A New Vector Quantization Clustering Algorithm

WILLIAM H. EQUITZ, MEMBER, IEEE

Abstract—**The Pairwise Nearest Neighbor (PNN) algorithm is presented as an alternative to the Linde–Buzo–Gray (generalized Lloyd) algorithm for vector quantization clustering. The PNN algorithm derives a vector quantization codebook in a diminishingly small fraction of the time previously required, without sacrificing performance. In addition, the time needed to generate a codebook grows only like $O(N \log N)$ in training set size, and is independent of the number of code words desired. Using this new method, one can either minimize the number of code words needed subject to a maximum allowable distortion or minimize the distortion subject to a maximum rate. The PNN algorithm can be used with squared error and weighted squared error distortion measures. Simulations on a variety of images encoded at 1/2 bit per pixel indicate that PNN codebooks can be developed in roughly 5 percent of the time required by the LBG algorithm.**

I. Vector Quantization

VECTOR quantization [1], [2] is a process in which data to be encoded are broken into small "blocks," or vectors, which are then sequentially encoded vector by vector. The idea is to identify a set, or "codebook," of possible vectors which are representative of the information to be encoded. The vector quantization encoder pairs up each source vector with the closest matching vector from the codebook, thus "quantizing" it. The actual encoding is then simply a process of sequentially listing the identity of the code words which were deemed to most closely match the vectors making up the original data. The decoder has a codebook identical to the encoder, and decoding is a trivial matter of piecing together the vectors whose identity has been specified. The key to this method, of course, is to have a good codebook of representative vectors, typical of the data to be sent. To date, the method used almost exclusively for developing a codebook has been the algorithm known as the Linde–Buzo–Gray (LBG) algorithm [3]. This algorithm is also sometimes referred to as the generalized Lloyd algorithm (GLA), since it is a vector generalization of a clustering algorithm due to Lloyd [4].

A. The Generalized Lloyd, or "LBG," Algorithm

The LBG algorithm for deriving a codebook based on a set of training vectors is iterative, and can be described as follows. The initialization step involves choosing the starting codebook of vectors. This could be a codebook used previously, or something arbitrary, such as evenly spaced points in the vector space. The iteration begins by assigning each training vector to its "best fit" code word, based on some distortion measure and an exhaustive search. Next, given the set of vectors assigned to a particular code, that code is modified to minimize its error relative to the training vectors currently assigned to it. This two step process continues iteratively. The process is terminated when the overall error between the training vectors and the codes they are assigned to changes by a small enough fraction between one iteration and the next. The codebook is then considered to be determined and, given arbitrary error tolerances, this codebook reduces the coding error to at least a local minimum. The execution time of this algorithm is uncertain because the required number of iterations cannot be predicted ahead of time. Experience indicates that execution time grows quickly as the training set gets larger, as the number of code words increases, and as the vector dimension increases.

In this algorithm, by far the most complicated task is to come up with an acceptable initial codebook. Since the LBG algorithm can only find *local* minima, it is important that one start out in the general neighborhood of the correct solution, lest one become stranded at a local minimum distant from the global minimum. As stated previously, one possible way to initialize the LBG algorithm is to use a codebook previously developed for some other purpose. Alternatively, one might initialize with evenly spaced code words in the vector space or use a so-called "splitting" technique as described in [1]. However, perhaps the best simple initialization is to randomly choose a sampling from the training set for use as the initial codes. In practice, the "random initialization" is typically implemented by choosing evenly spaced elements in the training sequence (e.g., $\vec{x}$, $\vec{x}_{k+1}$, $\vec{x}_{2k+1}$, $\cdots$, $\vec{x}_{(C-1)k+1}$ where $\vec{x}_i$ is the ith training vector, N = the number of training vectors, C = the number of code words desired, and $k = N/C$).

B. Other Clustering Algorithms

Other authors have mentioned that one could develop vector quantization codebooks based on pattern recognition "clustering" techniques, and at least one has been implemented [5]. However, the efficacy of these techniques has not been demonstrated, and these techniques often suffer from the defect that they cannot specify ahead of time how many clusters will result. Additionally, many typical clustering algorithms are no less complicated than the LBG algorithm and suffer from the defect that they

Manuscript received April 15, 1987; revised December 17, 1988. This paper is based in part on work supported by a National Science Foundation Graduate Fellowship, and was presented in part at 1987 ICASSP.

