SOM_PAK: The Self-Organizing Map Program Package

Teuvo Kohonen, Jussi Hynninen, Jari Kangas, and Jorma Laaksonen

Helsinki University of Technology
Faculty of Information Technology
Laboratory of Computer and Information Science **Report A31**

Otaniemi 1996

SOM_PAK: The Self-Organizing Map Program Package

Teuvo Kohonen, Jussi Hynninen,
Jari Kangas, and Jorma Laaksonen

Helsinki University of Technology
Faculty of Information Technology
Laboratory of Computer and Information Science
Rakentajanaukio 2 C, SF-02150 Espoo, FINLAND

Report A31
January 1996

ISBN 951-22-2947-1
ISSN 0783-7445
TKK OFFSET

SOM_PAK: The Self-Organizing Map Program Package

Teuvo Kohonen, Jussi Hynninen,
Jari Kangas, and Jorma Laaksonen

SOM Programming Team of the
Helsinki University of Technology
Laboratory of Computer and Information Science
Rakentajanaukio 2 C, SF-02150 Espoo
FINLAND

Abstract: The Self-Organizing Map (SOM) represents the result of a vector quantization algorithm that places a number of reference or codebook vectors into a high-dimensional input data space to approximate to its data sets in an ordered fashion. The SOM_PAK program package contains all programs necessary for the correct application of the Self-Organizing Map algorithm in the visualization of complex experimental data. The first version 1.0 of this program package was published in 1992 and since then the package has been updated regularly to include latest improvements in the SOM implementations. This report that contains the last documentation was prepared for bibliographical purposes.

Contents

1 General

This software package contains all programs necessary for the correct application of the Self-Organizing Map (SOM) algorithm [Kohonen 1989][Kohonen 1990] [Kohonen 1995] in the **visualization of complex experimental data**.

NEW BOOK:

Complete description, with over 1500 literature references, of the SOM and LVQ (Learning Vector Quantization) algorithms can be found in the recently published book Kohonen: Self-Organizing Maps (Springer Series in Information Sciences, Vol 30, 1995). 362 pp.

The Self-Organizing Map represents the result of a vector quantization algorithm that places a number of reference or codebook vectors into a high-dimensional input data space to approximate to its data sets **in an ordered fashion**. When local-order relations are defined between the reference vectors, the relative values of the latter are made to depend on each other as if their neighboring values would lie along an "elastic surface". By means of the self-organizing algorithm, this "surface" becomes defined as a kind of nonlinear regression of the reference vectors through the data points. A mapping from a high-dimensional data space $\Re^n$ onto, say, a two-dimensional lattice of points is thereby also defined. Such a mapping can effectively be used to **visualize** metric ordering relations of input samples. In practice, the mapping is obtained as an asymptotic state in a learning process.

A typical application of this kind of SOM is in the analysis of complex experimental vectorial data such as process states, where the data elements may even be related to each other in a highly nonlinear fashion.

The process in which the SOM is formed is an unsupervised learning process. Like any unsupervised classification method, it may also be used to find clusters in the input data, and to identify an unknown data vector with one of the clusters. On the other hand, if the data are *a priori* known to fall in a finite number of classes, identification of an unknown data vector would optimally be done by some supervised learning algorithm, say, the Learning Vector Quantization (LVQ), which is related to the SOM. The corresponding software package LVQ_PAK is available at the same address as this package.

The present program package is thus not intended for optimal classification of data, but mainly for their interactive monitoring.

NOTE: This program package is copyrighted in the sense that it may be used freely for scientific purposes. However, the package as a whole, or parts thereof, cannot be included or used in any commercial application without written permission granted by its producents. No programs contained in this package may be copied for commercial distribution.

This program package is distributed in the hope that it will be useful, but **without any warranty**. No author or distributor accepts responsibility to anyone for the consequences of using it or for whether it serves any particular purpose or works at all.

2 The principle of the SOM

There exist many versions of the SOM. The basic philosophy, however, is very simple and already effective as such, and has been implemented by the procedures contained in this package.

The SOM here defines a mapping from the input data space $\Re^n$ onto a regular two-dimensional array of nodes. With every node i, a parametric reference vector $m_i \in \Re^n$ is associated. The lattice type of the array can be defined as rectangular or hexagonal in this package; the latter is more effective for visual display. An input vector $x \in \Re^n$ is compared with the m_i, and the best match is defined as "response": the input is thus mapped onto this location.

The array and the location of the response (image of input) on it are supposed to be presented as a graphic display. For a more insightful analysis, each component plane of the array (the numerical values of the corresponding components of the m_i vectors) may also be displayed separately in the same format as the array, using a gray scale to illustrate the values of the components. Although this program package is intended for any UNIX or MS-DOS computer, some problems are caused by different standards used in the graphics systems. Therefore we give the output data as numerical arrays, and expect that the users can easily convert them into graphic displays. Two programs are provided to convert the resulting map files to encapsulated postscript image format (see Section 7.4).

One might say that the SOM is a "nonlinear projection" of the probability density function of the high-dimensional input data onto the two-dimensional display. Let $x \in \Re^n$ be an input data vector. It may be compared with all the m_i in any metric; in practical applications, the smallest of the Euclidean distances $||x - m_i||$ is usually made to define the best-matching node, signified by the subscript c:

$$\begin{aligned} ||x - m_c|| &= \min_i \{||x - m_i||\} \ , \qquad \text{or} \\ c &= \arg\min_i \{||x - m_i||\} \ . \end{aligned} \tag{1}$$

Thus x is mapped onto the node c relative to the parameter values m_i.

An "optimal" mapping would be one that maps the probability density function $p(x)$ in the most "faithful" fashion, trying to preserve at least the local structures of $p(x)$. (You might think of $p(x)$ as a flower that is pressed!) Definition of such m_i values, however, is far from trivial; a number of people have tried to define them as optima of some objective (energy) function (see e.g. [Ritter et al. 1988], [Luttrell 1989], [Kohonen 1991], and [Erwin et al. 1992]). As the existence of a satisfactory definition is still unclear, we have restricted ourselves in this package to the stochastic-approximation-type derivation [Kohonen 1991] that defines the original form of the SOM learning procedure.

