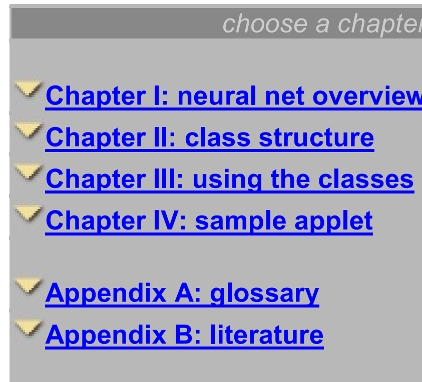
NEURAL NET COMPONENTS IN AN OBJECT ORIENTED CLASS STRUCTURE

By **Jochen Fröhlich**

Preface 5
Contents 9
Chapter I: Neural Net Overview 9
Chapter II: Class Structure 9
Chapter III: Using the Classes 10
Chapter IV: Sample Applet 10
Appendix A: Glossary 11
Appendix B: Literature 11
What Are Neural Nets? 11
The biological model: The human brain 11
The components of a neural net 12
What they can and where they fail 14
Types of Neural Nets 15
Perceptron 16
Sample Structure 16
Multi-Layer-Perceptron 17
Sample Structure 17
Backpropagation Net 18
Sample Structure 18
Hopfield Net 19
Sample Structure 19
Kohonen Feature Map 20
Sample Structure 20
The Learning Process 21
What does "learning" mean refering to neural nets? 21
Supervised and unsupervised learning 22
Forwardpropagation 22
Backpropagation 25
"What happens, if all values of an input pattern are zero?" 27
Selforganization 28
Class Structure 33
What is Object Orientation? 34
Classes 35
Objects 36
Methods 37
Encapsulation 38
Inheritance 39
Abstraction 40
The classes and their relationships 44
Neural Net classes 45
Class NeuralNet 45
Extends 45
Instantiated by 45
Constructors 45
Methods 45
Class BackpropagationNet 46
Extends 46
Instantiated by 46
Constructors 46

Methods 46
Class KohonenFeatureMap 48
Extends 48
Instantiated by 48
Constructors 48
Methods 48
Common components 49
Class Neuron 49
Extends 49
Instantiated by 50
Constructors 50
Methods 50
Class NeuronLayer 50
Extends 50
Instantiated by 50
Constructors 50
Methods 51
Class WeightMatrix 51
Extends 51
Instantiated by 51
Constructors 51
Methods 52
Net type specific components 52
Class Pattern 53
Extends 53
Instantiated by 53
Constructors 53
Methods 53
Class MapNeuron 53
Extends 53
Instantiated by 53
Constructors 53
Methods 54
Class NeuronMatrix 54
Extends 54
Instantiated by 54
Constructors 54
Methods 54
Class InputValue 55
Extends 55
Instantiated by 55
Constructors 55
Methods 55
Class InputMatrix 55
Extends 56
Instantiated by 56
Constructors 56
Methods 56
Using the Classes 57
Java = new ProgrammingLanguage() 57

Simple 57
Object-oriented 58
Network-savvy 59
Architecture neutral 59
Working with Java 60
The look of a class 61
Creating an object 63
Declaration 63
Instantiation 63
Initialization 64
Invoking a method 64
Using the classes in your own programs 66
Using the BackpropagationNet class 66
The structure of a conversion file 70
The structure of a pattern file 71
Using the KohonenFeatureMap class 72
Using the InputMatrix class 75
Sample Applet 76
A 3D Kohonen Feature Map 76
Description 76



Preface

In the last semester of my studies of Computer Science at the Fachhochschule Regensburg (March-July '96) there were two lectures I visited, because their contents seemed to be very interesting.

One of it was titled **Neural Networks**. For I always had been fascinated by artificial intelligence, I thought it could be a way to learn more about it, especially about the neural net theory.
The other one introduced the new programming language **Java**. That was the opportunity to get into the "mystic" world of object orientation I've heard of many times before, but didn't realize its advantages compared to traditional programming style. I should not regret my choice.
Besides, I was looking for a subject for my diploma that would finish my studies appropriately. (Of course, another management system for anything could have been written and probably never been used again, but that wouldn't be very motivating...)

The idea of "Neural Net Components in an Object Oriented Class Structure" appeared, when I had to do a project for the neural network lecture. I implemented the *Travelling Salesman Problem* using a Kohonen Feature Map neural net. Although it was written in Java, object orientation was no problem for me - because I didn't make use of it! So all ended up in one huge class with many unflexible methods.
To get some help on programming a neural net, I read several books which contained many examples, each suitable for just one special problem. However, some parts of the given source codes reappeared in other examples too.
In the neural networks lecture it was spoken of *neurons*, *weights* and *activation*, while the Java lessons introduced *classes*, *objects* and *methods* and how to write reusable software components.
And one thing lead to the other: Assuming that a neural net could come in handy in programs I would write later, it would be a pretty nice thing, if its components could be used like a construction kit instead of writing the same procedures again and again.
So I tried to find commonalities of different neural networks, especially the Backpropagation Net and the Kohonen Feature Map types, because they seem to be most useful for practical purposes.

The result was the Java class structure for neural net components which can be found on the following pages.

Chapter I neural net overview

This chapter gives general information on neural networks.
Besides the introduction of several neural net types, the different learning algorithms, suitable for the different net types, are also explained.

The chapter is divided into the following sections:

What are neural nets?
The biological model: The human brain
The components of a neural net
What they can and where they fail
Types of neural nets
Perceptron
Multi-Layer-Perceptron
Backpropagation Net
Hopfield Net
Kohonen Feature Map
The learning process
What does "learning" mean refering to neural nets?
Supervised and unsupervised learning
Forwardpropagation
Backpropagation
Selforganization

Chapter II class structure

This chapter shows the structure of the implemented classes.
At the beginning you will find a description of object orientation, for I used this method to design the featured classes.
After an overview of all classes and the relationships between them, each class will be explained in greater detail.

The chapter is divided into the following sections:

What is Object Orientation?
Classes
Objects
Methods
Encapsulation
Inheritance
Abstraction

The classes and their relationships
Neural net classes
Class NeuralNet
Class BackpropagationNet
Class KohonenFeatureMap
Common components
Class Neuron
Class NeuronLayer
Class WeightMatrix
Net type specific components
Class Pattern
Class MapNeuron
Class NeuronMatrix
Class InputValue
Class InputMatrix

Chapter III using the classes

This chapter explains how to work with the implemented classes.
First there are a few words about Java, the language I used to implement them. After a description of some basic aspects of Java, it is then shown how to use the neural network classes in your own programs.

The chapter is divided into the following sections:

Java = new ProgrammingLanguage()
Working with Java
The look of a class
Creating an object
Invoking a method
Using the classes in your own programs
Using the BackpropagationNet class
The structure of a conversion file
The structure of a pattern file
Using the KohonenFeatureMap class
Using the InputMatrix class
Download

Chapter IV sample applet

This chapter presents a Java applet that uses some of the implemented classes.
The applet demonstrates a Kohonen Feature Map that learns to span itself over a given set of points in three-dimensional space. Several parameters of the neural net can be changed while the example is running.

You will find a more detailed description of the neural

net's purpose and the applet's controls included in this page.

Appendix A glossary

This appendix explains expressions that are relevant in neural net theory.
The chapters have several links to the entries on this page. Once you have been transferred to the glossary page that way, click the right mouse key and choose 'Back' to return to where you came from.

Note that you can jump directly to a specific letter using the links in this page's title bar instead of scrolling down the whole page.
(This feature is not available if you linked from inside a chapter)

Appendix B literature

This appendix shows the literature I used while developing.
Listed are books and World Wide Web addresses, if a source is available online.
As we all know, such addresses are changing sometimes. Due to that fact, I added the date of my last visit on a certain page.

Contents

Chapter I: Neural Net Overview

This chapter gives general information on neural networks. Besides the introduction of several neural net types, the different learning algorithms, suitable for the different net types, are also explained.

The chapter is divided into the following sections:

What are neural nets?
» The biological model: The human brain
» The components of a neural net
» What they can and where they fail

Types of neural nets
» Perceptron
» Multi-Layer-Perceptron
» Backpropagation
» Net Hopfield Net
» Kohonen Feature Map

The learning process
» What does "learning" mean refering to neural nets?
» Supervised and unsupervised learning
» Forwardpropagation
» Backpropagation
» Selforganization

Chapter II: Class Structure

This chapter shows the structure of the implemented classes. At the beginning you will find a description of object orientation, for I used this method to design the featured classes. After an overview of all classes and the relationships between them, each class will be explained in greater detail.

The chapter is divided into the following sections:

What is Object Orientation?
» Classes
» Objects
» Methods
» Encapsulation

» Inheritance
» Abstraction

The classes and their relationships

Neural Net classes
» Class NeuralNet
» Class BackpropagationNet
» Class KohonenFeatureMap

Common components
» Class Neuron
» Class NeuronLayer
» Class WeightMatrix

Net type specific components
» Class Pattern
» Class MapNeuron
» Class NeuronMatrix
» Class InputValue
» Class InputMatrix

Chapter III: Using the Classes

This chapter explains how to work with the implemented classes. First there are a few words about Java, the language I used to implement them. After a description of some basic aspects of Java, it is then shown how to use the neural network classes in your own programs.

The chapter is divided into the following sections:

Java = new ProgrammingLanguage()

Working with Java
» The look of a class
» Creating an object
» Invoking a method

Using the classes in your own programs
» Using the BackpropagationNet class
» The structure of a conversion file
» The structure of a pattern file
» Using the KohonenFeatureMap class
» Using the InputMatrix class

Chapter IV: Sample Applet

This chapter presents a Java applet that uses some of the implemented classes. The applet demonstrates a Kohonen Feature Map that learns to span itself over a given set of points in three-dimensional space. Several parameters of the neural net can be changed while the example is running.

You will find a more detailed description of the neural net's purpose and the applet's controls included in this page.

Appendix A: Glossary

This appendix explains expressions that are relevant in neural net theory. The chapters have several links to the entries on this page. Once you have been transferred to the glossary page that way, click the right mouse key and choose 'Back' to return to where you came from.

Note that you can jump directly to a specific letter using the links in this page's title bar instead of scrolling down the whole page.

Appendix B: Literature

This appendix shows the literature I used while developing. Listed are books and World Wide Web addresses, if a source is available online. As we all know, such addresses are changing sometimes. Due to that fact, I added the date of my last visit on a certain page.

What Are Neural Nets?

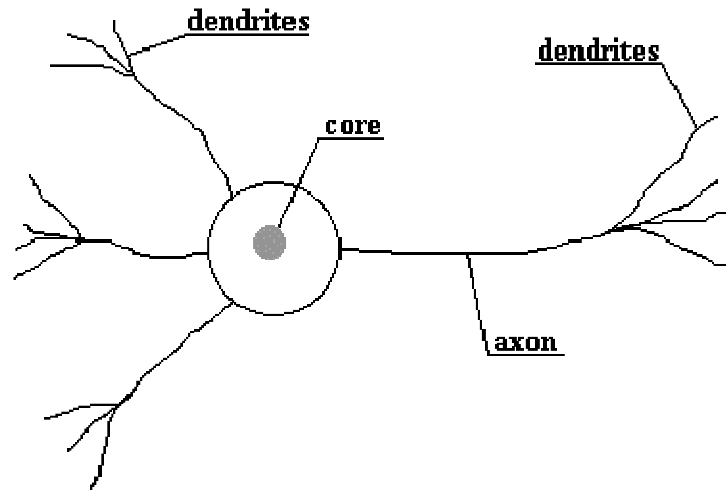
A neural net is an artificial representation of the human brain that tries to simulate its learning process.
The term "artificial" means that neural nets are implemented in computer programs that are able to handle the large number of neccessary calculations during the learning process.

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 information. 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.

