Clustering Algorithms and their Effect on Edge Preservation in Image Compression

Nothabo Elizabeth Ndebele

A thesis submitted in fulfillment of the requirements for the Degree of

Master of Science

in the Department of Statistics at

Rhodes Univeristy

February 2008

Abstract

Image compression aims to reduce the amount of data that is stored or transmitted for images. One technique that may be used to this end is vector quantization. Vectors may be used to represent images. Vector quantization reduces the number of vectors required for an image by representing a cluster of similar vectors by one typical vector that is part of a set of vectors referred to as the codebook. For compression, for each image vector, only the closest codebook vector is stored or transmitted. For reconstruction, the image vectors are again replaced by the the closest codebook vectors. Hence vector quantization is a lossy compression technique and the quality of the reconstructed image depends strongly on the quality of the codebook. The design of the codebook is therefore an important part of the process.

In this thesis we examine three clustering algorithms which can be used for codebook design in image compression: c-means (CM), fuzzy c-means (FCM) and learning vector quantization (LVQ). We give a description of these algorithms and their application to codebook design.

Edges are an important part of the visual information contained in an image. It is essential therefore to use codebooks which allow an accurate representation of the edges. One of the shortcomings of using vector quantization is poor edge representation. We therefore carry out experiments using these algorithms to compare their edge preserving qualities. We also investigate the combination of these algorithms with classified vector quantization (CVQ) and the replication method (RM). Both these methods have been suggested as methods for improving edge representation. We use a cross validation approach to estimate the mean squared error to measure the performance of each of the algorithms and the edge preserving methods.

The results reflect that the edges are less accurately represented than the non - edge areas when using CM, FCM and LVQ. The advantage of using CVQ is that the time taken for codebook design is reduced particularly for CM and FCM. RM is found to be effective where the codebook is trained using a set that has larger proportions of edges than the test set.

Contents

List of Figures

List of Tables

Acknowledgements

The research presented here would not have been possible without the assistance of Professor G. Jäger. Many thanks to him for introducing this work to me and for his guidance and support. I would also like to thank the Rhodes University Department of Statistics, my family and friends who have kept me inspired, assisted and supported me in various ways throughout my research. I am also grateful for the financial assistance that I received from Rhodes University and the Canon Collins Educational Trust.

Clustering Algorithms and their Effect on Edge Preservation in Image Compression

1 Vector Quantization

Vector quantization (VQ) is a coding technique that is used in data compression. We begin this chapter by describing data compression and the necessity of techniques that allow data compression to be performed. A description of VQ as a generalisation of scalar quantization is then given, with a focus on the design and the performance of a vector quantizer in image compression. A new approach to VQ is presented, with VQ being viewed as a multiple regression. The last section then describes the estimation of the mean squared error using cross validation.

1.1 Data compression and quantization

Digital systems are used in a variety of applications in engineering, computer science and other fields [1],[2],[3],[4]. The prevalence of their use for handling data has made it necessary to find ways of improving efficiency of data storage and transmission. In this section the different types of data compression are described briefly. An introduction to scalar quantization is given as a technique applied to data compression.

1.1.1 Data compression

Data compression is a way that has been employed to reduce redundancy in data without losing important information in the data. There are two types of data compression that may be used: lossy and lossless compression [1],[5]. Lossless compression allows all the information from the original data to be restored after decompression [5]. Although the data is in a different form after compression all the information in the data is maintained. This is used to minimise storage or transmission without losing any of the information in the data. An example of the use of lossless compression is Huffman encoding with Morse code [1],[5]. Morse code compresses letters of the alphabet by using short binary codewords for letters that are more likely to be used and long codewords for those less likely. Lossy data compression is used to minimise storage or transmission of data, where

the reconstruction after compression does not contain all the original information [5]. This may be done by eliminating redundancy in the data or using approximations to represent the original data. Lossy compression is used where data may be lost during compression, without a loss of essential information. Joint Photographic Experts Group (JPEG) [5] is a lossy compression technique used in image compression. Although data may come in various forms, of particular interest here is data compression applied to digital signals.

1.1.2 Scalar quantization

Quantization is a technique that may be used for lossy data compression. A quantizer compresses a set of numbers by approximating them using a smaller set of predetermined values. A scalar quantizer Q [1] is a mapping $Q : R \longrightarrow C$ where R is the set of real numbers and

$$C = \{c_1, c_2, ..., c_c\} \subset R$$

is the *codebook* with size $|C| = c$. The values c_i are referred to as the *code words*. Every quantizer is associated with a partition of R into c subsets $U_1, U_2, ..., U_c$ such that $U_i = \{x \in R : Q(x) = c_i\} = Q^{-1}(\{c_i\})$ where U_i is the inverse image of $\{c_i\}$ under Q. For any scalar quantizer we have the following conditions:

- $\bigcup_{i=1}^{c} U_i = R$
- $U_i \cap U_j = \emptyset$ for $i \neq j$

An example of scalar quantization is shown in figure 1.1. Figure 1.1(a) shows a function of x, where $f(x) = sin(x)$ for $x \in [0, 360]$ with $f(x) \in [-1, 1]$. Figure 1.1(b) shows a quantized form of the sine wave where $f(x)$ is now limited to 6 levels, $Q(x) = \{-1, -0.69, -0.3, 0.3, 0.69, 1\}$.

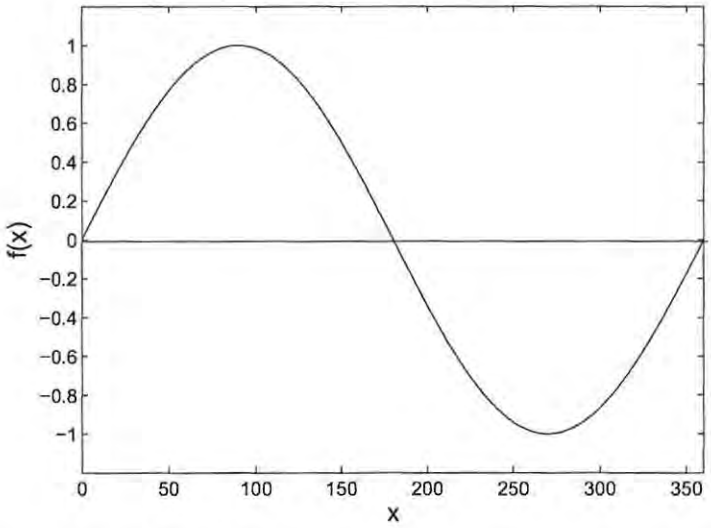
Quantization is not limited to scalar values. This technique may be extended to vectors. The following section gives a description of vector quantization and its application to image compression.

1.2 Vector quantization

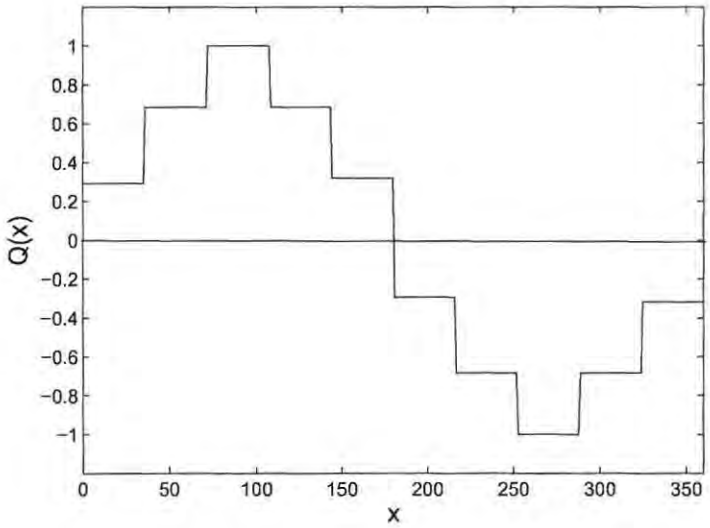
Vector quantization (VQ) is a generalisation of scalar quantization which allows us to perform data compression on vectors. In digital systems, vectors are often used to describe

Figure 1.1: An example of the quantization of a sine wave

(a) Sine wave $f(x) = sin(x)$ for $x \in [0, 360]$



(b) Quantized sine wave $Q(x)$ for $x \in [0, 360]$



data of a variety of types including images or audio and video signals. VQ has been used successfully for applications in speech processing [6], image processing [7],[8], fascimile transmission [2],[9] and weather satellites [9]. VQ is used for the following reasons [1]:

- to minimise the bandwidth that is required to transmit data or to get the best possible quality for data that is transmitted.
- to minimise data storage space or to get the best possible quality of a stored image in a limited space.
- to produce a representation of the data that contains all the necessary information and thus reduce the complexity of any other processing to be done on it.

VQ reduces the number of vectors required to represent information by mapping a set of input vectors $X = \{x_1, x_2, ..., x_N\}$ such that $x_k \in R^n$, to a finite set of predetermined vectors that "approximate" the input vectors [12]. Gersho and Gray [1] define a vector quantizer Q of dimension n and size c as a mapping from an n dimensional Euclidean space R^n onto a finite set C containing c output vectors called *code vectors*. Therefore

$$Q : R^n \longrightarrow C$$

where $C = \{c_1, c_2, ..., c_c\}$ and $c_i \in R^n$ for each $i = 1, 2, ..., c$. The set of code vectors has c distinct elements and is referred to as the *codebook*. Associated with the quantizer is the partition of the set R^n into c subsets denoted by U_i. The *ith* subset of the partition is defined by

$$U_i = \{x \in R^n : Q(x) = c_i\} = Q^{-1}(\{c_i\}) \tag{1.1}$$

and it follows that

$$\bigcup_{i=1}^{c} U_i = R^n \text{ and } U_i \cap U_j = \emptyset \text{ for } i \neq j.$$

Conversely, given a partition of R^n into c pair wise disjoint sets $\{U_1, U_2, ..., U_c\}$, a vector quantizer can be defined as $Q(x) = c_i$ if $x \in U_i$ with fixed $\{c_1, c_2, ..., c_c\} \in R^n$.

There are two main operations that a vector quantizer performs: *encoding* and *decoding*. The *encoder* selects a matching code vector c_i to represent an input vector x_k, which then becomes an element of U_i. An index is then used as a label of the code vector. This index is transmitted to the decoder which generates the code vectors c_i that will be used to represent the corresponding sets U_i from the partition.

A selector function $S_i(x)$ can be defined that describes the operation of the encoder [1]:

$$S_i(x) = \begin{cases} 1, & if\ x \in U_i \\ 0, & otherwise \end{cases} \tag{1.2}$$

$S_i(x)$ is also called the characteristic function of U_i.

The quantizer operation can be represented as follows

$$Q(x) = \sum_{i=1}^{c} c_i S_i(x) \tag{1.3}$$

For any given input vector only one term in the sum is non-zero as the $\{U_i : i = 1, 2, ..., c\}$ form a partition of R^n.

1.3 Vector Quantizer Design

In designing a vector quantizer the aim is to maximise a given measure of performance. The design of the codebook is critical in achieving this aim. It is therefore necessary to find a codebook for which the expected distortion between the input and the code vectors is minimised. Let $X = \{X_1, X_2, ..., X_N\}$ denote a set of continuously distributed random vectors. The expected distortion D is given by

$$D = E(d(X, Q(X))) \tag{1.4}$$

where d is a distortion measure between the input X and the output $Q(X)$ and E the statistical expectation. Distance measures are often used as distortion measures. A codebook that minimises the average distortion satisfies the nearest neighbour condition [1],[10],[11] and the centroid condition [1],[4].

Theorem 1.3.1 (Nearest neighbour (NN) condition):

For a given codebook $C = \{c_1, c_2, ..., c_c\}$ and a vector quantizer Q with $Q(x) = \sum_{i=1}^{c} c_i S_i(x)$, the partition which minimises [1]

$$D = E(d(X, Q(X))$$

where $X = \{x_1, x_2, ..., x_N\}$ *is randomly selected from a distribution* $F \subset R^n$ *is given by*

$$U_i \subset \{x \in R^n : d(x, c_i) \leq d(x, c_j) \ for\ all\ j\}, \tag{1.5}$$

that is

$$Q(x) = c_i\ only\ if\ d(x, c_i) \leq d(x, c_j)\ for\ all\ j \tag{1.6}$$

where $d(x, c_i)$ *is the distortion measure from* x *to* c_i.

Theorem 1.3.2 (The centroid condition):

For a given partition $\{U_i; i = 1, 2, ..., N\}$, *the optimal code vectors satisfy* [1],[3],[10]

$$c_i = cent(U_i)$$

that is the optimal code vectors must be the centroids of the partitions.

$$cent(U_i) = E(X|X \in U_i). \tag{1.7}$$

Clustering algorithms are often used in codebook design. They iteratively generate partitions until the code vectors stabilise. The iterative process is started with an initial codebook which is either chosen randomly or is a subset of the input vectors. A new codebook is then found, which results in a vector quantizer which has a distortion not greater than the previous quantizer [1]. The process continues until there is no significant change in the distortion between the successive iterations. There are several clustering algorithms that can be used in generating codebooks. A description of three algorithms considered in this thesis is given in Chapter 2.

1.4 Measuring the performance of the quantizer

Let $X = \{X_1, X_2, ..., X_N\}$ denote a set of continuously distributed random vectors in R^n. The performance of a quantizer is usually measured using the average distortion between the input vectors X and the output vectors $Q(X)$ that are reproduced using a codebook. In general when comparing quantizers, the quantizer with a smaller distortion is considered better. If the distortion is given by $d(X, Q(X))$ then overall performance

is measured by the long term performance

$$\bar{d} = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} d(X_k, Q(X)_k). \tag{1.8}$$

If the vector process is stationary and ergodic then with a probability of one the limit exists and $\bar{d} = E(d(X, Q(X)))$ where E is the statistical expectation [1],[12].

A measure that is often used to measure the distortion between an input vector X and the quantized vector $Q(X)$ is the squared Euclidean distance which is defined as

$$\begin{aligned} d(X, Q(X)) &= ||X - Q(X)||^2 \\ &= (X - Q(X))'(X - Q(X)) \\ &= \sum_{k=1}^{N} (X_k - Q(X_k))^2 \end{aligned} \tag{1.9}$$

The average distortion for the squared Euclidean distance is then called the mean squared error (MSE) [13] and is given by

$$\begin{aligned} MSE^* &= E(||X - Q(X)||^2) \\ &= \frac{1}{N^2} \sum_{k=1}^{N} (X_k - Q(X_k)) \end{aligned} \tag{1.10}$$

where

$$X = \begin{pmatrix} X_1 \\ X_2 \\ . \\ . \\ . \\ X_N \end{pmatrix} \quad \text{and} \quad Q(X) = \begin{pmatrix} Q(X_1) \\ Q(X_2) \\ . \\ . \\ . \\ Q(X_N) \end{pmatrix}$$

is the input and $Q(X) = \{Q(X_1), Q(X_2), ...Q(X_N)\}$ is the quantized output.

Another measure that can be used particularly for gray scale images with pixel values $\{0, 1, ..., 255\}$ is the peak signal to noise ratio (PSNR) that is inversely related to the MSE [1],

$$PSNR^* = 10 \times log_{10} \left(\frac{255^2}{MSE^*} \right). \tag{1.11}$$

1.5 Vector quantization and image compression

Typically the storage and transmission of digital images requires large amounts of space and a wide bandwidth [1]. Vector quantization is a technique that can be used for image compression. Of particular interest for us here is the use of VQ with gray scale images. A grayscale image is represented digitally by an array of integers in the range $[0, 255]$ (black (0), white (255)). Therefore a digital image is an array of $(x_{r,s})_{r=1,2,...,R,s=1,2,...,S} \in \{0, 1, 2, ..., 255\}^{R \times S}$ where $x_{r,s}$ is the pixel value at location (r, s). For quantization the image array is converted into vectors. This is done by subdividing the image into small blocks of size $p \times q$, where $p = q$ for square blocks. These blocks are then rearranged as vectors with each component representing a pixel in the block. Clustering algorithms may be used to find code vectors that will represent clusters of similar image vectors. Each code vector is assigned a label which is used for storage, transmission or any other processing done to the image. The image is stored or transmitted in the form of an array of labels with the codebook. In decoding the codebook is used to find the code vectors that are associated with the labels. These are then used to reproduce the compressed input. For each image block the code vector used to represent it is found and then instead of the original image vector the code vector is used for reconstruction. The quantized vectors are then converted into blocks to reconstruct the image. Often in practice the original image data is transformed. The fast -Fourier, Z or planar transforms and other transforms [14],[15],[16], may be used to transform the data prior to quantization and the corresponding inverse transform then used for decoding. For a 2×2 block in an image the process would be as follows:

- Convert the block to a vector:

$$\begin{pmatrix} x_{r,s} & x_{r,s+1} \\ x_{r+1,s} & x_{r+1,s+1} \end{pmatrix} \longrightarrow \begin{pmatrix} x_{r,s} \\ x_{r+1,s} \\ x_{r,s+1} \\ x_{r+1,s+1} \end{pmatrix}$$

- Use clustering algorithms to find closest code vector:

$$\begin{pmatrix} x_{r,s} \\ x_{r+1,s} \\ x_{r,s+1} \\ x_{r+1,s+1} \end{pmatrix} \longrightarrow \begin{pmatrix} c_{1i} \\ c_{2i} \\ c_{3i} \\ c_{4i} \end{pmatrix}$$

- Code vector is stored as a label:

$$c_i = \begin{pmatrix} c_{1i} \\ c_{2i} \\ c_{3i} \\ c_{4i} \end{pmatrix} \longrightarrow i$$

- For reconstruction, using the code vector label, look up the code vector to represent the vector and convert to a block:

$$i \longrightarrow c_i = \begin{pmatrix} c_{1i} \\ c_{2i} \\ c_{3i} \\ c_{4i} \end{pmatrix} \longrightarrow \begin{pmatrix} c_{1i} & c_{3i} \\ c_{2i} & c_{4i} \end{pmatrix}$$

Figure 1.2 shows the processes involved in VQ for image compression.

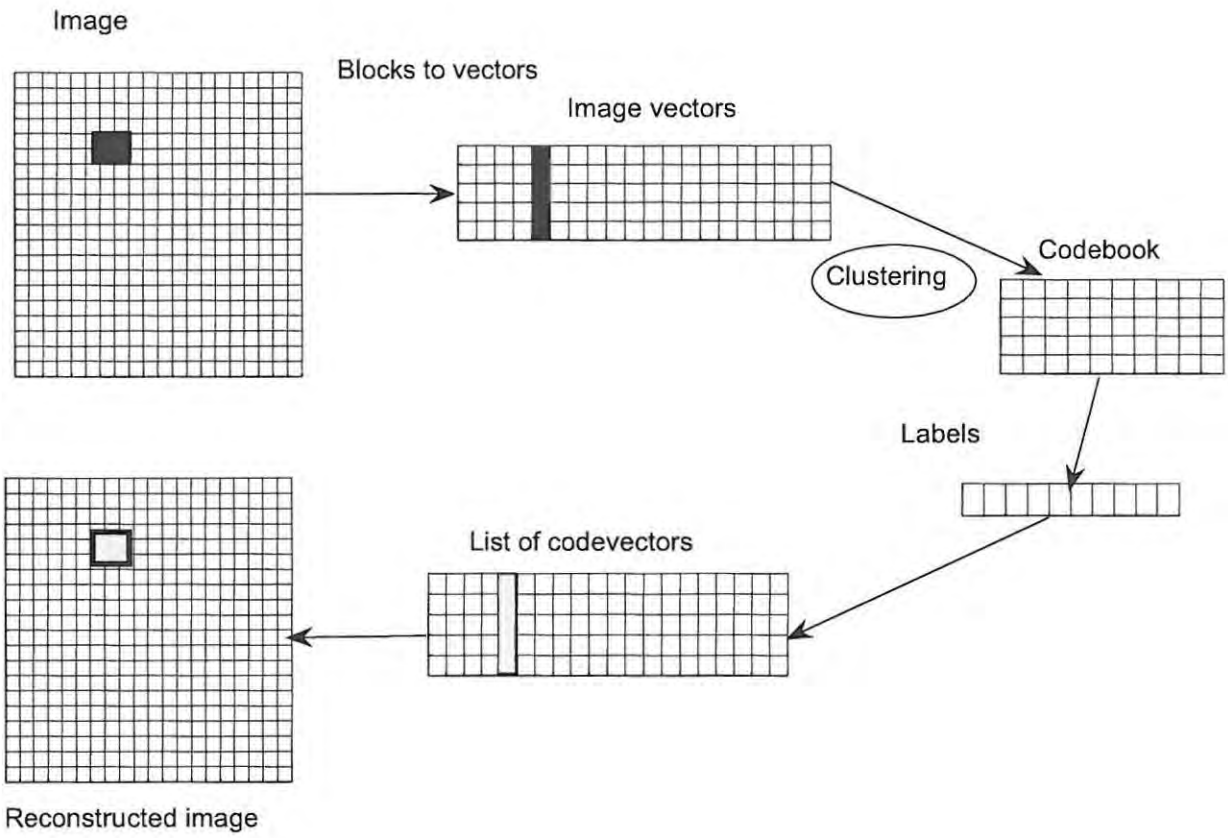
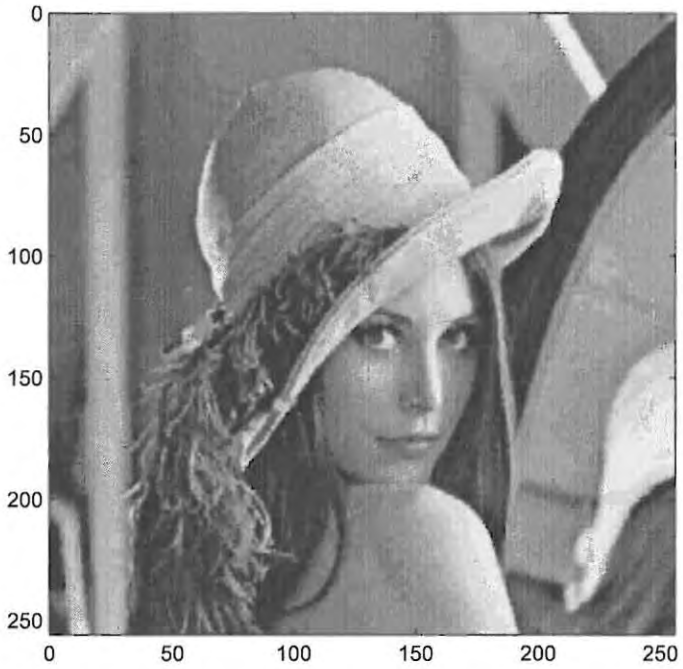


Figure 1.2: Vector quantization

Figure 1.3: 256×256 Lenna image reconstructed using VQ with (a) 2×2 (b) 4×4 blocks
(a) 2×2 blocks



(b) 4×4 blocks

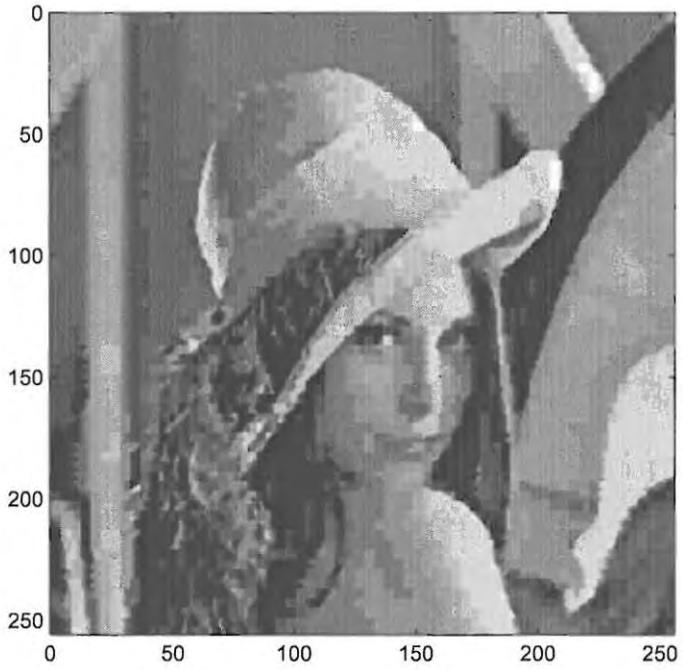
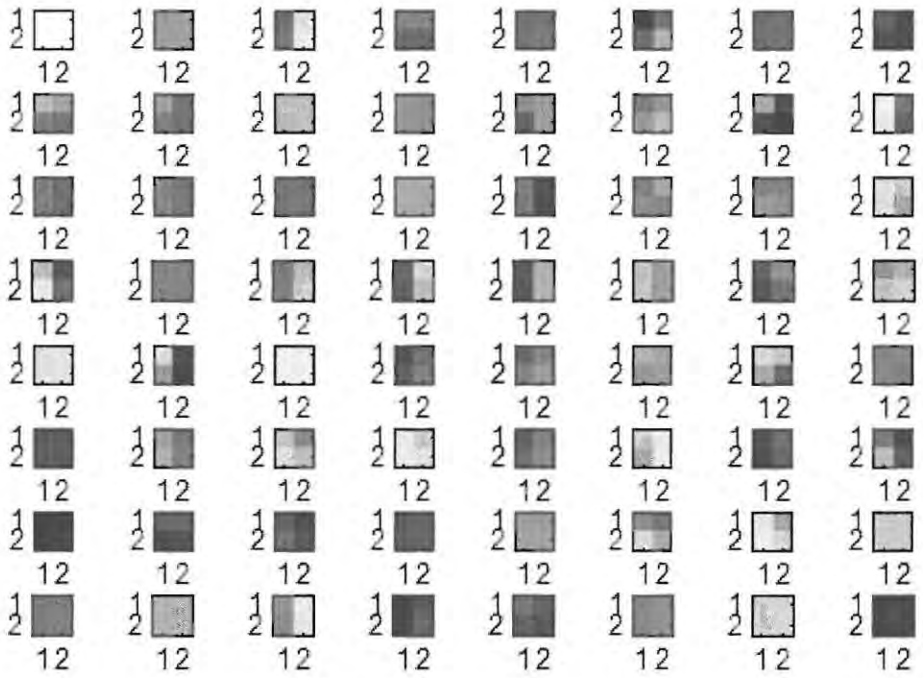


Figure 1.4: Codebooks created using Lenna image, codebook size = 64
(a) Codebook with 2×2 blocks



(b) Codebook with 4×4 blocks

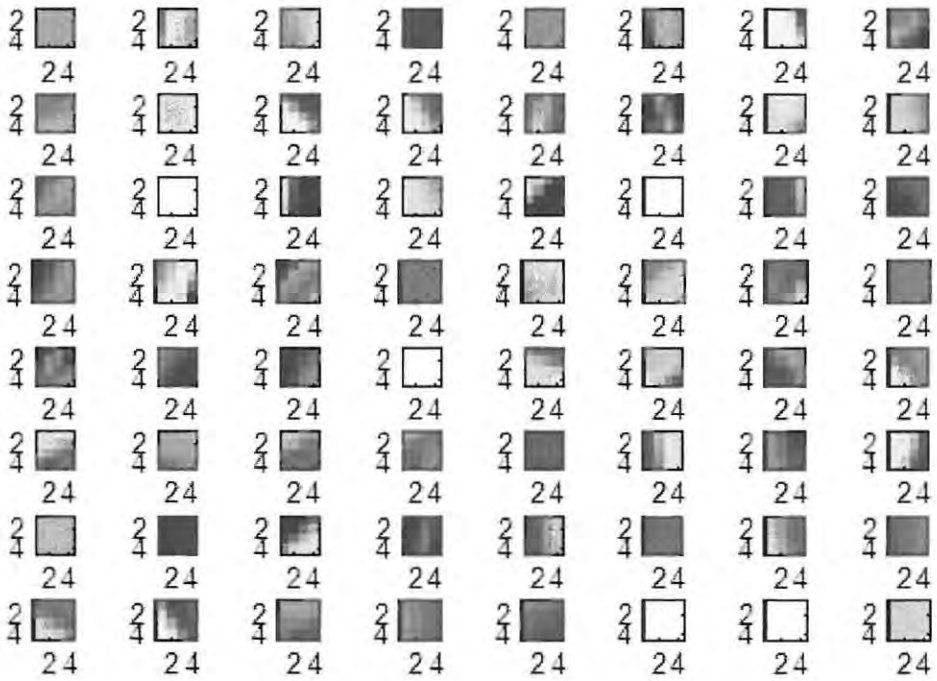


Figure 1.3 shows examples of images compressed using VQ. Each image has a different block size. We observe that the reconstruction is affected by the block size. If the block size of the compressed image is $p \times q = n$, during VQ, a block is converted to vector of size n. The extent to which compression is achieved may be measured using the bit rate R. The bit rate of a quantizer for an image with vectors of dimension n, is given by [1],[17]

$$R = \left(\frac{1}{n}\right) log_2 c \tag{1.12}$$

where c is the number of code vectors in the quantizer and n is the dimension of the code vectors. In figure 1.3(a) 64 code vectors were used with 2×2 blocks, thus the quantizer has a bit rate of

$$\begin{aligned} R &= \left(\frac{1}{4}\right) log_2 64 \\ &= 1.5 \end{aligned}$$

which would be referred to as a bit rate of 1.5 bits per pixel (1.5 bpp).

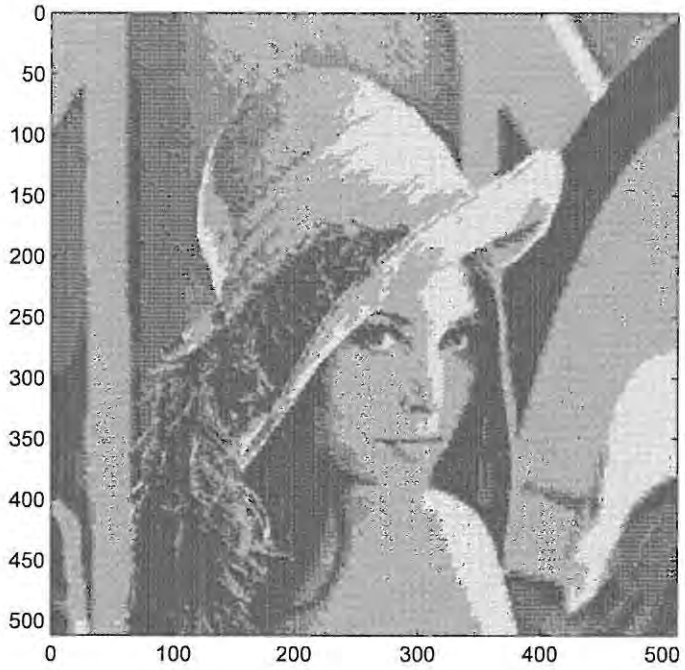
Examples of codebooks are given in figure 1.4. The choice of the codebook is crucial as it determines the quality of the decompressed output. Figure 1.5 shows an image reconstructed from a randomly generated codebook. We observe that the image is a very poor representation of the original image. A good codebook is necessary for an effective compression.

1.6 Vector Quantization as a Multiple Regression

The classical view of vector quantization has been given thus far. Vector quantization however, may also be approached as a vector-valued multiple regression. This idea to the best of my knowledge is new. It is drawn from regression trees [18]. Given a learning task

x_1	x_2	...	...	...	x_N
y_1	y_2	...	...	...	y_N

Figure 1.5: Reconstructed Lenna image using randomly generated codebook, $c = 256$, block size $= 2 \times 2$



where (x_i, y_i) are samples from a distribution on $\mathcal{F} \times \mathcal{L}$ where $\mathcal{F} \subset R^n$ and $\mathcal{L} \subset R^m$ are finite we can approximate the learning task by a function

$$\phi : R^n \rightarrow R^m$$

such that $\phi(x_i) \approx y_i$. The error measure for ϕ would be given by the mean squared error of ϕ which is given by

$$MSE^*(\phi) = E(||Y - \phi(X)||^2)$$

where (X, Y) are randomly chosen from the distribution. If we let $Z = ||Y - \phi(X)||^2$ then Z is a real valued random variable and $MSE^*(\phi) = E(Z)$. Thus an unbiased estimator for the $MSE(\phi)$ is the sample mean

$$MSE(\phi) = \frac{1}{N}\sum_{k=1}^{N} Z_k = \frac{1}{N}\sum_{k=1}^{N} ||Y_k - \phi(X_k)||^2 = \overline{Z}$$

In the context of VQ the class of functions that are involved are limited to vector-valued step functions.

Definition 1.6.1

A function $\phi : R^n \rightarrow R^m$ is a vector-valued step function if the set $\phi(R^n)$ is finite.

If $\phi(R^n) = \{c_1, c_2, ..., c_c\}$ the set of code vectors and $\phi^{-1}(\{c_i\}) = U_i$ for $i = \{1, 2, ..., c\}$ then:

1. $\phi(x) = c_i \Longleftrightarrow x \in U_i$

2. $U_1 \cup U_2 \cup ... \cup U_c = \phi^{-1}(\{c_1\}) \cup ... \cup \phi^{-1}(\{c_c\}) = \phi^{-1}(\{c_1, c_2, ..., c_c\}) = R^n$

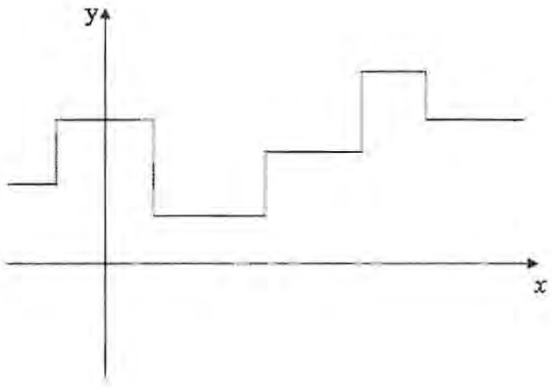
3. $C_i \cap C_j = \phi^{-1}(\{c_i\}) \cap \phi^{-1}(\{c_j\}) = \phi^{-1}(\{c_i\} \cap \{c_j\}) = \phi^{-1}(\emptyset) = \emptyset \quad \text{for } i \neq j$

We can thus characterize vector-valued step functions as follows:

Lemma 1.6.2 *A vector-valued step function is given by a partition $U_1, U_2, ..., U_c$ of R^m for function values $\{c_1, c_2, ..., c_c\} \in R^n$ with $\phi(x) = c_i$ if $x \in U_i$.*

An example of a graph of a step function where $m = n = 1$ is given in figure 1.6(a). An example for the case where $n = 2$ and $m = 1$ is given in figure 1.6(b).

(a) $m = n = 1$



(b) $n = 1$ and $m = 2$

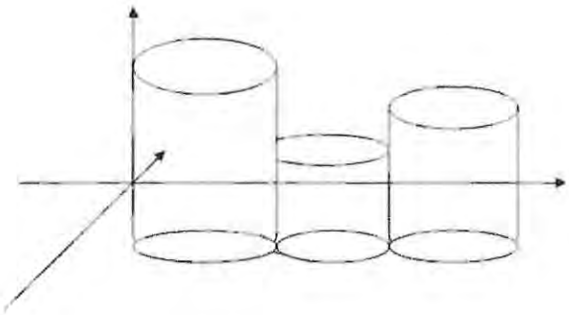


Figure 1.6: Step functions $\phi : R^m \rightarrow R^n$

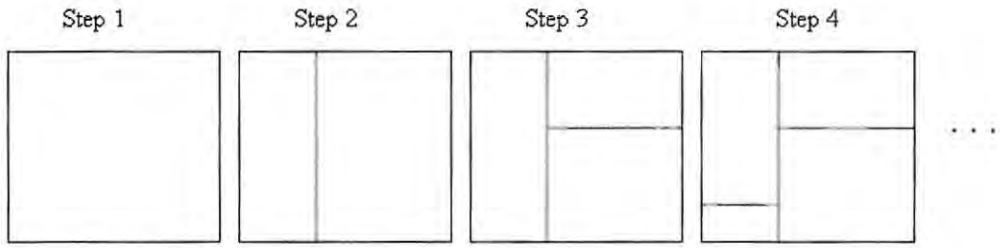


Figure 1.7: Splitting regions for a regression tree

In the context of regression trees, the regression function is defined by the partitions U_i for $i = 1, 2, ..., c$ where the U_i are bounded by hyperplanes of the form $\{x \in R^n : x_k = constant\}$. For this particular approach the partition is not derived by clustering but by

successively splitting regions. An example is given in figure 1.7.

Theorem 1.6.3

Given a partition $U_1, U_2, ..., U_c$, the step function $\phi : R^m \rightarrow R^n$ with $\phi(x) = c_i$ for $x \in U_i$ has the smallest MSE for a given learning task

x_1	x_2	...	...	...	x_N
y_1	y_2	...	...	...	y_N

if and only if

$$c_i = \frac{1}{N_i} \sum_{x_k \in U_i} y_k \tag{1.13}$$

where $N_i = |U_i|$ is the number of vectors $x_k \in U_i$.

Proof :

$$\begin{aligned}
MSE(\phi) &= \frac{1}{N} \sum_{k=1}^{N} ||y_k - \phi(x_k)||^2 \\
&= \frac{1}{N} \sum_{i=1}^{c} \sum_{x_k \in U_i} ||y_k - c_i||^2 \\
&= \frac{1}{N} \sum_{i=1}^{c} \sum_{x_k \in U_i} \sum_{j=1}^{n} (y_{kj} - c_{ij})^2 \\
\frac{\partial MSE(\phi)}{\partial c_{ji}} &= \frac{2}{N} \sum_{x_k \in U_i} (y_{kj} - c_{ij})(-1)
\end{aligned} \tag{1.14}$$

Then

$$\begin{aligned}
\frac{\partial MSE(\phi)}{\partial c_{ji}} &= 0 \\
\iff \sum_{x_k \in U_i} y_{kj} &= \sum_{x_k \in U_i} c_{ij} \\
\iff \sum_{x_k \in U_i} y_{kj} &= |U_i| c_{ij} \\
\iff \frac{1}{|U_i|} \sum_{x_k \in U_i} y_{kj} &= c_{ij}
\end{aligned} \tag{1.15}$$

Hence

$$c_i = \frac{1}{|U_i|} \sum_{x_k \in U_i} y_k \tag{1.16}$$

In order to check for sufficiency we look at the second order partial derivatives.

$$\frac{\partial^2 MSE(\phi)}{\partial c_{ji} \partial c_{lm}} = \begin{cases} 0 & m \neq i, l \neq j \\ -\frac{2}{N} \sum_{x_k \in U_i} (-1) = \frac{2N_i}{N} & m = i, l = j \end{cases}$$

The Hessian matrix is a diagonal matrix with positive diagonal entries if there are no empty clusters. Therefore there are only positive eigenvalues and the Hessian matrix is positive definite. Hence $\{c_1, c_2, ..., c_c\}$ are minimisers of the MSE.

Theorem 1.6.4

Given a partition $U_1, U_2, ..., U_c$ and the step function $\phi : R^m \rightarrow R^n$ with $\phi(x) = c_i$ for $x \in U_i$ for a given learning task

x_1	x_2	...	...	...	x_N
y_1	y_2	...	...	...	y_N

Then the $MSE(\phi)$ is minimised if we define

$$U_i \supseteq \{x_k \in R^m / ||y_k - c_i|| \leq ||y_k - c_l|| \ \forall l\} \tag{1.17}$$

such that

$$U_i \cap U_l = \phi \text{ for } \ i \neq l \text{ and } \bigcup_{i=1}^{c} U_i = R^m \tag{1.18}$$

Proof :

$$MSE(\phi) = \frac{1}{N} \sum_{i=1}^{c} \sum_{x_k \in U_i} ||y_k - c_i||^2 \tag{1.19}$$

is minimal if and only if all $\sum_{x_k \in U_i} ||y_k - c_i||^2$ are minimal. Each of these sums are minimal if for $x_k \in U_i$ we have that

$$\begin{aligned} ||y_k - c_i||^2 &\leq ||y_k - c_l||^2 \ \forall \ l \\ \iff ||y_k - c_i|| &\leq ||y_k - c_l|| \ \forall \ l \end{aligned} \tag{1.20}$$

1.6.1 The standard error of the estimate for the MSE

We let $Z_k = ||X_k - \phi(Y_k)||^2$ then

$$\begin{aligned} var(MSE(\phi)) &= Var(\frac{1}{N}\sum_{k=1}^{N} Z_k) \\ &= \frac{1}{N^2}\sum_{k=1}^{N} Var(Z_k) \\ &= \frac{1}{N^2} N Var(Z) \\ &= \frac{1}{N} Var(Z) \end{aligned} \tag{1.21}$$

We estimate $Var(Z)$ by the sample variance of the values $Z_k = ||Y_k - \phi(X_k)||^2$

$$\begin{aligned} s^2 &= \frac{1}{N}\sum_{k=1}^{N}\left(Z_k - \frac{1}{N}\sum_{k=1}^{N} Z_k\right) \\ &= \frac{1}{N}\sum_{k=1}^{N}\left(Z_k - \overline{Z}\right)^2 \\ &= \frac{1}{N}\sum_{k=1}^{N}\left(Z_k^2 - 2\overline{Z}Z_k + \overline{Z}^2\right) \\ &= \frac{1}{N}\left\{\sum_{k=1}^{N} Z_k^2 - 2\overline{Z}\sum_{k=1}^{N} Z_k + N\overline{Z}^2\right\} \\ &= \frac{1}{N}\left\{\sum_{k=1}^{N} Z_k^2 - 2N\overline{Z}^2 + N\overline{Z}^2\right\} \\ &= \frac{1}{N}\sum_{k=1}^{N} Z_k^2 - \overline{Z}^2 \end{aligned}$$

Note :

- Any statistic whose mathematical expectation is equal to the parameter is an unbiased estimator for the parameter. Therefore s^2 will be biased for $Var(Z)$ since [19]

$$E(s^2) = \frac{(N-1)var(Z)}{N} \tag{1.22}$$

- A statistic $\widehat{\theta}$ for a population parameter θ is consistent if and only if for any positive error bound, $lim_{N\longrightarrow\infty}P(|\widehat{\theta}-\theta|<b)=1$ [19]. $\frac{1}{N}\sum_k Z_k^2$ is consistent for $E(Z^2)$ and $\frac{1}{N}\sum_k Z_k$ is consistent for $E(Z)$ and since $E(Z^2)-E(Z)^2=Var(Z)$ then s^2 is consistent $Var(Z)$.

- $\overline{Z}=MSE(\phi)$

Therefore a consistent estimator for the standard error of the $MSE(\phi)$ is given by

$$\begin{aligned} SE(MSE(\phi)) &= \sqrt{\frac{s^2}{N}} \qquad (1.23)\\ &= \sqrt{\frac{1}{N}\left[\frac{1}{N}\sum_{k=1}^{N}||y_k-\phi(x_k)||^4-\left(\frac{1}{N}\sum_{k=1}^{N}||y_k-\phi(x_k)||^2\right)^2\right]} \end{aligned}$$

1.6.2 Application to vector quantization

Given an image that is to be quantized, as in the approach described before, we would subdivide the image into blocks which would then be represented by vectors x_k. The learning task would then be given by

x_1	x_2	...	x_N
x_1	x_2	...	x_N

where for $p\times q$ blocks with $p\times q=n$, $x_k\in R^n$. For this learning task

- $x_k=y_k$ because we attempt to reconstruct our image exactly the same as the original image.

- $\phi(x)$ is a step function.

From Theorem 1.6.3 and 1.6.4 we obtain that for a given partition $U_1,U_2,...,U_c$, $\phi(x)=c_i\Longleftrightarrow||x-c_i||\leq||x-c_l||\ \ \forall l=1,2,...,c$ if $x\in U_i$.

Moreover we estimate the MSE by

$$\begin{aligned} MSE(\phi) &= \frac{1}{N}\sum_{i=1}^{c}\sum_{x_k \in U_i} ||x_k - c_i||^2 \\ &= \frac{1}{N}\sum_{i=1}^{c}\sum_{x_k \in U_i}\sum_{j=1}^{n}(x_{kj} - c_{ij})^2 \end{aligned} \tag{1.24}$$

We define the pixelwise MSE for an image by

$$\begin{aligned} MSE(image) &= \frac{1}{PQ}\sum_{i=1}^{P}\sum_{k=1}^{Q}(x_{ik} - \widetilde{x}_{ik})^2 \\ &= \frac{1}{p.qN}\sum_{i=1}^{P}\sum_{k=1}^{Q}(x_{ik} - \widetilde{x}_{ik})^2 \\ &= \frac{1}{p.q}MSE(\phi) \\ &= \frac{1}{p.q}\overline{Z} \end{aligned} \tag{1.25}$$

where the image is of size $P \times Q$.

1.6.3 The standard error of the pixelwise MSE

$$MSE(image) = \frac{1}{PQ}\sum_{i=1}^{P}\sum_{k=1}^{Q}(x_{ik} - \widetilde{x}_{ik})^2$$

We estimate the standard error of the MSE

$$\begin{aligned} Var(MSE(image)) &= \frac{1}{(PQ)^2}\sum_{i=1}^{P}\sum_{k=1}^{Q} Var\left((x_{ik} - \widetilde{x}_{ik})^2\right) \\ &\approx \frac{1}{(PQ)^2}PQ.\widetilde{s}^2 \\ &= \frac{\widetilde{s}^2}{PQ} \\ \Longrightarrow SE(MSE(image)) &\approx \frac{\widetilde{s}}{\sqrt{PQ}} \end{aligned} \tag{1.26}$$

given that $\widetilde{s}^2$ = estimate of the variance of $(x_{ik} - \widetilde{x}_{ik})$

We now estimate $Var(MSE(\phi))$ by $\frac{s^2}{N}$. As a result the sample variances are different

and therefore

$$Var(MSE(\phi)) \neq Var(MSE(image))$$

In VQ it is the blocks that are used so it would perhaps be more appropriate to use

$$SE(MSE) = \sqrt{Var(MSE(\phi)}$$

as the standard error. However, it then becomes difficult when dealing with different block sizes and when using approaches that do not use the blocks. The standard error is not usually reported in papers that describe VQ applied to image compression.

1.7 Estimating the MSE : Cross-validation and the test set approach

As we have shown in section 1.6, a regression function is usually obtained by minimising the mean squared error on a random sample $\{(x_i, y_i) : i = 1, 2, ..., N\}$ taken from $\mathcal{F}\times\mathcal{L}$. Estimating the MSE from the same data that the regression function was drawn from, leads to an overly optimistic estimate. As a means of solving this problem, we may draw a second random sample from $\mathcal{F}\times\mathcal{L}$, that is independent of the sample used for approximating the regression function and estimate $MSE^*(\phi)$. This however may not be feasible as the data is often a finite set. Thus practically we would split the data in the sample into a training set and a test set. The training set would then be used to approximate the regression function. The mean squared error is estimated on the test set. To obtain a good approximation of a function, a large training set is required but then this leaves a small test set. A small test set would result in a poor estimate for the MSE^*. To alleviate this problem especially if the data set is small, we make use of cross-validation [20].

