Appendix

The program package is available at the Internet site "cochlea.hut.fi" and will be updated continuously. The indexing of the latest release will always be of the form "lvq_pak-X.Y". The instructions below are indexed as in the version lvq_pak-1.1, which was released on December 31, 1991.

The programs are available in two archive formats, one for the UNIX-environment, the other for MS-DOS, respectively. Both archives contain exactly the same files. These files can be accessed via FTP as follows:

1. Create an FTP connection from wherever you are to machine "cochlea.hut.fi". The Internet address of this machine is 130.233.168.48, for those who need it.
2. Log in as user "anonymous" with your own e-mail address as password.
3. Change the remote directory to "/pub/lvq_pak".
4. At this point the FTP should be able to get a listing of files in this directory with DIR and fetch the ones you want with GET. (The exact FTP commands depend on your local FTP program.) Remember to use the binary transfer mode for compressed files.

The LVQ_PAK program package includes the following files:

- Documentation:

README	short description of the package and installation instructions
document.ps	documentation in (c) Postcript format
document.ps.Z	same as above but compressed
document.txt	documentation in ASCII format

- Source file archives (which contain the documentation, too):

lvq_p1r1.exe	self-extracting MS-DOS archive file
lvq_pak-1.1.tar	UNIX tape archive file
lvq_pak-1.1.tar.Z	same as above but compressed

An example of FTP access is given below:

```
unix> ftp cochlea.hut.fi (or 130.233.168.48)
Name: anonymous
Password: <your e-mail address>
ftp>  cd /pub/lvq_pak
ftp>  binary
ftp>  get lvq_pak-1.1.tar.Z
ftp>  quit
unix> uncompress lvq_pak-1.1.tar.Z
unix> tar xvfo lvq_pak-1.1.tar
```

3.3 Test data files

The package also contains two data sets, *ex1.dat* and *ex2.dat*, which can be used to experiment with the programs. An exemplary procedure is given in the documentation of the package. All data files (input vectors and codebooks) are stored in ASCII format for their easy editing and checking.

3.4 Diagnostics

With all the programs, it is possible to set the *verbose* parameter (*-v n*), where the value of *n* determines the amount of diagnostic output the program generates.

For the visualization of topological relationships of data and codebook vectors, the monitoring program *sammon* generates a list of coordinates of points that can be displayed two-dimensionally by any standard graphics program. The two-dimensional Sammon mapping approximates to Euclidean distances of the data space, and thus visualizes the clustering of the data.

3.5 Obtaining and running the programs

All the programs have been written in the ANSI C. They are supposed to compile in any UNIX or MS-DOS machine. Instructions for their use are contained in the documentation. The complete package can be obtained as explained in the Appendix.

3.6 Hardware considerations

The archive files are about 300 kbytes in size, whereas the extracted files take about 1 Mbytes. When compiled and linked in MS-DOS, the executables are about 45 kbytes each. It is recommended to have at least 640 kbytes of RAM, when using the *lvq_pak* in MS-DOS.

The execution times of the programs depend heavily on the hardware. For example, the execution of the subprogram *olvq1* for the given data, with 164 twenty-dimensional codebook vectors, takes about 10 seconds in the Cray XMP at normal loading conditions, about 20 seconds in Sun or Silicon Graphics workstations at low loading, about 550 seconds in a PC/386SX provided with a math co-processor and running at 16 MHz, and about 7960 seconds with the same PC without co-processor.

References

[1] T. Kohonen, The Self-Organizing Map, Proceedings of the IEEE, Vol. 78, No. 9, pp. 1464-1480, Sept. 1990.

[2] T. Kohonen, Learning Vector Quantization for Pattern Recognition, Helsinki University of Technology, Laboratory of Computer and Information Science, Report TKK-F-A601, 1986.

[3] T. Kohonen, Statistical Pattern Recognition Revisited, in *Advanced Neural Computers*, R. Eckmiller (Ed.) (Elsevier, 1990) pp. 137-144.

[4] J.W. Sammon Jr., A Nonlinear Mapping for Data Structure Analysis, IEEE Transactions on Computers, Vol. C-18, No. 5, pp. 401-409, May 1969.

2.4 Stopping rule

Neural-network algorithms often "overlearn", i.e., with extensive learning, the recognition accuracy is first improved, but may then very slowly start decreasing again. Various reasons account for this phenomenon; in the present case, especially with a limited number of training samples that are being recycled in learning, the codebook vectors become very specifically tuned to the training data (i.e., they are optimized only with respect to this subset), with the result that their ability to generalize for other data values suffers from that. It is therefore necessary to stop the learning process after some 'optimal' number of steps, say, 30 to 50 times the total number of codebook vectors in the case of OLVQ1, or 100 to 500 times the number of codebook vectors in the cases of LVQ1 and LVQ2.1 (depending on the learning rate and type of data).

