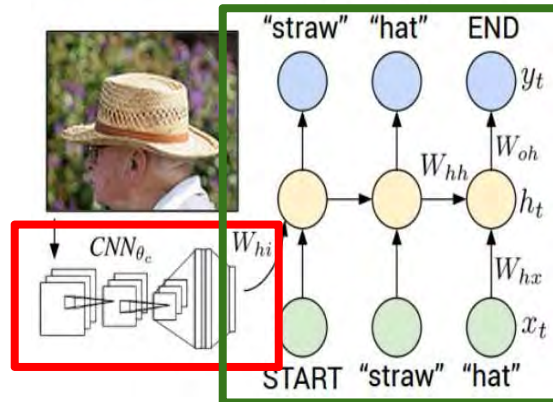
MALIS 2017

276



## NeuralTalk

### ◆ (Stanford 2015) Recurrent Neural Network



## Convolutional Neural Network

MALIS 2017

277

 <p>"man in black shirt is playing guitar."</p>	 <p>"construction worker in orange safety vest is working on road."</p>	 <p>"two young girls are playing with lego toy."</p>	 <p>"boy is doing backflip on wakeboard."</p>
 <p>"girl in pink dress is jumping in air."</p>	 <p>"black and white dog jumps over bar."</p>	 <p>"young girl in pink shirt is swinging on swing."</p>	 <p>"man in blue wetsuit is surfing on wave."</p>

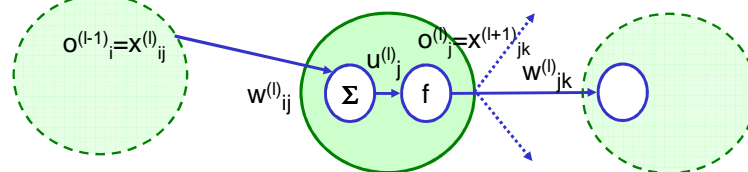
MALIS 2017

278

## MLP Feed-forward Computation

◆ For neuron  $j$  of layer  $l$ :

- Inputs:  $x_{ij}^{(l)}$  from neuron  $i$  at level  $l-1$
- Activation:  $u_j^{(l)}$
- Output:  $o_j^{(l)}$  to all neurons at level  $l+1$

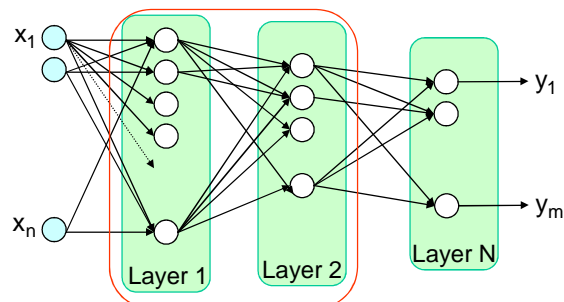


- Inputs:  $x_{ij}^{(l)} = o^{(l-1)}_i$
- Activation:  $u_j^{(l)} = \sum_i w_{ij}^{(l)} x_{ij}^{(l)}$
- Output:  $o_j^{(l)} = f(u_j^{(l)})$

## MLP Feed-forward Computation

◆ Assume MLP is given (layers, weights)

- Assume inputs  $(x_1, x_2, \dots, x_n)$  are known
- How to compute output values  $(y_1, \dots, y_m)$  ?



## MLP Feed-forward Computation

### ◆ For layer 1:

- Inputs:  $x^{(1)}_{ij} = x_i$
- Activation:  $u^{(1)}_j = \sum_i w^{(1)}_{ij} x^{(1)}_{ij}$
- Output:  $o^{(1)}_j = f(u^{(1)}_j)$

### ◆ For layer 2:

- Inputs:  $x^{(2)}_{jk} = o^{(1)}_j$
- Activation:  $u^{(2)}_k = \sum_j w^{(2)}_{jk} x^{(2)}_{jk}$
- Output:  $o^{(2)}_k = f(u^{(2)}_k)$

◆ ...

### ◆ For layer N:

- Inputs:  $x^{(N)}_{kh} = o^{(N-1)}_k$
- Activation:  $u^{(N)}_h = \sum_k w^{(N)}_{kh} x^{(N)}_{kh}$
- Output:  $y_h = o^{(N)}_h = f(u^{(N)}_h)$

## MLP Training

### ◆ Training set:

- $(x^j_1, x^j_2, \dots, x^j_n, t^j_1, t^j_2, \dots, t^j_m) \quad j = 1, 2, \dots, T$

### ◆ Assume NN structure is given

- layers, neurons per layer, possible connections

### ◆ Training:

- find best values of weights  $w^{(l)}_{ij}$

- best = minimize error:

- error for sample j:  $E_j = \sum_k (y^j_k - t^j_k)^2$

- total error:  $E = \sum_j E_j = \sum_j \sum_k (y^j_k - t^j_k)^2$

## MLP Training

- ◆ Find  $\min_{w^{(l)}_{ij}} E(\{w^{(l)}_{ij}\})$
- ◆ Use gradient descent:
  - initialize  $w^{(l)}_{ij}$  randomly
  - iterate  $w^{(l)}_{ij} \rightarrow w^{(l)}_{ij} - \lambda \frac{\partial E}{\partial w^{(l)}_{ij}}$  with  $\lambda > 0$
  - stop when (for example):
    - max number of iteration is reached
    - max duration is elapsed
    - not enough improvement  $\Delta E < \theta$

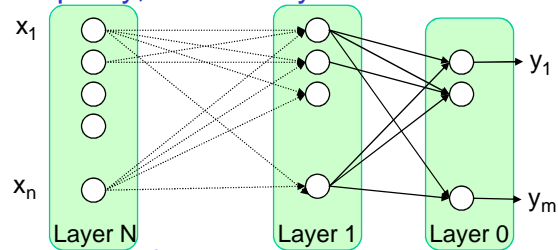
## MLP Training

- ◆ Problem: how to compute  $\frac{\partial E}{\partial w^{(l)}_{ij}}$  ?
- ◆ Analytical formulation: too complex
 
$$E = \sum_j \sum_k (y^j_k - t^i_k)^2$$

$\downarrow$   
 $y^j_k = o^{(N)}_k = f(u^{(N)}_k) = f\left(\sum_h w^{(N)}_{hk} x^{(N)}_{hk}\right)$   
 $\downarrow$   
 $x^{(N)}_{hk} = o^{(N-1)}_h = f(u^{(N-1)}_h) = f\left(\sum_i w^{(N-1)}_{ih} x^{(N-1)}_{ih}\right)$   
 $\downarrow \dots$
- ◆ Intractable problem until a numerical solution was found: the Back-Propagation algorithm

## Back-Propagation Notations

- For simplicity, number layers in reverse order:



- For neuron  $j$  of layer  $l$ :
  - Inputs:  $x_{ij}^{(l)}$ , activation:  $u_j^{(l)}$ , output  $o_j^{(l)}$
  - Equation:  $u_j^{(l)} = \sum_i w_{ij}^{(l)} x_{ij}^{(l)}$
- Network:
  - Inputs:  $x_i = x_{ij}^{(N)}$
  - Outputs:  $y_j = o_j^{(0)}$

## Reminder

- Differential of compound functions:

$$\diamond z = f(y) \quad y = g(x) \quad \frac{dz}{dx} = \frac{df}{dy} \frac{dg}{dx}$$

$$\diamond z = f(y_1, y_2) \quad y_1 = g_1(x) \quad y_2 = g_2(x)$$

$$\frac{dz}{dx} = \frac{\partial f}{\partial y_1} \frac{dg_1}{dx} + \frac{\partial f}{\partial y_2} \frac{dg_2}{dx}$$

$$\diamond z = f(y_1, y_2, \dots, y_n) \quad y_i = g_i(x)$$

$$\frac{dz}{dx} = \sum_i \frac{\partial f}{\partial y_i} \frac{dg_i}{dx}$$

## Back-Propagation Algorithm

◆ Consider one sample:

- Inputs  $(x_1, x_2, \dots, x_n)$
- Expected outputs  $(t_1, t_2, \dots, t_m)$
- Real outputs  $(y_1, y_2, \dots, y_m)$

◆ Error for one sample

$$E = \sum_k (y_k - t_k)^2$$

- $y_k$  is a function of the weights  $\{w_{ij}\}$
- we can change the weights to minimize the error

$$w_{ij}^{(0)} \rightarrow w_{ij}^{(0)} - \lambda \frac{\partial E}{\partial w_{ij}^{(0)}}$$

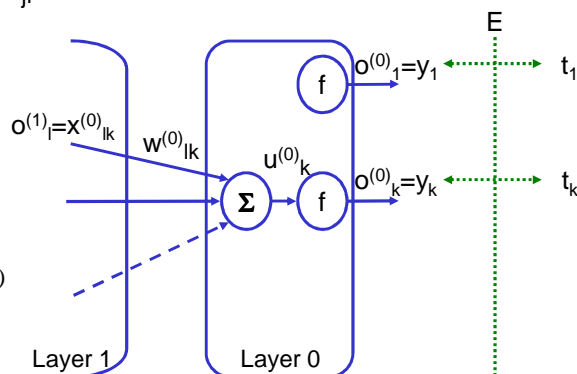
## Back-Propagation: Layer 0

◆ Computing  $\frac{\partial E}{\partial w_{ji}^{(0)}}$

$$E = \phi(y_1, y_2, \dots, y_m)$$

$$y_k = \psi(u_k^{(0)})$$

$$u_k^{(0)} = \zeta(w_{1k}^{(0)}, w_{2k}^{(0)}, \dots, w_{lk}^{(0)})$$



## Back-Propagation: Layer 0

### ◆ Computing $\frac{\partial E}{\partial w_{ji}^{(0)}}$

- E is a function of all  $y_k$

$$E = \sum_k (y_k - t_k)^2 \quad \frac{\partial E}{\partial w_{ji}^{(0)}} = \sum_k \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial w_{ji}^{(0)}} \quad \frac{\partial E}{\partial y_k} = 2(y_k - t_k)$$

- $y_k$  is a function of  $u_k^{(0)}$  only

$$y_k = f(u_k^{(0)}) \quad \frac{\partial y_k}{\partial w_{ji}^{(0)}} = \frac{\partial y_k}{\partial u_k^{(0)}} \frac{\partial u_k^{(0)}}{\partial w_{ji}^{(0)}} \quad \frac{\partial y_k}{\partial u_k^{(0)}} = f'(u_k^{(0)})$$

- $u_k^{(0)}$  is a function of  $w_{lk}^{(0)}$  for all l

$$u_k^{(0)} = \sum_l w_{lk}^{(0)} o_l^{(1)} \quad \frac{\partial u_k^{(0)}}{\partial w_{ji}^{(0)}} = 0 \text{ if } k \neq i$$

$$\frac{\partial u_i^{(0)}}{\partial w_{ji}^{(0)}} = o_j^{(1)}$$

## Back-Propagation: Layer 0

### ◆ Computing $\frac{\partial E}{\partial w_{ji}^{(0)}}$

$$\frac{\partial E}{\partial w_{ji}^{(0)}} = \sum_k \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial u_k^{(0)}} \frac{\partial u_k^{(0)}}{\partial w_{ji}^{(0)}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial u_i^{(0)}} \frac{\partial u_i^{(0)}}{\partial w_{ji}^{(0)}}$$

- So:

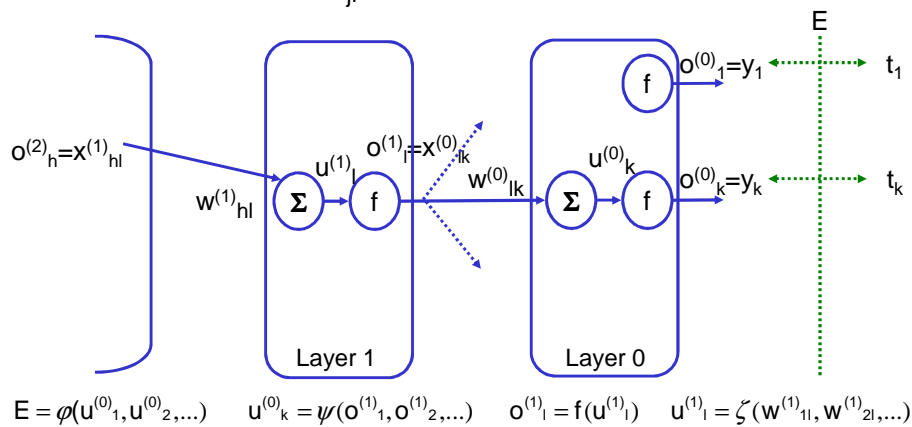
$$\frac{\partial E}{\partial w_{ji}^{(0)}} = 2(y_i - t_i) f'(u_i^{(0)}) o_j^{(1)}$$

- Note that:

$$\frac{\partial E}{\partial u_i^{(0)}} = \sum_k \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial u_i^{(0)}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial u_i^{(0)}} = 2(y_i - t_i) f'(u_i^{(0)})$$

## Back-Propagation: Layer 1

### ◆ Computing $\frac{\partial E}{\partial w^{(1)}_{ji}}$



## Back-Propagation: Layer 1

### ◆ Computing $\frac{\partial E}{\partial w^{(1)}_{ji}}$

$$E = \text{fct}(u^{(0)}_1, u^{(0)}_2, \dots) \quad u^{(0)}_k = \text{fct}(o^{(1)}_1, o^{(1)}_2, \dots) \quad o^{(1)}_i = f(u^{(1)}_i) \quad u^{(1)}_i = \text{fct}(w^{(1)}_{1i}, w^{(1)}_{2i}, \dots)$$

$$\begin{aligned}
 \frac{\partial E}{\partial w^{(1)}_{ji}} &= \sum_l \frac{\partial E}{\partial u^{(1)}_l} \frac{\partial u^{(1)}_l}{\partial w^{(1)}_{ji}} = \frac{\partial E}{\partial u^{(1)}_i} \frac{\partial u^{(1)}_i}{\partial w^{(1)}_{ji}} &= \sum_k \frac{\partial E}{\partial u^{(0)}_k} w^{(0)}_{ik} f'(u^{(1)}_i) o^{(2)}_j \\
 &\downarrow &\uparrow \\
 \frac{\partial E}{\partial u^{(1)}_i} &= \sum_k \frac{\partial E}{\partial u^{(0)}_k} \frac{\partial u^{(0)}_k}{\partial u^{(1)}_i} &= \sum_k \frac{\partial E}{\partial u^{(0)}_k} w^{(0)}_{ik} f'(u^{(1)}_i) \\
 &\downarrow &\uparrow \\
 \frac{\partial u^{(0)}_k}{\partial u^{(1)}_i} &= \sum_l \frac{\partial u^{(0)}_k}{\partial o^{(1)}_l} \frac{\partial o^{(1)}_l}{\partial u^{(1)}_i} = \sum_l w^{(0)}_{lk} \frac{\partial o^{(1)}_l}{\partial u^{(1)}_i} = w^{(0)}_{ik} \frac{\partial o^{(1)}_i}{\partial u^{(1)}_i} = w^{(0)}_{ik} f'(u^{(1)}_i)
 \end{aligned}$$



## Back-Propagation: Layer 1

◆ Computing  $\frac{\partial E}{\partial w_{ji}^{(1)}}$

- The main idea is to note that:

$$\frac{\partial E}{\partial u_j^{(1)}} = \sum_k \frac{\partial E}{\partial u_k^{(0)}} w_{ik}^{(0)} f'(u_i^{(1)})$$

- And then:

$$\frac{\partial E}{\partial w_{ji}^{(1)}} = \frac{\partial E}{\partial u_j^{(1)}} o_j^{(2)}$$

- ◆ The same idea can be applied to layers 2, 3, ...

