http://researchspace.auckland.ac.nz

ResearchSpace@Auckland

Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. http://researchspace.auckland.ac.nz/feedback

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form.

The University of Auckland

School of Engineering

The Application of Support Vector Machines to Compression of Digital Images

Jonathan Robinson

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

Doctor of Philosophy

February 2004

Abstract

Methods exploring the application of neural networks to still image compression are detailed in both the spatial and frequency domains. In particular the sparse properties of Support Vector Machine (SVM) learning are exploited in the compression algorithms.

A classic radial basis function (RBF) neural network requires that the topology of the network be defined before training. An SVM has the property that it will choose the minimum number of training points to use as centres of the Gaussian kernel functions. It is this property that is exploited as the basis for image compression algorithms presented in this thesis.

Several novel algorithms are developed applying SVM learning to both directly model the colour surface and model transform coefficients after the surface has been transformed into the frequency domain. It is demonstrated that compression is more efficient in frequency space.

The Discrete Cosine Transform (DCT) is used to transform the colour surface into the frequency domain. A counter-intuitive result is shown where mapping the DCT coefficients to a 1-dimensional function for SVM modelling produces better results than SVM modelling of the 2-dimensional transform surface.

Results are presented in comparison to the JPEG image compression algorithm. In the frequency domain, results are superior to that of JPEG. For example, the quality of the 'Lena' image compressed 63:1 for JPEG is slightly worse quality than the same image compressed 192:1 with the RKi-1 algorithm presented in this thesis. Due to the commercial value of the algorithms detailed in this thesis, a patent has been filed.

Acknowledgements

I would like to thank my supervisor Assoc. Prof. Vojislav Kecman for his support and guidance during the course of this thesis. I would also like to thank my family for their love, support and encouragement. In particular, I would like to thank my wife Simone for proof-reading the text and my daughter Claudia who graciously agreed to allow her photograph to be used as a test image in this thesis.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**
1.1 Spatial and Frequency Domains 3
1.2 Statement of the Problem 9
1.3 Previous Work 9
1.4 Support Vector Machines 11
1.5 Research Objectives and Contributions 13
1.6 Organisation of this Thesis 14

2 LOMON: LOcal MOdelling Network **16**
2.1 Training 19
2.2 Results 24
2.2.1 Effect of the Gaussian Shape Parameter 28
2.3 LOMON With Multiple Gaussians 29
2.3.1 Four-Gaussian LOMON 35
2.4 Parameters Required to be Transmitted 38
2.5 Discussion 41

3 Support Vector Machine Learning **43**
3.1 Introduction 43
3.2 Classification by SVM 44
3.2.1 Optimal Canonical Separating Hyperplane 46
3.2.2 The Maximal Margin 48
3.2.3 Solution by Lagrangian Multipliers 50
3.2.4 Non-separable Classification 53
3.3 Function Approximation by SVM 56
3.4 Summary 62

4 SVM Modelled Colour Surface 63
4.1 SVM Model . . . 63
4.1.1 Implementation . . . 64
4.1.2 Non-Negative Least Squares (NNLS) . . . 68
4.1.3 Transformation of the QP problem to NNLS . . . 69
4.2 Results . . . 71
4.3 Implementation . . . 76
4.3.1 Run Length and Huffman Encoding . . . 79
4.3.2 Encoding the Biases . . . 83
4.3.3 Encoding Weights . . . 84
4.4 Discussion . . . 87

5 LOMON combined with SVM 88
5.1 Introduction . . . 88
5.2 The LOMON-SVM Algorithm . . . 88
5.3 Results . . . 92
5.4 Discussion . . . 92

6 Compression in the Frequency Domain: The RKi-1 Algorithm 94
6.1 Introduction . . . 94
6.2 Discrete Cosine Transform . . . 95
6.2.1 Transformation of the DCT Matrix to 1-D . . . 97
6.3 Combining SVM with DCT . . . 98
6.3.1 The 'Inversion' bit . . . 102
6.4 Implementation . . . 104
6.4.1 Encoding Data for Storage . . . 106
6.4.2 Encoding the DC Coefficients . . . 107
6.5 Results . . . 110
6.5.1 Performance . . . 110
6.5.2 Effect of Reducing the Number of Inversion Bits . . . 113
6.6 Discussion . . . 116

