choose a section

What are neural nets?

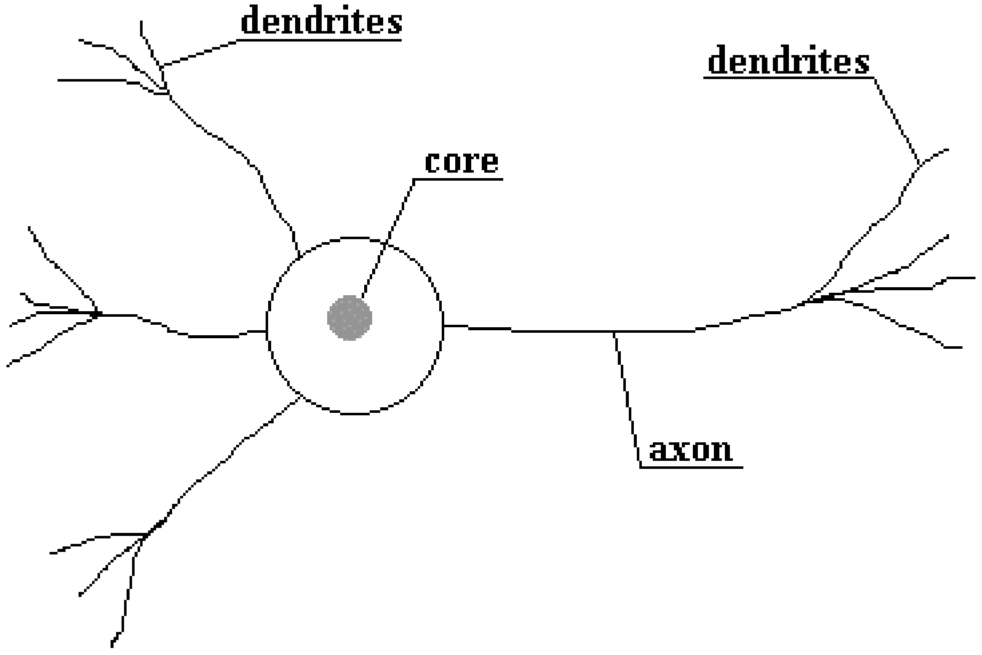
A neural net is an artificial representation of the human brain that tries to simulate its learning process.
The term "artificial" means that neural nets are implemented in computer programs that are able to handle the large number of neccessary calculations during the learning process.

To show where neural nets have their origin, let's have a look at the biological model: the human brain.

The biological model: The human brain

The human brain consists of a large number (more than a billion) of neural cells that process informations. Each cell works like a simple processor and only the massive interaction between all cells and their parallel processing makes the brain's abilities possible.

Below you see a sketch of such a neural cell, called a neuron:



Structure of a neural cell in the human brain

As the figure indicates, a neuron consists of a core, dendrites for incoming information and an axon with dendrites for outgoing information that is passed to connected neurons.
Information is transported between neurons in form of electrical stimulations along the dendrites.
Incoming informations that reach the neuron's dendrites is added up and then delivered along the neuron's axon to the dendrites at its end, where the information is passed to other neurons if the stimulation has exceeded a certain threshold. In this case, the neuron is said to be *activated*.
If the incoming stimulation had been too low, the information will not be transported any further. In this case, the neuron is said to be *inhibited*.

The connections between the neurons are *adaptive*, what means that the connection structure is changing dynamically. It is commonly acknowledged that the learning ability of the human brain is based on this adaptation.

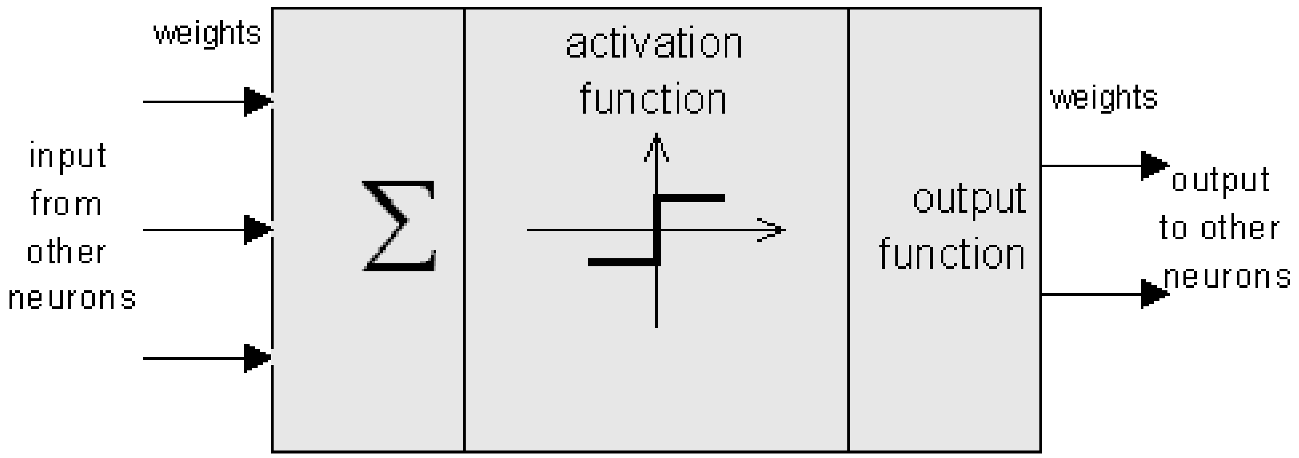
The components of a neural net

Generally spoken, there are many different types of neural nets, but they all have nearly the same components.
If one wants to simulate the human brain using a neural net, it is obviously that some drastic simplifications have to be made:
First of all, it is impossible to "copy" the true parallel processing of all neural cells. Although there are computers that have the ability of parallel processing, the large number of processors that would be necessary to realize it can't be afforded by today's hardware.
Another limitation is that a computer's internal structure can't be changed while performing any tasks.

And how to implement electrical stimulations in a computer program?

These facts lead to an idealized model for simulation purposes.
Like the human brain, a neural net also consists of neurons and connections between them. The neurons are transporting incoming information on their outgoing connections to other neurons. In neural net terms these connections are called weights. The "electrical" information is simulated with specific values stored in those weights.
By simply changing these weight values the changing of the connection structure can also be simulated.

The following figure shows an idealized neuron of a neural net.

