Analysis of Self-Organizing Neural Network Learning and the Design of a Visualization Tool (VisualNnet)

Petr Hummel

Czech Technical University in Prague — Faculty of Electrical Engineering — Master's Thesis, 2001

Contents

UNOFFICIAL ENGLISH TRANSLATION

This document is an **unofficial, unauthorized English translation** of a Czech master's thesis defended at the Czech Technical University in Prague (ČVUT), Faculty of Electrical Engineering, in **2001** (author: Petr Hummel). It has no official academic standing of its own and was produced decades after the original.

The Czech source was recovered by **OCR** from a PDF whose embedded text layer was unrecoverable (the file was generated by an old `dvips`/LaTeX pipeline using Type 3 fonts with no Unicode mapping, so the page rendered correctly but could not be copied or extracted). Prose was recovered at high fidelity; **mathematical equations were reconstructed** by hand from the rendered pages and should be checked against the original. The original document's **title was not present on the recovered title page**; the title above is a descriptive reconstruction based on the chapter headings.

Figures are not reproduced here. Each figure is represented by a captioned placeholder; the diagrams, plots, and screenshots remain in the original PDF and can be embedded on request.

All equations were transcribed from the original; the two that were illegible in the first scan (2.18–2.19) were recovered from the author's original 2001 PostScript file and are reproduced exactly. Translator's notes appear at the end.

Declaration

I declare that I have worked out the assigned master's thesis independently, with the contribution of the thesis supervisor, and that I used only the literature cited in this thesis. I declare that I have no objection to the use of the results of this thesis by the faculty, nor to its publication or lending with the consent of the thesis supervisor.

_____

Introduction

The phenomenon of neural networks began to emerge as early as the middle of the twentieth century. One of the driving forces behind the development and exploration of this field has undoubtedly been the desire to understand human thought. Human thinking is truly one of the wonders of our nature, and so from the very beginning of the human race it has attracted many scientists and researchers. Almost everyone has asked: how is it even possible that a person can think, learn, analyze situations, and derive procedures for solving problems? Perhaps that is also why I set out on a path that may lead to a deeper understanding of how the human brain works.

At present the spectrum of neural networks is quite vast, and given the differences between the individual paradigms it is almost impossible to design a single visualization tool for all types. My choice was influenced to some extent by previous work on the topic of self-organizing networks, in which I tried to answer the question of which kinds of examples Kohonen self-organizing maps are suitable for and which not. I began to study this problem in more detail and gradually deepened my knowledge of further self-organizing networks. Today self-organizing neural networks are a very large area, and so my work is focused on only some types. For visualization I chose two types of self-organizing network: a network based on Kohonen learning (often also called Hard Competitive Learning) and the rather little-known Dot Product SOM, of which the literature contains only a brief mention without any context or proofs.

In the literature one can find a great deal of information about Kohonen learning, but this is much less true of simple practical examples. There is, for instance, no comprehensive set of simple examples capturing all the essential features of Kohonen learning that would noticeably speed up the understanding of properties that are not evident at first sight. Of the Dot Product network even less can be found. In the literature [6] there is only a short article giving the algorithm without any explanation or proofs. The aim of my thesis is therefore to create a visualization tool that would demonstrate, on examples, all the essential features of both paradigms. For this reason the entire second chapter is devoted to the principles of the selected self-organizing networks. Through theoretical analysis I show the individual properties and problems of each of the mentioned networks. This is extremely important for the subsequent visualization, because only through careful analysis of the individual models can I create a tool covering the whole area. My analysis of the Dot Product SOM network is supported by my own proofs, because it was not sufficiently clear from the algorithm why this network works as the literature [6] states.

A further important step is the analysis of existing visualization tools. It is always better to learn from existing solutions, and so the third chapter discusses the individual methods of visualizing neural networks. I also evaluated each method in terms of its usefulness for my demonstration applet, VisualNnet.

The fourth chapter is devoted to the design of the VisualNnet demonstration applet. On the basis of the theoretical analysis in Chapter 2 and the analysis of current visualization tools, I formulated the requirements the proposed applet should satisfy. On the basis of these requirements I specify a more detailed design of the individual parts, including a description of my own implementation in the JAVA language. In the design I also placed great emphasis on the possibility of extending the applet. In the implementation description I give instructions and new ideas on how the individual classes can be used to extend new functions and methods. In this way, for example, multilayer neural networks can easily be implemented, and the described Sammon transformation can easily be added for displaying multidimensional data.

Programming the last line of the program is far from the end of development. The program must be subjected to careful testing. What tests the program passed, and with what results, can be read in the conclusion of the fourth chapter.

In the fifth chapter I then summarize the contribution of the applet and propose test examples whose aim is to demonstrate the individual characteristic properties identified in the second chapter. In conclusion I evaluate my achieved results and indicate the possibilities of further use and possible extension of the VisualNnet applet.

1 Introduction to Neural Networks

A neural network is composed of neurons interconnected so that the output of a neuron is the input of, in general, several neurons. The mathematical model of a neuron derives from the nature of the neurophysiological neuron. A neuron has n generally real-valued inputs. The inputs are weighted by corresponding, generally real, synaptic weights that determine their permeability. In agreement with the neurophysiological motivation, synaptic weights may be negative, thereby expressing an inhibitory character. A neuron has its internal potential, which in general can be computed from a function whose input values are the inputs of the neurons and their weights. This function differs depending on the type of neural network, but most often it is the weighted sum of the input values. Once the internal potential reaches a threshold value, it induces the output of the neuron.

The number of neurons and their mutual interconnection in the network determine the so-called architecture of the neural network. A neural network evolves over time: the interconnection and the state of the neurons change, and the weights adapt. This evolution is called the learning of the neural network.

1.1 Basic classification

The learning of neural networks can be divided into two basic types — supervised learning and unsupervised learning. For a neural network to be able to learn, it needs "something" that will tell it how it should change its configuration so that it converges to a target state. The learning of a neural network is an iterative process, and in each iteration the correct change of configuration must be decided. In the ideal state, the neural network produces the required outputs when the correct input values are presented. In other words, it is known what outputs the neural network should produce in some cases. If the neural network is suitably designed, then from the differences between the required outputs and the obtained outputs one can estimate how the configuration should change so as to give better results. If a neural network is based on the obtained output being classified from the outside, then this is supervised learning. A typical representative of supervised learning is the Learning Vector Quantization network. The second type are networks based on unsupervised learning; I analyze some of their variants in the following chapter.

2 Analysis of Neural Network Learning

2.1 Selected self-organizing neural networks

2.1.1 Motivation

Self-organizing networks have undergone considerable growth since the 1980s and today constitute a fairly extensive chapter of neural networks. The common principle of self-organizing network

models is that the output neurons compete over which of them will be active. Unlike other learning principles, then, only one neuron is active at any given time. First, however, we shall clarify some basic definitions and concepts on which the theory of self-organizing networks itself then builds.

2.1.2 Vector quantization

One of the cornerstones is undoubtedly vector quantization. The aim of vector quantization is to approximate the probability density $p(x)$ of real input vectors $x \in \mathfrak{R}^n$ using a finite number of units, or representatives, $w_i \in \mathfrak{R}^n$. Once the representatives have somehow been determined, then to each vector $x \in \mathfrak{R}^n$ the unit w_i that is closest is assigned as its representative, that is:

$$c = \arg\min_i \|x - w_i\| \tag{2.1}$$

To determine the quality of the approximation we introduce a so-called error function, which we define as follows:

$$E = \int \|x - c\|^2 \, p(x) \, dx \tag{2.2}$$

where $\|\cdot\|$ is the Euclidean norm and c is defined by (2.1). The most common cases, however, are those in which the probability density $p(x)$ is unknown and the problem is given by a finite training set of samples $T = x^{(t)};\ t = 1, \ldots, k$. In that case we compute the vector-quantization error as follows:

$$E = \frac{1}{k} \sum_{t=1}^{k} \|x - w_c\|^2 \tag{2.3}$$

Although we know how to compute the error function, its minimization is not simple, because the index c depends functionally on both the samples x and the units w. For these reasons, too, it is not easy to express explicitly the gradient of the error E with respect to the parameters w and then apply the usual minimization procedure. Therefore, heuristic iterative algorithms seeking an approximate solution were proposed — such as Lloyd's algorithm, the Kohonen neural network, and so on. Vector quantization is thus the reformulation of the problem of approximating a probability density by means of representatives, where the representatives are placed on the basis of minimizing the error function. The significance of the error function is that it makes it possible to define the "quality" of the approximation; sometimes this error function is also called the quantization error. The manner of minimizing this error is then a matter of the algorithms designed for it.

2.1.3 Kohonen self-organizing network

One of the first attempts to minimize the error function (2.3) is Lloyd's algorithm [1]. This algorithm is disadvantageous precisely in that changes to the representatives occur only after a pass through the whole training set. Therefore an on-line variant was developed by Kohonen, called the Kohonen self-organizing network. The Kohonen self-organizing network is a simpler variant of Kohonen self-organizing maps. The difference between them is that the Kohonen self-organizing network introduces no connections between the individual neurons, and the weights are adjusted only at the winning neuron.

The structure of the network is simple. It is a two-layer network with full connection of the units between layers. The input layer is formed by n neurons, which serve to distribute the input values $x \in \mathfrak{R}$. The units in the output layer compare according to the Euclidean distance. The weights $m_i = (m_{i1}, \ldots, m_{in}),\ i = 1, \ldots, n$ belong to the i-th output unit and determine its position in the input space.

While the inputs $x \in \mathfrak{R}$ of the network may be arbitrary real numbers, the outputs y_i are usually only the values 0 or 1, where exactly one output neuron is always active with the value 1. The active neuron is precisely the one that won the competition. The output of each neuron is obtained, depending on its distance from the input vector x, as follows:

$$y_i = \begin{cases} 1 & i = \arg\min_k \|x - w_k\| \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

In the literature this algorithm is also often referred to as Hard Competitive Learning, the term I use in my work. In Czech it means "winner takes all," and one of the mechanisms by which it is realized is so-called lateral inhibition. For a clearer picture we may introduce lateral connections between the output neurons that transmit inhibitory signals between them. Each output neuron thus tries to weaken the other neurons in the competition with a force directly proportional to its potential. The end result is therefore that the output neuron with the greatest potential suppresses the other output neurons and itself remains active.

The result of the competition is the neuron that best represents the presented input. The aim of this neuron is to improve its relative position with respect to the presented input still further, so as to represent the given input even better. We therefore move the winning neuron toward the selected input vector by a certain proportional distance $h(t)$:

$$w_i(t+1) = \begin{cases} w_i(t) + h(t)\,[x(t) - w_i(t)] & i \text{ is the winner} \\ w_i(t) & \text{otherwise} \end{cases} \tag{2.5}$$

A measure of the quality of the network's learning is the difference between the minimal error function for a given set of input vectors and a given number of neurons, and the error function of the learned network. Because it is very difficult to find the minimal function in general, no uniform procedure can be obtained for absolutely assessing how well a network has learned. It is, however, possible to select the best of several obtained solutions on the basis of comparing the individual error functions. Here we may ask: is it true that in the ideal case the result should give the minimal error function? The answer is not unambiguous. Tsypkin showed [3] that Kohonen's algorithm finds a k-means clustering solution if the value h is inversely proportional to the share of vectors x lying closer to the winning representative than to the others. There are, however, situations in which this does not hold. Hecht-Nielsen in [4] gives a simple example of two isolated, sufficiently distant regions, where a sample from the second region attracts one of the representatives, which then always wins the competition for samples from this set.

Let us therefore imagine the described situation, where the neurons are placed "at random" so as to deliberately disregard the density of occurrence of the set of input samples. We place one neuron (call it neuron A) closer to one group of input vectors (call it group 1), and the other neurons are placed near the second group (group 2). After the algorithm is started, the weakness of this algorithm immediately becomes apparent. As soon as some input vector from group 1 is selected

for the computation, the nearest neuron A moves toward this group, and if the diameter of this group is smaller than the distance of the nearest other neuron from this group, the already-shifted neuron A will win every competition for all input vectors of group 1. The other neurons then divide up (unless a similar situation has arisen here too) the second group of input samples.

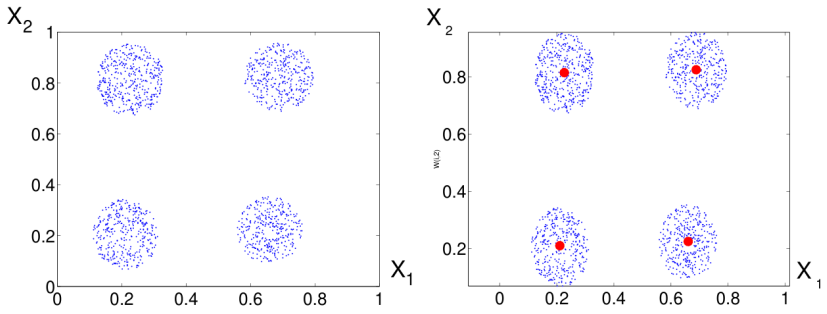

Figure 2.1. Placement of neurons after Kohonen learning on 4 isolated regions.

The input is 4 isolated regions of input vectors, and the output layer is formed by 4 randomly placed neurons. After learning with the Kohonen algorithm, each of the output neurons places itself in the center of each region. So far the input vectors have been defined as isolated points. The input set can, however, also be defined as whole areas that have the area probability density of occurrence as their function. Training samples are then chosen at random with regard to the probability density of occurrence in the individual regions.

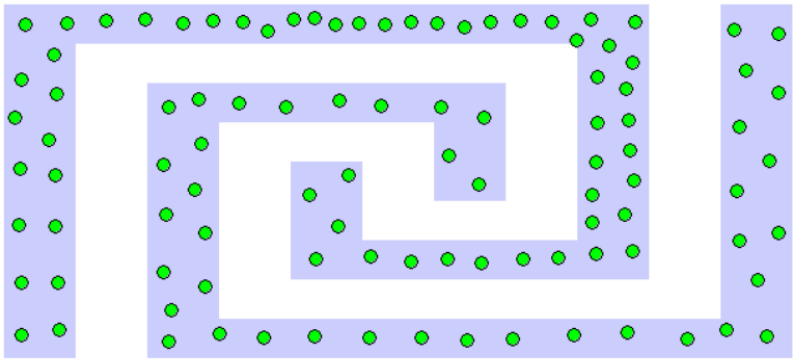

Figure 2.2. Hard Competitive Learning on a spiral.