In cross validation, we split the data set X into K sets referred to as K folds

$$X = X_1 \cup X_2 \cup ... \cup X_K$$

with $X_i \cap X_j \neq \emptyset$ for $i \neq j$ with all sets approximately equal to each other in size i.e $|X_i| = \frac{N}{K}$ for $i = 1, 2, ..., K$. We learn regression functions ϕ on the training sets for $\phi_k : \tau_K = X|X_k$ (training set) and $T_k = X_k$ (test set). We make the assumption that

the method is stable under perturbations i.e that

$$MSE^* \approx MSE^*(\phi_k) \tag{1.27}$$

for all $k = 1, 2, ..., K$. Then also for the estimates $MSE(\phi) \approx MSE(\phi_k)$ and we obtain an estimate for $MSE^*(\phi)$ as follows:

$$\begin{aligned} MSE(\phi) &= \frac{1}{N}\sum_{k=1}^{K}\sum_{X_i \in T}(Y_i - \phi(X_i))^2 \\ &= \frac{1}{N}\sum_{k=1}^{K} N_k MSE(\phi_k) \\ &= \sum_{k=1}^{K}\frac{N_k}{N} MSE(\phi_k) \\ &= \frac{1}{K}\sum_{k=1}^{K} MSE(\phi_k) \end{aligned} \tag{1.28}$$

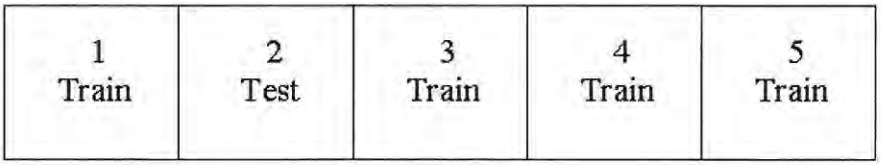
So therefore we can estimate $MSE^*(\phi)$ as the sample mean of $MSE(\phi_1)$, $MSE(\phi_2)$,..., $MSE(\phi_K)$.

1.7.1 Cross validation application to vector quantization

In VQ for image compression we employ cross-validation to measure the *generalisation* performance of VQ. By generalisation we refer to the ability for VQ to perform on an unseen data set. In image compression we would like to generate a codebook that will work well on image data that is unseen. In order to measure the performance of a vector quantizer we need to separate the data used to create a codebook, the training set from the data used to test the codebook, the test set. This is necessary because we want to ascertain the performance of the quantizer on an unknown data set. Cross-validation may be used to increase the amount of data available for training in codebook creation. For our application in image compression, K separate images may be used for a K fold cross validation approach. We would thus train the quantizer on $K-1$ images and then test it on the remaining unused image. The MSE^* estimate using cross-validation would then be given by equation (1.28)

$$MSE(\phi) = \frac{1}{K}\sum_{k=1}^{K} MSE(\phi_k)$$

Figure 1.8: Test and training sets with cross validation



$MSE(\phi_k)$ is the MSE estimate obtained by testing the quantizer on image k using a codebook trained on the other $K-1$ images. A similar approach may be done with one image by splitting the image into K parts to do a K fold cross validation. This however may result in inadequate data for the test set and result in poor estimates. We also use separate images in order to be able to visualise the test image after reconstruction. In image compression it is not only the minimal MSE that is important but also good visual quality of the image.

A problem that may arise in using cross validation in VQ for image compression is that the images that are used may not be very similar in the distribution of their pixel values. If a few images are used for the estimate this might result in poor estimations of the MSE because the test set may not be similar to the training set. This would then give MSE estimates that have wide variation and thus violate the assumption that is given in equation (1.27).

1.7.2 Estimating the MSE using random selection

An alternative approach that could be used in estimating the MSE is to get a random selection of image vectors to create a codebook and to test the quantizer. For each image we would subdivide the image into blocks. These blocks would then be represented as vectors for VQ. From these image vectors, a set of image vectors is randomly selected that will be used as training set to create a codebook. This codebook is then tested on the remaining set of vectors that were not in the training set. The training set may be made to be a larger set so that there are adequate image vectors available to create the codebook. The MSE may then be calculated for the test set of vectors to estimate the MSE for the quantizer. In order to get a more accurate estimate of the MSE, we would carry out this same procedure several times. The mean of the MSE's obtained would

then be taken as the estimate of the MSE for the quantizer.

When applied to image compression this approach has the disadvantage that it does not allow us to be able to visualise the quantized image, since the image vectors are randomly selected from the images. In other applications of VQ it may not be necessary to have the same order in the quantized data so this method would be more suitable in such instances. For our application here we would like to be able to view the quantized images as the visual quality of the image is important in determining the performance of the quantizer. A low MSE may also not necessarily mean good visual quality [14] thus a visual representation is important in assessing the performance of a quantizer.

2 Clustering algorithms for codebook design

We have stated that in VQ the design of the codebook is important as the codebook determines the performance of the quantizer. We focus here on clustering algorithms for codebook design. Although there are other algorithms that may be used, for our purposes we have chosen to use the c means, fuzzy c-means and learning vector quantization algorithms. This chapter gives a description of these algorithms.

2.1 Clustering algorithms

In general clustering is used to group similar data from a large data set in order to be able to extract information or process the data efficiently [21],[22]. Clustering algorithms are used as a way of generating the clusters. The purpose of clustering is to create clusters such that data points within a cluster are as similar to each other as possible and data from different clusters are as dissimilar as possible [21]. Given a data set S, we seek a *partition* of S

$$S = \{U_1, U_2, ..., U_c\}$$

with $U_i \subset S$.

Definition 2.1.1

If S is a set in R^n then a set of subsets $\mathcal{U} = \{U_1, U_2, ..., U_c\}$ is called a *partition* of S if

- (C1) there are no empty clusters: $U_i \neq \emptyset$ for all $i = 1, 2, ..., c$
- (C2) there are no overlapping clusters, $U_i \cap U_j = \emptyset$ for $i \neq j$
- (C3) the union of all clusters is the set S: $\bigcup_{i=1}^{c} U_i = S$

A clustering is a partition that satisfies the requirements of similarity mentioned above. An important part in the identification of clusters is a measure of similarity. There are

various measures that may be used to measure similarity that are dependent on the data type. For continuous data, often distance measures are used for similarity [21]. Examples of distance measures are the City Block distance, Euclidean distance and the Minkowski distance:

D1. City Block distance [21],[23],[24]

$$d(x_k, x_l) = \sum_{j=1}^{n} |x_{kj} - x_{lj}| \tag{2.1}$$

D2. Euclidean distance [21]

$$d(x_k, x_l) = \left(\sum_{j=1}^{n} (x_{kj} - x_{lj})^2 \right)^{\frac{1}{2}} \tag{2.2}$$

D3. Minkowski distance [21]

$$d(x_k, x_l) = \left(\sum_{j=1}^{n} |x_{kj} - x_{lj}|^h \right)^{\frac{1}{h}} \quad (h \geq 1) \tag{2.3}$$

where x_k and x_l are n dimensional vectors. In the context of image compression, we use clustering to reduce the amount of data used in image processing. We seek to partition an image, in the form of an array of vectors, into c clusters by identifying code vectors (cluster centres) to represent clusters of similar vectors. In this thesis we will measure the similarity of the image vectors using the Euclidean distance (D2). If $X = \{x_1, x_2, ..., x_n\}$ is the array of image vectors the partition is obtained from the code vectors $\{c_1, c_2, ..., c_c\}$ by

$$x_k \in U_i \iff d(x_k, c_i) \leq d(x_k, c_j) \text{ for all } i = j = 1, 2, ..., c$$

2.2 The c-means algorithm (CM)

When presented with a set of input vectors $S = \{x_1, x_2, ..., x_N\}$, the aim of the c-means algorithm [1],[4],[21],[26] is to partition S into c clusters $U_1, U_2, ..., U_c$ such that conditions

(C1), (C2), and (C3) are satisfied. The objective function W is defined by

$$\begin{aligned} W(U_1, U_2, ..., U_c, c_1, c_2, ..., c_c) &= \sum_{i=1}^{c} \sum_{x_k \in U_i} ||x_k - c_i||^2 \\ &= \sum_{i=1}^{c} \sum_{x_k \in U_i} \sum_{j=1}^{n} (x_{kj} - c_{ij})^2 \end{aligned} \tag{2.4}$$

where $\sum_{x_k \in U_i} ||x_k - c_i||^2$ is the sum of squared distances of the samples in cluster U_i to the centre c_i. Hence W is the sum of squares of the within cluster distances. Since we desire to have samples within a cluster to be as similar to each other as possible, we want to minimise W. We therefore seek a partition and centres such that W is minimised. Assuming clusters $U_1, U_2, ..., U_c$ are given, we can define a regression function by $\phi(x) = c_i$ for $x \in U_i$. Then

$$MSE(\phi) = W(U_1, U_2, ..., U_c, c_1, c_2, ..., c_c)$$

then by Theorem 1.6.3 W is minimised if and only if

$$c_i = \frac{1}{|U_i|} \sum_{x_k \in U_i} x_k \tag{2.5}$$

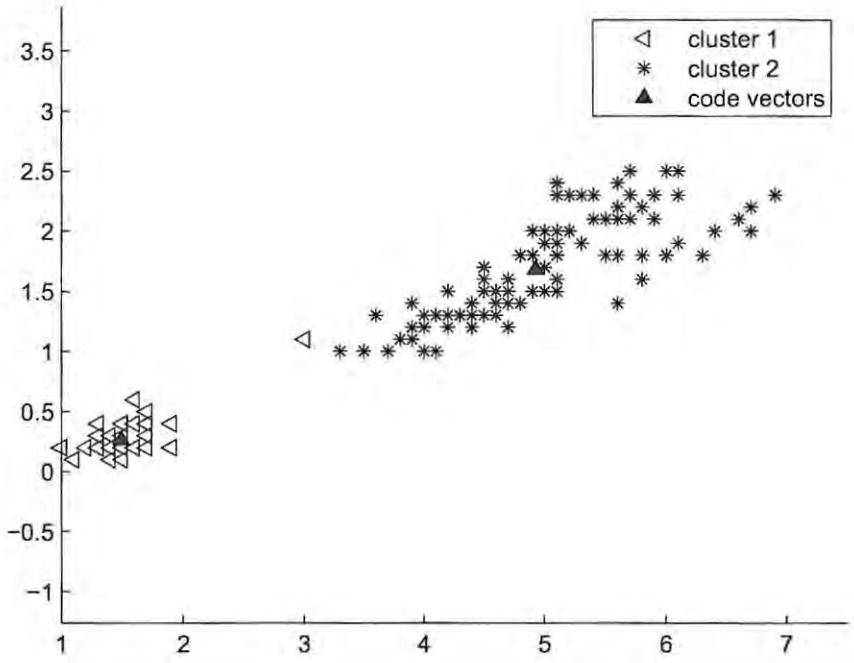
Conversely given cluster centres $c_1, c_2, ..., c_c$, by Theorem 1.6.4 W is minimised if U_i are chosen such that

$$x_k \in U_i \iff ||x_k - c_i||^2 \leq ||x_k - c_j||^2 \; \forall j = 1, 2, ..., c \tag{2.6}$$

CM therefore steps through equation (2.5) and (2.6) iteratively to generate a codebook that minimises W. To start the CM algorithm an initial set of code vectors $C^0 = \{c_1^0, c_2^0, ..., c_c^0\}$ is required. The Euclidean distance between the input vectors and each of the c code vectors is computed using equation (2.2). The distances are stored in a matrix D where D_{ik} is the distance between x_k and c_i. The input vectors x_k are assigned to a cluster U_i^0 whose cluster centre is c_i^0, if they satisfy the condition (2.6). Once all x_k have been assigned to a cluster, new code vectors are found by computing the mean vectors for each cluster using equation (2.5). The process of clustering begins again, this time using the new code vectors. The algorithm does this iteratively until there is no significant change between the Euclidean distances of code vectors from subsequent iterations. CM tries to identify compact clusters [29]. Figure 2.1 and 2.2 give examples of clustering using CM with varying numbers of code vectors for the Iris data [21].

Figure 2.1: Iris data with c-means clustering

(a) Iris data with 2 code vectors



(b) Iris data with 3 code vectors

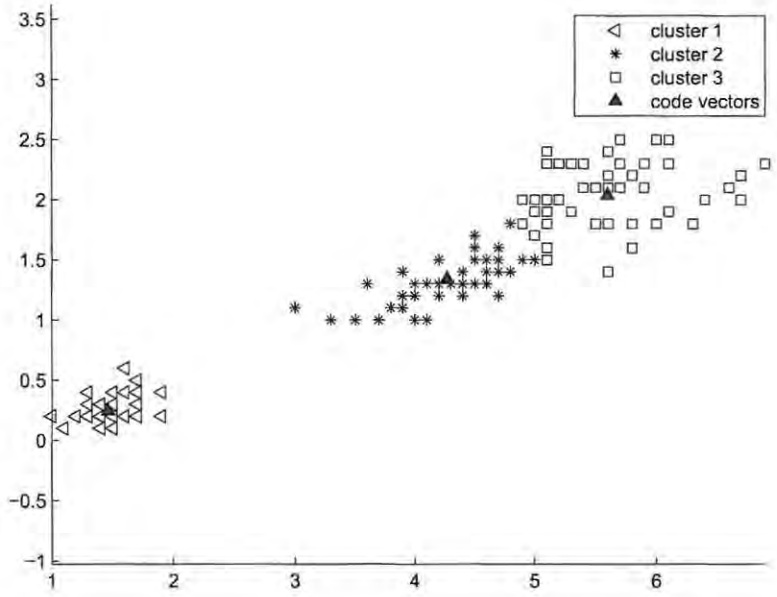
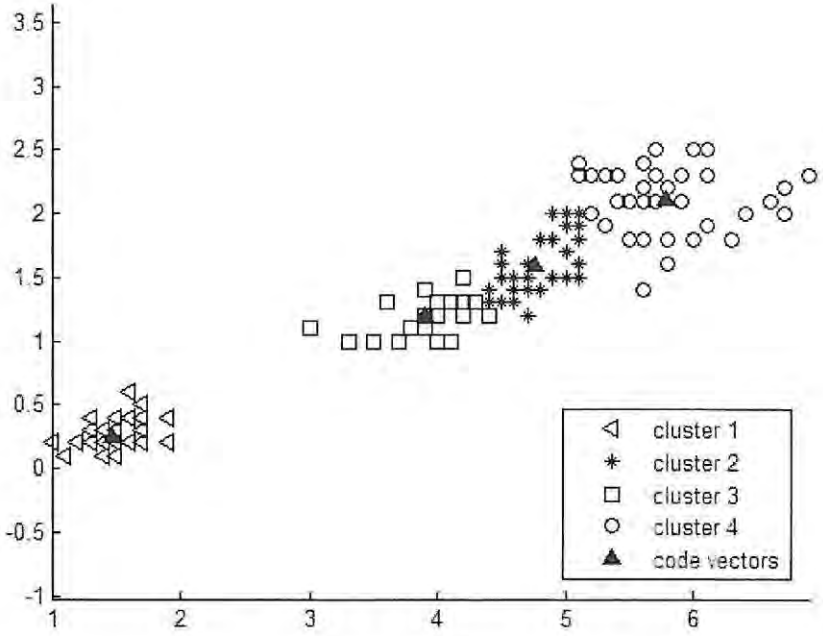
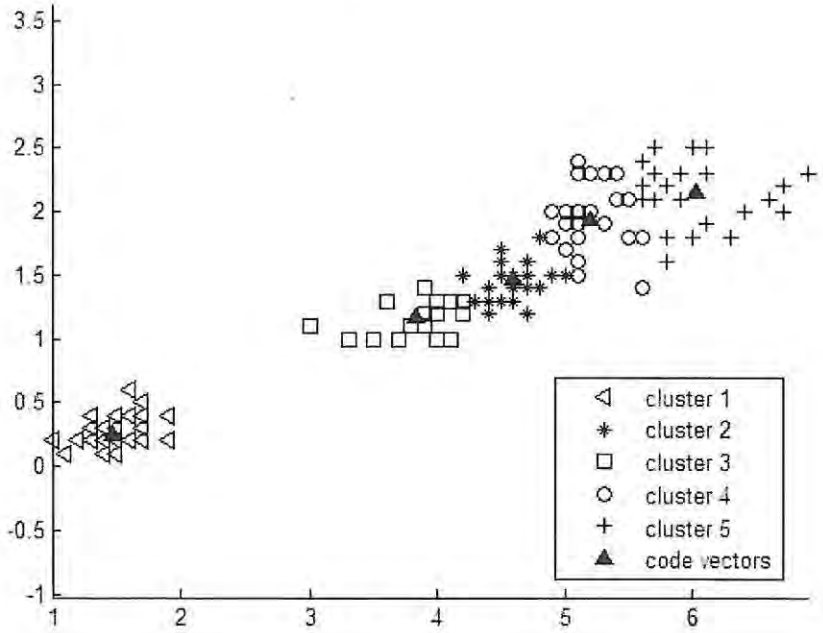


Figure 2.2: Iris data with c-means clustering

(a) Iris data with 4 code vectors



(b) Iris data with 5 code vectors



The following pseudo-code is a summary of the c-means algorithm:

Input: initial codebook $C^0 = \{c_1^0, c_2^0, ..., c_c^0\}$, *data* $X = \{x_1, x_2, ..., x_N\}$

while $t < t_{max}$ *do the following*

for $i = 1, 2, ..., c$

for $k = 1, 2, ..., N$

compute matrix of distances D using equation (2.2)

endfor

endfor

for $i = 1, 2, ..., c$

determine U_i *with equation (2.6)*

endfor

for $i = 1, 2, ..., c$

determine c_i *using equation (2.5)*

endfor

endwhile

Output : partition $U_1, U_2, ..., U_c$ *and codebook* $c_1, c_2, ..., c_c$

There are some problems that may arise in using CM. Firstly, a poor choice for an initial codebook may result in empty clusters. To avoid this problem, the initial code vectors may be chosen from the data so that the chances of having empty clusters are reduced. When there are empty clusters we effectively reduce the code vectors that are available to the data. This is because a code vector cannot be updated if the cluster is empty. Moreover in equation (2.5) we divide by $|U_i|$ which is the size of the cluster. For an empty cluster this value is zero, so the code vector update then becomes an undefined result. Linde, Buzo and Gray [27] suggest another method that may be used for initialisation that is referred to as *splitting*. Using splitting we derive the initial codebook from one code vector that is the centroid of the whole data set. This code vector is split into two close code vectors. The CM algorithm is then used to generate a codebook with two code vectors. Each of these two code vectors is also split into another two vectors that are close to each other so that there are four code vectors in total. Again these are used as the initial codebook and CM is used to generate a codebook with four code vectors.

This splitting process is done iteratively until the desired number of code vectors that are required for the initial codebook c is met. The codebook generated with c code vectors is then used as the initial codebook.

Another problem that may arise in CM is in the use of the matrix of distances D. For large data sets D becomes very large and thus it is difficult to store the array. To overcome this problem a sequential algorithm may be used such that we do not store the entire array. To do this, in each iteration, an individual data point x_k is presented. We calculate $d^k = (d_1, d_2, ..., d_c)$, the Euclidean distances between x_k and the c code vectors where for example $d_1 = d(x_k, c_1)$ is the Euclidean distance between x_k and c_1. Equation (2.5) is then used to determine the cluster U_i that x_k belongs to and this is stored as a label in the vector U. This is done for all $k = 1, 2, ..., N$. The following pseudo-code gives the sequential c-means algorithm:

Input: initial codebook $C^0 = \{c_1^0, c_2^0, ..., c_c^0\}$, *data* $X = \{x_1, x_2, ..., x_N\}$

while $t < t_{max}$ *do the following*

for $k = 1, 2, ..., N$

for $i = 1, 2, ..., c$

compute d^k *using equation (2.2)*

endfor

determine U_i *with equation (2.6)*

endfor

for $i = 1, 2, ..., c$

determine c_i *using equation (2.5)*

endfor

endwhile

Output : partition $U_1, U_2, ..., U_c$ *and codebook* $\{c_1, c_2, ..., c_c\}$

2.3 The fuzzy c-means (FCM)

The FCM algorithm is an application of fuzzy set theory to clustering, that was first published by Dunn [29] in 1974, and in Bezdek's thesis in 1973 [30]. The main idea of

fuzzy clustering is that data are allowed a grade of membership in more than one cluster. The characteristic function of a cluster $U_i \subset S$ is given by

$$1_{U_i}(x) = \begin{cases} 1, & \text{if } x \in U_i; \\ 0, & \text{elsewhere.} \end{cases}$$

If we define $u_{ik} = 1_{U_i}(x_k)$ then $u_{ik} = 1$ if $x_k \in U_i$ and $u_{ik} = 0$ if $x_k \notin U_i$ where u_{ik} is interpreted as the strength of the membership of x_k in the cluster U_i. Hence $U = u_{ik} \in \{0,1\}^{c \times N}$ is a partition matrix that contains all the information of the partition. U however, does not contain the actual cluster centres for each partition.

Definition 2.3.1: A fuzzy clustering of $S = \{x_1, x_2, ..., x_N\}$ can be described as a set $\mathcal{U} = \{U_1, U_2, ..., U_c\}$ of fuzzy sets such that $U_i : S \longrightarrow [0,1]$ where $u_{ik} = U_i(x_k)$ is the grade of membership of x_k in U_i [30],[31] . The fuzzy cluster matrix or partition matrix U is shown below

$$U = \begin{pmatrix} u_{11} & u_{12} & . & . & . & u_{1N} \\ u_{21} & u_{22} & . & . & . & u_{2N} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ u_{c1} & u_{c2} & . & . & . & u_{cN} \end{pmatrix}$$

and this matrix should satisfy the following conditions [32]:

- (FC1) $0 \leqslant u_{ik} \leqslant 1$
- (FC2) $\sum_{i=1}^{c} u_{ik} = 1$, for all $k = 1, 2, ..., N$
- (FC3) $\sum_{k=1}^{n} u_{ik} > 0$

These conditions allow the input vectors x_k to belong to more than one cluster and ensure that there are no empty clusters. For a crisp clustering the $u_{ik} = 0$ or 1 because an input vector is either in the cluster ($u_{ik} = 1$), or not in the cluster ($u_{ik} = 0$). For a crisp clustering the within groups sum of squares is given by

$$W_m(U, c_1, c_2, ..., c_c) = \sum_{i=1}^{c} \sum_{x_k \in U_i} ||x_k - c_i||^2 = \sum_{i=1}^{c} \sum_{k=1}^{N} u_{ik}^m ||x_k - c_i||^2 \qquad (2.7)$$

as $u_{ik}^m = u_{ik}$ for $u_{ik} \in \{0,1\}$. For fuzzy clustering this can be generalised by defining the within groups sum of squares as

$$W_m(U, c_1, c_2, ..., c_c) = \sum_{i=1}^{c} \sum_{k=1}^{N} u_{ik}^m ||x_k - c_i||^2 \tag{2.8}$$

where $m \in (1, \infty)$ is a weighting exponent on each fuzzy membership and is called the fuzzifier. FCM tries to minimise W [32]. The conditions that allow the minimisation of W_m are given in theorem 2.3.1.

Theorem 2.3.1 [30]

Assume $||.||$ to be inner product induced. Fix $m \in (1, \infty)$, let S have at least $c < N$ distinct points, and define for all $k \in \{1, 2, ..., N\}$ the sets

$$I_k = \{i : 1 \leq i \leq c; \quad d_{ik} = ||x_k - c_i|| = 0\} \tag{2.9}$$

$$\tilde{I}_k = \{1, 2, ..., c\} - I_k \tag{2.10}$$

Then (U, C) may be globally minimum for W_m only if

$$I_k = \emptyset \Longrightarrow u_{ik} = \left(\sum_{j=1}^{c} \left(\frac{d_{ik}}{d_{jk}} \right)^{2/(m-1)} \right)^{-1} \tag{2.11}$$

or

$$I_k \neq \emptyset \Longrightarrow u_{ik} = 0 \, \forall i \in \tilde{I}_k \quad \text{and} \quad \sum_{i \in I_k} u_{ik} = 1 \tag{2.12}$$

and

$$c_i = \frac{\sum_{k=1}^{N} (u_{ik})^m x_k}{\sum_{k=1}^{N} (u_{ik})^m} \quad \forall i \tag{2.13}$$

Proof :

We quote the proof given in Bezdek [30]. First, fix $C = \{c_1, c_2, ..., c_c\}$ and define $f_m(U) = W_m(U, C)$ for any U in a fuzzy partition space. Since U is degenerate its columns are

independent, and therefore

$$\begin{aligned}\min_U\{f_m(U)\} &= \min_U\left\{\sum_{k=1}^{N}\sum_{i=1}^{c}(u_{ik})^m(d_{ik})^2\right\} \\ &= \sum_{k=1}^{N}\left(\min_{U_k\in conv(B_c)}\left\{\sum_{i=1}^{c}(u_{ik})^m(d_{ik})^2\right\}\right) \qquad (2.14)\end{aligned}$$

where

$$conv(B_c) = \left\{U_k \in R^c : \sum_{i=1}^{c} u_{ik} = 1; u_{ik} \geq 0\right\} \qquad (2.15)$$

Solution of equation (2.14) is effected with Lagrange multipliers. For each term, let

$$f_{mk}(U_k) = \sum_{i=1}^{c} u_{ik}^m(d_{ik})^2 \qquad (2.16)$$

and let its Lagrangian be

$$F_k(\lambda, U_k) = \sum_{i=1}^{c}(u_{ik})^m(d_{ik})^2 - \lambda\left(\sum_{i=1}^{c} u_{ik} - 1\right) \qquad (2.17)$$

(λ, U_k) is stationary for F_k only if $\nabla_{\lambda,U_k}F_k(\lambda, U_k) = 0$, where ∇_{λ,U_k} is the gradient with respect to λ and U_k. Setting this gradient equal to zero yields

$$\frac{\partial F_k}{\partial \lambda}(\lambda, U_k) = \left(\sum_{i=1}^{c} u_{ik} - 1\right) = 0 \qquad (2.18)$$

$$\frac{\partial F_k}{\partial U_{st}}(\lambda, U_k) = \left[m(u_{ik})^{m-1}(d_{st})^2 - \lambda\right] = 0 \qquad (2.19)$$

From equation (2.19) we have that

$$u_{st} = \left[\frac{\lambda}{m(d_{st})^2}\right]^{1/(m-1)} \qquad (2.20)$$

Using equation (2.20) we obtain

$$\begin{aligned}\sum_{j=1}^{c} u_{jt} &= \sum_{j=1}^{c} \left(\frac{\lambda}{m}\right)^{\frac{1}{m-1}} \left(\frac{1}{(d_{ij})^2}\right)^{\frac{1}{m-1}} \\ &= \left(\frac{\lambda}{m}\right)^{\frac{1}{m-1}} \left\{\sum_{j=1}^{c} \left(\frac{1}{(d_{ij})^2}\right)^{\frac{1}{m-1}}\right\} \\ &= 1 \end{aligned} \tag{2.21}$$

Thus

$$\left(\frac{\lambda}{m}\right)^{\frac{1}{m-1}} = \frac{1}{\sum_{j=1}^{c} \left(\frac{1}{(d_{jt})^2}\right)^{\frac{1}{m-1}}} \tag{2.22}$$

Returning to equation (2.19),

$$\begin{aligned} u_{st} &= \left(\sum_{j=1}^{c} \left(\frac{1}{(d_{ij})^2}\right)^{\frac{1}{m-1}}\right) \left(\frac{1}{(d_{st})^2}\right)^{\frac{1}{m-1}} \\ &= \left(\sum_{j=1}^{c} \left(\frac{d_{st}}{d_{jt}}\right)^{\frac{2}{m-1}}\right) \end{aligned} \tag{2.23}$$

At this point there are two possibilities: if $I_t = \emptyset$, then equation (2.11) follows for column t: if $I_t \neq \emptyset$, then choosing u_{st} as in equation (2.12) results in $f_{mt}(U_t) = 0$, because the non-zero weights are placed on zero distances, while positive distances will increase $f_{mt}(U_t)$, contradicting minimality. Continuing in this way, we arrive at N vectors $\{U_1, U_2, ..., U_N\}$, which, taken together as an array U, define a stationary point for g_m. If all of the U_k's are computed by equation (2.11), $u_{ik} \in (0,1) \forall i,k$. If singularities occur, necessitating the use of equation (2.12) for some columns, U is non-degenerate. To see this, suppose, to the contrary, that for row r, $U_{rk} = 0 \, \forall k$. Define

$$c_i^* = \begin{cases} c_i & 1 \leq i \leq c;\ i \neq r \\ x_p & i = r \end{cases} \tag{2.24}$$

where $x_p \in X$, $x_p \neq c_i$ for $i \neq r$. Such a vector exists because X contains c distinct points. Now

$$0 = \sum_{k=1}^{N} (u_{rk})^m (d_{rk})^2 = \sum_{k=1}^{N} (u_{rk})^m (d_{rk}^*)^2 \tag{2.25}$$

where $d^*_{rk} = ||x_k - c^*_r|| \ \forall r$. Then (U, C^*) is a global minimum of W_m. Since $c^*_r = x_p$, the set $I_p = \{r\}$ with $u_{rp} = 1$ from equation (2.12). This contradicts that U is in a fuzzy partition space whenever equation (2.11) and (2.12) are used to construct it.

To establish equation (2.13), fix $U \in M$ and set $h_m(C) = W_m(U, C)$. Minimisation of h_m is unconstrained so we have

$$\begin{aligned} h_m(C) &= \sum_{k=1}^{N}\sum_{i=1}^{c}(u_{ik})^m (d_{ik})^2 \\ &= \sum_{k=1}^{N}\sum_{i=1}^{c}(u_{ik})^m \langle x_k - c_i, x_k - c_i\rangle \end{aligned} \tag{2.26}$$

where $\langle x_k - c_i, x_k - c_i\rangle$ is the norm inducing inner product. Then for each i, it is necessary that the directional derivatives $h'_m(c_i; w)$ vanish for all unit vectors $w \in R^c$:

$$\begin{aligned} \frac{\partial h_m}{\partial w}(c_i) &= \sum_{k=1}^{N}(u_{ik})^m \frac{d}{dt}\left(\langle x_k - c_i - tw, x_k - c_i - tw\rangle\right)|_{t=0} \\ &= -2\left[\sum_{k=1}^{N}(u_{ik})^m \langle x_k - c_i, x_k - c_i\rangle\right] = 0 \quad \forall w \\ \iff &\left\langle \sum_{k=1}^{N}(u_{ik})^m (x_k - c_i), w \right\rangle = 0 \quad \forall w \\ \iff &\sum_{k=1}^{N}(u_{ik})^m (x_k - c_i) = \theta \end{aligned} \tag{2.27}$$

from which equation (2.13) follows.

This theorem was previously proved by Dunn (1974) [29] for $m = 2$ and was later generalised by Bezdek for $m = (1, \infty)$ as is shown above.

We now state the conditions that FCM steps through during clustering:

- $(WM1)$

$$U_{ik} = \sum_{j=1}^{c}\left(\left(\frac{||x_k - c_i||}{||x_k - c_j||}\right)^{2/(m-1)}\right)^{-1} \ \forall \ i,k \tag{2.28}$$

- $(WM2)$

$$c_i = \sum_{i=1}^{c} \left[\frac{u_{ik} x_k}{\sum u_{ik}^m} \right] \tag{2.29}$$

To start the algorithm initial cluster centres are selected outside of the data set. In equation (2.28) if $x_k = c_j$ then $||x_k - c_j|| = 0$ and this gives an undefined result as we divide by zero. Initial code vectors must therefore be chosen to be similar to the data set but not part of the data set. Starting with initial code vectors $C^0 = \{c_1^0, c_2^0, ..., c_c^0\}$ a fuzzy clustering is performed by computing the grade of membership using equation 2.28 for all i and k and the code vectors are updated using equation (2.30).

$$c^{t+1} = \sum_{i=1}^{c} \left[\frac{u_{ik}^m x_k}{\sum u_{ik}^m} \right] \tag{2.30}$$

where c_i are weighted cluster means and $t = \{0, 1, 2, ..., t_{max}\}$ is the number of iterations. This process is done iteratively until there is no change in the partition matrix for subsequent iterations. FCM like CM is a non-sequential or a batch algorithm, as the cluster centres are only updated after a pass through the whole data set.

The fuzzy c-means algorithm may be summarised by the following pseudo code:

Input: initial codebook $C^0 = \{c_1^0, c_2^0, ..., c_c^0\}$, *data* $X = \{x_1, x_2, ..., x_n\}$, $1 < m < \infty$

while $t < t_{max}$ *do the following*

for $i = 1, 2, ..., c$

for $k = 1, 2, ..., n$

compute matrix of distances D using equation (2.2)

end

end

for $i = 1, 2, ..., c$

determine U_i *with equation (2.28)*

endfor

for $i = 1, 2, ..., c$

determine c_i *using equation (2.29)*

endfor

endwhile

Output : partition $U_1, U_2, ..., U_c$ *and codebook* $c_1, c_2, ..., c_c$

The fuzzifier m affects the distribution of the memberships and thus the performance of FCM is dependent on the choice of m [32]. As $m \rightarrow 1$ FCM becomes like a crisp clustering and thus becomes like CM. Bezdek [30] claimed that FCM converges to a minimum of equation (2.8). This was proved to be incorrect by Tucker [30]. The corrected result is given by Hathaway and Bezdek [33]. The result is given in theorem 2.3.2 and 2.3.3.

Theorem 2.3.2 (Global Convergence Theorem for FCM)

Let the sample X contain at least $c < n$ distinct points, and let (U, C) be any starting point in $M \times \mathbb{R}$ for the FCM iteration sequence $\{(U_1, c_1), (U_2, c_2), ...\}$. If (U^*, c^*) is any limit point of the sequence then it is either a minimum or saddle point of equation (2.8) [33].

Theorem 2.3.3 (Local Convergence Theorem for FCM)

Let the sample X contain at least $c < n$ distinct points and (U^*, C^*) be any minimum of (2.8) such that $d_{ik} > 0$ for all i, k and at which the Hessian of W_m is positive definite relative to all feasible directions . Then FCM is convergent to (U^*, C^*) [33].

Practically when using FCM the fuzzy clusters are created but only the centres are used. The input vectors belong to the clusters for which they have the highest grade of membership. For large data sets the matrix of distances D becomes very large and thus it is difficult to store the array. To overcome this problem a sequential algorithm may be used such that we do not store the entire array. For each x_k we calculate the following

$$d_i(x_k) = \left((x_k - c_i)'(x_k - c_i)\right)^{\frac{1}{m-1}} \tag{2.31}$$

We use this to calculate the membership function for x_k

$$U_i = \left(d_i(x_k) \sum_k \frac{1}{d_i(x_k)}\right)^{-1} \tag{2.32}$$

The update for the code vectors is calculated using the following equations:

$$c_i^{nm} = c_i^{nn} + U_i^m x_k \tag{2.33}$$

$$c_i^{dm} = c_i^{dm} + U_i \tag{2.34}$$

$$c_i = \frac{c_i^{nm}}{c_i^{dm}} \tag{2.35}$$

The sequential algorithm is summarised by the following pseudo code:

Input: initial codebook $C^0 = \{c_1^0, c_2^0, ..., c_c^0\}$, *data* $X = \{x_1, x_2, ..., x_n\}$, $1 < m < \infty$,

while $t < t_{max}$ *do the following*

for $k = 1, 2, ..., N$

for $i = 1, 2, ..., c$

compute d_i *using equation (2.31)*

end

for $i = 1, 2, ..., c$

determine U_i *with equation (2.32)*

calculate c_i^{nm} *using equation (2.33)*

calculate c_i^{dm} *using equation (2.34)*

endfor

endfor

for $i = 1, 2, ..., c$

determine c_i *using equation (2.35)*

endfor

endwhile

Output : partition $U_1, U_2, ..., U_c$ *and codebook* $c_1, c_2, ..., c_c$

We now consider the *butterfly* data to illustrate the differences between FCM and CM. The name of the data is derived from the arrangement of the data points as shown in figure 2.3. Figure 2.4 shows the clusters and centres that would be generated using CM and FCM on the butterfly data to produce 2 clusters.

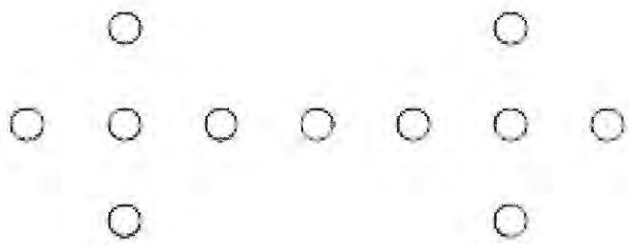


Figure 2.3: Butterfly data

For CM the cluster centres are

$$c_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \text{and} \quad c_2 = \begin{pmatrix} 4.667 \\ 1 \end{pmatrix}$$

For coordinates refer to figure 2.4. We note that there seems to be a problem in allocating the midpoint $(3, 1)$ to a cluster because it is midway between each of the clusters. Thus we cannot justify which cluster would be more appropriate as the point is equidistant from points in both clusters. It would be more appropriate to assign this point to both clusters. However with crisp clustering, in this case CM, this is not possible. We apply the FCM algorithm to the same data. The results are shown in figure 2.4.b. The clusters still remain the same, with the midpoint belonging to one cluster, so in that sense the result is similar to the CM result. However the centres that are obtained using FCM are

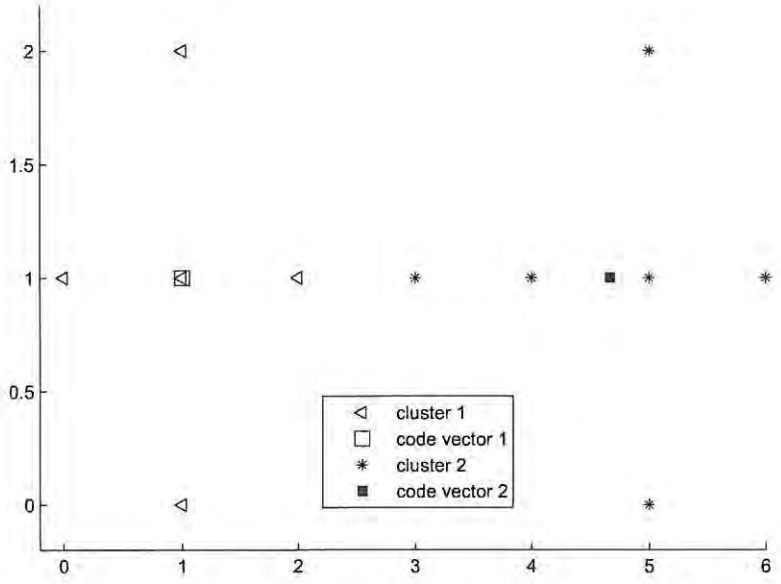
$$c_1 = \begin{pmatrix} 1.1023 \\ 1 \end{pmatrix} \quad \text{and} \quad c_2 = \begin{pmatrix} 4.8976 \\ 1 \end{pmatrix}$$

and these are not the same as those obtained using CM. We note here that the centres are more symmetric. Although the actual clusters are the same, the positioning of the centres has changed. The membership grade for the midpoint is

$$U(x) = \begin{pmatrix} 0.4998 \\ 0.5002 \end{pmatrix}$$

which is approximately equal membership grade for both clusters. FCM is therefore able to generate better cluster centres in situations where there are data points that could belong to more than one cluster.

(a) Butterfly data, c-means $c = 2$



(b) Butterfly data, FCM $c = 2$, $m = 2$

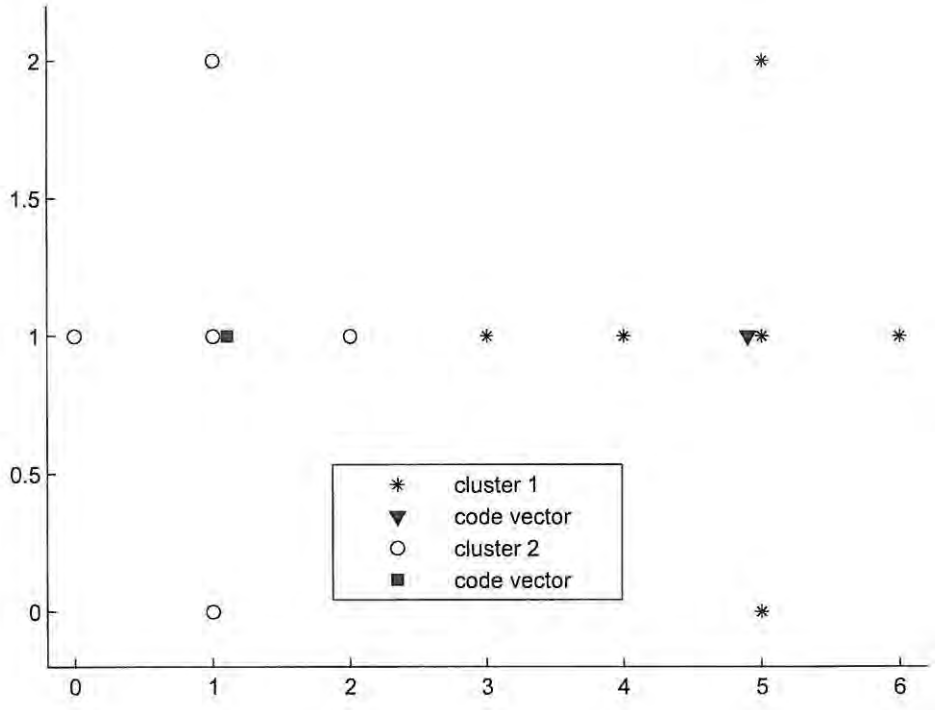
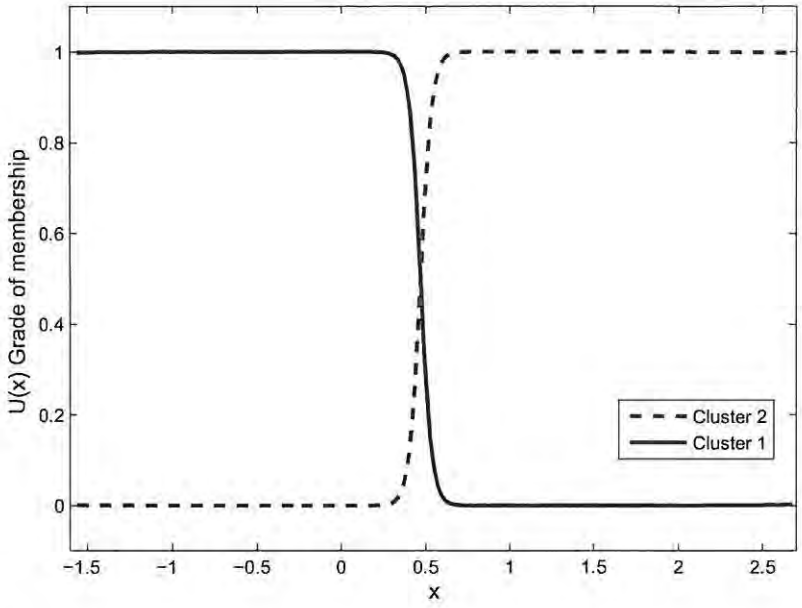


Figure 2.4: Butterfly data (a) CM (b) FCM with 2 clusters

Figure 2.5: FCM $X \sim N(0, 0.5)$ with $c = 2$

(a) $m = 1.1$



(b) $m = 1.5$

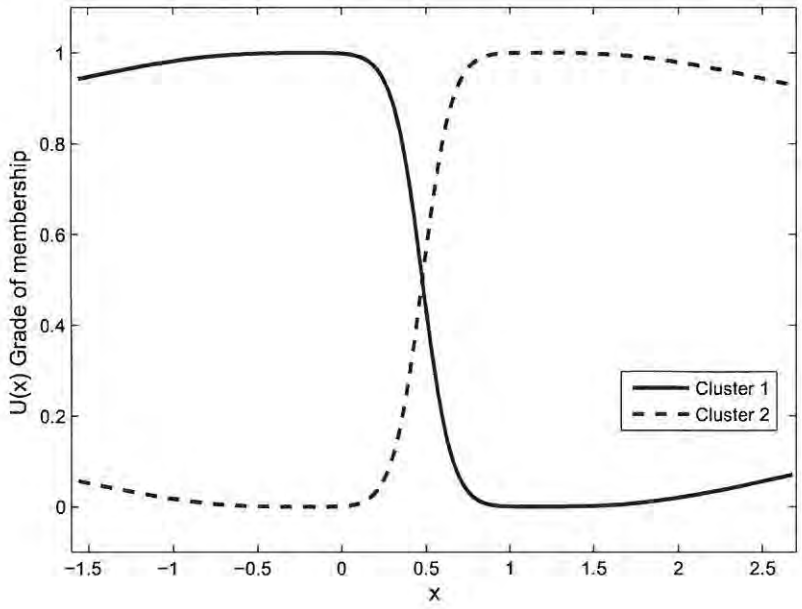
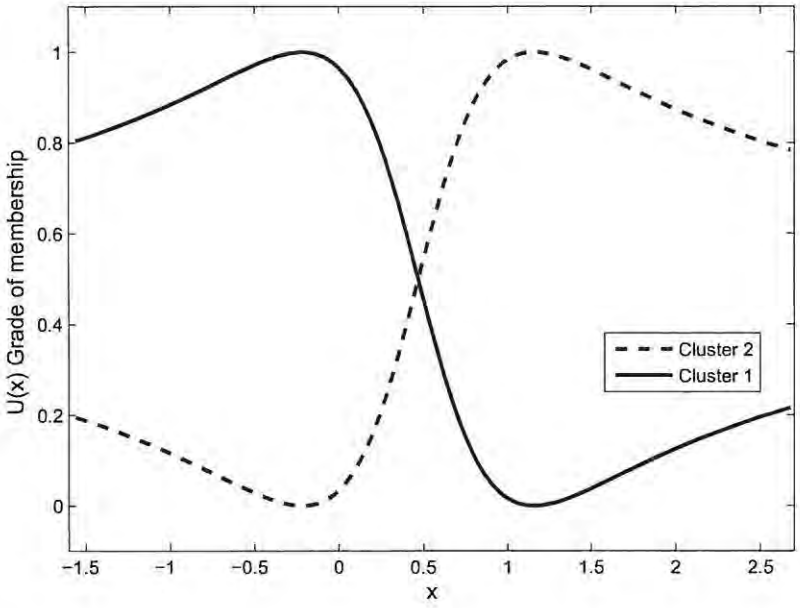
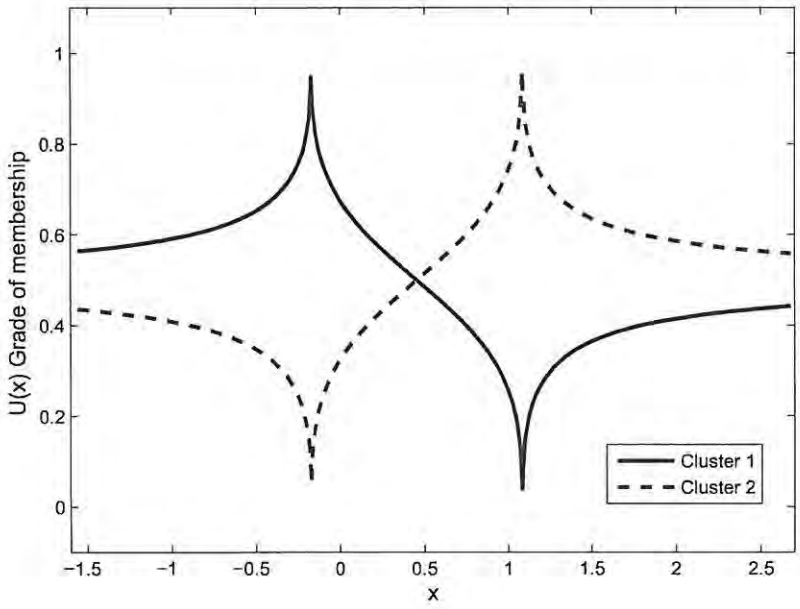


Figure 2.6: FCM on $X \sim N(0, 0.5)$ with $c = 2$

(a) $m = 2$



(b) $m = 6$



We now consider one dimensional data set that has two clusters, one normally distributed with a mean of zero and a variance of 0.5 and the other with a mean = 1 and variance of 0.5. We examine the effect of the fuzzifier on the grades of membership for the clusters in this data set. Figure 2.5-2.6 display the grade of membership for various values of the fuzzifier m. We observe that as we move towards the centres $U(x)$ is closer to 1 for the data points in the cluster and closer to zero for those pionts that are not in the cluster. In figure 2.5(a) with $m = 1.1$ we observe that for most data points $U(x) = 0$ or 1 which shows the tendency towards crisp clusters for m close to one. As m is increased there is a greater overlap in the clusters as is seen in figure 2.6(a) .