During learning, those nodes that are topographically close in the array up to a certain distance will activate each other to learn from the same input. Without mathematical proof we state that useful values of the m_i can be found as convergence limits of the following learning process, whereby the initial values of the $m_i(0)$ can be arbitrary, e.g., random:

$$m_i(t+1) = m_i(t) + h_{ci}(t)[x(t) - m_i(t)] \ , \tag{2}$$

where t is an integer, the discrete-time coordinate, and $h_{ci}(t)$ is the so-called **neighborhood kernel**; it is a function defined over the lattice points. Usually $h_{ci}(t) = h(||r_c - r_i||, t)$, where $r_c \in \Re^2$ and $r_i \in \Re^2$ are the radius vectors of nodes c and i, respectively, in the array. With increasing $||r_c - r_i||$, $h_{ci} \to 0$. The average width and form of h_{ci} defines the "stiffness" of the "elastic surface" to be fitted to the data points. Notice that it is usually not desirable to describe the exact form of $p(x)$, especially if x is very-high-dimensional; it is more important to be able to automatically find those dimensions and domains in the signal space where x has significant amounts of sample values!

This package contains two options for the definition of $h_{ci}(t)$. The simpler of them refers to a neighborhood set of array points around node c. Let this index set be denoted N_c (notice that we can define $N_c = N_c(t)$ as a function of time), whereby $h_{ci} = \alpha(t)$ if $i \in N_c$ and $h_{ci} = 0$ if $i \notin N_c$, where $\alpha(t)$ is some monotonically decreasing function of time ($0 < \alpha(t) < 1$). This kind of kernel is nicknamed "bubble", because it relates to certain activity "bubbles" in laterally connected neural networks [Kohonen 1989]. Another widely applied neighborhood kernel can be written in terms of the Gaussian function,

$$h_{ci} = \alpha(t) \cdot \exp\left(-\frac{||r_c - r_i||^2}{2\sigma^2(t)}\right) \ , \tag{3}$$

where $\alpha(t)$ is another scalar-valued "learning rate", and the parameter $\sigma(t)$ defines the width of the kernel; the latter corresponds to the radius of N_c above. Both $\alpha(t)$ and $\sigma(t)$ are some monotonically decreasing functions of time, and their exact forms are not critical; they could thus be selected linear. In this package it is furher possible to use a function of the type $\alpha(t) = A/(B + t)$, where A and B are constants; the inverse-time function is justified theoretically, approximately at least, by the so-called stochastic approximation theory. It is advisable to use the inverse-time type function with large maps and long training runs, to allow more balanced finetuning of the reference vectors. Effective choices for these functions and their parameters have so far only been determined experimentally; such default definitions have been used in this package.

The next step is **calibration** of the map, in order to be able to locate images of different input data items on it. In the practical applications for which such maps are intended, it may be usually self-evident from daily routines how a particular input data set ought to be interpreted. By inputting a number of typical, manually analyzed data sets and looking where the best matches on the map according to Eq. (1) lie, the map or at least a subset of its nodes can be labeled to delineate a "coordinate system" or at least a set of characteristic reference points on it according to their manual interpretation. Since this mapping is assumed to be continuous along some hypothetical "elastic surface", it may be self-evident how the unknown data are interpreted by means of interpolation and extrapolation with respect to these calibrated points.

3 Practical advices for the construction of good maps

Although it is possible to obtain some kind of maps without taking into account any precautions, nonetheless it is advisable to pay attention to the following arguments in order that the resulting mappings were stable, well oriented, and least ambiguous.

Form of the array: As stated earlier, the hexagonal lattice is to be preferred for visual inspection. The edges of the array ought to be rather rectangular than square, because the "elastic network" formed of the reference vectors m_i must be oriented along with $p(x)$ and stabilize in the learning process. Notice that if the array were, e.g., circular, it would have no stable orientation in the data space; so any oblongated form is to be preferred. On the other hand, since the m_i have to approximate to the $p(x)$, it would be desirable to find a form for the edges of the array the dimensions of which roughly correspond to the major dimensions of $p(x)$. Therefore, visual inspection of the rough form of $p(x)$, e.g., by **Sammon's mapping** (cf. [Sammon Jr. 1969] and Section 7.4) ought to be done first.

Learning with a small number of available training samples: Since for a good statistical accuracy the learning process (2) may require an appreciable number, say, 100'000 steps, and the number of available samples is usually much smaller, it is obvious that the samples must be used reiteratively in training. Several alternatives then exist: the samples may be applied cyclically or in a randomly permuted order, or picked up at random from the basic set (so-called bootstrap learning). It has turned out in practice that ordered cyclic application is not noticeably worse than the other, mathematically better justifiable methods.

Enhancement of rare cases: It may be obvious from the above that the SOM in some way tends to represent $p(x)$. However, in many practical problems there may occur important cases (input data) with small statistical frequency, whereby they would not get a representation on the SOM. Therefore, such important cases can be enhanced in learning by an arbitrary amount by taking a higher value of h_{ci} for these samples, or repeating these samples in a random order a sufficient number of times during the learning process. The weight parameters that correspond to the enhancement may be given in the training data file, and their determination should be done in cooperation with the users of these maps (see Section 5.1).

Quality of learning: Very different learning processes can be defined starting with different initial values $m_i(0)$, and applying different sequences of the training vectors $x(t)$ and different learning parameters. It is obvious that some optimal map for the same input data must exist. It may also be obvious that when comparing maps **that have the same "stiffness"** (same h_{ci}), the best map is expected to yield the smallest average quantization error because it is then fitted best to the same data. **The average quantization error**, or the mean of $||x - m_c||$ defined via inputting the training data once again is then a useful performance index. Therefore, an appreciable number (say, several tens) of random initializations of the $m_i(0)$ ought to be tried, and the map with the minimum quantization error selected. For this automatic choice there is a procedure in this package.

Especially with large maps, it is sometimes advisable to select the "best" SOM by computing a weighted distance measure $\sum h_{ci}||x-m_i||^2$ where h_{ci} is the neighborhood function, and use that value instead of the quantization error in comparison. This measure is called **the average distortion measure**. In this package it is possible to use either the usual quantization error, or the weighted distance measure (see Section 7.3) for the selection of the "best" match.