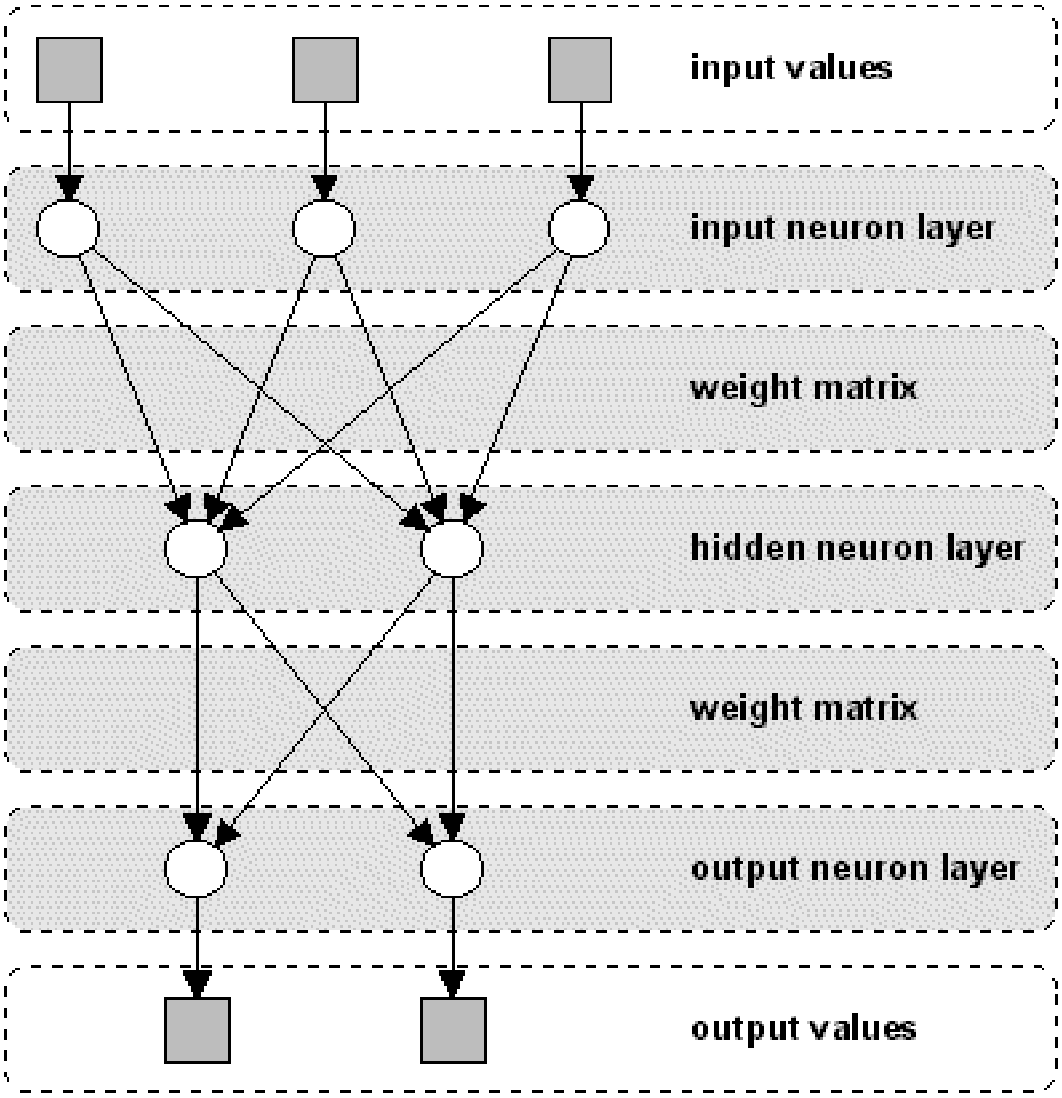


Structure of a neuron in a neural net

As you can see, an artificial neuron looks similar to a biological neural cell. And it works in the same way.
Information (called the input) is sent to the neuron on its incoming weights. This input is processed by a propagation function that adds up the values of all incoming weights.
The resulting value is compared with a certain threshold value by the neuron's activation function.
If the input exceeds the threshold value, the neuron will be activated, otherwise it will be inhibited.
If activated, the neuron sends an output on its outgoing weights to all connected neurons and so on.

In a neural net, the neurons are grouped in layers, called neuron layers. Usually each neuron of one layer is connected to all neurons of the preceding and the following layer (except the input layer and the output layer of the net).
The information given to a neural net is propagated layer-by-layer from input layer to output layer through either none, one or more hidden layers. Depending on the learning algorithm, it is also possible that information is propagated backwards through the net.

The following figure shows a neural net with three neuron layers.



Neural net with three neuron layers

Note that this is not the general structure of a neural net. For example, some neural net types have no hidden layers or the neurons in a layer are arranged as a matrix.
What's common to all neural net types is the presence of at least one weight matrix, the connections between two neuron layers.

Next, let's see what neural nets are useful for.

What they can and where they fail

Neural nets are being constructed to solve problems that can't be solved using conventional algorithms.
Such problems are usually optimization or classification problems.

The different problem domains where neural nets may be used are:

- pattern association
- pattern classification
- regularity detection
- image processing
- speech analysis
- optimization problems
- robot steering
- processing of inaccurate or incomplete inputs
- quality assurance
- stock market forecasting
- simulation
- **...**

There are many different neural net types with each having special properties, so each problem domain has its own net type (see Types of neural nets for a more detailed description).
Generally it can be said that neural nets are very flexible systems for problem solving purposes. One ability should be mentioned explicitely: the *error tolerance* of neural networks. That means, if a neural net had been trained for a specific problem, it will be able to recall correct results, even if the problem to be solved is not exactly the same as the already learned one. For example, suppose a neural net had been trained to recognize human speech. During the learning process, a certain person has to pronounce some words, which are learned by the net. Then, if trained correctly, the neural net should be able to recognize those words spoken by another person, too.

But all that glitters ain't gold. Although neural nets are able to find solutions for difficult problems as listed above, the results can't be guaranteed to be perfect or even correct. They are just approximations of a desired solution and a certain error is always present.
Additionaly, there exist problems that can't be correctly solved by neural nets. An example on pattern recognition should settle this:
If you meet a person you saw earlier in your life, you usually will recognize him/her the second time, even if he/she doesn't look the same as at your first encounter.
Suppose now, you trained a neural net with a photograph of that person, this image will surely be recognized by the net. But if you add heavy noise to the picture or rotate it to some degree, the recognition will probably fail.

Surely, nobody would ever use a neural network in a sorting algorithm, for there exist much better and faster algorithms, but in problem domains, as those mentioned above, neural nets are always a good alternative to existing algorithms and definitely worth a try.

Types of Neural Nets

As mentioned before, several types of neural nets exist.
They can be distinguished by

their type (feedforward or feedback),
their structure
and the learning algorithm they use.

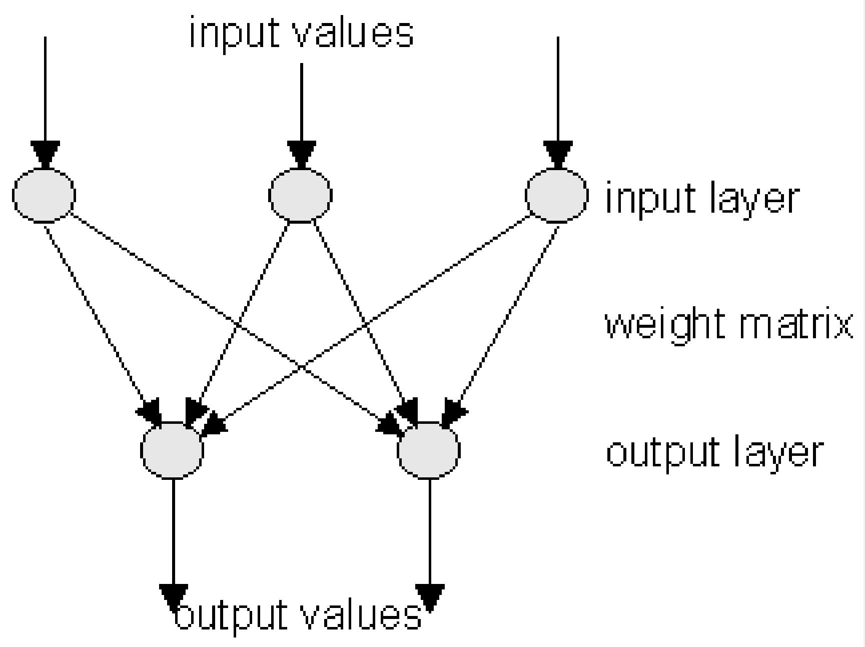
The *type* of a neural net indicates, if the neurons of one of the net's layers may be connected among each other. Feedforward neural nets allow only neuron connections between two different layers, while nets of the feedback type have also connections between neurons of the same layer.

In this section, a selection of neural nets will be described.