Figure 2.2 shows how the neurons place themselves in the given space. The placement of the output neurons is analogous to the placement of a gas in cavities formed by the areas of the input data. On closer inspection we see in the figure a slight densification of neurons in some places. The cause is unsuitable initialization in combination with the setting of the initial parameters of the neural network. During learning the neurons spread into all parts of the areas, but because only one winning neuron adapts, the edge neurons usually occupy a larger area than the neurons in the middle of a cluster. With the Hard Competitive Learning method it is difficult to find parameters such that these phenomena are entirely suppressed. Kohonen learning therefore began to be modified so as to suppress the undesirable properties described.

2.1.4 Kohonen self-organizing map

Kohonen self-organizing maps are a more complex type of self-organizing network. The organizational dynamics is similar to that of Kohonen networks. The output units, however, are additionally arranged into some topological structure — most often a two-dimensional grid or a one-dimensional array of units. This topological structure determines which units are neighbors in the network, and it is important in the learning process.

The motivation for modifying the learning of the Hard Competitive Learning method was to suppress the above-mentioned problem of poor initialization and its negative influence on subsequent learning. If several neurons adapt at once, the migration into isolated regions will be greater and the ossification of the other neurons outside a given region is thereby prevented. This brings more dynamics into the learning of the neural network. The fundamental difference is therefore that in the learning process we adapt the weights not only of the winning neuron but also of its neighboring neurons. The neighborhood of the output neurons is defined virtually — it is independent of the actual layout in the input space. At the start of learning we adjust a large neighborhood, and then we shrink the neighborhood over time so as to achieve convergence of the solution. The whole learning algorithm is analogous to algorithm (2.4); the adaptation of the weights is governed by the following equation:

$$m_i(t+1) = \begin{cases} m_i(t) + h_c(i)(t)\left[x(t) - m_i(t)\right] & j \in N_s(c) \\ m_i(t) & \text{otherwise} \end{cases} \tag{2.6}$$

The function $h_c(i)$ is often defined more generally, so that the transition between zero and non-zero values is continuous, which better corresponds to biological interactions. A typically used function is the Gaussian:

$$h_c(i) = h_0 \cdot e^{\left(-\frac{d(i,c)^2}{\sigma^2}\right)} \tag{2.7}$$

Unlike Kohonen networks, the output units now mutually attract one another, so they try to "hold" the structure together. If a Kohonen self-organizing map is applied to 2 isolated regions of input vectors, the output neurons do not place themselves exactly into the centers of both isolated regions, but rather settle between the two isolated regions. If we use the Kohonen-map method on the input vectors distributed as in Figure 2.1, the final placement of the output neurons (Figure 2.3) will differ from that of the Hard Competitive Learning method. In the case of Hard Competitive Learning, in the ideal case the output units are placed in the center of each region. In the Kohonen self-organizing map these units are deflected from the centers of the isolated regions toward the regions belonging to their neighbors.

In Figure 2.3 it is clearly visible how the output neurons mutually attract one another. Both outermost output neurons are shifted only horizontally (they have only one neighbor influencing them), whereas the remaining two neurons are shifted both horizontally and vertically — each has two neighboring neurons by which it is influenced.

Kohonen maps are most often used as two-dimensional networks, but the structure and arrangement can in principle be arbitrary. In the literature one also encounters one-dimensional Kohonen maps. The chosen map structure significantly influences the resulting arrangement of neurons. I therefore generated further input data, shown in Figure 2.4. Figure 2.5 shows the use of a 1D Kohonen map

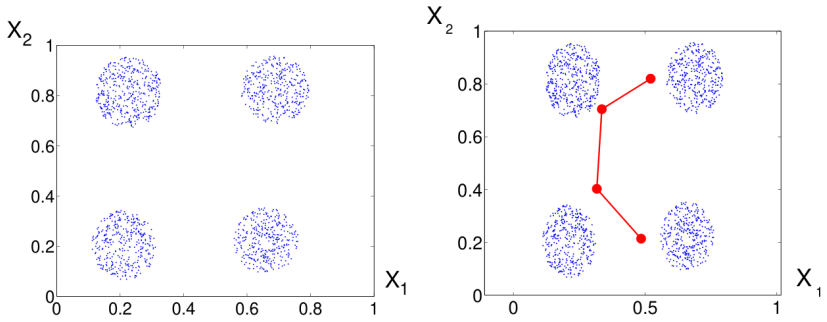

Figure 2.3. Placement of neurons on the given set after learning with the Kohonen-map method.

(the neurons are connected into an open chain). The result is an effort to fit the set with a curve. Figure 2.6 shows how the neurons arrange themselves, after the input data are presented, when a two-dimensional grid is used.

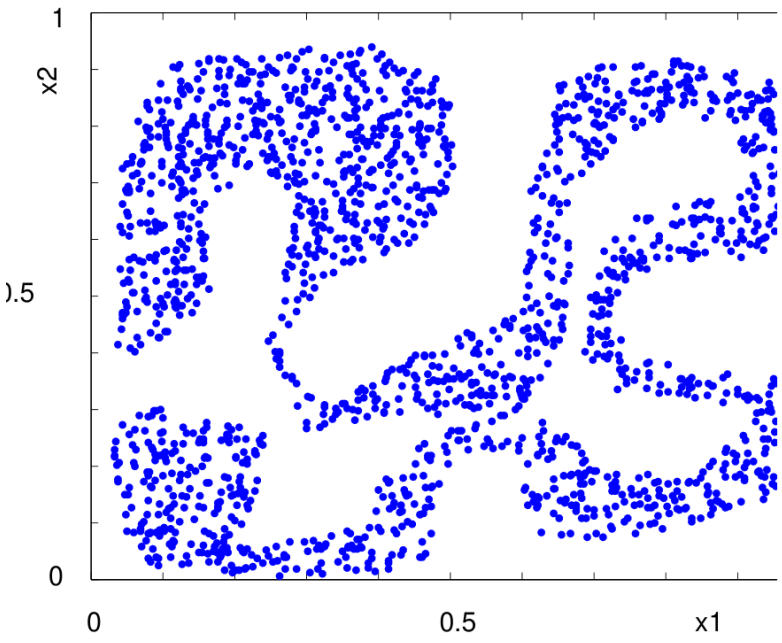

Figure 2.4. Input training set for experiments with various structures.

2.1.5 Neural gas

The idea used in the neural gas is a combination of Hard Competitive Learning and the self-organizing map, but it has small differences. The results when using Kohonen maps depend on the neuron structure used. In the neural gas a different learning method is used, carried out in a different way than with Kohonen maps. During learning we select an input vector. We determine the distance to each output neuron and sort the output neurons by Euclidean distance. We then adjust the weights similarly to the self-organizing map, with the difference that the structure (i.e., the neighborhood of a neuron) is defined by the ordered sequence of neurons according to the current

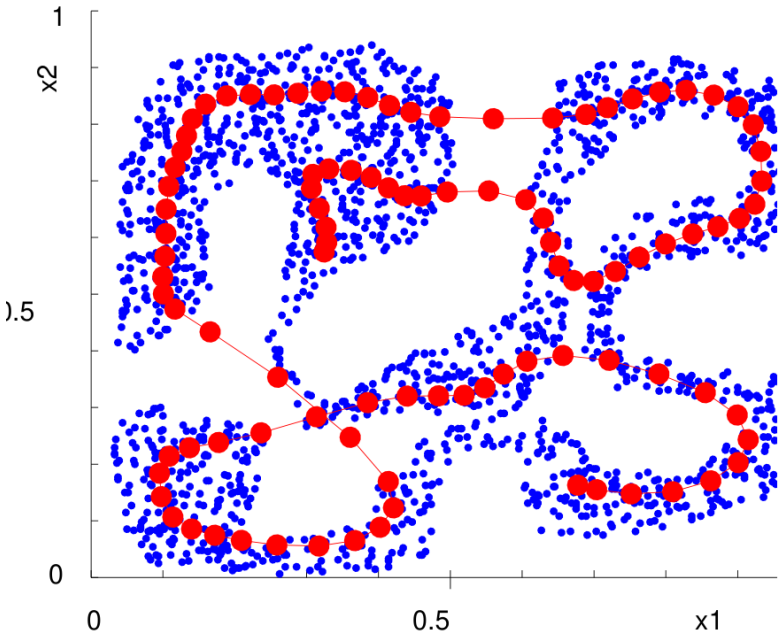

Figure 2.5. 1D SOM — a Kohonen 1D map attempting to capture a 1D structure.

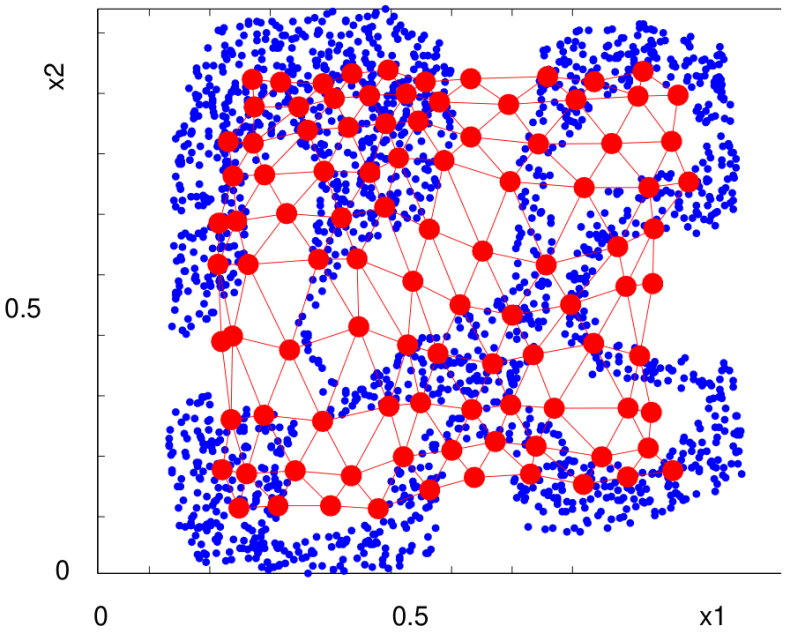

Figure 2.6. 2D SOM — a Kohonen map attempting to capture a 2D structure.

Euclidean distance. Herein lies the fundamental difference from self-organizing maps, where the neighborhood is given by a structure already before learning itself begins. The use of the neural gas can be tried out with the DemoGNG applet [5], created by two programmers from a German university. Using their predefined input sets (Figure 2.2), we perform a simulation with their applet. Figure 2.7 shows how the output neurons are evenly spread over the whole input set of vectors and how no densified places occur, as was the case in Figure 2.2 with the Hard Competitive Learning method.

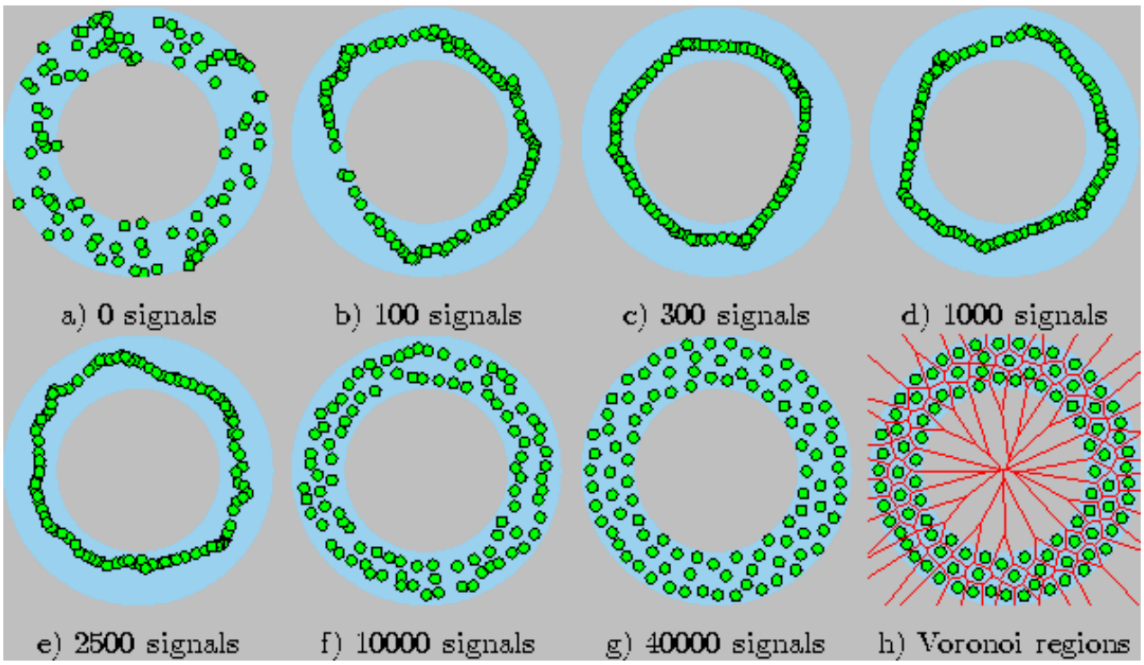

Figure 2.7. Development of the neural gas.

Figure 2.2 was produced with the DemoGNG applet, which can be found on the Internet page [5]. The authors there give further examples of the use of the neural gas — see Figure 2.7, which captures very well the development of the neural gas during learning. The next figure, 2.8, captures the results of the neural gas on various training sets.