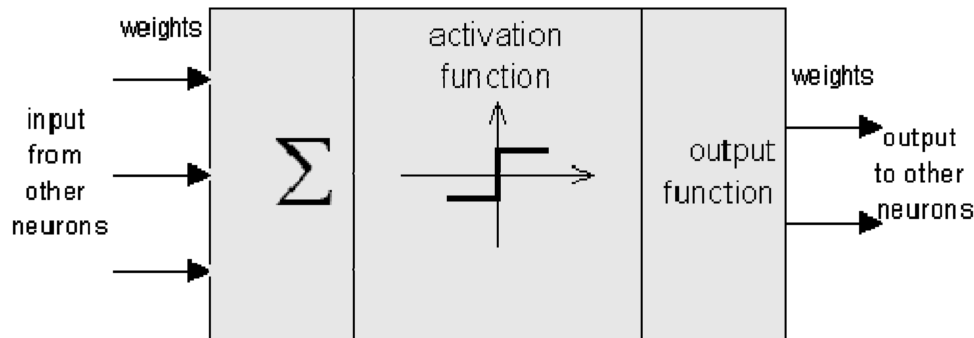
The components of a neural net

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.

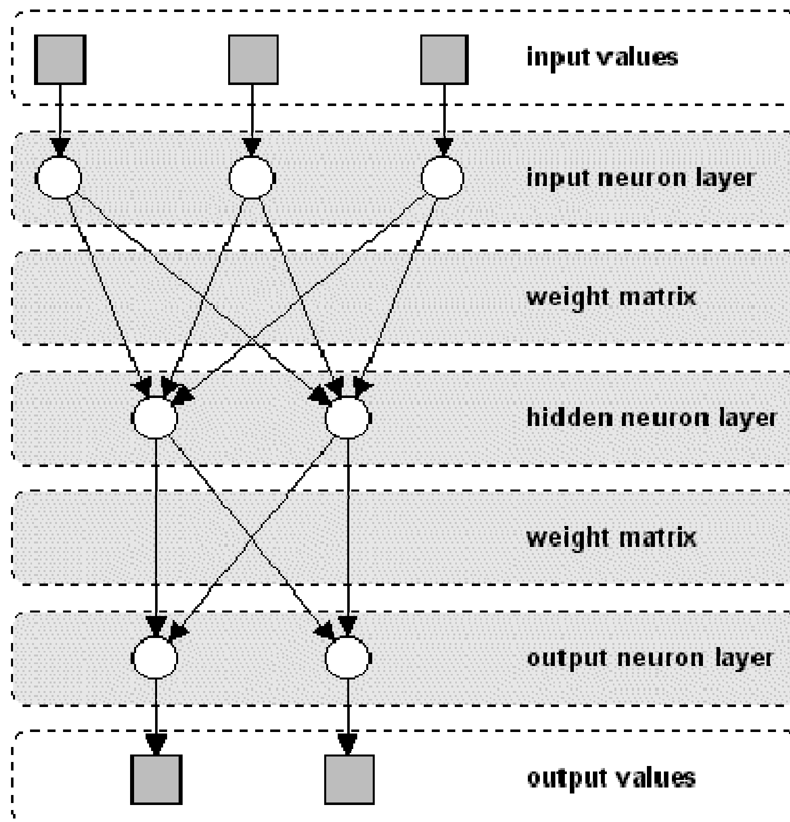


Structure of a neuron in a neural net

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

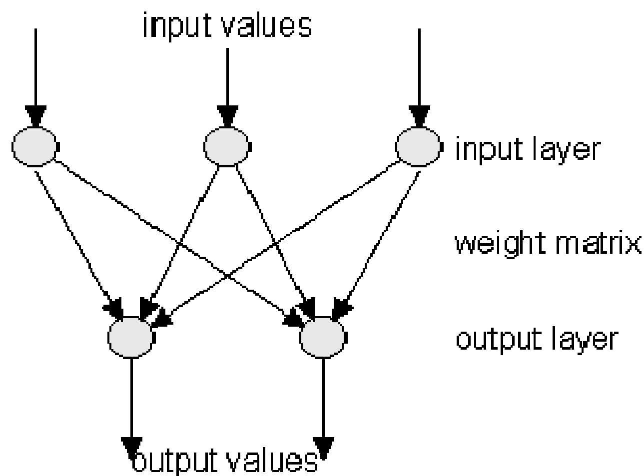
layer.

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.

Sample Structure



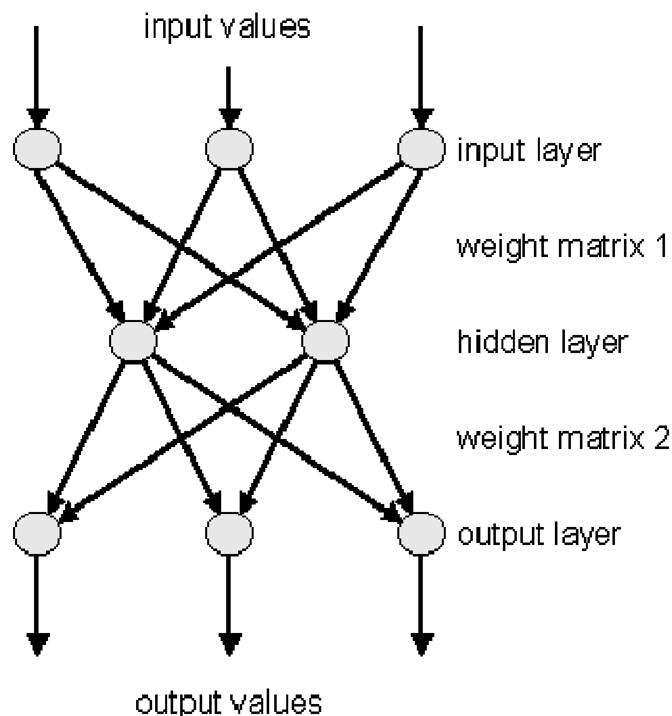
Perceptron

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.

Sample Structure



Multi-Layer-Perceptron

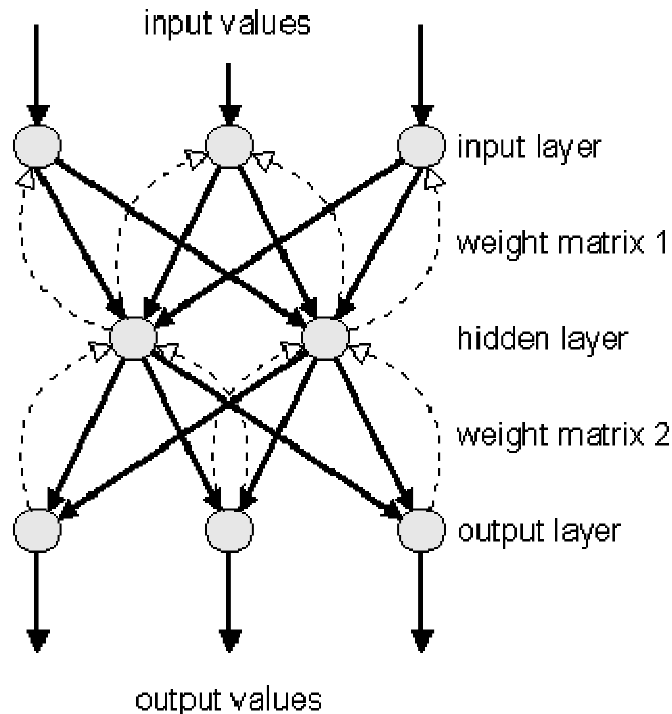
Type	feedforward
Neuron layers	1 input layer, 1 or more hidden layer(s), 1 output layer
Input value types	binary
Activation function	hard limiter / sigmoid
Learning method	supervised
Learning algorithm	delta learning rule, backpropagation (mostly used)

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.

Sample Structure



Backpropagation Net

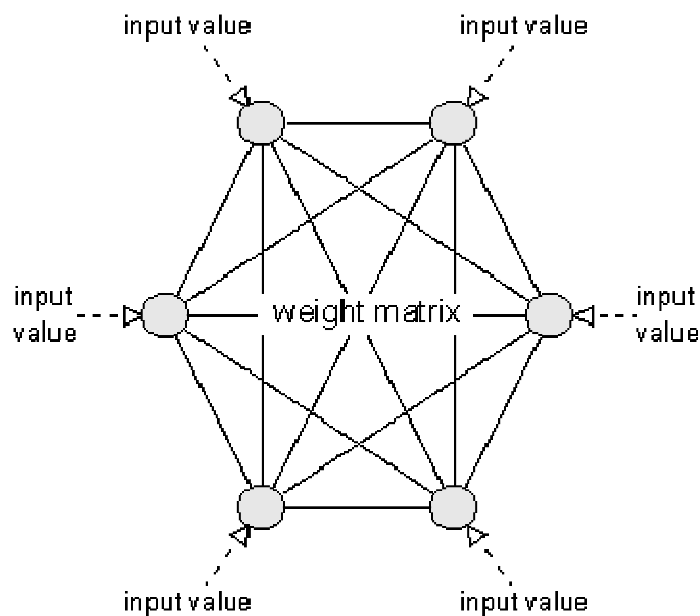
Type	feedforward
Neuron layers	1 input layer, 1 or more hidden layer(s), 1 output layer
Input value types	binary
Activation function	sigmoid
Learning method	supervised

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.

Sample Structure



Hopfield Net

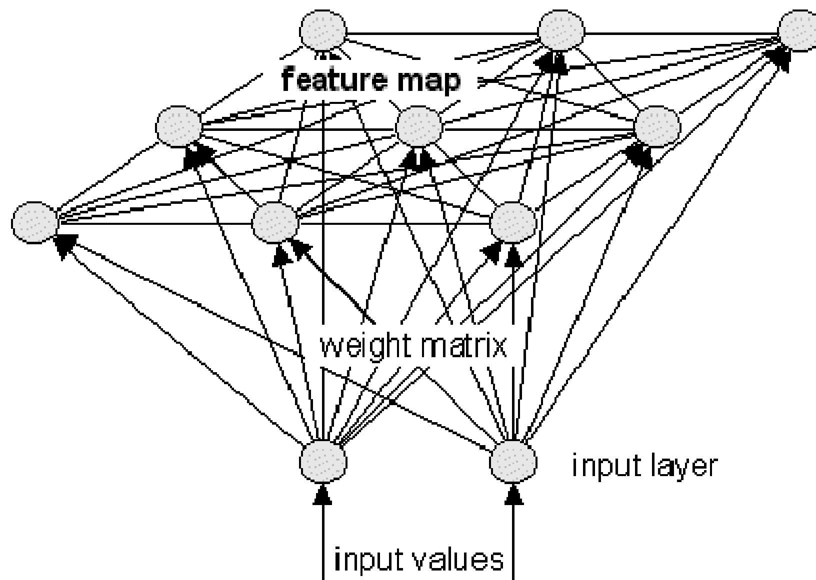
Type	feedback
Neuron layers	1 matrix
Input value types	binary
Activation function	signum / hard limiter
Learning method	unsupervised

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)

Sample Structure



Kohonen Feature Map / Self-organizing Map

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

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.

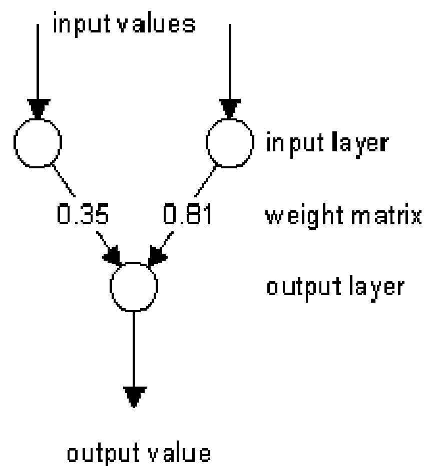
Forwardpropagation

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
4. Repeat this until the output layer is reached
5. Compare the calculated output pattern to the desired target pattern and compute an error value
6. Change all weights by adding the error value to the (old) weight values
7. Go to step 2
8. The algorithm ends, if all output patterns match their target patterns

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



Forwardpropagation in a 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: $(-0.81)^2 + (0.0425)^2 =$ **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.