7 The RKi-2 Compression Algorithm 117
7.1 2-Dimensional SVM . . . 117
7.2 Results . . . 120
7.3 Discussion . . . 124

8 The RKi-3 Algorithm 126
8.1 Results . . . 128
8.2 Discussion . . . 134

9 Summary and Discussion 135
9.1 Further Work . . . 136

Appendix I: Overview of the JPEG Compression Standard **139**

Appendix II: Contents of the Accompanying CD **141**

References **141**

List of Figures

1.1 TOP: Original 512×512 'Lena' image. BOTTOM: Rotated 512×512 'Lena' image. 5
1.2 Rotated 512×512 'Lena' image illustrating the complexity of the surface to be modeled. 6
1.3 TOP: Blocking effects seen in the JPEG compression algorithm at high compression (ratio = 27:1). BOTTOM: Smearing effects seen in wavelet compression at high compression (ratio = 114:1) . 8
1.4 Feed-forward RBF network 10
1.5 Two class classification problem 12
1.6 Example of a 1-dimensional SVM 14

2.1 The LOMON algorithm where $\epsilon =$ `max_image_error` is the maximum error in a pixel. 17
2.2 LOMON Strategy 18
2.3 Largest pixel error during LOMON training (No Bias) 22
2.4 LOMON trained on Lena image (no bias) 23
2.5 Largest pixel error during LOMON training (With Bias) 24
2.6 256×256 Lena image trained using LOMON 26
2.7 400×400 Claudia image trained using LOMON 27
2.8 Number of blocks versus Gaussian shape parameter for LOMON . 29
2.9 Definition of splitting in LOMON for two Gaussians 30
2.10 256×256 Lena image trained using LOMON (two Gaussians per block) 31
2.11 256×256 Lena image showing blocks generated by LOMON (two Gaussians per block) 32
2.12 Number of blocks versus Gaussian width for two Gaussians 34
2.13 LOMON results for two Gaussians per block 35
2.14 Definition of splitting in LOMON into four Gaussians 36
2.15 256×256 Lena image trained using LOMON (four Gaussians per block) 37
2.16 256×256 Lena image showing blocks generated by LOMON (four Gaussians per block) 39
2.17 Number of blocks versus Gaussian width or four Gaussian LOMON 40

3.1 Example of a 1-dimensional SVM 45
3.2 Maximal Margin 46
3.3 Optimal canonical separating hyperplane in 1-dimension 47
3.4 Maximal margin 48
3.5 Parameters used in the computation of the maximal margin . . . 49
3.6 Non-separable binary classification 54
3.7 Non-separable binary classification with slack variables 54
3.8 The parameters used in (1-dimensional) support vector regression (from [1]). 58

4.1 Radial Basis Function network 65
4.2 256×256 Lena Image with no bias 72
4.3 Surface patch produced by SVM learning 73
4.4 Plot of number of support vectors versus Gaussian width 74
4.5 Plot of SNR versus compression ratio for the 400×400 Claudia image 75
4.6 Plot of SNR versus compression ratio for the 512×512 Lena image 75
4.7 256×256 Lena Image with bias 76
4.8 512×512 Lena Image with bias 77
4.9 400×400 Claudia image with bias 78
4.10 Huffman coding algorithm 82
4.11 Huffman encoding example 83
4.12 Huffman table data storage 84
4.13 Too few quantisation levels 86
4.14 Encoding the Weights 87

5.1 Modified LOMON Strategy 90
5.2 The LOMON algorithm where `max_image_error` is the maximum error in a pixel. 91
5.3 Lena image using LOMON combined with SVM learning 93

6.1 The Discrete Cosine Transform 97
6.2 Zig-zag mapping of DCT coefficients 98
6.3 The RKi-1 Algorithm 99
6.4 1-dimensional plot of DCT coefficients 100
6.5 DCT real values as input to the SVM 101
6.6 DCT absolute magnitude as input to the SVM 103
6.7 Input block to SVM and output block 106
6.8 XML export file 108
6.9 Plot of SNR for RKi-1 and JPEG with Lena Image 112
6.10 Plot of SNR for RKi-1 and JPEG with Claudia Image 112
6.11 Effect of reducing the number of inversion bits for the 400×400 Claudia image 113