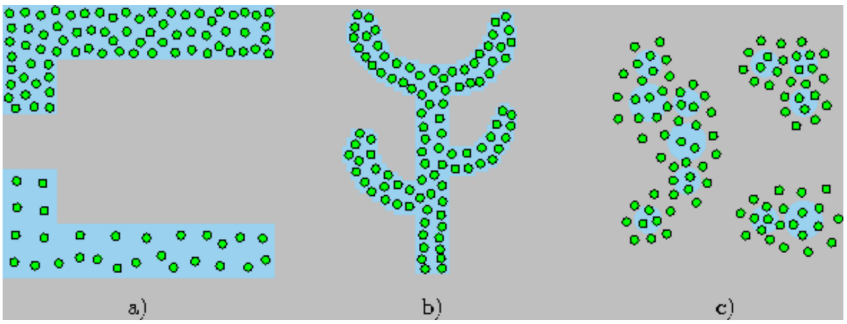

Figure 2.8. Placement of the neural gas on various training sets.

2.1.6 Dot Product SOM

The self-organizing networks given so far naturally have certain disadvantages too, described in the previous analyses. Hard Competitive Learning is sensitive to correct initialization, and the methods based on adapting several neurons at once (Kohonen self-organizing maps and the neural gas) have another unpleasant property that follows from the very principle of vector quantization. If one input sample in the training set has an order-of-magnitude larger value in one of its coordinates than the others, it strongly influences the other neurons (except in Hard Competitive Learning), and thereby worsens the learning in the other regions of the input set of samples.

Therefore the method "Dot Product SOM" was developed, which is based on a different understanding of the meaning of sample similarity and which solves the mentioned problem. The algorithm is given in [6] without any more detailed explanation or proofs. Therefore one of the pillars of my thesis is the analysis of this type of neural network and the proof of several regularities that are not evident at first sight.

The Dot Product SOM is again a two-layer network with full connection of the units between layers. The input layer is formed by n neurons, which serve to distribute the input values $x \in \mathfrak{R}$. The units in the output layer, however, compare according to a different key than the Euclidean distance. The scoring function is in fact the dot product:

$$f(m, x) = \vec{x}(t) \cdot \vec{m}_i(t) \tag{2.8}$$

The weights $m_i = (m_{i1}, \ldots, m_{in}),\ i = 1, \ldots, n$ belong to the i-th output unit and determine its position in the input space. The competition is won by the neuron with the highest scoring function:

$$y_i(t) = \begin{cases} 1 & i = \arg\max_{k=1\ldots l} \vec{x}(t) \cdot \vec{m}_k(t) \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

The reason for choosing the dot product as the function intended for the competition is the fact that, after normalization of the vectors of the output neurons, the similarity is based on the size of the angle that the neuron forms with the input sample. The following consideration, supported by a proof, shows how the neurons divide the training data in the two-dimensional input space.

At the beginning we randomly select one input sample $x = (x_1, x_2)$ and seek the vectors $m = (m_1, m_2)$ for which the dot product is constant:

$$c = \vec{x} \cdot \vec{m} = x_1 m_1 + x_2 m_2 \tag{2.10}$$

From (2.10) we express the unknown m_2 in terms of the variable m_1:

$$m_2 = \frac{c - x_1 m_1}{x_2} = \frac{c}{x_2} - \frac{x_1}{x_2} m_1 \tag{2.11}$$

From (2.11) it follows that the sought sets of points with a constant dot product are lines with direction vector $\left(1, -\frac{x_1}{x_2}\right)$ and constant $\frac{c}{x_2}$. Now it suffices to compute the dot product between the computed direction vector and the input vector x characterizing the presented sample:

$$s = x_1 + x_2\left(-\frac{x_1}{x_2}\right) = x_1 - x_1 = 0 \tag{2.12}$$

Equation (2.12) is the proof of the preceding conclusion. The whole situation can be seen in Figure 2.9.

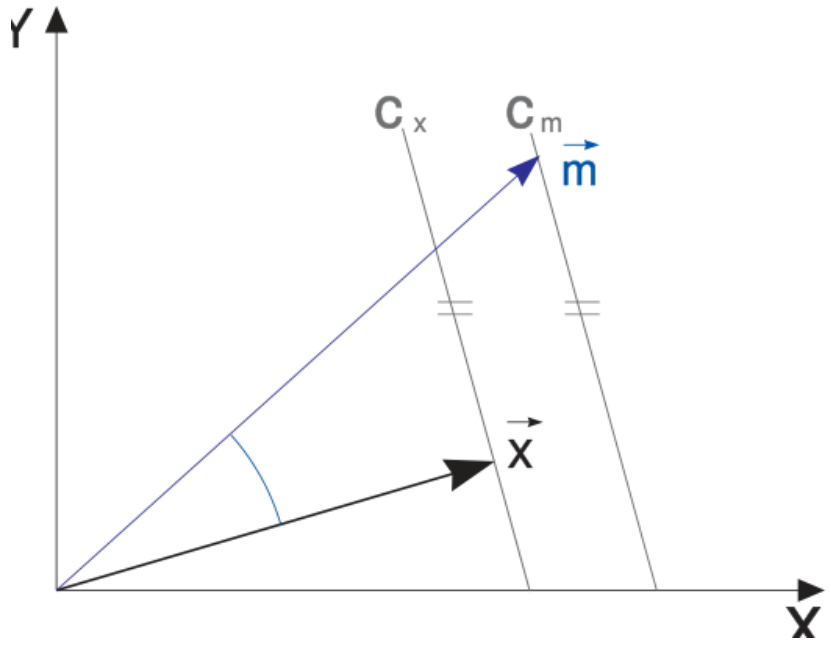

Figure 2.9. Sets of points for constant dot products with the vector X.

For such general values, however, the required proportionality will not hold — namely, that the smaller the angle $\varphi(x, m)$, the more similar the vectors $\vec{m}$ and $\vec{x}$. Figure 2.10 shows a case in which the vector $\vec{m}$ forms a larger angle with the vector X and yet its dot product with X is larger than the dot product $\vec{n} \cdot X$.

This problem can be solved simply by normalizing all vectors except the vector X. The normalized vectors will then always lie on a single circle of radius 1. The situation is aptly illustrated in Figure 2.11. The red line labeled $C_{\max}$ represents the maximal dot product we can obtain in the case of normalized vectors.

The endpoints of the vectors of the given line lie at the intersections of that line c with the circle. The more the line is shifted toward the center, the larger the angle between the obtained vectors and the input vector X. If the line is shifted past the center $[0, 0]$, further to the opposite side from the direction of vector x, the value of the dot product will keep decreasing to negative values. I proved this observation for the first quadrant in the following proof; by the same procedure, however, it can be generalized to the other quadrants.

The idea of the proof is simple. The point is to determine the behavior of the dot-product function in terms of the vectors X and m and to find any maximum. Let us denote the dot product by s and take vectors $\vec{x} = (x_1, x_2)$, $\vec{m} = (m_1, m_2)$ such that $x_1 > 0,\ x_2 > 0,\ m_1 \in (0, 1),\ m_2 = \sqrt{1 - m_1^2}$ (m_2 can be expressed through m_1 thanks to the normalization of vector $\vec{m}$). Then we substitute into (2.8):

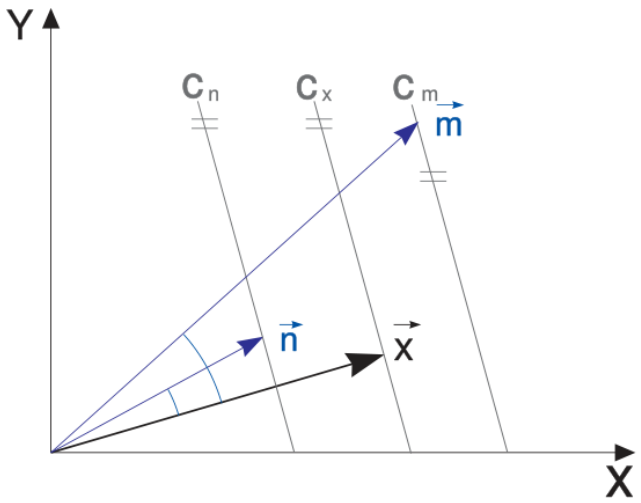

Figure 2.10. The general case of choosing vectors m, n.

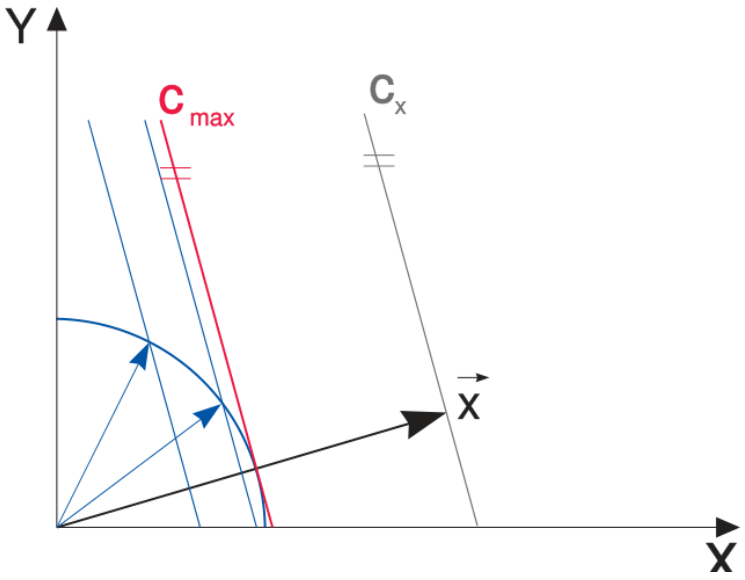

Figure 2.11. Normalized vectors.

$$s = x_1 m_1 + x_2 m_2 = x_1 m_1 + x_2 \sqrt{1 - m_1^2} \tag{2.13}$$

We seek when the dot product s is maximal. We determine the extremes from (2.13) using the first derivative:

$$\frac{ds}{dm_1} = x_1 + x_2 \frac{m_1}{\sqrt{1 - m_1^2}} \tag{2.14}$$

For extremes it holds that:

$$\frac{ds}{dm_1} = 0 \tag{2.15}$$

After substituting (2.15) into (2.14) it then suffices to rearrange suitably:

$$x_1 + x_2 \frac{m_1}{\sqrt{1 - m_1^2}} = 0 \tag{2.16}$$

$$\frac{x_2}{x_1} = \frac{\sqrt{1 - m_1^2}}{m_1} \tag{2.17}$$

We determine the type of extreme using the second derivative $\frac{d^2 s}{dm_1^2}$:

Equations (2.18)–(2.19) below are transcribed exactly from the author's original 2001 PostScript file.

$$\frac{d^2 s}{dm_1^2} = -\frac{x_2}{\sqrt{1 - m_1^2}} \left(1 - \frac{1}{2} \frac{m_1}{1 - m_1^2}\right) \tag{2.18}$$

From equation (2.17) we express x_2 and substitute into (2.18):

$$\frac{d^2 s}{dm_1^2} = -\frac{1}{m_1^2} - \frac{1}{2} \frac{1}{m_1^2 \left(1 - m_1^2\right)} \tag{2.19}$$

From (2.19) and the assumption $m_1 \in (0, 1)$ it follows that, when all assumptions are satisfied, the obtained second derivative is always negative. Therefore the sought extreme is a maximum.

The maximum of the dot product occurs if and only if condition (2.16)/(2.17) holds. After rearranging this condition (2.20) it is evident that the vectors $\vec{m}$ and $\vec{x}$ are parallel — that is, they form a zero angle between them:

$$\frac{x_2}{x_1} = \frac{\sqrt{1 - m_1^2}}{m_1} = \frac{m_2}{m_1} \tag{2.20}$$

Therefore the competition in the Dot Product SOM is based not on Euclidean similarity but on "directional" similarity. From this also follows the procedure for learning. The aim of learning is

to move the position of the neuron toward the position of the selected input sample, so that it represents the given input even better. The simplest operation that satisfies the requirement is the plain addition of the input vector $\vec{x}$ to the vector of the selected neuron $\vec{m}$. Since in general the proportionality between the enclosed angle and the scoring function does not hold, we must normalize the resulting vector. A learning algorithm designed this way would, however, not converge to any solution, because the positions of the neurons would never stabilize. To stabilize the position of the neurons we use a coefficient $\alpha(t)$, by which we specify what part of the vector $\vec{x}$ should change the new position of the vector $\vec{m}$. The coefficient α can be set at the beginning to any positive value and decreased toward zero over time. For a similar purpose the function (2.7) is also used, which is used in the adaptation in Kohonen maps on page 10 [of the original]. Adaptation therefore proceeds according to the relation:

$$m_i(t+1) = \begin{cases} \dfrac{m_i(t) + \alpha'(t)\, x(t)}{\left\| m_i(t) + \alpha'(t)\, x(t) \right\|} & i \in N_c(t) \\ m_i(t) & i \notin N_c(t) \end{cases} \tag{2.21}$$

In the Dot Product SOM method, the coefficient α may at the beginning be greater than one. The author [6] states that thanks to this, normalization of the vectors happens automatically. As soon as a not-yet-used neuron is adapted, it is automatically normalized from that time on by relation (2.21). If $\alpha(t) > 1$, the learned neuron is "un-learned," and thereby gives another neuron the chance to win. Thus it is highly probable that at the beginning of learning all neurons become normalized. After the value of α is gradually reduced, learning then proceeds in a scenario similar to Kohonen learning. As with Kohonen maps, here too we can introduce virtual connections and adjust the neighborhood N_c, or adjust analogously according to the rank determined by the obtained dot-product values (the same principle can be found in the neural gas).

3 Current Visualization Tools

3.1 Definition

Thinking and imagination are inseparably linked. Imagination is a very broad concept, and on the rational level it is above all the ability to model any problem in the human mind. Visualization tools help us to understand the problems under study better. Neural networks are an area that is quite difficult to understand, because complex mathematics is hidden behind them, and moreover they often involve iterative processes. A visualization tool can be said to be anything that graphically helps to represent a given problem. In order to design a new tool, I needed to familiarize myself with existing tools, ascertain their advantages and disadvantages, and on the basis of that evaluation design a tool that would bring new possibilities for explaining the selected paradigms. Roughly, visualization tools can be divided into two categories: single-purpose and universal.