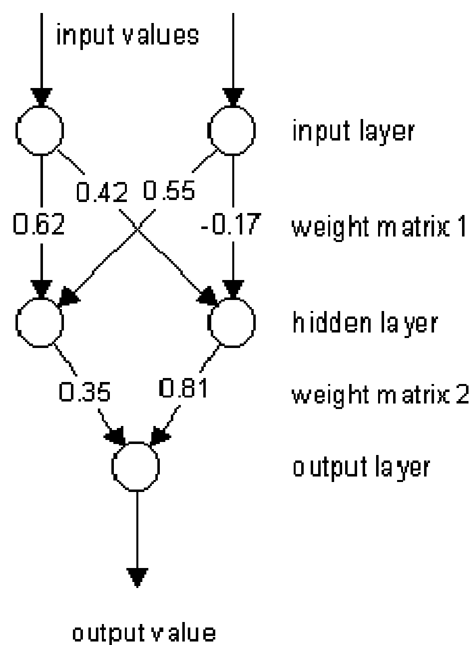
The formula of **sigmoid activation** is:

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

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))
3. Go to step 1
4. The algorithm ends, if all output patterns match their target patterns

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



Backpropagation in a 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.

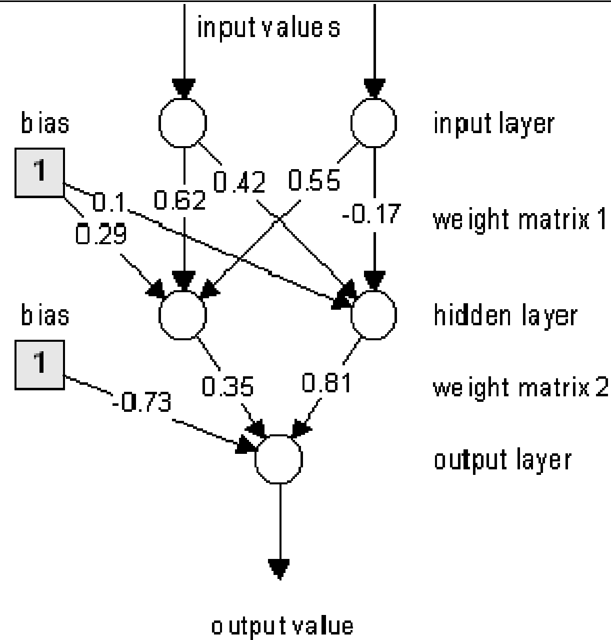
The algorithm is successfully finished, if the net error is zero (perfect) or approximately zero.

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:



Backpropagation in a 3-layered Multi-
Layer-Perceptron using Bias values

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.

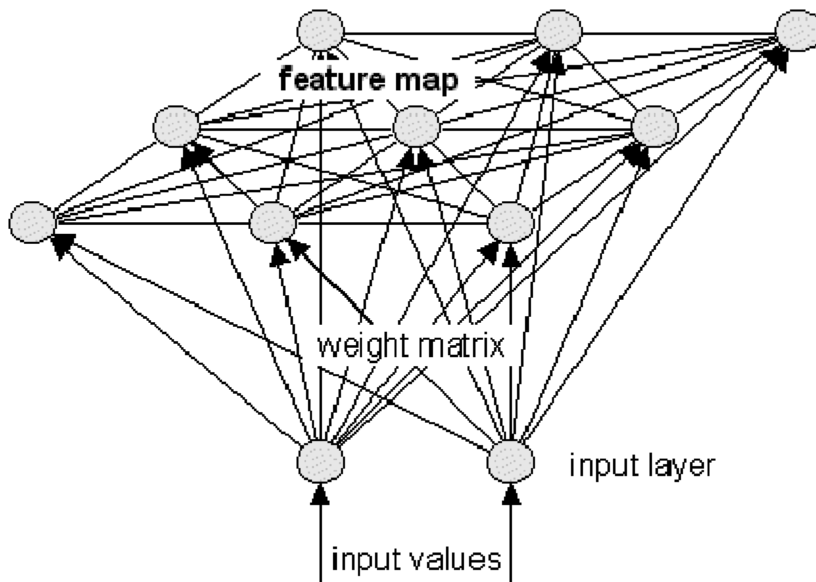
Selforganization

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:

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

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.

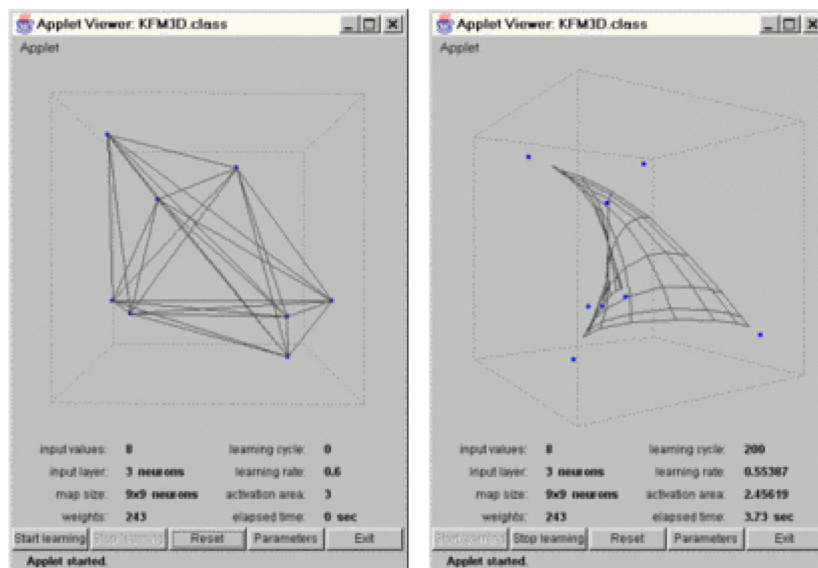
Unlike the biological model, the map neurons don't change their positions on the map. The "arranging" is simulated by changing the values in the weight matrix (the same way as other neural nets do).
Because selforganization is an unsupervised learning algorithm, no input/target patterns exist. The input 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
6. Change the weight values using the formula:
   weight(old) + feedback value * (input value - weight(old)) * learning rate
7. Decrease the activation area
8. Go to step 4
9. The algorithm ends, if the activation area is smaller than a specified value

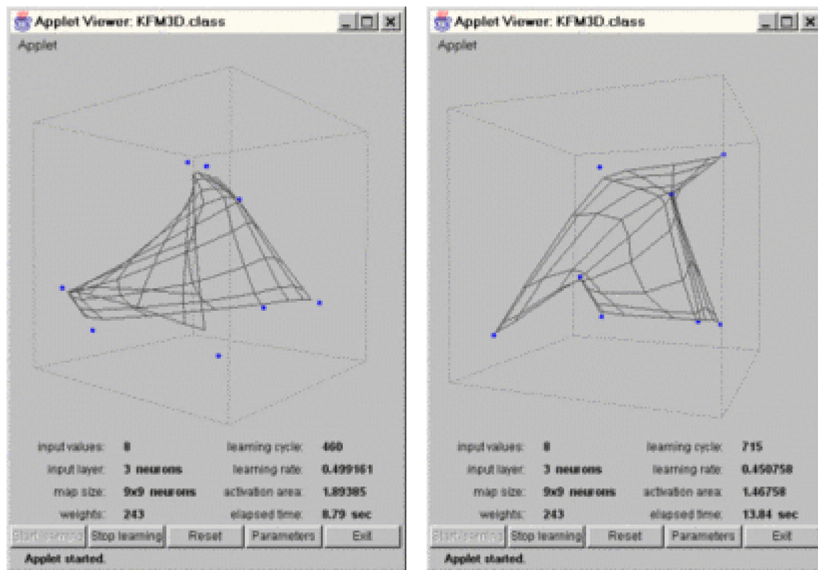
Example: see sample applet

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.



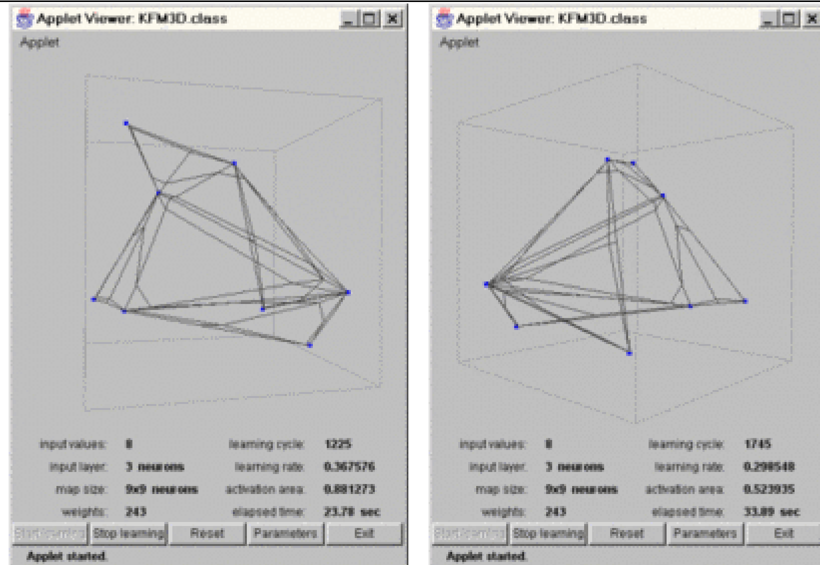
Screenshots of the sample applet showing a 3D Kohonen Feature Map

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.



Screenshots of the sample applet showing a 3D Kohonen Feature Map

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.



Screenshots of the sample applet showing a 3D Kohonen Feature Map

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.

Class Structure

This chapter shows the structure of the implemented classes. At the beginning you will find a description of object orientation, for I used this method to design the featured classes. After an overview of all classes and the relationships between them, each class will be explained in greater detail.

The chapter is divided into the following sections:

What is Object Orientation?
» Classes
» Objects
» Methods
» Encapsulation
» Inheritance
» Abstraction

The classes and their relationships

Neural Net classes
» Class NeuralNet
» Class BackpropagationNet
» Class KohonenFeatureMap

Common components
» Class Neuron
» Class NeuronLayer
» Class WeightMatrix

Net type specific components
» Class Pattern
» Class MapNeuron
» Class NeuronMatrix
» Class InputValue
» Class InputMatrix

What is Object Orientation?

This section gives a short introduction on Object Orientation, its terms and methodology.

Object Orientation (OO) is a concept for developing reusable, easy-to-handle and high-quality software components.
Unlike conventional software development, Object Orientation goes a different way that is more related to real-world problems.

Since the first serious steps in software engineering were done in the 1950's, the nature of software changed through the years.

As Booch describes it:

"As we look back upon the relatively brief yet colorful history of software engineering, we cannot help but notice two sweeping trends:
» The shift in focus from programming-in-the-small to programming-in-the-large
» The evolution of high-order programming languages

Most new industrial-strength software systems are larger and more complex than their predecessors were even just a few years ago. This growth in complexity has prompted a significant amount of useful applied research in software engineering, particularly with regard to decomposition, abstraction, and hierarchy. The development of more expressive programming languages has complemented these advances. The trend has been a move away from languages that tell the computer what to do (imperative languages) toward languages that describe the key abstractions in the problem domain (declarative languages)."
[BOO91, p. 26]

The method is subdivided in several parts:

Object Oriented Analysis (OOA):
"A method of analysis in which requirements are examined from the perspective of the classes and objects found in the vocabulary of the problem domain." [BOO91, p. 516]

Object Oriented Decomposition:
"The process of breaking a system into parts, each of which represents some class or object from the problem domain. The application of object-oriented design methods leads to an object-oriented decomposition, in which we view the world as a collection of objects that cooperate with one another to achieve some desired functionality." [BOO91, p. 516]

Object Oriented Design (OOD):
"A method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design; specifically, this notation includes class diagrams, object diagrams, module diagrams, and process diagrams." [BOO91, p. 516]

Object Oriented Programming (OOP):

"A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. In such programs, classes are generally viewed as static, whereas objects typically have a much more dynamic nature, which is encouraged by the existence of dynamic binding and polymorphism." [BOO91, p. 517]

While traditional structured languages were using functions and procedures that operate on (global) data, object oriented languages are using classes and objects that consist of methods, which operate only on their own internal data.

