Connectionist Computing COMP 30230/41390

Gianluca Pollastri
office: E0.95, Science East.
email: gianluca.pollastri@ucd.ie

Credits

- **Geoffrey Hinton, University of Toronto.**
  - borrowed some of his slides for "Neural Networks" and "Computation in Neural Networks" courses.

- **Ronan Reilly, NUI Maynooth.**
  - slides from his CS4018.
- **Paolo Frasconi, University of Florence.**
  - slides from tutorial on Machine Learning for structured domains.

Lecture notes on Brightspace

- Strictly confidential...
- Slim PDF version will be uploaded later, typically the same day as the lecture.
- If there is demand, I can upload onto Brightspace last year's narrated slides.. (should be very similar to this year's material)

Books

- No book covers large fractions of this course.

- Parts of chapters 4, 6, (7), 13 of Tom Mitchell's "Machine Learning"
- Parts of chapter V of Mackay's "Information Theory, Inference, and Learning Algorithms", available online at:

http://www.inference.phy.cam.ac.uk/mackay/itprnn/book.html

- Chapter 20 of Russell and Norvig's "Artificial Intelligence: A Modern Approach", also available at:

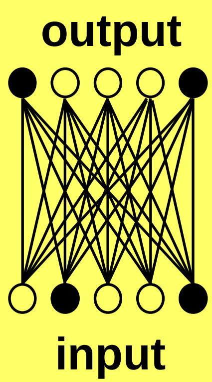
http://aima.cs.berkeley.edu/newchap20.pdf

- More materials later..

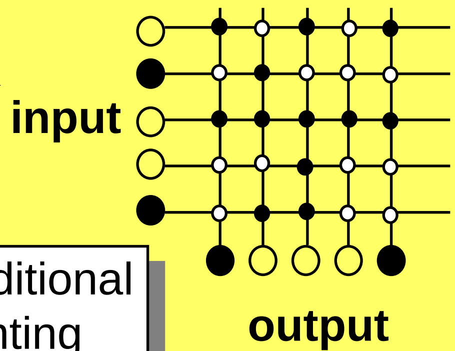
Marking

- **3 landmark papers to read, and submit a 10-line summary on Brightspace about: each worth 6-7%**
- **a connectionist model to build and play with on some sets, write a report: 30%**
- **Final Exam in the RDS (50%)**

Associators



A grid is the traditional way of representing associators



Learning in associators

- Learning involves a variation of Hebb's rule:

$$\Delta w_{ji} = \eta y_j x_i$$

- y_j : j-th output
- x_i : i-th input

.. finally

- If the input patterns are orthogonal the associator shows perfect memory:

$$x^{(k)T} \cdot x^{(p)} = 0$$

$$y^{(p)'} = y^{(p)}$$

- If not orthogonal, “crosstalk”

Iterative learning procedure

- When the number of input patterns increases and the patterns themselves are not orthogonal, the "one-shot" learning approach ceases to be optimal.
- An alternative (suggested by Kohonen, 1977) is to repeatedly iterate through a set of patterns, making small weight changes, and attempting to minimise some error measure.

Minimise squared error

- **A possible error function:**

$$E = \frac{1}{2}\sum_{p=1}^{P}\sum_{k}(y_k^{(p)} - y_k^{(p)'})^2$$

Gradient descent

- We can measure the slope of the error function with respect to the parameters, and follow the direction of steepest descent:

$$\nabla_w E = \left(\frac{\partial E}{\partial w_{ji}}\right)$$

after a minor struggle

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}}\Big(\frac{1}{2}\sum_{p=1}^{P}\sum_{k}(y_k^{(p)} - y_k^{(p)'})^2\Big) =$$

$$\frac{1}{2}\sum_{p=1}^{P}\sum_{k}\frac{\partial}{\partial w_{ji}}(y_k^{(p)} - y_k^{(p)'})^2 =$$

$$\sum_{p=1}^{P}(y_j^{(p)'} - y_j^{(p)})\frac{\partial y_j^{(p)'}}{\partial w_{ji}} =$$

