Music Machine Learning

II – Neural networks

Master ATIAM - Informatique

Philippe Esling (esling@ircam.fr)

Maître de conférences – UPMC

Equipe représentations musicales (IRCAM, Paris)

UPMC SORBONNE UNIVERSITÉS master ATIAM ircam Centre Pompidou

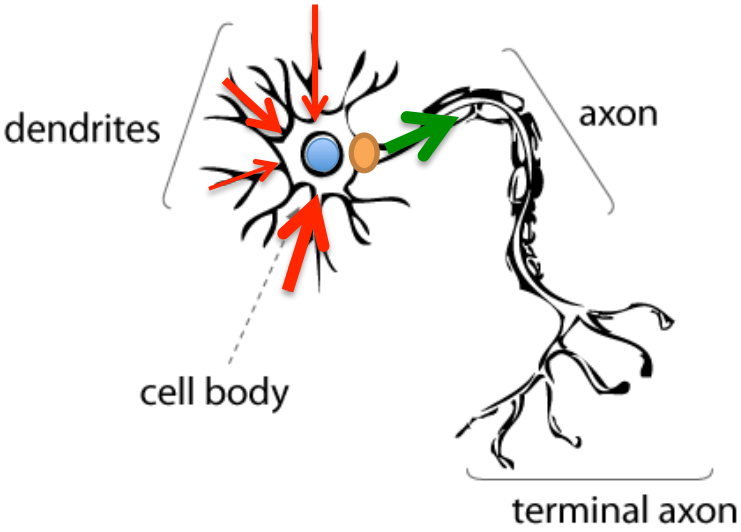
Neural Networks

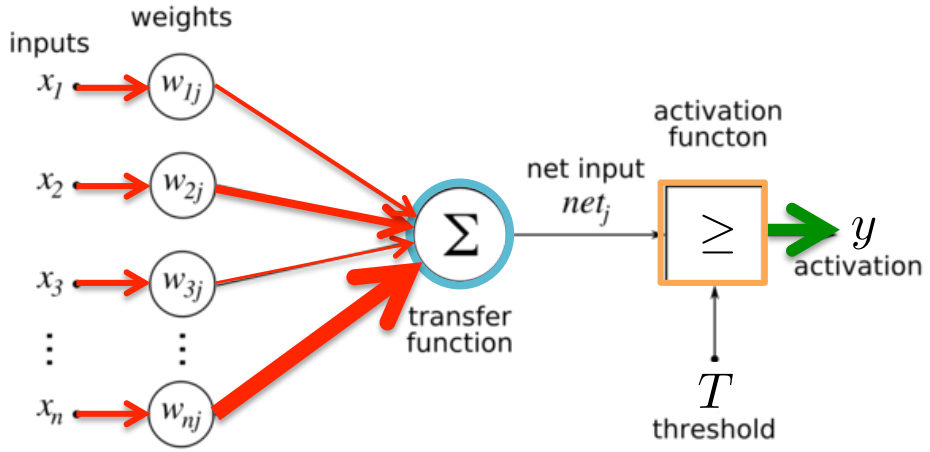
- An attempt to mimic « intelligence » in biology
- You might have seen this in crappy movies
- So why modeling biological neurons ?...
- Well it is the only « intelligent » thing we know ☺

- First model of neuron trace back to 1943 (McCulloch & Pitts)
- Then went slightly into disappearance
- Backpropagation algorithm & NetTalk
- Sparked back the interest ~1990

- Nowadays **deep learning** and **connexionnism**
- Seems like the best way to go upwards strong AI.

Neural Networks





$$y = \left(\sum_{i \in in_x} w_i . x_i \geq T \right)$$

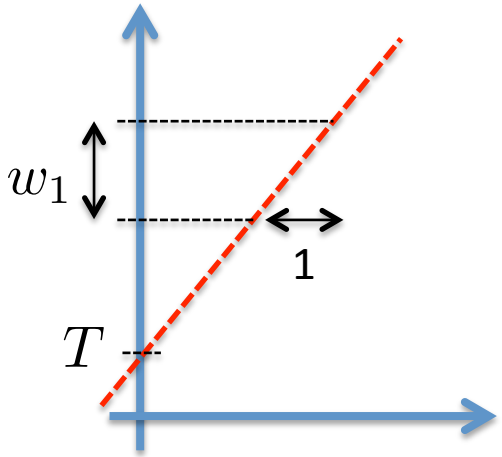
- Seminal model by McCulloch & Pitts (**dating back to 1943**)
- So what does a single neuron mathematically do ?

Neural Networks

$$y = \left(\sum_{i \in in_x} w_i.x_i \geq T \right) \qquad \sum_{i \in in_x} w_i.x_i = T$$

So what is the interpretation of this equation?
Let's say I only have a single weight (+ threshold)?

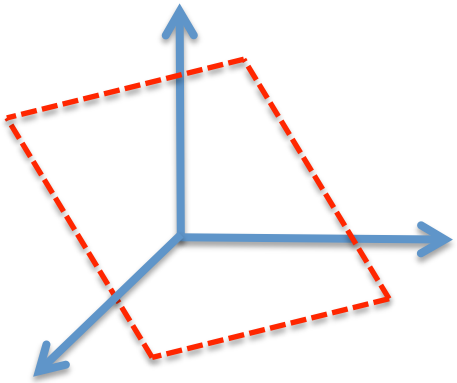
$$y = w_1.x_1 + T$$



With N inputs, a single neuron simply defines a **N-dimensional hyperplane**

$$y = w_1.x_1 + w_2.x_2 + T$$

$$y = \sum_i w_i x_i + T$$



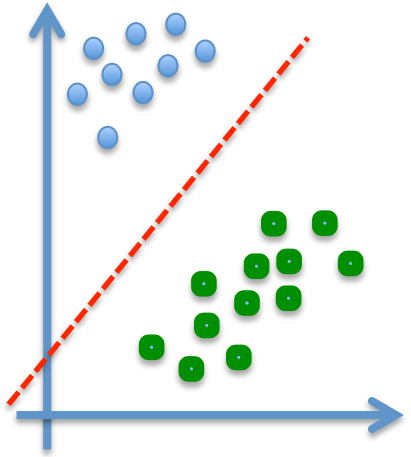
So essentially a space division (classification) but also a function approximation (as the sum will give an output whatever X comes in)