In the following, the terms of Object Orientation will be described.

Classes

A class is the fundamental element in an object oriented system.
All classes of an object oriented system are arranged in a class hierarchy with one root class at its top.

Campione and Walrath define a class in the following way:

"A class is a blueprint or prototype that defines the variables and the methods common to all objects of a certain kind."
[CW96, "What are classes?"]

What does that mean?
» blueprint: A class can't do anything on its own.
» defines: A class provides something that can be used later.
» objects: A class can only be used, if it had been "brought to life" by instantiating it.

Example: Class "Car"

The class Car

As can be seen, the class, representing a car, has variables (brand, speed, gear, ...) and methods (accelerate, brake, ...). This class serves as a generic description of any existing car, because each real-world thing that is a car, has, for example, a speed, a number of wheels, ... And when driving a car, you can accelerate it or change the current gear.

But you don't drive a generic car with a number of wheels at a speed.
You usually drive a specific car with, for example, four wheels at 50 mph.

This specific car is an instance of the class "Car" and is then called an object.

Objects

Synonym: instance

Booch defines an object as

"Something you can do things to. An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class. [...]"
[BOO91, p. 516]

What does that mean?
» state: An object has a condition, determined by the current values of its variables.
» behavior: An object's state can be changed by applying a certain method.
» identity: Each object can be distinguished from other objects.

Example: Object "MyCar"

An instance of a car:
The object MyCar

The object looks similar to the class it had been instantiated from. This is, because the structure didn't change. By applying certain values to the variables, the object MyCar has now a certain state as well as an identity. It is still a car, for it has the same behaviour as all other cars, which had been defined in the methods of their common class Car.

Now we have created our real-world object and are driving constantly with 50 mph. After a while, we enter the highway, because we want to move faster. So we have to change the car's speed (change the object's state).

This can be done by applying one of an object's methods.

Methods

Synonym: message

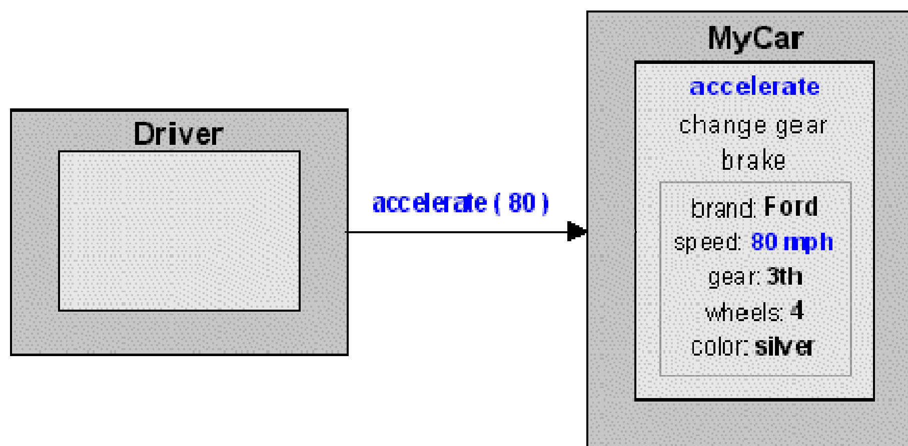
Booch says, a method is

"An operation upon an object, defined as part of the declaration of a class; [...]"
[BOO91, p. 515]

The methods, defined in a class, indicate, what the instantiated objects are able to do.
An object's method is called by other objects of the system.

As Campione and Walrath explain:
"Three components comprise a message:
1. the object to whom the message is addressed
2. the name of the method to perform
3. any parameters needed by the method"
[CW96, "What are messages?"]

As mentioned before, we want to change our car's speed. To achieve this, the car's driver (also an object) has to call the "accelerate" method:



Using the "accelerate" method of the object MyCar

As the figure shows, the "accelerate" method was called with the parameter "80", which indicates the desired speed. By applying that method, the value of the variable "speed" had been changed. You don't change the state of an object (its variables) directly.

The fact that variables of an object are invisible to the outside is called encapsulation.

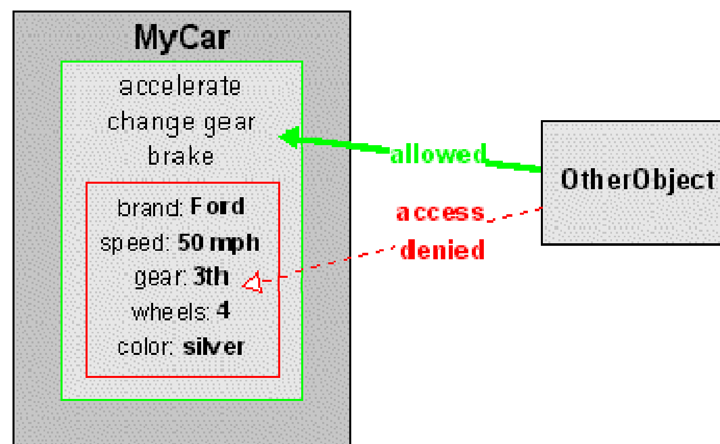
Encapsulation

Synonym: information hiding

Encapsulation is one of the most important characteristics of an object oriented system.

Booch describes encapsulation as

"The process of hiding all of the details of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods. [...]"
[BOO91, p. 513]



Encapsulation of variables in object MyCar

As the figure of the object MyCar shows, all of its variables are enclosed inside the object's methods and may only be changed by those methods. As it is, when you want to accelerate a real-world car, you don't have to know exactly how the car's engine works. You just have to know that you have to step on the gas pedal and change the gear.

This leads to an easy-to-use interface, with benefits for the user, as well as for the developer:

» The user doesn't have to know details of the internal structure and functionality of an object. What he/she must know is, which operations can be done on the object to change its state.

» The developer can change or improve any implementation details, as long as the interface doesn't change. Another advantage is that all functionality is enclosed in the same object, what makes object handling easier.

As mentioned before, an object oriented system consists of "classes arranged in a class hierarchy". One class at the top of this hierarchy serves as the base for other classes, which should have the same or additional properties.

This process is called inheritance.

Inheritance

Inheritance is a very useful mechanism in software development, which is used to create the class hierarchy of an object oriented system.

Booch defines inheritance as

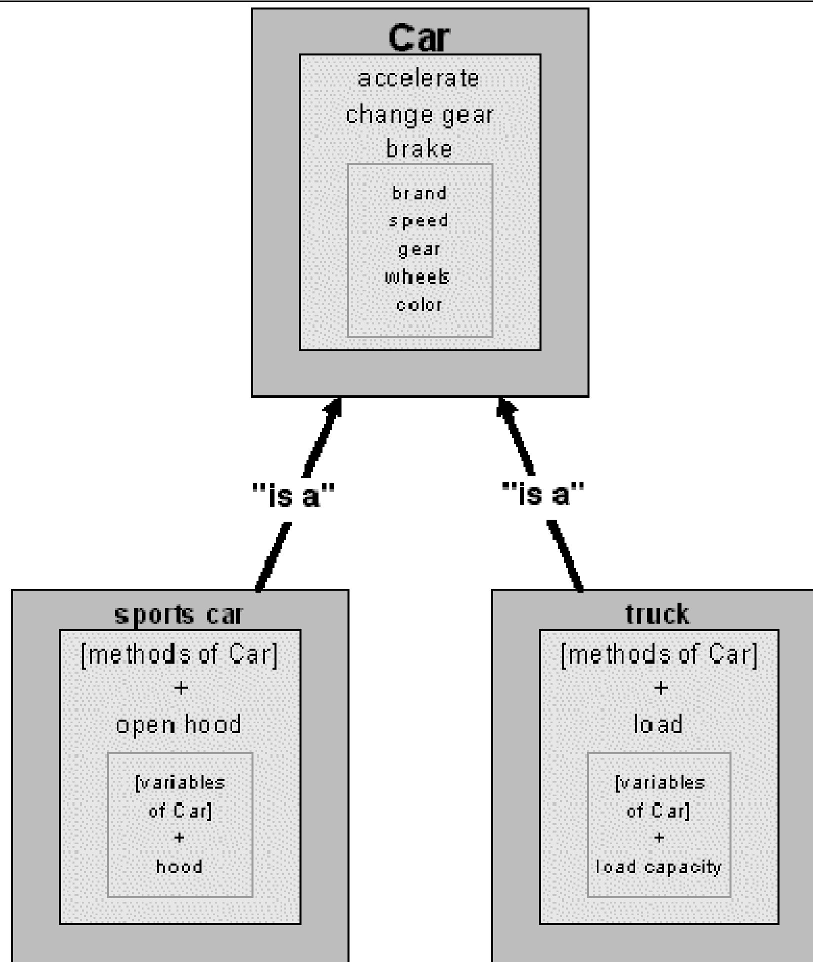
"A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a 'kind of' hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of its superclasses." [BOO91, p. 514]

Note that Java does not support multiple inheritance by definition. Due to that fact, it will not be described here.

Inheritance offers great advantages for software development:

» reusability: The methods and variables once defined in a class, can be used or extended by all subclasses of that class.
» reliability: Methods that worked in the superclass will also work the same way in each inherited class.

Example: Subclasses of the class "Car"



Inheritance: A "sports car" and a "truck" are subclasses of the class Car

Although a sports car is unlike a truck, they both have something in common and thus were defined to be subclasses of a "Car". For example, both types have "a number of wheels" and also can be "accelerated". As can be seen, only the variables and methods specific for one type had to be added in the subclass.
The common methods (defined in Car) may be used by instantiated objects of the subclasses and must not be rewritten. By defining further subclasses, a class hierarchy arises, where classes in the lower part are more and more specialized. On the other hand, classes in the upper part are more generalized.

This leads us to another aspect of object orientation, which is called abstraction.

Abstraction

Abstraction is a very important element of object oriented software development and is used in the design phase to find a suitable hierarchy for a set of classes.

Booch describes abstraction in the following way:

"The essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply-defined conceptual boundaries relative to the perspective of the viewer; the process of focusing upon the essential characteristics of an object. [...]"
[BOO91, p. 512]

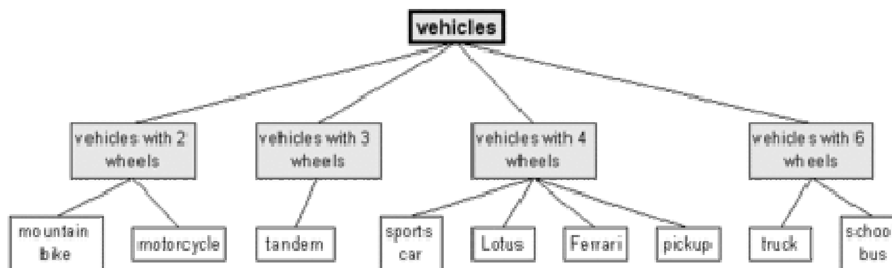
As the definition says, abstractions are "relative to the perspective of the viewer", what means that it depends a lot on the problem domain for which a class structure shall be created.

The search for abstractions starts after the objects of the problem domain have been found in the analysis phase. What follows is the most difficult and critical part of the development process, because the design of the abstraction levels determines the quality of the whole system.

Example:
Suppose, the result of the analysis phase were the following objects:
» sports car
» mountain bike
» motorcycle
» tandem
» school bus
» pickup
» Ferrari
» truck
» Lotus

Obviously, the objects represent different kinds of vehicles. By realizing that fact, we have found our first abstraction and will take a class "vehicles" as the base class of our hierarchy.