3.2 Single-purpose tools

Among single-purpose tools we may classify the Sammon transformation, SOM (The Self-Organizing Map Pack), LVQ PAK, and others. They are characterized by being intended for only one type of problem; other problems cannot be solved with them. There are a great many such tools, so I mention only those I have had the opportunity to become acquainted with.

3.2.1 Sammon projection

I began to study the Sammon projection more deeply during experiments with Kohonen maps. During my experiments it often happened that the map was twisted after learning. In some cases even retraining the whole network did not help. Originally I attributed the twisting of the network to poor training of the network, but during analysis it turned out that the actual cause lies in the Sammon transformation itself.

In practice we often need to work with data whose domain is more than three-dimensional. To be able to gain at least a partial idea of the structure of the data, we must have a tool that converts this data into a representable form. One of these views is precisely the Sammon projection. It tries to transform data from an n-dimensional space into an m-dimensional space ($m \leq 3$) while preserving the structure of the data as much as possible, in the form of an effort to preserve the ratios of distances between the individual vectors in the input space.

3.2.1.1 Definition The Sammon projection is a nonlinear projection from an n-dimensional space into a d-dimensional space, where $n > d$, so that the ratios of distances between the individual vectors are preserved as much as possible. For this it is necessary to define a scoring function that will determine the "quality" of the transformation. This function is again the error function:

$$E = \frac{1}{\sum_{i<j} \left[d^*_{ij}\right]} \sum_{i<j}^{N} \frac{\left[d^*_{ij} - d_{ij}\right]^2}{d^*_{ij}} \tag{3.1}$$

By minimizing this error we obtain the best possible projection. Because we cannot express the global minimum exactly, a gradient method is used.

The Sammon projection is also used in the visualization of data in Kohonen neural networks and maps. In a Kohonen neural network, the Sammon projection is used to display the output neurons (the represented classes), which are n-dimensional vectors, as two-dimensional vectors. These two-dimensional vectors are then visualized and thus allow one to "peek" at the given structure.

3.2.1.2 Properties The Sammon transformation is a means of viewing multidimensional data. It also has its shortcomings, however, some of which arise from the very principle of the Sammon projection. This can best be demonstrated on the following examples.

In the first example we use the vertices of a tetrahedron as the input set (an object that has 4 vertices, all equidistant from one another — i.e., all its faces are formed by equilateral triangles). In Figure 3.1 we see the result of the Sammon projection from 3D into 2D.

In 2D space it is not possible to find an object satisfying all the parameters of a tetrahedron. In 2D space there exists only an object with at most three vertices that are mutually equidistant — the equilateral triangle. When displaying more points under the given condition, it is no longer possible to introduce a projection in which the ratios of distances are identical. A fundamental error therefore arises in this projection that cannot be suppressed.

From this error follows the following unpleasant fact. If the structure in the input space is nontrivial, then the Sammon projection cannot guarantee the property that the most distant points in the input space will also be the most distant in the transformed space. Sometimes it even happens that the most distant points in the input space are placed side by side in the transformed space. Confirmation

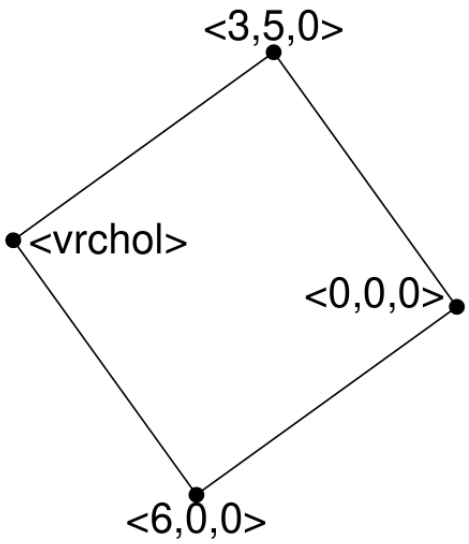

Figure 3.1. A 3D tetrahedron after applying the Sammon projection.

of this fact can be seen on a 4-dimensional cube. After applying the Sammon projection from 4D into 2D we obtain the following figure.

The most distant points are those that differ in all four positions. The vertices 0000 and 1111 are indeed the most distant from each other, but the vertices 1001 and 0110 are far closer to each other, even though they are the most distant in the input space. This is even more striking in higher-dimensional space. As an example we use a 6-dimensional cube. After the transformation, the vertices 110000 and 001111 come close to each other, even though they are the most distant in the input space.

In Figures 3.4 and 3.5 we can see how vertices at distance 1 and 2 are scattered from a chosen vertex.

Another unpleasant property of the Sammon projection follows from the use of the gradient method in minimizing the error function. The gradient method does not guarantee finding the global minimum; it may get stuck in a local minimum, at the expense of a better solution.

3.2.1.3 Evaluation The Sammon transformation is thus a means that makes it possible to view multidimensional data visually and to gain at least an approximate idea of the structure of the data. It also has its limitations, however, which follow from the very principle of the impossibility of preserving all properties when projecting into lower dimensions; therefore one cannot always rely on the displayed structure.

An implementation of the Sammon projection can be found, for example, in the SOM package.

3.2.2 SOM (SOM Pack)

SOM Pack, or The Self-Organizing Map Program Package, is a set of programs intended for working with Kohonen maps. It was created at the University of Helsinki, which is one of the leaders in the field of self-organizing networks. SOM Pack is implemented for the operating systems UNIX and

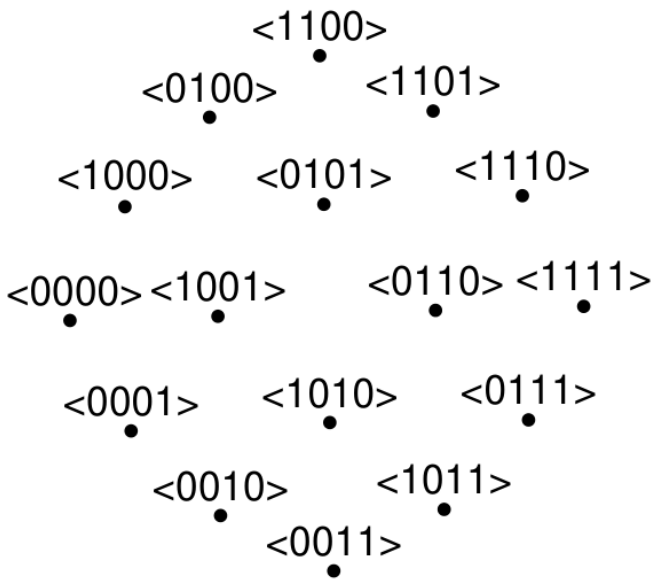

Figure 3.2. Sammon projection of a 4-dimensional cube.

DOS, and libraries for Matlab were even released. All interaction between the user and the programs under UNIX and DOS proceeds via the command line using parameters, so this program package does not have a very user-friendly environment. Nevertheless, this tool offers very good possibilities for using Kohonen maps. Under DOS, SOM Pack is composed of the following programs:

- **Randinit.exe** initializes the vectors characterizing the neurons.
- **Lininit.exe** introduces virtual connections between neurons (see page 10 of the original). The structure of the connections can be specified via input parameters.
- **Vsom.exe** is an implementation of Kohonen-map learning. The input consists of the learning parameters and two files: one file contains the training samples, the other the generated layer of neurons. The output is then a file with the learned neurons.
- **Calibration** is the process in which, for each neuron, it is determined which input sample belongs to it, and from a calibration file the neuron is assigned the corresponding name representing the sample.
- **Qerror.exe** essentially computes the error function for a given learned network. The error function is computed according to formula (2.3) on page 6 of the original. The value, however, cannot be assessed in absolute terms; it is meaningful when comparing the learning quality of the network for different parameters with the same configuration and input training set.
- **Vcal.exe** sets the maximum number of labels for a single neuron.
- **Umat.exe** displays the distances between neighboring neurons using gray levels into a PostScript file.
- **Planes.exe** performs so-called two-dimensional slices. Two dimensions are selected via parameters, and into them the vectors characterizing the positions of the neurons in n-dimensional space are projected.
- **Sammon.exe** performs the Sammon projection. It generates output into a PostScript file. The Sammon projection is described in the previous chapter.

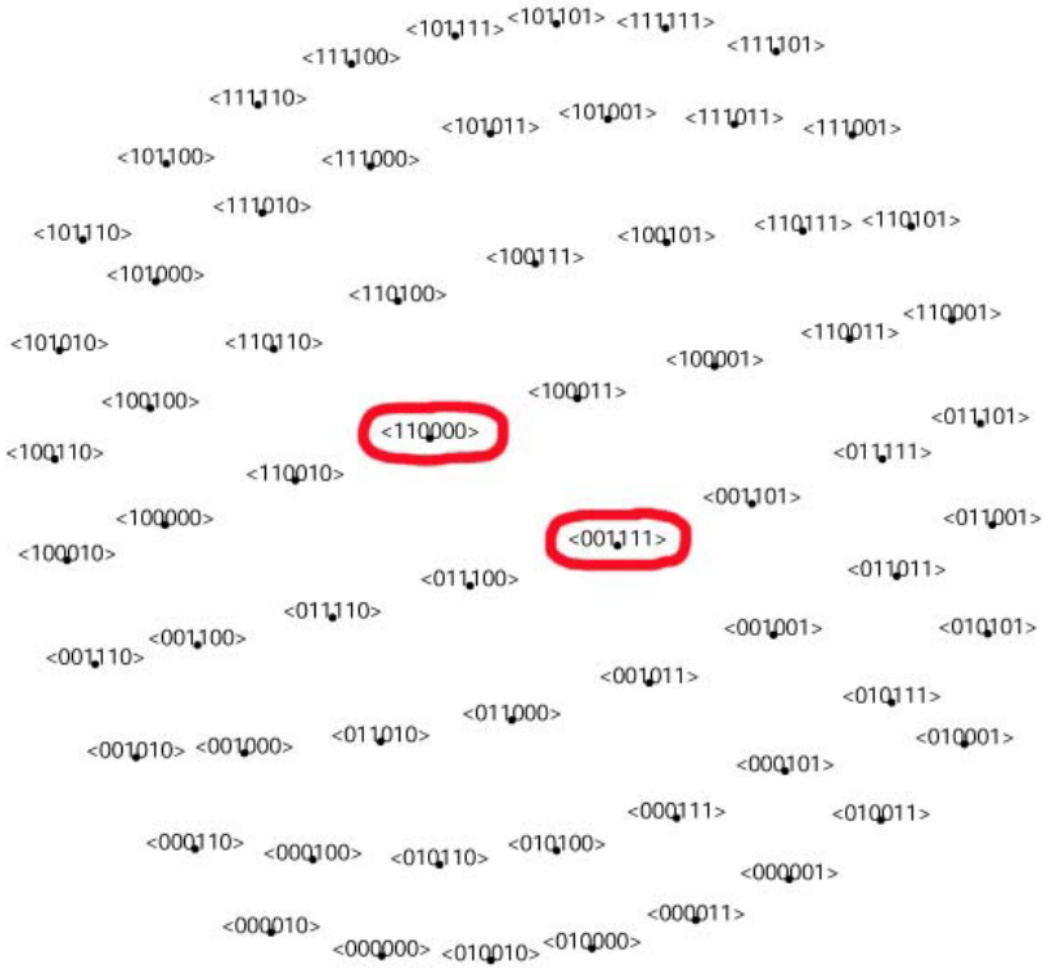

Figure 3.3. Sammon projection of a 6-dimensional cube.

3.2.2.1 Evaluation SOM Pack is a computational tool primarily intended for processing data. It follows that the implementation is optimized for speed and does not allow the results to be displayed continuously. Animation of learning significantly slows down the computations, and so it is not included in the implementation. Therefore it is unsuitable as a tool for visualizing learning.

3.2.3 The DemoGNG applet

The tools described above did not provide graphical output during computations, which is of course a great disadvantage for representing the activity of a neural network. Therefore two German programmers decided to create an applet [5] that would provide graphical output during simulation. Using this applet I simulated, for example, the demonstration of the behavior of a Hard Competitive Learning network on a spiral (Figure 2.7 on page 14 of the original). DemoGNG enables the simulation of selected self-organizing networks, among them the neural gas and Hard Competitive Learning. The applet is very nicely designed and is a very good tool for visualizing the learning of neural networks.

One of the few disadvantages is the fact that in the so-called "teach" mode an iteration (one competition–adaptation cycle) is not divided into several phases. Several changes are thus displayed on the screen at once, and we can easily lose some of the connections.

3.3 Universal tools

3.3.1 Matlab

Matlab is a very good universal tool for any kind of computation. With each higher version its possibilities keep growing, and so I, too, used Matlab — especially in the early stages of my experiments, when I did not yet know about the DemoGNG applet. Matlab provides simple functions for plotting, and so the programmer need not worry about the graphical part; Matlab does it instead, at the cost of speed, of course. The team led by Kohonen implemented SOM Pack under Matlab, but because its goal is rather the resulting layout, it does not allow graphical output during simulation. Thanks to the universality of Matlab, however, this can be programmed, or the necessary libraries can be modified. Thanks to Matlab I obtained the results that can be seen in Figures 2.1, 2.3, 2.4, 2.5, and 2.6.

