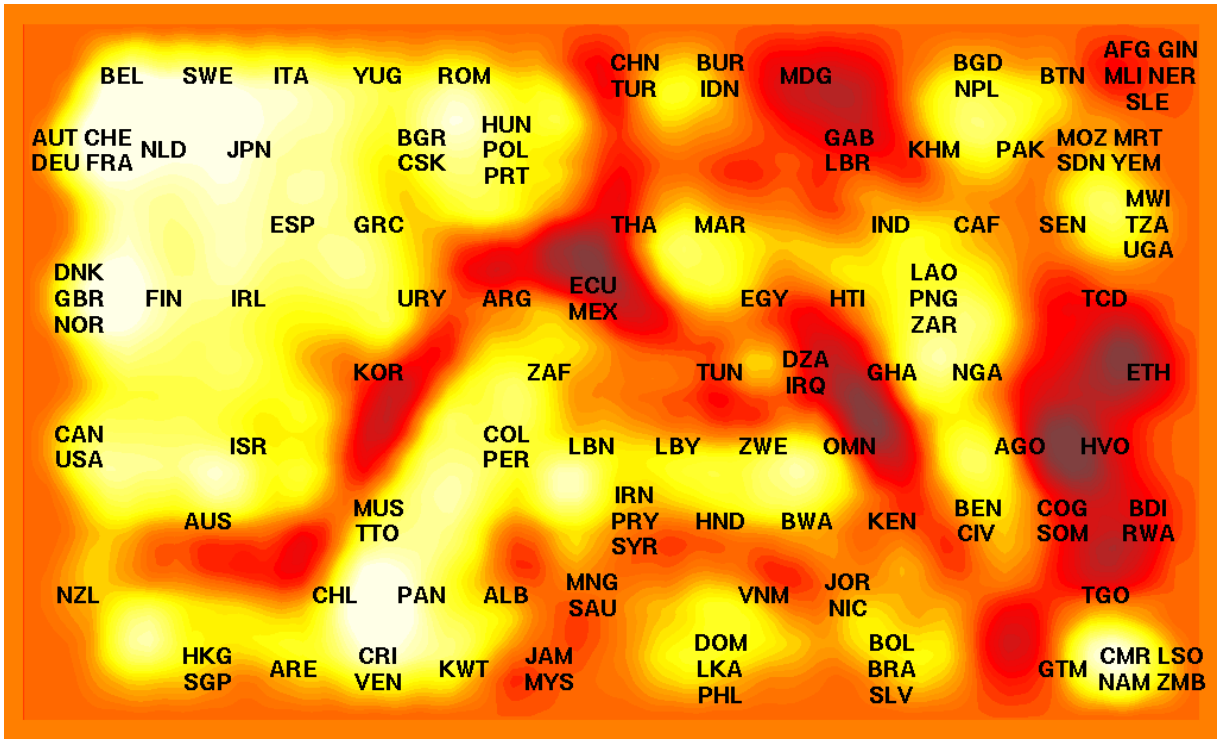
Teuvo Kohonen

MATLAB Implementations and Applications of the *Self-Organizing Map*



Teuvo Kohonen

MATLAB Implementations and Applications of the *Self-Organizing Map*

Aalto University, School of Science
P.O. Box 11000, FI-00076 AALTO, FINLAND

Unigrafia
Helsinki, Finland, 2014

ISBN 978-952-60-3678-6
e-ISBN 978-952-60-3679-3

Unigrafia Oy, Helsinki, Finland, 2014

THIS BOOK SHOULD BE CITED AS:
Kohonen, T., MATLAB Implementations and
Applications of the Self-Organizing Map
Unigrafia Oy, Helsinki, Finland, 2014.

DISTRIBUTION AND SALES (PRINTED BOOK):
Unigrafia Bookstore Helsinki
http://kirjakauppa.unigrafia.fi/
books@unigrafia.fi
P.O. Box 4 (Vuorikatu 3)
FI-00014 UNIVERSITY OF HELSINKI, FINLAND
Tel +358 9 7010 2366
Fax +358 9 7010 2374

http://docs.unigrafia.fi/publications/kohonen_teuvo/index.html

Foreword

This is not a mathematical treatise but a guide. It is neither any handbook of the syntax of computer programming, but rather a starter*, which contains a number of exemplary program codes. You may call it a recipe book.*

The method discussed here, the *Self-Organizing Map (SOM)* introduced by the author, is a *data-analysis* method. It produces low-dimensional *projection images* of high-dimensional *data distributions*, in which the *similarity relations* between the data items are preserved. In other words, it is able to *cluster* the data, but at the same it *orders the clusters*. In a way, it forms reduced *abstractions* of complex data. This method has been in practical use in science, technology, finance and many other areas since 1982. Over 10 000 scientific papers and more than 20 books on it have been published. Plenty of mathematical descriptions and justifications of its simpler versions have been presented, but for a general dimensionality and distribution of the data items, no stringent proofs of the convergence of the algorithm yet exist. Nonetheless, with a proper choice of its parameters, the algorithm normally converges. In spite of being mathematically "ill-posed," the mappings produced by this algorithm have turned out illustrative and useful in practice, and the correctness of the maps thereby formed can be analyzed and verified.

Since 1989, many SOM software packages have been published by various parties. Usually some diagnostic and other auxiliary programs have been included with the basic SOM algorithms. Some of these packages are freeware, others are commercial, and many researchers use specific SOM programs developed by themselves for particular applications. One might think that the methodology would already have been established and standardized, but in practice one has encountered following kinds of problems:

1. Some widely-spread general-purpose algorithms, like those contained in the SOM Toolbox developed in our laboratory, are only available as a set of *functions* programmed in particular languages, e.g., MATLAB. Some people may at first be confused about how to write proper and correct *scripts* that use these functions to implement a specific SOM algorithm, and what should be included in those scripts.

2. Frequently asked questions concern the selection of *features* for the input-data items and proper *dimensions* for the SOM array: how many nodes should be included in it, and what structure of the array should be used in a particular application.

3. In order to achieve good results, one has to pay special attention to the *selection and preprocessing of the input data*. It means that the training samples ought to be carefully verified and validated in order that they represent the true statistics of input, and do not contain "outliers" that may be due to systematic errors or faults. The data should also have *metric properties* for its self organization. On the other hand, it may not be harmful, at least in the preliminary

studies, to involve *redundant features* to describe the data items, because after the SOM has been constructed and analyzed, it is easy to ignore, for instance, those features that depend on the other features, in order to save computing resources. However, it may sometimes be advantageous to assign different *weights* to the different features for better resolution in the map.

4. It may not have been realized generally that the basic SOM algorithm implements a *complex, nonlinear computational process*. It is easy to overlook the importance and meaning of its *training parameters*, because they affect, e.g., the speed of convergence and even the correctness of the final result. The values selected for these parameters are usually not told in publications! Some SOM maps that have been reported seem to represent only temporary states achieved during the course of the learning process, and they may still change if the training is continued.

5. However, it is not generally known that the *batch computation*, which can be used to speed up the construction of the SOM, *will converge in a finite number of cycles*, if carried out properly. This result can be used as a *stopping rule* of the algorithm.

6. One has often not been aware of the existence of helpful tools by which the *quality of the resulting SOMs* can be checked.

The SOM has mainly been used by experts of mathematical statistics and programming. However, with a little of guidance, even non-specialists are expected to be able to use it correctly. So this not a textbook, which is trying to define the syntax of the complete SOM Toolbox. The purpose of this discourse is to give the first advice in the correct application of the SOM, using exemplary scripts relating to different application areas.

Espoo, Finland, December 10, 2014
Teuvo Kohonen

Contents

Preface

I am sure that all of you know what *smoothing* means in numerical mathematics. For example, in a two-dimensional array of numbers, you first compute all of the *local averages* around every node of the array (whereupon the corners and edges of the array have to be taken into account in a separate way). In forming the averages, the numbers are usually *weighted* by coefficients, the values of which depend on the distance of the array node to be processed from the neighboring nodes over which the average is taken. These coefficients are called the *kernel*. After that, all of the old numbers in the array are replaced by the weighted local averages. Smoothing can also be regarded as a *convolution* of the original array with the smoothing kernel. These phases are reiterated a wanted number of times.

Many of you are also familiar with the *k-means clustering*, also called the *vector quantization*, in which k *local averages* are computed for a finite set of variables in such a way that when each one of the original variables is approximated by the *closest* local average, the *average quantization error* thereby made is minimum. In the k-means clustering the original variables, usually metric vectors, are not ordered in any way, and do not belong to any array.

Around 1981 I was wondering what would happen if the k-means clustering and the smoothing would be combined. Actually this happened in the context when I was studying the theory of *artificial neural networks* and especially the *brain maps*. In an attempt to simulate the learning processes that take place in the brain I contrived an *unsupervised learning scheme* called the *Self-Organizing (topographic) Map* that was supposed to describe how the brain maps might be formed by adaptation to various sensory features.

As the scheme that combined k-means clustering with smoothing also worked well in many practical data-analysis tasks, without any direct connection to brain physiology, I started to advance it as a general analysis tool; many mathematicians had already asked me why I need to refer to the neural networks! This method, in a general form, has now been proliferated to many fields of science as a *data-analysis method*, as you can read from this book.

One of my graduate students once said that to understand the Self-Organizing Map one has to internalize half a dozen new principles simultaneously.

Maybe the threshold in trying to understand the SOM would be lower, if one could first internalize a single basic idea which I got in the beginning of 1981, and which quickly materialized in the SOM algorithms. Consider a set of input data vectors that we want to analyze. Also consider a two-dimensional regular array of "cells", each of which contains an *adaptive parameter vector*, a "model." These "models" together shall *represent* the set of input data vectors. The "models" shall have the same dimensionality as the input data vectors.

Especially *in the analysis of clustered input data*, the number of "models" is assumed to be much smaller than that of the input data vectors, whereupon each cluster is represented by one or a few models. The objective is then to determine the values of the "models" as *local averages of the input data vectors* in such a way that *the distribution of the "models" approximates the distribution*

of the input vectors. Moreover, *the "models" shall be ordered two-dimensionally to reflect the similarities between the input data items*. Using a relatively small number of models it is possible to visualize the similarity relations of even a very large data base.

On the other hand, there also exist problems in which we have only one unique sample of each input item, and these items can then be *projected* onto the self-organizing map nonlinearly; in this case, usually, the SOM array has more nodes than input data.

In both types of problems, the "models" constitute a *similarity diagram* of the set of input items. It is necessary to realize that the SOM can handle these two rather different types of problems using the saame algorithm!

In order to construct a set of such "models" adaptively, one can start even with random initial values for the "model vectors". (If the models are preliminarily ordered, for instance along with their principal components, the self-organizing process proceeds much quicker, as we will see.) In the simplest adaptive process in which such "models" are formed, the input vectors are taken one at a time and compared with all of the "models" concurrently. A correction to a particular subset of the "models" is made in the following way, and this "training" of the "models" is continued iteratively, always picking up a new input vector and making the corrections:

The basic idea in the formation of the SOM:

Every input data vector shall select the "model" that matches best with it, and this "model", called the winner, as well as a subset of "models" that are its spatial neighbors in the array, shall be modified for better matching.

It is obvious that this is a kind of a *smoothing process* that tries to increase the *continuity* of the "models" in the two dimensions. At the same time the "models" around the "winner" tend to *approximate* the input data vectors. The result is the Self-Organizing Map.

We intend to define this training principle mathematically, and then implement it by program codes. It will turn out that this principle works in a great many different applications. What is different from case to case is a proper *preprocessing* and *representation* of the input data, as well as the *similarity measures*.

A road map to the contents of this book

Progression of ideas

Here is a suggestion for a "road map" to the reading of this book. You do not have to read all of the sections at once before you can use the algorithm, or understand how you could apply it to your own problems. Secs. 1 through 8 constitute the theoretical part, and the basic idea already transpires from Secs. 3,4, and 5.

Sec. 4 contains fundamental mathematical discussions, but if you don't like them, you might skip them first and continue with Sec. 7, which tries to describe the most widely used "Batch Map" algorithm in an illustrative fashion, without mathematics.

When you are ready to start with concrete applications, you should proceed to the description of software that starts at Sec.10. I recommend that before you start working on a problem of your own, you should also run one of the given examples first, e.g., the one described in Sec.12, to get some hands-on experience of the behavior of the algorithm. After that you may continue with another example that looks most similar to you own problem, and start adapting its program code to your own case.

Section 1 tries to explicate the *philosophy* of the SOM, because these aspects may remain unnoticed if you encounter the description of the SOM algorithm first time in some technical review. If you want to be a serious SOM worker, you must be aware of what is actually taking place in the SOM, and to what category of methods the SOM belongs.

Section 2 is trying to convince you about the practical importance of the scientific, industrial, and financial importance of the SOM algorithm, and also to tell something about the current progress in its use.

Section 3 has been written to answer the most frequently asked questions on the SOM. For to obtain good results it is very important to "internalize" these details.

Section 4 is intended to provide the mathematical justification of the method. I am using the term "justification," although the convergence proof of the SOM algorithms has not yet been given for general dimensionality and distributions of the input data. It must be understood that the SOM describes a *nonlinear decision process*, and like many other nonlinear dynamic processes, it is very hard to prove mathematically. The mathematics in this section defines the state of the SOM process as a function of training steps, and specifies the asymptotic (equilibrium) state after sufficiently long training. It is this particular result on which the practical *batch computation process* of the SOM ("Batch Map") is based. Nonetheless, you need not be worried about managing the SOM method

mathematically, if you use standard software and follow some basic advice that I am attending to give you in this book. The SOM algorithm is unbelievably robust in practice, and there exist diagnostic methods to assess the quality of the maps produced by the algorithm.

Section 5 illustrates the self-organizing process by a very simple, two-dimensional example, in which we can see a *topographic order* ensuing in the set of models. This example gives rise to a practical application, to be discussed in Section 6.

Section 6, as mentioned above, demonstrates that the SOM algorithm can be used to materialize an effective *adaptive demodulator of signals* used in telecommunications. This demodulator has a number of discrimination levels, which are automatically adjusted to take into account various distortions in the transmitted signals levels, for instance due to the attenuation and reflection of radio waves through many paths.

Section 7 defines the batch computation algorithm of the SOM, and this part is very central to the understanding of the SOM. It can be read almost without mathematical background. What is most important to understand is that the *neighborhood function* has a very central role in the operation of the SOM, and *its radius should never go to zero*, because otherwise the algorithm would lose its ordering power and would be reduced to the classic k-means algorithm. This section also propagates the idea that if the set of input data vectors is fixed, and the neighborhood function is held constant during the final training cycles, *the SOM algorithm will terminate in a finite and not too large number of training cycles.*

Section 8 lists various alternatives of *similarity* between the input data items, and many of them occur in the examples given in this book. The purpose of this section is to demonstrate the applicability of the SOM method to a wide range of practical problems and applications.

Section 9 emphasizes that a variety of new self-organizing map principles have been introduced and studied in recent years, and a number of competing software packages have been published, commercial as well as freeware. It is not my purpose to evaluate or compare these products. It is even impossible for me or for anybody else to review the complete range of these products, because of a large amount of material and sheer lack of time to read it. I hope that the reader would be content if I give her or him working examples of my own method and descriptions of applications that I have personally encoded and verified.

Neither does this book contain extensive examples from the industry or finance, because the explicit program codes, which are the essence of this discourse, would be too big in those applications to be copied and explained in full in this kind of a tutorial presentation. I firmly believe that "upscaling" of the codes is no problem, once you understand how they work in smaller-scale appli-

cations, and I also believe that the behavior of the SOM algorithm is principally not very different in "toy problems" and in big applications. The present-day and future computers have unbelievable capacities for straightforward upscaling of the computations.

Section 10 finally starts with the description of the *SOM Toolbox* and its central functions, extensively applied in this book.

Section 11 solves the QAM demodulator problem using the Batch Map algorithm and the SOM Toolbox.

Section 12 describes a practical case example, the mapping of input data items onto the SOM according to their *physical attributes*. This example, as well as the others selected to this book, can be seen to contain a result, which is typical to data mining: we can see that *a detailed result, which was completely unexpected, will emerge*. In this example we see that the ferromagnetic metals *Ni, Co* and *Fe* are mapped into the same SOM cell although no magnetic properties were included in the set if input attributes. So there must exist some hidden properties in the other physical attributes that are common to this subset of metallic elements and have a strong implicit correlation with them.

The program code given with this example may serve almost as such to a number of other related applications; the only variable detail is a different input data matrix in different applications.

Section 13 contains another physical example, the self-organization of *colors*, but the representation of input data items is completely different from that of the previous example. In Sec. 12 we had a *finite set of input data items*. In this example *the items are shades of color*, which we can define in indefinite quantities. On the other hand, the dimensionality of the color vectors is low, three, and so we can display the models of color shades as such in the SOM.

A surprising feature that ensues in this experiment is that if we *scale down* the input signals properly (e.g., as square roots of the RGB values), the SOM of colors starts to resemble the *chromaticity diagram of human vision*! This representation is also found in a particular visual area of the human brain, and so we may have here a possible (but of course, rather coarse) *neural model of color vision*.

Section 14 describes an SOM that is computationally similar to the one discussed in Section 12, but it has been applied to another important area of applications, namely, to *finance*. The six attributes taken from the Internet describe the present *financial state of 50 countries or unions*.

The locations of the countries in this map look different from those of the Welfare map shown in Fig. 1. The main reason for these cases looking different is that in the Welfare map, attributes such as the levels of education and health services were included, whereas the few financial indicators included there were

not emphasized. The example represented in Sec. 14 is based *solely* on financial indicators. Also the years from which these data sets were collected were quite different: the Welfare map stems from 1992, while the financial data relate to the year 2014.

Section 15 introduces a graphic method for the demarcation and emphasis of *cluster borders* on the SOM array using *shades of gray*. It is called the *U matrix*. These shades represent *differences between neighboring model vectors*: a dark shade means that this difference is large, i.e., there exists a cluster border.

Section 16 does not contain anything new, except that it points out that the attributes need not represent quantitative or measurable properties of items, but the input information can be given in the form of *nonnumerical, logic statements*.

Section 17 is another example of the use of binary attributes. We only show this example in order to exemplify the various "dimensions" of the SOM, and the representation of models by *images of the best-matching items*. This example may also give a hint to new applications of the SOM.

Section 18 extends the use of discrete symbolic attributes in the SOM. It is a generalization of the binary attributes, but the symbolic attributes may here assume *a multitude of discrete values*. The most important new idea is that one can represent multivalued discrete attributes, which are not related quantitatively, by *different unit vectors*. This example is based on the familiar *mushroom classification* data.

The *distributions of input data on the SOM* can be represented using *shades of gray*. The falling of input items into a particular class is shown by a so-called *hit diagram*. When the known data vectors are used as inputs to the SOM, the number of "winners" on each of the nodes is indicated by a shade of gray.

The surprising result transpiring in this example is that the attribute that indicates the *edibility* of the mushrooms *was not involved in the computation of the SOM*; nonetheless, on the basis of the visible attributes, the models were automatically *separated* into the clusters of poisonous vs. edible species, which are shown by two separate histograms drawn onto identical SOM lattices. For to check this, the hit diagrams of the edible vs. poisonous mushrooms were constructed by first dividing the input data according to the edibility attribute into two separate sets, and then drawing *separate hit diagrams* for them. We can see that the histograms of these two classes are *segregated* to a reasonable accuracy: their intersection is almost empty.

Section 19 exemplifies the use of the SOM in *statistical classification*. The input data that represent *mutually related scientific articles* fall in several categories or classes, in this example four. The SOM is first computed using all available input data, not yet regarding the class information. Nonetheless the SOM models were

almost completely segregated into four classes, the histograms of which had an almost empty intersection.

In this example we also introduce new kinds of features used in the classification of textual documents. Each document is identified by the *word statistics* of its contents. First a *histogram of words* used in the document is recorded. Then the words in the histograms, or the *elements of the histogram* regarded as vectorial elements are *weighted* by statistical arguments. Finally, rare and very common words are dropped, and the weighted histograms of words, believed to provide the largest class separation, are used as "input features" of the documents.

In Sections 21 and 22 we shall show how the different areas of a single SOM can be labeled by different *pseudo-colors* to demarcate zones that are supposed to represent the various classes most credibly.

Section 20 is an "interlude." Before proceeding with bigger examples, it was felt necessary to establish rules for the selection of certain parameter values, such as *the number of coarse and fine training cycles vs. array size*. This benchmarking was made using the Reuters document data, which seemed well verified and well validated.

Section 21 shows that it is possible to divide the area of the same SOM into *zones* according to the *classes of input data items* mapped onto the SOM. The principle described here first is based on the Bayesian classification rule: a cell in the SOM is labeled according to the majority of classes of items that have selected this node as a winner. In the classification of a new, unknown input sample, its classification is decided to comply with the label of the corresponding winner model.

Section 22 presents another labeling criterion. A cell is labeled by *k-nearest-neighbors classification*, i.e., according to the majority of labels in the k input data items that are most similar to the model in the cell. One may see that the k-nearest-neighbors labeling is smoother, whereas the Bayesian decision provides a more accurate labeling.

Section 23 introduces an idea, which is new in the SOM research. Instead of classifying an unknown input item according to the winner model, a more informative visualization of the input data item is to define a small set of models that *together fit to the input best*. This is done by a least-squares fitting procedure in which *non-negative fitting coefficients* are used. When using the non-negativity constraint, the number of nonzero terms in this optimized linear mixture is usually very small, on the order of one per cent of all models. The example discussed in this section demonstrates the method using the Reuters document data.

Section 24 applies the same idea to the visual analysis of *mobile phone data*. It seems that this principle might work well in the monitoring the performance

of any complex system, machine, or machinery.

Section 25 moves to a completely new subject, namely, *contextual SOMs*. In other words, the *semantic similarities between words* used in various text corpora are usually reflected in the *similarities of the local contexts* of neighboring words, in which they occur. Thereby we only mean contexts extending to a distance of a few word positions. The example discussed in this section is still a "toy problem," in which the source text is generated artificially, on the basis of an extremely small vocabulary. This example has mainly been included here for historical reasons,because it was the first of this type, but it already shows how complex the preprocessing may be that is needed in the "contextual SOMs."

Section 26 then describes a rather large experiment which I carried out in 2009-2010. This work was prepared for WSOM 2011 (Workshop on Self-Organizing Maps), but in this book I have made big changes to the original program codes, to make its explanation more illustrative.

The text corpus was very big, on the order of 1,500,000 words, all of which were assigned to one of 89 linguistic classes by my Chinese colleagues. Incidentally, this text was written in Chinese that I do not master, but since the words were provided with linguistic meanings, the SOM could nonetheless analyze the words automatically. The Chinese text was selected for two reasons: 1. It was the only classified corpus that was available to me. 2. Since the words in Chinese are not inflected and have no endings, the semantics ensue from the local contexts in totally pure form.

It may be possible to try this same example for other languages, too.

A result of this analysis shows that: 1. The words are clustered in the SOM on basis of their local contexts, not only by their word classes but also according to their *roles as sentence constituents*, which sounds reasonable. 2. In the original article [45] we could further show that the frequency of a word in the text corpus has a strong effect on its classification; this result is not presented in this book.

Section 27 tries to replicate the "Welfare map" experiment shown in Fig. 1. Unfortunately the SOM Toolbox functions do not have the provision for using incomplete input data (with random elements missing from the input vectors). In this experiment I tried to *patch* the incomplete data by *estimation of the values of the missing elements on the basis of neighboring models in the SOM where these elements were given*. This method is nicknamed "inputting." It turned out that almost reasonable results were obtained, although the SOM did not quite comply with Fig.1, computed by our earlier SOM_PAK software package.

Section 28 takes us to a still different, major problem area, namely *SOMs of symbol strings*. When I introduced the Batch Map algorithm, it occurred to me that this nonmathematical formulation of the algorithm might also apply to nonvectorial variables, e.g., strings of symbols, if some kind of *"mean" over a set of*

symbol strings could be defined. In 1985 I had published the idea of a "generalized median" of string variables, which seemed to fit well to this algorithm. Since then a genre of "string SOMs" has been studied and used especially in bioinformatics.

The worst problem in constructing string-SOMs is that the strings are *discrete-valued entities*. This problem may be possible to tolerate if the strings are long, like in bioinformatics. In this book I have selected an example in which the domain of values of the elements of strings is very small, namely, (1, 2, ... , 26) (the letters), and the strings are very short (names). The *initialization* of the models is another problem. Accordingly, there occur frequently various *ties* in the comparison of strings. I had to invent all shorts of unusual tricks to make this example computable.

Section 29 describes a special algorithm. Already in 1984 we were trying to apply the SOM to automatic speech recognition, or more accurately, to the recognition of *phonemes* from continuous speech, in order to transcribe speech. This method seemed to work promisingly for Finnish and Japanese, which are very phonemic languages. In order to increase class separation, we were *supervising* the construction of the SOM by including the labels of the phonemes (as unit vectors) in the input patterns: in this way, the class information enhanced the clustering of the acoustic vectors into phonemic groups. The simple artificial-data example is trying to illustrate this effect. Our "Phonetic Typewriter," which was based on this idea, was published in the *IEEE Spectrum* in 1988 [36].

Section 30, "The Learning Vector Quantization," describes a class of learning algorithms that produce near-optimal class separation in the sense of Bayesian statistics. I invented this idea in trying to improve the accuracy of our recognizer of phonemes, or the "Phonetic Typewriter," and it indeed increased the recognition accuracy, when compared with the supervised SOM discussed in Sec. 30. If you are mainly interested in the SOM and not so much in statistical pattern recognition, you may skip this section.

Section 31 demonstrates that an SOM can act as a *filter bank* for the preprocessing and extraction of features for waveform analysis, e.g., in the analysis, synthesis, and perception of speech.

In the book in presentation we compute for segments of waveforms so-called *linear predictor coding (LPC) coefficients*, which are a viable alternative to frequency analysis by the Fast Fourier Transform, or to the so-called *cepstra* which are applied widely as features in speech recognition [39]. The SOM array is *calibrated* by known samples of the waveform, e.g., by known *phonemes* extracted from continuous speech. An unknown signal, e.g., speech is first *segmented* into parts, for each of which the LPC coefficients are computed. The *best-matching LPC coefficients of the SOM* are then identified, and the label of the corresponding node indicates the recognition result, e.g., the phoneme or pseudo-phoneme represented by that segment.

In order to improve class separation, the models of the SOM can further be fine tuned, either by the Supervised SOM, or preferably by Learning Vector Quantization.

Section 32 gives practical hints of how to improve the speed of computation by shortcut computing methods and eventual parallel processing. All of these ideas cannot be used in the SOM Toolbox functions, but they may stimulate future research.

1 The Self-Organizing Map; an overview

The Self-Organizing Map represents a set of high-dimensional data items as a quantized two-dimensional image in an orderly fashion. Every data item is mapped into one point (node) in the map, and the distances of the items in the map reflect similarities between the items.

The Self-Organizing Map (SOM) is a *data-analysis method that visualizes similarity relations in a set of data items*. For instance in economy, it has been applied to the *comparison of enterprises at different levels of abstraction*, to assess their relative financial conditions, and to profile their products and customers. On the other hand, in industry, the monitoring of processes, systems and machineries by the SOM method has been a very important application, and there the purpose is to describe the masses of different input states by *ordered clusters of typical states*. In science and technology at large, there exist unlimited tasks where the research objects must be classified on the basis of their inherent properties, to mention the classification of proteins, genetic sequences and galaxies. A comprehensive listing of the most important applications can be found in Sec.2.

It is assumed that you have already got some information about the SOM (e.g., [39], [46]) and you are now interested in writing program codes for its application. The purpose of this guide is to help you to start with it.

1.1 Is the SOM a projection or a clustering method?

The SOM as a nonlinear projection. When I gave my first conference talk on the SOM at a Pattern Recognition conference in 1982, a remark from the audience pointed out that the SOM belongs to the *nonlinear projection* methods, such as *multidimensional scaling (MDS)*, especially the *Sammon projection* [76]. That is true, but only partially. In the projective methods the data vectors, often with a very high dimensionality, are mapped onto a two-dimensional Euclidean plane in such a way that the mutual distances of the projections on the 2D Euclidean plane are approximately the same as the mutual distances of the original vectors in the high-dimensional input-data space. Similar items are located close to each other, and dissimilar items farther apart in the display, respectively. It is said that the items are then represented in an *abstract topographic order*.

However, the SOM represents input data by models, which are local averages of the data. Only in some special cases the relation of input items with their projection images is one-to-one in the SOM. More often, especially in industrial and scientific applications, the mapping is *many-to-one*: i.e., the projection images on the SOM are *local averages of the input-data distribution*, comparable to the *k-means averages in classical vector quantization (VQ)*([19], [20]). In the VQ, the local averages are represented by a finite set of *codebook vectors*. The SOM also uses a finite set of "codebook vectors," called the *models*, for the representation of local averages. An input vector is mapped

into a particular node on the SOM array by comparing it with all of the models, and the *best-matching model*, called the *winner*, is identified, like in VQ. The most essential difference with respect to the k-means clustering, however, is that the models of the SOM also reflect *topographic relations* between the projection images which are *similar to those of the source data*. So the SOM is actually a *data compression method*, which represents the topographic relations of the data space by a finite set of models on a topographic map, in an orderly fashion.

In the standard-of-living diagram shown in Fig. 1, unique input items are mapped on unique locations on the SOM. However, in the majority of applications, there are usually many statistically distributed variations of the input items, and the projection image that is formed on the SOM then represents *clusters* of the variations. We shall make this fact more clear in examples. So, in its *genuine* form the SOM differs from all of the other nonlinearly projecting methods, because it usually represents a big data set by a much smaller number of models, sometimes also called "weight vectors" (this latter term comes from the theory of artificial neural networks), arranged as a rectangular array of nodes. Each model has the same number of parameters as the number of features in the input items. However, an SOM model may not be a replica of any input item but only a *local average* over a subset of items that are most similar to it. In this sense the SOM works like the k-means clustering algorithm, but in addition, in a special learning process, the SOM also arranges the k means into a *topographic order* according to their similarity relations. The parameters of the models are variable and they are adjusted by learning such that, in relation to the original items, *the similarity relations of the models finally approximate or represent the similarity relations of the original items*. It is obvious that an insightful view of the complete data base can then be obtained at one glance.

The SOM classifies feature vectors. Assume now generally that we have a large set of some input-data items and each of them is represented by several *features*. The features may consist of numerical attributes, such as statistical descriptors of an item, but many other types of features can also be used. The simplest measure of the similarity of two items is then the similarity of their feature sets in some metric, but again, more complex definitions of similarity can be delineated.

The SOM display is quantized. The SOM does not map high-dimensional items onto a Euclidean plane but onto a regular array or network of *nodes*. In the first illustrative example, shown in Fig. 1, we demonstrate how the SOM compares the *standard of living in different countries* of the world, labeled by three-letter symbols (which may be understandable without any separate legend). From the statistics of the World Development Record of the World Bank of the year 1992, 39 statistical indicators, which describe factors like health, education, consumption and social services, were picked up, forming a 39-element *feature vector* for each country. All indicators are relative to population. As will be explained shortly, an abstract map, a nonlinear projection of the countries

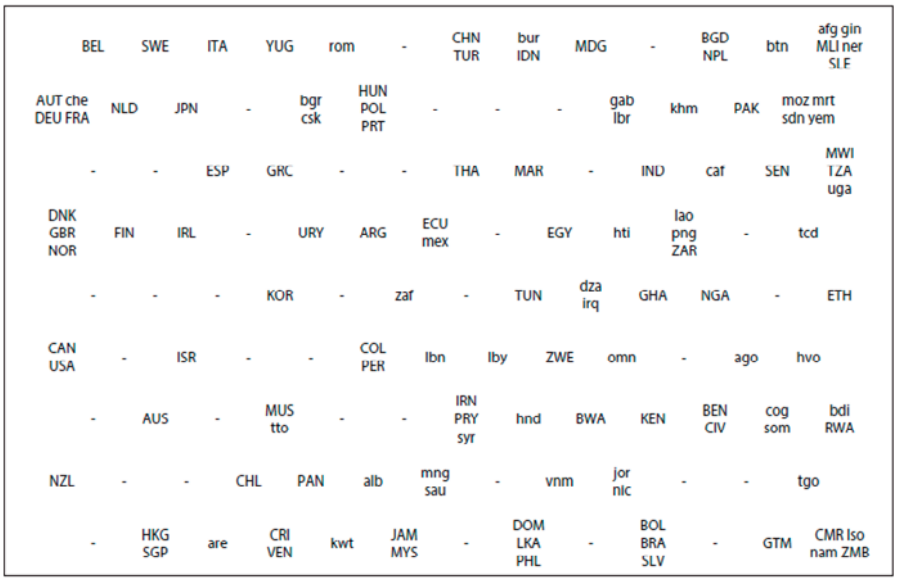


Fig. 1. Structured diagram of the data set chosen to describe the standard of living in 126 countries of the world in the year 1992. The abbreviated country symbols are concentrated onto locations in the (quantized) display computed by the SOM algorithm. The symbols written in capital letters correspond to those 78 countries for which at least 28 indicators out of 39 were given, and they were used in the real computation of the SOM. The symbols written in low case letters correspond to countries for which more than 11 indicator values were missing, and these countries are projected to locations based on the incomplete comparison of their given attributes with those of the 78 countries.(Cf. The legend on symbols on page 4.)

onto a rectangular array was computed by the SOM algorithm. This implementation of the SOM has been computed by our older SOM_PAK software package, which has provisions for dealing with missing data.

The overall order of the countries on the map can be seen to illustrate the traditional conception of *welfare*. In fact, the horizontal dimension of the map seems to correlate fairly closely with the gross national product per capita of the countries. Refined interpretations about the fine structures of welfare and poverty types in different areas of the map can be made based on some traditional methods like factor analysis applied on selected subsets of countries. Nonetheless, this two-dimensional display is more easily understandable than the ordinary linear tables that are based on econometric functions. One might say that countries that are mapped close to each other in the SOM have a *similar state of development, expenditure pattern, and policy.*

The SOM models are developed, not moved. It shall be emphasized that unlike in the other projective methods, *in the SOM the representations of*

Table 1. Legend of symbols used in Fig. 1:

AFG	Afghanistan	GRC	Greece	NOR	Norway
AGO	Angola	GTM	Guatemala	NPL	Nepal
ALB	Albania	HKG	Hong Kong	NZL	New Zealand
ARE	United Arab Emirates	HND	Honduras	OAN	Taiwan, China
ARG	Argentina	HTI	Haiti	OMN	Oman
AUS	Australia	HUN	Hungary	PAK	Pakistan
AUT	Austria	HVO	Burkina Faso	PAN	Panama
BDI	Burundi	IDN	Indonesia	PER	Peru
BEL	Belgium	IND	India	PHL	Philippines
BEN	Benin	IRL	Ireland	PNG	Papua New Guinea
BGD	Bangladesh	IRN	Iran, Islamic Rep.	POL	Poland
BGR	Bulgaria	IRQ	Iraq	PRT	Portugal
BOL	Bolivia	ISR	Israel	PRY	Paraguay
BRA	Brazil	ITA	Italy	ROM	Romania
BTN	Bhutan	JAM	Jamaica	RWA	Rwanda
BUR	Myanmar	JOR	Jordan	SAU	Saudi Arabia
BWA	Botswana	JPN	Japan	SDN	Sudan
CAF	Central African Rep.	KEN	Kenya	SEN	Senegal
CAN	Canada	KHM	Cambodia	SGP	Singapore
CHE	Switzerland	KOR	Korea, Rep.	SLE	Sierra Leone
CHL	Chile	KWT	Kuwait	SLV	El Salvador
CHN	China	LAO	Lao PDR	SOM	Somalia
CIV	Cote d'Ivoire	LBN	Lebanon	SWE	Sweden
CMR	Cameroon	LBR	Liberia	SYR	Syrian Arab Rep.
COG	Congo	LBY	Libya	TCD	Chad
COL	Colombia	LKA	Sri Lanka	TGO	Togo
CRI	Costa Rica	LSO	Lesotho	THA	Thailand
CSK	Czechoslovakia	MAR	Morocco	TTO	Trinidad and Tobago
DEU	Germany	MDG	Madagascar	TUN	Tunisia
DNK	Denmark	MEX	Mexico	TUR	Turkey
DOM	Dominican Rep.	MLI	Mali	TZA	Tanzania
DZA	Algeria	MNG	Mongolia	UGA	Uganda
ECU	Ecuador	MOZ	Mozambique	URY	Uruguay
EGY	Egypt, Arab Rep.	MRT	Mauritania	USA	United States
ESP	Spain	MUS	Mauritius	VEN	Venezuela
ETH	Ethiopia	MWI	Malawi	VNM	Viet Nam
FIN	Finland	MYS	Malaysia	YEM	Yemen, Rep.
FRA	France	NAM	Namibia	YUG	Yugoslavia
GAB	Gabon	NER	Niger	ZAF	South Africa
GBR	United Kingdom	NGA	Nigeria	ZAR	Zaire
GHA	Ghana	NIC	Nigaragua	ZMB	Zambia
GIN	Guinea	NLD	Netherlands	ZWE	Zimbabwe

the items are not moved anywhere in their "topographic" map for their ordering. Instead, *the adjustable parameters of the models* are associated with fixed locations of the map once and for all, namely, with the *nodes of a regular, usually two-dimensional array* (Fig. 2). A hexagonal array, like the pixels on a TV screen, provides the best visualization. Initially the parameters of the models can even have random values. *The correct final values of the models or "weight vectors" will develop gradually by learning.* The representations, i.e., the models, become more or less exact *replica of the input items* when their sets of feature parameters are *tuned* towards the input items during learning. The SOM algorithm constructs the models (in this picture denoted generally by M_i) such that:

> *After learning, more similar models will be associated with nodes that are closer in the array, whereas less similar models will be situated gradually farther away in the array.*

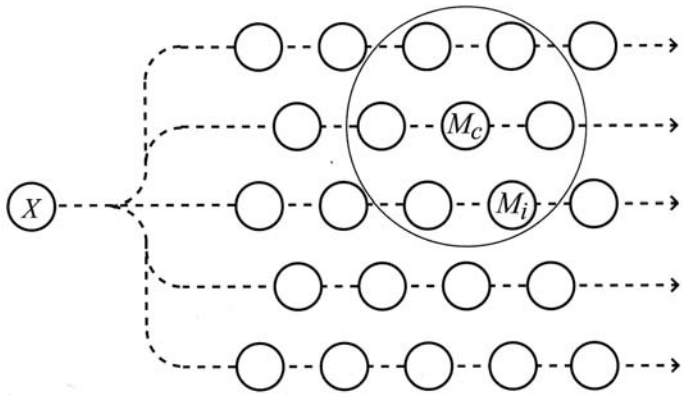


Fig. 2. Illustration of a Self-Organizing Map. An input data item X is broadcast to a set of models M_i, of which M_c matches best with X. All models that lie in the neighborhood (larger circle) of M_c in the array will be updated together in a training step and finally match better with X than with the rest.

It may be easier to understand the rather involved learning principles and mathematics of the SOM, if the central idea is first expressed in the following simple illustrative form. Let X denote a general input item, which is broadcast to all nodes for its concurrent comparison with all of the models.

> *Every input data item shall select the model that matches best with the input item, and this model, called the winner (denoted by M_c in Fig. 2), as well as a subset of its spatial neighbors in the array, shall be modified for better matching.*

Like in the k-means clustering, the modification is concentrated on a selected node that contains the winner model. On the other hand, since *a whole spatial neighborhood around the winner in the array is modified at a time*, the degree of local, differential ordering of the models in this neighborhood, due to a smoothing action, will be increased. The successive, different inputs cause corrections in different subsets of models. The local ordering actions will gradually be propagated over the array. However, the real mathematical processes are a bit more complicated than that, and will be discussed in the following sections.

1.2 Is the SOM a model, a method, or a paradigm?

The SOM as a neural model. Many principles in computer science have started as models of neural networks. The first computers were nicknamed "giant brains," and the electronic logic circuits used in the first computers, as contrasted with the earlier electromechanical relay-logic (switching) networks, were essentially nothing but networks of threshold triggers, believed to imitate the alleged operation of the neural cells.

The first useful *neural-network models* were *adaptive threshold-logic circuits*, in which the signals were weighted by adaptive ("learning") coefficients. A significant new aspect introduced in the 1960s was to consider *collective effects in distributed adaptive networks*, which materialized in new distributed associative-memory models, multilayer signal-transforming and pattern-classifying circuits, and networks with massive feedbacks and stable eigenstates, which solved certain optimization problems.

Against this background, the Self-Organizing Map (SOM) introduced around 1981-82 may be seen as a *model of certain cognitive functions*, namely, a network model that is able to create *organized representations of sensory experiences*, like the *brain maps* in the cerebral cortex and other parts of the central nervous system do. In the first place the SOM gave some hints of how the brain maps could be formed postnatally, without any genetic control. The first demonstrations of the SOM exemplified the adaptive formation of *sensory maps* in the brain, and stipulated what functional properties are most essential to their formation.

The SOM as a data-mining method. In the early 1970s there were big steps made in *pattern recognition (PR) techniques*. They continued the idea of adaptive networks that started the "artificial intelligence (AI)" research. However, after the introduction of large time-shared computer systems, a lot of computer scientists took a new course in the AI research, developing complex decision rules by "heuristic programming", by which it became possible to implement, e.g., expert systems, computerized control of large projects, etc. However, these rules were mainly designed manually. Nonetheless there was a group of computer scientists who were not happy with this approach: they wanted to continue the original ideas, and to develop computational methods for new analytical tasks in information science, such as remote sensing, image analysis in medicine, and speech recognition. This kind of Pattern Recognition was based on mathematical statistics, and with the advent of new powerful computers, it too could be applied to large and important problems.

Notwithstanding the connection between AI and PR research broke in the 1970ies, and the AI and PR conferences and societies started to operate separately.

Although the Self-Organizing Map research was started in the neural-networks context, its applications were actually developed in experimental pattern-recognition research , which was using *real data*. It was first found promising in speech recognition, but very soon numerous other applications were found in industry, finance, and science. The SOM is nowadays regarded as a general *data-analysis method* in a number of fields.

Data mining has a special flavor in data analysis. When a new research topic is started, one usually has little understanding of the collected data. With time it happens that new, unexpected results or phenomena will be found. The meaning often given to *automated data mining* is that the method is able to discover *new, unexpected and surprising results*. Even in this book I have tried to collect simple experiments, in which something quite unexpected will show up. Consider, for

instance, Sec. 12 ("The SOM of some metallic elements") in which we find that the ferromagnetic metals are mapped to a tight cluster; this result was not expected, but the data analysis suggested that the nonmagnetic properties of the metals must have a very strong correlation with the magnetic ones! Or in Sec. 19 ("Two-class separation of mushrooms on the basis of visible attributes") we are clustering the mushrooms on the basis of their visible attributes only, but this clustering results in a dichotomy of edible vs. poisonous species. In Sec. 13 we are organizing color vectors by the SOM, and if we use a special scaling of the color components, we obtain a color map that coincides with the chromaticity map of human color vision, although this result was in no way expected. Accordingly, it may be safe to say that the SOM is a genuine data mining method, and it will find its most interesting applications in new areas of science and technology.

I may still mention a couple of other works from real life. In 1997, Naim et al. published a work [61] that clustered middle-distance galaxies according to their morphology, and found a classification that they characterized as a new finding, compared with the old standard classification performed by Hubble. In Finland, the pulp mill industries were believed to represent the state of the art, but a group of our laboratory was conducting very careful studies of the process by the SOM method [1], and the process experts payed attention to certain instabilities in the flow of pulp through a continuous digester. This instability was not known before, and there was sufficient reason to change the internal structures of the digester so that the instability disappeared.

The SOM principle as a new paradigm in information science. It seems possible that the SOM will open new vistas into the information science. Not only does it already have numerous spin-offs in applications, but its role *in the theory of cognition* is intriguing. However, its mathematics is still in its infancy and offers new problems especially for mathematical statistics. A lot of high-level research is going on in this area. Maybe it is not exaggerated to assert that the SOM presents *a new information processing paradigm*, or at least a philosophical line in bioinformatics.

To recapitulate, the SOM is a clustering method, but unlike the usual clustering methods, it is also a topography-preserving nonlinear projection mapping. On the other hand, while the other nonlinear projection mappings are also topography-preserving mappings, they do not average data, like the SOM does.

2 Main application areas of the SOM

More than 10 000 scientific papers on the SOM have been published. Recently, the number of new publications has been on the order of 800 yearly.

Before looking into the details, one may be interested in knowing the justification of the SOM method. Briefly, by the end of the year 2005 we had documented 7768 scientific publications (cf. [32], [64] and [69]) that analyze, develop, or apply the SOM. The following short list gives the main application areas:

1. Statistical methods at large
   (a) exploratory data analysis
   (b) statistical analysis and organization of texts
2. Industrial analyses, control, and telecommunications: [40]
3. Biomedical analyses and applications at large
4. Financial applications: [13]

In addition to these, one may mention a few specific applications, e.g., profiling of the behavior of criminals, categorization of galaxies ([61]), categorization of real estates, etc.

A very important application area of the SOM has been the exploration of full-text databases, i.e., document organization and retrieval. These publications can only be cited here, because showing the scripts for the management of such big text corpora is not possible in this tutorial book. One should take a look at the original articles, e.g., [54], [24], [25], [26], [27], [28], and [59]. A smaller application has been expounded in Sec. 17 in full.

It is neither possible to give a full account of the theory and different versions of the SOM, or applications of the SOM in this article. We can only refer to the above lists of publications (today, their number is over 10,000, and about 800 new papers are being published yearly), and to more than ten textbooks, monographs, or edited books, e.g. [37], [73], [60], [63], [84], [88], [39], [2], [62], [78], [85], and a great number of PhD Theses.

Two special issues of the journal *Neural Networks* have been dedicated to the SOM: The 2002 Special Issue with the subtitle "New Developments in Self-Organizing Maps," *Neural Networks*, Vol. 1, Numbers 8-9, October/November 2002, and the 2006 Special Issue "Advances in Self-Organizing Maps - WSOM'05," *Neural Networks*, Vol.1, Numbers 6-7, July/August 2006. Moreover, the journal *Neurocomputing* has published a special SOM issue in Vol.21, Numbers 1-3, October 1998.

Quite recently, this author has published an updated review article on the SOM [46].

A series of meetings named the *WSOM (Workshop on Self-Organizing Maps)* has been in progress since 1997. They have been organized in the following venues: Otaniemi, Finland (1997 and 1999) [94], [63]; Lincoln, U.K. (2001) [2], Kitakyushu, Japan (2003) [95]; Paris, France (2005) [96]; Bielefeld, Germany (2007) [97]; St Augustine, FL, USA (2009) [70], Espoo, Finland (2011) [49], Santiago, Chile (2012) [16], and Mittweida, Germany (2014) [92].

3 How to select the SOM array

Usually the SOM display is formed onto a two-dimensional rectangular array, where the nodes are organized as a hexagonal grid.

3.1 Size of the array

One of the most frequently asked questions concerning the structure of the SOM is how many nodes one needs in the array. If the SOM is used to map unique items such as the countries in Fig. 1, one may have even more nodes as there are items, because some items are clustered on the same node, while there will be empty space between the occupied nodes. However, there may exist better visualization methods for them, like the so-called *nonlinear projections* (cf. [76]).

Maybe it is necessary to state first that the SOM is visualizing the *the entire input-data space*, whereupon its *density function* ought to become clearly visible.

The SOM is a quantizing method. Assume next that we have enough statistical data items to visualize the *clustering structures* of the data space with sufficient accuracy. Then it should be realized that the SOM is a *quantizing method*, and has a limited *spatial resolution* to show the details of the clusters. Sometimes the data set may contain only few clusters, whereupon a coarse resolution is sufficient. However, if one suspects that there are interesting fine structures in the data, then a larger array would be needed for sufficient resolution.

Histograms can be displayed on the SOM array. However, it is also necessary to realize that the SOM can be used to represent a *histogram*. The number of input data items that is mapped onto a node is displayed as a shade of gray, or by a pseudo-color. The statistical accuracy of such a histogram depends on how many input items are mapped per node on the average . A very coarse rule-of-thumb may be that *about 50 input-data items per node on the average* should be sufficient, otherwise the resolution is limited by the sparsity of data. So, in visualizing clusters, a compromise must be made between resolution and statistical accuracy. These aspects should be taken into account especially in statistical studies, where only a limited number of samples are available.

Sizing the SOM by a trial-and-error method. It is not possible to estimate or even guess the exact size of the array beforehand. It must be determined by the trial-and-error method, after seeing the quality of the first guess. One may have to test several sizes of the SOM to check that the cluster structures are shown with a sufficient resolution and statistical accuracy. Typical SOM arrays range from a few dozen to a few hundred nodes.

In special problems, such as the mapping of *documents* onto the SOM array, even larger maps with, say, thousands of nodes, are used. The largest map produced by us has been the SOM of seven million patent abstracts, for which we constructed a one-million-node SOM.

On the other hand, the SOM may be at its best in the visualization of industrial processes, where unlimited amounts of measurements can be recorded. Then the size of the SOM array is not limited by the statistical accuracy but by the computational resources, especially if the SOM has to be constructed periodically in real time, like in the control rooms of factories.

3.2 Shape of the array

Because the SOM is trying to represent the distribution of high-dimensional data items by a two-dimensional projection image, it may be understandable that the scales of the horizontal and vertical directions of the SOM array should approximately comply with the extensions of the input-data distribution in the two *principal dimensions*, namely, those two orthogonal directions in which the variances of the data are largest. In complete SOM software packages there is usually an auxiliary function that makes a traditional two-dimensional image of a high-dimensional distribution, e.g., the *Sammon projection* (cf., e.g. [39] in our SOM Toolbox program package. From its main extensions one can estimate visually what the approximate ratio of the horizontal and vertical sides of the SOM array should be.

Special shapes of the array. There exist SOMs in which the array has not been selected as a rectangular sheet. Its topology may resemble, e.g., a cylinder, torus, or a sphere (cf., e.g., [80]). There also exist special SOMs in which the structure and number of nodes of the array is determined *dynamically*, depending on the received data; cf., e.g., [18].

The special topologies, although requiring more cumbersome displays, may sometimes be justified, e.g., for the following reasons. 1. The SOM is sometimes used to define the control conditions in industrial processes or machineries automatically, directly controlling the actuators. A problem may occur with the boundaries of the SOM sheet: there are distortions and discontinuities, which affect the control stability. The toroidal topology seems to solve this problem, because there are then no boundaries in the SOM. A similar effect is obtained by the spherical topology of the SOM. (Cf. Subsections.4.5 and 5.3, however.) 2. There may exist data, which are cyclic by their nature. One may think, for example of the application of the SOM in *musicology*, where the degrees of the scales repeat by octaves. Either the cylindrical or toroidal topology will then map the tones cyclically onto the SOM.

The dynamical topology, which adjusts itself to structured data, is very interesting in itself. There is one particular problem, however: one must be able to define the condition on which a new structure (branching or cutting of the SOM network) is due. There do not exist universal conditions of this type, and any numerical limit can only be defined arbitrarily. Accordingly, the generated structure is then not unique. This same problem is encountered in other neural-network models.

