University of Salzburg, Institute for Computer Science

PS Introduction to Bioinformatics
SS 1999

Applications of Hopfield Networks

Mayer August	ags@gmx.net
Wiesbauer Gerald	gerald.wiesbauer@sbg.ac.at
Spitzlinger Markus	mspitzl@cosy.sbg.ac.at
Leader:	
Dipl.Ing. Mag. Roland Schwaiger	rschwaiger@cosy.sbg.ac.at

Version from November 4, 1999

Abstract

This document gives an overview of the structure of Hopfield Networks (HN) and describes some applications in detail. After a short general description of neural networks, the hopfield networks and the available learning methods are described to some detail. A number of applications such as the TSP problem and pattern recognition and finally a programming project concerning the usage of HN in pattern recognition are discussed. The project includes some algorithms of image processing and feature space transformation to prepare the input picture for the network.

Contents

Chapter 1

Introduction

1.1 About Neural Networks

According to [Fau94] an artificial neural network is an information-processing system that has certain performance characteristics in common with biological neural networks. Artificial neural networks have been developed as generaliziations of mathematical models of human cognition or neural biology, based on the assumptions that:

1. Information processing occurs at many simple elements called *neurons, units, cells or nodes*.
2. *Signals* are passed between neurons over directed communication links.
3. Each connection link has an associated *weight*, which, in a typical neural network, the signal transmitted is multiplied by. The weights represent the information which is used by the network to solve the given problem.
4. For each neuron there is an activation function (which is usually nonlinear) which is applied to its network input (i.e. the sum of weighted input signals) to determine its output signal.

A neural network is characterized by the *architecture*, which is formed by the pattern of the connections between the neurons, the *training* or *learning algorithm*, which is the method of determining the weights of the connections, and the *activation function*.

Each neuron has an internal state, called its activation or activity level, which is a function of the inputs it has received. Typically, a neuron sends its activation as a signal to several other neurons. It is important to note that a neuron can send only one signal at a time, although that signal is broadcast to several other neurons.

1.1.1 A simple Adaptable Node

According to [Ale93] a simple *Adaptable Node* or *Toy Adaptive Nod (TAN)* can be seen as some sort of electronic component.

Figure 1.1 shows a TAN. The actual values of voltages or currents in this circuit aren't considered by convention, as well as it is assumed that the neurons can have exactly two states: 0 and 1.

There are N inputs labelled X_1, X_2, ..., X_N, and one output F. The actual 0 or 1 value present at the input will often be referred to as the *input bit pattern* since they can be thought of as binary digits or bits, as in conventional computing. The inputs of one node are connected either to the outputs of other nodes or to some external source. The function F of a node is a list (or boolean table) which shows what the output signal F will be for each of the possible patterns.

For example, the function of a 3-input node can have the following boolean table:

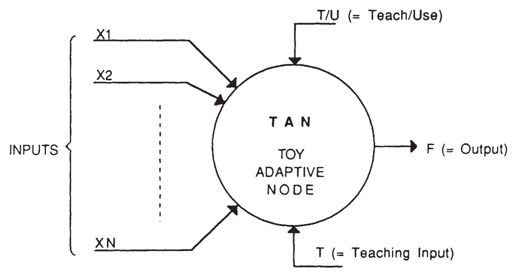

Figure 1.1: The Toy Adaptive Node (TAN).

X_1	X_2	X_3	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

The table is a simple means of showing that:

- when $X_1 = 0$, $X_2 = 0$ and $X_3 = 0$, then $F = 0$
- when $X_1 = 0$, $X_2 = 0$ and $X_3 = 1$, then $F = 1$

et cetera.

The node may operate in two modes: it can be *taught* (during that time its function is possibly changed) or *used* (then the function doesn't change). The mode which the TAN is currently operating in is determined by the *T/U signal*, which is set to 1 for teaching mode or to 0 for usage mode. While teaching, the node associates whatever is present (i. e. 0 or 1) at the teaching input T with whatever pattern is present at the terminals X_1, X_2, ... This means that whenever that particular input pattern occurs again in a subsequent using phase, the learnt or associated output symbol (i. e. 0 or 1) is output at F. There is also an undefined output, 0 or 1, chosen at random which exists either if the particular input has not been taught at all, or if there is some sort of confusion about the desired output for a particular input. For example, if the node is taught to output a 1 in response to a particlular input pattern and is then taught to output a 0 in response to the same pattern the output will be undefined.

1.1.2 Network Stuctures

There are two types of networks: *feed-forward* (or *associative*) networks and *feedback* networks.

A feed-forward network is one in which there are no closed loops. A single-layer feed-forward net is a net where every output node is connected to every input terminal. In other network types, there are nodes which are not directly connected to the output; they are called *hidden nodes*. They can cause problems during training because the training information, consisting only of bit patterns of inputs and their corresponding outputs, does not contain a specification of what should be fed to the teaching inputs of these hidden units. And to achive appropriate links between input and output, they cannot act at random.

A feed-back net is one where information can travel back from the output to the input. There is one special type of feed-back nets called *autoassociative* because they are trained using a selected pattern on internal connections as data both for inputs and desired outputs of the nodes. Instead of associating a pattern at an overall input with a pattern at the overerall output (as occurs in the other nets) the net associates the training pattern with itself.

1.2 History of Neural Networks

Accoring to [Ale93] the beginning of neural computing was in 1936, when Alan Turing, a British mathematician laid down the principles for defining what is and what is not computable. He also wrote about the way in which the brain did its computations through cycles of activity in neural nets. Another mathmatician, John von Neumann, also talked about neural nets, seeing them as providing a way of doing computations that emerges from the structure of the net itself.

Norbert Wiener, whose name is associated with the definition of cybernetics, thought that the fundamental structure to a mathematical understanding of computation in both brains and machines is the interplay of logic and network structure. But the algorithmic method has always been seen as the well-behaved, predictable kind of computing and so neural networks laid largely dormant for a long time.

Years of definition (1943 - 1969) followed, where a model of the fundamental cell of the living brain, the neuron, was suggested from the neurophysiologist Warren Mc Culloch and the logician Walter Pitts. Another important idea was put forward in 1949 by Donald Hebb, who suggested that a group of neurons could reverberate in different patterns, each of these patterns relating to the recall of a differnet experience. This was the first time a model of dynamic memory had been used by a neuropsychologist.

With this and a few other theories the capabilities of the nets where understood and they could be simulated on contemporary computers. This empirical approach was severely challenged in 1969 by Marvin Minsky and Seymour Papert. They wrote that neural nets are able to learn to perform tasks that requires internal represantations not explicity specified by the training data. The net must somehow work out these representation for itself and this training turns out to be a difficult task and so it is very much part of current debate and most modern learning algorithmus and net structures are being designed with the possibility of solving this difficulty in mind.

Today most nets are studied as programs running on serial computers, but the future turns upon the ability of engineers to make physically parallel systems. There is a little doubt that John Hopfield of the California Institute of Technoloy was responsible for the breakthrough of neurals nets, because his orientation was very practical. He thought that neural nets are a good way of making advanced computer memories.