4 Design and Implementation of a New Visualization Tool

In the previous chapter we became acquainted with current tools that graphically represent some parts of the whole learning and recognition process of self-organizing networks. We showed that the tools can be divided into several categories based on their characteristic features. The first category comprises single-purpose batch programs, which are unsuitable for visualization purposes because they do not allow continuous graphical display of the important aspects of the individual iterations. The second category contains universal tools that allow work not only in the area of neural networks. A typical representative is Matlab, which is a very powerful tool for computation and modeling. Because it is a language primarily intended for computation, backed by support for a graphical interface, it allows us to create a tool fairly easily that can continuously display the required outputs. The graphical interface, however, is primarily intended for comfortable control and cannot be used to display fast changes, which is one of the properties we need for visualizing learning. The third category consists of simulation programs, which may take the form of applications or Java applets — for example, the DemoGNG applet, which is far more suitable for

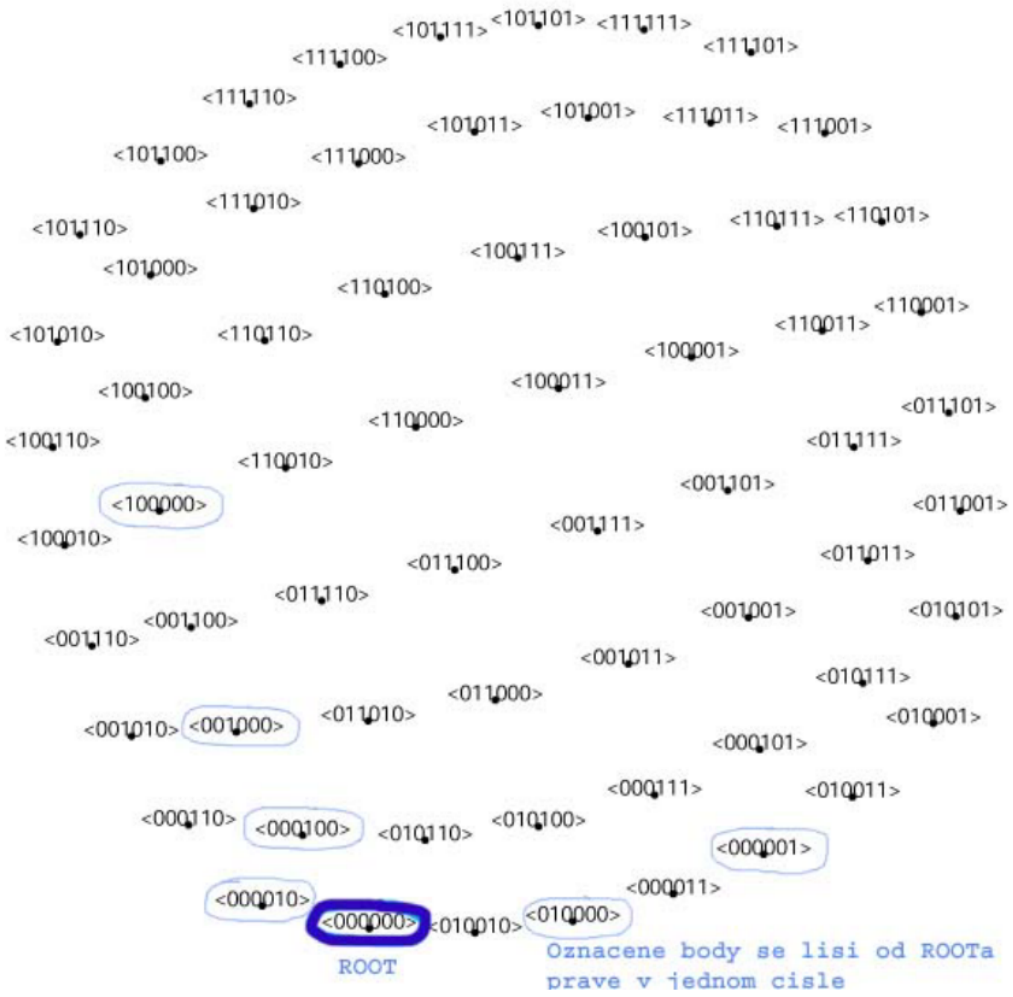

Figure 3.4. Vertices marked at distance 1 hop from a chosen vertex.

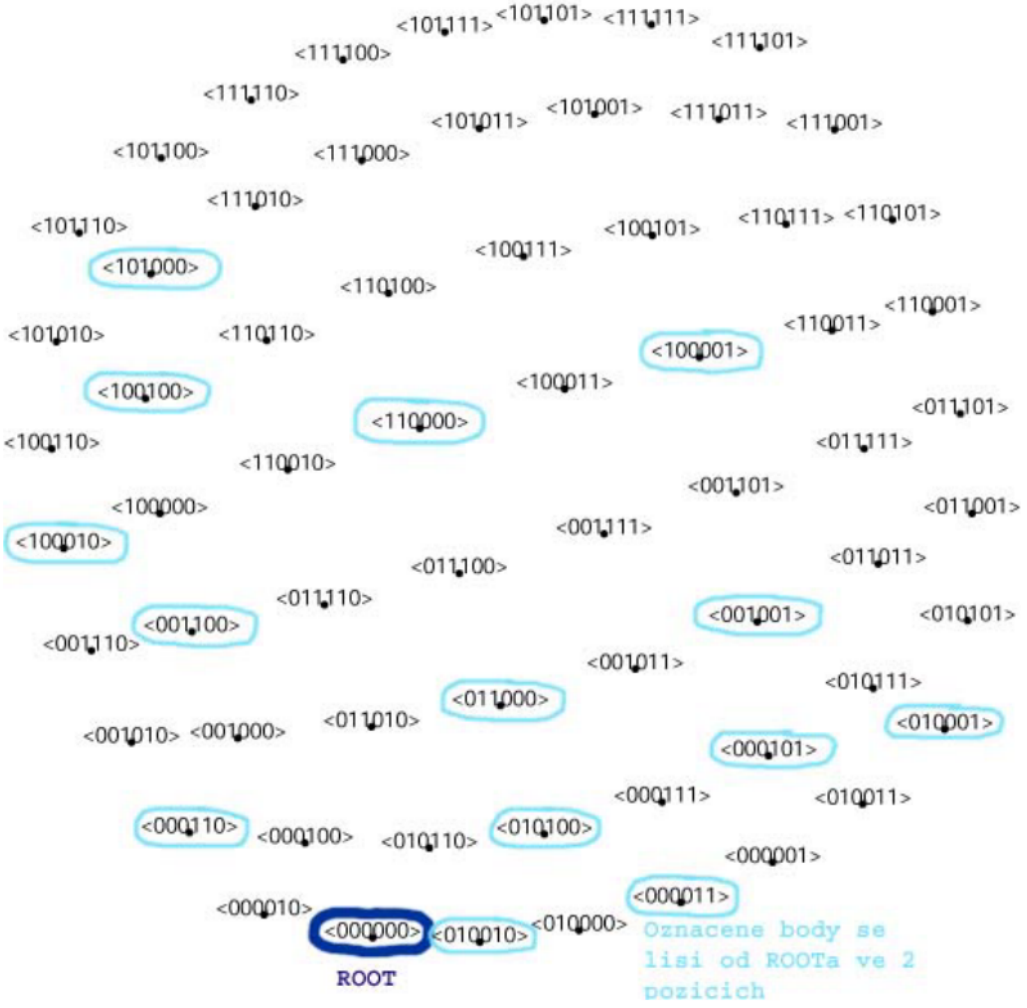

Figure 3.5. Vertices marked at distance 2 hops from a chosen vertex.

demonstrating the visualization of learning. In the previous chapter we also noted its disadvantages. Each iteration in this applet, composed of competition and adaptation, takes place at once, and only the result is displayed on the screen, so that we often cannot perceive the dynamics and the reasons for the iteration being performed. We therefore need to create a visualization tool that satisfies all the aspects necessary for a good visualization tool.

4.1 Formulation of goals

Any system being created, regardless of how complex it is, has its elementary purpose. The aim of my learning-visualization design is two paradigms of self-organizing networks: Hard Competitive Learning (page 6 of the original) and Dot Product SOM (page 14 of the original). The reasons for choosing these two paradigms are as follows. Methods based on Kohonen learning give very good results in the area of data preprocessing and finding relationships. Most applications implementing these paradigms do not allow a continuous animated approach. It is thus not possible to study in detail the dynamics of the whole model. Therefore one of the aims is the continuous animation of the processes, so that the dynamics of the whole model and its parts are evident.

With methods based on Kohonen learning, an additional problem arose in which some input samples had extreme values of some of their inputs, thereby significantly and negatively influencing the correct placement of neurons for the other input samples. The question was how this negative phenomenon could be suppressed.

One solution is the so-called zoom method. Having trained the network on the given input training set, we label, with the appropriate categories, those input samples that significantly influenced the learning of the whole network. Then we remove these input samples and train the network again. We again label the prominent input samples, and so we proceed to any desired level.

The second solution is based on finding a learning model that is in some sense independent of such extremes. One of the attempts is precisely the Dot Product SOM mentioned above, which is based on directional similarity and not on Euclidean similarity. The literature contains only a mention of the algorithm, not of all its properties; therefore the Dot Product SOM was chosen precisely so as to make it possible to ascertain its advantages and disadvantages, which may form the basis for further use of this paradigm. Moreover, I wanted to verify the correctness of the assumptions I proved in the Dot Product SOM chapter. So that the user would have the possibility of comparison, I chose as the reference model precisely the Hard Competitive Learning method, on which the principles based on Euclidean similarity are best and most clearly visible. On this method it is moreover possible to show the basic features common to self-organizing networks. Primarily, then, it is necessary to define all the problems and situations of the given paradigms that are important from the standpoint of explaining learning.

4.1.1 Definition of problems

On the Hard Competitive Learning method we can demonstrate several interesting problems and situations:

- **Demonstration of the basic activity of HCL on a simple example** — the simplest situation: there are 2 input samples and only one neuron in the output layer.
- **Illustration of the correct approximation of regions with increased density of occurrence.**

- **Dependence on correct initialization** — a typical situation connected with the principial weakness of the method.
- **Influence of the method of selecting input samples** — by selecting input samples with different methods (randomly, in a selected order, …) we may obtain different results.

For the Dot Product SOM method:

- **Demonstration of the basic activity of Dot Product SOM on a simple example** — the same as for HCL.
- **Dependence on the order of selecting input samples.**
- **The difference from HCL in the perception of data clusters.**

4.1.2 Visualization

The visualization of learning can be understood as the graphical representation of the causes and consequences of any changes in the states of the network. This is the basis from which we should proceed and from which further requirements follow. First of all, the tool must be able to simulate the required types of neural networks so that, at specified steps, current information about the state of the program and the network can be displayed. By "information" the following important aspects are meant:

- **What is being visualized** — it must always be clear what the given task concerns.
- **State and mode of the program** — the user should always have information about the mode and state the program is in.
- **A listing of the current parameters** — depends on the type of neural network used; for self-organizing networks this is usually the current time and the so-called learning coefficient mentioned in Chapter 2.
- **Information about the input samples** — in our case this will be a graphical representation of the placement of the input samples in 2D space. In the general case, however, graphical representation cannot be carried out, especially for multidimensional data; then it is possible either to list values or to preprocess the input data, for example using the Sammon projection.
- **Information about the hidden and output neurons** — similarly to the input samples. The program should moreover be able to display the layers for different types of neural networks independently and at once. We thereby make it possible to observe better the differences in the recognition and learning of the individual paradigms.
- **An apt description of the essence of the current processes** — during any operation the user should know what is happening and, above all, why it is happening. Describing the cause is often underestimated and is thus a common error encountered in describing any kind of problem.
- **Supplementary information:**
  - *Legends for the graphical objects* — the user should be able to obtain at least basic information about each displayed object.
  - *Axes and scales* — a seemingly obvious and also often underestimated matter. The author certainly knows all the details of the simulated network, yet the user may be at a loss without basic information about the axes and scale.

4.1.3 Simulation

One of the pillars of the sought visualization tool is undoubtedly simulation. It would make no sense to display the states of a system without being able to simulate it. Simulation can be divided, by

the manner of simulation, into synchronous and asynchronous. The implementation of synchronous simulation is fairly simple and will fully suffice for our purposes; moreover, it better corresponds to the description of processes in models of self-organizing neural networks. For synchronous simulation it is necessary to divide the iteration cycle of the neural network itself into elementary atomic operations. It is also important to bear in mind the requirement that it be possible to simulate several types of neural networks "in parallel." One of the goals of the program is, after all, to represent the differences between the individual paradigms on the same training set. When demonstrating the problems mentioned so far, the individual animations of the system will be graphically represented in each iteration. Because the learning process often consists of thousands to tens of thousands of iterations, the user would have to wait too long for the final placement. It is therefore good to introduce two basic modes of simulation:

- **1. Slow mode** — i.e., animation mode. If we choose this mode, the simulation program will animate all the dynamics of the system in detail.
- **2. Fast mode** — some animation elements will not be displayed. The aim of this mode is the fastest possible convergence to the final solution, so as to display the final placement of the neurons of the individual layers.

We divide the states of both paradigms in slow mode as follows:

- **Hard Competitive Learning**
  - Initial state
  - Selection of the input sample — we highlight the input sample.
  - Selection of the winning neuron — we highlight the neuron that won the competition.
  - Adaptation of the winning neuron — we animate the movement of the winning neuron.
- **Dot Product SOM**
  - Initial state
  - Selection of the input sample — we highlight the input sample.
  - Selection of the winning neuron — we highlight the neuron that won the competition.
  - Adaptation of the winning neuron — we animate the movement of the winning neuron. Because the geometric construction of the new position is more complex than for HCL, we divide this operation into several partial parts that correspond to the gradual composition of vectors according to relation (2.21) on page 19 of the original:
    - drawing the vector characterizing the neuron with weights m;
    - adding the vector x of the input image to the vector m;
    - shortening the added vector x by the proportional coefficient α;
    - drawing the new vector obtained by adding the shortened vector x to the vector m;
    - normalizing the new drawn vector;
    - moving the selected neuron to the new position determined by the normalized vector.

In fast mode it then suffices to display only the final state of each iteration. Because drawing is, after all, time-consuming, it is possible to introduce a parameter whose value determines after how many iterations the current state is to be displayed. In this way we achieve even greater simulation speed.

4.1.4 Controls

Suitable interaction with the user is also an essential part of every program. We should give the user the possibility of intervening in the processes so that he can change parameters, pause the animation, and change the speed according to his needs — because those who have already

encountered animated paradigms will prefer faster animation, whereas those just beginning with the given paradigms may have problems understanding some of the processes. Another important matter is providing a sufficient number of training examples that show all the interesting properties of the individual paradigms. We should give the user the possibility of choosing the individual paradigms, including the choice of simulation mode.

