6. Learning Vector Quantization

Closely related to VQ and SOM is *Learning Vector Quantization (LVQ)*. This name signifies a class of related algorithms, such as LVQ1, LVQ2, LVQ3, and OLVQ1. While VQ and the basic SOM are *unsupervised* clustering and learning methods, LVQ describes *supervised learning*. On the other hand, unlike in SOM, no neighborhoods around the "winner" are defined during learning in the basic LVQ, whereby also no spatial order of the codebook vectors is expected to ensue.

Since LVQ is strictly meant for a *statistical classification or recognition method*, its only purpose is to define *class regions* in the input data space. To this end a subset of similarly labeled codebook vectors is placed into each class region; even if the class distributions of the input samples would overlap at the class borders, the codebook vectors of each class in these algorithms can be placed in and shown to stay within each class region for all times. The quantization regions, like the Voronoi sets in VQ, are defined by midplanes (hyperplanes) between neighboring codebook vectors. An additional feature in LVQ is that for class borders one can only take such borders of the Voronoi tessellation that separate Voronoi sets into different classes. The class borders thereby defined are piecewise linear.

6.1 Optimal Decision

The problem of *optimal decision* or *statistical pattern recognition* is usually discussed within the framework of the Bayes theory of probability (Sect. 1.3.3). Assume that all samples of x are derived from a finite set of classes $\{S_k\}$, the distributions of which usually overlap. Let $P(S_k)$ be the *a priori* probability of class S_k, and $p(x|x \in S_k)$ be the conditional probability density function of x on S_k, respectively. Define the *discriminant functions*

$$\delta_k(x) = p(x|x \in S_k)P(S_k) \; . \tag{6.1}$$

Let it be recalled that on the average, the unknown samples are classified optimally (i.e., the rate of misclassifications is minimized) if a sample x is determined to belong to class S_c according to the decision

$$\delta_c(x) = \max_k \{\delta_k(x)\} \; . \tag{6.2}$$

Let us recall that the traditional method in practical statistical pattern recognition was to first develop approximations for the $p(x|x \in S_k)P(S_k)$ and then to use them for the $\delta_k(x)$ in (6.1). The LVQ approach, on the other hand, is based on a totally different philosophy. Consider Fig. 6.1. We first assign *a subset of codebook vectors to each class* S_k and then search for that codebook vector m_i that has the smallest Euclidean distance from x. The sample x is thought to belong to the same class as the closest m_i. The codebook vectors can be placed in such a way that those ones belonging to different classes are not intermingled, although the class distributions of x overlap. Since now only the codebook vectors that lie closest to the class borders are important to the optimal decision, obviously a good approximation of $p(x|x \in S_k)$ is not necessary everywhere. It is more important to place the m_i into the signal space in such a way that the nearest-neighbor rule used for classification minimizes the average expected misclassification probability.

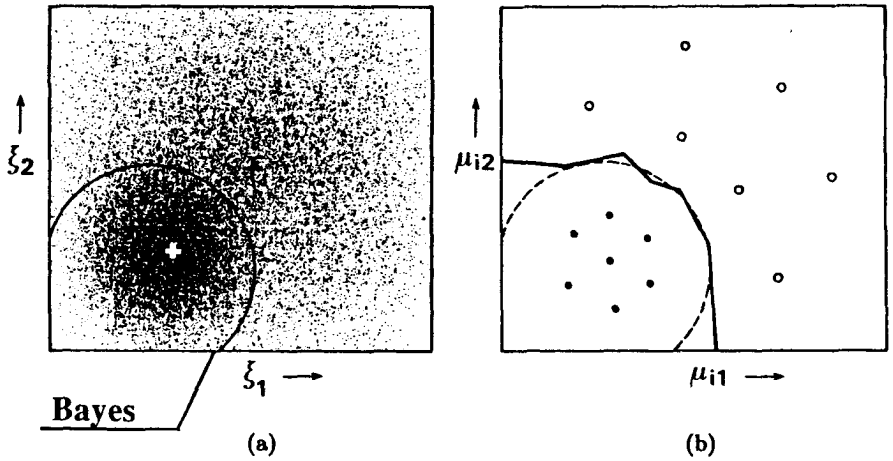

Fig. 6.1. **(a)** Small dots: Superposition of two symmetric Gaussian density functions corresponding to classes S_1 and S_2, with with their centroids shown by the white and dark cross, respectively. Solid curve: Bayes decision border. **(b)** Large black dots: reference vectors of class S_1. Open circles: reference vectors of class S_2. Solid curve: decision border in the learning vector quantization. Broken curve: Bayes decision border

