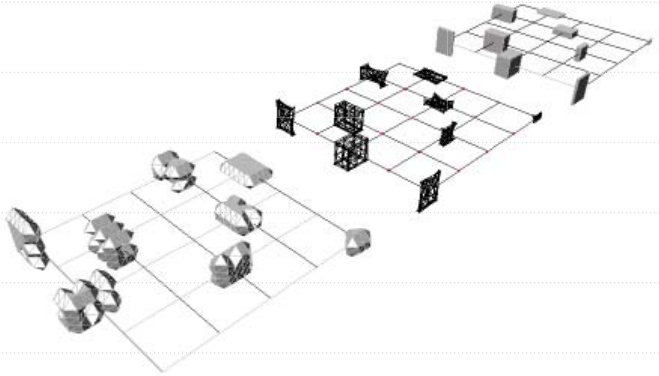
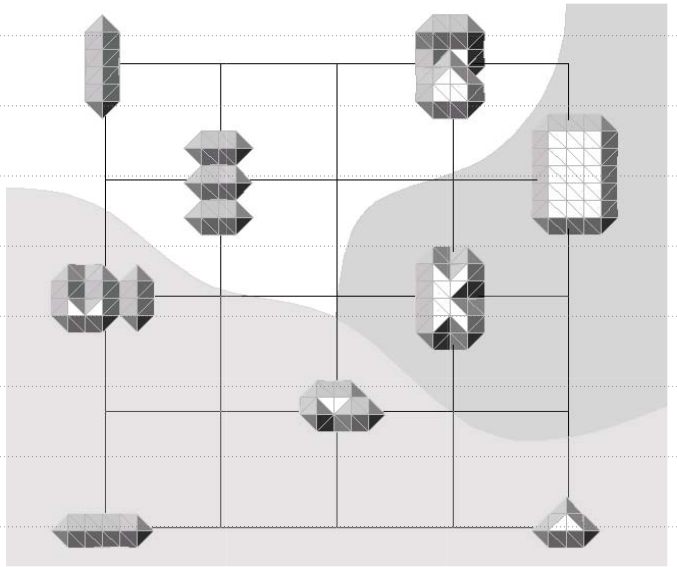
ANN = [artificial/ Architectural Neural Networks]

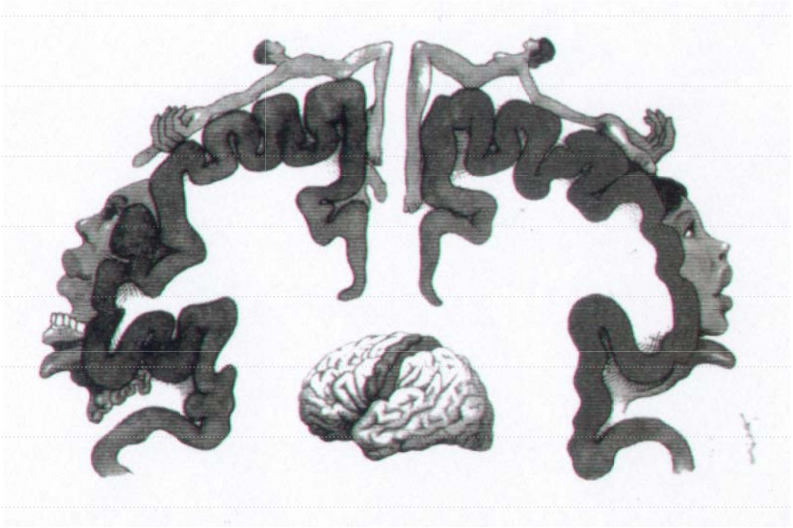
Artificial neural networks emulate the mechanics of the human brain, which with its 10^{11} neurons is believed to be the most complex living organism. In their digital simulation they are rather similar to the principles of a cellular automaton or a swarm, in that the single members (nodes or neurons) of the network are processing simultaneously and therefore possess a 'limbo' world. Additionally, the single member of the network does in isolation not contain any meaning at all, but the system makes only sense when observed as a whole from the outside (therefore called a ***distributed*** or ***connectionist*** model)! The single members communicate with their neighbours or all of the members but don't know what the others are communicating about. Thus, each member has only a local description of what is going on in the system. None of them can see the global picture.