2.4 Learning vector quantization (LVQ)

An *unsupervised* clustering is one in which underlying clusters or groups within a data set are found without prior knowledge on the clusters [7],[21],[22]. LVQ [3],[4] is a name used to refer to algorithms that are used for unsupervised learning [10],[34]. Learning refers to the process of determining of weights in a neural network [22]. The LVQ neural network consists of an input layer and a competitive layer. The input layer consists of n input neurons for the input vector

$$x_k = \begin{pmatrix} x_{k1} \\ x_{k2} \\ . \\ . \\ . \\ x_{kn} \end{pmatrix}$$

The input layer is fully connected to the competitive layer that consists of c neurons for the c cluster centres. Neuron i of the competitive layer computes the distance $||x_k - c_i||$ between the x_k and the vector of weights to this neuron,

$$c_i = \begin{pmatrix} c_{i1} \\ c_{i2} \\ . \\ . \\ . \\ c_{in} \end{pmatrix}$$

which is centre i. There is also one decision neuron which computes the index of the *winner* neuron c_j such that c_j satisfies the NN condition. The winner neurone is then updated. The neural network is shown in figure 2.7 [31],[35].

The initial codebook for LVQ is usually selected from the data. During the process of clustering only the winner code vector is updated. If the code vector is not close enough to any data it may never be chosen as winner. This often results in under utilised code vectors when using LVQ [8]. If the initial code vectors are chosen from the data set then this is less likely to occur. Starting with initial centres that are randomly chosen from S, each x_k is presented and the Euclidean distance between the x_k and the cluster centres are calculated. The clusters are said 'to compete' and the *winning* cluster centre is the one that has the minimum Euclidean distance from x_k. The winner c_j is then updated as follows:

$$c_j^t = c_j^{t-1} + \eta_t \left(x_i - c_j^{t-1} \right) \tag{2.36}$$

where t is the iteration number and η_t is the learning rate at time t, $0 < \alpha_t < 1$, which is reduced monotonically. The winner is adjusted towards the input image vector x_k by a factor determined by the learning rate [20]. The update is shown in figure 2.8. This is then done for all x_k for $k = 1, 2, ..., N$.

LVQ tries to minimize the local error of x_k with respect to the winning code vector, L_k, given by

$$L_k(c_1, c_2, ..., c_c) = \frac{1}{2} \sum_{i=1}^{c} \delta_{ik} ||x_k - c_i||^2 \tag{2.37}$$

with

$$\delta_{ik} = \begin{cases} 1, & \text{if } c_i \text{ wins} \\ 0, & \text{elsewhere} \end{cases} \tag{2.38}$$

$\Longrightarrow$

$$\frac{\partial L_k}{\partial c_i} = \begin{cases} -(x_{kj} - c_{ij}), & \text{if } c_i \text{ wins} \\ 0, & \text{elsewhere} \end{cases} \tag{2.39}$$

Going in the direction of the steepest descent, the negative gradient direction,

$$c_i \leftarrow c_i - \alpha \frac{\partial L_k}{\partial c_i} = c_i + \eta(x - c_i) \text{ if } c_i \text{ wins} \tag{2.40}$$

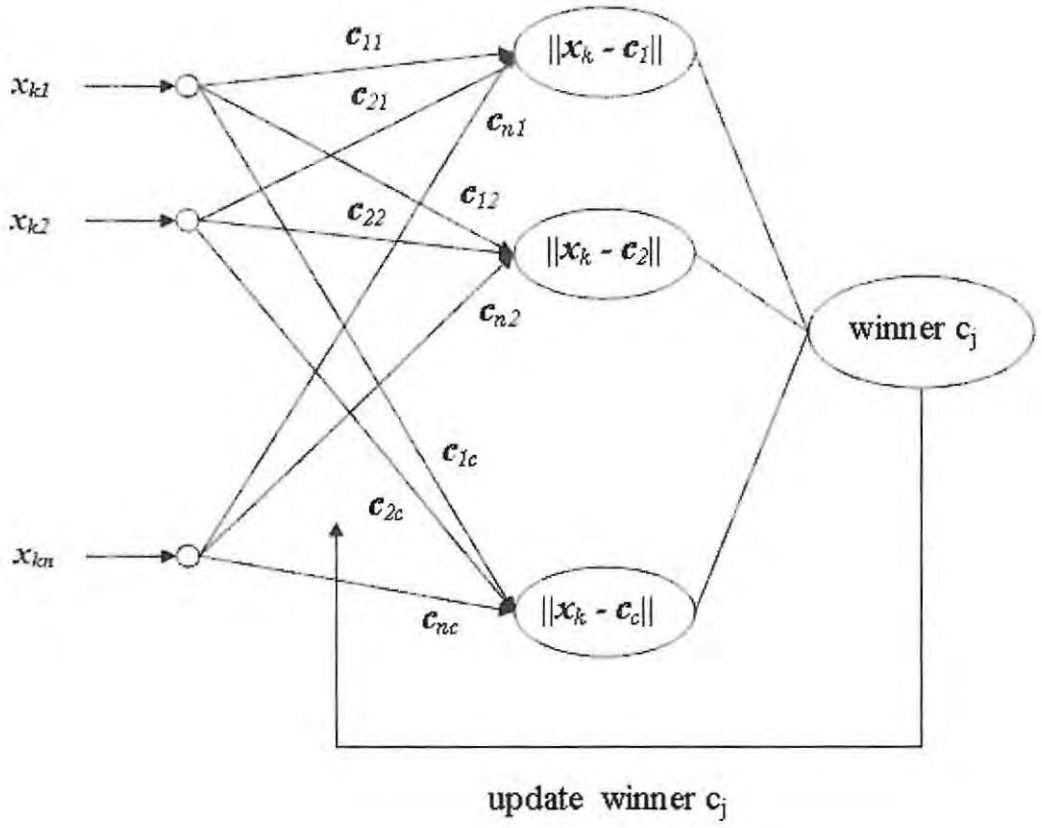


Figure 2.7: LVQ network

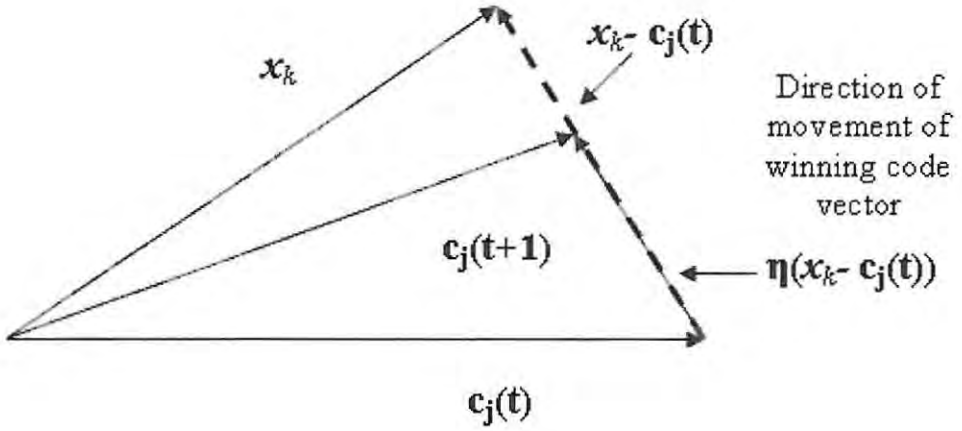


Figure 2.8: LVQ update

$$c_i \leftarrow c_i \text{ if } c_i \text{ does not win} \tag{2.41}$$

Since the cluster centres are updated every time that a vector is submitted to the network, LVQ is a sequential algorithm. The algorithm is terminated after a set number of iterations.

The algorithm can be summarised using the following pseudo code:

Input: initial codebook $C^0 = \{c_1^0, c_2^0, ..., c_c^0\}$, *data* $X = \{x_1, x_2, ..., x_n\}$

for $t = 1, 2, ..., t_{max}$ *do the following*

for $k = 1, 2, ..., n$

compute matrix of distances $d(k)$ *using equation (2.2)*

determine winner c_j *using equation (2.6)*

update c_j *using equation (2.36)*

endfor

endfor

Output : codebook $c_1, c_2, ..., c_c$

Relationship between LVQ and CM.

We consider the steepest descent for the CM functional given by:

$$\begin{aligned} L &= W(U_1, U_2, ..., U_c, c_1, c_2, ..., c_c) \\ &= \sum_{i=1}^{c} \sum_{x_k \in U_i} ||x_k - c_i||^2 \\ &= \sum_{i=1}^{c} \sum_{x_k \in U_i} \sum_{l=1}^{n} (x_{kl} - c_{il})^2 \end{aligned} \tag{2.42}$$

The first partial derivative of L is

$$\begin{aligned} \frac{\partial L}{\partial c_{il}} &= \sum_{x_k \in U_i} -2(x_{kl} - c_{il}) \\ \Longrightarrow \frac{\partial L}{\partial c_{il}} &= -2 \sum_{x_k \in U_i} (x_k - c_i) \end{aligned} \tag{2.43}$$

If the code vectors are not updated for one pass through the data and the updates would then be accumulated to give the following

$$\sum_{k=1}^{N} \frac{\partial E}{\partial c_i} = -\sum_{k=1}^{N} \delta_{ik}(x_k - c_i) \tag{2.44}$$

where

$$\begin{aligned} \delta_{ik} &= 1 \Longleftrightarrow c_i \text{ wins at presentation of } x_k \\ &\Longleftrightarrow ||x_k - c_i|| \leq ||x_k - c_j|| \quad \forall j = 1, 2, ..., c \\ &\Longleftrightarrow x_k \in U_i \end{aligned} \tag{2.45}$$

Therefore δ_{ik} is the characteristic function $1_{U_i}(x_k)$. Hence the accumulated rates of change are

$$\begin{aligned} \sum_{k=1}^{N} \frac{\partial L}{\partial c_i} &= -\sum_{k=1}^{N} 1_{u_i}(x_k)(x_k - c_i) \\ &= -\sum_{x_k \in U_i} (x_k - c_i) \end{aligned} \tag{2.46}$$

So the steepest descent update for CM is

$$c_i \longleftarrow c_i + \eta \sum_{x_k \in U_i} (x_k - c_i) \tag{2.47}$$

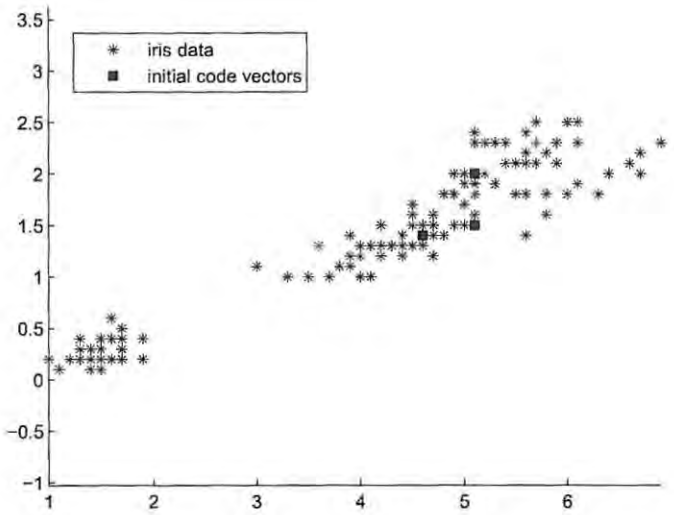
This is the same update rule as is given in equation (2.36) if we accumulate the updates. The result is that LVQ may be considered as a sequential steepest descent for the CM function.

Figure 2.9-2.10 give an example where the LVQ algorithm is used, to illustrate the movement of initial cluster centres into the centre of clusters.

2.4.1 Self organising feature map algorithm

Kohonen's self organising feature map algorithm (SOFM) is an algorithm that belongs to the class of competitive learning algorithms [2],[22]. It has also been used in codebook design. Upon the presentation of data the code vector that satisfies the NN condition is identified as the winner. The winner and the code vectors in the neighbourhood of the winner are updated. The neighbourhood of a code vector is defined according

Figure 2.9: LVQ example using Iris data with 3 code vectors
(a) Initial codebook



(b) Codebook after 2 iterations

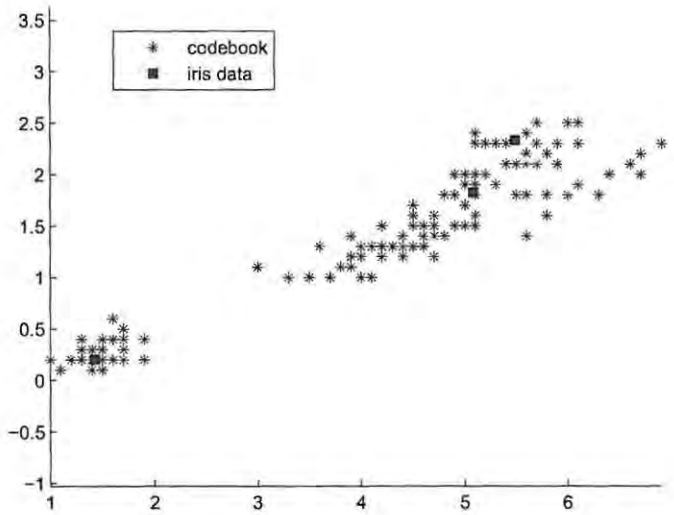
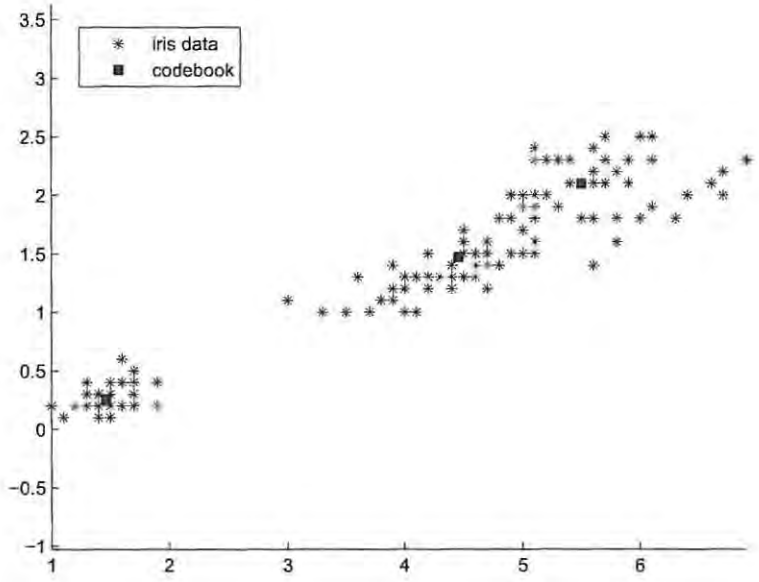


Figure 2.10: LVQ example using Iris data with 3 code vectors
Codebook after 10 iterations



to a distance measure on a topological ordering of the code vectors. The size of the neighbourhood is decreased with each iteration in the algorithm. The update of the code vectors is given by

$$c_j^{t+1} = \begin{cases} c_j^t + \eta_t \left(x_i - c_j^t \right), & i \in N(j) \\ c_j^t, & \text{otherwise} \end{cases} \tag{2.48}$$

where c_j^t is the winner code vector at time t, $N(j)$ is the set of code vectors that belong to the neighbourhood of c_j and η is the learning rate that is decreased with time. Note that if the neighbourhood only contains j then the update is reduced to the LVQ update equation (2.36). Thus the LVQ algorithm is a special case of SOFM, where each neighbourhood only contains one element which is the code vector itself. The disadvantage of using this algorithm is that there is additional computation during training which is as a result of having to calculate the neighbourhood in each iteration and also updating all the code vectors within the neighbourhood. The neighbourhood function however ensures that many of the code vectors are affected by the input vectors and thus aids as a solution to the under utilisation of some of the code vectors [8].

3 Edge detection and edge preserving codebook design

Edges play an important role in the visual perception of an image. We thus think it is necessary to examine the effect of image compression on the regions that contain edges. We begin this chapter by describing the properties of edges and giving a method of detecting edges in images. In the first section we also look at how the codebooks that are generated using algorithms in Chapter 2 affect the edges. In the second section we describe two methods that have been suggested to improve edge preservation in codebook design. We further give a description of the methods that we implemented which were drawn from these two methods.

3.1 Edge Detection

3.1.1 Properties of edges

In a gray scale image the edges are the regions where there is an abrupt gray level change between pixels. They are an important part of the visual quality of an image [17],[40]. It is therefore necessary that the edges are reconstructed well in image compression. One of the problems that arise in using vector quantization in image compression, is edge degradation [17],[40]. The edge pixels constitute a small portion of image pixels and are poorly reconstructed because most code vectors are used for the non edge areas where the vectors are more concentrated. Figure 3.1 shows an example of the distribution of image vectors for the two and three dimensional case (i.e 1×2 blocks and 1×3 blocks) respectively.

If (α, β) are the pixel values for a 1×2 block then the blocks that lie along the diagonal in Figure 3.1, are those that have similar values for α and β. These blocks are either part of horizontal features or taken from homogenous regions of the image. Similarly with 1×3 blocks, if (α, β, γ) were to represent the pixel values in a block where $\alpha \approx \beta \approx \gamma$,

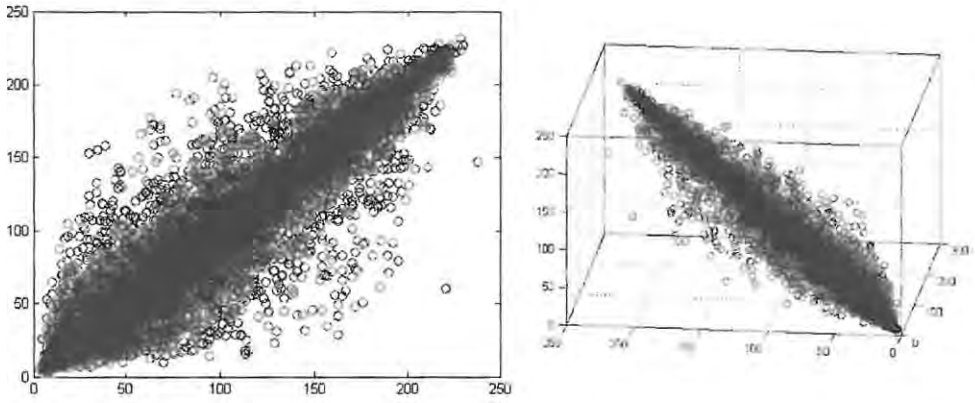


Figure 3.1: Lenna image, 256 × 256, Left: 1 × 2 blocks Right: 1 × 3 blocks

then the block is either taken from a homogenous region or part of a horizontal feature and would also lie along the diagonal of the 3 dimensional plot in Figure 3.1. The blocks that lie further away from the diagonal are those that represent edges. In the context of the pixel values the edges are represented by blocks that have varying pixel values. For 1 × 2 blocks this would be where either $\alpha << \beta$ or where $\alpha >> \beta$. Although we cannot visualise it, this may also be extended to the case where we have larger block sizes. During VQ most of the code vectors are assigned to the vectors that lie along the diagonal $\triangle$, where $\triangle$ is the set of points $(x_1, x_2, ..., x_n)^T$ such that $x_1 = x_2 = \ldots = x_n$, because there is a greater density of vectors located there. Figure 3.2 shows a typical distribution of code vectors obtained by the CM algorithm with $c = 64$ centres.

The type of figure that has been used to show the distribution of cluster centres is called a *Voronoi* diagram. The 'dots' represent the clusters centres, while the demarcated regions around the centres are the clusters derived using the NN condition. Thus Voronoi regions are nearest neighbour regions [1],[15] and a Voronoi diagram is used to represent these regions graphically. In the example shown in figure 3.2 most of the code vectors are assigned to the diagonal where there are non-edge vectors, hence the edges are poorly reconstructed. As a result, images reconstructed from VQ often exhibit the staircase effect [17] along object boundaries. This effect is shown in figure 3.3.

To preserve the edges it is necessary to be able to identify them. Edge detection techniques may be used to this end. There are various ways of implementing edge detection, one of which is using gradient operators. The next section gives details on the use of gradient operators in edge detection. One of the methods that will later be discussed

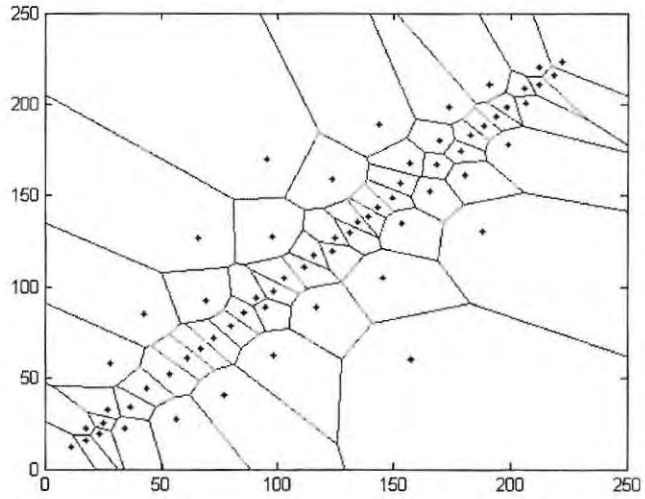


Figure 3.2: Distribution of code vectors, Lenna image, 256 × 256 and 1 × 2 blocks using 64 code vectors

Figure 3.3: Upper left: Detail of hat (reconstructed); Upper right: Detail of hat (original); Lower: Original 512 × 512 reconstructed Lenna image, 4 × 4 blocks, codebook size = 256

considers separating edges and quantizing them separately. In this method, edge detection is used prior to quantization to separate the edge vectors from the non-edge vectors and thus be able to quantize the edges separately.

3.1.2 Edge detection using gradient operators

Gradient operators may be used to measure the pixel value gradient in different directions. There are various gradient operators including the Prewitt [41], Kirsch [42] and Sobel [43],[44] operators. For us to be able to use the gradient, it is necessary to be able to get an estimate for it.

Estimating the gradient

Let f be a real valued function which has continuous derivatives up to order 3. We consider the Taylor approximation of $f'(r)$ [39]

$$f(r+h) = f(r) + hf'(r) + \frac{h^2}{2}f''(\omega) \tag{3.1}$$

with some $r \leq \omega \leq r+h$ and $h > 0$. From this we obtain

$$\begin{aligned} f'(r) &= \frac{f(r+h)-f(r)}{h} - \frac{1}{2}hf''(\omega) \quad \text{for} \quad h > 0 \\ \Longrightarrow f'(r) &\approx \frac{f(r+h)-f(r)}{h} \end{aligned} \tag{3.2}$$

We can obtain a better approximation if we consider the following Taylor approximations:

$$f(r+h) = f(r) + hf'(r) + \frac{1}{2}h^2f''(r) + \frac{1}{6}h^3f'''(\omega_1) \tag{3.3}$$

$$f(r-h) = f(r) - hf'(r) + \frac{1}{2}h^2f''(r) - \frac{1}{6}h^3f'''(\omega_2) \tag{3.4}$$

with $\omega_1 \in [r, r+h]$ and $\omega_2 \in [r-h, r]$. Subtracting these equations, we obtain

$$f(r+h) - f(r-h) = 2hf'(r) + \frac{1}{6}h^3(f'''(\omega_1) + f'''(\omega_2)) \tag{3.5}$$

and then solving for $f'(r)$,

$$f'(r) = \frac{f(r+h) - f(r-h)}{2h} - \frac{1}{6}h^2 \frac{f'''(\omega_1) + f'''(\omega_2)}{2} \tag{3.6}$$

If $f'''(r)$ is continuous then by the Intermediate Value Theorem there $\omega \in [r-h, r+h]$ such that

$$f'''(\omega) = \frac{f'''(\omega_1) + f'''(\omega_2)}{2}$$

and

$$f'(r) = \frac{f(r+h) - f(r-h)}{2h} - \frac{1}{6}h^2 \frac{f'''(\omega)}{2}$$

Therefore we can approximate $f'(r)$:

$$f'(r) \approx \frac{f(r+h) - f(r-h)}{2h} \tag{3.7}$$

We use this approximation to estimate the gradient for each pixel location. If $I(r,s)$ is the gray value of an image at pixel location (r,s) then the partial derivative is given by

$$\frac{\partial I}{\partial r}(r_o, s_o) \approx \frac{1}{2}\left[I(r_o+1, s_o) - I(r_o-1, s_o)\right]$$

where $h = 1$ (the stepsize $h = 1$ because the minimum pixel distance is 1). To get an approximation which is less sensitive to noise we average (smooth) in the s direction.

$$I(r_o+1, s_o) \approx \frac{1}{4}\left(I(r_o+1, s_o-1) + 2I(r_o+1, s_o) + I(r_o+1, s_o+1)\right)$$

If we use Sobel weights

$$\begin{aligned} \Longrightarrow \frac{\partial I}{\partial r}(r_o, s_o) \approx\ & [I(r_o+1, s_o-1) + 2I(r_o+1, s_o) + I(r_o+1, s_o+1) - I(r_o-1, s_o-1) - \\ & 2I(r_o-1, s_o) - I(r_o-1, s_o+1)] \end{aligned} \tag{3.8}$$

With the convolution mask,

$$S_\varepsilon = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.9}$$

where S_ε is the Sobel mask, we obtain

$$S_\varepsilon = \begin{bmatrix} I(r_o-1, s_o-1) & I(r_o-1, s_o) & I(r_o-1, s_o+1) \\ I(r_o, s_o-1) & I(r_o, s_o) & I(r_o, s_o+1) \\ I(r_o+1, s_o-1) & I(r_o+1, s_o) & I(r_o+1, s_o+1) \end{bmatrix} = 8\frac{\partial I}{\partial r}(r_o, s_o) \quad (3.10)$$

This then gives us an estimate for the gradient for each pixel location (r_o, s_o) but this is only in one edge direction. A similar derivation may also be used to get the gradient in other directions by rotating the boundary elements of the mask.

The convolution operator assigns a value to a pixel that is derived from the weighted average of the pixel values in a $v \times v$ neighbourhood around the central pixel [15]. The weights are usually in the form of a square matrix that is called the convolution mask. The convolution operation is defined by the relationship between the elements of the input image $I(r, s)$, the elements of the convolution kernel $S_\varepsilon(\rho, \tau)$ and the elements of the output image $g(r, s)$ [15]:

$$g(r, s) = \sum_{\rho=-\infty}^{\infty} \sum_{\rho=-\infty}^{\infty} I(r, s) S_\varepsilon(r-\rho, s-\tau). \quad (3.11)$$

The square array S_ε is usually such that v is odd and much smaller than the image dimension. The convolution operation may then be written as

$$g(r, s) = \sum_{\rho=-(k-1)/2}^{(v-1)/2} \sum_{\rho=-(k-1)/2}^{(v-1)/2} I(r, s) S_\varepsilon(r-\rho, s-\tau). \quad (3.12)$$

For the Sobel operator the convolution mask is given by [43] equation (3.9) which gives us the mask to get the gradient of edges in the north direction. When this mask is rearranged by rotating the eight boundary elements, we get masks for seven more different directions [45]: south (S) , north west (NW), south east (SE), west (W), east (E), north east (NE) and south west (SW).

$$S_1 = S_N = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} S_2 = S_S = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} S_3 = S_{NW} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$$

$$S_4 = S_{SE} = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \quad S_5 = S_W = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad S_6 = S_E = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$S_7 = S_{NE} = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix} \quad S_8 = S_{SW} = \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix}$$

If $g_\varepsilon(r,s)$ is the gradient in the direction ε where

$$\varepsilon = \begin{cases} 1 & north \\ 2 & south \\ 3 & north\ west \\ 4 & south\ east \\ 5 & west \\ 6 & east \\ 7 & south\ west \\ 8 & north\ east \end{cases} \tag{3.13}$$

then the gradient at pixel location (r,s) is defined as [43]

$$g(r,s) = \max_{\varepsilon}\{|g_\varepsilon(r,s)|\} \tag{3.14}$$

where $g_\varepsilon(r,s)$ is the convolution of the mask in direction ε with the image I at pixel location (r,s) calculated using equation (3.14).

Pixel location is defined as an edge location if $g(r,s)$ exceeds a threshold t_ε such that those pixels with the largest gradients are identified as edges. The edge map $\epsilon(r,s)$ of an image is defined as

$$\epsilon(r,s) = \begin{cases} 1 & (r,s) \in I_g \\ 0 & \text{otherwise} \end{cases} \tag{3.15}$$

where

$$I_g \triangleq \{(r,s); g(r,s) > t_\varepsilon\} \tag{3.16}$$

The edge map $\epsilon_\varepsilon(r,s)$ of a image in direction ε is given by

$$\epsilon_\varepsilon(r,s) = \begin{cases} 1 & (r,s) \in I_\varepsilon \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

where

$$I_\varepsilon = \{(r,s) : g(r,s) = g_\varepsilon(r,s) > t_\varepsilon\} \tag{3.18}$$

Figure 3.4 displays the edge maps for different thresholds for the Lenna image. Figure 3.5-3.6 show edge maps for different edge directions for the Lenna image.

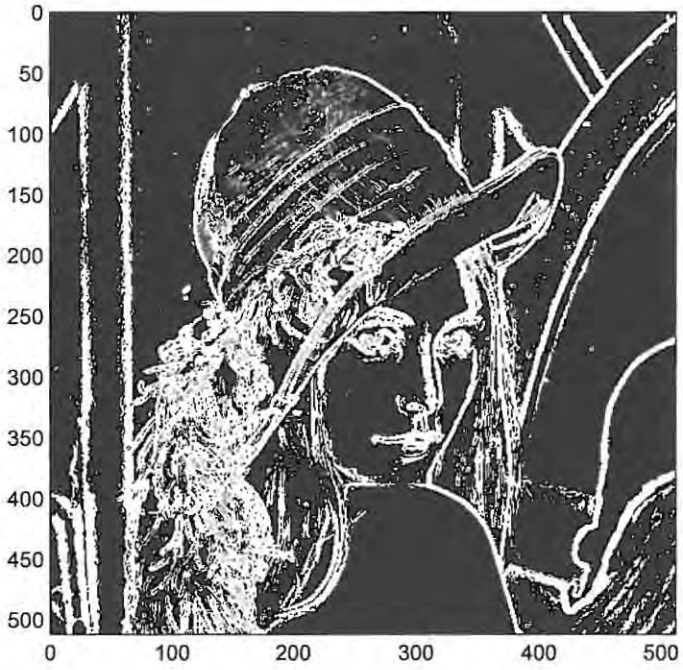
Edge detection may be used to improve edge reconstruction. A method that employs this was suggested by Ramamurthi and Gersho [1],[17] who suggested that blocks are separated into edge classes depending on the orientation of edges in their classes, and then VQ performed on each class separately. Another method that may be used to overcome the problem of poor edge reconstruction was suggested by Chou, Su and Lai [40] that does not use the edge detection approach. They suggested that edge vectors are replicated so that more code vectors are forced into the regions with edges during quantization. The following sections give descriptions of these two approaches.

3.2 Classified vector quantization

3.2.1 Ramamurthi and Gersho Method

Ramamurthi and Gersho [17] employ classified vector quantization (CVQ) as a method to overcome poor edge reconstruction. In their approach the image blocks are separated into edge classes and VQ performed on each class separately. This ensures that for any given image vector only a code vector that belongs to the same edge class as the input vector may be used for coding. Before quantization may be performed, the edge orientation of each image block is determined. They restrict the number of edge orientations to four directions: horizontal, vertical and two diagonals. They further distinguish all possible locations for each orientation. These are subdivided further into two classes, depending on the polarity, that is from high to low pixel value and vice versa. The edge blocks are therefore classified according to these three criteria: edge orientation, location and polarity. The suggested classification algorithm has two stages: the edge enhancement stage and a decision tree. The first stage uses an edge detector and thus the gradients

Figure 3.4: Edge maps for Lenna image using Sobel operator
(a) Lenna image with $t = 50$



(b) Lenna image with $t = 150$

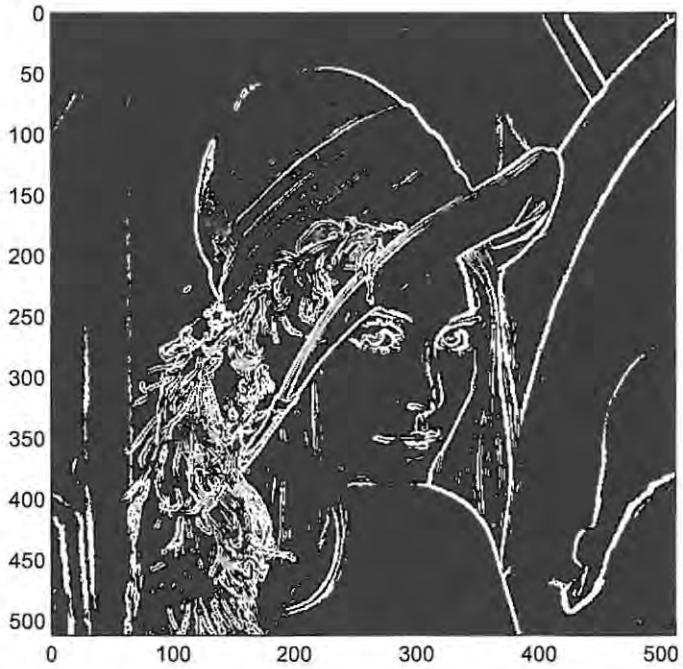
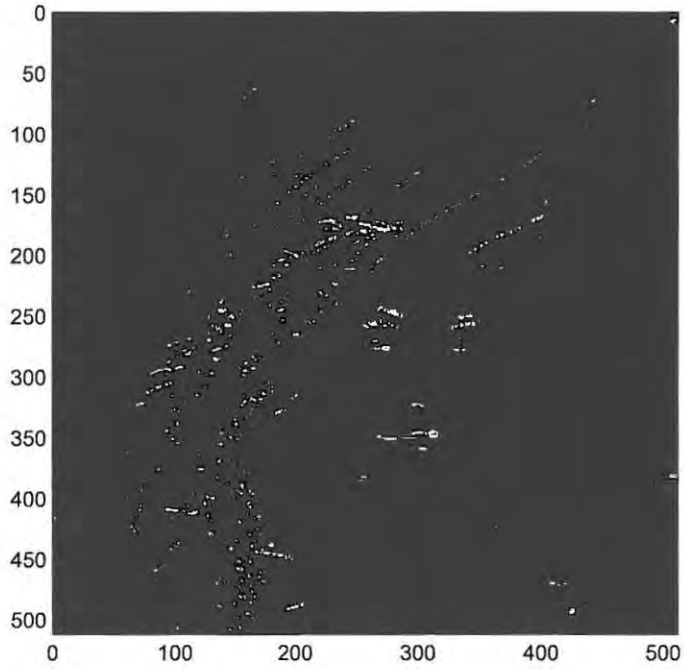


Figure 3.5: Edge maps for Lenna image using Sobel operator
(a) North edges



(b) North west edges

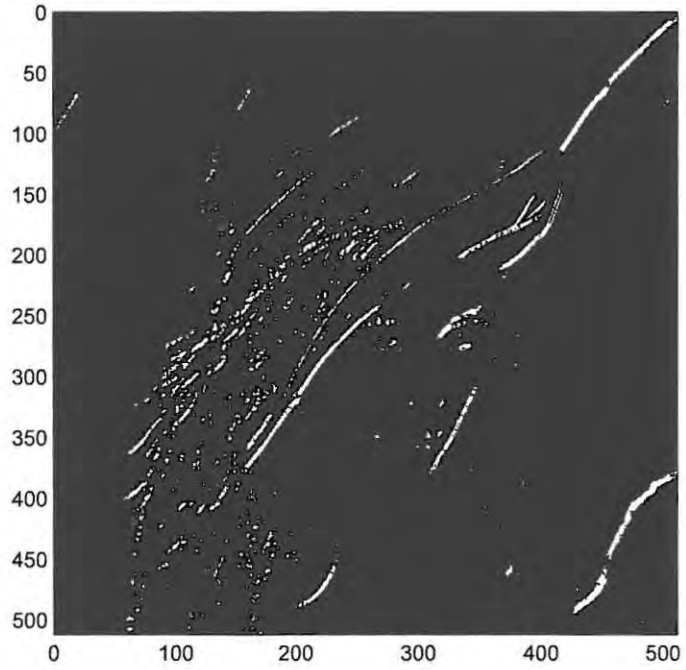
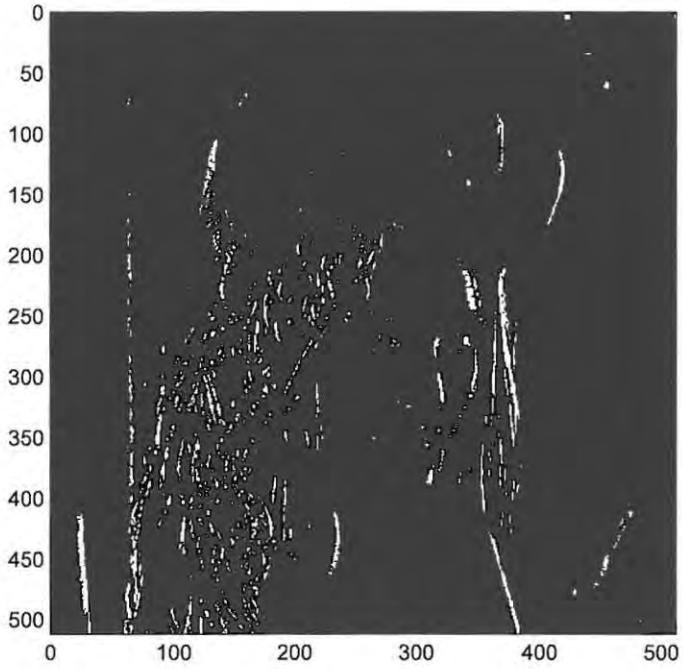
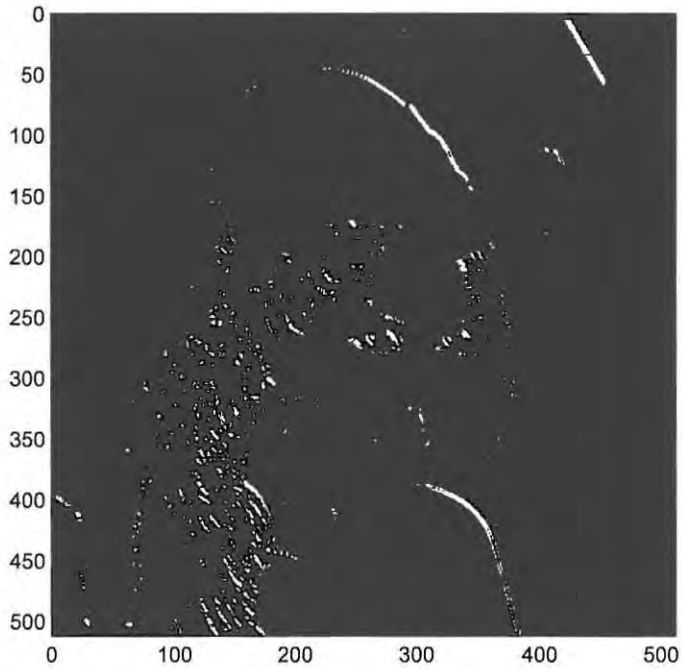


Figure 3.6: Edge maps for Lenna image using Sobel operator
(a) West edges



(b) South west edges



for each pixel are estimated. The decision tree is then used to decide the edge class that each vector will belong to depending on the criteria mentioned before. For classification there are a total of 7 classes that are subdivided into the possible locations and polarity as described below.

1. Shade - non-edge class for all blocks that have no significant gradient.
2. Midrange - non-edge class for all blocks with moderate gradient.
3. Horizontal edges
4. Vertical edges
5. 45^o diagonal edges
6. 135^o diagonal edges
7. Mixed class for all blocks that do not have a definite edge orientation or a mixture of more than one edge orientation.

Figure 3.7 gives some examples of the blocks that would belong to each class for the classification of 4×4 blocks.

In designing the codebook the blocks in the training set are separated by edge orientation, location and polarities. For 4×4 blocks 3 locations are suggested for the horizontal and vertical classes and 4 locations for the diagonal classes with 2 polarities for each location. This together with the mixed class and 2 non-edge classes would give a total of 31 classes. Each of these classes will have training done separately and thus produce 31 subcodebooks that are combined to create a codebook that is used for testing. In testing the codebook, the classifier is also used to subdivide the test set into edge classes. Only the subcodebook that is associated with each class is used to reconstruct the blocks in that class. Thereafter the reconstructed blocks are rearranged to form the reconstructed image.