Perceptron

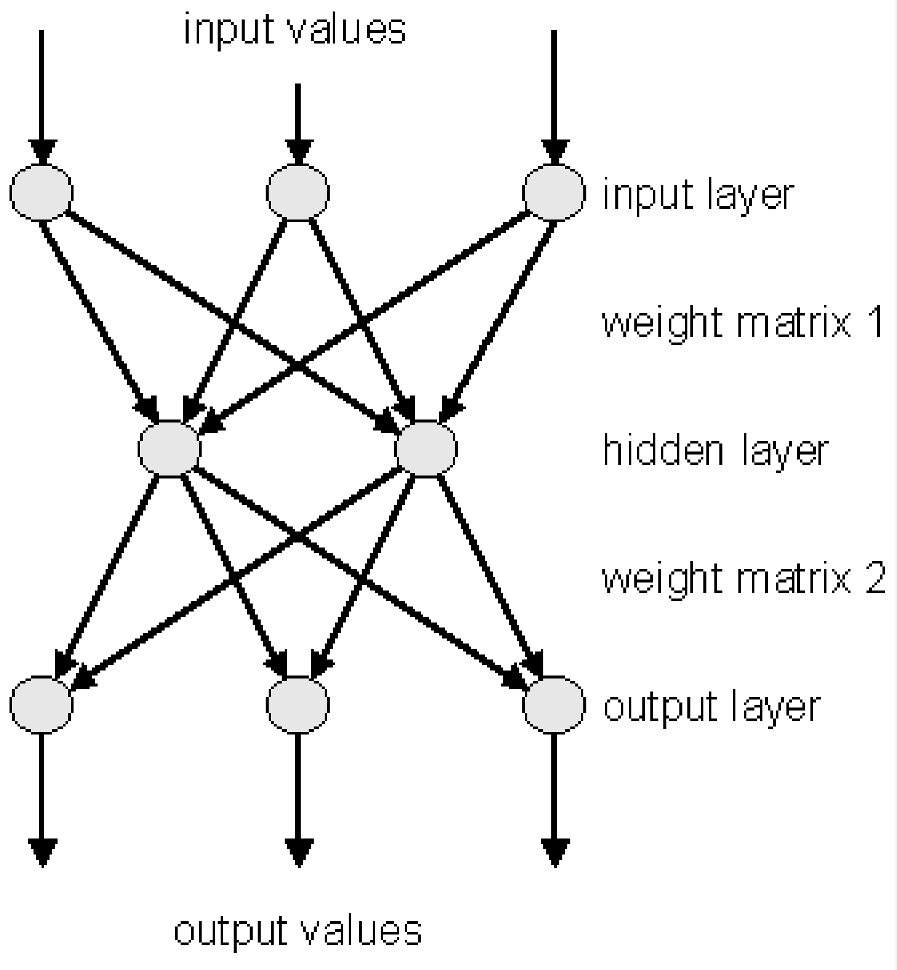
The Perceptron was first introduced by F. Rosenblatt in 1958.
It is a very simple neural net type with two neuron layers that accepts only binary input and output values (0 or 1). The learning process is supervised and the net is able to solve basic logical operations like AND or OR. It is also used for pattern classification purposes.
More complicated logical operations (like the XOR problem) cannot be solved by a Perceptron.

Perceptron characteristics

sample structure	 input values input layer weight matrix output layer output values
type	feedforward
neuron layers	1 input layer 1 output layer
input value types	binary
activation function	hard limiter
learning method	supervised
learning algorithm	Hebb learning rule
mainly used in	simple logical operations pattern classification

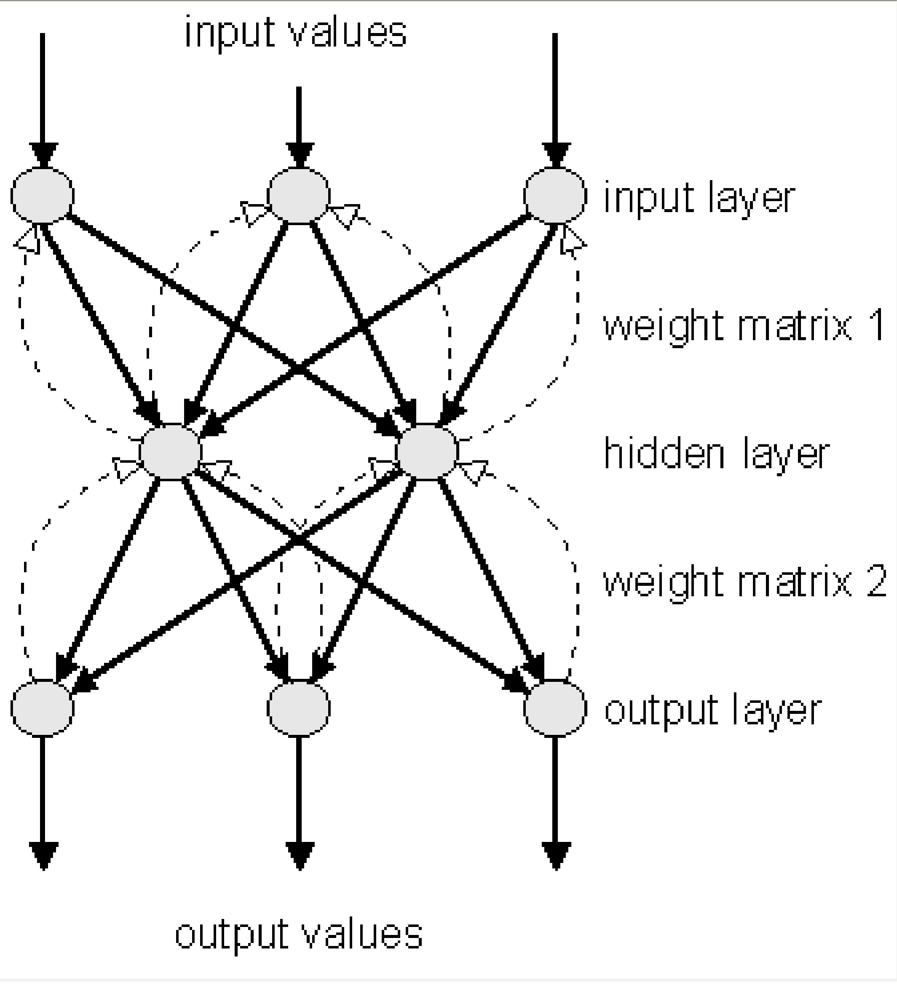
Multi-Layer-Perceptron

The Multi-Layer-Perceptron was first introduced by M. Minsky and S. Papert in 1969.
It is an extended Perceptron and has one ore more hidden neuron layers between its input and output layers.
Due to its extended structure, a Multi-Layer-Perceptron is able to solve every logical operation, including the XOR problem.