Neurals nets are used in a wide range of applications. One of the most successful and proven demonstrations of the ability of neural nets is to take symbolic representation of words as letters and generate phoneme (sound segment) code that may be used to drive a voice synthesizer. The opposite problem is the speech recognition and another thing is that a net could be made to follow conversations aubout simple scences and paraphrase simple stories. But probably the major application of neural nets lies in the recognition of visual patterns.

The future of neural computing are hybrid systems, which find out from the communication between the user and the machine what the user wants.

1.3 About Hopfield Networks

In short[1], the main topic in Hopfield networks is that they are recurrent neural networks, where the term ”recurrent” means that the output values are feeded back to the input of the network in an undirected way. Hopfield Networks are mainly used in pattern recognition and optimization. Learning is performed by modifying the strength of the connections between the neurons and the parameters of the activation function (*thresholds*). The network structure is built in a way that the outputs of each neuron are connected to the input of every other neuron.

1.3.1 Construction

A Hopfield network consists of binary neurons which are connected by a symmetric network structure. Binary means that the neurons can be active (”firing”, state 1) or inactive (”not firing”, 0). The connections are weighted, and depending on the sign of the weight they can be intercepting or activating; e.g. a firing neuron activates all neurons which are connected to it with a positive weight. There is a threshold value for every neuron which the sum of the input values must reach to produce activity.

[1] see [Bäu97] and [DN96, pp.137ff]

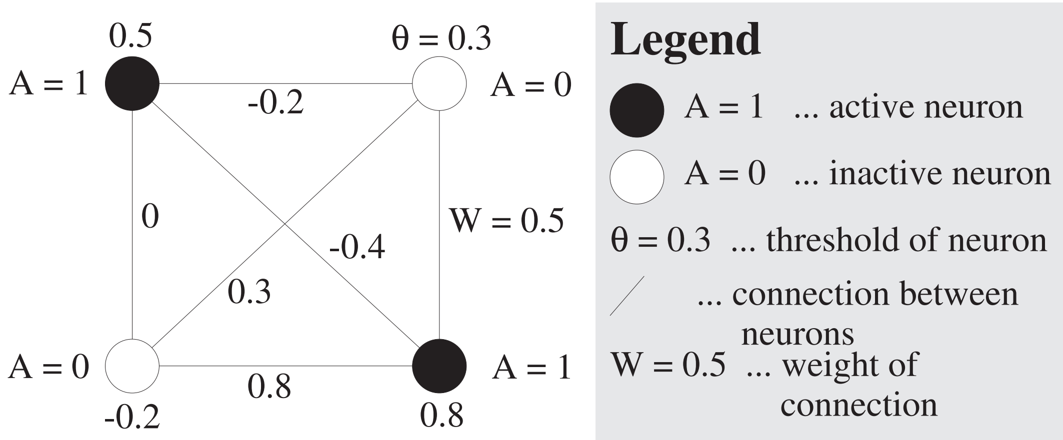

Figure 1.2: A simple example of a Hopfield network.

At the beginning of the calculation of the network output, the neuron's activation corresponds to the pattern to recognize. Then the network is iterated, which means that the state of the neurons is recalculated until the network is stable, i.e. the network state doesn't change any more. This is possible in a finite amount of time and iterations for Hopfield networks[2]. This can also be seen as the minimization of the energy in the net, so that the final state is a minimum of an energy function called *attractor*.

1.3.2 Formalism

As described in [DN96], a Hopfield Network $HN = (U, W, A, O, NET, ex)$ is a Neural Network, where

- the *weight matrix* (*network structure*) $W : U \times U \to \mathbf{R}$ is symmetric and doesn't connect any neuron to itself, i.e. $W(u, u') = W(u', u)$ and $W(u, u) = 0$,
- there is an activation function $A_u \in A$ for every neuron (unit) $u \in U$ which gives each neuron an activation state a_u. This is $+1$ if $net_u \geq \theta_u$ and -1 otherwise; the threshold $\theta_u \in \mathbf{R}$ is defined statically for every neuron;
- there is an output function $O_u \in O$ for every neuron $u \in U$, which is the identity function.
- there is a network input function (propagation function) $NET_u \in NET$ which maps the network input net_u to every neuron; this function is calculated by multiplying the outputs of all the other neurons with the weights of the connections to that neuron and then summarizing them up, i.e.

  $$net_u = \sum_{u' \in U} W(u', u) \cdot o'_u$$

  (the weight of the connection from a neuron to itself is always 0 in a HN)
- there is a function $ex(u)$ which specifies the external input value for every neuron, i.e. $ex : U \to \{-1, +1\}$; sometimes also the value set $ex : U \to \{0, 1\}$ is used, which results in some formula changes (set $\tilde{a}_u := 2a_u - 1$ and $a_u := \frac{1}{2}\tilde{a}_u + \frac{1}{2}$ to convert between the two systems).

The inventor of the HN, John Hopfield, thought more in terms of an energy-based analysis of the magnetic anomalies of the so-called spin-glasses. He calculates the energy of a neuron u as follows:

$$E_u = -a_u \cdot \left(\left(\sum_{u' \in U} W(u', u) \cdot a_{u'} \right) - \theta_u \right) = -a_u \cdot \text{(activation)}$$

[2]See [Hof93, p95].

The energy of the whole system is then calculated as follows (note the factor $\frac{1}{2}$ because of the symmetry of the net):

$$E = -a_u \cdot \sum_{u \in U} \left(\left(\frac{1}{2} \cdot \sum_{u' \in U} W(u', u) \cdot a_{u'} \right) - \theta_u \right)$$

A change in the network activation always results in an energy decrease, because the activation change Δa_u always has the same sign as the network input minus the threshold $net_u - \theta_u$, so that ΔE_u is always negative or zero.

If all neurons of a HN are updated at the same time, there can be cyclic state changes. An alternative is to update the neurons in a defined or random order. Then it can be proven (see [DN96, p142]) that a final state is reached after at most $n \cdot 2^n$ steps.

According to [Hof93, p95], the maximum number of patterns a HN is able to memorize is

$$p_{\max} = 0,138 \cdot N,$$

where N is the number of neurons in the net. For big N, the formula

$$p_{\max} = \frac{N}{2 \cdot \ln N}$$

is valid.

A Hopfield Network also has local minima other than the learned patterns. A common example is the exact inverse pattern, but there are also other *spurios states*. However, the attractivity of those local minima is usually rather small compared to the learned patterns if the capacity of the net has not been exceeded.

1.3.3 Example for using a trained net

As previously described, the network detects a pattern which is mapped as the initial activity states of the neurons step-by-step in a finite amount of time. Here is an example how this works.

First, a neuron is selected, either using a fixed order or a random permutation of the neurons. Then the outputs of all the other neurons are multiplied by the weight of the connections to the active neuron, and these products are summed up. If the sum exceeds the threshold, the new state of the neuron is active, else it is inactive.