6.2 The LVQ1

Assume that several codebook vectors are assigned to each class of x values, and x is then determined to belong to the same class to which the nearest m_i belongs. Let

$$c = \arg\min_i \{||x - m_i||\} \tag{6.3}$$

define the index of the nearest m_i to x.

Notice that c, the index of the "winner", depends on x and all the m_i. If x is a natural, stochastic, continuous-valued vectorial variable, we need not consider multiple minima: the probability for $||x - m_i|| = ||x - m_j||$ for $i \neq j$ is then zero.

Let $x(t)$ be an input sample and let the $m_i(t)$ represent sequential values of the m_i in the discrete-time domain, $t = 0, 1, 2, \ldots$. Values for the m_i in (6.3) that approximately minimize the rate of misclassification errors are found as asymptotic values in the following learning process [2.38], [2.40], [2.41]. Starting with properly defined initial values (as will be discussed in Sect. 6.9), the following equations define the basic Learning Vector Quantization process; this particular algorithm is called LVQ1.

$$\begin{aligned} m_c(t+1) &= m_c(t) + \alpha(t)[x(t) - m_c(t)] \\ &\quad \text{if } x \text{ and } m_c \text{ belong to the same class,} \\ m_c(t+1) &= m_c(t) - \alpha(t)[x(t) - m_c(t)] \\ &\quad \text{if } x \text{ and } m_c \text{ belong to different classes,} \\ m_i(t+1) &= m_i(t) \quad \text{for } i \neq c\,. \end{aligned} \tag{6.4}$$

Here $0 < \alpha(t) < 1$, and $\alpha(t)$ (learning rate) is usually made to decrease monotonically with time. It is recommended that α should already initially be rather small, say, smaller than 0.1. The exact law $\alpha = \alpha(t)$ is not crucial, and $\alpha(t)$ may even be made to decrease linearly to zero, as long as the number of learning steps is sufficient; see, however, Sect. 6.3. If also only a restricted set of training samples is available, they may be applied cyclically, or the samples presented to (6.3)-(6.4) may be picked up from the basic set of training samples at random.

The general idea underlying all LVQ algorithms is *supervised learning*, or the reward-punishment scheme. However, it is extremely difficult to show what the exact convergence limits are. The following discussion is based on the observation that the classical VQ tends to approximate $p(x)$ (or some monotonic function of it). Instead of $p(x)$, we may also consider approximation of *any* other (nonnegative) density function $f(x)$ by the VQ. For instance, let the optimal decision borders, or the Bayesian borders (which divide the signal space into class regions B_k such that the rate of misclassifications is minimized) be defined by (6.1) and (6.2); *all such borders together* are defined by the condition $f(x) = 0$, where

$$\begin{aligned} &\text{for } x \in B_k \text{ and } h \neq k, \\ &f(x) = p(x|x \in S_k)P(S_k) - \max_h \{p(x|x \in S_h)P(S_h)\}\,. \end{aligned} \tag{6.5}$$

Let Fig. 6.2 now illustrate the form of $f(x)$ in the case of scalar x and three classes S_1, S_2, and S_3 being defined on the x axis by their respective distributions $p(x|x \in S_k)P(S_k)$. In Fig. 6.2(a) the optimal Bayesian borders

have been indicated with dotted lines. The function $f(x)$ has zero points at these borders according to (6.5), as shown by Fig. 6.2(b); otherwise $f(x) > 0$ in the three "humps."