Neural Networks

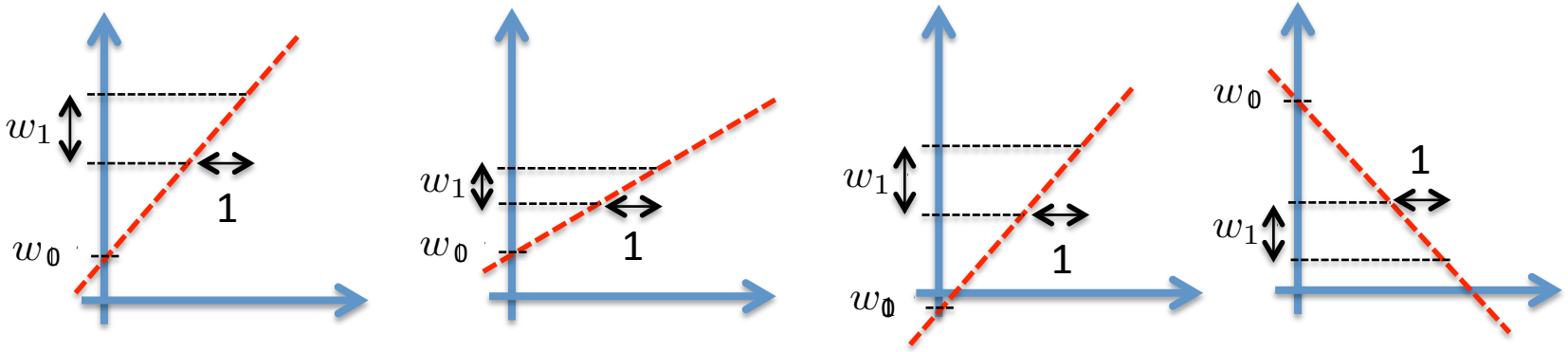
- So neurons allow to divide the space …
- So neurons allow to perform classification
- But also function approximation

(as the sum will give an output for all X)

- Only *linearly separable problems*



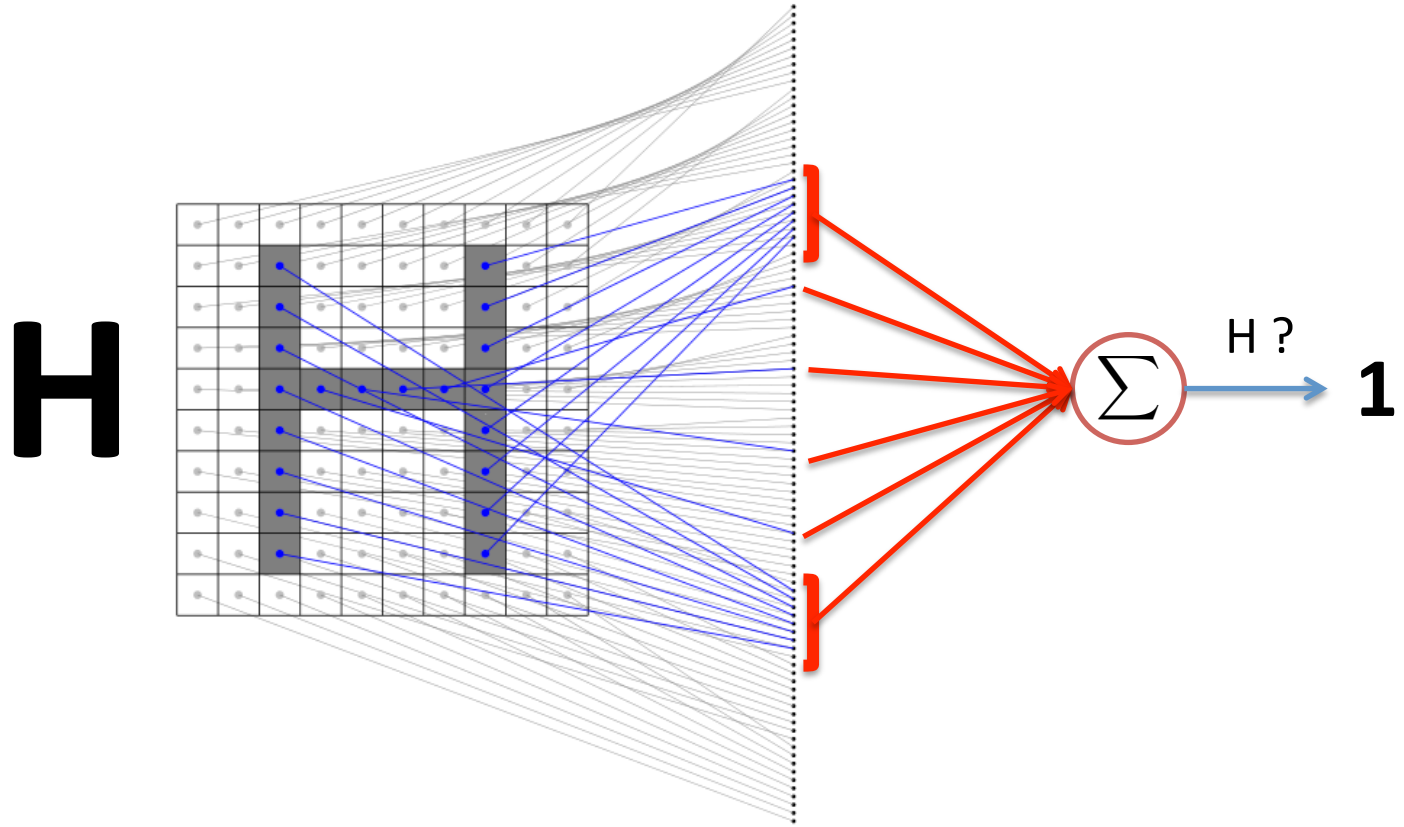
But how can a neuron *learn* ?