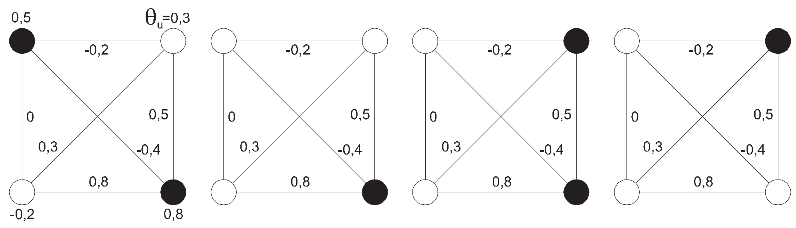

Figure 1.3: First pattern recognition steps. (Described in Section 1.3.3)

As shown in figure 1.3, the upper left neuron is selected and its state is recalculated (filled circles mean active neurons, empty inactive ones). So the neuron gets the inputs $0 \cdot -0.2$ (from the upper right neuron), $1 \cdot -0.4$ (lower right neuron) and $0 \cdot 0$ (lower left neuron), which sums up to -0.4. This is smaller than the threshold $\theta_u = 0.5$, so the activation of the upper left neuron switches to 0. The threshold values stay the same through this whole example and are only shown on the first image above or below the respective neuron.

The next neuron is the upper right neuron, and its input $-0.2 \cdot 0 + 0.3 \cdot 0 + 0.5 \cdot 1 = 0.5$ is greater than the threshold value $\theta_u = 0.3$, which results in the neuron being turned active. After that, the lower right neuron is calculated, and it stays deactivated, and the lower left neuron's activation state becomes 1 because it gets enough input (can be seen on the next figure 1.4).

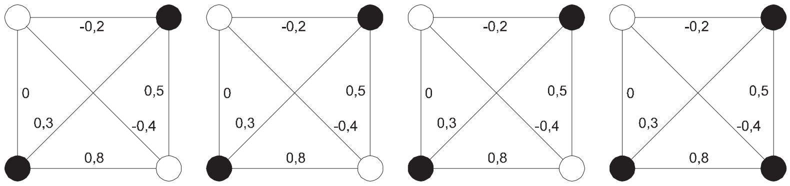

Figure 1.4: Next pattern recognition steps, as described in section 1.3.3

Remind that the state of the neurons is calculated one after the other, and the activation values are *not* taken from the network state at the beginning of the iteration, but the actual states are used. Otherwise, the recalculation would be parallel for all neurons, and then the network needn't reach a stable state.

The procedure is continued until a full iteration is done without changing a neuron, which means that a stable state has been reached. The last state of Fig.1.4 is such a state. Note that it is possible to recalculate any neuron without its state getting changed. The resulting states of the neurons form the recognized pattern.

1.3.4 Learning in Hopfield Networks

Neural networks, in particular HNs, can learn their stable states from examples (self-learning network)[3]. The network performs learning by feeding a desired state to the network; if this state isn't stable, the values of the thresholds and inter-neuron connection strengths (weights) are changed until the desired stability is reached. Rules which can be used for this are *Hebb's learning rule* (1949) and the *Widrow-Hoff* or *Delta learning rule* (1960).

Hebb's rule roughly states that the connection strength between two neurons changes proportionally to the frequency of the neurons firing together. Training is performed[4] by loading the desired n-dimensional stabile state vectors $x_1, x_2, \ldots$ into the network and then recalculating the weights (which are initially zero) according to the formula

$$w_{ij} \to w_{ij} + x_{ki} \cdot x_{kj}, \qquad i, j = 1, \ldots, n \text{ and } i \neq j,$$

where x_{ki} is the ith component of the vector x_k. Because the diagonal elements of the matrix have to be zero in a HN, the identity matrix I is subtracted from the result of the above (as the state vector $-1, +1$ is used, a multiplication of an element with itself always results to 1), and the final formula is

$$W_i = x_i^T x_i - I.$$

The best results with the Hebbian learning rule are reached if any two input vectors x_i and x_j are nearly orthogonal.

The Hebb rule is a very simple learning rule, and if the vectors $x_1, x_2, \ldots$ are rather near to each other, it is possible that some vectors are not really learnt. But for a HN, the Perceptron learning rules can also be used[5], which often leads to better results.

At every iteration of a Perceptron learning algorithm, the activations of the neurons are set to the desired stable state, and if the activation of the single considered neuron is wrong, the weights and threshold of the neuron are changed, i.e. the Delta-Rule or the Perceptron learning algorithm is locally used. The threshold of a neuron is treated like an additional weight, but its value is inverted.

The Perceptron learning algorithm[6] works by testing the resulting activation of a given input vector. If the activation is wrong, the input vector is added to the weight vector for positive activation and subtracted for negative activation.

[3] See also [Ale93, pp.105–109] and [Olm98].
[4] See [Roj93, pp.296f].
[5] see [Roj93, pp.298ff]
[6] see [Roj93, p.83].

The Delta learning rule tries to speed up convergence. Here, the input vectors are normalized. On a falsely classified input vector $x \in P$ (i.e. a vector which should result in a positive activation), the errorδ is calculated by

$$\delta = -w_t \cdot x,$$

and the new weight vector w_{t+1} is

$$w_{t+1} = w_t + (\delta + \epsilon)x,$$

where ϵ is a very small positive value. After that, x is classified right[7]. The value ϵ guarantees that the new weight vector descends over the edge of the error function to a lower level; if it is too big, the weight vector could be moved too far and the resulting error could be even bigger. If the falsely classified input vector x should result in a negative activation (i.e.$x \in N$), then ϵ is subtracted from δ:

$$w_{t+1} = w_t + (\delta - \epsilon)x.$$

By adding a constant γ, which is called the *learning factor*, and using unnormalized vectors, we have the method of Aleksander and Morton or the *Delta learning rule* proper:

$$w_{t+1} = w_t + \gamma \cdot (\delta + \epsilon)x.$$

So if an unit has to fire, but its activation is negative, the Widrow-Hoff rule causes it to be made positive, which decreases the total energy. If a unit is firing and should not, the activation is made negative along with the value of A_u, which again leads to a reduction of energy[8].

The effect of the Widrow-Hoff learning rule is that the energy of the appropriate state is lowered in comparison with its 'neighbouring' states (i.e. states differing by only one node).

It is theoretically possible to understand the network by making a step-for-step calculation and evaluation of the network states and the resulting error.

Another aspect is that such networks can be considered as inequalities, and training the former provides an automatic way of solving them, which would be relatively time-consuming using standard methods.

1.3.5 The training algorithms

This section describes two ways to train the network for defined patterns:

- calculating the values of thresholds and strengths by hand — *handcrafted values*
- training the network with examples

1.3.6 Network training — The calculating method

In this example, we use a HN with 4 nodes. The desired stable states are:

$a_1 = 1, a_2 = 0, a_3 = 1, a_4 = 0$ and
$a_1 = 1, a_2 = 0, a_3 = 0, a_4 = 1,$

where a_u is the activation of the node, $W_{uu'}$ is the connection weight between two nodes and θ_u is the threshold of a given neuron u. The value set used is $ex : U \to \{0, +1\}$.