The neural network that we are looking at over the next two weeks is a ***associative*** model. It reads topographic/ geometric/ statistic differences as input into its neurons but organises that input via its topology into categories of associations.
The difference then between an ANN and a CA or swarm is the way the members communicate with neighbours and how they relate to their environment they are embedded in. CAs don't communicate with their context and have a fixed transition function. Swarms don't have a topological order of structure.
ANNs and swarms are adapting to their environment which puts them into the category of learning systems. ANNs are generally used for pattern recognition and statistic analysis.



spatial feature maps &
categories of associations

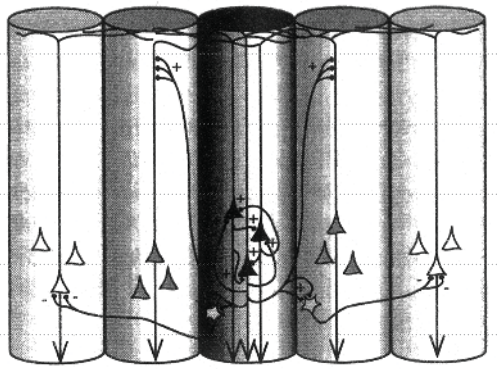




Homunculus

The ANN we are discussing here is based on the 'self-organizing *feature* map' (SOM) developed by the Finnish neuroscientist Teuvo Kohonen. The network consists of a 2 dimensional array of geometric points in space which make up their members or nodes. Each node communicates, thus is connected to every other node in a topological rather than topographic manner. Swarms change their neighbours dynamically in accordance to topographic distances whereas the SOM preserves its topology by respecting the initially laid out structure of the array. CAs are connected topologically as well as topographically (nearest neighbours).
The SOM gets input in the form of system external 3 dimensional points which form a 'vector-space'. This vector space is 'read' by all the nodes. Whichever node is very similar to any of the input points (commonly an *activation function*) organizes his neighbours according to the read point. When all of the nodes are doing that simulaneously, the result of such an input reading is the adaption of the SOM's nodes to the vector space - it has learned the input.

The learning is a mathematical function applied by the nodes that find similar input points to all the nodes in the network. In other words, a feedback between the nodes. The learning algorithm used here is called Hebb's learning. The way that all nodes read all input points but only some of them are 'winners' who get to excert feedback onto the network is called 'competition'.



neural tubes in cortex

the SOM algorithm

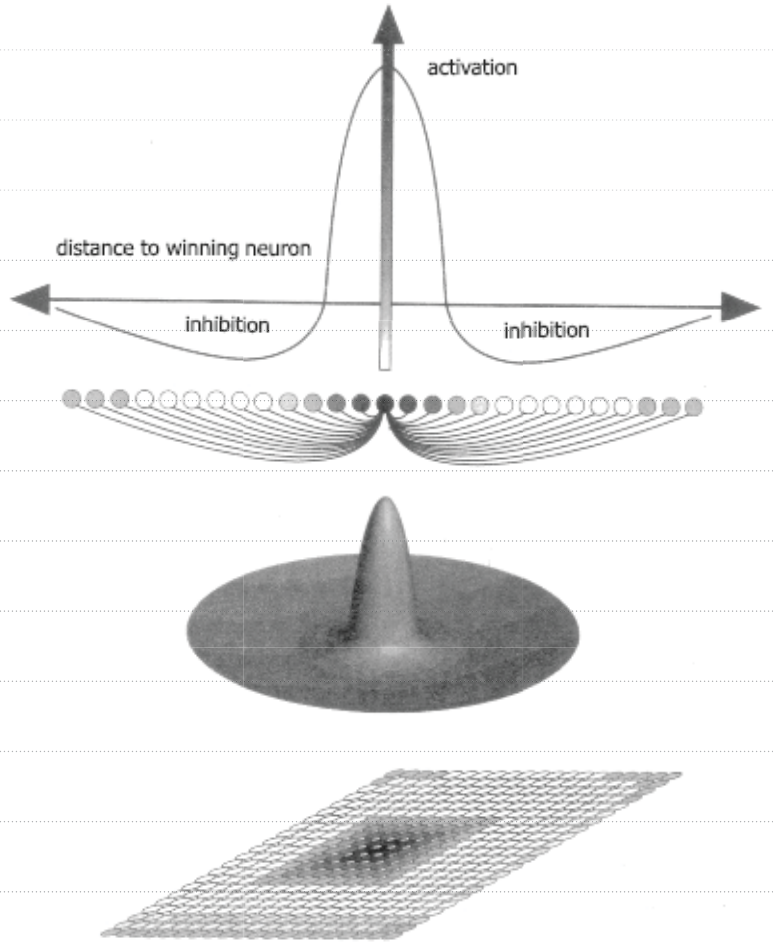
At first, the network itself is set-up through a 2 dimensional surface - an acBezierSurfaceMesh, and the input which constitutes the vector space is defined interactively by the user by clicking on screen.
Then the learning parameters are defined through the learning rate for the winning nodes and a learning rate for all the other nodes. Additionally, the neighbourhood radius needs establishing.