If we then use VQ to define the point density of the m_i that approximates $f(x)$, *this density also tends to zero at all Bayesian borders.* Thus VQ and (6.5) together define the Bayesian borders with arbitrarily good accuracy, depending on the number of codebook vectors used.

The optimal values for the m_i in the classical VQ were found by minimizing the average expected quantization error E, and in Sect. 1.5.2 its gradient was found to be

$$\nabla_{m_i} E = -2 \int \delta_{ci} \cdot (x - m_i) p(x) dx \; ; \tag{6.6}$$

here δ_{ci} is the Kronecker delta and c is the index of the m_i that is closest to x (i.e., the "winner"). The gradient step of vector m_i is

$$m_i(t+1) = m_i(t) - \lambda \cdot \nabla_{m_i(t)} E \; , \tag{6.7}$$

where λ defines the step size, and the so-called sample function of $\nabla_{m_i} E$ at step t is $\nabla_{m_i(t)} E = -2\delta_{ci}[x(t) - m_i(t)]$. One result is obvious from (6.6): only the "winner" $m_c(t)$ should be updated, while all the other $m_i(t)$ are left intact during this step.

If $p(x)$ in E is now replaced by $f(x)$, the gradient steps must be computed separately in the event that the sample $x(t)$ belongs to S_k, and in the event that $x(t) \in S_h$, respectively.

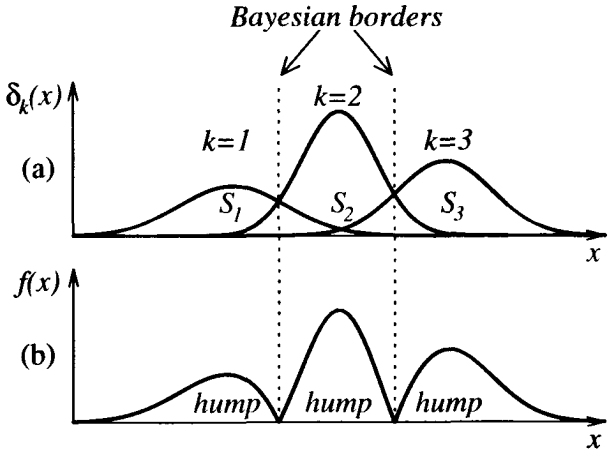

Fig. 6.2. **(a)** Distribution of scalar samples in three classes S_1, S_2, and S_3. **(b)** Illustration of the respective function $f(x)$

The gradient of E, with $p(x)$ replaced by $f(x)$, is

$$\begin{aligned} \nabla_{m_i} E &= -2 \int \delta_{ci}(x - m_i) f(x) dx \\ &= -2 \int \delta_{ci}(x - m_i)[p(x|x \in S_k)P(S_k) \\ &\quad - \max_h \{p(x|x \in S_h)P(S_h)\}] dx \; . \end{aligned} \tag{6.8}$$

In the event that $x(t) \in S_k$ we thus obtain for the sample function of $\nabla_{m_i} E$ *with the a priori probability* $P(S_k)$:

$$\nabla_{m_i(t)} E = -2\delta_{ci}[x(t) - m_i(t)] \; . \tag{6.9}$$

If the class with $\max_h \{p(x|x \in S_h)P(S_h)\}$ is signified by index r meaning the "runner up" class, and in the event that $x(t) \in S_r$, the sample function of $\nabla_{m_i} E$ is obtained with the *a priori* probability $P(S_r)$ and reads

$$\nabla_{m_i(t)} E = +2\delta_{ci}[x(t) - m_i(t)] \; . \tag{6.10}$$

Rewriting $\alpha(t) = 2\lambda$, there results

$$\begin{aligned} m_c(t+1) &= m_c(t) + \alpha(t)[x(t) - m_c(t)] \quad \text{if } x(t) \in B_k \text{ and } x(t) \in S_k \; , \\ m_c(t+1) &= m_c(t) - \alpha(t)[x(t) - m_c(t)] \quad \text{if } x(t) \in B_k \text{ and } x(t) \in S_r \; , \\ m_c(t+1) &= m_c(t) \quad \text{if } x(t) \in B_k \text{ and } x(t) \in S_h \; , \; h \neq r \; , \\ m_i(t+1) &= m_i(t) \quad \text{if } i \neq c. \end{aligned} \tag{6.11}$$

If the m_i of class S_k were already within B_k, and we take into account the humped form of $f(x)$ (Fig. 6.2b), the $m_i \in S_k$ would further be attracted by VQ to the hump corresponding to the B_k, at least if the learning steps are small.