6.12 Subjective comparison of RKi-1 and JPEG for the 512×512 Lena image . . . 114
6.13 Subjective comparison of RKi-1 and JPEG for the 400×400 Claudia image . . . 115

7.1 The RKi-2 Algorithm . . . 119
7.2 Plot of SNR versus compression ratio for the 400×400 Claudia image . . . 121
7.3 Plot of SNR versus compression ratio for the 512×512 Lena image 121
7.4 Subjective comparison of RKi-2 and JPEG for the 512×512 Lena image . . . 122
7.5 Subjective comparison of RKi-2 and JPEG for the 400×400 Claudia image . . . 123
7.6 Influence of Gaussians in RKi-1 and RKi-2 . . . 125

8.1 Shuffling DCT coefficients . . . 127
8.2 Reduction of a 'Super-Block' in RKi-3 . . . 130
8.3 Subjective comparison of RKi-3 and JPEG for the 512×512 Lena image . . . 131
8.4 Subjective comparison of RKi-3 and JPEG for the 384×384 Claudia image . . . 132
8.5 Plot of SNR versus compression ratio for the 384×384 Claudia image . . . 133
8.6 Plot of SNR versus compression ratio for the 512×512 Lena image 133

9.1 Plot of SnR versus compression ratio for the 512×512 Lena image for all algorithms . . . 136
9.2 Plot of SnR versus compression ratio for the 400×400 Claudia image for all algorithms . . . 137

List of Tables

4.1 Huffman encoding biases 85

6.1 Huffman encoding DC values 109

9.1 The Y component quantisation table 140

Chapter 1

Introduction

"A picture is worth a thousand words" is an often used expression. However if the picture is required to be stored in an electronic retrieval system then it is likely that many more than a thousand words are required. Consider an image 512 pixels[1] in height by 512 pixels in width. The image contains 262,144 pixels. In computer terms the most basic piece of information is an individual 0 or 1 – a bit. Eight bits comprise a byte and two bytes comprise a 'word'. Thus if our 512 X 512 image is a gray-scale image (ie black and white) with 256 different gray levels then 262,144 bytes are required, or a little over 131,000 words. If the image is a full colour image with 256 levels each for red, green and blue then three bytes are required for each pixel and a total of around 400,000 words are needed to store the image (786,432 bytes).

A standard floppy disk stores 1.38 megabytes of data. A full colour (24-bit, approx. 16 million colours) image of dimension 512 X 512 would occupy 53% of this if stored uncompressed. Now consider a one second video clip with the standard 24 frames per second. To store 24 frames uncompressed of a 24 bit colour video stream of size 512×512 requires 19 megabytes of storage space – thirteen floppy disks would be needed to store this one second clip. If stored

[1]The term 'Pixel' is loosely derived from 'Picture Element'. It refers to an individual point in an image.

raw, then a standard compact disk (CD) with a capacity of 650 megabytes could only store about 35 seconds of video. Even a digital versatile/video disk (DVD) would only be able to store around 7 minutes of video. The one second video clip would take around 45 minutes to download on a 56k modem.

The FBI is in the process of converting around 200 million inked fingerprint cards into electronic format [2]. Fingerprint cards are scanned at 500 dots per inch and 256 level gray scale. This results in each fingerprint card occupying about 10 megabytes of data. Thus the existing fingerprint database would require around 2000 terabytes (2^{15} bytes) if stored raw. Coupled with the fact that 50,000 new fingerprint cards are added daily to the database, it is apparent why compression is required.

Fortunately an image contains spatial redundancy (a video stream also contains temporal redundancy). These redundancies, together with the way the human eye perceives images are exploited in image compression algorithms.

The rapid growth of the internet has created demand for faster and more efficient data compression algorithms. The internet is being used more and more to deliver services such as video on demand to devices such as personal computers, PDAs and mobile phones. A recent edition of IEEE's 'Computer' magazine [3] was devoted to 'Media on Demand' emphasising the importance of delivering multimedia to a variety of devices.