4.2 Choice of programming environment

Today there are quite a number of programming tools that allow implementation [of such applications], and the tool should not be platform-dependent. JAVA is very suitable for this purpose: it is platform-independent and moreover has a fairly uniform graphical and user interface. In combination with the WWW environment it creates a balanced means for building applications that can be integrated into larger, interlocking wholes accessible directly on the Internet.

4.3 Program structure design and implementation

A good program design is a perennial puzzle for most programmers. Almost everyone who has ever sat down at a keyboard and tried to write a program from scratch has found that this is often not the ideal solution. Having an idea is a good thing, but without proper analysis one can easily get lost in a tangle of swarming problems. Many books have been written about application design, but a universal one of course does not exist, since different programming tools provide different possibilities. I therefore started from the premise that the program is to be written in the JAVA language. It follows that the design will take the form of object-oriented analysis.

4.3.1 Highest-level data flow diagram

To begin with, it is appropriate to start with a simple data flow diagram (Figure 4.1). In this figure we can see a characteristic situation. The user watches the graphical interface and tries to influence the running of the program via the control panel. On the basis of user events, the control panel triggers actions that influence both the graphical interface and the simulation activity itself.

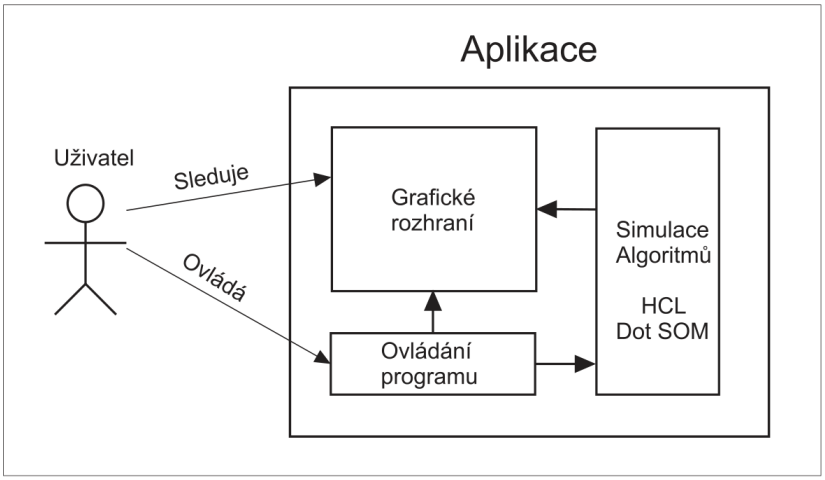

Figure 4.1. Simple Data Flow Diagram.

It follows that the application will be divided into these parts:

- Graphical interface
- Controls
- Simulation of algorithms

It is also very advantageous to use the means offered by the WWW environment. We can thus integrate all the necessary information about the applet and the problems addressed into WWW pages so that they form a coherent learning text. Therefore one of the subsections is devoted precisely to the presentation on the WWW.

4.3.2 Representation of the neural network

A correct design is one of the cornerstones of the whole application. In the design we start from the recognition and learning algorithms of the individual paradigms described in the theoretical part. Hard Competitive Learning is described by relations (2.4) and (2.5), and Dot Product SOM by relations (2.8) and (2.9). We thus have a sufficient mathematical description that just needs to be translated into the JAVA language. Further requirements are placed on the design, however, which we largely defined in the Simulation chapter (page 34 of the original). One requirement is the time division of the individual iterations into atomic operations. The reason is the effort to graphically display the individual stages of an iteration. The method that handles the graphical display — the `paint()` method — must know exactly what it is to draw (which object to highlight, conversely which element to suppress) and hence in which part of the iteration the algorithm is. Here two solutions present themselves for telling the `paint()` method exactly what to draw.

The first option is the so-called state variable. This variable stores a value indicating which part of the iteration we are in. In the method handling the graphical interface it then suffices to implement the display for the individual states. During the algorithm the state variable will thus change and be passed to the graphical method.

The second option is the so-called action flags. In slow-animation mode it will always be a matter of highlighting or, conversely, hiding a given object so as to represent the process as much as possible. Therefore it is possible to introduce flags for the required objects that will say whether a given object is active or passive. This is best demonstrated on a simple example. On the screen we have a set of input data. At the beginning of an iteration all the input data are displayed identically. If we want to emphasize which input sample we have selected, then it must be highlighted; otherwise the user can hardly recognize which input sample was added. Here the solution presents itself of remembering the state for each input sample. If it is necessary to highlight the selected input sample, it then suffices to change the state variable of that input sample to "active," and the display method `paint()` will, when drawing, know which element to highlight. In this case the number of necessary variables can also be reduced. If, for example, we know that there will be only one active input sample, it suffices to remember its index in a single variable, and we need not have a state variable for each one.

The first variant has the advantage that we need not maintain any flags and thereby save the need to introduce further variables. The disadvantage, however, is the rather complex conditions in the `paint()` method. The second variant mentioned provides far greater comfort in controlling the drawing of the individual objects, so that for any request to change the drawing it suffices to change the flag settings. It is also possible to use both options at once and thus take advantage of both.

An interesting observation is that the Hard Competitive Learning and Dot Product SOM methods

have very similar algorithms. They essentially differ only in the scoring and adaptation functions; otherwise the structure of the algorithm is the same. It is therefore good to exploit this property and design a common abstract class `Nnet`, which will define common methods for the derived classes `NnetHCL` and `NnetDot` — Figure 4.2.

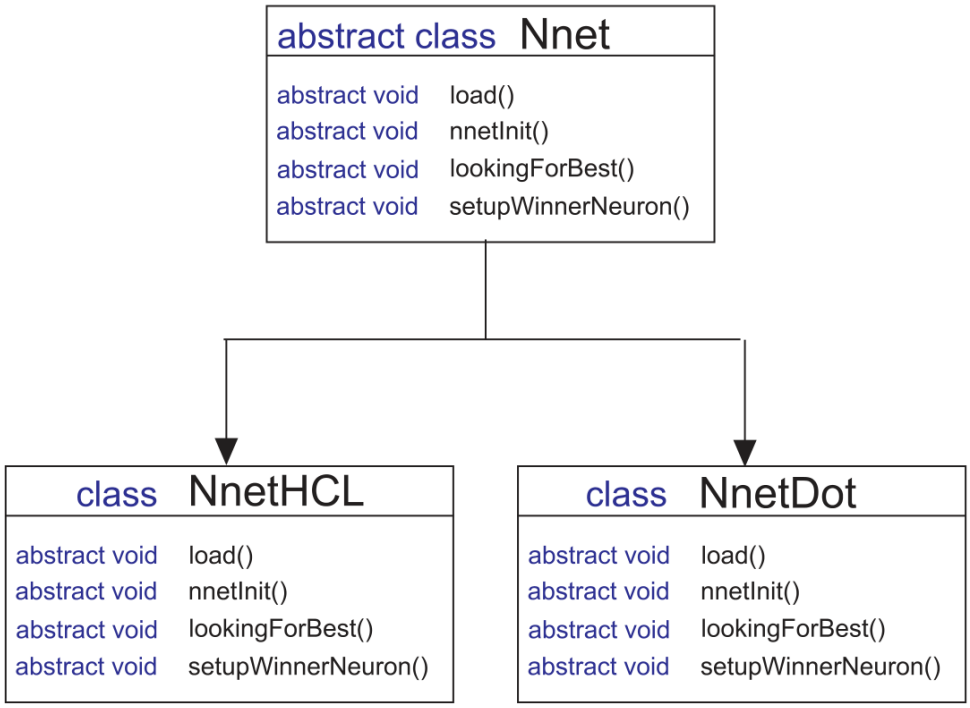

Figure 4.2. Classes defining and implementing parts of the HCL and Dot Product SOM algorithms.

In Figure 4.2 we can see, in each class, 4 methods:

- The **`load()`** method serves to load the given layer of neurons into the computer's memory. To simplify extensibility, this method is defined separately and is not implemented directly in the **`nnetInit`** method. The **`load()`** method can be overloaded as needed; for example, loading data from files or from other inputs can easily be implemented and then called with the appropriate parameters.
- The **`nnetInit()`** method initializes all the necessary objects and, using the **`load()`** method, loads the given layer of neurons into memory.
- The **`lookingForBest()`** method carries out the competition of the winning neuron. For each neuron the value of the scoring function is determined, and on the basis of the results the best neuron is determined.
- The **`setupWinnerNeuron()`** method computes the new position of the winning neuron.

4.3.3 Layers of neural networks

Now it is necessary to define a layer of neurons for each `Nnet` object. In general there may be several types of layers of any objects. The advantage is the fact that in our problem only three very similar types of layers occur, namely:

- input samples
- output neurons of the HCL type
- output neurons of the Dot Product SOM type

Each element of both these types has the same dimension. We exploit this to advantage in the design as well. First we design an abstract class `Layer`, which will be common to all derived classes `LayerInput`, `LayerHCL`, and `LayerDot`. `LayerInput` will define the layer of input samples, `LayerHCL` the layer of HCL neurons, and `LayerDot` the layer of Dot Product SOM neurons. Everything is shown in Figure 4.3.

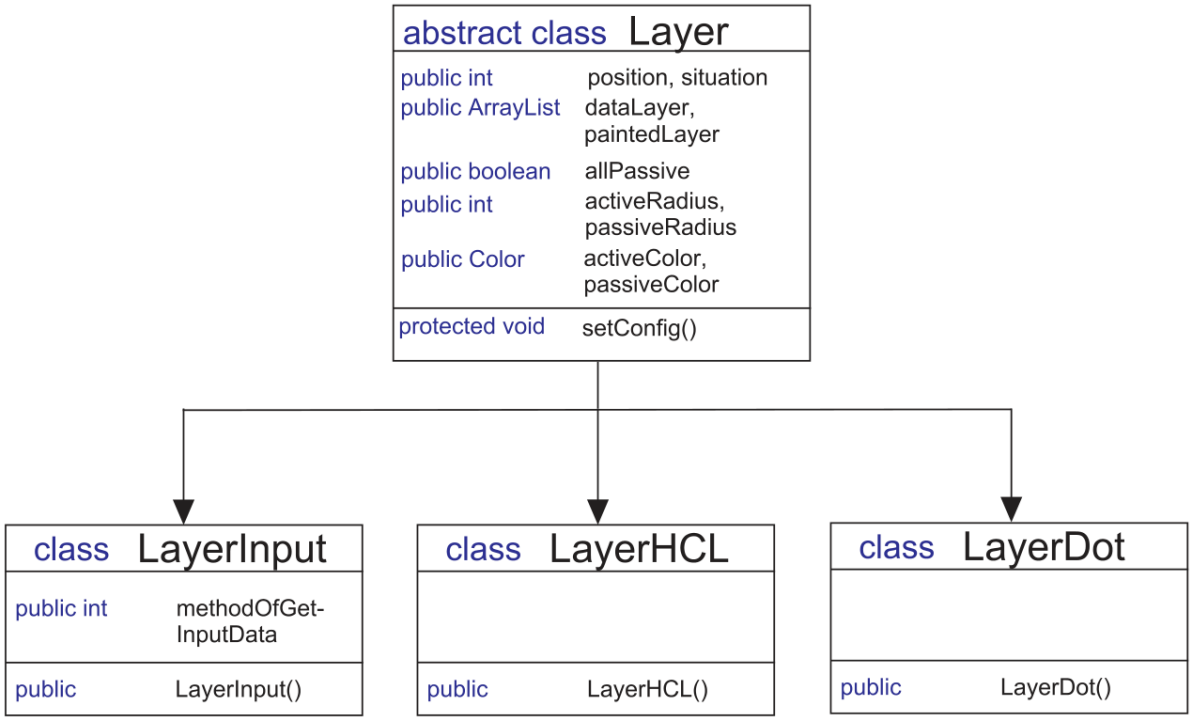

Figure 4.3. Classes defining and implementing the individual data layers of neural networks.

Now we explain the meaning of the individual fields and methods:

- The **`position`** field defines which element of the layer is active. In the input layer it points to the position of the selected input sample; in both types of neural network it points to the winning neuron.
- The **`situation`** field determines in which phase of the iteration the given layer is. According to the type of phase, both the activity of the algorithm is controlled and, in the animation, certain parts are highlighted.
- The **`dataLayer`** instance is a set of vectors. The vectors characterize either the values of the input samples or the values of the output neurons of both layers.
- The **`paintedLayer`** instance is a set of vectors obtained by recomputing the vectors of `dataLayer` so that they can be displayed on the graphical output. In general the input values may take any values, so it is necessary to recompute all the values of the input vectors so that they can be drawn appropriately. Here two options present themselves: either recompute these values from the `dataLayer` set on every drawing (saving the `paintedLayer` instance, which has the same number of elements), or recompute these values at the beginning

and, on any change of the `dataLayer` instance, propagate that change to the `paintedLayer` instance. The second option has the great advantage of saving the overhead of recomputing all objects on every redraw, and is therefore the one implemented.

- The `allPassive` field means that no object is active. In some phases of the algorithm's iteration no object of the layer is active (at the beginning of each iteration). The `position` field, however, always has some defined value; for this reason we must introduce a correcting variable that "switches off" the activity of an object.
- The `activeRadius`, `activeColor` fields define the appearance of the highlighted active element. In view of the possibility of extending the application to several neural networks (even of the same type), we must define a different appearance for each layer; otherwise we would not be able to tell which object belongs to what. Therefore information about the appearance of all types of objects of a given layer is also included in the layer definition. The activity of an object is uniquely given by the combination of the `position` and `allPassive` fields.
- The `passiveRadius`, `passiveColor` fields define the appearance of inactive objects.