Near equilibrium, close to the borders at least, (6.4) and (6.11) can be seen to define almost similar corrections; notice that in (6.4), *the classification of x was approximated by the nearest-neighbor rule*, and this approximation will be improved during learning. Near the borders the condition $x \in S_r$ is approximated by looking for which one of the m_i is second-closest to x; in the middle of B_k this method cannot be used, but there the exact values of the m_i are not important. However, notice that in (6.4), the minus-sign corrections were made every time when x was classified incorrectly, whereas (6.11) only makes the corresponding correction if x is exactly in the runner-up class. This difference may cause a small bias to the asymptotic values of the m_i in the LVQ1. As a matter of fact, the algorithms called LVQ2 and LVQ3 that will be discussed below are even closer to (6.11) in this respect. *Two* codebook vectors m_i and m_j that are the nearest neighbors to x are eventually updated in them at every learning step; the one that is classified correctly is provided with the plus sign, whereas in the case of incorrect classification the correction is opposite.

6.3 The Optimized-Learning-Rate LVQ1 (OLVQ1)

The basic LVQ1 algorithm will now be modified in such a way that an individual learning-rate factor $\alpha_i(t)$ is assigned to each m_i, whereby we obtain the following learning process [3.15]. Let c be defined by (6.3). Then we assume that

$$\begin{aligned} m_c(t+1) &= m_c(t) + \alpha_c(t)[x(t) - m_c(t)] \text{ if } x \text{ is classified correctly,} \\ m_c(t+1) &= m_c(t) - \alpha_c(t)[x(t) - m_c(t)] \text{ if } x \text{ is classified incorrectly,} \\ m_i(t+1) &= m_i(t) \text{ for } i \neq c. \end{aligned} \tag{6.12}$$

The problem is whether the $\alpha_i(t)$ can be determined optimally for fastest convergence of (6.12). We express (6.12) in the form

$$m_c(t+1) = [1 - s(t)\alpha_c(t)]m_c(t) + s(t)\alpha_c(t)x(t) , \tag{6.13}$$

where $s(t) = +1$ if the classification is correct, and $s(t) = -1$ if the classification is wrong. It may be obvious that the *statistical accuracy* of the learned codebook vector values is approximately optimal if all samples have been used with equal weight, i.e, if the effects of the corrections made at different times, when referring to the end of the learning period, are of approximately equal magnitude. Notice that $m_c(t+1)$ contains a trace of $x(t)$ through the last term in (6.13), and traces of the earlier $x(t'), t' = 1, 2, \ldots, t-1$ through $m_c(t)$. In a learning step, the magnitude of the last trace of $x(t)$ is scaled down by the factor $\alpha_c(t)$, and, for instance, during the same step the trace of $x(t-1)$ has become 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:

$$\alpha_c(t) = [1 - s(t)\alpha_c(t)]\alpha_c(t-1) \ . \tag{6.14}$$

If this condition is made to hold for all t, by induction it can be shown that the traces collected up to time t of all the earlier $x(t')$ will be scaled down by an equal amount at the end, and thus the 'optimal' values of $\alpha_i(t)$ determined by the recursion

$$\alpha_c(t) = \frac{\alpha_c(t-1)}{1 + s(t)\alpha_c(t-1)} \ . \tag{6.15}$$

A precaution is necessary, however: since $\alpha_c(t)$ can also increase, it is especially important that it shall not rise above the value 1. This condition can be imposed in the algorithm itself. For the initial values of the α_i one may then take 0.5, but it is almost as good to start with something like $\alpha_i = 0.3$.