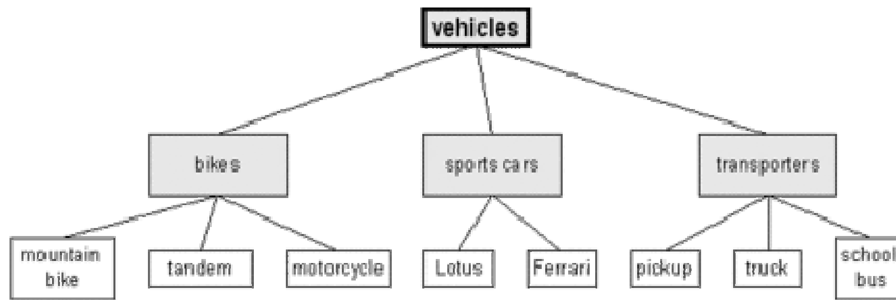
A first attempt in building a hierarchy may look like:



First attempt in building a class hierarchy (Click to enlarge)

What can be said about this hierarchy is: "Don't try this at home!". Although it contains all objects and therefore it's "correct", it is not very intelligent. An evident violation of the abstraction principle is, for example, that a "sports car" can be found on the same level as a "Ferrari" and a "Lotus". This must be wrong, because they both are kinds of a "sports car".

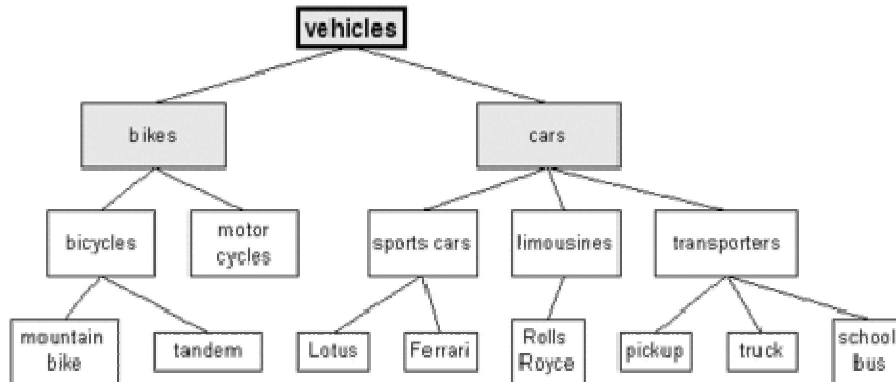
So we change our hierarchy to the following:



Second attempt in building a class hierarchy (Click to enlarge)

Now we have got a more structured hierarchy with the subclasses "bikes", "sports cars" and "transporters". But they are still too specific. For instance, if a new object "Rolls Royce" is to be added to the hierarchy, where should it be inserted, for it is no sports car?

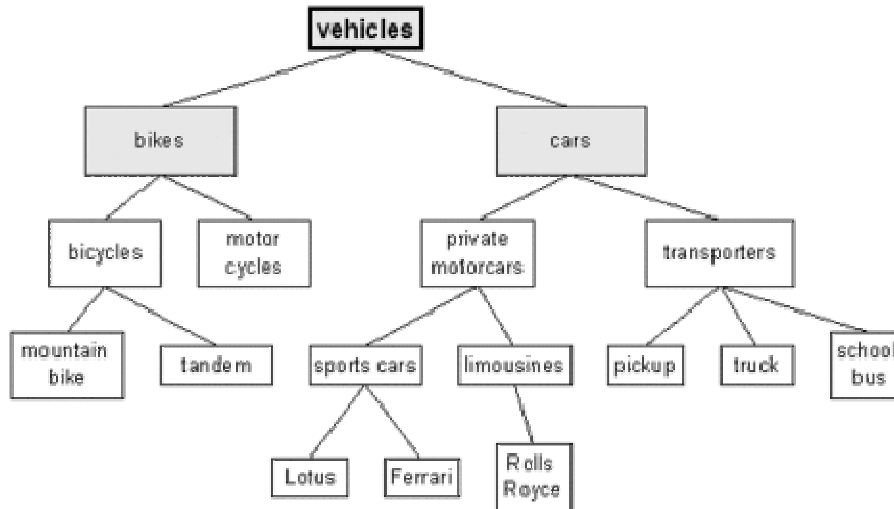
We rearrange our structure by introducing a new class "cars":



An important abstraction: The new class "cars" (Click to enlarge)

Now there are four levels of abstraction. The new class "cars" may contain the properties that are common to all of its subclasses. But there is still something wrong with our sports cars and limousines. As you can see, they are arranged on the same abstraction level as, for example, a "truck". But a truck is a more abstract thing than a Ferrari or a Rolls Royce.

We try to improve our hierarchy by doing another abstraction:



A new abstraction: The class "private motorcars" (Click to enlarge)

The fifth abstraction level had been created and our class hierarchy is now much more structured than it was at our first attempt, but is still far from being finished.

This was an example of the abstractioning process that should only demonstrate, how a class hierarchy can be built up and which thoughts have to be made.

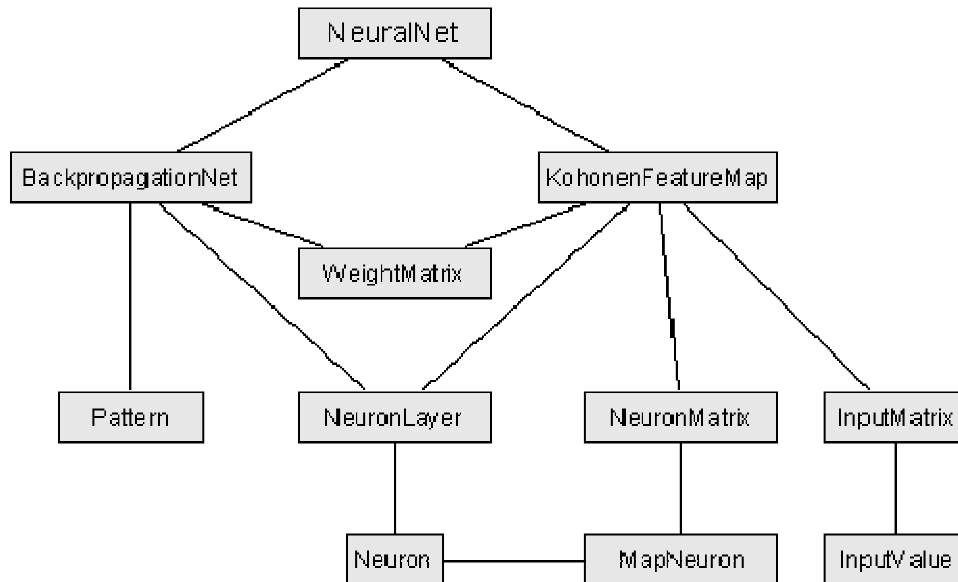
The next sections consist of my class hierarchy for neural networks.
Because neural nets are themselves abstract things, the design process was, literally, an "abstraction of abstractions" and, unlike the previous example, took me a few more attempts to get any results.

The classes and their relationships

First a few words about my motivation to write this class structure.
Many applications were written for the existing neural network types. Each of those programs is focussing on a special problem to be solved.
So if you want to implement a similar neural net to one already written, you might either try to understand the source code (that means "fun" for many hours) or copy the source code and modify it until it fits your own net's purpose (what is not the best way of developing software).

Due to the fact that most of the existing neural net types have some common components, I thought it would be nice, if I had to implement those components once and may use them later in different applications, like some sort of a construction kit.
I decided to write classes suitable for the Backpropagation Net and the Kohonen Feature Map, because (in my honest opinion) these are the most powerful types of neural networks.

And here is the result of my work:
(You should always keep in mind that this was my first object oriented piece of software, so don't blame me!)



The classes and their relationships

This is not to be read as a class hierarchy, as explained in the previous section. The levels do not represent levels of abstraction.
The figure only shows, which classes are present and what classes have relationships among each other.

Furthermore, it shall demonstrate my primary goal: The integration of common components.

As you can see, for example, the WeightMatrix class and the NeuronLayer class are commonly used by the two net classes (BackpropagationNet and KohonenFeatureMap) although their structure differs in both net types. Object Orientation rules!

The following sections explain the classes in greater detail.

Neural Net classes

These classes represent two neural net types (Backpropagation Net and Kohonen Feature Map).

There is also an abstract class called "NeuralNet", which is the superclass of both types. This class contains generic methods, e.g. a method to set the initial learning rate of a neural net.

Class NeuralNet

This class is the first class in the structure.

abstract class NeuralNet

Extends

java.lang.Object

Instantiated by

(can't be instantiated)

Constructors

(none)

Methods

boolean displayNow ()
Indicates, whether the net should be drawn or not, depending on its display step. True, if the net should be drawn. False otherwise.

boolean finishedLearning ()
Indicates that the net has finished learning. True, if the learning process is finished. False otherwise.

String getElapsedTime ()
Returns the elapsed learning time of a neural net.

int getLearningCycle ()
Returns the current learning cycle of a neural net.

double getLearningRate ()
Returns the current learning rate of a neural net.

int getMaxLearningCycles ()
Returns the number of maximum learning cycles of a neural net.

void resetTime ()
Resets the net's learning time.

void setDisplayStep (int displayStep)
Sets a value that indicates the interval to display the net.

void setLearningRate (double learningRate)
Sets the learning rate of a neural net.

void setMaxLearningCycles (int maxLearningCycles)
Sets the number of learning cycles, the net shall perform.
If -1, the net has no maximum cycle.

Class BackpropagationNet

This class represents a Backpropagation Net neural net.

public class BackpropagationNet

Extends

java.lang.Object
  NeuralNet

Instantiated by

Application

Constructors

public BackpropagationNet ()

Methods

void addNeuronLayer (int size)
Adds a neuron layer with size neurons.
Note that neuron layers are sequentially added to the net.

void connectLayers ()
Connects all neuron layers with weight matrices.
Must be called after all neuron layers have been added.

double getAccuracy ()
Returns the accuracy value.

double getError ()
Returns the current error of the net.

String getInputPattern (int patternNr)
Returns the input pattern with number patternNr.

double getMinimumError ()
Returns the minimum error of a neural net.

float[] getNeuronOutputs (int layerNr)
Returns the output values of all neurons in layer layerNr.

int getNumberOfLayers ()
Returns the number of neuron layers.

int getNumberOfNeurons (int layerNr)
Returns the number of neurons in layer layerNr.

int getNumberOfPatterns ()
Returns the number of patterns.

int getNumberOfWeights ()
Returns the number of weights of all weight matrices.

int getNumberOfWeights (int matrixNr)
Returns the number of weights in weight matrix matrixNr.

String getOutputPattern (int patternNr)
Returns the output pattern with number patternNr.

float getPatternError (int patternNr)
Returns the error of output pattern patternNr.

String getTargetPattern (int patternNr)
Returns the target pattern with number patternNr.

float[][] getWeightValues (int matrixNr)
Returns the weight values of weight matrix matrixNr.
The values for matrixNr start with zero!

void learn ()
Performs one learning step.

synchronized void readConversionFile (String conversionFileName)
Reads a conversion table for ASCII-binary values from file conversionFileName.

synchronized void readPatternFile (String patternFileName)

Reads input and target patterns from file patternFileName.

String recall (String recallInput)
Tries to recall the correct output for a learned input pattern recallInput.

void setAccuracy (double accuracy)
Sets an accuracy value for the net, which is something like a "fuzzy border" for output/recall purposes (default is 0.2).

void setMinimumError (float minimumError)
Sets the minimum error of a neural net.