In this guide we do not discuss special shapes of SOMs.

4 The original, stepwise recursive SOM algorithm

4.1 The algorithm

The first SOMs were constructed by a stepwise-recursive learning algorithm, where, at each step, a selected patch of models in the SOM array was tuned towards the given input item, one at a time.

Consider again Fig. 2. Let the input data items X this time represent a sequence $\{\mathbf{x}(t)\}$ of real n-dimensional Euclidean vectors $\mathbf{x}$, where t, an integer, signifies a step in the sequence. Let the M_i, being variable, successively attain the values of another sequence $\{\mathbf{m}_i(t)\}$ of n-dimensional real vectors that represent the successively computed approximations of model $\mathbf{m}_i$. Here i is the spatial index of the node with which $\mathbf{m}_i$ is associated. The original SOM algorithm assumes that the following process converges and produces the wanted ordered values for the models:

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + h_{ci}(t)[\mathbf{x}(t) - \mathbf{m}_i(t)] \ , \tag{1}$$

where $h_{ci}(t)$ is called the *neighborhood function*. The neighborhood function has the most central role in self organization. This function resembles the kernel that is applied in usual smoothing processes. However, in the SOM, the subscript c is the index of a particular node (winner) in the array, namely, the one with the model $\mathbf{m}_c(t)$ that has the smallest Euclidean distance from $\mathbf{x}(t)$:

$$c = \text{argmin}_i\{||\mathbf{x}(t) - \mathbf{m}_i(t)||\} \ . \tag{2}$$

Equations (1) and (2) can be illustrated as defining a recursive step where first the input data item $\mathbf{x}(t)$ defines or selects the *best-matching model (winner)* according to Eq.(2). Then, according to Eq.(1), the model at this node as well as at its *spatial neighbors in the array* are modified. The modifications always take place in such a direction that the modified models will match better with the input.

The rates of the modifications at different nodes depend on the mathematical form of the function $h_{ci}(t)$. A much-applied choice for the neighborhood function $h_{ci}(t)$ is

$$h_{ci}(t) = \alpha(t) \exp[-\text{sqdist}(c,i)/2\sigma^2(t)] \ , \tag{3}$$

where $\alpha(t) < 1$ is a monotonically (e.g., hyperbolically, exponentially, or piecewise linearly) decreasing scalar function of t, $\text{sqdist}(c,i)$ is the square of the geometric distance between the nodes c and i in the array, and $\sigma(t)$ is another monotonically decreasing function of t, respectively. The true mathematical form of $\sigma(t)$ is not crucial, as long as its value is fairly large in the beginning of the process, say, on the order of 20 per cent of the longer side of the SOM array, after which it is gradually reduced to a small fraction of it, usually in a few thousand steps. The topographic order is developed during this period. On the other hand, after this initial phase of *coarse ordering*, the *final convergence* to

nearly optimal values of the models takes place, say, in an order of magnitude more steps, whereupon $\alpha(t)$ attains values on the order of .01. For a sufficient statistical accuracy, every model must be updated sufficiently often. *However, we must give a warning: the final value of σ shall never go to zero, because otherwise the process loses its ordering power. It should always remain, say, above half of the array spacing.* In very large SOM arrays, the final value of σ may be on the order of five per cent of the shorter side of the array.

There are also other possible choices for the mathematical form of $h_{ci}(t)$. One of them, the "bubble" form, is very simple; in it we have $h_{ci} = 1$ up to a certain radius from the winner, and zero otherwise.

4.2 Stable state of the learning process

In the stationary state of learning, every model is the average of input items projected into its neighborhood, weighted by the neighborhood function.

Assuming that the convergence to some stable state of the SOM is true, we require that the expectation values of $\mathbf{m}_i(t+1)$ and $\mathbf{m}_i(t)$ for $t \to \infty$ must be equal, while h_{ci} is nonzero, where $c = c(\mathbf{x}(t))$ is the index of the winner node for input $\mathbf{x}(t)$. In other words we must have

$$\forall i,\ E_t\{h_{ci}(\mathbf{x}(t) - \mathbf{m}_i(t))\} = 0\ . \tag{4}$$

Here E_t is the mathematical expectation value operator over t. In the assumed asymptotic state, for $t \to \infty$, the $\mathbf{m}_i(t)$ are independent of t and are denoted by $\mathbf{m}_i^*$. If the expectation values $E_t(.)$ are written, for $t \to \infty$, as $(1/t)\sum_t(.)$, we can write

$$\mathbf{m}_i^* = \frac{\sum_t h_{ci}(t)\mathbf{x}(t)}{\sum_t h_{ci}(t)}\ . \tag{5}$$

This, however, is still an *implicit* expression, since c depends on $\mathbf{x}(t)$ and the $\mathbf{m}_i$, and must be solved *iteratively*. Nonetheless, Eq.(5) shall be used for the motivation of the iterative solution for the $\mathbf{m}_i$, known as the *batch computation of the SOM ("Batch Map").*

4.3 Initialization of the models

The learning process can be started with random vectors as the initial values of the model vectors, but learning is sped up significantly, if certain regular initial values are given to the models.

A special question concerns the selection of the *initial values* for the $\mathbf{m}_i$. It has been demonstrated by [39] that they can be selected even as *random vectors*, but a significantly faster convergence follows if the initial values constitute a *regular, two-dimensional sequence of vectors taken along a hyperplane spanned by the two*

largest principal components of $\mathbf{x}$ (i.e., the principal components associated with the two highest eigenvalues); cf. [39]. This method is called *linear initialization.*

The initialization of the models as random vectors was originally used only to demonstrate the capability of the SOM to become ordered, starting from an arbitrary initial state. In practical applications one expects to achieve the final ordering as quickly as possible, so the selection of a good initial state may speed up the convergence of the algorithms by orders of magnitude.

4.4 Point density of the models (one-dimensional case)

It was stated in Subsec. 4.2 that *in the stationary state of learning, every model vector is the average of input items projected into its neighborhood, weighted by the neighborhood function.* However, this condition does not yet tell anything about the *distribution* of the model vectors, or their *point density.*

To clarify what is thereby meant, we have to revert to the *classical vector quantization*, or the k-means algorithm [19], [20], which differs from the SOM in that *only the winners are updated in training*; in other words, the "neighborhood function" h_{ci} in k-means learning is equal to δ_{ci}, where $\delta_{ci} = 1$, if $c = i$, and $\delta_{ci} = 0$, if $c \neq i$.

No *topographic order* of the models is produced in the classical vector quantization, but its mathematical theory is well established. In particular, it has been shown that the *point density* $q(\mathbf{x})$ of its model vectors depends on the *probability density function* $p(\mathbf{x})$ *of the input vectors* such that (in the Euclidean metric)

$$q(\mathbf{x}) = C \cdot p(\mathbf{x})^{1/3} \text{ , where } C \text{ is a scalar constant.}$$

No similar result has been derived for general vector dimensionalities in the SOM. In the case that (i) when the input items are scalar-valued, (ii) when the SOM array is linear, i.e., a one-dimensional chain, (iii) when the neighborhood function is a box function with N neighbors on each side of the winner, and (iv) if the SOM contains a very large number of model vectors over a finite range, *Ritter* and *Schulten* [74] have derived the following formula, where C is some constant:

$$q(x) = C \cdot p(x)^r \text{ , where}$$

$$r = 2/3 - 1/(3^2 + 3(N+1)^2) \text{ .}$$

For instance, when $N = 1$ (one neighbor on each side of the winner), we have $r = 0.60$.

For Gaussian neighborhood functions, *Dersch* and *Tavan* have derived a similar result [15].

In other words, the point density of the models in the display is usually proportional to the probability density function of the inputs, but not linearly: it is flatter. This phenomenon is not harmful; as a matter it means that the SOM display is more uniform than it would be if it represented the exact probability density of the inputs.

4.5 Border effects of the SOM

Another salient effect in the SOM, namely, in the planar sheet topology of the array, is that *the point density of the models at the borders of the array is a little distorted*. This effect is illustrated in the case in which the probability density of input is constant in a certain domain (support of $p(x)$) and zero outside it. Consider again first the one-dimensional input and linear SOM array.

In Fig. 3 we show a one-dimensional domain (support) $[a, b]$, over which the probability density function $p(x)$ of a scalar-valued input variable x is constant. The inputs to the SOM algorithm are picked up from this support at random. The set of ordered scalar-valued models μ_i of the resulting one-dimensional SOM has been rendered on the x axis.

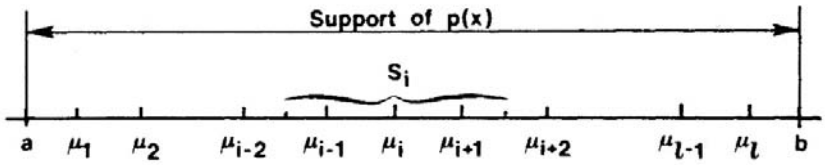


Fig. 3. Ordered model values μ_i over a one-dimensional domain.

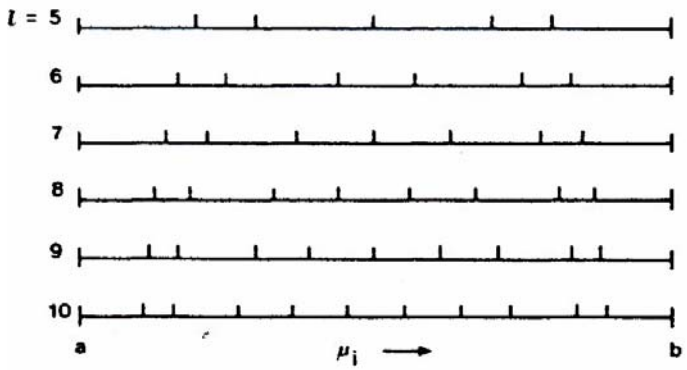


Fig. 4. Converged model values μ_i over a one-dimensional domain of different lengths.

Numerical values of the μ_i for different lengths of the SOM array are shown in Fig.4. It is discernible that the μ_i in the middle of the array are reasonably evenly spaced, but close to the borders the first distance from the border is bigger, the second spacing is smaller, and the next spacing is again larger than the average. This effect is explained by Fig. 3: in equilibrium, in the middle of the array, *every* μ_i *must coincide with the centroid of set* S_i, where S_i represents

all values of x that will be mapped to node i or its neighbors $i-1$ and $i+1$. However, at the borders, all values of x that are mapped to node 1 will be in the interval ranging from a to $(\mu_2 + \mu_3)/2$, and those values of x that will be mapped to node 2 range from a to $(\mu_3 + \mu_4)/2$. Similar relations can be shown to hold near the end of the chain for μ_{l-1} and μ_l. Clearly the definition of the above intervals is unsymmetric in the middle of the array and at the borders, which makes the spacings different.

In Sec.5 we will be dealing with two-dimensional input vectors. If their density function is constant in a square domain and zero outside it, we see similar border effects in both dimensions as what we have in Fig. 4. For other forms of the input density function the border effects are there, too, with a size that is relative to the value of the density function near the borders.

In higher dimensions we encounter similar border effects in every dimension of the input data vectors. Normally we are not concerned about them, because, especially in larger arrays, the relative fraction of the border units is small, and the model vectors are not shown. At most we may notice that the border nodes of the SOM are occupied more densely than the nodes in the middle of the array, but that may well be due to the fact that all input vectors that lie outside the model vectors in the input space will be mapped to the side or corner nodes of the SOM. On account of the border effects, some researchers have come to suggest *cyclic SOM arrays*.

However, it will be interesting to notice that *even the cyclic arrays are not free of effects that resemble the border effects*. I have in Fig. 5 a reproduced picture from my book [39] where a ring-topology SOM is trying to approximate the rectangular 2D density function. We can clearly see that *at the ends of the 2D support where the ring has the sharpest folds we have similar point density effects that are present at the ends of the linear 1D topology in Fig. 4*. These too are due to the neighborhood function. So, what is the argument for cyclic topologies?

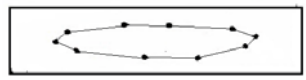


Fig. 5. Demonstration of "fold effects" in ring topology.

At the end of Sec.5 we demonstrate that the ordering strength of the sheet-topology SOM is substantial. Notice that in the k-means classification we cannot define any 2D constellation of the model vectors, and thus there are no border effects close to the convex hull of the k-means vectors. However, if we apply the SOM algorithm only at every 50th training step and let the rest of the steps be classical vector quantization steps, the SOM will anyway be materialized in the 2D form, and *the border effects will be practically nonexistent*.

5 Practical construction of a two-dimensional SOM

5.1 Construction of a 2D SOM in one training phase

The first example describes an SOM process, in which the input vectors are *two-dimensional random vectors*, and they have a *uniform distribution over a square area* in the (ξ_1, ξ_2) space. Outside this area the density function is zero.

In this example, the SOM process will consist of one phase only, which takes 1,000,000 steps, and during which the learning-rate coefficient $\alpha(t)$ defined in Eq. (3) is defined as a hyperbolically decreasing function. A step-size parameter that is a hyperbolical function of time has been shown to provide the fastest convergence in the simplest gradient-descent optimization problems., i.e., in the minimization of error functions. Our heuristic choice in the present case is the same:

$$\alpha(t) = .3/(1 + t/300000).$$

The original stepwise-recursive SOM algorithm is now written in the MATLAB code. The SOM array is rectangular and two dimensional, but since the input vectors and models are also two dimensional, we might resort to higher-dimensional arrays in computation, as we will see. However, *we start by indexing the model vectors using a one-dimensionally running index i*, and initialize their values by random numbers.

```
M = rand(64,2); % initialization of a 64-model SOM array
```

In this example the process consists of 1,000,000 steps. At each step, first, the input vector `X` is generated. After that, the index `c` of the winner node is computed. Let `Q` represent the *quantization error*, i.e., the norm of the difference of an input vector and the best-matching model. This error and the winner index `c` are computed as

```
Q = zeros(64,1);   % quantization error
for t = 1:1000000
    X = rand(1,2); % training input

    % Winner search
    for i = 1:64
        Q(i,1) = norm(X(t,:) - M(i,:));
    end
    [C,c] = min(Q);
```

Here `c` is the index of the winner. Next we have to define a *neighborhood set* around the winner. The models in this neighborhood are updated at step `t`.

Let the learning-rate coefficient α be denoted by `a` in the MATLAB code. Let `denom` be its time-dependent parameter (denominator in the expression of `a`), and let `r` stand for the time-dependent value of the half-width of the neighborhood set (when using a bubble-function neighborhood). In order to make the

SOM array explicit, let us *reshape* it such that the running index of the model vector is converted into the rectangular coordinates of the SOM array. So, first we reshape the `M` and `X` matrices into three-dimensional arrays. After that we compute the *indices of the horizontal row and vertical column* `ch` and `cv`, respectively, of the SOM array, and define the *neighborhood set* around the winner node `c`, over which the updating shall be made. The size of the neighborhood radius `r` and the learning rate `a` decrease hyperbolically in time.

```
    % Updating the neighborhood
    denom = 1 + t/300000; % time-dependent parameter
    a = .3/denom; % learning coefficient
    r = round(3/denom); % neighborhood radius
    M = reshape(M,[8 8 2]);
    X = reshape(X,[1 1 2]);
    ch = mod(c-1,8) + 1;      % c starts at top left of the
    cv = floor((c-1)/8) + 1;  % 2D SOM array and runs downwards!
    for h = max(ch-r,1):min(ch+r,8)
        for v = max(cv-r,1):min(cv+r,8)
            M(h,v,:) = M(h,v,:) + ...
                a*(X(1,1,:) - M(h,v,:));
        end
    end
```

Both `M` and `X` have to be reshaped again to their original dimensionalities, which are needed in the winner search at the next training step `t`:

```
    M = reshape(M,[64 2]);
    X = reshape(X,[1 2]);
end
```

This concludes the SOM algorithm. Its computation on a 2 GHz home computer took 117 seconds. Next we plot the model vectors by asterisk symbols (*), and connect them by horizontal and vertical auxiliary lines that link together the nodes of the SOM array. These lines are used to show which model vectors are horizontal and vertical neighbors in the SOM array. The model vectors are represented by the reshaped coordinate vector `M` in the MATLAB code.

```
plot(M(:,:,1), M(:,:,2),'k*',M(:,1,1),M(:,1,2),'k-',M(:,2,1), ...
M(:,2,2),'k-',M(:,3,1),M(:,3,2),'k-',M(:,4,1),M(:,4,2),'k-', ...
M(:,5,1),M(:,5,2),'k-',M(:,6,1),M(:,6,2),'k-',M(:,7,1), ...
M(:,7,2),'k-',M(:,8,1),M(:,8,2),'k-',M(1,:,1),M(1,:,2),'k-', ...
M(2,:,1),M(2,:,2),'k-',M(3,:,1),M(3,:,2),'k-',M(4,:,1), ...
M(4,:,2),'k-',M(5,:,1),M(5,:,2),'k-',M(6,:,1),M(6,:,2),'k-', ...
M(7,:,1),M(7,:,2),'k-',M(8,:,1),M(8,:,2),'k-',M(:,3,1), ...
M(:,3,2),'k-',M(:,4,1),M(:,4,2),'k-',0,0,'.',1,1,'.');
```

The drawing of the the points `0,0,'.',1,1,'.'` has been added in order to force the coordinates of the framed area to range from 0 to 1.

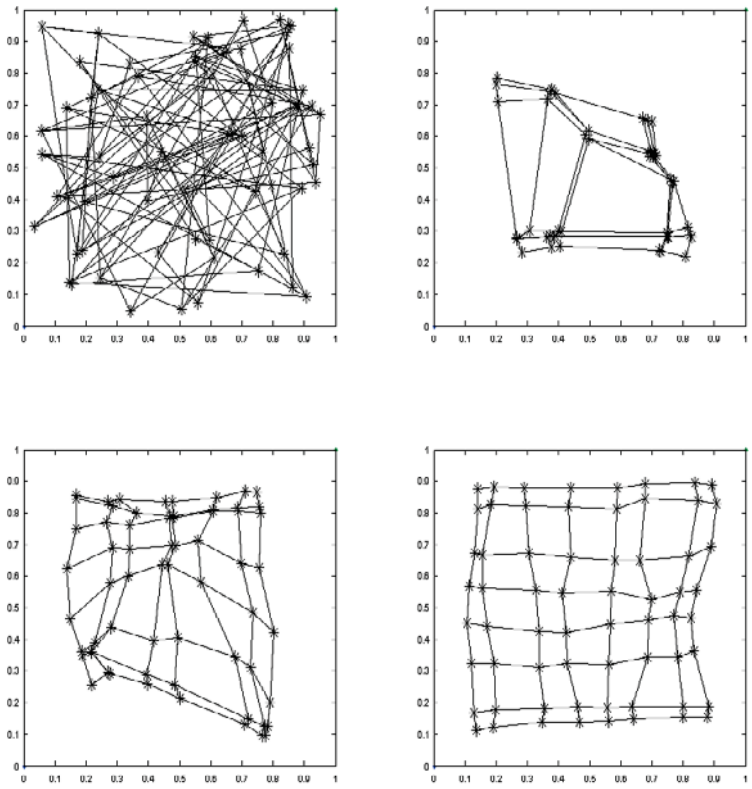


Fig. 6. A sequence of model vectors `[M(:,1) M(:,2)]` resulting in the one-phase training process after zero, 10,000, 100,000, and 1,000,000 training steps, respectively. The model vectors have been drawn by the asterisk symbols. The models that are horizontal or vertical neighbors in the SOM array have been connected by the network of auxiliary lines, to show their mutual order. The fourth subimage is directly produced by the above script, the rest have been sampled during the process.

In Fig. 6 we plot the model vectors in the `(M(:,1), M(:,2))` coordinate system and represent them by asterisk symbols (*), which are linked by auxiliary lines in order to indicate which model vectors are horizontal and vertical neighbors in the SOM array. So this is a representation of the model vectors *in the input space*. The density function of the input vectors was constant in the framed square area and zero outside it. On the top left we see the random model vectors as they were initialized, and then *a sequence of intermediate states of the SOM process after 10,000, 100,000 and 1,000,000 training steps, respectively*. The above script produces the fourth image directly, and the rest have been sampled during the process. There are still random deviations left after 1,000,000 steps; however, the two-phase process to be explained next will provide a slightly smoother and accurate organization result. Also, if we had used

the *Batch Map algorithm* as will be explained later, the convergence would have taken place faster and more robustly.

5.2 Coarse and fine training phases

It has turned out that we can obtain a more accurate equilibrium state if *two training phases, a coarse and a fine one* are used. If α were changing all the time like in the one-phase process, the convergence of the process would be slower, because the radius of the neighborhood function depends on time. However, if we apply a much shorter *coarse training process*, during which the topographic order of the $\mathbf{m}_i$ is formed, and continue training with a *fine training process*, in which the neighborhood function is narrow but *constant in time*, we obtain a more accurate final state with the same total number of training steps, as shown in Fig. 7.

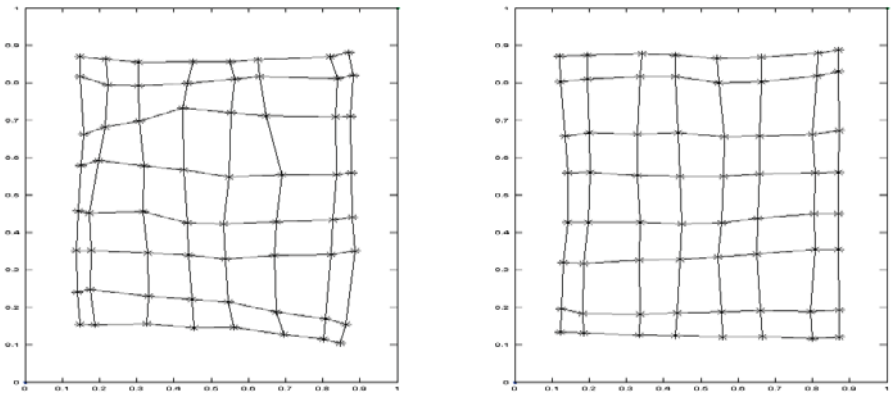


Fig. 7. The same process as described in Fig. 6, but the neighborhood function was made to decrease only during the coarse training phase, which in this example consisted of 100,000 training steps. After that the neighborhood function was kept constant (containing only the nearest nodes) for 900,000 steps. Left image: The SOM after the coarse training phase. Right image: The SOM after fine training, whereupon the total number of training steps was the same as in the previous example, namely, equal to 1,000,000.

5.3 Compensation for the border effects

It may have to emphasized that in practical problems, in which the dimensionality of input vectors is higher than two, we usually don't show the model vectors. Instead we usually show the *nonlinear projections of the input vectors on the 2D SOM array*. Therefore the border effects, with an appreciably-sized SOM, do not become visible and do not play any significant role in visualization. I have

come to think of one particular application only, namely, the *demodulation of quantized signals* to be discussed in the next section, in which *the model vectors define the discrimination levels for the demodulation of quantized signals*, and in which the elimination of the border effects is of utmost importance.

An attempt to fine tune the SOM by k-means clustering. The k-means clustering has been used extensively as an adaptive method in digital signal transmission. It has the particular advantage that it has no border effects in quantization; on the other hand, *it has no ordering power neither.* Also, although the initial values were topographically ordered, this order may be lost partly in fine tuning by the k-means clustering.

In the above script, the k-means clustering process is obtained by setting the neighborhood radius `r` to the value zero.

The result of this experiment is shown by the right-hand subimage of Fig. 8. In the first instance, the k-means clustering is trying to *minimize the quantizing error*, which happens *when the constellation of the models is a hexagonal grid in the input space*; that is clearly what is happening in Fig. 8.

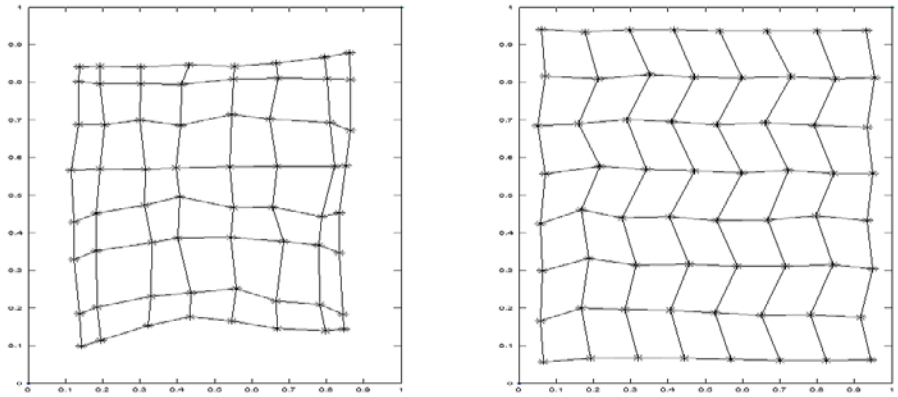


Fig. 8. The same process as described in Fig. 7, but in fine training, 900,000 steps of the *k-means clustering* algorithm were used. The constellation of the models now tends to be *hexagonal*, which minimizes the mean quantization error. The coarse training, 100,000 steps, was similar as in Fig. 7. Notice that with a different sequence of random numbers the result of coarse training is slightly different.

Compensation for border distortion by a mixture of SOM and k-means algorithms. A compromise between accurate adaptation and self-ordering is obtained if most training steps are made by the k-means clustering, but, say, every 50th step is an SOM step. Even such rare SOM steps will be able to stabilize the topographic ordering and smooth the data effectively, while the k-means clustering steps compensate for the border effects, as shown in the

right-hand subimage of Fig. 9. Although the k-means algorithm was applied for 98 per cent of the steps, nonetheless it was not able to minimize the mean quantization error, which needs the hexagonal constellation. The effect of the rare SOM steps is so strong that in the first instance they try to keep the grid form as square, because it was so determined by the neighborhood function. So, if one is really concerned about the border effects, one might try this same idea also in all of the other applications.

Comment. If we had used the *Gaussian neighborhood function* and a usual two-phase training, and if we had defined its radius (during the 100,000 steps in this example) like above, but during the final 900,000 steps we had kept the radius at a value that is significantly smaller than unity (but at any rate greater than zero), we would probably have obtained the same result as in Fig. 9.

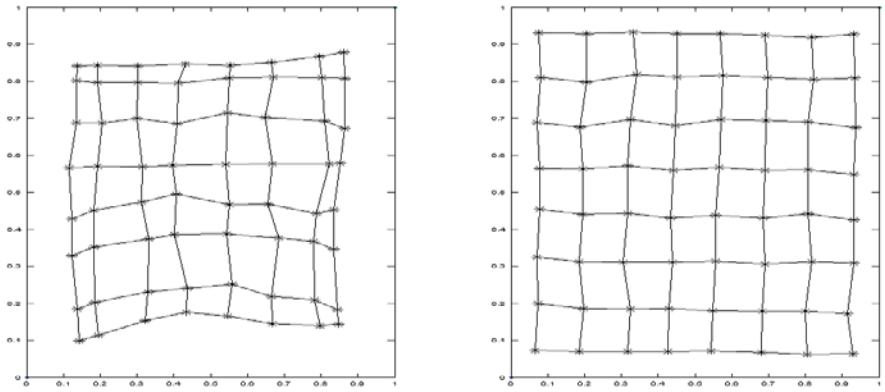


Fig. 9. The coarse training, 100,000 steps, was similar as in Figs. 7 and 8. In fine training, *a mixture of k-means clustering and SOM training steps was used*. The k-means algorithm tends to approximate the input density function without border effects, but the SOM algorithm, although applied only for two per cent of the training steps, overrides it, provides for more effective convergence and takes care of keeping the model vectors as a topographically ordered square grid.

6 Preliminary example: Demodulation of TV signals

6.1 The task

Before we start with the SOM Toolbox software package and its application to various practical problems, which is the main topic in this book, it may be helpful to discuss a very simple, low-dimensional application of the SOM. It is used in the demodulation of TV signals, and at the same time we will be able to gain a very concrete understanding of the operation of the SOM.

Practically all telecommunications of today are based on *quantized signals and their digital processing*. However, instead of using streams of *bits* as usual in wired transmission, the capacities of wireless communications are far more effectively utilized if the *analog signal values* are approximated by *several quantized reference levels*. For instance, in the transmission of TV signals, the so-called *QAM (quadrature-amplitude modulation)* is nowadays a standard.

Fig. 10 represents the principle of the sixteen-level quadrature-amplitude modulation, abbreviated *16QAM*. There are two independent carrier waves that have the same frequency but a mutual phase shift of 90 degrees. They are transmitted concurrently through the same frequency channel. Each one of these carrier waves is modulated by four quantized discrimination levels: one wave by the *in-phase (I)* components, and the other wave by the *quadrature (Q)* components, respectively. A combination of these partial waves can encode and decode 16 bits concurrently.

In the contemporary TV techniques, using 16 quantization levels in each subchannel, one is able to implement a 256QAM, or 256-bit parallel transmission.

Since the amplitude modulation must anyway be accomplished by analog means, various disturbances and variations of signal amplitudes cause many kinds of distortions to the signal levels. In radio traffic the signals may reach the receiver through multiple (reflected) paths. Nonetheless the signals can mostly be kept *clustered*. In Fig. 10 we see two kinds of nonlinear *distortions* of the signal levels and their relative phases, resulting in typical QAM.

The I and Q signals are demodulated by discriminating them in the 16QAM into four levels each, using *four reference levels*. Obviously the main problem in the compensation of distortions is to *let the reference levels to follow up the centroids of the clusters*. This could be accomplished by the k-means clustering (vector quantization) algorithm, but some extra stabilization would then be needed. A new idea is to use the SOM for the same purpose, because no extra stabilization is then necessary, and the convergence of the SOM algorithm is significantly faster and more robust than that of the vector quantization.

6.2 Using the SOM as an adaptive demodulator

Now we will now see in what way the SOM can accomplish this same task effectively. In the practical example we use the 64QAM which quantizes the analog signal values into 64 clusters. The input data consist of a stream of noisy

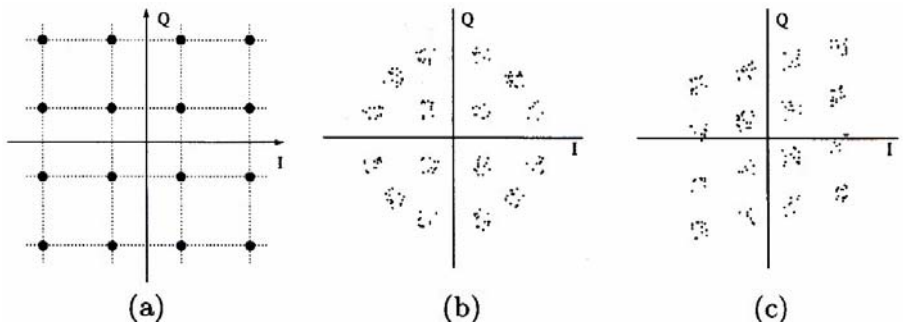


Fig. 10. (a) Ideal signal constellation of a sixteen-level quadrature-amplitude modulation (16QAM) used in digital communications systems when in-phase (I) and quadrature (Q) components occupy discrete lattice points in the IQ coordinate system. (b) Typical "corner collapse" distortion, in which the corner clusters are shifted towards the center, due to saturation of amplitudes at high signal level.. (c) The "lattice collapse" distortion, in which the relative phase of the I and Q components is changed, resulting in an oblique lattice.

two-dimensional samples $\mathbf{x}(t) = [\xi_1(t), \xi_2(t)]$ labeled by the sampling time t and clustered as shown on the left of Fig. 11.

There are 64 two-dimensional model vectors, which are also functions of time: $\mathbf{m}_i(t) = [\mu_{i1}(t), \mu_{i2}(t)], i = 1, 2, ..., 16$. These model vectors are computed by the mixture of the k-means clustering and SOM algorithms discussed earlier. This method keeps the model vectors topographically ordered all the time and does not exhibit any significant border distortions. The right-hand side of Fig. 11 shows how the noisy signals are discriminated into 64 clusters. In the right image the lattice points have adaptively converged into a closely optimal constellation, and the discrimination levels between them have been computed by the *Voronoi tessellation* [93]. The lines of this tessellation separate the clusters optimally, every signal into its closest lattice point.

We may still be interested in seeing the program code of this simulation. It starts with the definition of the *64 ideal lattice points* used in the quantization of the signals. We may imagine that the lattice points are ideal at the sending end, and the noise comes from transmission only. Their values range equidistantly from -3.5 to +3.5 units in each channel:

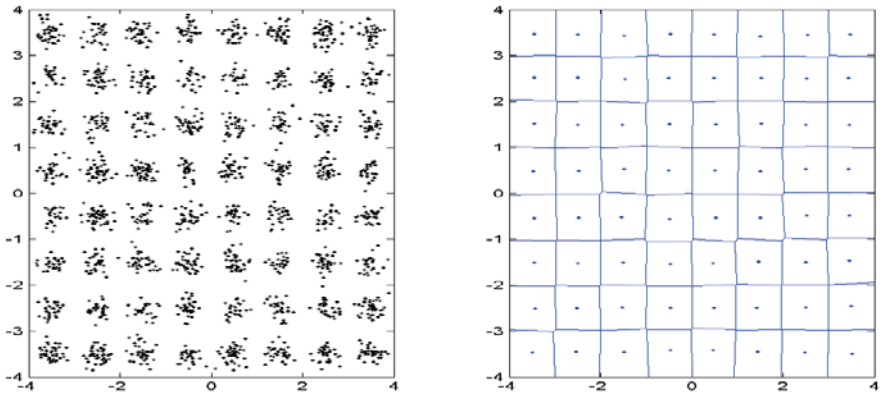


Fig. 11. Left image: Noisy signal constellation in a 64-level quadrature-amplitude modulation (64QAM) used in digital communications. Right image: Voronoi diagram, which shows the two-dimensional *discrimination levels* used to demodulate the QAM signals. Any of the clusters in the left image will be discriminated into one of 64 discrete sets.

```
M = zeros(64,2);

M(:,1) = [1 3 5 7 9 11 13 15 ...
          1 3 5 7 9 11 13 15 ...
          1 3 5 7 9 11 13 15 ...
          1 3 5 7 9 11 13 15 ...
          1 3 5 7 9 11 13 15 ...
          1 3 5 7 9 11 13 15 ...
          1 3 5 7 9 11 13 15 ...
          1 3 5 7 9 11 13 15 ];

M(:,2) = [1  1  1  1  1  1  1  1 ...
          3  3  3  3  3  3  3  3 ...
          5  5  5  5  5  5  5  5 ...
          7  7  7  7  7  7  7  7 ...
          9  9  9  9  9  9  9  9 ...
         11 11 11 11 11 11 11 11 ...
         13 13 13 13 13 13 13 13 ...
         15 15 15 15 15 15 15 15];

for i = 1:64
        M(i,:) = (M(i,:)-8)/2;
end
```

After that, 100,000 samples of noisy (received) signal values `X` are defined.

```
X = zeros(1,2);
```

```
% Winner search
for t = 1:100000
    X(1,1) = floor(8*rand) + .15*randn -3.5;
    X(1,2) = floor(8*rand) + .15*randn -3.5;
```

Now we start adapting the reference values of the QAM to the received signals, using a constant learning-rate coefficient; since we want to make this system *continuously adapting to new signals*, we do not want the coefficient change in time, and we keep it at a tentative small value `a = .01`.

```
    % Winner search
     Q = zeros(64,1);
     for i = 1:64
          Q(i,1) = norm(X(1,:) - M(i,:));
     end
     c = min(Q);

    % Updating
    M = reshape(M,[8 8 2]);
    X = reshape(X,[1 1 2]);
    ch = mod(c-1,8) + 1;
    cv = floor((c-1)/8) + 1;
    if mod(t,50) == 0  % for every 50th input
       r = 1; % SOM neighborhood radius
    else
       r = 0; % k-means clustering step
    end
    a = .01; % learning rate
    for h = max(1,ch-r):min(8,ch+r)
         for v = max(1,cv-r):min(8,cv+r)
              M(h,v,:) = M(h,v,:) +  a*(X(1,1,:) - M(h,v,:));
       end
    end

    M = reshape(M,[64 2]);
    X = reshape(X,[1 2]);
end
```

The final two lines were needed to reshape `M` and `X` for the next winner search. This concludes the adaptation algorithm. What we still need are the discrimination levels shown at right in Fig. 11:

```
figure(1);
for subimage = 1:2
    subplot(1,2,subimage)
    if subimage == 1
       for t = 1:64
```

```
            Y(t,1) = floor(8*rand) + .15*randn-3.5); % Y = X
            Y(t,2) = floor(8*rand) + .15*randn-3.5);
        end
        plot(Y(:,1),Y(:,2),'k.')
    end
    if subimage == 2
      voronoi(M(:,1),M(:,2))
    end
end
filename  'QAM';
print('-dpng', [filename '.png']);
```

Next we show that the SOM lattice, starting from its standard position in Fig. 11, *will follow up to the collapsed signals* and produce new, optimal discrimination levels. It will do that in 10,000 training steps, as demonstrated in Fig. 12.

The collapsed inputs `X`, which contain an even larger noise amplitude `.2*randn`, can be simulated by the following lines:

```
for t = 1:10000
    E = floor(8*rand);
    F = floor(8*rand);
    X(1,1) = E + .2*randn -3.5;
    X(1,2) = .08*E + F + .2*randn -3.75;
```

In order to fit the noisy signals (with outliers) to the same size of a frame as the Voronoi diagram of the SOM models, we cut the `Y` signals by the instruction:

```
Y = Y(find(Y(:,1)>-4 & Y(:,1)<4 & Y(:,2)>-4 & Y(:,2)<4),:);
```

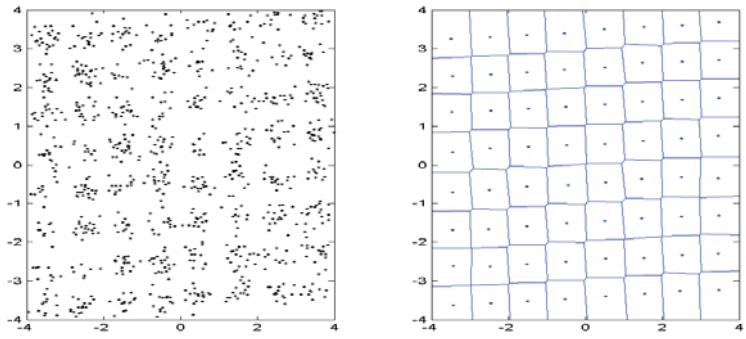


Fig. 12. Left subimage: Noisy signal constellation, where the lattice of QAM modulation is further deformed like in "lattice collapse.". Right subimage: Voronoi diagram of the deformed QAM signals.

7 Batch computation of the SOM

7.1 The algorithm

In practice, the batch training algorithm is usually preferred.

In this section we discuss the *batch computation* of the SOM. If the input vectors were Euclidean, the principal-component method is recommended for the initialization. For general distance measures the random initialization is always possible; the optimization of initialization for general metrics, however, is very tricky, and cannot be discussed here.

In this guide we introduce a *batch computation process* for the construction of the SOM. There are several reasons for it: 1. There is no time-variable learning-rate parameter $\alpha(t)$ in it. 2. The batch algorithm converges faster than the stepwise "gradient" method. 3. If a two-phase learning is used, as described in the next subsection, if the neighborhood function during the fine-tuning phase is held constant, and if the set of training inputs is the same at each iteration, *the algorithm terminates exactly in a finite number if iterations*, which can be utilized for the stopping of computation. 4. We will later see that the batch algorithm can be generalized for *non-vectorial data*, too.

Consider Fig. 13, where a two-dimensional hexagonal array of nodes, depicted by the circles, is shown. With each node i, a model $\mathbf{m}_i$ is associated.

Also a *buffer memory*, depicted by a rectangular symbol, is associated with each node. In the beginning of *one training cycle* each buffer first *sums up* the values of all of those input vectors $\mathbf{x}(t)$ that are mapped to this node, and also stores the number of addends.

The updating of the $\mathbf{x}(t)$ is now made in the following way. In this illustrative example we assume a "bubble" neighborhood function that has the value 1 in a *neighborhood set* N_i of nodes, consisting of the nodes up to a certain distance from node i, and is equal to zero otherwise. In Fig. 13 only the nearest neighbors of the winner belong to N_i. According to Eqs.(4) and (5), when using this simplified neighborhood function h_{ci}, the equilibrium value of every model must coincide with the *mean* of the $\mathbf{x}(t)$ falling into its neighborhood set N_i. We try to approach to this equilibrium state in an iterative way. In one cycle of iteration, we first compute the *sum* of all $\mathbf{x}(t)$ over N_i, that is, the sum of all of the *partial sums* that have been accumulated in the buffers of N_i. Then we divide this total sum by the total number of input vectors mapped to N_i, taken from the buffers. A similar mean is computed for every node i, i.e. over the neighborhoods around all of the nodes. *Updating* of the $\mathbf{m}_i$ then means that the old values of the $\mathbf{m}_i$ are replaced by the computed means *in one concurrent computing operation over all nodes of the array.* This kind of an iterative cycle takes us closer to the equilibrium, and concludes one training cycle.

Training cycles of the above kind are repeated, always first clearing all buffers and thereafter distributing new copies of the input vectors under those nodes, the (updated) models of which match best with the new input vectors. New means over the N_i are computed and made to replace the old $\mathbf{m}_i$, and these

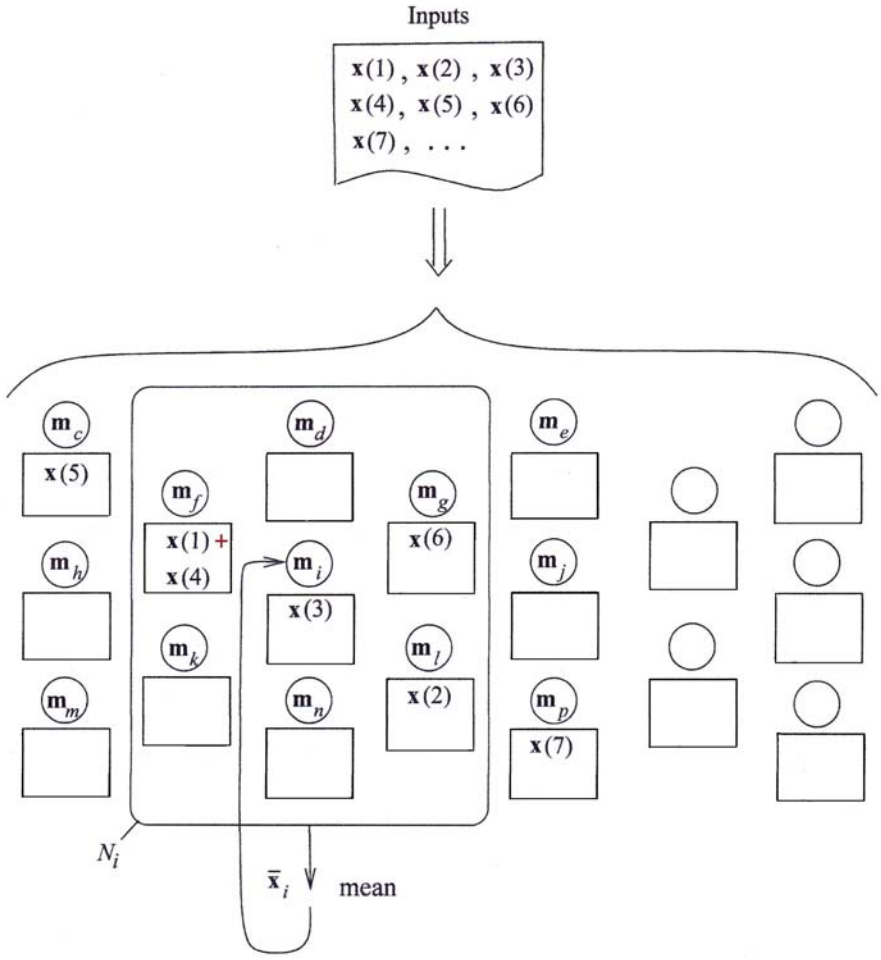


Fig. 13. Illustration of one cycle in the batch-training process. The input data items $\mathbf{x}(t)$ are first distributed into the buffer memories (rectangular symbols) associated with their best-matching models, and summed up there with the earlier contents (like $\mathbf{x}(1) + \mathbf{x}(4)$ at node $\mathbf{m}_f$). The number of addends is also stored with the sum in the buffer. According to Eqs.(4) and (5), when using the definition of the simplified neighborhood N_i, the equilibrium value of every model must now be the *mean* of the vectors $\mathbf{x}(t)$ over the corresponding neighborhood set N_i. Therefore we form the total sum of the partial sums in N_i and divide this total sum by the total number of addends in N_i. Such sums, computer for every node, then replace all of the old values $\mathbf{m}_i$ in one concurrent operation over the whole SOM array. This training cycle is repeated iteratively, always first clearing up all buffer memories and replacing the old means by the new means, until the wanted equilibrium is reached.

cycles are repeated until the wanted equilibrium is reached. Usually we need a few dozens of iterative cycles.

A process that complies even better with the stepwise recursive learning is obtained if the means are formed as *weighted averages*, where the weights are related to each other like the h_{ci}. Here c is the index of the node, the model of which is updated, and i stands for the indices of the nodes in its neighborhood.

A discussion of the convergence of the batch-computing method has been presented in [8].

7.2 Two training phases, a coarse and a fine one

In practice, the training process usually consists of two phases: coarse and fine training, respectively.

It has still to be emphasized that so far there does not exist any mathematical definition of the optimal training process. A successful use of the SOM is only based on a large amount of experience, and the advice given here is also based on a large number of observations from practice. The recommendations given below are believed to work in usual applications and for SOM arrays of modest size (up to a few thousand nodes), and when this advice has been followed, no problems have been encountered in practice.

It has been realized ever since the introduction of the SOM that the width of the neighborhood function cannot be selected too small in relation to the dimensions of the SOM array, because then the ordering may be disturbed and various kinds of "folds" to the map may be formed and left permanently. This handicap is an intrinsic property of the nonlinear process on which the SOM is based. However, if the width of the neighborhood function is initially large enough and constant, say, on the order of 20 per cent of the larger dimension of the SOM array or more, these "folds" will be smoothed out in a modest number of training cycles. Then, however, although the SOM seems to be globally ordered, the map is still too "stiff" and cannot describe the fine details of the input data. Nonetheless, if the "folds" have once been smoothed out, they will never appear again, however long the training is continued.

Narrowing of the neighborhood function in coarse training. Imagine now that the training is made to consist of successive major phases, each consisting of several training cycles. During each phase the neighborhood function is held constant, but its width is made monotonically smaller in the consecutive phases. If the neighborhood width is reduced sufficiently slowly, the global order achieved in the first phase will not be destroyed in continuation. On the other hand, since the neighborhood function is narrower during the successive phases, the ordering starts to take place at a finer resolution. Finally the SOM will be smoothed out globally, with increasing resolution, and no folds will appear. This effect was already found in the first SOM experiments around 1982.

It is possible to use a single training phase, during which the width of the neighborhood function is made a function of the training step. However, a quicker and more accurate convergence is obtained, if a limited number of training cycles are divided into coarse and fine training phases.

Convergence in a finite number of cycles. Another, more recent experimental observation is that if the neighborhood function is held constant during the last iterations, whatever its width is, and the same input data are applied iteratively, the ordering process will be stabilized (converge) *in a finite number of training cycles*. So far we have found no exceptions to this observation.

Based on plenty of experiences , we have thus decided to use *two main training phases, a coarse and a fine one*. During the coarse-training phase, the width of the neighborhood function is made to decrease linearly from an initial value that is, say, 20 per cent of the longer side of the SOM array, to a final value which is not smaller than, say, five per cent of the shorter side (but at least a half array spacing). With modest sizes of the SOM arrays (say, at most a few thousands of nodes), the number of batch training cycles required during this phase is usually not more than a few dozen.

During the fine-training phase the neighborhood function shall have the value that was used last in coarse training, and is held *constant*. If the training set is the same at each iteration, the fine training phase is continued until the corrections to the SOM weight vectors become zero. This usually occurs in less than a few dozens of cycles. The SOM process has then reached a stable state exactly.

7.3 Dot-product maps

The similarity of two metric vectors is often expressed as their dot product.

For metric vectors, a practical computation of the SOM may be based on their *dot products*. For Euclidean vectors this method is particularly advantageous, if there are plenty of zero elements in the vectors, because they are skipped in the evaluation of similarities. However, the model vectors $\mathbf{m}_i$, for their comparison with the input $\mathbf{x}$, must be kept normalized to constant length all the time. Instead of eq.(2), the index of the winner location is now defined by

$$c = \text{argmax}_i\{\text{dot}(\mathbf{x}, \mathbf{m}_i)\} \ . \tag{6}$$

Since the computation of the SOM is in practice carried out by the batch algorithm, the mapping of all of the input items onto the respective winner nodes (i.e., the associated lists) is made in a similar way as described before.

A normalization of the $\mathbf{m}_i$ to constant length shall be made after each iteration cycle.

It must be mentioned that the SOM Toolbox does not have provisions for computing the dot-product maps.

8 Various measures of distance and similarity

The distance between two vectors is a measure of their dissimilarity. Inner products, especially the dot product are direct measures of similarity. More specific measures of distance and similarity are defined in this section.

There exist many versions of the SOM, which apply different definitions of "similarity." This property deserves first a short discussion. "Similarity" and "distance" are usually opposite properties.

If we deal with concrete objects in science or technology, we can then base the definition of *dissimilarity* on basic mathematical concepts of, say, *distance measures between attribute vectors*. However, if the attributes have a different physical or other nature, different *units* are used for them, and then their values are expressed in *different scales*. The scales must first be *normalized* as discussed in the next paragraph. After that, eventually, the various features must be *weighted* by information-theoretic measures, or experimentally selected factors to emphasize particular features in self organization. Such weighting is often necessary when analyzing problems in which human decisions are involved, e.g., problems in finance. On the other hand, in science and technology, normalization of the scales without weighting is often a sufficient strategy.

Sets of various *indicators* collected in statistical studies are usually also expressed as real vectors, consisting of numerical results or other statistical data, which have to be normalized, too.

In scientific problems, various kinds of spectra and other *transformations* can be regarded as multidimensional vectors of their components.

Scaling of features. For normalization, the simplest method is to *rescale* the variables so that either their *variances*, or their *maxima and minima*, respectively, become identical. After that, some standard distance measure, such as the Euclidean, or more generally, the Minkowski distance, etc., can be tried, the choice depending on the nature of the data. It has turned out that the Euclidean distance, with normalization, is already applicable to most practical studies, since the SOM is able to display even complex interdependencies between the variables in its display. The *local magnification* of the SOM areas depends mainly on the *density function of the corresponding input items*, and not so much on the metric chosen to describe them.

Inner products and normalization. A natural measure of the *similarity* of vectorial items is in general some *inner product*. In the SOM research, the *dot product* is frequently used, especially in very large SOMs. This measure also complies better with the biological neural models than the Euclidean distance. However, all of the the model vectors $\mathbf{m}_i$ of the SOM, for their correct comparison with the same input $\mathbf{x}$, must then be kept *normalized to constant length* all the time. If the vector dimensionality is high, and also the input vectors are

normalized to constant length, the difference between SOMs based on the Euclidean distances and the dot products is insignificant. (For the construction of Euclidean and dot-product SOMs, cf. Subsections 7.1 and 7.3, respectively.) On the other hand, if there are plenty of zero elements in the vectors, the computation of dot products is correspondingly faster, because the zero elements can simply be skipped. This property was utilized effectively especially in the fast computation of *document maps* [31].