The `Layer` class is designed with extensibility in mind. If we want to add another type of layer, it suffices to create a class derived from `Layer` and additionally define any unique parameters of the layer. Even in the case where the `setupWinnerNeuron()` method is based on adjusting a set of neurons (Kohonen maps, …) and not only the winner, it is possible to extend this layer simply by adding another `ArrayList` instance that will define the order of the adjusted objects, or the objects themselves. Thanks to this design, implementing multilayer networks as well is not a problem.

4.3.4 Algorithms of neural networks

We have a prepared implementation of all parts of the neural networks, and it remains to bring these networks to life — that is, to design a class (or classes) whose methods control and carry out the activity of the neural networks' learning. Again we exploit the similarity of the learning methods of the two networks and create a common abstract class `ExecuteNnet`, from which we derive the classes `ExecuteNnetHCL` and `ExecuteNnetDot` implementing the control of the activity for the individual networks. In addition, the `ExecuteNnetAll` method implements the simulation of the algorithm for both neural networks at once.

The `ExecuteNnet` class contains the common elements for all derived classes. The basis is two abstract methods, `initNnet` and `selectPartOfIteration`, which are unique for the individual computation implementations. In the `ExecuteNnet` class we can also define the input layer of reference samples `layerInput`, because in every model there is exactly one such layer. Further methods are implemented in the abstract class which, for the sake of clarity, are not shown in Figure 4.4. These are mostly auxiliary methods for creating the input layer of samples, namely:

- The `AddInputVector(double x, double y)` method serves to insert a vector into the layer of input samples.
- The `fetchFullBox(double s1, double s2, double r1, double r2, int countOfPoints)` method generates a region of input vectors bounded by a rectangle whose center is determined by the coordinates $[s_1, s_2]$ and whose dimensions are r_1, r_2 from the center. The vectors have constant density and their number is regulated by the `countOfPoints` parameter. The generated vectors are inserted into the input layer via `AddInputVector(double x, double y)`.
- The `fetchFullCircle(double s1, double s2, double radius, int countOfPoints)` method generates a region bounded by a circle with center $[s_1, s_2]$ and radius r. The vectors

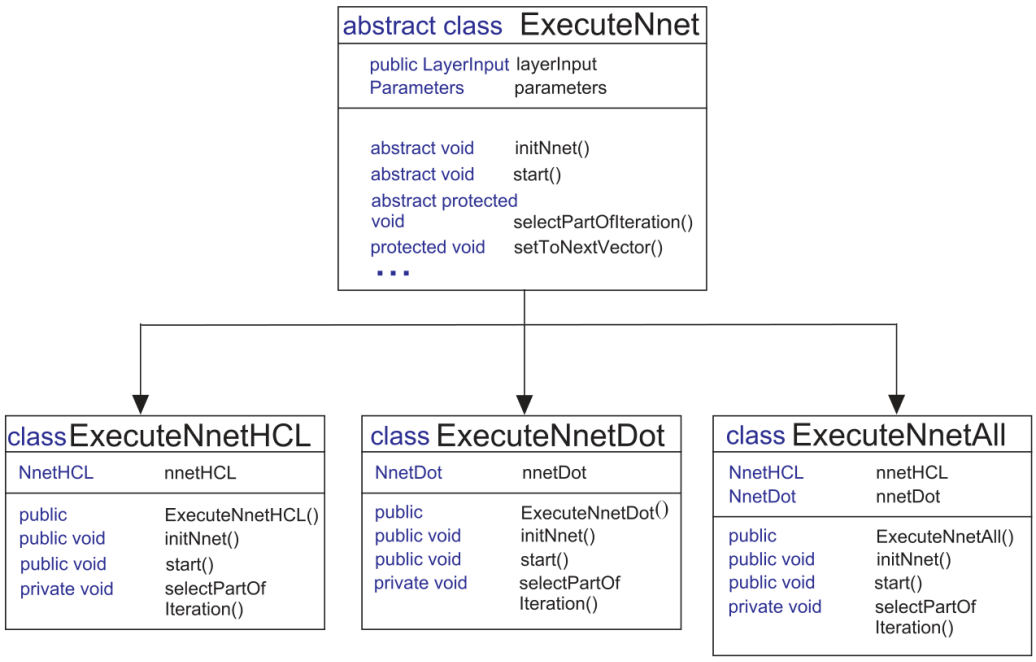

Figure 4.4. Classes that control the computation algorithms themselves.

have constant density and their number is regulated by `countOfPoints`. The generated vectors are again inserted into the input layer via `AddInputVector(double x, double y)`.

- The methods **`fetchData1()`** through **`fetchData7()`** serve to load the input data for the various example cases.
- The **`loadData()`** method calls and determines which of the methods `fetchData1()` through `fetchData7()` will be called. It decides according to the `setOfInputData` parameter, found in the `Parameters` class. This value changes when the corresponding menu in the user controls is changed.

The derived methods `ExecuteNnetHCL`, `ExecuteNnetDot`, and `ExecuteNnetAll` establish, in their constructors, the appropriate objects implementing the algorithm for the given type of neural network. Because this is the core of the whole system, for better understanding it is appropriate to give, as a source-code excerpt, the `ExecuteNnetHCL` class, which implements the algorithm for computing the Hard Competitive Learning network. *(The source-code excerpt is in an appendix of the original.)*

4.3.5 Global parameters

The running of the algorithms can also be influenced by parameters common to all implemented types of neural networks. It is advantageous to group these parameters into a single class, so that we can pass all the parameters via a pointer to an instance and need not pass each parameter separately. In the applet implementation they can be found in the `parameters` instance, which is of course of type `Parameters`. Because these parameters influence the operation of the whole algorithm, I give here at least the most important ones:

- **`public int timeActual`** — gives the number of iterations since the start of running the neural-network algorithm. The value increases by 1 on each new selection of an input sample.

From this value the so-called learning coefficient α (also denoted $h_c(t)$) is also derived.

- `public int timeOfFinish` — the number of iterations at which the computation ends.
- `public int modeOfExecution` — determines which type of simulation is to be started. In the current implementation it is possible to start 3 kinds of simulation: Hard Competitive Learning (`modeOfExecution = 0`), Dot Product SOM (`modeOfExecution = 1`), or both at once (`modeOfExecution = 2`).
- `public LayerInput layerInput` — creates the instance of input data. In the classes derived from `ExecuteNnet`, a pointer to this input-layer instance is passed in the constructors, thereby allowing the algorithm to work directly with the given input samples.
- `public ArrayList allLayers` — a list of output (in general, any) layers. As soon as any layer representing neurons is created, a reference to it is added to the `allLayers` list. The `paint` method then has references to all layers and, when redrawing, traverses this list and draws them all. Thanks to this solution, any number of layers can be drawn universally.
- `public ArrayList allLayersInfo` — information about the layers. Since various types of layers can in general be inserted into `allLayers`, a mechanism is needed that uniquely identifies these layers when retrieving them; into `allLayersInfo` we must store information about the type of layer.
- `public boolean fastMode` — controls the manner of animation. The applet can work in a "fast mode" (value `true`) and a "slow mode" (value `false`). In fast mode all objects are drawn passively and all supporting graphical elements serving to visualize learning are turned off. Slow mode, conversely, draws in detail all the particulars necessary for an apt representation of learning.
- `public int setOfInputData` — gives the type of input samples to be used at initialization. The applet has 7 predefined types of input-sample layouts for the example cases.
- `public int setOfHCLData` — determines the type of HCL neuron layout to be used at initialization of this network. The applet has 5 predefined types of HCL neuron layouts for the example cases.
- `public int setOfDotData` — determines the type of Dot Product SOM neuron layout to be used at initialization of the Dot Product SOM network. The applet has 5 predefined types.

For operations with the time `timeActual` we introduce several methods with the following functions:

- The `incrementTime()` method increases `timeActual` by 1 on each call.
- The `getFinishStatus()` method returns `true` when `timeActual` reaches `timeOfFinish`; in that case it is a signal to end the simulation. Otherwise it returns `false` and the computation may continue.
- The `setTime(...)` method, called with the appropriate parameters, sets `timeActual` and `timeOfFinish` to the required values.

Now it remains only to create a `RunNnet` class that will, on the basis of the `modeOfExecution` parameter, start the correct type of simulation. The whole simulation must run in a separate thread so that we can control the application via the menu in parallel. It is therefore good to implement the `RunNnet` class as `Runnable`. In this class it suffices to implement one method and define a custom constructor. In the `RunNnet()` constructor, on the basis of the `modeOfExecution` parameter, we create the necessary objects of type `ExecuteNnetHCL`, `ExecuteNnetDot`, or `ExecuteNnetAll`. In the `run()` method we then start the `start()` method of the appropriate object, thereby handing control over to the simulation.

4.3.6 GUI

Even the best program, without a good graphical interface and controls, has only half the effectiveness. This is especially true of applications meant to visualize simulations of some models. We divide the interaction with the user into two parts: graphical output and controls. The graphical output must contain all the important information but should remain clear. In our case the graphical output can be further divided into three components:

- **(1) Animation** — in this area the visualization of the learning of the neural networks is displayed.
- **(2) Information panel** — this section provides the user with all the necessary information about the state of the program and of the individual displayed objects.
- **(3) Control menu** — allows control and influencing of the applet's activity.

Figure 4.5. Basic layout of the individual parts of the graphical output.

The whole animation is handled by the `MainGraphicsPanel` class. It contains two public methods, `void paintComponent()` and `void setNewParameters(...)`.

We call `void setNewParameters(...)` when changing the type of neural network, changing any layers, or restarting. It ensures that references pointing to the new parameters are passed on any change of data or types of neural networks. Therefore it does not suffice to put this initialization in the constructor, since we would call it only once, at the applet's startup. The `void paintComponent` method is called whenever we call `repaint()`. The `repaint` method can be called from several classes, so two implementation options present themselves here. Either we pass each object a pointer to the `MainGraphicsPanel` class, so that each object obtains a reference to the `repaint` method; or we exploit the fact that each object is already passed a reference to a `Parameters` object, and include the reference to the graphical panel of type `MainGraphicsPanel` in the `Parameters` object, which gathers all important global parameters. In this variant we pass only one parameter at object initialization, so this variant is better. Into the `MainGraphicsPanel` class we also put auxiliary methods used by `paintComponent`, which are only private:

- `private void paintTextBox(...)` draws the information box where all the essential data concerning the simulation are displayed.

- `paintHCLInfo(...)` writes, into the information box, information about the simulation of the Hard Competitive Learning network itself. On the basis of the current iteration phase, in slow mode the essential features of the given phase are described.
- `paintDotInfo(Graphics g, LayerInput layerInput, LayerDot layerDot)` fulfills the same function as `paintHCLInfo`, but for the Dot Product SOM network.
- `paintAllInfo(...)` — used when we simulate both neural networks at once. It is not possible to display using `paintHCLInfo()` and `paintDotInfo()` at once, because both methods write some data at the same positions. Therefore the listing must be rearranged and defined in a new method. When creating a new type of simulation, we must define a new method that will list the required data according to our needs.
- `private void paintVector(...)` graphically represents a vector arrow with its start at point $[X_1, Y_1]$, its end at point $[X_2, Y_2]$, and color `myColor`.
- `private void paintInputDataComponent(...)` displays the input layer of reference samples `layerInput`.
- `paintHCLNeuronComponent(...)` displays a neuron layer of the Hard Competitive Learning type. The methods are designed so that several layers of neural networks of the same type can be displayed independently of one another. Thus this method can be called repeatedly with different `layerHCL` references and thereby draw several layers on top of one another. For this purpose there is the main loop in the `repaintComponent` method, which traverses all references to layers (stored in the `parameter.allLayers` list) and, according to type, repeatedly calls the methods `paintHCLNeuronComponent(...)` and `paintDotNeuronComponent(...)` with the obtained references to the individual layers. Correct recognition of the layer type is ensured by the `parameter.allLayersInfo` parameter, which stores information about the layer type. Because the current implementation supports 3 types of simulation (HCL, Dot Product SOM, and both at once), the choice of methods is made on the basis of the `parameter.modeOfExecution` parameter.
- `paintDescriptionAxes(...)` displays the axes with the appropriate labels, using the auxiliary methods `paintXMark()` and `paintYMark()`.

The control panel should provide the user with comfortable and clear control. The design of the individual parts of the control panel derives from the basic requirements on the applet. The user should be able to influence the applet's processes so that he can choose the individual parameters or change the animation speed as needed. The controls and appearance of the main window, including the components, are implemented in the main class `MainProgram`. The controls can be divided into the following components:

- **Pause/start the simulation.** The user should be able to pause the simulation at any time. We implement pausing/starting via the button `JButton startStopButton`.
- **Restart the simulation** — we do not wait for the current simulation to finish; rather, by setting the parameter `parameter.stop` to the value 2 we give the thread processing the simulation the command to terminate, and in addition we send the thread an interrupt signal, since it may have been in a wait state called during the animation process. As soon as the thread terminates, new values are initialized via the method `private void initAnimationVariables(...)`.
- **Switching between fast and slow mode** is implemented via a `JCheckBox fastModeChBox` instance; on the given event the value `parameter.fastMode` is changed to `true`/`false` as required.
- **Switching between selecting predefined examples and combining the individual configurations.** The selection is provided by instances of type `JComboBox`. Predefined

example types can be selected via the `setOfExample` instance, and the individual data configurations via the instances `setOfInputData`, `setOfHCLData`, or `setOfDotData`. If a new data set needs to be added to one of the instances, it suffices to call the `add` method on the given instance with the name as parameter (e.g. `setOfInputData.add("Circle 10")`).

- **Changing the animation speed** is also a necessary condition for comfortable control of the applet. The simplest solution is to use a suitable component of type `JSlider`, designed precisely for this. When an event is generated, one of its methods returns the relative position of the slider, which we must recompute and store in the variable `parameter.timeWait`. During animation, the thread then waits exactly this time (in milliseconds) between individual frames, so that the dynamics of the system is slower or faster according to the value of `parameter.timeWait`.