Data compression algorithms can be divided into two categories: Lossy and Lossless. The key difference between these two categories is that lossless compression produces a 100% accurate duplicate of the original data when decompressed whereas lossy algorithms produce an approximation of the original data. Lossless compression algorithms are used where data cannot be lost. The decompressed data must be identical to the original. The 'deflate' algorithm [4, 5] (used in the 'WinZip' program) is an example of a lossless compression algorithm. Lossy algorithms are generally engineered so that they take advantage of particular

aspects of and patterns in the data being compressed to minimise the apparent data loss. For example the FBI wavelet image compression algorithm [2, 6, 7] is used to compress images representing fingerprints. The algorithm takes advantage of the nature of a finger print image and produces very good compression for this type of image. This does not extend to the general case of any image.

Another example of a lossy image compression algorithm is JPEG standard [8, 9]. The JPEG algorithm exploits the way the human eye perceives images by concentrating the data loss to higher frequencies in the image which are less noticeable to the human eye than lower frequencies. This works well for photographic images but does not work as well for drawing (cartoon) style images as sharp lines tend to be blurred. The GIF (Graphical Image Format) [10, 11] is another example of a commonly used lossy data compression algorithm[2]. GIF works well for images of drawings or cartoons but does not perform as well as JPEG for photographic images. GIF is also limited to 256 colours whereas JPEG supports 24-bit colour (approx. 16 million colours)

1.1 Spatial and Frequency Domains

The term 'spatial domain' applies to an image colour surface where each individual pixel has a position in an (x_1, x_2) plane. Mathematically an image can be viewed as a 3-dimensional image surface. The x_1- and x_2- coordinates identify the position of the pixel and the $y-$ coordinate represents the colour value of the pixel. Figures 1.1 and 1.2 show the benchmark 'Lena' image and the associated 3-d colour surface[3]. These images illustrate the complexity of a typical colour surface. There are many sharp peaks and troughs which need to be accurately modeled in a recovered image for the recovered image to be an acceptable

[2]GIF is actually a file format utilising a compression algorithm patented by Unisys. The term 'GIF' normally refers to both the file format and the algorithm.

[3]The 'Lena' image [12] is the defacto standard image used to compare image processing techniques.

representation of the original.

The colour surface can be expressed as:

$$y = f(x_1, x_2) \tag{1.1}$$

where y_i is the colour of the pixel at point (x_{1i}, x_{2i}) in the image and f is the function that defines the image. It is the purpose of all image compression algorithms to somehow store/transmit f. It can be seen in figure 1.1 that some pixels have similar colours to bordering pixels. This is spatial redundancy. Spatial redundancy in images is exploited in Run Length Encoding (RLE) which is used in the popular BMP (BitMaP) image compression algorithm.[4] For example if an image contains a row of pixels with colours:

$$10\ 11\ 13\ 13\ 13\ 13\ 13\ 13\ 13\ 11\ 11\ 11\ 11\ 11\ 11 \tag{1.2}$$

the row of pixels could be stored as:

$$10\ 00\ 11\ 00\ 13\ 06\ 11\ 05 \tag{1.3}$$

where the first number is the pixel colour followed by the number of times it is repeated.

[4]Most BMP images do not use compression, however there is a mode in the BMP scheme which employs RLE as the compression algorithm.