Then the main loop starts that has an stop-condition which is fulfilled when the winning nodes have adapted to the read input points sufficiently as for the observer to recognize the vector space through the network:

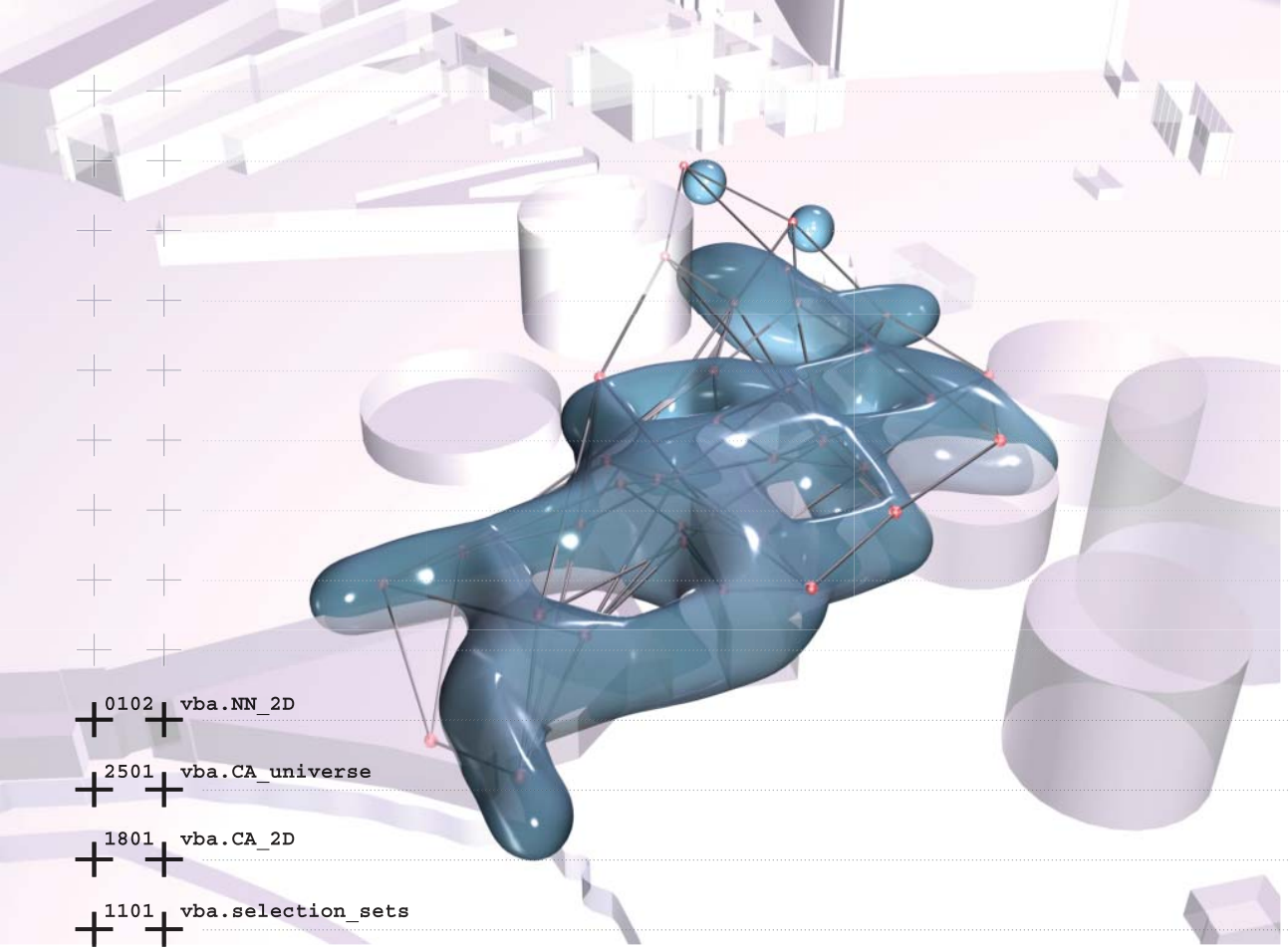
```
Loop While (winlearn > 0.05)
```

At the end of each main loop run, the learning parameters are updated, usually through monotonously decreasing their values.

Within the main loop, each node reads the input points with the call to the subroutine multm() (multiply matrix). This subroutine returns the winners.
Now, the feedback between the neighbours starts with the call to the subroutine neighbours(). Everytime this subroutine is called within the main loop, the learning parameters have diminished. Thus, a slow parallel re-organization of the whole network is garuanteed.
Next, all we do is to change the location in space of the nodes - the re-organization of the network - and place the present network data into a parallel array upon which the next round of feedback calculations is taking place:



Hebb's learning(mexican head function)



learned contextual space

```
net(i, j).node(k) = limbo(i, j).node(k)
```

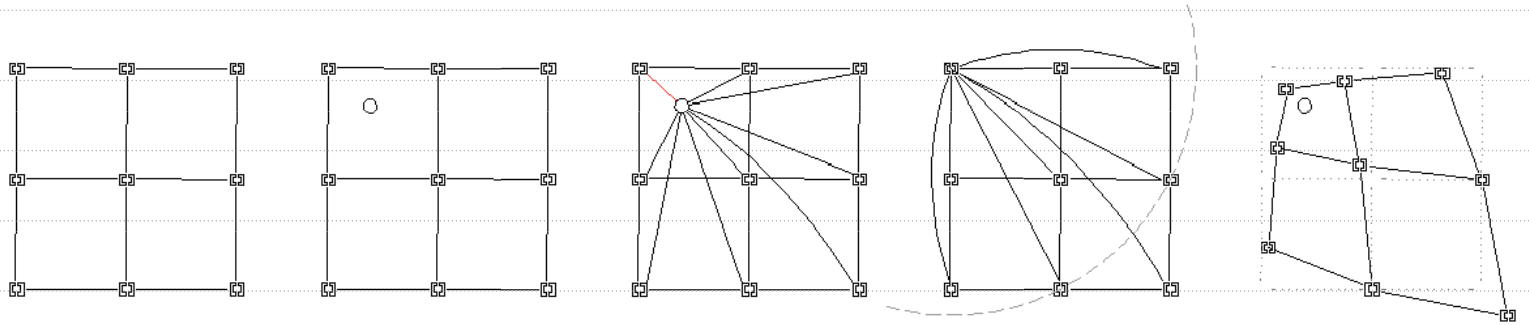
As you might notice, the learning parameters are proportional to time (the variable *zeit* being incremented around every main loop is translated to *time* in English).

New Syntax

The only new syntax, which forms part of the Acad object, is the method to retrieve an 3D point from the user interactively:

```
point = ThisDrawing.Utility.GetPoint_
(, "Enter a point: ")
```

You can see that the method GetPoint() has two arguments. The first - an array of 3 doubles - is left out in our example since we don't AutoCad to translate the coordinate systems from UCS to WCS (don't worry about it). Since it is optional we can leave it out but have to keep the comma to indicate a *null* input. The second is the prompt that appears to the user. Also the prompt is optional.



winner feedback