Without defining events for the individual components, no events will be generated; therefore we must implement these events. In the VisualNnet applet two types are defined: `actionListener` and `SlideListener`. `SlideListener` serves to control the animation speed; the other events are defined in the `myEventListener` method, which is of type `actionListener`. When the `myEventListener` method is called, it determines which component triggered the event and, on that basis, performs the given actions.

4.4 Testing

4.4.1 Goal of testing

It is said that in every program there is at least one error. Since we are by nature fallible, errors often occur in our programs. Some errors can be detected immediately; others manifest themselves only after several days of agonizing over the program. Of course every author tries to test and tune his program so that it contains no errors, but the problem is often that the author is biased toward his program. Subconsciously he is governed by his knowledge of the program's structure, and so he often focuses on problems where there is in fact no problem. For this reason it is better if the testing is carried out by another person. Therefore I asked the tester, Ing. Jan Ingerle, to test my applet. The aim of the tests was three basic parameters:

- 1. comprehensibility
- 2. correct functionality
- 3. stability

4.4.2 Testing procedure

First the VisualNnet applet was tested for comprehensibility. The applet was shown to the tester without any explanation, and his task was to try out and guess what each control component means. He then tried to describe, in his own words, the meaning of the individual animated objects. The tester described all the objects without any problems. Then the tester was given several tasks meant to show how difficult it is to move around in the given environment and whether the goal would always be reached. The tester was presented with the following tasks:

- **Task 1:** Familiarize yourself with the basic information from the WWW presentation.
- **Task 2:** Start the simulation of the 3rd example and try out the individual simulation modes — stopping the simulation, restarting the simulation, fast mode, slow mode, changing the animation speed, changing the mode of selecting input samples. Repeat these actions several

times and observe whether the application responds appropriately. Then increase the speed to maximum and observe whether the simulation runs to the end.

- **Task 3:** Turn off the option of selecting example cases and set up a simulation enabling the display of both paradigms at once. Select the input set `data3`, the HCL neuron layer `HCL2`, and the layer `Dot 2`. Try out all possible modes, similarly to Task 2.
- **Task 4:** Switch between the individual selections of the layouts of the individual layers and observe whether the application behaves stably. At the same time, go through all possible modes, similarly to points 2 and 3.

4.4.3 Contribution and conclusions from testing

During testing, the first two requirements (comprehensibility and correct functionality) were met without problems, and no defects or remarks were found. A problem arose during the testing of stability. The tester recorded, on repeated restarting of the applet and selection of the individual sets, an exception when working with the `parameter.allLayers` list of type `ArrayList`. After detailed examination, the following synchronization error was found. During the simulation, the algorithm itself calls, after each phase, the `repaint()` method, which ensures the correct display of all objects. When restarting the application (and thus also selecting new layers), the main program sends a signal (via the `parameter.stop` variable) commanding the thread to terminate. If the `repaint()` method is called, the simulation thread continues and does not wait for the drawing by `repaint` to finish. If the simulation thread receives the termination signal, the thread terminates. The drawing itself, however, is time-consuming and may take far longer than the termination of the simulation thread. Because, when starting a new simulation thread (including the initialization of variables), this possibility had not been allowed for, it happened that the simulation thread — including the initialization (nulling) of all lists — ran before the animation thread finished. In that case the thread requested data from lists that had already been reset to the initial value `null`, and therefore the given exception was raised. Thanks to the conscientious work of the tester, Ing. Jan Ingerle, an error was thus revealed that occurred only occasionally and was therefore harder to localize. After fixing this error the application was already stable.

The design and implementation of any product in the vast majority of cases contains errors, and here too it turned out that testing is an absolutely indispensable part of the development of any type of program, and it thus helped to increase the quality and reliability of the resulting product.

4.5 WWW presentation

We exploit, to advantage, the fact that the applet can be placed directly on a WWW page. We can thus provide the user with complete information concerning the given topic. The structure can be divided into several subtopics:

- **Introductory information** should introduce the user to the problem — that is, why this project came into being, what can be found on these pages, and what contribution the VisualNnet demonstration applet has.
- **Theoretical part** — we should introduce the user to the basics of the individual paradigms.
- **Description of functions and reference manual** — a list and description of the applet's activities.
- **User guide** — a manual for using the applet. We acquaint the user with the prepared example cases. Optionally we can add a quiz concerning the basic properties of the individual paradigms. The quiz should be designed so that the user can find the answers precisely by

means of the simulations that the VisualNnet applet enables.

4.6 User documentation

By means of the user documentation the user usually becomes acquainted with the system for the first time. The user documentation should provide a precise idea of how the application works. The user should not be forced to study the whole documentation if he wants to use simple functions. The documentation should be structured so that the user can study precisely only what he needs for his work. The WWW presentation can also be considered user documentation, and therefore its structure is identical to the structure of the WWW presentation.

5 Using the Proposed Tool

The VisualNnet applet is, by its nature, more an educational application than a tool used for scientific computation. Its purpose is to visualize the learning of a neural network and thereby to convert the simulation of a given model into a more easily imaginable and comprehensible form. If we are to maintain this philosophy, we should provide the user with example demonstration cases that guide the user and practically illustrate the essential features and problems of the individual paradigms.

5.1 Proposing example cases

The introductory examples should be truly simple. The aim is to focus on the basic idea of the individual paradigms and then gradually move on to examples that show the characteristic properties of the individual paradigms. The conclusion should show the differences when comparing the individual paradigms.

- **Example 1 — Introduction to Hard Competitive Learning.** In this example we acquaint the user with the basic principle of competition and adaptation of an HCL network. For this we use only two input samples and one output neuron. The user will observe how the neuron's position changes on the basis of the selection of the input sample. The user thus has the opportunity to see that at first the neuron moves all the way to the position of the selected input sample, and that over time the change in the neuron's position keeps decreasing.
- **Example 2 — Introduction to Dot Product SOM.** This example has the same aim as the previous one, with the difference that the demonstrated network will be of the Dot Product SOM type. The user will thus have the opportunity to understand the derivation of the position on the basis of graphical illustrations of vector composition.
- **Example 3 — The influence of poor initialization on learning in HCL.** In the second part of the thesis some disadvantages of HCL are given. One of these disadvantages is the influence of the initial setting of the neurons' positions on the final placement. It is therefore important that we make this phenomenon sufficiently clear too. The example illustrates 2 isolated regions of input samples and 6 HCL neurons. The neurons are deliberately placed so that one of them grasps one isolated region and the others cannot reach it. This is guaranteed precisely when the other neurons are farther from the nearest point in the observed isolated region than the diameter of that isolated region. Once a "selfish" neuron has grasped a given isolated region, the distance of this neuron from all input samples will be at most the diameter of the given group. Therefore, if we want to achieve the effect of isolated regions, we must place the other neurons so that their distance is greater than the mentioned diameter.

- **Example 4 — Suitable initialization in HCL learning.** One way of suppressing the influence of the previous example is to suitably place the initial positions of the neurons. We thereby achieve that the neurons spread evenly in both isolated regions.
- **Example 5 — The influence of the input-sample selection method on HCL learning.** One way of partially suppressing the influence of neuron initialization is to change the method of selecting input samples. Random selection of input samples for the competition turns out best, and therefore one exercise should be experimenting with the influence of selection methods. The choice of input samples does influence the resulting learning of the network, but with poor initialization it helps only exceptionally. In the example, the input samples are placed on a circle centered at $[0, 0]$. The input samples are selected by going around the circle in sequence; the winning neuron is thus dragged behind the input sample. If a second neuron is farther from the nearest input sample than the radius of the circle, this neuron has no chance of winning the competition under the given ordered selection. This influence is suppressed by random selection, because there is a high probability that the first neuron will sometimes be on the opposite side from the second neuron, and when selecting an input sample close to the second neuron (closer than the diameter of the circle), it wins the competition and thus joins the learning process.
- **Example 6 — The influence of the input-sample selection method on Dot Product SOM learning.** It turns out that a similar problem of selection influence can be found in the Dot Product SOM network too. A simple illustration with two Dot Product SOM neurons is based on the fact that, under ordered selection, the competition always selects the input sample closest to the neuron that has won the competition so far. The input reference set of samples is formed by a circle, and the competition proceeds by gradually selecting the nearest samples.
- **Example 7 — Four isolated regions** provides a practical verification of the example presented in Figure 2.1 on page 8 of the original.
- **Example 8 — Different conception of clusters after learning in HCL and Dot Product SOM.** One of the primary reasons the Dot Product SOM algorithm was created is the suppression of the influence of extreme values in some components of the input samples. Therefore, between the two paradigms, the manner of understanding the similarity of two vectors in space is completely different. We therefore create an example in which this difference is clear. First we create 3 clusters of input vectors and place them so that their centers lie on a single line passing through the center of the coordinate system. Then we create another triple satisfying the same conditions, but the angle enclosed by the two imaginary lines should be at least 130 degrees. We then place both neuron layers (HCL and Dot Product SOM). The whole example is thus ready for simulation. During the simulation, each HCL neuron will try to grasp one cluster and place itself in its center. Conversely, a Dot Product SOM neuron will try to orient itself in the direction of the grouping of the triple of clusters and thus grasp the whole triple, located in a cone emanating from the center of the coordinate system. In this way we best demonstrate to the user the difference in the conception of similarity.

6 Conclusion

Neural networks are a complex and today already very extensive area. In the middle of the 20th century, when this scientific field arose, it was predicted that in the near future we would be able to model human thought. But despite great technical and scientific progress, we still do not have this goal within reach.

In this work we became acquainted with some types of self-organizing neural networks. We showed how and why these neural networks work, and what properties, advantages, and disadvantages they have. On the basis of this analysis we created, using existing visualization tools, some example cases that were the basis for the proposed visualization tool. In analyzing the selected neural networks, we chose two for visualization. The first is a network based on the "Hard Competitive Learning" method. It is fairly well described in the current literature, but the literature often lacks a set of example cases that would sufficiently clarify the properties and dynamics of this method. We therefore designed a set of test examples capturing all the essential features of Kohonen learning. The second network is referred to in the literature as "Dot Product SOM" and is described very briefly and without any proofs. We therefore analyzed this method in more detail and supported the results with our own proofs. Although both types of network are based on cluster analysis, each of them uses a different method for recognizing clusters. This was the main reason we chose precisely these two types of neural network.

In the third part we became acquainted with current visualization tools and, on the basis of their comparison, designed a new visualization tool. In the design we placed great emphasis on simplicity, extensibility, and flexibility. The design is conceived so that we can simulate several neural networks at once. Although the implemented networks have only one layer, multilayer networks can easily be created. This ensures that in further development of the application we can use new types of neural networks. We implemented the visualization tool in the JAVA language, thereby giving the possibility of wide use over the Internet. For the applet we created a WWW presentation that provides basic information about the application and the problems addressed.

The applet was also subjected to tests whose aim was comprehensibility, correct functionality, and stability. For such a check it is more appropriate that the testing be carried out by someone other than the author. Therefore we asked Ing. Jan Ingerle, who tested the application. Thanks to this, two errors were revealed, one of which had a direct influence on the stability of the application. All findings, fixes, and results are given at the end of the fourth chapter. Finally, thanks to the analyses in the second chapter, we proposed sets of examples that show all the characteristic properties and problems of both paradigms.

In conclusion, it can be said that we successfully managed to create a visualization tool that helps to represent the learning of neural networks, and thereby contributes to a better understanding of the given problem.

References

[1] Miroslav Šnorek. *Neuronové sítě a neuropočítače* [Neural Networks and Neurocomputers].

[2] Jiří Šíma, Roman Neruda. *Teoretické otázky neuronových sítí* [Theoretical Questions of Neural Networks].

[3] Y. Z. Tsypkin (Cypkin). *Foundation of the Theory of Learning Systems.* Academic Press, New York, 1973.

[4] R. Hecht-Nielsen. *On the algebraic structure of feedforward network weight spaces.* In *Advanced Neural Computers*, pp. 129–135. Elsevier, 1990.

[5] Hartmut S. Loos, Bernd Fritzke. *DemoGNG applet.* `http://www.neuroinformatik.ruhr-uni-bochum.de/ini`

[6] Teuvo Kohonen. *Self-Organizing Maps.* Springer, 2nd edition.

[7] Teuvo Kohonen, Jussi Hynninen, Jari Kangas, Jorma Laaksonen. *The Self-Organizing Map Program (SOM_PAK / LVQ_PAK).* `http://www.cis.hut.fi/research/som_lvq_pak.shtml`

Translator's Notes

1. **Source recovery.** The original PDF (generated by `dvips`/LaTeX with Type 3 fonts and no Unicode mapping) could not be copied or text-extracted; the page rendered visually but yielded only garbage on extraction. The Czech text was therefore recovered by OCR (Tesseract, Czech model) from page rasterizations. Prose recovery was of high quality; equations and figures were not.

2. **Equations.** Equations were transcribed from the original; (2.18)–(2.19), illegible in the first scan, were recovered from the author's original 2001 PostScript file and are reproduced exactly. Equations (2.4)–(2.5) are standard renderings of the original competition/update rules. *Note on (2.14) and (2.16):* as printed in the original they carry a "+" where differentiating (2.13) yields "−"; the corrected sign reappears from (2.17) onward, and the conclusion (a maximum) is unaffected, since (2.19) is manifestly negative for $m_1 \in (0, 1)$.

3. **Figures.** None of the 15+ figures are reproduced; each is represented by a captioned placeholder. The figures (plots, screenshots, vector diagrams, and Sammon-projection hypercube layouts) remain in the original PDF and can be embedded on request.

4. **Terminology.** The original distinguishes *Kohonenova síť* ("Kohonen network" — winner-only competitive learning, close to online k-means) from *Kohonenova mapa* ("Kohonen map" — the topology-preserving SOM); this distinction is preserved. "Hard Competitive Learning" follows the terminology of Fritzke's DemoGNG.

5. **Cross-references.** "Page N of the original" refers to the original Czech pagination, which is not preserved in this translation.

6. **Title.** The original title was not present on the recovered title page; the descriptive title on the cover of this translation is a reconstruction based on the chapter headings.