Class KohonenFeatureMap

This class represents a Kohonen Feature Map neural net.

public class KohonenFeatureMap

Extends

java.lang.Object
  NeuralNet

Instantiated by

Application

Constructors

public KohonenFeatureMap ()

Methods

void connectLayers (InputMatrix inputMatrix)
Connects the feature map and the input layer (which is generated depending on the size of the inputMatrix) with a weight matrix.

void createMapLayer (int xSize, int ySize)
Creates a two-dimensional feature map with xSize*ySize map neurons.

double getActivationArea ()
Returns the current activation area.

double getInitActivationArea ()
Returns the initial activation area.

double getInitLearningRate ()
Returns the initial learning rate.

int getMapSizeX ()
Returns the number of neurons in the map layer's x-dimension.

int getMapSizeY ()
Returns the number of neurons in the map layer's y-dimension.

int getNumberOfWeights ()
Returns the number of weights in the weight matrix.

double getStopArea ()
Returns the final activation area.

float[][] getWeightValues ()
Returns the weight values of the net's weight matrix.

void learn ()
Performs a learning step.

void setInitActivationArea (double initActivationArea)
Sets the initial activation area.

void setStopArea (double stopArea)
Sets the final activation area at which the net stops learning.

void setInitLearningRate (double initLearningRate)
Sets the initial learning rate.

Common components

These classes represent neural net components that are commonly used by neural nets, especially the Backpropagation Net and the Kohonen Feature Map. The three classes listed here are: Neuron, NeuronLayer and WeightMatrix.

Note that these classes are only used by the neural net classes in the structure. This means, you don't have to worry about instantiation and things like that, for it's all handled by other classes.

Class Neuron

This class represents a neuron of a neural net's neuron layer.

public class Neuron

Extends

java.lang.Object

Instantiated by

NeuronLayer

Constructors

public Neuron ()

Methods

void activateSigmoid ()
Performs a sigmoid activation on a neuron's input and computes its output.

void computeOutputError (float targetValue)
Computes the output error of a neuron in an output neuron layer by comparing the neuron's current output with the targetValue.

float getInput ()
Returns the input value of a neuron.

float getOutput ()
Returns the output value of a neuron.

float getOutputError ()
Returns the output error value of a neuron.

void init (float input)
Initializes a neuron with an input value.

Class NeuronLayer

This class represents a neuron layer of a neural net.

public class NeuronLayer

Extends

java.lang.Object

Instantiated by

BackpropagationNet
KohonenFeatureMap

Constructors

public NeuronLayer (int size)
Creates a neuron layer with size neurons.

Methods

void computeLayerError (Pattern pattern)
Computes the output error values of all neurons in a layer by comparing the output with a target pattern.

void computeOutput ()
Computes the output values of all neurons in a layer.

float[] getLayerError ()
Returns the output error values of all neurons in a layer.

float[] getOutput ()
Returns the output values of all neurons in a layer.

void setInput (InputValue value)
Sets a value as an input to a neuron layer.
Only used by the input layer of a Kohonen Feature Map.

void setInput (Pattern pattern)
Sets a pattern as an input to a neuron layer.
Only used by the input layer of a Backpropagation Net.

int size ()
Returns the number of neurons in a layer.

Class WeightMatrix

This class represents a weight matrix between two neuron layers of a neural net.

public class WeightMatrix

Extends

java.lang.Object

Instantiated by

BackpropagationNet
KohonenFeatureMap

Constructors

public WeightMatrix (int xSize, int ySize, boolean withBias)
Creates a weight matrix with xSize*ySize weights. The withBias flag indicates, whether Bias values shall be used or not.

Methods

void changeWeights (float[] precLayer, float[] succLayer, double learningRate)
Changes all weights using the Backpropagation learning algorithm.
Only used by BackpropagationNet.

void changeWeightsKFM (float[] precLayer, float[] succLayer, double learningRate)
Changes all weights using the Selforganization learning algorithm.
Only used by KohonenFeatureMap.

float[] getBiases ()
Returns the values of all biases in a weight matrix.

float[] getInputWeights (int targetNeuron)
Returns the values of all incoming weights that lead to the neuron with number targetNeuron.

float[] getOutputWeights (int sourceNeuron)
Returns the values of all outgoing weights that lead away from the neuron with number sourceNeuron.

float[][] getWeights ()
Returns the values of all weights in a weight matrix.

void init ()
Initializes all weights with values ranging from -1.0 to 1.0.

void init (float[][] iweight)
Initializes all weights with specified values stored in iweight.

void init (InputValue[] iv, int dimension)
Initializes all weights with InputValues stored in iv that have the dimension dimension.
Only used by KohonenFeatureMap.

int size ()
Returns the number of weights in a weight matrix.

Net type specific components

These classes represent neural net components that are specific either to the Backpropagation Net or the Kohonen Feature Map.

The specific class for the BackpropagationNet is: Pattern
The specific classes for the KohonenFeatureMap are: MapNeuron, NeuronMatrix, InputValue, InputMatrix

Class Pattern

This class represents a pattern that contains input or target values for a neural net. It is a specific class for supervised learning neural nets.

public class Pattern

Extends

java.lang.Object

Instantiated by

BackpropagationNet

Constructors

public Pattern (String patString, String[][] conversionTable)
Creates a pattern from patString using a conversionTable.

Methods

int size ()
Returns the number of pattern values in a pattern.

Class MapNeuron

This class represents an element of a NeuronMatrix. It is a specific class for a Kohonen Feature Map neural net.

public class MapNeuron

Extends

java.lang.Object
  Neuron

Instantiated by

NeuronMatrix

Constructors

public MapNeuron ()
public MapNeuron (int x, int y)
Creates a map neuron with a x and y position on the feature map.

Methods

int getXPos ()
Returns the x-coordinate of a map neuron's position on the map.

int getYPos ()
Returns the y-coordinate of a map neuron's position on the map.

void init (int x, int y)
Initializes a map neuron's position on the map with a x and y coordinate.

Class NeuronMatrix

This class represents the map layer of a Kohonen Feature Map. It is a specific class for a Kohonen Feature Map neural net.

public class NeuronMatrix

Extends

java.lang.Object

Instantiated by

KohonenFeatureMap

Constructors

public NeuronMatrix (int size)
Creates a one-dimensional neuron matrix with size map neurons.

public NeuronMatrix (int xSize, int ySize)
Creates a two-dimensional neuron matrix with xSize*ySize map neurons.

Methods

MapNeuron[] getMapNeurons ()
Returns all map neurons of a neuron matrix.

void init (InputValue[] inputValues)
Initializes a neuron matrix with input values from inputValues.

int size ()
Returns the number of map neurons in this neuron matrix.

int xSize ()
Returns the number of map neurons in x-direction.

int ySize ()
Returns the number of map neurons in y-direction.

Class InputValue

This class represents an element of an InputMatrix. It is a specific class for a Kohonen Feature Map neural net.

public class InputValue

Extends

java.lang.Object

Instantiated by

InputMatrix

Constructors

public InputValue ()
public InputValue (int x, int y, int z)
Creates an input value with x, y and z coordinate.

Methods

int getX ()
Returns the x coordinate of an input value.

int getY ()
Returns the y coordinate of an input value.

int getZ ()
Returns the z coordinate of an input value.

void setX (int x)
Sets the x coordinate of an input value.

void setY (int y)
Sets the y coordinate of an input value.

void setZ (int z)
Sets the z coordinate of an input value.

Class InputMatrix

This class represents the input of a Kohonen Feature Map. Although it is called a "matrix", the elements are stored in an array. It is a specific class for a Kohonen Feature Map neural net.

public class InputMatrix

Extends

java.lang.Object

Instantiated by

KohonenFeatureMap

Constructors

public InputMatrix (int size, int dimension)
Creates an input matrix with size input values and dimension dimensions.

Methods

int getDimension ()
Returns the dimension of an input matrix.

void setInputX (int[] inputX)
Initializes all x-dimension values of an input matrix with values from inputX.

void setInputY (int[] inputY)
Initializes all y-dimension values of an input matrix with values from inputY.

void setInputZ (int[] inputZ)
Initializes all z-dimension values of an input matrix with values from inputZ.

void setInputValues (int[] inputX, int[] inputY, int[] inputZ)
Initializes the values for all dimensions of an input matrix with values from inputX, inputY and inputZ.

int size ()
Returns the number of input values in an input matrix.

Using the Classes

This chapter explains how to work with the implemented classes. First there are a few words about Java, the language I used to implement them. After a description of some basic aspects of Java, it is then shown how to use the neural network classes in your own programs.

The chapter is divided into the following sections:

Java = new ProgrammingLanguage()

Working with Java
» The look of a class
» Creating an object
» Invoking a method

Using the classes in your own programs
» Using the BackpropagationNet class
» The structure of a conversion file
» The structure of a pattern file
» Using the KohonenFeatureMap class
» Using the InputMatrix class

Java = new ProgrammingLanguage()

"Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language."
[JS96]

This is how JavaSoft, the Java-related organization of Sun Microsystems, describes the new programming language in its White Paper.

I don't want to explain each of these "buzzwords" in detail, as other authors do [see FLA96, pp. 4-9], but only those that were relevant for my decision to use Java as the language for the implementation of my classes.

Simple

Java isn't a completely new language. Its syntax and constructs look very familiar to those in other languages like C or C++, what makes learning easy for programmers of these languages (I've experienced that by myself).

By speaking of a simple language, it is meant that some (critical or rarely used) features of C or C++ are no longer present in Java. Examples for such simplifications are that Java doesn't support struct or union constructs or multiple inheritance.

Flanagan mentiones other advantages that Java provides:

"Perhaps the most important simplification, however, is that Java does not use pointers. Pointers are one of the most bug-prone aspects of C and C++ programming. Since Java does not have structures, and arrays and strings are objects, there's no need for pointers. Java automatically handles the referencing and dereferencing of objects for you. Java also implements automatic garbage collection [sometimes...; author's remark], so you don't have to worry about memory management issues. All of this frees you from having to worry about dangling pointers, invalid pointer references, and memory leaks, so you can spend your time developing the functionality of your programs."
[FLA96, p. 5]

These facts indeed make Java a lot easier to work with than other languages do.

Working with Java does not neccessarily require programming experience in other languages, although it would be helpful. I think, it's also a good way for novices to make first steps in programming.

Object-oriented

When working with Java, you can't help but going the object-oriented way, for nearly everything in Java is an object.

As Flanagan describes it:

"Unlike C++, Java was designed to be object-oriented from the ground up. Most things in Java are objects; the simple numeric, character, and boolean types are the only exceptions."
[FLA96, p. 6]

To provide a quick approach, the Java API offers a lot of already designed classes, arranged in packages.

The packages of the JDK, Version 1.0.2, are the following:
java.lang
Package that contains essential Java classes, including numerics, strings, objects, compiler, runtime, security, and threads. This is the only package that is automatically imported into every Java program.

java.io
Package that provides classes to manage input and output streams to read data from and write data to files, strings, and other sources.

java.util
Package that contains miscellaneous utility classes, including generic data structures, bit sets, time, date, string manipulation, random number generation, system properties, notification, and enumeration of data structures.

java.net
Package that provides classes for network support, including URLs, TCP sockets, UDP sockets, IP addresses, and a binary-to-text converter.

java.awt
Package that provides an integrated set of classes to manage user interface components such as windows, dialog boxes, buttons, checkboxes, lists, menus, scrollbars, and text fields. (AWT = Abstract Window Toolkit)

java.image
Package that provides classes for managing image data, including color models, cropping, color filtering, setting pixel values, and grabbing snapshots.

java.peer
Package that connects AWT components to their platform-specific implementations (such as Motif widgets or Microsoft Windows controls).

java.applet
Package that enables the creation of applets through the Applet class. It also provides several interfaces that connect an applet to its document and to resources for playing audio.
[GY96]

The classes within these packages can be used as the base for own programs to be written. In addition, the source codes of all classes are freely available and may be studied by advanced programmers to understand Java in detail.

Network-savvy

This aspect is probably the one, which made Java such a popular language.