Notice too that there would be no sense in comparing quantization errors for different h_{ci}, because, e.g., it is a trivial fact that the error is minimum if $h_{ci} = \delta_{ci}$ (Kronecker delta). With this kernel, however, there is no self-organizing power left. In general the quantization error depends strongly on h_{ci}.

Missing input vector components: In many applications, sensor failures, recording errors and resource limitations can prevent data collection to complete each input vector. Such incomplete training examples still contain useful information, however. For example, partial data can still be used to determine the distribution statistics of the available vector components. Using the Self-Organizing Map algorithm one can easily utilize partial training data [Samad et al. 1992] [Kaski 1995].

For incomplete input data vectors the SOM_PAK has the possibility to mark the missing values by a predefined string ('*x*' by default). The SOM_PAK routines will compute the distance calculations and reference vector modification steps using the available data components.

NOTE: If some specific component is missing in *all* input data vectors, the results conserning that component are meaningless. The component should be removed from the data files.

Scaling of the components: This is a very subtle problem. One may easily realize that the orientation, or ordered "regression" of the reference vectors in the input space must depend on the scaling of the components (or dimensions) of the input data vectors. However, if the data elements have already been represented in different scales, there does not exist any simple rule to determine what kind of optimal rescaling should be used before entering the training data to the learning algorithm. One may try many heuristically justifiable rescalings and check the quality of the resulting maps by means of Sammon's mapping or average quantization errors.

Forcing representations to a wanted place on the map: Sometimes, especially for monitoring purposes, it may be desirable to map "normal" data onto specified locations (say, into the middle) of the map. It is possible to affect the orientation and translation of the reference vectors in the input space, for instance, by forcing some "normal" data samples to be mapped to some specified nodes. The *fixed* parameters that correspond to specified locations may be given in the training data file (see Section 5.1).

4 Installation of the program package

In the implementation of the SOM_PAK programs we have tried to use as simple a code as possible. Therefore the programs are supposed to compile in various machines without any specific modifications made on the code. All programs have been written in ANSI C.

The programs included in this basic package have not been provided with explicit graphics facilities; this makes it possible to run the programs equally well in all computers ranging from PCs to Cray supercomputers. The display programs generate lists of coordinates of points, which can be visualized by any standard graphics program.

4.1 Getting the program code

The latest version – currently Version 3.1 – of the *som_pak*-program package is available for anonymous ftp user at the Internet site *cochlea.hut.fi*. All programs and this documentation are stored in the directory */pub/som_pak*. The files are in multiple formats to ease their downloading and compiling.

The directory */pub/som_pak* contains the following files:

README	–	short description of the *som_pak* package
som_doc.ps	–	this document in ©PostScript format
som_doc.ps.Z	–	same as above but compressed
som_doc.txt	–	this document in ASCII format
som_p3r1.exe	–	self-extracting MS-DOS archive file
som_pak-3.1.tar	–	UNIX tape archive file
som_pak-3.1.tar.Z	–	same as above but compressed

An example of FTP access is given below

```
unix> ftp cochlea.hut.fi
Name: anonymous
Password: <your email address>
ftp> cd /pub/som_pak
ftp> binary
ftp> get som_pak-3.1.tar.Z
ftp> quit
unix>
```

4.2 Installation in UNIX

The archive file *som_pak-3.1.tar.Z* is intended to be used when installing *som_pak* in UNIX systems. It needs to be uncompressed to get the file *som_pak-3.1.tar*. If your system doesn't support the BSD *compress* utility, you may download the uncompressed file directly.

The *tar* archive contains the source code files, makefiles, and example data sets of the package, all in one subdirectory called *som_pak-3.1*. In order to create the subdirectory and extract all the files you should use the command *tar xovf som_pak-3.1*. (The switches of *tar* unfortunately vary, so you may need omit the '*o*'.)

The package contains a makefile called makefile.unix for compilation in UNIX systems. Before executing the *make* command, one or the other of them has to be copied to the name *makefile*. The makefile.unix is rather generic and should work as such in most systems.

We have written the source code for an ANSI standard C compiler and environment. If the *cc* compiler of your system doesn't fulfill these requirements, we recommend you to port the public domain GNU *gcc* compiler in your computer. When using *gcc*, the makefile macro definition CC=cc has to be changed accordingly to CC=gcc. The *makefile* also contains some other platform specific definitions, like optimizer and linker switches, that may need to be revised.

In order to summarize, the installation procedure is as follows:

> *uncompress som_pak-3.1.tar.Z*

> *tar xovf som_pak-3.1.tar*

> *cd som_pak-3.1*

> *cp makefile.unix makefile*

> *make*

After a successful make of the executables, you may test them by executing

> *make example*

which performs the commands as listed in section "6.4 Using the command lines (an example)".

4.3 Installation in DOS

The archive file *som_p3r1.exe* is intended to be used when installing *som_pak* in MS-DOS computers. It is a self-extracting packed archive compatible with the public domain *lha* utility. If your system supports UNIX *tar* archiving and *compress* file compressing utilities, you may as well use *som_pak-3.1.tar* and *som_pak-3.1.tar.Z* archives, as described in the previous section.

The *som_p3r1.exe* archive contains the source code files, makefiles, and example data sets of the package, all in one subdirectory called *som_pak.3r1*. In order to create the subdirectory and extract all the files simply use the command *som_p3r1*.

The package contains a makefile called *makefile.dos* for building up the object files. Before using the *make* command, *makefile.dos* has to be copied to the name *makefile*. It is intended to be used with the Borland Make Version 3.6 and the Borland C++ compiler Version 3.1, and may need to be revised if used with other compilation tools. Even with Borland C you may want to set some compiler switches, e.g., floating point options, according to your hardware.

In order to summarize, the installation procedure is as follows:

> *som_p3r1*

> *cd som_pak.3r1*

> *copy makefile.dos makefile*

> *make*

After a successful make of the executables, you may test them by executing

> *make example*

which performs the commands as listed in section "6.4 Using the command lines (an example)".

Some of the more advanced features are not available in DOS Version of the programs. These include the reading and writing of compressed files and usage of piped commands.