By tweaking the weights we can separate any two classes (only if they are linearly separated)

Neural Networks

Optical Character Recognition (OCR) example



Neural Networks

- But how to learn the weights?
- Learning is possible through weight *optimization*

This is possible as $\bar{y} = f(\bar{x}, \bar{w})$

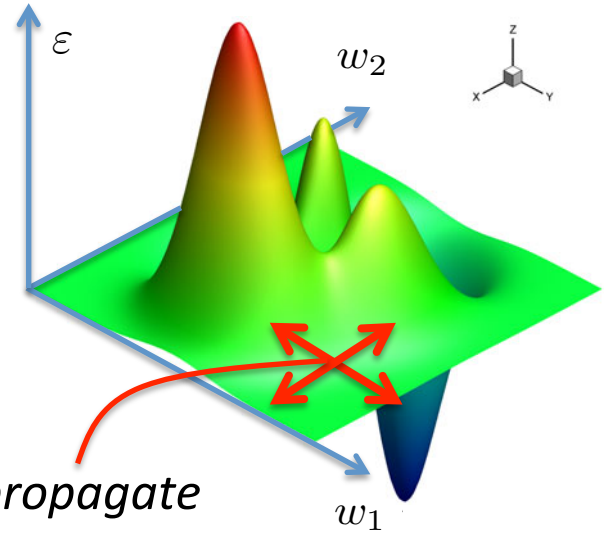
- This is done through a **training phase**
- We have a set of input vectors $\mathbf{X} = \{x_1, ..., x_n\}$
- For which we know the desired output $\mathbf{D} = \{d_1, ..., d_n\}$
- Therefore we can compute the error $\varepsilon = \sum_i (d_i - x_i)^2$

- So training amounts to adjust $\bar{w}_i$ so that $y_i \sim d_i$

- We can do that easily using hill-climbing search ☺

Neural Networks

- Gradient descent is path following
- Need to find the lowest error point
- This amounts to error minimization
- Look around different directions
- Compute the difference in error



$$\Delta\bar{w} = r\left(\frac{\delta\mathcal{P}}{\delta w_1}, \frac{\delta\mathcal{P}}{\delta w_2}\right)$$

« Climb hills » and *back-propagate*

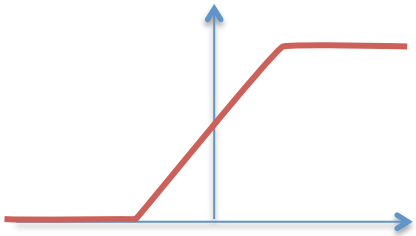
Two problems remain from the original definition.
Problem 1 = There is a threshold T for each neuron
Problem 2 = Conditions on the error shape for hill climbing
The space needs to be continuous

Neural Networks

Problem 1 = There is a threshold T for each neuron
Solution : We consider T as a virtual weight (w_0)

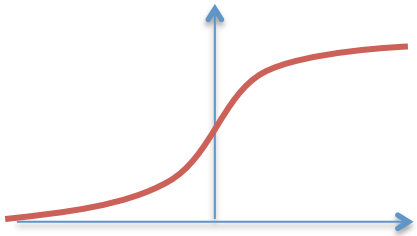
Problem 2 = Conditions on the error shape for hill climbing
The space needs to be continuous
Solution : Simply needs to use different activation functions

Piecewise linear



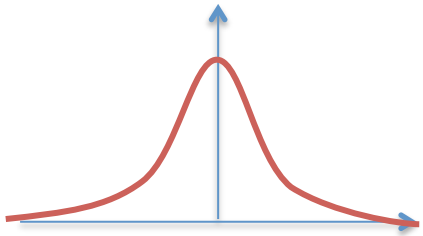
$$\mathcal{T}(x) = \begin{cases} 0 & \text{if } x \le x_{\min} \\ mx+b & \text{if } x_{\max} > x > x_{\min} \\ 1 & \text{if } x \ge x_{\max} \end{cases}$$

Sigmoïd



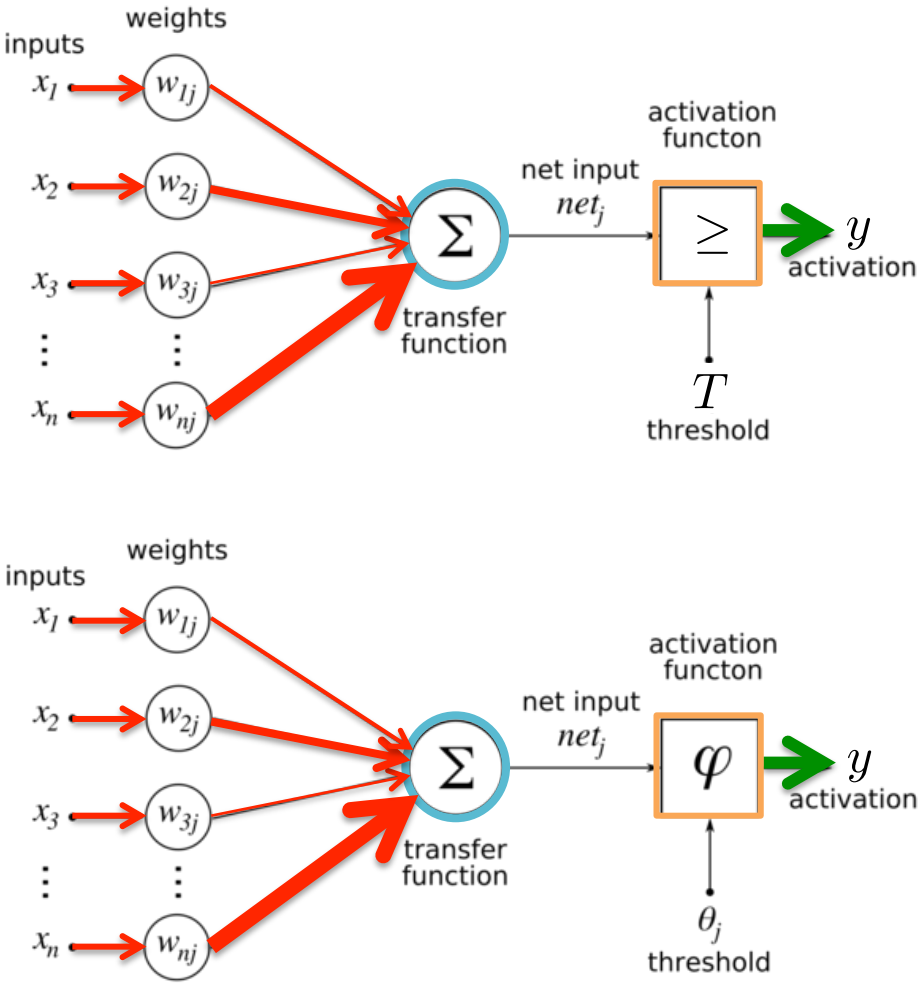
$$\mathcal{T}(x) = \frac{e^x}{1+e^{-\beta x}}$$