$$\sum_{p=1}^{P}(y_j^{(p)'} - y_j^{(p)})x_i^{(p)}$$

- **Fairly similar to Hebb's law**

$$\Delta w_{ji} = -\eta \sum_{p=1}^{P} (y_j^{(p)'} - y_j^{(p)}) x_i^{(p)} =$$

$$\eta \sum_{p=1}^{P} (y_j^{(p)} - y_j^{(p)'}) x_i^{(p)}$$

- **Iterate until satisfied:**

$$w_{ji} = w_{ji} + \Delta w_{ji}$$

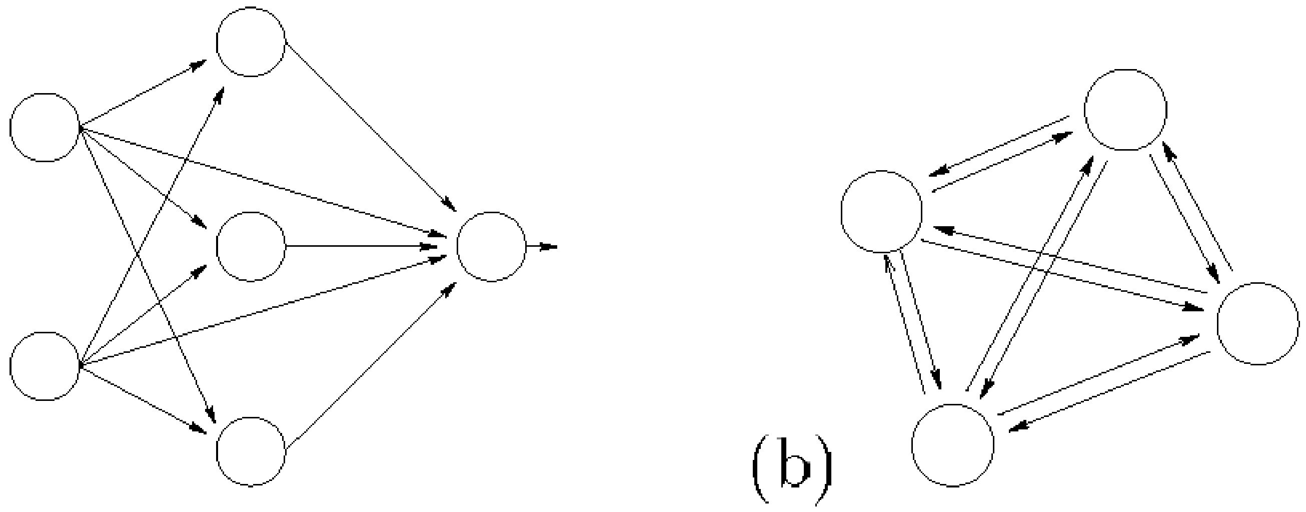
Gradient descent and associators

- **Complexity of gradient computation is minimal: o(nm)**
- **BUT, it is unclear how many steps along the gradient need to be taken…**

- **Much better storage of examples than with one-shot learning, though typically there will be residual error.**

Feedforward and feedback networks

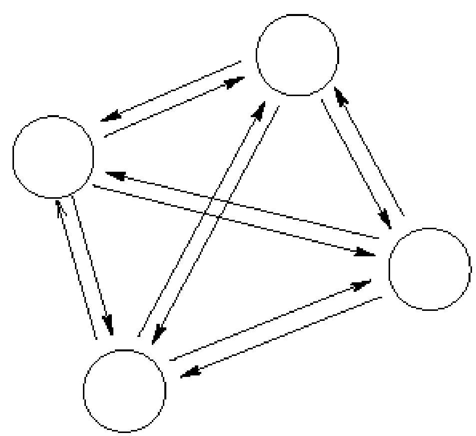
- **FF is a DAG (Directed Acyclic Graph). Perceptrons, Associators are FF networks.**
- **FB has loops (i.e., not Acyclic)**



(a) (b)