Natural variations in pictures. Before proceeding further, it will be necessary to emphasize a basic fact. A *picture*, often given as a set of pixels or other structural elements, will usually not be applicable as such as an input vector. The natural variations in the pictures, such as translations, rotations, variations of size, etc., as well as variations due to different lighting conditions are usually so wide that a direct comparison of the objects on the basis of their appearances does not make any sense. Instead, the classification of natural items shall be based on the extraction and classification of their characteristic *features* which must be as *invariant* as possible. Features of this type may consist of color spectrograms, expansions of the images in Fourier transforms, wavelets, principal components, eigenvectors of some image operators, etc. If one can describe the input objects by a restricted set of such *invariant features*, the dimensionality of the input representations, and the computing load are reduced drastically.

The selection of a characteristic set of features and their automatic extraction from the primary observations must often be based on heuristic rules. In biology, various feature detectors have been developed in a very long course of evolution.

Structural features. Especially in the recognition and understanding of images, various *artificial-intelligence methods* have been developed in order to achieve the highest possible degree of invariance with respect to different *image transformations* such as lighting conditions, shades, translations, rotations, scales, of deformations. The first step is *segmentation* of a picture into areas in which the shade of gray or color is homogeneous, and *edge detection* in which steep changes of shades are identified. Typical *forms* from these segments are identified by comparison with previously recorded forms, and so a collection of *primitives* for the picture are found. The topological relations between the primitives are *parsed* by a *picture grammar*, and the result is usually presented as a *parsing tree*. A parsing tree can be converted into a coded entity, and by the comparison of the parsing tree with a collection of standard parsing trees one is able to identify simultaneously occurring objects in the pictures, and to classify them invariantly with respect to various transformations and distortions.

The application of the SOM to structural recognition is still in its infancy, and we do not discuss any problems of that type in this book.

.

Features of texts. For more complex comparisons one may also look for other kinds of features to be used as the vector elements. For instance, in *text*

analysis, complete *documents* can be distinguished from each other based on their word statistics, e.g., *word histograms*, whereupon very careful attention must be paid to the relative occurrence of the words in different texts; cf. [75]. So, the elements of the histogram, corresponding to the various words, must be *weighted* by multiplicative factors derived from the relative occurrences of the words. For weighting, one can use the statistical *entropy* (actually, *negentropy*) of a word, but the words in histograms can also be weighted (and thus, rare and very common words can be ignored) based on their *inverse document frequency (IDF)*. The "document frequency" means in how many documents in a text corpus a particular word occurs, and IDF is the inverse of this figure. With proper weighting, the word histograms, which constitute the feature vectors, can be restricted to, say, some hundreds of dimensions.

Features of symbol strings. The *strings of symbols* constitute another common type of variables. Except in texts, string variables occur, e.g., in bioinformatics and organic chemistry: in genetic codes, sequences of atoms in macromolecules etc.: cf., e.g., [65] and [42]. Normally the strings are of very different length. Some kind of *edit distance*, i.e., the number of elementary editing operations needed to transform one string into the other is a very effective definition of the distance between string variables. These operations must normally be weighted based on the statistics of the various errors. For very long strings, such as the protein sequences, some heuristic shortcut computations of distances such as those applied in the wide-spread FASTA method ([67] and [68]) may be necessary. Such distance measures have often been precomputed in the databases.

Contextual similarity. There also exist other, more abstract kinds of similarity measures. One of them is the *contextual similarity of words*. Consider a word in a text, within the context of its neighboring words. If each word in the vocabulary is represented by a random code, the mutual correlations between the *representations of the words* remain very small. However, the measure of similarity of two *local contexts*, e.g., triplets of three successive words in the text, then ensues from the *occurrence of the same random codes in identical positions* in the triplets. Analyses of the semantic values of words can be based on contextual-similarity studies, and very deep linguistic conclusions can be drawn from such analyses, as demonstrated in [45].

Dynamical features. An important task is to compare *dynamic phenomena*. This becomes possible, if the models are made to represent *dynamic states*. A very important discussion of dynamic SOMs has been presented by [21].

Various *transformations in the time scale* constitute an elementary category of dynamic features. For instance, at the end of this book we have Section 32 in which the SOM is regarded as a filter bank for *temporal features*.

9 A view to SOM software packages and related algorithms

The SOM discussed here is not the only version of self-organizing maps, but especially its batch computation is believed to be fastest, especially for very large maps.

The basic self-organizing map (SOM) principle as discussed in this book has been used extensively as an analytical and visualization tool in exploratory data analysis. It has had plenty of practical applications ranging from industrial process control and finance analyses to the management of very large document collections. New, very promising applications exist in bioinformatics. The largest applications so far have been in the management and retrieval of textual documents.

SOM software packages. Several commercial software packages as well as plenty of freeware on the SOM are available. This author strongly encourages the use of two public-domain software packages developed by the team of our laboratory: the SOM_PAK [81] and the SOM Toolbox [82]. Both packages contain auxiliary analytical procedures, and especially the SOM Toolbox, which makes use of the MATLAB functions, is provided with good and versatile graphics as well as thoroughly proven statistical analysis programs of the results.

This book will mainly apply the basic version of the SOM. Nonetheless there may exist at least theoretical interest in different versions of the SOM, where some of the following modifications have been introduced.

The SOM_PAK. Our first SOM software package, the SOM_PAK was published in the late 1980ies. It was written in C++, and meant for a professional package for big problems: the C++ implementation is significantly faster than MATLAB. On the other hand, although it had scripts for defining the SOM arrays, it did not contain complete graphic programs of its own, a so the users had to resort to graphic tools of their own. Neither does it contain the batch training algorithm.

The construction of the SOM in SOM_PAK is carried out by *command lines*, like the `som_lininit, som_randinit, som_seqtrain` and `som_batchtrain` functions of MATLAB. An example of initialization is

lininit -xdim 16 -ydim 12 -din file.dat -cout file.cod -neigh gaussian -topol hexa

Here *-din* is the input data file and *-cout* the model data file; the rest may be self-explanatory. A typical simple training sequence is defined by

vsom -din file.dat -cin file1.cod -cout file2.cod -rlen 10000 -alpha 0.03 -radius 10

where *cin* is the file where the initialized models are taken from, *-rlen* is the number of training steps, *-alpha* is the value of the initial training-rate parameter α which decreases to zero during learning, and *-radius* is the initial neighborhood radius which decreases to one during training.

Various topologies of the SOM array. In this book, the SOM array is mostly taken as two dimensional, regular, and hexagonal. This form of the array is advantageous if the purpose is to *visualize* the overall structure of the whole data base in one image. One of the different versions of the array, developed, e.g., by the BLOSSOM Team in Japan in Tottori University [80], is *spherical*. Such cyclic "topologies," cylindrical, spherical, or toroidal, may have some meaning if the data themselves have a cyclic structure, or if the purpose is to avoid border effects of the noncyclic array of nodes. This may be the case if the SOM is used for process control, for the continuous and homogeneous representation of all possible process states.

The SOM Toolbox contains parameters in the initializing functions by which the topology of the array can be defined as a straight sheet, a cylinder, or a toroid.

Another, often suggested version of the SOM is to replace the regular array by a *structured graph* of nodes, where the structure and the number of nodes are determined dynamically; cf., e.g., [18].

Other mathematical principles. Then, of course, there arises a question whether one could define a SOM-like system based on quite different mathematical principles. One of the interesting suggestions is the *generative topographic mapping (GTM)* introduced in [6]. It is based on direct computation of the topological relations of the nodes in the array. A different, theoretically deep approach has been made by [88], using information-theoretic measures in the construction of the SOM topology.

Perhaps one of the main virtues of the basic SOM algorithm is that one can compute really large mappings in reasonable time, using only personal computers.

Clustering methods. Finally we must remind that the traditional methodology for the representation of similarity relations between data items is to *cluster* them according to some similarity or distance measure. The classical clustering algorithms as described by [3], [22], [30], and [86], however, are usually rather heavy computationally, since every data item must be compared with all of the other ones, maybe reiteratively. For masses of data this is obviously time-consuming. The remedy provided by the SOM is to represent the set of all data items by a much smaller set of *models*, each of which stands for a subset of similar or almost similar data items. One also has to realize that the SOM forms a nonlinear projection of the input density function onto the SOM array, whereupon the similarity relationships between all data items in the input data base become explicit, which is not due in usual clustering methods.

10 The SOM Toolbox

The main reason for concentrating on a special software package, called the SOM Toolbox, is that it contains all of the main SOM functions and good graphics tools in a concise form. We strongly recommend this program package, developed in our laboratory, because it is based on a long experience and justified by many demanding projects. It has been intended for professional use and contains a lot of auxiliary diagnostic programs. Since this guidebook is intended for a starter, we shall concentrate on the basic SOM functions.

10.1 General

In order to obtain a "hands-on" experience of the SOM algorithm, many people like to program it themselves. In general this is a good practice, and if you like it, you are welcome to do it first. However, the SOM algorithm defines a highly nonlinear dynamic process, and like many other nonlinear dynamic phenomena, it may behave in unexpected ways, depending on the training sequences and parameters defined. The main reason for recommending the use of readily available SOM software packages is that there are many details in them chosen after a long experience. In good software packages there are also programs for the monitoring of the training process and diagnostic programs for the checking and testing of the results.

One of the widest-spread SOM software packages is the *SOM Toolbox* developed in our laboratory. It is compatible with the MATLAB, so it can utilize all of the functions of the latter, including versatile graphics and diagnostic programs. A lot of auxiliary functions, not only for the SOM itself, but also for auxiliary tools have been developed by ourselves.

The SOM Toolbox was created for very pragmatic reasons. The SOM algorithm had been used by us in many cooperative projects, with industry as well as in financial applications, and the first SOM software package, the SOM_PAK, was also developed in our laboratory. We needed a good standard platform for experimentation as well as for industrial implementations. Also good general-purpose visualization tools were needed. The first version of the SOM Toolbox was released in 1996, and is has been updated a few times since then, taking the development of the MATLAB into account [89] [90] [91].

How to get the SOM Toolbox? The SOM Toolbox can be downloaded freely from the Internet:

http://www.cis.hut.fi/projects/somtoolbox/documentation/;
http:// www.cis.hut.fi/projects/somtoolbox/package/papers/techrep.pdft.

It needs the MATLAB version 5 or higher, and a GUI interface is also needed. It is a public-domain software with very mild restrictions to commercial application. For scientific purposes it can be used completely freely.

10.2 The SOM scripts

In this subsection we define the most essential features of the SOM scripts and introduce the most central SOM Toolbox functions.

A MATLAB *script* is a complete program code that consists of general instructions and *functions*. The MATLAB program package contains a great number of its own general functions. The program package SOM Toolbox contains extra MATLAB functions developed by the SOM Programming Team of our laboratory.

A script normally contains following kinds of parts:

1. Definition of parameters.
2. Loading of input data.
3. Preprocessing of input data.
4. Calling of functions.
5. Plotting.
6. Saving the results in files.

The central functions in the computation of the SOM are usually: 4.1. Initialization of the models. 4.2. Coarse training of the models. 4. 3. Fine training of the models. 4.4. Functions used in graphic displays of the SOM. These functions are specified by several *parameters* defined below.

Form and size of the SOM lattice. Let us start with the definition of the *lattice topology* of the SOM array. It is most often selected as a *rectangular plane*, called the `sheet`, with open edges. Other possible forms of the SOM lattice are a *cylinder* and a *toroid*, where the lattice is closed cyclically along one or two dimensions, respectively. The *definition of the lattice topology* is made within the *initializing function* `som_lininit` or `som_randinit`, as exemplified soon. If we want to create an SOM with rectangular plane topology, we give to the parameter `shape` the value `'sheet'`. Otherwise the value `'cyl'` or `'toroid'` is given to the `shape` parameter.The SOM Toolbox does not have any provisions for dynamically growing or other special lattice forms.

The *network structure*, or *the topological relation of the SOM nodes* is another structural feature, which is defined by the parameter `lattice`. The alternatives in the SOM Toolbox for network structures are `'rect'` and `'hexa'`. The former means a rectangular grid, and the latter a hexagonal grid, respectively. Normally we prefer the hexagonal grid because of its better isotropy in graphics, but sometimes (especially in the case of SOMs for strings of symbols) the rectangular lattice is simpler for typographic reasons.

The *number of nodes* in the SOM array, and their division in the *horizontal* and *vertical directions* is defined by the parameter `msize`, for which the value `[hor vert]` is given. Here `hor` means the number of (horizontal) *rows*, and

`vert` the number of (vertical) *columns*, respectively.

Neighborhood function. There are several options for the form of the *neighborhood function* in the SOM Toolbox. They are selected by the parameter `neigh`. The most common of them is the the Gaussian form, defined by the value `'gaussian'` of the parameter `neigh`. Another value, mainly restricted to the use of the batch training version of the SOM, is the `'bubble'`, which defines a *neighborhood set* of nodes around the winner, and has the constant value of 1 up to a certain radius from the winner, and zero outside it. However, in very large problems, in which the computing time is critical, one may *cut* the flanks of the Gaussian function at the mean radius; hence the name `'cutgauss'`. In the original documentation of the SOM Toolbox, the following definitions occur: if `Ud` is the abbreviation for the *squared radius from the winner*, and if, for brevity, we write `radius(t)` for the *squared-radius parameter*, then the definitions of the neighborhoods are written in SOM Toolbox as

```
case 'bubble',   H = (Ud<=radius(t));
case 'gaussian', H = exp(-Ud/(2*radius(t)));
case 'cutgauss', H = exp(-Ud/(2*radius(t))) .* (Ud<=radius(t));
case 'ep',       H = (1-Ud/radius(t)) .* (Ud<=radius(t));
```

Other training parameters. The most essential feature of the SOM is the *value of its neighborhood radius as a function of the number of training cycles*. As already mentioned in the theoretical part, we prefer two training phases, the coarse and the fine one, and the neighborhood function is defined differently during each. In coarse training we want to achieve global ordering of the map as quickly and surely as possible, whereas the *exact* values of the model vectors are reached in the fine-training cycles. During the latter we usually keep the neighborhood radius constant, and if the set of training inputs is the same during each fine training cycle, the algorithm is expected to terminate in a finite (and not too large) number of fine training cycles.

During *coarse training*, the (mean) radius of the neighborhood is defined by the parameter `radius_coarse`, and usually we define the *initial value* `init` and the *final value* `final`, between which the radius decreases linearly with training steps. These two values are given as the vector `[initial final]`. Also other time functions of the neighborhood radius can be defined in the SOM Toolbox.

The number of training cycles during the coarse training phase is given by the parameter `trainlen_coarse`. Correspondingly, during the fine-training phase we define the radius by the parameter `radius_fine = [final final]`, and the number of cycles during the fine training phase is `trainlen_fine`.

An example. In the example of Sec.12, which is a typical one, we have

```
msize = [6 6];
lattice = 'hexa';
neigh = 'gaussian';
radius_coarse = [4 .5];  % [initial final]
trainlen_coarse = 50;
radius_fine = [.5 .5];   % [initial final]
trainlen_fine = 10;
```

Note that the radius of the Gaussian neighborhood function need not be an integer, and .5 is a typical final value for small maps (and also the smallest value in general). On the other hand, the length of the fine-training phase above seems rather short, but it is usually followed by a few extra training phases associated with the *stopping rule* of the algorithm, which will be introduced in forthcoming examples.

The above parameter values can also be given explicitly numerically *within the training functions themselves*, as we will see next.

Initialization of the SOM. The initialization of the SOM models is made by the function `som_lininit` or `som_randinit`. The latter initializes the models by random values. This kind of initialization may only have theoretical interest in itself. The idea of using `som_randinit` is that when using it, one can prove that the self-organization of the SOM is possible starting with arbitrary initial values. However, in practice, a much quicker computation of the SOM ensues using `som_lininit`, i.e., picking the initial values in a regular fashion from the hyperplane spanned by the two largest principal components, as already explained in Subsec. 4.3.

A typical initialization command is

```
smI = som_lininit(X, 'msize', [10 15], 'lattice', 'hexa', ...
      'shape',  'sheet');
```

where `X` is the *input data matrix*, and the SOM array size is 10 by 15 nodes. The parameters are here given directly in the command. This function returns the initial values of the SOM matrix as the variable `smI`. It is a data type named *structure* (see next paragraph).

MATLAB structs. A remark concerning the representation of matrices in MATLAB functions should now be made. In the MATLAB at large, and in some SOM Toolbox functions in particular, the matrix arrays are represented as so-called *structures*, briefly called "structs." A MATLAB struct is an array organization with named fields that can contain data of varying types and sizes.

For instance, you can create a new struct `s1` like shown in the following example:

```
s1.a = 12.7;
s1.b = {'abc', [4 5; 6 7]};
s1.c = 'Hello!';
save('newstruct.mat', '-struct', 's1');
```

In the SOM Toolbox there is a file named `codebook`, which contains the information about the SOM matrix `M` at the different phases of its computation. After initialization, the matrix `M` is stored and returned as the struct `smI`, which is not a mathematical matrix variable, but which can be readily converted into the matrix form by the command `M = smI.codebook`, if we need the explicit matrix form in computations. Similar structs are `smC`, which represents `M` after coarse training, and `sm`, which stands for `M` after fine training, and they too can be converted into mathematical matrix variables by the commands `M = smC.codebook` and `M = sm.codebook`, respectively.

Except for the initialization and training functions, we do not discuss structs in this book.

Training functions. Next we define the SOM Toolbox function `som_batchtrain`. This function, with various parameters, can be applied for both coarse training and fine training. Below we have a typical set of commands for initialization and training:

```
% Initialization:

smI = som_lininit(X, 'msize', msize, 'lattice', lattice, ...
      'shape',  'sheet');

% Coarse training:

smC = som_batchtrain(smI, X, 'radius', radius_coarse, ...
      'trainlen', trainlen_coarse, 'neigh', neigh);

% Fine training:

sm = som_batchtrain(smC, X, 'radius', radius_fine, ...
     'trainlen', trainlen_fine, 'neigh', neigh);
```

In the SOM Toolbox there is also the function `som_seqtrain`, which implements the original *stepwise recursive algorithm* discussed in Sec.4. This function has only theoretical interest, since for practical computation the batch training algorithm has many virtues over it: the batch algorithm does not involve any learning-rate parameter, and it is quicker and more robust.

Winner search. The winner search in the SOM training algorithms is built in the training functions themselves. However, we need a separate winner search script in the *calibration* of the SOM, i.e., when testing what SOM model matches best with a given input item.

The MATLAB has been designed for a very effective handling of matrices, and so we can find the best-matching nodes by a couple of command lines. If the model vectors are denoted (in the usual mathematical notation) as $\mathbf{m}_i$, where i runs linearly over the nodes (not yet regarding the two-dimensional SOM array, which is not needed until in the graphics), and if $\mathbf{x}$ is the calibration (or input) vector, then we have to find the minimum over the set of vectorial differences $\{\mathbf{x} - \mathbf{m}_i\}$. The *locations of the minima* are the same as the *locations of the minima of the squares of the differences*, which we write:

$$||\mathbf{x}||^2 + ||\mathbf{m}_i||^2 - 2(\mathbf{x} \cdot \mathbf{m}_i) \quad .$$

But in the winner search, $\mathbf{x}$ is the same for all differences and can be ignored. Let now `X` be the matrix of all *calibrating inputs*; if the calibration is made using all of the columns of the input data matrix, then `X` is the data matrix itself.

In MATLAB we can perform many simultaneous computations very conveniently. Let `M` be the SOM matrix. The squares of the norms of all of the models $\mathbf{m}_i$, denoted `norms2` in the scripts in the sequel, are computed *simultaneously* as the expression `sum(M.*M,2)`, and so the index `c` of the winner model for each calibration input `X(u,:)` is obtained by

```
M = sm.codebook;
norms2 = sum(M.*M,2);
for u = 1:size(X,1)
    X1 = X(u,:)';
    Y = norms2 - 2*M*X1;
    [C,c]= min(Y);
end
```

where the expression `[C,c]` defines both the value `C` of the minimum, and its index `c`.

However, this piece of script becomes even simpler if we define the matrix `Norms`:

```
Norms = norms2;
for u = 1:size(X,1)-1
    Norms = [Norms norms2];
end
```

In other words, `Norms` is now a matrix in which the column vectors `norms2` are repeated `size(X,1)` times. Then we obtain *all of the winner indices* `c` *simultaneously by these two lines*:

```
Y = Norms - 2*M*X';
[C,c] = min(Y);
```

There are also SOMs in which the dot-product matching criterion is used. In that case the whole winner search is even simpler and faster, because no `norms2` need be computed:

```
M = sm.codebook;
Y = M*X';
[C,c] = max(Y);
```

Graphic representations of the SOM models. The SOM models $\mathbf{m}_i$, being usually vectors, can be represented by the SOM graphics in many different ways, e.g.:

1. In low-dimensional problems one can plot the model vectors referring to the input space (cf. Secs. 6, 11 and 13).

2. One can display the components of the model vectors as graphic diagrams such as bar diagrams, pie charts, etc.

3. If the vectorial models have a distinctive semantic meaning, such as the name of a class, *textual labeling* of the nodes can be used (cf., e.g., Sec. 12).

4. If the number of classes to be represented by the models is small, one can use *shades of gray or pseudo-colors* to paint those nodes (locations) of the SOM, the models in which belong to particular classes. In this way, classified areas of the SOM become clearly visible.

5. In a special case as discussed in Sec. 11, i.e., the self-organization of colors, the nodes (locations) of the SOM array are directly painted by the color represented by the models.

6. If, on the other hand, the SOM is used to represent *histograms* of items falling in different classes (e.g. the mushroom example in Sec. 19), the numbers of input items mapped into particular SOM locations are usually represented by shades of gray.

11 The QAM problem recomputed by the SOM Toolbox

Before we attack larger-scale problems, just for curiosity and for comparison we carry out the computation of the 64 QAM by the SOM Toolbox functions.

Unfortunately, without rewriting the SOM Toolbox, we cannot mix the k-means clustering and SOM algorithms easily. However, to try to reach a compromise between topographic ordering and elimination of the border effects, we can use the Gaussian neighborhood function, *but select a small final radius for the neighborhood function*, say, 0.2, which reduces the border effects. Fig. 14 shows the resulting Voronoi diagram. The computation on a 2 GHz computer took 0.6 seconds, and was carried out by the script

```
X = zeros(10000,2); % training input
for t = 1:10000
    X(t,1) =  floor(8*rand) + .1*randn - 3.5;
    X(t,2) =  floor(8*rand)+ .1*randn - 3.5;
end
smI = som_lininit(X, 'msize', [8 8], 'lattice', 'rect', ...
      'shape',  'sheet');
smR = som_batchtrain(smI, X, 'radius', [4 .2], 'trainlen', 20, ...
      'neigh', 'gaussian');
sm = som_batchtrain(smR, X, 'radius', [.2 .2], 'trainlen', 3, ...
      'neigh', 'gaussian');
M = sm.codebook;
figure(1);
voronoi(M(:,1),M(:,2))
```

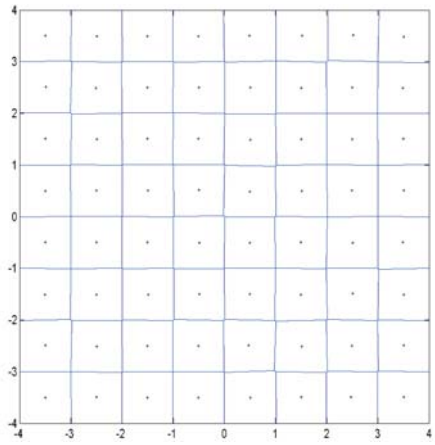


Fig. 14. The Voronoi diagram of the 64QAM solved by the SOM Toolbox.

12 The SOM of some metallic elements

Clustering of items according to their physical properties

Objective:

The attributes of the items (metallic elements) in this example are physical, macroscopic properties of materials. We want to see to what degree the items are clustered on the basis of their observable common physical properties.

12.1 Description of items on the basis of their measurable properties

As mentioned in the introductory chapters of this book, the SOM has been applied to a very large range of problems, many of which deal with very big data bases. Unfortunately I cannot include extensive applications from the practice in this book, but I hope that you will be able to move on to practice after understanding how the SOM operates.

It is my purpose that you could first get some hands-on experience of fair-sized cases and see in what way the SOM can illustrate and visualize data. I have intentionally tried to pick up examples from different application areas with different types of data. The *preprocessing* and preparation of the data to the SOM algorithm is mostly different in the various cases, and I am starting from simple and concrete problems, proceeding to cases in which the similarity relations between the data items are more abstract. The majority of SOM applications deals with variables with concrete, measurable attributes.

For the first clustering example we shall take a typical case in which the items are described by their *measurable properties*. These properties constitute the *attributes* of the metals, on the basis of which we intend to *cluster* the metals in a way that is different from the classical systematics of the elements. The results ensuing in this problem may be understandable to us, at least intuitively.

I have selected 17 metallic elements: *Al, Sb, Ag, Ir, Cd, Co, Au, Cu, Pb, Mg, Ni, Pd, Pt, Fe, Zn, Sn,* and *Bi.* For these metals, all of the following 12 attribute values could be found from physical tables:

1. Density (kg/dm^3) at 18°C
2. Coefficient of thermal expansion x 10^6(per cent /C)
3. Compressibility x 10^6 (relative change/at)
4. Velocity of sound in it (m/s)
5. Modulus of elasticity (kp/cm^2)
6. Thermal conductivity at 18°C ($kcal/m \cdot h \cdot C$)
7. Specific heat ($kcal/kg \cdot c$)
8. Melting point (C)
9. Fusion heat ($kcal/kg$)
10. Boiling point at 760 $mmHg$ (C)

11. Boiling heat (kg/kg)
12. Specific resistance ($ohm \cdot mm^2/m$)

The first thing in a SOM script is to define the input matrix of the attributes. Since this is still a problem of small dimensionality, we can write the input matrix explicitly: the rows represent the above elements, and the columns represent the above physical properties, respectively.

```
X = [2.7   23.8 1.34 5105  .7  187   .22    658   92.4  2450 2800   2.72
     6.68  10.8 2.7  3400  .79  16   .051   630.5 38.9  1380  300  39.8
    10.5   18.8  .99 2700  .8  357   .056   960.5 25    1930  520   1.58
    22.42   6.59 .27 4900 5.2   51.5 .032  2454   28    4800  930   5.3
     8.64  31.6 2.25 2300  .51  84   .08    320.5 13     767  240   7.25
     8.8   12.6  .54 4720 2.08  60   .1    1489   67    3000 1550   6.8
    19.3   14.3  .58 2080  .81 268   .031  1063   15.7  2710  420   2.21
     8.92  16.2  .73 3900 1.2  338   .0928 1083   50    2360 1110   1.72
    11.34  28.9 2.37 1320  .17  30   .03    327    5.9  1750  220  20.7
     1.734 25.5 2.95 4600  .45 145   .25    650   46.5  1097 1350   4.3
     8.9   12.9  .53 4970 2.03  53   .108  1450   63    3075 1480   7.35
    12.16  11.04 .53 3000 1.15  60   .059  1555   36    2200  950  10.75
    21.45  15.23 .36 2690 1.6   60   .032  1773   27    4300  600  10.5
     7.86  12.3  .59 5100 2.2   50   .11   1530   66    2500 1520   9.9
     7.14  17.1 1.69 3700  .43  97   .092   419.5 26.8   907  430   5.95
     7.3   27   1.88 2600  .55  57   .05    231.9 14.2  2270  620  11.3
     9.8   13.4 2.92 1800  .32   8.6 .029   271.3 14.1  1490  200 118];
```

Normalization of the variables. Because the physical entities have been measured by different units and are thus given in different scales, we must *normalize* them. This we do by a change of all scales, where each variable shall have an identical minimum and maximum:

```
for i = 1:12
    mi = min(X(:,i));
    ma = max(X(:,i));
    X(:,i) = (X(:,i)-mi)/(ma - mi);
end
```

Making the SOM. Now we are ready to move to the initialization and computing of the SOM, which begins with the declaration of the parameters, and continues with the execution of the functions `som_lininit` and `som_batchtrain`:

```
msize = [6 6];
lattice = 'hexa';       % hexagonal lattice
neigh = 'gaussian';     % neighborhood function
radius_coarse = [4 .5]; % [initial final]
trainlen_coarse = 50;   % cycles in coarse training
radius_fine = [.5 .5];  % [initial final]
trainlen_fine = 10;     % cycles in fine training
```

```
smI = som_lininit(X, 'msize', msize, 'lattice', lattice, 'shape', ...
'sheet');
smC = som_batchtrain(smI, X, 'radius', radius_coarse, 'trainlen', ...
trainlen_coarse, 'neigh', neigh);
sm = som_batchtrain(smC, X, 'radius', radius_fine, 'trainlen', ...
trainlen_fine, 'neigh', neigh);
```

The SOM has now been computed and expressed as the struct `sm`. Next we show how it can be shown explicitly.

Plotting of the SOM array and its labeling. The plotting of the SOM may take place in many different ways. In this example we first define a *blank hexagonal SOM graphic* of a correct lattice size, the cells of which we have to label by the due symbols of the elements. The blank hexagonal network is drawn by the command

```
som_cplane('hexa',msize, 'none')  .
```

The symbols of the elements are then written into the due hexagons. They are defined in the following way. We first define two strings `labels1` and `labels2`, which define the two letters that define the elements:

```
labels1 = 'ASAICCACPMNPPFZSB';
labels2 = 'lbgrdouubgidtenni';
```

The first letter of the element numbered by the parameter `u`, `u = 1 ... 17`, is the `u`:th element in the string `labels1`; the second letter is the `u`:th element in the string `labels2`, respectively. For instance, `'Al' = [labels1(1) labels2(1)]`, and `'Bi' = [labels1(17) labels2(17)]`.

But first we have to *calibrate* the SOM nodes using the original input data `X`, for which we have to find the *winner nodes*. As described in Sec.10, these are found by the piece of script

```
M = sm.codebook;
norms2 = sum(M.*M,2);
for u=1:17
    X1 = X(u,:)';
    Y = norms2 - 2*M*X1;
    [C,c] = min(Y);
```

(continues)

The locations on the SOM display, into which the symbols of the elements have to be written, are defined by the *horizontal rows* `ch` and *vertical columns*

cv of the SOM array. However, in the text command used for labeling the cells, ch takes the role of the x coordinate and cv the role of the y coordinate, respectively. These coordinates are resolved as

```
ch = mod(c-1,6) + 1;
cv = floor((c-1)/6) + 1;
```

(continues)

Now we can write the symbols automatically into their correct places onto the SOM, defined by the coordinates ch and cv, by the following script. Because the even and odd rows of the hexagonal SOM are mutually displaced in the horizontal direction, we have to use the shift1 parameter for the horizontal shift to position the texts correctly. However, since there were a few *collisions* of different labels in the same cells, we have to use the shift2 parameter in these locations to position the colliding symbols correctly also in the vertical positions. The result is shown in Fig. 15.

```
    if mod(cv,2) == 1
       shift1 = -.15;
    else
       shift1 = .35;
    end
    if u==9 || u==11
      shift2 = -.3;
    else if u==5 || u==14
      shift2 = .3;
    else
      shift2 = 0;
    end
    text(ch+shift1,cv+shift2,[labels1(u) ...
    labels2(u)], 'FontSize',15);
end
```

What we may need further are the commands to print and store the figure.

Discussion. A surprising result in this example is that we can find a tight *ferromagnetic cluster* of *Ni, Co* and *Fe* at the top, although we did not consider the magnetic susceptibility or any other magnetic properties of the metals. This is obviously due to some strong correlation between the physical properties of the ferromagnetic metals.

The series of noble metals *Pt, Pd, Au* and *Ag* is also discernible, and if the chemical reactivity properties are not taken into account (and we had no attributes of them), *copper* is *physically* close to the noble metals. The physical properties of the rest of the metals also seem to be related correctly. Note that *Sn, Pb, Sb,* and *Bi* are used in low-melting-point alloys.

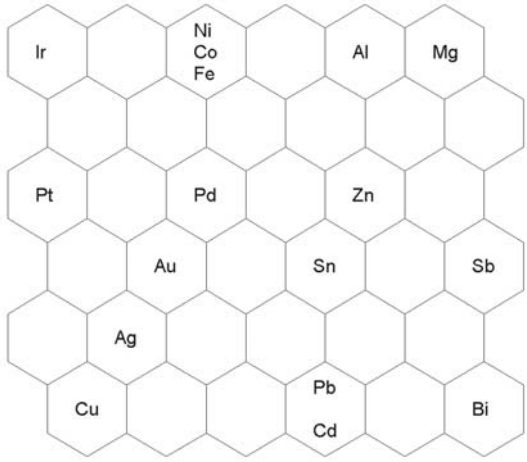


Fig. 15. The SOM of some metallic elements, the properties of which may be familiar to us. One can see the grouping of metals according to their physical properties.

What is the rationale of using the SOM for this kind of analysis and showing this kind of a figure? Certainly it does not add much to our knowledge, compared, e.g., to the Mendelejev table and the well-known physics of metals, but it may give us a hint of how the SOM may in general discover new, interesting and unexpected phenomena from experimental data. The SOM comes in handy especially in the *preliminary analysis* of data.

Naturally this same analysis could be carried out for a wider range of elements, and more complete tables of physical properties are nowadays available. Nonetheless we wanted to keep this example, as well as most of the examples taken to this book simple, in order that the program structures would be more transparent and easier to follow, and to demonstrate that even rather small SOMs, with a relatively small data set, may demonstrate new dimensions in data analysis.

13 Self organization of color vectors

Ordered 2D projection of random 3D color vectors

Objective:

In this second example we demonstrate how we can map (project) 3D color vectors onto a 2D plane in an orderly fashion by the SOM algorithm. The color vectors are mixtures of red, green and blue colors in which the color components vary. These colors become topographically ordered on a 2D plane with respect to both intensity and hue to produce the so-called chromaticity diagram that occurs in human color vision.

13.1 Different nature of items and their attributes

Although we shall be dealing with physical attributes in this example, too, nonetheless the nature of the problem is completely different from the first case. First of all we do not have a finite set of concrete objects like the metals in the first example; *each input item is only a color shade*, and there exists an indefinite number of them in practice. On the other hand, each shade is represented only by three attributes, namely, the intensities of the three components of the basic colors, which are red, green, and blue. Nonetheless, although the mathematical representation of the input items is different from that in the previous example, the self-organization of the colors occurs formally in a similar way.

13.2 Color vectors

A three-dimensional *RGB (red-green-blue) color vector* is a *digital code* for any mixtures of shades and intensities of visible colors. In digital representation, the intensities of the three *basic color components* (R, G, B) are given as real scalars in the range $[0, 1]$.

Our aim is to show that if the input data represent quite random color vectors, the SOM algorithm is able to produce a representation of colors such that their *distribution* remains the same, but the colors become *spatially ordered* on the SOM array such that both the *hue* and the *intensity* in neighboring models change gradually; it is said that a *topographic order* of the colors has ensued. Because we can understand the relations between colors intuitively, we use this color example as an abstract model for more general *topographic self organization*.

With a proper scaling of the input vectors, an SOM can be produced that represents the generally known *chromaticity diagram* or *CIE diagram*, where the hue and the saturation of color become represented in polar coordinates like in the human vision.

Consider a MATLAB variable C(a,b,c), which represents a *three-dimensional table*. The parameters `a` and `b` are the indices of a row and a column in the table, and `c` is a vector that has three elements: its first element represents the intensity of *pure red*, the second element represents the intensity of *pure green*, and the third component is the intensity of *pure blue*, respectively.

Let us exemplify the digital coloring by the following concrete example. The MATLAB function `image(C)` defines the coloring of the square `(a,b)` by that pure color component, the intensity of which is defined by the vector `c`. If one needs *mixed colors*, the same location `(a,b)` must be painted *separately* by the intensities of the elements of `c`; in other words, the mixed colors are defined digitally by *superimposing* basic color components.

However, unlike in wet painting, where the mixture of blue and yellow produces the green color, in digital color definition the mixture of *red and green* produces the *yellow* color. Therefore yellow in the area `(a,b)` is produced by the combination of the functions `C(a,b,1)` and `C(a,b,2)`. Consider the following MATLAB commands:

```
C = zeros(1,4,3);       % 1 by 4 table, three color components

C(1,1,1) = 1;           % leftmost (1,1) area is pure red C(:,:,1)
C(1,2,2) = 1;           % second (1,2) area is pure green C(:,:,2)
C(1,3,3) = 1;           % third (1,3) area is pure blue C(:,:,3)

C(1,4,1) = 1; C(1,4,2) = 1;  % fourth area is painted pure yellow:
                             % combination of pure red = C(:,:,1)
                             % and pure green = C(:,:,2)

image(C)                     % painting
```

These commands produce the color picture in Fig. 16.

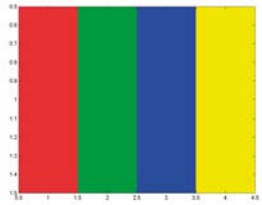


Fig. 16. Coloring example.

Another picture, in which a 25 by 25 array is colored by *random colors*, is shown in Fig. 17.

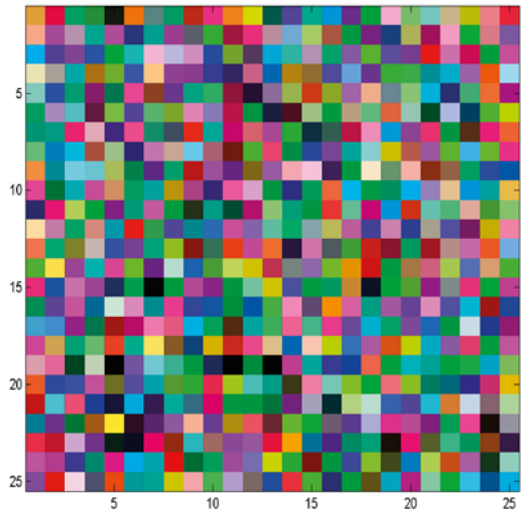


Fig. 17. Coloring of a 25 by 25 array by random color vectors.

13.3 The SOM script for the self organization of colors

In this subsection we initialize the SOM by the function `som_lininit`, *and continue with the training function* `som_batchtrain`, *which constructs the SOM of color vectors.*

As the *input data* we use 10'000 *random color vectors*, which constitute the 10000 by 3 *input data matrix* `X`. The SOM algorithm computes the data matrix `M`, which is a 625 by 3 matrix. It is to be denoted that in the SOM algorithm, for mathematical reasons, all of the model vectors are *concatenated* into a *vertical array* that has as many rows as there are nodes in the SOM array. It is not until we *display* the SOM array that we reshape the vertical array as a rectangular array, in this case 25 by 25.

Training parameters. The definition of the training parameters of the SOM is made first. We may decide to use a square SOM array (lattice) of the size 25 by 25 nodes, which is defined by the vector `msize =[25 25]`. In this simple example we define the SOM array as *rectangular*, `lattice = 'rect'`. We may prefer to use the *Gaussian neighborhood function*. Its definition `'gaussian'` follows the `'neigh'` parameter name in the function `som_batchtrain` defined below. Other parameters are the average *radius* of the neighborhood function, which decreases linearly with the coarse training cycles. In this self-organization example we have found it proper to use the initial value of 10 and the final value of 7, and during the fine training cycles let the radius decrease from 7 to 5, respectively. In other words, we did not make use of any stopping rule. The number of training cycles in both coarse and fine training shall tentatively be 50.

```
msize = [25 25];
lattice = 'rect';
radius_coarse = [10 7];
radius_fine = [7 5];
trainlen_coarse = 50;
trainlen_fine = 50;
```

Training functions. Next we define the SOM Toolbox functions `som_lininit` and `som_batchtrain`. The former carries out the *linear initialization* of the SOM. The `som_batchtrain` function is then applied twice: first for coarse training, and then for fine training. The last parameters `'shape','sheet'` in `som_lininit` mean that the topology of the SOM array is a plane.

For the training inputs `X` we use *random values* of the color vectors: `X = rand(10000,3)`.

The following commands, which form the *MATLAB SOM Toolbox script*, may now be self-explanatory. These instructions form the complete script or program which computes the SOM array of *self-organized colors*.

```
X = rand(10000,3); % random input (training) vectors to the SOM
smI = som_lininit(X,'msize',msize,'lattice',lattice,'shape', ...
'sheet');
smC = som_batchtrain(smI,X,'radius',radius_coarse,'trainlen', ...
trainlen_coarse, 'neigh','gaussian');
sm = som_batchtrain(smC,X,'radius',radius_fine,'trainlen', ...
trainlen_fine, 'neigh','gaussian');
```

Display of the SOM. This time we do not need the SOM graphic function, since we are not calibrating the SOM array in the usual way. The matrix values `M` represent color shades, which can displayed directly by the `image` function. The result is shown in Fig. 18.

```
M = sm.codebook;
C = zeros(25,25,3);
for i = 1:25
    for j = 1:25
        p = 25*(i-1) + j;
        C(i,j,1) = M(p,1); C(i,j,2) = M(p,2); C(i,j,3) = M(p,3);
    end
end
image(C)
```

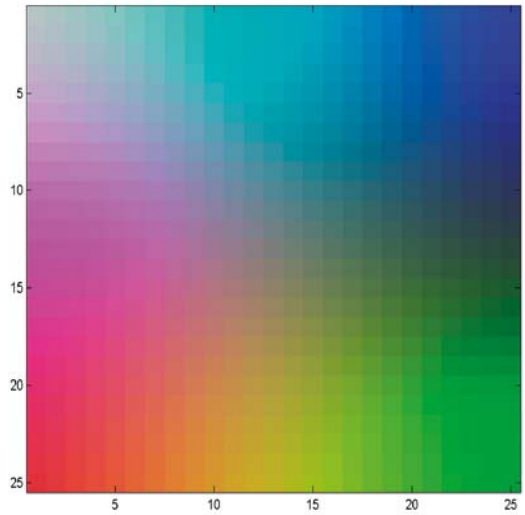


Fig. 18. Self organization of random colors in the SOM.

Computation of the chromaticity diagram. Another experiment, in which the *square root* of the previous color vectors was used as input data matrix `X`, is shown in Fig. 19.

Fig. 19 resembles the *chromaticity diagram*, which is a representation of *intensity* and *hue* of colors in two-dimensional polar coordinates. This diagram, also called the CIE diagram, is experimentally verified to exist in the human brain, in the visual cortex.

Like in the CIE diagram of colors, the pale area in Fig. 19 is also in the middle, and we have an ordered representation of intensity and hue of and mixed colors. There is no black area in this illustration like in Fig. 18. Obviously the square root emphasizes light colors to produce this result.

The square root resembles the *logarithm*, and it is known that in the biological sensory systems the subjectively experienced signal intensities are often logarithmically related. This might explain why Fig. 19 resembles the experimentally constructed CIE diagram. However, we cannot use logarithmic scales in digital signal processing, because small signals are transformed into very large negative logarithmic values.

Discussion. This example showed how a *three-dimensional color solid* was projected onto a *two-dimensional plane*. The plot includes areas where also the intensities of colors vary: for example, in Fig. 8 there is a pale area in the upper left corner, as well as a dark area on the right side. In the former, all of the numerical values of color components are close to unity, while in the latter, the numerical values of all of the components of the color vectors are low.

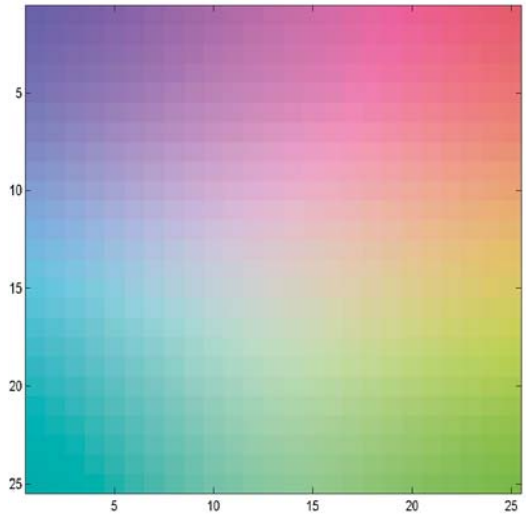


Fig. 19. Another self organization of random colors by the SOM, when the *square root* of the color vectors was used as input data to the SOM. This image resembles the *chromaticity diagram* or the *CIE color diagram of the human color vision.*

The somewhat more realistic color diagram in Fig. 19 was constructed using color vectors with an enhanced distribution of whiter shades.

Certainly this projection is *nonlinear*. One may imagine that the nodes of the SOM array form a *flexible two-dimensional network* that is trying to approximate the three-dimensional color solid. Like the *fractal surfaces*, also called *space-filling surfaces*, this "flexible" SOM network behaves somewhat in the same way. It is trying to approximate the higher-dimensional structures, but is succeeding only partly. The SOM network is more or less "stiff," depending on the *width of the neighborhood function*, and its ability of approximating higher-dimensional structures depends on the choice of this neighborhood function. In this example the final width of the neighborhood function was equal to 5, which is a rather large value compared with what is normally used. On the other hand, with this value the color maps shown in Figs.18 and Fig. 19 have become *globally ordered*, i. e., they do not contain partially disorganized areas.

One may also understand that while the SOM network is a *projection surface* onto which the higher-dimensional data distribution is projected nonlinearly, this projection surface, in the two-dimensional display, is straightened up.

This example also shows that although the SOM projection is able to illustrate higher-dimensional distributions, it is not always unique. Especially depending on the choice of the neighborhood function, but also on the scaling of the input data and different random-number sequences, the detailed projections may vary from one mapping to another. The *topographic relations* between the items, however, tend to be preserved.

14 The SOM of the present financial status of 50 countries or unions

Ordering of items with respect to a set of statistical indicators

Objective:

We try to construct an SOM that represents and compares the financial status of selected countries, based on data that are available in the Internet for the present.

We had available in the Internet several financial indicators for many countries or unions from the turn of the year 2013/2014, as well as from August, 2014. Based on the data for the 50 richest of them, we now construct SOMs that describe the *clustering* of these countries. First, however, it will be necessary to emphasize a few facts.

1. When compared with the "welfare map" presented in Fig. 1, the main difference is that the "welfare map" was mainly based on nonfinancial indicators*. There were only few financial indicators, such as the gross national product (GNP) per capita, whereas the majority of the 39 indicators represented health services such as the number of doctors per capita, as well as educational services, all of which correlated strongly with GNP and were reasonably stable with time. Contrary to that, the present example is completely based on* financial indicators*, among which there are strongly interrelated variables such as the rate of interest and inflation, but there is also a different time behavior between them. For instance, if the inflation is high, the central banks are forced to raise the rate of interest, but its effect is delayed. Also, since this occurs at different times in different countries, the countries may be* in a different phase of periodic economical development*, and actually their indicators could then not be compared directly.*

2. On the other hand, if the objective had been to compare the economical status *of the countries, one should have taken into account the trade balance, the volumes of domestic and international markets etc. Unfortunately these figures are not easy to obtain without hard work, and this context is not quite suitable for such studies. We decided to be content with the information that was available in general sources.*

3. One also has to emphasize the fact that the scale of the SOM display is not homogeneous, and one usually cannot plot any coordinate axes onto it. Therefore, sometimes the projections of the countries seem to make big jumps, but it does not mean that the corresponding true vectorial differences of their inputs were that big. The quantitative relations in the SOM vectors can be visualized with the aid of some auxiliary graphic aids such as the U matrix *introduced in Sec. 15.*

The raw data. In this example, the following countries and unions were included. Below are their codes used in the maps:

ARE	United Arab Emirates	IRL	Ireland
ARG	Argentina	IRN	Iran
AUS	Australia	IRQ	Iraq
AUT	Austria	ISR	Israel
BEL	Belgium	ITA	Italy
BRA	Brazil	JPN	Japan
CAF	Central African Rep.	KOR	Korea, Rep.
CAN	Canada	MEX	Mexico
CHE	Switzerland	MYS	Malaysia
CHL	Chile	NGA	Nigeria
CHN	China	NLD	Netherlands
COL	Colombia	NOR	Norway
CSK	Czechoslovakia	OAN	Taiwan tiny .
DEU	Germany	PAK	Pakistan
DNK	Denmark	PHL	Philippines
EGY	Egypt. Arab Repub.	POL	Poland
ESP	Spain	PRT	Portugal
EUR	European Union	RUS	Russia
FIN	Finland	SAU	Saudi Arabia
FRA	France	SGP	Singapore
GBR	United Kingdom	SWE	Sweden
GRC	Greece	THA	Thailand
HKG	Hong Kong	TUR	Turkey
IDN	Indonesia	USA	United States
IND	India	VEN	Venezuela

The original data obtained from the Internet for the various countries consisted of the following indicators: gross national product in millions of dollars (GNP), rate of interest (ROI), inflation (INFL), unemployment (UNEMP), debts in relation to (DEBTS), liquid deposits in relation to GNP (DEPOS), and population (POP). They are listed in the table below:

Country	GNP	ROI	INFL	UNEMP	DEBTS	DEPOS	POP
USA	16800	.25	2.00	6.20	101.53	-2.30	317.30
EUR	12750	.15	0.30	11.50	92.60	2.40	332.88
CHN	9240	6.00	2.30	4.10	22.40	2.00	1354.04
JPN	5960	0.00	3.40	3.80	227.20	0.70	127.22
DEU	3635	0.15	0.80	4.90	78.40	7.50	81.84
FRA	2735	0.15	0.50	10.10	91.80	-1.30	65.28
GBR	2522	0.50	1.60	6.40	90.60	-4.40	63.26
BRA	2246	11.00	6.50	4.90	56.80	-3.66	193.94
RUS	2097	8.00	7.50	4.90	13.41	1.56	143.35
ITA	2071	0.15	-0.10	12.60	132.60	1.00	9.39
IND	1877	8.00	7.96	5.20	67.72	-1.70	1233.00

Country	GNP	ROI	INFL	UNEMP	DEBTS	DEPOS	POP
CAN	1825	1.00	2.10	7.00	89.10	-3.20	35.06
AUS	1561	2.50	3.00	6.40	20.48	-2.90	22.79
ESP	1358	0.15	-0.50	24.47	93.90	0.80	46.20
KOR	1305	2.25	1.60	3.40	33.80	5.80	50.00
MEX	1261	3.00	4.07	5.47	36.90	-1.80	116.90
IDN	868	7.50	4.53	5.70	26.11	-3.30	245.90
TUR	820	8.25	9.32	8.80	35.85	-7.90	75.62
NLD	800	0.15	0.89	8.20	73.50	10.40	16.73
SAU	745	2.00	2.60	5.50	2.68	18.00	29.55
CHE	651	0.00	0.00	2.90	35.40	13.50	7.95
ARG	612	15.61	10.90	7.50	45.60	-0.90	41.28
SWE	558	0.25	0.00	7.10	40.60	6.20	9.48
NGA	523	12.00	8.30	23.90	11.00	7.10	166.21
POL	518	2.50	-0.20	11.90	57.00	-1.30	38.53
NOR	513	1.50	2.20	3.30	29.	11.00	4.99
BEL	508	0.15	0.00	8.50	101.50	-1.60	11.08
OAN	489	1.88	1.75	3.95	40.98	11.73	23.31
VEN	438	16.56	60.90	7.10	49.80	7.10	29.72
AUT	416	0.15	1.80	7.30	74.50	2.70	8.44
THA	387	2.00	2.16	1.15	45.70	-0.70	66.79
ARE	384	1.00	2.30	4.20	16.70	14.91	9.21
COL	378	4.50	2.89	9.30	31.80	-3.40	47.10
IRN	369	14.60	10.70	-0.78	10.30	8.12	75.10
CAF	351	5.75	6.30	25.50	46.10	-5.80	52.20
DNK	331	0.20	0.80	4.10	44.50	7.30	5.57
MYS	312	3.25	3.20	2.80	54.80	4.70	29.20
SGP	298	0.08	1.20	2.00	105.50	18.00	5.31
ISR	291	0.25	0.30	6.20	67.40	2.47	7.91
CHL	277	3.50	4.50	6.50	12.80	-3.40	17.40
HKG	274	0.50	4.00	3.30	33.84	2.10	7.22
EGY	272	9.25	10.61	12.30	87.10	-2.40	83.66
PHL	272	3.75	4.90	7.00	49.20	3.50	95.80
FIN	257	0.15	0.80	7.00	57.00	-1.10	5.43
GRC	242	0.15	-0.70	27.20	175.10	0.70	11.29
PAK	237	10.00	7.88	6.00	63.30	-1.10	178.91
IRQ	223	2.30	15.10	-5.55	31.30	6.71	32.58
PRT	220	0.15	-0.90	13.90	129.00	0.50	10.54
IRL	218	0.15	0.30	11.50	123.70	6.60	4.588
CSK	198	0.05	0.50	7.40	46.04	-1.40	10.52

The indicators used in the computation of the SOM. We shall now denote the seven original raw indicators by `X(i,j)`, where `i` is the row denoting the country, and `j` is the number of the column where the indicator is written.