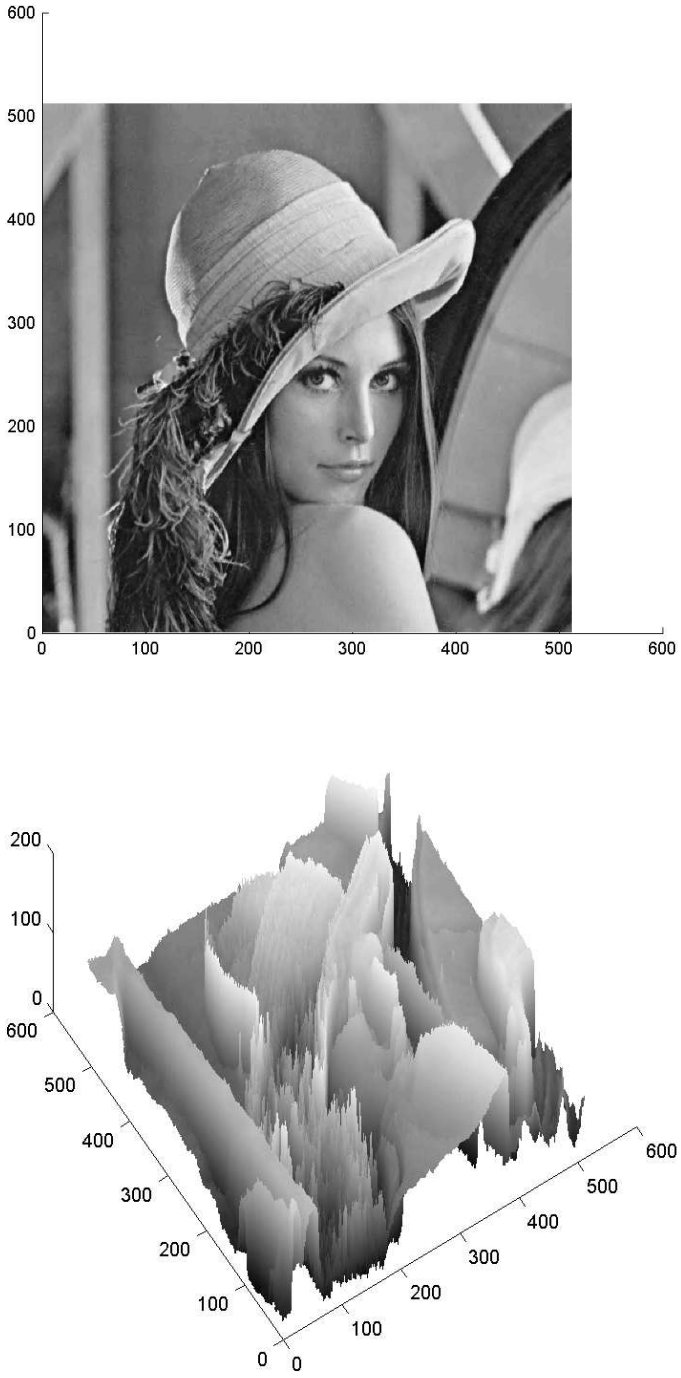

Figure 1.1: TOP: Original 512×512 'Lena' image. BOTTOM: Rotated 512×512 'Lena' image.

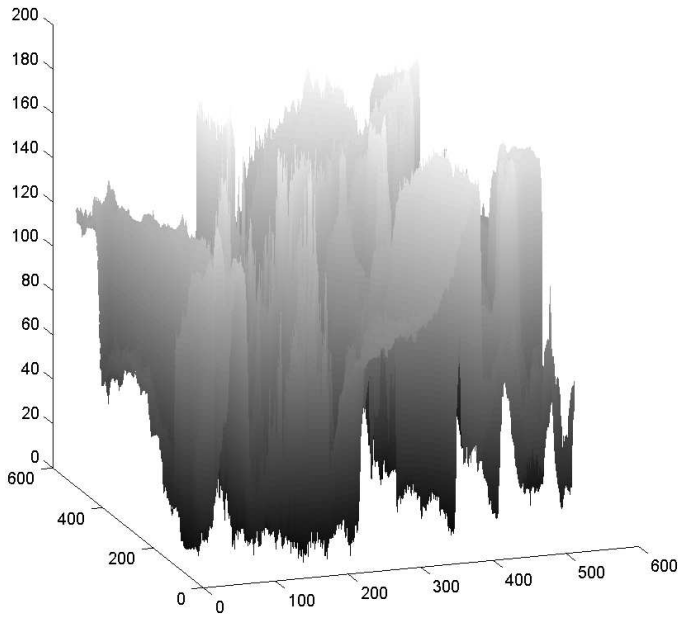

Figure 1.2: Rotated 512×512 'Lena' image illustrating the complexity of the surface to be modeled.