The author is with the IBM Almaden Research Center, 650 Harry Rd., Department K52/802, San Jose, CA 95120-6099.

IEEE Log Number 8929999.

0096-3518/89/1000-1568$01.00 © 1989 IEEE

cluster points with the aim of classifying the points rather than minimizing reconstruction error.

II. Pairwise Nearest Neighbor Clustering

In this section, a new algorithm, the Pairwise Nearest Neighbor (PNN) algorithm, is presented as a substitute for the LBG algorithm. This new algorithm significantly reduces needed computation without sacrificing performance. This algorithm can also be considered as an initializer for the LBG algorithm, providing better performance than either algorithm can achieve separately. The PNN algorithm is designed for use with squared error and weighted squared error distortion measures. The full search version of this algorithm will first be presented, and then an efficient approximation will be described. A preliminary version of this work first appeared in [6].

A. Full Search Pairwise Nearest Neighbor Clustering

The process of generating vector quantization code words from a training set is equivalent to the process of grouping the training set into "clusters," where each cluster is to be represented by a single code word. The Pairwise Nearest Neighbor (PNN) algorithm begins with a separate cluster for each vector in the training set and merges together two clusters at a time until the desired codebook size is achieved. At the start of the clustering process, one converts N clusters, each containing one vector, into the optimal $(N-1)$ cluster codebook by merging together into a single cluster the two closest training vectors. The code word for this new cluster is chosen to minimize the error incurred by replacing these two vectors with a single code word. In other words, it is the centroid of the two vectors now in the new cluster.

For instance, in Fig. 1 we start with six training vectors of two-dimensional data. We consider each training vector to be a separate cluster. The two components of each vector are represented as x and y coordinates on the graph and each cluster centroid is represented in the diagram with an X. Before any merges, each cluster centroid has the number "1" next to it, signifying that it is made up of just one training vector. After one merging step there are five clusters, with the two closest cluster centroids deleted and replaced by a single cluster centroid half way between the two deleted X's. This new centroid has a "2" next to it, signifying that it now represents two training vectors.

Unfortunately, once clusters have more than one member, things become more complicated. However, given K clusters, we can always optimally move to $K-1$ by merging the two clusters which result in the best tradeoff between merging close clusters and affecting few training vectors. If the members of a cluster can be approximated by their centroid, then this step-by-step optimality will lead us to a good overall clustering.

Consider the example in Fig. 2. This diagram represents a typical merge in the PNN algorithm. We start with five clusters and merge together the clusters consisting of four and one training vectors, respectively. We merge together these two clusters rather than the clusters with four and one-hundred training vectors, which appear "closer," because the larger error introduced to each training vector in our chosen two clusters is outweighed by the fact that our choice of clusters affects many fewer training vectors.

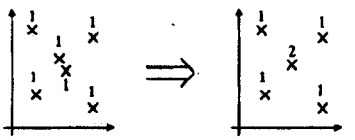

Fig. 1. First merge in PNN algorithm.

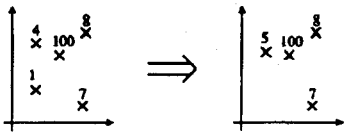

Fig. 2. Typical merge in PNN algorithm.

The clustering process is halted when we are satisfied with our clusters, and then the centroid of each cluster is used as a code word. Stopping criteria are described in more detail below. Of course, it is *not* true that the "optimal" size C codebook will necessarily be achievable by sequentially developing the optimal codebooks of decreasing size, but this is the approximation on which the algorithm is built.

The pair of clusters which will introduce the least error when merged can be calculated as follows. If our distortion measure is squared error (the calculations are similar for weighted squared error), and we use the following notation:

C_i = ith cluster of training vectors
C_{ij} = cluster formed by merging ith and jth clusters
n_i = number of training vectors in C_i
n_{ij} = number of training vectors in C_{ij}
$\bar{x}_i$ = centroid (mean) of the training vectors in C_i
$\bar{x}_{ij}$ = centroid (mean) of the training vectors in C_{ij}
S_i^2 = average squared error between $\bar{x}_i$ and the training vectors in C_i
S_{ij}^2 = average squared error between $\bar{x}_{ij}$ and the training vectors in C_{ij}
$\langle x, y \rangle$ = inner product of x and y,

then

$$n_{ij} = n_i + n_j \tag{1}$$

$$\bar{x}_{ij} = \frac{n_i \bar{x}_i + n_j \bar{x}_j}{n_i + n_j} \tag{2}$$

$$n_{ij} S_{ij}^2 = \sum_{x \in C_{ij}} |x - \bar{x}_{ij}|^2 \tag{3}$$

$$= \sum_{x \in C_i} |x - \bar{x}_{ij}|^2 + \sum_{x \in C_j} |x - \bar{x}_{ij}|^2, \tag{4}$$

where

$$\sum_{x \in C_i} |x - \bar{x}_{ij}|^2 = \sum_{x \in C_i} \left(|x|^2 - 2\langle x, \bar{x}_{ij} \rangle + |\bar{x}_{ij}|^2 \right) \quad (5)$$

$$= n_i \left(S_i^2 + |\bar{x}_i|^2 \right) - 2n_i \langle \bar{x}_i, \bar{x}_{ij} \rangle + n_i |\bar{x}_{ij}|^2 \quad (6)$$

$$= n_i S_i^2 + n_i |\bar{x}_i - \bar{x}_{ij}|^2 \quad (7)$$

$$= n_i S_i^2 + n_i \left| \frac{n_i \bar{x}_i + n_j \bar{x}_i - n_i \bar{x}_i - n_j \bar{x}_j}{n_i + n_j} \right|^2 \quad (8)$$

$$= n_i S_i^2 + n_i \left| \frac{n_j \bar{x}_i - n_j \bar{x}_j}{n_i + n_j} \right|^2 \quad (9)$$

$$= n_i S_i^2 + \frac{n_i n_j^2}{(n_i + n_j)^2} \cdot |\bar{x}_i - \bar{x}_j|^2. \quad (10)$$

Consequently,

$$n_{ij} S_{ij}^2 = n_i S_i^2 + n_j S_j^2 + \frac{n_i n_j^2}{(n_i + n_j)^2} |\bar{x}_i - \bar{x}_j|^2 + \frac{n_i^2 n_j}{(n_i + n_j)^2} |\bar{x}_j - \bar{x}_i|^2 \quad (11)$$

$$= n_i S_i^2 + n_j S_j^2 + \frac{n_i n_j (n_i + n_j)}{(n_i + n_j)^2} |\bar{x}_i - \bar{x}_j|^2 \quad (12)$$

$$= n_i S_i^2 + n_j S_j^2 + \frac{n_i n_j}{n_i + n_j} |\bar{x}_i - \bar{x}_j|^2. \quad (13)$$

We interpret the last term in (13) as the squared error introduced by merging clusters C_i and C_j, and the idea is to choose the clusters C_i and C_j which minimize this quantity. Notice that the only statistics one need keep track of for each cluster are $\bar{x}_i$ and n_i. In fact, C_i can be considered to be a vector of "weight" n_i located at the centroid of the cluster $\bar{x}_i$. In this way, the distortion introduced by merging two clusters can be considered a "weighted distance" between the two centroids.

If one is interested in tracking the error introduced as the clustering proceeds, one might also keep track of S_i^2 for each cluster. However, one should recognize that this value is just an upper bound on the distortion the real encoder will introduce. This is because the calculation above assumes that all the training vectors in a cluster are closer to the centroid of their own cluster than to the centroid of a different cluster. In practice, this may not be the case, and the training vectors may thus be encoded with strictly less distortion. Using this method allows one to terminate the clustering process when a certain distortion relative to the training set is reached, rather than when a certain number of clusters are obtained.

We can now see that there are two possible termination criteria. First, we may choose to terminate when we have reduced our training set to a predetermined number of clusters. Alternately, we may choose to continue merging clusters as long as the average error introduced by representing the training data by the cluster centroids stays below some predetermined threshold. Since the number of clusters determines the coding rate, we see that these termination criteria correspond respectively to minimizing distortion, subject to a rate constraint, and to minimizing rate, subject to a distortion constraint.