A more insightful display is obtained, however, if the following six *relative* indicators, denoted `V(:,j)`, are used in the computation of the SOM:

1. Gross national product(GNP) per capita.
2. Rate of interest (in percentages)
3. Inflation per year (in percentages).
4. Unemployment, (in percentages)
5. Debts, in relation to the GNP.
6. Deposit accounts, in relation to the GNP.

These six indicators are first computed from the raw data:

```
V = zeros(50,6);
V(:,1) = X(:,1)./X(:,7);
for j = 2:6
   V(:,j) = X(:,j);
end
```

Equalizing the scales of the variables. In attacking a multivariate case like this, the first problem is usually a different scale of different indicators; they may be measured using different units. The first task in *rescaling* is to choose dimensionless scales, in which the extrema for each variable are identical. In other words, take, for instance, the unemployment, which is expressed in percentages. Let this variable, for the different countries i, be named $V(i,4)$. Since the training of the SOM is based on the inspection of *differences* of vectors, it does not matter where we put the origin of each variable, and it can be the same for all countries. The new value of $V1(i,4)$ is first expressed as a dimensionless variable

$$V1(i,4) = (V(i,4) - \min_c\{V_c\})/(\max_c\{V_c\} - \min_c\{V_c\}) \ .$$

This renormalization is often sufficient as such.

Weighting of the variables. However, especially when one is analyzing statistics associated with *human behavior*, such as political or economic relations, the straightforward equalization of the indicator values may not be sufficient. For instance, in the appraisal of land value as made by Carlson [7], he had to use experimentally determined weights for the different indicators that varied as much as by one order of magnitude, in order to get realistically-looking SOMs.

In the present problem it was found necessary to put a bit higher weight on the *gross national product (GNP) per capita*, namely, 2, as well as on the *debts* in percentage of the GNP. The latter weight was equal to 3. One might ask how I ended up with these values. Intuitively it was clear that these indicators

are more important than the others, but the exact values were decided after a few preliminary experiments, after seeing how the countries were located on the SOM relative to each other. *Especially when looking at the U matrix, these choices seemed to result in the most continuous distributions of the components and best contrasts in the U matrix.*

The first display relates to data collected in August, 2014. The script continues with the equalization and weighting of the six SOM indicators:

```
for j = 1:6
    mi = min(V(:,j));
    ma = max(V(:,j));
    V(:,j) = (V(:,j)-mi)/(ma - mi);
end
V(:,1) = 2*V(:,1);
V(:,5) = 3*V(:,5);
```

The script for the construction of the SOM is otherwise almost similar to that used in the previous examples. The following specifications of the parameters have to be mentioned:

```
msize = [17 17];
lattice = 'hexa';
neigh = 'gaussian';
radius_coarse = [7 1]; % neighborhood radius, coarse [initial final]
trainlen_coarse = 100; % cycles in coarse training
radius_fine = [1 1];   % neighborhood radius, fine [initial final]
trainlen_fine = 50;    % cycles in fine training
```

The training took place in the standard way:

```
smI = som_lininit(V, 'msize', msize, 'lattice', lattice, 'shape', ...
'sheet');
smC = som_batchtrain(smI, V, 'radius', radius_coarse, 'trainlen', ...
trainlen_coarse, 'neigh', neigh);
sm = som_batchtrain(smC, V, 'radius', radius_fine, 'trainlen', ...
trainlen_fine, 'neigh', neigh);
```

Plotting of the SOM of August 2014. In the same way as in the metal example, we use the blank hexagonal SOM graphic, the cells of which we label by the codes of the countries. Since there is no big difference in computing times, from now on we use the component-form definition of `V` to compute the winners. The horizontal and vertical coordinates `ch,cv` of the labels are determined and then the texts are written into these locations from the strings `labels1` and `labels2`. Just for a change, we take the *first two letters* of the country code from `labels1` and *the third letter* from `labels2`.

```
labels1 = 'USEUCHJPDEFRGBBRRUITINCAAUESKOMEIDTUNLSACHARSW ...
           NGPONOBEOAVEAUTHARCOIRZADNMYSGISCHHKEGPHFIGRPACSIR ...
           POIRCS';
labels2 = 'ARNNUARASADNSPRXNRDUEGEALRLNNTAELNFKSPRLGYLNCKKQRLK';
som_cplane('hexa',msize, 'none')
M = sm.codebook;
norms2 = sum(M.*M,2);
for u = 1:50
    V1 = V(u,:)';
    Y = norms2 - 2*M*V1;
    [C,c] = min(Y);
    ch = mod(c-1,17) + 1;
    cv = floor((c-1)/17) + 1;
    if mod(cv,2)==1
        shift1 = -.4;
    else
        shift1 = .1;
    end
    text(ch+shift1,cv,[labels1(2*u-1) labels1(2*u) ...
    labels2(u)],
    'FontSize',8);
end
filename = 'financemap_new';
print('-dpng', [filename '.png']);
save(filename, 'sm');
```

Fig. 20 represents the projections of the countries as of August, 2014 on the SOM.

Plotting of the SOM at the turn of the year 2013/14. I had copied the corresponding data for the same countries from the turn of the year 2013/2014, i.e., eight months earlier; now these data were no longer available in the Internet. They had been stored in my computer as `save(filename, 'sm'`. Another SOM, using parameters that were identical with those in making Fig. 20, was computed for the older input data and is shown in Fig. 21.

```
V = zeros(50,6);
for i=1:50
    filename = 'financemap_new';
    load(filename, 'sm');
end
```

In order to guarantee that the SOM represented *the same local optimum that materialized in Fig. 20, I initialized the SOM with the* `sm` *struct* obtained in the previous example. Let this representation of the SOM be stored as the file `SOM_August2014`. The fine tuning took place using 50 cycles of `som_batchtrain`

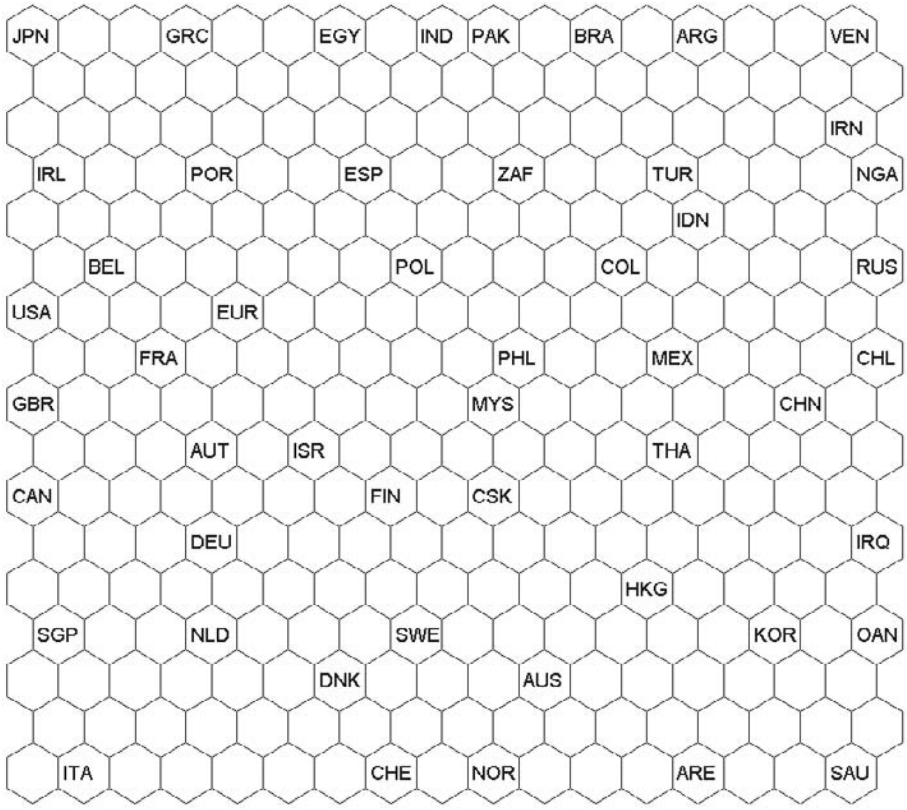


Fig. 20. The SOM of the richest 50 countries in August 2014. The data were taken from the Internet from generally available statistics, which are updated several times in a year, but they did not contain important financial indicators such as the volumes of the domestic and international markets, the trade balance etc., which would have changed the relative position of many countries.

with the value of `radius = [1 1]`. Now the script for the computation of the SOM from the turn of the year 2013/14 reads

```
filename =  'SOM_August2014';  % data used for initialization
load([filename 'sm'])
sm = som_batchtrain(sm, V, 'radius', [1 1], 'trainlen', 50, ...
'neigh', neigh);
```

Fig. 21 shows the *older* map. The SOM display was made in the same way as before.

Discussion. One cannot expect that the "Welfare map" shown in Fig. 1 and the financial maps show in Fig. 20 and Fig. 21 would be similar. First of all, they represent quite different statistics; Fig. 1, in addition to a few financial

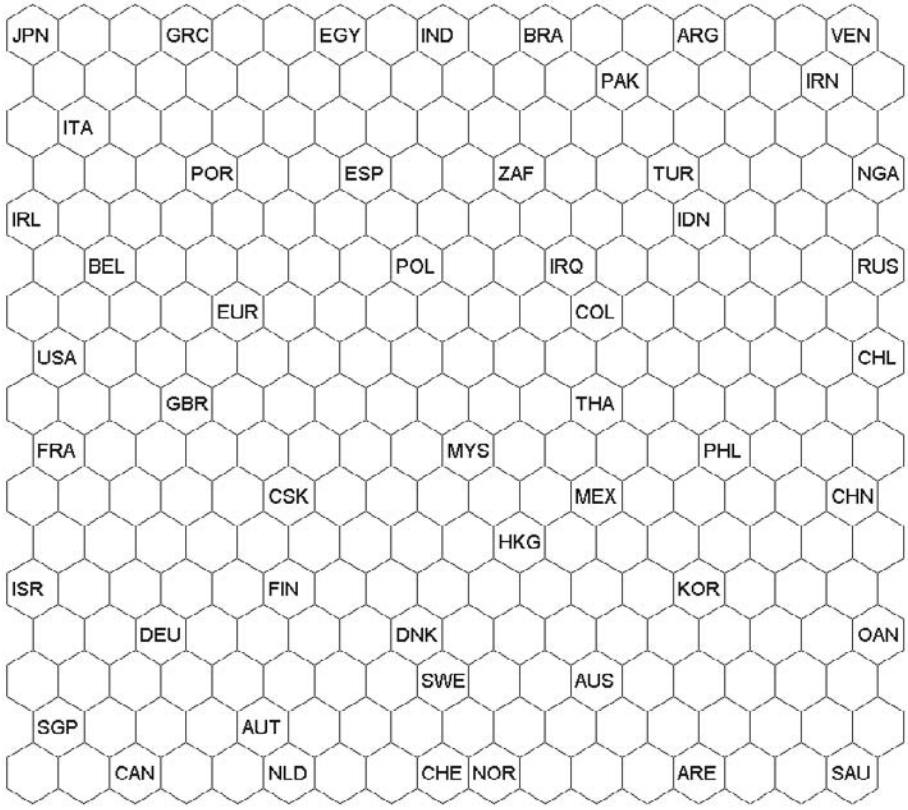


Fig. 21. The SOM of the richest 50 countries at the turn of the year 2013/14.

indicators, was strongly based on educational and health care data, which were completely missing from the "financial" maps. Second, the data stemmed from very different eras, and especially in the year 2014 there have been many kinds of severe political and financial crises, some of which have not been settled yet. Third, the collection of countries in these examples was different.

15 Using shades of gray to indicate the clustering of models on the SOM

Introduction of the U matrix

Objective:

In the maps shown in Fig. 20 and Fig. 21 we do not yet see any borders between the clusters of countries. However, the vectorial distances of the neighboring models, if we can make use of them, tell the clustering tendency of the models. A graphic display based on these distances, called the U matrix, is explained in this section.

The *clustering tendency* of the data, or the models to describe them, can be shown graphically based on the *magnitudes of vectorial distances between neighboring model*s in the map, as shown in Fig. 22 below. In it, the number of hexagonal cells has first been increased from 18 by 17 to 35 by 33, in order to create *blank interstitial cells* that can be colored by shades of gray or by pseudocolors to emphasize the cluster borders. This creation of the interstitial cells is made automatically, when the instructions defined below are executed.

The U matrix. A graphic display called the *U matrix* has been developed by *Ultsch* [87], as well as *Kraaijveld et al.* [47], to illustrate the degree of clustering tendency on the SOM. In the basic method, interstitial (e.g. hexagonal in the present example) cells are added between the original SOM cells in the display. So, if we have an SOM array of 18 by 17 cells, after addition of the new cells the array size becomes 35 by 33. Notice, however, that the extra cells are not involved in the SOM algorithm, only the original 18 by 17 cells were trained.

The *average (smoothed) distances between the nearest SOM models* are represented by light colors for small mean differences, and darker colors for larger mean differences. A "cluster landscape," formed over the SOM, then visualizes the degree of classification. The groundwork for the U matrix is generated by the instructions

```
colormapigray = ones(64,3) - colormap('gray');
colormap(colormapigray);
msize = [18 17];
Um = som_umat(sm);
som_cplane('hexaU', sm.topol.msize, Um(:));
```

In this case, we need not draw the blank SOM groundwork by the instruction `som_cplane('hexa',msize, 'none')` as for Figs. 20 and 21. The example with the countries in August 2014 will now be represented together with the U-matrix "landscape" in Fig. 22. It shows darker "ravines" between the clusters.

Annotation of the SOM nodes. After that, the country acronyms are written on it by the `text` instruction. The complete map is shown in Fig. 22.

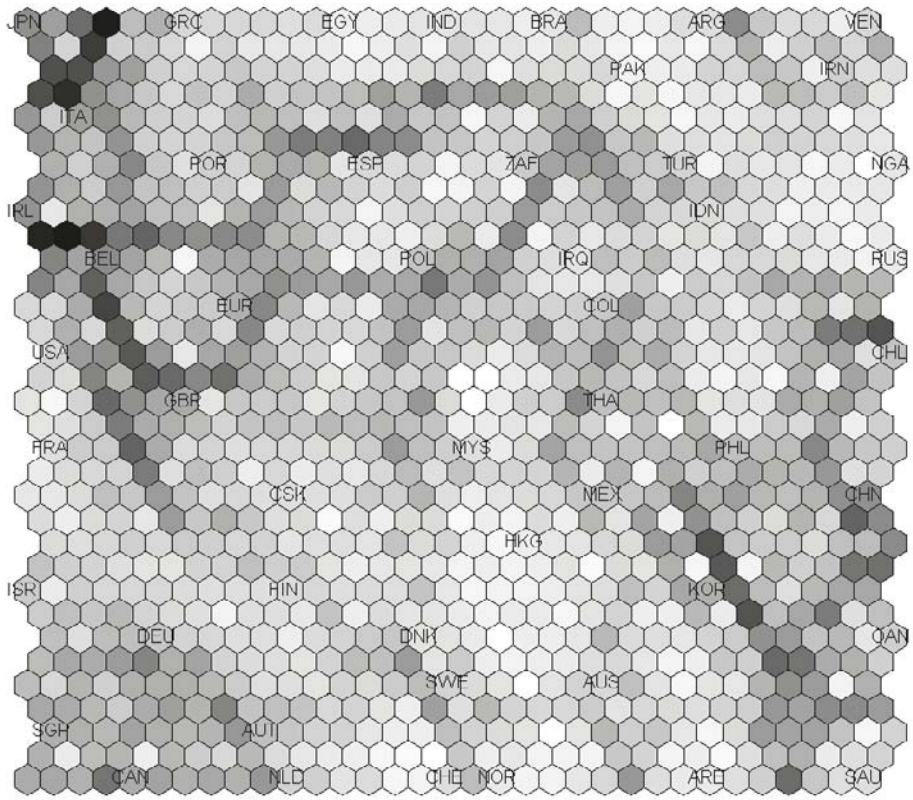


Fig. 22. The SOM, with the U matrix, of the financial status of 50 countries

A particular remark is due here. Notice that there are plenty of unlabeled areas in the SOM. During training, when the model vectors had not yet reached their final values, the "winner nodes" for certain countries were located in completely different places than finally; nonetheless the models at these nodes gathered memory traces during training, too. Thus, these nodes have learned more or less wrong and even random values during the coarse training, with the result that the vectorial differences of the models in those places are large. So the SOM with unique items mapped on it and having plenty of blank space between them is not particularly suitable for the demonstration of the U matrix, although some interesting details can be found in it.

16 Using binary attributes as input data to the SOM

Clustering of discrete items based on logic statements

Objective:

In this section we construct an SOM for items that are described by binary, symbolic attributes, resulting from simple logic statements.

Clustering is a rather common task in data analysis. It is sometimes based on the verification of a set of *properties* that the items have. Most often the properties (attributes, descriptors, indicators, or other similar qualities) are real-valued numerical arguments resulting in statistical studies or scientific experiments. However, in this very simple case example we use *binary attributes* for the description of the items. A binary attribute represents a discrete and distinct property that the object X has or does not have, and here it is given in a *statement:*: e.g., the statement " X has hair" has a *truth value* that is either 1 or 0. The *similarity* of two items is defined by the number of truth values (0 or 1) that are identical in the same statements about the two items.

However, when we use the SOM for the clustering of the items, we construct *adaptive models* for the items, i.e., for the sets of their attributes. But the models consist of *continuous-valued variables*: how do they then comply with the binary attributes? The answer is that in the algorithm we only *approximate* binary attributes by continuous-valued model parameters: both the attributes and the models are treated as real vectors, and in their matching, the *comparison for similarity* is still made by computing the *vectorial differences between the sets of binary attributes and the elements of the model vectors.*

We continue with a very simple example, a toy problem, which has been presented in [39]; this example has been slightly revised here. In it, 13 animals are presented by 11 *binary attributes* with the values `1` and `0`, respectively. The *input data* are given as a *binary data matrix* `X(animal, attribute)`:

```
X = [1 0 0 1 0 0 0 0 0 1 0
     1 0 0 1 0 0 0 0 0 0 0
     1 0 0 1 0 0 0 0 0 1 1
     1 0 0 1 0 0 0 1 0 1 0
     0 1 0 1 0 0 0 1 0 1 0
     0 1 0 0 1 0 0 1 0 0 0
     0 1 0 0 1 0 0 0 1 0 0
     0 1 0 0 1 0 1 1 1 0 0
     1 0 0 0 1 0 0 1 0 0 0
     0 0 1 0 1 0 0 1 1 0 0
     0 0 1 0 1 0 1 1 1 0 0
     0 0 1 0 1 1 1 0 1 0 0
     0 0 1 0 1 1 0 0 0 0 0]
```

The *rows* of `X` represent the following 13 `animals` :

{*dove, hen, duck, hawk, eagle, fox, dog, wolf, cat, tiger, lion, horse, cow*},

and the binary values on the *columns* of X correspond to the *truth values* of the statements, respectively:

{*is small, is medium, is big, has 2 legs, has 4 legs, has hooves, has mane, likes to hunt, likes to run, likes to fly, likes to swim*}.

Loading of the data matrix. Let us next assume that we had already prepared the data matrix `X` and saved it by the filename `'animaldata'`. In saving, we have also defined the name of the data matrix to be `X`. The data are loaded by the instructions

```
filename  = 'animaldata';
load (filename, X)
```

We do not have to define the dimensions of `X`, because they are automatically defined in the loading instruction.

Initialization and training of the SOM. Contrary to what we made in the first example, we shall now try to get along with a *single training phase*. First we use the linear initialization `som_linit`. After that we apply the batch training function `som_batchtrain` for 50 cycles. This time, for a change, we write the explicit parameter values directly into the instructions. A hexagonal lattice with the parameter value `'hexa'` is chosen. The parameter `'shape'` below shall again have the value `'sheet'`.

```
smI = som_lininit(X, 'msize', [7 7] , 'lattice', 'hexa', ...
'shape', 'sheet');
sm = som_batchtrain(smI, X, 'radius', [3 1], 'trainlen', ...
50, 'neigh', 'gaussian');
M = sm.codebook;
```

Calibration of the SOM nodes. The next task is to label those nodes of the SOM array that correspond to the various animals. What we again have to do first is to locate the *winner nodes* for each of the row vectors of the data matrix `X` that correspond to the animals:

```
norms2 = sum(M.*M,2);
for u = 1:13
    X1 = X(u,:)';
    Y = norms2 - 2*M*X1;
     [C,c] = min(Y);
end
```

When we plot the SOM, we again first draw the blank hexagonal array:

```
som_cplane('hexa', [7 7], 'none');
```

After that we automate the labeling like in the example of Sec. 12. The maximum number of letters in a name is five, so we need the following five strings of symbols to define the words (you can read the names from top to bottom from these five strings, e.g., the first name is `L1(1) L2(1) L3(1) L4(1) L5(1) = 'dove '` :

```
 L1 = 'dhdhefdwctlhc';
 L2 = 'oeuaaoooaiioo';
 L3 = 'vncwgxgltgorw';
 L4 = 'e kkl  f ens ';
 L5 = '    e    r e ';
```

The labeling continues by determination of the row and column coordinates of the winners:

```
    ch = mod(c-1,7) + 1;
    cv = floor((c-1)/7) + 1;
```

In labeling, the offset of the even rows with respects of the odd rows must be taken into account by using the `shift` parameter:

```
    if mod(cv+1,2)== 0
        shift = -.2;
    else
        shift = .3;
    end
    text(ch+shift,cv, [L1(u) L2(u) L3(u) L4(u) L5(u)], ...
    'FontSize',10)
end
```

The computed SOM is shown in Fig. 23. The U matrix for the binary-attribute example is shown in Fig. 24.

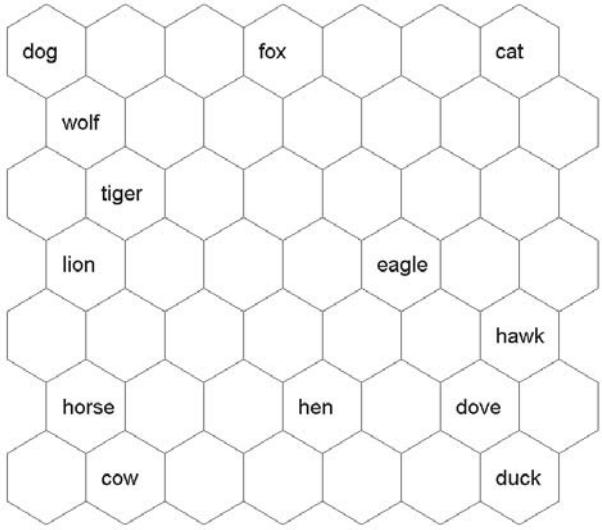


Fig. 23. The SOM of 13 animals.

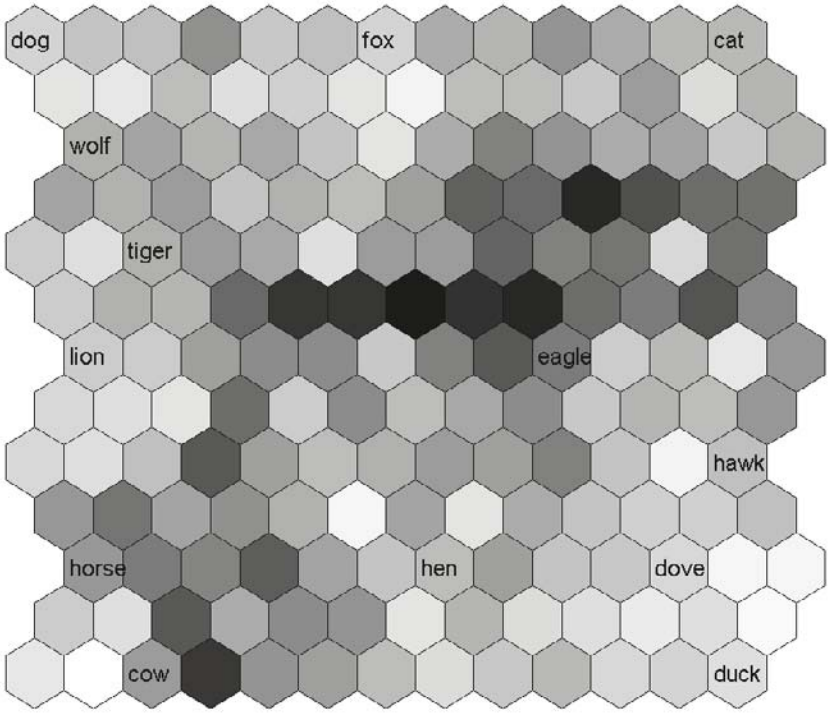


Fig. 24. The U matrix for the 13 animals.

17 Ordering of items by their functional value

Mapping of footware according to their distinctive features

Objective:

The similarities of items can also be based on their functional values in use. We exemplify this by a collection of different footware.

In this example we want to order 20 pieces of footware, not according to their visual properties, but by their *purpose* and *functional value*. The items are shown in Fig. 25.

Fig. 25. Collection of footware

Top row:
1. Baby boy's shoe. 2. Dress shoe. 3. Roman soldier's sandal. 4.Dutch wooden sandal (clomp). 5. Japanese wooden sandal (geta).

Second row:
6. Baby girls shoe. 7. Fashion shoe. 8. Boot for rough terrain. 9. Rubber overshoe. 10. Rubber boot.

Third row:
11. Ladies' fashion shoe. 12. Men's ankle boot. 13. Men's sporting sandal.
14. Ball player's shoe. 15. Health sandal.

Fourth row:
16. Ladies' casual shoe. 17. Quaint ladies' fashion shoe. 18. Ladies' walking shoe. 19. Espadrille. 20. Beach sandal.

The *distinctive features* of these footware are:

1. Designed for babies
2. " men and women
3. " men only
4. " women only
5. " all seasons
6. " cold season
7. " hot season
8. " indoor use
9. " fashion
10. " leisure
11. " work
12. " for sporting
13. " for military or heavy outdoor use
14. Highly waterproof
15. Against rough and rocky ground

The computation of the SOM is made by the following script In the beginning the data matrix `X` is defined. Its *rows* correspond to the various footware defined above, and its *columns* define the distinctive features.

```
X = [1 0 1 0 1 0 0 0 0 0 0 0 0 0 0
     0 0 1 0 0 0 0 1 1 0 0 0 0 0 0
     0 0 1 0 1 0 0 0 0 0 0 0 1 0 1
     0 1 0 0 1 0 0 0 0 0 1 0 0 0 1
     0 1 0 0 1 0 0 0 0 0 1 0 0 0 1
     1 0 0 1 0 0 0 1 0 0 0 0 0 0 0
     0 0 1 0 0 0 0 1 1 0 0 0 0 0 0
     0 0 1 0 0 1 0 0 0 0 0 1 1 0 1
     0 0 1 0 0 1 0 0 0 0 0 0 0 1 0
     0 1 0 0 1 0 0 0 0 0 0 0 0 1 0
     0 0 0 1 0 0 0 1 1 0 0 0 0 0 0
     0 0 1 0 0 1 0 0 0 0 0 0 0 0 0
     0 0 1 0 0 0 1 0 0 0 0 1 0 0 1
     0 1 0 0 0 0 0 1 0 0 0 1 0 0 0
     0 1 0 0 0 0 0 1 0 0 1 0 0 0 0
     0 0 0 1 0 0 1 0 0 1 0 0 0 0 0
     0 0 0 1 0 0 0 1 1 0 0 0 0 0 0
     0 0 0 1 0 0 1 0 0 1 0 0 0 0 0
     0 0 0 1 0 0 1 0 0 1 0 0 0 0 0
     0 1 0 0 0 0 1 0 0 1 0 0 0 0 0];

msize = [8 8];
lattice = 'hexa';       % hexagonal lattice
neigh = 'gaussian';     % neighborhood function
radius_coarse = [3 1]; % " radius, coarse [initial final]
radius_fine = [1 1];   % " radius, fine [initial final]
```

```
trainlen_coarse = 30;  % cycles in coarse training
trainlen_fine = 20;    % cycles in fine training
smI = som_lininit(X, 'msize', msize, 'lattice', lattice,...
    'shape', 'sheet');
smC = som_batchtrain(smI, X, 'radius', radius_coarse,...
    'trainlen', trainlen_coarse, 'neigh', neigh);
sm = som_batchtrain(smC, X, 'radius', radius_fine,...
    'trainlen', trainlen_fine, 'neigh', neigh);
M = sm.codebook;
norms2 = sum(M.*M,2);
som_cplane('hexa', msize, 'none');
for u = 1:20
    X1 = X(u,:)';
    Y = norms2 - 2*M*X1;
    [C,c]= min(Y);
    ch = mod(c-1,8) + 1;
    cv = floor((c-1)/8) + 1;
```

Now we have obtained the coordinates `(ch,cv)` into which the symbols of the various footware have to be plotted. We are drawing the pictures manually, but the corresponding places on the `som_cplane` are computed first numerically. Because there are multiple collisions, the `shift2` parameters separate the item numbers in the display.

```
    if mod(cv,2) == 1
        shift1 = -.2;
    else
        shift1 = .3;
    end
    if  u == 11 || u == 12 || u == 16
        shift2 = -.3;
    else if  u == 4 || u == 7 || u == 9 ...
            || u == 17 ||  u == 18
        shift2 = .3;
        else
            shift2 = 0;
        end
    end
    text(ch+shift1, cv+shift2, num2str(u),  'FontSize',10)
end
```

The illustrative clustering, the order in which may be self-explanatory, is provided in Fig. 26.

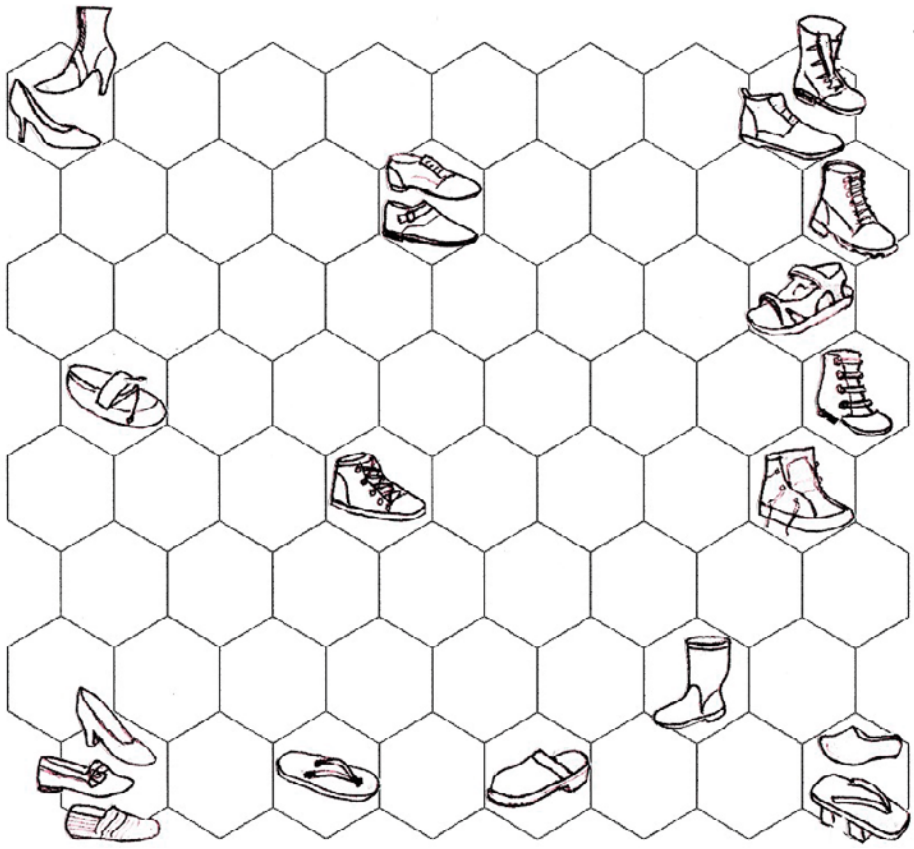


Fig. 26. Ordering of various footwear on the SOM according to their usage. The pictures of different footwear were drawn at the coordinates `ch, cv` determined by the above SOM script. Note that it would have been impossible to define dichotomies with respect to all of the 15 distinctive features. However, it is possible to discern various polarizations: mens' footware on the right, ladies' footware on the left. Note in particular the locations of the baby shoes! Fashion shoes lie at top-left and top-middle, light shoes at bottom-left, strong protective shoes at top-right, and various sandals as well as the rubber booth that are worn by both ladies and men lie at the bottom. The sporting shoe that is also worn by ladies as well as by men lies in the middle.

18 Two-class separation of mushrooms on the basis of visible attributes

This is an extension of binary attributes to symbolic attributes that have more than two values. The distributions of mushroom data, represented by symbolic attributes, are shown as histograms on the SOM groundwork, and the edible and poisonous mushrooms are separated automatically

Objective:

This is a well-known benchmarking example of classifying North-American mushrooms into edible and poisonous species, based solely on their visible attributes. This example is now handled by the SOM.

The problem. This example classifies 23 species of gilled mushrooms growing in North-America. The Audubon Society Field Guide [55] describes them in terms of their physical characteristics, and classifies them as *definitely edible*, *definitely poisonous*, or of unknown edibility and *not recommended*.

The present realistically-looking benchmarking data set has been designed and donated in 1987 for mathematical studies by Jeff Schlimmer (Jeffrey.Schlimmer @ a.gp.cs.cmu.edu). From that homepage you can find the attribute information given also below, and via the link *agaricus-lepiota.data* on Schlimmer's homepage the attribute values are downloadable. The mushrooms are described by 22 categorical (discrete) attributes. The present test data contain 8124 instances where the "definitely poisonous" and "not recommended" classes are combined into a single "poisonous" class.

For example, the first attribute "cap-shape" can have a value of {b, c, x, f, k, s} which stand for {bell, conical, convex, flat, knobbed, sunken}, respectively.

Attribute information:

1. cap-shape: bell=b, conical=c, convex=x, flat=f, knobbed=k, sunken=s
2. cap-surface: fibrous=f, grooves=g, scaly=y, smooth=s
3. cap-color: brown=n, buff=b, cinnamon=c, gray=g, green=r, pink=p, purple=u, red=e, white=w, yellow=y
4. bruises?: bruises=t, no=f
5. odor: almond=a, anise=l, creosote=c, fishy=y,foul=f, musty=m, none=n, pungent=p, spicy=s
6. gill-attachment: attached=a, descending=d, free=f, notched=n
7. gill-spacing: close=c, crowded=w, distant=d
8. gill-size: broad=b, narrow=n
9. gill-color: black=k, brown=n, buff=b, chocolate=h, gray=g, green=r, orange=o, pink=p, purple=u, red=e, white=w, yellow=y
10. stalk-shape: enlarging=e, tapering=t

11. stalk-root: bulbous=b, club=c, cup=u, equal=e, rhizomorphs=z, rooted=r,
. missing=?
12. stalk-surface-above-ring: fibrous=f, scaly=y, silky=k, smooth=s
13. stalk-surface-below-ring: fibrous=f, scaly=y, silky=k, smooth=s
14. stalk-color-above-ring: brown=n,buff=b, cinnamon=c, gray=g, orange=o,
. pink=p, red=e, white=w, yellow=y
15. stalk-color-below-ring: brown=n, buff=b, cinnamon=c, gray=g, orange=o,
. pink=p, red=e, white=w, yellow=y
16. veil-type: partial=p, universal=u
17. veil-color: brown=n, orange=o, white=w, yellow=y
18. ring-number: none=n, one=o, two=t
19. ring-type: cobwebby=c, evanescent=e, flaring=f, large=l, none=n, pendant=p,
. sheathing=s, zone=z
20. spore-print-color: black=k, brown=n, buff=b, chocolate=h, green=r, orange=o,
. purple=u, white=w, yellow=y
21. population: abundant=a, clustered=c, numerous=n, scattered=s, several=v,
. solitary=y
22. habitat: grasses=g, leaves=l, meadows=m, paths=p, urban=u, waste=w,
. woods=d

The *agaricus-lepiota.data* data set is given as a 8124-row symbolic data matrix, of which the first five rows are copied below. The first symbol `e` or `p` on each row stands for "edible" or "poisonous", respectively. The next 22 symbols represent the *values* of the 22 physical attributes. Attribute No. 11 contains about 30 per cent of missing data, and is simply *ignored* in the following computations.

Mushroom table `Mt`:

```
Mt = [p,x,s,n,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,k,s,u
      e,x,s,y,t,a,f,c,b,k,e,c,s,s,w,w,p,w,o,p,n,n,g
      e,b,s,w,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,n,m
      p,x,y,w,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,k,s,u
      e,x,s,g,f,n,f,w,b,k,t,e,s,s,w,w,p,w,o,e,n,a,g
      ...            8124 rows)                  ...  ]
```

Conversion of symbols into unit vectors. We see that the source data are given in *symbolic form*, but the SOM works with *real numbers*. The models in the SOM are usually *metric vectors*, and in the training process they modify their values *gradually*.

Notice that you cannot compare the magnitudes of two symbolic attributes: either you have them or you don't. Now we may think that these attributes are *unit vectors in higher-dimensional spaces*. To explain what I mean, consider the first attribute "cap-shapedness" that has one of the *six* possible values of {b, c, x, f, k, s}. Let us introduce the six units vectors in this six-dimensional feature space. They are formally: `b = [1 0 0 0 0 0], c = [0 1 0 0 0 0], x = [0 0 1 0 0 0], f = [0 0 0 1 0 0], k = [0 0 0 0 1 0]` and `s =[0 0 0 0 0 1]`. The second attribute "cap-surfaceness" has one of *four* possible values {f,g,y,s} and is described by one of the four unit vectors `f = [1 0 0 0 ], g = [0 1 0 0`

], y = [0 0 1 0], and s = [0 0 0 1]. The third attribute corresponds to *ten-dimensional unit vectors*, and so on. So, when we then *concatenate* the unit vectors that correspond to the due abstract symbols, the first four input items (not regarding the class symbols p, e and e in the first location) start with

```
x = [0 0 1 0 0 0], s = [0 0 0 1],  ...
x = [0 0 1 0 0 0], s = [0 0 0 1],  ...
b = [1 0 0 0 0 0], s = [0 0 0 1],  ...
x = [0 0 1 0 0 0], y = [0 0 1 0],  ...
```

Concatenated in the horizontal and vertical directions, the unit vectors constitute the *input data matrix*

```
X = [0 0 1 0 0 0 0 0 0 1  ...
     0 0 1 0 0 0 0 0 0 1  ...
     1 0 0 0 0 0 0 0 0 1  ...
     0 0 1 0 0 0 0 0 1 0  ...
    ...

    ...                                  ]
```

When we neglect attribute No.11 that has missing values, the concatenated unit vectors that form the input vectors have the dimensionality of 119. Although they are *binary* vectors, at the same time they can also be regarded as *real* vectors, and the *real model vectors of the SOM* are then trying to *approximate* them metrically.

Automatic conversion. The conversion of the large symbolic mushroom table Mt into the large numerical input data matrix X of the SOM can be made conveniently by the following script. Below, the script starts with the *attribute array* A. Both Mt and A are *vertical symbol arrays*, the former with the dimensionality of 8124 by 22 (note: the first symbol on each row of M is the symbol of classification "edible/poisonous"), and the latter array A with the dimensionality of 8124 by 12. (In vertical symbol arrays in which the strings of symbols have different lengths, all rows must be filled with blank symbols so that the length is always the same, here equal to 12):

```
A = ['bcxfks      '
     'fgys        '
     'nbcgrpuewy  '
     'tf          '
     'alcyfmnps   '
     'adfn        '
     'cwd         '
     'bn          '
     'knbhrgopuewy'
     'et          '
```

```
'fyks        '
'fyks        '
'nbcgopewy   '
'nbcgopewy   '
'pu          '
'nowy        '
'not         '
'ceflnpsz    '
'knbhrouwy   '
'acnsvy      '
'glmpuwd     '];

X = zeros(81241,119);
for s = 1:8124
    d = 0;
    for a = 2:22
        A1 = A(a-1,:);
        A2 = A1(find(A1~=' '));
        for t = 1:length(A2)
            if Mt(s,a) == A2(t)
               X(s,t+d) = 1;
            end
        end
        d = d + length(A2);
    end
end
```

The SOM script. Assuming that the input data matrix `X` has been constructed in the above fashion and saved as the file `mushroomdata.mat`, the rest of the script is straightforward. With the comments it may be understandable as such.

Because in this example we are using so-called hit diagrams which separate the input items according to their classification, it is better to use the component vectors of the input data matrix `X` *in the computation of the winners.*

```
file = 'mushroomdata';
load ([file '.mat'])
msize = [15 20];
lattice = 'hexa';
neigh = 'gaussian';
radius_coarse = [5 1];
trainlen_coarse = 30;
radius_fine = [1 1];
trainlen_fine = 30;
smI = som_lininit(X, 'msize', msize, 'lattice', lattice, 'shape', ...
'sheet');
```

```
smR = som_batchtrain(smI, X, 'radius', radius_coarse, 'trainlen', ...
trainlen_coarse, 'neigh', neigh);
sm = som_batchtrain(smR,X, 'radius', radius_fine, 'trainlen', ...
trainlen_fine, 'neigh', neigh);
M = sm.codebook;
norms2 = sum(M.*M),2);
hits = zeros(8124, 2);   %  No. of winners mapped into an SOM node
                         %  in either of the two subplots
for u=1:8124
     U = X(u,:)';
     Y = norms2 - 2*M*U;
     [C,c]= min(Y);
```

After that, the winners of the edible mushrooms are mapped into node `c` in one subplot, and the winners of the poisonous mushrooms into node `c` in another subplot, respectively. These subplots use the *hit variables* `hits(c,1)` and `hits(c,2)`, which accumulate and display the hit diagrams.

```
     if Mt(u,1) == 'e'
        hits(c,1) = hits(c,1) + 1;
     end
     if Mt(u,1) == 'p'
        hits(c,2) = hits(c,2) + 1;
     end
end
```

The plotting of Fig. 27 starts with two rows that define the gray scales. The commands after that may be self-explanatory.

```
figure;
colormapigray = ones(64,3) - colormap('gray');
colormap(colormapigray);
for k=1:2
    subplot(1,2,k);
    som_cplane(sm, hits(:,k));
    set(gca,'FontSize',10);
    if k == 1
       title('Edible');
    end
    if k == 2
       title('Poisonous');
    end
end
savefilename = 'mushroomplots';
print('-dpng', [savefilename '.png']);
```

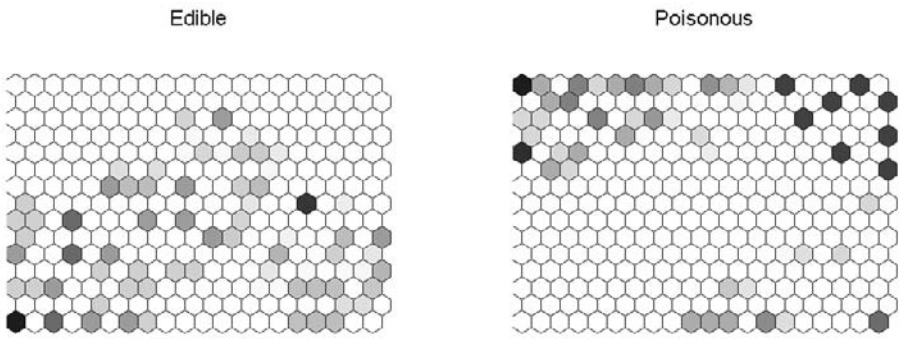


Fig. 27. The distributions of hits of the two classes of mushrooms on the SOM.

Discussion. The results of classification are shown as the two histograms on the SOM, "Edible" vs. "Poisonous." The numbers of winners on each node of the two subplots are shown by gray-level diagrams. There is very small overlapping in these subplots: only one dot (fifth row from bottom, third dot from the right) coincides in both graphs. So practically the SOM separates these classes.

In the earlier benchmarking studies of algorithms that analyzed these data, one was evaluating the classification accuracy, in a search of distinctive attributes that would clearly separate the edible mushrooms from the poisonous ones. Such a combination of key attributes was never found. I think that such a study should also be made by using test data that are *statistically independent* of the training data. If the data set is very restricted, one can used the "leave-one-out" method: if we have N items in the data base, here $N = 8124$, then one may repeat the test by constructing the SOM $N - 1$ times, every time leaving one of the items for a test item and constructing the SOM out of the rest of the $N - 1$ items. That would mean a lot of work in this case. However, it is not the purpose to carry out any statistical studies here, so Fig. 27 may suffice for a demonstration.

19 Clustering of scientific articles

Separating the histograms of four related classes of articles on the SOM groundwork

Objective:

So far we have based the clustering of items on their attribute values. In this section we introduce a new set of features, namely, the vocabulary that is used in a document. We show that it is possible to distinguish and classify documents by means of weighted word histograms, and show the distributions of the document classes on the SOM groundwork.

Almost every SOM computation starts with a more or less extensive preprocessing of the raw data. The next example has been chosen to demonstrate how *classes of documents*, such as scientific articles, can be represented on the SOM and distinguished from each other on the basis of the usage of words. We describe how *sets of documents* can be mapped onto the SOM array as *histograms plotted onto the SOM groundwork*. Each input data item is a *document*. One frequently used method to identify documents in text analysis is to describe them by their *word histograms*. In order to increase class separation, all words are first reduced into their stem forms. Then there exist *stopwords*, i.e., short function words, such as 'a', 'the', 'is', 'and', 'which', 'on,' 'like' etc. that can be removed from the text without decreasing the statistical information contained in it. Furthermore the words in the histograms can be *weighted* by statistical arguments: e.g., one frequently used weight is the *negentropy* of the words. Another viable choice is the *inverse document frequency (IDF)* of the word. The *document frequency* is the number of documents in which a particular word occurs, and the IDF is its inverse. These weights are very effective in increasing class separation of the documents.

Reuters data. The text corpus used in this experiment was taken from a collection of articles published by the *Reuters corporation*. No original articles were available to us; however, Lewis et al. [53], who prepared this corpus for benchmarking purposes, have preprocessed the textual data, ignoring stopwords and reducing the remaining word forms into their stems. The word histograms of the documents were weighted by specific statistical coefficients introduced by Manning and Schütze [58]. J. Salojärvi from our laboratory then selected a 4000-document subset from the Reuters collection, restricting only to such articles that could be assigned to one of the following classes:

1. Corporate-Industrial.
2. Economics and Economic Indicators.
3. Government and Social.
4. Securities and Commodities Trading and Markets.

There were 1000 documents in each class. Salojärvi then picked up those 233 words that appeared most often in the selected texts and made a corresponding 233-element selected-word histogram for each document. He then weighted the elements of the 233-dimensional histograms of the documents by the coefficients used by Manning and Schütze [58]. In this way we have obtained the vectorial representations of the input data items: 4000 vectors with 233 real-valued input components each.

Since this classification problem is already more demanding than the previous ones and contains a lot of more data, the different parts of the script will now be explained in detail.

Loading of input data. The input data were given as the 4000 by 234 matrix `documentdata`, in which each row is of the form `[label element(1) ... element(233)]`, this time without any stored variable names.

First, the input data are loaded. The classifications of the data into the four classes are defined by the first elements on each row of the `documentdata` matrix, and loaded by the instructions

```
load('documentdata.mat');
labels = documentdata(:,1);
```

The 4000-element vector `labels` now identifies the classes of the documents. Then, the numerical input data `X` are defined by

```
X = zeros(4000,233);
for i = 1:233
     X(:,i) = documentdata(:,i+1);
end
```

Checking for missing data. Especially with big statistical data bases, one cannot be always sure whether accidentally some of the input vectors might contain only zero elements and should be ignored. Therefore we start the present problem with a typical operation to verify and validate only nonzero data. As a matter of fact, it indeed turned out in posterior checking that in this application, in spite of careful (automatic) selection of the input data, one input vector out of the 4000 was zero!

The following lines detect all zero input vectors and form a new vector `labels`, in which the value of the class label is set to 0 for the zero vector.

```
nonzeroindex = find(sum(X'));
X = X(nonzeroindex,:);
labels = labels(nonzeroindex);
```

Initialization and the two training phases. Only in very simple problems one training phase will suffice. In a bigger problem like this we already have to use two training phases, a coarse and a fine one, for one thing because we want to make use of an *automatic stopping rule*. During the coarse training phase, the radius of the neighborhood function decreases linearly; during the fine training phase it must be held constant, whereupon sooner or later the learning terminates exactly.

The size of the SOM array for this problem was selected to be 10 by 15 nodes. Let us start with initialization, which preferably ought to be linear:

```
smI = som_lininit(X, 'msize', [10 15], 'lattice', ...
'hexa', 'shape', 'sheet');
```

For the coarse training we start with the SOM values obtained in initialization, which are still in the *struct* form smI. Nonetheless they can be inserted into the `som_batchtrain` function. Now we must use separate parameters in coarse and fine training, respectively. Let us take

```
radius_coarse = [4 1] ;
radius_fine = [1 1];
trainlen_coarse = 30;
trainlenlen_fine = 10;
```

In coarse training we start with the initialized SOM `smI` and end up with the SOM `smC`. In fine training we start with `smC` and end up with the final value `sm`:

```
smC = som_batchtrain(smI, X, 'radius_coarse', [4 1], ...
'trainlen_coarse',30, 'neigh', 'gaussian');
sm = som_batchtrain(smC, X, 'radius_fine', [1 1], ...
'trainlen_fine', 1,  'neigh', 'gaussian' )
```

Notice that the coarse and fine training functions are formally similar, they only use different parameters.