4.4 Hardware requirements

The compressed archive files are about 200 kbytes in size, whereas the extracted files take about 500 kbytes. When compiled and linked in MS-DOS, the executables are about 65 kbytes each. It is recommended to have at least 640 kbytes RAM, when using *som_pak* in MS-DOS. The execution times of the programs depend heavily on the hardware.

5 File formats

All data files (input vectors and maps) are stored as ASCII files for their easy editing and checking. The files that contain training data and test data are formally similar, and can be used interchangeably.

The data and map file formats are similar to the formats used in the *LVQ_PAK* program package. Thus the same data files can be used in both (*LVQ_PAK* and *SOM_PAK*) packages.

5.1 Data file format

The input data is stored in ASCII-form as a list of entries, one line being reserved for each vectorial sample.

The first line of the file is reserved for status knowledge of the entries; in the present version it is used to define the following items (these items MUST occur in the indicated order):

- Dimensionality of the vectors (integer, compulsory).
- Topology type, either *hexa* or *rect* (string, optional, case-sensitive).
- Map dimension in x-direction (integer, optional).
- Map dimension in y-direction (integer, optional).
- Neighborhood type, either *bubble* or *gaussian* (string, optional, case-sensitive).

In data files the optional items are ignored.

Subsequent lines consist of n floating-point numbers followed by an optional class label (that can be any string) and two optional qualifiers (see below) that determine the usage of the corresponding data entry in training programs. The data files can also contain an arbitrary number of comment lines that begin with '#', and are ignored. (One '#' for each comment line is needed.)

If some components of some data vectors are missing (due to data collection failures or any other reason) those components should be marked with 'x' (replacing the numerical value). For example, a part of a 5-dimensional data file might look like:

1.1	2.0	0.5	4.0	5.5
1.3	6.0	x	2.9	x
1.9	1.5	0.1	0.3	x

When vector distances are calculated for winner detection and when codebook vectors are modified, the components marked with x are ignored.

An example data file: Consider a hypothetical data file *exam.dat* that represents shades of colors in a three-component form. This file contains four samples, each one comprising a three-dimensional data vector. (The dimensionality of the vectors is given on the first line.) The labels can be any strings; here 'yellow' and 'red' are the names of the classes.

exam.dat:

```
3
# First the yellow entries
181.0  196.0   17.0  yellow
251.0  217.0   49.0  yellow
# Then the red entries
248.0  119.0  110.0  red
213.0   64.0   87.0  red
```

Each data line may have two optional qualifiers that determine the usage of the data entry during training. The qualifiers are of the form *codeword=value*, where spaces are not allowed between the parts of the qualifier. The optional qualifiers are the following:

- Enhancement factor: e.g. *weight=3*.
  The training rate for the corresponding input pattern vector is multiplied by this parameter so that the reference vectors are updated as if this input vector were repeated 3 times during training (i.e., as if the same vector had been stored 2 extra times in the data file).

- Fixed-point qualifier: e.g. *fixed=2,5*.
  The map unit defined by the fixed-point coordinates ($x = 2, y = 5$) is selected instead of the best-matching unit for training. (See below for the definition of coordinates over the map.) If several inputs are forced to known locations, a wanted orientation results in the map.

The optional qualifiers are not used by default; see the definition of the parameters *-fixed* and *-weights*.

5.2 Map file format

The map files are produced by the SOM_PAK programs, and the user usually does not need to examine them by hand.

The reference vectors are stored in ASCII-form. The format of the entries is similar to that used in the input data files, except that the optional items on the first line of data files (topology type, x- and y-dimensions and neighborhood type) are now compulsory. In map files it is possible to include several labels for each entry.

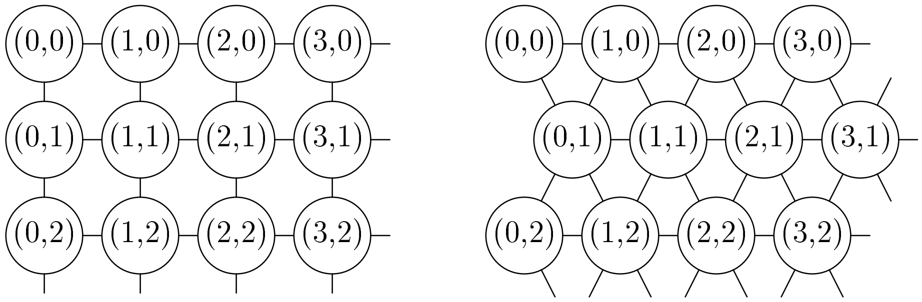
An example: The map file *code.cod* contains a map of three-dimensional vectors, with three times two map units. This map corresponds to the training vectors in the *exam.dat* file.

code.cod:

```
3 hexa 3 2 bubble
191.105  199.014  21.6269
215.389  156.693  63.8977
242.999  111.141  106.704
241.07   214.011  44.4638
231.183  140.824  67.8754
217.914  71.7228  90.2189
```

The x-coordinates of the map (column numbers) may be thought to range from 0 to $n-1$, where n is the x-dimension of the map, and the y-coordinates (row numbers) from 0 to $m-1$, respectively, where m is the y-dimension of the map. The reference vectors of the map are stored in the map file in the following order:

1	The unit with coordinates $(0,0)$.
2	The unit with coordinates $(1,0)$.
	...
n	The unit with coordinates $(n-1,0)$.
$n+1$	The unit with coordinates $(0,1)$.
	...
nm	The last unit is the one with coordinates $(n-1,m-1)$.



Rectangular Hexagonal

In the picture above the locations of the units in the two possible topological structures are shown. The distance between two units in the map is computed as an Euclidean distance in the (two dimensional) map topology.

6 Application of this package

6.1 The searching program *vfind*

The easiest way to use the *som_pak*-programs is to run the *vfind*-program, which searches for good mappings by automatically repeating different random initializing and training procedures and their testing several times. The criterion of a good mapping is a low quantization error.

The *vfind*-program asks all the required arguments interactively. The user only needs to start the program without any parameters (except that the verbose parameter (*-v*), the learning rate function type parameter (*-alpha_type*), the quantization error type parameter (*-qetype*) and the qualifier parameters (*-fixed* and *-weights*) can be given).