## Back-Propagation: Sigmoid Case

- ◆ If  $f(t) = \frac{1}{1 + e^{-t}}$        $f'(t) = f(t)[1 - f(t)]$        $f'(u_i^{(k)}) = o_i^{(k)}[1 - o_i^{(k)}]$

- ◆ at layer 0:

$$\frac{\partial E}{\partial u_i^{(0)}} = 2(y_i - t_i) o_i^{(0)} [1 - o_i^{(0)}]$$

$$\frac{\partial E}{\partial w_{ji}^{(0)}} = 2(y_i - t_i) o_i^{(0)} [1 - o_i^{(0)}] o_j^{(1)}$$

- ◆ at layer 1:

$$\frac{\partial E}{\partial w_{ji}^{(1)}} = \sum_k \frac{\partial E}{\partial u_k^{(0)}} w_{ik}^{(0)} o_i^{(1)} [1 - o_i^{(1)}] o_j^{(2)}$$

## Back-Propagation Training

- ◆ Initialize with random weights
- ◆ Iterate Back-propagation:
  - Feed-forward pass to compute activations and outputs
  - Back-propagation pass to compute partial derivatives

(be careful about layer numbering)

$$\frac{\partial E}{\partial u_i^{(0)}} = 2(y_i - t_i) f'(u_i^{(0)})$$

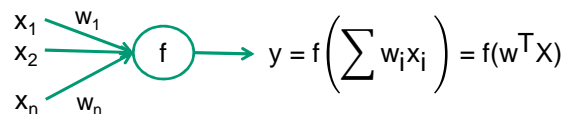
$$\frac{\partial E}{\partial u_i^{(l+1)}} = \sum_k \frac{\partial E}{\partial u_k^{(l)}} \cdot w_{ik}^{(l)} f'(u_i^{(l+1)})$$

$$\frac{\partial E}{\partial w_{ji}^{(l)}} = \frac{\partial E}{\partial u_i^{(l)}} o_j^{(l+1)}$$

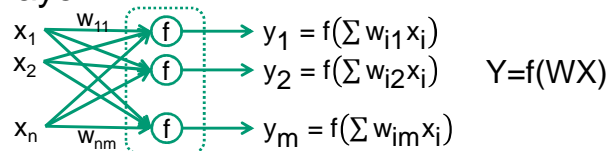
- Update  $w_{ij}^{(l)} \rightarrow w_{ij}^{(l)} - \lambda \frac{\partial E}{\partial w_{ij}^{(l)}}$

## Matrix Form

- ◆ Neuron:



- ◆ Layer



- ◆ Network

$$X \rightarrow Y_1 = f(W_1 X) \rightarrow Y_2 = f(W_2 Y_1) \dots \rightarrow Y_l = f(W_l Y_{l-1})$$

## Matrix Form

- ◆ Back-propagation:

$$dY = \left( \frac{\partial E}{\partial y_1}, \frac{\partial E}{\partial y_2}, \dots, \frac{\partial E}{\partial y_n} \right)^T$$

- ◆ Matrix form:

$$\begin{cases} y=f(z) & \frac{\partial E}{\partial z} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial z} = \frac{\partial E}{\partial y} f'(z) \\ Y = f(Z) & dZ = f'(Z) \circ dY \end{cases}$$

*( $\circ$  is the term-by-term matrix multiplication)*

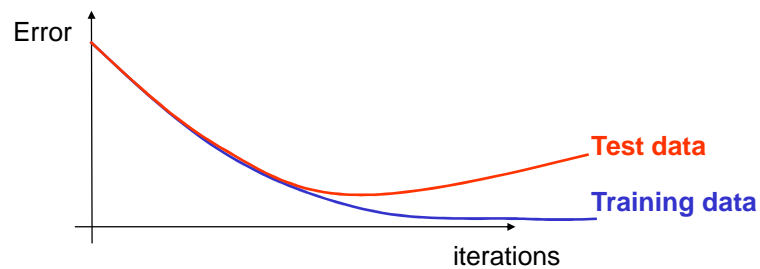
$$\begin{cases} z = \sum w_i x_i & \frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial z} \frac{\partial z}{\partial w_i} = \frac{\partial E}{\partial z} x_i \\ Z = WX & dW = dZ X^T \quad dX = W^T dZ \end{cases}$$

## Back-Propagation Training

- ◆ Converges to a local minimum
  - Sometimes, try several random initial points and keep best result overall
- ◆ When to stop iterations ?
  - Until maximum number of iterations
  - Until error improvement is small
  - Until maximum time is reached
- ◆ Problem : overfitting
  - If training size is insufficient with regard to model parameters, the model may overfit to the training data and not generalize well to other test data

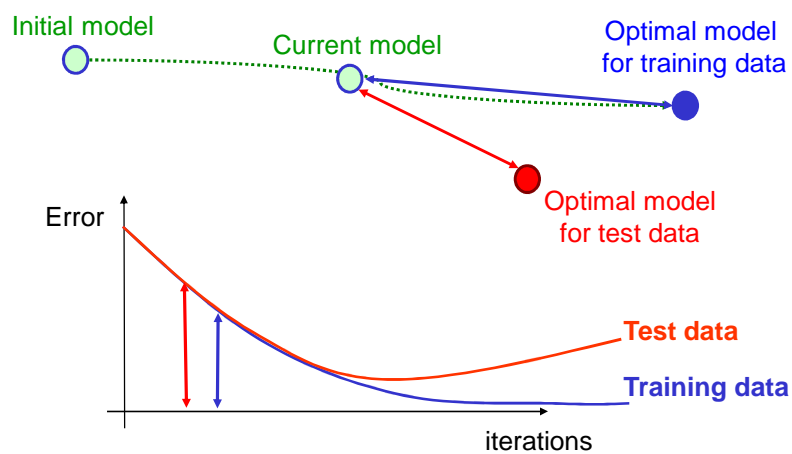
## Overfitting

- ◆ During training:
  - Error on training set always improves
  - Error on test set does not necessarily
- ◆ After some time, the model may overfit



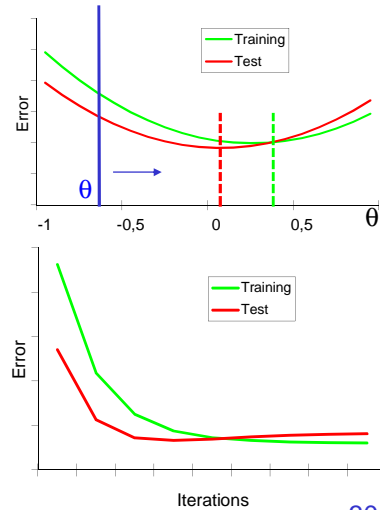
## Overfitting

- ◆ Intuitive explanation:



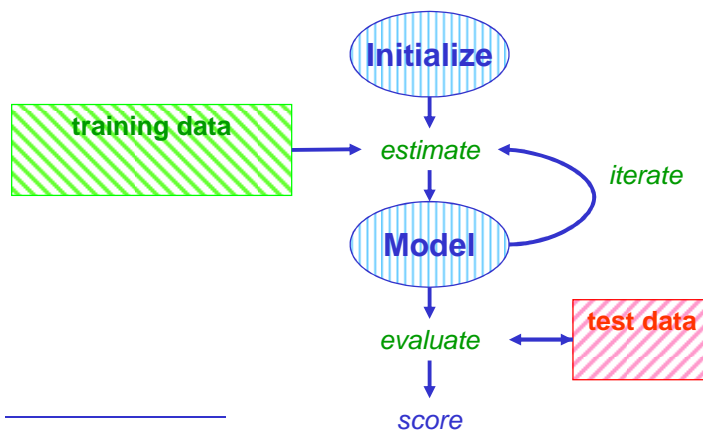
## Overfitting Example

Linear classifier  $x_1 > \theta$ ?



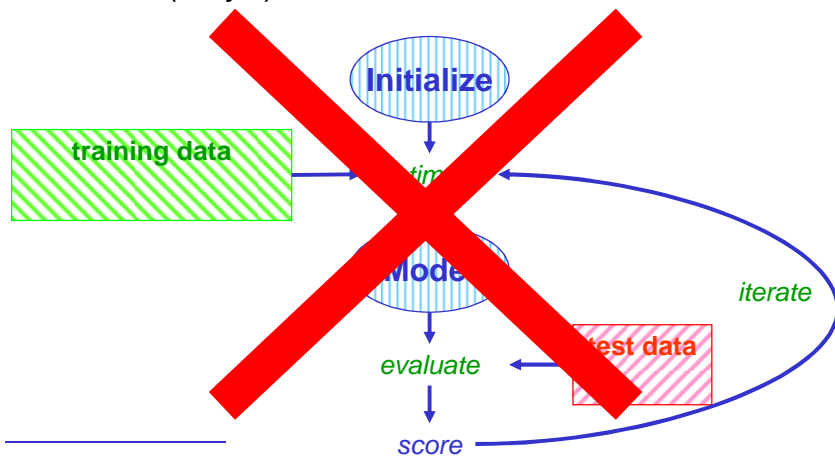
## How to avoid overfitting ?

- ◆ Training data: to estimate model parameters
- ◆ Test data: to evaluate model performance



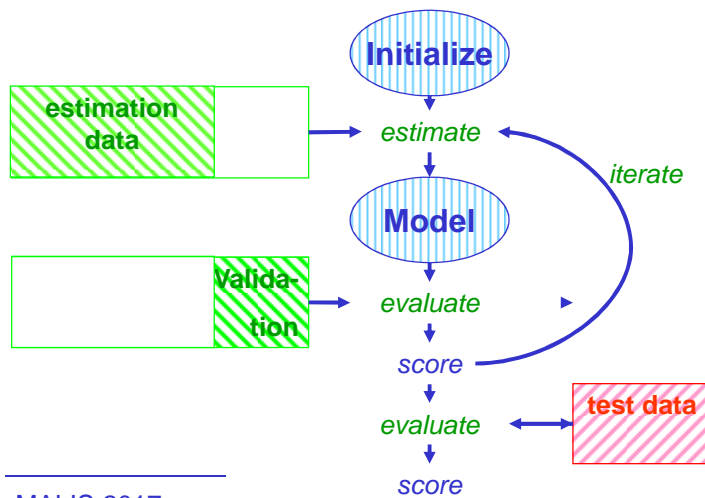
## How to avoid overfitting ?

- ◆ **Bad solution:** iterate to find best score on test data (why?)



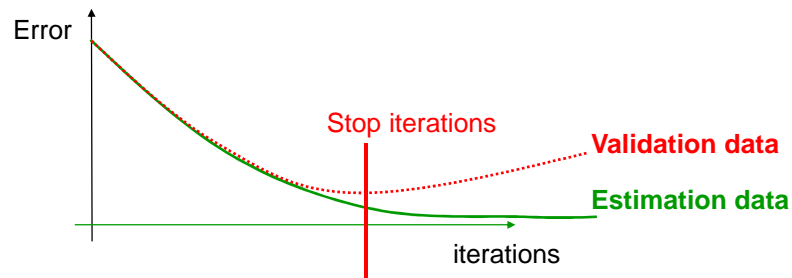
## How to avoid overfitting ?

- ◆ **Good solution:** check on validation data



## Overfitting

- Split the training data into estimation data and validation data
- At each iteration, modify model based on estimation data and validate on validation data
- Stop when improvement on validation is not enough

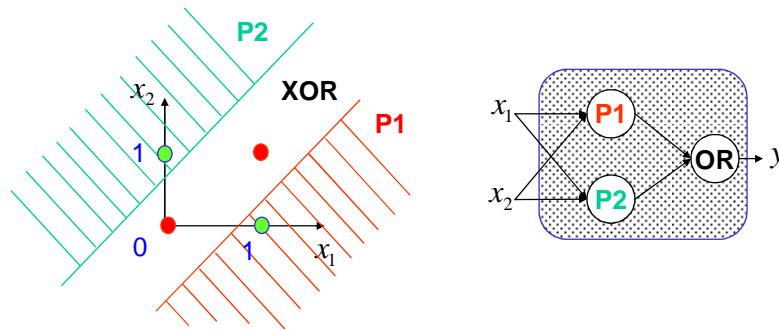


## Overfitting Summary

- ◆ Split training data into:
  - estimation data
  - validation data
- ◆ Estimate model from **estimation** data
- ◆ Compute performance on **validation** data
- ◆ Iterate as long as performance on **validation** data improves (enough)
- ◆ Model performance is the score on **test** data

## MLP Potential

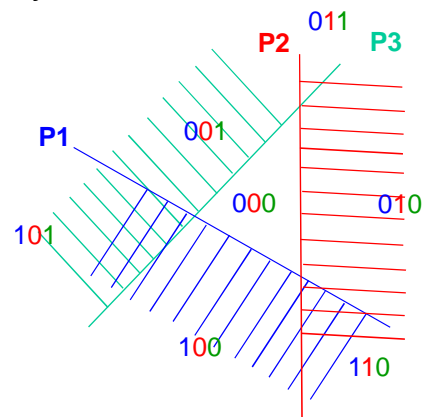
- ◆ MLP can model XOR



- ◆ and much more...