Primarily, the first desired stable state is trained (1, 0, 1, 0). So there should be

$W_{12} \cdot a_2 + W_{13} \cdot a_3 + W_{14} \cdot a_4 - \theta_1 \geq 0$
$W_{21} \cdot a_1 + W_{23} \cdot a_3 + W_{24} \cdot a_4 - \theta_2 \leq 0$

[7] see [Roj93, p.89].
[8] See [Ale93, p.109]

$W_{31} \cdot a_1 + W_{32} \cdot a_2 + W_{34} \cdot a_4 - \theta_3 \geq 0$
$W_{41} \cdot a_1 + W_{42} \cdot a_2 + W_{43} \cdot a_3 - \theta_4 \leq 0$

This formula can be simplified as some values of a_u are zero. The following inequality system remains:

$a_1 : W_{13} - \theta_1 \geq 0$
$a_2 : W_{21} + W_{23} - \theta_2 \leq 0$
$a_3 : W_{31} - \theta_3 \geq 0$
$a_4 : W_{41} + W_{43} - \theta_4 \leq 0$

Secondarily, the second state should also be trained (1,0,0,1). Thus,

$a_1 : W_{12} \cdot a_2 + W_{13} \cdot a_3 + W_{14} \cdot a_4 - \theta_1 \geq 0$
$a_2 : W_{21} \cdot a_1 + W_{23} \cdot a_3 + W_{24} \cdot a_4 - \theta_2 \leq 0$
$a_3 : W_{31} \cdot a_1 + W_{32} \cdot a_2 + W_{34} \cdot a_4 - \theta_3 \leq 0$
$a_4 : W_{41} \cdot a_1 + W_{42} \cdot a_2 + W_{43} \cdot a_3 - \theta_4 \geq 0$

Again, the inequalities can be simplified:

$a_1 : W_{13} - \theta_1 \geq 0$
$a_2 : W_{21} + W_{24} - \theta_2 \leq 0$
$a_3 : W_{31} + W_{34} - \theta_3 \leq 0$
$a_4 : W_{41} - \theta_4 \geq 0$

The next step is to find values for the weights and thresholds to make the systems above valid. A set which does this is as following:

$$\begin{array}{rclrcl} & & & W_{12} & = & 0.3 \\ \theta_1 & = & -0.2 & W_{31} & = & 0.3 \\ \theta_2 & = & 0.6 & W_{41} & = & 0.4 \\ \theta_3 & = & -0.2 & W_{43} & = & -0.6 \\ \theta_4 & = & 0.3 & W_{23} & = & 0.1 \\ & & & W_{24} & = & 0.1 \end{array}$$

We can test the set by inserting them into the inequality systems above:

$$\left.\begin{array}{lrrrl} a_1: & 0.3 & & +0.2 & \geq 0 \\ a_2: & 0.3 & +0.1 & -0.6 & \leq 0 \\ a_3: & 0.3 & & +0.2 & \geq 0 \\ a_4: & 0.4 & -0.6 & -0.3 & \leq 0 \end{array}\right. \quad \text{ok.}$$

$$\left.\begin{array}{lrrrl} a_1: & 0.3 & & +0.2 & \geq 0 \\ a_2: & 0.3 & +0.1 & -0.6 & \leq 0 \\ a_3: & 0.3 & -0.6 & +0.2 & \leq 0 \\ a_4: & 0.4 & & -0.3 & \geq 0 \end{array}\right. \quad \text{ok.}$$

1.3.7 Network training — Training by giving examples

This method uses the Widrow-Hoff rule and starts as follows: The strengths and thresholds are equal in each node and set to a default value, (i.e 0.5 for the strengths and 0.0 for the thresholds)

The next step is to apply these values to the formulas, according to the example described at the calculating method. 2 of 4 nodes are enough to show the algorithm, the whole calculation is too long to be described here.

$a_1 : W_{13} - \theta_1 \geq 0$
$a_2 : W_{21} + W_{23} - \theta_2 \leq 0$

this means:

$a_1 : 0.5 - 0.1 = 0.4$
$a_2 : 0.5 + 0.5 - 0.0 = 1.0$

These variables (0.4, 1) are the condition values for firing, but not the desired ones (0.1, -0.2). The difference between these two tuples is the error-signal of the Widrow-Hoff rule, keep these desired values as small as possible.

$a_1 : 0.5 - 0.0 = 0.5$, desired $= 0.1, e = 0.4$

$a_2 : 0.5 + 0.5 - 0.0 = 1.0$, desired $= -0.2, e = 1.2$

The recalculation of the 'active' nodes is as follows:

$a_1 : 0.3 - 0.2 = 0.1 \geq 0$
$a_2 : 0.1 + 0.1 - 0.4 = -0.2 \leq 0$

The algorithm to perform this recalculation is simple. Divide the error-signal by the number of values at the right side and substract each value on this side by the 'divided error signal'.

This recalculation has to be applied to each node until all error-signals are zero.

1.4 Hopfield Nets and the Boltzmann Machine

Hopfield nets suffer from a tendency to stabilize to a local rather than a global minimum of the energy function. This problem is largely solved by a class of networks known as *Boltzmann machines*, in which the neurons change state in a statistical rather than a deterministic fashion. There is a close analogy between these methods and the way in which a metal is annealed; hence, the methods are often called *simulated annealing*.

Chapter 2

Applications of Hopfield Networks

2.1 The Travelling Salesperson Problem (TSP)

2.1.1 About the Travelling Salesman Problem

According to [Rob95] and [Hio96] the Travelling Salesman Problem is probably the most well-known member of the wider field of combinatorial optimization (CO) problems. The problem is that a salesperson wants to travel through N cities and he wants to know the shortest way through those cities.

These are difficult optimization problems where the set of feasible solutions (trial solutions which satisfy the constraints of the problem but are not necessarily optimal) is a finite, though usually very large set. The number of feasible solutions grows as some combinatoric factor such as , where N characterizes the size of the problem.

Example:

You are a salesman and you must visit 20 cities spread across North America. Yo must visit each city once and only once. The question is this: In what order should you visit them to minimize the total distance that you have to travel?

The answer is that there is no simple answer. Reasonable people will make a reasonable choice and accept a reasonably short path.

However there is only one way to find the absolute shortest path, and that is to write down every possible ordering of cities, compute the distance for each of those orderings, and pick the shortest one.

How many orderings are there? They can be counted this way:

For the first city to visit you have 20 choices.

For the second city to visit you then have only 19 choices (because you can't visit the first city again).

For the third city you have 18 choices, and so on . . .

For the last city you have 1 choice.

These numbers must be multiplied together to give the total number of orderings:

$20 * 19 * 18 * ... = 20! = 2,432,902,008,176,640,000$ possible orderings.

This number is so big that if your computer could check 1 million orderings every second it would still take 77,000 years to check them all! Thus even though we know how to solve the Travelling Salesman Problem we still can't do it.