Multi-Layer-Perceptron characteristics	
sample structure	 input values input layer weight matrix 1 hidden layer weight matrix 2 output layer output values
type	feedforward
neuron layers	1 input layer 1 or more hidden layers 1 output layer
input value types	binary
activation function	hard limiter / sigmoid
learning method	supervised
learning algorithm	delta learning rule backpropagation (mostly used)
mainly used in	complex logical operations pattern classification

Backpropagation Net

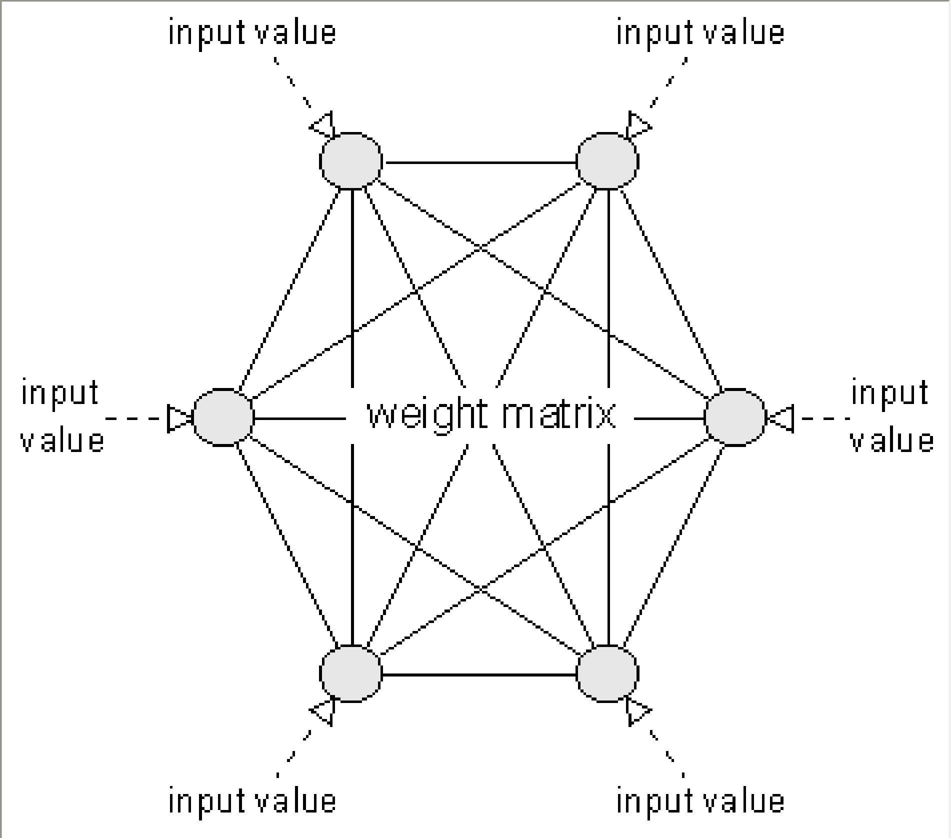
The Backpropagation Net was first introduced by G.E. Hinton, E. Rumelhart and R.J. Williams in 1986
and is one of the most powerful neural net types.
It has the same structure as the Multi-Layer-Perceptron and uses the backpropagation learning algorithm.

Backpropagation Net characteristics	
sample structure	 input values input layer weight matrix 1 hidden layer weight matrix 2 output layer output values
type	feedforward
neuron layers	1 input layer 1 or more hidden layers 1 output layer
input value types	binary

activation function	sigmoid
learning method	supervised
learning algorithm	backpropagation
mainly used in	complex logical operations pattern classification speech analysis

Hopfield Net

The Hopfield Net was first introduced by physicist J.J. Hopfield in 1982 and belongs to neural net types which are called "thermodynamical models".
It consists of a set of neurons, where each neuron is connected to each other neuron. There is no differentiation between input and output neurons.
The main application of a Hopfield Net is the storage and recognition of patterns, e.g. image files.

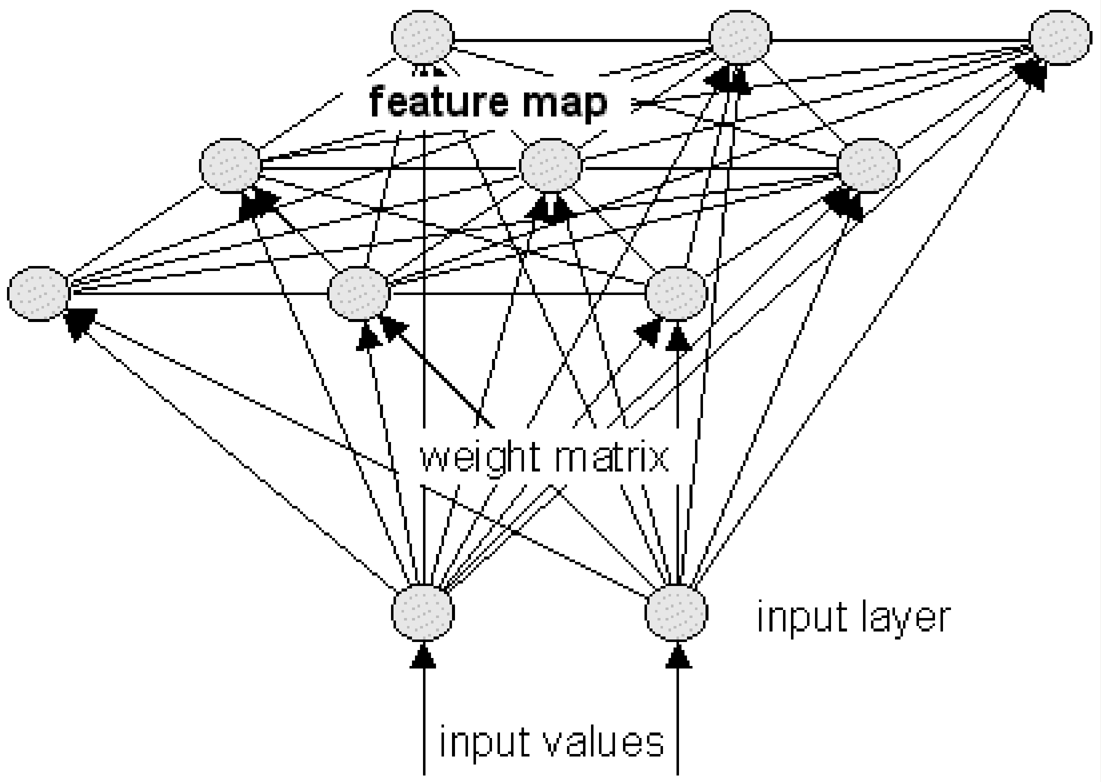
Hopfield Net characteristics	
sample structure	 input value input value input value weight matrix input value input value input value

type	feedback
neuron layers	1 matrix
input value types	binary
activation function	signum / hard limiter
learning method	unsupervised
learning algorithm	delta learning rule simulated annealing (mostly used)
mainly used in	pattern association optimization problems

Kohonen Feature Map

The Kohonen Feature Map was first introduced by finnish professor Teuvo Kohonen (University of Helsinki) in 1982.
It is probably the most useful neural net type, if the learning process of the human brain shall be simulated. The "heart" of this type is the *feature map*, a neuron layer where neurons are organizing themselves according to certain input values.
The type of this neural net is both feedforward (input layer to feature map) and feedback (feature map).
(A Kohonen Feature Map is used in the sample applet)

Kohonen Feature Map characteristics

sample structure	
type	feedforward / feedback
neuron layers	1 input layer 1 map layer
input value types	binary, real
activation function	sigmoid
learning method	unsupervised
learning algorithm	selforganization
mainly used in	pattern classification optimization problems simulation

Types of neural nets bottom of page

The learning process

This section describes the learning ability of neural networks.
First, the term *learning* is explained, followed by an overview of specific learning algorithms for neural nets.

What does "learning" mean refering to neural nets?

In the human brain, information is passed between the neurons in form of electrical stimulation along the dendrites. If a certain amount of stimulation is received by a neuron, it generates an output to all other connected neurons and so information takes its way to its destination where some reaction will occur. If the incoming stimulation is too low, no output is generated by the neuron and the information's further transport will be blocked.

Explaining how the human brain learns certain things is quite difficult and nobody knows it exactly.
It is supposed that during the learning process the connection structure among the neurons is changed, so that certain stimulations are only accepted by certain neurons. This means, there exist firm connections between the neural cells that once have learned a specific fact, enabling the fast recall of this information.
If some related information is acquired later, the same neural cells are stimulated and will adapt their connection structure according to this new information.
On the other hand, if a specific information isn't recalled for a long time, the established connection structure between the responsible neural cells will get more "weak". This had happened if someone "forgot" a once learned fact or can only remember it vaguely.