## MLP Intuitive Potential

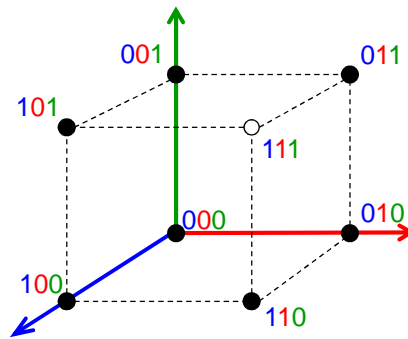
- ◆ First layer:





## MLP Intuitive Potential

◆ Second layer:



## MLP Intuitive Potential

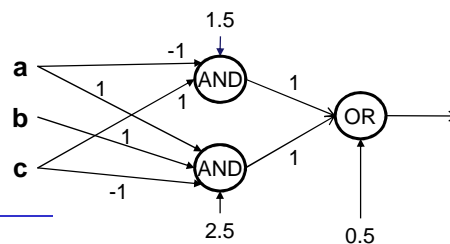
No hidden layer			Half-space
One hidden layer			Convex sets (intersections of half-spaces)
Two hidden layers			Concave and non-connex sets (union of intersections of half-spaces)

## Representative Power of NN

- ◆ Boolean functions:
  - can be represented with one hidden layer (but number of neurons can be very large)
- ◆ Continuous functions (Cybenko 1989) :
  - can be approximated with one hidden layer (with sigmoids on hidden layer and linear output)
  - (but two hidden layers maybe more efficient)
- ◆ Classifiers:
  - measurable partition  $\{P_i\}$ ,  $g(x) = i$  iff  $x \in P_i$
  - $g$  can be approximated with one hidden layer

## Boolean Functions

- ◆ 1 hidden layer is sufficient to represent any boolean function:
    - Use disjunctive normal form  $(\bar{a} \wedge c) \vee (a \wedge b \wedge \bar{c})$
    - For each term, create a hidden neuron for AND
- $$\bar{a} \wedge c \Leftrightarrow -a + c \geq 2$$
- $$a \wedge b \wedge \bar{c} \Leftrightarrow a + b - c \geq 3$$
- Create one output neuron for the main OR



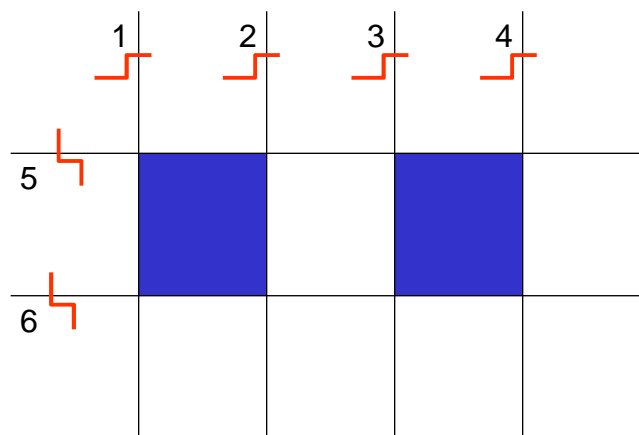
## One hidden layer example

- ◆ Find NN with output:



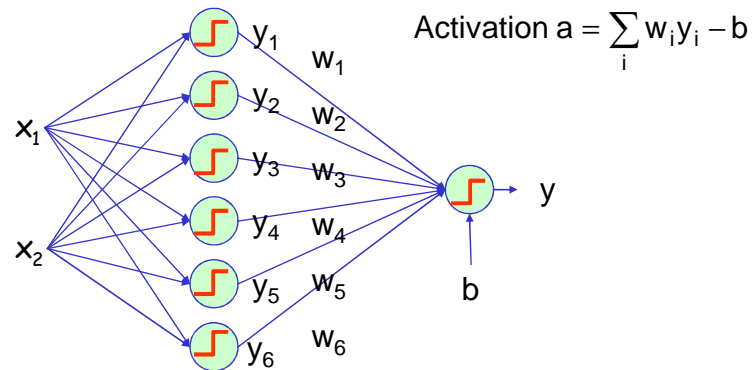
## One hidden layer example

- ◆ Consider delimiting hyperplanes:



## One hidden layer example

- ◆ Build one hidden layer network:



## One hidden layer example

- ◆ Values of activation:

	1	2	3	4	
	-	-	-	-	-
5	-	+	-	+	-
6	-	-	-	-	-

## One hidden layer example

$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$a$
0	0	0	0	0	0	-
1	0	0	0	0	0	-
1	1	0	0	0	0	-
1	1	1	0	0	0	-
1	1	1	1	0	0	-
0	0	0	0	1	0	-
1	0	0	0	1	0	+
1	1	0	0	1	0	-
1	1	1	0	1	0	+
1	1	1	1	1	0	-
0	0	0	0	1	1	-
1	0	0	0	1	1	-
1	1	0	0	1	1	-
1	1	1	0	1	1	-
1	1	1	1	1	1	-

## One hidden layer example

- ◆  $-b < 0$
- ◆  $w_1$   $-b < 0$
- ◆  $w_1 + w_2$   $-b < 0$
- ◆  $w_1 + w_2 + w_3$   $-b < 0$
- ◆  $w_1 + w_2 + w_3 + w_4$   $-b < 0$
- ◆  $+ w_5$   $-b < 0$
- ◆  $w_1$   $+ w_5$   $-b \geq 0$
- ◆  $w_1 + w_2$   $+ w_5$   $-b < 0$
- ◆  $w_1 + w_2 + w_3$   $+ w_5$   $-b \geq 0$
- ◆  $w_1 + w_2 + w_3 + w_4$   $+ w_5$   $-b < 0$
- ◆  $+ w_5 + w_6$   $-b < 0$
- ◆  $w_1$   $+ w_5 + w_6$   $-b < 0$
- ◆  $w_1 + w_2$   $+ w_5 + w_6$   $-b < 0$
- ◆  $w_1 + w_2 + w_3$   $+ w_5 + w_6$   $-b < 0$
- ◆  $w_1 + w_2 + w_3 + w_4$   $+ w_5 + w_6$   $-b < 0$

## One hidden layer example

♦ For example, choose:

♦		$-b = -3$	$b = 3$
♦	$w_1$	$-b = -1$	$w_1 = 2$
♦	$w_1 + w_2$	$-b = -3$	$w_2 = -2$
♦	$w_1 + w_2 + w_3$	$-b = -1$	$w_3 = 2$
♦	$w_1 + w_2 + w_3 + w_4$	$-b = -3$	$w_4 = -2$
♦	$w_5 = 2$	$+w_5$	$-b = -1 < 0$
♦	$w_1$	$+w_5$	$-b = +1 \geq 0$
♦	$w_1 + w_2$	$+w_5$	$-b = -1 < 0$
♦	$w_1 + w_2 + w_3$	$+w_5$	$-b = +1 \geq 0$
♦	$w_1 + w_2 + w_3 + w_4$	$+w_5$	$-b = -1 < 0$
♦	$w_6 = -2$	$+w_5 + w_6$	$-b = -3 < 0$
♦	$w_1$	$+w_5 + w_6$	$-b = -1 < 0$
♦	$w_1 + w_2$	$+w_5 + w_6$	$-b = -3 < 0$
♦	$w_1 + w_2 + w_3$	$+w_5 + w_6$	$-b = -1 < 0$
♦	$w_1 + w_2 + w_3 + w_4$	$+w_5 + w_6$	$-b = -3 < 0$

## Cybenko's weak theorem

- ♦ Let  $f$  a continuous function  $[-1, +1]^n \rightarrow \mathbb{R}$
- ♦  $\forall \varepsilon > 0 \exists N$  MLP neural net with 2 hidden layers, s.t.

$$\forall x \in [-1, +1]^n \quad |f(x) - N(x)| < \varepsilon$$

♦ Proof:

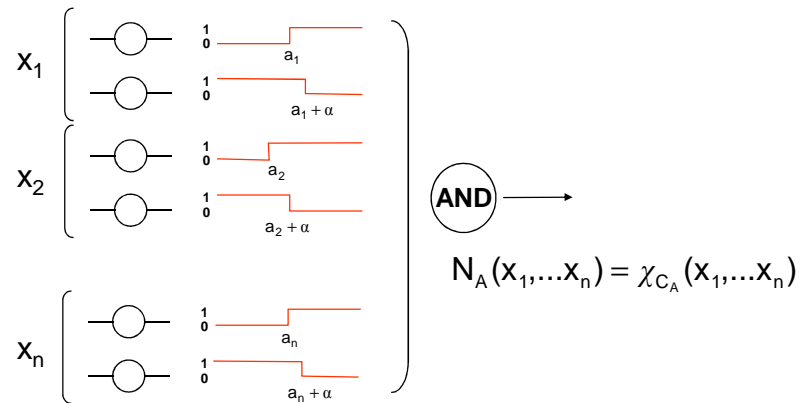
- let  $A = (a_1, a_2, \dots, a_n)$
- let  $C_A(\alpha)$  be the hypercube

$$[a_1, a_1 + \alpha] \times [a_2, a_2 + \alpha] \times \dots \times [a_n, a_n + \alpha]$$

- we can partition  $[-1, +1]^n$  with such hypercubes  $C_A(\alpha)$

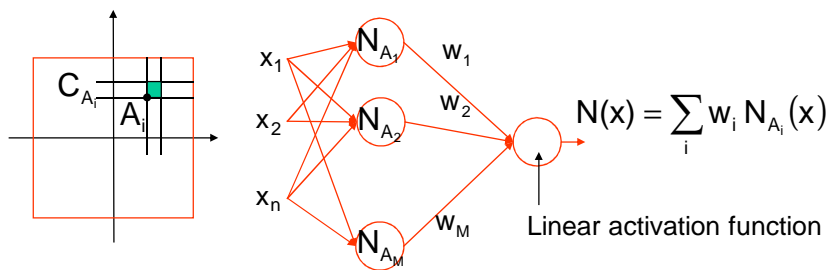
## Cybenko's weak Theorem

- we can construct the characteristic function of  $C_A$  as a neural net  $N_A$ :



## Cybenko's weak Theorem

- ◆ Build a compound Neural Net  $N$ :



- ◆ if  $x \in C_{A_i}$  then  $N(x) = w_i$
- ◆ We choose  $w_i = f(A_i)$ , so  $N(x) = f(A_i)$

## Cybenko's weak Theorem

---

- ◆  $f$  is uniformly continuous:  
$$\forall \varepsilon > 0 \exists \alpha > 0 \forall x_0, x \in [-1, +1]^n \|x - x_0\|_\infty < \alpha \Rightarrow |f(x) - f(x_0)| < \varepsilon$$
- ◆ Let  $x \in [-1, +1]^n$ 
  - $x$  belongs to one hypercube  $\exists i : x \in C_{A_i}$
  - by construction  $\|x - A_i\|_\infty < \alpha$
  - so  $|f(x) - N(x)| = |f(x) - f(A_i)| < \varepsilon$
- ◆ the network  $N$  approximates  $f$  with an accuracy of  $\varepsilon$
- ◆  $N$  has two hidden layers

Note:  $\|x\|_\infty = \max_i |x_i|$

## Summary on MLP

---

- ◆ Advantages
  - Very general, can be applied in many situations
  - Neural interpretation
  - Powerful according to theory
  - Efficient according to practice
- ◆ Drawbacks
  - Training is often slow
  - Choice of optimal number of layers & neurons difficult
  - Little understanding of real model



## Deep Learning Networks

- ◆ 2009 DN wins Character Recognition competition
- ◆ 2011 DN wins Traffic Sign Recognition competition
- ◆ 2011 Microsoft DN breakthrough in Speech Recognition
- ◆ 2012 Google DN recognizes Cats in images (unsupervised)
- ◆ 2012 DN wins ImageNet competition (15% errors, second best is 26%)
- ◆ 2013 G. Hinton hired by Google
- ◆ 2013 Y. LeCun hired by Facebook

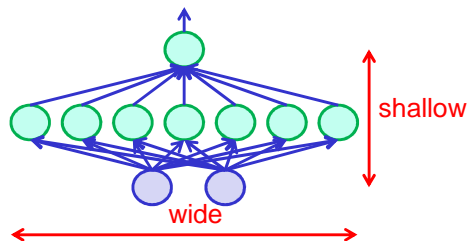
## Deep Learning Networks

### ◆ Shallow architecture:

- Few layers
- Many neurons

$$f(x) \approx \sum_j g_j(x)$$

- **Combination** of simpler functions

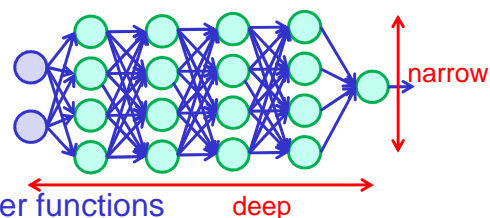


### ◆ Deep architecture:

- Many layers

$$f(x) \approx g_1(g_2(\dots g_n(x)))$$

- **Composition** of simpler functions



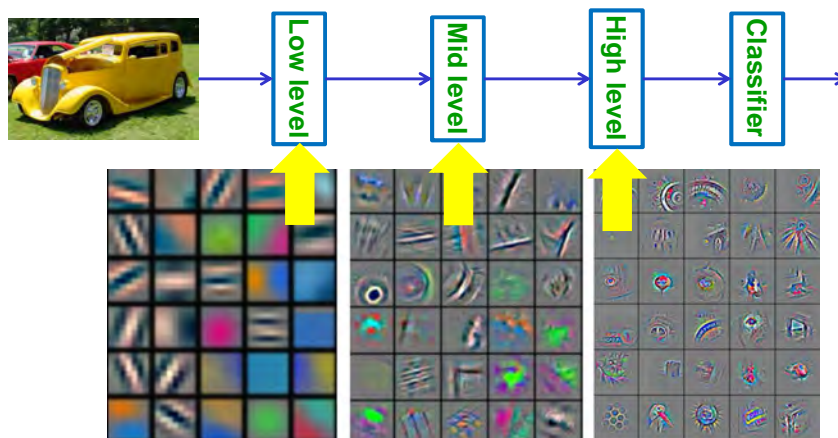
## Deep Learning Networks

- ◆ Basic idea:

For some problems, composition is more efficient than combination

## Deep Learning Networks

- ◆ Allow hierarchical analysis



## Deep Learning Networks

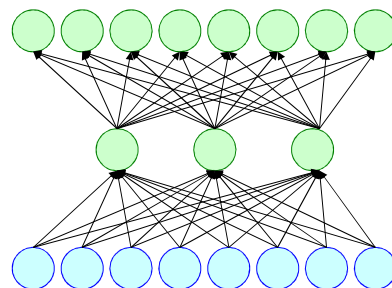
---

- ◆ Problem: How to train ?
  - Random initialization + Backpropagation
    - Numerically unstable when too many layers
    - Random initialization gives poor local minimum
    - Many parameters requires a lot of data
- ◆ Idea: train one layer after each other
  - Allows unsupervised training (more data is available)
  - Avoids random initialization
  - Often fine tune with labeled data at the end

## Deep Learning Networks

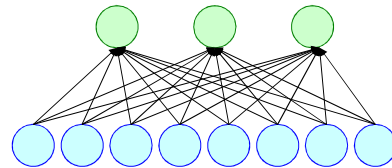
---

- ◆ One technique: stacking auto-encoders  
(other techniques exist)
  - Train auto-encoder



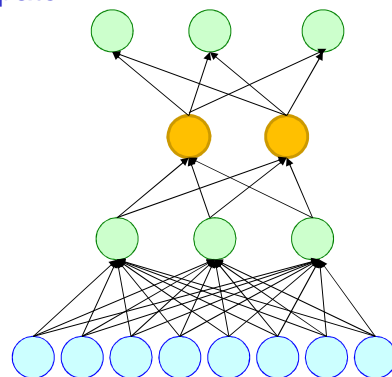
## Deep Learning Networks

- ◆ One technique: stacking auto-encoders  
(other techniques exist)
  - Train auto-encoder on a patch
  - Remove last layer



## Deep Learning Networks

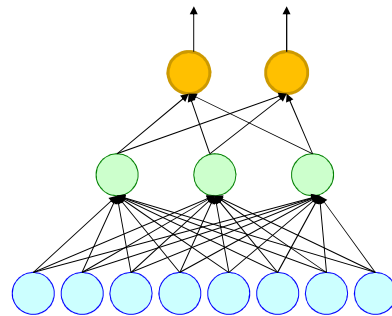
- ◆ One technique: stacking auto-encoders  
(other techniques exist)
  - Train auto-encoder on a patch
  - Remove last layer
  - Build new auto-encoder on last layer



## Deep Learning Networks

### ◆ One technique: stacking auto-encoders (other techniques exist)

- Train auto-encoder on a patch
- Remove last layer
- Build new auto-encoder on last layer
- Remove last layer
- Etc...