B. Fast Search PNN Clustering

The PNN codebook development algorithm is a matter of progressively merging together pairs of clusters with minimal weighted distance between their centroids. The key to quick execution of this algorithm is quickly finding the closest pairs of centroids among an essentially randomly distributed set. The obvious way is to explicitly find each point's nearest neighbor, but this is very slow (a log N search at best) and leads to closest pair computation cost on the order of N log N for each merge. However, it is possible to efficiently find good, if suboptimal, pairs of clusters to merge as described below. We do not care if we merge the absolute closest pair of clusters at each step, as long as close clusters get merged eventually. We call this approximated PNN algorithm the "fast" PNN algorithm.

The way we accomplish the computational saving is to search for a vector's near neighbors only within a small region, with neighborhoods being defined by a *k-d* tree partitioning of the multidimensional space. In the past [6]–[8], *k-d* trees have been used in vector quantization coding as a means of performing the nearest neighbor searches required by the LBG algorithm, but in our case we will merely be using the partition of k-dimensional space induced by organizing the training vectors into a *k-d* tree structure.

1) K-d Trees:

Structure: *K-d* trees (short for k-dimensional trees) were developed by Bentley [9], [10] and provide a data structure which allows for $\log(N)$ multidimensional "nearest-neighbor" searches to be accomplished. *K-d* trees (see Fig. 3) consist of a set of interconnected nodes and a set of terminal nodes, or "buckets," located at the lowest level of the tree. The nodes serve to organize the data, the buckets hold the data. Similar to binary search trees, each node in a *k-d* tree partitions data into two sets based on some scalar threshold, but unlike simple binary trees, Bentley's *k-d* trees partition *vector* data at each node by performing a threshold test on a single coordinate of each vector. The coordinate being tested is the same for all vectors being "partitioned" at a given node, but can be different at each node of the *k-d* tree.

A node in a *k-d* tree is defined by four elements. The first element is the index i of the coordinate used for partitioning the data at this node. The second element is the threshold t for partitioning data. Any vector in the subtree "below" this node whose ith coordinate has a value less than t is located in the left branch of this subtree and,

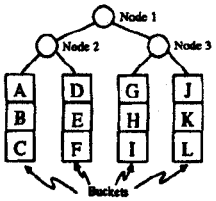

Fig. 3. Example of a *K-d* tree.

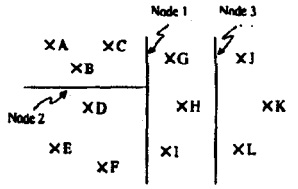

Fig. 4. *K-d* partition corresponding to example.

conversely, any vector whose ith coordinate is greater than t is located in the right branch. It is a matter of convention in which branch a vector belongs if its ith coordinate happens to equal t. The last two elements of a node are pointers to the left and right branches, or children. These children are *k-d* trees in their own right, organizing their "half" of the data.

Buckets are special nodes which point to, or "contain," data rather than other nodes. In a completely organized tree, each bucket would contain only one vector, but usually it is more efficient to have a bucket contain a small number of vectors which are all similar. This way fewer nodes (and levels in the tree) are needed. The vectors in a given bucket can be stored in a finite array if it is guaranteed that there will never be more than a certain number of vectors per bucket or they can be stored as a linked list.

One can consider each particular *k-d* tree to be a partitioning of k-dimensional with each node corresponding to a hyper-plane parallel to all but one coordinate axis. For example, consider the two-dimensional case in Fig. 4 which corresponds to the tree in Fig. 3. Here there are 12 vectors, represented by the letters A–L. The top node acts as the first partition, dividing the two-dimensional region into two half-planes. The next level of the tree divides these half-planes once again, and so on. The lines (hyperplanes in general) dividing up the "space" correspond to nodes, and the regions defined by the nodes correspond to buckets. Each partition could, in theory, be done with respect to any coordinate, but to get the most effective partitions one would normally try to use different coordinates at different nodes.

Building a K-d Tree: K-d trees, like binary trees, are constructed recursively from the top down, and are designed specifically to partition a particular set of data. The first parameter one fixes is the number of vectors one will allow in each bucket. Then, if one wants to build a tree with this many or fewer vectors, they are all assigned to a single bucket. If there are more vectors than will fit into a single bucket, a single coordinate i and a threshold t are chosen and the set of vectors is divided into two halves based on the value of their ith coordinate. As described above, if a vector's ith coordinate is less than t it is assigned to the left child, and if it is greater than t it goes to the right child. If the ith coordinate *equals* t it goes to one child or the other based on convention. Each of these two sets of vectors are then recursively formed into a new *k-d* tree, and so on, until all the vectors are assigned into buckets.