As mentioned before, neural nets *try to simulate* the human brain's ability to learn. That is, the artificial neural net is also made of neurons and dendrites. Unlike the biological model, a neural net has an unchangeable structure, built of a specified number of neurons and a specified number of connections between them (called *"weights"*), which have certain values.
What changes during the learning process are the values of those weights. Compared to the original this means:
Incoming information "stimulates" (exceeds a specified threshold value of) certain neurons that pass the information to connected neurons or prevent further transportation along the weighted connections. The value of a weight will be increased if information should be transported and decreased if not.

While learning different inputs, the weight values are changed dynamically until their values are *balanced*, so each input will lead to the desired output.
The training of a neural net results in a matrix that holds the weight values between the neurons. Once a neural net had been trained correctly, it will probably be able to find the desired output to a given input that had been learned, by using these matrix values.
I said "probably". That is sad but true, for it can't be guaranteed that a neural net will recall the correct results in any case.
Very often there is a certain error left after the learning process, so the generated output is only a good approximation to the perfect output in most cases.

The following sections introduce several learning algorithms for neural networks.

Supervised and unsupervised learning

The learning algorithm of a neural network can either be supervised or unsupervised.

A neural net is said to learn *supervised*, if the desired output is already known.
Example: pattern association
Suppose, a neural net shall learn to associate the following pairs of patterns. The input patterns are decimal numbers, each represented in a sequence of bits. The target patterns are given in form of binary values of the decimal numbers:

input pattern	**target pattern**
0001	001
0010	010
0100	011
1000	100

While learning, one of the input patterns is given to the net's input layer. This pattern is propagated through the net (independent of its structure) to the net's output layer. The output layer generates an output pattern which is then compared to the target pattern. Depending on the difference between output and target, an error value is computed.
This output error indicates the net's learning effort, which can be controlled by the "imaginary supervisor". The greater the computed error value is, the more the weight values will be changed.

Neural nets that learn *unsupervised* have no such target outputs.
It can't be determined what the result of the learning process will look like.
During the learning process, the units (weight values) of such a neural net are "arranged" inside a certain range, depending on given input values. The goal is to group similar units close together in certain areas of the value range.
This effect can be used efficiently for pattern classification purposes.
See Selforganization for details.

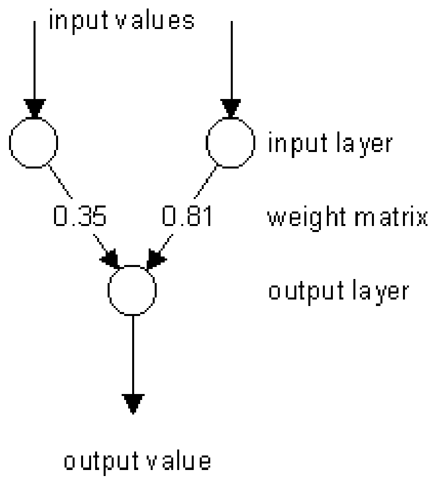
Forwardpropagation

Forwardpropagation is a supervised learning algorithm and describes the "flow of information" through a neural net from its input layer to its output layer.

The algorithm works as follows:

1. Set all weights to random values ranging from -1.0 to +1.0
2. Set an input pattern (binary values) to the neurons of the net's input layer
3. Activate each neuron of the following layer:
   Multiply the weight values of the connections leading to this neuron with the output values of the preceding neurons
   Add up these values
   Pass the result to an activation function, which computes the output value of this neuron
7. Repeat this until the output layer is reached
8. Compare the calculated output pattern to the desired target pattern and compute an error value
9. Change all weights by adding the error value to the (old) weight values
10. Go to step 2
11. The algorithm ends, if all output patterns match their target patterns

Example:
Suppose you have the following 2-layered Perceptron:



Patterns to be learned:

input	target
0 1	0
1 1	1

First, the weight values are set to random values (0.35 and 0.81).
The learning rate of the net is set to 0.25.
Next, the values of the first input pattern (0 1) are set to the neurons of the input layer (the output of the input layer is the same as its input).
The neurons in the following layer (only one neuron in the output layer) are activated:

```
        Input 1 of output neuron:       0 * 0.35 = 0
        Input 2 of output neuron:       1 * 0.81 = 0.81
        Add the inputs:                 0 + 0.81 = 0.81            (= output)
        Compute an error value by
        subtracting output from target: 0 - 0.81 = -0.81
        Value for changing weight 1:    0.25 * 0 * (-0.81) = 0     (0.25 = learning
rate)
        Value for changing weight 2:    0.25 * 1 * (-0.81) = -0.2025
        Change weight 1:                0.35 + 0         = 0.35    (not changed)
        Change weight 2:                0.81 + (-0.2025) = 0.6075
```

Now that the weights are changed, the second input pattern (1 1) is set to the input layer's neurons and the activation of the output neuron is performed again, now with the new weight values:

```
        Input 1 of output neuron:       1 * 0.35   = 0.35
        Input 2 of output neuron:       1 * 0.6075 = 0.6075
        Add the inputs:                 0.35 + 0.6075 = 0.9575     (= output)
        Compute an error value by
        subtracting output from target: 1 - 0.9575 = 0.0425
        Value for changing weight 1:    0.25 * 1 * 0.0425 = 0.010625
        Value for changing weight 2:    0.25 * 1 * 0.0425 = 0.010625
        Change weight 1:                0.35   + 0.010625 = 0.360625
        Change weight 2:                0.6075 + 0.010625 = 0.618125
```

That was one learning step. Each input pattern had been propagated through the net and the weight values were changed.
The error of the net can now be calculated by adding up the squared values of the output errors of each pattern:

Compute the net error: $(-0.81)^2 + (0.0425)^2 = \mathbf{0.65790625}$

By performing this procedure repeatedly, this error value gets smaller and smaller.

The algorithm is successfully finished, if the net error is zero (perfect) or approximately zero.

Backpropagation

Backpropagation is a supervised learning algorithm and is mainly used by Multi-Layer-Perceptrons to change the weights connected to the net's hidden neuron layer(s).

The backpropagation algorithm uses a computed output error to change the weight values in backward direction.
To get this net error, a forwardpropagation phase must have been done before. While propagating in forward direction, the neurons are being activated using the *sigmoid activation function*.

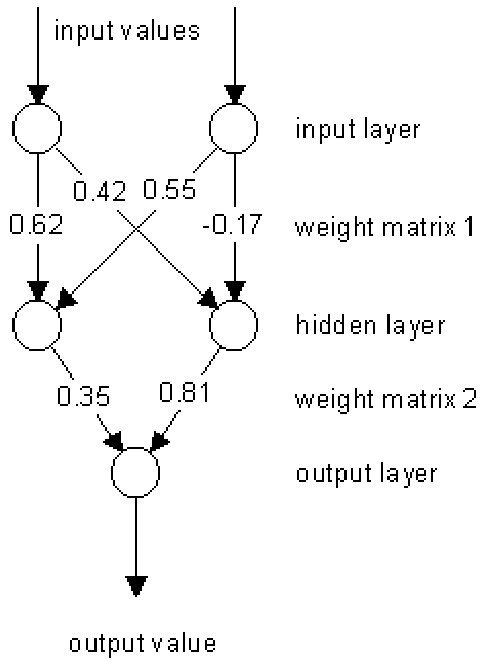
The formula of **sigmoid activation** is:

```
              1
f(x) = ---------
         1 + e-input
```

The algorithm works as follows:

1. Perform the forwardpropagation phase for an input pattern and calculate the output error
2. Change all weight values of each weight matrix using the formula
   weight(old) + learning rate * output error * output(neurons i) * output(neurons i+1) * (1 - output(neurons i+1))
4. Go to step 1
5. The algorithm ends, if all output patterns match their target patterns

Example:
Suppose you have the following 3-layered Multi-Layer-Perceptron:



Patterns to be learned:

input	target
0 1	0
1 1	1

First, the weight values are set to random values: 0.62, 0.42, 0.55, -0.17 for weight matrix 1 and 0.35, 0.81 for weight matrix 2.
The learning rate of the net is set to 0.25.
Next, the values of the first input pattern (0 1) are set to the neurons of the input layer (the output of the input layer is the same as its input).