Since the past few years, the World Wide Web has caused some kind of enthusiasm all over the world. Everybody speaks of the "Global Village" and the "Information Superhighway". And Java supports the requirements of distributed applications perfectly.

Providing the neccessary classes and routines for network-wide operations in the java.net package, Java makes programming for the Internet much more easier than it was in languages as C and C++.

For instance, connecting to a resource and transmitting its content can now be done the same way as opening and reading a local file.

Besides, the possibility of integrating Java programs into Web pages in form of applets is a great and useful way to visualize different subjects, users can interact with over the net.

Architecture neutral

As mentioned before, Java was designed to support "programming for the Internet". Because the Internet is an Open System, consisting of a large number of different computer systems, you need applications that are running on any of these systems.

Gone are the times where software had to be rewritten/recompiled if it should run on a different architecture, as Flanagan explains:

"[...] Java programs are compiled to an architecture neutral byte-code format. The primary advantage of this approach is that it allows a Java application to run on any system, as long as that system implements the Java Virtual Machine. Since Java was designed to create network-based applications, it is important that Java applications be able to run on all of the different kinds of systems found on the Internet."
[FLA96, p. 8]

From the developer's view this means a great improvement, because the specific properties of different systems (like numeric representation or threads control) have not to be worried about any longer. This leads to faster software development.

Once an application had been written and compiled on, for example, an IBM PC running Windows95/NT, the produced byte-code runs also on a UNIX workstation or a PowerPC Macintosh without any modifications. Using the classes of the java.awt package, the application also has the appropriate look for each system.

After all that praising, a few words must be said about Java's "other side":

They speak of high-performance. You probably don't agree to that statement immediately, but there is a reason why Java programs seem to run slowly.
"Java is an interpreted language, so it is never going to be as fast as a compiled language like C. In fact, Java is on the average about 20 times slower than C. But before you throw up your arms in disgust, be aware that this speed is more than adequate to run interactive, GUI and network-based applications, where the application is often idle, waiting for the user to do something, or waiting for data from the network."
[FLA96, p. 8]

A real disadvantage of Java that might lead developers to frustration is obvious, when a user interface is to be implemented.

Java provides so called "Layout Managers" for different purposes that are very hard to work with. For example, if you want to arrange layout components in a window, there is no possibility to specify absolute coordinates. Instead, you have to deal with relative positions, what permanently leads to "funny" results. (Don't ask me, how long I worked on the tiny Parameters window of the sample applet. A "relative" long time...)

But the Java language specification is changing constantly.
While writing these pages, the new JDK Version 1.1 had been announced, which will provide some additional packages and, hopefully, removes the lacks of previous releases.

Working with Java

This section describes some basics of Java and is by no means a complete language description.

Assuming that you have already worked with Java, it shall only remind you of the most important things you have to know, when you use the neural network classes.

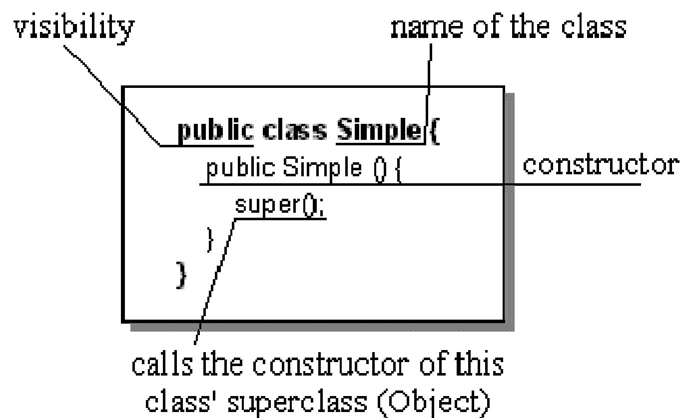
If you want to get more information about Java, I strongly recommend you to take a closer look on The Java Tutorial, written by Mary Campione and Kathy Walrath. It explains each feature of the Java language in detail, is structured very well, and had been written in an easy-to-read way.

The look of a class

As we have already heard, an object-oriented system consists of a hierarchy of classes with one root class at its top.

The root class in the Java API is the Object class, located in the java.lang package. It is a class with very general behaviour and all other classes inherit and extend this behaviour.

Now it will be shown, what a class looks like, by defining a simple class:



A simple class

This simple class has a "header" that consists of the visibility (indicates, from where this class may be instantiated), the word class and a name for this class.

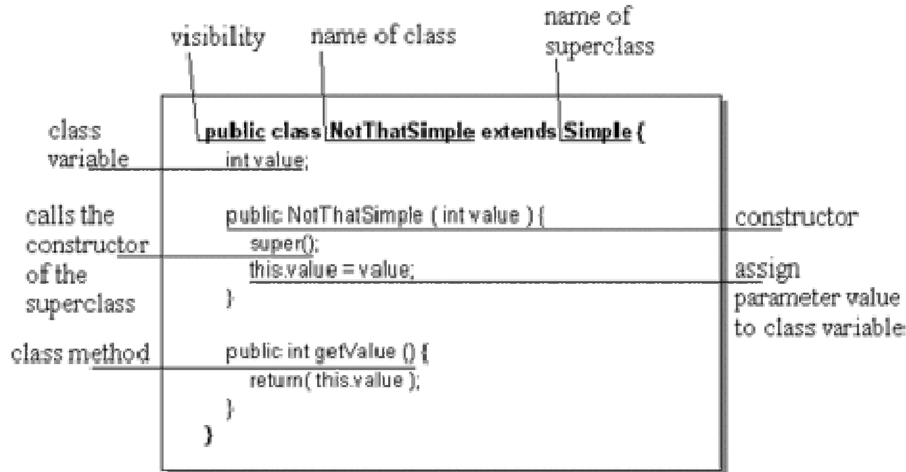
The "body" of this class only has a constructor that is used, when this class is instantiated. Each statement inside the constructor will be executed and usually contains statements that initialize variables or call methods.

The statement super() (always called implicitely) simply calls the constructor of this class' superclass. Because the Simple class is not explicitely derived from any other class by adding

extends [name of superclass], Java automatically declares the Object class to be the superclass.

If you are a C++ programmer, you are probably looking for a destructor. You won't find one, because, as mentioned in the previous section, Java automatically handles the dereferencing of objects with its garbage collection.

For this simple class is pretty useless, let's extend it to a not that simple class:



A not that simple class (Click to enlarge)

As you can see, this class is derived from the Simple class by adding extends Simple after the class name. Thus, our new class inherits all the functionality and behaviour of the Simple class, as well as of the Object class.

This class now has one class variable (or member variable) value and a method getValue(), which returns the variables' value.

The constructor has now something more to do: First, the superclass constructor is called by super() (which calls its own superclass constructor; that's inheritance!) and then assigns the parameter value to the class variable value. For the names of the parameter and the class variable are the same, the prefix this. must be added to the variable's name to indicate that it belongs to a variable of this class.

The method getValue() is also given a visibility, which indicates from where this method may be invoked. Giving the method a public visibility means that this method may be called from any other class.

The possible visibilities of a method, regarding to same class, subclass, package, and world are:

public: The method may be called from anywhere.
protected: The method may be called from anywhere but from the world.

friendly: The method may be called from inside the same class, as well as from the same package.
private: The method may be called only from inside the same class.

Now we want to use this class and thus have to create an object.

Creating an object

To create an object, a Java class is instantiated by using the new operator. So, if we want to create an object of the NotThatSimple class, we would have to write:
NotThatSimple notThatSimpleObject = new NotThatSimple(42);

This statement creates a new NotThatSimple object and performs the following three actions:

1. declaration,
2. instantiation (object creation), and
3. initialization.

Declaration

The declaration part could have been written separately by writing:
NotThatSimple notThatSimpleObject;

This statement only creates a variable named notThatSimpleObject to hold an object of type NotThatSimple, but does not instantiate it. As it is common use in Java (and the Style Guide fanatic I am), I will use names that begin with an uppercase letter for class names and a lowercase letter for object names.

Instantiation

The object is instantiated by the new operator, followed by a class constructor call. The instantiation part is the following statement:
new NotThatSimple(42);

The new operator creates an object and returns a reference to that object:
notThatSimpleObject = new NotThatSimple(42);

Note that before you can write the object instantiation in this form, you must have declared notThatSimpleObject to be of type NotThatSimple earlier. Otherwise, you'll get an exception (error), every Java programmer is familiar with: the NullPointerException.

Initialization

The initialization of an object is done by the class constructor call after the new operator:
NotThatSimple(42);

Each class has at least one constructor, which can easily be distinguished from a method, because it has the same name as the class and has no return type. Our NotThatSimple class has only one constructor that takes one integer as an argument (the number "42" in the example).
[see CW96, "Creating Objects"]

Now we have got our object notThatSimpleObject we can work with.

As mentioned in the previous chapter, an object has state, behaviour, and identity.
The state of our object is determined by the current value of the variable value. To make clear that a variable belongs to an object (instance of a class), it is explicitely called instance variable.
The object's behaviour is determined by the methods it (or any of its superclasses) can perform.
Our object has only one method getValue(), which may be used to get the object's current state.
And finally, our object also has an identity, determined by the object's name.

As we heard before, a class serves as a "blueprint" for similar objects.
This means, we are free to create as many objects of the NotThatSimple type as we want. For example, the statement
NotThatSimple similarObject = new NotThatSimple(27);

creates another object that is similar to the notThatSimpleObject. It has the same behaviour, because it is also of the NotThatSimple type, but its state and identity are different.

Now, if we want to change the state of an object, we have to call its methods.

Invoking a method

An object's method is called by using a statement of the following form:
notThatSimpleObject.getValue();

To specify, which object's method you want to call, the name of that object must be written. After the object's name, the name of the method follows. To call the same method, but now for similarObject, you would have to write:
similarObject.getValue();

(Do you remember the term reusability, mentioned in the "Object Orientation" section?)

Our example class offers no way to change our objects' states, but we may get some information about the states. The getValue() method returns an integer value, which represents an object's current state.
Thus, if you want to get the state of similarObject, you might write something like:

```
int currentState;
currentState = similarObject.getValue();
```

Of course, the variable that receives the return value must be of the same type as the returned value.

After this short description of some Java basics, let's go on to the serious stuff. The next section shows you, how to work with the neural network classes.

Using the classes in your own programs

This section explains, how you can use the neural network classes in your own programs.

While designing the classes, I asked myself, what would be the easiest way to build a neural net? I remembered the neural net lessons, where my professor used to say things like:
"Now we want to build a 3-layered Backpropagation Net with 4 neurons in its input layer and 3 neurons in its output layer. The hidden layer consists of 2 neurons. Each neuron of one layer is connected to all neurons of the following layer..."

While he said this, he drew a sketch of the net on the blackboard in the following way:
1. He drew the neurons of the input layer
2. He drew the neurons of the hidden layer
3. He drew the neurons of the output layer
4. He connected the input neurons with the hidden neurons
5. He connected the hidden neurons with the output neurons
6. He assigned random values to the weights of both weight matrices
7. He wrote down the input and target patterns
8. He demonstrated a few learning cycles of the net

For this procedure was always the same, and only the net structure changed, it seemed to be a possible solution for an implementation. So I came to the conclusion, building a neural net in a program should be done the same way as you would describe it with your own words.

The whole class structure consists of 11 classes, as can be seen in Class Structure/The classes and their relationships, but you actually need to know more about only three of them to get your neural net running.
The classes you explicitely use in your programs are: BackpropagationNet, KohonenFeatureMap, and InputMatrix.

Using the BackpropagationNet class

Features of the BackpropagationNet class

Number of neuron layers	minimum: 2, maximum: N
Number of neurons in each layer	minimum: 1, maximum: N
Weight matrices	Automatically created and initialized. The weights are connecting the neurons of two consecutive layers
Bias values	Automatically used
Number of input/target patterns	minimum: 1, maximum: N
One learning step includes	Forward propagation and backpropagation of all input patterns (learning-by-epoch), changing of the weights

Steps to create a Backpropagation Net

1. Object declaration
BackpropagationNet bpn;
Required. Declares the object bpn to be of type BackpropagationNet.

2. Constructor call
bpn = new BackpropagationNet();
Required. Creates an instance of BackpropagationNet. This is the only constructor of the class and takes no arguments.