Stopping rule. If we hold `radius_fine` constant during fine training, the algorithm terminates in a finite number of cycles, at least if the topographical order of the SOM has been achieved in coarse training. To that end the number of coarse training cycles must be sufficient, say, 30 (we shall test this in a number of benchmarking runs below).

The stopping criterion is that the SOM matrix does not change in any further training cycles. In practice we can test this quicker by following the norm of the SOM matrix, which has to achieve a constant value, too. So we attach the following lines after the fine training cycles:

```
R = 1000000;  % any large number bigger than norm(sm.codebook)
while R - norm(sm.codebook) > 0
      R = norm(sm.codebook);
      sm = som_batchtrain(sm, X, 'radius_fine', [1 1],  ...
      'trainlen', 1, 'neigh', gaussian);
end
```

Forming the histograms. The SOM was computed using all of the 4000 input vectors for training. After that, the distributions of the documents of each of the four classes on the SOM were determined. In testing, the vectors of each class in turn are input to the SOM (but this time without any further training of the SOM), finding the *winners* (hits) for each input vector on the SOM array, and accumulating the hits on each winner node of the array for each of the four classes. The number of hits on each node is shown by a shade of gray (Fig. 28). The hit diagrams are automatically normalized by the MATLAB, relative to their maximum value. So the four hit diagrams have different scales.

The documents in this problem were classified into four classes, and each document was provided with a numerical classifier (label) 1 through 4.

One way of using the SOM is to show by several subplots how the different data classes are mapped onto the SOM array. The number of "hits" of data items on the various nodes constitute a kind of "histogram," which is visualized by shades of gray.

Let `c` denote a node and let `i` be a class label. Then the four histograms ("hits"), defined by the matrices `hits(c,i)` are computed by the following script:

```
M = sm.codebook;
norms2 = sum(M.*M,2);
for u = 1:50
     hits = zeros(size(M, 1), 4);
     for i = 1:4
          classvectors = X(find(labels == 1), :);
          for u = 1:size(classvectors, 1)
             X1 = classvectors(u, :)';
             Y = norms2 - 2*M*X1;
            [C,c] = min(Y);
            hits(c, i) = hits(c, i) + 1;
          end
```

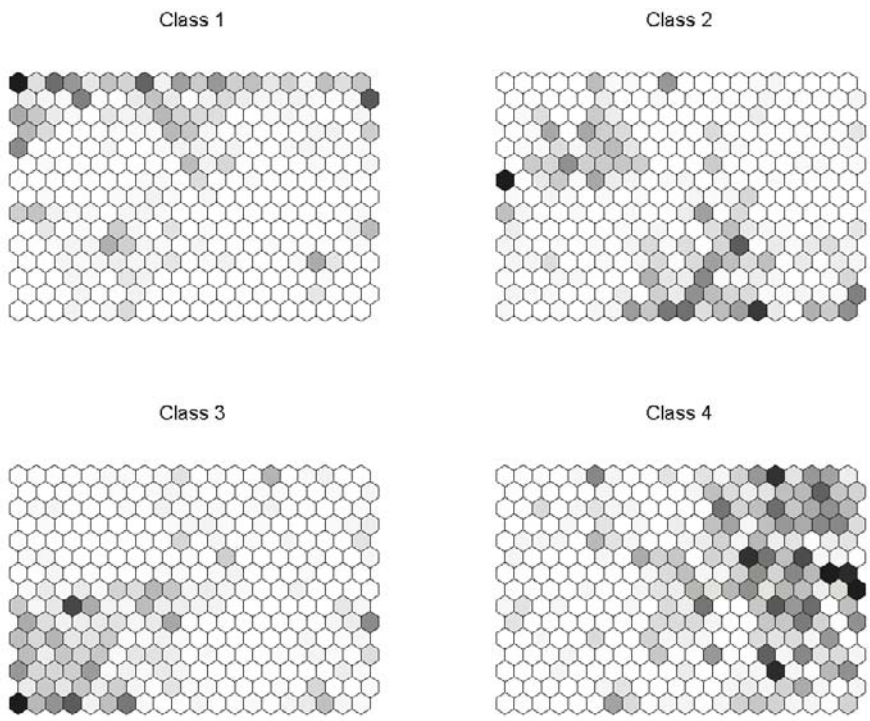


Fig. 28. Distribution of the documents of the four classes on an SOM. Class 1: Corporate-Industrial. Class 2: Economics and Economic indicators. Class 3: Government and Social. Class 4: Securities and Commodities Trading and Markets. The SOM was computed using all of the documents. The mapping of documents of each class onto the same SOM is shown in the four subplots, in which the number of documents mapped on a particular node is shown by a shade of gray. The process has converged fully.

```
    end
end
```

The four histograms, `hits(c, i)` are plotted by the following lines:

```
figure;
colormapigray = ones(64, 3) - colormap('gray');
colormap = colormapigray);
for i = 1:4
     subplot(2,2,i);
     som_cplane(sm, hits(:, i));
     set(gca, 'FontSize'. 10);
     title([Class ' num2str(i)]);
```

```
end
```

19.1 The complete script

Because this is a good typical example of the application of the SOM in practice, we write the complete script here once again:

```
% Parameters
msize = [15 20];
lattice = 'hexa';
neigh = 'gaussian';
radius_coarse = [4 1];
trainlen_coarse = 20;
radius_fine = [1 1];
trainlen_fine = 20;

% Data loading
load('documentdata.mat');
labels = documentdata(:,1);
X = zeros(4000,233);
for i = 1:233
    X(:,i) = documentdata(:,i+1);
end

% Preprocessing
nonzeroindex = find(sum(X'));
X = X(nonzeroindex,:);
labels = labels(nonzeroindex);

% Initialization
smI = som_lininit(X, 'msize', msize, 'lattice', ...
'hexa', 'shape', 'sheet')

% Coarse training
smC = som_batchtrain(smI, X, 'radius', radius_coarse, ...
'trainlen', trainlen_coarse, 'neigh', 'gaussian' )

% Fine training
sm = som_batchtrain(smC, X, 'radius', radius_fine, ...
'trainlen', trainlen_fine, 'neigh', 'gaussian' )

% Stopping rule
R = 1000000; % any large number bigger than...
norm(sm.codebook)
while R - norm(sm.codebook) > 0
```

```
R = norm(sm.codebook);
sm = som_batchtrain(sm, X, 'radius', radius_fine, ...
'trainlen', 1, 'neigh', neigh);
end

% Computation of the histograms
hits = zeros(size(M, 1), 4);
norms2 = sum(M.*M,2);
for i = 1:4
    classvectors = X(find(labels == 1), :);
    for u = 1:size(classvectors, 1)
        X1 = classvectors(u, :)';
        Y = norms2 - 2*M*X1;
        [C,c] = min(Y);
        hits(c,i) = hits(c,i) + 1;
    end
end

% Plotting
figure(1);
colormapigray = ones(64, 3) - colormap('gray');
colormap = colormapigray);
for i = 1:4
    subplot(2,2,i);
    som_cplane(sm, hits(:, i));
    set(gca, 'FontSize', 10);
    title([Class ' num2str(i)]);
end

% Saving the results
savefilename = '4histograms';
save(savefilename, 'M');
print('-.dpng', [savefilename '.png']);
```

This completes the script.

20 Convergence tests; benchmarking

Computing times vs. parameter values

Objective:

One frequently asked question is how many training cycles one needs to build up an SOM. If no prior experience exists, it is best to use a trial-and-error method to study the first example. However, there seem to exist some general rules to answer this question, and for larger problems, one can also construct a stopping rule that tells when the SOM matrix does not change any longer in training. This section contains benchmarking studies on training lengths.

It was stated earlier that a two-phase SOM converges in a finite number of batch cycles, if the neighborhood function is held constant in fine training. We believe that the document-classification problem is statistically rather typical, and we perform a small set of benchmarking experiments on it. The number of input items was 4000, and this figure represents an experiment of a modest size. Also the size of the SOM array, [10 15] is typical for many SOM studies published in the literature.

20.1 The original Reuters example

The following table (Table 2) shows the number of fine training cycles necessary for convergence vs. typical coarse training. The first column represents the initial radius in coarse training, and the second column the number of coarse training cycles used.

The number of fine training cycles for full convergence (when the SOM is no longer altered in further fine training cycles) was determined for a set of combinations of the initial neighborhood radius in coarse training, and the number of batch cycles in coarse training, respectively. It is generally known that the benchmarking of computing times is a bit questionable, because from one run to another, the use of the memory hierarchy is affected by the history of the computations.The given figures are averages from several runs, but still only approximative. Nonetheless, one might expect that the more coarse cycles one is using, the less fine cycles one would need. However, as the figures show, this is neither always true. There seems to be an *optimal combination* of *both* coarse *and* fine cycles for each initial neighborhood radius. The explanation is the following. In the linear initialization, an *even* distribution of the model vectors is set. The coarse training phase initially *disturbs* this evenness, because it is starting to match the model vectors with the eventually "crooked" density function, and this disturbance is the larger, the wider the neighborhood function is in the beginning and the longer the coarse training phase is. So it is clear from these figures that it is a better policy to be "gentle" in coarse training, and to let the SOM adjust itself to the density function of the input vectors smoothly.

Table 2. The number of fine training cycles, as a function of the initial neighborhood radius in coarse training and the number of coarse training cycles

Init. coar. rad.	Coar. cycl.	Time	Fine cycl.	Time	Total
3	23	4.6	17	3.2	7.8
3	24	5.0	14	2.4	7.4
3	25	5.0	24	4.6	9.6
4	17	3.0	23	3.7	6.7
4	18	3.3	14	3.0	6.3
4	19	4.7	16	3.1	7.8
5	25	5.0	34	6.5	11.5
5	26	4.6	20	3.6	8.2
5	27	5.0	26	5.1	10.1

One notices that with the initial coarse radius 4, the minimal computing time is 6.3 seconds, which is obtained with 18 coarse batch cycles and 14 fine batch cycles.

From this experiment we obtain a rule of thumb, which seems to be true rather generally:

> *With batch training and linear initialization, the initial coarse radius should be on the order of 20 per cent of the longer side of the SOM array.*

Unfortunately there does not exist any expression for the optimal number of coarse training cycles. However, a quick check can be made. With the initial coarse neighborhood radius 4 and 30 coarse training cycles, the algorithm terminated in 35 fine training cycles, and the computing time was 15.2 seconds. So, if the computing time is not important, one might first try a few dozens of coarse training cycles.

20.2 Upscaling the SOM matrix

In this subsection we study the effect of the array size on the convergence of the SOM algorithm. First we *double* the *linear dimensions* of the SOM array, i.e., `msize = [30 40]`. The initial neighborhood radius must also be doubled: `radius_coarse = [8 1]`. Now we carry out the computations for the coarse radius 8 and compare the results with those of Table 2.

Table 3. The number of fine training cycles for a 1200-node SOM array, as a function of the number of coarse training cycles

Init. coar. rad.	Coar. cycl.	Time	Fine cycl.	Time	Total
8	26	22.6	17	17.8	40.4
8	27	23.7	19	9.4	33.1
8	28	25.2	9	9.4	34.6
8	29	25.4	11	11.3	36.7
8	30	26.0	13	13.6	39.6

So, the SOM array has now four times as many nodes as before, and the computing time has become roughly fivefold. The optimal number of coarse cycles is on the same order of magnitude as with the smaller SOM array. At least it does not grow rapidly with the size of the SOM array.

Next we *quadruple* the linear dimensions of the original SOM array, i.e., `msize = [60 80]`. The initial neighborhood radius must also be quadrupled: `radius_coarse = [16 1]`. Now we carry out the computations for the coarse radius 16 and compare the results with those of Tables 2 and 3.

Table 4 . The number of fine training cycles for a 4800-node SOM array, as a function of the number of coarse training cycles

Init. coar. rad.	Coar. cycl.	Time	Fine cycl.	Time	Total
16	8	64	12	275	339
16	9	69	10	225	294
16	10	73	9	200	273
16	11	79	9	200	279
16	12	85	8	176	261
16	13	91	8	177	268
16	14	97	9	200	297
16	15	103	10	225	328
16	16	109	10	225	334
16	17	116	10	227	343

So, the SOM array has now 16 times as many nodes as initially, but the computing time has only become roughly seven- or eight-fold. This may be due to the fact that in the large maps the neighborhoods, during the coarse training, contain plenty of nodes, which has a strong smoothing action on learning. This also indicates that for large sizes of the SOM array, the number of coarse training cycles needed may be smaller than in the small maps.

21 Calibration of the nodes by the Bayesian decision rule

Indication of classes by pseudo-colors

Objective:

Sometimes we need a compressed representation of the distribution of the classes on the SOM. In a single image we cannot show the forms of the class distributions, but we may try to create the class borders onto the SOM groundwork. The statistically most justified determination of the borders is based on the Bayesian method.

Again we use the Reuters example for demonstration. First we explain how the nodes of an SOM can be calibrated using *majority voting* over the class labels.

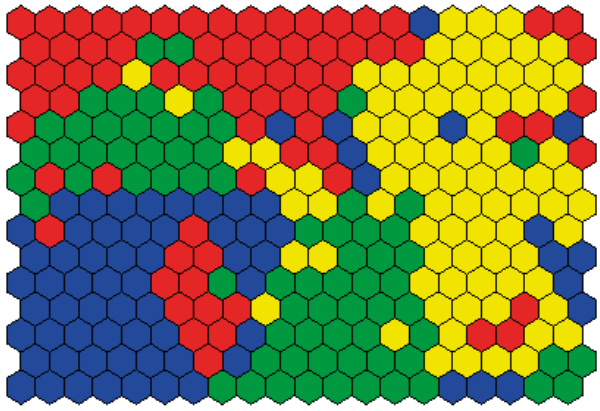


Fig. 29. Calibration of the SOM according to the majority of classes of documents that have been mapped on the various nodes. The majority of classes has neen indicated by colors. Red: Corporate-Industrial. Green: Economics and Economic indicators. Blue: Government and Social. Yellow: Securities and Commodities Trading and Markets.

We show in Fig. 29 how the mapping of the four classes onto the SOM can be illustrated in one color display. First we carry out a *majority voting* over the class labels: i. e., we check what is the majority of class labels of those documents that have been mapped onto a particular node, and we color the node correspondingly: red for samples of Class 1, green for Class 2 samples, blue for Class 3 samples, and yellow for Class 4 samples, respectively. This kind of classification of the nodes is said to be based on the *Bayesian decision rule*, and it is one kind of *calibration* of the SOM nodes.

Naturally, if we associate with each document a *pointer* to the location (node) where it is mapped, we can easily check what its most probable classification is,

as esteemed on the basis of its vocabulary, and what are those documents that are most similar to it, because they can be found at the neighboring nodes. For this purpose the nodes must be provided with *symmetric* or *double-linked pointers*, i.e., pointers from the documents to the nodes, and pointers from the nodes to the documents, respectively. This arrangement is particularly important if we receive a new, unknown document, and want to see the classification of that node into which the document is mapped on the SOM.

The following piece of script implements the computation of the color picture of Fig. 29:

```
% Majority of class labels at a node
nodelabels = zeros(300,1);
for i=1:length(nodelabels)
    [C,c] = max(hits(i,:));
    nodelabels(i) = c;
end

% Range of colors
mixturelabelcolormap = [ones(1,64); ...
 [1:-1/63:0];[1:-1/63:0]] [[1:-1/63:0]; ...
ones(1,64); ...
[1:-1/63:0]] [[1:-1/63:0]; [1:-1/63:0]; ...
ones(1,64)][ones(1,64); ones(1,64); [1:-1/63:0]] ...
[0 1; 0 1; 0 1]]';

% MATLAB structure for SOM Toolbox instructions
som_topology = struct('type', 'som_topol', ...
'msize', msize, 'lattice', ...
'hexa', 'shape', 'sheet');
mixturecolor = nodelabels*64;

% Plotting
figure;som_cplane(som_topology, ...
mixturelabelcolormap(round(mixturecolor),:));

% Saving
savefilename = 'Bayespicture' ;
print('-dpng', [savefilename '.png']);
```

22 Calibration of the nodes by the kNN rule

Another common calibration of the SOM nodes

Objective:

Sometimes we are more interested in a smooth labeling of the SOM, instead of the most accurate classification of unknown input items. Then we can use the kNN labeling, explained next.

Especially if there is only a small number of input data items available, in relation to the size of the SOM array, so that the above majority voting makes no sense (e.g., there are too many ties, or there are no hits at some of the models), one can apply the so-called *k-nearest-neighbors (kNN)* labeling method. For each model, those k input data items that are closest to it (in the metric applied in the construction of the SOM) are searched, and a majority voting over them is carried out to determine the most probable labeling of the node. In the case of a tie, the value of k is increased until the tie is resolved. Usually k is selected to be on the order of half a dozen to a hundred, depending on the number of input data items and the size of the SOM array.

In the present Reuters-data example, we had about 13 input items mapped into each node on the average, so there is actually no need to resort to the kNN rule, but it is anyway applied here for comparison. The following script is needed, where the earlier notations of `norms2`, `M` and `X` are used:

```
% Take originally k nearest neigbors into account
k = 10;
kOrg = k;
nodelabels = zeros(size(M,1),1);

% Find the labels of nearest neighbors ('in')
 for i=1:length(nodelabels)
     [v,in] = sort(norms2 - 2*M*X');
     k = kOrg;
     v21 = 0;
     v22 = 0;
     while v21 == v22  % if there is a tie,  increment k

             % No. of varous classes ('labs') in ...
             % the neigboring documents
              labs = [length(find(labels(in(1:k)) == 1)) ...
              length(find(labels(in(1:k)) == 2)) ...
              length(find(labels(in(1:k)) == 3)) ...
              length(find(labels(in(1:k)) == 4))];

             % Sorting 'labs' into decreasing order
```

```
            [v2,in2] = sort(-labs);
            v21 = v2(1); %  documents in largest class
            v22 = v2(2); %  documents in second-largest class
            k = k + 1; % if there is a tie,  increment k
    end
    nodelabels(i) = in2(1);  % most probable labeling by kNN
end
```

The vector `nodelabels` now contains the classifications of the nodes. Its plot in Fig. 30 looks smoother than that in Fig. 29, but the Bayesian method may define deviating classes of nodes within larger homogeneous areas more reliably.

Fig. 30. Calibration of the SOM by the kNN rule, $k = 10$. Red: Corporate-Industrial. Green: Economics and Economic indicators. Blue: Government and Social. Yellow: Securities and Commodities Trading and Markets.

23 Approximation of an input data item by a linear mixture of models

Text analysis by least-squares fitting of documents using non-negative coefficients

Objective:

An interesting problem in text analysis is whether a given text follows too closely any old text or texts. Then its is thinkable to expand the given text in terms of vocabularies of a number of known texts. This would otherwise be impractical, if it did not exist a modern method of finding such optimal linear mixtures of models, where the coefficients in the expansion are restricted to non-negative values. It turns out that this principle also automatically restricts the expansion to a very small number of possible reference texts to be taken into account.

The text classification example discussed in the above section demonstrated that it is possible to assign an unknown text into some class of known texts by classifying it on the basis of its use of words, i.e., weighted vocabulary.

Nonetheless, an even more intriguing application would be if one could point out *several sources of text, of which the unknown text is a combination.* The method discussed next does not only indicate that this is possible, but it also measures the relative contents of the foreign sources.

An analysis hitherto generally unknown is introduced in this chapter; cf. also [43], [44]. The purpose is to extend the use of the SOM by showing that instead of a single winner model, one can approximate the input data item more accurately by means of a set of *several models* that *together* define the input data item more accurately. It shall be emphasized that we do not mean k winners that are rank-ordered according to their matching. Neither do I suggest parallel winners, each defined over a local area of the SOM. Instead, the input data item is approximated by an *optimized linear mixture of the models, using a nonlinear constraint.*

Consider the n-dimensional SOM models $\mathbf{m}_i, i = 1, 2, \ldots, p$, where p is the number of nodes in the SOM. Their general linear mixture is written as

$$k_1\mathbf{m}_1 + k_2\mathbf{m}_2 + \ldots + k_p\mathbf{m}_p = \mathbf{M}\,\mathbf{k} \ , \tag{7}$$

where the k_i are scalar-valued weighting coefficients, $\mathbf{k}$ is the p-dimensional column vector formed of them, and $\mathbf{M}$ is the matrix with the $\mathbf{m}_i$ as its columns. Now $\mathbf{M}\,\mathbf{k}$ shall be the *estimate* of some input vector $\mathbf{x}$. The vectorial fitting error is then

$$\mathbf{e} = \mathbf{M}\,\mathbf{k} - \mathbf{x} \ . \tag{8}$$

Our aim is to minimize the norm of $\mathbf{e}$ in the sense of least squares. *However, the special nonlinear constraint must then be taken into account in this optimization.*

Much attention has recently been paid to least-squares problems where the fitting coefficients are constrained to *non-negative values*. Such a constraint is natural, when the *negatives* of the items have no meaning, for instance, when the input item consists of statistical indicators that can have only non-negative values, or is a weighted word histogram of a document. In these cases at least, the constraint contains additional information that is expected to make the fits more meaningful.

23.1 The lsqnonneg function

The present fitting problem belongs to the broader category of *quadratic programming* or *quadratic optimization*, for which numerous methods have been developed in recent years. A much-applied one-pass algorithm is based on the *Kuhn-Tucker theorem*, as explained in [51], but it is too involved to be reviewed here in full. Let it suffice to mention that it has been implemented in MATLAB as the function named the *lsqnonneg*. Below, the variables $\mathbf{k}$, $\mathbf{M}$, and $\mathbf{x}$ must be understood as being defined in the MATLAB format. Then we obtain the weight vector $\mathbf{k}$ as

$$\mathbf{k} = \text{lsqnonneg}(\mathbf{M}, \mathbf{x}) \ . \tag{9}$$

The *lsqnonneg* function can be computed, and the result will be meaningful, for an *arbitrary rank* of the matrix $\mathbf{M}$. Nonetheless it has to be admitted that there exists a rare theoretical case where the optimal solution is not *unique*. This case occurs, if some of the $\mathbf{m}_i$ *in the final optimal mixture* are *linearly dependent*. In practice, if the input data items to the SOM are stochastic, the probability for the optimal solution being not unique is negligible. At any rate, the *locations* of the nonzero weights are unique even in this case!

23.2 Description of a document by a linear mixture of SOM models

The following analysis applies to most of the SOM applications. Here it is exemplified using the Reuters data.

In text analysis, one intriguing task is to find out whether a text comes from different sources, whereupon its word histogram is expected to be a linear mixture of other known histograms.

The example that demonstrates the fitting of a linear mixture of models to a given document is based on the lsqnonneg function. The text corpus was derived from the Reuters data as described earlier.

The piece of script that is needed to find and display the linear-mixture coefficients is very short, indeed. Let `X(art,:)` represent the unknown article, and let `M` be the SOM matrix. Then the linear-mixture coefficients `K` are obtained by the MATLAB `lsqnonneg(M', X1)` function directly; they are displayed as gray-shade dots onto the SOM array into due places.

```
X1 = X(art,:)';
K = lsqnonneg(M', X1);
figure;
som_cplane(sm, K);
```

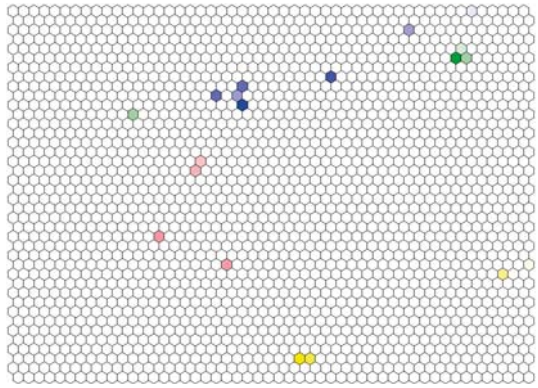


Fig. 31. A linear mixture of SOM models fitted to a new, unknown document. The weighting coefficients k_i in the mixture are shown by intensities of shade in the dots. (From [43].)

Fig. 31 shows a typical example, where a linear mixture of SOM models was fitted to a new, unknown document. The values of the weighting coefficients k_i in the mixture are shown by dots with intensities of shade that correspond to the weights of the corresponding SOM model vectors. The weights are displayed in the locations of the model vectors.

It is to be emphasized that this fitting procedure also defines the *optimal number* of the nonzero coefficients. In the experiments with large document collections, this number was usually very small, less than a per cent of the number of models.

When the models fall in classes that are known a priori, the weight of a model in the linear mixture also indicates the *weight of the class label associated with that model*. Accordingly, by summing up the weights of the various types of class labels one then obtains the *class affiliation* of the input with the various classes.

Discussion. This principle has also been applied with success to the evaluation of grant applications submitted to the Academy of Finland. (No official decisions were based on it, though.) The purpose was not so much to find plagiarisms but to analyze to what areas of science or its subfields an application belongs, and to find the most proper reviewers. There seems to exist an important area of application here.

24 The SOM of mobile phone data

Using lsqnonneg components instead of winners to display the state of a system

Objective:

The `lsqnonneg` *principle might find applications also in the control of processes and in the monitoring of machineries. We have made a pilot study on the monitoring of mobile-phone cells, the performance of which (especially finding mixtures of faulty states) can be detected by this principle.*

The second concrete example of forming linear mixtures of models comes from the study of the *continuous performance of a cell in a mobile-telephone network*. The input vector to the SOM was defined by 22 variables that describe the *key performance indices (KPI)* such as signal qualities in inward and outward transmission, frequencies of breaks in operation relating to different kinds of faults, and loadings of the cell. We had data from 110 such cells available, and each of the records was an average of the respective measurement or evaluation over an hour.

The particular SOM constructed for this study consisted of 80 models with the dimensionality of 22. The operation of one cell over nine successive hours has been exemplified here.

One might find similar applications in industry and medicine, where continuous processes are monitored. In some of the applications that we have studied, e.g., continuous casting of steel, milling of steel stripes, continuous cooking of cellulosa, and operation of a power transformer and an anaesthesia machine, the various states of the machinery that occurred during an appreciable time of operation defined an SOM. The present state was classified by following the sequence of the "winner" during a period of interest.

However, we now want to show that a more accurate view of the state of the equipment can be obtained if instead of a single "winner" on the SOM, *a linear mixture of the responses, defined by the lsqnonneg function* is displayed. *I do not thereby mean that the "winner," together with a few of its strongest "runners-up" should be displayed. The lsqnonneg function defines all of the nonzero coefficients in the linear mixture, and the "winner" may not even exist among them!*

The first of the two pictures, Fig. 32 depicts a sequence of the *winners* over nine hours. The second one, Fig. 33 , is the corresponding sequence of the *lsqnonneg coefficients*, where the values of the coefficients are shown by the shade of gray. One can see that the "winners" are not always included among the coefficients, but if they are (subimages 3, 6, 7, and 8), the linear mixture normally consists of fewer components. In other words, the state is then more "pure."

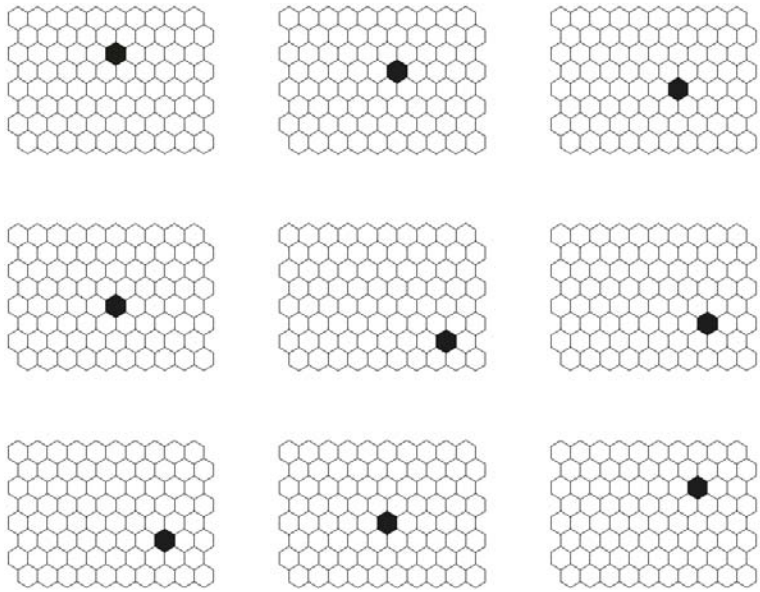


Fig. 32. A sequence of nine successive "winners" in the operation of a cell in a mobile telephone network over nine hours.

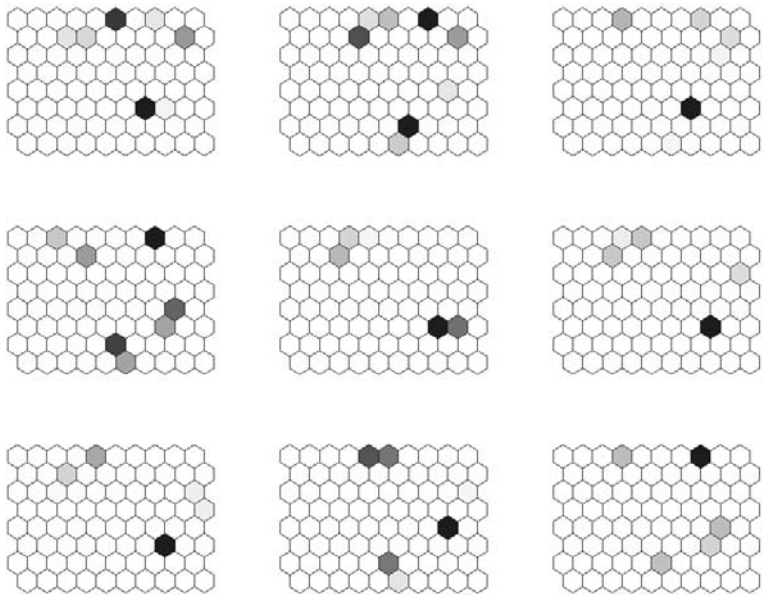


Fig. 33. The sequence of *all* nonzero lsqnonneg coefficients.

25 Categorization of words by their local contexts; toy example

Clustering of words on the basis of their local contexts

Objective:

In this and the next section we demonstrate that the "similarities" between input items can be defined in an indirect way, which may give rise to new data mining experiments. The "contextual SOMs"of words belong to psycholinguistic studies, in which the semantic meaning of words ensues from their occurrence in the contexts of their neighboring words in the text.

Description of the principle. A little known area of SOM applications is *psycholinguistics*. In the analysis of texts, the semantic role of a word transpires from its occurrence in the *local context* of other words. The local context around a particular word in the text, called the *target word*, can be defined in several ways. In early works it was made to consist of three successive words centered around the target word, or of two words of which the latter was the target word. *In this toy example we take for the context the previous and the next word to the target word, respectively.* E.g., consider a piece of text "it will rain here today ." If the word "rain" is selected for the target word, the local context is [will ... here]. If the target word is "here", the local context is [rain ... today]. The local context is a kind of *pattern vector*, and its words are the *pattern elements*. In statistical text analysis with the SOM, the local contexts around all of the consecutive words in the text corpus constitute the input data items.

Consider then the following pieces of text: "it will rain here today", "it will snow here today", "it will rain here tomorrow", and "it will snow here tomorrow". The words "rain" and "snow" occur twice in similar contexts, and are thus *contextually similar*. Obviously there exist very delicate and complex correlations between the various words in natural contexts.

One may try to *cluster* the local contexts according to their mutual contextual similarities. To that end, each of the contexts, as an input item, is identified by its target word.

A simple contextual SOM. A simple example [71] shows that the SOM can cluster target words of different word classes into separate areas on the SOM array, in an orderly fashion, on the basis of the local contexts in which they occur. In that work, three-word sentences of the type subject-predicate-object or subject-predicate-predicative were constructed *artificially*. In this "toy example" the vocabulary consisted only of 30 words, which were mapped onto the SOM as shown in Fig. 34. The words became *segregated* according to their linguistic roles in an orderly fashion: for instance, if you pay attention to the nouns, the proper nouns are in their own corner, then follow the names of animals, and after

that the materials. (The three curves separating the classes were drawn by hand.)

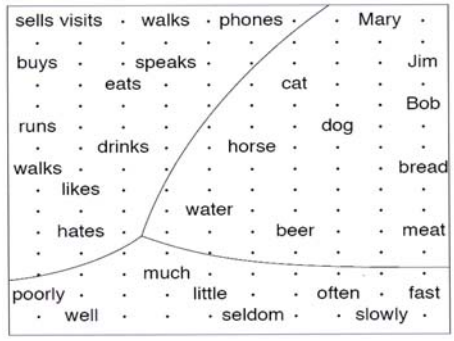


Fig. 34. A simple contextual SOM

Artificially generated clauses. For this demonstration, a small vocabulary, shown in Fig. 35(a), was defined. Its words are divided into *categories.* Each category is shown on a separate line. First there are three proper names of humans, and then three names of animals. These constitute all *subjects* in the artificially constructed sentences (clauses). The vocabulary continues with two names of liquids and two names of food, which are *objects.* Then follow 12 verbs, of which two categories constitute *transitive verbs.* These are followed by four categories of *intransitive verbs.* Finally there are listed four categories of adverbs.

The categories restrict the use of the words in such a way that words on the same row are *freely interchangeable as sentence constituents* in order to form *meaningful clauses.* This grouping brings about the contextual patterns.

All of the *logically possible sentence patterns* are listed in Fig. 35(b). In Fig. 35(a) we have the vocabulary, in which the numerals refer to the rows, or *word categories.* Examples of artificially generated, meaningful clauses are shown in Fig. 35(c).

Nonetheless the sentence patterns may give rise to unusual clauses, e.g., "Bob likes Bob," or "horse hates water." This example was only meant for the explanation of the principle, and its mending would mean a significant extension of the set of sentence patterns.

Since we want to reproduce this experiment, our first task is to transform these source data into a convenient MATLAB format.

Random word codes. In order that only the *word combinations* would make sense in the local contexts, the *writing* of the words must not effect the context pattern. In other words, for this experiment we must use *codes* for the words, which are as *independent of each other* as possible.

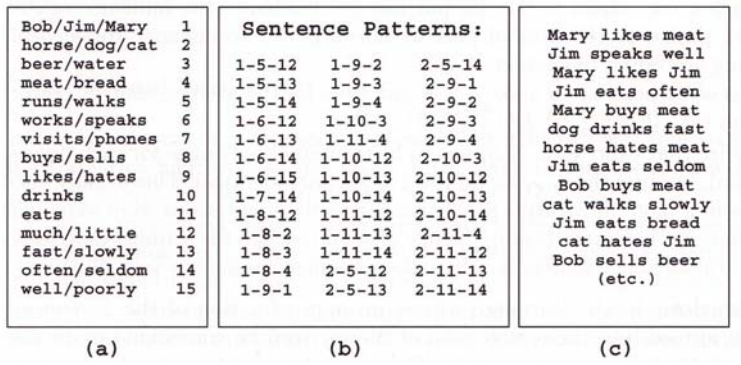


Fig. 35. (a) List of used words (nouns, verbs, and adverbs), (b)Legitimate sentence patterns, and (c) Some examples of generated three-word clauses. (From [39])

The most usual input patterns in SOM studies are *metric vectors*. Therefore we use *random vectors* for the word codes. If the dimensionality of the random vectors is high enough, the *correlation*, or the *dot product* of two random vectors, is negligible and guarantees the dissimilarity of the words. It is most effective to take random vectors which have a *normal distribution*. We use MATLAB vectors `randn`, which are (0,1)-normal random vectors.

The *random-code vocabulary* corresponding to Fig. 35(a) is shown below. In our preliminary studies it turned out that the dimensionality of 10 of the random vectors is quite sufficient for this experiment; with larger vocabularies the dimensionality must be higher, as will be shown in Section 24.

In the original vocabulary shown in Fig. 35(a) we had 15 rows, and we can imagine that there are three columns, some of which are sparse. When all words are reorganized as a single-column array, the random vocabulary shall have 45 rows, but we are using only those random vectors that have the same first indices `i` for which `Z(i) = 1`.

```
len = 100000;              % number of sentence patterns selected
dim = 10;                  % dimensionality of  vocabulary vectors
randn('seed',30000)
rand('seed',30000)
L1 = '   h   w b wwsvp  lhd  l sos p';
L2 = 'BJModcbamrraopihbsiaremiflfewo';
L3 = 'oiaroaeteeulresouektiautaotleo';
L4 = 'bmrsgteeaankkainyleentctswedlr';
L5 = '  ye  rrtdsssksteslsskshltlnoll';
L6 = '             sss s  s  e y m y';
```

```
% Vocabulary:

randword = randn(45,dim);  % random values of the vocabulary

Z = ones(45,1);
Z(9)  = 0; Z(12) = 0; Z(15) = 0; Z(18) = 0; Z(21) = 0;
Z(24) = 0; Z(27) = 0; Z(29) = 0; Z(30) = 0; Z(32) = 0;
Z(33) = 0; Z(36) = 0; Z(39) = 0; Z(42) = 0; Z(45) = 0;
```

Stream of clauses. A great number (100,000) of artificial clauses were generated by randomly selecting a sentence pattern from Fig. 35(b) and substituting randomly selected alternatives for words from the due categories in Fig. 35(a). After that, the words were replaced by the corresponding random-code vectors. *These three-member clauses are then concatenated into a simple stream of 10-dimensional random-code vectors, without any separating delimiters between the clauses*.

In our MATLAB demonstration we pick up the random word codes from the above vocabulary. First we have to form the sentence patterns.

Forming the clauses in MATLAB. The sentence patterns shown in Fig. 35(b) can be written as decimal numbers: e.g., 1-5-12 is written as `10512`.

```
% Sentence patterns:

   clause(1)  = 10512; clause(14) = 10902; clause(27) = 20514;
   clause(2)  = 10513; clause(15) = 10903; clause(28) = 20901;
   clause(3)  = 10514; clause(16) = 10904; clause(29) = 20902;
   clause(4)  = 10612; clause(17) = 11003; clause(30) = 20903;
   clause(5)  = 10613; clause(18) = 11104; clause(31) = 20904;
   clause(6)  = 10614; clause(19) = 11012; clause(32) = 21003;
   clause(7)  = 10615; clause(20) = 11013; clause(33) = 21012;
   clause(8)  = 10714; clause(21) = 11014; clause(34) = 21013;
   clause(9)  = 10812; clause(22) = 11112; clause(35) = 21014;
   clause(10) = 10802; clause(23) = 11113; clause(36) = 21104;
   clause(11) = 10803; clause(24) = 11114; clause(37) = 21112;
   clause(12) = 10804; clause(25) = 20512; clause(38) = 21113;
   clause(13) = 10901; clause(26) = 20513; clause(39) = 21114;
```

If we know the number `k` of the clause, *the numbers of the rows in the original vocabulary, Fig. 35(a), are obtained by first regarding these five digits as a decimal number.* The rows in the old vocabulary can then be computed as

row of the **first** *member in the clause:* `floor(clause(k)/10000) + 1 = row1;`
row of the **third** *member in the clause:* `mod(clause(k),100) = row3;`

row of the **second** *member in the clause:* `(clause(k) - 10000*row1 - row3)/100;`

For instance, for `clause(25)` we have `floor(20512/10000) + 1 = 2, mod(20512,100) = 12,` and `(20512 - 10000*2 - 12)/100 = 5`.

If we denote the *row* in the old vocabulary by `oldrow` and the *column* in the old vocabulary by `column`, we have *in the new one-column vocabulary* :

`newrow` of any member in the clause: `3*(oldrow-1) + column;`

One of the `columns` (1, 2, 3) is selected randomly, as we shall see.

Forming the random-code text words and the text stream. Next we start generating the artificial *text stream*, which consists of 10-dimensional random vectors `randword` obtained from the new vocabulary.

1. We select the number `argclause` of the sentence pattern randomly.

2. For each member of the pattern in turn, we determine the category `oldrow` (row in the old vocabulary).

3. As a function of `oldrow`, we define three successive rows in the new vocabulary as `newrow, newrow+1,` and `newrow+2`, respectively.

4. A `textword` is defined as a random-code vector from the new vocabulary, the choice depending on the number of alternatives in the category.

5. The random-code vector so obtained is appended to the text stream.

```
textword = zeros(3,dim);
tex = zeros(3*len,dim);
XL = zeros(30,dim);
XR = zeros(30,dim);
N = zeros(30,1);
X = zeros(30,2*dim);

% Construction of text stream

for S = 1:len
    argclause = floor(39*rand) + 1; % number of the clause
    for memb = 1:3
          if memb == 1
             oldrow = floor(clause(argclause)/10000);
             newrow =  3*(oldrow-1) + 1;
```

```
            V1 = oldrow;  % V1, V2 auxiliary variables
         end
         if memb == 2
            V2 = mod(clause(argclause),100);
            oldrow = (clause(argclause) - 10000*V1 - V2)/100;
            newrow = 3*(oldrow-1) + 1;
         end
         if memb == 3
            oldrow = V2;
            newrow = 3*(oldrow-1) + 1;
         end

         if Z(newrow+1) == 1 && Z(newrow+2) == 1
            if rand < 1/3
               textword = randword(newrow,:);
            else if rand < 2/3
                      textword = randword(newrow+1,:);
                 else
                      textword = randword(newrow+2,:);
                 end
            end
         end
         if Z(newrow+1) == 1 && Z(newrow+2) == 0
            if rand < .5
               textword = randword(newrow,:);
            else
               textword = randword(newrow+1,:);
            end
         end
         if Z(newrow+1,:) == 0 && Z(newrow+2) == 0
            textword = randword(newrow,:);
         end
         tex(3*(S-1) + memb,:) = textword;
    end
end
```

Forming the input patters to the SOM. The inputs to the SOM will now be picked up from the above `tex` vector in the following way. A *gliding window of three successive word positions* in the text is defined, and the three successive random vectors found in the window are *concatenated* into a single 21-dimensional input vector `X` to the SOM.

Note that when scanning the stream of word codes, only in one of three cases the obtained input vector `X` is positioned correctly with respect to the sentence pattern and contains a true clause, whereas in two cases out of three the code triple extends over the clause limits, and the "clause" does not make any sense. However, since the choice of successive clauses is random, the illegitimate word

triples picked up from adjacent clauses can be regarded as *pure noise*. So only every third word triple is *effective* in organizing the SOM. One can become convinced about that when carrying out the experiment.

Comment: Naturally one could have used separately generated clauses for SOM inputs, but the whole idea of this experiment was to show that the contextual SOM can make sense out of a continuous stream of words, where occasionally there are meaningful pieces of code triples.

Accelerated learning using averaged contexts as inputs. The whole experiment could have been performed as described above, but a faster learning, especially in saving computing time, is obtained using *average context vectors of words*. This will be very necessary in the large experiment to be reported in the next Sec. 27.

For speedup, the training data can be *clustered around the words of the vocabulary*. A particular word occurring in the text is called the *target word*. The produced text is scanned, and an average of the random-code vectors on both sides of a particular target word, averaged over the whole text corpus, is formed. In this way we obtain the *averaged contexts* for each unique word separately, and the number of averaged contexts, i.e., the number of training input data items, is the same as the number of words in the vocabulary (i.e., only 30 in this example).

Let us again start from the text vector of size `[99999 10]`. Let us define `X` to be the *averaged input vector* of size `[30 14]`. We scan the text 30 times, each time finding the indices of the various `X` in the text, and sum up the adjacent random vectors, one sum `XL` for the words on the left side of the target word, and one sum `XR` for the words on the right side of the target word, respectively. At the same time we count the occurrences `N` of the target words, and divide the sums by them.

```
% Inputs to the SOM
u = 0;
for v = 1:45              % word in vocabulary
    if Z(v) == 1
       u = u + 1;
       for w = 2:3*len - 1
           if tex(w,1) == randword(v,1)
              XL(u,:) = XL(u,:) + tex(w-1,:);
              XR(u,:) = XR(u,:) + tex(w+1,:);
              N(u) = N(u) + 1;
           end
       end
       XL(u,:) = XL(u,:)/N(u);
       XR(u,:) = XR(u,:)/N(u);
```

Now we concatenate `XL` and `XR` to form the 14-dimensional input vectors to the SOM:

```
        for j = 1:dim
            X(u,j) = XL(u,j);
        end
        for j = dim+1:2*dim
            X(u,j) = XR(u,j-dim);
        end
    end
end
```

Computation of the SOM. Since the training data `X` are now known, the SOM can be computed by the `som_lininit` and `som_batchtrain` functions.

```
% Computation of the SOM
msize = [9 12];
lattice = 'hexa';
smI = som_lininit(X, 'msize', msize, 'lattice', lattice, ...
'shape', 'sheet');
smC = som_batchtrain(smI,X,'radius',[4 1],'trainlen',100, ...
'neigh', 'gaussian');
sm = som_batchtrain(smC,X,'radius',[1 1],'trainlen',50, ...
'neigh', 'gaussian');
```

Plotting and labeling the SOM. In this very simple example, we input the averaged context vectors `X` again, this time not executing any training, and find the *node indices* `c` of the corresponding winners. From these the *horizontal and vertical rows*, `ch` and `cv` of `c`, are computed. With the aid of these coordinates it is possible to insert texts, namely, the explicit writings of the words, into the due locations `(cv,ch)` on the SOM plane, given as labels `L`(in the beginning of the script):

```
% Calibration
som_cplane('hexa',msize,'none');
M = sm.codebook;
norms2 = sum(M.*M,2);
for u = 1:30
    X1 =X(u,:)';
    Y = norms2 - 2*M*X1;
    [C,c] = min(Y);
    ch = mod(c-1,9) + 1;
    cv = floor((c-1)/9) + 1;
    if mod(ch,2) == 1
        shift1 = -.45;
    else
        shift1 = .1;
    end
    if u == 1 || u == 4
        shift2 = -.3
```

```
    else if u == 3 || u == 6
            shift2 = +.3
        else if u == 7 || u == 9 || u == 11 || u == 13 ...
                    || u == 15 || u == 17 || u == 19 ...
                    || u == 25 || u == 27 || u == 29
            shift2 = -.2;
            else if u == 8 || u == 10 || u == 12 || u == 14 ...
                    || u == 16 || u == 18 || u == 20 ...
                    || u == 26 || u == 28 || u == 30
                    shift2 = .2;
                else
                    shift2 = 0;
                end
            end
        end
    end
    string = [L1(u) L2(u) L3(u) L4(u) L5(u) L6(u)];
    text(cv+shift1,ch+shift2,string,'FontSize',8);
end
```

The computed SOM, corresponding to Fig. 34, is shown below as Fig. 36. Some minor differences are there, but one should pay attention to the almost perfect *clustering*: the pairs of words with opposite meaning or being closely related to each other are mapped into the same or adjacent nodes.

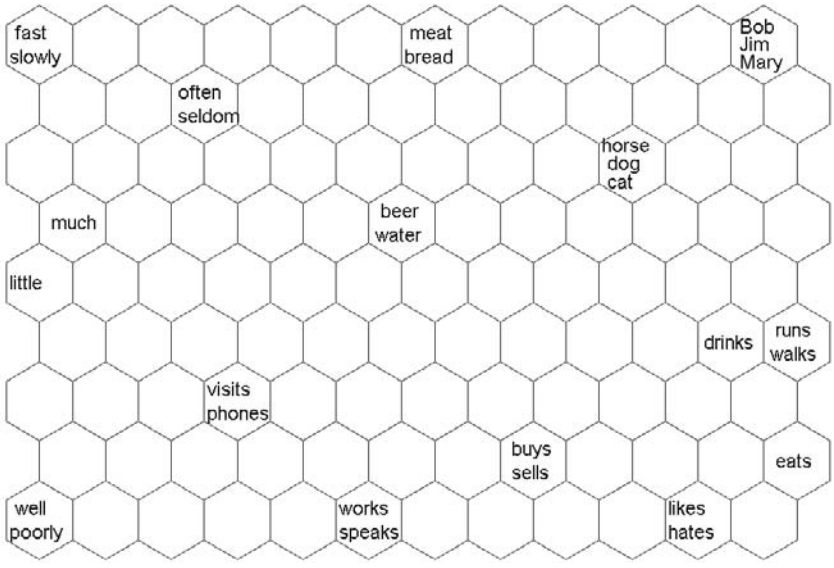


Fig. 36. A simple contextual SOM, recomputed in MATLAB.

26 Contextual maps of Chinese words

This is a big experiment in which word classes, based on contextual features, were plotted as histograms on the SOM

Objective:

In this section we describe a scientific project that uses a very large corpus of linguistically parsed text. It follows the idea explained in the previous section, but instead of displaying individual words on the SOM, it plots the histograms of various word classes on the SOM groundwork.

Inflexions of words. In most languages the words are *inflected*, and many linguistic forms are indicated by *endings*. This would be a problem in the unique encoding of words. One simple solution is to regard each inflected form as a unique word. A more effective method is to transform each word form into its *base form* or *stem*, and to assume that the contextual relations are not changed.

Notwithstanding there exist languages such as Chinese, in which the words are not inflected at all, and which are then ideal for context experiments: even deep semantic meanings ensue solely from the contexts of the words. The Chinese texts consist of *characters*, which are highly standardized pictographs, each one with semantic loading. They act like letters, but their number is many thousands. The *words*, on the other hand, consist of one, two, or even many more successive characters, but *there are usually no spaces between the words* in texts. For the segmentation of texts into words there exist nowadays many automatic text-processing methods, but the Chinese-speaking people are able to carry out the segmentation instinctively (cf Fig. 37).

Fig. 37. A Chinese greeting

The Modern Chinese Research Corpus. We were very lucky for having at our disposal a very large Chinese text corpus called the *MCRC*, or the *Modern*

Chinese Research Corpus [83], which contained 1,524,121 words in electronic form, collected from newspapers, novels, magazines, TV subtitles, folktales, and other material from modern Chinese media. Each word in this text corpus was classified into one of 114 classes, of which 89 were genuine linguistic classes, while the rest consisted of punctuation marks and nonlinguistic symbols. Each character was represented by a standard Unicode consisting of one to five decimal digits.

In this experiment, in order to take more contextual information into account, the local contexts were made to consist of *five* successive words.

Averaged contexts of words. In principle, one might expect that all of the successive local-context vectors of the text corpus should have been used as input data in training the SOM. However, since the experiment was carried out using MATLAB scripts in which the SOM Toolbox extension was included, there was simply no memory reserved for that size of input data. A reasonable approximation is to form the *averaged contexts* for all unique words that occurred in the text: their number was 48,191. The averages can be formed by scanning the text corpus and forming the averages over 1,524,117 words (notice that the target words for the full five-word contexts must start at the third word and end two words before the end of the corpus). So the input data vectors are of the form $\mathbf{x}(w) = \mathrm{avg}_{i(w)}[\mathbf{r}_{i-2}, \mathbf{r}_{i-1}, \mathbf{r}_i, \mathbf{r}_{i+1}, \mathbf{r}_{i+2}]$. Here i is the word position in the text, w is the word at it, the $\mathbf{r}$ are the random-vector representations of the successive words, and avg_i is the operator averaging over i.