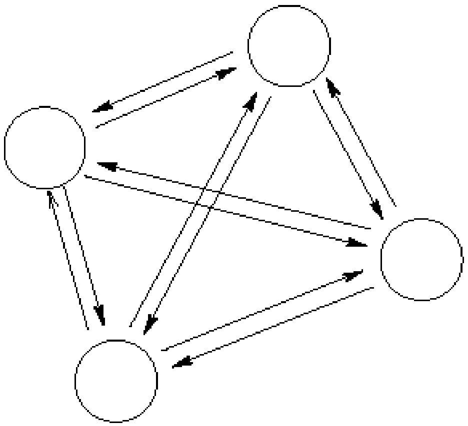
Hopfield Nets

- **Networks of <u>binary threshold</u> units.**
- **Feedback networks: each units has connections to all other units except itself.**



Hopfield Nets

- w_{ji} is the weight on the connection between neuron i and neuron j.
- Connections symmetric, i.e. $w_{ji} = w_{ij}$



Stable states in Hopfield nets

- **These networks are not FF. There is no obvious way of sorting the neurons from inputs to outputs (every neuron is input to all other neurons).**
- **In which order do we update the values on the units?**
  - ***Synchronous* update: all neurons change their state simultaneously, based on the current state of all the other neurons.**
  - ***Asynchronous* update: e.g. one neuron at a time.**
- **Is there a stable state (i.e. a state that no update would change)?**

Energy function in Hopfield nets

- **Given that the connections are symmetric ($w_{ij} = w_{ji}$), it is possible to build a global *energy function*. According to it each configuration (set of neuron states) of the network can be scored.**
- **It is possible to look for configurations of (possibly locally) minimal energy. In fact the whole space of weights is divided into *basins of attraction,* each one containing a minimum of the energy.**

The energy function

- **The global energy is the sum of many contributions. Each contribution depends on one connection weight and the binary states of two neurons:**

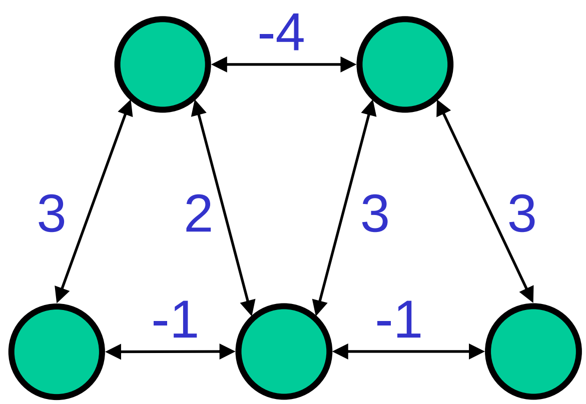
$$E = -\frac{1}{2}\sum_{i,j} w_{ij} y_i y_j - \frac{1}{2}\sum_i b_i y_i$$

- **The simple energy function makes it easy to compute how the state of one neuron affects the global energy (it is the activation of neuron!):**

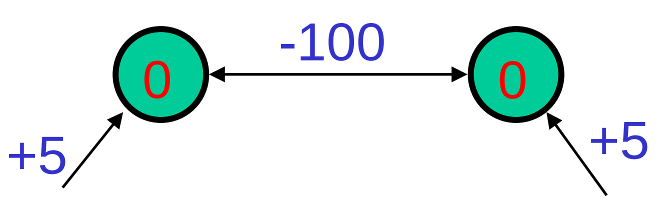
$$E(y_i = -1) - E(y_i = 1) = \sum_j w_{ij} y_j + b_i$$

Settling into an energy minimum

- **Pick the units one at a time (asynchronous update) and flip their states if it reduces the global energy.**