The neurons in the hidden layer are activated:

```
        Input of hidden neuron 1:       0 * 0.62 + 1 * 0.55    = 0.55
        Input of hidden neuron 2:       0 * 0.42 + 1 * (-0.17) = -0.17
        Output of hidden neuron 1:      1 / ( 1 + exp(-0.55) ) = 0.634135591
        Output of hidden neuron 2:      1 / ( 1 + exp(+0.17) ) = 0.457602059
```

The neurons in the output layer are activated:

```
        Input of output neuron:         0.634135591 * 0.35 + 0.457602059 * 0.81 =
0.592605124
        Output of output neuron:        1 / ( 1 + exp(-0.592605124) ) = 0.643962658
        Compute an error value by
        subtracting output from target: 0 - 0.643962658 = -0.643962658
```

Now that we got the output error, let's do the backpropagation.
We start with changing the weights in weight matrix 2:

```
        Value for changing weight 1:    0.25 * (-0.643962658) * 0.634135591
                                        * 0.643962658 * (1-0.643962658) = -
0.023406638
        Value for changing weight 2:    0.25 * (-0.643962658) * 0.457602059
                                        * 0.643962658 * (1-0.643962658) = -
0.016890593
        Change weight 1:                0.35 + (-0.023406638) = 0.326593362
        Change weight 2:                0.81 + (-0.016890593) = 0.793109407
```

Now we will change the weights in weight matrix 1:

```
        Value for changing weight 1:    0.25 * (-0.643962658) * 0
                                        * 0.634135591 * (1-0.634135591) = 0
        Value for changing weight 2:    0.25 * (-0.643962658) * 0
                                        * 0.457602059 * (1-0.457602059) = 0
        Value for changing weight 3:    0.25 * (-0.643962658) * 1
                                        * 0.634135591 * (1-0.634135591) = -
0.037351064
        Value for changing weight 4:    0.25 * (-0.643962658) * 1
                                        * 0.457602059 * (1-0.457602059) = -
0.039958271
        Change weight 1:                0.62 + 0 = 0.62         (not changed)
        Change weight 2:                0.42 + 0 = 0.42         (not changed)
        Change weight 3:                0.55 + (-0.037351064) = 0.512648936
        Change weight 4:                -0.17+ (-0.039958271) = -0.209958271
```

The first input pattern had been propagated through the net.
The same procedure is used for the next input pattern, but then with the changed weight values.
After the forward and backward propagation of the second pattern, one learning step is complete and the net error can be calculated by adding up the squared output errors of each pattern.
By performing this procedure repeatedly, this error value gets smaller and smaller.

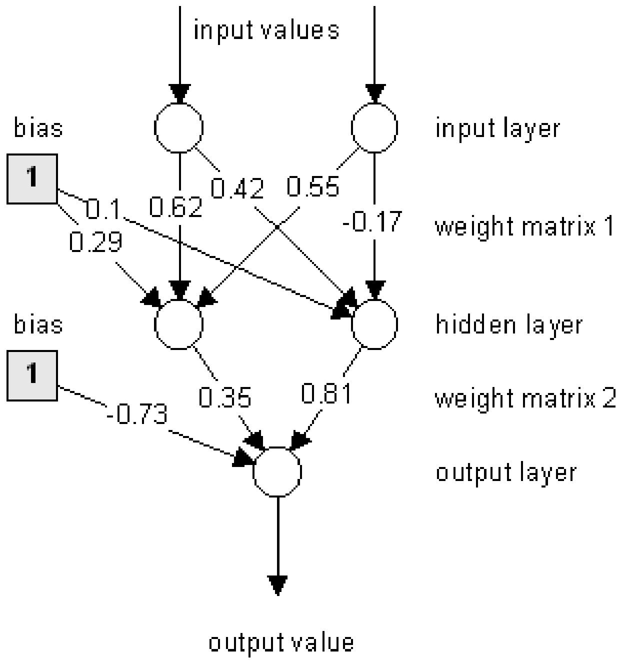
The algorithm is successfully finished, if the net error is zero (perfect) or approximately zero.

Note that this algorithm is also applicable for Multi-Layer-Perceptrons with more than one hidden layer.

"What happens, if all values of an input pattern are zero?"

If all values of an input pattern are zero, the weights in weight matrix 1 would never be changed for this pattern and the net could not learn it. Due to that fact, a "pseudo input" is created, called *Bias* that has a constant output value of 1.

This changes the structure of the net in the following way:



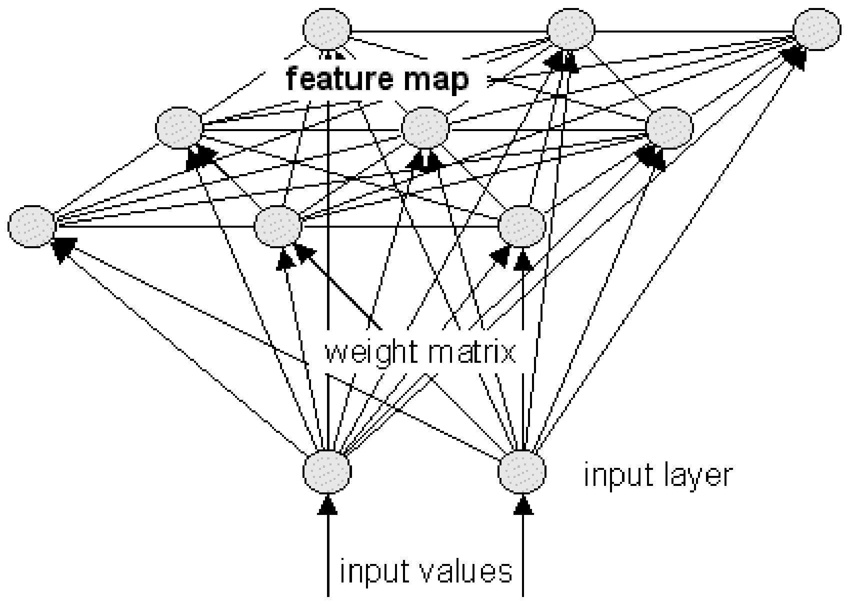
These additional weights, leading to the neurons of the hidden layer and the output layer, have initial random values and are changed in the same way as the other weights. By sending a constant output of 1 to following neurons, it is guaranteed that the input values of those neurons are always differing from zero.

Selforganization

Selforganization is an unsupervised learning algorithm used by the Kohonen Feature Map neural net.

As mentioned in previous sections, a neural net tries to simulate the biological human brain, and selforganization is probably the best way to realize this.
It is commonly known that the *cortex* of the human brain is subdivided in different regions, each responsible for certain functions. The neural cells are organizing themselves in groups, according to incoming informations.
Those incoming informations are not only received by a single neural cell, but also influences other cells in its neighbourhood.
This organization results in some kind of a map, where neural cells with similar functions are arranged close together.