It must be warned that (6.15) is not applicable to the LVQ2 algorithm, since thereby the α_i, on the average, would not decrease, and the process would not converge. The reason for this is that LVQ2 is only a *partial* approximation of (6.11), whereas LVQ3 will be more accurate and probably could be modified like LVQ1. If LVQ3 is modified, then it should be called "OLVQ3."

6.4 The Batch-LVQ1

The basic LVQ1 algorithm can be written in a compressed form as

$$\begin{aligned} m_i(t+1) &= m_i(t) + \alpha(t)s(t)\delta_{ci}[x(t) - m_i(t)] \ , \\ \text{where } s(t) &= +1 \text{ if } x \text{ and } m_c \text{ belong to the same class,} \\ \text{but } s(t) &= -1 \text{ if } x \text{ and } m_c \text{ belong to different classes.} \end{aligned} \tag{6.16}$$

Here δ_{ci} is the Kronecker delta ($\delta_{ci} = 1$ for $c = i$, $\delta_{ci} = 0$ for $c \neq i$).

The LVQ1 algorithm, like the SOM, can be expressed as a batch version. In a similar way as with the Batch Map (SOM) algorithm, the equilibrium condition for the LVQ1 is expresssed as

$$\forall i, \ \mathrm{E}_t \{s\delta_{ci}(x - m_i^*)\} = 0 \ . \tag{6.17}$$

The computing steps of the so-called *Batch-LVQ1* algorithm (in which at steps 2 and 3, the class labels of the nodes are redefined dynamically) can then be expressed, in analogy with the Batch Map, as follows:

1. For the initial reference vectors take, for instance, those values obtained in the preceding unsupervised SOM process, where the classification of $x(t)$ was not yet taken into account.
2. Input the $x(t)$ again, this time listing the $x(t)$ *as well as their class labels* under each of the corresponding 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 the $x(t)$ by the corresponding factors $s(t)$ that indicate whether $x(t)$ and $m_c(t)$ belong to the same class or not.
5. At each node i, take for the new value of the reference vector the entity

$$m_i^* = \frac{\sum_{t'} s(t')x(t')}{\sum_{t'} s(t')} \ , \tag{6.18}$$

   where the summation is taken over the indices t' of those samples that were listed under node i.
6. Repeat from 2 a few 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 usual LVQ, the labeling of the nodes was allowed to change in the iterations. This has sometimes yielded slightly better classification accuracies than if the labels of the nodes were fixed at first steps. Alternatively, the labeling can be determined permanently immediately after the SOM process.

6.5 The Batch-LVQ1 for Symbol Strings

Consider that $S = x(i)$ is the fundamental set of strings $x(i)$ that have been assigned to different classes. Let m_i denote one of the reference strings. The identity vs. nonidentity of the classes of $x(i)$ and m_i shall be denoted by $s(i)$, as in Sect. 6.4. Then the equilibrium condition that corresponds to (6.17) in 6.4 is assumed to read

$$\forall i, \quad \sum_{x(i) \in S} s(i) d[x(i), m_i^*] = \min! \,, \tag{6.19}$$

where d is some distance measure defined over all possible inputs and models, $s(i) = +1$ if $x(i)$ and m_i^* belong to the same class, but $s(i) = -1$ if $x(i)$ and m_i^* belong to different classes.

In accordance with the Batch-LVQ1 procedure introduced in Sect. 6.4 we obtain the *Batch-LVQ1 for strings* by application of the following computational steps:

1. For the initial reference strings take, for instance, those strings obtained in the preceding SOM process.
2. Input the classified sample strings once again, listing the strings as well as their class labels under the winner nodes.
3. Determine the labels of the nodes according to the majorities of the class labels in these lists.
4. For each string in these lists, compute an expression equal to the left side of (6.19), where the distance of the string from every other string in the same list is provided with the plus sign, if the class label of the latter sample string agrees with the label of the node, but with the minus sign if the labels disagree.
5. Take for the set median in each list the string that has the smallest sum of expressions defined at step 4 with respect to all other strings in the respective list. Compute the generalized median by systematically varying each of the symbol positions in the set median by replacement, insertion, and deletion of a symbol, accepting the variation if the sum of distances (provided with the same plus and minus signs that were used at the previous step) between the new reference string and the sample strings in the list is decreased.
6. Repeat steps 1 through 5 a sufficient number of times.