- **If units make simultaneous decisions the energy could go up.**

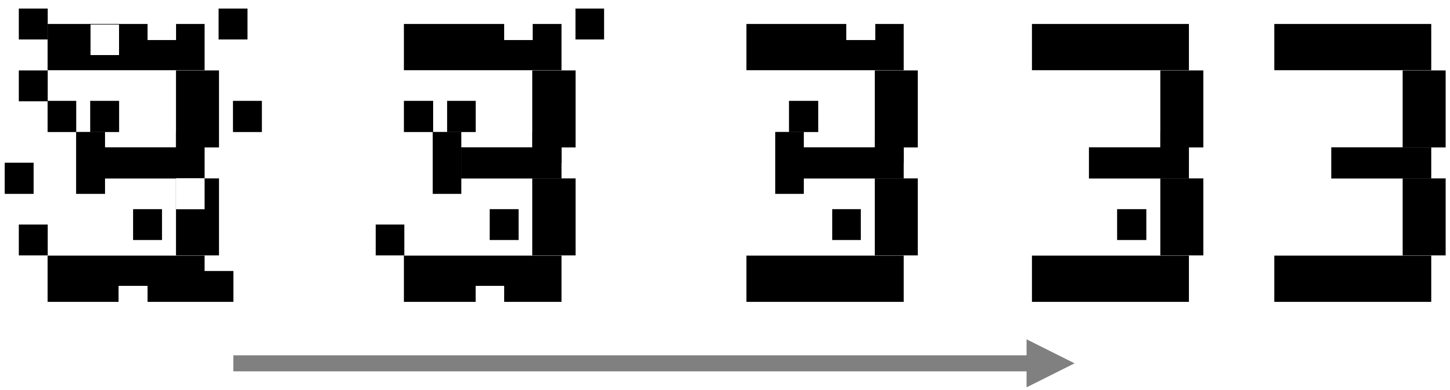


Hopfield network for storing memories

- Memories could be energy minima of a neural net.
- The binary threshold decision rule can then be used to clean up incomplete or corrupted memories.
  - This gives a content-addressable memory in which an item can be accessed by just knowing part of its content
  - Is it robust against damage?

Example

Training set



The corrupted pattern for "3" is input and the network cycles through a series of updates, eventually restoring it.

Storing memories (learning)

- If we want to store a set of memories:

$$y^{(1)}, .., y^{(p)}, .., y^{(P)}$$
$$y^{(p)} = (y_1^{(p)}, .., y_m^{(p)})$$

- if the states are –1 and +1 then we can use the update rule:

$$\Delta w_{ji} = \eta \sum_p y_i^{(p)} y_j^{(p)}$$

Example

- **Two patterns:**

$y^{(1)}$=(1 –1 1) and $y^{(2)}$= (-1 1 –1)

Say we want η=1/neurons=1/3

What is W?

Example

0	–2/3	2/3
-2/3	0	–2/3
2/3	–2/3	0

?

Storing memories (learning)

- **If neuron states are 0 and 1 the rule becomes slighty more complicated:**

$$\Delta w_{ji} = 4\eta \sum_p (y_i^{(p)} - \frac{1}{2})(y_j^{(p)} - \frac{1}{2})$$

Hopfield nets with sigmoid neurons

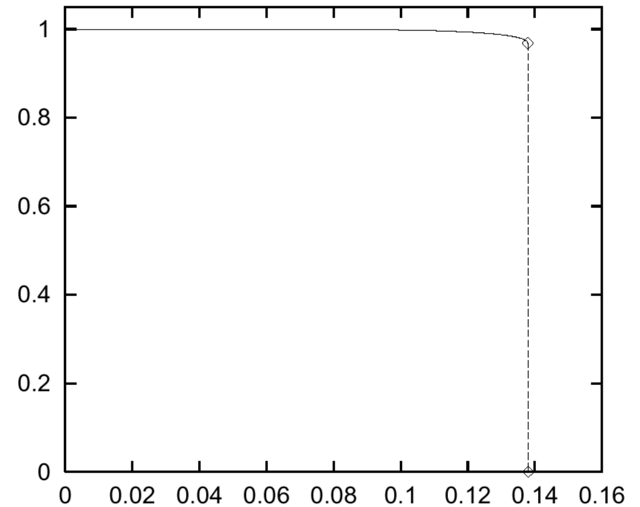
- Perfectly legitimate to use Hopfield nets with sigmoid neurons instead of binary-threshold- ones.
- The learning rule remains the same.