A two dimensional transform such as the fourier transform or discrete cosine transform can be applied to the colour surface. This transforms the colour surface from the spatial domain into the frequency domain. The associated inverse transform will recover the original image. In the frequency domain, compression algorithms operate on frequency coefficients (for example fourier coefficients if a fourier transform is used). This is generally more efficient than working in the spatial domain as higher frequency components can be attenuated or removed with little apparent degradation to the image. This phenomenon arises due to the way the human eye perceives images. The human eye is less perceptive to higher frequencies compared with lower frequencies, thus higher frequency components do not contain as much image information as the lower frequencies. Exactly how and which frequency components are attenuated/removed is the main focus of a frequency domain image compression algorithm.

Traditional compression algorithms (JPEG for example) sub-divide the image into smaller blocks, typically 8×8 pixels (ie 64 pixels in a block). Each block is

treated independently of all other blocks. Sub-dividing the image is partly due to historical factors such as limitations in computing power available at the time of designing the algorithm and partly due to practical limitations of mathematical methods used. For example the discrete cosine transform can theoretically work on any size block, however rounding errors occur when very large or very small numbers across a large number range are required to be stored as a series of bytes. While it is possible to accurately store these numbers, the extra bytes required decreases the compression efficiency.

At high compression ratios a phenomenon known as blocking becomes apparent. This applies to techniques which subdivide the image into (usually 8×8) blocks. It arises because the compression algorithm operates on each block independently. An example of this is shown in figure 1.3. This image shows the Lena image compressed using the JPEG algorithm at a compression rate where the blocking becomes apparent.

In recent years, wavelet methods [7] have been used in compression algorithms. Wavelet methods work on the whole image rather the traditional way of sub-dividing the image into blocks. This is generally more efficient in terms of compression ratios than methods which subdivide the image into blocks. At high compression ratios, the blocking effect of algorithms which subdivide the image into blocks is replaced in wavelet methods by a 'smearing' of the image. This usually becomes apparent at compression ratios much higher than when blocking is apparent in traditional methods. The smearing effect is illustrated in figure 1.3.

Figure 1.3: TOP: Blocking effects seen in the JPEG compression algorithm at high compression (ratio = 27:1). BOTTOM: Smearing effects seen in wavelet compression at high compression (ratio = 114:1)

1.2 Statement of the Problem

This thesis uses neural networks to compress still images. The neural network compression problem can be stated as: how to model the colour surface using the fewest parameters while still retaining an acceptable approximation of the surface. In the spatial domain this can be formulated as a nonlinear regression problem as we are attempting to model a function of the form $y = f(x_1, x_2)$ where y is a colour value on the plane (x_1, x_2) and f is the function to be modelled.

Figures 1.1 and 1.2 demonstrate the complexity of the colour surface to be modelled.

Another aspect of the problem is how to store the approximated colour surface. In particular, as neural networks are used in this thesis to approximate the colour surface, how to store the parameters of the neural network(s). Clearly the parameters need to be stored such that the space required to store the parameters is much less than the space required to store the original image.

1.3 Previous Work

Image compression using a single hidden layer feed-forward neural network was described in [13]. The architecture of this network is shown in figure 1.4. In this network the number of inputs matches the number of outputs and the number of hidden layer neurons is less than the number of inputs/outputs. The image is subdivided into small images (blocks) of dimension 8×8 so there are 64 pixels in a block. The colour values of these 64 pixels are both the input and desired output so the network is acting as an encoder of patterns. Thus there are 64 inputs and 64 outputs. There are two training phases to the learning process. In the first phase the network is trained on random blocks to set the output layer weights. In the second phase the network is trained on each 8×8 block one-by-one. It is assumed that the receiver will already know what the output layer

weights are as these can be obtained by training the network on random blocks from different images. The important information to be stored (or transmitted) are the hidden layer weights. If we choose the number of hidden layer neurons to be 16 then we only need to store 16 data elements for each 8×8 block.