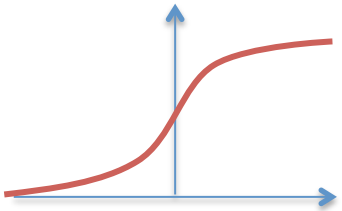
Gaussian



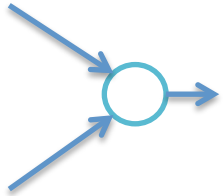
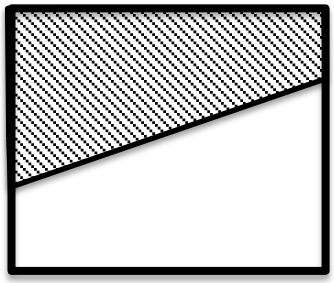
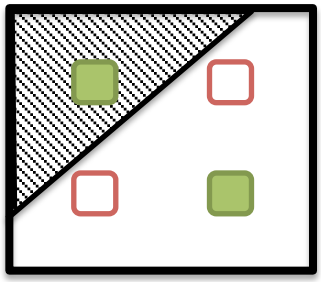
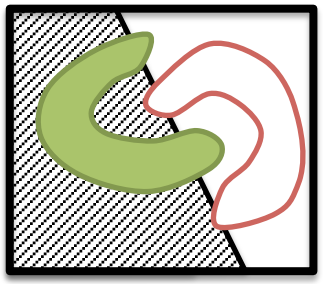
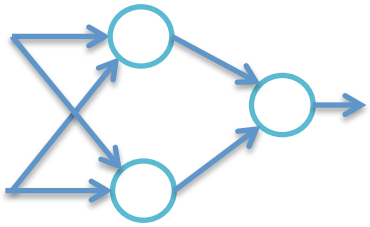
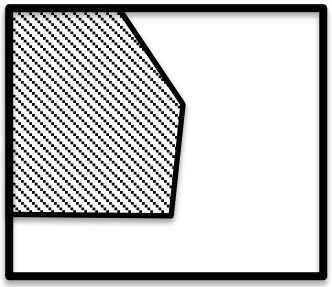
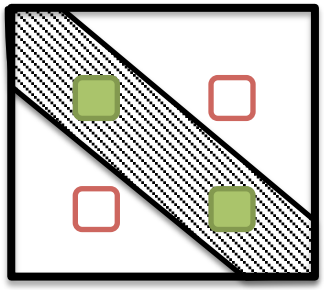
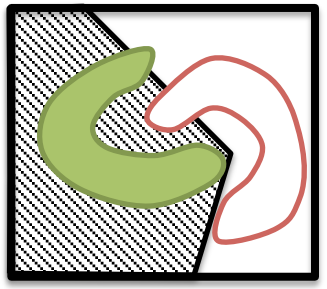
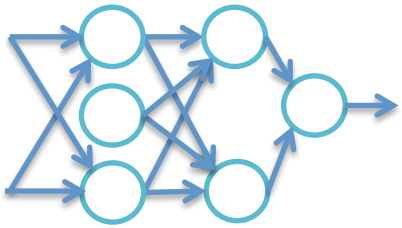
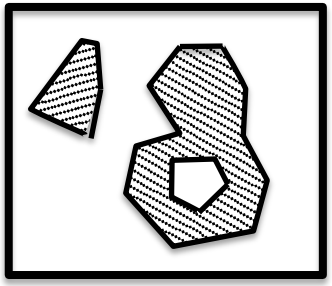
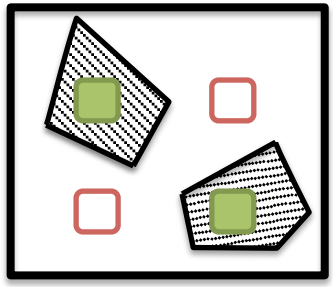
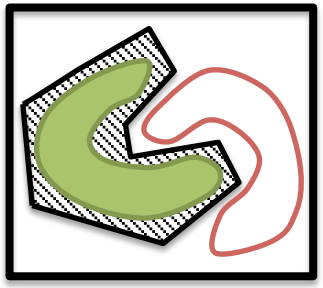
$$\mathcal{T}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Neural Networks

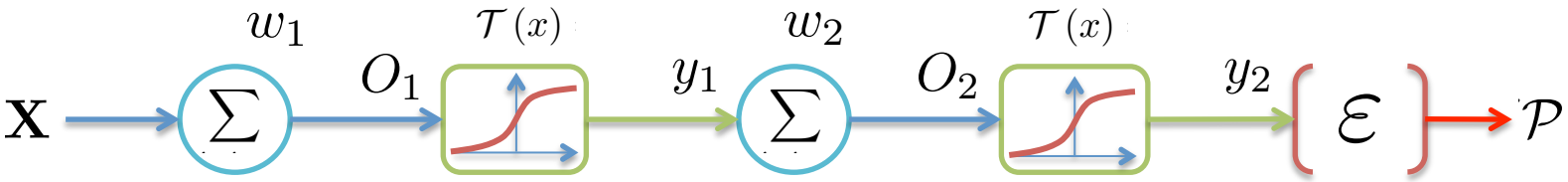




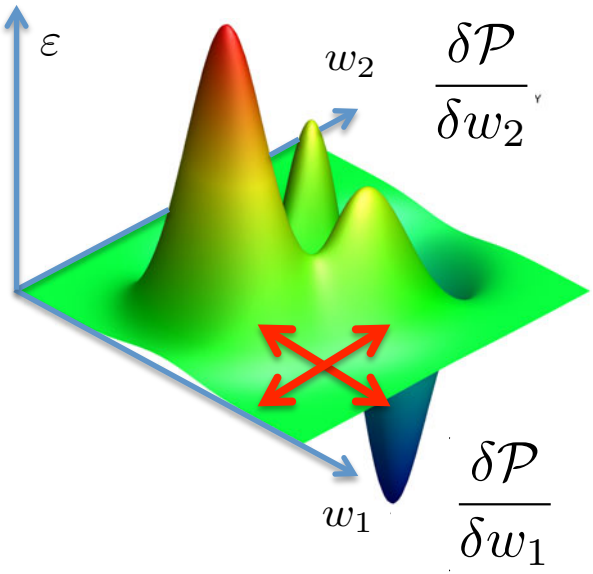
Neural Networks

Architecture	Space region	XOR Problem	General shapes
 *Single layer*			
 *Two layers*			
 *Three layers*			