## Deep Learning Networks

### ◆ Benefits of layer-wise training

- Train one layer : gradient optimization works well
- Initialization: random start on one level only, the other levels are initialized with previously trained values: less risk to get a bad minimum
- Can use unlabelled data (e.g. millions of images)

### ◆ Generally followed by a step of fine tuning with labelled data

- But the network has been pre-trained, so optimization gives a better minimum
- Current architectures: 5 – 150 – « infinite » levels

## Deep Learning Networks

---

- ◆ Active field of research
- ◆ Many new ideas:
  - RELU activation function
  - Drop-out training
  - Contrastive learning
  - Residual networks
  - Long-Short-Term Memory
  - ...
  - And GPUs
- ◆ If interested, wait for Deep Network course in Spring

## Hopfield Networks

## Associative Memory Networks

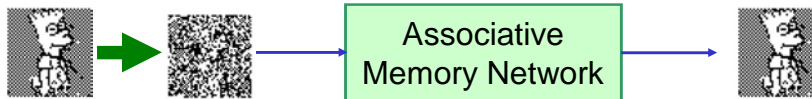
### ◆ Principle:



#### • Hetero-association



#### • Auto-association (noisy input)



## Associative Memory Networks

### ◆ Bipolar data representation:

- Bipolar pattern:  $x \in \{-1, +1\}^n$ 
  - -1 = inactive
  - +1 = active
- Properties:
  - $x = \text{sign}(x)$
  - $x^T x = n$

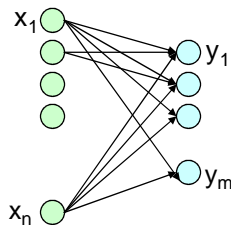
### ◆ Bipolar vs binary representation:

- More orthogonal vector pairs:

$$\text{bipolar: } 2^n \binom{n}{n/2} (n \text{ even}) \quad \text{binary: } 3^n$$

## Associative Memory Networks

- Simple example:  $F: \{-1, +1\}^n \rightarrow \{-1, +1\}^m$



$$u_j = \sum_i w_{ij} x_i$$

$W$  weight matrix  $m \times n$

$$F(x) = \text{sign}(W \cdot x) = y$$

- Given  $(a, b)$  we want to find  $W$  s.t.  $F(a) = b$

- Note that:  $(ba^T)a = b$   $(a^T a) = n$

$$b = \text{sign}(b) = \text{sign}\left(\frac{1}{n}(ba^T)a\right)$$

$$W = \frac{1}{n}ba^T$$

$\longleftrightarrow W$

## Associative Memory Networks

- Slightly more complex example:
- Assume we want to encode a set of  $T$  patterns  $(a_i, b_i)$ ,  $i=1, \dots, T$
- Also, assume that the  $a_i$  are orthogonal:  $a_i^T a_j = 0$

- Let 
$$W = \frac{1}{n} \sum_i b_i a_i^T$$

- Then

$$Wa_k = \frac{1}{n} \left( \sum_i b_i a_i^T \right) a_k = \frac{1}{n} \sum_i b_i (a_i^T a_k) = \frac{1}{n} b_k (a_k^T a_k) = b_k$$

$$F(a_k) = \text{sign}(Wa_k) = \text{sign}(b_k) = b_k$$



## Associative Memory Networks

- ◆ Stability: What is the effect if the input is slightly pertubated by a noise vector  $d$ :  $F(a_k+d)$  ?

$$\begin{aligned} W(a_k + d) &= \frac{1}{n} \left( \sum_i b_i a_i^T \right) (a_k + d) = \frac{1}{n} \sum_i b_i (a_i^T a_k) + \frac{1}{n} \left( \sum_i b_i a_i^T \right) d \\ &= b_k + \frac{1}{n} \left( \sum_i b_i a_i^T \right) d \end{aligned}$$

$$F(a_k + d) = \text{sign}(W(a_k + d)) = \text{sign} \left( b_k + \frac{1}{n} \left( \sum_i b_i a_i^T \right) d \right)$$

- If  $d$  is small or uncorrelated with  $\sum b_i a_i^T$  then  

$$F(a_k + d) = \text{sign}(b_k) = b_k$$

## Associative Memory Networks

- ◆ Assume now that the  $a_i$  have low correlation

$$\begin{aligned} W a_k &= \frac{1}{n} \left( \sum_i b_i a_i^T \right) a_k = \frac{1}{n} \sum_i b_i (a_i^T a_k) \\ &= b_k + \frac{1}{n} \sum_{i \neq k} b_i (a_i^T a_k) \end{aligned}$$

$$F(a_k) = \text{sign}(W a_k) = \text{sign} \left( b_k + \frac{1}{n} \sum_{i \neq k} b_i (a_i^T a_k) \right)$$

- If  $(a_i^T a_k)/n \approx 0$  then  

$$F(a_k) = \text{sign}(b_k) = b_k$$

## Associative Memory Networks

### ◆ Summary up to now:

- We considered 1 layer network
- Training patterns  $(a_i, b_i)$ ,  $i=1, \dots, T$
- Weight matrix (called Hebbian matrix):

$$W = \frac{1}{n} \sum_i b_i a_i^T$$

### ◆ Properties:

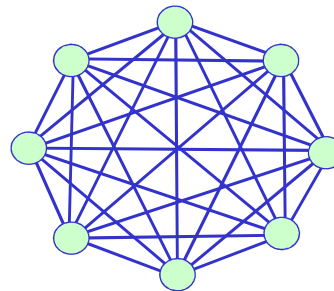
Network associates  $b_i$  to  $a_i$  when:

- $a_i$  orthogonal (uncorrelated)
- Stable under small noise perturbation
- $a_i$  low correlation

## Hopfield Networks

### ◆ Hopfield Networks are:

- Auto-associative networks  
 $a = b$  (and  $m=n$ )
- Contains  $n$  neurons
- Each neuron activates all others except itself
- Weights are symmetric
- Operation:
  - Network is initialized
  - Neurons fire, network evolve
  - Until final stable state

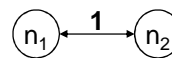


## Hopfield Networks

- ◆ Neuron  $i$ : two possible states  $x_i \in \{-1, +1\}$
- ◆ Network: state  $x = (x_1, x_2 \dots x_n)$
- ◆ Fully connected with symetric connections  $w_{ij}$ 
  - Hopfield net properties:  $w_{ij} = w_{ji}$ ,  $w_{ii} = 0$
- ◆ Non-deterministic evolution:
  - Select a neuron  $k$
  - Change state: 
$$x_i^{(t+1)} = \begin{cases} \text{sign}(\sum_j w_{kj} x_j^{(t)} - \theta_k) & \text{if } i = k \\ x_i^{(t)} & \text{else} \end{cases}$$
  - Trajectory:  $x^{(0)} \rightarrow x^{(1)} \rightarrow \dots \rightarrow x^{(t)} \rightarrow$

## Hopfield Net Example

- ◆ 2 neurons
  - Weight matrix  $w = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
  - Thresholds  $\theta_i = 0$

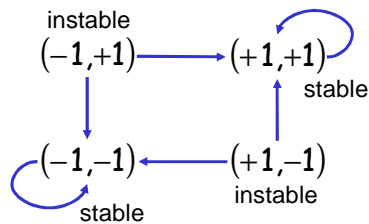


State		Next state	
		Select $n_1$	Select $n_2$
$S_1$	-1, -1	$S_1$	$S_1$
$S_2$	+1, -1	$S_1$	$S_4$
$S_3$	-1, +1	$S_4$	$S_1$
$S_4$	+1, +1	$S_4$	$S_4$

## Hopfield Net Example



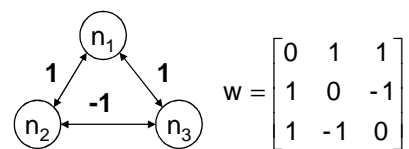
### ♦ Trajectories:



## Hopfield Net Example

### ♦ 3 neurons

- Thresholds  $\theta_i = 0.5$
- 8 states:  $\{\pm 1, \pm 1, \pm 1\}$
- Assume initial state:  $(+1, -1, -1)$   
 $x_1^{(0)} = +1, x_2^{(0)} = -1, x_3^{(0)} = -1$



- Non-deterministic evolution: for example,  $k = 2$

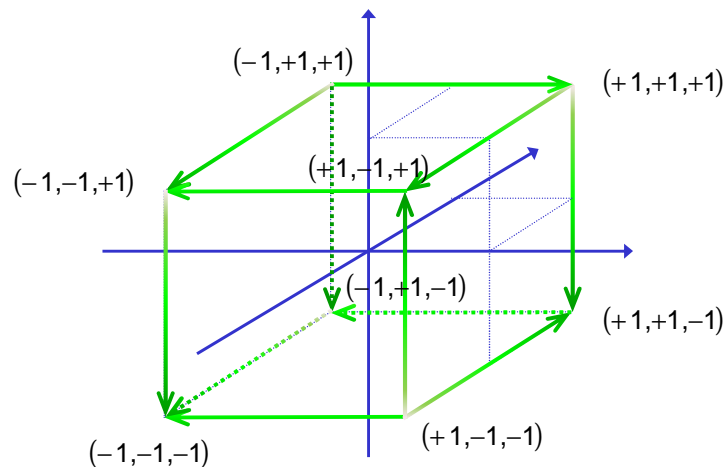
$$\begin{aligned} x_1^{(1)} &= x_1^{(0)} = +1 \\ x_2^{(1)} &= \text{sign}\left(\sum_j w_{2j}x_j^{(0)} - \theta_2\right) = +1 \\ x_3^{(1)} &= x_3^{(0)} = -1 \end{aligned}$$

- Next state  $(+1, +1, -1)$

## Hopfield Net Example

		Next state		
State		Select $n_1$	Select $n_2$	Select $n_3$
$S_1$	-1,-1,-1	$S_1$	$S_1$	$S_1$
$S_2$	+1,-1,-1	$S_1$	$S_4$	$S_6$
$S_3$	-1,+1,-1	$S_3$	$S_1$	$S_3$
$S_4$	+1,+1,-1	$S_3$	$S_4$	$S_4$
$S_5$	-1,-1,+1	$S_5$	$S_5$	$S_1$
$S_6$	+1,-1,+1	$S_5$	$S_6$	$S_6$
$S_7$	-1,+1,+1	$S_8$	$S_5$	$S_3$
$S_8$	+1,+1,+1	$S_8$	$S_6$	$S_4$

## Hopfield Network Example



## Hopfield Net Trajectories

---

- ◆  $2^n$  state vectors
- ◆ Trajectories may either:
  - 1: converge to a stable point
  - 2: cycle
- ◆ Limit in case 1: fundamental memory
- ◆  $x$  is a fundamental memory iff

$$x_i^{(t+1)} = \text{sign}\left(\sum_j w_{ij}x_j^{(t)} - \theta_i\right) = x_i^{(t)} \quad \forall i$$

- ◆ (we will see that cycles are impossible)
- ◆ We assume that  $\text{sign}(0) = -1$

## Hopfield Net Trajectories

---

- ◆ Theorem:
  - all trajectories terminate in a stable point
- ◆ Proof: define energy  $E = -\frac{1}{2} \sum_{ij} w_{ij}x_i x_j + \sum_i \theta_i x_i$ 
  - during evolution, assume neuron  $k$  changes:  
(remember that  $w_{kk} = 0$ )

$$\begin{aligned} E^{(t+1)} - E^{(t)} &= -\frac{1}{2} \sum_{ij} w_{ij} (x_i^{(t+1)} x_j^{(t+1)} - x_i^{(t)} x_j^{(t)}) + \sum_i \theta_i (x_i^{(t+1)} - x_i^{(t)}) \\ &= -\sum_i w_{ik} x_i^{(t)} (x_k^{(t+1)} - x_k^{(t)}) + \theta_k (x_k^{(t+1)} - x_k^{(t)}) \\ &= -(x_k^{(t+1)} - x_k^{(t)}) \left( \sum_i w_{ik} x_i^{(t)} - \theta_k \right) \end{aligned}$$

## Hopfield Net Trajectories

$$E^{(t+1)} - E^{(t)} = -\left(x_k^{(t+1)} - x_k^{(t)}\right) \left( \sum_i w_{ik} x_i^{(t)} - \theta_k \right) \\ = -\left(\text{sign}(u_k^{(t+1)}) - x_k^{(t)}\right) u_k^{(t+1)}$$

- ◆ when neuron k changes its state:

$x_k^{(t)}$	$x_k^{(t+1)}$	$u_k^{(t+1)}$	$E^{(t+1)} - E^{(t)}$
+1	-1	$\leq 0$	$\leq 0$
-1	+1	$> 0$	$< 0$

- in all cases:  $E^{(t+1)} \leq E^{(t)}$       energy decreases

## Hopfield Net Trajectories

- ◆ Energy along a trajectory always decreases
  - there are only finite states,
  - so trajectory ends with constant energy
- ◆ Cycles with constant energy are not possible
  - this is true, but why ?
- ◆ So each trajectory terminates in a single stable state
  - this is a fundamental memory of the network

## Hopfield Net Example

### ◆ 3 neurons

- 8 states:  $\{\pm 1, \pm 1, \pm 1\}$

thresholds = 0.5

Energy

3.5

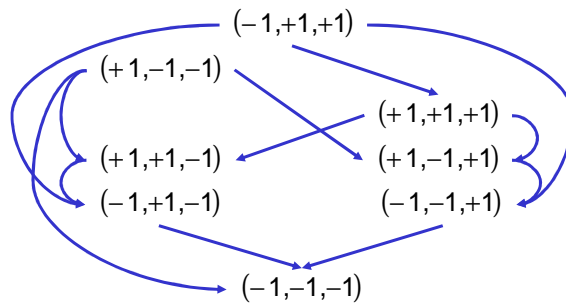
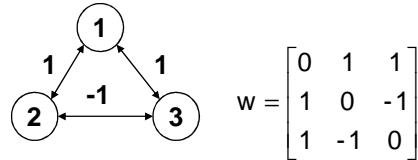
2.5

0.5

-0.5

-1.5

-2.5

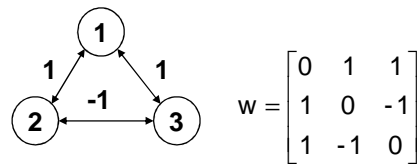


## Hopfield Net Example

### ◆ 3 neurons

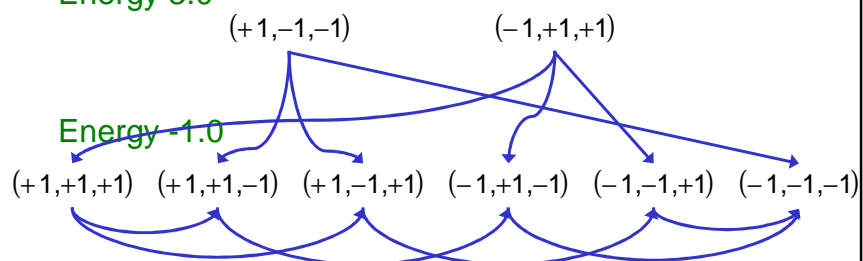
- 8 states:  $\{\pm 1, \pm 1, \pm 1\}$

- thresholds zero



Energy 3.0

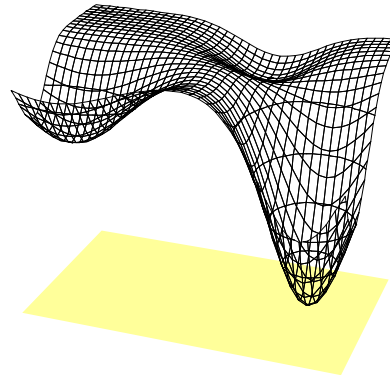
Energy -1.0





## Intuitive Idea

- ◆ Fundamental memories are attractors on the energy surface
- ◆ A basin is the set of trajectories which end in a given memory
- ◆ End of trajectory depends on initial point
- ◆ Fundamental memories which are too close are confusable



## Associative Memory

- ◆ Example: image restoration
  - assume 1 pixel = 1 neuron
  - assume image I is a fundamental memory
  - assume J is a noisy version of image I
  - then, initialize network state to J, let evolve
  - it is likely that the trajectory will end in I

initial

degraded

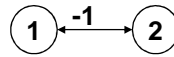
restored



## Synchronous Update

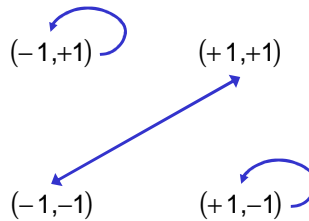
- ◆ All neurons fire simultaneously
- ◆ The trajectory property does not hold: cycles are possible

- two neurons
- threshold zero



$$w = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$$

- ◆ Trajectories:



## Hebb Weights

- ◆ How to build a network with known fundamental memories ?