This selforganization process can also be performed by a neural network. Those neural nets are mostly used for classification purposes, because similar input values are represented in certain areas of the net's map.
A sample structure of a Kohonen Feature Map that uses the selforganization algorithm is shown below:



Kohonen Feature Map with 2-dimensional input and 2-dimensional map (3x3 neurons)

As you can see, each neuron of the input layer is connected to each neuron on the map. The resulting weight matrix is used to propagate the net's input values to the map neurons.
Additionally, all neurons on the map are connected among themselves. These connections are used to influence neurons in a certain area of activation around the neuron with the greatest activation, received from the input layer's output.

The amount of feedback between the map neurons is usually calculated using the Gauss function:

$$\text{feedback}_{ci} = e^{\frac{-|x_c - x_i|^2}{2 * sig^2}}$$

where x_c is the position of the most activated neuron
x_i are the positions of the other map neurons
sig is the activation area (radius)

In the beginning, the activation area is large and so is the feedback between the map neurons. This results in an activation of neurons in a wide area around the most activated neuron.
As the learning progresses, the activation area is constantly decreased and only neurons closer to the activation center are influenced by the most activated neuron.

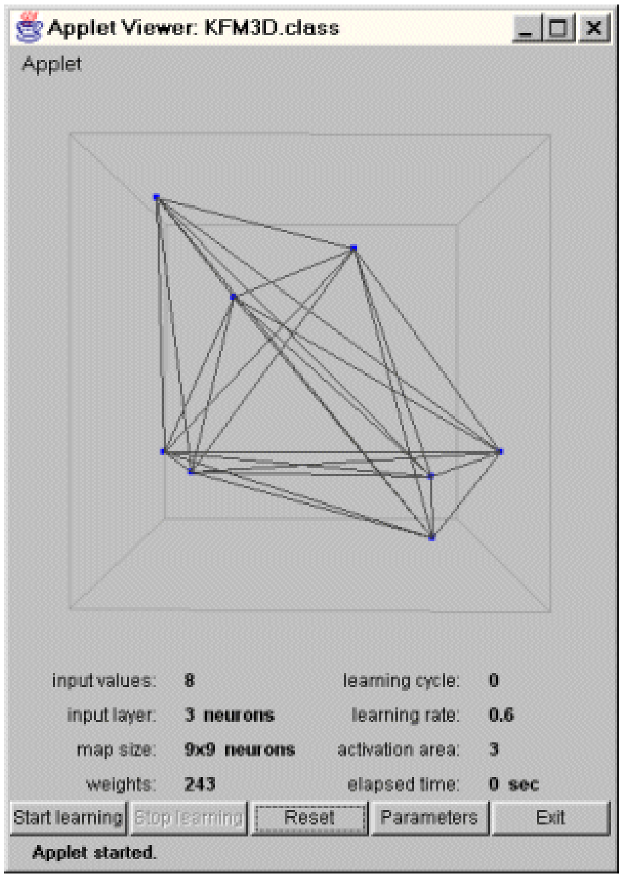
Unlike the biological model, the map neurons don't change their positions on the map. The "arranging" is simulated by changing the values in the weight matrix (the same way as other neural nets do).
Because selforganization is an unsupervised learning algorithm, no input/target patterns exist. The input values passed to the net's input layer are taken out of a specified value range and represent the "data" that should be organized.

The algorithm works as follows:

1. Define the range of the input values
2. Set all weights to random values taken out of the input value range
3. Define the initial activation area
4. Take a random input value and pass it to the input layer neuron(s)
5. Determine the most activated neuron on the map:
   Multiply the input layer's output with the weight values
   The map neuron with the greatest resulting value is said to be "most activated"
   Compute the feedback value of each other map neuron using the Gauss function
9. Change the weight values using the formula:
   weight(old) + feedback value * (input value - weight(old)) * learning rate

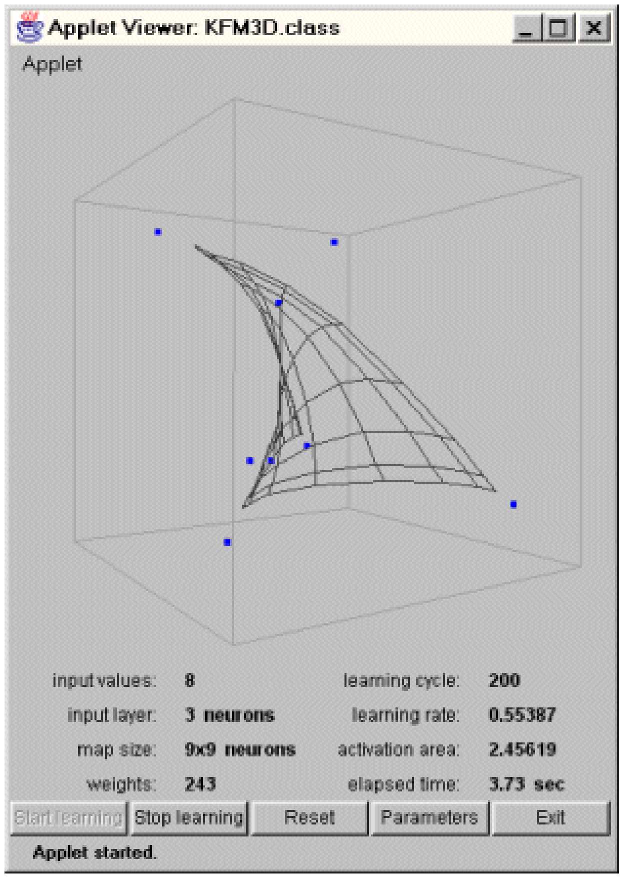
11. Decrease the activation area
12. Go to step 4
13. The algorithm ends, if the activation area is smaller than a specified value

Example: see sample applet
The shown Kohonen Feature Map has three neurons in its input layer that represent the values of the x-, y- and z-dimension. The feature map is initially 2-dimensional and has 9x9 neurons. The resulting weight matrix has 3 * 9 * 9 = 243 weights, because each input neuron is connected to each map neuron.



input values: 8
input layer: 3 neurons
map size: 9x9 neurons
weights: 243
learning cycle: 0
learning rate: 0.6
activation area: 3
elapsed time: 0 sec

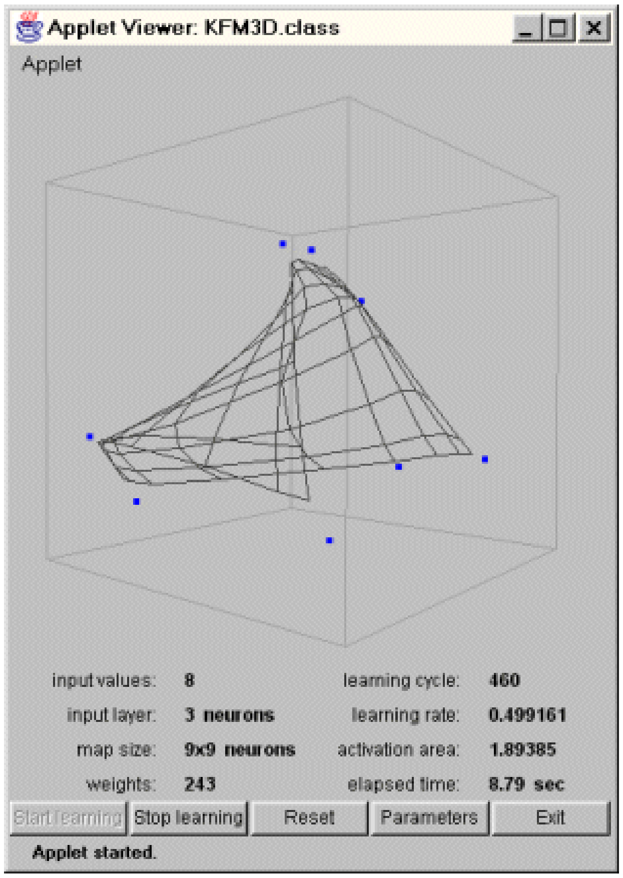
Applet started.