3.2.2 Our approach to classified vector quantization

For the purposes of our experiments here a similar approach is used with less classes for classification. Ramamurthi and Gersho [17] suggest that any edge detector may be used for edge classification hence we use the edge detector described in section 3.1.2 to classify the pixels in an image. The classification of pixels is done by calculating the convolution

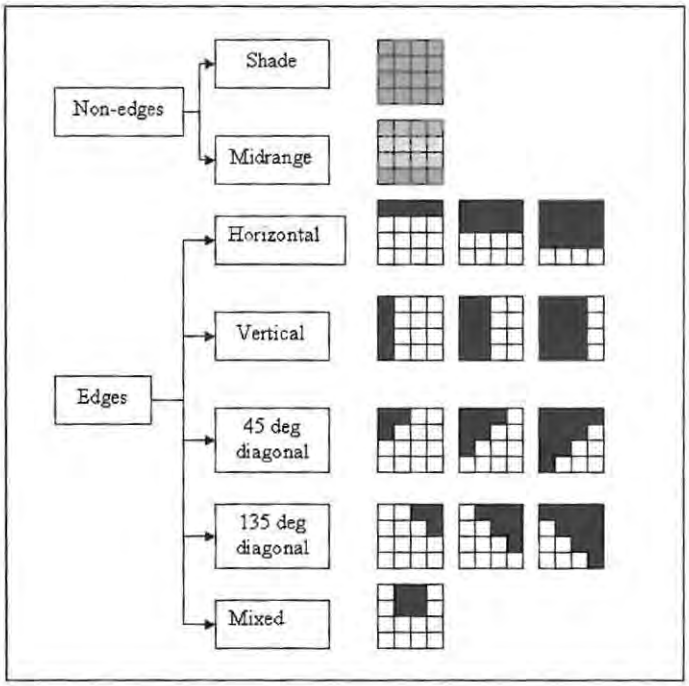


Figure 3.7: Edge classification for 4 × 4 blocks with examples of different locations

$\langle I, S_\varepsilon \rangle$ of the image I, with the Sobel masks S_ε for $\varepsilon = 1, 2, ..., 8$ to estimate the gradient $g(r, s)$ using equation (3.14). The threshold t_ε is then used to decide if the pixel is an edge pixel. If $g(r, s) > t$ then pixel (r, s) is an edge pixel. If $g(r, s) = g_\epsilon(r, s)$ then pixel (r, s) is classified as a pixel that belongs to edge class ε that represents all the edge pixels in edge direction θ_ε. This then allows us to determine the edge direction of the pixel by taking the direction that has the largest gradient. If the pixel does not belong to any of these edge classes then it is assigned to the non edge class, $\varepsilon = 9$. The pixel classification process is summarised in the following pseudo code:

Input I, *masks* S_m^ϵ *for* $\varepsilon = 1, 2, ..., 8$

for $\varepsilon = 1, 2, ..., 8$

    Calculate $g_\varepsilon(r, s) = \langle I, S_m^\epsilon \rangle$

end for

$g(r, s) = max_\varepsilon |g_\varepsilon(r, s)| \quad \forall r, s$

if $g(r, s) > t_\varepsilon$ *and* $g(r, s) = g_\epsilon(r, s)$

    $I(r, s) \in E_\varepsilon$

else $I(r, s) \in E_9$

end if

Output: Pixel classes $I_p(r,s)\ \forall r,s$

Once the pixels are classified then we are able to classify the blocks. The training image/s is subdivided into blocks which are represented by vectors. First the edge blocks are identified by counting the number of pixels classified as edges. If the number of edge pixels N_ε in the block is equal to or exceeds the minimum number of pixels (t_p) required for a block to be identified as an edge block, then it is identified as an edge block. The value for t_p varies depending upon the block size that is used. For 4×4 blocks in Ramamurthi and Gersho [17] and Chou, Su, and Lai [40], $t_p = 6$, that is, for a block to be classified as an edge it must contain at least 6 edge pixels in it. The blocks are then classified according to the dominant edge direction of the pixels contained in the block. The variance of the gray values of the pixels in each block may also be used to check the blocks that have been identified as edges to ascertain whether they are true edge blocks. If the variance of the block exceeds the variance threshold V_t and satisfies the rest of the criteria given above then it is classified as an edge block otherwise it is put into the non edge class. The classification of the blocks (4×4) may be summarised as follows:

Input : training image vectors $I = x_1, x_2, ..., x_N$, t_p, V_t, *pixel classes* I_p

convert array of pixel classes I_p *to vectors* I_pv

count number of edge pixels $N_e(x_k)\ \forall k$

If $N_e(x_k) > t_p$

$x_k =$ *edge block*

else $x_k =$ *non-edge block*

end if

Calculate $N_\varepsilon(x_k) =$*number of pixels in class* $\varepsilon\ \forall k$

If $N_m = max(N_\varepsilon(x_k))$ *for* $m = 1, 2, ..., 9$

$x_k \in E_m$

end if

Output: Block classes B_ε

The edge class numbers and their edge directions for the blocks are given in equation (3.13) and the non-edge class is represented by $\varepsilon = 9$.

$$\varepsilon = \begin{cases} 1 & north \\ 2 & south \\ 3 & north\ west \\ 4 & south\ east \\ 5 & west \\ 6 & east \\ 7 & south\ west \\ 8 & north\ east \\ 9 & non\ edge \end{cases}$$

To create the codebook, an initial codebook that is subdivided into 9 subcodebooks is employed. There are eight subcodebooks for the eight directional edge classes and one subcodebook for the non-edge class. Each subcodebook is worked with separately. Clustering algorithms are used to create a subcodebook for each class separately using training vectors in the respective classes. The codebook can then be tested on a test image. Each pixel in the test image is classified. The test image is divided into blocks which are then classified according the type of edges that they have. The blocks are then separated into their edge classes. For a block in edge class ε the vector is quantized by looking up the closest code vector in the class subcodebook ε. The block vectors are quantized and reconstructed. All the quantized vectors are then combined to reconstruct the new image. The creation and testing of the codebook may be summarised as follows:

- Start with 9 initial subcodebooks one for each class
- Using vectors from the associated classes, use clustering algorithms to create subcodebooks for each class
- Subdivide test image into blocks and convert into vectors
- Classify test image vectors
- Look up code vectors in sub codebook corresponding to the class that image vectors belong to.
- Reconstruct test image

In this method we only look up the codebook of the corresponding class of each vector, thus the computational load is reduced during VQ. This is one of the advantages that are

given in [17] about using CVQ. The CVQ algorithm may be summarised in the following way:

Input : initial codebooks $C_{\varepsilon 1}^0, C_{\varepsilon 2}^0, ..., C_{\varepsilon 9}^0$, *training image* I_{tr}, *test image* I, t, t_p, V_t

classify pixels by estimating g(r,s) for I_{tr} *and* I

classify blocks for I_{tr} *and* I

Check edge blocks using V_t *for* I_{tr} *and* I

Separate blocks into edge classes for training image $I_{tr,\varepsilon}$ *and test image* I_ε

for $\varepsilon = 1, 2, ..., 9$ *do the following*

Initialise codebook $C_\varepsilon^0 = c_1, c_2, ..., c_{c\varepsilon}$

Use clustering algorithms to create codebook C_ε *for each class with* $I_{tr,\varepsilon}$

quantize test image vectors I_ϵ

reconstruct $Q(I_\varepsilon)$

endfor

Combine $Q(I_\varepsilon)$ *for all* ε

Output: reconstructed image $Q(I)$

3.3 Replication method

VQ usually assigns a greater number of code vectors to the areas where there is a high density of vectors [40]. The areas that contain edges are usually sparse because they are a small portion of the whole data set. Chou, Su and Lai [40] suggested that the vectors in the sparse regions be replicated so that more code vectors are used to code the edge vectors and thus improve the quality of the edges in the reconstructed image. The sparse areas are identified by measuring the density of the region that is centred at each image vector. The density $D(x_j)$ of the region centered at the image vector x_j is given by [40]

$$D(x_k) = \sum_{j=1, j\neq k}^{N} exp\left(-\frac{||x_j - x_k||^2}{\sigma_d^2}\right) \tag{3.19}$$

where σ_d is chosen to be

$$\sigma_d^2 = E[(d_{jk} - E(d_{jk}))^2] \tag{3.20}$$

where $d_{jk} = ||x_j - x_k||$. The larger the value of $D(x_k)$ the denser the region around x_k. A virtual data set can then be created by replicating each image vector x_k by a factor $N_d(x_k)$ that is given by [40]

$$N_d(x_k) = \left\lfloor 10 \times log_{10} \left(\frac{max_{j=1,2,...,N} D(x_j)}{D(x_k)} \right) \right\rfloor + 1 \tag{3.21}$$

where $\lfloor \rfloor$ is the floor operator. The lower the value of $D(x_k)$ the higher the value of N_d which means there are more replicates of that vector. If x_k belongs to a dense region the value of $N_d(x_k) = 1$ meaning that x_k will not be replicated. The total number of vectors in the virtual data set is $N^v = \sum_{k=1}^{N} N_d(x_k)$. The virtual data set is then used to create a codebook using clustering algorithms. Since the virtual data set is always larger than the initial data set this increases the computational load and thus slows down the clustering process.

Codebook design

The training image/s are subdivided into blocks that are converted to vectors $I_{tr} = \{x_1, x_2, ..., x_n\}$. The density $D(x_k)$ around each image vector x_k is calculated using equation (3.19). The replication number $N_d(x_j)$ is calculated for each image vector x_k according to equation (3.21). A virtual data set $I^v = x_1, x_2, ..., x_{N^v}$ is then created by replicating each image vector x_k by the number given by $N_d(x_k)$. Clustering algorithms are employed to create a codebook which tested on another image. The codebook design may be summarised as follows:

- Subdivide training image/s X into blocks and convert to vectors
- Calculate $D(x_k)$ and $N_d(x_k)$ for all k
- Replicate X according to $N_d(x_k)$ to create virtual set X^v
- Use clustering algorithms to create codebook using X^v
- Test the codebook on a test image

The summary of this method is given as pseudo code in the following:

Input : training image I_{tr}, test image I

Convert training image to vectors $I_{tr} = x_1, x_2, ..., x_N$

for $k = 1, 2, ..., N$

Calculate replication number $N_d(x_k)$ using equation (3.21)

end for

Replicate I to create I^v

Initialise codebook $C^o = \{c_1^o, c_2^o, ..., c_c^o\}$

Generate codebook using I^v with clustering algorithms

Quantize I

Reconstruct I

Output: reconstructed image $Q(I)$

4 Implementation and Results

This chapter gives a detailed description of experiments that were carried out to compare the performance of different clustering algorithms in codebook design, with respect to the preservation of edges. The clustering algorithms described in Chapter 2 are employed to generate codebooks for typical gray scale images. The performance of the algorithms is measured using the MSE, not only on the whole image but also on the edge regions. We take a closer look at the regions that contain edges in the image, and apply the modifications suggested in Chapter 3 to ascertain if the methods are effective in improving edge preservation in image compression. The results obtained are given, as well as an assessment of the results. In section 4.1 we describe the set up of the experiments. In section 4.2 the results of the c-means (CM), fuzzy c-means (FCM) and learning vector quantization (LVQ) experiments are given and discussed. The results of the classified vector quantization (CVQ) method and the replication method (RM) are given in section 4.3 and section 4.4 respectively.

The aims of the experiments may be summarised by the following:

1. To assess the performance of CM, FCM and LVQ in generating codebooks for image compression by applying them in the compression of gray scale images. This is done by:

    - measuring the MSE between the original and quantized image for different size codebooks.
    - measuring the MSE between the original edge vectors and the quantized edge vectors to see how they perform in the edge regions of the image.
    - reconstructing the images to observe their visual quality
    - comparing the results obtained using the different algorithms.

2. To assess the performance of CVQ and RM in edge preservation by

- carrying out image compression with codebooks generated using CVQ and RM.
- measuring the MSE's obtained on edge areas and non edge areas.
- observing the reconstructed images.
- comparing results obtained with these methods with the results obtained without using CVQ or RM.

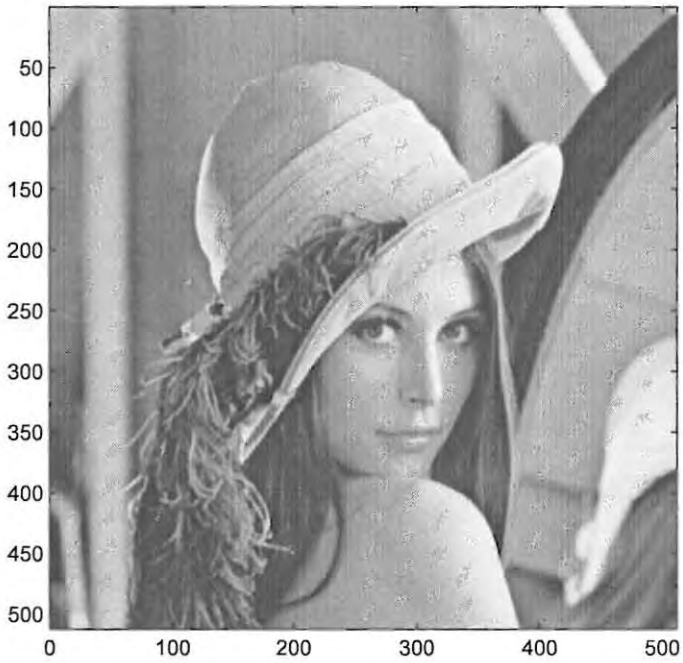
4.1 Conditions for experiments

For this investigation five gray scale images were used, each of size 512×512 pixels. The images are displayed in figure 4.1 and figure 4.2. These are typical gray scale images used for test purposes in image processing from USC-SIPI Image Database (`http:\\www.usc-sipi.edu`, accessed 10 Oct 2006). Matlab programming is used for all the experiments reported here. A cross validation approach is used to obtain a "good" estimate of the MSE. Therefore in each experiment four images are used as a training set to generate a codebook and this codebook is tested on the fifth image. Table 4.1 shows the training sets for each image.

As we would like to compare the algorithms and the results obtained using the methods for edge preservation, there are some parameters that are fixed for the main set of experiments. The investigation is carried out using 4×4 blocks and codebooks of size $2^8 = 256$, $2^9 = 512$ and $2^{10} = 1024$. As the bit rate (defined in section 1.5) is determined by the number of bits used for the codebook, we use codebook sizes that are powers of 2. This ensures that we have the maximum codebook size for a particular number of bits and thus have the largest codebook size for a particular bit rate. For example, codebooks with $129 \leq c \leq 255$ will still require 8 bits for coding. We use the maximum codebook size that uses 8 bits which is 256. The aim in image compression is a bit rate that is as low as possible without losing too much visual information and thus achieve a minimal MSE. Large codebooks are preferable in maintaining a low MSE, but however do not have low bit rates.

To be able to make an assessment of the reconstruction of the edges, we need to estimate the MSE not only for the whole image, but also for the vectors that belong to each edge class. Before the experiments are performed, the vectors that belong to each edge class are identified for each set of training images and the associated test image, using an edge detector as described in Chapter 3. These classes are maintained throughout

Figure 4.1: Images
(a) Lenna image sometimes described as 'the girl in a hat'



(b) Boat image

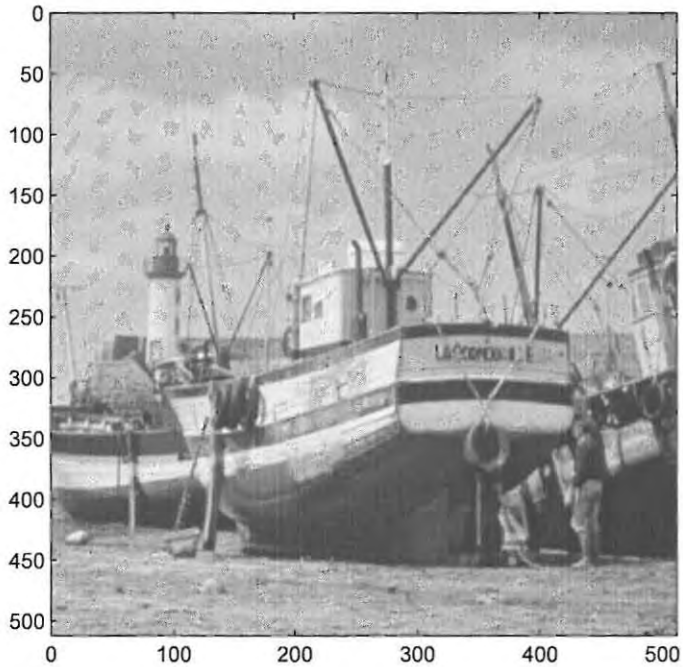
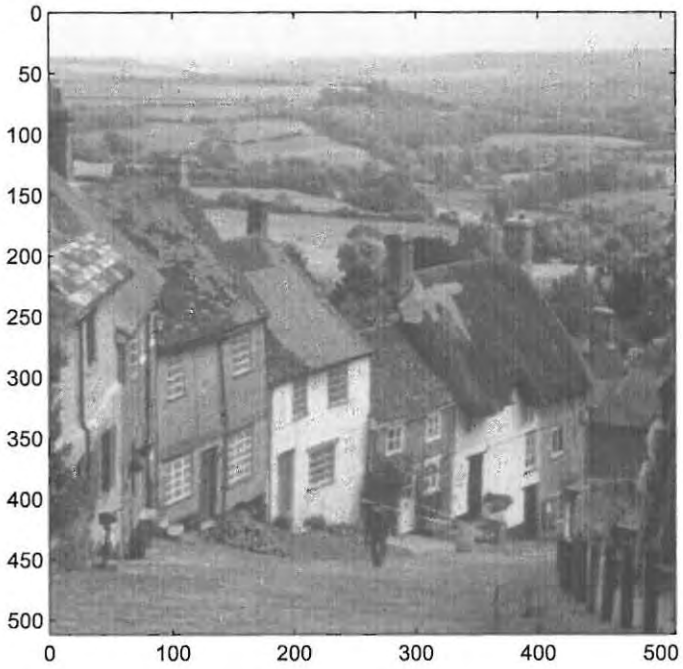


Figure 4.2: Images (continued)

(a) Goldhill



(b) Peppers

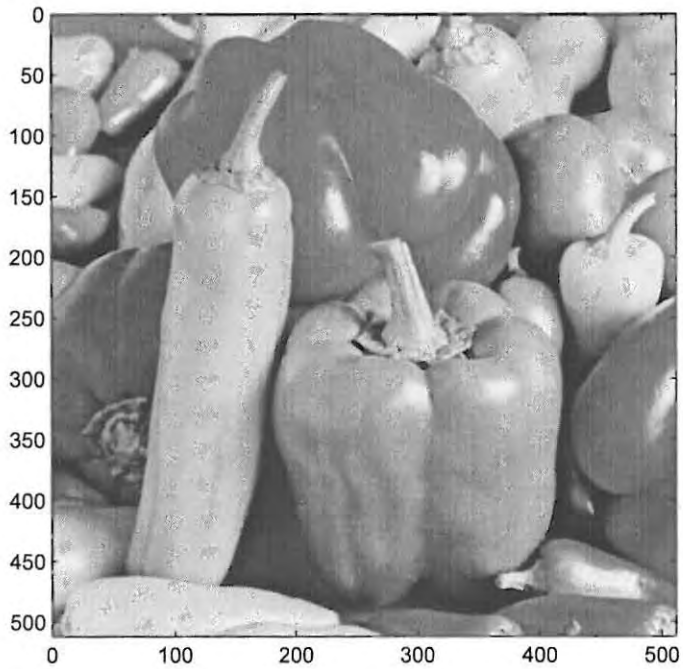
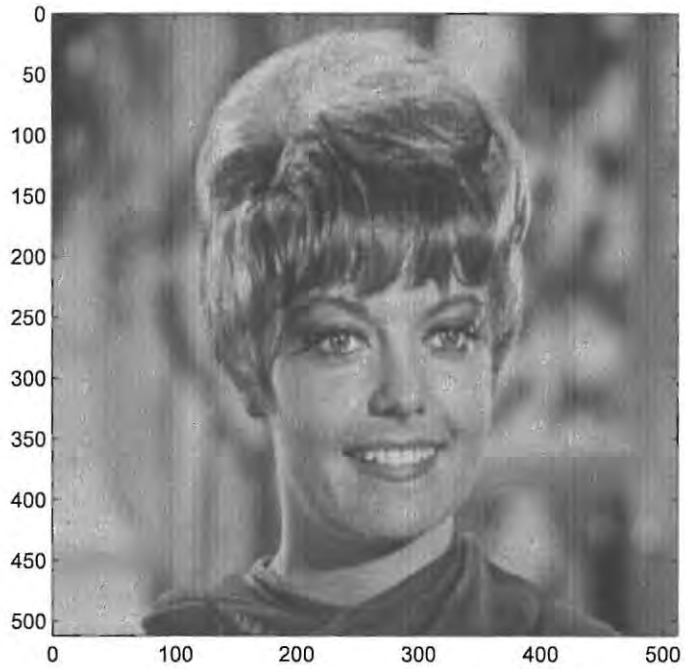


Table 4.1: Training and test sets

Test image	Training set images			
Lenna	Boat	Goldhill	Peppers	Zelda
Boat	Lenna	Goldhill	Peppers	Zelda
Goldhill	Lenna	Boat	Peppers	Zelda
Peppers	Lenna	Boat	Goldhill	Zelda
Zelda	Lenna	Boat	Goldhill	Peppers

Figure 4.3: Images (continued)

Zelda



the experiments. In the identification of the edge classes, there are 3 parameters of importance:

- the threshold t_ε for the edge map for the classification of pixels: pixels with an edge strength $> t_\varepsilon$ in the edge map are edge pixels,
- the minimum number of pixels, t_p that is necessary for a block to be classified as an edge block
- the variance threshold, V_t that must be used in order to check whether an edge block has been correctly classified: blocks that have a variance that is below V_t are not considered edge blocks.

In Ramamurthi and Gersho [17] for a set of 8 images the approximate proportion of edge pixels was found to be 27%. For the set of images used for our purposes we sought a threshold that would give us a similar proportion of edge pixels. Figure 4.4 shows the percentage of pixels that are classified as edges pixels for all five images. The threshold is chosen to be $t_\varepsilon = 40$ which allows us to have approximately 30% of the pixels as edges as is shown in figure 4.4. Once pixels have been classified, the next stage is block classification. To reduce the complexity in selecting parameters, we use here $t_p = 6$, which means that a block is considered an edge block if it contains a minimum of 6 edge pixels. This is also used in the examples given in Ramamurthi and Gersho [17] and Chou, Su and Lai [40].

The block variance is used to eliminate blocks that have been chosen as edge blocks when they do not actually have a significant gray value gradient. The block variance is given by

$$var(x) = \frac{\sum_{j=1}^{n} x_j^2 - n(\overline{x})^2}{n-1}$$

where x is the block vector and n is the number of pixels in the block. Edge blocks will have a larger variance as there is greater variation in the pixel values. A visualisation of the blocks that would belong to each class, for all the images is made for different variances to try and get a suitable threshold. We found that there were some that would be more suitably classified as non-edge blocks than as edge blocks for a threshold lower than 130. Figure 4.1 shows an example of the blocks that are eliminated from one of the diagonal classes by raising the threshold from 60 to 130. The top layer shows blocks that are homogenous that would be included in the diagonal edge class if the threshold $V_t = 60$. We observe that these blocks are not actually edge blocks when they are

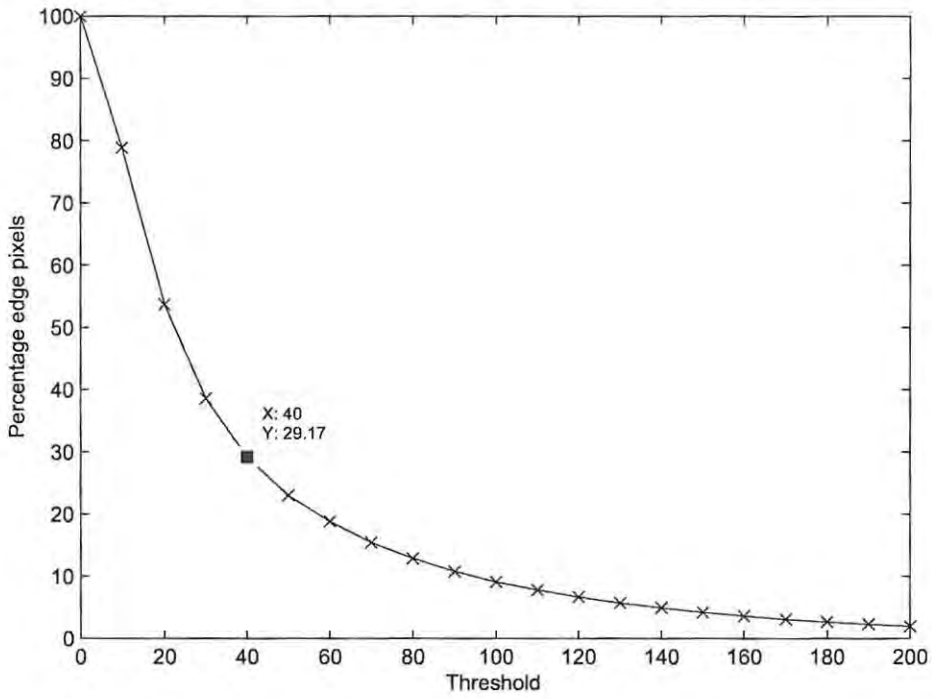


Figure 4.4: Percentage edge pixels against threshold (5 images)

Top row : Blocks that are included in E7 when $V_t = 60$
Bottom row: Blocks that are in E7 when $V_t = 130$

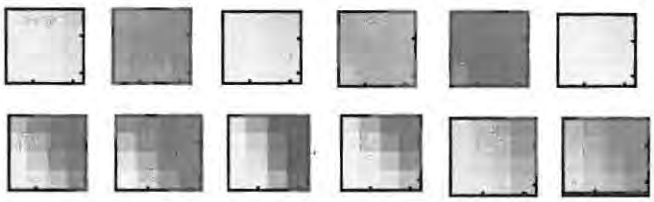
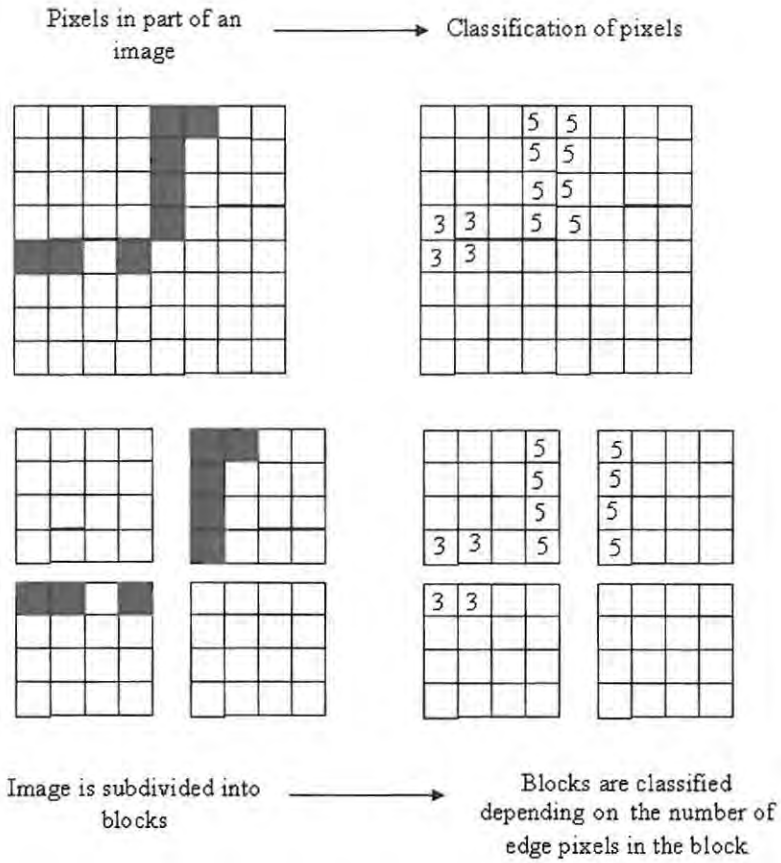


Figure 4.5: Image blocks in edge class E7 for $V_t = 60$ and $V_t = 130$

Figure 4.6: Block classification



visualised. Using a higher threshold aids us in eliminating misclassified blocks from the edge classes. To try and understand the reason for misclassification we examine the edge detector. The edge detector classifies each pixel in relation to the neighbouring pixels. At an edge location, neighbouring pixels are labeled as edge pixels. In VQ however we do not use the pixels but blocks from the image. This means that at times we may separate neighbouring pixels that are at an edge location when we subdivide the image into blocks. As a result we may end up with the scenario shown in figure 4.6, where the top left block is labelled as an edge block when it is homogenous block. The variance is thus used for the purpose of distinguishing homogenous blocks that are misclassified with those that do contain edges.

Table 4.2 shows the edge statistics, as percentages, of each image that is used in this

Table 4.2: Edge statistics of (a) training sets (b) test sets

(a) Training sets

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	Total
Lenna	3.1	3.2	2.6	2.3	3.5	3.2	1.6	1.7	78.7	100
Boat	2.2	2.2	2.6	2.4	3.2	3.0	1.6	1.5	81.3	100
Goldhill	2.2	2.2	2.7	2.5	3.4	3.3	1.6	1.4	80.8	100
Peppers	3.0	2.9	2.8	2.5	3.9	3.5	1.6	1.7	78.1	100
Zelda	3.2	3.2	3.1	2.7	4.0	3.7	1.8	1.8	76.5	100
Total	2.7	2.8	2.8	2.5	3.6	3.3	1.6	1.6	79.1	100

(b) Test sets

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	Total
Lenna	1.1	0.8	3.5	3.3	3.9	3.7	1.7	1.4	80.4	100
Boat	4.7	5.0	3.5	2.7	5.3	4.7	1.8	2.1	70.4	100
Goldhill	5.1	4.9	2.9	2.5	4.4	3.6	1.9	2.4	72.3	100
Peppers	1.8	2.2	2.6	2.2	2.2	2.8	1.9	1.3	83.0	100
Zelda	1.1	0.8	1.3	1.6	2.2	1.8	1.0	0.9	89.3	100
Total	2.7	2.8	2.8	2.5	3.6	3.3	1.7	1.6	79.1	100

analysis, for $t = 40$, $t_p = 6$, $V_t = 130$. Figure 4.7 - 4.11 display bar charts of the edge statistics of the training set and the test set for each image. A separate bar chart for the non-edge classes for all the images is displayed in figure 4.12. The edge maps in figure 4.13-4.14 show the regions in each image which have blocks regarded as edge blocks.

We estimate the MSE between the original image vectors and the quantized image vectors by calculating

$$MSE = \frac{1}{N} \sum_{i=1}^{N} ||x_k - Q(x_k)||^2 \tag{4.1}$$

where N is the number of image vectors and $Q(x)$ is the vector that is used to represent x_k in the quantized image.

To estimate the MSE for each edge class, the vectors from the original and the quantized image are used in the following:

$$MSE = \frac{1}{N_\varepsilon} \sum_{i=1}^{N_\varepsilon} ||x_k^\varepsilon - Q(x_k^\varepsilon)||^2 \tag{4.2}$$

where N_ε for $\varepsilon = 1, 2, ..., 9$ is the number of vectors in the edge class ε and x_k^ε for $k = 1, 2, ..., N_\varepsilon$ are the image vectors that belong to the edge class.

Figure 4.7: Bar chart of test and training sets for Lenna image

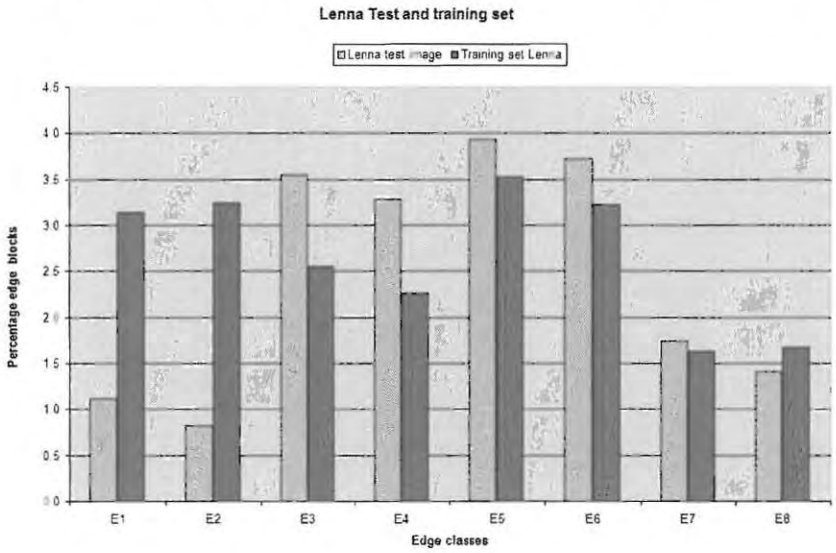


Figure 4.8: Bar chart of test and training sets for Boat image

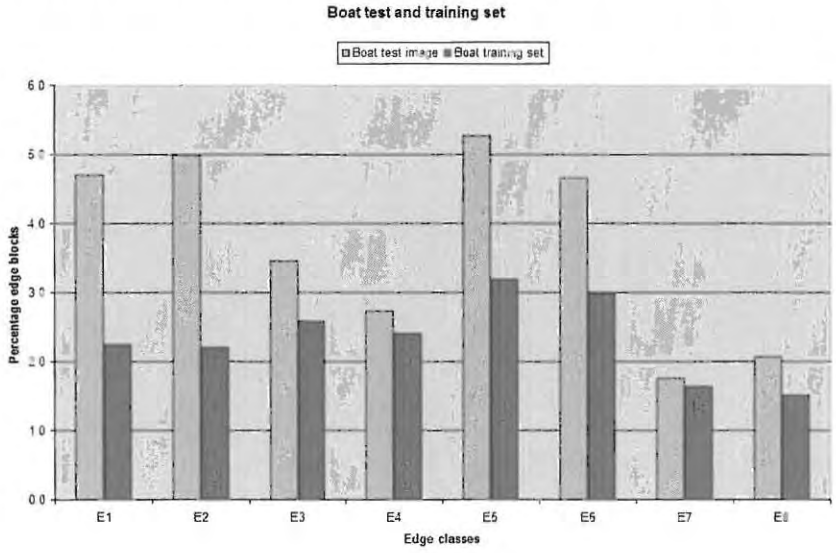


Figure 4.9: Bar chart of test and training sets for Goldhill image

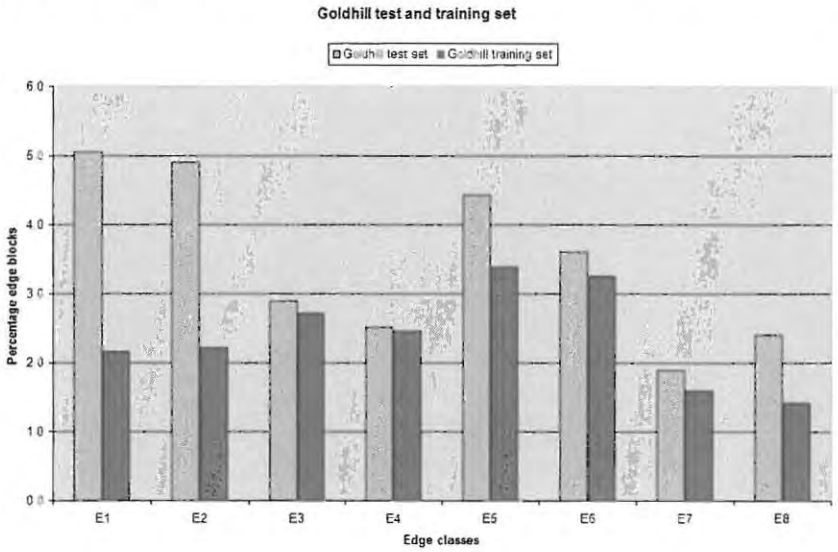


Figure 4.10: Bar chart of test and training sets for Peppers image

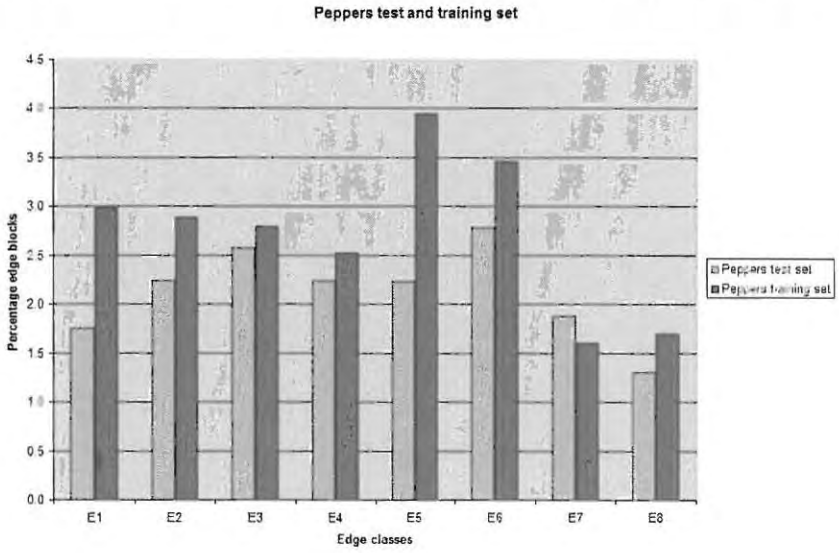


Figure 4.11: Bar chart of test and training sets for Zelda image

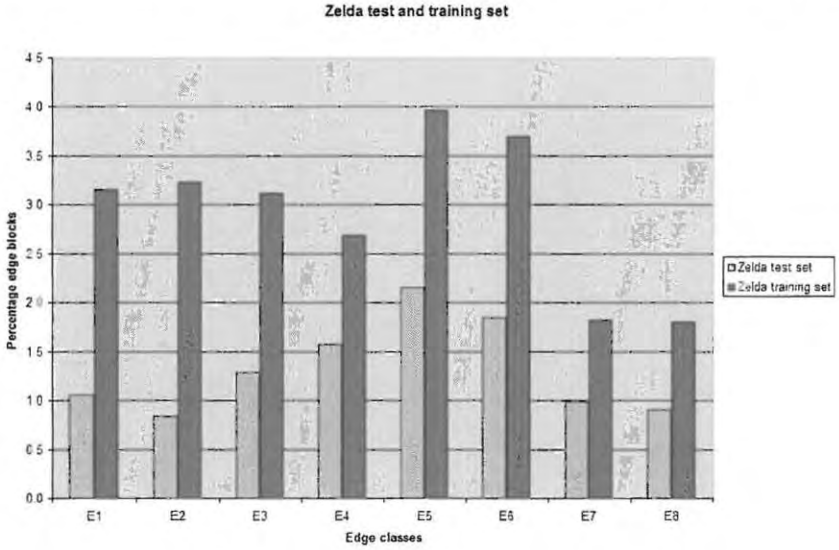


Figure 4.12: Non edge class training and test sets for all images

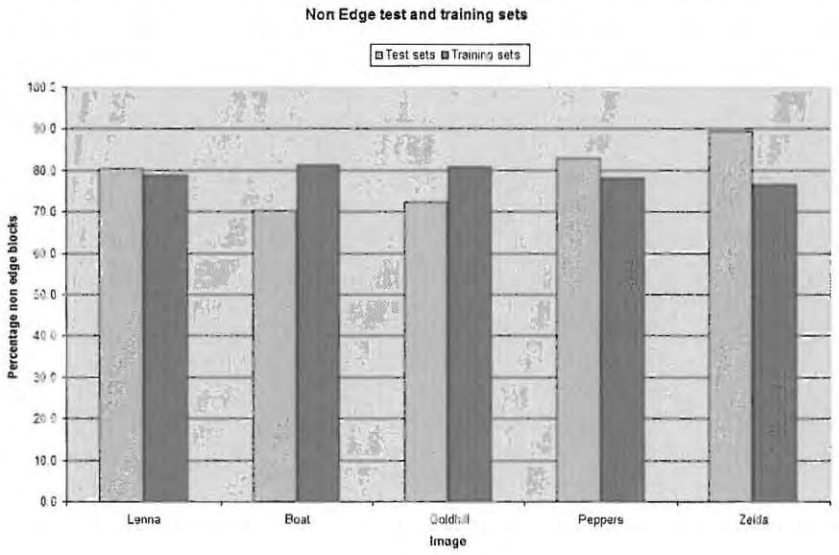
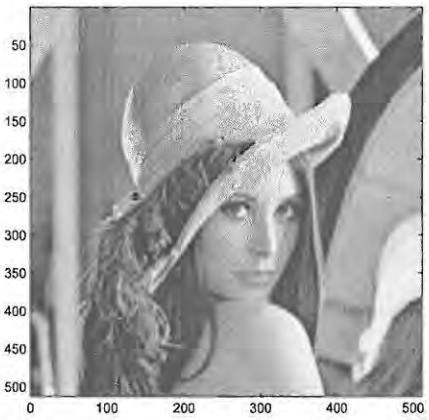
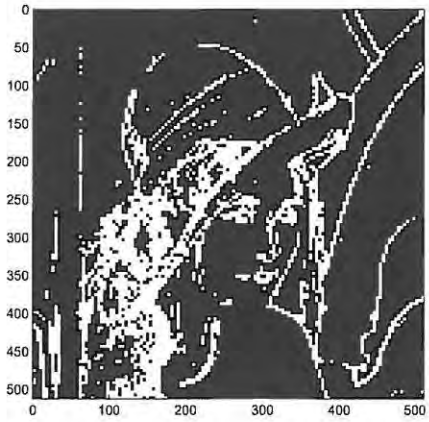
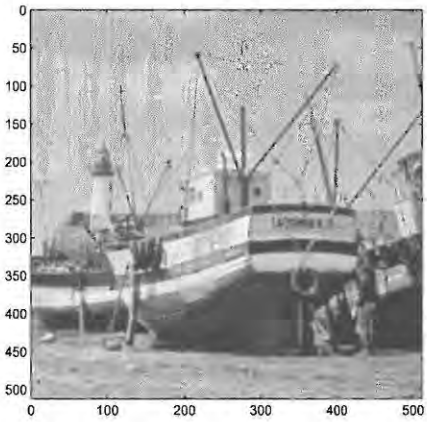
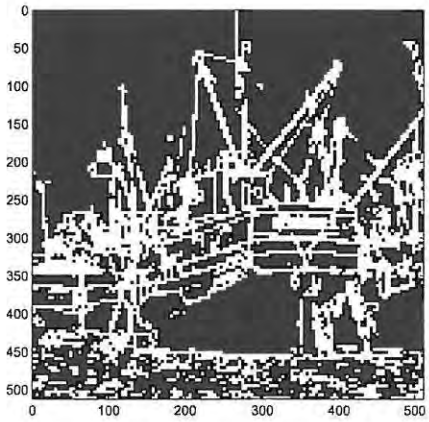


Figure 4.13: Edge maps for $t_\varepsilon = 50$, $t_p = 6$, $V_t = 130$ Left: original image; Right: edge map

(a) Lenna

(b) Boat

(b) Goldhill

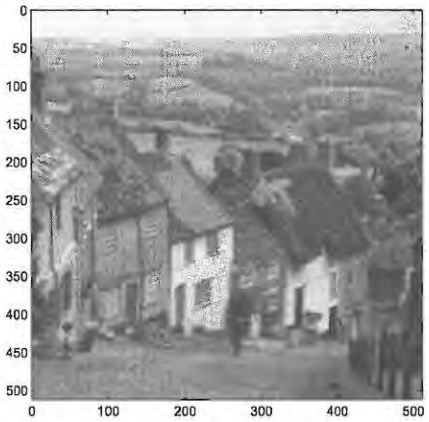
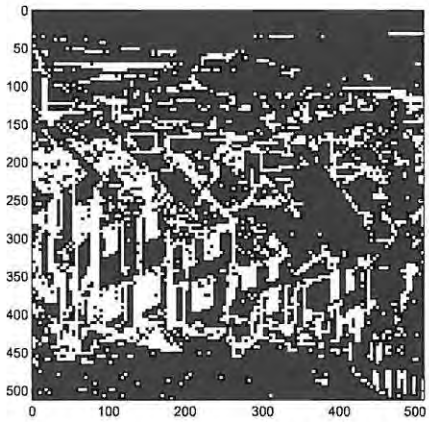
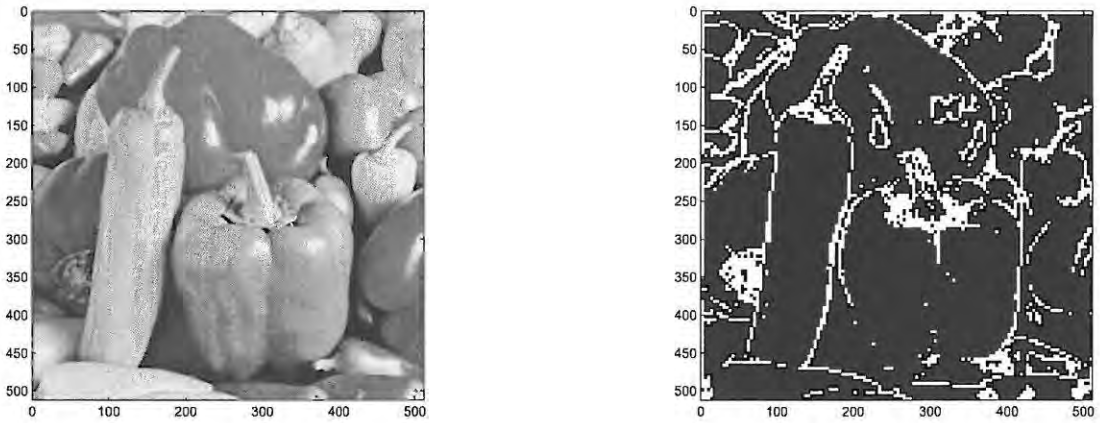
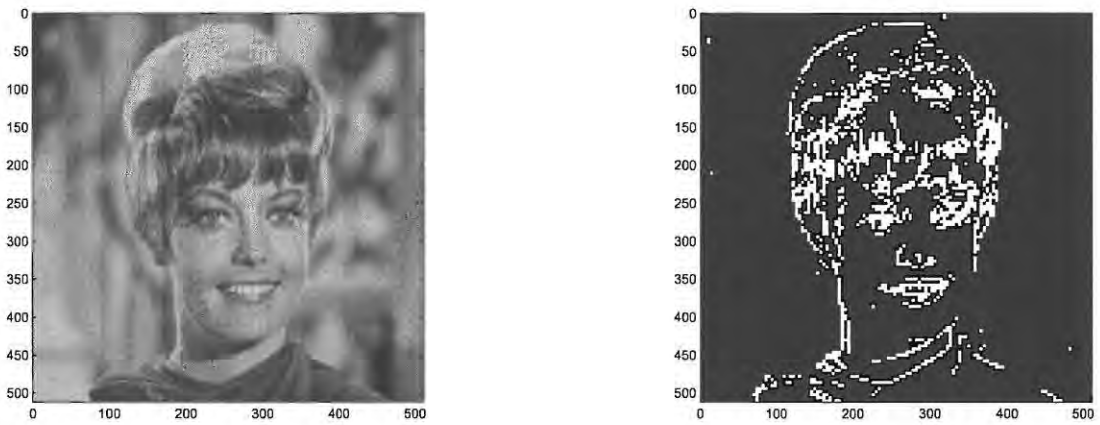



Figure 4.14: Edge maps $t_\varepsilon = 50$, $t_p = 6$, $V_t = 130$ Left: original image; Right: edge map
(a) Peppers



(b) Zelda



4.2 Learning Vector Quantization, C-means and Fuzzy c-means experiments

The first set of experiments was done using learning vector quantization (LVQ), c-means (CM) and fuzzy c-means (FCM) without any modifications as described in Chapter 2 for codebook design. For the experiments using LVQ and CM, the initial codebook is randomly chosen from the training set. For CM, the update for the code vectors is given by

$$c_i = \frac{1}{|U_i|} \sum_{x_k \in U_i} x_k \tag{4.3}$$

If there are no vectors that belong to U_i then $|U_i| = 0$ which gives an undefined result because we divide by zero. In order to reduce the chances of this happening, initialisation is done using vectors from the training set. When using LVQ a code vector is only updated if it is the closest code vector to the training vector that has been presented. This however means that some code vectors may not be used but this is left undetected. If the codebook is taken from the data set then it is less likely that there are no other data points that are close to it than in the case where the code vectors are chosen outside of the data set. For FCM the initial codebook is randomly chosen. In FCM the partition is given by

$$U_{ik} = \sum_{j=1}^{c} \left(\left(\frac{||x_k - c_i||}{||x_k - c_j||} \right)^{\frac{2}{m-1}} \right)^{-1} \tag{4.4}$$

If $x_k = c_j$ then the denominator becomes zero and we get an undefined result. To avoid this we select the initial code vectors randomly in $[0, 255]^{16}$. The random numbers that are generated are decimals and since the entire set of pixel values are integers, our code vectors are not in the training set. All training sessions are 70 iterations. The LVQ experiments are done for initial learning rate $\eta_i = 0.75$ being decreased to $\eta_f = 0.001$ the final learning rate. The fuzzifier m for FCM is set at 1.1. The algorithms are used to generate codebooks of size 256, 512 and 1024.

The initial conditions for the experiments are summarised in the following:

For LVQ:

- Initial codebook randomly chosen from the training vectors.
- Maximum number of iterations $t = 70$
- Initial learning rate $\eta_i = 0.75$

- Final learning rate $\eta_f = 0.001$
- Initial codebook size $c = 256$, $c = 512$, $c = 1024$

For CM:

- Initial codebook randomly chosen from the training vectors
- Maximum number of iterations $t = 70$
- Initial codebook size $c = 256$, $c = 512$, $c = 1024$

For FCM

- Initial codebook randomly chosen in $[0, 255]^{16}$
- Maximum number of iterations $t = 70$
- Fuzzifier $m = 1.1$
- Initial codebook size $c = 256$, $c = 512$, $c = 1024$

MSE Estimates for Edge classes

The MSE estimates obtained for each edge class and the reconstructed image are listed in table 4.3 for the Lenna image. The reconstructed image MSE is represented by W. We observe that the MSE estimates for non edge areas are much lower than the MSE over the edge classes. For instance, using LVQ with 256 code vectors, over the edge classes the MSE is in the range $202 \leq MSE \leq 322$, yet the MSE for the non edge class is 23 (refer to table 4.3 and the appendix). We see this trend for all codebook sizes and for all three algorithms. Figure 4.15(a) shows the reconstructed image for a codebook size of 512 using LVQ. A display of the difference image is given in figure 4.15(b). This difference image is constructed by taking the absolute difference between the original and the reconstructed images:

$$diff(r, s) = |image(r, s) - reconimage(r, s)| \tag{4.5}$$

where $image(r, s)$ is pixel value in the original image at pixel location (r, s) and $reconimage(r, s)$ is the pixel value in the reconstructed image at location (r, s) for all values of (r, s). The

difference image allows us to visualise those areas that are poorly reconstructed. The greater the difference between the original and the quantized image, the darker the pixel. Therefore the gray areas in the difference image, indicate those areas that are poorly reconstructed in the reconstructed image. In the figure 4.15(b) we observe that the areas that are poorly reconstructed include the outline of Lenna's hat, the feathers in her hat, her eyes and her shoulders. These are places where there are abrupt changes in the gray levels in the pixels. This result corresponds to the high MSE's mentioned previously, showing that the edge regions are less accurately coded than the rest of the image. A closer look at a part the quantized image further illustrates this, as is shown in figure 4.16. We observe in this image that the detail of Lenna's eyes, the feathers in her hat, and lips are not very clear and that the outline of the hat has a jagged edge.