Learning problems

- Each time we memorise a configuration, we hope to create a new energy minimum.
- But what if two nearby minima merge to create a minimum at an intermediate location (spurious minima)?
- How many minima can we store in a network before they start interfering with each other?
- Can other minima coexist with the learned ones?

Critical state

- **There is a critical state around P/N=0.14, where P=memories and N=number of neurons. Above it the probability of failure increases drastically.**



Critical state

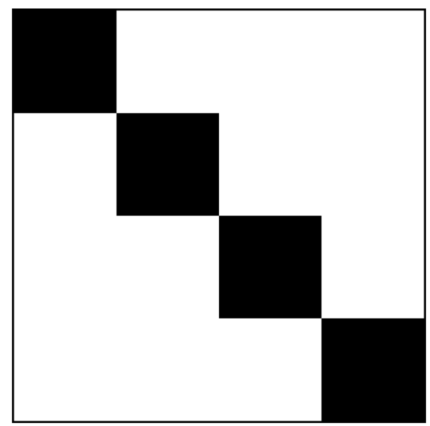
- P/N > 0.14 : no minimum is related to the learned patterns.
- 0.14 > P/N > 0.05 : both learned and other minima. Other minima tend to dominate.
- 0.05 > P/N > 0 : both learned and other minima. Learned minima dominate (lower energy).

An iterative storage method

- Instead of trying to store vectors in one shot as Hopfield does, cycle through the training set many times and make small weight changes.
  – This uses the capacity of the weights more efficiently.
  – Very much like Kohonen's extension to Linear Associators.

Example

- **Say we have 4 patterns of size 4:**
- **(1, -1, -1, -1)**
- **(-1, 1, -1, -1)**
- **(-1, -1, 1, -1)**
- **(-1, -1, -1, 1)**



- **We build the sigmoid Hopfield net based on the 4 patterns (Matlab nnet toolbox).**
- **Incidentally, here one-shot learning would not work: try.**

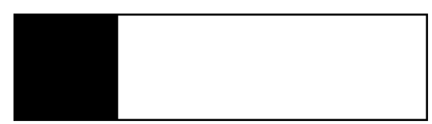
$$\Delta w_{ji} = \eta \sum_p y_i^{(p)} y_j^{(p)}$$

Example

- **Let’s now start from some state Y for the neurons, and watch the network evolve.**
- **Y=(1 0 0 0)**

Steps (1 update/neuron):

1: (0.4999 -0.6620 -0.6620 -0.6620)
2: (0.5022 -0.8476 -0.8476 -0.8476)
3: (0.6351 -0.9332 -0.9332 -0.9332)
4: (0.8186 -1.0000 -1.0000 -1.0000)
5: (1 -1 -1 -1) **converged to pattern 1!**



Example (2)

- **Different starting point:**
- **Y=(0 0 0 0)**

Steps:

1: (-0.4273 -0.4273 -0.4273 -0.4273)