Exclusion of nonlinguistic symbols. As a matter of fact, since our aim was to carry out a linguistic context analysis, we further decided to ignore all such local contexts which involved at least one nonlinguistic symbol. This happened especially at the beginning and at the end of sentences where full stops could not be taken to the contexts. In all, after such "purification" there were still 27,090 linguistically pure local contexts left, and this amount of input data was considered as sufficient for forming a 2000-node SOM array.

After the SOM had been computed, we constructed the histograms of responses to various sets of test words.

26.1 Preparation of the input data

The amount of preprocessing in the contextual SOMs, compared with the other applications, is rather high. However, I shall describe this experiment in as many details as possible, hoping that this method would be applicable to text corpora of other languages as well.

The Unicode. The Chinese characters, like the special letter symbols of many other languages, are stored and transmitted in electronic form encoded by the Unicode. We decided to handle only words that contain up to *four characters*. The relative number of Chinese words that consist of more than four characters is less than 1/1000 of all words, at least in the MCRC corpus, so we decided

to ignore such words in this experiment. At any rate the error caused by this omission is negligible, on the order of magnitude of "noise," and was not expected to cause any visible changes in the SOMs produced.

Below we see an example of such decimal codes: each horizontal line corresponds to one word in the text. The T1 through T4 are the Unicodes for characters used in the text. The first word contains only one character T1; therefore T2 = T3 = T4 = 0. The second word consists of three characters, and so on. The fifth column tells the decimal codes of the word classes. For instance, 34 means general nouns and 82 verbs without objects. The label 103 is a nonlinguistic symbol, etc.

There are in total 1,524,121 entries in Table 5.

Table 5. An excerpt of the Unicode-encoded file of text words.

character T1	character T2	character T3	character T4	label
...	...	...	...	...
30340	0	0	0	63
20013	25104	33647	0	34
12290	0	0	0	103
22269	21153	38498	0	39
37319	21462	0	0	82
30340	0	0	0	63
...	...	...	...	...

The next task is the construction of a *lexicon*, where all *unique* words of the text are entered only once. A particular problem is that about 6 per cent of the words in the text have an ambiguous classification: although the word form is the same, they are assigned to either of two possible classes, usually a general or a more specific one. It was decided that each word is assigned to that class that occurs most often with it. The physical appearance of the lexicon is otherwise similar to Table 5, except that each unique word occurs in it only once. The order of the words in the lexicon can be arbitrary. The characters in the lexicon are denoted L1, L2, L3 and L4, respectively. There are 48,191 entries in the lexicon.

26.2 Computation of the input data files for the SOM

As mentioned earlier, for the SOM method in general, it is most effective to represent the input data as *metric vectors*. On the other hand, the representations of all unique words should be as *uncorrelated* as possible; it is the *combination of the word codes* in the local context that carries the contextual information. In other words, *the occurrence of the same word codes in different local contexts then defines the degree of similarity of different local contexts*. Thus, the basic idea in the computation of contextual SOMs is to assign a unique random vector

to every unique lexical word. This assignment is made only once, not every time when the word is used. The random vectors must have a high dimensionality, say, on the order of a couple hundred, and be preferably normally distributed with zero mean, in order to be mutually as *orthogonal* as possible.

Choosing variable-length random coding for words. In earlier works the random vectors always had the same dimensionality for every word independent of their position in the local context. In those works the length of the local context was two or three words. Later studies have shown that there might exist an optimal length for the local context, because the strongest contextual effect comes from the nearby words and gets weaker when the words are more distant from each other. It has also turned out that if the local-context vector is longer, say, being composed of five segments as mentioned before, it may be advisable to use a dimensionality of the random vectors that is a function of the position of the word in the longer context. In the present study we have made the following choice, which is solely based on experience:

> *For the middle word (denoted by the index w), the random vector had the dimensionality of 50. For the words with the indices $w-1$ and $w+1$, the dimensionality of their random vectors was 200, and for the words with the indices $w-2$ and $w+2$, the dimensionality of their random vectors was 100, respectively.*

```
% Global parameters
len_text = 1524121;   % length of text
len_lex = 48191;      % length of lexicon
NewLex = 27090;       % length of compressed lexicon
Dim = 50;             % unit of dimensionality

% Definition of the context segments
XL2 = zeros(len_lex,2*Dim);  % leftmost segment of context
XL1 = zeros(len_lex,4*Dim);  % left segment of context
XM0 = zeros(len_lex,Dim);    % middle segment of context
XR1 = zeros(len_lex,4*Dim);  % right segment of context
XR2 = zeros(len_lex,2*Dim)   % rightmost segment of context
```

Encoding the lexicon. First of all we must have a *lexicon* of all unique words n used in the text and for their class labels. Let the lexicon be represented by the matrix $L(n, j)$, were the elements $L(n, 1)$ through $L(n, 4)$ correspond to the decimal-valued Unicodes of the word $n = 1, 2, ..., 48191$, and $L(n, 5)$ is the *class label* of the word n, with the values $1, 2, ..., 114$. As mentioned earlier, the class labels 1 through 89 are labels of pure linguistic classes and the labels 90 through 114 are labels of punctuation marks and nonlinguistic symbols.

```
% Lexicon: Unicodes of lexical words
filename = 'lexicon';
load ([lexicon  '.dat']);
L = lexicon;  % labels in lexicon = L(:,5)
words = zeros(len_lex,1); % frequency of  target words in text
```

Construction of random context codes for the text words. Let us denote the text corpus by another matrix $T(w, j)$, where the elements $T(w, 1)$ through $T(w, 4)$ correspond to the decimal-valued Unicodes of the word $w = 1, 2, ..., 1524121$, and $T(w, 5)$ is the class label of word w, with the values $1, 2$, ..., 114.

We start the construction of the input vectors X to the SOM by scanning the text from left to right and determining the indices $n1$, $n2$, $n3$, $n4$, and $n5$ of the five successive words $w - 2$, $w - 1$, w, $w + 1$, and $w + 2$. In this scanning we ignore the words with labels 90 though 114, the word class 1, which is a special punctuation mark, and the class 23, which in this corpus was empty.

```
% Input data: Unicodes  of  text words and labels
filename = 'textfile';
load([filename '.dat']);
T = textfile;
labels = textfile(:,5);

% Computation of random segments

% Unique random codes for all lexical words: initial values
R2 = randn(len_lex,2*Dim);
R1 = randn(len_lex,4*Dim);
R0 = randn(len_lex,Dim);

for w = 3:len_text-2  % scanning the  text corpus, ends excluded

   % Exclusion of certain classes
  if labels(w-2)<90 && labels(w-1)<90 && labels(w)<90 ...
  && labels(w+1)<90 && labels(w+2)<90,

  if labels(w-2)~=1 && labels(w-1)~=1 && labels(w)~=1 ...
  && labels(w+1)~=1 && labels(w+2)~=1,

  if labels(w-2)~=23 && labels(w-1)~=23 && labels(w)~=23 ...
  && labels(w+1)~=23 && labels(w+2)~=23,
```

The following indices `n1` through `n5` are the indices of words in the vocabulary, with the aid of which the random words codes can be located.

```
 % location of context words (Unicodes) in lexicon
n1 = find(L(:,1)== T(w-2,1)&L(:,2)== T(w-2,2) ...
& L(:,3)== T(w-2,3)&L(:,4)== T(w-2,4));

n2 = find(L(:,1)== T(w-1,1)&L(:,2)== T(w-1,2) ...
& L(:,3)== T(w-1,3)&L(:,4)== T(w-1,4));

n3 = find(L(:,1)== T(w,  1)&L(:,2)== T(w,  2) ...
& L(:,3)== T(w,  3)&L(:,4)== T(w,  4));

n4 = find(L(:,1)== T(w+1,1)&L(:,2)== T(w+1,2) ...
& L(:,3)== T(w+1,3)&L(:,4)== T(w+1,4));

n5 = find(L(:,1)== T(w+2,1)&L(:,2)== T(w+2,2) ...
& L(:,3)== T(w+2,3)&L(:,4)== T(w+2,4));

words(n3) = words(n3) + 1; % no. of target words found
```

The next task is to form the averages of the random context vectors. It starts with the summing up of random codes of the words into the context segments, after which they are divided by the word frequencies. At the same time, if some lexical words do not occur in the text (due to our restriction to "pure" contexts, which do not contain punctuation marks or other nonlinguistic symbols), the new lexicon is reduced to the length `NewLex`, and saved for the construction of the SOM:

```
 % superposition of random vectors to segments of vector X
XL2(n3,:) = XL2(n3,:) + R2(n1,:);
XL1(n3,:) = XL1(n3,:) + R1(n2,:);
XM0(n3,:) = XM0(n3,:) + R0(n3,:);
XR1(n3,:) = XR1(n3,:) + R1(n4,:);
XR2(n3,:) = XR2(n3,:) + R2(n5,:);
end
end
end
end

% Averages of word codes
NewLex = 0;
for n = 1:len_lex
     if words(n) > 0                      % target words used
        for j = 1:2*Dim
            XL2(n,j) = XL2(n,j)/words(n);
            XR2(n,j) = XR2(n,j)/words(n);
        end
```

```
        for j = 1:Dim
        XM0(n,j) = XM0(n,j)/words(n);
        end

        for j = 1:4*Dim
            XL1(n,j) = XL1(n,j)/words(n);
            XR1(n,j) = XR1(n,j)/words(n);
        end

        NewLex = NewLex + 1;
    end
 end
```

The input vectors `X` to the SOM are now formed by *concatenation* of the computed segments into input vectors of higher dimensionality of 13:

```
% Concatenation of word segments to form X
X = zeros(NewLex,13*Dim);
lexlabels = zeros(NewLex,1);

v = 1;                                    % index of input item
  for w = 1:len_lex
       if words(w) > 0

% Leftmost T
       for d = 1:2*Dim
           X(v,d) = XL2(w,d);
       end

% Left T
       for d = 2*Dim+1:6*Dim
           X(v,d) = XL1(w,d-2*Dim);
       end

% Middle T
       for d = 6*Dim+1:7*Dim
           X(v,d) = XM0(w,d-6*Dim);
       end

% Right T
       for d = 7*Dim+1:11*Dim
           X(v,d) = XR1(w,d-7*Dim);
       end
```

```
% Rightmost T
      for d = 11*Dim+1:13*Dim
          X(v,d) = XR2(w,d-11*Dim);
      end

 % Labels
      L1(v) = L(w,5);
      v = v+1;
   end
end

filename = 'mcrc_inputs';
save([filename '.mat'], 'X');

filename = 'mcrc_labels';  % new labels with indices 1 ... NewDim
save([filename '.mat'], 'L1');
```

26.3 Computation of the SOM

The SOM is computed along similar lines as before. We want to have a reasonable resolution in the map, so the array size is chosen as 40 by 50. It is to be noted that since the neighborhood function is Gaussian, its radius need not be an integer; for better resolution the final radius could be .5 units, and the initial (coarse) radius could be, say,equal to 6. We start by 30 training cycles in both coarse and fine learning, which turns out to be sufficient.

The input data `X` were computed by the previous script.

```
msize = [40 50];
lattice = 'hexa';          % hexagonal lattice
neigh = 'gaussian';        % neighborhood fuction
radius_coarse = [6 .5];    % neighb. radius, coarse [initial final]
trainlen_coarse = 30;      % cycles in coarse training
radius_fine = [.5 .5];     % neighb. radius, fine [initial final]
trainlen_fine = 30;        % cycles in fine training

% Linear initialization
smI = som_lininit(X, 'msize', msize, 'lattice', lattice, 'shape', ...
'sheet');

% Coarse training
smC = som_batchtrain(smI, X, 'radius', radius_coarse, 'trainlen', ...
trainlen_coarse, 'neigh', neigh);

% Fine training
sm = som_batchtrain(smC, X, 'radius', radius_fine, 'trainlen', ...
trainlen_fine, 'neigh', neigh);
```

```
% Stopping rule
R = 1000000;
while R - norm(sm.codebook) > 0
R = norm(sm.codebook);
sm = som_batchtrain(sm, X, 'radius', radius_fine, 'trainlen', 1, ...
    'neigh', neigh);
R1 = norm(sm.codebook);
end

filename = 'mcrc_som';
save([filename '.mat'], 'sm');
```

Notice the last line above: it was advisable to save the SOM memory in the struct form `sm`, because the `som_cplane` instruction then becomes very simple, and the matrix form of the memory can always be recovered by the expression `M = sm.codebook`.

Forming word histograms. When the SOM matrix now has been computed and stored, it can simply be loaded, together with the computed labels, in further scripts to make any number of histograms, without having to repeat the above computations. Below is the script for making the histograms. One may write it as a `function` to give the number of the linguistic class and the title of the picture as its arguments (notice that "class" and title" may be reserved words):

```
function  histograms(Class, Title)

filename = 'mcrc_som';
load([filename '.mat'], 'sm');
M = sm.codebook;
filename = 'mcrc_somlabels';
load([filename '.mat'], 'L1');
M = sm.codebook;

hits = zeros(2000,1);          % hit diagram on the SOM
norms2 = sum(M.*M,2);

for u=1:NewDim
     if L1(u) == Class  %  decimal code of class to be displayed
        X1 = X(u,:)';
        Y = norms2 - 2*M*X1;
        c = min(Y);
        hits(c,1) = hits(c,1) + 1;
     end
end
```

```
% Display
colormapigray = ones(64,3) - colormap('gray');
colormap(colormapigray);
figure(1);
som_cplane(sm, hits(:,1));   % simplest declaration of SOM plane
set(gca,'FontSize',10);
title(Title);    % give the title of class to be written
```

To this script we may add the instructions for making multiple diagrams with texts, for saving space.

26.4 The histograms of four main word classes

At this point we may already be interested in seeing some results of computation. We start with showing the histograms of all adjectives, all adverbs, all nouns, and all verbs, respectively. All adjectives belong to the classes 2 through 8, all adverbs to the classes 10 through 12, all nouns to the classes 33 through 42, and all verbs to the classes 72 through 89, respectively.

Fig. 38 combines the above four diagrams. The modification of the script "Display" to make a combination of four maps has not been shown explicitly. We can see that the overlapping of these four histograms is very small.

The MATLAB graphics has a default setting of normalizing the shades of gray: the maximum value in the histogram is always painted in black. Therefore the absolute values cannot be compared directly.across the different subplots. Of course there exists an option for changing the gray scales individually, but a drawback would be that some histograms would then not be visible at all.

The noun histogram is widest of all the four. It extends over the whole SOM, which shows that the nouns occur in many kinds of contexts. The verbs are contextually more specific, an one can discern some very tight clusters of verbs, as the case also is with the adverbs. We will see that there exist many specific subclasses of verbs.

We can see that to a reasonable extent the four main word classes are segregated on the SOM and have a small overlap. In particular one may pay attention to the areas where the verbs have a cluster; at least the corresponding areas in the noun display are white.

26.5 Subsets of nouns, verbs, and pronouns

Next we show that the main classes are further divided into finer subclasses. In the following two combination pictures, four specific noun classes, and four specific verb classes are depicted. The noun classes are: "Names of people" (given names), "Surnames," "Names of places,", and "Names of organizations." They are shown in Fig. 39.

The areas where the specific nouns are situated are more sparse than the areas of general nouns, but the clusters are also very narrow. They are well segregated

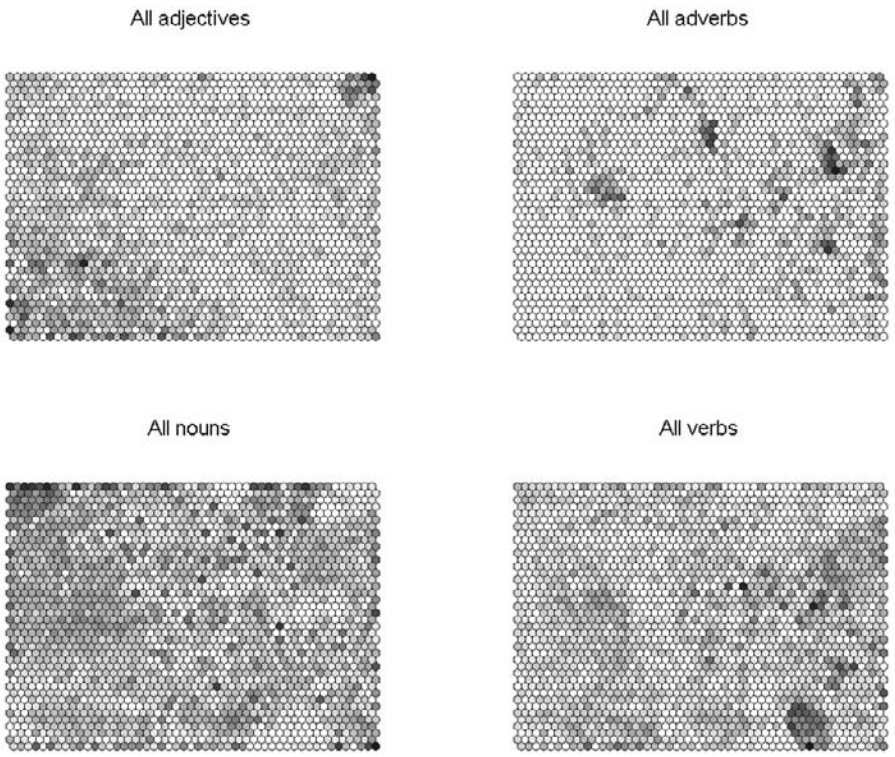


Fig. 38. The hit histograms of all adjectives, all adverbs, all nouns, and all verbs, respectively. The shades of gray cannot be directly compared across the pictures, because the MATLAB graphics automatically draws the maximum value in black, and all the other shades are only relative to that. Nonetheless it is obvious that the nouns have the widest distribution. The four distributions do not overlap significantly, which shows that the words of the four main word classes normally occur in different kinds of contexts.

from each other. The "names of people" in Chinese mean given names, which at least occur in different kinds of contexts compared with those of the surnames.

In addition to these, there are still several classes for specific nouns, and idioms that act like nouns. The number of words in these classes is so small that the histograms cannot be formed.

The specific verb classes shown here are: "Verbs followed by nouns" (in some languages they would be called transitive verbs), "Verbs without objects" (intransitive verbs), "Modal verbs," and "Linking verbs" (Fig. 40.) It should be noted that the MATLAB graphics *normalizes* the histogram scales so that the maximum is always painted in black; therefore it is not possible to directly compare the shades of gray between the different subplots.

In addition to these subclasses, there are still classes like "Verbs as objects," "Verbs as subjects," "Verbs as adverbs," "Verbs as modifying components in

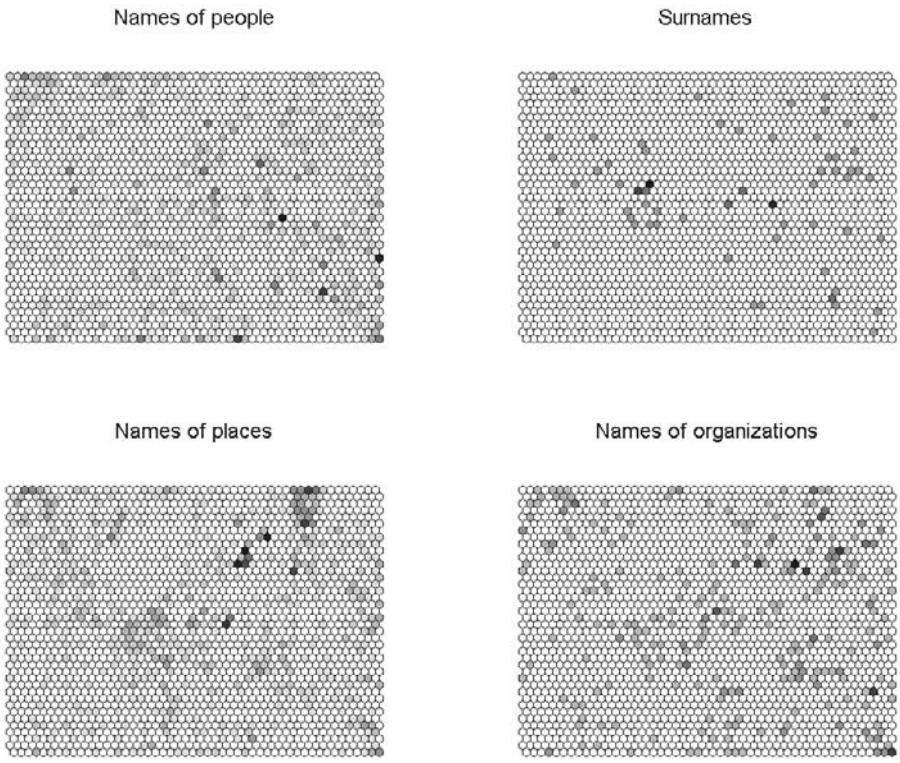


Fig. 39. The hit histograms of "Names of people," "Surnames," "Names of places," and "Names of organizations." These histograms are more sparse than the histograms of general word classes, which is understandable: there are a great many specific noun classes defined in the MCRC, but in each of them there is only a small subset of words that belong to the general noun class. At any rate, these classes are well segregated: one cannot find occupied areas in the four histograms that would coincide.

noun phrases," "Verbs as the core of noun phrases," and various idioms acting like verbs. Usually the number of words in these subclasses is so small that the histograms do not look realistic.

We still plot the subcategories of pronouns as parsed in this corpus: "Pronouns as adjectives," "Pronouns as adverbs," Pronouns as subjects or objects," and "Pronouns as attributes." . In two of the subclasses there were so few words that there is only one exemplar of each; this is indicated by the dots being completely black. The subclasses have been defined according to the role of a word as a sentence constituent, and not so much as a basic linguistic word class.

The four subclasses of pronouns are shown in Fig. 41.

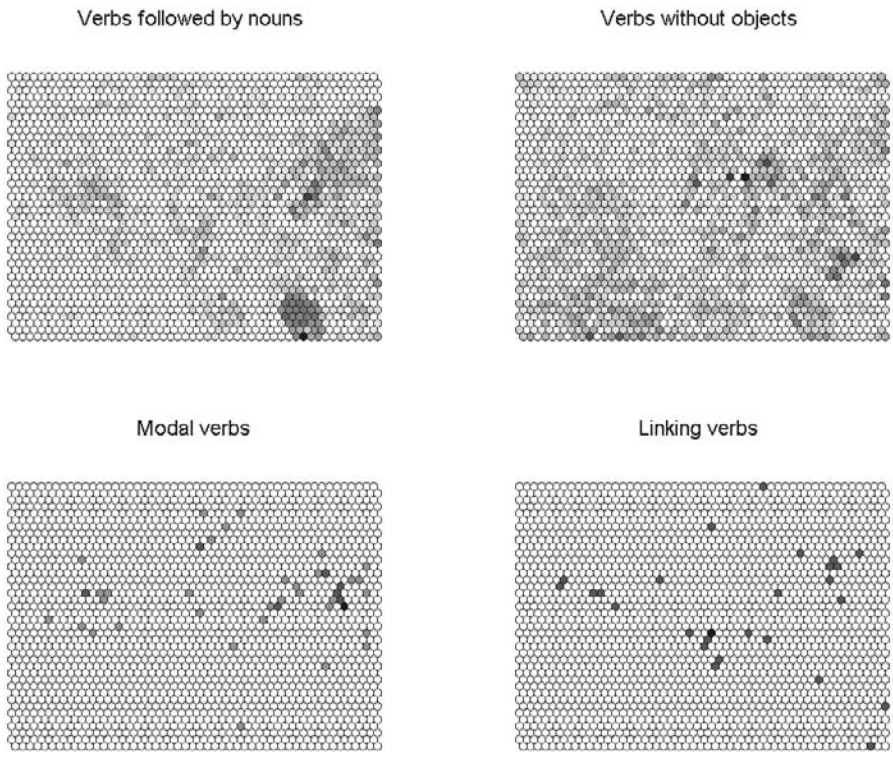


Fig. 40. The hit histograms of "Verbs followed by nouns," "Verbs without objects," "Modal verbs," and "Linking verbs.".

Discussion. One might ask what is the rationale behind showing this kind of complicated example. One reason, of course, is that it might stimulate and help analytical research in psycholinguistics. However, the study of the Chinese language was not in central role. If one had access to parsed text corpora of comparable size or bigger, this analysis could apply to many languages.

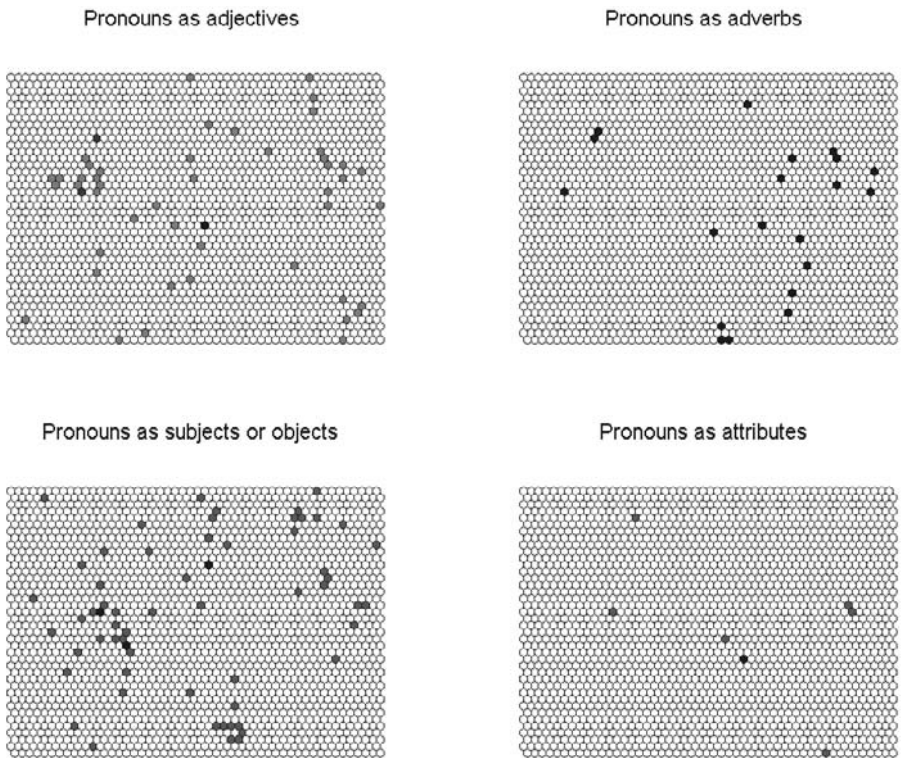


Fig. 41. The hit histograms of "Pronouns as adjectives," "Pronouns as adverbs," "Pronouns as subjects or objects," and "Pronouns as attributes."

27 Computation of the "Welfare map" of 1992 by the SOM Toolbox

Estimation of missing data ("Inputting")

Objective:

In the SOM Toolbox there is no provision for matching items on the basis of incompletely defined data, like in our SOM_PAK program package. In this section we try to mend this problem by estimating the missing values by referring to the nearest neighbors on the SOM array.

27.1 The problem of missing data (Incomplete data matrices)

It has turned out that the SOM is a rather robust algorithm. It tolerates slightly wrong parameter values and too few iteration cycles in training, and still produces almost correct-looking SOMs. It can also handle one frequently encountered problem in practice, namely, the problem caused by *missing data*. Especially in large statistical studies some of the elements of the input data matrix may not be available. For instance, in the "Welfare map" example mentioned at the beginning of this book, about 30 per cent of the indicators were missing from the statistics of some countries submitted to the Word Bank. For the 126 countries taken to the statistics, only 27 or more of the indicator values of the total of 39 were given to 77 countries. The rest of the countries had even less given indicators. Nonetheless, an SOM could be computed for these 77 countries by a special arrangement. This possibility was built in the programs of SOM_PAK in the following way:

The SOM matrix is dimensioned for the full input data matrix (for all possible components of the input vectors). However, if some of the components of the input vectors are missing, they are indicated by a special symbol in the data file, and the comparison with the model vectors of the SOM, when searching for the "winner", is only made on the basis of the given components*. Then, only those components of the model vector in the SOM that correspond to the given input data are updated. Naturally, some information is thereupon missing, but one does not know what, and this is the best way to proceed, if one wants to take also all of the incompletely given information into account. In the similar way, when* calibrating *the SOM for a country that has only incomplete data given, only the components of the SOM model corresponding to the given components of the input vector are taken into account in searching for the "winner" and labeling it on the SOM.*

The above method was possible in the SOM_PAK whose scripts were written in the *C* language. The SOM Toolbox, on the other hand, is based on MATLAB, and incomplete vectors cannot be used in the vector- and matrix-valued variables, at least in the `som_linit` and `som_batchtrain` functions.

I am now suggesting another method for the SOM Toolbox to deal with the incomplete input data matrices. It makes use of the conventional `som_lininit` and `som_batchtrain` functions. *The missing elements of the input vectors are* estimated *on the basis of the corresponding components in the most similar vectors, where these elements are given.*

27.2 Handling of incomplete data matrices in SOM Toolbox

It may now be best to explain the principle by means of a MATLAB script. It is written completely in the component form; this is recommendable, since the script that prepares the input vectors can be executed in less than .02 seconds on a 2 GHz computer. The operations are better understandable in the component form.

The file `welfare2` contains the input variables X and the labels of the countries.

```
file = 'welfare2';
load ([file '.mat']) % X, labels
```

It may be necessary to tell already at this point that the elements 33 through 38 of the data vectors were missing for a number of countries, especially the poorest ones. In order that this would not cause an imbalance in comparison, we decided to neglect these components, so instead of 39-dimensional vectors we will have 33-dimensional ones. It will also turn out that the the handling of the incomplete data becomes significantly easier after this.

At first we make a binary "mask" over the `X` matrix such that the existing elements are indicated by the values 1, and the missing elements by 0, respectively. However, since we henceforth compute in MATLAB, also the missing components are given the value 0. This is possible, because the original data were normalized by having the same variance in every component scale, and since the values are then real-valued and nonzero, they are not confused with the "dont care" value 0.

```
for i = 1:77
    for j = 1:33
        if X(i,j) == 0
           P(i,j) = 0;
        else
           P(i,j) = 1;
        end
    end
end
```

Next, for every input vector indexed by `i0`, we search for a subset of other input vectors that are most similar to `X(i0,:)` on the basis of their *nonzero elements*. The number of these neighboring vectors is indicated by the parameter `Dist` (in this example, `Dist = 4`). We do this searching for all of the 77 input vectors. The parameter `Ref` is a gliding reference value that is large in the beginning of the search, but it is updated after every magnitude comparison operation. The vector `c` is the index vector of all "winners" that identifies the corresponding input vectors relative to `i0`. There are in this example `t = 4` of them.

```
for i0 = 1:77
    Ref = 10000;
    Dist = 4;
    c = zeros (Dist,1);
```

Now we start looking for neighbors for each vector indexed by `i0`. We could save time by doing that only for incomplete vectors, but winning less than a millisecond is not important: so we do this for all pairs of elements of the input matrix. However, an element need not be compared with itself. The variable `d` is another gliding reference value. It accumulates the value of the distance, but for incomplete vectors, only those elements that are nonzero in both `X(i0,:)` and `X(i,:)` must be taken into account. It means that the squares of differences between the vectors must only be accumulated for nonzero elements: hence the conditions set up by the masks `P`. But as a consequence, the number `N` of squares of differences must be counted, and the sum of squares must be divided by `N`.

```
    for i = 1:77
        d = 0;
        N = 0;
        if i ~= i0
           for j = 1:33
               if P(i0) == 1 && P(i) == 1
                  d = d + (x(i,j) - x(i0,j))^2;
                  N = N+1;
               end
           end
           d = d/N;
```

We have to prepare a *sorted list* of the nearest vectors, identified by their indices `c(1)` through `c(Dist)` in the descending order. It could be made and updated by the MATLAB `sort` instruction. In this example we continue the programming by the components. The following piece of script also does the sorting, and the two extra `end` lines terminate the script written so far.

```
            if d < Ref
               Ref = d;
               for k = 2:Dist
                   c(Dist-k+2) = c(Dist-k+1);
               end
               c(1) = i;
            end
        end
    end
```

Next we form the *means* of those elements in the neighboring vectors that are supposed to replace the missing elements. As a matter of fact, we were able to patch all of the missing data by this method. So the script is after that ready to *produce the preprocessed inputs that can be used as regular inputs* to the `lininit` and `batchtrain` functions on a normal SOM.

```
    for j = 1:33
        if P(i0,j) == 0
            Mean = 0;
            N = 0;
           for k = 1:Dist
               if c(k) ~= 0 && x(c(k),j) ~= 0
                  Mean = Mean + x(c(k),j);
                  N = N + 1;
               end
           end
           Mean = Mean/N;
           x(i0,j) = Mean;     % replacement of the missing element
        end
    end
end

file = 'welfaremap_inputs';
save([file '.mat'], 'x')
```

27.3 Making the SOM

The estimated input data, stored in the file `'welfaremap_inputs'`, can now be used *as such* in the conventional way in making the SOM. That SOM can also be calibrated by finding the winners to the inputs `x`:

```
file = 'a00pov_inputs_x1';
load ([file '.mat']) % x, names
```

```
U = x1;
for i = 1:33
    mi = min(U(:,i));
    ma = max(U(:,i));
    U(:,i) = (U(:,i)-mi)/(ma - mi);
end
H = 17;
V = 17;
msize = [V H];
lattice = 'hexa';      % hexagonal lattice
neigh = 'gaussian';    % neighborhood fuction
radius_coarse = [7 1]; % neighborhood radius, coarse [initial final]
trainlen_coarse = 50;  % cycles in coarse training
radius_fine = [1 1];   % neighborhood radius, fine [initial final]
trainlen_fine = 30;    % cycles in fine training
smI = som_lininit(U, 'msize', msize, 'lattice', lattice, 'shape', 'sheet');
smC = som_batchtrain(smI, U, 'radius', radius_coarse, 'trainlen', trainlen_coarse
sm = som_batchtrain(smC, U, 'radius', radius_fine, 'trainlen', trainlen_fine, 'ne
M = sm.codebook;
som_cplane('hexa',msize, 'none')
norms = sum(M.*M,2);
for u = 1:77
    U1 = U(u,:)';
    Y = norms - 2*M*U1;
    [C,c] = min(Y);
    ch = mod(c-1,H) + 1;
    cv = floor((c-1)/H) + 1;
    if u==3 || u==4 || u==9 || u==10 || u==37 || ...
       u==39 || u==49 || u==71 || u==73  || u==74
       shift2 = -.2;
    else if u==1 || u==5 || u==6 || u==8 || u==20 || u==21 ...
                || u==35 || u==69 || u==70 || u==72
        shift2 = .2;
        else
        shift2 = 0;
        end
    end
    if mod(cv,2)==1
        shift1 = -.22;
    else
        shift1 = .22;
    end
   text(ch+shift1-.15,cv+shift2,[names(u,1) names(u,2) ...
  names(u,3)], 'FontSize',8);
end
```

```
filename = 'Welfaremap';
print('-dpng', [filename '.png']);
```

In Fig. 42 we see the new "Welfare map.". Remember that this map describes the situation in 1992, which is quite different from the present one.

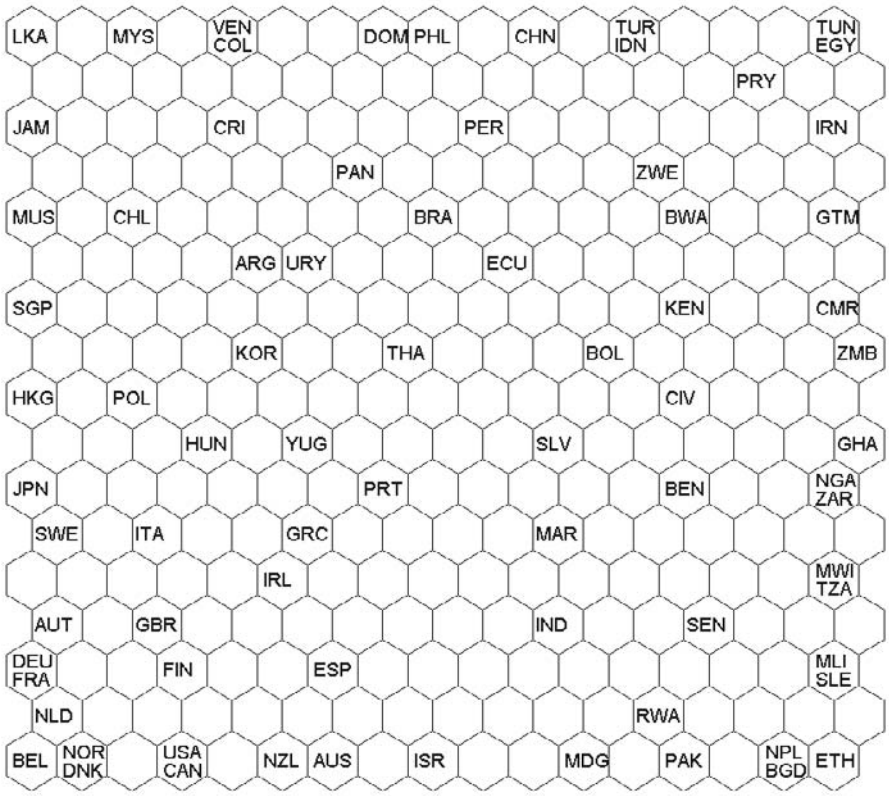


Fig. 42. Location of the countries on the new "Welfare SOM." For a legend of the abbreviations, see Table 1 on page 4.

The U matrix (without labeling) of the "Welfare map" is shown in Fig. 43. Notice the big ravine between certain poor countries on the right.

Discussion. One must now make a few comments when comparing the new map with Fig. 1. First of all *the new map is rotated left by 90 degrees, but this is natural, because the SOM can be materialized in any rotated form, without losing its topographic relations.* The form of the present map is a square, while that of Fig. 1 is rectangular. The square form (`msize[17 17]`) was necessary, because in the old format (`msize[13 9]`), the resolution of the texts on the `som_cplane` function would have been too poor.The detailed locations of the

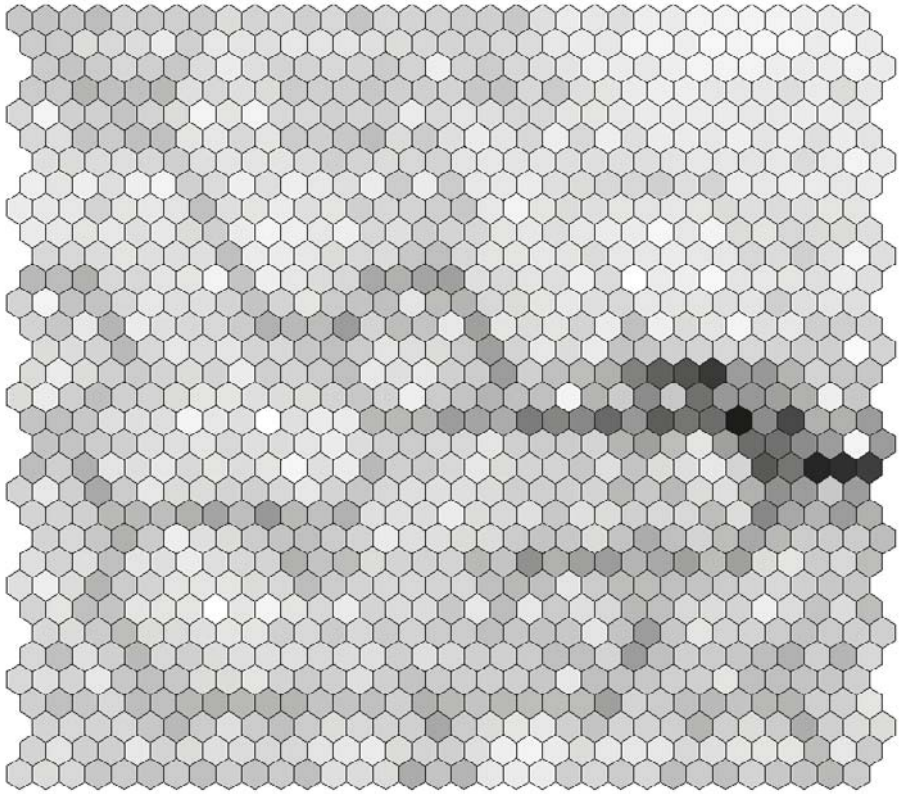


Fig. 43. The U matrix of the new "Welfare SOM." Notice the big ravine between certain poor countries on the right.

countries in the present map also slightly differ from those in Fig. 1. That is neither alarming, since the input data were not quite the same. The original components 33 through 38 were completely ignored in this example, since there were many missing data in them for some countries, and the "inputting" might not have resolved them. Remember that some missing components were only *estimated* in the present map. Nonetheless, there is much similarity in the maps. For instance, the locations of the richest (OECD) countries are concentrated on the same cluster in both maps (here in the lower left-hand corner; in the old map they were in the upper left-hand corner), and this is understandable, because the most significant *financial indicators* like the gross national product per capita, which are very dominant, were identical in both maps. Among the missing indicators there were some medical and educational data, which would have made a difference especially between the poorest countries.

28 SOMs of symbol strings

Non-vectorial items can be mapped onto the SOM, too

Objective:

Construction of a SOM for strings of symbols is very tricky, because one cannot use vector-matrix-operations for the computation of similarities, and training is not based on continuous-valued corrections of the variables. Therefore we cannot use the SOM Toolbox for this task. On the other hand, the SOM of symbol strings belongs to a larger category of SOMs of items, in which the geometric distances on the SOM are derived from the distance matrix *of the items.*

28.1 Special problems with strings of symbols

As stated above, the SOM of symbol strings belongs to a larger category in which the SOMs are supposed to reflect *similarities derived from the distance matrices* of the input items. One of the most demanding cases is the *SOM of human endogenous retroviruses* [65], which is based on precomputed distance matrices of DNA sequences. Related SOMs have been constructed in bioinformatics e.g. for sequences of macromolecules in proteins.

The corpora of data in bioinformatics are huge. The scripts for the computation of such SOMs also tend to be long and complex and are not suitable for tutorial examples like those presented here. It is not possible to use SOM Toolbox functions in them, and they have not even been written in MATLAB.

Nonetheless I wanted to include a reasonably simple example of a symbol-string SOM in this book. The set of input items that I selected was extremely trivial, namely, *a small set of given names of SOM researchers (!)*. With the aid of these data I at least hoped to be able to describe some basic concepts about the strings, such as the *distance measure*, which is often expressed in the *Levenshtein metric*, and computation of *averages* as so-called *medians of strings*. These operations are necessary when we modify the *Batch Map algorithm* for non-vectorial data.

On the other hand, one important problem is a proper *initialization of the SOM*. In principle it is possible to initialize the SOM by randomly chosen strings, but significant speedups can be gained if the initial values somehow comply with the statistics of the input items, and further, if they are provisionally *ordered*. One special operation that we need in initialization is *interpolation between strings*.

The worst problem is a tie. However, I must warn you beforehand that even though the data are simple, the self-organizing process is not. Due to the shortness of the strings and their small number, all kinds of problems will appeared in training. The worst of them is a *tie* in matching and selection.

Especially when the strings of symbols are short, as the names usually are, the set of possible values for the distance between two strings is very small. For

instance, if the (unweighted) Levensthtein distances discussed next are used, the possible distances between strings of symbols of length L are $0, 1, 2, ..., L$. Consequences of this are of following types: 1. In matching an input with the models, there will usually be *multiple winners* (i.e., a tie between them), and we have to find a good strategy to *break* the tie, or select the winner that guarantees the best ordering. 2. In the basic Batch Map, updating meant that an old model had to be replaced by the *mean* of input data mapped into the neighborhood of the winner. Now it has been shown that we can modify the Batch Map for strings where we use the *median* of a set of strings in place of the mean, but again there can occur ties in the definition of the median.

In large problems like the SOMs of bioinformatics data, due to the long strings of symbols thereby used, the ties occur more seldom. However, in this example which was supposed to be very simple, you will actually encounter these problems often, and therefore I had to invent several tricks to alleviate them. But do not let these tricks frighten you; at least we shall solve the whole task!

But first we need a *similarity measure*, or at least some *distance measure* for the comparison of strings, and it is discussed in the next subsection.

28.2 The Levenshtein metric for symbol strings

The lengths of the strings to be compared may vary in wide limits, but it is possible to define a *distance* between them. This problem was first discussed in the theory of communications. The statistically most accurate measure of distances between strings of symbols is the *Levenshtein distance* [52] (see also [39], pp. 22-23], which, in its *unweighted* form, for strings A and B is defined as

$$LD(A, B) = \min\{a(i) + b(i) + c(i)\}.$$

Here string B is obtained from string A by $a(i)$ replacements, $b(i)$ insertions, and $c(i)$ deletions of a symbol. There exists an indefinite number of combinations of $a(i)$, $b(i)$ and $c(i)$ to do this, and the minimum is sought, e.g., by the following *dynamic programming* method, that is shown below as a piece of MATLAB script.

Actually, since there may occur various types of stochastic *errors* in strings, and the probabilities for these errors depend on the occurring symbols, too, a statistically evaluated measure of distance is more accurate if the various types of editing operations (replacement, insertion, and deletion) are provided with different *statistical weights* p, q and r, respectively. This then results in the definition of the *weighted Levenshtein distance (WLD)*

$$WLD(A, B) = \min\{pa(i) + qb(i) + rc(i)\} \quad ,$$

where the coefficients p, q and r for the respective types of error may be obtained, e.g., from the so-called *confusion matrix* of the alphabet, as the inverse probability for a particular error to occur.

In the following we use the unweighted Levenshtein distance for simplicity, and take

$p(A(i), B(j)) = 0$ if $A(i) = B(j)$,
$p(A(i), B(j)) = 1$ if $A(i) \neq B(j)$,
$q(B(j)) = 1$,
$r(A(i)) = 1$,

where $A(i)$ is the ith symbol of string A, and $B(j)$ is the jth symbol of string B, respectively.

```
% The script for the computation of LD(A,B)

function  levenshtein(A,B)

m = zeros(3);
LA = length(A);
LB = length(B);
LD = 0;
D = zeros(LA+1,LB+1);
for i = 2:LA+1,
    D(i,1) = D(i-1,1) + 1;
end
for j = 2:LB+1;
     D(1,j) = D(1,j-1) + 1;
end
for i = 2:LA+1,
    for j = 2:LB+1,
        if A(i-1) == B(j-1),
            r = 0;
        else
            r = 1;
        end
        m1 = D(i-1,j-1) + r;
        m2 = D(i,j-1) + 1;
        m3 = D(i-1,j) + 1;
        m = [m1 m2 m3];
        D(i,j) = min(m);
    end
end
LD = D(LA+1,LB+1);
```

For intance, LD('erhardt','leonardo') = 4, because 'leonardo' can be transformed into 'erhardt' by deletion of 'l', replacement of 'on' by 'rh', and replacement of 'o' by 't' .

This script, the function `levenshtein(A,B)`, now comes in handy, because it can be called many times at the various stages of computations.

28.3 The median of a set of symbol strings

In the introduction of learning to a string SOM we first of all encounter the problem of how to *average* sets of signal strings. Even if the strings are regarded as vectors, like one assumes in MATLAB, their "dimensionalities" change often. However, in elementary arithmetics we already know how the *median*of numbers is defined: it is a member of the set of numbers that has the same number of elements which are smaller than and greater than it, respectively. (If there is an even number of elements, there are two elements that can be called a median.) We also know how the median is constructed arithmetically: *the median is that number which has the smallest sum of absolute values of differences with respect to each of the members of the set.*

Fron the above we can now generalize the definition of the median for any items, between which the *distance matrix* has been defined:

> *The median of a set of elements is that member in the set, which has the smallest sum of distances from all the other elements.*

Computation of the median of symbol strings. Next we need an algorithm for the computation of the *median of a set of symbol strings*. Let us call the symbol strings `w(i,:)`.

The purpose is to find that string `Median`, the sum of Levenshtein distances of which from all of the other strings is minimum. For computational reasons we must define the set of strings as an *indexed vertical array* of the strings. Let us recall that *only strings of the same length can be represented as a vertical array*, so when in our example the maximum length of a string is 9, the other strings must be filled in with a suitable number of blanks (`' '`) to define the length 9 for all of the strings. Below are the string data used:

```
w = ['takashi  '
     'leonardo '
     'andreas  '
     'argyris  '
     'fernando '
     'guilherme'
     'erhardt  '
     'michael  '
     'yoonsuck '
     'heeyoul  '
     'francesco'
     'alexander'
     'hiroshi  '
     'patrice  '
     'william  '
     'kouichi  '
     'geoffroy '
     'barbara  '
```

```
     'shigeomi '
     'roberto  '
     'leticia  '
     'rodrigo  '
     'nicolai  '
     'shinsuke '
     'toshiyuki'];
```

On the other hand, to execute the function `levenshtein(A,B)` below, the `A` and `B` must have no blanks, and therefore we need, e.g., the lines `A1 = w(a,:); A = A1(find(A1 =' '));` and `B = w(a,:); B = B1(find(A1 =' '));` below to remove the blanks. The Levenshtein distances are computed by calling the function `LD = levenshtein(A,B)`, after which the sums of distances from all of the names to all of the other names are formed, and the `sort` function puts the sums to an increasing order.

```
summa = zeros(25,1);
for a = 1:25,
    A1 = w(a,:);
    A = A1(find(A1~=' '));
    for b = 1:25,
        B1 = w(a,:);
        B = B1(find(A1~=' '));
        LD = levenshtein(A,B);
        summa(a) = summa(a) + LD;
    end
end
[MED,med] = sort(summa);
```

The vector `MED` contains all of the sums of distances to be compared. It is possible that among its elements, there exist multiple minima. All of these median words `w` will be found and listed as

```
w(med(find(MED==MED(1))),:)
```

These words still contain the blanks as they appear in the input data list. It turns out that in this case there is only one median among these 25 names, namely, `'hiroshi '`.

With a smaller set of strings, especially of equal lengths, e.g.,

```
w = ['takashi  '
     'andreas  '
     'argyris  '
     'erhardt  '
     'michael  '
     'heeyoul  '];
```

we may obtain a relatively large set of multiple medians: for this set all of the medians are

```
'takashi '
'andreas '
'michael '
'heeyoul '
```

This example shows clearly that since the intermediate results are often not unique, we might run into problems with ordering, especially with short strings such as the names. That is, *the optimal SOM will usually not be unique.*

It is to be noted that *the medians computed above are members of the set.* This may seem to restrict the averaging method, but actually it only means *quantization*, and the SOM using this method can still become topographically ordered and visually correct-looking.

In [35] I have shown that it is possible to construct a string, the sum of distances of which from the other strings is minimum, and *which does not belong to the set of given strings.* We shall skip it here.