- Patterns  $p_1, p_2, \dots, p_T$   $p_i \in \{-1, +1\}^n$
- Hebb weights:

$$w_{ij} = \frac{1}{n} \left( \sum_{k=1}^T p_{ki} p_{kj} - T \delta_{ij} \right)$$

$\delta_{ij}$  is the « Kronecker symbol »;  
 $\delta_{ij} = 1$  if  $i=j$   
 $0$  else

- intuitive:
  - if  $p_{ki} = p_{kj}$ ,  $p_{ki} p_{kj} = +1$  contributes to  $w_{ij} > 0$
  - if  $p_{ki} \neq p_{kj}$ ,  $p_{ki} p_{kj} = -1$  contributes to  $w_{ij} < 0$

## Hebb Weights

- ◆ If we set state to  $p_k$

- ◆ activation for neuron  $i$ :

$$y_i = \sum_j w_{ij} p_{kj} = \sum_{j \neq i} \left( \frac{1}{n} \sum_h p_{hi} p_{hj} \right) p_{kj} = \frac{1}{n} \sum_h p_{hi} \sum_{j \neq i} p_{hj} p_{kj}$$

when  $h = k$ ,  $\sum_{j \neq i} p_{hj} p_{kj} = \sum_{j \neq i} p_{kj}^2 = n - 1$

$$y_i = p_{ki} - \frac{1}{n} p_{ki} + \frac{1}{n} \sum_{h \neq k} p_{hi} \sum_{j \neq i} p_{hj} p_{kj}$$

- if patterns are not correlated,  $\sum_{j \neq i} p_{hj} p_{kj}$  is low

so,  $p_{ki} = \text{sign}(y_i)$

the pattern  $p_k$  is a fundamental memory

## Hebbian Training

- ◆ Iterative training algorithm (heuristic)

- Patterns  $p_1, p_2, \dots, p_K$   $p_i \in \{-1, +1\}^n$

- initialize weights  $w_{ij}$

- select a pattern  $p_k$

- update weights (Hebbian rule):

$$w_{ij} \rightarrow w_{ij} + \lambda p_{ki} p_{kj} \quad \text{with } \lambda > 0$$

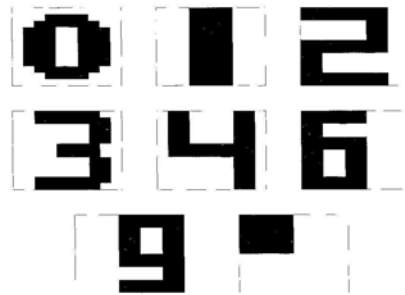
- intuitive:

- if  $p_{ki} = p_{kj}$ , increase  $w_{ij}$
- if  $p_{ki} \neq p_{kj}$ , decrease  $w_{ij}$

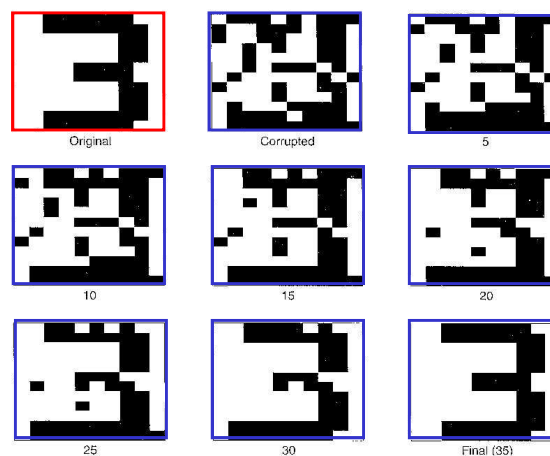
- iterate

## Hopfield Example

- ◆  $n = 120$  neurons  
 $n^2 - n = 12280$  weights
- ◆ Trained to retrieve 8 digitlike black and white patterns
- ◆ Weights computed using the Hebb's rule.

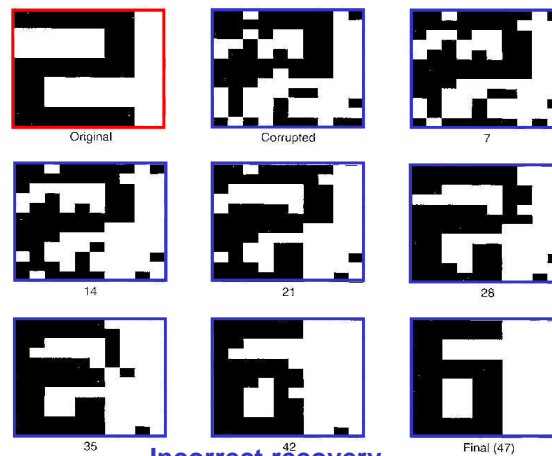


## Hopfield Example



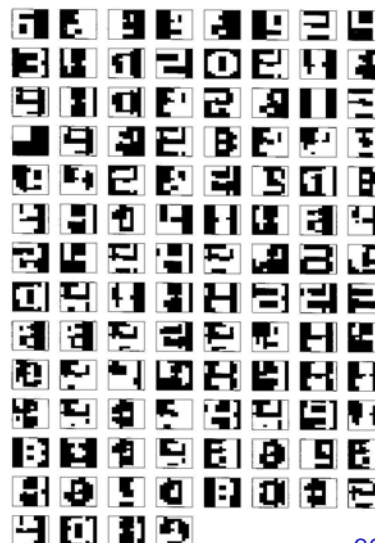
Correct recovery

## Hopfield Example



## Hopfield Example

- ◆ Spurious States:
  - Reversed fundamental memories
  - Mixture states
  - “Spin-glass states”



## Spurious States

---

- ◆ Training may produce local energy minima which are not patterns: those are spurious states
- ◆ Spurious states maybe:
  - opposite of stable states (if zero thresholds)
  - odd mixture of stable states  
ex:  $\pm s_1 \pm s_2 \pm s_3$
  - “spin glass” states

## Capacity of Hopfield Nets

---

- ◆ How many fundamental memories in a net ?
- ◆ Assuming  $p$  random patterns for  $n$  neurons:
  - if we want more than 99% bits correct:  
 $p < 0.138 n$
  - if we want more than 99% patterns correct:

$$p \leq \frac{n}{2 \log n}$$

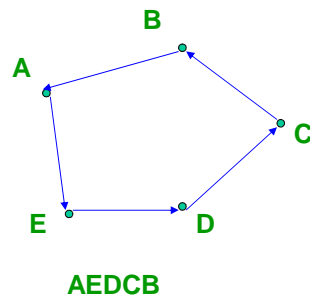
- if we want more than 99% chances that all patterns are correct:

$$p \leq \frac{n}{4 \log n}$$

## Optimization with Hopfield Networks

### ◆ Design network for TSP with n cities:

- $N=n \times n$  neurons
- Neuron  $ij$  is 1 if city  $i$  was visited at time  $j$ , 0 else



	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

## Optimization with Hopfield Networks

### ◆ Constraints:

- At any given time only one town is visited

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

- Each town is visited only once

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

### ◆ Cost function:

- Tour length = sum of distances between towns visited

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

## Optimization with Hopfield Networks

- ◆ Include constraints in cost function:

- Constrained optimization:

$$C(Y) = \sum_{i=1}^n \sum_{k=1, k \neq i}^n \sum_{j=1}^n c_{ik} y_{ij} (y_{k,j-1} + y_{k,j+1}) \quad \text{with} \quad \sum_{j=1}^n \left( \sum_{i=1}^n y_{ij} - 1 \right)^2 = 0$$

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

- Unconstrained optimization:

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

## Optimization with Hopfield Networks

- ◆ Represent cost function as network energy:

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

$$E = -\frac{1}{2} \sum_{i,j} \sum_{k,l} w_{ij,kl} y_{ij} y_{kl} + \sum_{i,j} \theta_{ij} y_{ij}$$

Results in:

$$\begin{cases} w_{ij,kl} = -ac_{ik} (\delta_{l,j-1} + \delta_{l,j+1}) - b(\delta_{ik} + \delta_{jl} + \delta_{ik} \delta_{jl}) \\ w_{ij,ij} = 0 \\ \theta_{ij} = 2b \end{cases}$$



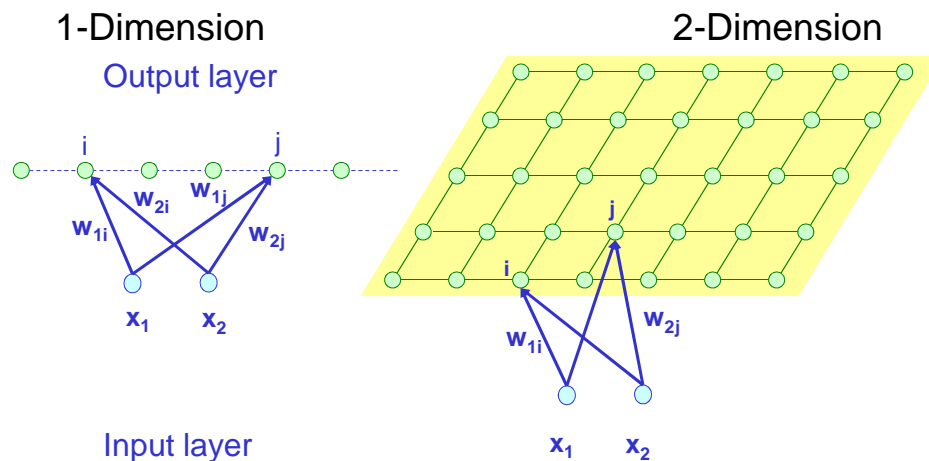
# SOM: Self Organizing Maps

## Self Organizing Maps

---

- ◆ SOM: Kohonen (1982)
- ◆ Neural networks with 2 layers:
  - Input layer
  - Output layer
- ◆ Full connections
- ◆ Topology on output layer (generally 2D)
  - Define neighbours of an output neuron
- ◆ Specific training algorithm:
  - Called Competitive Learning
  - It is an unsupervised method

## Self Organizing Maps



## Competitive Learning

- ◆ Unsupervised training algorithm
  - Training set of input vectors
  - We don't know the corresponding output vectors
- ◆ Best Matching Unit:
  - Input vector  $x^j = (x_1^j, x_2^j, \dots, x_m^j)$
  - Output neuron  $k$ : weight vector  $w_k = (w_{1k}, w_{2k}, \dots, w_{mk})$
  - Distance between output neuron and input vector:
 
$$\|w_k - x^j\|^2 = \sum_i (w_{ik} - x_i^j)^2$$
  - Closest output neuron:  $\underset{k}{\operatorname{argmin}} \|w_k - x^j\|^2$

## Competitive Learning

### ♦ Idea: « Winner Takes All » rule:

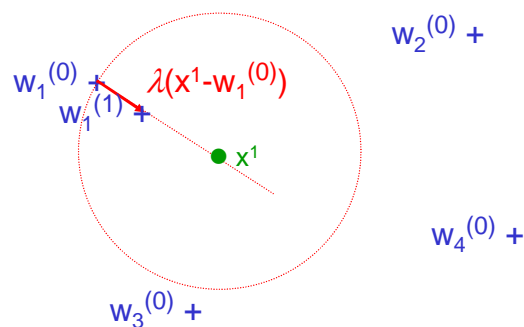
- Initialize random weights  $w_{ij}^{(0)}$
- Iterate over all training samples: select  $x^{i(t)}$
- Set input values to  $x^{i(t)}_1, x^{i(t)}_2, \dots, x^{i(t)}_m$
- Compute best neuron  $k^*$ :  

$$k^* = \underset{k}{\operatorname{argmin}} \|w_k^{(t)} - x^{i(t)}\|^2$$
- Update weights for  $k^*$ :  

$$w_{ik^*}^{(t+1)} = w_{ik^*}^{(t)} + \lambda(x_i^{i(t)} - w_{ik^*}^{(t)}) \quad \lambda > 0 \quad i=1,2,\dots,n$$
- Iterate

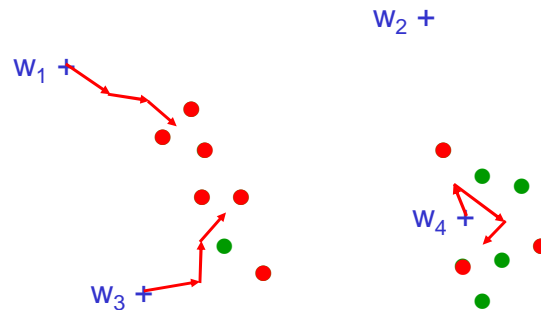
Note: often, for better convergence,  $\lambda$  decreases with  $t$

## Competitive Learning Illustration



## Competitive Learning Illustration

- ◆ Weight vectors will converge to center of clusters of input vectors



## Neuronal Interpretation

- ◆ Distance with input vectors:

$$\|w_k - x^j\|^2 = \|w_k\|^2 - 2 \sum_i w_{ik} x_i^j + \|x^j\|^2$$

- ◆ If weight vectors are normalized  $\|w_k\| = 1$ , then:

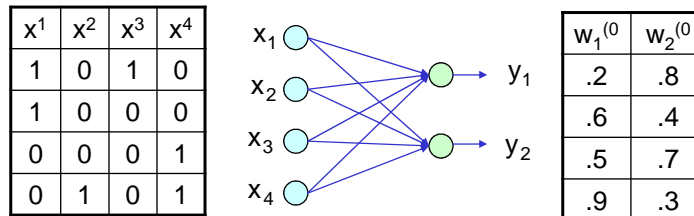
$$k^* = \operatorname{argmax}_k \sum_i w_{ik} x_i^j = \operatorname{argmin}_k \|w_k - x^j\|^2$$

- ◆ Minimum distance is also maximum activation
- ◆ The winner neuron is also the neuron with largest output:

$$y_k^j = f\left(\sum_i w_{ik} x_i^j\right)$$

## Competitive Learning Example

- ◆ 4 input values, 2 output neurons
- ◆ 4 input vectors: initial weights:



- ◆ Competitive training with  $\lambda = 0.6$

## Competitive Learning Example

- ◆  $\|w_1^{(0)} - x^1\|^2 = (.2-1)^2 + (.6-1)^2 + (.5-0)^2 + (.9-0)^2 = 1.86$
- ◆  $\|w_2^{(0)} - x^1\|^2 = (.8-1)^2 + (.4-1)^2 + (.7-0)^2 + (.3-0)^2 = 0.98$
- ◆  $w_2^{(0)}$  is the winner, update it:

$$w_2^{(1)} = w_2^{(0)} + \lambda (x^1 - w_2^{(0)})$$

$w_2^{(1)}$
.92
.76
.28
.12

$w_2^{(0)}$
.8
.4
.7
.3

$\lambda$
.6
.6
.6
.6

$x^1$
1
1
0
0

$w_2^{(0)}$
.8
.4
.7
.3

➔

$w_1^{(1)}$	$w_2^{(1)}$
.2	.92
.6	.76
.5	.28
.9	.12

$w_1^{(1)} = w_1^{(0)}$

## Competitive Learning Example

- ◆ Sample input vector 2:
- ◆  $\|w_1^{(1)} - x^2\|^2 = 0.66$
- ◆  $\|w_2^{(1)} - x^2\|^2 = 2.28$
- ◆  $w_1^{(1)}$  is the winner, update it:

$$w_1^{(2)} = w_1^{(1)} + \lambda (x^2 - w_1^{(1)})$$

.08	.2
.24	.6
.20	.5
.96	.9

$$=$$

.2	.6
.6	.6
.5	.6
.9	.6

$$+$$

.6	.6
.6	.6
.6	.6
.6	.6

$$\times$$

0	0
0	0
0	0
1	1

$$-$$

.2	.6
.6	.5
.5	.9
.9	.9

$$\rightarrow$$

$w_1^{(2)}$	$w_2^{(2)}$
.08	.92
.24	.76
.20	.28
.96	.12

$w_2^{(2)} = w_2^{(1)}$

## Competitive Learning Example

- ◆ After 4 samples:

$w_1^{(4)}$	$w_2^{(4)}$
.03	.97
.10	.30
.68	.11
.98	.05

- ◆ Note: we can reduce  $\lambda$  for example, each time the training set has been entirely processed:
  - $\lambda = \lambda / 2$
  - This gives the sequence  $\lambda = 0.6, 0.3, 0.15, \dots$

## Competitive Learning Example

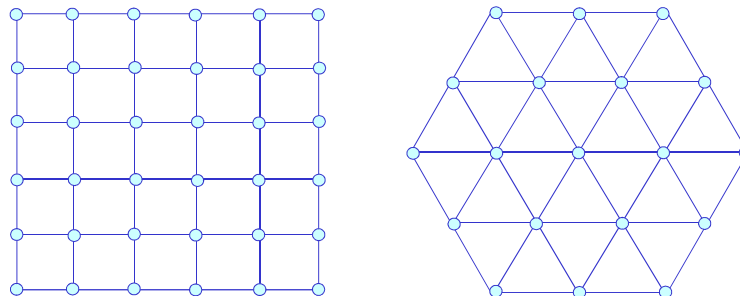
- ◆ After 10 x 4 iterations

$w_1^{(40)}$	$w_2^{(40)}$		$(x^2+x^4)/2$	$(x^1+x^3)/2$	$x^1$	$x^2$	$x^3$	$x^4$
.01	.99	→	0.	1.	1	0	1	0
.03	.41		0.	0.5	1	0	0	0
.58	.02		.5	0.	0	0	0	1
.99	.01		1.	0.	0	1	0	1

- ◆  $x^1$  and  $x^3$  belong to class 2
- ◆  $x^2$  and  $x^4$  belong to class 1

## Self Organizing Maps

- ◆ Output neurons are organized in a topological structure
  - usually 1D, 2D or 3D
- ◆ Example of 2D structures:



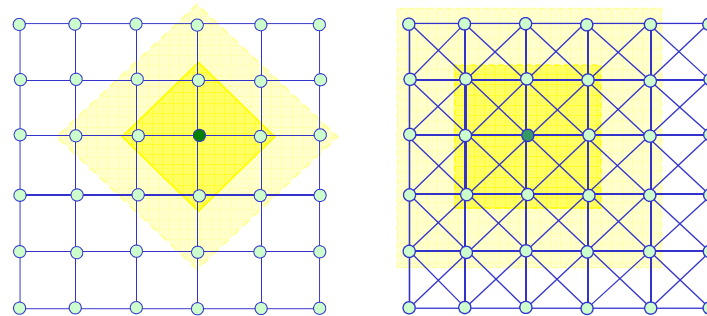
## Self Organizing Maps

### ◆ Definition of neighborhoods

#### ◆ 1-Dimension



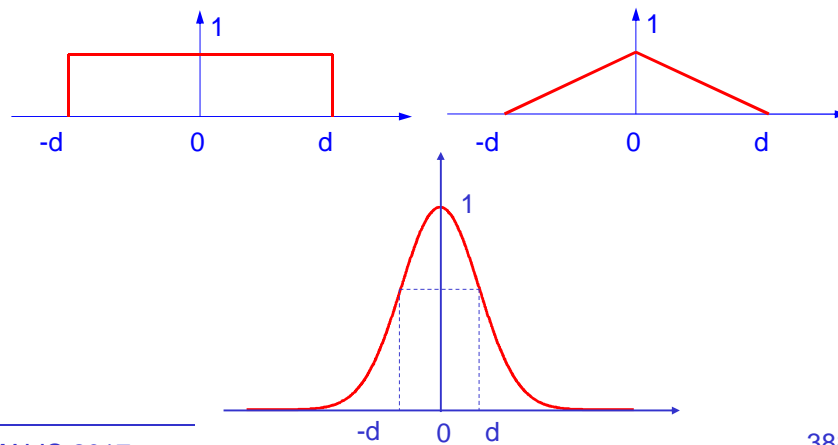
#### ◆ 2-Dimension



## Self Organizing Maps

### ◆ Neighborhood Functions $\phi_k(i)$

#### ◆ 1-Dimension





## Self Organizing Maps

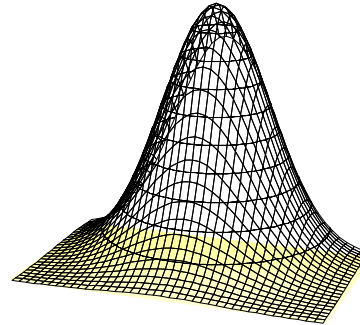
- ◆ Neighborhood Functions  $\varphi_{k^*}(i)$

- ◆ 2-Dimension

- Gaussian-like

$$\varphi_{k^*}(i) = e^{-\frac{\|k^* - i\|^2}{2\sigma^2}}$$

$$\varphi_{k^*}(i) = \frac{1}{1 + \frac{\|k^* - i\|^2}{\sigma^2}}$$



## Complete SOM Training Algorithm

- ◆ Idea:

- Neighbours of Best Matching Unit are also updated
  - Update depends on neighboring function:

- For Best Matching Unit  $k^*$

$$w_{k^*}^{(t+1)} = w_{k^*}^{(t)} + \lambda (x^{j(t)} - w_{k^*}^{(t)})$$

- For Neighbours  $i$

$$w_i^{(t+1)} = w_i^{(t)} + \lambda \varphi_{k^*}(i) (x^{j(t)} - w_i^{(t)})$$

(note that generally  $\varphi_{k^*}(k^*) = 1$ )

## Complete SOM Training Algorithm

### ♦ Algorithm:

- Initialize random weights  $w_{ij}^{(0)}$
- Iterate over all training samples: select  $x^{(t)}$
- Set input values to  $x^{(t)}_1, x^{(t)}_2, \dots, x^{(t)}_m$
- Compute best neuron  $k^*$ :

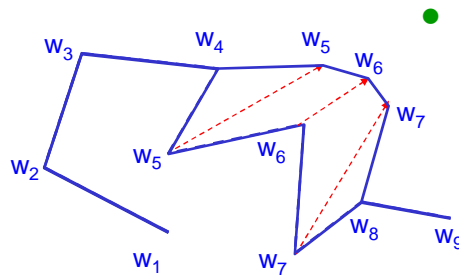
$$k^* = \underset{k}{\operatorname{argmin}} \|w_k^{(t)} - x^{(t)}\|^2$$

- Update weights for  $k^*$  and neighbours  $i$ :

$$w_i^{(t+1)} = w_i^{(t)} + \lambda \varphi_{k^*}(i) (x^{(t)} - w_i^{(t)})$$

- Iterate

## 1-D Illustration

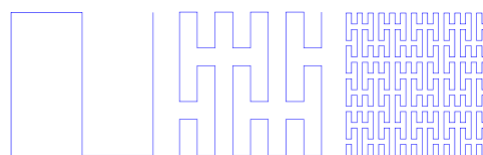


## Peano Curve Example

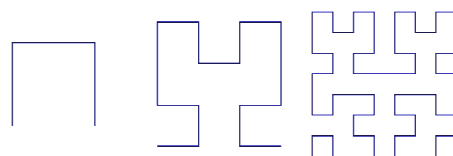
- ◆ George Cantor [1878]: **there exists** a bijective function between  $\mathbb{R}$  and  $\mathbb{R}^2$ 
  - also between the unit interval  $[0,1]$  and the unit square  $[0,1] \times [0,1]$
- ◆ Netto [1879]: **there is no** continuous bijective function between  $[0,1]$  and  $[0,1] \times [0,1]$
- ◆ Peano [1890]: **there exists** a surjective continuous function from  $[0,1]$  onto  $[0,1] \times [0,1]$ 
  - Called « space filling curves »

## Peano Curve Example

- ◆ Peano



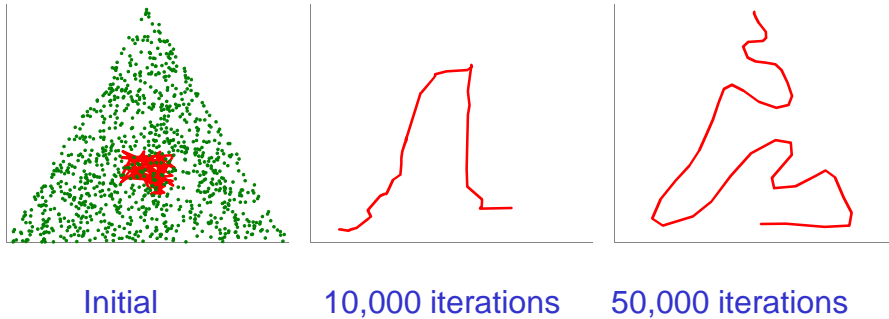
- ◆ Hilbert



## Peano Curve Example

### ◆ 1D classification of 2D surface

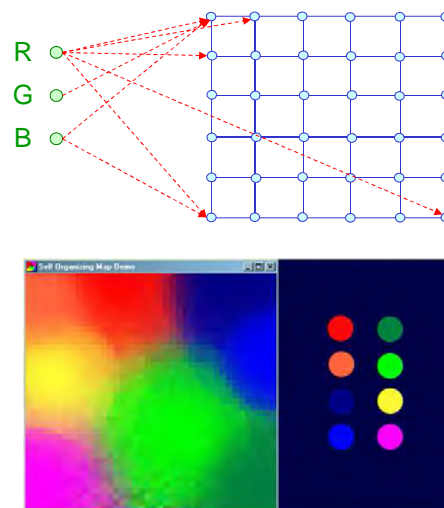
- 1000 random input vectors within triangle
- 100 output neurons in 1D chain



## Color Projection Example

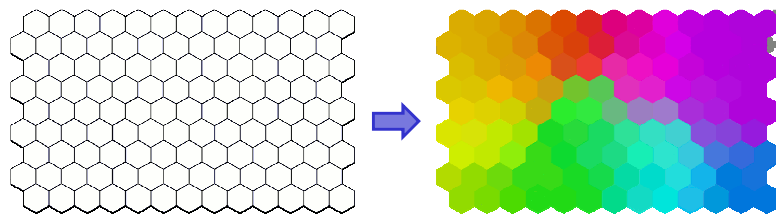
### ◆ Project 3D colors on a 2D surface

Input: 8 colors  
Output: color map



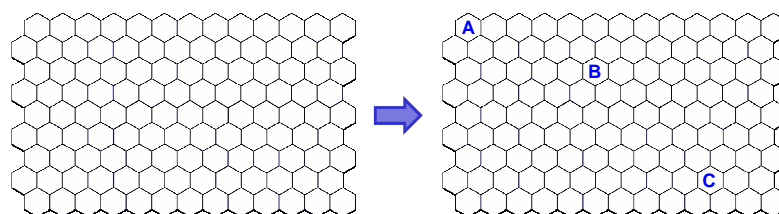
## SOM Poverty Map

- ◆ 1992 World Bank statistics of countries
- ◆ 169 countries
- ◆ 39 economic indicators for each country
  - 169 input vectors with dimension 39
- ◆ Hexagonal 9x13 2D color output map
  - Train with RGB colors



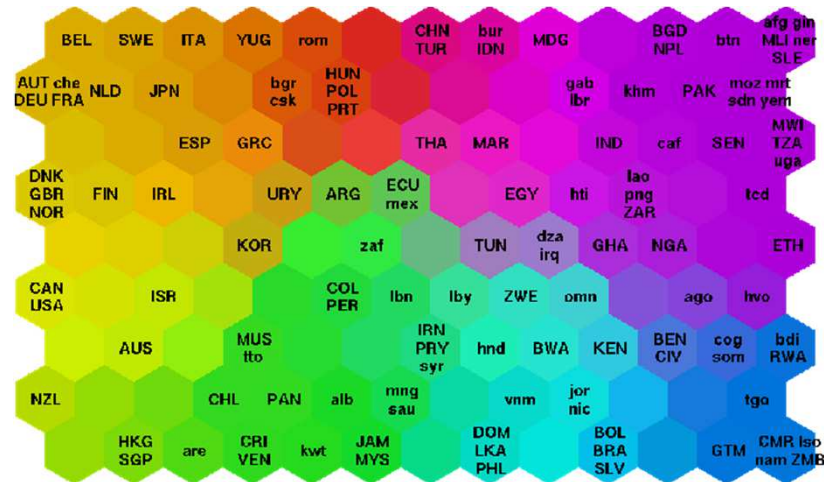
## SOM Poverty Map

- ◆ 169 countries, input vectors with dimension 39
- ◆ Hexagonal 9x13 2D color output map
  - Train with dim 39 vectors
  - After training, assign each country to closest neuron



- Countries with similar economic vectors are assigned the same or neighbour neurons

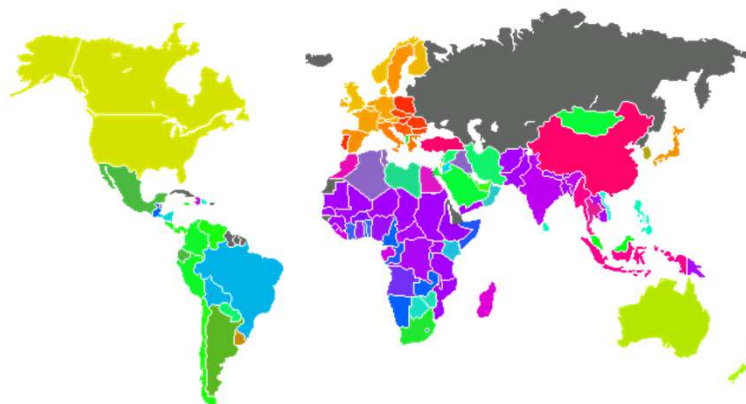
## SOM Poverty Map



MALIS 2017

399

## SOM Poverty Map



MALIS 2017

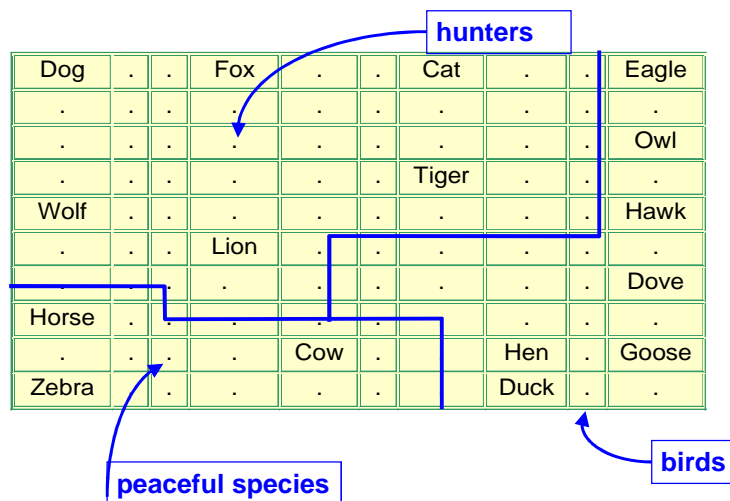
400

## SOM Animals Semantic Map

- ◆ 13 characteristics per animal
  - 16 input vectors with dimension 13
- ◆ Rectangular 10x10 grid of output neurons

		Dove	Hen	Duck	Goose	Owl	Hawk	Eagle	Fox	Dog	Wolf	Cat	Tiger	Lion	Horse	Zebra	Cow
is	Small	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0
	Medium	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
	Big	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
has	2 legs	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
	4 legs	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	Hair	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	Hooves	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
	Mane	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0
	Feathers	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
	Hunt	0	0	0	0	1	1	1	1	0	1	1	1	1	0	0	0
likes to	Run	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0
	Fly	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
	Swim	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

## SOM Animals Semantic Map



## Simulated Annealing

## Heuristic Search

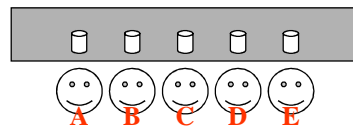
---

- ◆ Used for “difficult” problems, typically NP-hard ones.
- ◆ Optimal solutions are very computationally expensive, or impossible to find.
- ◆ Use approximation algorithms, “heuristics”, to find “good” solutions.
- ◆ Many such problems are combinatorial in nature. The solution is a sequence or grouping of discrete objects.