Neural Networks



To learn, we have to look at the effect of weight modification on errors

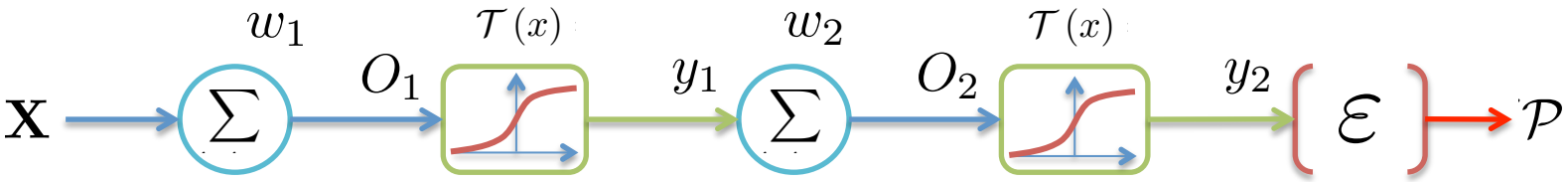


So we need the derivatives

$$\Delta \bar{w} = r \left(\frac{\delta \mathcal{P}}{\delta w_1}, \frac{\delta \mathcal{P}}{\delta w_2} \right)$$

- Look at the shape of error surface
- Evaluate this in each dimension
- Then follow the hills (change weights)
- Back-propagate the effect

Neural Networks



$$\frac{\delta \mathcal{P}}{\delta w_1} = \frac{\delta \mathcal{P}}{\delta y_2} \cdot \frac{\delta y_2}{\delta w_1} - \frac{\delta y_2}{\delta O_2} \cdot \frac{\delta O_2}{\delta w_1} - \frac{\delta O_2}{\delta y_1} \cdot \frac{\delta y_1}{\delta w_1} - \frac{\delta y_1}{\delta O_1} \cdot \frac{\delta O_1}{\delta w_1}$$

$2\,(y_2 - d)$ $\quad y_2\,(1 - y_2)$ $\quad w_2$ $\quad y_1\,(1 - y_1)$ $\quad \mathbf{X}$

$$\frac{\delta \mathcal{P}}{\delta w_2} = \frac{\delta \mathcal{P}}{\delta y_2} \cdot \frac{\delta y_2}{\delta w_2} - \frac{\delta y_2}{\delta O_2} \cdot \frac{\delta O_2}{\delta w_2}$$

$2\,(y_2 - d)$ $\quad y_2\,(1 - y_2)$ $\quad y_1$

- Amount of work is a linear function of the depth of the network
- Sigmoïd helps to simplify the computations

Neural Networks

So how to code your own (single) neuron?

1. Initialize $\bar{w}_i$ and $\mathcal{T}$ to small random values
2. While !(stop condition)
   1. Present patterns $\mathbf{X} = \{x_1, ..., x_n\}$ to neuron
   2. **Propagation:** Evaluate its output and error
   3. **Back-propagation:** Update the weights based on

$$w_j^{t+1} = w_j^t + \eta . (d - y) . x_j$$

learn rate ([0,1]) — **desired** — **output**

Stop conditions
- Iterations number
- $\varepsilon < \tau$
- $\Delta\varepsilon < \tau$

- This extends quite simply to multi-layer networks
- We simply need to take into account the multiple layers
- So the propagation is iterative and back-propagation also

Neural Networks

So how to code your multilayer perceptron

1. Initialize $\bar{w}_i$ to small random values
2. Randomly choose an input pattern from the training set
   1. Present patterns $\mathbf{X} = \{x_1, ..., x_n\}$ to the network
   2. **Propagation:** Evaluate its output and error
   3. Compute the δ_i^L in the *output (last) layer* ($o_i = y_i^L$)

$$\delta_i^L = g'\left(h_i^L\right)\left[\delta_i^u - y_i^L\right]$$

derivative of activation — **Net input to ith unit in Lth layer**

   4. **Back-propagate** by computing deltas for preceding layers

$$\delta_i^l = g'\left(h_i^l\right)\sum_j w_{ij}^{l+1}\delta_j^{l+1} \quad l \in [1 \ldots L-1]$$

   5. Update weights in the corresponding layers

$$\Delta w_{ij}^l = \eta.\delta_i^l y_j^{l-1}$$

Learning rules

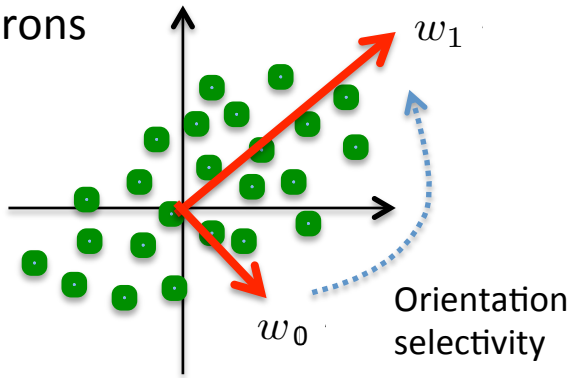
So far we have used the simplest learning rule

1. Error-correction
    1. Learning occurs when the perceptron makes an error
    2. We use the error signal $(d-y)^2$

But other rules can be implemented

2. Hebbian rules
    1. If neurons on both sides of a synapse are activated
    2. Then the synapse strength is increased
    3. Learning is done locally with synchronicity
    4. Weights change only for two connected neurons
    5. This amounts to learn the maximal variance