At each node, one is faced with the difficult question of which coordinate to use for partitioning the data. One simple approach is to choose cyclically among the coordinates. On the top level of the tree partition on the basis of the first coordinate; on the second level, use the second, etc. When all the coordinates have been used once, start over again with the first one. A better method is to choose for each node the coordinate which best "spreads out" the data. That is to say, one might choose to partition the data on the basis of the coordinate which has the largest variance associated with it. The reason for choosing this as the "split coordinate" is that if the data are very spread out along a particular dimension, then presumably differences in that coordinate are more "significant" in some sense than differences in another, more densely grouped coordinate. Choosing in this second manner has the effect of splitting the data on the basis of uncorrelated coordinates.

To achieve a maximally balanced tree, one should use the median coordinate value from among the vectors involved as the split threshold. Friedman *et al.* [10] report that a bucket size averaging around eight entries seems to be optimal for a wide range of problems.

2) Using K-d Trees to Perform Fast PNN Clustering: We use *k-d* trees in the following way (see Fig. 5). When looking for pairs of clusters to merge, we only consider pairs where both cluster centroids are assigned to the same region of k-dimensional space as defined by the buckets of a *k-d* tree.

We start by organizing the training set in a *k-d* tree. Then we repeatedly perform the following steps until we are satisfied with our clustering. As usual, we consider the training vectors to be the first set of cluster centroids. After the tree is created, candidate pairs for merging are generated by doing local comparisons within each *k-d* bucket. Within each bucket, the pair of clusters which will introduce the least distortion when merged is called the bucket's "candidate." Next, a fixed fraction of these candidate pairs (such as 50 percent) are merged based on the distortion their merge would introduce. The candidate pair which would introduce the least distortion is always merged, and then the pair which would introduce the second least amount, and so on, until the desired fraction of candidates have been merged. The reason for merging only a fraction of the candidate pairs at each pass is that some partitions (buckets) will not have *any* close pairs. At this point, we can stop if we want, either because we have the correct number of clusters or because we have reached the maximum desired distortion between the

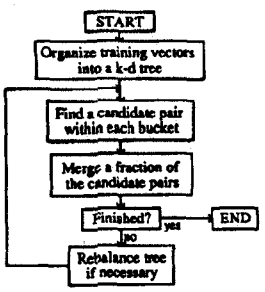

Fig. 5. The PNN algorithm.

training set and our cluster centroids. If we decide to continue, the *k-d* tree is then rebalanced to account for the loss of the merged cluster centroids and the addition of the results of these merges. Buckets which now contain too many cluster centroids are split, and buckets which now contain too few cluster centroids are combined with neighboring buckets. This tree readjustment has the effect of keeping the number of vectors in each bucket roughly constant. With this newly adjusted tree, new candidate pairs are generated and the process continues.

3) Complexity of the Fast PNN Algorithm: Bentley shows that the amount of computation needed to build the *k-d* tree is $O(kN \log N)$ where there are N k-dimensional vectors to be organized, so if T equals the number of vectors in the training set, the complexity of the first step in the fast PNN algorithm is $O(T \log T)$. This computation is performed once.

If there are initially T clusters (one for each training vector) and if at the end there are C clusters, there must be a total of $(T - C)$ merges performed. Since T is typically much larger than C, we lose little by upper bonding the number of merges performed by T. The cost of an individual merge is the cost of deleting two vectors from the tree and adding one. This is of complexity $O(\log T)$ so the complexity of all the merges combined is $O(T \log T)$.

At each pass in the fast PNN algorithm, a fixed fraction of the candidate pairs of cluster centroids are actually merged, so the total number of buckets searched during the entire clustering process is $(T - C)/\gamma$, where γ is the fraction of candidates merged at any given step. Since buckets are kept at a small constant size, searching within a bucket incurs a constant computational cost, and the overall complexity attributable to searching for candidates is $O(T)$. In addition, the operation of determining which candidates to merge is linear in the number of candidates produced at any stage, since it involves only finding a percentile (median, for example) and comparing all candidates to this percentile. Since finding a percentile is linear is the size of the data set, at any stage the complexity of determining which candidates to merge is proportional to the number of buckets searched. As above, the total number of buckets searched for a candidate is $O(T)$, so the overall cost of generating candidate pairs is $O(T)$.