6.2 Using the programs directly

Usually the subprograms contained in this package are run separately and directly from the console using command lines defined in Section 7. The user should take care of that the programs are then run in the correct order: first initialization, then training, and then tests; and that the correct parameters are given (correspondence of the input and output files of subsequent programs is particularly important).

Each program requires some parameters: file names, learning parameters, sizes of maps, etc. All these must be given to the program in the beginning; the programs are not interactive in the sense that they do not ask for any parameters during their running.

6.3 Program parameters

Various programs need various parameters. All the parameters that are required by any program in this package are listed below. The meaning of the parameters is obvious in most cases. The parameters can be given in any order in the commands.

-din	Name of the input data file.
-dout	Name of the output data file.
-cin	Name of the file from which the reference vectors are read.
-cout	Name of the file to which the reference vectors are stored.
-rlen	Running length (number of steps) in training.
-alpha	Initial learning rate parameter. Decreases linearly to zero during training.
-radius	Initial radius of the training area in som-algorithm. Decreases linearly to one during training.
-xdim	Number of units in the x-direction.
-ydim	Number of units in the y-direction.
-topol	The topology type used in the map. Possible choices are hexagonal lattice (*hexa*) and rectangular lattice (*rect*).
-neigh	The neighborhood function type used. Possible choices are step function (*bubble*) and Gaussian (*gaussian*).
-plane	The component plane of the reference vectors that is displayed in the conversion routines.

- *-fixed* Defines whether the fixed-point qualifiers are used in the training programs. The value one means that fixed-point qualifiers are taken into account. Default value is zero.
- *-weights* Defines whether the weighting qualifiers are used in the training programs. The value one means that qualifiers are taken into account. Default value is zero.
- *-alpha_type* The learning rate function type (in *vsom* and *vfind*). Possible choices are linear function (*linear*, the default) and inverse-time type function (*inverse_t*). The linear function is defined as $\alpha(t) = \alpha(0)(1.0 - t/rlen)$ and the inverse-time type function as $\alpha(t) = \alpha(0)C/(C+t)$ to compute $\alpha(t)$ for an iteration step t. In the package the constant C is defined $C = rlen/100.0$.
- *-qetype* The quantization error function type (in *qerror* and *vfind*). If a value greater than zero is given then a weighted quantization function is used.
- *-version* Gives the version number of SOM_PAK.

In addition to these, it is always possible to give the *-v n* parameter (verbose parameter), which defines how much diagnostic output the program will generate. The value of n can range from 0 upwards, whereby greater values will generate more output; the default value is 1.

- *-v* Verbose parameter defining the output level.

In most programs it is possible to give the *-help 1* parameter, which lists the required and optional parameters for the program.

- *-help* Gives a list where the required and optional parameters are described.

In the initialization and training programs the random-number generator is used to select the order of the training samples, etc. The parameter *-rand* defines whether a new seed for the random number generator is given; when any other number than zero is given, that number is used as the seed, otherwise the seed is read from the system clock. The default value is zero.

- *-rand* Parameter that defines whether a new seed for the random-number generator is defined.

Some examples of the use of the parameters:

> *randinit -xdim 12 -ydim 8 -din exam1.dat -cout code1.map -topol hexa -neigh bubble*

An initialization program was called above to create a map of 12 times 8 units. The input entries used in the initialization were read from the file *exam1.dat* and the map was stored to the file *code1.map*. The topology of the map was selected to be hexagonal and the neighborhood function was step function.

> *vsom -din exam1.dat -cin code1.map -cout code1.map -rlen 10000 -alpha 0.05 -radius 10*

A training program was called. The training entries were read from the file *exam1.dat*; the map to be trained was read from the file *code1.map* and the trained reference vectors were resaved to the same file *code1.map*. Training was defined to take 10000 steps, but if there were fewer entries in the input file, the file was iterated a sufficient number of times. The initial learning rate was set to 0.05 and the initial radius of the learning "bubble" was set to 10.

> *qerror -din exam2.dat -cin code1.map*

The quantization error relating to the reference vectors stored in the map file *code1.cod* was tested using the test data file *exam2.dat*.

6.4 Using the command lines (an example)

The example given in this section demonstrates the direct use of command lines. It is meant for an introduction to the application of this package, and it may be helpful to study it in detail. (The example may also be run directly and automatically by the command *make example*.)

The data items used in the example stem from practice and describe measurements taken from a real device. They are contained in this package and consist of four data sets: *ex.dat*, *ex_fts.dat*, *ex_ndy.dat* and *ex_fdy.dat*. The first file (*ex.dat*) contains 3840 vectorial samples of the device state measurements and is used for training the map, while the second file (*ex_fts.dat*) contains 246 examples of faulty states of the device and is used for the calibration of the map. The other two files contain samples collected during 24 hours of operation of the device. In *ex_ndy.dat* the samples are from normally operating device and in *ex_fdy.dat* from a device that is overheating. These sample files have been intended to demonstrate how the map can be used for device monitoring.

Each vector in the files has a dimensionality of 5.

Below, the map is initialized and trained, the quantization error is evaluated, and the resulting map is visualized.

6.4.1 First stage: Map initialization

The reference vectors of the map are first initialized to tentative values. The lattice type of the map and the neighborhood function used in the training procedures are also defined in the initialization.

In the example the map is initialized using random numbers (the seed for the random number generator is specified with the *-rand* parameter). The lattice type is selected to be hexagonal (*hexa*) and the neighborhood function type is step function (*bubble*). The map size is here 12 times 8 units.

> *randinit -din ex.dat -cout ex.cod -xdim 12 -ydim 8 -topol hexa -neigh bubble -rand 123*

Now the map has been initialized.

6.4.2 Second stage: Map training

The map is trained by the self-organizing map algorithm using the program *vsom*.

Training is done in two phases. The first of them is the ordering phase during which the reference vectors of the map units are ordered. During the second phase the values of the reference vectors are fine-tuned.

In the beginning the neighborhood radius is taken almost equal to the diameter of the map and decreases to one during training, while the learning rate decreases to zero. (With *lininit* initialization the first phase (ordering) can be ignored and only the second phase of training is needed.)