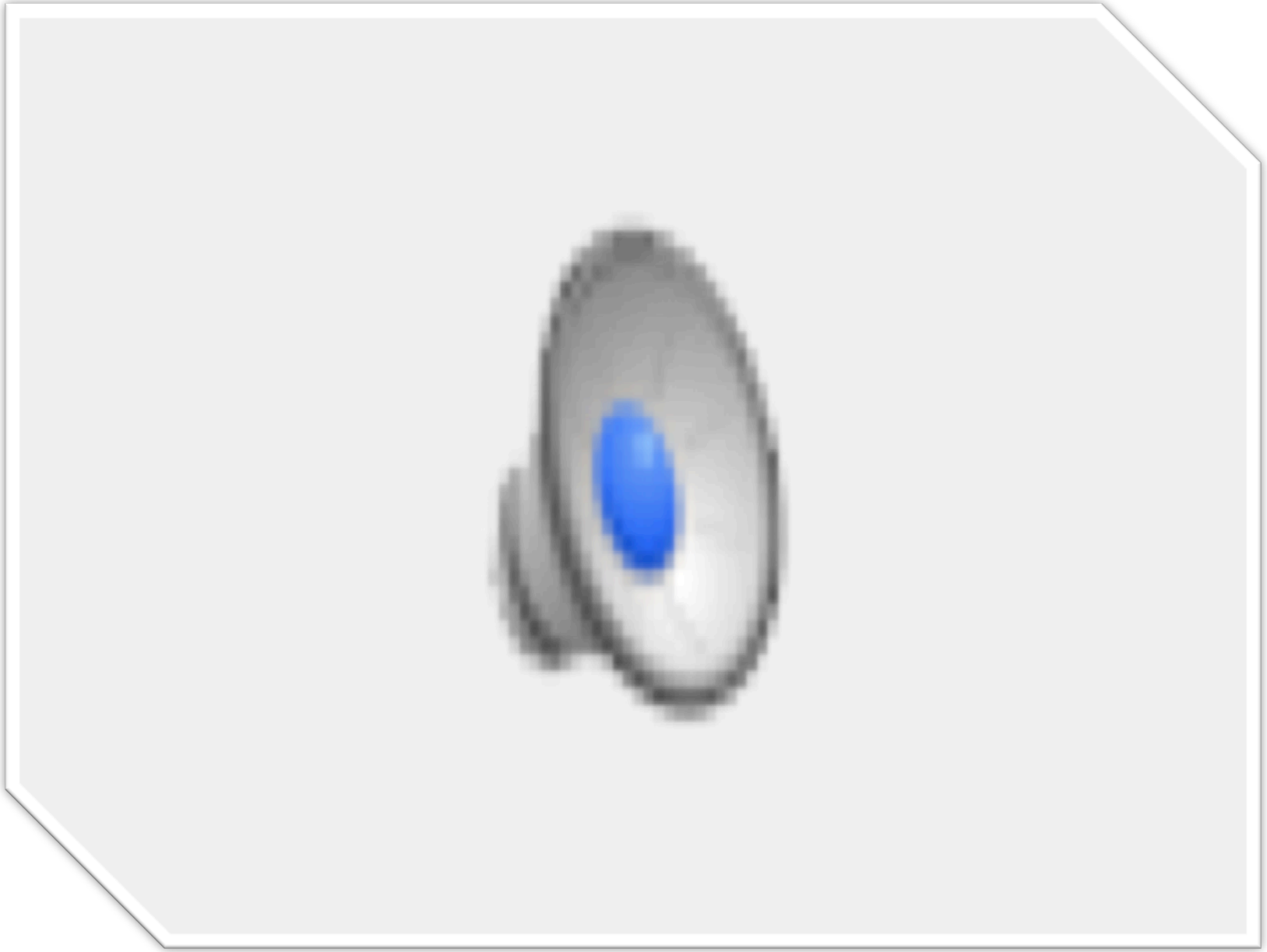
$$\Delta \bar{w} = \eta O \bar{I} \qquad O = \bar{w}^T \bar{I}$$



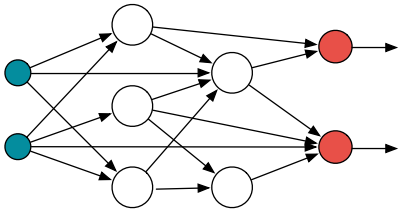
Learning rules

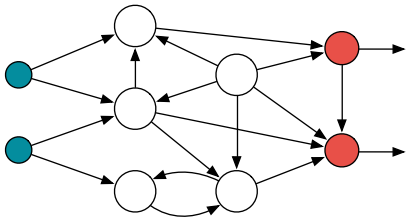
3. Boltzmann learning
    1. Works on symmetric recurrent networks
    2. Each neuron is a stochastic unit
    3. Output based on Boltzmann distribution of statistical mechanics
    4. Two modes exist for each neuron
        1. Clamped: fixed to specific states
        2. Free-runing: Adjust weight so that clamped states satisfy PDF

4. Competitive learning
    1. Unlike other rules, here neurons compete
    2. Only one output unit is active at any given time
    3. Each output connected to all inputs and all other outputs
    4. This implies inhibitory weights and self-feedback
    5. The unit with largest output wins
    6. Implies tradeoff between plasticity and stability