6.6 The LVQ2 (LVQ2.1)

The classification decision in this algorithm is identical with that of the LVQ1. In learning, however, *two* codebook vectors m_i and m_j that are the nearest neighbors to x are now updated simultaneously. One of them must belong to the correct class and the other to a wrong class, respectively. Moreover, x must fall into a zone of values called a 'window' that is defined around the midplane of m_i and m_j. Assume that d_i and d_j are the Euclidean distances of x from m_i and m_j, respectively; then x is defined to fall in a 'window' of relative width w if

$$\min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s, \text{ where } s = \frac{1-w}{1+w} \,. \tag{6.20}$$

A relative 'window' width w of 0.2 to 0.3 is recommended. The version of LVQ2 called LVQ2.1 below is an improvement of the original LVQ2 algorithm [2.39] in the sense that it allows *either* m_i *or* m_j be the closest codebook vector to x, whereas in the original LVQ2, m_i had to be closest.

Algorithm LVQ2.1:

$$\begin{aligned} m_i(t+1) &= m_i(t) - \alpha(t)[x(t) - m_i(t)] \,, \\ m_j(t+1) &= m_j(t) + \alpha(t)[x(t) - m_j(t)] \,, \end{aligned} \tag{6.21}$$

where m_i and m_j are the two closest codebook vectors to x, whereby x and m_j belong to the same class, while x and m_i belong to different classes, respectively. Furthermore x must fall into the 'window.'

6.7 The LVQ3

The LVQ2 algorithm was based on the idea of *differentially* shifting the decision borders toward the Bayesian limits, while no attention was paid to what might happen to the location of the m_i in the long run if this process were continued. Therefore it seems necessary to introduce corrections that ensure that the m_i continue approximating the class distributions, or more accurately the $f(x)$ of (6.5), at least roughly. Combining the earlier ideas, we now obtain an improved algorithm [6.1–3] that may be called LVQ3:

$$\begin{aligned} m_i(t+1) &= m_i(t) - \alpha(t)[x(t) - m_i(t)] \,, \\ m_j(t+1) &= m_j(t) + \alpha(t)[x(t) - m_j(t)] \,, \end{aligned}$$

where m_i and m_j are the two closest codebook vectors to x, whereby x and m_j belong to the same class, while x and m_i belong to different classes, respectively; furthermore x must fall into the 'window';

$$m_k(t+1) = m_k(t) + \epsilon\alpha(t)[x(t) - m_k(t)] \,, \tag{6.22}$$

for $k \in \{i, j\}$, if x, m_i, and m_j belong to the same class.

In a series of experiments, applicable values of ϵ between 0.1 and 0.5 were found, relating to $w = 0.2$ or 0.3. The optimal value of ϵ seems to depend on the size of the window, being smaller for narrower windows. This algorithm seems to be self-stabilizing, i.e., the optimal placement of the m_i does not change in continued learning.

Comment. If the idea were only to approximate the humps of $f(x)$ in (6.5) as accurately as possible, we might also take $w = 1$ (no window at all), whereby we would have to use the value $\varepsilon = 1$.

6.8 Differences Between LVQ1, LVQ2 and LVQ3

The three options for the LVQ-algorithms, namely, the LVQ1, the LVQ2 and the LVQ3 yield almost similar accuracies in most statistical pattern recognition tasks, although a different philosophy underlies each. The LVQ1 and the LVQ3 define a more robust process, whereby the codebook vectors assume stationary values even after extended learning periods. For the LVQ1 the learning rate can approximately be optimized for quick convergence (as shown in Sect. 6.3). In the LVQ2, the relative distances of the codebook vectors from the class borders are optimized whereas there is no guarantee of the codebook vectors being placed optimally to describe the forms of the class distributions. Therefore the LVQ2 should only be used in a *differential* fashion, using a small value of learning rate and a restricted number of training steps.