3. Read conversion file
bpn.readConversionFile("fileName");
Required. Reads the ASCII-binary conversion file. See The structure of a conversion file for further explanation of conversion files.

4. Create input layer
bpn.addNeuronLayer(i);
Required. Creates the net's input layer with i neurons. The neuron layers are sequentially added to the net structure!

Create hidden layer(s)
bpn.addNeuronLayer(h);
Creates the net's hidden layer(s) with h neurons. This step is optional (if your net should have no hidden layers) or may be done one or more times (if your net has one or more hidden layers). If your net has more than one hidden layers, the number of neurons in each layer must not always be the same.

5. Create output layer
bpn.addNeuronLayer(o);
Required. Creates the net's output layer with o neurons.

6. Connect all layers
bpn.connectLayers();
Required. Connects the neurons of all consecutive layers. All weight matrices are created and initialized with random values.

7. Read pattern file
bpn.readPatternFile("fileName");
Required. Reads the pattern file that contains the input and target patterns. See The structure of a pattern file below for further explanation of pattern files.

Set learning rate
bpn.setLearningRate(x);
Sets the net's learning rate to x.

Set minimum error
bpn.setMinimumError(x);
Sets the net's minimum error to x. The net learns, until its error is smaller than this value.

Set accuracy
bpn.setAccuracy(x);
Sets the net's accuracy value to x.

Set maximum learning cycles
bpn.setMaxLearningCycles(x);
Sets the maximum number of learning cycles to x. The default value is -1 (no maximum).

Reset learning time
bpn.resetTime();
Resets the learning time.

8. Perform a learning cycle
bpn.learn();
Required. Performs one learning cycle. This method is usually called within a loop, which exits, if the finishedLearning() method returns true.

Recall a pattern
bpn.recall("inputPattern");
Tries to recall the correct output of the input pattern inputPattern.

Information is available on:

Accuracy
bpn.getAccuracy();
Returns the accuracy value of the net.

Elapsed time
bpn.getElapsedTime();
Returns the time that elapsed since the learning process started.

Input pattern
bpn.getInputPattern(i);
Returns the input pattern with number i. Pattern numbers start with zero!

Learning cycle
bpn.getLearningCycle();
Returns the current learning cycle of the net.

Learning rate
bpn.getLearningRate();
Returns the learning rate of the net.

Minimum error
bpn.getMinimumError();
Returns the minimum error of the net.

Net error
bpn.getError();
Returns the current error of the net.

Neuron outputs
bpn.getNeuronOutputs(i);
Returns the output values of all neurons in layer i.

Number of layers
bpn.getNumberOfLayers();
Returns the number of neuron layers.

Number of neurons
bpn.getNumberOfNeurons(i);
Returns the number of neurons in layer i. Neuron layer numbers start with zero!

Number of patterns
bpn.getNumberOfPatterns();
Returns the number of input/target patterns.

Number of all weights
bpn.getNumberOfWeights();
Returns the number of all weights.

Number of weights
bpn.getNumberOfWeights(m);
Returns the number of weights in weight matrix m.

Output pattern
bpn.getOutputPattern(i);
Returns the output pattern with number i. Pattern numbers start with zero!

Pattern error
bpn.getPatternError(i);
Returns the error of the output pattern with number i. Pattern numbers start with zero!

Target pattern
bpn.getTargetPattern(i);
Returns the target pattern with number i. Pattern numbers start with zero!

Weight values
bpn.getWeightValues(m);
Returns all weight values of weight matrix m. Weight matrix numbers start with zero!

Here is the source code of the BPN Application to show you an example: BPN.java

The structure of a conversion file

The conversion file of a Backpropagation Net must be used to convert ASCII characters (the net input) to an internal binary representation. The number of binary values that represent the ASCII characters can be changed freely, but has to be the same for each character.

The general structure of a conversion file is as follows:

In the first line:
Number of conversions

Each following line:
- At first position: The ASCII character
- and then: The binary representation (the length of this representation doesn't matter, but must be the same for each conversion)

Below you see the conversion file ascii2bin.cnv that is used in the BPN application. It contains 64 conversions (as can be seen in the first line) and each ASCII character is converted to 6 binary digits.

64
0000000
1000001
2000010
3000011
4000100
5000101
6000110
7000111
8001000
9001001
a001010
b001011
c001100
d001101
e001110
f001111
g010000
h010001
i010010
j010011
k010100
l010101
m010110
n010111
o011000
p011001
q011010
r011011

s011100
t011101
u011110
v011111
w100000
x100001
y100010
z100011
A100100
B100101
C100110
D100111
E101000
F101001
G101010
H101011
I101100
J101101
K101110
L101111
M110000
N110001
O110010
P110011
Q110100
R110101
S110110
T110111
U111000
V111001
W111010
X111011
Y111100
Z111101
?111110
?111111

Note: A conversion file must be read **before** you add neuron layers to the net, because the number of binary digits must already be available when a neuron layer is created.

The structure of a pattern file

The pattern file of a Backpropagation Net contains the input and target patterns for the net. The patterns should contain ASCII characters that are defined in the net's conversion file. The number of patterns can be changed freely.
The length of an input pattern must be the same as the number of neurons in the net's input neuron layer. The length of a target pattern must be the same as the number of neurons in the net's output neuron layer.

The general structure of a pattern file is as follows:

In the first line:
Number of patterns

In the second line:
Length of input patterns (same as number of neurons in input layer)

In the third line:
Length of target patterns (same as number of neurons in output layer)

Each following line:
The input pattern [whitespace] the target pattern

Below you see the pattern file towns.pat that is used in the BPN application. It contains 15 patterns (first line). Each input pattern consists of 10 ASCII characters (second line) and each target pattern consists of 7 characters (third line).

```
15
10
7
Bonn000000 Germany
Brasilia00 Brasil0
Brussels00 Belgium
Helsinki00 Finland
London0000 England
Madrid0000 Spain00
Moscow0000 Russia0
New0Delhi0 India00
Oslo000000 Norway0
Paris00000 France0
Rome000000 Italy00
Stockholm0 Sweden0
Tokyo00000 Japan00
Vienna0000 Austria
Washington USA0000
```

Note: A pattern file must be read **after** the connectLayers() method had been called.

Using the KohonenFeatureMap class

Features of the KohonenFeatureMap class

Number of neuron layers	1 input layer, 1 feature map
Input dimension	minimum: 1, maximum: 3
Number of neurons in feature	minimum: 1*1, maximum: N*M

map	
Number of input values	minimum: 1, maximum: N
Weight matrices	Automatically created and initialized. One matrix connects input layer and feature map. The other matrix is not of the WeightMatrix type and connects the neurons of the feature map among themselves.
Biases	Not used
One learning step includes	Selection of a random input value, finding the most activated map neuron, changing of the weights

Steps to create a Kohonen Feature Map

1. Object declaration
KohonenFeatureMap kfm;
Required. Declares the object kfm to be of type KohonenFeatureMap.

2. Constructor call
kfm = new KohonenFeatureMap();
Required. Creates an instance of KohonenFeatureMap. This is the only constructor of the class and takes no arguments.

3. Create feature map
kfm.createMapLayer(xSize,ySize);
Required. Creates the net's feature map with xSize*ySize map neurons.

4. Define input matrix
see Using the InputMatrix class
Required. Creates the input matrix.

5. Connect input layer with feature map
kfm.connectLayers(im);
Required. Connects each input neuron (automatically created, depending on the dimension of the input matrix im) with each neuron of the feature map. Besides, all map neurons are connected among themselves. All weight matrices are created and initialized with random values, taken from the input matrix im (created in step 4).

Set initial learning rate
kfm.setInitLearningRate(x);
Sets the net's initial learning rate to x. The default value is 0.6.

Set initial activation area
kfm.setInitActivationArea(x);
Sets the net's initial activation area to x. The default value is the greater of both map sizes divided by 2.

Set final activation area
kfm.setStopArea(x);

Sets the net's final activation area to x. The default value is initActivationArea divided by 10.

Set maximum learning cycles
kfm.setMaxLearningCycles(x);
Sets the maximum number of learning cycles to x. The default value is -1 (no maximum).

Reset learning time
kfm.resetTime();
Resets the learning time.

6. Perform a learning cycle
kfm.learn();
Required. Performs one learning cycle. This method is usually called within a loop, which exits, if the finishedLearning() method returns true.

Information is available on:

Activation area
kfm.getActivationArea();
Returns the current activation area of the feature map.

Elapsed time
kfm.getElapsedTime();
Returns the time that elapsed since the learning process started.

Final activation area
kfm.getStopArea();
Returns the final activation area of the feature map.

Initial activation area
kfm.getInitActivationArea();
Returns the initial activation area of the feature map.

Initial learning rate
kfm.getInitLearningRate();
Returns the initial learning rate of the net.

Learning cycle
kfm.getLearningCycle();
Returns the current learning cycle of the net.

Learning rate
kfm.getLearningRate();
Returns the current learning rate of the net.

Map size in x
kfm.getMapSizeX();
Returns the size of the feature map in x-dimension.

Map size in y
kfm.getMapSizeY();
Returns the size of the feature map in y-dimension.

Number of weights
kfm.getNumberOfWeights();
Returns the number of weights in the weight matrix that connects the input neurons with the neurons of the feature map.

Weight values
kfm.getWeightValues();
Returns all weight values of the weight matrix that connects the input neurons with the neurons of the feature map.

Using the InputMatrix class

Features of the InputMatrix class

Number of input values	minimum: 1, maximum: N
Dimensions	minimum: 1, maximum: 3

Steps to create an Input Matrix

1. Object declaration
InputMatrix im;
Required. Declares the object im to be of type InputMatrix.

2. Constructor call
im = new InputMatrix(size,dim);
Required. Creates an instance of InputMatrix. This is the only constructor of the class and has the following arguments:
size: the number of input values (1...N)
dim: the dimension of this matrix (1...3)

3. Set values (version 1)
im.setInputX(x); (and)
im.setInputY(y); (and)
im.setInputZ(z);
Required. Sets input values for each dimension to this matrix. These methods can be used sequentially, if you want an input matrix with 1, 2, or 3 dimensions. The arguments x, y, and z are arrays of type int. To define a 3-dimensional input matrix, you may use the setInputValues(x,y,z) method.

3. Set values (version 2)
im.setInputValues(x,y,z);

Required. Sets input values for all three dimensions to this matrix. This method is used, when you want an input matrix with three dimensions. The arguments x, y, and z are arrays of type int.

Information is available on:

Number of input values
im.size();
Returns the number of input values in this input matrix.

Dimension
im.getDimension();
Returns the dimension of this input matrix.

Sample Applet

A 3D Kohonen Feature Map

Description

This applet demonstrates a Kohonen Feature Map neural net in three-dimensional space.

The task of this neural net is to span its map over all blue points (the input values of the net) in an even way and without crossing one point twice. This problem is similar to the classic Travelling Salesman Problem, where the shortest path between a certain number of cities is to be found without passing one city twice. Using conventional methods, like the backtracking algorithm, this problem can't be solved on any computer in time for a number of cities greater than about 25. The presented neural net is able to solve this problem for even 50 or more cities. And all that in three dimensions! The result may not definitely be the optimal solution but it's always a good approximation if all parameters have been properly set.

The structure of the feature map may be changed in the Parameters window. It can either be one-dimensional (displayed as a black line) or two-dimensional (displayed as a black grid). (see Sample Applet Controls for a detailed description of all changeable parameters)

If you want to know, how the learning algorithm of a Kohonen Feature Map works, take a closer look on the "Selforganization" section in neural net overview/The learning process.

See the applet at http://www.nnwj.de/sample-applet.html