The MSE of the whole image is related to the MSE of the edges classes in the following way

$$\begin{aligned} MSE &= \frac{1}{N}\sum_{\varepsilon=1}^{9}\sum_{x_k\in E_\varepsilon}(x_k-\phi(x_k))^2 \\ &= \frac{1}{N}\sum_{\varepsilon=1}^{9}N_\varepsilon.\frac{1}{N_\varepsilon}\sum_{x_k\in E_\varepsilon}(x_k-\phi(x_k))^2 \\ &= \frac{1}{N}\sum_{\varepsilon=1}^{9}N_\varepsilon MSE_\varepsilon \end{aligned} \tag{4.6}$$

where N is total number of test vectors, N_ε is the total number of vectors in class ε and MSE_ε is the MSE estimate in edge class ε. We also take note here that our edge classes are small and that a high value for MSE_ε can compensate for a small value of N_ε.

The classes with the highest MSE for the Lenna image are the north west (E3) and south west (E7) classes for all three algorithms. So we can see that there is a similar trend on the effect on the edge classes in the algorithms. Referring to table 4.3 we see however that the MSE values are not the same. We then looked at the detail in the images to see if there was a significant difference between the visual qualities of images reconstructed using the different algorithms. Figure 4.17 displays Zelda's face for LVQ, CM and FCM. We observe that the effect on the edges is the same as there are no major differences in the reconstructed images for 512 code vectors. Zelda image has the lowest MSE's for all edge classes and overall. This could be attributed to that the Zelda image contains only 11% of edge blocks compared to the other images that have proportions that range between 17% and 29%.

Table 4.3: MSE estimates for Lenna image

(a) LVQ

Number of code vectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	206	206	322	222	270	202	319	281	23	69
512	191	187	298	200	241	182	278	257	21	62
1024	163	157	261	170	196	163	231	221	18	53

(b) CM

Number of code vectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	219	210	334	235	279	227	343	300	23	72
512	200	199	321	211	267	198	309	273	20	66
1024	174	164	290	190	213	177	274	235	17	56

(c) FCM

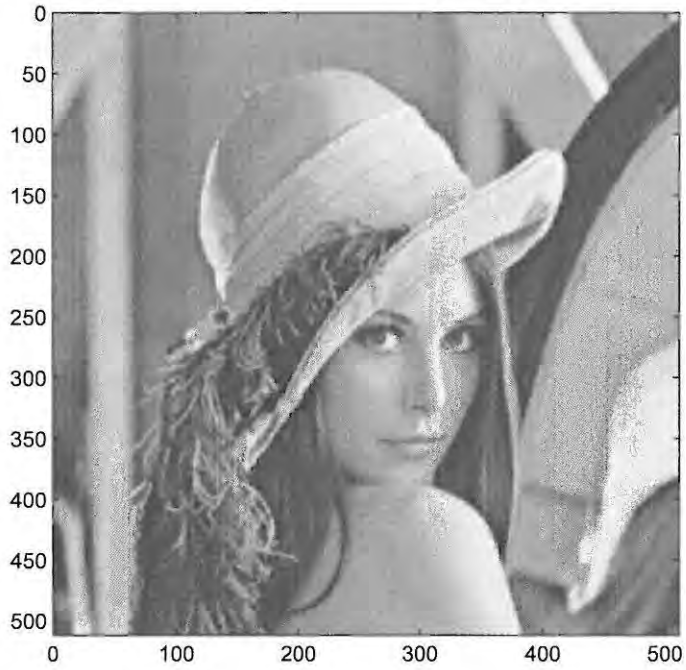
Number of code vectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	186	182	285	224	185	163	301	234	20	59
512	166	167	242	191	161	139	252	196	17	50
1024	145	149	214	163	137	120	207	170	15	44

Table 4.4: MSE estimates using LVQ with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	206	206	322	222	270	202	319	281	23	69
Boat	198	192	256	271	401	326	251	296	24	99
Goldhill	170	173	195	221	170	189	172	172	49	85
Peppers	269	531	209	267	280	390	319	248	26	76
Zelda	89	95	120	122	118	126	138	123	22	32
Mean MSE	186	239	220	221	248	247	240	224	29	72

Figure 4.15: Reconstructed Lenna image $c = 512$ using LVQ

(a) Reconstructed image



(b) Difference Image

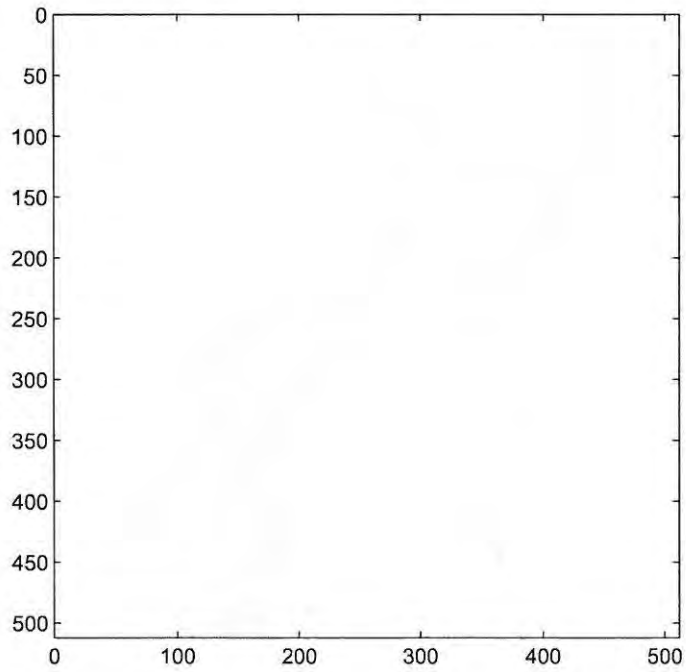
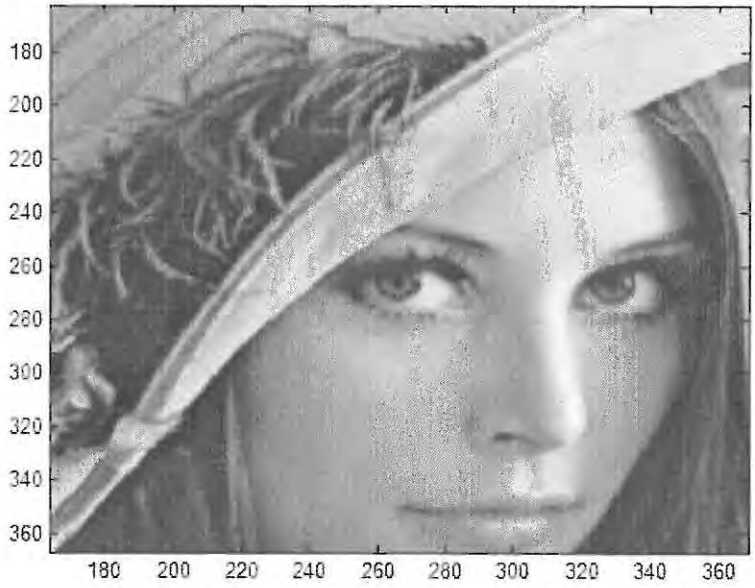


Figure 4.16: Detail of Lenna's face. Top: Original image Bottom: Image reconstructed with 512 code vectors and 4×4 blocks using LVQ

Original image



Reconstructed image

The compression ratio between the original and the reconstructed image is 16 : 1.

Figure 4.17: Zelda image detail for (a) LVQ (b) CM (c) FCM

(a) LVQ $c = 256$

(b) CM $c = 256$

Figure 4.18: Zelda image detail (continued)

FCM $c = 256$

Table 4.5: Time taken using LVQ, CM and FCM with 256 code vectors

Algorithm	Time taken
LVQ	13 minutes
CM	1 hour 22 minutes
FCM	4 hours and 39 minutes

The cross validation estimate of the MSE's is the mean of the MSE estimates for the five images. We see here that we violate the assumption that

$$MSE(\phi) \approx MSE(\phi_k) \qquad \text{for} \quad k = 1, 2, ..., K \tag{4.7}$$

as we see from table 4.4 that there is large variation in the MSE's that are obtained. For example for edge class E2 in table 4.4 we observe that the lowest MSE is 95 for Zelda and the highest is 531 for the Peppers image. We however can make some deductions from the cross-validation estimate. We see that the LVQ and CM algorithms give very similar results. The FCM seems to have $MSE's$ that are slightly higher than those for the other two algorithms. We still however see the same trends, over the edge classes in relation to codebook sizes and the effect on edge classes as mentioned previously. We also see visually that there is no noticeable difference between the three algorithms.

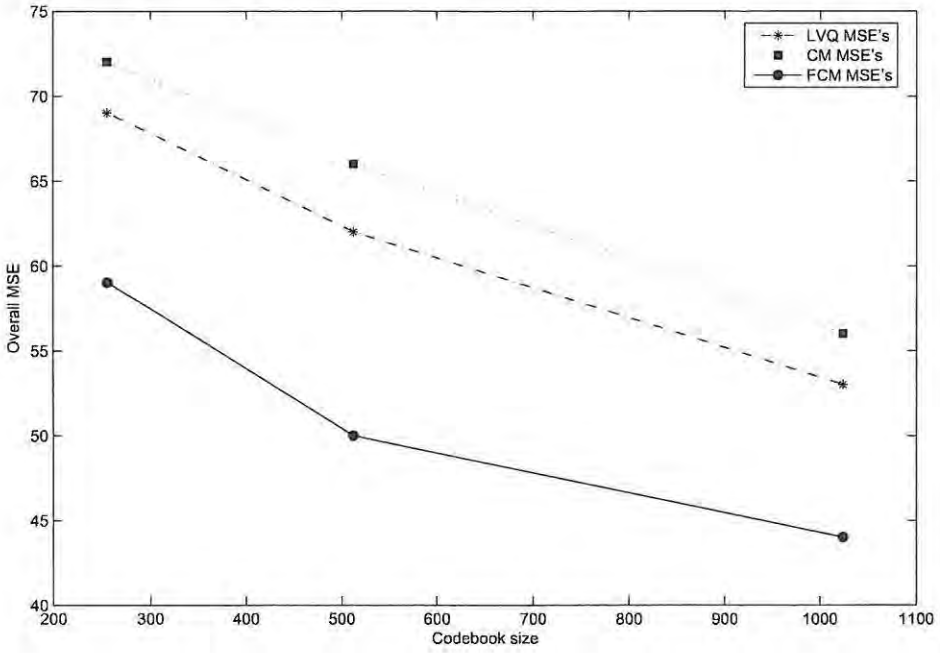
Dependence of MSE on Codebook

For all three algorithms the MSE for the whole image decreases as the size of the codebook is increased as shown in figure 4.19. This we would expect, as a larger codebook means that there are more code vectors available to represent the image vectors. Figure 4.20 shows part of the reconstructed images from the compression of the Goldhill image, for the different codebook sizes. We observe here that the visual quality of the reconstructed image is better for larger codebooks. The windows, doors, door steps and the roof edges are not clearly shown especially for the case with a codebook size of 256. Comparing figure 4.20(b) and 4.20(d) we note that the roof edge and the windows for the house to the left are more visible for the latter image.

Dependence of computation time on algorithms

We recorded the total time taken for clustering, coding and reconstruction of the images, to see if there were any computational advantages in the algorithms. LVQ, a sequential algorithm, is the fastest, taking approximately 13 minutes. The times are displayed in

Figure 4.19: Codebook size and MSE using FCM on the lenna image



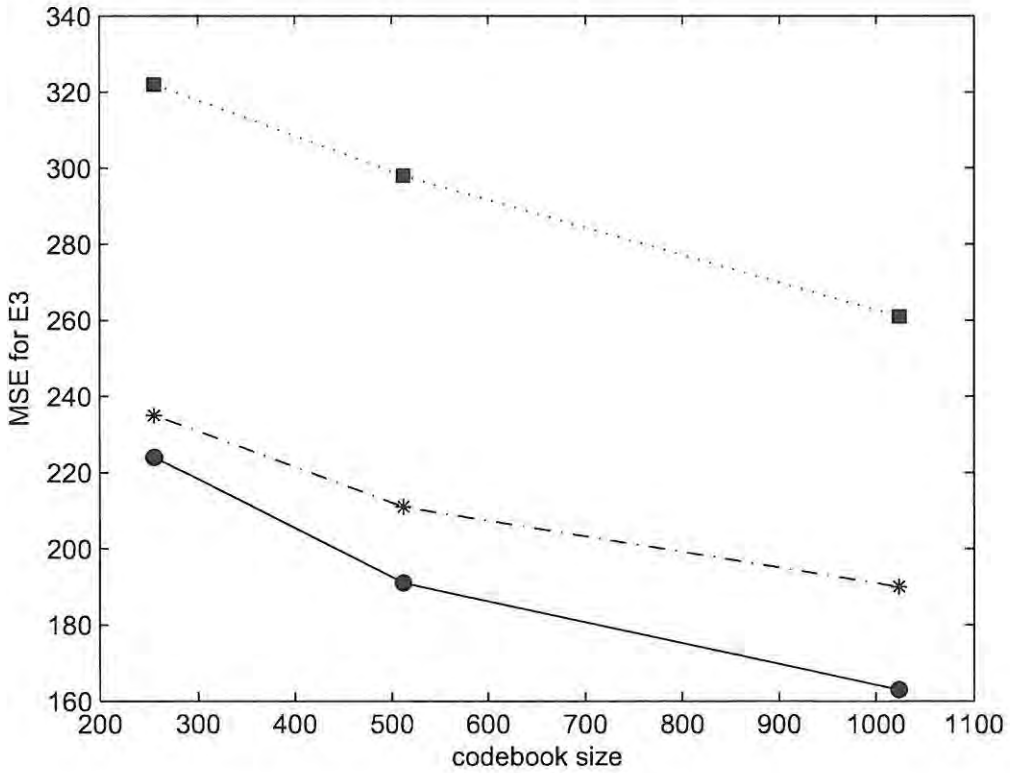


Figure 4.20: Detail of Goldhill, reconstructed image using CM with (a) original (b) 256 (c) 512 (d) 1024 code vectors

(a)

(b)

(c)

(d)

Table 4.6: MSE estimates for portion of Lenna using FCM with 256 code vectors

	m	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
FCM	1.1	68	81	115	63	207	61	425	79	22	37
FCM	1.5	227	387	234	158	263	162	719	221	30	61
FCM	2	216	458	441	256	481	267	1045	228	23	76
FCM	6	314	520	655	377	734	592	1294	324	25	300

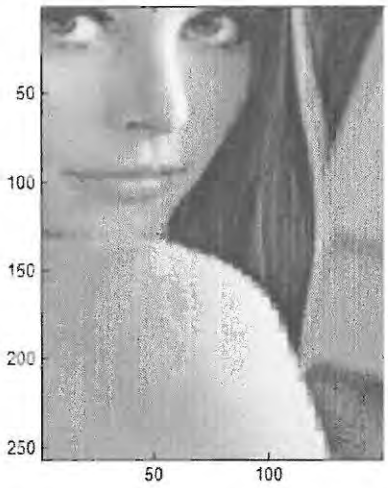
table 4.5. FCM is the slowest algorithm of the three algorithms taking 4 hours and 39 minutes with codebook size of 256. Regardless of which clustering algorithm is used for codebook creation, only 51 seconds of the time is used for reconstruction. We also note that having a smaller codebook is an advantage over a larger codebook as the time for codebook design is reduced. Although the algorithms do not take the same amount of time to execute there is a similarity in that time to carry out VQ increases as the codebook sizes increases.

A distinct parameter that FCM has is the fuzzifier m. We decided to take a look at the effect of the fuzzifier on the performance of FCM in image compression. We used the bottom left quarter of the Lenna image as a training set and the bottom right as a test set. The fuzzifier is set at $m = 1.1$, $m = 1.5$, $m = 2$ and $m = 6$ for the experiments. We used a smaller training set for this illustration to reduce the computational load and thus increase the speed of the experiment. Figure 4.21 shows the reconstructed images that were obtained. The MSE's are also given in table 4.6.

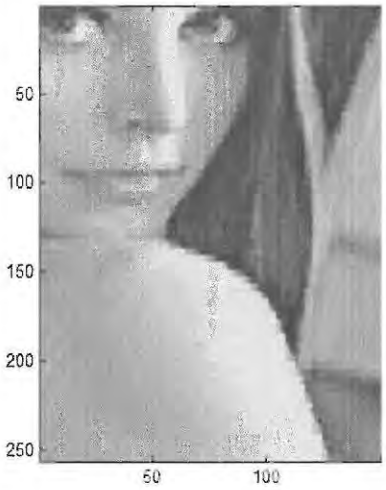
The MSE's for the edge classes increase as m increases, with a difference of up to 869 between $m = 1.1$ and $m = 6$ for the worst edge class as shown in the table 4.6. We display a graph showing the relationship between the fuzzifier and the MSE in figure 4.22 for the east (E6) edge class. When we take a close look at the reconstructed images in figure 4.21, we see that as m increases there are more areas containing edges that are poorly reconstructed. The outline of Lenna's face and her shoulder show the staircase effect described previously. There is less of this effect shown in figure 4.21(a). We also note that Lenna's eyes are blurry in figure 4.21(b). The homogenous areas in figure 4.21(d) also have a poorer visual quality to those in figure 4.21(a). There is not much detail that is observed with $m = 6$ in figure 4.21. We observe then that the fuzzifier does have an effect on the visual quality of the reconstructed image and that $m \approx 1$ seems to give the best reconstruction.

Figure 4.21: Part of Lenna image with variation in m using FCM with $c = 256$ (a) $m = 1.1$ (b) $m = 1.5$ (c) $m = 2$ (d) $m = 6$

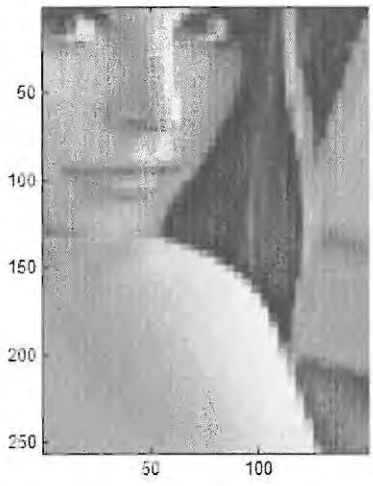
(a)



(b)



(c)



(d)

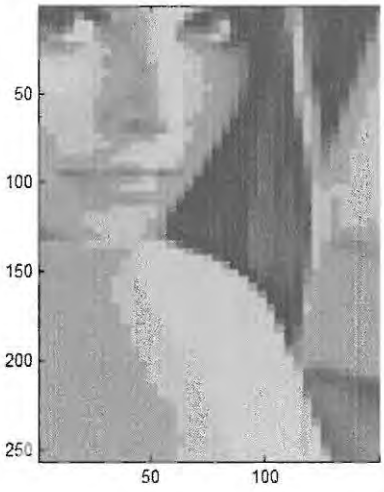
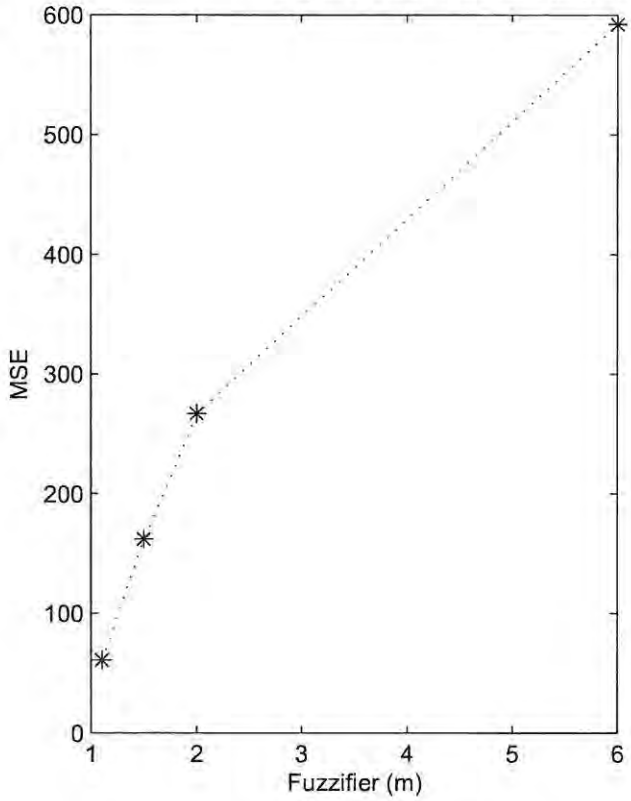


Figure 4.22: Fuzzifier(m) vs MSE



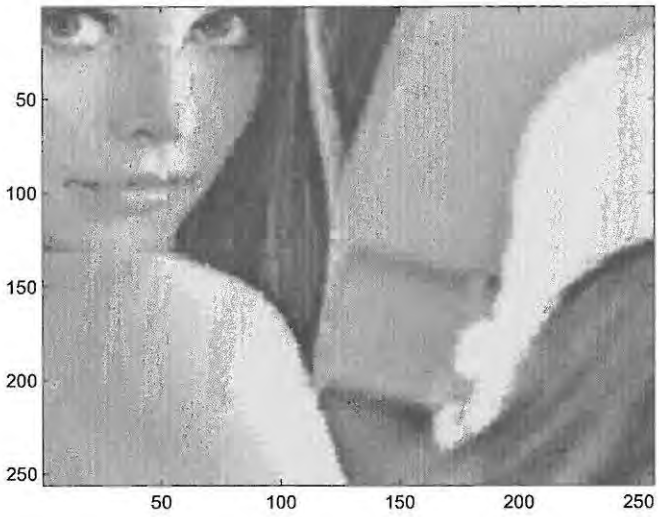
4.3 Classified Vector Quantization (CVQ) Experiments

The first step in the CVQ is to classify the test and training set. Each image is subdivided into blocks that are converted to vectors which are classified using the method described in section 3.2. For the training set, the image vectors are separated into their edge classes. The training set is therefore made up of 9 arrays of vectors, with each array containing the vectors that belong to one edge class. The total number of code vectors used is the same as the previous experiments, that is 256, 512 and 1024. However, since each class has a separate codebook, a decision has to be made first about the number of code vectors that are required for each class.

Prior to carrying out the experiments with the five images we used the Lenna image in an attempt to get an indication of the proportion of code vectors necessary for each class. The lower left part of the Lenna image was used as a training set to generate a codebook. This codebook was tested on the lower right part of the image. The number of code vectors allocated was the same for each edge class. The training set does not necessarily have the same proportion of vectors in each edge class as the test image, thus we keep the proportions in the codebook the same for all edge classes. The proportion of code vectors for the non edge class was varied between 25% and 92%. The results that were obtained are given in table 4.7. The results show that we need a greater proportion of code vectors for the edge regions as there is a noticeable improvement in the MSE's for the edge classes. Although the MSE for the non edge class becomes higher for an allocation of 75% to the edge classes, our particular interest here is to improve the results for edge classes thus we choose the proportion that favours the edge classes. The reconstructed images in figure 4.23 visually show an improvement in the edge rconstruction for the case where 75% code vectors were allocated to the edge classes. Percentages higher than 75% do not show any noticeable improvement. Based on these results, for the main set of experiments, we decided to have 25% of code vectors to be allocated to the non edge class and the remaining 75% of code vectors to be evenly distributed between the 8 edge classes.

As done previously we use all three algorithms with the CVQ approach for three codebook sizes. For convenience we abbreviate the three algorithms to classified vector quantization with learning vector quantization (CVQ-LVQ), classified vector quantization with c-means (CVQ-CM), and classified vector quantization with fuzzy c-means (CVQ-FCM). Experiments that were carried out using CVQ-LVQ and CVQ-CM had initial code vectors that were randomly selected from the training set for each class. For CVQ-FCM the initial code vectors were randomly selected outside of the training set for each edge class.

Figure 4.23: Part of Lenna quantized with 256 code vectors using LVQ
(a) Part of Lenna image quantized using 75% code vectors for the edge classes



(b) Quantized image using 25% code vectors for edge classes

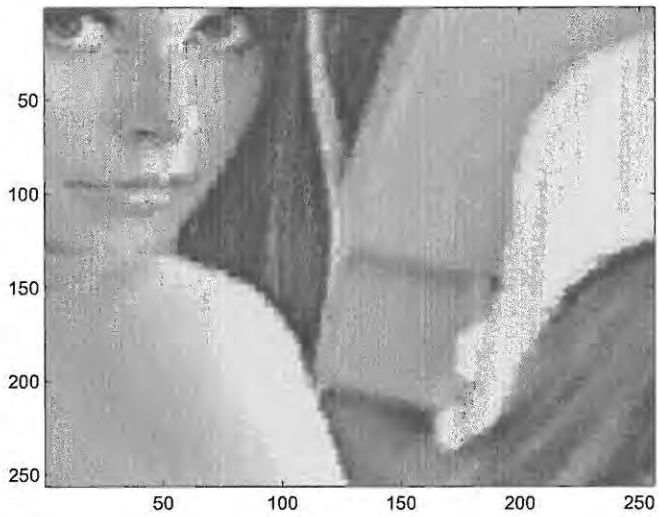


Table 4.7: Proportion of edge code vectors, Lenna, $c = 256$, LVQ

%age edges	E cv	NE cv	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
25%	8	192	260	330	363	210	458	271	732	296	36	80
50%	16	128	255	318	322	183	357	245	557	315	37	74
75%	24	64	239	331	278	179	324	208	551	239	45	77
81.25%	26	48	239	331	268	185	323	201	463	239	48	78
87.5%	28	32	239	331	266	185	327	228	463	239	46	78
93.75%	30	16	233	320	268	184	322	201	463	279	62	90

Table 4.8: Codebook sizes

Edge classes	Non Edge class	Total
24	64	256
48	128	512
96	256	1024

Initial conditions are the same as those given in section 4.2 except that these conditions are for each codebook created for each edge class. The total codebook size is 256, 512 and 1024 as we had before, but now this is distributed into the edge classes, with 75% distributed evenly amongst the edge classes and 25% in the non edge class. Table 4.8 shows the number of code vectors in each edge class, the non edge class and the total number of codevectors used in the CVQ experiments. All training sessions are for 70 iterations.

MSE estimates for edge classes

Once the codebook for each edge class has been generated and the test image vectors quantized, the MSE's for each edge class are estimated. The quantized test image vectors are combined to make up the reconstructed image. The MSE estimates obtained for the boat image using CVQ-LVQ, CVQ-CM and CVQ-FCM are shown in table 4.9. The results using LVQ, CM, FCM described in section 4.1 are also given for comparison. Using LVQ we observe lower MSE estimates for the edge classes: west (E5) and north east (E8). We observe for all 3 codebook sizes, for all the edge classes that the MSE's are higher for the method utilising CVQ-CM and CVQ-FCM than for CM and FCM respectively. We also observe that there are improvements only for some edge classes with the other images using LVQ (Refer to the appendix for results). We then observe the reconstructed images to see if there is a noticeable difference in the visual quality of the images. The images using CVQ-LVQ are shown in figure 4.24 and 4.25. The

original and the reconstructed image using a codebook generated with LVQ are also given for comparison. Visually there is no noticeable difference between the images obtained using LVQ and CVQ-LVQ . We then decided to increase the percentage allocated to the edge classes from 75% to 93% to see if perhaps we would see an improvement. The reconstructed image is shown in figure 4.25(b). Again we observe no difference in the visual quality of the images.

Ramamurthi and Gersho [17] used double the number of code vectors for the diagonal classes than the other classes. We then tried this distribution of codevectors instead of having equal vectors using the LVQ algorithm with $c = 512$. The distribution of the code vectors is given in table 4.10. E3, E4, E7 and E8 are the diagonal classes so they have twice as many code vectors as the E1, E2, E5 and E6. The non edge class as was the case previously has 25% of the total codebook. Table 4.10 gives the MSE estimates LVQ, CVQ- LVQ with 75% of the codevectors as edge vectors and CVQ-LVQ2 with double the codebook size for the diagonal classes. The MSE's are higher for 6 out of the 8 edge classes indicating greater edge degredation. The classes that show improvement are the north west (E3) and south west (E8) class. The boat image that is reconstructed with a codebook that has double the codebook size for the diagonal clases is shown in figure 4.26. Again we observe that there is no major difference between the images that are reconstructed with a codebook using LVQ especially in the diagonal structures.

We take a look at the cross validation estimates of the MSE's shown in table 4.11. The estimates are derived from the mean of the MSE estimates for the five images. The MSE estimates using CVQ are higher than those that are obtained without using CVQ for all three algorithms. We deduce from the cross validation estimate of the MSE that the MSE's increase when using CVQ and thus generally we canot say that there is any improvement in using CVQ. We have however shown that in the visual quality of the images there is no evidence of any major diffrences between the cases without CVQ and those that employ CVQ. The CVQ algorithm is however much faster than the method without edge separation. The time taken to create a codebook and reconstruct images using CVQ is given for the three algorithms in table 4.12. Although we do not see an improvement in the edges for most cases we observe that there is an advantage in using CVQ when we consider the time that it takes to create the codebook. As we have alluded to in Chapter 3, this is as a result of the fact that the algorithm does not perform clustering with the whole codebook but through subcodebooks which allows faster computation. CVQ thus allows greater efficiency in the use of the clustering algorithms.

Table 4.9: MSE estimates for the boat image using CVQ for codebook design

LVQ

Number of codevectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	198	192	256	271	401	326	251	296	23.9	99
512	179	171	226	250	362	305	223	267	22.1	90
1024	153	143	203	215	317	277	192	235	18.5	78

CVQ-LVQ

Number of codevectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	230	232	276	320	397	338	287	278	24.1	105
512	197	196	236	279	344	284	243	240	20.9	90
1024	174	170	202	242	273	250	222	212	18.2	77

CM

Number of codevectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	207	198	240	267	331	263	248	244	21.5	90
512	182	174	204	238	275	232	224	213	19	78
1024	151	151	177	208	243	216	193	187	16.2	68

CVQ-CM

Number of codevectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	222	226	259	326	384	349	280	272	23	103
512	197	192	228	267	317	284	240	236	19	87
1024	175	161	200	242	273	233	216	212	17.2	75

FCM

Number of codevectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	207	198	237	277	310	269	238	256	19.7	88
512	179	164	205	235	270	241	214	212	17.1	76
1024	153	144	177	199	234	212	188	189	15.6	66

CVQ-FCM

Number of codevectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	216	210	267	321	404	338	274	269	23.6	103
512	187	175	224	274	317	269	238	232	20	85
1024	165	154	196	223	259	226	205	203	17.3	73

Table 4.10: MSE estimates using CVQ-LVQ2

(a) Class sizes for CVQ-LVQ2

Edge class	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
No. of code vectors	32	32	64	64	32	32	64	64	128	512

(b) MSE Estimates

Edge class	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
LVQ	179	171	226	250	362	305	223	267	22.1	90
CVQ-LVQ	197	196	236	279	344	284	243	240	20.9	90
CVQ-LVQ2	213	211	221	266	376	316	232	223	20.9	93

Figure 4.24: Quantized boat detail image with 512 code vectors using LVQ
(a) Original image

(b) Using LVQ without edge separation

Figure 4.25: Quantized boat detail image with 512 code vectors using LVQ (cont.)
(a) 75% code vectors edges

(b) 93.75% code vectors edges

Figure 4.26: Reconstructed boat image using CVQ-LVQ2

Table 4.11: Cross validation estimates of the MSE

Without CVQ , $c = 512$

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
LVQ	164	210	197	196	218	221	207	200	25	64
CM	167	219	199	194	214	211	218	189	23	62
FCM	173	250	211	217	245	228	240	186	20	62

With CVQ, $c = 512$

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
LVQ	188	240	212	217	232	240	219	193	24	66
CM	185	233	205	210	221	234	215	186	24	64
FCM	183	229	205	215	224	226	212	186	23	64

Table 4.12: Time taken for codebook creation and reconstruction $c = 256$

Algorithm	Time taken	Algorithm	Time Taken
LVQ	13 minutes	CVQ-LVQ	9 minutes
CM	1 hour 22 minutes	CVQ-CM	26 minutes
FCM	4 hours and 39 minutes	CVQ-FCM	38 minutes

Table 4.13: MSE estimates using FCM on part of Lenna image

	m	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
FCM	1.1	68	81	115	63	207	61	425	79	22	37
FCM	2	216	458	441	256	481	267	1045	228	23	76
CVQ-FCM	1.1	245	287	259	177	308	200	491	217	32	63
CVQ_FCM	2	352	299	253	159	480	235	1154	463	36.7	83

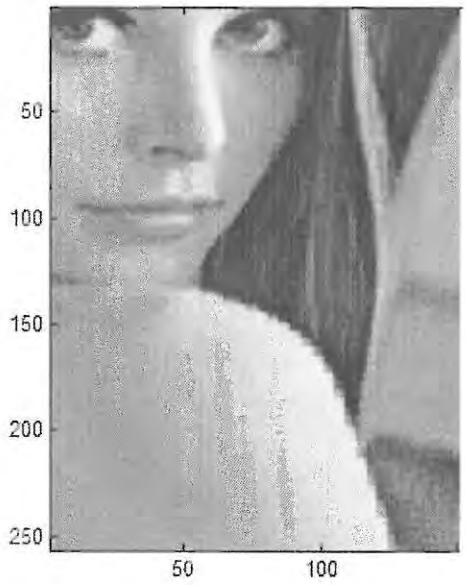
FCM: CVQ performance dependence on fuzzifier

We once again examine the effect that the fuzzifier has on the effectiveness of FCM when using CVQ-FCM. The lower left quarter of the Lenna image was used as a training set to generate a codebook using FCM. The lower right quarter of the Lenna image was used as a test image as was done previously in section 4.1. We use the FCM algorithm with $m = 1.1$ and $m = 2$, then we compare these results with those obtained using CVQ -FCM with FCM. When CVQ is performed for the case $m = 1.1$ less detail is observed in the eyes as shown in figure 4.27(c) than in FCM shown in figure 4.27(a). The outline of Lenna's face and shoulder show more of the staircase effect as well. On the other hand, for $m = 2$ we see a difference in the reconstruction, as the finer details in the eye are become more visible. There is also better reconstruction of the outline of Lenna's face. This however is masked by the poor reconstruction of the homogenous areas. The CVQ-FCM approach shows no significant visible improvement on reconstruction when compared to FCM.

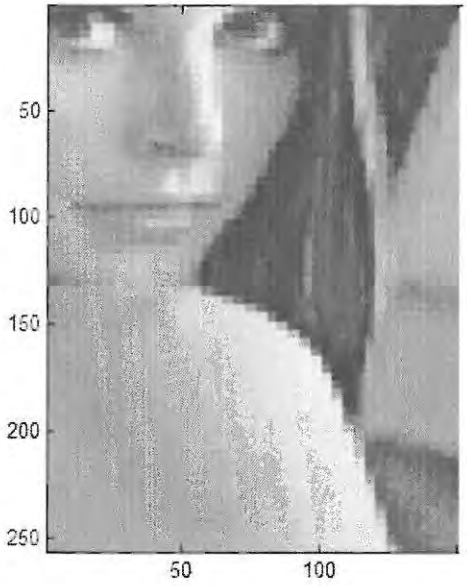
MSE estimates using random selection

Thus far we have looked at the cross validation estimates for the MSE. As mentioned in section 4.2 the assumption in equation (4.7) is violated here. In order to ascertain whether there was an effect on the MSE estimates due to using the cross validation approach, we carried out some experiments by taking a random selection of vectors from our five images. The procedure that is done is described previously in Section 1.7.2. We create a training set by randomly selecting 16384 image vectors from the five images. The test set is also created in a similar fashion, but is chosen such that all the vectors are independent of those in the training set. We choose to use 16384 image vectors as this is the number of image vectors that is in an image that is 512×512 pixels in size. We carried out 20 experiments using LVQ to ascertain if there was variation in the number of vectors that belong in each edge class and if there was an effect on the MSE. We give the results for one of the experiments. The number of image vectors in each edge class

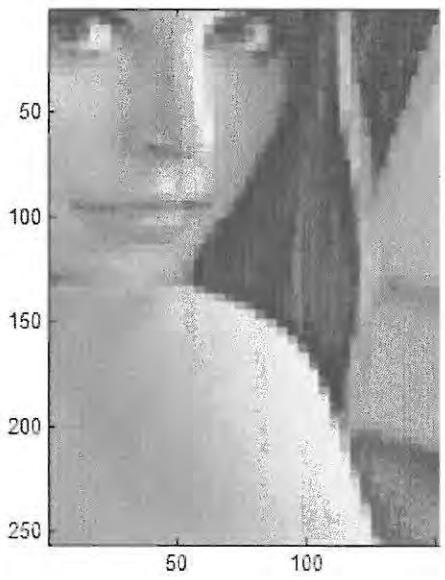
Figure 4.27: Part of Lenna image for $m = 1.1$, $m = 2$ with and without edge separation using FCM. (a) $m = 1.1$ with FCM (b) $m = 2$ with FCM (c) $m = 1.1$ with CVQ-FCM (d) $m = 2$ with CVQ-FCM



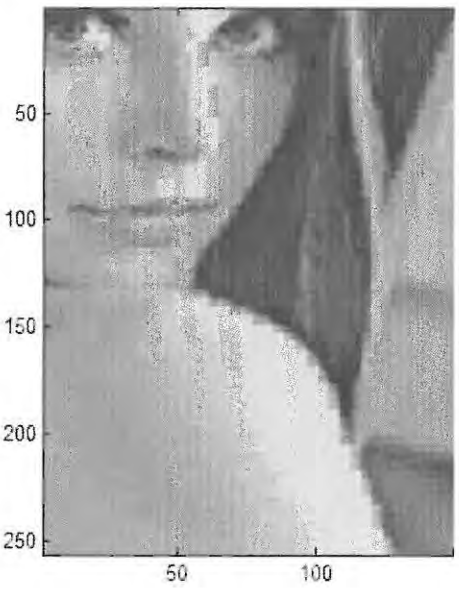
(a)



(b)



(c)



(d)

Table 4.14: Edge classes and MSE estimates using random selection

(a) Edge class sizes using random selection

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Training set	452	444	460	408	598	534	265	249	12974	16384
Test set	421	474	448	418	595	561	287	265	12915	16384

(b) MSE Estimates with $c = 256$

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
LVQ	179	221	219	222	238	242	222	214	25	66
CM	180	176	206	214	208	207	237	195	22	60
FCM	189	177	214	220	205	213	232	203	23	60

(c) CVQ MSE Estimates with $c = 256$

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
CVQ-LVQ	215	205	254	271	256	261	260	219	25	71
CVQ-CM	215	217	249	260	250	265	272	227	26	71
CVQ-FCM	433	481	512	472	436	443	444	403	34	122

for a random selection of image vectors for the training and test set is given in table 4.14. The corresponding MSE estimates are also given in table 4.14 for the cases with and without CVQ.

First we see that the edge class sizes in the training set and test set are similar for all classes, with south west (E7) and north east (E8) having the smallest class size and west (E5) and east (E6) with the largest class sizes. This corresponds to the edge statistics that are in table 4.2. We observe that for CM and FCM there are similar MSE's for all of the classes. LVQ generally has higher estimates than the other other two algorithms. When comparing the CVQ MSE's to those without CVQ, again we get higher MSE's for all algorithms and over all the edge classes. This is similar to the results that we obtain using the cross-validation approach. With this method however we cannot visualise the images to observe the effect on the visual quality because the image vectors are randomly selected from all five images. We are, however, able to see that the higher MSE's that we get using CVQ are not a result of using a cross-validation approach even though the main assumption of cross-validation is violated.

CVQ with 1×2 blocks

Further experiments were done to understand the reason for the increase in the MSE's when CVQ is employed. The 16 dimensional vectors that we use are difficult to visualise so for the purposes of viewing the data we used the lower left quarter of Lenna for training and lower right quarter for testing, now subdivided into 1×2 blocks. For this block size

only 2 edge classes are possible: where the gray value moves from high to low and vice versa. We used a similar edge detector for our data with the edge classes limited to only 2 classes. The parameters used to classify edge blocks were as follows: $t_p = 1$, $t = 50$ and $V_t = 15$. We plotted each edge class separately. These plots are shown in figure 4.28.

The first edge class is represented by those points that are predominantly below the diagonal as shown in figure 4.28(a). The second edge class consists of the points that are mostly above the diagonal. The non edge class contains all the homogenous blocks that lie along the diagonal as shown in figure 4.29. We note here that there seems to be an overlap in the edge classes 1 and 2. This may occur if both pixels are classified as edge pixels with different classes for each pixel. The edge class is always chosen to be the edge class that the majority of edge pixels in the block. In the case where there are two classes with an equal number of edge pixels, the class is chosen as the class labeled first. In our example, class 1 will be chosen as the edge class if there is a block that contains a class 1 edge pixel and class 2 edge pixel.

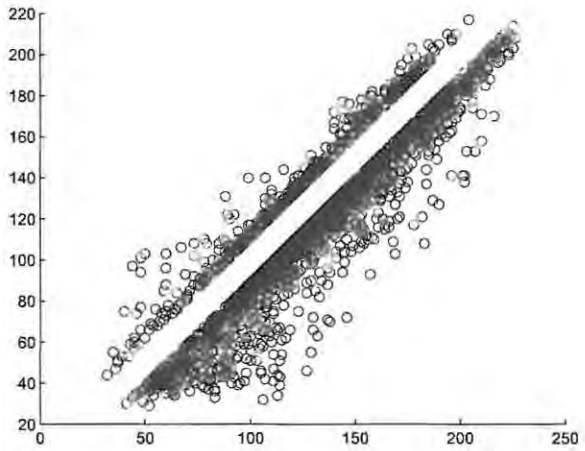
We then used LVQ and plotted the distribution of the code vectors using Voronoi diagrams to show the partition of the data. Figure 4.30 shows the code vectors for the cases with and without edge separation. For the case without edge separation we see that most of the code vectors are concentrated in the region with the non edge class. Once separation of the classes is performed we observe in figure 4.30(b) that the code vectors are moved away from the diagonal. Taking a closer look at the plot as shown in figure 4.31, we observe that there are code vectors that are very close to each other especially above the diagonal. We suspect this occurs as a result of the overlap that occurs in the class 1 and class 2. If this idea is extended to the 16 dimensional case that we have worked with here, there could be many instances that there is an overlap in the classes and thus we reduce our efficiency in the use of the code vectors as we have duplicate code vectors in some instances. When code vectors are too close to each other, effectively the codebook size is reduced as the code vectors code for the same image vectors. The resulting codebook has redundant code vectors.

4.4 Experiments using the replication method (RM)

The procedure in using the replication method is very similar to that used in section 4.2 except that the training sets are modified before being used in generating the codebooks. Before any of the codebooks are generated, for each block x_k in the training set the replication factor $N_d(x_k)$ given in Chapter 3 is calculated. As the training set is large,

Figure 4.28: Distribution of edge classes with 1×2 blocks

(a) Edge class 1



(b) Edge class 2

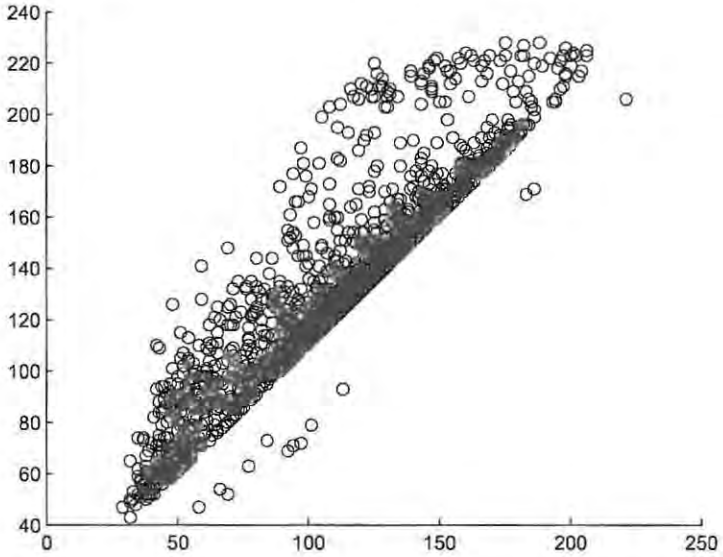


Figure 4.29: Non edge class distribution for 1×2 blocks

(c) Non edge class

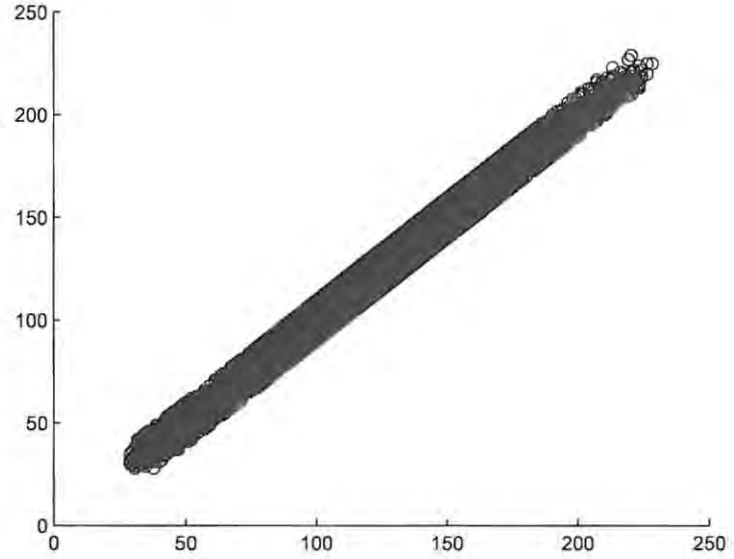


Table 4.15: Statistics of Nd

Test image	Number of training vectors	Maximum N_d
Lenna	142142	14
Boat	159990	15
Goldhill	120965	17
Peppers	136437	10
Zelda	161419	19

65536 vectors in each case, the computation takes a long time. These factors are set for all the experiments. We then replicate each block x_k in the training set using $N_d(x_k)$.