2.1.2 Approximation of TSP with Hopfield Networks

Here are a few adresses of WWW-pages where to find programs which simulate the TSP.

A JAVA-Program:

http://www.essex.ac.uk

A C-File:

http://chopin.bme.ohio-state.edu/holgi/projects/C/C.html

Implementation of a Hopfield network to solve the TSP Problem for 10 cities and 20 different input patterns. The network is designed after the model proposed by Hopfield in his paper "Neural Computation of Decisions in Optimization Problems", in "Biological Cybernetics, 1985" using 100 analog units which are arranged in a 10 times 10 grid.

Others:

http://www.CSI.utoronto.ca/~mackay/itprnn

2.2 Pattern Recognition

2.2.1 About Pattern Recognition

According to [Fau94] many interesting problems fall into the general area of pattern recognition. One specific area in which many neural network applications are used is the automatic recgonition of handwritten characters (digits or letters). Humans can perform relatively easily the large information of variation in sizes, positions and styles of writing but it is very difficult problem for traditional techniques.

General-purpose multilayer neural nets, such as the backprogagation net (a multilayer net trained by backprogation) have been used for handwritten zip codes. Even when an application is based on a standard training algorithm, it is quite common to customize the architecture to improve the performance of the application. This backprogation net has several hidden layers, but the pattern of connections form one layer to the next is quite localized.

An alternative approach to the problem of recognizing handwritten characters is the neocognitron. This net has several layers, each with a highly structured pattern of connections form the previous layer and to the subsequent layer. However, its training is a layer-by-layer process, specialized for such an application.

2.2.2 Pattern Recognition with Hopfield Networks

According to [Tve98] there is a program to make Hopfield networks and a recall program that completes an incomplete pattern using the Hopfield, Boltzman and interactive activation network methods.

The Software may be found at: http://www.dontveter.com/nnsoft/nnsoft.html

There are two programs included in this package, hop.cpp and recall.cpp and the greatest number of units you need is only 25.

The purpose of hop.cpp is to take a set of patterns in and produce the weight matrix you get from the simple weight making algorithm in the text, the same algorithm used by Hopfield. The sbp program in the backprop package can also be used to produce a weight matrix that has symmetric weights between the units, simply save the weights to the file weights. The recall program implements the Hopfield and Boltzman machine pattern completion algorithms and an interactive activation algorithm as well. The interactive activation method is said to be better at avoiding local minima although with it you do not get a guarantee of finding any minimum at all.

2.3 Associative memory

Associative memory has to have the capability to "remember" patterns. For example, a human immediately remembers a chain of properties and events related to a person if he only hears his/her name. Also, if somebody sees a film on TV which he has seen in the cinema, even many years ago, he immediately recalls this.

On a computer level, Autoassociative memory can recognize an even partially obliterated letter (with high noise level), for example a 'T', as in picture 2.3.

Figure 2.1: Example of image reconstruction by associative memory

The method of recognizing is as follows: There is always some sort of 'underlying' information in a picture which lies in the ways that picture elements are ordered. This information is stored in the memory, and when a part of the picture is presented to the network, it may be able to 'associate' and restore the whole picture (*autoassociative memory*)

Another example of an associative memory is a libary system; if somebody wants to find a book of which he only knows a part of the title, associative memory can be used to find the rest of it.

Associative memory is increasingly used in search engines; possibly the one of the web index 'Altavista' uses it too.

Chapter 3

Programming project: Object recognition with Hopfield Networks

The goal of this programming project is to create a working object recognition program using Hopfield Networks. The code will be written in the C programming language, and it should be compact and fast enough to be executed on a microcontroller such as the Siemens C166. Along with the main part, the network, also some "glue" will have to be created, so that it is possible to feed a picture into the final program and get back signals for the location and identity of the recognized objects in this picture.

3.1 Parts and limitations

Three main parts of the algorithm are evident:

1. Object Separation. The Hopfield Network concept doesn't seem capable of separating objects in a picture (which is not proven), so more widely used approaches will be used. To recognise dependent picture elements it is necassary to split the picture into parts containing one or more objects. The objects are separated in a way that if objects overlap each other they are cut out together. This means that if there is the ball hiding partly behind the opponent, the final separation rectangle contains both the opponent and the ball. This greatly simplifies the separation process, but implicates that the final recognition process has more final states.
2. Transformation into Feature Space. After separation, the piece of the picture containg the object or objects is transformed by a Fourier Transformation (FFT or DFT). The image data is transformed into a single wave by using the "PET Method".
3. Object Recognition. The Hopfield Network is able to recognize a single transformed object as well as any combination of them.

The objects which should be recognized, also under suboptimal environment conditions (bad light, bad perspective, one object partially hiding the other), are a sperical form, a vertically striped cylinder and a grey rectangular shape. This is because these forms are used in a robot soccer playing arena, where the network should reliably recognize ball, opponent and goal.

3.2 Object Separation method

1. **Finding edges**

   To separate one or a group of objects a sobel edge detection algorithm is used, because it provides good performance and is relatively insensitive to noise. Other filters like a laplace edge detector algorithm are not so insensitive to noise and the edges are not detected very well. Smoothing noises in homogeneous image with e.g. a median filter is not nesassary because the sobel filter is insensitive enough.

A few example outputs after detecting the edges:

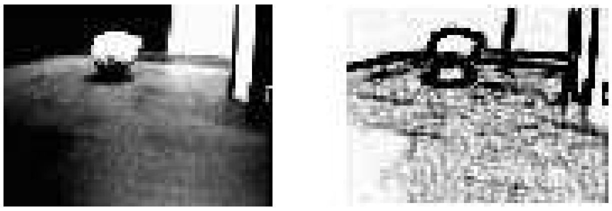

Figure 3.1: Original image, and image after detecting the edges

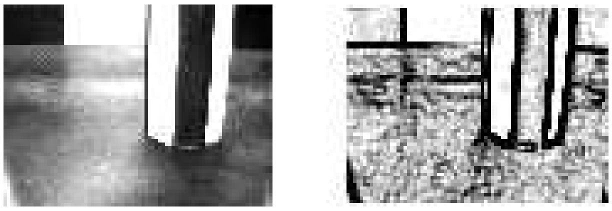

Figure 3.2: Original image, and image after detecting the edges

2. **Thinning**

   When the edges where found a one pass thinning algorithm is used because after this algorithm the objects can be separated easier. (Therefore the colour depth of the picture has to be decreased to a black and white picture.)