3 Overview of the programs in the LVQ_PAK

3.1 The interface program *lvq_run*

This package contains an interactive program *lvq_run*, which automatically executes some of the most important procedures. The program asks the user about the needed parameters and takes care of running the recommended subprograms in the correct order.

3.2 The subprograms

There are in total 16 subprograms, which can be run separately and directly at the console. Each subprogram needs some parameters: file names, learning parameters, sizes of codebooks, etc. Some of the parameters are obligatory, the others are optional. The most often used of these subprograms are:

eveninit selects an equal number of codebook vectors to each class and sets their initial values.

propinit selects the numbers of codebook vectors to each class in proportion to the *a priori* probabilities of the classes and sets their initial values.

mindist computes and displays the medians of the shortest distances between codebook vectors in each class and the standard deviations of the class samples.

balance adds or deletes codebook vectors so that the medians of the shortest distances between codebook vectors in each class are equalized.

olvq1, lvq1, lvq2 are training programs.

accuracy evaluates the recognition accuracy using an independent test data set.

sammon generates the Sammon mapping [4] from an n-dimensional data space to the two-dimensional plane for the visualization of distributions of data or codebook vectors (cf. Sec. 3.4).

For the program *olvq1* all the learning rate parameters α_i, for each set of codebook vectors, are automatically saved in a separate file.

be defined by Eq. (1). Then

$$\begin{aligned} m_c(t+1) &= m_c(t) + \alpha_c(t)[x(t) - m_c(t)] && \text{if } x \text{ is classified correctly,} \\ m_c(t+1) &= m_c(t) - \alpha_c(t)[x(t) - m_c(t)] && \text{if the classification of } x \text{ is incorrect,} \\ m_i(t+1) &= m_i(t) && \text{for } i \neq c \ . \end{aligned} \tag{2}$$

Next we address the problem of whether the $\alpha_i(t)$ can be determined optimally for fastest possible convergence of (2). If we express (2) in the form

$$m_c(t+1) = [(1 - s(t)\alpha_c(t)]m_c(t) + s(t)\alpha_c(t)x(t) \tag{3}$$

where $s(t) = +1$ if the classification is correct and $s(t) = -1$ if the classification is wrong, we first directly see that $m_c(t)$ is statistically independent of $x(t)$. It may also be obvious that the statistical accuracy of the learned codebook vector values is optimal if the effects of the corrections made at different times, when referring to the end of the learning period, are of equal weight. Notice that $m_c(t+1)$ contains a "trace" from $x(t)$ through the last term in (3), and "traces" from the earlier $x(t'), t' = 1, 2, \ldots, t-1$ through $m_c(t)$. The (absolute) magnitude of the last "trace" from $x(t)$ is scaled down by the factor $\alpha_c(t)$, and, for instance, the "trace" from $x(t-1)$ is scaled down by $[1 - s(t)\alpha_c(t)] \cdot \alpha_c(t-1)$. Now we first stipulate that these two scalings must be identical:

$$\alpha_c(t) = [1 - s(t)\alpha_c(t)]\alpha_c(t-1) \ . \tag{4}$$

If this condition is then made to hold for all t, by induction it can be shown that the "traces" collected up to time t from all the earlier x will be scaled down by an equal amount at the end, and thus the "optimal" values of $\alpha_i(t)$ are determined by the recursion

$$\alpha_c(t) = \frac{\alpha_c(t-1)}{1 + s(t)\alpha_c(t-1)} \ . \tag{5}$$

Any user of the LVQ_PAK can easily become convinced about that (5) really provides for fast convergence. A precaution must be made, however: since $\alpha_c(t)$ can also increase, it is especially important that it does not rise above the value 1; the learning program *olvq1* in this package is even more restrictive, it never allows any α_i to rise above its initial value. With this provision, the initial values of the α_i can be selected rather high, say, 0.3, whereby learning is significantly speeded up, especially in the beginning, and the m_i quickly find their approximate asymptotic values.

It must be warned, too, that (5) is not applicable to the LVQ2, since thereby the α_i, on the average, would not decrease, and the process would not converge.

2.3 Fine tuning using the basic LVQ1 or LVQ2.1

If the learning time is of no concern, the recognition accuracy may still be increased after OLVQ1 by fine tuning the codebook vectors. Two algorithms, the basic LVQ1 [1] and the LVQ2.1 [3] have been selected into this package for that purpose. The learning rate $\alpha(t)$ in these algorithms is identical for all the units, and decreases linearly to zero in a given number of learning steps. For the initial $\alpha(t)$, a value smaller than 0.1 should be selected.

functions by these quantized values. In the LVQ algorithms, however, the codebook vectors are not directly used to approximate to any density functions, but to provide for a sufficient number of reference points to describe various class zones. The *borders* of the zones are then defined (for an arbitrary input vector $x \in \Re^n$) by first searching the codebook vector that has the smallest vectorial difference from x. Let the $m_i \in \Re^n$ be codebook vectors that describe various classes and are labeled by the corresponding class symbols. Let the index of the "nearest" m_i be defined by

$$c = \arg \min_i \{||x - m_i||\}. \tag{1}$$

The class borders consists of those sets of points in $\Re^n$ at which the class label of c changes when crossing a border.