Consequently, the complexity of the PNN clustering process is $O(T \log T)$ in the size of the training set and is essentially independent of the size of codebook generated (even though larger codebooks take slightly less time than small ones).

III. Application to Image Coding

In this section we describe an application of the PNN algorithm to vector quantization coding of still images. We chose image coding because we felt this was the application in which the computational difficulties were most severe. However, in theory the PNN algorithm could serve as an effective substitute for the LBG algorithm in any vector quantization application.

A. Vector Quantization Picture Coding

We used the typical [11]–[14] approach of coding digitized images by dividing them into blocks of pixels and then using these blocks as vectors. Our blocks were 4×4 pixels in size and the images we used were 512×512 pixels. Our pictures were digitized to 8 bits per pixel (256 possible gray levels) and we chose to generate a codebook with 256 codewords, resulting in a coding rate of $1/2$ bit per pixel.

While organizing the training vectors in *k-d* trees, the coordinate used to partition the data at each node in the tree was the coordinate with the greatest variance. The split threshold was the median value of this coordinate. When a child was given eight or fewer vectors they were placed in a bucket.

All tests were performed on either a Vax 11/780 or a Vax 11/750 with programs written in the C programming language. Execution time was measured in Berkeley UNIX accounting units. These "units" are measured in seconds and are similar to CPU seconds, except that they are adjusted for system load. All performance tests shown in a single table were run on the same computer. All tests involving the LBG algorithm were run until there was less than a 0.1 percent change in the distortion introduced by representing the training set with the codebook being developed. Most tests were run using vectors from a single image as the training set, although larger training sets were also tested with similar results.

For LBG coding, a "trick" which increased performance significantly was to initialize not with evenly spaced vectors from the training set beginning with the first vector, but rather with evenly spaced vectors from the training set offset by a small number. The reason for the performance enhancement was that the first training vector corresponded to the block of pixels in the top left corner of the image, and evenly spaced vectors beginning with this vector included a large fraction of vectors lying along the left boundary of the picture. This was a problem because digitized images often have artifacts along the edges corresponding to windowing or other effects, and initializing the LBG algorithm with unrepresentative vectors degraded its performance significantly. We also al-

ways ran the LBG algorithm as enhanced by the use of *k-d* tree nearest neighbor searching. This alone reduced computation by a factor of two as reported in [6] and [7]. We also used other computational enhancements such as those cited in [15] and [16].

B. Full Search PNN versus "Fast" PNN

The "full search" PNN algorithm of Section II-A is a slow and costly process. Since it requires an unacceptable amount of computation, the pertinent question is how much the "fast" PNN algorithm degrades performance as compared to the full search algorithm. It was found that the fast PNN algorithm increased coding error (squared error in this case) by approximately 0.4–0.6 dB for single image encoding. This figure increased as the overall coding error increased.

C. Execution Time of PNN versus LBG

The fast implementation of the PNN algorithm was compared to the LBG algorithm (enhanced to use *k-d* tree searching) and was found in this application to require less than 5 percent of the amount of time needed by LBG algorithm (see Table I). Recall that for these examples, the training set is a single picture ($T = 16{,}384$ for 512×512 pictures by 4×4 codewords), although in practice the training set might be composed of several pictures which belong to a certain "class of pictures."

D. Numerical PNN Quality versus LBG Quality

The quality of images generated by any VQ design algorithm is of great importance because even an extremely fast algorithm is useless if it produces bad pictures. What is desired is a fast alternative with performance equivalent to LBG performance, and as Table II shows, PNN distortion is comparable in all cases. It should be noted that the codebooks generated by the PNN algorithm are suboptimal since the LBG algorithm can always improve on them by running a few iterations. On the other hand, the LBG algorithm is not necessarily globally optimal. A typical picture is shown in Fig. 6 along with the images produced by each algorithm blown up to show detail (see Figs. 6–8). These images were coded at 1/2 bit per pixel.