   A few example outputs after thinning:

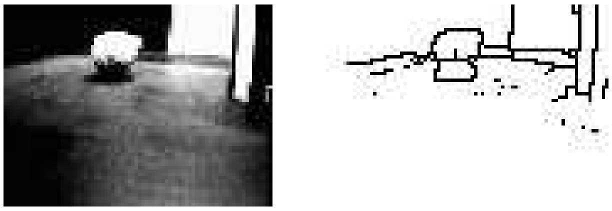

Figure 3.3: Original image, and image after thinning

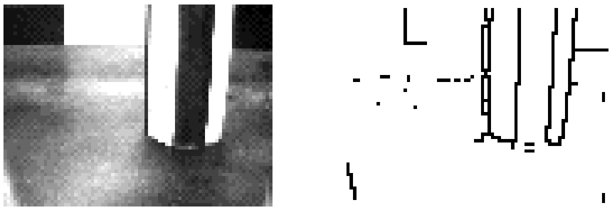

Figure 3.4: Original image, and image after thinning

3. **Object Separation**

   The picture including only thin edges is used the separate the different objects. The algorithm searches a point of the edge and follows this edge unitl no more participiant points where found. If this object is smaller than a definited size it is distroyed, otherwise it is returned. But there is the problem that objects can interleave and this is solved by an other procedure which fuses the interleaving objects.

   A few example outputs after thinning:

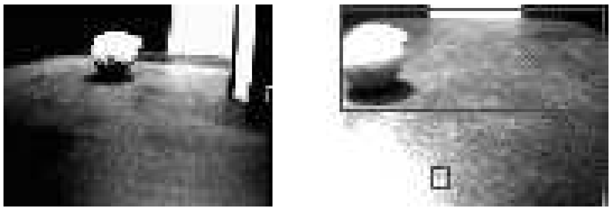

Figure 3.5: Original image, and image after objekt seperation

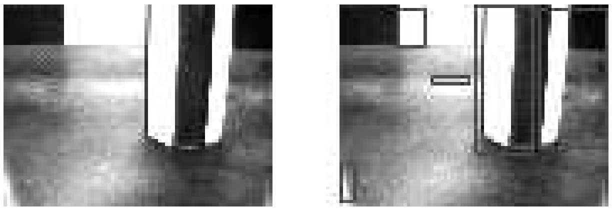

Figure 3.6: Original image, and image after objekt seperation

3.3 Transformation into feature space

Description of the algorithm that performs the transformation:

The Problem:

There is a picture, or slices of a picture with a maximum resolution of 80*60, and a Hopfield-Netowrok that contains 256 neurons, more neurons could be a performance problem. A possible solution of this problem is to “compress“ this image, so that the new resolution fits the resolution of the HFN (80*60) → (16*16) But a major problem of this algorithm is the compressing, what and how should be compressed. The “information“ (in EMMAS case: ball, goal, opponent) of the image must not be disturbed.

Porbably this algorithm is the appropriate one.

There are 3 major steps:

- **The “tomograph“**

  A slice of the picture be placed in the middle of the round thomograph-window, the procedure is equal to a medical tomograph. But as distinct form a medical thomograph the algorithm uses only one beam (normaly a thomograph uses more then 20 X-Rays). The strength of the beam at the reciever is the ”compressed” image. As the result of the ”compressing” the ”thomograph” retuns the beam strength at each degree of it’s round window.

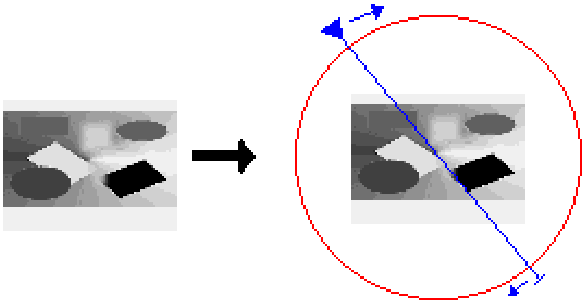

Figure 3.7: “Tomograph output“

The result is a sin-like wave (distortion of a sin wave). Now this wave has to be normalized, before it could be fed into the FFT/DFT filter Normalising means discard the “DC-part“ of the signal and fit the signal magnetude to a value of 1.

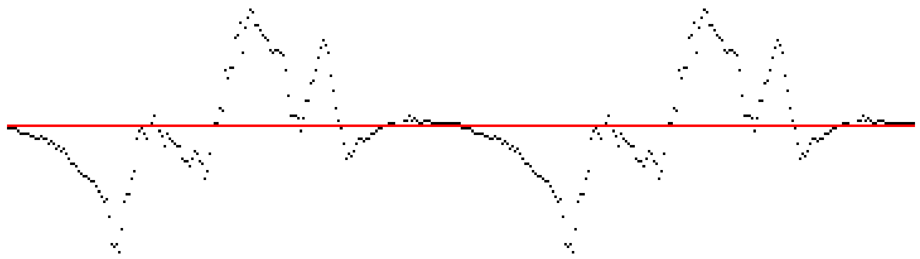

Figure 3.8: Normalized output

- **The FFT/DFT Filter**

  Transforms the “signal“ of the “thomograph“ into the feature space. All odd frequences are zero caused by the signal itselfs (symetric left-right). So it is enough to porcess the even frequences. The normalising is also nessercery in the feature-space, but as distinct form a not feature-space-signal it is enough to fit the maximum value to 1.

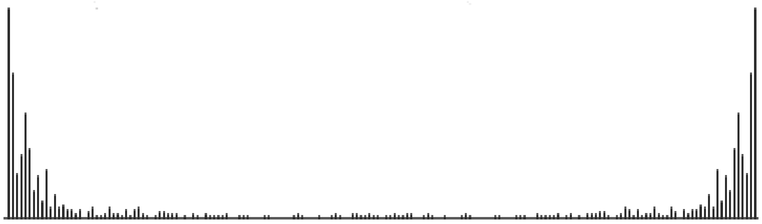

Figure 3.9: Feature space

- **Feature-space to ANN pattern**

  The last step is the transfer of the frequences of the feature-space into a neuronal network pattern. The first 16 even frequences were transformed into a VU-meter (like the one in your stereo) like matrix with a power 2 scale factor. Now like the LEDs in a VU-meter (light/no light) the pattern for the network is generated

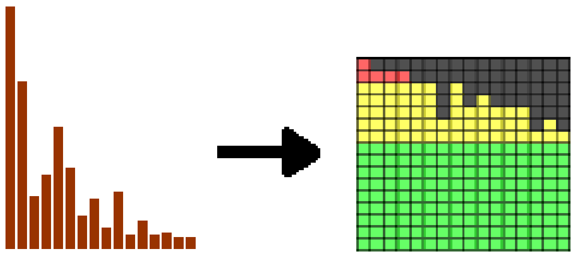

Figure 3.10: Pattern for the network

3.4 The hopfield network

The Hopfield network used is represented as a square matrix of neurons of size 16 by 16, which results in a size of 256 neurons and 32,640 weights. Since the weights are symmetric and null for the connections from a neuron to itself, the size of the weight matrix can be diminiushed to $\big(N \cdot (N-1)\big)/2$ values, where N is the size of the network and number of neurons. For example, in a 4×4 Matrix, there are 16 neurons, and $\big(16 \cdot (16-1)\big)/2 = 120$ distinct weights. The position of a weight in this matrix can be calculated as following: Be m_1, m_2 the numbers of the neurons the weight is between (starting from 1), and $m_1 < m_2$, i.e. m_1 is the smaller element. Then the index is

$$(m_1 - 1) \cdot N - \frac{(m_1 - 1) \cdot m_1}{2} + m_2 - m_1 - 1.$$

Additionally, there needn't be a fixed memory location for storing the activation matrix, since the activation is specific to the proper task the routines are actually doing.