## An Example

- ◆ Five colleagues (named A, B, C, D and E) enter a pub to celebrate. They seat themselves along the bar to enjoy a drink.
- ◆ Unfortunately, they have strong preferences as to whom they sit beside. The friends want to organise themselves to minimise the total dislike.



$d(i,j)$	A	B	C	D	E
A	-	13	4	13	6
B	13	-	5	7	3
C	4	5	-	5	2
D	13	7	5	-	6
E	6	3	2	6	-

## Some maths/formalism

- ◆ Let  $S$  be the set of all possible set of configurations.
- ◆ In our case,  $S$  is the set of all possible orderings of the students;  $S = \{(A,B,C,D,E), (A,B,C,E,D), \dots, (E,D,C,B,A)\}$ .
- ◆ How many different such solutions are there?

## Some maths/formalism (cont.)

---

- ◆ If we consider solutions such as  $(A,B,C,D,E)$  and  $(E,D,C,B,A)$ , what do we see?
- ◆ So, what is the real total number of solutions?

## More maths...

---

- ◆ Let  $f(x) : x \in S \rightarrow \mathbb{R}$  be a cost function that assigns a real number to each configuration  $x \in S$ .
- ◆ In our case, let a solution be denoted by  $x=(x_1, x_2, x_3, x_4, x_5)$ , where  $x_q$  gives the person sitting in seat  $q$ . If  $d(i,j)$  is the dislike between person  $i$  and  $j$ , then the cost of solution  $x$  is:  
$$f(x) = d(x_1, x_2) + d(x_2, x_3) + d(x_3, x_4) + d(x_4, x_5)$$

Eg, for  $x=(A,B,C,D,E)$ ,

$$f(x) = d(A,B) + d(B,C) + d(C,D) + d(D,E) = 13+5+5+6 = 29.$$
- ◆ Our goal is to find a configuration  $x$  for which  $f(x)$  achieves a minimum (or maximum) value,  $f^*$
- ◆ ie find some optimal configuration  $x^*$  satisfying:  $f(x^*) = f^*$

## Solution Method One

---

- ◆ Complete enumeration
  - Choose best of every possible solution
- ◆ Advantages:
- ◆ Disadvantages:
  - How many solutions are there if 20 not-so-good mates go? If  $10^4$  solutions are tried each second, how long does it take to try all solutions?

## Solution Method Two

---

- ◆ Greedy sequential algorithm:
  - A greedy sequential algorithm constructs a solution by a series of steps that are 'greedy'.
  - They make a decision that looks like a good thing to do *now*, and 'sequentially'
  - Makes a sequence of decisions, never reversing an earlier decision.
  - In general, it does not guarantee optimality.

## Solution Method Two:

---

### ◆ Greedy sequential algorithm:

- 1. Pick the best pair and sit them together, call this solution  $x$ .
- 2. Repeat:
  - From the unseated people, find the person  $y$  who best fit current solution  $x$ , either from left (gives,  $(y,x)$ ), or from right (gives,  $(x,y)$ ) which ever gives better objective value.
- 3. Until all people are seated.

### ◆ Try that out now on the colleagues in the bar eg.!

## Solution Method Three:

---

### ◆ Random Search:

Choose the best of many random examples

### ◆ Advantages

### ◆ Disadvantages

## Solution Method Four:

---

- ◆ **Local Search** (aka Hill Climbing/Steepest Descent)
  - Repeatedly make small changes to the solution, accepting the changes that give improvements.
- ◆ Advantage:
- ◆ Disadvantage:

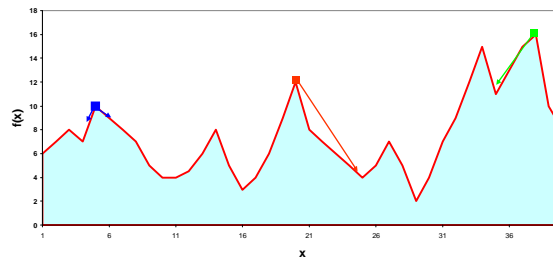
## Solution Method Four

---

- ◆ **Local Search:**
- ◆ Basic algorithm: (steepest descent)
  - initialize  $i=0$ , random  $x_0$
  - search x neighbor of  $x_i$  which minimizes  $f(x)$
  - $x_{i+1} = x$
  - iterate

## Steepest-Descent

- ◆ Also called hill-climbing for maximization
- ◆ Gradient search is a special case (if  $f$  diff.)
- ◆ Problems with steepest descent:
  - depends on initial point
  - converges to local minimum



## How to avoid local minima ?

- ◆ Need to allow up-hill moves
- ◆ should be controlled, otherwise process might never converge

→ simulated annealing idea

- ◆ inspired from thermodynamics
- ◆ introduce temperature parameter
  - high temperature → large movements allowed
  - low temperature → small movements allowed
- ◆ cool slowly

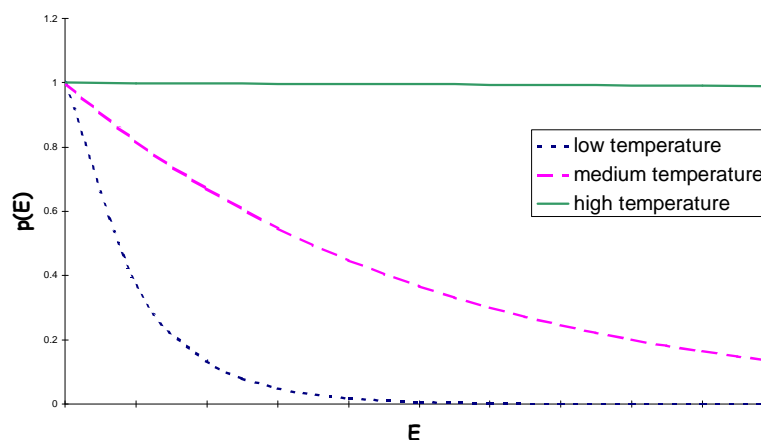
## Thermodynamics

- ◆ Dynamical system  $S$  with states  $\{x\}$
- ◆ Energy function  $E(x)$
- ◆ For temperature  $T > 0$ , system  $S$  converges to thermal equilibrium
- ◆ At thermal equilibrium,  $S$  randomly fluctuates around average value:

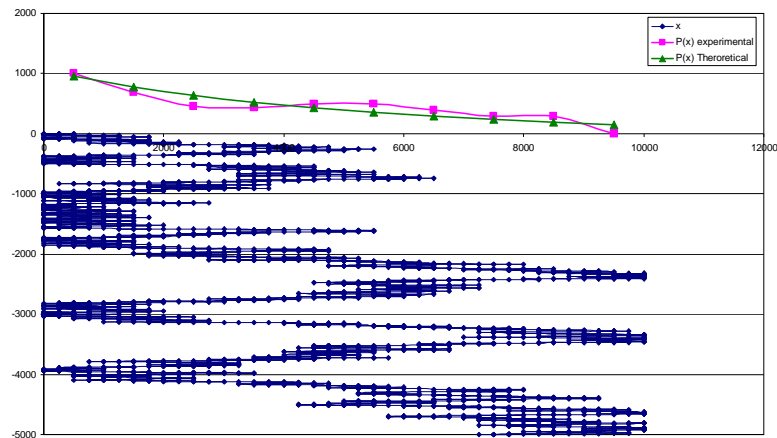
$$P(x) = \lambda e^{-\frac{E(x)}{kT}}$$

- $P(x)$  is the Boltzmann-Gibbs distribution
- $k$  is the Boltzmann constant

## Metropolis' Algorithm



## Metropolis' Algorithm



## System Evolution

- ◆ At thermal equilibrium:
  - at time  $t$ ,  $S$  in state  $x(t) = x_i$
  - at time  $t+1$ ,  $S$  in state  $x(t+1) = x_j$
  - probability to move from  $x_i$  to  $x_j$ :

$$\begin{aligned}
 P(x_j|x_i) &= \frac{P(x(t+1)=x_j, x(t)=x_i)}{P(x(t)=x_i)} \\
 &= \frac{P(x(t+1)=x_j)}{P(x(t)=x_i)} \quad \text{(Assume system is free to move)} \\
 &= \frac{\lambda e^{\frac{E(x_j)}{kT}}}{\lambda e^{\frac{E(x_i)}{kT}}} = e^{\frac{E(x_j)-E(x_i)}{kT}} = e^{\frac{\Delta E}{kT}}
 \end{aligned}$$



## Metropolis Algorithm

◆ How to get a system to thermal equilibrium ?

◆ Metropolis algorithm:

- put system in initial state  $x_0$ ,  $i=0$
- select state  $x$  neighbor of  $x_i$
- compute  $\Delta E = E(x) - E(x_i)$ 
  - if  $\Delta E < 0$ , accept move:  $x_{i+1} = x$
  - if  $\Delta E > 0$ , accept move with probability  $p = e^{-\frac{\Delta E}{kT}}$
- iterate

◆ converges to thermal equilibrium.

## Annealing

◆ Metallurgy:

- metal heated near melting point
- cooling strategy:
  - if cooled slowly, gives large crystal (global minimum energy)
  - if cooled quickly, gives crystal with imperfection (local minimum)
- cooling schedule:
  - hard to find the right one

## The Annealing Analogy

- ◆ Use thermodynamic simulation to solve optimization problems (Kirkpatrick et al., early 80's).

Thermodynamic simulation	Combinatorial Optimization
System States	Feasible Solution
Energy	Cost
Change of State	Neighboring Solution
Temperature	Control Parameter
Frozen State	Heuristic Solution

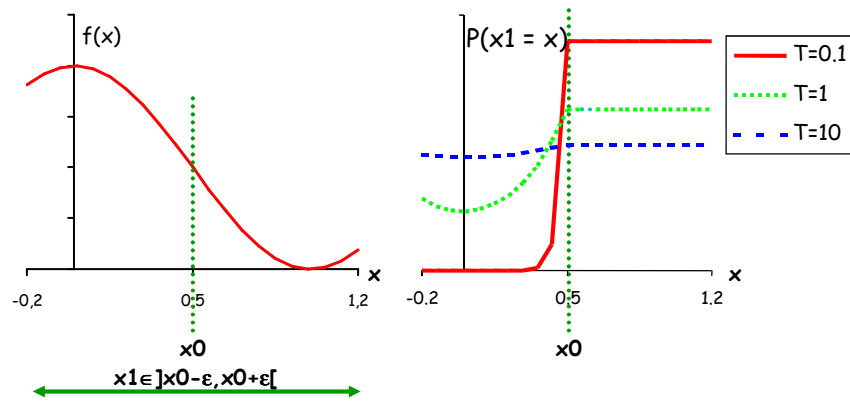
## Simulated Annealing

- ◆ Metropolis with temperature cooling:

- initialize temperature  $T_0$ , point  $x_0$ ,  $i=0$
- select  $x$  neighbor of  $x_i$
- compute  $\Delta E = f(x) - f(x_i)$ 
  - if  $\Delta E < 0$ , accept move:  $x_{i+1} = x$
  - if  $\Delta E > 0$ , accept move with probability  $p = e^{-\frac{\Delta E}{kT_i}}$
- eventually reduce temperature
- iterate

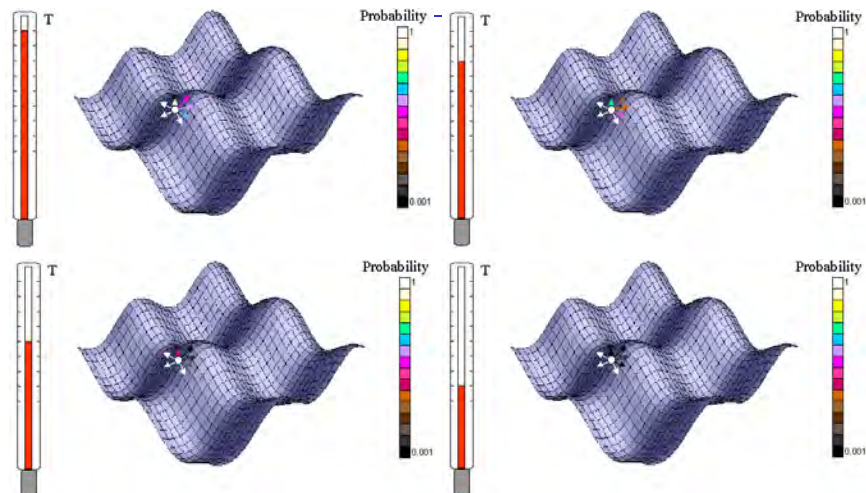
## Interpretation

- ♦ Is able to escape local minima:



MALIS 2017

425

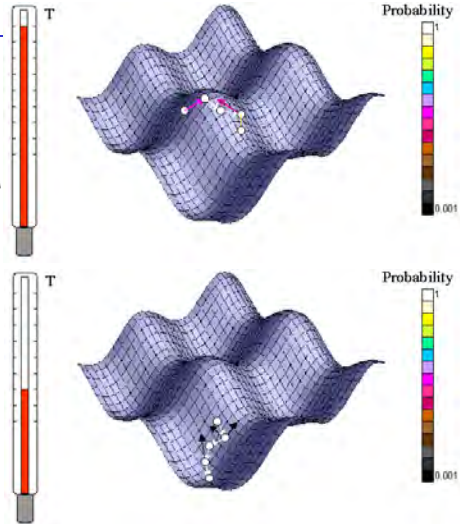


MALIS 2017

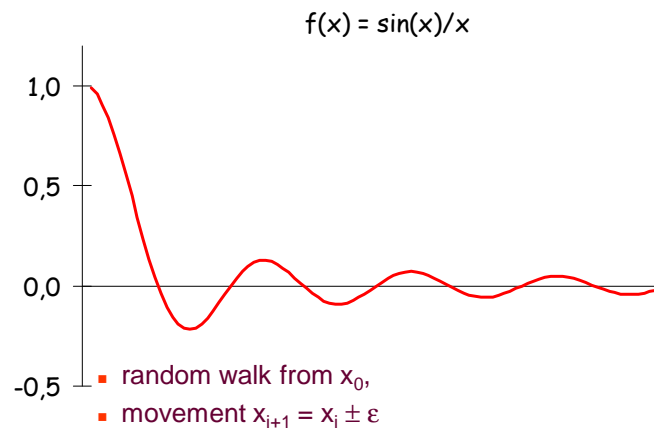
426

At the beginning up-hill moves are frequent:  
the temperature is high

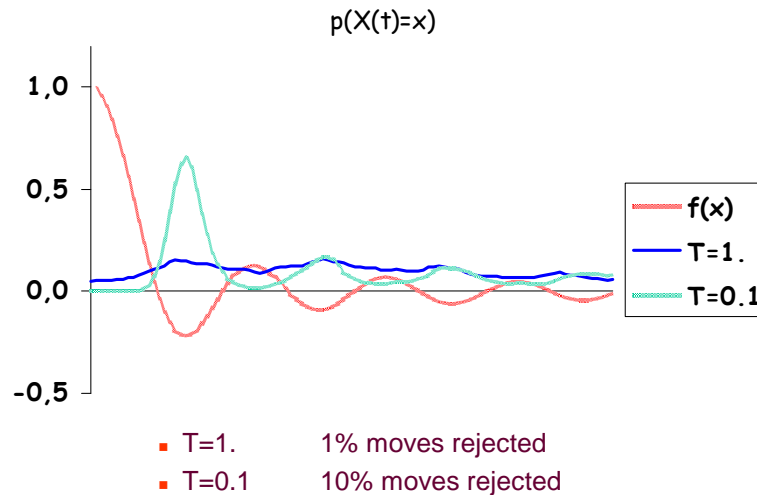
The temperature is decreased as the  
algorithm iterates:  
up-hill moves are rare



## Simulated Annealing Example



## Simulated Annealing Example



## SA Algorithm Parameters

### ♦ Algorithm implementation choices

- Annealing parameters:
  - initial temperature,
  - cooling schedule:  $T(t)$  and nb cycles per temp.
  - stopping condition.
- Problem specific:
  - choice of the space of feasible solutions,
  - form of the cost function,
  - neighborhood structure employed.