Neural Networks





Neural networks

Feed-forward

Recurrent/feedback

Single-layer perceptron

Multilayer perceptron

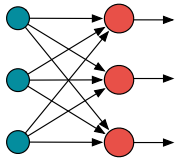
Radial Basis Functions

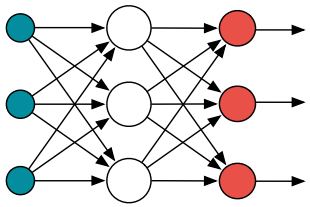
Competitive networks

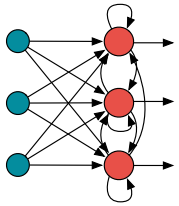
Kohonen's SOM

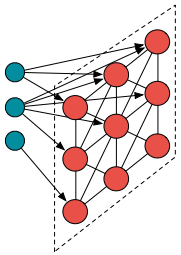
Hopfield network

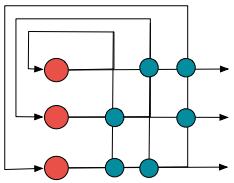
ART models

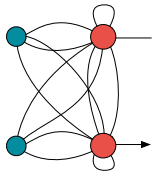








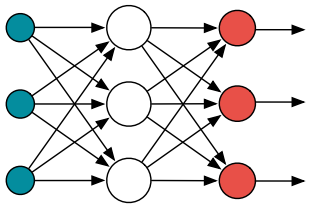


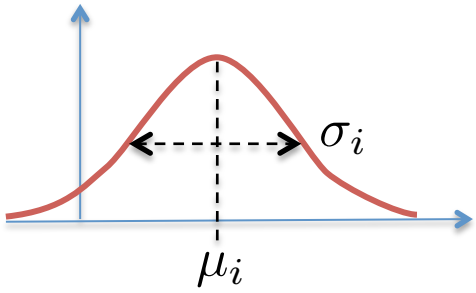


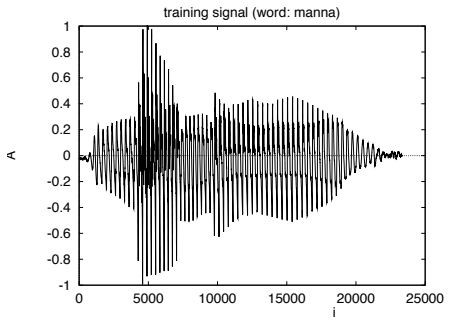
Different architectures

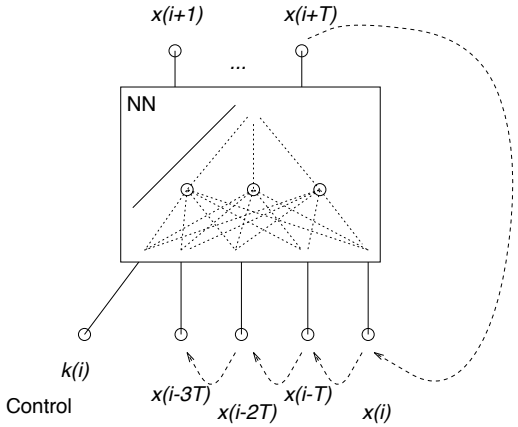
Radial Basis Function (RBF) Network

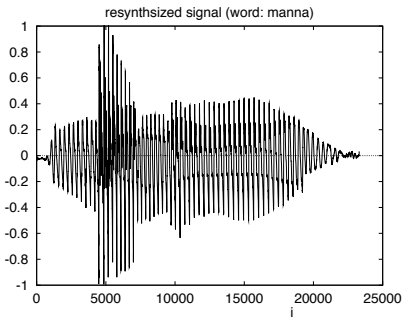
- Special case of feed-forward network
- Follows the multilayer perceptron structure
- Usually 2 layers architecture
- Activation function is a **gaussian kernel**
- Both μ_i and σ_i must be learned
- 2-step hybrid learning strategy
  1. Unsupervised clustering for μ_i and σ_i
  2. Supervised Least Mean Squares (LMS)







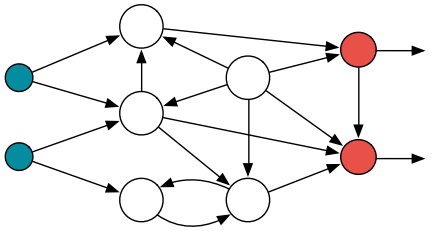


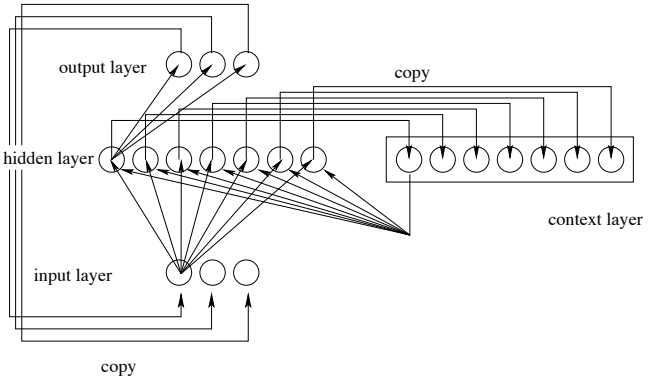


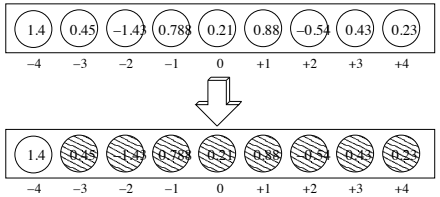
Different architectures

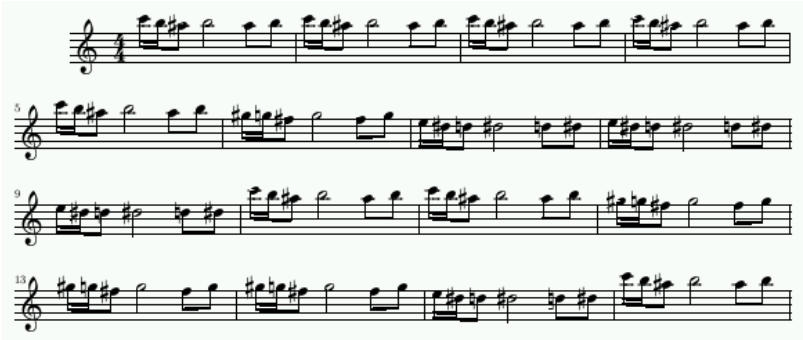
Recurrent/Feedback Network

- Connections can go backwards
- Self-connections are also allowed
- Allows retro-action but can also model
  - Time dependency notions
  - Different network states