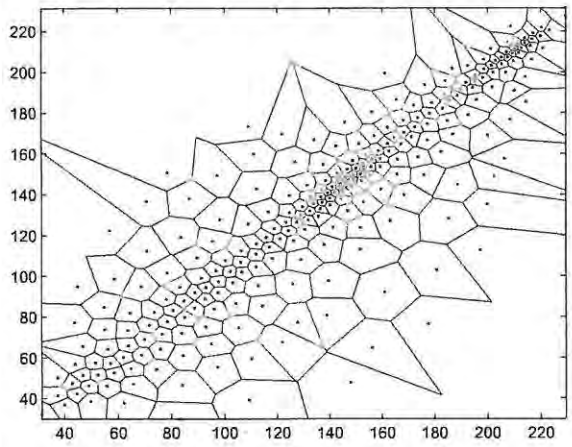
The maximum value of N_d and the total number of vectors in the training set for the 5 images is given in table 4.15.

MSE Estimates for edge classes

The three algortithms LVQ, CM and FCM are then employed with similar initialisation as in section 4.2 and all other parameters specific to each algorithm, remaining as stated previously. The results using LVQ with replication are similar to the results obtained without replication, with some classes being slightly higher for the former. Generally for

Figure 4.30: Distribution of code vectors

(a) LVQ with 256 code vectors



(b) LVQ with CVQ approach 256 code vectors

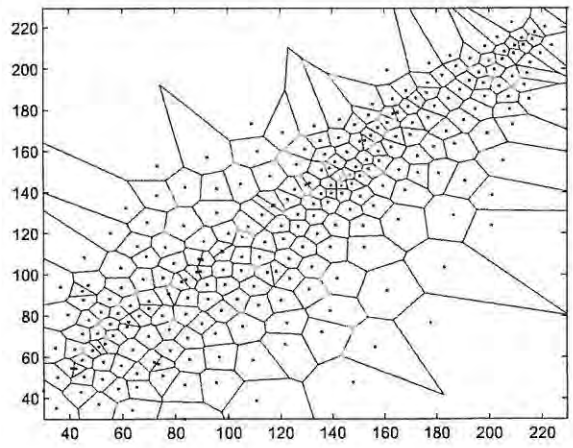
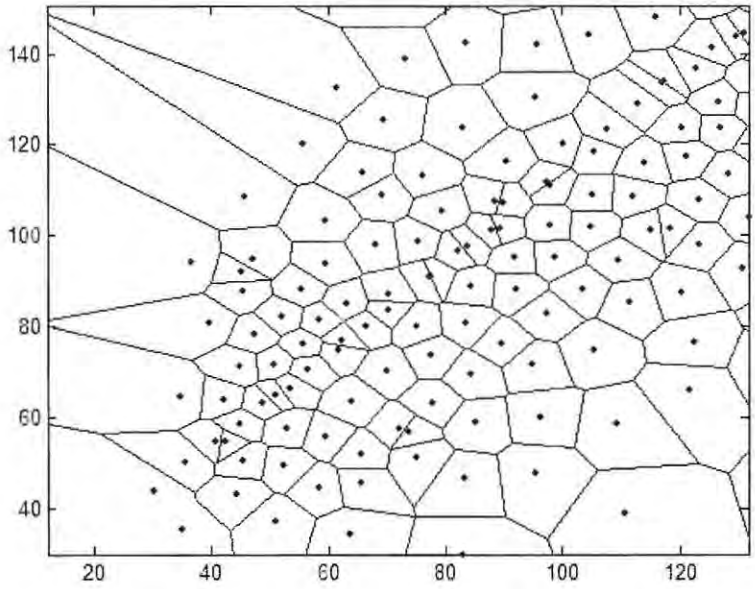


Figure 4.31: CVQ approach
(c) A closer look at the distribution of the code vectors using the CVQ approach



all the images there is no visible difference between the case with LVQ and using RM-LVQ. Tables of the results are in the appendix. Replication using CM also only shows slight improvements for the Peppers and Zelda image particularly for a larger codebook. The results for the Peppers image are shown table 4.16. When using FCM the results are similar only for the Lenna and Boat image with and without replication. The Goldhill has MSE's which are significantly higher for the replication method, therefore we have a poorer reconstruction for it compared to the method without replication. We however noted that similar to the case with CM, there was an improvement for the Peppers and Zelda images. The results using FCM that were attained for the Peppers image are shown in table 4.16.

Figure 4.32 and figure 4.33 show the peppers images reconstructed after using FCM with the replication of the training set. We observe the staircase effect here in the image using FCM without replication as mentioned before. When replication was applied we see that the areas showing the boundaries of the peppers are clearer and as indicated by the lower MSE's there is a noticeable improvement in the reconstruction.

In examining the mean cross-validation estimates given in table 4.17 we observe that the MSE's for RM-LVQ are slightly higher than those for LVQ for all edge classes. The MSE

Table 4.16: MSE estimates for Peppers image with 512 code vectors using RM-CM and RM-FCM

(a) CM

Number of code vectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	254	559	212	254	303	393	324	240	23	74
512	228	478	190	227	259	355	281	209	21	66
1024	203	461	169	201	254	338	249	190	18	60

(b) RM-CM

Number of code vectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	259	524	214	259	270	384	312	251	25	75
512	223	479	188	223	239	354	277	267	22	66
1024	203	427	167	197	213	333	236	185	20	59

(c) FCM

Number of code vectors	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
256	328	703	351	368	554	490	486	282	22	96
512	283	655	316	345	487	460	435	254	20	88
1024	264	624	285	314	441	420	406	228	19	81

(d) RM-FCM

Number of code vectors	E1	E2	E3	E4	E5	E6	E7	E8	E	W
256	258	497	215	253	267	376	307	246	26	74
512	225	459	187	222	224	345	273	209	22	66
1024	201	398	163	189	193	303	226	184	20	57

Table 4.17: Mean MSE estimates using cross-validation with $c = 512$

(a) Without RM

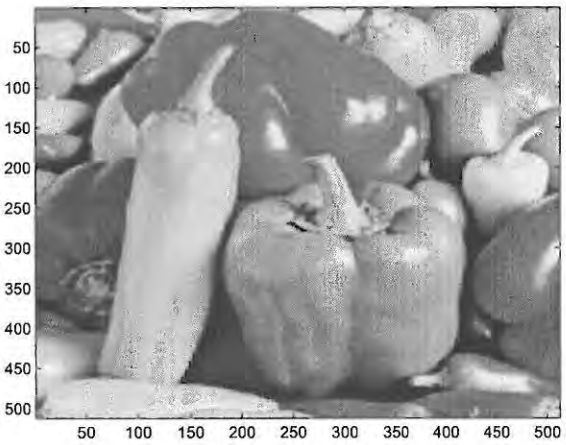
	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
LVQ	164	211	197	196	218	221	207	200	25	64
CM	167	219	199	194	214	211	218	189	23	62
FCM	173	250	211	217	245	228	240	186	20	62

(b) With RM

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
LVQ	172	245	209	211	221	229	223	207	29	69
CM	165	216	187	198	209	216	205	195	23	61
FCM	170	212	187	199	194	214	205	189	23	61

Figure 4.32: Quantized Peppers image (a) FCM (b) RM-FCM

(a) Peppers with FCM 512 code vectors



(b) Peppers RM-FCM 512 code vectors

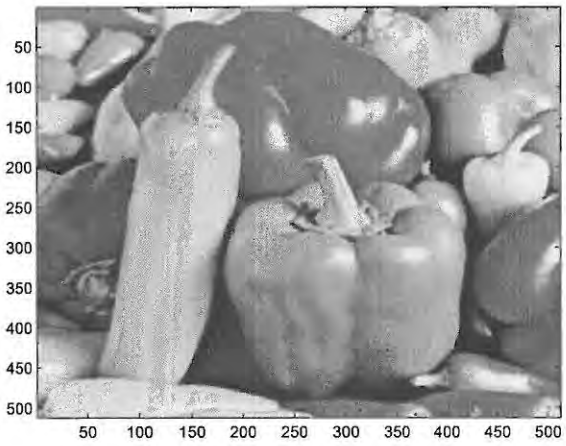
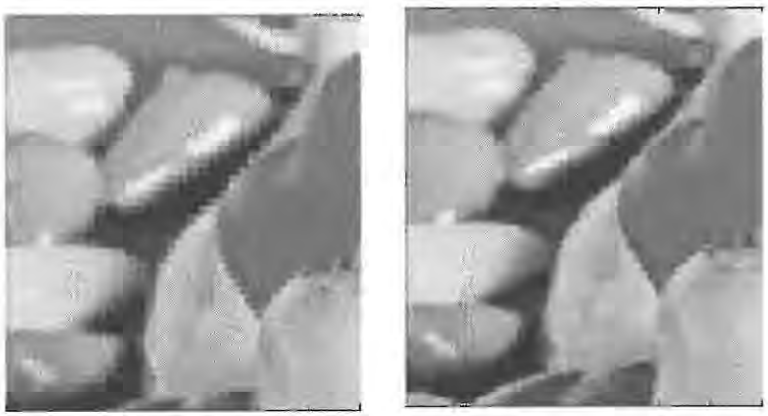


Figure 4.33: Detail of Peppers using RM-FCM with 512 code vectors. Left: without replication; Right: with replication



estimates using RM-CM are lower for edge class north west (E3), west (E5) and south west (E7). The MSE estimates using RM-FCM are lower than using FCM. We deduce from these estimates that FCM seems to show a slight improvement when using RM.

In order to understand the results we looked again at the statistics of the edge classes for each test set, comparing the test image statistics with those of the corresponding training set. Referring to table 4.2 we observe that Goldhill and the Boat have larger proportions of vectors in their edge classes than their training sets. Consequently when replication is done there are inadequate vectors that are supplied to those areas and thus there is no improvement using any of the algorithms. The Peppers and Zelda image however have smaller proportions of vectors in their edge classes than they do in their training sets, therefore there are more code vectors that are used for the edge areas. We can conclude then that the training set must have a very similar edge distribution to the test set for the replication method to be successful. In the cross-validation approach, we see that there are only lower MSE's for the images with RM that have training sets that contain a greater proportion of edge vectors in their edge classes. This could then indicate that it is necessary to have a training set that has a greater proportion of edge vectors in the edge classes than the test set to be able to observe any improvement in the visual quality of the images reconstructed using RM.

MSE estimates using various factors of N_d

We further went on and investigated using the LVQ algorithm to see whether results improve by manipulating the factors $N_d(x_k)$. As the computation time is long for a large

Table 4.18: MSE estimates for Peppers image using LVQ with 512 code vectors

Factor of N_d	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
N_d	277	572	296	308	456	437	410	243	23	84
$N_d^1 = \frac{1}{2}N_d$	273	575	295	312	456	433	408	240	22	84
$N_d^2 = 1.5N_d$	277	572	292	308	456	436	410	243	22	84
$N_d^3 = 2N_d + 1$	268	57	290	307	450	434	400	239	23	84
$N_d^4 = (N_d)^{0.5}$	272	573	292	310	457	435	406	239	22	84
$N_d^5 = log(N_d) + 1$	271	572	291	312	455	435	408	240	22	84

Table 4.19: RM-FCM MSE's with $m = 1.1$ and $m = 2$ on part of Lenna

	m	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
FCM	1.1	68	81	115	63	207	61	425	79	22	37
FCM	2	216	458	441	256	481	267	1045	228	23	76
RM-FCM	1.1	185	189	196	133	207	143	397	171	31.9	53.5
RM-FCM	2	268	500	343	249	376	305	923	250	34.2	80.6

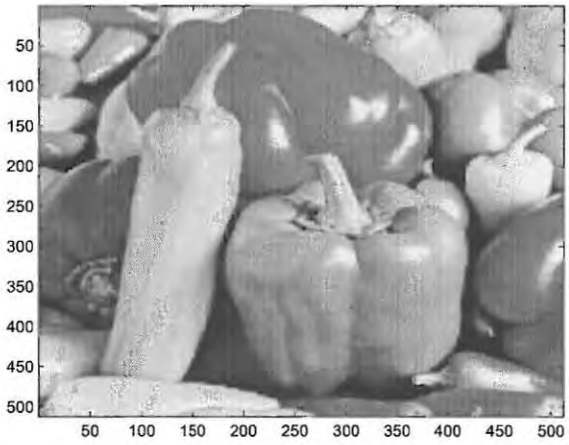
data set we then limited the training set 16384 vectors, the size of an image, with the vectors being randomly selected from the training set. Functions of N_d used for these experiments are shown in table 4.18. The results for the Peppers image using LVQ with 512 code vectors are given in table 4.18.

We observe similar results for all the functions that were applied to the Peppers images using LVQ for all the classes. We still get a similar effect as that achieved using N_d when using the various factors applied in the images that are reconstructed using replication. Some of the figures obtained using the different factors on the peppers image are shown in figure 4.35-4.36. As indicated by the MSE's there is no noticeable difference for any of the factors.

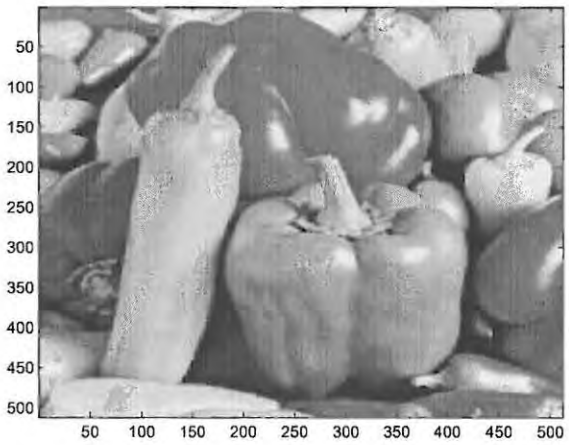
FCM: RM performance dependence on fuzzifier

We examine the effect that the fuzzifier has on the effectiveness of FCM when using RM-FCM. The lower left quarter of the Lenna image was used as a training set to generate a codebook using FCM. The lower right quarter of the Lenna image was used as a test image as was done previously in section 4.2 and 4.3. We use the FCM algorithm with $m = 1.1$ and $m = 2$, then we compare these results with those obtained using RM-FCM with FCM. When RM-FCM is performed for the case $m = 1.1$ we see an improvement in the reconstruction of the outline of Lenna's shoulder as shown in figure 4.37(c) than in FCM shown in figure 4.37(a). The homogenous areas show poorer reconstruction for $m = 1.1$

Figure 4.34: Peppers image reconstructed using replication with different factors
(a) Replication N_d^1



(b) Replication using N_d^2



(c) Replication using N_d^5

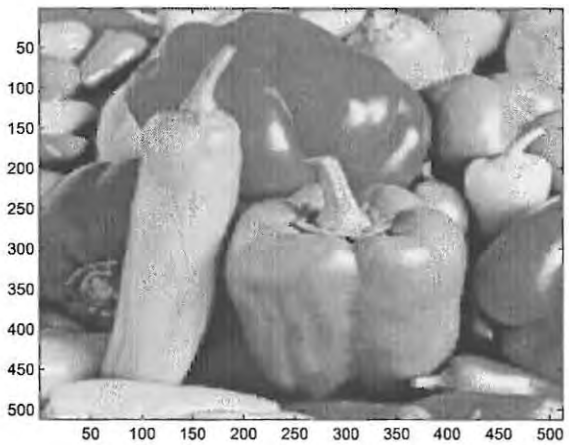


Figure 4.35: Detail of Peppers image reconstructed using replication with different factors (a) Replication using N_d^1

(b) Replication using N_d^2

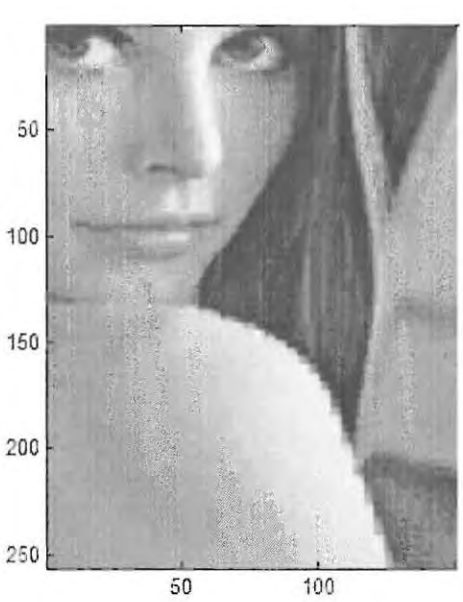
Figure 4.36: Detail of Peppers image reconstructed using replication with different factors Replication using N_d^5

with RM-FCM. The rest of the image does not show any noticeable improvement. The MSE's for RM-FCM with $m = 1.1$ indicate an improvement only for one edge class E7. All other MSE's are higher than the case with FCM. For $m = 2$ we do not observe any noticeable difference in the reconstructed image using RM-FCM compared to the case with FCM. There are however lower MSE's for four of the edge classes E3, E4, E5 and E7 as shown in table 4.19. The effect is not seen visually because although the MSE's are lower they are still very high compared to the case with $m = 1.1$. The RM approach shows some improvement for the case with $m = 2$.

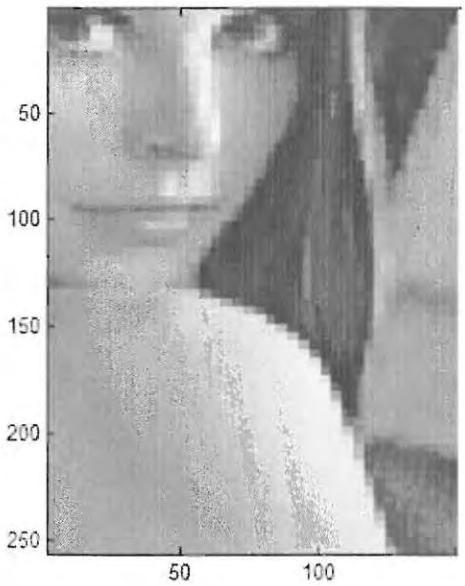
MSE estimates using random selection

Thus far we have looked at the cross validation estimates for the MSE. As mentioned in section 4.2 the assumption in equation (4.7) is violated here. In order to ascertain whether there was an effect on the MSE estimates due to using the cross validation approach, we carried out some experiments by taking a random selection of vectors from our five images as was described previously. Table 4.20 gives the MSE estimates for RM-LVQ, RM-CM and RM-FCM using 512 code vectors. We observe from these MSE's

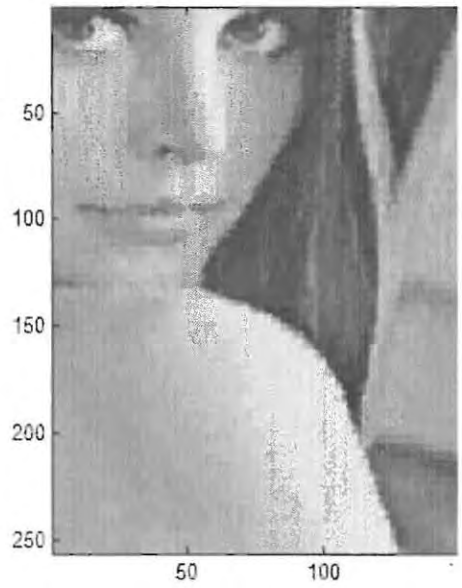
Figure 4.37: Part of Lenna image for $m = 1.1$, $m = 2$ with FCM and RM-FCM (a) $m = 1.1$ with FCM (b) $m = 2$ with FCM (c) $m = 1.1$ with RM-FCM (d) $m = 2$ RM-FCM



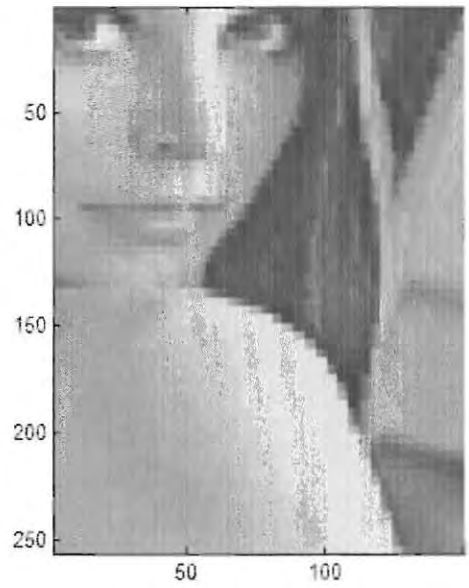
(a)



(b)



(c)



(d)

Table 4.20: MSE estimates random selection using RM

(a) Edge classes sizes using random selection

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Training set	452	444	460	408	598	534	265	249	12974	16384
Test set	421	474	448	418	595	561	287	265	12915	16384

(b) MSE Estimates with $c = 512$

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
LVQ	179	221	219	222	238	242	222	214	25	66
CM	180	176	206	214	208	207	237	195	22	60
FCM	189	177	214	220	205	213	232	203	23	60

(c) RM MSE Estimates with $c = 512$

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
RM-LVQ	186	218	226	228	240	250	224	219	27	69
RM-CM	189	183	215	223	207	210	231	205	25	63
RM-FCM	189	183	212	226	199	214	233	211	26	63

that there is no major difference between the MSEs with and without RM for all of the algorithms. When using the cross validation approach there is no major difference in the MSEs for the cases where the proportions for the edge classes are the same for the training set and the test set. We obtain a similar result using the approach with random selection.

A full set of the results from all the experiments is given in Appendix A. The Matlab programs used are also given in Appendix B. In the next chapter we give a discussion of the results that are reported here and draw conclusions from the implementation of these clustering algorithms. We also give a summary of all the algorithms and methods.

5 Discussions and Conclusion

In this chapter we give a discussion of the methods used in the implementation and draw conclusions based on the results. In section 5.1 we give a summary of Chapter 1 to Chapter 3 which describes and gives the background for the methods used in the experiments described in Chapter 4. We then give an overview of learning vector quantization (LVQ), c-means (CM) and fuzzy c-means algorithm (FCM) and their implementation in section 5.2. We discuss the results that were obtained from their implementation. In section 5.3 we examine the results that were obtained using classified vector quantization (CVQ) and hence deduce the effectiveness and advantages of using it with a particular focus on edge preservation. Section 5.4 dicusses the replication method (RM) as a method for edge preservation and gives the advantages and shortcomings of this method.

5.1 Summary

We have looked into vector quantization (VQ) as a technique used for data compression. We began by describing the neccessity of data compression in the use of digital systems. We gave a brief overview of scalar quantization from which VQ is drawn. Our focus in this thesis was VQ applied to image compression. To this end we gave a description of the use of VQ for images. In order to apply VQ the image must first be transformed into a set of vectors. This is done by subdividing the image into blocks, and it is these blocks that are then represented as vectors. VQ is employed by seeking a finite set of vectors, a codebook, to represent these image vectors. We are able to do this because images often contain redundant data, that arises from the fact that neighbouring pixels will often have similar pixel values. We also mentioned that an important part of VQ is the clustering process in which a codebook is found. There are others ways that may be employed in codebook creation but for our purposes we have focused on the use of clustering algorithms. The performance of the quantizers was measured using the mean squared error (MSE) between the original image and the quantized image.

We introduced a new approach that views VQ as a vector-valued multiple regression.

This is an idea that is drawn from regression trees. We used this method to derive the MSE and the standard error of the MSE. Cross validation is a method of testing the generalisation properties of a predictor and is employed here as a method of estimating the MSE.

The algorithms that we investigated were learning vector quantization (LVQ), c-means (CM) and the fuzzy c-means (FCM) algorithms. LVQ is a sequential algorithm that employs unsupervised learning using a neural network whose weight vectors are the code vectors. CM is a batch algorithm in which the codebook is made up of the mean vectors of the partition. Fuzzy c-means is a batch algorithm that employs fuzzy set theory and generalises c-means to fuzzy clusters.

In Chapter 3 we look into edge detection and methods that have been employed for edge preserving codebook design. The edges in the image often constitute object boundaries. In terms of the pixel values, the edges are the areas where there is a steep gradient between the pixel values in neighbouring pixels. The edges are a important part of an image as our eyes are more sensitive to them than to homogenous areas. We described the use of gradient operators in edge detection where the example used was the Sobel operator. During edge detection the gradient of each pixel in the image is approximated. A threshold is used to determine which pixels are edge and non edge pixels. The edge direction of each edge pixel is also determined using the gradient operators.

One of the shortcomings of the algorithms that employ VQ is poor edge preservation. Thus we described two methods that have been suggested in order to improve the edge preservation qualities. In section 3.3 we described classified vector quantization (CVQ) suggested by Ramamurthi and Gersho [17]. This method uses edge detection to identify the edge areas in the image. The edge areas are then quantized separately from homogenous areas. Since VQ uses the blocks that are converted into vectors, it is necessary to determine the blocks that contain edges. To determine an edge block for our purposes we had to have a minimum of 6 edge pixels in a 4×4 block. The edge direction of the block is determined by the edge direction that most edge pixels have. We are thus able to classify the edge blocks in 8 directions and non-edge class. A problem that may arise in edge detection is the overlap of classes. In CVQ, quantization is done separately for each class. A codebook is created for each class separately and the blocks are quantized separately. For reconstruction all the blocks are brought back together to make up the reconstructed image.

Another method that was suggested by Chou, Su and Lai [40] is also described as a method that may be used for edge preservation. This method is referred to here as the

replication method (RM). The edges are not as well represented as the non-edge areas during VQ because there are not enough code vectors that are used for the edge areas. The idea is to replicate all the vectors that are in sparse areas so that more code vectors are allocated to those areas. Since the edges are in sparse areas this would then increase the number of code vectors that are used to represent these image vectors.

In Chapter 4 we gave a description of the experiments carried out in order to investigate the ideas presented in Chapter 1 to Chapter 3. In the next section we give the conclusions that are drawn for the various experiments that were conducted.

5.2 LVQ, CM and FCM

We begin by discussing the initialisation of these algorithms. For LVQ and CM the initial codebook is randomly chosen from image vectors that are in the training set to avoid empty clusters and under utilised code vectors. We found that, for most of the edge classes different random initialisations do not influence the quality of the reconstructed images. We did however have instances using the CM that there were empty clusters even though we had employed this method. Linde, Buzo and Gray suggested a different method for initialisation that involves splitting [27]. For FCM we used an initial codebook that was randomly chosen to be independent of the training set. We found that out of the three algorithms FCM has less variation in the MSE due to the initialisation. We also also observed here that the trends in the classes remained the same for all algorithms.

In Chapter 2 we discussed the connection between the LVQ and CM. We showed that LVQ could be considered as a sequential steepest descent algorithm of the CM function. This would mean that the results obtained using LVQ and CM should be similar. We went on to show the relationship between CM and FCM and saw that for a fuzzifier that is close to 1, the partition that is generated using CM is similar to FCM. As the algorithms are connected in this manner we would expect some similarities in the clustering trends for the three algorithms.

For all the algorithms the edge classes exhibit higher MSE estimates than the non edge class. We saw evidence of this for all the images used in the experiments. We used the difference images and viewed reconstructed images and in all cases we saw that the edges were not as clearly represented as the non edge regions. We can conclude from this that using VQ it is more difficult to code the edge areas than the non-edge areas in image compression. We then scrutinised the individual edge classes to find that each algorithm

had a similar effect on the edge class performance. This we have seen in the classes that have the highest and lowest MSE's which are the same for all three algorithms.

Large codebooks are preferred in VQ because they produce lower MSE's. The MSE's are lower because there are more code vectors that are available to represent the original vectors and thus the image vectors are better represented. We see evidence of this in our experiments. We observed that as we increased the codebook size that the MSE's became lower. This was observed for all edge classes and overall.

The shortcoming of using larger codebooks is the increased amount of time that is required to generate the codebook. The time to determine the nearest code vector also increases with the size of the codebook. We showed that the time that is needed to generate a codebook increases as we increase the codebook size. We observe that the three algorithms take different times to generate the codebook and reconstruct the test image. LVQ was found to be far more time efficient than the other two algorithms taking only 13 minutes for codebook creation and reconstruction compared to CM and FCM which takes 4 hours and 39 minutes for a codebook with 256 code vectors. We also found that we have to convert CM and FCM into sequential algorithms especially for large codebooks because it requires a lot of memory space to store the distance matrix for a large codebook. Converting the algorithms to sequential algorithm adds to the amount of time that is needed to generate the codebook. This however is the case when using Matlab software. This may perhaps be overcome by using another programming language such as C or C++ that executes for loops faster.

FCM has a parameter that is unique to it, the fuzzifier m. We observed the effect that the fuzzifier had on the image compression in experiments that were conducted on the Lenna image. We found that for $m = 6$ the visual quality of the image is very poor. The homogemous areas are not well represented and there are no edge details that are clearly visible. We see that as m is decreased towards 1, the MSE's become lower and there is an improvement in the visual quality that is observed. As m get closer to 1 there is more detail that is observed in the reconstruction. For $m \approx 1$ we observe that the FCM partitions approach the CM partitions [32].

The MSE's especially for the edge classes show great variation for all the algorithms between the different images. When we employed cross validation we violated the assumption that the MSE's for each test set are approximately the same. We however see that using the cross validation estimate we can still draw the same conclusions about trends in the edge classes and the effect of increasing codebook size on the MSE. We can also compare the average performance between the algorithms using this estimate.

5.3 Classified vector quantization

In CVQ we employed an edge detector to classify the image vectors into edge classes. We were unable to classify the pixels that are on the border of the image, that is the first and the last, row and column. This is because we use a 3×3 convolution mask to calculate the gradient of each pixel in relation to its neighbouring pixel. This affects up to 3% of the blocks in edge block classification. Once we had classified the image vectors we separated our image vectors into edge classes and each class was quantized separately. For the set of images that we used the distribution for image vectors was found to be uneven. We found that the south west (E7) and north east edge classes had the least edge vectors.

For the experiments done using CVQ we allocated 75% of the code vectors to the edge classes. We found that it was necessary to use a larger proportion of the code vectors for the edge classes than for the non edge class. There are higher MSE estimates for the edge classes when there is a smaller proportion of code vectors that are allocated to the edge classes than to the non-edge class. We decided to distribute the code vectors evenly between the edge classes because for all the images the edge statistics were not the same for the training set and the test set. We did experiments in which we allocated double the number of code vectors to the diagonal edge classes than the non-diagonal classes as suggested by Ramamurthi and Gersho [17]. The MSE's for the latter case were higher than for the case with an even number of code vectors allocated to each edge class.

The MSE estimates obtained using CVQ were still higher for the edge classes than for the non edge classes. In comparison with the results obtained without using CVQ we observed that the MSE's were higher using CVQ. This means that there is even more degredation using CVQ. The Lenna and Boat images had some edge classes that had lower MSE's. On viewing the reconstructed images however we found that there was no noticeable difference in the visual quality of the images created with a codebook generated using CVQ. We then increased the number of code vectors allocated to the edge classes from 75%. This yielded slightly better MSE's over the edge classes. Visually though, there was no difference evident in the images particularly when using CVQ-LVQ.

Although we found that CVQ did not improve the edges in terms of the MSE, there is an advantage in the computational time taken in generating the codebook for CM and FCM. Using CM we reduce the time taken for codebook creation and reconstruction from an hour and 22 minutes to 26 minutes for a codebook of size 256. The improvement is even moreso for FCM for which the time reduces from 4 hours and 39 minutes to 38

minutes for 256 code vectors.

For FCM we examined the effect of CVQ on the MSE's for different values of the fuzzifier m. We displayed the reconstructed image for the case where $m = 1.1$ and where $m = 2$. We observed that for $m = 1.1$ there was no improvement on the visual quality of the reconstructed image. There were however more details that could be seen in the reconstructed image with $m = 2$. In analysing the MSE's we also saw that half of the edge classes showed a considerable difference when CVQ was employed. The homogenous areas were not well represented with $m = 2$ and thus this overshadowed the effect that arose when we use CVQ-FCM. We can conclude though, that CVQ-FCM is effective in improving the quality of the edges for a fuzzifier that is not close to 1.

We calculate the cross validation estimate of the MSE by calculating the mean of the MSE estimates for the five images. When the CVQ estimates were compared to the previous estimates, the MSE's were higher for CVQ for all the edge classes. This corresponds to the deductions that we have made using the estimates from the individual images. As mentioned previously, we violated the key assumption for the use of cross validation because of the large variation that is in the MSE estimates for each image. We therefore considered taking a random selection of vectors from the five images to create a training set and a test set of vectors. We did this to ascertain whether the results are affected by using the cross-validation approach to estimate the MSE. Using a random selection approach we saw that the distribution edge/non edge vectors was roughly the same for the training and test set. We performed CVQ using the training set, and tested it on the test vectors. We found that the MSE estimates were slightly higher or for some classes similar when using CVQ than when using the method without CVQ. This result is the same as the result that we obtained using the cross-validation approach.

5.4 Replication method (RM)

LVQ, CM and FCM allocate more code vectors to areas where vectors are densely populated during codebook creation because they are algorithms that minimise the MSE during training. This then leads to the poor representation of the edge regions in quantized images because there are inadequate code vectors that are used to represent the edges that are in the sparsely populated areas. RM allows us to replicate the vectors that are in sparse regions in order to force more code vectors to be used for these areas that contain edges. To do this a replication factor N_d was calculated for each image vector. Those vectors that were located in sparse regions had a high value of N_d. For

the five images that we used the replicated training set was approximately twice the size of the original training set. The maximum number of replicates for an image vector was between 15 and 19 depending upon the training set.

The first thing that we note about RM is that we increase the size of the training data. This means that there is an increase in the amount of time that is used to create a codebook. Using CM it took 7 hours for codebook creation an reconstruction, of which 4 hours was used to calculate N_d and to replicate the data set. For FCM it takes approximately 14 hours and 45 minutes to generate a codebook that had 256 code vectors. This method is therefore not time efficient and this is a disadvantage in comparison to the other two methods.

RM however does have some favourable results for some of the images. The MSE estimates for the peppers and Zelda image were lower for the edge classes when RM was employed, compared to the case without modifications and with CVQ. The other images had MSE's that were similar to the methods discussed in section 5.2. We then examined the test and the training sets used for Peppers and Zelda images. We observed that the edge classes of the training set had larger proportions compared to the proportions that were in the test set for all the edge classes. The other three images had inadequate edges in their training sets as their training sets had smaller proportions of edges than the test set. We tried to make some adaptations to the replication factor so that there would be more replicates for the edge vectors. The five factors that we used gave results that were similar to those using the original value N_d.

We then examined the cross-validation estimate of the MSE's using RM. The MSE's were higher using RM than in the method without any modifications for RM-CM and RM-LVQ. The MSE's were slightly lower when using FCM which indicates that the method is more favourable for FCM than for the other two algorithms. We also estimated the MSE by taking a random selection of image vectors from the five images for the training and test sets in a similar way as described in section 5. The MSE estimates with a random selection using RM were similar to those that were achieved without using RM.

5.5 Conclusion

We have shown that LVQ, CM and FCM may be used effectively in image compression. The disadvantage of VQ is that the edges are not represented well because there are inadequate code vectors that are allocated to the regions with edges. CVQ may be used in order to improve the edge preservation in VQ but it may only be effective if the

edges classes do not overlap. CVQ has the advantage that it reduces the computational time that is required to generate a codebook especially for CM and FCM which take a long time to generate a codebook. RM may also be used to preserve edges in image compression. It is however sensitive to the training set. The distribution of the training set must have a greater proportion of vectors in the edge classes for this method to be effective. It however takes a longer time to generate a codebook using this method for image compression.

Bibliography

[1] A. Gersho and R.M. Gray, "Vector Quantization and Signal Compression", Kluwer Academic Publishers, 1992.

[2] R.D. Dony and S. Haykin, "Neural Network Approach to Image Compression", in proceedings of the IEEE, vol.83, no.2, 1995, pp.288-303.

[3] A. Gersho, "On the Structure of Vector Quantizers", IEEE Trans. Inform. Theory vol. IT-28, 1982, pp.157-166.

[4] R.M. Gray, "Vector Quantization", IEEE ASSP Mag., vol.1, 1984, pp.4-29.

[5] M. Sonka, V. Hlavac and R. Boyle, "Image Processing, Analysis and Machine Vision", Brookes/Cole Publishing Company, 1999.

[6] F. Jelinek, "Statistical Methods for Speech Recognition", MIT Press, 1997.

[7] B.D. Ripley, "Pattern Recognition and Neural Networks", Cambridge Univerisity Press, 1996.

[8] S.C. Ahalt and J.E. Fowler, "Vector Quantization using Artificial Neural Networks", in proceedings of the International Workshop on Adaptive Methods and Emergent Techniques for Signal Processing and Communications, 1993, pp.42-61.

[9] S.C. Ahalt and J. Fowler, "Vector Quantization using Artificial Neural Network Models", in proceedings of the 7th Tyrrhenian International Workshop on Digital Communications, 1995, pp.346-357.

[10] E. Yair, K. Zerger and A. Gersho, "Competitive Learning and Soft Competition Vector Quantizer Design", IEEE Trans. on Signal Processing, vol.40, no.2, 1992, pp.294-308.

[11] R.O. Duda, P.E. Hart and D.G. Stork, "Pattern Classification", Wiley Interscience, 2001.

[12] N.B. Karayiannis and P-I Pai, "Fuzzy Algorithms for Learning Vector Quantization", IEEE Trans. on Neural Networks, vol.7, no.5, 1996, pp.1196-1211.

[13] R.C. Gonzalez and P. Wintz, "Digital Image Processing", Addison and Wesley, 1977.

[14] S.J. Solari, "Digital Video and Audio Compression", McGraw-Hill, 1997.

[15] M. Seul, L. O'Gorman and M.J. Sammon, "Practical Algorithms for Image Analysis", Cambridge University Press, 2000.

[16] E.L. Hall, "Computer Image Processing and Recognition", Academic Press, 1979.

[17] B. Ramamurthi and A. Gersho, "Classified Vector Quantization in Images", IEEE Trans. on Communications, vol.34, no.11, 1986, pp.1105-1115.

[18] L. Breiman, J. Friedman, C.J. Stone and R.A. Olshen, "Classification and Regression Trees", Chapman and Hall, 1993.

[19] R.V. Hogg and A.T. Craig, "Introduction to Mathematical Statistics", Macmillan Publishing, 1978.

[20] T. Hastie, R. Tibshirani and J.H. Friedman, "Elements of Statistical Learning: Data mining, Inference and Prediction", Springer, 2001.

[21] B.S. Everitt, S. Landau and M. Leese, "Cluster Analysis", Oxford University Press, 2001.

[22] S. Abe, "Pattern Classification, Neuro-fuzzy Methods and their Comparison", Springer, 2001.

[23] M.L. Brandeau and S.S. Chiu, "Parametric Facility Location in a Tree Network with an L_p norm Cost Function", Transportation Science, vol.22, no.1, 1988, pp.59-69.

[24] E.F. Krausse, "Taxicab Geometry", Addison and Wesley, 1975.

[25] S. Theodoridis and K. Koutrombus, "Pattern Recognition", Elsevier Academic Press, 2003.

[26] J.A. Hartigan, "Clustering Algorithms", John Wiley, 1975.

[27] Y. Linde, A. Buzo and R.M. Gray, "An Algorithm for Vector Quantizer Design", IEEE Trans. on Communications, vol.28, no.1, 1980, pp 1964-1967.

[28] K.K. Paliwal and V. Ramasubramanian, "Comments on Modified K-means Algorithm for Vector Quantization Design", IEEE Trans. on Image Processing, vol.9, no.11, 2000, pp.1964-1967.

[29] J.C. Dunn, "A Fuzzy Relative of the ISODATA Process and its Use in Detecting Compact Well Separated Clusters", Journal of Cybernetics, vol.3, no.3, 1974, pp.32-57.

[30] J.C. Bezdek, "Pattern Recognition with Fuzzy Objective Function Algorithms", Plenum Press, 1981.

[31] N.B. Karayiannis and J.C. Bezdek, "An Integrated Appoach to Fuzzy Learning Vector Quantization and Fuzzy c-means Clustering", IEEE Trans. on Fuzzy Systems, vol.3, no.4, 1997, pp.1196-1211.

[32] E.C. Tsao, J.C. Bezdek and N.R. Pal, "Fuzzy Kohonen Clustering Networks", Pattern Recognition, vol.27, no.5, 1994, pp.757-764.

[33] R. J. Hathaway and J.C. Bezdek, "Recent Convergence Results for the Fuzzy c-means Clustering Algorithms", Journal of Classification, vol.5, 1988, pp.237-247.

[34] D.W. Patterson , "Artificial Neural Networks", Prentice Hall, 1996.

[35] I. Pitas, C. Kotropoulos, N. Nikolaidis, R. Yang and M. Gabbouj, "Order Statistics Learning Vector Quantizer", IEEE Trans. on Image Processing, vol.5, no.6, 1996, pp.1048-1053.

[36] N.B. Karayiannis, "A Methodology for Constructing Fuzzy Algorithms for Learning Vector Quantization", IEEE Trans. on Neural Networks, vol.8, no.3, 1997, pp.505-518.

[37] N. Karayiannis, J.C. Bezdek, N.R. Pal, R.J. Hathaway and P. Pai, "Repairs to GLVQ : A New Family of Competitive Learning Schemes", IEEE Trans. on Neural Networks, vol.7, no.5, 1996, pp.1062-1071.

[38] N.R. Pal, J.C. Bezdek, and E.C.Tsao, "Generalized Clustering Networks and Kohonen's Self Organizing Scheme", IEEE Trans. on Neural Networks, vol.4, no.4, 1993, pp.549-557.

[39] R.L. Burden and J.D. Faires, "Numerical Analysis", Brookes/Cole Publishing, 1997.

[40] C.H. Chou, M.C. Su and E.Lai, "A New Cluster Validity Measure and its Application to Image Compression", Pattern Analysis Application, vol.7, 2004, pp.205-220.

[41] B.S. Lipkin and A.Rosenfeld, "Picture Processing and Psychopictorics", Academic Press, 1970.

[42] R. Kirsch, "Computer Determination of the Constituent Structure in Biological Images", Compt. Biomed Res, vol.4, no.3, 1971, pp.315-328.

[43] A.K. Jain, "Fundamentals of Digital Image Processing", Prentice Hall, 1989.

[44] L.S. Davis, "A Survey of Edge Detection Techniques", Computer Graphics and Image Processing, vol.4, 1975, pp.248-270.

[45] C. Watkins, A. Sadun and S. Marenka, "Mordern Image Processing : Warping, Morphing, and Classical Techniques", Academic Press Professional, 1993.