## SA Algorithm Parameters

---

- ◆ Influence on:
  - Convergence speed,
  - Quality of the solutions obtained.
- ◆ From theoretical research results:
  - to guarantee convergence to a global optimum SA may require more iterations than exhaustive search,
  - good advice as to which factor to take into account for deciding on the algorithm details.

## Choice of Annealing Parameters

---

- ◆ Initial Temperature:
  - “hot” enough to allow an almost free exchange of neighboring solutions.
  - Physical analogy: a material is quickly heated up to its liquid state before being cooled slowly
  - What is “hot” enough? Two methods:
    - Known problem so  $T$  can be computed (rare).
    - Decide on proportion of up-hill moves, then heat up the system until the desired proportion of accepted moves is obtained.

## Initial Temperature (Example)

- ◆ Suppose that  $N=1000$  iterations (moves) are going to be generated per temperature value. Suppose that an initial solution is generated, and from this, 400 of the first 1000 solutions had a cost higher than the solution that generated it.
- ◆ Let  $\delta(j)$  be the change in cost of the  $j$ -th solution from the  $(j-1)$ -th, and  $\bar{\delta}$  be the average change in cost over the 400 cost-increasing solutions, that is:

$$\bar{\delta} = (1/400) \sum_{j:\delta(j)>0} \delta(j)$$

## Initial Temperature (Example Cont.)

- ◆ Choose initial temperature,  $T$ , such that 200 of these 400 would be accepted. That is,

$$e^{-\frac{\bar{\delta}}{T}} = \frac{1}{2},$$
$$T = \frac{\bar{\delta}}{\ln 2}.$$

## Cooling Schedule

---

◆ Theory:

To guarantee convergence to global minimum:

- $T(t) = N\Delta/\log t$  [Geman and Geman 1984]
- $N\Delta$  generally large  $\Rightarrow$  computationally expensive

- ◆ Both empirical evidence and theoretical research show that the combination cooling rate / number of cycles per iteration is the key to optimal convergence.

## Cooling Schedule (in practice)

---

- Cooling rate
  - Geometric reduction:  
 $T(t+1) = aT(t)$  with  $0.8 \leq a \leq 0.99$
  - Slow reduction:  
 $T(t+1) = T(t) / (1 + \beta T(t))$  with  $\beta \rightarrow 0$
  - Logarithmic reduction:  
 $T(t+1) = c/\log(1+t)$
- Temperature updates:
  - T changes at every iteration ( $n = 1$ ),
  - T unchanged for  $n$  iterations, as T lowers increase  $n$  until local optimum is fully explored.



## Stopping Criterion

---

- ◆ As  $T \rightarrow 0$ , probability of accepting any up-hill moves is small, therefore chance of a change of local optimum is negligible.
- ◆ Possible criteria:
  - Stop iterations when  $T < \text{threshold}$ ,
  - Fix a total number of iterations. (sufficiently large to reach small  $T$  to ensure convergence).
  - Stop after no up-hill moves have been accepted for a given number of iterations.

## Problem specific parameters

---

- ◆ Neighborhood Structure:
  - To ensure convergence every solution should be “reachable” from every other.
  - At each iteration, the algorithm requires the generation of a neighboring feasible solution. Potentially computationally expensive.
  - Small neighborhoods are to be encouraged.

## Problem specific parameters

---

- ◆ The Cost Function: (Energy)
  - Evaluated at every iteration  
⇒ quick and efficient computation.
  - Often unnecessary to recalculate the complete cost function for the new solution.
- ◆ Solution Space:
  - From empirical results:
    - the number of iterations required depends on the size of the solution space.
    - Solution space should be kept small.  
(reducing problem to sub-problems)

## The Travelling Salesman Problem

---

- ◆ Simulated annealing has been applied with success to the travelling salesman problem (TSP).
- ◆ Aim: to find the shortest path that visits all cities from a given set only once.
- ◆ This is a discrete problem
  - (gradient descent cannot be used because there is no way to represent a non-continuous function as n-dimensional surfaces)

## The Travelling Salesman Problem

---

- ◆ Cities are numbered  $1 \dots N$
- ◆ Any permutation  $\pi$  of the numbers  $1 \dots N$  is a valid route.
- ◆ Cost function is the total length of the trip

$$E = c_{\pi(1)\pi(N)} + \sum_{i=1}^{N-1} c_{\pi(i)\pi(i+1)}$$

where  $c_{ij}$  is the distance between  $i$  and  $j$

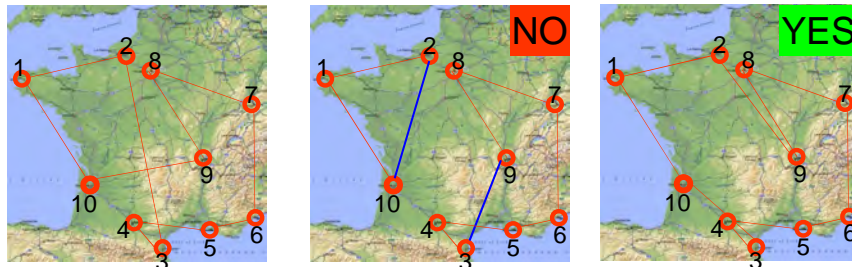
## The Travelling Salesman Problem

---

- ◆ A  $k$ -neighborhood is defined by removing  $k$  links and replacing them by a different set of  $k$  links
  - The feasibility of the trip should be maintained
  - $k > 3$   
There are a number of ways of reconnecting.
  - $k = 2$   
There is only one way of reconnecting the links.

## The Travelling Salesman Problem

- ◆ Links (2,3) and (9,10) are removed, new valid trip iff replaced by (2,9) and (3,10).



$(\pi(i)\pi(i+1))$  and  $(\pi(j)\pi(j+1)) \rightarrow (\pi(i)\pi(j))$  and  $(\pi(i+1)\pi(j+1))$

## TSP: 2-Neighborhoods (1/2)

- ◆ All 2-neighbors can be completely defined by 2 indices  $i$  and  $j$  such that  $i < j$
- ◆ The size of the solution space is  $(N-1)!/2$
- ◆ The size of the neighborhood is  $N(N-1)/2$ 
  - large reduction of neighborhood size.
  - Any tour can be obtained from any other by a sequence of such exchanges.
  - Neighboring solution can be easily generated by randomly selecting values for both  $i$  and  $j$ .

## TSP: 2-Neighborhoods (2/2)

---

- ◆ The computational requirement of the cost function can also be reduced by computing

$$\delta E = c_{\pi(i)\pi(j)} + c_{\pi(i+1)\pi(j+1)} - c_{\pi(i)\pi(i+1)} - c_{\pi(j)\pi(j+1)}$$

instead of

$$\delta E_{\theta} = E_{\theta} - E_{\theta-1} \text{ where } E_{\theta} = c_{\pi_{\theta}(i)\pi_{\theta}(N)} + \sum_{i=1}^N c_{\pi_{\theta}(i)\pi_{\theta}(i+1)}$$

- ◆ Starting solution for simulated annealing can be obtained by generating a random permutation of the city indices.

## TSP: Remarks

---

- ◆ Alternative neighborhood structures:
  - Path reversal
  - Path replacement
- ◆ Although TSP is NP-complete, simulated annealing finds either the global minimum or a local minimum that cannot be significantly improved upon in polynomial time.
- ◆ Simulated annealing can incorporate additional constraints by changing the cost function.

## SA for constrained problems

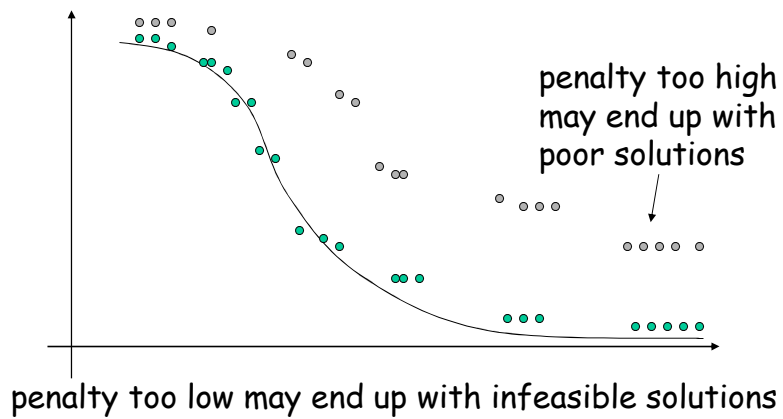
### ◆ Examples:

- The colleagues: Guys C and E will pull each other's hair if seated next to each other.
- The TSP: city 2 has to be visited after city 6, city 7 has to be visited between 7am and 9am, no roads go from city 8 to city 3, etc
- What do we do then?

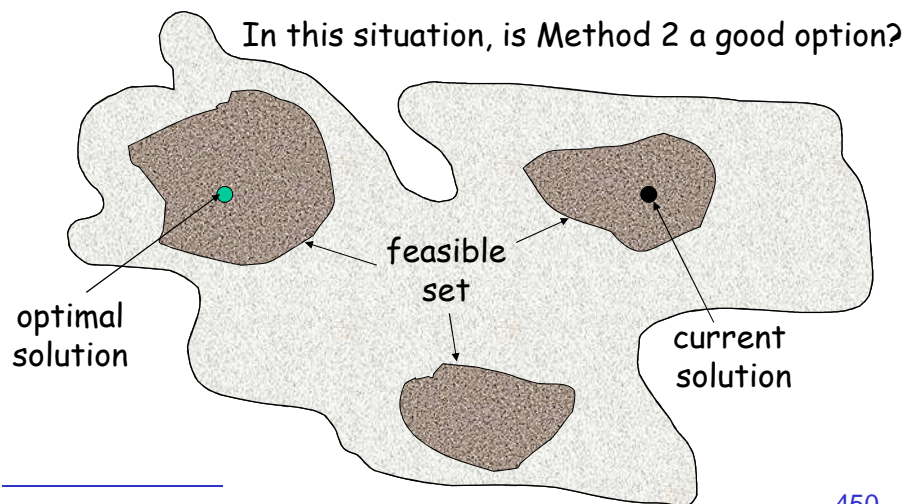
## Method One: penalty

- ◆ How should be penalise infeasibilities?
- ◆ What happen if the penalty is too big?
- ◆ What happen if the penalty is not big enough?
- ◆ How do we determine what penalty to use?
- ◆ What if there are more than one constraints?

## Method One: penalty

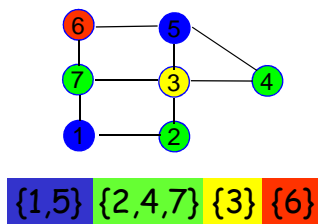


## Method Two: Avoid infeasibility



## The Node Coloring Problem (NCP)

- ◆ Well-known problem in graph theory (applications in time-tabling and scheduling)
- ◆ Problem: Allocate a “color” (label) to nodes s.t. adjacent nodes have different color.



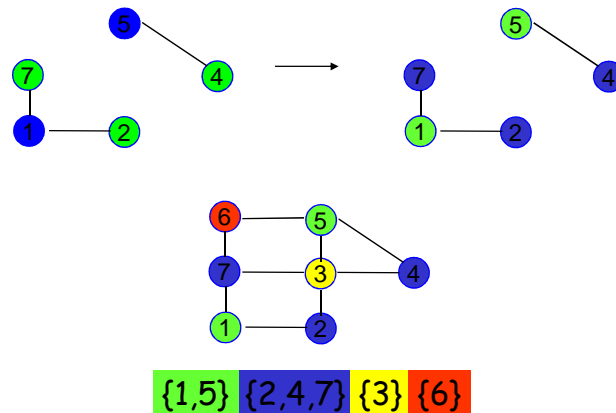
## The NCP: Variants (1/2)

- ◆ Neighborhood of a solution:
  - swapping some nodes between 2 subsets
  - BUT: must be done in a feasible manner.
  - solutions may fall outside the solution space!
- ◆ Use Kempe chains to partition the graph into disconnected graphs
  - BUT: computationally expensive process



## Kempe Chains

### ◆ Blue-Green Kempe Chains



## The NCP: Variants (2/2)

- ◆ Use cost functions that ensures feasible solution are CPU expensive.
  - Natural cost function:  
The number of subsets in the partition
  - But: not a good choice since unaffected unless a subset becomes empty!
- ◆ The cost function should encourage large sets but discourage infeasible coloring

## Applications

---

- ◆ Simulating thermodynamics
- ◆ Finding optimal schedules for classes,
- ◆ Locating routers and repeaters for telecommunication networks
- ◆ Planning land development (forest, roads,...)
- ◆ Determining the best position for equipment (aerials, water-towers, sirens, etc...)
- ◆ Computer vision (from low level segmentation to high level matching)
- ◆ VLSI design (partitioning, placement,...)

## Pros of Simulated Annealing

---

- ◆ Deals with any cost (energy) function,
  - well suited for large, unstructured problems like Computer Vision and VLSI design.
- ◆ Easy to implement, even for complex problems.
- ◆ Generally gives a 'good' solution.
- ◆ It is possible to give statistical guarantees of finding the optimal solution

## Cons of Simulated Annealing

---

- ◆ Speed.
  - Due to the required number of iteration.
- ◆ Complexity of the cost (energy) function.
- ◆ Random number generator of high quality (i.e. have a very long cycle) is required for good results.