> *vsom -din ex.dat -cin ex.cod -cout ex.cod -rlen 1000 -alpha 0.05 -radius 10*

During the second phase the reference vectors in each unit converge to their 'correct' values. The second phase is usually longer than the first one. The learning rate is thereby smaller. The neighborhood radius is also smaller on the average: in the beginning the units up to a distance of three are covered. In this example the training time of the second phase is ten times longer than in the first phase.

> *vsom -din ex.dat -cin ex.cod -cout ex.cod -rlen 10000 -alpha 0.02 -radius 3*

After these two phases of training the map is ready to be tested and to be used in monitoring applications.

6.4.3 Third stage: Evaluation of the quantization error

When the entries in the map have been trained to their final values, the resulting quantization error can be evaluated. The training file is used for this purpose. The program *qerror* is used to evaluate the average quantization error.

> *qerror -din ex.dat -cin ex.cod*

This program computes the quantization error over all the samples in the data file. The average quantization error with the training set in this example is expected to be 3.57.

6.4.4 Fourth stage: Map visualization

The trained map can now be used for visualization of data samples. In this package there are visualization programs, which make an image of the map (actually one selected component plane of it) and plot the trajectory of the best-matching units vs. time upon it.

Before visualization, the map units are calibrated using known input data samples. The sample file *ex_fts.dat* contains labeled samples from states of an overheating device.

> *vcal -din ex_fts.dat -cin ex.cod -cout ex.cod*

After calibration some of the units in the map have labels showing an area in the map which corresponds to fatal states.

The program *visual* generates a list of coordinates corresponding to all the best-matching units in the map for each data sample in the data file. It also returns the quantization errors and the class labels of the best-matching units, if the latter have been defined. The list of coordinates can then be processed for various graphical outputs.

The data file *ex_ndy.dat* contains samples collected during 24 hours from a device operating normally. The data file *ex_fdy.dat* contains samples collected during 24 hours from a device that has overheating problems during the day.

> *visual -din ex_ndy.dat -cin ex.cod -dout ex.nvs*

> *visual -din ex_fdy.dat -cin ex.cod -dout ex.fvs*

The program *visual* stores the three-dimensional image points (coordinate values of the responses and the quantization errors) in a similar fashion as the input data entries are stored.

The package also includes program *planes* to convert the map planes to encapsulated postscript (eps) images and program *umat* to compute so called u-matrix visualization [Ultsch, 1993] of the SOM reference vectors and to convert it to encapsulated postscript (eps) image.

7 Description of the programs of this package

7.1 Initialization programs

The initialization programs initialize the reference vectors.

- *randinit* - This program initializes the reference vectors to random values. The vector components are set to random values that are evenly distributed in the area of corresponding data vector components. The size of the map is given by defining the x-dimension (*-xdim*) and the y-dimension (*-ydim*) of the map. The topology of the map is defined with option (*-topol*) and is either hexagonal (*hexa*) or rectangular (*rect*). The neighborhood function is defined with option (*-neigh*) and is either step function (*bubble*) or Gaussian (*gaussian*).

  > *randinit -xdim 16 -ydim 12 -din file.dat -cout file.cod -neigh bubble -topol hexa*

- *lininit* - This program initializes the reference vectors in an orderly fashion along a two-dimensional subspace spanned by the two principal eigenvectors of the input data vectors.

  > *lininit -xdim 16 -ydim 12 -din file.dat -cout file.cod -neigh bubble -topol hexa*

7.2 Training programs

- *vsom* - This program trains the reference vectors using the self-organizing map algorithm. The topology type and the neighborhood function defined in the initialization phase are used throughout the training. The program finds the best-matching unit for each input sample vector and updates those units in the neighborhood of it according to the selected neighborhood function.

  The initial value of the learning rate is defined and will decrease linearly to zero by the end of training. The initial value of the neighborhood radius is also defined and it will decrease linearly to one during training (in the end only the nearest neighbors are trained). If the qualifier parameters (*-fixed* and *-weight*) are given a value greater than zero, the corresponding definitions in the pattern vector file are used. The learning rate function α can be defined using the option *-alpha_type*. Possible choices are *linear* and *inverse_t*. The linear function is defined as $\alpha(t) = \alpha(0)(1.0 - t/rlen)$ and the inverse-time type function as $\alpha(t) = \alpha(0)C/(C + t)$ to compute $\alpha(t)$ for an iteration step t. In the package the constant C is defined $C = rlen/100.0$.

> *vsom -din file.dat -cin file1.cod -cout file2.cod -rlen 10000 -alpha 0.03 -radius 10 [-fixed 1] [-weights 1] [-alpha_type linear] [-snapinterval 200] [-snapfile file.snap]*

Notice that the degree of forcing data into specified map units can be controlled by alternating "fixed" and "nonfixed" training cycles.

7.3 Quantization accuracy program

- *qerror* - The average quantization error is evaluated. For each input sample vector the best-matching unit in the map is searched for and the average of the respective quantization errors is returned.

  > *qerror -din file.dat -cin file.cod [-qetype 1] [radius 2]*

  It is possible to compute a weighted quantization error $\sum h_{ci}||x - m_i||^2$ for each input sample and average these over the data files. If option *-qetype* is given a value greater than zero, then a weighted quantization error is used. Option *-radius* can be used to define the neighborhood radius for the weighting, default value for that is 1.0.

7.4 Monitoring programs

- *visual* - This program generates a list of coordinates corresponding to the best-matching unit in the map for each data sample in the data file. It also gives the individual quantization errors made and the class labels of the best matching units if the latter have been defined. The program will store the three-dimensional image points (coordinate values and the quantization error) in a similar fashion as the input data entries are stored. If a input vector consists of missing components only, the program will skip the vector. If option *-noskip* is given the program will indicate the existence of such line by saving line '*-1 -1 -1.0 EMPTY_LINE*' as a result.