For bigger networks, the size of the weight matrix is easily bigger than 64Kbytes, as there are double values used for precision (perhaps a reasonable result could also be achieved by using only float values), and this is an upper limit for 16-bit DOS systems (and presumably also C166 ones). So the weight matrix is split up into a number of smaller matrices, and the previous linear form is obtained by a macro which calculates the real index out of the linear one. But for this, memory has to be allocated, which is performed in the init_net function. Also, the thresholds are set to 0 here, and the weights to 0.5. Another operation is testing the weight matrix if it really works (there seem to be problems with Borland C).

Teaching the patterns to the HN is done using the Widrow-Hoff learning rule in a way that, for a given pattern and for every neuron, the activation is calculated. The error signal is then the difference to the correct activation value, divided by the number of contributors to the activation plus 1. Then, the error signal is added to the threshold of the neuron and subtracted from all the weights which contributed to the error. Finally, the network usually is normalized, which means that the weights are trimmed to sum up to exactly 1. However, the normalization is disabled, because the weights become very small this way. This procedure is repeated several times until the network doesn't change any more (stable state) or the specified cycle limit is reached.

The recognition of a pattern is performed by setting the activation to the pattern to recognize and then recalculating it using the actual weight and threshold values until the activation doesn't change anymore. This final state is the pattern recognized. The Hopfield network routine works directly on the given activation pattern, i.e. it modifies it directly. The returned pointer is a pointer to the previously specified pattern.

Finally, there are also procedures for loading and saving networks and activations. They store the data in a C compatible format, so that the patterns or weights/thresholds can be easily included into a C project. On loading, some fscanf magic is performed to regain the data. The files can be modified by hand, although some places (beginning of a number block) may not be modified (e.g. more spaces inserted between = and {).

The hopfield network routines are saved in a file called Hopf_net.cpp. There is also a header file called neur_net.h which is included by modules using them. Here, all of the functions which may be used are declared.

3.5 The main programs

The main programs are front ends for various tasks in dealing with the network structures. *hmkpat32* creates activation pattern files from picture files for *hteach32*, which teaches a number of patterns to a network. It is required for Hopfield networks that all the patterns are taught to it at once, because otherwise, the weights are driven into the direction of the last pattern taught. Finally, *huse32* reads a picture file and filters the result it through the learnt network. The result is saved to an output activation file. The final step would be comparing the output activation to the set of learnt activations to decide which picture the network has recognized.

hmkpat32 is given basically a picture file name on the command line on execution. The picture is separated (sobel filter, thinning, edge detection, object separation). Then the picture is transformed into feature space by the PET Method. The final activations (there can be some if the separation detected more objects) are saved to the output activation files, which are named in a scheme like "se-0.act", "se-1.act", "se-2.act" and so on. A output scheme mask can also be specified on the command line; a program call can look like this: "hmkpat32 emma1.ppm sep0-%d.act". There can only be one picture separated at a time, which may result in some work to correctly rename the files for the teaching process.

hteach32 takes as arguments the number of files to teach, the number of cycles (default 50) to teach the patterns, and a file mask for the input pattern files (default “se-%d.act”). The hopfield network is stored to the file “hopfnet.net”. The program first initializes the network and loads a possibly existing network file. If not, the network parameters stay at their defaults. Then it reads all of the activation patterns, stores them internally in a linked list and then and performs the learning cycles. The network is then stored to a file, and the program is finished.

huse32 is designed for testing and using the learnt network. So it is given as command line arguments a picture file name, optionally a network file name also optionally an output activation file mask. The program first initializes and loads the network, then separates the given picture, finally transforms every separated object into feature space and the iterates this through the net. The result is written to an output activation file (default mask is “ca-%d.act”).

All programs print a help message and exit if called without parameters. However, there is only one parameters mandatory; all parameters have more or less sensitive default values predefined. A future extension will be to provide a more common syntax, i.e. specifiers for what follows next (“-if my.act -p emma1.ppm”). Another extension will be the comparison of the result activation to the learning patterns or optionally some other sort of analyzation of the output pattern.

3.5.1 Organization of the programs

The program is divided into the sections “shared files”, “hmkpat32”, “hteach32” and “huse32”, which are realized as Microsoft Visual C++ project files. On the disk, there is also a separate “hinclude32” directory for the commonly used header files. This directory is put into the include path of every project. The code itself is basically C and should be easily portable, as no Microsoft or Borland specialities are used. The files are stored in CPP files because of using single-line comments, variable declarations inside of the function and other specialities only available in C++. There aren’t, however, any object structures (yet).

Appendix A

Records

A.1 Record of March 10th, 1999

Leader: DI Mag.Roland Schwaiger
Recorder: Mayer August
Participants: Spitzlinger Markus
Wiesbauer Gerald
Place: Salzburg
Time: 12^{30}

- General discussion
- Website under http://www.cosy.sbg.ac.at on Mr. Schwaigers Homepage
- Requirements for the paper
  - Structure
  - Records in the appendix
  - Goal: Understanding of the topic, (detailed) description of some applications
- LaTeX-file until Di 13^{00} as TeX-Source
- Possible Applications:
  - TSP (Travelling Salesman Problem)
  - Associative Memory: Picture searching (IBM), denoising of pictures
- Should programs be written?
  - not required, but appreciated
  - also: search ready programs in the Internet
  - utilization of Java
  - would be good for presentations of the institute
- Are there ready-made simulators?
  - SNNS (Stuttgarter Neuronale Netze Simulator) is/will be installed
  - optional: search in the net
- Literature
  - Igor Aleksander, "'Introduction to Neural Computing"'

- Activities until next week:
  - Compressed overview of Hopfield networks
  - Later: search applications in the Internet

A.2 Record of March 17th, 1999

Leader: DI Mag. Roland Schwaiger
Recorder: Spitzlinger Markus
Participants: Mayer August
Wiesbauer Gerald
Place: Salzburg
Time: 12^{30}

- Old points:
  - Title-page
  - Photos (Wiesbauer, Mayer)
  - Picture
- New points:
  - Glossary
  - Studying Igor Aleksander, "'Introduction to Neural Computing"'
  - Detailed description of Hopfield Networks
    * Example
    * How learns a HN?
  - Simulate a HN with SNNS

A.3 Record of March 24th, 1999

Leader: DI Mag. Roland Schwaiger
Recorder: Wiesbauer Gerald
Participants: Mayer August
Spitzlinger Markus
Place: Salzburg
Time: 12^{30}

- Consider the interface between human and machine, and the useability
- Improve the example for the HFN
- Example with 4 neurons to test the SNNS
- Consider the possibility of calculating the attracktors (stabile state)
- Rework the text, which consider the learning of a HFN
- List the kinds of ANN
- Consider a formular description of ANN