6.9 General Considerations

In the LVQ algorithms, vector quantization is not used to approximate the density functions of the class samples, but to directly define the class borders according to the nearest-neighbor rule. The accuracy achievable in any classification task to which the LVQ algorithms are applied and the time needed for learning depend on the following factors:

- an approximately optimal number of codebook vectors assigned to each class and their initial values,
- the detailed algorithm, a proper learning rate applied during the steps, and a proper criterion for the stopping of learning.

Initialization of the Codebook Vectors. Since the class borders are represented piecewise linearly by segments of midplanes between codebook vectors of neighboring classes (a subset of borders of the Voronoi tessellation), it may seem to be a proper strategy for optimal approximation of the borders that the average distances between the adjacent codebook vectors (which depend on their numbers per class) should be the same on both sides of the borders. Then, at least if the class distributions are symmetric, this means that the average shortest distances of the codebook vectors (or alternatively, the medians of the shortest distances) should be the same everywhere in every class. Because, due to unknown forms of the class distributions, the final placement of the codebook vectors is not known until at the end of the learning process, their distances and thus their optimal numbers cannot be determined before that. This kind of assignment of the codebook vectors to the various classes must therefore be made *iteratively*.

In many practical applications such as speech recognition, even when the a priori probabilities for the samples falling in different classes are very different, a very good strategy is thus to start with the same number of codebook vectors in each class. An upper limit to the total number of codebook vectors is set by the restricted recognition time and computing power available.

For good piecewise linear approximation of the borders, the medians of the shortest distances between the codebook vectors might also be selected somewhat smaller than the standard deviations (square roots of variances) of the input samples in all the respective classes. This criterion can be used to determine the minimum number of codebook vectors per class.

Once the tentative numbers of the codebook vectors for each class have been fixed, for their initial values one can use first samples of the real training data picked up from the respective classes. Since the codebook vectors should always remain inside the respective class domains, for the above initial values too one can only accept samples that are not misclassified. In other words, a sample is first tentatively classified against all the other samples in the training set, for instance by the K-nearest-neighbor (KNN) method, and accepted for a possible initial value only if this tentative classification is the same as the class identifier of the sample. (In the learning algorithm itself, however, no samples must be excluded; they must be applied independent of whether they fall on the correct side of the class border or not.)

Initialization by the SOM. If the class distributions have several modes (peaks), it may be difficult to distribute the initial values of the codebook vectors to all modes. Recent experience has shown that it is then a better strategy to first form a SOM, thereby regarding all samples as unclassified, for its initialization. After that the map units are labeled according to the class symbols by applying the training samples once again and taking their labels into account like in the calibration of the SOM discussed in Sect. 3.2.

The labeled SOM is then fine-tuned by the LVQ algorithms to approximate the Bayesian classification accuracy.

Learning. It is recommended that learning always be started with the optimized LVQ1 (OLVQ1) algorithm, which converges very fast; its asymptotic recognition accuracy will be achieved after a number of learning steps that is about 30 to 50 times the total number of codebook vectors. Other algorithms may continue from those codebook vector values that have been obtained in the first phase.

Often the OLVQ1 learning phase alone may be sufficient for practical applications, especially if the learning time is critical. However, in an attempt to ultimately improve recognition accuracy, one may continue with the basic LVQ1, the LVQ2.1, or the LVQ3, using a low initial value of learning rate, which is then the same for all the classes.

Stopping Rule. It often happens that the neural-network algorithms 'overlearn'; e.g., if learning and test phases are alternated, the recognition accuracy is first improved until an optimum is reached. After that, when learning is continued, the accuracy starts to decrease slowly. A possible explanation of this effect in the present case is that when the codebook vectors become very specifically tuned to the training data, the ability of the algorithm to generalize for new data suffers. It is therefore necessary to stop the learning process after some 'optimal' number of steps, say, 50 to 200 times the total number of the codebook vectors (depending on particular algorithm and learning rate). Such a stopping rule can only be found by experience, and it also depends on the input data.