Appendix A

MSE Estimates

Table 0.1: LVQ with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	206	206	322	222	270	202	319	281	23.4	69.2
Boat	198	192	256	271	401	326	251	296	23.9	98.8
Goldhill	170	173	195	221	170	189	172	172	49.1	85.4
Peppers	269	531	209	267	280	390	319	248	25.6	75.6
Zelda	89	95	120	122	118	126	138	123	21.6	31.9
Mean	186	239	220	221	248	247	240	224	28.7	72.2

Table 0.2: CM with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	219	210	340	235	279	227	343	300	22.7	71.6
Boat	207	198	240	267	331	263	248	244	21.5	89.5
Goldhill	173	174	195	216	177	183	180	173	48.4	85.4
Peppers	254	559	212	254	303	393	324	240	23.2	74.3
Zelda	83	92	121	113	121	119	132	118	20.2	30.3
Mean	187	247	222	217	242	237	245	215	27.2	70.2

Table 0.3: FCM with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	212	220	298	232	231	197	289	235	23.8	66.3
Boat	216	210	267	321	404	338	274	269	23.6	103
Goldhill	190	209	242	261	225	239	193	190	42.9	91.2
Peppers	322	621	268	279	371	448	335	271	26.7	85.4
Zelda	94	117	147	139	146	162	143	130	23.6	35.9
Mean	197	275	242	245	278	253	272	216	23.5	70.5

Table 0.4: CVQ-LVQ with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	220	223	299	237	232	211	296	229	22.5	66.3
Boat	230	232	276	320	397	338	287	278	24.1	105.2
Goldhill	197	223	239	262	226	252	196	201	46.3	95.5
Peppers	321	663	261	298	356	468	343	275	26.9	87.2
Zelda	88	164	152	141	162	169	139	126	21	34
Mean	211	301	245	252	275	288	252	222	28.2	77.6

Table 0.5: CVQ-CM with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	219	224	297	239	228	194	287	224	22.7	65.3
Boat	222	226	259	326	384	349	280	272	23.2	102.9
Goldhill	192	203	245	264	228	248	201	199	49	96.5
Peppers	314	642	288	272	367	450	366	276	25.9	86.1
Zelda	86	121	145	149	151	157	157	119	22.7	35.2
Mean	207	283	247	250	272	280	258	218	28.7	77.2

Table 0.6: CVQ-FCM with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	212	220	298	232	231	197	289	235	23.8	66.3
Boat	216	210	267	321	404	338	274	269	23.6	103
Goldhill	190	209	242	261	225	239	193	190	42.9	91.2
Peppers	322	621	268	279	371	448	335	271	26.7	85.4
Zelda	94	117	147	139	146	162	143	130	23.6	35.9
Mean	207	275	244	246	275	277	247	219	28.1	76.4

Table 0.7: RM-LVQ with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	209	212	323	260	261	229	328	293	36.8	81.2
Boat	203	192	264	280	397	340	247	319	24.3	101
Goldhill	185	191	210	239	187	200	186	193	56.2	95.1
Peppers	278	670	248	323	301	390	376	259	30.4	86.7
Zelda	103	110	131	139	127	145	148	136	27.3	38.4
Mean	196	275	235	201	255	261	257	240	35	80.4

Table 0.8: RM-CM with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	197	195	272	220	183	162	285	228	22.3	60.1
Boat	194	196	244	287	317	288	248	255	26.1	89.1
Goldhill	210	208	217	252	240	248	197	202	43.0	93.0
Peppers	259	524	214	259	270	384	312	251	25.3	74.5
Zelda	87	105	129	132	129	135	138	125	22.5	33.4
Mean	189	246	215	230	228	243	236	212	27.8	70.0

Table 0.9: RM-FCM with $c = 256$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	194	184	267	224	181	168	283	233	22.2	60
Boat	202	198	253	297	312	277	252	261	20.6	90
Goldhill	220	205	225	250	235	242	195	215	42.4	92.6
Peppers	258	497	215	253	267	376	307	246	25.6	73.5
Zelda	81	108	123	138	119	137	140	137	22.4	33.3
Mean	191	238	217	232	223	240	235	218	26.6	69.9

Table 0.10: LVQ with $c = 512$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	191	187	298	200	241	182	278	257	21	62.4
Boat	179	171	226	250	362	305	223	267	22.1	89.9
Goldhill	149	153	167	194	154	166	158	154	42	75
Peppers	227	456	187	233	233	344	261	217	23.3	66.1
Zelda	76	86	106	102	101	108	117	103	18.9	27.7
Mean	164	211	197	196	218	221	207	200	25.5	64.2

Table 0.11: CM with $c = 512$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	200	199	321	211	267	198	309	273	20.1	65.5
Boat	182	174	204	238	275	232	224	213	19	77.9
Goldhill	155	157	175	194	165	166	162	158	40.1	74.6
Peppers	228	478	190	227	259	355	281	209	20.6	65.5
Zelda	71	86	105	100	102	103	114	94	17.1	25.8
Mean	167	219	199	194	214	211	218	189	23.4	61.9

Table 0.12: FCM with $c = 512$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	166	167	242	191	161	139	252	196	16.9	50.3
Boat	179	164	205	235	270	241	214	212	17.1	75.7
Goldhill	156	161	177	194	164	170	165	156	31.5	68.8
Peppers	283	655	316	345	487	460	435	254	20.3	87.5
Zelda	79	101	116	118	141	129	136	113	16.4	27.5
Mean	173	250	211	217	245	228	240	186	20.4	62

Table 0.13: CVQ-LVQ with $c = 512$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	198	201	255	204	196	165	246	201	18.9	55.8
Boat	197	196	236	279	344	284	243	240	20.9	90.2
Goldhill	176	183	212	237	189	207	181	178	39	81.7
Peppers	287	511	233	253	303	401	295	235	22.8	73.6
Zelda	81	108	125	114	129	141	129	109	17.4	28.3
Mean	188	240	212	217	232	240	219	193	23.8	65.9

Table 0.14: CVQ-CM with $c = 512$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	196	192	253	200	190	154	246	189	18.7	54.5
Boat	197	192	228	267	317	284	240	236	19.4	86.7
Goldhill	177	176	205	230	185	196	178	173	39.5	80.7
Peppers	280	498	224	233	298	384	287	232	22.5	71.5
Zelda	77	106	116	121	114	150	123	104	18.6	28.7
Mean	185	233	205	210	221	234	215	187	23.7	64.4

Table 0.15: CVQ-FCM with $c = 512$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	199	187	255	197	189	156	235	193	20	55.4
Boat	187	175	224	274	317	269	238	232	20	85
Goldhill	172	173	201	232	186	193	175	169	34.8	76.6
Peppers	280	505	230	261	300	385	292	228	23.4	73.2
Zelda	77	103	114	112	128	127	118	107	18.7	28.8
Mean	183	229	205	215	224	226	212	186	23.4	63.8

Table 0.16: RM-LVQ with $c = 512$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	190	181	288	197	215	192	271	245	28.8	67.2
Boat	177	175	230	254	354	302	223	276	22.2	90
Goldhill	162	168	184	211	160	176	169	168	45.7	80.8
Peppers	248	607	227	280	266	355	336	238	26.7	77.5
Zelda	82	94	116	114	112	120	118	109	22.2	31.6
Mean	172	245	209	211	221	229	223	201	29.1	69.4

Table 0.17: RM-CM with $c = 512$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	164	171	231	188	157	140	241	194	18.8	51.1
Boat	171	164	214	246	275	249	220	228	17.2	77.2
Goldhill	193	177	198	224	266	223	172	186	37.7	82
Peppers	223	479	188	223	239	354	277	267	22.3	66
Zelda	74	87	103	108	108	115	114	100	19	28
Mean	165	216	187	198	209	216	205	195	23	60.9

Table 0.18: RM-FCM with $c = 512$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	174	168	226	190	156	142	242	200	18.8	51.2
Boat	177	163	213	245	270	246	217	233	17.6	77.2
Goldhill	193	183	196	226	213	222	176	195	35.6	81.2
Peppers	225	459	187	222	224	345	273	209	22.1	65.8
Zelda	79	87	111	110	106	116	117	110	18.6	27.9
Mean	170	212	187	199	194	214	205	189	22.5	60.7

Table 0.19: LVQ with $c = 1024$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	163	157	261	170	196	163	231	221	17.9	53.2
Boat	153	143	203	215	317	277	192	235	18.5	78.1
Goldhill	133	133	150	172	131	142	140	138	34.5	64
Peppers	205	422	166	200	194	291	229	186	21.2	58.5
Zelda	62	71	88	88	89	92	93	88	16.6	24
Mean	143	185	174	169	185	193	177	174	21.8	55.6

Table 0.20: CM with $c = 1024$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	174	168	290	190	213	177	274	235	16.7	56.3
Boat	151	151	177	208	243	216	193	187	16.2	68
Goldhill	135	135	152	175	145	145	139	133	31.4	62.4
Peppers	203	461	169	201	234	338	249	190	18.4	59.9
Zelda	61	72	85	85	88	91	102	85	14.3	21.9
Mean	145	197	175	172	185	193	191	166	19.4	53.7

Table 0.21: FCM with $c = 1024$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	145	149	214	163	137	120	207	170	15	43.7
Boat	153	144	177	199	234	212	188	189	15.6	66.3
Goldhill	138	135	133	178	145	150	145	140	27.9	60.8
Peppers	264	624	285	314	441	420	406	228	18.9	80.8
Zelda	65	85	103	100	128	116	119	101	14.5	24.3
Mean	153	227	182	191	217	204	213	166	18.4	55.2

Table 0.22: CVQ-LVQ with $c = 1024$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	178	177	215	182	165	144	219	178	16.4	48.4
Boat	174	170	202	242	273	250	222	212	18.2	77.4
Goldhill	155	163	190	208	169	175	167	158	33	71.2
Peppers	263	454	209	225	254	346	251	203	20.4	60.8
Zelda	72	98	108	102	108	122	107	96	15.1	20.6
Mean	168	212	185	192	194	207	193	169	20.6	55.7

Table 0.23: CVQ-CM with $c = 1024$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	171	166	218	175	167	130	222	172	16.3	47.6
Boat	175	161	200	242	273	233	216	212	17.2	75.3
Goldhill	152	155	181	201	157	169	160	153	33	69.2
Peppers	255	468	203	215	244	342	255	203	20.2	64.2
Zelda	70	87	100	97	106	109	106	191	15.9	24.7
Mean	165	207	180	186	189	197	192	186	20.5	56.2

Table 0.24: CVQ-FCM with $c = 1024$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	174	163	226	169	166	134	201	174	16.5	47.5
Boat	165	154	196	223	259	226	205	203	17.3	72.5
Goldhill	148	148	199	202	158	169	157	150	30.1	66.4
Peppers	256	463	205	232	263	362	260	218	20.1	65.7
Zelda	65	91	99	103	111	104	110	90	15.7	24.6
Mean	162	204	185	186	191	199	187	167	20	55.5

Table 0.25: RM-LVQ with $c = 1024$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	159	153	249	169	182	157	219	209	23	55.7
Boat	156	157	210	225	328	279	206	250	19.4	81.3
Goldhill	144	146	161	184	140	147	150	149	37.7	68.9
Peppers	209	539	196	249	246	331	291	209	23.6	68.8
Zelda	71	81	98	93	95	102	99	93	18.5	26.4
Mean	148	215	183	184	198	203	193	182	24.4	60.2

Table 0.26: RM-CM with $c = 1024$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	151	153	205	166	139	125	209	171	16	44.7
Boat	157	145	189	218	244	222	202	207	15.5	69.1
Goldhill	174	164	182	212	192	208	157	170	34.1	75.2
Peppers	203	427	167	197	213	333	236	185	19.6	59
Zelda	64	75	90	94	89	97	95	82	15.6	23.3
Mean	150	193	167	177	175	197	180	163	20.2	54.3

Table 0.27: RM-FCM with $c = 1024$

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	153	149	191	162	135	124	200	172	16.2	43.9
Boat	153	144	187	209	242	219	201	204	15.8	68.4
Goldhill	170	160	179	203	192	193	157	168	32.2	72.5
Peppers	201	398	163	189	193	303	226	184	19.8	56.7
Zelda	68	77	97	91	94	100	101	88	15.8	23.8
Mean	149	186	163	171	171	188	177	163	20	53.1

Table 0.28: LVQ with $c = 512$ Double diagonal

Image	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	213	221	240	199	215	186	238	194	18.9	56.7
Boat	213	211	211	266	376	316	232	223	20.9	93.5
Goldhill	191	209	199	226	315	232	173	166	39.0	84.8
Peppers	310	562	220	248	329	436	284	225	22.8	75.9
Zelda	86	117	116	113	142	158	120	105	17.4	28.8

Table 0.29: Initialisation of LVQ with part of Lenna $c = 256$

Initialisation	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
1	225	362	265	171	270	200	606	233	68.1	94.9
2	241	353	271	168	280	205	642	242	64.5	93.0
3	243	312	287	161	296	197	640	223	71.1	98.8
4	221	365	298	166	275	201	625	212	68.08	96.4
5	239	319	269	155	265	216	599	239	68.7	95.7
6	248	347	273	167	280	202	648	232	68.3	96.2
7	244	356	277	161	283	210	656	229	64.0	93.0
8	224	347	270	159	270	200	595	216	67.3	93.9
9	229	360	286	171	287	203	670	209	67.6	96.2
10	228	365	279	186	297	211	632	203	68.3	97.1
11	233	359	286	162	281	198	655	219	67.7	95.7
12	224	359	262	161	278	210	654	239	67.2	95.1
13	240	348	275	163	277	200	622	227	67.0	94.5
14	221	305	290	164	280	203	621	227	66.9	94.6
15	241	343	287	157	272	203	645	219	67.5	95.3
16	15	348	281	167	282	203	648	225	67.5	95.4
17	221	368	275	166	275	202	637	219	69.7	97
18	237	354	261	170	267	197	626	242	68.1	95.0
19	243	314	278	158	283	205	593	222	68.0	95.3
20	229	353	274	172	271	201	614	223	67.6	94.8

Table 0.30: Random initialisation of CM with part of Lenna $c = 256$

Experiment	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
1	228	336	276	168	325	215	722	221	64.2	95.2
2	223	314	267	172	301	203	645	221	61.6	90.6
3	240	383	267	171	312	197	674	228	64.2	93.8
4	224	399	266	178	315	199	694	250	57.6	88.6
5	207	319	268	169	300	224	669	223	74.4	102.6
6	24711	358	283	188	323	206	700	224	63.3	94.4
7	216	391	270	175	300	200	637	215	64.8	93.5
8	234	326	279	203	331	245	679	218	169.6	187.7
9	232	360	283	179	312	210	704	233	61.5	92.6
10	215	436	292	192	335	221	762	206	68.5	101.3
11	231	376	288	184	281	217	609	216	60.7	90.3
12	231	295	258	166	320	212	652	214	67.2	96.1
13	228	329	275	173	311	197	677	219	62.8	92.3
14	232	395	287	198	331	214	707	242	65.2	97.2
15	225	384	284	193	327	210	624	214	68.9	98.7
16	240	349	290	171	326	222	673	233	66.9	98
17	211	383	295	168	305	221	714	222	64.1	95
18	218	397	277	183	320	215	706	215	70.8	101
19	240	378	284	163	282	197	665	219	61.5	90.6
20	237	317	283	168	304	200	647	233	61.5	91

Table 0.31: Initialisation of FCM with part of Lenna $c = 256$

Experiment	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
1	227	396	264	164	281	205	645	233	62.9	91.5
2	234	398	252	159	277	218	588	236	46.6	77.0
3	231	300	274	147	280	210	610	215	53.5	82.6
4	235	383	263	163	287	208	659	227	56.7	86.6
5	234	361	262	174	284	196	609	235	59.9	88.2
6	223	398	272	163	283	214	623	237	47.8	78.8
7	234	381	271	158	278	197	626	231	46.6	76.8
8	213	370	265	166	292	198	631	215	58.9	87.6
9	231	379	277	172	279	194	662	222	57.4	86.7
10	234	404	273	165	290	196	701	230	62.1	91.5
11	231	393	274	164	285	201	662	235	46.8	77.8
12	228	384	264	168	289	205	652	233	62.0	91.0
13	235	372	287	160	284	210	643	235	50.1	81.2
14	238	390	263	159	278	205	661	230	59.3	88.4
15	230	388	269	161	275	205	654	231	51.9	82.0
16	229	385	270	171	291	210	661	234	52.7	83.6
17	235	352	267	169	285	203	639	227	46.5	77.3
18	233	395	261	158	283	207	664	221	49	79.8
19	220	382	262	169	279	215	606	228	46.9	77.4
20	230	390	269	177	282	191	631	231	61	89.4

Table 0.32: Number of image vectors in test and training sets

(a) Test sets

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	183	135	581	538	644	610	286	231	13176	16384
Boat	771	816	567	447	864	764	288	339	11528	16384
Goldhill	828	803	474	411	727	592	311	394	11844	16384
Peppers	287	367	422	367	366	456	308	214	13597	16384
Zelda	173	137	211	258	353	303	163	149	14637	16384
Total	2242	2258	2255	2021	2954	2725	1356	1327	64782	16384

(b) Training sets

	E1	E2	E3	E4	E5	E6	E7	E8	E9	W
Lenna	2059	2123	1674	1483	2310	2115	1070	1096	51606	65536
Boat	1471	1442	1688	1574	2090	1961	1068	988	53254	65536
Goldhill	1414	1455	1781	1610	2227	2133	1045	933	52938	65536
Peppers	1955	1891	1833	1654	2588	2269	1048	1113	51185	65536
Zelda	2069	2121	2044	1763	2601	2422	1193	1178	50145	65536

Appendix B

Matlab Programs

Block classifier

```
function[block_classes] = block_classifier(W, Np)
%classifies image blocks into an edge class
%Input:  W which gives the classification of each pixel in each block,
%Np = minimum number of edge pixels required for edge block
%Output:  Block class
count = [];
[xsize, ysize] = size(W);
n = xsize*ysize;
block_classes = [];
for k = 1:ysize
  count0 = find(W(:,k)== 0);
  if (length(count0) >= n - Np)% identify no edge blocks
    block_classes(k) = 0;
else
    count(1) = sum(W(:,k)== 1);
    count(2) = sum(W(:,k)== 2);
    count(3) = sum(W(:,k)== 3);
    count(4) = sum(W(:,k)== 4);
    count(5) = sum(W(:,k)== 5);
    count(6) = sum(W(:,k)== 6);
```

```
    count(7) = sum(W(:,k)== 7);
    count(8) = sum(W(:,k)== 8);
    count(9) = sum(W(:,k)== 0);
    [max_count, index_max] = max(count);
    if index_max == 9
        block_classes(k) = 0;
    else
    block_classes(k)= index_max;
    end
  end
end
return
```

Blocks to vectors

```
function V = blocks2vectors(array,xsize,ysize)
% transforms an image array to an array of xsize*ysize vectors
% by scanning the image array blockwise and rearranging the blocks
array_size = size(array);
% n rows, m columns
n = array_size(1);
% number of rows (y direction)
m = array_size(2);
% number of columns (x direction)
xsteps = floor(m/xsize);
ysteps = floor(n/ysize);
V = [];
for k=1:ysize:n
  for i=1:xsize:m
    block = array(k:k+ysize-1,i:i+xsize-1);
    % cut out block of image vector = reshape(block,xsize*ysize,1);
```

```
    % make column vector from block V = [V,vector];
    % append column
  end
end
return
```

Class edge blocks

```
function[E1, E2, E3, E4, E5, E6, E7, E8] = class_edge_blocks( block_classes)
% Separates the edge classes
% Input:  blocks classes
% Output:  Indices of the edge blocks that belong to each edge class
E1 = find(block_classes ==1);
E2 = find(block_classes ==2);
E3 = find(block_classes ==3);
E4 = find(block_classes ==4);
E5 = find(block_classes ==5);
E6 = find(block_classes ==6);
E7 = find(block_classes ==7);
E8 = find(block_classes ==8);
return
```

Pixel Classifier

```
function [edge_array,m, edge_classes] = classifier(image, threshold)
% classifies image pixels into edge classes where each class is labelled as
% follows:  0:  no edges
% 1:  edges in north direction
% 2:  edges in south direction
% 3:  edges in north west direction
% 4:  edges in south east direction
% 5:  edges in west direction
```

```
% 6:  edges in east direction
% 7:  edges in south west direction
% 8:  edges in north east direction
% Input:  image, Threshold
edge_classes = [];
edge_array = zeros(512, 512, 8);
edge_image = image;
for k = 1:8
  edge_array(:, :, k) = edge_enhancer(image, k);
end
[m, edge_classes] = max(edge_array, [], 3);
return
```

Classify train images

```
function [E1_v, E2_v, E3_v, E4_v, E5_v, E6_v, E7_v, E8_v, E9_v] = classify_train(trainimages,
threshold, Vthreshold, xsize, ysize, Np)
% Used to classify the training images for CVQ-LVQ, CVQ-CM and CVQ-FCM
% Inputs:  training images, threshold for pixel classification,
%        Vthreshold = variance threshold used to eliminate non edge blocks from
%         edge classes, block size = xsize × ysize, Np = minimum number of pixels
%         for an edge block
% Outputs:  training vectors separated into their respective edge classes
image1 = trainimages(:, 1:512);
image2 = trainimages(:, 513:1024);
image3 = trainimages(:, 1025:1536);
image4 = trainimages(:, 1537:2048);
%*************************classify image1 ******************************
[edge_array1,m1, pixel_classes1] = classifier2(image1, threshold);
% convert pixel class array to block-vectors
class_vectors1 = blocks2vectors(pixel_classes1, xsize, ysize);
```

```
% convert image array to block-vectors
image_vectors1 = blocks2vectors(image1, xsize, ysize);
% classify each block as an edge or non-edge block
[block_classes1] = block_classifier(class_vectors1, pcnt0);
% check whether blocks have been correctly identified as edge blocks using
% the variance and then get indices for edge and non-edge blocks
% variance threshold:  (take block vectors with variance above vthreshold to
% be true edge blocks and get new block classes
[edge_blocks1, non_edge_blocks1, block_classes1] = edge_check(image_vectors1, block_classes1,
Vthreshold);
% Separate edge blocks to directions N, S, NW, SE, W, E, SW, NE
[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes1);
E9 = non_edge_blocks1;
image1_E1 = image_vectors1(:, E1);
image1_E2 = image_vectors1(:, E2);
image1_E3 = image_vectors1(:, E3);
image1_E4 = image_vectors1(:, E4);
image1_E5 = image_vectors1(:, E5);
image1_E6 = image_vectors1(:, E6);
image1_E7 = image_vectors1(:, E7);
image1_E8 = image_vectors1(:, E8);
image1_E9 = image_vectors1(:, E9);
%**********************classify Image 2 ********************************** [edge_array2,m2,
pixel_classes2] = classifier2(image2, threshold);
% convert pixel class array to block-vectors
class_vectors2 = blocks2vectors(pixel_classes2, xsize, ysize);
% convert image array to block-vectors
image_vectors2 = blocks2vectors(image2, xsize, ysize);
% classify each block as an edge or non-edge block
[block_classes2] = block_classifier(class_vectors2, pcnt0);
```

```
% check whether blocks have been correctly identified as edge blocks using
% the variance and then get indices for edge and non-edge blocks
% variance threshold (take block vectors with variance above vthreshold to be true
edge blocks and get new block classes
[edge_blocks2, non_edge_blocks2, block_classes2] = edge_check(image_vectors2, block_classes2,
Vthreshold);
% Separate edge blocks to directions N, S, NW, SE, W, E, SW, NE
[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes2);
E9 = non_edge_blocks2;
image2_E1 = image_vectors2(:, E1);
image2_E2 = image_vectors2(:, E2);
image2_E3 = image_vectors2(:, E3);
image2_E4 = image_vectors2(:, E4);
image2_E5 = image_vectors2(:, E5);
image2_E6 = image_vectors2(:, E6);
image2_E7 = image_vectors2(:, E7);
image2_E8 = image_vectors2(:, E8);
image2_E9 = image_vectors2(:, E9);
%************************classify image3 *********************************
[edge_array3,m3, pixel_classes3] = classifier2(image3, threshold);
% convert pixel class array to block-vectors
class_vectors3 = blocks2vectors(pixel_classes3, xsize, ysize);
% convert image array to block-vectors
image_vectors3 = blocks2vectors(image3, xsize, ysize);
% classify each block as an edge or non-edge block
[block_classes3] = block_classifier(class_vectors3, pcnt0);
% check whether blocks have been correctly identified as edge blocks using
% the variance and then get indices for edge and non-edge blocks
% variance threshold (take block vectors with variance above vthreshold to
% be true edge blocks and get new block classes [edge_blocks3, non_edge_blocks3,
```

```
block_classes3] = edge_check(image_vectors3, block_classes3, Vthreshold);
% Separate edge blocks to directions N, S, NW, SE, W, E, SW, NE
[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes3);
E9 = non_edge_blocks3;
image3_E1 = image_vectors3(:, E1);
image3_E2 = image_vectors3(:, E2);
image3_E3 = image_vectors3(:, E3);
image3_E4 = image_vectors3(:, E4);
image3_E5 = image_vectors3(:, E5);
image3_E6 = image_vectors3(:, E6);
image3_E7 = image_vectors3(:, E7);
image3_E8 = image_vectors3(:, E8);
image3_E9 = image_vectors3(:, E9);
%************************classify image4 *********************************
[edge_array4,m4, pixel_classes4] = classifier2(image4, threshold);
% convert pixel class array to block-vectors
class_vectors4 = blocks2vectors(pixel_classes4, xsize, ysize);
% convert image array to block-vectors
image_vectors4 = blocks2vectors(image4, xsize, ysize);
% classify each block as an edge or non-edge block
[block_classes4] = block_classifier(class_vectors4, pcnt0);
% check whether blocks have been correctly identified as edge blocks using
% the variance and then get indices for edge and non-edge blocks
% variance threshold (take block vectors with variance above vthreshold to
% be true edge blocks and get new block classes
[edge_blocks4, non_edge_blocks4, block_classes4] = edge_check(image_vectors4,
block_classes4, Vthreshold);
% Separate edge blocks to directions N, S, NW, SE, W, E, SW, NE
[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes4);
```

```
E9 = non_edge_blocks4;
image4_E1 = image_vectors4(:, E1);
image4_E2 = image_vectors4(:, E2);
image4_E3 = image_vectors4(:, E3);
image4_E4 = image_vectors4(:, E4);
image4_E5 = image_vectors4(:, E5);
image4_E6 = image_vectors4(:, E6);
image4_E7 = image_vectors4(:, E7);
image4_E8 = image_vectors4(:, E8);
image4_E9 = image_vectors4(:, E9);
E1_v = [image1_E1, image2_E1, image3_E1, image4_E1];
E2_v = [image1_E2, image2_E2, image3_E2, image4_E2];
E3_v = [image1_E3, image2_E3, image3_E3, image4_E3];
E4_v = [image1_E4, image2_E4, image3_E4, image4_E4];
E5_v = [image1_E5, image2_E5, image3_E5, image4_E5];
E6_v = [image1_E6, image2_E6, image3_E6, image4_E6];
E7_v = [image1_E7, image2_E7, image3_E7, image4_E7];
E8_v = [image1_E8, image2_E8, image3_E8, image4_E8];
E9_v = [image1_E9, image2_E9, image3_E9, image4_E9];
```

Learning vector quantization

```
function codebook = clusternet(data,init_codebook, epoch_number,waitnumber)
% function codebook = clusternet(data,init_codebook, epoch_number)
% performs lvq clustering of a data set.  initial clustercenters
% (=codevectors) are iteratively moved towards optimal clustercenters
% movements of centers is plotted in first two dimensions
% inputs:  data (array of format (length of vectors times number of data)
% init_codebook (initial array of clustercenters)
% epoch_number (number of runs through data, must be >=2)
% waitnumber:  number of seconds for pausing the plot
```

```
% (recommendend 0.5 for visualization, 0 for fast processing)
% outputs:  codebook (array of optimal clustercenters)
plotyes = 0; % 0 = don't plot , 1 = plot
% controls plotting (for larger data sets do not plot!)
codebook = double(init_codebook);
data = double(data);
size_data = size(data);
size_codebook = size(codebook);
number_of_data = size_data(2);
number_of_codevectors = size_codebook(2);
initial_learning_rate = 0.75;
final_learning_rate = 0.001;
if size_data(1) ~= size_codebook(1)
%     disp('formats of codevectors and data does not match')
return
end
if plotyes == 1
    figure
    % open new window
    hold on
    % hold window active for plotting
end
for epoch=1:epoch_number
    eta=(initial_learning_rate-final_learning_rate)/(1-epoch_number)*(epoch-1)+
% initial_learning_rate;
% learning rate (=cooling scheme)
%******** plot the data and codevectors (first two dimensions) redraw in each epoch
*
if plotyes == 1
    plot(data(1,:),data(2,:),'ok')
```

```
    %plot the data as small black circles plot(codebook(1,:),codebook(2,:),'sr')
    % plot the codevectors as little red squares
end %**************************************************************************
for k=1:number_of_data
%********* plot datum in green *************
    if plotyes == 1 plot(data(1,k),data(2,k),'og')
end
%****************************************** % find closest code vector to datum:
c v = repmat(data(:,k),1,number_of_codevectors);
d = sum((v-codebook).^2,1);
% columnwise sum winner = min(find(d == min(d))); %******** plot winning center in
blue *********
if plotyes == 1
    plot(codebook(1,winner),codebook(2,winner),'sb')
    pause(waitnumber)
    % short pause
    plot(codebook(1,winner),codebook(2,winner),'sw')
    % make winner invisible
end %*********************************************
codebook(:,winner) = codebook(:,winner) + eta*(data(:,k)-codebook(:,winner));
% update
%******** plot updated winner *****************
if plotyes == 1
    plot(codebook(1,winner),codebook(2,winner),'sr')
    plot(data(1,k),data(2,k),'ok')
    % change color of datum again
end
%*********************************************
end
end
```

```
hold off

return
```

C-means algorithm

```
function [U,codebook] = cmeans(x,c0)

% c-means clustering

% inputs:  (n x N) data matrix x of N data x in R^n % initial cluster

centers c0=(c01,...,c0c), (n x c) matrix

% output:  partition matrix U (c x N), cluster centers c

plotyes = 0;

% change to 1 if plots are wanted

[n,N] = size(x);

[n1,c] = size(c0);

if abs(n - n1)>0

  disp('dimensions do not agree')

  return

end

dist = 1;

it = 1;

while dist > 0.001

U = zeros(c,N);

D = zeros(c,N);

%************** compute matrix of distances ******************

for i=1:c

  for k=1:N

    D(i,k) = norm(x(:,k) - c0(:,i));

  end

end

%************ compute partition matrix ************************

[m,index] = min(D,[],1);
```

```
for k=1:N
  U(index(k),k) = 1;
end
%**************Check for empty clusters ***********************%
sumU = sum(U,2);
% row sums
index0 = find(sumU == 0);
%find empty clusters
randN = randperm(N);
for i=1:length(index0)
  U(:, randN(i)) = 0;
  %make sample belong to no cluster
  U(index0(i), randN(i)) = 1;
  % assign sample to cluster i
end
%**** plot the data ***************************************************
if plotyes == 1
  figure(it)
  colors = ['r','b','y','c','k','g','m'];
  marks = ['+', '<', '*', 's', 'd', 'o'];
  hold on
  for i=1:c
    color = colors(mod(i,7)+1);
    mark = marks(mod(i,6)+1);
    index = find(U(i,:)>0);
    %plot(x(1,index),x(2,index),['o',color])
    plot(x(1,index),x(2,index),[mark,color])
    %plot(c0(1,i),c0(2,i),['+',color])
    plot(c0(1,i),c0(2,i), ['p', 'b'])
```

```
  end
  hold off
  disp('press any key to continue!')
  pause
end
%*********** compute new cluster centers *********************
for i=1:c
  index = find(U(i,:)  > 0);
  c1(:,i) = sum(x(:,index),2)/sum(U(i,:));
end
dist = norm(c0-c1);
it = it+1;
c0 = c1;
end (while)
codebook = c0;
return
```

C-means sequential

```
function [U,codebook] = cmeans(x,c0)
% c-means clustering
% inputs:  (n x N) data matrix x of N data x in R^n
% initial cluster centers c0=(c01,...,c0c), (n x c) matrix
% output:  partition matrix U (c x N), cluster centers c
plotyes = 1;
% change to 0 if plots are not wanted
[n,N] = size(x);
[n1,c] = size(c0);
if abs(n - n1)>0
  disp('dimensions do not agree')
  return
```

```
end
dist = 1;
it = 1;
while dist > 0.001
  U = zeros(c,N);
  D = zeros(c,N);
%************** compute matrix of distances ******************
  for i=1:c
    for k=1:N
      D(i,k) = norm(x(:,k) - c0(:,i));
    end
  end
%************ compute partition matrix ************************
  [m,index] = min(D,[],1);
  for k=1:N
    U(index(k),k) = 1;
  end
%**************Check for empty clusters ************************%
  sumU = sum(U,2);
  % row sums index0 = find(sumU == 0);
  %find empty clusters randN = randperm(N);
  for i=1:length(index0)
    U(:, randN(i)) = 0;
    %make sample belong to no cluster U(index0(i), randN(i)) = 1;
    % assign sasmple to cluster i end %**** plot the data
************************************************
  if plotyes == 1 figure(it) colors = ['r','b','y','c','k','g','m'];
    marks = ['+', '<', '*', 's', 'd', 'o'];
    hold on
```

```
    for i=1:c color = colors(mod(i,7)+1);
      mark = marks(mod(i,6)+1);
      index = find(U(i,:)>0);
      %plot(x(1,index),x(2,index),['o',color])
      plot(x(1,index),x(2,index),[mark,color])
      %plot(c0(1,i),c0(2,i),['+',color])
      plot(c0(1,i),c0(2,i), ['p', 'b'])
    end
    hold off
    disp('press any key to continue!')
    pause
  end
%********** compute new cluster centers **********************
  for i=1:c
    index = find(U(i,:)  > 0);
    c1(:,i) = sum(x(:,index),2)/sum(U(i,:));
  end
  dist = norm(c0-c1);
  it = it+1;
  c0 = c1;
end
codebook = c0;
return
```

Convolution

```
function out = convolution(image, mask)
size_image = size(image);
out = image * 0;
for i = 2 :size_image(1) - 1
  for k = 2:  size_image(2) - 1
```

```
    out(i,k) = sum(sum(image(i-1:i+1, k-1:k+1).*mask));
  end
end
```

Edge check

```
function[edge_blocks, non_edge_blocks, block_classes] = edge_check(V, block_classes,
Vthreshold)
% Checking using the variance whether blocks have been correctly classified
% as edges.  A block with a variance lower that Vthreshold is not considered
% an edge block
% Input:  V(image blocks), block classes, variance threshold
%Outputs:  Edge/Non edge blocks = gives indices of edge/non edge blocks
edge_blocks = [];
non_edge_blocks = [];
block_variance = var(V);
i= 1;
non_edge_blocks = find(block_classes == 0);
edges = find(block_classes > 0);
for k = 1:  length(edges)
  if block_variance(edges(k)) > Vthreshold
    edge_blocks(i) = edges(k);
    i = i + 1;
  else
    non_edge_blocks= [non_edge_blocks, edges(k)];
    block_classes(edges(k)) = 0;
  end
end
return
```

Edge enhancer

```
function out = edge_enhancer(image, direction)
%performs the convolution of the image with the mask in the direction given
%Inputs:  Inputs :  image and edge direction as number btwee 1 and 8
%Output :  Out = convolution of image with edge mask
switch upper(direction)
case 1 mask = [1,2,1; 0,0,0;-1,-2,-1];% North
case 2 mask = [-1,-2,-1; 0,0,0;1,2,1]; % South
case 3 mask = [2,1,0; 1,0,-1;0,-1,-2]; % North West
case 4 mask = [-2,-1,0; -1,0,1;0,1,2]; % South East
case 5 mask = [1,0,-1; 2,0,-2;1,0,-1]; % West
case 6 mask = [-1,0,1; -2,0,2;-1,0,1]; % East
case 7 mask = [0,-1,-2; 1,0,-1;2,1,0]; %South west
case 8 mask = [0,1,2;-1,0,1;-2,-1,0]; % North East
end
out = convolution(image, mask); return
```

Error measures

```
function [MSE] = Error_Measures(image, quant_image)
% Given two images in the form of M*N arrays,
%the original image and the reconstructed image
% The mean squared eror (MSE) between the two images is calculated
% Inputs :Array of Original image and array of reconstructed image
% Outputs:  MSE
[xsize, ysize] = size(image);
MSE =sum(sum((quant_image-image).^2)) /(xsize * ysize);
```

Fuzzy c-means

```
function [U,c,defuzzind] = fuzzy_cmeans(x,c0,exponent,maxit)
% fuzzy c-means clustering
% inputs:  (n x N) data matrix x of N data x in R^n
```

```
% initial cluster centers c0=(c01,...,c0c), (n x c) matrix
% exponent:  fuyyzifier (recommended:  1<exponent<=6, Use 1.1 for image compression)
% output:  fuzzy partition matrix U (c x N), cluster centers c,
% defuzzind:  cell array with indices of defuzzified clusters
%            (x(:,defuzzind{k}) = samples of cluster k)
plotyes = 0; % change to plotyes = 1 to plot the data
[n,N] = size(x);
[n1,c] = size(c0);
if abs(n - n1)>0
  disp('dimensions do not agree')
  return
end
dist = 1;
it = 1;
while (it <= maxit) & dist >0.001
  U = zeros(c,N);
  D = zeros(c,N);
%************** compute matrix of distances ******************
  for i=1:c
    for k=1:N
      D(i,k) = norm(x(:,k) - c0(:,i));
    end
  end
%************ compute partition matrix ************************
  for i=1:c
    for k=1:N
      U(i,k) = (sum((D(i,k)./D(:,k)).^(2/(exponent-1))))^(-1);
    end
  end
```

```
%**** plot the data ************************************************
  if plotyes == 1
  figure(it)
  colors = ['r','k','y','c','b','g','m'];
  marks = ['+', '<', '*', 's', 'd', 'o'];
  [m,maxind] = max(U,[],1);
  hold on if n>=2
  % two dim case:  plot only crisp clusters
    for k=1:c
      color = colors(mod(k,7)+1);
      mark = marks(mod(i,6)+1);
      index = find(maxind == k);
      %plot(x(1,index),x(2,index),['o',color])
      plot(x(1,index),x(2,index),[mark,color])
      %plot(c0(1,k),c0(2,k),['p',color])
      plot(c0(1,k),c0(2,k), ['p', 'b'])
    end
  end
  if n==1
  % one dim case:
  plot fuzzy clusters
  for k=1:c color = colors(mod(k,7)+1);
    plot(x,U(k,:),['.',color])
  end
  end
  hold off
c0
% display the centers
  disp('strike any key to continue')
```

```
  pause
end
%********** compute new cluster centers *********************
Unew = U.^exponent;
for i=1:c
  for j=1:n
    %dimension of data
    temp(j) = Unew(i,:)*x(j,:)';
  end
  c1(:,i) = temp'/sum(Unew(i,:));
end
dist = norm(c0-c1);
it = it+1;
c0 = c1;
end
%**** determine defuzzified clusters
[m,maxind] = max(U,[],1);
defuzzind = {} ;
%empty cell array for index of defuzzified clusters
for k=1:c
  defuzzind{k} = find(maxind == k);
end
c = c0;
return
```

Fuzzy c-means sequential

```
function [U,c] = fuzzy_cmeans_seq(x,c0,exponent,maxit)
% fuzzy c-means clustering
% does not return the fuzzy partion matrix but only the crisp partition
% ( applications:  codebook design for lvq )
```

```
% inputs:  (n x N) data matrix x of N data x in R^n
% initial cluster centers c0=(c01,...,c0c), (n x c) matrix
% exponent:  fuyyzifier (recommended:  1<exponent<=6, best:  2)
% output:  matrix U (1 x N) with cluster label per sample, cluster centers c
% remark:  no check if datum sits on cluster center!!!
no termination in % case of non-convergence!!!!
[n,N] = size(x); [n1,c] = size(c0);
if abs(n - n1)>0
  disp('dimensions do not agree')
  U = [];
  c = [];
  return
end
dist = 1;
it = 1;
while (it <= maxit) & dist >0.001
  c1 = c0*0;
  % initialization
  for k=1:c
    ck = 0;
    denomk = 0;
    for i=1:N
    % determine U(k,i) (not needed later therefore not stored)
    dik = (x(:,i)-c0(:,k))'*(x(:,i)-c0(:,k));
    sum = 0;
    for j=1:c
      sum = sum + (dik/((x(:,i)-c0(:,j))'*(x(:,i)-c0(:,j))))^(1/(exponent-1));
    end
    Uki = sum^(-1);
```

```
    ck = ck + (Uki^exponent)*x(:,i);
    denomk = denomk + Uki^exponent; end c1(:,k) = ck/denomk;
  end
  dist = norm(c0-c1);
  it = it+1;
  c0 = c1;
end
% determine the crisp partition
U = zeros(1,N);
for i=1:N
  di = zeros(1,c);
  for k=1:c
  di(k) = (x(:,i)-c0(:,k))'*(x(:,i)-c0(:,k));
  % squared distance end [m,label] = min(di);
  U(i) = label;
end
c = c0;
return
```

Quantize image

```
function q = quantize_image(image, codebook, xsize,ysize)
% given a codebook, an image is compressen.  each image block (of xsize % times ysize)
% is reformed to an array of length xsize*ysize and compared
% to all codebook vectors.  the number of the closest codebook vector is
% stored %
% inputs:  image :  image to be compressed
% codebook:  array of code vectors (= cluster centers)
% xsize, ysize:  blocksize used for codebook creation % (clustering)
image = double(image);
[n,m] = size(image);
```

```
size_codebook = size(codebook);
number_of_codevectors = size_codebook(2);
if xsize*ysize ~= size_codebook(1)
  disp('dimensions of blocks and code vectors do not match')
  return
end
xsteps = floor(m/xsize);
ysteps = floor(n/ysize);
q = zeros(ysteps,xsteps);
% array for quantization numbers
for k=1:ysteps
  for i=1:xsteps
    xpos = (i-1)*xsize+1;
    ypos = (k-1)*ysize+1;
    block = image(ypos:ypos+ysize-1,xpos:xpos+xsize-1);
    vector = reshape(block,xsize*ysize,1);
    % make block to column
    v = repmat(vector,1,number_of_codevectors);
    d = sum((v - codebook).^2,1);
    winner = min(find(d == min(d)));
    q(k,i) = winner;
  end
end
return
```

Quantize image vectors

```
function q = quantize_image(image, codebook, xsize,ysize)
% given a codebook, an image is compressed.  each image block (of xsize
% times ysize) is reformed to an array of length xsize*ysize and compared
% to all codebook vectors.  the number of the closest codebook vector is
```

```
% stored
% inputs:  image :  image to be compressed
% codebook:  array of code vectors (= cluster centers)
% xsize, ysize:  blocksize used for codebook creation
% (clustering)
image = double(image);
[n,m] = size(image);
size_codebook = size(codebook);
number_of_codevectors = size_codebook(2);
if xsize*ysize ~= size_codebook(1)
  disp('dimensions of blocks and code vectors do not match')
  return
end
xsteps = floor(m/xsize);
ysteps = floor(n/ysize);
q = zeros(ysteps,xsteps);
% array for quantization numbers
for k=1:ysteps
  for i=1:xsteps
    xpos = (i-1)*xsize+1;
    ypos = (k-1)*ysize+1;
    block = image(ypos:ypos+ysize-1,xpos:xpos+xsize-1);
    vector = reshape(block,xsize*ysize,1);
    % make block to column
    v = repmat(vector,1,number_of_codevectors);
    d = sum((v - codebook).^2,1);
    winner = min(find(d == min(d)));
    q(k,i) = winner;
  end
```

```
end
return
```

Reconstruct image

```
function quant_image = reconstruct(compressed_image,codebook,xsize,ysize)
% reconstructs a quantized image
% inputs:  compressed_image:  array with labels of clusters
% codebook:  list of cluster centers
% xsize, ysize:  blocksizes
[m,n] = size(compressed_image);
quant_image = zeros(m*ysize,n*xsize);
for k=1:m
  for i=1:n
    winner = compressed_image(k,i);
    block = reshape(codebook(:,winner),ysize,xsize);
    xpos = (i-1)*xsize+1;
    ypos = (k-1)*ysize+1;
    quant_image(ypos:ypos+ysize-1,xpos:xpos+xsize-1) = block;
  end
end
return
```

Reconstruct image vectors

```
function quant_image = reconstruct_vectors(compressed_image,codebook)
% reconstructs a quantized image
% inputs:  compressed_image:  array with labels of clusters
% codebook:  list of cluster centers [m,n] = size(compressed_image);
quant_image =[];
for k=1:n winner = compressed_image(:,k);
  block = codebook(:,winner);
  quant_image(:,k) = block;
```

```
end
return
```

Replicate image vectors

```
function [imageV_vectors] = replicate(image_vectors, Nd)
%replicates the sparse regions of the data creating a virtual data set
%using a replication factor Nd
% Inputs:  image_vectors of training set, Nd = replication number
% Outputs:  imageV_vectors = virtual training data set
[M,N] = size(image_vectors);
Nv = sum(Nd);
imageV_vectors = [];
for j = 1:N
  R = repmat(image_vectors(:,j), 1, Nd(j));
  imageV_vectors = [imageV_vectors, R];
end
return
```

Replication number

```
function [Nd] = replication_number(image_vectors)
% Calculates the replication factor for each image vector in the training data
[m,n] = size(image_vectors);
E = 0;
E2 = 0;
for i=1 :  n
  for j = i:n
    E = E + norm(image_vectors(:,i) - image_vectors(:,j));
    E2 = E2 + norm(image_vectors(:,i)-image_vectors(:, j))^2;
  end
end
```

```
E = E/(n*(n+1)/2);
E2 = E2/(n*(n+1)/2);
sigma2 = E2 - E^2;
for j= 1 :n
  D(j) = 0;
  for i = 1:  n
    D(j) = D(j) + exp(-norm(image_vectors(:,i)- image_vectors(:,j))^2/sigma2);
  end D(j) = D(j) - 1;
end
MD = max(D);
Nd = floor(10 * log10(MD./D)) + 1;
return
```

Vectors to image

```
function[new_image] = vectors2image(recon_i_vectors, image, ysize);
%converts image vectors to blocks then back to original image size
%( only for squareblocks)
[m, n] = size(image);
% size image
new_image = zeros(m,n);
% initilize new image
% reshape into array (row) of blocks
i_vectors = reshape(recon_i_vectors, ysize, (m*n)/ysize);
% get size of array of blocks
[mi, ni] = size(i_vectors);
% reshape blocks to image
for k = 0 :  (ni/n - 1)
new_image((mi*k)+1:mi*(k+1), :  )  = i_vectors(:, (n*k)+1:n*(k+1));
end
return
```

C-means

```
function [quant_image, MSE, SNR, sizeEs] = VQ_cm_eN( imagetest,imagetrain, xsize,
ysize,codevectors,threshold, Vthreshold, Np)
% calculates the codebook using fuzzy cmeans for 2 by 2 and 4 by 4 blocks
% Inputs:  image, xsize and ysize which give the block size, and the number
% of codevectors used for the non edge and edge vectors and Np is the
% minimum number of pixels classified edges required for an edge block
% Outputs:  reconstructed image, the error measures given by the MSE and SNR
% Difference image
% imagetest = testing image
% imagetrain = training image vectors (images must be converted to vectors first)
% classify each pixel
[edge_array,m, pixel_classes] = classifier2(imagetest, threshold);
% convert pixel class array to block-vectors
class_vectors = blocks2vectors(pixel_classes, xsize, ysize);
% convert image array to block-vectors
image_vectors = blocks2vectors(imagetest, xsize, ysize);
% classify each block as an edge or non-edge block
[block_classes] = block_classifier(class_vectors, pcnt0);
% check whether blocks have been correctly identified as edge blocks using
% the variance and then get indices for edge and non-edge blocks
% variance threshold (take block vectors with variance above vthreshold to
% be true edge blocks and get new block classes
[edge_blocks, non_edge_blocks, block_classes] = edge_check(image_vectors, block_classes,
Vthreshold);
% Separate edge blocks to directions N, S, NW, SE, W, E, SW, NE
[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes);
sizeEs = [length(E1), length(E2),length(E3),length(E4),length(E5),length(E6),length(E7),lengtl
sizeEs = [sizeEs, length(non_edge_blocks)];
E1_vectors = image_vectors(:, E1);
E2_vectors = image_vectors(:, E2);
```

```
E3_vectors = image_vectors(:, E3);
E4_vectors = image_vectors(:, E4);
E5_vectors = image_vectors(:, E5);
E6_vectors = image_vectors(:, E6);
E7_vectors = image_vectors(:, E7);
E8_vectors = image_vectors(:, E8);
NE_vectors = image_vectors(:, non_edge_blocks);
%select training set
t_vectors = imagetrain;
% ********************create initial codebook*****************************%
[p, q] = size(t_vectors);
steps = ceil(q/codevectors);
init_codebook = t_vectors(:, 1:steps:q);
%************************codebook creation ********************************%
% codebook using cmeans
[partmat,codebook]=cmeans_seq(t_vectors,init_codebook);
% ********************quantization using codebook*************************%
% gives the cluster label for each image vector disp('quantize...')
compressed_image_vectors=quantize_image_vectors(image_vectors, codebook);
%fuzzy c-means
%********************reconstruction of compressed image*******************%
quant_image_vectors = double(reconstruct_vectors(compressed_image_vectors, codebook));
quant_image_E1 = quant_image_vectors(:, E1);
quant_image_E2 = quant_image_vectors(:, E2);
quant_image_E3 = quant_image_vectors(:, E3);
quant_image_E4 = quant_image_vectors(:, E4);
quant_image_E5 = quant_image_vectors(:, E5);
quant_image_E6 = quant_image_vectors(:, E6);
quant_image_E7 = quant_image_vectors(:, E7);
```

```
quant_image_E8 = quant_image_vectors(:, E8);
quant_image_NE = quant_image_vectors(:, non_edge_blocks);
quant_image = vectors2image(quant_image_vectors, imagetest, xsize);
%************************** Difference image*****************************%%%
disp('difference image...')
[m,n] = size(quant_image);
diff_image = 255 - abs(imagetest(1:m, 1:n)- quant_image);
figure()
imagesc(diff_image, [0, 255])
colormap(gray)
%****************** Error measures ***************************************%
[MSE, SNR]= Error_Measures(imagetest, quant_image);
[MSE1, SNR1] = Error_Measures(E1_vectors, quant_image_E1);
[MSE2, SNR2] = Error_Measures(E2_vectors, quant_image_E2);
[MSE3, SNR3] = Error_Measures(E3_vectors, quant_image_E3);
[MSE4, SNR4] = Error_Measures(E4_vectors, quant_image_E4);
[MSE5, SNR5] = Error_Measures(E5_vectors, quant_image_E5);
[MSE6, SNR6] = Error_Measures(E6_vectors, quant_image_E6);
[MSE7, SNR7] = Error_Measures(E7_vectors, quant_image_E7);
[MSE8, SNR8] = Error_Measures(E8_vectors, quant_image_E8);
[MSE9, SNR9] = Error_Measures(NE_vectors, quant_image_NE);
MSE = [MSE1, MSE2, MSE3, MSE4, MSE5, MSE6, MSE7, MSE8, MSE9, MSE];
SNR = [SNR1, SNR2, SNR3, SNR4, SNR5, SNR6, SNR7, SNR8, SNR9, SNR];
return
```