28.4 The most distant strings

An opposite to the median of a set of strings is the string that is *the most distant from all of the other strings.* The algorithm for finding all of them is otherwise identical with the median algorithm, except that when sorting the `summa`, the *maxima* instead of the *minima* are determined:

```
[MDIST,mdist] = sort(summa,'descend');
w(mdist(find(MDIST==MDIST(25))),:)
```

Again it happened that only one string, namely, `'alexander'` was found to be the most distant string in the set of 25 names.

28.5 Interpolation between symbol strings

Another operation that is sometimes needed in the SOMs of strings, especially in their initialization, is *interpolation between two strings.* It comes in handy if we *enlarge* the SOM, for example if we first make a small SOM and then approximately double its horizontal and vertical dimensions. For the initial values of the new, interstitial, "blank" cells of the larger SOM we can take "averages" of the neighboring cells of the original SOM and continue training of the new SOM. Naturally we cannot use any arithmetic averaging, but the distances between strings are still describable, e.g., by *Levenshtein metrics.* We may say that *the average of two strings is a string that has a distance from both of the former, which is half of the distance of the original strings.* Because the strings are quantized entities, we can halve the distance only approximately.

The simplest way to interpolate between two strings is to proceed in the following way. Consider different strings `A` and `B`. If the symbols in the same

symbol position in A and B are different, and we make the symbol in one string identical with the symbol in the other, the similarity of the strings is increased, in other words, the resulting string has a distance from both of the original ones that is smaller than the distance between the original strings. We repeat this operation on randomly chosen different symbols until B has reached a distance from A that is about half of the distance between A and the original B.

Since we must always find symbols that have the same position in both strings, we must make changes in the shorter string only. Therefore we shall *rename* the strings so that the longer one becomes A and the shorter one B, respectively.

Let us denote the Levenshtein distance between A and the original B by LD, and the variable Levenshtein distance between A and the new B by LD1, respectively. Let x be a randomly drawn symbol position in B. The script for interpolation reads as follows:

```
function[LD1] =  interpolation(A,B)

LA = length(A);
LB = length(B);
if LB > LA
   C = A;
   A = B;
   B = C;
end
LD = levenshtein(A,B);
LD1 = LD;
while LD1 > ceil(LD/2)
      x = floor(length(B)*rand + 1);
      if A(x) ~= B(x)
         B(x) = A(x);
      end
      LD1 = levenshtein(A,B);
end
```

Notice that when we select the symbol position x randomly, we do not obtain every time different A and B, but we shall anyway continue the random selection until LD1 = ceil(LD/2). (It is also possible to use the function floor instead of ceil.)

If A = 'washington', B = 'lissabon', the original Levenshtein distance between them is 7. By the above algorithm we obtain (with a certain random sequence of x) for the interpolation 'wassanon', which has the Levenshtein distance from A equal to 4, and from the original B equal to 3, respectively. Naturally there may exist a great number of interpolations that fulfill the above condition, but any one of them will serve our purposes.

The Ordering Index (OI). First we define a very simple criterion of two-dimensional ordering, which is utilized in the initialization of the string SOM.

Assume that we have the same number of strings and SOM nodes, and our objective is to put the strings into the SOM array in such an order that *the sum of Levenshtein distances between all of the closest nodes in the horizontal and vertical directions is minimum*. This sum is now called the *ordering index (OI)*.

For the string SOMs we use here rectangular arrays only, because they are simple to program, and the visual anisotropy of the map is not important, when the data are non-vectorial. Moreover we can use simple tabular operations in the display of the SOM.

Assume that we have 25 names and a 4 by 5 SOM array. Let `m(i,j)` denote a model, which is a string. The script for the computation of the `OI` is straightforward:

```
function[OI] = orderindex(m)

OI = 0;
for i = 1:5
    for j = 1:4
        c = 5*(j-1) + i;
        c1 = 5*j + i;
        A1 = m(c,:);
        A = A1(find(A1~=' '));
        B1 = m(c1,:);
        B = B1(find(B1~=' '));
        LD = levenshtein(A,B);
        OI = OI + LD;
    end
end
for i = 1:4
    for j = 1:5
        c = 5*(j-1) + i;
        c1 = 5*(j-1) + i + 1;
        A1 = m(c,:);
        A = A1(find(A1~=' '));
        B1 = m(c1,:);
        B = B1(find(B1~=' '));
        LD = levenshtein(A,B);
        OI = OI + LD;
    end
end
```

28.6 Semi-manual initialization of SOMs for symbol strings

In the SOM for symbols, the models are symbols, too. The first problem is how one should *initialize* them. There do not exist any principal components for non-vectorial variables, and it might seem that choosing *random strings* for the models would be the only possibility. Nonetheless we would like to have even an

approximative order to the array, in order to speed up the computations and to guarantee better ordering results.

Estimation of initial values from the Sammon projection. One method used by us earlier was to first construct a *nonlinear projection* of the strings. In the *Sammon mapping* [76], each string (or other symbolic item) is represented by a *point on a two-dimensional plane*, and analytically as a Euclidean vector $\mathbf{x}(i)$. Let the distance of two symbols $s(i)$ and $s(j)$ be $d(i, j)$. The idea in the Sammon mapping is to to approximate the $d(i, j)$ by the vectorial distances $||\mathbf{x}(i) - \mathbf{x}(j)||$ of the corresponding points on a *two-dimensional plane*. Only an optimal solution is generally possible, in which the approximation error is minimized. In the original method, the optimum is sought by a gradient-descent method of the error function. In our case we must start from the distance matrix of symbolic items. Then it is possible to pick up at random two items (points), and since their vectorial difference in general does not comply with the $d(i, j)$, to move these points (symmetrically) towards or away from each other to match $d(i, j)$. These corrections must be repeated for other randomly chosen pairs of items and making only small corrective steps, and this may require hundreds of thousands of correction operations. After that, the resulting two-dimensional Sammon projection may be copied, e.g., to the `som_cplane` of a suitable size and form. This still requires some manual fitting. Naturally this is still a nonlinear projection, no SOM.

However, it has turned out that in this method, also the initial values of the Sammon mapping ought to be roughly ordered, not only because its will speed up the computations, but also to guarantee a good ordering result.

Especially for large SOMs, automatic initialization methods are needed.

28.7 The GENINIT projection method

In this subsection we introduce a simple and fast projection method. First, the four strings that are most distant from the other strings and also from each other are searched. After that, the horizontal and vertical coordinates of the items in the projection are determined on the basis of distances from these four points.

Special coordinate systems with focal points. Before we introduce the new projection method for strings, it may be useful to remind about certain generally known special *confocal coordinate systems* in geometry. Consider a Cartesian (x, y, z) space where two *focal points* are defined, say, at $x = -1, y = 0, z = 0$ and $x = +1, y = 0, z = 0$. If we define a *coordinate surface* as the set of points, whose *difference of distances from the two focal points is constant*, we find out that this surface is a *cylindrical hyperboloid*, symmetric with respect to the x axis. However, if we define the coordinate surface as the set of points, whose *difference of squares of distances from the two focal points is constant*, we obtain a surface that is a *plane perpendicular to the x axis*. With different values of the constant we obtain a family of such coordinate surfaces.

Now we define two families of coordinate surfaces of the latter type. Imagine a sphere, similar to the globe, and *two pairs of focal points*, one pair placed at the two poles, and the second pair on the equator, at opposite sides of the globe. Then imagine a set of latitudes and another set of longitudes and their crossing points. If we want to project these crossing points onto a plane that passes through the two pairs of focal points, we define two families of coordinate surfaces: one defined as constant differences of squared distances from the poles, and second coordinate system as constant differences of squared differences from the equatorial focal points, respectively. All of these distances are measured directly through the three-dimensional space, not along the surface of the globe. When the crossing points of the latitudes and longitudes are plotted on a two-dimensional plane using these two coordinates, we obtain the image shown in Fig. 44.

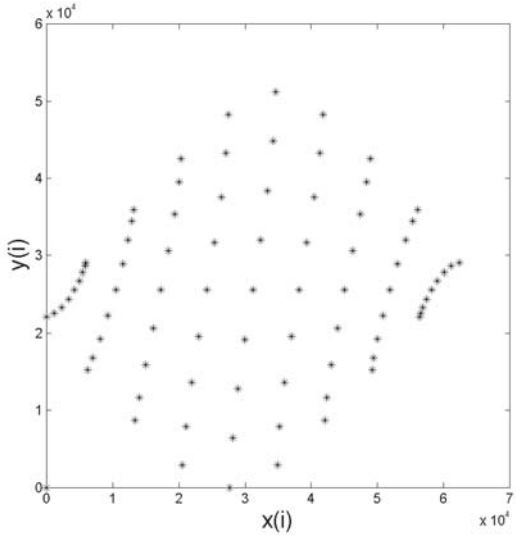


Fig. 44. The GENINIT projection of a globe onto a plane, by plotting the latitudes and longitudes using differences of their squared distances from two pairs of focal points, one pair of focal points placed to the poles, and the second pair to the opposite points of the equator, respectively.

Application of the focal-point principle to strings of symbols. Of course the strings of symbols do not behave like Euclidean vectors, and the space of strings is not three-dimensional, but we may see here a possibility to produce projections of items for which only the mutual distances can be defined. First we look for strings that are *most distant* from each other, and which also would have as large a sum of all strings as possible. We find four such strings,

all of which have the distance of *nine* units from each other, and which also happen to have the largest sums of distances from the other points. They are: `alexander, guilherme, toshiyuki` and `francesco`. Their sums of distances from all of the other strings are 193, 190, 189, and 185, respectively. Let us call these string as *focal points*.

From these four strings we select *two pairs*, e.g., `(6,11)` and `(12,25)`. The Levenshtein distance of two items `a` and `b` is denoted `D(a,b)`.

First we sort the *differences of distances from the first pair of items* and obtain the sequences of ordered values `H` and their indices `Kh`, respectively. Let us call these the *horizontal coordinates* of the items on the display. In the same manner we sort the *differences of distances from the second pair of items* and obtain the second sequences of ordered values `V` and their indices `Kv`, which we call the *vertical coordinates* of the items on tte display. The question is, whether we should use the true distances `(H,V)` or the sorting indices `(Kh,Kv)` as the coordinates. In the geometric example we used the squares of distances, but since we are mainly interested in the topographic relations of the strings, we shall choose the indices. There will be an extra benefit of this choice, as will be shown shortly. Neither is there then any reason to square the differences, since the strings are not Euclidean entities anyway. So, first we do the sorting:

```
  I1 = 6;
  J1 = 11;
  [H,Kh] = sort(D(:,I1) - D(:,J1));
  I2 = 12;
  J2 = 25;
  [V,Kv] = sort(D(:,I2) - D(:,J2));
```

Let us now plot the names on a two-dimensional plane using the coordinates `(Kh,Kv)`. The results are shown in Fig. 45.

```
xScale = 7/24;
yScale = 8/24;
 for i = 1:25,
    x(i) = find(Kh == i);
    y(i) = find(Kv == i);
 end
 plot(x,y,'k.');

% Construction of the  texts T for the names
 A = 'abcdefghijklmnopqrstuvwxyz';
 for i = 1:25,
     T = A(w(i,1));
      for j = 1:p(i)
         T = [B A(w(i,1+j))];
      end
      if i == 1 || i == 2 || i == 8
```

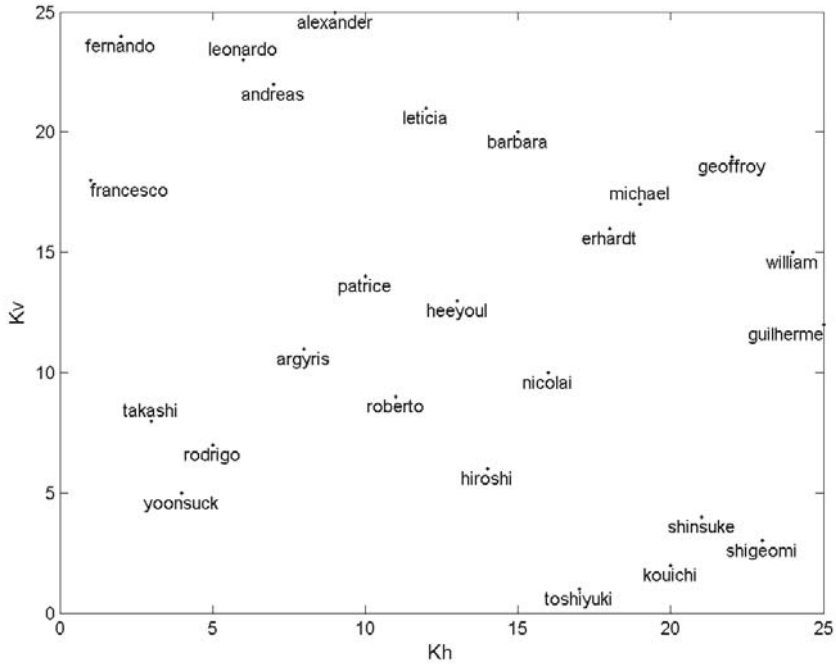


Fig. 45. The GENINIT projection of the 25 names based on their differences of Levenshtein distances from four items that are most distant from each other.

```
        text(x(i),y(i),T,'horizontalalignment', ...
        'center','verticalalignment','bottom');
    else if i == 11
        text(x(i),y(i),T,'horizontalalignment', ...
        'left','verticalalignment','top');
      else if i == 6
         text(x(i),y(i),T,'horizontalalignment', ...
         'right','verticalalignment','top');
        else
          text(x(i),y(i),T,'horizontalalignment', ...
          'center','verticalalignment','top');
        end
      end
    end
end
xlabel('Kh','FontSize',12);
ylabel('Kv','FontSize',12);
```

Compression of the GENINIT projection. One of the benefits of using sorting indices in defining the coordinates of the GENINIT method is that the plot becomes more even, when compared with the use of true distances. However, an even bigger benefit can be seen upon closer inspection of the plot: the image is divided into an 25 by 25 array onto which the items are projected, and there is only one item on each row and in each column! Such a sparse 25 by 25 matrix can be compressed, e.g., by the `find` function into a 5 by 5 array, where the items are still coarsely ordered. This kind of an array is shown below. It has been made as a `tabular` array of LaTeX for convenience.

fernando	alexander	leticia	michael	geoffroy
francesco	leonardo	barbara	erhardt	william
takashi	andreas	heeyoul	nicolai	guilherme
rodrigo	patrice	roberto	kouichi	shinsuke
yoonsuck	argyris	hiroshi	toshiyuki	shigeomi

The compressed GENINIT initialization of the 25 names.

The "small SOM." In order that we can call the following case an "SOM", at least the *neighborhood function* must somehow be involved with its training. In a square array of 5 by 5 models, with one square dedicated for every input item, the neighborhood function shall be equal to 1 in a *neighborhood set* consisting of the closest neighbors, and 0 outside it. In the inside of the array the neighborhood set has 3 by 3 nodes, and fewer at the borders. Like in the batch-training SOM, the middle element of the neighborhood set is updated by the "average" over the neighborhood set, which in this case, for strings of symbols, is defined as the *median* of the strings in the neighborhood set. Actually the updating operation is a bit more complex, "supervised," in order that the convergence is guaranteed, as will be seen next.

Supervised training of the "SOM." Notice that in this case we have no external inputs, because all of the inputs are already there in the map as the initial values of the models, and if we would input the same items again, they would only identify themselves as the "winners." So in this example we do not have lists of "winners" associated with the nodes, only the models themselves. *If we would replace the old item at a node by the median of the due neighborhood, we would generally create a new copy of an item and lose one item, so after that we would not have all of the 25 names left in the map any longer.* The only

sensible thing to do then is to *swap the median and the contents of the node that we are processing*. But even then it is not sure that the *global* degree of order in the SOM is increased, because increasing the order in a neighborhood may disturb the order elsewhere in the map. So *we must do the swapping only conditionally, provided that the global index of order* `OI` is thereby decreased. The following script will guarantee that. Before applying this script, we need to execute the `makemedian(w)` function to obtain the median `w(med(1),:)`. (Note that `aux` is a temporary storage used in swapping.)

```
oldw = w;
oldOI = orderindex(m);
aux = w(med(1),:);
w(med(1),:) = w(v,:);
w(v,:) = aux;
m = w;
OI = orderindex(m);
if OI < oldOI
    w = w;
else
    w = oldw;
end
m = w;
```

The tabular array below represents the "SOM" array after supervised training. While the order of index with the original, randomly ordered array was 278, and in the compressed GENINIT array it was 264, after 1000 *supervised training step* it was decreased to 248. However, there may still exist several alternatives for the (local) optima of ordering, and the optimal constellation of the strings seems to be very vague, *also depending from which direction we look at the projection*. Anyway, this 5 by 5 array may now serve as a starting point for the training of an enlarged SOM.

nicolai	hiroshi	takashi	fernando	alexander
argyris	heeyoul	barbara	erhardt	geoffroy
andreas	roberto	leonardo	william	guilherme
rodrigo	francesco	leticia	michael	shinsuke
patrice	yoonsuck	kouichi	toshiyuki	shigeomi

The "SOM" array after supervised training.

28.8 Computation of a genuine SOM for strings of symbols

As said, we have earlier succeeded in producing ordered maps starting with *randomly generated strings as initial values of the models*, but then we have had significantly larger data bases for the input items. Also the initialization methods have been somewhat different from those discussed in this book. Here we have introduced a simple and rather straightforward method for automatic initialization. Since this initialization is coarsely ordered, too, the final training is expected to take place significantly faster. At least it seems better than the semi-manual Sammon-map initialization, and its is well compressed.

Initializing a large SOM for this example. Now we want to increase the size of the SOM array. *The added 16 interstitial nodes will be initialized by interpolation, and an extra garbling is performed on them.* This garbling might not be necessary, but since we are now demonstrating a genuine SOM process, a garbling is supposed to give an extra proof for that the algorithm is really capable of ordering the strings topographically. The 9 by 9 initialized array is shown in the tabular array below.

nicoli	nicyrisc	arygyris	ardyeiz	andreasu	axodrego	rodrio	adrigo	patrce
niceoshi	nrrohal	vegyrul	aeeeyil	rjndrrto	aoanco	foarqcgo	paonto	pooniece
hyiroshi	herfosul	heeyonl	reeetl	robdrto	ranceo	francesyo	yognrto	yoonsjuck
tirashoi	harosrt	berbouoa	neenara	reonro	reanaaia	fruicea	yoeocia	younrhi
takahi	xakbsra	brbara	lernarl	eonardo	letnaa	lgeticia	koticixa	kduichi
aknahi	eahwadt	brharnra	legarda	gillaad	wexlaea	mechceam	kohiel	osichi
fecrnando	ehnadt	erhagdt	wiardt	qilliam	wicliell	mictael	mshiel	toshiyuki
lexando	glhfladt	gehnfrdt	gullrt	guliam	siliel	shcnpael	mmsgiul	sshioui
alfxander	gleyanoy	geoxfroy	guilfroyc	guilherme	suislseke	shinsuke	shingsuke	uhigeomi

Initialization of the SOM of 25 names on a 9 by 9 array using interpolation of the interstitial position. For more random initial values, the strings have been garbled.

The Batch Map for strings. The final SOM algorithm to be constructed is very much similar to the "Batch Map," where instead of the *mean* over the winner lists in the neighborhood of a node, *medians* of the strings in the same

neighborhood lists are used to replace the old values of the strings in the nodes. However, since the strings are discrete entities and their domain of values is rather narrow, there often occur *ties*, i.e. the distances to be compared are often equal. The ties may result (1) in the *winner search*, (2) in the *identification of the medians*, and (3) when *calibrating* the SOM, or labeling the best-matching nodes with standard strings, or strings with known identity.

A tie occurs very frequently if the strings are short like the names in this example. The ties are most severe if the comparisons are based on *unweighted Levenshtein distances*. If the strings would be obtained, say, in speech recognition, the probabilities for the various types of error (replacement, insertion, deletion) would be different even for each symbol separately, and the *weighted Levenstein distances* were used, the probability for a tie would then be significantly smaller. The example we are now discussing is one of the worst, because the names used as data are short and have a unique form.

First we present a couple of methods to break the ties.

Comparison with several nodes. In winner search, when the tie is due, one could switch from the search of a single winner to the identification of *that set of neighboring nodes, which have the smallest average distance to the input*. This strategy resembles the winner-search criterion of Heskes and Kappen [23].

Of cause this kind of a tie break causes complications in programming, and is different at the borders of the SOM array compared with the middle of the array. Also one must be prepared, in principle at least, for that if the first choice of comparison set does not yet break the tie, more nodes must be involved in comparison, and so on. So let us not use this method in this rather artificially constructed example which had to be kept as transparent as possible.

Random choice for tie breaking. Since the determination of the winner in terms of average distances from a neighborhood of the target node is so complicated, one is tempted to look for simpler alternatives for tie break. One of them is the *random choice from the ties*. Another speculation is to give the preference to either longer or shorter strings. In our numerous experiments we have seen no benefits of favoring the candidates on the basis of the lengths of the strings, whereas the random tie break is almost as good as the enlargement of the neighborhood in winner search, especially if we are not dealing with statistical classification of strings. In the present demonstration we thus use the random choice between the ties in selecting the winner, as well as in searching for the median in a neighborhood set in updating the SOM.

Definition of the winner lists at the nodes. In the Batch Map for vectorial data we could use a single *accumulator buffer* associated with every node of the SOM. In these buffers we immediately formed the sums of inputs mapped to that node. A counter of the number of addends was also needed at each node. This method is not possible in the Batch Map for strings, because we do not form sums and means of the mapped inputs but their *medians*. In the

first phase of computation we must therefore keep up *replica* of all of the input strings that are mapped to the winner nodes. These replica, in this example, are stored in 81 *winner lists* associated with the SOM nodes. We must also store *the lengths of their occupied parts* which are given by the auxiliary variables `len` for each node. In this example, for safety, 200 locations have been reserved for each winner list, although it turns out that we will need only a fraction of them.

> *However, we now run into a particular problem. When defining the array for the winner lists, we have used one index to denote the node, and the second index to define the position in the winner list. But the strings to be stored are formally* vectors of symbols*, with as many components as there are symbols in the string, and for that we would need the third index. There is an option in MATLAB to use three-dimensional arrays, as we have seen in the QAM example, but there exists a much easier and simpler solution to this problem. We need not store the strings as such in the lists but only the* indices of these items (i.e., pointers), the index being identifiable by its order when this string was generated. *We define this index as* `v = 81*(repet-1) + s`, *which is a scalar number.* The original strings can be restored any time on the basis of this index.

```
% Construction of the winner lists at all nodes
winlist = zeros(81,100);
len = zeros(81,1);
```

Garbling of input data. We want to make a genuine experiment, where the training data of the SOM are randomly distributed variables. Since we are dealing with an artificial example, we simulate the erroneous input strings by generating artificial editing errors (replacements, insertions, and deletions) to the set of standard input strings. We could have garbled those 25 original names, but it seems that we have a better fidelity with real examples if we use garbled versions of the 81 strings used as initial values of the SOM, as shown in the tabular array on p. 148. We generate a great number of garbled inputs automatically. However, it would be valuable if the streams of random numbers were repeatable for experimenting and diagnosing. In MATLAB there are special options for making repeatable random-number streams. Since we need not have a high quality of random numbers in this demonstration, we use the old initialization of the random-number generator using, e.g., the simple command line `rand('seed',20000)`. The MATLAB depreciates this method, but we use it here for simplicity.

When we generate garbled versions of the strings, we identify each version with the running index `v` of each of the 81 words for a reason that will be explained shortly. We decided to make exactly three random errors to each word. The letter to be garbled in the word is defined by its position `p`. The error type `errtype` can be a replacement, insertion, or deletion of that letter, with equal probability for each error type. A set of 81 thrice garbled names is repeated three times, each time having a different random garbling. In this way we obtain 243

training patterns at each cycle. This training episode, with different randomization, is repeated 50 times; so the number of random training patterns was in total 12'150.

```
% Generation of garbled input strings

ABC = 'abcdefghijklmnopqrstuvwxyz';
M = M0;

for cycl = 1:50  % cycles belonging to teaching

    winlist = zeros(81,25); % winlists reset at each cycle
    len = zeros(81,1);

  for repet1 = 1:3 % three statistically independent input sets

    for s = 1:81 % one of three input sets

        v = 81*(repet1-1) + s;
        A1 = M0(s,:);
        A = A1(find(A1~=' '));

      for repet2 = 1:3  % each input set is garbled thrice

        P = floor(length(A)*rand) + 1;
        errtype  = floor(3*rand) + 1;
        Lett = ABC(floor(26*rand) + 1);
        if errtype == 1
           A(P) = Lett;
        end
        if errtype == 2
           C = '';
           if length(A)<9
              for i = 1:P
                  C(i) = A(i);
              end
              C(P+1) = Lett;
              if length(C)<9
                 for i = P+2:length(A)+1
                     C(i) = A(i-1);
                 end
              end
              A = C;
           end
        end
        if errtype == 3
```

```
          A(P) = '';
        end
      end  % end of repet2 (garbling)

        A2(v,:) = '          ';
        for P = 1:length(A)
            A2(v,P) = A(P);
        end
```

As you may have noticed, the trailing blanks from the strings `A` have now already been removed, and we continue by winner search:

```
        % Matching with SOM nodes and making winnerlists
        for j = 1:81
            B1 = M(j,:);
            B = B1(find(B1 ~= ' '));
            LD = levenshtein(A,B);
            D(j) = LD;
        end
        [W,w] = sort(D);
```

Random tie break in sorting. The function `sort` produces a sorted list of the matches. In the case that there are `n` ties (minima) in sorting, we choose the index `r` among them randomly, increment the corresponding list length, and store the winner.

```
         n = length(find(w==w(1)));
         r = floor(n*rand) + 1;
         len(w(r)) = len(w(r)) + 1;
         winlist(w(r),len(w(r))) = v;
    end
end
```

Indices of the neighborhoods. The winner lists have now been completed in this way. Next we define the *neighborhood sets* around the nodes `c`.

```
% Definition of neighborhoods
        for c = 1:81
            cj = floor((c-1)/9) + 1;
            ci = mod(c-1,9) + 1;
            if c == 1
               c1 = [1 2 10 11];
               lenc = 4;
            end
            if c == 9
               c1 = [8 9 17 18];
```

```
   lenc = 4;
end
if c == 73
   c1 = [64 65 73 74];
   lenc = 4;
end
if c == 81
   c1 = [71 72 80 81];
   lenc = 4;
end
if cj == 1 && ci > 1 && ci < 9
   c1 = [c-1 c c+1 c+8 c+9 c+10];
   lenc = 6;
end
if cj == 9 && ci > 1 && ci < 9
   c1 = [c-10 c-9 c-8 c-1 c c+1];
   lenc = 6;
end
if ci == 1 && cj > 1 && cj < 9
   c1 = [c-9 c-8 c c+1 c+9 c+10];
   lenc = 6;
end
if ci == 9 && cj > 1 && cj < 9
   c1 = [c-10 c-9 c-1 c c+8 c+9];
   lenc = 6;
end
if ci > 1 && ci < 9 && cj > 1 && cj < 9
   c1 = [c-10 c-9 c-8 c-1 c c+1 c+8 c+9 c+10];
   lenc = 9;
end
```

Computation of the medians and their substitution to the nodes. The following pieces of script need a very detailed explanation. First we define the running indices of the neighborhoods, for each map node `c` separately, the indices relating to the original SOM. Notice that the indices `v` of the winner lists are only numerical indices of the true strings that were given in `A2(v,:)` earlier. The true strings also contain a number of blanks to make all strings 9 symbols long. The sum of distances of strings in each sublist in a particular neighborhood from the strings in all other sublists in the same neighborhood are computed. But we cannot be absolutely sure whether all these sublists are nonempty, and therefore we have to use the condition `if v>0` . Notice also that the sum of distances from all elements in one list to all elements in all of the other lists in the neighborhood is only related to index `a`of the first sublist, and therefore the sums to be compared are components `a` of vectors `summa`.

We found out that the random tie breaking was not necessary in this final training, due to the big length of the winnerlists and all-random repetitions.

```
    % Computation of medians
           summa = zeros(length(c1),1);
           for a = 1:length(c1)
               for z = 1:len(c1)
                   v = winlist(c1(a),z);
                   if v>0
                      A1 = A2(v,:);
                      A = A1(find(A1 ~= ' '));
                   end
                   for b = 1:length(c1)
                       for z = 1:len(c1)
                           v = winlist(c1(b),z);
                           if v>0
                              B1 = A2(v,:);
                              B = B1(find(B1 ~= ' '));
                              LD = levenshtein(A,B);
                              summa(a) = summa(a) + LD;
                           end
                       end
                   end
               end
           end
       end
           [Med,med] = sort(summa);
           mediaani(c,:) = M(med(1),:);
   end
       M(c,:) = mediaani(c,:);
 end
end
```

Now *all of the 50 training cycles* have been completed. The training was made to continue by repeating similar training cycles until the convergence of the algorithm was satisfactory. This time no exact convergence was expected, since new random garbling of the inputs was carried out for all of the 50 training cycles, and the inputs were not the same during all cycles.

Calibration of the nodes. Next we have to *calibrate* the nodes of the SOM. This we want to do using the original, errorless names as test strings, and we input them looking for the winners. The result shown in the tabular array below almost completely coincided with the initialization, *although all training strings were three-times garbled versions.* However, we lost one name, `shigeomi`, which was replaced by `nicolai`.

Discussion. Although the construction of the SOM for symbol strings contained many complicated phases, the present problem was still only a "toy example." For one thing, the density function of the erroneous input strings, which was defined artificially, was almost ideally *uniform*. This was reflected in the almost rectangular constellation of the calibration items on the map. It is plausible that with longer, statistically distributed strings and a bigger input data base, not only the *ties* would occur less often, but the SOM would also reflect *meaningful clusters of data* more clearly.

In the case that the dissimilarities of the data items would be defined directly by distance matrices, whether theoretical, or computed by approximate methods like the FASTA method, many phases described above could be followed.

nicolai	argyris	andreas	rodrigo	patrice
hiroshi	heeyoul	roberto	francesco	yoonsuck
takashi	barbara	leonardo	leticia	kouichi
fernando	erhardt	william	michael	toshiyuki
alexander	geoffroy	guilherme	shinsuke	nicolai

Genuine SOM of 25 names on a 9 by 9 array.

29 The Supervised SOM

Simple supervised training of the SOM

Objective:

The original SOM is definitely an unsupervised learning method. Therefore it does not classify input patterns at statistically optimal accuracy. However, it can be made to perform better; the first remedy used in applications was to add class information to the input patterns, whereupon the decision borders between the classes were emphasized almost optimally.

It has been stated generally that the SOM is an *unsupervised classification method*. Its main operation is a nonlinear projection from a high-dimensional data space onto a usually two-dimensional array of nodes. Therefore it is not expected that it would simultaneously cluster input data items at an optimal accuracy, when referred to the probabilistic methods of classification. Nonetheless its computation is very easy, and with some extra means it *can* be made to classify objects at a near-optimal accuracy. We noticed that already in our first attempts to recognize *speech by phonemes* around 1984 (cf. [36]).

In order to make an SOM to operate in a *supervised* manner, we have to give *class information* already in connection with the input data. The simplest way to do this is to add extra components to the input pattern vector that indicate the class-affiliation of the input pattern. If `X` is the input pattern, then its class-affiliation is defined by a *class vector* `C` that has as many components as there are classes. Usually the class vectors are *unit vectors* that have a 1 in the component showing the class, and 0 elsewhere. The combined input vectors of the SOM are then of the form `U = [X C]`.

The effect of the unit vector part is to increase the *clustering tendency* of those input vectors that belong to the same class, and to make vectors of foreign classes expel each other. When the `X` part and the `C` part are weighted suitably, one can optimize the class separation of the samples. We demonstrate this with two classes of two-dimensional artificial vectors that obey the Gaussian distributions but overlap significantly. One of the classes is centered at $(-2, -2)$ and has the standard deviation equal to 1, while the second class is centered at $(+2, +2)$ and has the standard deviation equal to 2.

The script of the Supervised SOM is the following. It starts with the definition of the simulation inputs `X` and `C`. For the latter we give the values -3 and -3, respectively (to put a little more weight to the class parts). Then, for a reference, we generate 1000 random samples `Y` of input data in order to delineate the form of the input density function. These samples are plotted to the left-hand subimage of Fig. 46, but for graphic reasons, the Gaussian distributions have been cut at the framing when drawing the left-hand subimage. (The `Y` values are not involved in the computation of the SOM.)

```
X = zeros(10000,2);
C = zeros(10000,1);
Y = zeros(1000,2);
for i = 1:10000
    if rand <.5
       C(i) = 3;
       X(i,:) = randn(1,2) - 2;
    else
       C(i) = - 3;
       X(i,:) = 2*randn(1,2) + 2;
    end
end
for i = 1:1000
    if rand <.5
       Y(i,:) = randn(1,2) - 2;
    else
       Y(i,:) = 2*randn(1,2) + 2;
    end
end
Y = Y(find(Y(:,1)>-4 & Y(:,1)<-5 & Y(:,2)>-4 & Y(:,2)<5),:);
```

This was the input-data part of the script. Next the pattern parts `X` and the class parts `C` are concatenated into combined input vectors `U` to the SOM:

```
U = zeros(10000,3);
for u = 1:10000
    for v = 1:2
        U(u,v) = X(u,v);
    end
    U(u,3) = C(u);
end
```

Now we are ready to compute the SOM:

```
smI = som_lininit(U,'msize',[7 7],'lattice',...
      'hexa','shape','sheet');
smC = som_batchtrain(smI,U,'radius',[3 1],...
    'trainlen',50);
sm = som_batchtrain(smC,U,'radius',[1 1],...
    'trainlen',200);
```

We are plotting the models as points, which are connected with auxiliary lines that link the nodes that are horizontal or vertical neighbors in the SOM. For the plotting instructions we must re-dimension the `M` vectors.

We also want to identify those models , which belong to class C_1. For that we use the command `M1 = M1(find(M1(:,3)>0),:);` below. We mark these models by an asterisk.

The `plot` instructions end up with `...-5,-4,'k.',5,5,'k.'` to set up similar scales to the two subimages.

```
M = sm.codebook;
M1 = M;
M1 = M1(find(M1(:,3)>0),:);
M = reshape(M, [7 7 3]);
for k=1:2
    subplot(1,2,k);
    if k == 2
       plot(M(:,1,1),M(:,1,2),'k-',M(:,2,1),M(:,2,2), ...
       'k-',M(:,3,1),M(:,3,2),'k-',M(:,4,1), ...
       M(:,4,2),'k-',M(:,5,1),M(:,5,2),'k-', ...
       M(:,6,1),M(:,6,2),'k-',M(:,7,1),M(:,7,2), ...
       'k-',M(1,:,1),M(1,:,2),'k-',M(2,:,1), ...
       M(2,:,2),'k-',M(3,:,1),M(3,:,2),'k-', ...
       M(4,:,1),M(4,:,2),'k-',M(5,:,1),M(5,:,2), ...
       'k-',M(6,:,1),M(6,:,2),'k-',M(7,:,1), ...
       M(7,:,2),'k-',M1(:,1),M1(:,2),'k*',-5,-4,'k.',5,5,'k.');
    end

    if k == 1
       plot(Y(:,1),Y(:,2),'k.',-5,-4,'.', 5,5,'.');
    end
end
file = 'SupervisedSom';
print('-dpng',[file '.png']);
```

Since the artificial training inputs and model vectors (not counting the class information) were two dimensional, we can easily show their constellation in the right-hand subimage of a two-dimensional diagram, Fig. 46. Class C_1 is emphasized by asterisks. We can see that the SOM network is *stretched* between the classes C_1 and C_2. If we had been able to plot the SOM three dimensionally, we could have seen that C_1 would pop up from the plane by 3 units and C_2 would be sunken by 3 units, respectively.

Especially with higher-dimensional input data the organization of the SOM would be improved due to the `C` parts of the training inputs.

A new, unknown vector does not yet have the `C` part, and it must be classified (i.e., by determination of the winner node with known classification) based on its known `X` part only. Its classification result is found in the `C` part, *indicated by the maximum value of the component that was supposed to be equal to* 1 *in the unit vector.*

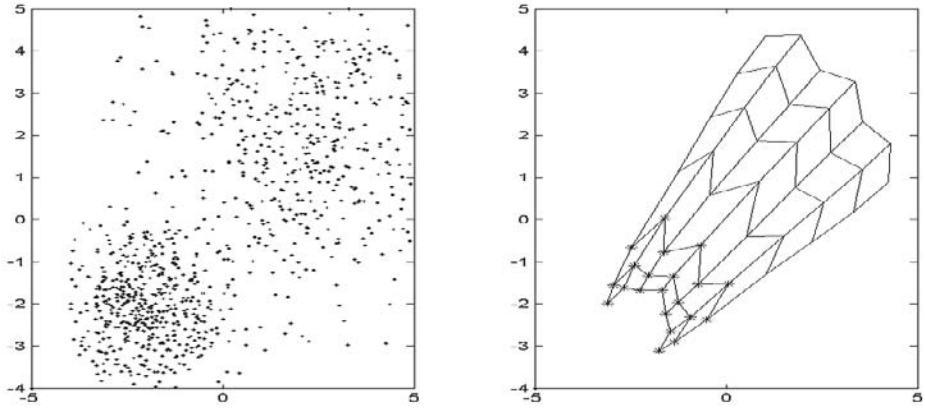


Fig. 46. This picture shows the *first two components* of the input and model vectors of a Supervised SOM. The third component represents the *classification* of the model vectors, and is not shown in this picture. In the left-hand subimage, the distribution (with 1000 samples) of the input vectors is demonstrated. In the right-hand subimage, the SOM model vectors after 10,000 training steps are shown; they are situated at the crossings of the auxiliary lines. There are in all 7 by 7 model vectors, and they have been adapted to the input density function. The models corresponding to the classes C_1 and C_2 have further been expelled automatically from each other in training, due to the different class vector parts of the input vectors, which impose bigger vectorial differences between the vectors of different classes. In the right-hand subimage, the 21 model vectors of class C_1 have been emphasized by an asterisk, and the rest of the model vectors belong to class C_2. The model vectors of the two classes are separated by a bigger gap between them. However, in the automatic classification of unknown input vectors, the classification result is mainly determined by those model vectors that are closest to the discrimination limit (between the model vectors of class C_1 and class C_2), i.e., those points that have been labeled by the asterisk or not. Cf. the Learning Vector Quantization to be discussed in the next section.)

Discussion. The Supervised SOM was the first *supervised* algorithm in the SOM category, and it was used in the demonstration of typing out text from unlimited Finnish and Japanese speech ("Phonetic Typewriter.") In a later series of experiments the Supervised SOM was replaced by various versions of Learning Vector Quantization algorithms (cf. the next section). For best class separation it would be necessary to experiment with different sizes of the `C` parameters. Notice that if the Learning Vector Quantization is used, there will be no `C` parts any longer.

30 The Learning Vector Quantization

Error-controlled supervised training of the models

Objective:

There are two functions `lvq1` *and* `lvq3` *in the SOM Toolbox, which are related to the SOM but differ from it in two important aspects. Unlike the SOM which is definitely an unsupervised learning method, these two functions belong to supervised learning methods; they are called Learning Vector Quantization. They have been designed for near-optimal class separation, but the models they construct are not topographically ordered. They just describe class density functions like the codebook vectors in k-means clustering, but normally their objective is to define a near-optimal separating surface between two classes.*

Although we have seen that the SOM is able to carry out a classification of the input items, nonetheless we must say that the classification accuracy is not as good as for special classification algorithms, say, the ones that are based on the *Bayes* theory of conditional probabilities. On the other hand, like we saw in the fourth example that dealt with the classification of mushrooms, the SOM was able to display the roles of the individual models in approximating the *forms of the class distributions.*

Now we switch to another class of algorithms that were developed concurrently with the SOM. They are called by the generic name *Learning Vector Quantization (LVQ)*, and there are at least three different kinds of them, LVQ1, LVQ2 and LVQ3, which differ from each other in minor details of the training process. First we introduce the LVQ1, which implements the basic idea in reduced form.

The Bayesian decision borders. The problem of *optimal decision in statistical pattern recognition* is basically discussed within the framework of the *Bayes theory of probability.* Assume that the input data vectors $\mathbf{x}$ ensue from a finite set of sources or *classes* $\{S_k\}$, and the *distributions* of the latter are being described by the *conditional probability density functions* $p(\mathbf{x}|\mathbf{x} \in S_k)$. In general, the density functions of the classes overlap, which is reflected in *classification errors.* The objective in statistical pattern recognition is to minimize these errors. Without strict and complete discussions, it is simply stated that the minimization is made by introducing the *discriminant functions*

$$\delta_k(\mathbf{x}) = p(\mathbf{x}|\mathbf{x} \in S_k)P(S_k) \quad .$$

Here $P(S_k)$ is the *a priori* probability of class S_k. Then the unknown samples $\mathbf{x_i}$ are classified optimally (i.e., the rate of misclassification errors is minimized on the average), if the sample $\mathbf{x_i}$ is decided to belong to class S_k when

$$\delta_c(\mathbf{x}_i) = \max_k\{\delta_k(\mathbf{x}_i)\} \quad .$$

To illustrate what these equations mean, let us look at Fig. 47. We are demonstrating the classification of *scalar-valued samples* x. There are three classes S_1, S_2 and S_3. They are defined on the x axis by three Gaussian-formed discriminant functions $\delta_1(x)$, $\delta_2(x)$, and $\delta_3(x)$, respectively. The optimal Bayesian borders are indicated by dotted lines, and they divide the x axis into three zones. In the first zone the discriminant function $\delta_1(x)$ has the largest value, and in the second and third zone, $\delta_2(x)$ and $\delta_3(x)$ are largest, respectively.

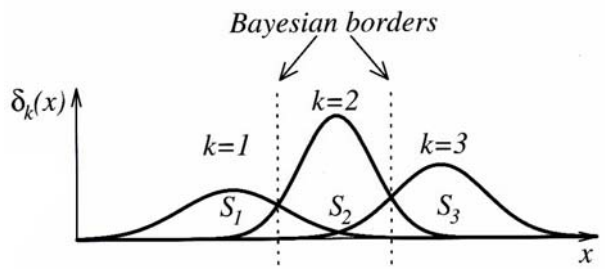


Fig. 47. Distributions of the scalar samples in three classes S_1, S_2 and S_3, and definition of the due Bayesian borders.

A preliminary example. The idea of the Learning Vector Quantization (LVQ) may become clearer if we first illustrate it with a simple example. Consider Fig. 48, which shows the density functions (Gaussian) of two overlapping classes.

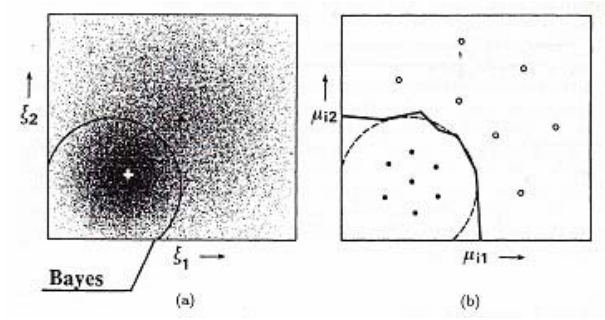


Fig. 48. (a) Small dots : Superposition of two symmetric Gaussian density functions corresponding to the classes S_1 and S_2, with their centroids shown by the white and black cross, respectively. Solid line: the Bayes decision border. (b) Large black dots: model vectors of class S_1. Open circles: model vectors of class S_2. Dashed line: Bayes decision border. Solid line: decision border in the Learning Vector Quantization. (From [39].)

The samples are *two dimensional*, with their values denoted by $\mathbf{x} = [\xi_1, \xi_2]$. These distributions are plotted in Fig. 48(a) as small dots. In the Bayesian theory of probability, these distributions would be separated optimally, with the minimum number of misclassifications, if the separating curve would be a circle. Now the essential policy, illustrated in Fig. 48(b), is to place a number model vectors $\mathbf{m}_i = [\mu_{i1}, \mu_{i2}]$ into the distributions of both classes such that *the class borders are defined by those points on the* $[\mu_{i1}, \mu_{i2}]$ *-plane that have equal distances from the closest model vectors of each class.* The $\mathbf{m}_i$ will be assigned permanently to either class S_1 or class S_2. The exact values of these models are determined in the supervised learning by the LVQ algorithm.

30.1 The Learning Vector Quantization algorithm LVQ1

Initialization. Selection of the *number* of codebook vectors (models) in the Learning Vector Quantization into each class follows similar rules as determining the number of models for the SOM. As a matter of fact, a proven policy is to carry out first the computation of an SOM with all of the tentative model vectors assigned into the same class. When the SOM models have acquired steady values with the final neighborhood radius, the models are *calibrated* by the class symbols. After that the topographic order acquired in the SOM training process is forgotten, and the training continues from these values, regarded as the intial values for the LVQ algorithm. By the above method we also obtain an approximately *optimal number* of models in each class, which may not be the same for all classes, due to the different forms of the density functions of the classes.

The LVQ1. The LVQ1 algorithm contains the Learning Vector Quantization idea in the simplest reduced form. We cannot present its mathematical derivation here because of space limitations, and since this is not a mathematical textbook. However, the principle is closely related to the k-means classification:

> *Give the training data one at a time together with its classification. If the winner model has the same classification, increase the matching of the model with the input. If the classifications disagree, reduce the matching (i.e., carry out a correction).*

Let us now first write the training equations in the stepwise recursive form; later we will show that the batch training procedure is possible for the LVQ1, too.

$$\mathbf{m}_c(t+1) = \mathbf{m}_c(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{m}_c(t)]$$

if c is the index of the winner model and $\mathbf{x}$ and $\mathbf{m}_c$ belong to the same class,

$$\mathbf{m}_c(t+1) = \mathbf{m}_c(t) - \alpha(t)[\mathbf{x}(t) - \mathbf{m}_c(t)]$$

if c is the index of the winner model and $\mathbf{x}$ and $\mathbf{m}_c$ belong to different classes.

Here $\alpha(t)$ is the *learning rate* that must fulfill the condition $0 < \alpha(t) < 1$, and $\alpha(t)$ is made to decrease monotonically with time: for instance, $\alpha(t) = .5A/(A + t)$, where A is a parameter that depends on the number of training steps and the magnitude of final correction. The learning rate is based on experience, and it is not needed if the following batch-training algorithm is used.

The Batch-LVQ1. The LVQ1 algorithm can also be written shorter as

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + \alpha(t)s(t)\beta_{ci}[\mathbf{x}(t) - \mathbf{m}_i(t)] \ ,$$

where $s(t) = +1$ if $\mathbf{x}$ and $\mathbf{m}_i$ belong to the same class,
but $\quad s(t) = -1$ if $\mathbf{x}$ and $\mathbf{m}_i$ belong to different classes,
and where $\beta_{ci} = 1$ for $c = i, \beta_{ci} = 0$ for $c \neq i$.

Its equilibrium condition is written as

$$\forall i, \mathrm{E}_t\{s\beta_{ci}(\mathbf{x} - \mathbf{m}_i^*)\} = 0 \quad .$$

The LVQ1 algorithm, like the SOM, can now be written as the so-called *Batch-LVQ1* algorithm as described by the following steps:

1. For the initial model vectors take, for example, those values obtained in an SOM process, where the classification of the $\mathbf{x}(t)$ is not yet taken into account.

2. Input the $\mathbf{x}(t)$ again, this time listing the $\mathbf{x}(t)$ *as well as their class labels* under each of the winner nodes.

3. Determine the labels of the nodes according to the majorities of the class labels of the samples in these lists.

4. Multiply in each partial list all of the $\mathbf{x}(t)$ by the corresponding factors $s(t)$ that indicate whether $\mathbf{x}(t)$ and $\mathbf{m}_i(t)$ belong to the same class or not.

5. At each node i, take, concurrently for the new value of the model vector the entity

$$\mathbf{m}_i^* = \textstyle\sum_{t'} s(t')\mathbf{x}(t') / \sum_{t'} s(t') \quad ,$$

where the summation is taken over the indices t' of those samples that were listed under node i.

6. Repeat from 2 a suitable number of times.

Comment 1. For stability reasons it may be necessary to check the sign of $\sum_{t'} s(t')$. If it becomes negative, no updating of this node is made.

Comment 2. Unlike in the usual LVQ1, the labeling of the nodes is allowed to change during the iterations. This has sometimes yielded slightly better classification accuracies than if the labels of the nodes had been fixed at the first steps. Alternatively, the labeling can be determined permanently immediately after the SOM initialization process.

The LVQ1 script. A simple MATLAB script for the computation of LVQ1 is shown below. Although it executes 10000 learning steps, it only takes some 10 seconds on a PC to do it.

The script starts with the definition of 10000 training vectors X and their classification (with class symbols 1 and -1, respectively). Class S_1 has the Gaussian form with standard deviation equal to 1, and it is centered at the coordinates $(-2, -2)$. Class S_2 has the standard deviation of 2 and is centered at $(2, 2)$. Also $N * N$ SOM model vectors ($N = 4$) are declared.

The initialization of the models vectors of the LVQ1 is carried out by first computing an SOM with $N * N$ model vectors, whereupon the classification of the X is not yet taken into account. In this way, the number of model vectors to be assigned to each class in the LVQ1 is determined automatically, depending o the forms of the class distributions, to guarantee roughly optimal spacings of the model vectors near the decision border, as will be seen. The labeling of the model vectors is made by a simple majority voting of the class symbols 1 and -1 of the winners.

After that the LVQ1 algorithm is applied for 10000 steps. The learning rate *alpha* is let to decrease with the training steps according to an almost hyperbolic law, which has been found suitable experimentally.