Optimal values for the reference points are determined in adaptive learning processes. For the initial guess of the m_i values, one may take copies of samples of the input vectors. Then, however, these samples should already lie within the correct class zones. Since the latter are not known before the end of the learning process, we have applied the following approximative criterion. By performing a *knn* (k-nearest-neighbor) classification of an input sample against all the other available input samples (or a randomly selected subset of them), we accept a sample for an initial m_i value if and only if the majority of its k nearest input samples has the same class label that the sample itself has.

The number of codebook vectors to be assigned to each class is another problem, and no simple answer to it is known. (Naturally one might try all the possible combinations of numbers and perform the comparative classifications, but this method is formidably slow.) Experimentally we have found out that a very good strategy in practice, even when the class distributions are very dissimilar, is to select an identical number of codebook vectors to each class! This choice is based on the view that since the forms of the class borders are based on nearest-neighbor classification (or the Voronoi tessellation), then for optimal quantization error of the class borders, the piecewise linear segments or edges of the borders should roughly be of equal size (independent of densities of samples within the classes). The program *eveninit* performs this kind of initialization.

In an attempt to still better equalize the sizes of the above edges, there exists a program named *balance* in this package that iteratively adds codebook vectors to or deletes them from the various classes, followed by a tentative short learning cycle, after which the shortest distances between the codebook vectors in each class are again evaluated; they are then supposed to be even more equal. This strategy has been found to slightly improve the final classification accuracy.

The total number of codebook vectors should at least be so high that the average distances between the adjacent codebook vectors in each class are smaller than the standard deviations (square roots of variances) of the respective class samples. This can be checked by the *mindist* program. If the number of codebook vectors is increased much beyond this, at least three detrimental effects start to manifest themselves: (i) the time needed to search for the best-matching node increases, (ii) for a sufficient statistical accuracy in learning, a certain number of training steps per node is needed, and then also the total number of training steps must be bigger, (iii) if the number of available training data is limited, adding more codebook vectors only causes their clustering around the input samples, without any real improvement in accuracy.

2.2 The optimized-learning-rate LVQ1 (OLVQ1)

The basic LVQ1 algorithm (cf., e.g., [1]) is now modified in such a way that an individual learning rate $\alpha_i(t)$ is assigned to each m_i. We then get the following discrete-time learning process. Let c

LVQ_PAK: A program package for the correct application of Learning Vector Quantization algorithms

Teuvo Kohonen, Jari Kangas, Jorma Laaksonen, and Kari Torkkola

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

Abstract. This paper is an overview of the program package LVQ_PAK, which has been developed for convenient and effective application of Learning Vector Quantization algorithms. Two new features are included: fast conflict-free initial distribution of codebook vectors into the class zones, and the optimized-learning-rate algorithm OLVQ1.

1 Introduction

Statistical classification or pattern recognition of stochastic vectorial samples can be performed by the Learning Vector Quantization (LVQ) algorithms. Description of their three main variants, LVQ1, LVQ2, and LVQ3 can be found in Ref. [1].

There are certain delicate details in the application of these algorithms; if they are not taken into account, the recognition results and the learning speed may vary significantly. In particular, the following problems have to be solved properly:

- assignment of an approximately optimal number of codebook vectors to each class and setting of their initial values,
- selection of a proper learning rate during the steps and the stopping rule for learning.

Although the first of the LVQ algorithms, the LVQ1, was already introduced in 1986 [2], and applied to numerous practical tasks since then, no detailed instructions of how to properly solve all the above problems have been given earlier. For this reason we decided to develop this easily applicable program package, which contains recommended procedures and can be run in most contemporary computers ranging from PCs to supercomputers. This package is provided with diagnostic and monitoring facilities, for instance to visualize the learning results, and it also contains data files picked up from natural speech for convenient testing of the programs.

This program package, LVQ_PAK, has been available in the public domain since December 19, 1991, and can be copied freely from the Internet site "cochlea.hut.fi" (130.233.168.48) via anonymous FTP (file transfer protocol); see the instructions given in the Appendix.

2 The main problems in the application of the LVQ algorithms

2.1 Initialization of the codebook vectors

The basic idea in classical Vector Quantization methods is to place a number of reference vectors, called codebook vectors, into the space of input variables to approximate them or their density