- Exercises:
  - Work with the SNNS use example ......
  - List examples of HFN (in the real world)
  - How many stable states are there in an N neuron HFN?
  - Form an accurate description of the components of an ANN
- Additional:
  - EMMA: Discuss the algorithm of current pattern recogition, implemented in ”EMMA's-brain”
  - Feature space, results after a fourie-transformation
- Books:
  - D.Mauck F.Klawonow R.Kruse, ”Neuronale Netze + Fuzzy Logic”

A.4 Record of April 14th, 1999

Leader: DI Mag. Roland Schwaiger
Recorder: Mayer August
Participants: Wiesbauer Gerald
Spitzlinger Markus
Place: Salzburg
Time: 12^{30}

- Hopfield-Network has not yet been modelled with SNNS
- “Typos”:
  - TAN is only described in I. Aleksander's Book ⇒ Drawing
  - Table in introducing chapter as Boolean Table
  - incomplete sentence in the learning chapter
  - Salesman → Salesperson
- Style considerations:
  - BIB-file was missing
  - Make subsections at the network training chapter
  - Structure of the Pattern Recognition chapter like the TSP-Article
  - Move chapter about HN as Associative Memory into introduction
- Content:
  - more detailed: problems in error calculation – no signal for the desired state at the HN, learning by backpropagation; input, output and hidden layer combined in a unit
  - more detailed description of the used variables (θ); change a_u to A_u
  - big step in the example for the learning procedure
  - TSP – How is the problem mapped onto ANN implementations?
  - How is pattern recognition used for image reconstruction?
  - Description of Associative Memory
- Activities:
  - Model TSP and Hopfield Network with SNNS;

- Results in using Hopfield Networks for solving the TSP? Quality of programs?
- Search for further applications
- Search for applications as associative memory: image recognition, typing error resistant word search
- Extend Glossary

A.5 Record of April 28th, 1999

Leader: DI Mag. Roland Schwaiger
Recorder: Mayer August
Participants: Wiesbauer Gerald
Spitzlinger Markus
Place: Salzburg
Time: 12^{30}

- Theoretic text about Pattern Recognition with Hopfield Nets
- Programming project: Pattern Recognition
  - Pattern separation
  - Object recognition: Recognition of the objects found in the pattern separation step
- Will be programmed in the language C
- search for applications and methods in books, internet

A.6 Record of May 12th, 1999

Leader: DI Mag. Roland Schwaiger
Recorder: Spitzlinger Markus
Participants: Wiesbauer Gerald
Mayer August
Place: Salzburg
Time: 12^{30}

- Object Separation:

  The sobel edge detector works already.
  - Thinning algorithm?
  - Edge following algorithm?
- Transformation into Feature Space: testing our algorithm.
- Search for applications and methods in books, internet.
- Search for other solutions of this problem.

A.7 Record of June 2nd, 1999

Leader: DI Mag. Roland Schwaiger
Recorder: Mayer August
Participants: Wiesbauer Gerald
Spitzlinger Markus
Place: Salzburg
Time: 12^{30}

- Description of the program handed over (Edge detection, Object separation) by Mr. Spitzlinger
  - Book: "Mathematische Morphologie"
  - after thinning: make black and white picture?
  - object separation does yet work
  - poss. contact group "Fast object recognition" for pictures
  - try with original pictures because of poor quality, shadows
- Description of the feature space transformation
  - PET-Method by Mr. Wiesbauer
  - distinct cases of use
  - FFT still to realize
- How to feed the separated picture into the feature space?
  - fill the pictures/objects/contours?
  - noise? thresholds?
- Presentation on Thursday?

A.8 Record of June 16th, 1999

Leader: DI Mag. Roland Schwaiger
Recorder: Wiesbauer Gerald
Participants: Mayer August
Spitzlinger Markus
Place: Salzburg
Time: 12^{30}

- Presentation layout: see project homepage
  - titlepage
  - problem specification
  - possible solutions
  - 2 slides: own solution
  - results
- first results of the programs; final layout and method

A.9 Record of June 30th, 1999

Leader: DI Mag. Roland Schwaiger
Recorder: Spitzlinger Markus
Participants: Mayer August
Wiesbauer Gerald
Place: Salzburg
Time: 12^{30}

- Object Separation results:
  - edge detection with sobel algorithm
  - thinning algorithm
  - edge following algorithm (parts of objects; fuseing interleaving objects)
- Transformation into Feature Space results
- Hopfield network results (normalizing)
- presentation details see project homepage

Appendix B

Glossary

cybernetics: The comparative study of the internal workings of organic and machine processes in order to understand their similarities and differences. Cybernetics often refers to machines that imitate human behaviour.

firing: The word comes from the language of the biologist interested in the real neurons of the brain. A neuron is said to fire when it emits a buzz of electrical pulses at the rate of about 100 per second.

firing rule: It determines how one calculates whether the node should fire for any input pattern.

Neural Computing: Is the study of networks of adaptable nodes which, through a process of learning from task examples, store experiential knowledge and make it available for use.

neuron: Is the adaptive node of the brain.

nodes: Are adaptable of the nodes of the brain, they acquire knowledge through changes in the function of the node by being exposed to examples.

simulated annealing: A technique which can be applied to any minimisation or learning process based on successive update steps (either random or deterministic) where the update step length is proportional to an arbitrarily set parameter which can play the role of a temperature. Then, in analogy with the annealing of metals, the temperature is made high in the early stages of the process for faster minimisation or learning, then is reduced for greater stability.

strength: see weight

TSP: Travelling Salesperson Problem.

The problem is that a salesperson wants to travel through N cities and he wants to know the shortest way through those cities.

wells: Wells are stable states.

weight: Firing rules are couched in terms of some weight (or strength) associated with each connection in the net.

Bibliography

[Ale93] Igor Aleksander. *An Introduction to Neural Computing*. Chapman&Hall, 1993.

[Bäu97] Christian Bäumer. Hopfield netze. Vortrag im Seminar Neuronale Netze und Fuzzy-Systeme im Sommersemester 1997, 1997.

[DN96] Rudolf Kruse Detlef Nauck, Frank Klawonn. *Neuronale Netze und Fuzzy-Systeme*. vieweg, 1996.

[Fau94] Laurene Fausett. *Fundamentals of neural networks: architectures, alorithms and applications*. Prentice-Hall, 1994.

[Hio96] Eric Hiob. The travelling salesman problem . http://www.scas.bcit.bc.ca/scas/math/entrtain/tsp/t_tsp.htm, 1996.

[Hof93] Norbert Hofmann. *Kleines Handbuch Neuronale Netze*. vieweg, 1993.

[Olm98] David D. Olmsted. History and principles of neural networks . http://www.neurocomputing.org/history.htm, 1998.

[Rob95] Guy Robinson. An improved method for the travelling salesman problem . http://www.npac.syr.edu/copywrite/pcw/node260.html, 1995.

[Roj93] Raúl Rojas. *Theorie der neuronalen Netze*. Springer, 1993.

[Tve98] Donald R. Tveter. Hopfield and boltzman routines from the basis of ai . drt@christianliving.net – http://www.dontveter.com, 1998.