E. PNN as LBG Initializer

As can be seen by Table II, when the output from the PNN clustering algorithm was used as the initializer for the LBG algorithm, total coding error was lower in all cases than for the LBG codebook with random initialization. In addition, with the PNN initializer the LBG algorithm always converged in fewer iterations (often half as many), so the computation time overall was lower also, as one or two iterations alone take more time than the entire execution of the fast PNN algorithm. Consequently, it is clear that even if the PNN algorithm were to be considered unacceptable because it usually does not generate an "optimal" codebook, it appears to be an excellent alternative to random initialization for the LBG algorithm, and used this way results in excellent performance as well as computational savings.

TABLE I
EXECUTION TIME OF LBG VERSUS PNN

Execution time (in UNIX accounting units)			
Picture	Codebook Development Algorithm		
	LBG with random initialization		fast PNN
	iterations	time (secs)	time (secs)
baboon	17	10020	361
lake	25	8443	363
airport	27	16089	373
lena	25	6930	366
peppers	31	8602	364
plane	33	9577	400

TABLE II
CODED PICTURE ERROR OF LBG VERSUS PNN

Total Squared Pixel Error for Decoded Pictures (x 10^7)			
Picture	Codebook Development Algorithm		
	LBG initialized with random training vectors	fast PNN	LBG initialized with fast PNN codes
baboon	6.55	7.21	6.54
lake	2.60	2.65	2.38
airport	1.75	1.54	1.38
lena	1.17	1.26	1.11
peppers	1.27	1.33	1.20
plane	1.42	1.44	1.25

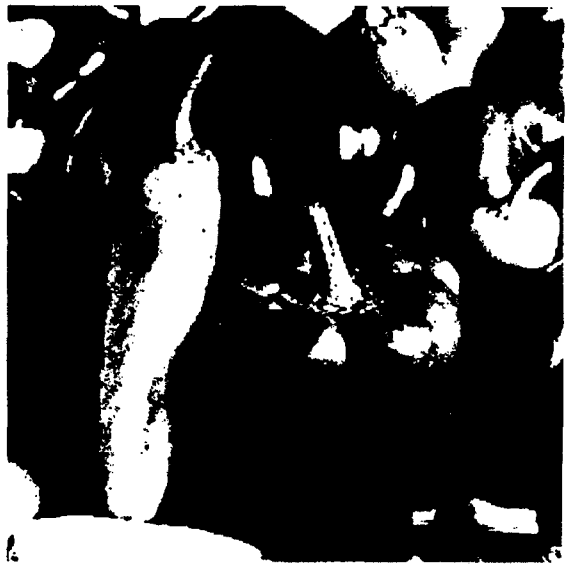

Fig. 6. "Peppers" original.

F. Performance Outside Training Set

Tests were performed to determine if the PNN algorithm would continue to perform comparably with the LBG algorithm when coding pictures outside the training set. In these cases, picture reconstruction was understandably worse, but picture quality with PNN codebooks remained comparable to that achieved using LBG codebooks. The computational advantage of the PNN

Fig. 7. Blowup (250 × 250 pixels) of "Peppers" coded with LBG codebook.

Fig. 8. Blowup (250 × 250 pixels) of "Peppers" coded with fast PNN codebook.

algorithm over the LBG algorithm became more pronounced as the size of the training set increased.

IV. Conclusions

The Pairwise Nearest Neighbor (PNN) algorithm was presented as a noniterative way to generate vector quantization codebooks comparable to those generated by the LBG algorithm. This algorithm is useful for applications involving squared error and weighted squared error distortion measures. The PNN algorithm's main "feature" is that it develops codebooks in a diminishingly small fraction of the time previously required. The PNN algorithm is not sensitive to initializations and is guaranteed to terminate in a finite amount of time. The PNN algorithm is shown to take slightly *less* time the larger the codebook desired and allows one the option of either minimizing coding distortion subject to a rate constraint, or minimizing rate subject to a distortion constraint. In addition, since practical implementations of the PNN algorithm use local nearest neighbor searching, processing can be done in parallel to take advantage of the added speed that extra hardware can bring.

When applied to image coding with small training sets, the time required to execute the PNN algorithm is demonstrated to be just 5 percent of that typically required by the LBG algorithm. Reconstructed pictures generated with this new algorithm were shown to be as good as those generated with the standard algorithm. It is interesting to note that the PNN algorithm makes it computationally feasible to develop a different codebook for each image. Sending the codebook to the receiver before the encoded image would take an additional $1/8$ bit per pixel with parameters as stated.