Let us recall that the OLVQ1 algorithm may generally be stopped after a number of steps that is 30 to 50 times the number of codebook vectors.

6.10 The Hypermap-Type LVQ

The principle discussed in this section could apply to both LVQ and SOM algorithms. The central idea is selection of candidates for the "winner" in sequential steps. In this section we only discuss the Hypermap principle in the context of LVQ algorithms.

The idea suggested by this author [6.4] is to recognize a pattern that occurs in the context of other patterns (Fig. 6.3). The context around the pattern is first used to select a *subset of nodes* in the network, from which the best-matching node is then identified on the basis of the pattern part. The contexts need not be specified at high accuracy, as long as they can be assigned to a sufficient number of descriptive clusters; therefore, their representations can be formed by unsupervised learning, in which unclassified raw data are used for training. On the other hand, for best accuracy in the final classification of the pattern parts, the representations of the latter should be formed in a supervised learning process, whereby the classification of the training data must be well-validated. *This kind of context level hierarchy can be continued for an arbitrary number of levels.*

The general idea of the Hypermap principle is thus that if different sources of input information are used, then each of the partial sources of information only defines a set of possible *candidates* for best-matching cells; the final classification decision is suspended until all the sources have been utilized. Each of the neural cells is assumed to receive inputs from a number of different kinds of sources, but these subsets of inputs are not used simultaneously. Instead, after application of one particular signal set to the respective inputs, the neural cell is only left in a state of *preactivation*, in which it will not yet elicit an output response. Preactivation is a kind of binary memory state or bias of the neural cell, in which the latter is facilitated to be triggered by further inputs. Without preactivation, however, later triggering would be impossible. There may be one or more phases in preactivation. When a particular set of inputs has been used, information about its accurate input signal values will be "forgotten", and only the facilitatory (binary) information of the cell is memorized; so preactivation is not a bias for "priming" like in some other neural models. One can thereby discern that no linear vector sum of the respective input signal groups is formed at any phase; combination of the signals is done in a very nonlinear but at the same time robust way. Only in the last decision operation, in which the final recognition result is specified, the decision operation is based on continuous-valued discriminant functions.

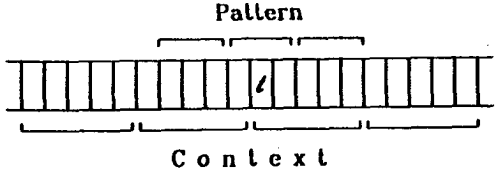

Fig. 6.3. Definition of pattern and context parts in a time series of samples

Example: Two-Phase Recognition of Phonemes. The operation of the LVQ-Hypermap is now exemplified in detail by the following phoneme recognition experiment. It must be emphasized, however, that the principle is much more general. Consider Fig. 6.3, which illustrates a time sequence of samples, such as spectra. For higher accuracy, however, it is better to use in this experiment another feature set named the *cepstrum*, Sects. 7.2 and 7.5. We have used such features, so-called cepstral coefficients (e.g., 20 in number) instead of simple spectra. Assume that the phonemic identity of the speech signal shall be determined at time t. A pattern vector x_{patt} can be formed, e.g., as a concatenation of three parts, each part being the average of three adjacent cepstral feature vectors, and these nine samples are centered around time t. In order to demonstrate the Hypermap principle, x_{patt} is now considered in the context of a somewhat wider context vector x_{cont}, which is formed as a concatenation of four parts, each part being the average of five adjacent cepstral feature vectors. The context "window" is similarly centered around time t.

Figure 6.4 illustrates an array of neural cells. Each cell has two groups of inputs, one for x_{patt} and another for x_{cont}, respectively. The input weight vectors for the two groups, for cell i, are then denoted $m_{i,\text{patt}}$ and $m_{i,\text{cont}}$, respectively. We have to emphasize that the cells in this "map" were not yet ordered spatially: they must still be regarded as a set of unordered "codebook vectors" like in classical Vector Quantization.

Classification of the Input Sample. For simplicity, the classification operation is explained first, but for that we must assume that the input weights have already been formed, at least tentatively. The matching of x and m_i is determined in two phases.