C-means (for varying codebook sizes)

```
function [quant_image, MSE, SNR, sizeEs] = VQ_cmN( imagetest,imagetrain, xsize, ysize,codevect
Vthreshold, pcnt0)
for k= 1:  length(codevectors)
[quant_image(:,:,k), MSE(k, :), SNR(k, :), sizeEs(k,:)]  = VQ_cm_eN( imagetest,imagetrain,
xsize, ysize,codevectors(k),threshold, Vthreshold, Np);
```

```
k = k+1;

end
```

Edges seperate FCM

```
function [reconstructed_image, MSE, SNR, sizeEs] = VQ_edges_sep_FCM( testimage, trainimage,xsi
ysize,codevectors,threshold, Vthreshold, Np)

% calculates the codebook using fuzzy cmeans for xsize by ysize blocks

% Inputs:  image, xsize and ysize which give the block size, and the number

% of codevectors used for the non edge and edge vectors and Np is the

% percentage of pixels classified as non edges and edges in 8 directions

% Outputs:  reconstructed image, the error measures given by the MSE

% Difference image

%Vthreshold = 130

% train image already in vectors

%********************classify each pixel in test image ******************%

%threshold = 30;

[edge_array,m, pixel_classes] = classifier2(testimage, threshold);

% convert pixel class array to block-vectors

class_vectors = blocks2vectors(pixel_classes, xsize, ysize);

% convert image array to block-vectors

image_vectors = blocks2vectors(testimage, xsize, ysize);

% classify each block as an edge or non-edge block

[block_classes] = block_classifier(class_vectors, pcnt0);

% check whether blocks have been correctly identified as edge blocks using

% the variance and then get indices for edge and non-edge blocks

% variance threshold (take block vectors with variance above vthreshold to

% be true edge blocks and get new block classes

[edge_blocks, non_edge_blocks, block_classes] = edge_check(image_vectors, block_classes,
Vthreshold); length(non_edge_blocks);

% Separate edge blocks to directions N, S, NW, SE, W, E, SW, NE

[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes);
```

```
sizeEs = [length(E1), length(E2),length(E3),length(E4),length(E5),length(E6),length(E7),length
sizeEs = [sizeEs, length(non_edge_blocks)];
%*******************classify pixels in training set ***********************%
[E1_v, E2_v, E3_v, E4_v, E5_v, E6_v, E7_v, E8_v, E9_v] = classify_train2(trainimage,
threshold, Vthreshold, 4, 4, Np);
% remove 2 and just have classify train if using 5 images
%For quantization epochs = 70;
% set number of iterations exponent = 1.5;
% change to 2 for comparison
%******************* Quantization of non-edge blocks*********************%
codevectorsNE = codevectors(9);
non_edge_vectors = image_vectors(:, non_edge_blocks);
%select training set ( 1/2 of non-edge vectors)
t_vectors = E9_v;
% create initial codebook (all codevectors on diagonal)
ic = rand(1, codevectorsNE)*255;
dim = xsize * ysize;
init_codebook = repmat(ic, dim, 1);
% codebook creation using fuzzy c-means
[U,codebook] = fuzzy_cmeans_seq2(t_vectors,init_codebook,exponent,epochs);
% gives the cluster label for each non-edge vector
ne_cluster_labels = quantize_image_vectors(non_edge_vectors, codebook);
%reconstructs non-edge vectors using codebook
quant_ne_vectors = double(reconstruct_vectors(ne_cluster_labels, codebook));
%*************************************************************************%
%************************Quantization of edge blocks ********************%
% North edges
codevectors1 = codevectors(1);
E1_vectors = image_vectors(:, E1);
%select training set ( 1/2 of vectors)
```

```
t1_vectors = E1_v;
% create initial codebook
dim = xsize * ysize;
init_codebook1 = rand(dim, codevectors1)*255;
% codebook creation using fuzzy c-means
[U1,codebook1] = fuzzy_cmeans_seq2(t1_vectors,init_codebook1,exponent,epochs);
% gives the cluster label for each north edge vector
E1_cluster_labels = quantize_image_vectors(E1_vectors, codebook1);
%reconstructs north vectors using codebook
quant_E1_vectors = double(reconstruct_vectors(E1_cluster_labels, codebook1));
%*********************************************************************%
%*********************************************************************%
codevectors2 = codevectors(2);
E2_vectors = image_vectors(:, E2);
%select training set ( 1/2 of vectors)
t2_vectors = E2_v;
% create initial codebook
dim = xsize * ysize;
init_codebook2 = rand(dim, codevectors2)*255;
% codebook creation using fuzzy c-means
[U2,codebook2] = fuzzy_cmeans_seq2(t2_vectors,init_codebook2,exponent,epochs);
% gives the cluster label for each south edge vector
E2_cluster_labels = quantize_image_vectors(E2_vectors, codebook2);
%reconstructs south edge vectors using codebook
quant_E2_vectors = double(reconstruct_vectors(E2_cluster_labels, codebook2));
%*********************************************************************%
%*********************************************************************%
codevectors3 = codevectors(3);
E3_vectors = image_vectors(:, E3);
```

```
%select training set ( 1/2 of vectors)
t3_vectors = E3_v;
% create initial codebook
dim = xsize * ysize;
init_codebook3 = rand(dim, codevectors3)*255;
% codebook creation using fuzzy c-means
[U3,codebook3] = fuzzy_cmeans_seq2(t3_vectors,init_codebook3,exponent,epochs);
% gives the cluster label for each NW edge vector
E3_cluster_labels = quantize_image_vectors(E3_vectors, codebook3);
%reconstructs NW edge vectors using codebook
quant_E3_vectors = double(reconstruct_vectors(E3_cluster_labels, codebook3));
%**************************************************************************%
%**************************************************************************%
codevectors4 = codevectors(4);
E4_vectors = image_vectors(:, E4);
%select training set ( 1/2 of vectors)
t4_vectors = E4_v;
%initial codebook creation
dim = xsize * ysize;
init_codebook4 = rand(dim, codevectors4)*255;
% codebook creation using fuzzy c-means
[U4,codebook4] = fuzzy_cmeans_seq2(t4_vectors,init_codebook4,exponent,epochs);
% gives the cluster label for each SE edge vector
E4_cluster_labels = quantize_image_vectors(E4_vectors, codebook4);
%reconstructs SE edge vectors using codebook
quant_E4_vectors = double(reconstruct_vectors(E4_cluster_labels, codebook4));
%**************************************************************************%
%**************************************************************************%
codevectors5 = codevectors(5);
```

```
E5_vectors = image_vectors(:, E5);
%select training set ( 1/2 of vectors)
t5_vectors = E5_v;
% create initial codebook
dim = xsize * ysize;
init_codebook5 = rand(dim, codevectors5)*255;
% codebook creation using fuzzy c-means
[U5,codebook5] = fuzzy_cmeans_seq2(t5_vectors,init_codebook5,exponent,epochs);
% gives the cluster label for each W edge vector
E5_cluster_labels = quantize_image_vectors(E5_vectors, codebook5);
%reconstructs W edge vectors using codebook
quant_E5_vectors = double(reconstruct_vectors(E5_cluster_labels, codebook5));
%**************************************************************************%
%**************************************************************************%
codevectors6 = codevectors(6);
E6_vectors = image_vectors(:, E6);
%select training set ( 1/2 of vectors)
t6_vectors = E6_v;
% create initial codebook
dim = xsize * ysize;
init_codebook6 = rand(dim, codevectors6)*255;
% codebook creation using fuzzy c-means
[U6,codebook6] = fuzzy_cmeans_seq2(t6_vectors,init_codebook6,exponent,epochs);
% gives the cluster label for each E edge vector
E6_cluster_labels = quantize_image_vectors(E6_vectors, codebook6);
%reconstructs E edge vectors using codebook
quant_E6_vectors = double(reconstruct_vectors(E6_cluster_labels, codebook6));
%**************************************************************************%
%**************************************************************************%
```

```
codevectors7 = codevectors(7);
E7_vectors = image_vectors(:, E7);
%select training set ( 1/2 of vectors)
t7_vectors = E7_v;
%initial codebook creation
dim = xsize * ysize;
init_codebook7 = rand(dim, codevectors7)*255;
% codebook creation using fuzzy c-means
[U7,codebook7] = fuzzy_cmeans_seq2(t7_vectors,init_codebook7,exponent,epochs);
% gives the cluster label for each SW edge vector
E7_cluster_labels = quantize_image_vectors(E7_vectors, codebook7);
%reconstructs SW edge vectors using codebook
quant_E7_vectors = double(reconstruct_vectors(E7_cluster_labels, codebook7));
%*************************************************************************%
%*************************************************************************%
codevectors8 = codevectors(8);
E8_vectors = image_vectors(:, E8);
%select training set ( 1/2 of vectors)
t8_vectors = E8_v;
dim = xsize * ysize;
init_codebook8 = rand(dim, codevectors8)*255;
% codebook creation using fuzzy c-means
[U8,codebook8] = fuzzy_cmeans_seq2(t8_vectors,init_codebook8,exponent,epochs);
% gives the cluster label for each NE edge vector
E8_cluster_labels = quantize_image_vectors(E8_vectors, codebook8);
%reconstructs NE edge vectors using codebook
quant_E8_vectors = double(reconstruct_vectors(E8_cluster_labels, codebook8));
%*************************************************************************%
%************************** Reconstruction ***************************%
```

```
% get reconstruced image block vectors
[r,s] = size(image_vectors);
recon_i_vectors = zeros(r,s);
recon_i_vectors(:, non_edge_blocks) = quant_ne_vectors;
recon_i_vectors(:, E1)= quant_E1_vectors;
recon_i_vectors(:, E2)= quant_E2_vectors;
recon_i_vectors(:, E3)= quant_E3_vectors;
recon_i_vectors(:, E4)= quant_E4_vectors;
recon_i_vectors(:, E5)= quant_E5_vectors;
recon_i_vectors(:, E6)= quant_E6_vectors;
recon_i_vectors(:, E7)= quant_E7_vectors;
recon_i_vectors(:, E8)= quant_E8_vectors;
% convert reconstructed image vectors to blocks then
to the reconstructed % image
[reconstructed_image] = vectors2image(recon_i_vectors, testimage, xsize);
%************************* Difference image*************************%%%
[m,n] = size(reconstructed_image);
diff_image = 255 - abs(testimage(1:m, 1:n)-reconstructed_image);
figure()
%set('Name', 'Difference image ')
imagesc(diff_image, [0, 255])
colormap(gray)
%****************** Error measures ************************************%
[MSE] = Error_Measures(testimage, reconstructed_image);
[MSE1] = Error_Measures(E1_vectors, quant_E1_vectors);
[MSE2] = Error_Measures(E2_vectors, quant_E2_vectors);
[MSE3] = Error_Measures(E3_vectors, quant_E3_vectors);
[MSE4] = Error_Measures(E4_vectors, quant_E4_vectors);
[MSE5] = Error_Measures(E5_vectors, quant_E5_vectors);
```

```
[MSE6] = Error_Measures(E6_vectors, quant_E6_vectors);
[MSE7] = Error_Measures(E7_vectors, quant_E7_vectors);
[MSE8] = Error_Measures(E8_vectors, quant_E8_vectors);
[MSE9] = Error_Measures(non_edge_vectors, quant_ne_vectors);
MSE = [MSE1, MSE2, MSE3, MSE4, MSE5, MSE6, MSE7, MSE8, MSE9, MSE];
return
```

Edges separate C-means

```
function [reconstructed_image, MSE, SNR, sizeEs] = VQ_edges_sep_CM( testimage,
trainimage, xsize, ysize,codevectors,threshold, Vthreshold, Np)
% calculates the codebook using fuzzy cmeans 4 by 4 blocks
% Inputs:  image, xsize and ysize which give the block size, and the number
% of codevectors used for the non edge and edge vectors and Np is the
% percentage of pixels classified as non edges and edges in 8 directions
% Outputs:  reconstructed image, the error measures given by the MSE
% Difference image
%Vthreshold = 130;
%
% classify each pixel
[edge_array,m, pixel_classes] = classifier(testimage, threshold);
% convert pixel class array to block-vectors
class_vectors = blocks2vectors(pixel_classes, xsize, ysize);
% convert image array to block-vectors
image_vectors = blocks2vectors(testimage, xsize, ysize);
% classify each block as an edge or non-edge block
[block_classes] = block_classifier(class_vectors, pcnt0);
% check whether blocks have been correctly identified as edge blocks using
% the variance and then get indices for edge and non-edge blocks
% variance threshold (take block vectors with variance above vthreshold to
% be true edge blocks and get new block classes
[edge_blocks, non_edge_blocks, block_classes] = edge_check(image_vectors,
```

```
block_classes, Vthreshold);
length(non_edge_blocks);
% Separate edge blocks to directions N, S, NW, SE, W, E, SW, NE
[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes);
sizeEs=[length(E1),length(E2),length(E3),length(E4),length(E5),
length(E6),length(E7),length(E8)];
sizeEs = [sizeEs, length(non_edge_blocks)];
%*****************classify pixels in training set **********************%
[E1_v, E2_v, E3_v, E4_v, E5_v, E6_v, E7_v, E8_v, E9_v] = classify_train(trainimage,
threshold, Vthreshold, 4, 4, Np);
%For quantization exponent = 1.1;
%****************** Quantization of non-edge blocks*********************%
codevectorsNE = codevectors(9);
non_edge_vectors = image_vectors(:, non_edge_blocks);
%select training set
t_vectors = E9_v;
% create initial codebook (all codevectors on diagonal)
%ic = rand(1, codevectorsNE)*255;
%dim = xsize * ysize;
%init_codebook = repmat(ic, dim, 1);
[p, q] = size(t_vectors);
steps = ceil(q/codevectorsNE);
init_codebook = t_vectors(:, 1:steps:q);
% codebook creation using c-means
[U,codebook] = cmeans(t_vectors,init_codebook);
% gives the cluster label for each non-edge vector
ne_cluster_labels = quantize_image_vectors(non_edge_vectors, codebook);
%reconstructs non-edge vectors using codebook
quant_ne_vectors = double(reconstruct_vectors(ne_cluster_labels, codebook));
```

```
%************************************************************************%
%************************Quantization of edge blocks *******************%
% North edges codevectors1 = codevectors(1);
E1_vectors = image_vectors(:, E1);
%select training set
t1_vectors = E1_v;
% create initial codebook
%dim = xsize * ysize;
%init_codebook1 = rand(dim, codevectors1)*255;
[p1, q1] = size(t1_vectors);
steps1 = ceil(q1/codevectors1);
init_codebook1 = t1_vectors(:, 1:steps1:q1);
% codebook creation using c-means
[U1,codebook1] = cmeans(t1_vectors,init_codebook1);
% gives the cluster label for each north edge vector
E1_cluster_labels = quantize_image_vectors(E1_vectors, codebook1);
%reconstructs north vectors using codebook
quant_E1_vectors = double(reconstruct_vectors(E1_cluster_labels, codebook1));
%************************************************************************%
%************************************************************************%
codevectors2 = codevectors(2);
E2_vectors = image_vectors(:, E2);
%select training set
t2_vectors = E2_v;
% create initial codebook
%dim = xsize * ysize;
%init_codebook2 = rand(dim, codevectors2)*255;
[p2, q2] = size(t2_vectors);
steps2 = ceil(q2/codevectors2);
```

```
init_codebook2 = t2_vectors(:, 1:steps2:q2);
% codebook creation using c-means
[U2,codebook2] = cmeans(t2_vectors,init_codebook2);
% gives the cluster label for each south edge vector
E2_cluster_labels = quantize_image_vectors(E2_vectors, codebook2);
%reconstructs south edge vectors using codebook
quant_E2_vectors = double(reconstruct_vectors(E2_cluster_labels, codebook2));
%**************************************************************************%
%**************************************************************************%
codevectors3 = codevectors(3);
E3_vectors = image_vectors(:, E3);
%select training set ( 1/2 of vectors) t3_vectors = E3_v;
% create initial codebook
%dim = xsize * ysize;
%init_codebook3 = rand(dim, codevectors3)*255;
[p3, q3] = size(t3_vectors);
steps3 = ceil(q3/codevectors3);
init_codebook3 = t3_vectors(:, 1:steps3:q3);
% codebook creation using c-means
[U3,codebook3] = cmeans(t3_vectors,init_codebook3);
% gives the cluster label for each NW edge vector
E3_cluster_labels = quantize_image_vectors(E3_vectors, codebook3);
%reconstructs NW edge vectors using codebook
quant_E3_vectors = double(reconstruct_vectors(E3_cluster_labels, codebook3));
%**************************************************************************%
%**************************************************************************%
codevectors4 = codevectors(4); E4_vectors = image_vectors(:, E4);
%select training set
t4_vectors = E4_v;
```

```
%initial codebook creation
%dim = xsize * ysize;
%init_codebook4 = rand(dim, codevectors4)*255;
[p4, q4] = size(t4_vectors);
steps4 = ceil(q4/codevectors4);
init_codebook4 = t4_vectors(:, 1:steps4:q4);
% codebook creation using c-means
[U4,codebook4] = cmeans(t4_vectors,init_codebook4);
% gives the cluster label for each SE edge vector
E4_cluster_labels = quantize_image_vectors(E4_vectors, codebook4);
%reconstructs SE edge vectors using codebook
quant_E4_vectors = double(reconstruct_vectors(E4_cluster_labels, codebook4));
%**************************************************************************%
%**************************************************************************%
codevectors5 = codevectors(5);
E5_vectors = image_vectors(:, E5);
%select training set
t5_vectors = E5_v;
% create initial codebook
%dim = xsize * ysize;
%init_codebook5 = rand(dim, codevectors5)*255;
[p5, q5] = size(t5_vectors);
steps5 = ceil(q5/codevectors5);
init_codebook5 = t5_vectors(:, 1:steps5:q5);
% codebook creation using c-means
[U5,codebook5] = cmeans(t5_vectors,init_codebook5);
% gives the cluster label for each W edge vector
E5_cluster_labels = quantize_image_vectors(E5_vectors, codebook5);
%reconstructs W edge vectors using codebook
```

```
quant_E5_vectors=double(reconstruct_vectors(E5_cluster_labels,codebook5));
%*************************************************************************%
%*************************************************************************%
codevectors6 = codevectors(6);
E6_vectors = image_vectors(:, E6);
%select training set
t6_vectors = E6_v;
% create initial codebook
%dim = xsize * ysize;
%init_codebook6 = rand(dim, codevectors6)*255;
[p6, q6] = size(t6_vectors);
steps6 = ceil(q6/codevectors6);
init_codebook6 = t6_vectors(:, 1:steps6:q6);
% codebook creation using c-means
[U6,codebook6] = cmeans(t6_vectors,init_codebook6);
% gives the cluster label for each E edge vector
E6_cluster_labels = quantize_image_vectors(E6_vectors, codebook6);
%reconstructs E edge vectors using codebook
quant_E6_vectors = double(reconstruct_vectors(E6_cluster_labels, codebook6));
%*************************************************************************%
%*************************************************************************%
codevectors7 = codevectors(7);
E7_vectors = image_vectors(:, E7);
%select training set
t7_vectors = E7_v;
%initial codebook creation
%dim = xsize * ysize;
%init_codebook7 = rand(dim, codevectors7)*255;
[p7, q7] = size(t7_vectors);
```

```
steps7 = ceil(q7/codevectors7);
init_codebook7 = t7_vectors(:, 1:steps7:q7);
% codebook creation using fuzzy c-means
[U7,codebook7] = cmeans(t7_vectors,init_codebook7);
% gives the cluster label for each SW edge vector
E7_cluster_labels = quantize_image_vectors(E7_vectors, codebook7);
%reconstructs SW edge vectors using codebook
quant_E7_vectors = double(reconstruct_vectors(E7_cluster_labels, codebook7));
%************************************************************************%
%************************************************************************%
codevectors8 = codevectors(8);
E8_vectors = image_vectors(:, E8);
%select training set
t8_vectors = E8_v;
%dim = xsize * ysize;
%init_codebook8 = rand(dim, codevectors8)*255;
[p8, q8] = size(t8_vectors);
steps8 = ceil(q8/codevectors8);
init_codebook8 = t8_vectors(:, 1:steps8:q8);
% codebook creation using c-means
[U8,codebook8] = cmeans(t8_vectors,init_codebook8);
% gives the cluster label for each NE edge vector
E8_cluster_labels = quantize_image_vectors(E8_vectors, codebook8);
%reconstructs NE edge vectors using codebook
quant_E8_vectors = double(reconstruct_vectors(E8_cluster_labels, codebook8));
%************************************************************************%
%*************************** Reconstruction ***************************%
% get reconstruced image block vectors
[r,s] = size(image_vectors);
```

```
recon_i_vectors = zeros(r,s);
recon_i_vectors(:, non_edge_blocks) = quant_ne_vectors;
recon_i_vectors(:, E1)= quant_E1_vectors;
recon_i_vectors(:, E2)= quant_E2_vectors;
recon_i_vectors(:, E3)= quant_E3_vectors;
recon_i_vectors(:, E4)= quant_E4_vectors;
recon_i_vectors(:, E5)= quant_E5_vectors;
recon_i_vectors(:, E6)= quant_E6_vectors;
recon_i_vectors(:, E7)= quant_E7_vectors;
recon_i_vectors(:, E8)= quant_E8_vectors;
% convert reconstructed image vectors to blocks then to the reconstructed
% image [reconstructed_image] = vectors2image(recon_i_vectors, testimage, xsize);
%************************** Difference image*****************************%%%
[m,n] = size(reconstructed_image);
diff_image = 255 - abs(testimage(1:m, 1:n)-reconstructed_image);
figure()
imagesc(diff_image, [0, 255])
colormap(gray)
%****************** Error measures ***************************************%
[MSE] = Error_Measures(testimage, reconstructed_image);
[MSE1] = Error_Measures(E1_vectors, quant_E1_vectors);
[MSE2] = Error_Measures(E2_vectors, quant_E2_vectors);
[MSE3] = Error_Measures(E3_vectors, quant_E3_vectors);
[MSE4] = Error_Measures(E4_vectors, quant_E4_vectors);
[MSE5] = Error_Measures(E5_vectors, quant_E5_vectors);
[MSE6] = Error_Measures(E6_vectors, quant_E6_vectors);
[MSE7] = Error_Measures(E7_vectors, quant_E7_vectors);
[MSE8] = Error_Measures(E8_vectors, quant_E8_vectors);
[MSE9] = Error_Measures(non_edge_vectors, quant_ne_vectors);
```

```
MSE = [MSE1, MSE2, MSE3, MSE4, MSE5, MSE6, MSE7, MSE8, MSE9, MSE];

return
```

Edge separate LVQ

```
function [reconstructed_image, MSE, sizeEs] = VQ_edges_sep_LVQ( testimage, trainimage,
xsize, ysize,codevectors,threshold, Vthreshold, Np)

% calculates the codebook using LVQ for 4 by 4 blocks

% Inputs:  image, xsize and ysize which give the block size, and the number

% of codevectors used for the non edge and edge vectors and Np is the

% minimum number of pixels needed classified as non edges and edges

% Outputs:  reconstructed image, the error measures given by the MSE and size of
each

% edge class

% Difference image

%Vthreshold = 130 % this threshold gives

% classify each pixel

[edge_array,m, pixel_classes] = classifier(testimage, threshold);

% convert pixel class array to block-vectors

class_vectors = blocks2vectors(pixel_classes, xsize, ysize);

% convert image array to block-vectors

image_vectors = blocks2vectors(testimage, xsize, ysize);

% classify each block as an edge or non-edge block

[block_classes] = block_classifier(class_vectors, pcnt0);

% check whether blocks have been correctly identified as edge blocks using

% the variance and then get indices for edge and non-edge blocks

% variance threshold (take block vectors with variance above vthreshold to

% be true edge blocks and get new block classes

[edge_blocks, non_edge_blocks, block_classes] = edge_check(image_vectors,

block_classes, Vthreshold); length(non_edge_blocks);

% Separte edge blocks to directions N, S, NW, SE, W, E, SW, NE

[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes);
```

```
sizeEs=[length(E1),length(E2),length(E3),length(E4),length(E5),
length(E6),length(E7),length(E8)];
sizeEs = [sizeEs, length(non_edge_blocks)];
%******************classify pixels in training set ***********************%
[E1_v, E2_v, E3_v, E4_v, E5_v, E6_v, E7_v, E8_v, E9_v] = classify_train2(trainimage,
threshold, Vthreshold, 4, 4, Np);
%For quantization epochs = 70;
% set number of iterations
%exponent = 1.1;
%****************** Quantization of non-edge blocks*********************%
codevectorsNE = codevectors(9);
non_edge_vectors = image_vectors(:, non_edge_blocks);
%select training set
t_vectors = E9_v;
% create initial codebook (all codevectors on diagonal)
[p, q] = size(t_vectors);
steps = ceil(q/codevectorsNE);
init_codebook = t_vectors(:, 1:steps:q);
% codebook creation using LVQ
[codebook] = clusternet(t_vectors,init_codebook, 70,0);
% gives the cluster label for each non-edge vector
ne_cluster_labels = quantize_image_vectors(non_edge_vectors, codebook);
%reconstructs non-edge vectors using codebook
quant_ne_vectors = double(reconstruct_vectors(ne_cluster_labels, codebook));
%*************************************************************************%
%************************Quantization of edge blocks *********************%
% North edges
codevectors1 = codevectors(1);
E1_vectors = image_vectors(:, E1);
```

```
%select training set
t1_vectors = E1_v;
% create initial codebook
[p1, q1] = size(t1_vectors);
steps1 = ceil(q1/codevectors1);
init_codebook1 = t1_vectors(:, 1:steps1:q1);
% codebook creation using fuzzy c-means
[codebook1] = clusternet(t1_vectors,init_codebook1, 70,0);
% gives the cluster label for each north edge vector
E1_cluster_labels = quantize_image_vectors(E1_vectors, codebook1);
%reconstructs north vectors using codebook
quant_E1_vectors = double(reconstruct_vectors(E1_cluster_labels, codebook1));
%**************************************************************************%
%**************************************************************************%
codevectors2 = codevectors(2);
E2_vectors = image_vectors(:, E2);
%select training set
t2_vectors = E2_v;
% create initial codebook
%dim = xsize * ysize;
%init_codebook2 = rand(dim, codevectors2)*255;
[p2, q2] = size(t2_vectors);
steps2 = ceil(q2/codevectors2);
init_codebook2 = t2_vectors(:, 1:steps2:q2);
% codebook creation using LVQ
[codebook2] = clusternet(t2_vectors,init_codebook2, 70, 0);
% gives the cluster label for each south edge vector
E2_cluster_labels = quantize_image_vectors(E2_vectors, codebook2);
%reconstructs south edge vectors using codebook
```

```
quant_E2_vectors = double(reconstruct_vectors(E2_cluster_labels, codebook2));
%************************************************************************%
%************************************************************************%
codevectors3 = codevectors(3);
E3_vectors = image_vectors(:, E3);
%select training set
t3_vectors = E3_v;
% create initial codebook
%dim = xsize * ysize;
%init_codebook3 = rand(dim, codevectors3)*255;
[p3, q3] = size(t3_vectors);
steps3 = ceil(q3/codevectors3);
init_codebook3 = t3_vectors(:, 1:steps3:q3);
% codebook creation using fuzzy c-means
[codebook3] = clusternet(t3_vectors,init_codebook3, 70,0);
% gives the cluster label for each NW edge vector
E3_cluster_labels = quantize_image_vectors(E3_vectors, codebook3);
%reconstructs NW edge vectors using codebook
quant_E3_vectors = double(reconstruct_vectors(E3_cluster_labels, codebook3));
%************************************************************************%
%************************************************************************%
codevectors4 = codevectors(4);
E4_vectors = image_vectors(:, E4);
%select training set
t4_vectors = E4_v;
%initial codebook creation
[p4, q4] = size(t4_vectors);
steps4 = ceil(q4/codevectors4);
init_codebook4 = t4_vectors(:, 1:steps4:q4);
```

```
% codebook creation using fuzzy c-means
[codebook4] = clusternet(t4_vectors,init_codebook4, 70,0);
% gives the cluster label for each SE edge vector
E4_cluster_labels = quantize_image_vectors(E4_vectors, codebook4);
%reconstructs SE edge vectors using codebook
quant_E4_vectors = double(reconstruct_vectors(E4_cluster_labels, codebook4));
%**************************************************************************%
%**************************************************************************%
codevectors5 = codevectors(5);
E5_vectors = image_vectors(:, E5);
%select training set
t5_vectors = E5_v;
% create initial codebook
[p5, q5] = size(t5_vectors);
steps5 = ceil(q5/codevectors5);
init_codebook5 = t5_vectors(:, 1:steps5:q5);
% codebook creation using fuzzy c-means
[codebook5] = clusternet(t5_vectors,init_codebook5, 70,0);
% gives the cluster label for each W edge vector E5_cluster_labels =
quantize_image_vectors(E5_vectors, codebook5);
%reconstructs W edge vectors using codebook quant_E5_vectors =
double(reconstruct_vectors(E5_cluster_labels, codebook5));
%**************************************************************************%
%**************************************************************************%
codevectors6 = codevectors(6);
E6_vectors = image_vectors(:, E6);
%select training set
t6_vectors = E6_v;
% create initial codebook
```

```
%dim = xsize * ysize;
%init_codebook6 = rand(dim, codevectors6)*255;
[p6, q6] = size(t6_vectors);
steps6 = ceil(q6/codevectors6);
init_codebook6 = t6_vectors(:, 1:steps6:q6);
% codebook creation using LVQ
[codebook6] = clusternet(t6_vectors,init_codebook6, 70, 0);
% gives the cluster label for each E edge vector
E6_cluster_labels = quantize_image_vectors(E6_vectors, codebook6);
%reconstructs E edge vectors using codebook
quant_E6_vectors = double(reconstruct_vectors(E6_cluster_labels, codebook6));
%**************************************************************************%
%**************************************************************************%
codevectors7 = codevectors(7);
E7_vectors = image_vectors(:, E7);
%select training set
t7_vectors = E7_v;
%initial codebook creation
%dim = xsize * ysize;
%init_codebook7 = rand(dim, codevectors7)*255;
[p7, q7] = size(t7_vectors);
steps7 = ceil(q7/codevectors7);
init_codebook7 = t7_vectors(:, 1:steps7:q7);
% codebook creation using fuzzy c-means
[codebook7] = clusternet(t7_vectors,init_codebook7, 70,0);
% gives the cluster label for each SW edge vector
E7_cluster_labels = quantize_image_vectors(E7_vectors, codebook7);
%reconstructs SW edge vectors using codebook
quant_E7_vectors = double(reconstruct_vectors(E7_cluster_labels, codebook7));
```

```
%*************************************************************************%
%*************************************************************************%
codevectors8 = codevectors(8);
E8_vectors = image_vectors(:, E8);
%select training set
t8_vectors = E8_v;
%dim = xsize * ysize;
%init_codebook8 = rand(dim, codevectors8)*255;
[p8, q8] = size(t8_vectors);
steps8 = ceil(q8/codevectors8);
init_codebook8 = t8_vectors(:, 1:steps8:q8);
% codebook creation using LVQ
[codebook8] = clusternet(t8_vectors,init_codebook8,70,0);
% gives the cluster label for each NE edge vector
E8_cluster_labels = quantize_image_vectors(E8_vectors, codebook8);
%reconstructs NE edge vectors using codebook
quant_E8_vectors = double(reconstruct_vectors(E8_cluster_labels, codebook8));
%*************************************************************************%
%*************************** Reconstruction ***************************%
% get reconstruced image block vectors
[r,s] = size(image_vectors);
recon_i_vectors = zeros(r,s);
recon_i_vectors(:, non_edge_blocks) = quant_ne_vectors;
recon_i_vectors(:, E1)= quant_E1_vectors;
recon_i_vectors(:, E2)= quant_E2_vectors;
recon_i_vectors(:, E3)= quant_E3_vectors;
recon_i_vectors(:, E4)= quant_E4_vectors;
recon_i_vectors(:, E5)= quant_E5_vectors;
recon_i_vectors(:, E6)= quant_E6_vectors;
```

```
recon_i_vectors(:, E7)= quant_E7_vectors;
recon_i_vectors(:, E8)= quant_E8_vectors;
% convert reconstructed image vectors to blocks then to the reconstructed
% image
[reconstructed_image] = vectors2image(recon_i_vectors, testimage, xsize);
%************************** Difference image**************************%%%
[m,n] = size(reconstructed_image);
diff_image = 255 - abs(testimage(1:m, 1:n)-reconstructed_image);
figure()
imagesc(diff_image, [0, 255])
colormap(gray)
%****************** Error measures ************************************%
[MSE] = Error_Measures(testimage, reconstructed_image);
[MSE1] = Error_Measures(E1_vectors, quant_E1_vectors);
[MSE2] = Error_Measures(E2_vectors, quant_E2_vectors);
[MSE3] = Error_Measures(E3_vectors, quant_E3_vectors);
[MSE4] = Error_Measures(E4_vectors, quant_E4_vectors);
[MSE5] = Error_Measures(E5_vectors, quant_E5_vectors);
[MSE6] = Error_Measures(E6_vectors, quant_E6_vectors);
[MSE7] = Error_Measures(E7_vectors, quant_E7_vectors);
[MSE8] = Error_Measures(E8_vectors, quant_E8_vectors);
[MSE9] = Error_Measures(non_edge_vectors, quant_ne_vectors);
MSE = [MSE1, MSE2, MSE3, MSE4, MSE5, MSE6, MSE7, MSE8, MSE9, MSE];
```

Edge separation using cross validation on 5 images

```
function [quant_image, MSE, SNR, sizeEs] = VQ_edges_sep_LVQ_C( images, xsize,
ysize,codevectors,threshold, Vthreshold, Np)
% performs CVQ with 5 images, using 5 fold cross validation
trainimage(:,:,1) = [images(:,:,2), images(:,:,3), images(:,:,4), images(:,:,5)];
trainimage(:,:,2) = [images(:,:,1), images(:,:,3), images(:,:,4), images(:,:,5)];
```

```
trainimage(:,:,3) = [images(:,:,1), images(:,:,2), images(:,:,4), images(:,:,5)];
trainimage(:,:,4) = [images(:,:,1), images(:,:,2), images(:,:,3), images(:,:,5)];
trainimage(:,:,5) = [images(:,:,1), images(:,:,2), images(:,:,3), images(:,:,4)];
for k= 1:  5
[quant_image(:,:,k), MSE(:,k), SNR(:,k), sizeEs(:,k)] = VQ_edges_sep6N(images(:,:,k),
trainimage(:,:,k), 4, 4,codevectors, 50, 130, 10);
k = k+1;
end
```

Edge separation using CM using 5 images

```
function [quant_image, MSE, SNR, sizeEs] = VQ_edges_sep_CM_C images, xsize,
ysize,codevectors,threshold, Vthreshold, pcnt0)
trainimage(:,:,1) = [images(:,:,2), images(:,:,3), images(:,:,4), images(:,:,5)];
trainimage(:,:,2) = [images(:,:,1), images(:,:,3), images(:,:,4), images(:,:,5)];
trainimage(:,:,3) = [images(:,:,1), images(:,:,2), images(:,:,4), images(:,:,5)];
trainimage(:,:,4) = [images(:,:,1), images(:,:,2), images(:,:,3), images(:,:,5)];
trainimage(:,:,5) = [images(:,:,1), images(:,:,2), images(:,:,3), images(:,:,4)];
for k= 1:  5
[quant_image(:,:,k), MSE(:,k), SNR(:,k), sizeEs(:,k)] = VQ_edges_sep5N(images(:,:,k),
trainimage(:,:,k), 4, 4,codevectors, 50, 130, 10);
k = k+1;
end
```

Edge separation with FCM

```
function [quant_image, MSE, SNR, sizeEs] = VQ_edges_sep_NN( images, xsize,
ysize,codevectors,threshold, Vthreshold, pcnt0)
trainimage(:,:,1) = [images(:,:,2), images(:,:,3), images(:,:,4), images(:,:,5)];
trainimage(:,:,2) = [images(:,:,1), images(:,:,3), images(:,:,4), images(:,:,5)];
trainimage(:,:,3) = [images(:,:,1), images(:,:,2), images(:,:,4), images(:,:,5)];
trainimage(:,:,4) = [images(:,:,1), images(:,:,2), images(:,:,3), images(:,:,5)];
trainimage(:,:,5) = [images(:,:,1), images(:,:,2), images(:,:,3), images(:,:,4)];
for k= 1:  5
```

```
[quant_image(:,:,k), MSE(:,k), SNR(:,k), sizeEs(:,k)] = VQ_edges_sep5N(images(:,:,k),
trainimage(:,:,k), 4, 4,codevectors, 50, 130, 10);
k = k+1;
end
```

Fuzzy c-means

```
function [quant_image, MSE, sizeEs] = VQ_fcm_eN( imagetest, imagetrain, xsize,
ysize,codevectors,threshold, Vthreshold, Np)
% calculates the codebook using fuzzy cmeans 4 by 4 blocks
% Inputs:  image, xsize and ysize which give the block size, and the number
% of codevectors used for the non edge and edge vectors and Np is the
% percentage of pixels classified as non edges and edges in 8 directions
% Outputs:  reconstructed image, the error measures given by the MSE
% Difference image
%Vthreshold = 130
% this threshold gives
% classify each pixel
[edge_array,m, pixel_classes] = classifier(imagetest, threshold);
% convert pixel class array to block-vectors
class_vectors = blocks2vectors(pixel_classes, xsize, ysize);
% convert image array to block-vectors
image_vectors = blocks2vectors(imagetest, xsize, ysize);
% classify each block as an edge or non-edge block
[block_classes] = block_classifier(class_vectors, Np);
% check whether blocks have been correctly identified as edge blocks using
% the variance and then get indices for edge and non-edge blocks
% variance threshold (take block vectors with variance above vthreshold to
% be true edge blocks and get new block classes
[edge_blocks, non_edge_blocks, block_classes] = edge_check(image_vectors,
block_classes, Vthreshold); length(non_edge_blocks);
% Separate edge blocks to directions N, S, NW, SE, W, E, SW, NE
```

```
[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes);
sizeEs=[length(E1),length(E2),length(E3),length(E4),length(E5),length(E6),
length(E7),length(E8)];
sizeEs = [sizeEs, length(non_edge_blocks)];
E1_vectors = image_vectors(:, E1);
E2_vectors = image_vectors(:, E2);
E3_vectors = image_vectors(:, E3);
E4_vectors = image_vectors(:, E4);
E5_vectors = image_vectors(:, E5);
E6_vectors = image_vectors(:, E6);
E7_vectors = image_vectors(:, E7);
E8_vectors = image_vectors(:, E8);
NE_vectors = image_vectors(:, non_edge_blocks);
%select training set
t_vectors = imagetrain;
% ********************create initial codebook*****************************%
ic = rand(1, codevectors)*255;
dim = xsize * ysize;
init_codebook = repmat(ic, dim, 1);
%**********************codebook creation ********************************%
epochs = 70;
% set number of iterations
exponent = 1.1;
% codebook creation using fuzzy c-means
[partmat,codebook] = fuzzy_cmeans_seq2(t_vectors,init_codebook,exponent,epochs);
% ********************quantization using codebook************************%
% gives the cluster label for each image vector
compressed_image_vectors = quantize_image_vectors(image_vectors, codebook);
%fuzzy c-means
```

```
%*********************reconstruction of compressed image*******************%
quant_image_vectors = double(reconstruct_vectors(compressed_image_vectors, codebook));
quant_image_E1 = quant_image_vectors(:, E1);
quant_image_E2 = quant_image_vectors(:, E2);
quant_image_E3 = quant_image_vectors(:, E3);
quant_image_E4 = quant_image_vectors(:, E4);
quant_image_E5 = quant_image_vectors(:, E5);
quant_image_E6 = quant_image_vectors(:, E6);
quant_image_E7 = quant_image_vectors(:, E7);
quant_image_E8 = quant_image_vectors(:, E8);
quant_image_NE = quant_image_vectors(:, non_edge_blocks);
quant_image = vectors2image(quant_image_vectors, imagetest, xsize);
%*************************** Difference image*****************************%%%
[m,n] = size(quant_image);
diff_image = 255 - abs(imagetest(1:m, 1:n)- quant_image);
figure()
imagesc(diff_image, [0, 255])
colormap(gray)
%****************** Error measures ***************************************%
[MSE]= Error_Measures(imagetest, quant_image);
[MSE1] = Error_Measures(E1_vectors, quant_image_E1);
[MSE2] = Error_Measures(E2_vectors, quant_image_E2);
[MSE3] = Error_Measures(E3_vectors, quant_image_E3);
[MSE4] = Error_Measures(E4_vectors, quant_image_E4);
[MSE5] = Error_Measures(E5_vectors, quant_image_E5);
[MSE6] = Error_Measures(E6_vectors, quant_image_E6);
[MSE7] = Error_Measures(E7_vectors, quant_image_E7);
[MSE8] = Error_Measures(E8_vectors, quant_image_E8);
[MSE9] = Error_Measures(NE_vectors, quant_image_NE);
```

```
MSE = [MSE1, MSE2, MSE3, MSE4, MSE5, MSE6, MSE7, MSE8, MSE9, MSE];
return
```

Fuzzy c-means for varying codebook sizes

```
function [quant_image, MSE, SNR, sizeEs] = VQ_fcmN( imagetest,imagetrain, xsize,
ysize,codevectors,threshold, Vthreshold, pcnt0)
for k= 1:  length(codevectors)
[quant_image(:,:,k), MSE(k, :), SNR(k, :), sizeEs(k,:)]  = VQ_fcm_eN
( imagetest,imagetrain, xsize, ysize,codevectors(k),threshold, Vthreshold, pcnt0);
k = k+1;
end
```

LVQ

```
function [quant_image, MSE, SNR, sizeEs, codebook] = VQ_lvq_eN
( imagetest, imagetrain, xsize, ysize,codevectors,threshold, Vthreshold, Np)
% calculates the codebook using fuzzy cmeans for xsize by ysize blocks
% Inputs:  image, xsize and ysize which give the block size, and the number
% of codevectors used for the non edge and edge vectors and Np is the
% minimum number of pixels classified as non edges and edges in 8 directions
% Outputs:  reconstructed image, the error measures given by the MSE
% Difference image
%Vthreshold = 130
% this threshold gives
% classify each pixel
%threshold = 30;
[edge_array,m, pixel_classes] = classifier(imagetest, threshold);
% convert pixel class array to block-vectors
class_vectors = blocks2vectors(pixel_classes, xsize, ysize);
% convert image array to block-vectors
image_vectors = blocks2vectors(imagetest, xsize, ysize);
% classify each block as an edge or non-edge block
```

```
[block_classes] = block_classifier(class_vectors, Np);
% check whether blocks have been correctly identified as edge blocks using
% the variance and then get indices for edge and non-edge blocks
% variance threshold (take block vectors with variance above vthreshold to
% be true edge blocks and get new block classes
[edge_blocks, non_edge_blocks, block_classes] = edge_check
(image_vectors, block_classes, Vthreshold);
length(non_edge_blocks);
% Separate edge blocks to directions N, S, NW, SE, W, E, SW, NE
[E1, E2,E3, E4, E5, E6, E7, E8] = class_edge_blocks2( block_classes);
sizeEs = length(E1),length(E2),length(E3),length(E4),length(E5),
length(E6),length(E7),length(E8)];
sizeEs = [sizeEs, length(non_edge_blocks)];
E1_vectors = image_vectors(:, E1);
E2_vectors = image_vectors(:, E2);
E3_vectors = image_vectors(:, E3);
E4_vectors = image_vectors(:, E4);
E5_vectors = image_vectors(:, E5);
E6_vectors = image_vectors(:, E6);
E7_vectors = image_vectors(:, E7);
E8_vectors = image_vectors(:, E8);
NE_vectors = image_vectors(:, non_edge_blocks);
%select training set
t_vectors = imagetrain;
% ********************create initial codebook*****************************%
[p, q] = size(t_vectors);
steps = ceil(q/codevectors(1));
init_codebook = t_vectors(:, 1:steps:q);
%**********************codebook creation ********************************%
```

```
epochs = 70;
% set number of iterations
codebook = double(clusternet(t_vectors,init_codebook, epochs, 0));
% ********************quantization using codebook************************%
% gives the cluster label for each image vector
compressed_image_vectors = quantize_image_vectors(image_vectors, codebook);
%fuzzy c-means
%********************reconstruction of compressed image******************%
quant_image_vectors = double(reconstruct_vectors(compressed_image_vectors, codebook));
quant_image_E1 = quant_image_vectors(:, E1);
quant_image_E2 = quant_image_vectors(:, E2);
quant_image_E3 = quant_image_vectors(:, E3);
quant_image_E4 = quant_image_vectors(:, E4);
quant_image_E5 = quant_image_vectors(:, E5);
quant_image_E6 = quant_image_vectors(:, E6);
quant_image_E7 = quant_image_vectors(:, E7);
quant_image_E8 = quant_image_vectors(:, E8);
quant_image_NE = quant_image_vectors(:, non_edge_blocks);
quant_image = vectors2image(quant_image_vectors, imagetest, xsize);
%************************************************************************%
%************************** Difference image*************************%%%
[m,n] = size(quant_image);
diff_image = 255 - abs(imagetest(1:m, 1:n)- quant_image);
figure()
imagesc(diff_image, [0, 255])
colormap(gray)
%****************** Error measures ***************************************%
[MSE]= Error_Measures(imagetest, quant_image);
[MSE1] = Error_Measures(E1_vectors, quant_image_E1);
```

```
[MSE2] = Error_Measures(E2_vectors, quant_image_E2);
[MSE3] = Error_Measures(E3_vectors, quant_image_E3);
[MSE4] = Error_Measures(E4_vectors, quant_image_E4);
[MSE5] = Error_Measures(E5_vectors, quant_image_E5);
[MSE6] = Error_Measures(E6_vectors, quant_image_E6);
[MSE7] = Error_Measures(E7_vectors, quant_image_E7);
[MSE8] = Error_Measures(E8_vectors, quant_image_E8);
[MSE9] = Error_Measures(NE_vectors, quant_image_NE);
MSE = [MSE1, MSE2, MSE3, MSE4, MSE5, MSE6, MSE7, MSE8, MSE9, MSE];
return
```

LVQ for varying codebook sizes

```
function [quant_image, MSE, SNR, sizeEs] = VQ_lvqN( imagetest,imagetrain,
xsize, ysize,codevectors,threshold, Vthreshold, Np)
for k= 1:  length(codevectors)
[quant_image(:,:,k), MSE(k, :), SNR(k, :), sizeEs(k,:)]  = VQ_lvq_eN
( imagetest,imagetrain, xsize, ysize,codevectors(k),threshold, Vthreshold, Np);
k = k+1;
end
```