The plotting of the model vectors of the LVQ1 and the location of the decision border between the classes is computed by the function named the `voronoi` (cf. *Voronoi tessellation* [93]). This tessellation defines the borders of those input regions such that all inputs that are falling to a particular region are mapped to the same winner node; in other words, all these inputs are closest to this node. The Voronoi tessellation automatically defines the class decision border, too, because it partly coincides with the tessellation. The decision border has been drawn with a thick black line to Fig. 49.

```
N = 4;
X = zeros(10000,2);
R = zeros(N*N,1);
S = zeros(10000,1);
for i = 1:10000

% Definition of training vectors and their classes
    if rand <.5
       S(i) = 1;
```

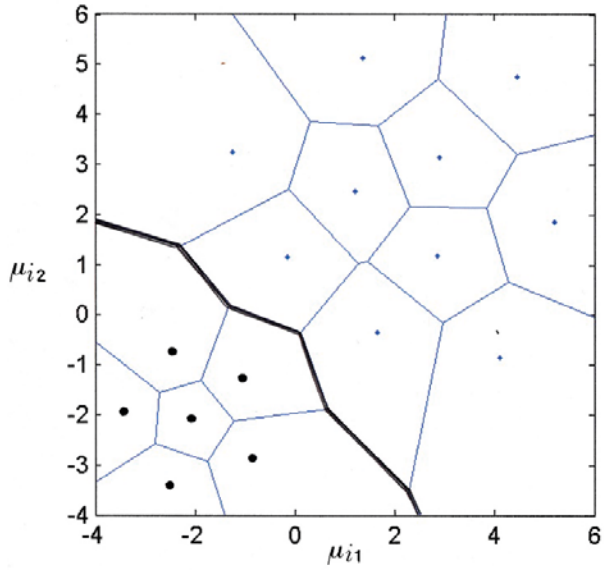


Fig. 49. The model vectors of class S_1 are drawn by large black balls and the model vectors of class S_2 by the smaller dots, respectively. The centroids of the class distributions are located at (-2,-2) and (2,2), respectively. The Voronoi tessellation defined by the LVQ1 model vectors is shown as a network of thin lines; the decision border between the classes, which partly coincides with the tessellation, is drawn by a thick black broken line. The centroids of the class distributions are located at (-2,-2) and (2,2), respectively. This example resembles the case shown in Fig. 48(b), except that the numbers of model vectors distributed between the classes (6 vs. 10) are determined automatically, in the initialization by the SOM. Notice how the model vectors of the different classes are automatically withdrawn from the region where the class density functions overlap. Especially the distribution of the model vectors of class S_2 is no longer spherically symmetric.

```
        X(i,:) = randn(1,2) -2;
    else
        S(i) = - 1; %
        X(i,:) = 2*randn(1,2) + 2;
    end
end

% SOM
smI = som_lininit(X,'msize',[N N],'lattice',...
      'hexa','shape','sheet');
smC = som_batchtrain(smI,X,'radius',[1 .01],...
    'trainlen',50);
sm = som_batchtrain(smC,X,'radius',[.01 .01],...
    'trainlen',200);
```

```
M = sm.codebook;
norms2 = sum(M.*M,2);

% Calibration of SOM
for u = 1:10000
    for v = 1:N*N
        X1 = X(u,:)';
        Y = norms2 - 2*M*X1;
        c = min(Y);
        R(c) = R(c) + S(u);  % class labels of the model vectors computed
    end
end
R = sign(R);  %  simple majority voting of the class symbols -1 and +1

% LVQ1
for u = 1:10000
    alpha = .01/(1 + u/10) + .00005;

    % Determination of winners
    for v = 1:N*N
        X1 = X(u,:)';
        Y = norms2 - 2*M*X1;
        c = min(Y);

       % Updating
        sigma = S(u)*R(c);
       M(c,:) = M(c,:) + alpha*sigma*(X(u,:) - M(c,:));
    end
end

% Voronoi tessellation
voronoi(M(:,1),M(:,2))
```

The Batch-LVQ script. The following script for the Batch-LVQ1 must be constructed a bit differently; nonetheless its relation to the LVQ1 script is obvious. The most salient difference is that *no learning-rate parameter is needed.*

```
N = 4;
X = zeros(10000,2);
R = zeros(N*N,1);
S = zeros(10000,1);
for i = 1:10000
    if rand <.5
       S(i) = 1;
       X(i,:) = randn(1,2) - 2;
    else
```

```
        S(i) = - 1;
        X(i,:) = 2*randn(1,2) + 2;
    end
end

% SOM
smI = som_lininit(X,'msize',[N N],'lattice',...
      'hexa','shape','sheet');
smC = som_batchtrain(smI,X,'radius',[1 .01],...
    'trainlen',50);
sm = som_batchtrain(smC,X,'radius',[.01 .01],...
    'trainlen',200);
M = sm.codebook;
norms2 = sum(M.*M,2);

% Calibration of the SOM
for u = 1:10000
    for v = 1:N*N
        X1 = X(u,:)';
        Y = norms2 - 2*M*X1;
        c = min(Y);
        R(c) = R(c) + S(u);
    end
end
R = sign(R);
sX = zeros(N*N,2);
sS = zeros(N*N,1);
```

In the Batch-LVQ1 we use the same 10000 training vectors as in the LVQ1, but we divide them into 100 batches, with 100 input vectors in each. This time the batches are not identical as usual, but they have the same statistics. Nonetheless we do not expect that the algorithm would terminate exactly, since the batches are not identical. Anyway the batch computation assigns to each input vector in the batches the same weight, since there are no learning rates in this algorithm. This may cause some differences in the learning results, when compared with the LVQ1.

```
% Batch-LVQ1
for cycle = 1:100
    for u = 1:100

        % Determination of winner
        for v = 1:N*N
            X1 = X(100*(cycle-1)+u,:)';
            Y = norms2 - 2*M*X1;
            c = min(Y);
```

```
            sigma = S(100*(cycle-1)+u)*R(c);
            sX(c,:) = sX(c,:) + sigma*X(100*(cycle-1)+u,:);
            sS(c) = sS(c) + sigma;
        end
    end
    for v = 1:N*N
        if sS(v) > 0
           for el = 1:2
               M(v,el) = sX(v,el)/sS(v);
           end
        end
    end
end

voronoi(M(:,1),M(:,2))
```

The plot of the Batch-LVQ1 computation is shown in Fig. 50. The decision border looks similar as in LVQ1; there may be some deviations in the detailed locations of the model vectors.

30.2 The Learning Vector Quantization algorithm LVQ3

Actually there exists an LVQ2 algorithm, too, but since it is less robust than LVQ1 and LVQ3, it is skipped here.

In the LVQ3 algorithm we try to ensure that the model vectors $\mathbf{m}_i$ keep on approximating the class distributions even in very long training sequences; this has been a problem in LVQ1. We solve it by adding a third recursion to the algorithm:

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{m}_i(t)] \quad ,$$
$$\mathbf{m}_j(t+1) = \mathbf{m}_j(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{m}_j(t)] \quad ,$$

where $\mathbf{m}_i$ and $\mathbf{m}_j$ are the two closest model vectors to $\mathbf{x}$, whereupon $\mathbf{x}$ and $\mathbf{m}_j$ belong to the same class, while $\mathbf{x}$ and $\mathbf{m}_i$ belong to different classes, respectively. Furthermore,

$$\mathbf{m}_k(t+1) = \mathbf{m}_k(t) + \epsilon\alpha(t)[\mathbf{x}(t) - \mathbf{m}_k(t)] \quad ,$$

for $k \in \{i, j\}$, and $\mathbf{x}$, $\mathbf{m}_i$ and $\mathbf{m}_j$ belonging to the same class.

In a series of experiments, applicable values of ϵ between .1 and .5 were found.

Although we do not show any simulation results from the LVQ3, it may anyway be interesting to see how the script has been changed from that of LVQ1:

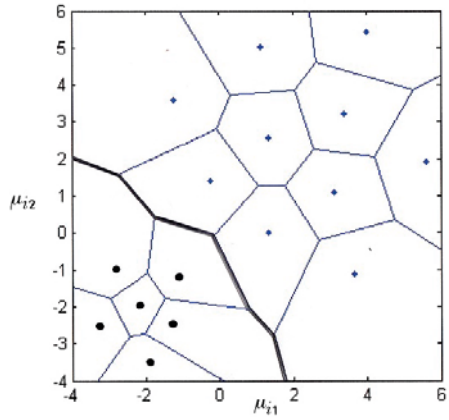


Fig. 50. This picture represents the model vectors of Batch-LVQ1. There are no big differences in the decision border (thick black line) when compared with the stepwise recursive LVQ1. The exact locations of the individual model vectors, however, may look somewhat different when compared with the LVQ1, but these locations are already different in various runs of both algorithms. Such differences are mainly due to the randomness of the training steps, and they do not affect the classification accuracy essentially: it is only the relative location of the pairs of model vectors in both classes that lie closest to the decision border, and that is controlled closely by both algorithms. Only the model vectors that are closest to the border define the location of the classification border and thus the classification accuracy. The differences in the model vectors are also due to different kinds of training steps in the two algorithms, because no time-variable learning rate is included in the Batch-LVQ1 algorithm.

```
N = 4;
X = zeros(10000,2);
S = zeros(10000,1);
R = zeros(N*N,1);
for i = 1:10000
    if rand <.5
       S(i) = 1;
       X(i,:) = randn(1,2) -2;
    else
       S(i) = - 1; %
       X(i,:) = 2*randn(1,2) + 2;
    end
end

% SOM
smI = som_lininit(X,'msize',[N N],'lattice',...
      'hexa','shape','sheet');
smC = som_batchtrain(smI,X,'radius',[1 .01],...
```

```
    'trainlen',30);
sm = som_batchtrain(smC,X,'radius',[.01 .01],...
    'trainlen',200);
M = sm.codebook;
norms2 = sum(M.*M,2);

% Calibration of SOM
for u = 1:10000
    for v = 1:N*N
        X1 = X(u,:)';
        Y = norms2 - 2*M*X1;
        c = min(Y);
        R(c) = R(c) + S(u);
    end
end
R = sign(R);
```

Now we must use the `sort` instruction instead of `min`:

```
% LVQ3
for u = 1:10000
    alpha = .01/(1 + u/10) + .00005;

    % Determination of winners
    for v = 1:N*N
        X1 = X(u,:)';
        Y = norms2 - 2*M*X1;
        [C,c] = sort(Y);

       % Updating
       cond = (S(u)==1 && R(c(1))==1 && R(c(1))==1||...
           S(u)== -1 && R(c(2))== -1 && R(c(2))== -1);
        e = .3;
        M(c(1),:) = M(c(1),:) + alpha*S(u)*R(c(1))*(X(u,:) -  ...
                            M(c(1),:));
        M(c(2),:) = M(c(2),:) + alpha*S(u)*R(c(2))*(X(u,:) - ...
                            M(c(2),:));
        M(c(1),:) = M(c(1),:) + e*alpha*cond*(X(u,:) - M(c(1),:));
        M(c(2),:) = M(c(2),:) + e*alpha*cond*(X(u,:) - M(c(2),:));
    end
end

% Voronoi tessellation
voronoi(M(:,1),M(:,2))
```

30.3 The "LVQ-SOM"

It may be interesting to find out that an LVQ algorithm and the SOM algorithm can be combined in a straightforward way. Consider the basic training algorithm of the SOM:

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + h_{ci}(t)[\mathbf{x}(t) - \mathbf{m}_i(t)] \quad .$$

The following supervised training scheme can be used if every training sample $\mathbf{x}(t)$ is known to belong to a particular class, and the $\mathbf{m}_i(t)$ have been assigned to respective classes, too. Like in the LVQ, if $\mathbf{x}(t)$ and $\mathbf{m}_i(t)$ belong to the same class, then in the "LVQ-SOM" $\mathbf{x}(t)$ is selected positive. On the other hand, if $\mathbf{x}(t)$ and $\mathbf{m}_i(t)$ belong to different classes, then the sign of $h_{ci}(t)$ is reversed. Notice that this sign-reversal rule is applied *individually to every* $\mathbf{m}_i(t)$ *in the topographic neighborhood of the "winner."*

It may be advisable to apply the "LVQ-SOM" scheme only after the unsupervised SOM phases, after the neighborhood has shrunk to its final value.

The script of the LVQ-SOM is left for an exercise to an advanced SOM programmer.

Difference between the "Supervised SOM" and the "LVQ-SOM". In the Supervised SOM discussed in Sec. 30 we concatenated the input vectors and the associated class vectors (weighted unit vectors), and then carried out *a normal SOM training procedure*. The class vectors increased the vectorial difference between the different classes, but no error corrections of the LVQ type were performed. If we look at Fig. 47, we may think that the basic SOM algorithm is directly trying to approximate the segments of class distributions, separated by the Bayesian borders. The mapping is still topographic, because the neighborhood function is trying to organize the models vectors topographically.

In the LVQ-SOM we make use of the neighborhood functions, too, which results in stronger training than in LVQ, because many model vectors are then updated in one training step, which increases their local correlation and statistical accuracy. On the other hand, the class information is not given in the data vectors but in the training algorithm. Although the LVQ-SOM seems more complex than the Supervised SOM, it is also more effective, and because it is trying to approximate the Bayes borders directly, its accuracy is better and theoretically justified. (We introduced the Supervised SOM mainly for historical reasons and also because it is easy to use.)

31 Optimization of a feature detector bank

Waveform analysis by LPC filters

Objective:

Especially in electro-acoustics, but also in other signals analyses, the waveform analysis is often based on Linear predictor coding (LPC) coefficients. *The LPC coefficients are light to compute, and there is a special function* `lpc` for it in MATLAB. Their efficacy, e.g., in speech coding and recognition is comparable with frequency analysis by the Fast Fourier Transform. In this section we shall discuss the optimization of an LPC filter bank by the SOM, eventually fine-tuned by LVQ.

Linear predictor coding (LPC) coefficients. Consider a sequence of scalar samples $x(t)$ that may represent, e.g., a waveform. Let t here stand for a *sampling instant*, an integer. We shall look for a linear approximation, called the *nth order autoregressive (AR) process*, which approximates the value $x(t)$ recursively as

```
x(t) = - a(2)*x(t-1) - a(3)*x(t-2) - ... - a(n+1)*x(t-n) .
```

Note that in this particular writing, which is used in the MATLAB, there are n terms on the right, but there are $n + 1$ coefficients: *the value of the first coefficient* $a(1)$ *is always equal to* 1, *and the numbering of the coefficients on the right-hand side starts with 2*. Notice also the minus signs on the right-hand side.

The coefficients `a` are called the *linear predictor coding (LPC) coefficients*. For their computation there is a special function

```
a = lpc(x,n)
```

in the MATLAB, where `a` is the vector of the coefficients `a`. An nth order process always produces $n + 1$ coefficients, namely,

$a(1), a(2), ..., a(n + 1)$, where always $a(1) = 1$.

When the waveform is produced by a physical system, such as the speech organs, the linear approximation is usually very good, provided that n is large enough (in speech analysis and synthesis, on the order of 20). The AR process has been used widely in signals analysis and especially in the digital analysis and synthesis of speech.

Since the LPC filter is linear, its coefficients can be computed easily for any given time-domain signal by minimizing the variance of the prediction errors. In MATLAB this can be made conveniently by the *Levinson-Durbin recursion* [29] [4].

An example of the performance of the LPC predictor is given in Fig. 51. One of the classic waveforms is the *relaxation of a second-order linear dynamic system*:

$$x(t) = e^{-At}\cos(Bt) \ .$$

The original waveform $x(t)$ has been defined for 30 steps of t, as shown in the top figure. However, only the steps 1 ... 15 have been taken into account in computing the LPC coefficients.

The sixth-order LPC coefficients are shown by the bar diagram in the central figure. Note that $a(4)$ and $a(5)$ are not visible, and $a(6)$ and $a(7)$ are very faint.

The 16 first samples of the waveform $y(t)$ in the bottom figure are copies of the waveform at the top. The rest of the bottom curve has been computed recursively: after the estimate of $y(16)$ has been computed, the estimate of $y(17)$ is computed on the basis of $y(2), y(3), ..., y(16)$, and so on. In the bottom figure, the waveform $y(t)$ is equal to that of $x(t)$ up to step 15, and steps 16 ... 30 represent the values y computed recursively on the basis of the LPC coefficients. All of the images in Fig. 51 have been produced by the following script:

```
len= 30;
x = zeros(1,len);
y = zeros(1,len);
A = .057;
B = .99;
for t = 1:len
    x(t) = exp(-A*t)*cos(B*t);
end
a = lpc(x,6);
for t = 1:15
    y(t)= x(t);
end
for t = 16:len
    y(t) = -a(2)*y(t-1) - a(3)*y(t-2) - a(4)*y(t-3)...
          - a(5)*y(t-4) - a(6)*y(t-5) - a(7)*y(t-6);
end
for pic = 1:3
    subplot(3,1,pic);
    if pic == 1
       plot(x(1,:));
    else if pic == 2
            bar(a(1,:));
        else
            plot(y(1,:));
        end
    end
end
```

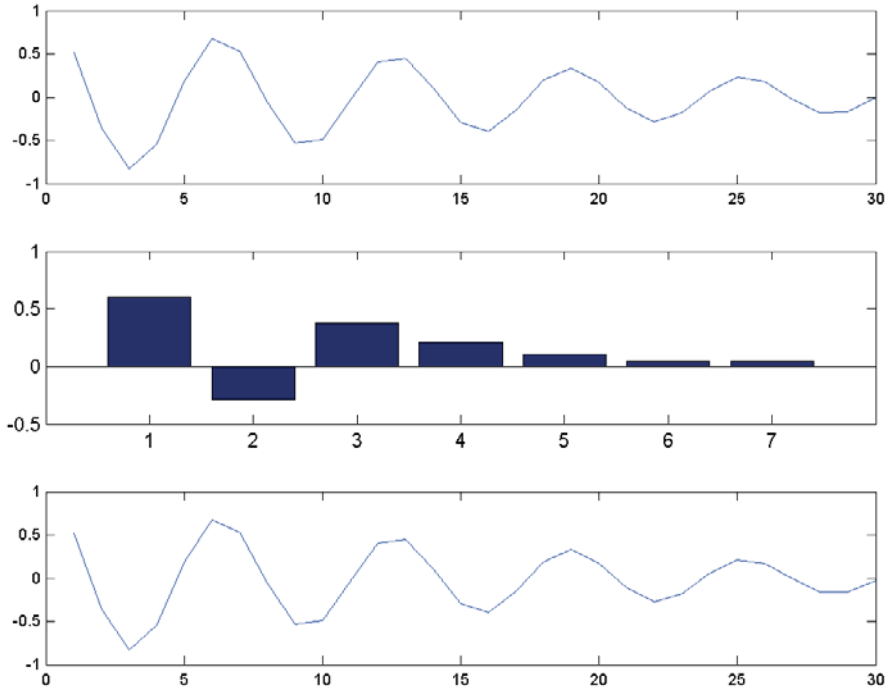


Fig. 51. Prediction of a waveform on the basis of the LPC coefficients. Top figure: $x(t)$. Middle figure: The LPC coefficients $a(k)$, which were computed from the waveform $x(t)$ from the interval $[1, 15]$. They are represented by a bar diagram. Note that in the MATLAB convention, the first coefficient is always equal to 1. Bottom figure: The waveform of $y(t)$. Up to step $t = 15$, $y(t)$ is a copy of $x(t)$, but after that ($t = 16...30$) the waveform has been *predicted* by the LPC coefficients.

31.1 The LPC-SOM

Now we are constructing an SOM for 25 sets of LPC coefficients. That is, the SOM operates like a *competitive filter bank* where the *winner* classifies the waveform.

For that purpose the SOM units should be *labeled*, for instance, by the symbols of *phonemes* that the segment of the speech waveform represents. When the unknown waveform selects the winner, it becomes *classified* at the same time.

In this tutorial example we do not deal with phonemes but with artificial waveforms, for the occurring variations of which we want to compute the representative SOM.

Training data. The coding starts with the definition of the training data. We shall pick up an example, in which the *distribution of the input signal patterns* is very simple, e.g., the signals are defined by only *two parameters, like the frequency and the attenuation of the waveform of Fig. 51.* (This is only an

illustrative example for programming, of course; the natural signals are much more multidimensional, but they may be described by a small number of LPC coefficients.)

When this waveform is restricted to, say, to the interval $[1, 15]$, we call this discrete-time temporal pattern an *episode*. One thousand of such episodes are generated by randomly selecting the parameters `A` and `B` as `A = .05 + .1*rand; B = .3 + 2*rand;` for the 1000 episodes we compute the LPC coefficients and use them as input data patterns for the SOM algorithm. Here we denote the input data used to train the SOM by `b`.

```
b = zeros(1000,7);
len= 30;
x = zeros(1,len);
y = zeros(1,len);
for epis = 1:1000
    A = .05 + .1*rand;
    B = .3 + 2*rand;
    for t = 1:len
        x(t) = exp(-A*t)*cos(B*t);
    end
    a = lpc(x,6);
    b(epis,:) = a(1,:);
end
smI = som_lininit(b,'msize', [5 5], 'lattice', ...
    'hexa', 'shape','sheet');
smC = som_batchtrain(smI, b, 'radius', [2 .5], ...
    'trainlen', 30,'neigh', 'gaussian');
sm = som_batchtrain(smC, b, 'radius', [.5 .5], ...
    'trainlen', 30,'neigh', 'gaussian');
M = sm.codebook;
for pic = 1:25
    subplot(5,5,pic)
    bar(M(pic,:));
end
```

Fig. 52 is a different type of an SOM picture that represents the LPC coefficients at the various SOM locations as bar diagrams produced by the MATLAB graphics.

The central motive in producing an LPC-SOM is that one is then able to *classify* input signals. Since the set of filters in the LPC-SOM is *optimized* for the statistics of waveforms used in the formation of the SOM, the computing resources are optimized for that task. Naturally the same kind of optimization would have been achieved by a filter bank based on the classical k-means clustering, too, but the SOM has two advantages over it: first, its training is more robust due to the local smoothing effects in the neighborhoods, and second, it is easy and quick to monitor the performance by the SOM display; we have plenty

of experience of this from our "Phonetic Typewriter" [36]. Further, *supervised tuning* of the LPC filters is possible, as will be mentioned below.

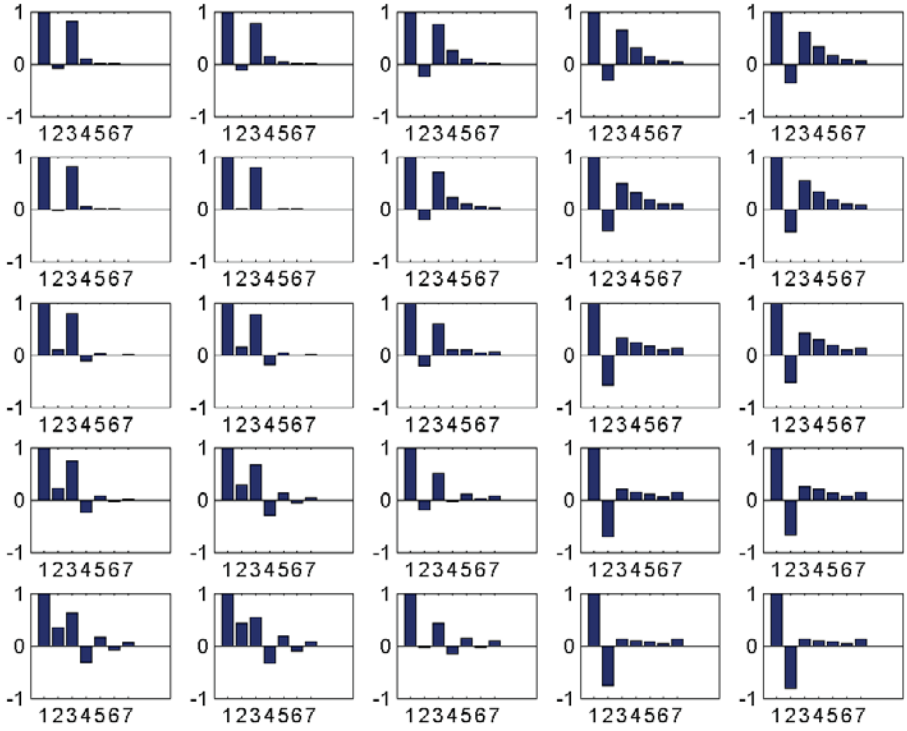


Fig. 52. 25 subplots, together representing the seven LPC coefficients in the various SOM locations. Note that in the MATLAB convention, the first coefficient is always equal to 1.

Actually Fig. 52 however, is not what was originally meant. In a genuine Predictor SOM [50], the *winners* were supposed to be defined by the *smallest estimation errors* when comparing the time-domain input waveform with the waveforms produced by the LPC filters at each node of the SOM. At least in the present example, the problem seemed to be that when the training of the SOM models was based on the minimization of the estimation errors based on the stochastic-gradient method, like suggested in [50], the self-organizing power remained rather weak and did not easily produce well-ordered maps. Perhaps a longer waveform to find the statistically best matches in training might have been needed. Then the derivation of the gradient-descend formula would have become rather complicated.

In Fig. 53 we give the original results from [50], in order to show what kinds of LPC filters are formed for real speech waveforms.

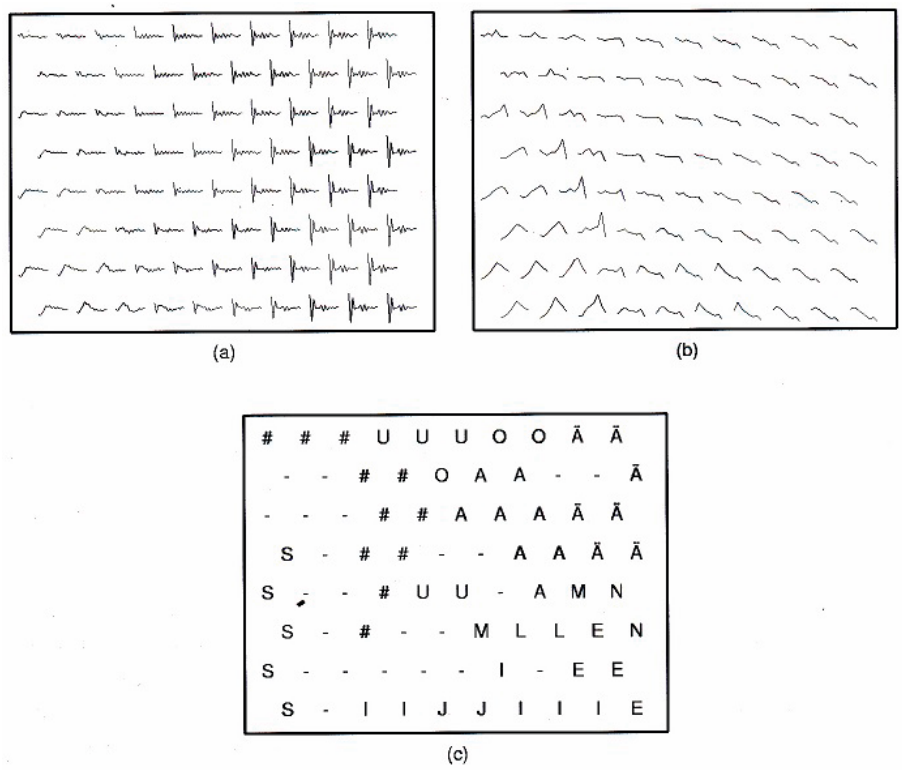


Fig. 53. The 16th-order LPC-coefficient vectors have been organized in this SOM. (a) The ordinate values of the plots drawn at each node of the map represent the LPC coefficients : $a_{i,1}$ on the left, $a_{i,16}$ on the right. (b) Frequency responses computed from the LPC coefficients are shown as a plot in each map position: low frequencies (300 Hz) on the left, high frequencies (8 kHz) on the right. (c) Labeling of the nodes by some phonemic symbols. The capital letters correspond to the Finnish phonemes, which sound almost similar as in Latin. (From [50].)

31.2 Supervised tuning of the LPC-SOM

If the purpose is to create an LPC-coefficient filter bank for pattern recognition or classification, I may suggest that the final class separation were improved by *supervised training, for instance by the Supervised SOM like in [36], or by LVQ-SOM*. To that end, we must have available *a sufficient number of samples from waveforms that are known to belong to a finite number of classes*, such as speech waveforms having been extracted from utterances of known phonemes.

32 How to make large SOMs

Some time ago I asked a group why they had started with one of the largest and most difficult SOM problems, without having experience from simpler ones. They answered: "But Professor Kohonen, don't you know that there is nowadays a hard competition in the world?"

Objective:

Sooner or later you might want to make really big SOMs. The SOM Toolbox functions have a limited memory space, and it is neither quite sure that the MATLAB is then the right programming system. Maybe it is advisable to program the problem, e.g., in C++, Java, or R. Nonetheless, before you start programming large problems, it is necessary to have an idea of what to program. One can increase the dimensions of the problem with simple tricks, and one can change the representation format. Here are some pieces of advice.

Whatever you intend to do in developing the SOM, it is necessary to *imagine* what the algorithms are doing. This you will learn best with simple toy problems. For this reason I have started with very simple examples and then increased the complexity of tasks gradually and progressively. It is highly recommendable that you run the simpler exemplary scripts first, unless you already have a lot of experience from the SOMs. The really large applications could not be described here; it is necessary to refer to the original scientific publications.

Multiplying the number of nodes in the SOM. The simplest method for to save computing time in constructing large SOMs is to compute first a smaller SOM array, and then to *multiply* its size. The small array is first let to converge, and then new, *interstitial nodes* are introduced between the old ones. In this way the size of the SOM is made roughly fourfold: cf. how this trick was applied in Subsec. 29.8. Since the converged model vectors are supposed to change smoothly, for the initial values of the interstitial models we can take *averages* of the neighboring old nodes. Then this larger SOM is already rather well initialized and the time for its convergence in the subsequent training phases becomes shorter. This enlargement of the SOM can be continued an arbitrary number of times, if after each multiplication the enlarged SOM is let to converge carefully. We have used this method in constructing the largest SOM known so far: the SOM of about 7 million patent abstracts, which were mapped onto an approximately one-million-node SOM.

Selection of the neighborhood function. The Gaussian neighborhood function yields better SOMs than the `bubble` form, or neighborhood set, be-

cause one can use a continuous range of values for the radius of the neighborhood. For very large maps the fact that the Gaussian function extends to the borders of the map makes the updating computations slow. A compromise is the `cutgauss` form of the neighborhood, or the Gausssian kernel whose flanks are cut at the radius corresponding to the standard deviation. The amount of computation is of the same order of magnitude as with the `bubble` neighborhood function, while the radius can be set at an arbitrary (real) value, e.g., smaller than unity.

Shortcut winner search. An effective method to reduce the number of winner-search operations is to *estimate the winner location* on the basis of previous searches. A prerequisite for this is that the batch of training vectors is the same at each iteration cycle.

Assume that we are somewhere in the middle of an iterative training process, whereupon the last winner corresponding to the training vector has been determined at an earlier training cycle. If the training vectors are expressed as a linear table, an *address pointer* to the tentative (earlier) winner location can be stored (and updated) with each training vector (cf. Fig. 54).

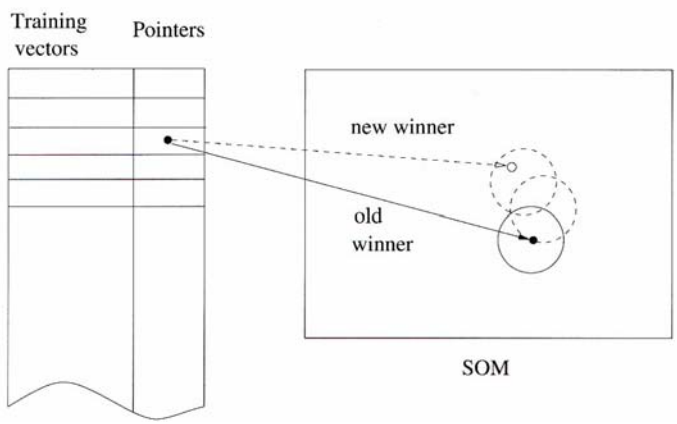


Fig. 54. Finding the new winner in the vicinity of the old one, whereupon the old winner is directly located by a pointer associated with the training vector. After the new winner is found, the old pointer is updated.

Assume further that the SOM is already smoothly ordered, although not yet asymptotically stable. This is the situation, e.g., during the fine-tuning phase, which may be long in large maps. However, the size of the neighborhood is constant and small. Then the new winner is found at or in the vicinity of the old one, and in searching for the new winner, it will suffice to carry out a local search in the neighborhood of the node located by the pointer. It may suffice to find a new winner whose matching with the training vector is better than for

the old winner. This will be significantly faster than performing an exhaustive global search over the whole SOM.

The search can first be made in the immediate surround of the old winner, and if the best match is found at its edge, searching is continued in the larger surround, until the best match is in the inside of the searching domain. After the new winner location has been identified, the associated pointer is is replaced by th address of the new winner location.

In a benchmarking study we found that the speedup factor in winner search due to the shortcut search could be on the order of a few dozen.

Speedup in sparse matrices. In the winner search, whether a dot-product SOM or a usual SOM with the simplified winner search, the most time-consuming task is to form the matrix-vector product `M*X`. There are cases, especially with word histograms, where the input vectors `X` are *sparse*, i.e., they contain plenty of zero elements. In the matching for the winner these zeros can be skipped. It is possible to tabulate the indices of the non-zero components of each input vector and use only those components in computing the matches.

Saving memory by reducing representation accuracy. In very large SOMs with high input dimensionality, the memory requirements can be reduced significantly by using a coarse numerical accuracy of the vectors and matrices, whereby even a very large SOM can be kept in the main memory of the computer system. Notice that since the models are rather regularly spaced in the data space, the numerical accuracy for the location of the winner can be incredibly low. For instance, in our largest SOM we had only 8 -bit accuracy in the representation of the vector and matrix components. If the dimensionality of the data vectors is large, the statistical accuracy with such quantized-value components may be quite sufficient in in the matching.

For further details of possible speedup and memory saving, cf. [39].

References

[1] Alhoniemi, E. (2000). Analysis of pulping data using the self-organizing map. *TAPPI Journal*, 83(7): 66.

[2] Allinson, N., Yin, H., Allinson, L., & Slack, J. (Eds.) (2001). *Advances in Self-Organizing Maps*. London, UK: Springer.

[3] Anderberg , M. (1973). *Cluster Analysis for Applications*. New York, NY: Academic.

[4] Bäckström, T. (2004). "2.2. LevinsonDurbin Recursion." *Linear Predictive Modeling of Speech Constraints and Line Spectrum Pair Decomposition*. Doctoral Thesis. Report no. 71 / Helsinki University of Technology, Laboratory of Acoustics and Audio Signal Processing. Espoo, Finland.

[5] Bairoch, A. & Apweiler, R. (1999). The SWISS-PROT protein sequence data bank and its supplement. TrEMBL in 1999. Nucleic Acids Res., 27:49-54.

[6] Bishop, C.M. , Svensen, M. & Williams, C.K.I. (1998). GTM: The generative topographic mapping. *Neural Comp.*, 10: 215-234.

[7] Carlson, E. (1991). Self-organizing feature maps for appraisal of land use of shore parcels. In Kohonen, T., Mäkisara, K., Simula, O. and Kangas, J., eds., *Artificial Neural Networks*, vol. II, pp. 1309-1312. Amsterdam, Netherlands: Noth-Holland.

[8] Cheng, Y. (1997). Convergence and ordering of Kohonen's Batch map. *Neural Computation*, 9: 1667-1676.

[9] Cole, R.A. , Muthusami, Y. , & Fanty, M.A. (1994). The ISOLET spoken letter database. Technical Report 90-004. Computer Science Department, Oregon Graduate Institute.

[10] Cottrell, M. & Fort, J.C. (1987). Étude d'un processus d'auto-organization. *Ann. Inst. Henri Poincaré*, 23: 1-20.

[11] Cottrell, M. , Fort, J.C., & Pagés, G. (1997). Theoretical aspects of the SOM algorithm. In *Proc. WSOM 97, Workshop on Self-Organizing Maps* (pp. 246-267). Helsinki University of Technology, Neural Networks Research Centre, Espoo, Finland, pp. 246-267.

[12] Cottrell, M. , Fort, J.C., & Pagés, G. (1998). Theoretical aspects of the SOM algorithm. *Neurocomputing*, 21(1): 119-138.

[13] Deboeck, G. & Kohonen,T. (1998). *Visual Explorations in Finance with Self-Organizing Maps*. London, UK: Springer. pp. 246-267.

[14] Deerwester, S. , Dumais, S., Furnas, G., & Landauer, K. (1990). Indexing by latent semantic analysis.*J. Am. Soc. Inform. Sci.*, 41: 391-407.

[15] Dersch, D. & Tavan, P. (1995). Asymptotic level density in topological feature maps. *IEEE Trans. on Neural Networks*, 6(1): 230-236.

[16] Estévex, P., Príncipe, J. & Zegers, P. (2012). *Advances in Self-Organizing Maps: Springer series in Advances in Intelligent Systems and Computing 198*. Heidelberg, New York, Dordrect, London: Springer.

[17] Forgy, E.W. (1965). Cluster analysis of multivariate data: efficiency vs. interpretability of classifications. *Biometrics*, 21: 768, abstract.

[18] Fritzke, B. (1994). Growing cell structures - a self-organizing network for unsupervised and supervised learning. *Neural Networks*, 7: 1441-1460.

[19] Gersho, A. (1979). On the structure of vector quantizers. *IEEE Trans. Inform. Theory*, IT-25: 373-380.

[20] Gray, R.M. (1984). Vector quantization, *IEEE ASSP Mag.*, 1: 4-29.

[21] Hammer, B., Micheli, A., Sperduti, A., & Strickert, M. (2004). Recursive self-organizing network models, *Neural Networks*. 17: 1061-1085.

[22] Hartigan, J. (1975). *Clustering Algorithms*. New York, NY: Wiley.

[23] Heskes, T.M. & Kappen, B. (1993). Error potential for self-organization, in *Proc. ICNN'93, Int. Conf. on Neural Networks, Vol. III* (pp. 1219-1223). Piscataway, NJ: IEEE Service Center.

[24] Honkela, T., Pulkki, V., & Kohonen, T. (1995). Contextual relations of words in Grimm tales, analyzed by self-organizing maps. In F. Fogelman-Souli & P. Gallinari, (Eds.), Proceedings of International Conference on Artificial Neural Networks, ICANN95, Vol II, (pp. 3-7). Nanterre, France: EC2.

[25] Honkela, T., Kaski, S., Lagus, K. & Kohonen, T. (1996). Exploration of full-text databases with self-organizing maps. In *Proc. ICNN'96, Int. Conf. on Neural Networks* (vol.1, pp. 56-61). Piscataway, N.J.: IEEE Service Center.

[26] Honkela, T., Kaski, S., Lagus, K. & Kohonen, T. (1997). WEBSOM - self-organizing maps for document collections. In *Proc. WSOM'97, Workshop on Self-Organizing Maps* (pp. 310-315). Espoo, Finland: Helsinki University of Technology, Neural Networks Research Centre.

[27] Honkela, T., Kaski, S., Kohonen, T. & Lagus, K. (1998). Self-organizing maps of very large document collections: Justification for the WEBSOM method. In Balderjahn, I., Mathar, R. & Schader, M., eds., *Classification, Data Analysis, and Data Highways* (pp.245-252). Berlin: Springer.

[28] Honkela, T., Lagus, K. & Kaski, S. (1998).Self-organizing maps of large document collections. In G. Deboeck and T. Kohonen, eds., *Visual Explorations in Finance with Self-Organizing Maps* (pp.168-178). London: Springer.

[29] Jackson, L. B. Digital Filters and Signal Processing. 2nd Edition. Boston: Kluwer Academic Publishers, 1989, pp. 255257.

[30] Jain, A.K. & Dubes, R.C. (1988). Algorithms for Clustering of Data. Englewood Cliffs, NJ: Prentice-Hall.

[31] Kaski, S., Honkela, T., Lagus, K., & Kohonen, T. (1998). WEBSOM - Self-organizing maps of document collections. *Neurocomputing*, 21(1): 101-117.

[32] Kaski, S., Kangas, J., & Kohonen, T. (1998). Bibliography of Self-Organizing Map (SOM) papers: 1981-1997, *Neural Computing Surveys*, 1: 1-176. Available in electronic form at http://www.cis.hut.fi/research/som-bibl/vol1_4.pdf

[33] Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biol. Cyb.*, 43: 59-69.

[34] Kohonen, T. (1982). Clustering, taxonomy, and topological maps of patterns. In *Proceedings of the Sixth Int. Conf. on Pattern Recognition* (pp. 114-128). Washington, D.C.: IEEE Computer Soc. Press.

[35] Kohonen, T. (1985). Median strings. *Pattern Recognition Letters*, 3: 309-313.

[36] Kohonen, T. (1988).The "neural" phonetic typewriter. *Computer* 21: 11-22.

[37] Kohonen, T. (1989). *Self-Organization and Associative Memory.* 3rd ed. Berlin-Heidelberg-, Germany: Springer.

[38] Kohonen, T. (1990). The Self-Organizing Map. *Proc. IEEE* 78: 1464-1480.

[39] Kohonen, T. (2001). *Self-Organizing Maps.* 3rd ed. Berlin-Heidelberg, Germany: Springer.

[40] Kohonen,T., Oja, E., Simula, O., Visa, A., & Kangas, J. (1996). Engineering applications of the Self-Organizing Map. *Proc. IEEE*, 84: 1358-1384.

[41] Kohonen,T., Hynninen, J., Kangas, J., & Laaksonen, J. (1996). *The Self-Organizing Map Program Package, Report A31*, Espoo, Finland: Helsinki University of Technology, Laboratory of Computer and Information Science.

[42] Kohonen, T. & Somervuo, P. (2002). How to make large self-organizing maps for non-vectorial data. *Neural Networks.* 15: 945-952.

[43] Kohonen, T. (2007). Description of input patterns by linear mixtures of SOM models. In *WSOM 2007 CD-ROM Proceedings*, Bielefeld, Germany: Bielefeld University. Also available at http://biecoll.ub-bielefeld.de.

[44] Kohonen, T. (2008). Data management by self-organizing maps. In J.M. Zurada, G.G. Yen, & J. Wang (Eds.), *Computational Intelligence: Research Frontiers* (pp. 309-332). Berlin, Heidelberg, New York: Springer.

[45] Kohonen, T. & Xing, H. (2011). Contextually self-organized maps of Chinese words. In J. Laaksonen & T. Honkela, *Advances in Self-Organizing Maps* (pp. 16-29). Berlin, Heidelberg: Springer.

[46] Kohonen, T. (2013) Essentials of the self-organizing map. *Neural Networks*. 37: 5265.

[47] Kraaijveld, M.A., Mao, J. & Jain, A.K. (1992).A non-linear projection method based on Kohonen's topology preserving maps. *Proc. 11ICPR, Int. Conf. on Pattern Recognition*, (pp 41-45)., Los Alamitos, CA: IEEE Comp. Soc. Press.

[48] Kruskal, J.B. & Wish, M. (1978). *Multidimensional Scaling*, Sage University Paper Series on Quantitative Applications in the Social Sciences No. 07-011, Newbury Park, CA: Sage Publications.

[49] Laaksonen, J. & Honkela, T. (Eds.) (2011). *Advances in Self-Organizing Maps, Springer Lecture Notes in Computer Science 6731*. Berlin, Heidelberg.

[50] Lampinen, J. & Oja. E. (1989). In *Proc. 6 SCIA, Scand. Conf. on Image Analysis* (p. 120), ed. by M. Pietikäinen, J. Röning. Suomen Hahmontunnistustutkimuyksen Seura, r.y.: Helsinki, Finland.

[51] Lawson, C.L., & Hanson, R.J. (1974). *Solving Least-Squares Problems*. Englewood Cliffs, NJ: Prentice-Hall.

[52] Levenshtein, V. (1966) Sov.Phys. Dokl. 10: 707.

[53] Lewis, D.D., Yang, Y., Rose, T.G., & Li, T. (2004). RCV1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.* 5: 361-397.

[54] Lin, X., Soergel, D. & Marchionini, G. (1991). Self-organizing semantic map for information retrieval. In *Proc. 14th Ann. Int. ACM/SIGIR Conf. on R & D In Information Retrieval* (pp.262-269).

[55] Lincoff, G.H. (1981). *The Audubon Society Field Guide to North American Mushrooms*. New York: Alfred A. Knopf

[56] Linde, Y., Buzo, A., & Gray, R.M. (1980). An algorithm for vector quantization. *IEEE Trans. Communications*, COM-28: 84-95.

[57] Makhoul, J., Roucos, S., & Gis, H. (1985). Vector quantization in speech coding. *Proc. IEEE*, PROC-73: 1551-1588.

[58] Manning, C.D. & Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press.

[59] Merkl, D. & Tjoa, A.M. (1994). The representation of semantic similarity between documents by using maps: Application of an artificial neural network to organize software libraries. In *Proc. FID'94, General Assembly Conf. and Congress of the Int. Federation for Information and Documentation*.

[60] Miikkulainen, R. (1993). *Subsymbolic Natural Language Processing: An Integrated Model of Scripts. Lexicon, and Memory*, Cambridge, MA: MIT Press.

[61] Naim, A., Ratnatunga, K.U., & Griffiths, R.E. (1997). Galaxy morphology without classification; Self-organizing maps, *Astrophys. J. Suppl. Series*. 111: 357-367.

[62] Obermayer, K. & Sejnowski, T. (Eds.) (2001). *Self-Organizing Map Formation: Foundations of Neural Computation*. Cambridge, MA: MIT Press.

[63] Oja, E. & Kaski, S. (1999). *Kohonen Maps*. Amsterdam, Netherlands: Elsevier.

[64] Oja, M., Kaski, S., & Kohonen, T. (2003). Bibliography of Self-Organizing Map (SOM) papers; 1998-2001 addendum. *Neural Computing Surveys*, 3: 1-156. Available in electronic form at http://www.cis.hut.fi/research/som-bibl/NCS_vol3_1.pdf

[65] Oja, M., Somervuo, P., Kaski, S., & Kohonen, T. (2003). Clustering of human endogeneous retrovirus sequences with median self-organizing map. In *Proceedings of the WSOM'03, Workshop on Self-Organizing Maps, Hibikino, Japan*.

[66] Okuda,T., Tanaka, E. & Kasai, T., A method for the correction of garbled words based on the Levenshtein metric. IEEE Trans. C-25, 172-178, 1976.

[67] Pearson, W. & Lipman, D. (1988). Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sc. (USA*, 85: 2444-2448.

[68] Pearson, W. (1999). The FASTA program package.

[69] Pöllä, M., Honkela, T., & Kohonen, T. (2009). Bibliography of SOM papers, available at http://users.ics.tkk.fi/tho/online-papers/TKK-ICS-R23.pdf.

[70] Príncipe, J. & Miikkulainen, R. (Eds.). Springer Series in Advances in Self-Organizing Maps, LNCS 5629. Berlin, Heidelberg, New York: Springer, 2009.

[71] Ritter, H. & Kohonen, T. (1989). Self-organizing semantic maps, *Biol. Cyb.*, 61: 241-254.

[72] Robbins, H. & Monro, S. (1951). *Ann. Math. Statist*, 22: 400.

[73] Ritter, H., Martinetz, T., & Schulten, K. (1992). *Neural Computation and Self-Organizing Maps: An Introduction*. Reading, MA: Addison-Wesley.

[74] Ritter, H. & Schulten, K. (1986). On the stationary state of Kohonen's self-organizing sensory mapping. *Biol. Cyb.*, 54: 99-106.

[75] Salton, G. & McGill, M.J. (1983). *Introduction to Modern Information Retrieval*. New York, NY: McGraw-Hill.

[76] Sammon, Jr., J.W. (1969). A nonlinear mapping for data structure analysis. *IEEE Trans. Computers*, C-18: 401-409.

[77] Scholtes, J.C. (1991). Unsupervised context learning in natural language processing. In *Proc. IJCNN'91, Int. Conf. on Neural Networks, Vol.I* (pp. 107-112). Piscataway, NJ: IEEE Service Center.

[78] Seiffert, U. & Jain, L.C. (Eds.) (2002). *Self-Organizing Neural Networks: Recent Advances and Applications*. Heidelberg, Germany: Physica-Verlag.

[79] Shepard, R.N. (1962). The analysis of proximities: multidimensional scaling with an unknown distance function. Psychometrica, 27:125-246.

[80] BLOSSOM Team (Japan) (2005). SOM Japan Co., Ltd. *The BLOSSOM software package*, http://www.somj. com/.

[81] SOM_PAK Team (1990). http://www.cis.hut.fi/research/som_pak/som_doc.ps,

[82] SOM Toolbox Team (1999). http://www.cis.hut.fi/projects/somtoolbox/documentation/ ; http://www.cis.hut.fi/projects/somtoolbox/package/papers/techrep.pdf

[83] Sun, H.L., Sun, D.J., Huang, J.P., Li. D.J., & Xing, H.B. (1996). Corpus for Modern Chinese Research. In Luo, Z.S., & Yuan, Y.L.(eds.), Studies of the Chinese language and characters in the era of computers (pp. 283-294). Beijing, China: Tsinghua University Press.

[84] Tokutaka, H., Kishida, S., & Fujimura, K. (1999). *Application of Self-Organizing Maps - Two-Dimensional Visualization of ulti-Dimensional Information* (in Japanese),Tokyo, Japan: Kaibundo.40

[85] Tokutaka, H., Ookita, M., & Fujimura, K. (2007). *SOM and the Applications* (in Japanese),Tokyo, Japan: Springer Japan.

[86] Tryon, R. & Bailey, D. (1973). *Cluster Analysis*. New York, NY: McGraw-Hill.

[87] Ultsch, A. (1993). Self-organizing neural networks for visualization and classification In O. Opitz, B. Lausen, & R. Klar (Eds.), *Information and Classification* (pp. 307-313). Berlin, Germany: Springer, pp. 307-313.

[88] Van Hulle, M. (2000). *Faithful Representations and Topographic Maps: From Distortion- to Information-Based Self-Organization*. New York, NY: Wiley.

[89] Vesanto, J., Himberg, J. Alhoniemi, E., & Parhankangas, J. (1999). Self-organizing map in Matlab: the SOM Toolbox, in *Proc. Matlab DSP Conference, Nov. 16-17, Espoo, Finland*, pp. 35-40.

[90] Vesanto, J., Alhoniemi, E., Himberg, J., Kiviluoto, K. & Parviainen, J. (1999). Self-organizing map for data mining in MATLAB: the SOM Toolbox, *Simulation News Europe*, 25:54, March1999.

[91] Vesanto, J., Himberg, J., Alhoniemi, E., & Parhankangas, J. (2000). SOM Toolbox for Matlab 5, Technical Report A57, Helsinki University of Technology, Neural Networks Research Centre.

[92] Villman, T., Schleif, F.-M. & Kaden, M. (2014). *Advances in Self-Organizing Maps and Learning Vector Quantization: Proceedings of WSOM 2014, Workshop on Self-Organizing Maps*. Berlin, Germany: Springer.

[93] Voronoi, G. (1907). Nouvelles applications des paramétres continus á la théorie des formes quadratiques, *J. Reine und Angew. Math.*, 133: 97-178.

[94] WSOM 1997 Program Committee (Eds.) (1997). *Proc. of the WSOM97, Proceedings of the Workshop on Self-Organizing Maps* . Espoo, Finland: Helsinki University of Technology, Neural Networks Research Centre.

[95] WSOM 2003 Program Committee (Eds.) (2003). *Proc. of the WSOM07, Proceedings of the Workshop on Self-Organizing Maps, Abstracts*. Kitakyushu, Japan: Kyushu Institute of Technology.

[96] WSOM 2005 Program Committee (Eds.) (2005). *Proc. of the WSOM05, Proceedings of the Workshop on Self-Organizing Maps*. Paris, France: SAMOS-MATISSE, Université Paris 1.

[97] WSOM 2007 Program Committee (Eds.) (2007). *Proc. of the WSOM07, Proceedings of the Workshop on Self-Organizing Maps (CD only)*. Bielefeld, Germany: Bielefeld University.

Index

The famous "Self-Organizing Map" (SOM) data-analysis algorithm developed by Professor Teuvo Kohonen has resulted in thousands of applications in science and technology. This book is the first-ever practical introduction to SOM programming, especially targeted to newcomers in the field. With several examples and in a clear detailed way, Prof. Kohonen here explains how various data-analysis problems can be approached with SOM analysis, what preprocessing steps are needed, and how the scripts of the SOM algorithm can be encoded. The text is particularly suitable for self-studying and as a collateral material for exercises in data mining courses.

Unigrafia Oy
Helsinki 2014

ISBN 978-952-60-3678-6