  > *visual -din file.dat -cin file.cod -dout file.vis [-noskip 1]*

- *sammon* - Generates the Sammon mapping [Sammon Jr. 1969] from n-dimensional input vectors to 2-dimensional points on a plane whereby the distances between the image vectors tend to approximate to Euclidean distances of the input vectors. If option *-eps* is given an encapsulated postscript image of the result is produced. Name of the eps-file is generated by using the output file basename (up to the last dot in the name) and adding the ending *_sa.eps* to the output filename. If option *-ps* is given a postscript image of the result is produced. Name of the ps-file is generated by using the output file basename (up to the last dot in the name) and adding the ending *_sa.ps* to the output filename.

In the following example, if the option *-eps 1* is given, an eps file named *file_sa.eps* is generated.

> *sammon -cin file.cod -cout file.sam -rlen 100 [-eps 1] [-ps 1]*

- *planes* - This program generates an encapsulated postscript (eps) code from one selected component plane (specified by the parameter *-plane*) of the map imaging the values of the components using gray levels. If the parameter given is zero, then all planes are converted. If the input data file is also given, the trajectory formed of the best-matching units is also converted to a separate file. The eps files are named using the map basename (up to the last dot in the name) and adding *_px.eps* (where x is replaced by the plane index, starting from one) to it. The trajectory file is named accordingly adding *_tr.eps* to the basename. If the *-ps* option is given a postscript code is generated instead and the produced files are named replacing *.eps* by *.ps*.

  In the following example a file named *file_p1.eps* is generated containing the plane image. If the *-din* option is given, another file *file_tr.eps* is generated containing the trajectory. If the *-ps* option is given then the produced file is named *file_p1.ps*.

  > *planes -cin file.cod [-plane 1] [-din file.dat] [-ps 1]*

- *umat* - This program generates an encapsulated postscript (eps) code to visualize the distances between reference vectors of neighboring map units using gray levels. The display method has been described in [Ultsch, 1993] [Iivarinen et al., 1994] [Kraaijveld et al., 1992]. The eps file is named using the map basename (up to the last dot in the name) and adding *.eps* to it.

  If the *-average* option is given the grey levels of the image are spatially filtered by averaging, and if the *-median* option is given median filtering is used. If the *-ps* option is given a postscript code is generated instead and *.ps* ending in filename is used.

  In the following example a file named *file.eps* is generated containing the image.

  > *umat -cin file.cod [-average 1] [-median 1] [-ps 1]*

7.5 Other programs

- *vcal* - This program labels the map units according to the samples in the input data file. The best-matching unit in the map corresponding to each data vector is searched for. The map units are then labeled according to the majority of labels 'hitting' a particular map unit. The units that get no 'hits' are left unlabeled. Giving the option *-numlabs* one can select the maximum number of labels saved for each codebook vector. Default value is one.

  > *vcal -din file.dat -cin file.cod -cout file.cod [-numlabs 2]*

8 Advanced features

Some more advanced features has been added into the SOM_PAK program package in Version 3.0. These features are intended to ease the usage of the package by offering ways to use e.g. compressed data files directly and to save snapshots of the map during the training run.

The advanced features include:

- Buffered loading (the whole data file need not be loaded into memory at once)
- Reading and writing of:
  - compressed files
  - stdin/stdout
  - piped command
- Snapshots of the codebook during teaching
- Environment variables

Buffered loading

This means that the whole data set doesn't have to be loaded in memory all the time. SOM_PAK can be set, for example, to hold max 10000 lines of data in memory at a time. When the 10000 data vectors have been used, the next 10000 data vectors are loaded over the old ones. The buffered reading is transparent to the user and it works also with compressed files.

Note that when the whole file has been read once and we want to reread it, the file has to be rewound (for regular files) or the uncompressing command has to be rerun. This is done automatically and the user need not to worry about it, but some restrictions are enforced on the input file: If the source is a pipe, it can't be rewound. Regular files, compressed files and standard input (if it is a file) work. Pipes work fine if you don't have to rewind them, ie. there is no end in the data, or the number of iterations is smaller than the number of data vectors.

-buffer Defines the number of lines of input data file that are read at a time.

Most programs support the buffered reading of data files. It is activated with the command line option *-buffer* followed with the maximum number of data vectors to be kept in memory. For example, to read the input data file 10000 lines at a time one uses:

> *vsom -buffer 10000 ...*

Reading and writing compressed files

To read or write compressed files just put the suffix *.gz* at the end of the filename. The file is automatically uncompressed or compressed as the file is being read or written. SOM_PAK uses 'gzip' for compressing and uncompressing. It can also read files compressed with regular UNIX compress-command. The commands used for compressing and decompressing can be changed with command line options or at compile time.

Example: with *vsom*, to use a compressed data file for teaching:

> *vsom -din data.dat.gz ...*

Reading and writing stdin/stdout

To use standard input or output, use the minus sign ('-') as a filename. Data is then read from *stdin* and written to *stdout*. For example, to read training data from *stdin* with vsom:

> *vsom -din - ...*

Reading and writing piped commands

If you use a filename that starts with the UNIX pipe character ('|'), the filename is executed as a command. If the file is opened for writing the output of the SOM command is piped to the command as standard input. Likewise, when the file is opened for reading the output of the command is read by the SOM programs.

For example:

> *vsom -cin "|randinit ..." ...*

would start the program *randinit* when it wants to read the initial codebook. However, the same thing could be done with:

> *randinit ...| vsom -cin - ...*

Snapshots

Saves snapshots of the codebook during training (in *vsom* program).

-snapinterval Interval between snapshots (in iterations).

-snapfile Name of the snapfile. If the name given contains string '%d', the number of iterations taken so far is included to the filename.

The interval between snapshots is specified with the option *-snapinterval*. The snapshot filename can be specified with the option *-snapfile*. If no filename is given, the name of the output code file is used. The filename is actually passed to 'sprintf(3)' as the format string and the number of iterations so far is passed as the next argument. For example:

> *vsom -snapinterval 10000 -snapfile "ex.%d.cod" ...*

gives you snapshots files every 10000 iterations with names starting with: *ex.10000.cod, ex.20000.cod, ex.30000.cod*, etc.