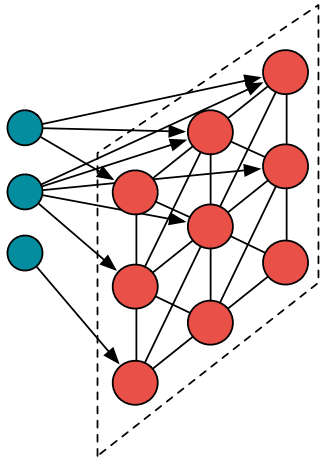


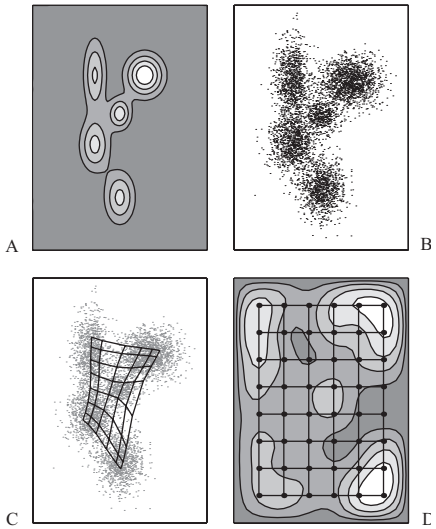


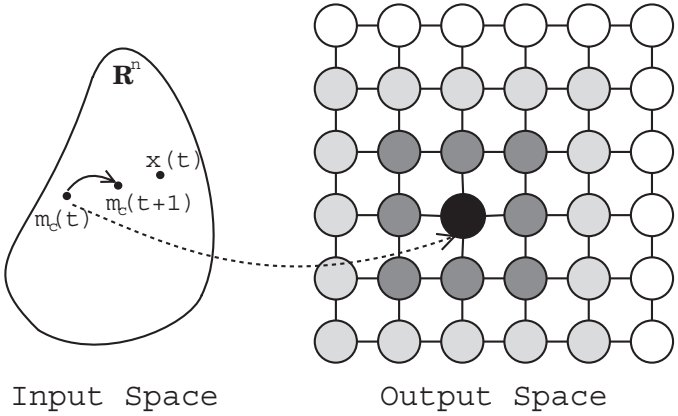
Different architectures

Kohonen's Self-Organizing Maps (SOM)

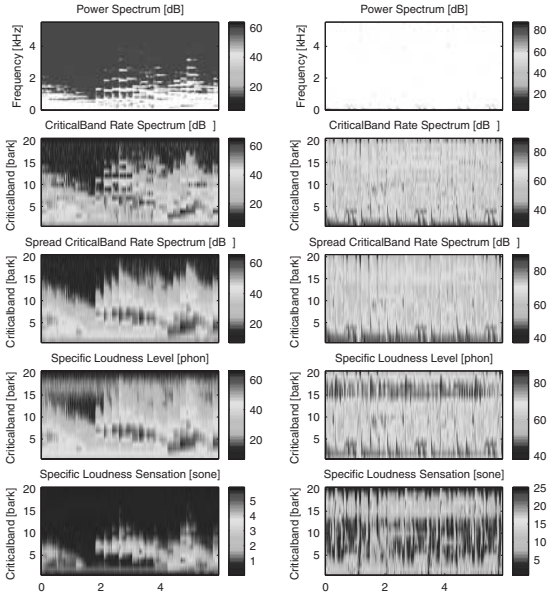
- Each neuron connected to all neurons
- But distance between neurons is kept
- And this distance influences the network
- Models relations of spatial neighborhood
- Local neurons influence more than global
- Embeds a property of topology preservation

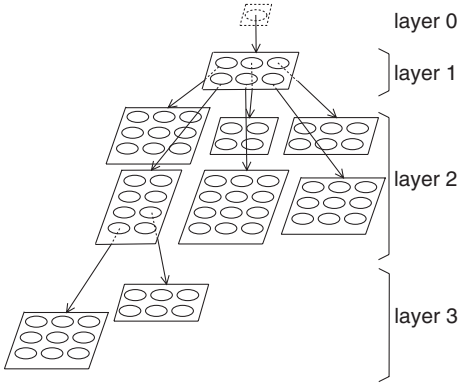


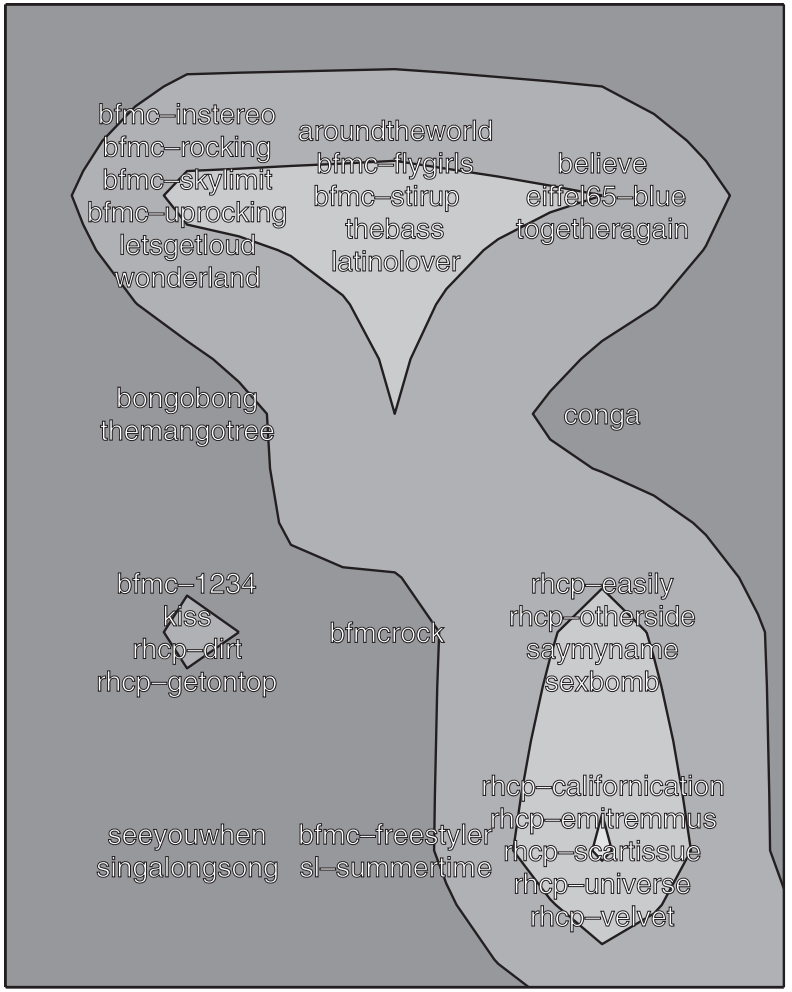




Different architectures







Different architectures

Adaptive Resonance Theory (ART) models

- Form of competitive network
- Not all outputs neuron are used (only if necessary)
- Difference between commited and uncommited
- Only update a category if prototypes simlar
- The extent of similarity is controled by vigilance P