input values: 8
input layer: 3 neurons
map size: 9x9 neurons
weights: 243
learning cycle: 200
learning rate: 0.55387
activation area: 2.45619
elapsed time: 3.73 sec

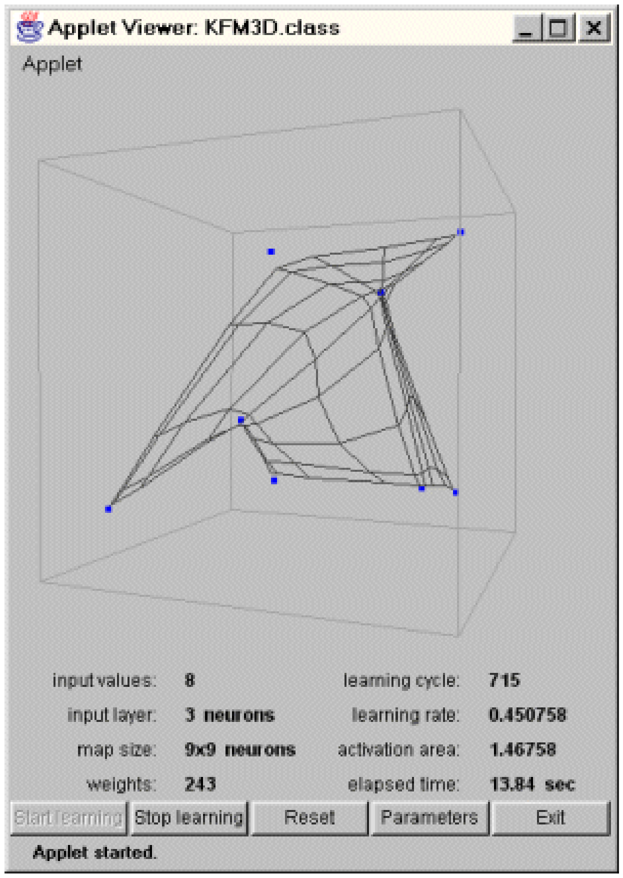
Applet started.

In the beginning, when the weights have random values, the feature map is just an unordered mess.
After 200 learning cycles, the map has "unfolded" and a grid can be seen.



```
input values:  8                learning cycle:   460
input layer:   3 neurons        learning rate:    0.499161
map size:      9x9 neurons      activation area:  1.89385
weights:       243              elapsed time:     8.79 sec
```

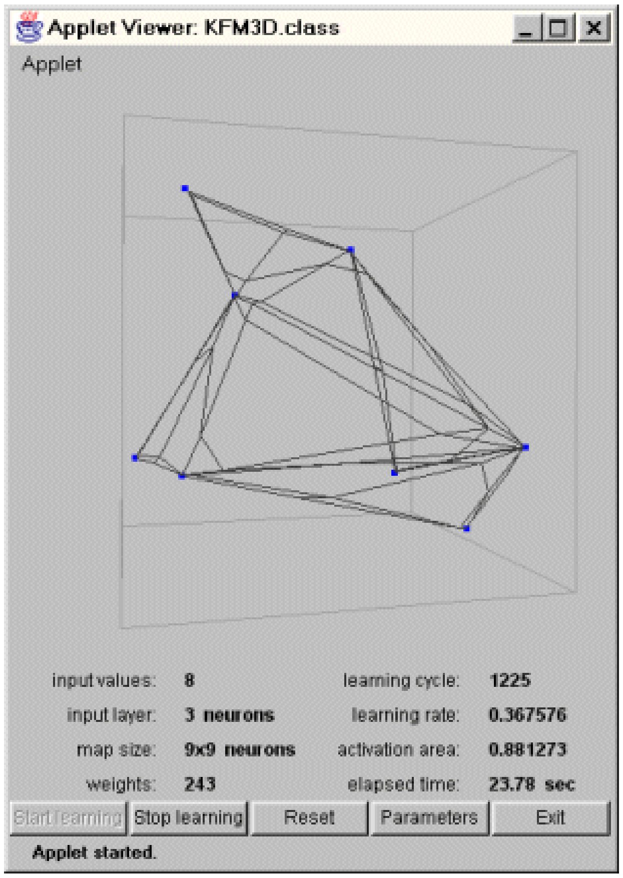
Applet started.



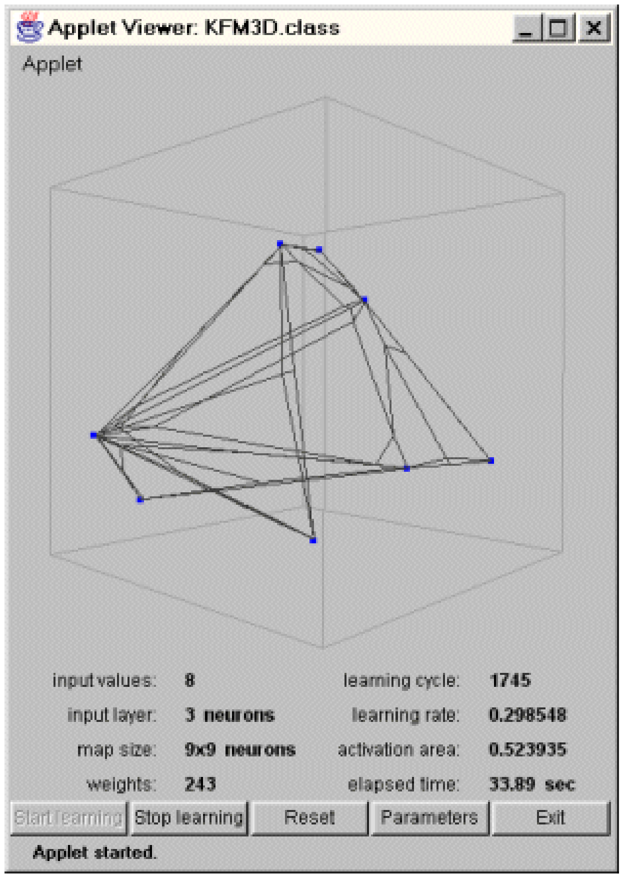
```
input values:  8                learning cycle:   715
input layer:   3 neurons        learning rate:    0.450758
map size:      9x9 neurons      activation area:  1.46758
weights:       243              elapsed time:     13.84 sec
```

Applet started.

As the learning progresses, the map becomes more and more structured.
It can be seen that the map neurons are trying to get closer to their nearest blue input value.



input values: 8
input layer: 3 neurons
map size: 9x9 neurons
weights: 243
learning cycle: 1225
learning rate: 0.367576
activation area: 0.881273
elapsed time: 23.78 sec
Applet started.



input values: 8
input layer: 3 neurons
map size: 9x9 neurons
weights: 243
learning cycle: 1745
learning rate: 0.298548
activation area: 0.523935
elapsed time: 33.89 sec
Applet started.

At the end of the learning process, the feature map is spanned over all input values.
The reason why the grid is not very beautiful is that the neurons in the middle of the feature map are also trying to get closer to the input values. This leads to a distorted look of the grid.
The selforganization is finished at this point.

I recommend you, to do your own experiments with the sample applet, in order to understand its behaviour. (A description of the applet's controls is given on the belonging page)
By changing the net's parameters, it is possible to produce situations, where the feature map is unable to organize itself correctly. Try, for example, to give the initial activation area a very small value or enter too many input values.

Types of neural nets   top of page

navigation
[main page] [content] [neural net overview]
[class structure] [using the classes] [sample applet]
[glossary] [literature] [about the author] [what do you think ?]