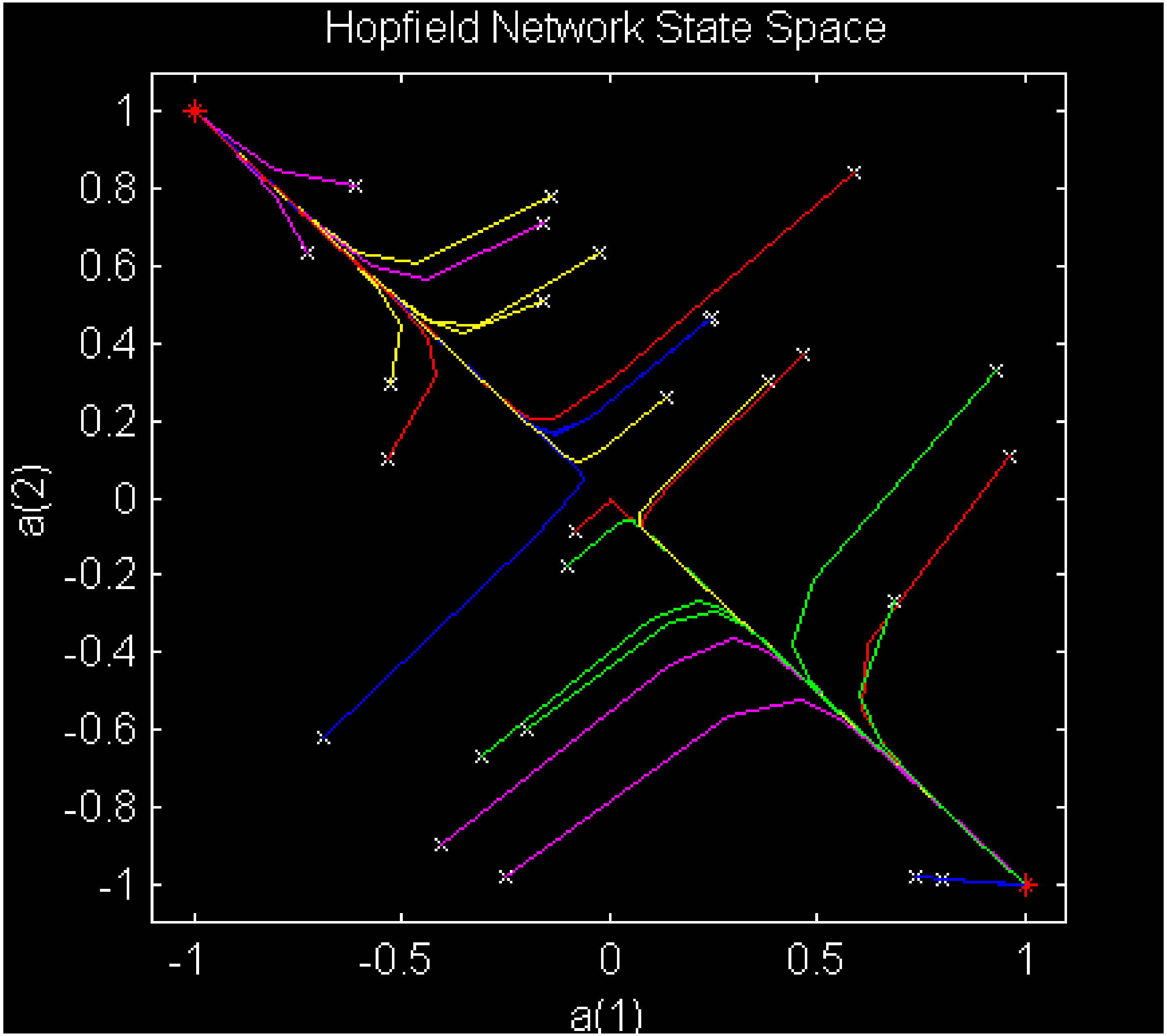
2: (-0.5226 -0.5226 -0.5226 -0.5226)

3: (-0.5439 -0.5439 -0.5439 -0.5439)

4: (-0.5486 -0.5486 -0.5486 -0.5486)

..

7: (-0.55 -0.55 -0.55 -0.55) **stuck in the middle!**



Hopfield Network State Space
a(2)
a(1)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1

Example (3)

- Let's now try train a Hopfield net on slightly nastier vectors (the previous ones were orthogonal):

(1.0000 -1.0000 -1.0000 0.3000)
(0.1000 1.0000 -1.0000 -0.1000)
(-1.0000 -1.0000 1.0000 -0.5000)
(-1.0000 -1.0000 -1.0000 1.0000)

Example (3)

- **Let's start from some state Y for the neurons, and watch the network evolve.**
- **Y=(1 0 0 0)**

Steps (1 update/neuron):

1:	**(0.9957**	**-0.3693**	**-0.3693**	**-0.0028)**	
5:	**(1.0000**	**-0.5332**	**-0.5332**	**-0.0040)**	
50:	**(1.0000**	**-0.5348**	**-0.5348**	**-0.0040)**	**odd plateau**
170:	**(1.0000**	**-0.5345**	**-0.5350**	**-0.0040)**	
5000:	**(1.0000**	**0.2032**	**-1.0000**	**1.0000)**	**spurious min!**