Environmental variables

Some defaults can be set with environment variables:

LVQSOM_COMPRESS_COMMAND Defines the command used to compress files. Default: "gzip -9 -c >%s"

LVQSOM_UNCOMPRESS_COMMAND Defines the command used to decompress files. Default: "gzip -d -c %s"

LVQSOM_MASK_STR Defines the string which is used to replace missing input vector components. Default: "x"

Other new options

- *-mask_str* Defines the string which is used to replace missing input vector components.
- *-compress_cmd* Defines the compress command.
- *-uncompress_cmd* Defines the uncompress command.
- *-numlabs* Defines the maximum number of labels a codebook entry can have (in *vcal*).
- *-noskip* Do not skip those entries where there are only missing components (in *visual*).
- *-average* Filter the u-matrix image using averaging (in *umat*).
- *-median* Filter the u-matrix image using median filtering (in *umat*).
- *-eps* Generate encapsulated postscript code (in *sammon*).
- *-ps* Generate postscript code (instead of encapsulated postscript) (in *umat*, *planes* and *sammon*).

By default the components of the data vectors that are marked with '*x*' are ignored. This string can be changed with the *-mask_str* option. For example,

> *vsom -mask_str "MIS" ...*

would ignore components thats are marked with string '*MIS*' instead of '*x*'. The string is case insensitive.

The command used to compress files can be changed by the option *-compress_cmd*. Similarly the uncompress command can be changed by the option *-uncompress_cmd*.

The *vcal* program can give several labels to each codebook entry. Using the option *-numlabs* one can restrict the number of labels. Default value is to use at most one label per code.

When input data has been automatically collected, it is possible that some vector components are missing. In extreme cases it is even possible that all components are missing (i.e. there are no numerical values left). These data vectors are not usable in training programs, but for visualization they might be useful for example to mark time steps where the collection has been unfunctional. In *visual* program the default behaviour is to skip the empty entries, but they can be included into the visual result file by giving an option *-noskip*. The resulting line would look out as '*-1 -1 -1.0 EMPTY_LINE*'.

9 Comments on and experiences of the use of this package

Comments on and experiences of the installation and use of these programs are welcome, and may be sent to the e-mail address *som@cochlea.hut.fi*.

9.1 Changes in the package

No changes to the central algorithms have been made. The following are the details that have been changed from the Version 1.0:

1. Those who have already used the Gaussian kernel, Eq. (3), in Version 1.0 may have noticed that the learning rate in the beginning has been rather small. From Version 1.1 on we have now revised Eq. (3) into the form in which it has originally appeared in publications. Anyway it will be necessary to use parameter values that are different from the 'bubble' case, and must be found experimentally. We recommend that for comparison, the 'bubble' kernel should always be tried first, and in the examples we have given recommendable parameter values for it.

2. The function 'strdup' that was used in some functions is not included in the ANSI C standard. From Version 1.2 on we have written a new function 'ostrdup' that is functionally equivalent to 'strdup' and is used throughout the program code.

3. There was an error in the function 'hexa_dist' that introduced a small 'random' factor in the distance calculation. That error has now been corrected.

4. In Version 3.0 it is possible to have missing components in input data vectors.

5. In Version 3.0 it is possible to use a weighted quantization function.

6. In Version 3.0 it is possible to use an inverse function as a learning rate function $\alpha(t)$.

7. In Version 3.0 it is possible to read the input data files in pieces, i.e. to have only a portion of the whole data in main memory at a time. This will enable using the SOM_PAK programs in PC-machines with large data files.

8. In version 3.0 there are several new 'advanced' features to allow reading and writing of compressed files, stdin and stdout, and piped commands.

9. In version 3.0 it is now possible to save 'snapshots' of the state of codebook during training.

10. The only change made to Version 3.1 was a bug fix in the random ordering of data.

References

[Erwin et al. 1992] Ed Erwin, Klaus Obermayer, Klaus Schulten. Self-organizing maps: Ordering, convergence properties and energy functions. *Biological Cybernetics*, 67(1):47–55, 1992.

[Iivarinen et al., 1994] J. Iivarinen, T. Kohonen, J. Kangas, S. Kaski Visualizing the clusters on the self-organizing map. *Multiple Paradigms for Artificial Intelligence (SteP94)*, 122–126. Finnish Artificial Intelligence Society, 1994.

[Kaski 1995] Sami Kaski, Teuvo Kohonen. *Structures of Welfare and Poverty in the World Discovered by the Self-Organizing Map.* Report A24, Helsinki University of Technology, Laboratory of Computer and Information Technology, 1995.

[Kohonen 1989] Teuvo Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, Berlin-Heidelberg-New York-Tokio, 3 edition, 1989.

[Kohonen 1990] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[Kohonen 1991] Teuvo Kohonen. Self-organizing maps: Optimization approaches. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 981–990, Espoo, Finland, June 1991.

[Kohonen 1995] Teuvo Kohonen. *Self-Organizing Maps.* Springer-Verlag, Heidelberg, 1995.

[Kraaijveld et al., 1992] M. A. Kraaijveld, J. Mao, A. K. Jain. A non-linear projection method based on Kohonen's topology preserving maps. *Proceedings of the 11th International Conference on Pattern Recognition (11ICPR)*, 41–45, Los Alamitos, CA. IEEE Comput. Soc. Press, 1992.

[Luttrell 1989] S. Luttrell. Self-organization: a derivation from first principles of a class of learning algorithms. In *Proceedings of International Joint Conference on Neural Networks*, pages II–495–498, Washington, D.C., 1989.

[Ritter et al. 1988] H. Ritter, K. Schulten. Kohonen self-organizing maps: exploring their computational capabilities. In *Proceedings of IEEE International Conference on Neural Networks*, pages 109–116, San Diego, California, July 1988.

[Samad et al. 1992] T. Samad, S. A. Harp. Self-organization with partial data. *Network: Computation in Neural Systems*, 3(2):205-212, 1992.

[Sammon Jr. 1969] John W. Sammon Jr. A nonlinear mapping for data structure analysis. *IEEE Transactions on Computers*, C-18(5):401–409, May 1969.

[Ultsch, 1993] A. Ultsch. Self organized feature maps for monitoring and knowledge aquisition of a chemical process. S. Gielen, B. Kappen, editors, *Proceedings of the International Conference on Artificial Neural Networks (ICANN93)*, 864–867, London. Springer-Verlag, 1993