It was also shown that using PNN code words as an LBG initialization for image coding results in much better codebooks (with fewer iterations) than does "random initialization." This provides for a way to generate better codebooks and indicates that the LBG algorithm using random training vectors as an initializer typically converges to suboptimal codebooks, a fact often not given adequate attention. It is the opinion of the author, however, that the main usefulness of the PNN algorithm is as a fast alternative to the LBG algorithm which allows vector quantization to be used in situations where it had previously been computationally prohibitive, such as in repetitive experimental work, or in situations with large training sets or codebooks.

References

[1] R. M. Gray, "Vector quantization," *IEEE ASSP Magazine*, pp. 4-29, Apr. 1984.
[2] A. Gersho, "On the structure of vector quantizers," *IEEE Trans. Inform. Theory*, vol. IT-28, pp. 157-166, Mar. 1982.
[3] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun.*, vol. COM-28, pp. 84-95, Jan. 1980.
[4] S. P. Lloyd, "Least squares quantization in pcm," *IEEE Trans. Inform. Theory*, vol. IT-28, pp. 129-137, Mar. 1982 (reprint of 1957 paper).
[5] L. Miclet and M. Dabouz, "Low bit rate transmission of speech by vector quantization of the spectrum," Dep. Commun. Ecole Nationale Superieure des Telecommunications, Paris, 1985, unpublished.
[6] W. H. Equitz, "Fast algorithms for vector quantization picture coding," Master's thesis, Mass. Inst. Technol., June 1984.
[7] ——, "Fast algorithms for vector quantization picture coding," in *Proc. ICASSP*, Dallas, TX, Apr. 1987, pp. 18.1.1-18.1.4.
[8] A. Lowry, S. Hossain, and W. Millar, "Binary search trees for vector quantization," in *Proc. ICASSP*, Dallas, TX, Apr. 1987, pp. 51.8.1-51.8.4.
[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509-517, Sept. 1975.

[10] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Software*, vol. 3, no. 3, pp. 209-226, Sept. 1977.
[11] R. L. Baker and R. M. Gray, "Image compression using non-adaptive spatial vector quantization," in *Proc. 16th Asilomar Conf. Circuits, Syst., Comput.*, 1982, pp. 55-61.
[12] T. Murakami, K. Asai, and E. Yamazaki, "Vector quantizer of video signals," *Electron. Lett.*, vol. 18, no. 23, pp. 1005-1006, Nov. 1982.
[13] A. Gersho and B. Ramamurthi, "Image coding using vector quantization," in *Proc. ICASSP*, Paris, France, 1982, pp. 428-431.
[14] Y. Yamada, K. Fujita, and S. Tazaki, "Vector quantizer of video signals," in *Proc. Annu. Conf. IECE*, 1980, p. 1031.
[15] D.-Y. Cheng, A. Gersho, B. Ramamurthi, and Y. Shoham, "Fast search algorithms for vector quantization and pattern matching," in *Proc. ICASSP*, San Diego, CA, Mar. 1984, pp. 9.11.1-9.11.4.
[16] C.-D. Bei and R. M. Gray, "An improvement of the minimum distortion encoding algorithm for vector quantization," *IEEE Trans. Commun.*, vol. COM-33, pp. 1132-1133, Oct. 1985.

William H. Equitz (M'89) was born in Wauwatosa, WI, on August 19, 1961. He received the S.B. degree in mathematics and the M.S. degree in electrical engineering from the Massachusetts Institute of Technology in 1983 and 1984, respectively. He received a National Science Foundation Graduate Fellowship to study at Stanford University beginning in 1984, and received the Ph.D. degree in electrical engineering from Stanford in June 1989.

From 1983 to 1984 he was employed at AT&T Bell Laboratories in Holmdel, NJ, where he worked on vector quantization image coding, and is currently employed at the IBM Almaden Research Center in San Jose, CA, doing work on data compression using arithmetic coding. His current research interests include data compression, rate distortion theory, image processing, and other forms of applied information theory and statistics.

Dr. Equitz is a member of Sigma Xi.