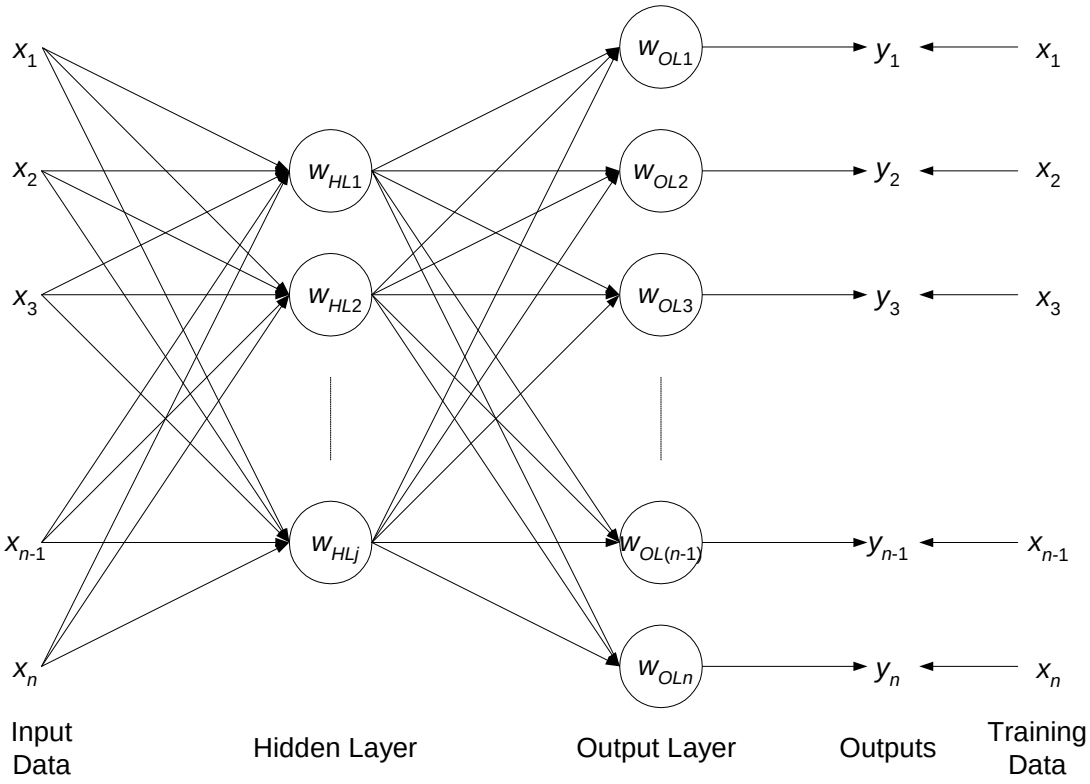

Figure 1.4: The feed forward network described in [13]. Inputs (and training data outputs) are pixel colour values. The output layer weights are common for all images so do not need to be stored. Compression is achieved by storing quantised values for the hidden layer weights.

In practice it was found that the maximum achievable compression ratio was 8:1 [13]. The compression is achieved by both the fact that only 16 data elements are required for each block of 64 pixels and by quantising these data elements so they only occupy 4 bits each, i.e. each transmitted hidden layer weight can be only one of 16 possible values. Where the weight does not match one of these values it is reassigned the value of the closest matching quantised value.

In the learning phase the input pixel values were normalised in the range

$[-1, 1]$. The learning consisted of between $50,000$ to $100,000$ iterations at learning rates of 0.01 and 0.1 for the hidden and output layer weights respectively.

There are several disadvantages of this back-propagation approach to image compression. Speed is a factor - back-propagation can take several minutes to several hours depending on the speed of the machine and the efficiency of the implementation. Back-propagation is not always guaranteed to find a solution and the training can be locked into a local minimum. The compression ratio of 8:1 falls far short of what is achievable with conventional (e.g. JPEG) methods. In short although refinements to the back-propagation algorithm in an image compression application are possible we conclude that the disadvantages far out-weigh any possible gains in performance achieved through these refinements (including batch-training an RBF network).

1.4 Support Vector Machines

A support vector machine (SVM) is a machine in the sense that it is given inputs which are processed by the machine to produce outputs. A SVM can also be seen (i.e. used) as a method of batch training an Radial Basis Function (RBF) network. In a standard RBF network the topology of the network must be defined before learning and is fixed during learning. In comparison SVM learning produces, as an interpretation of its outputs, the network topology.

Initially, support vector machines have mostly been applied to classification problems and SVM learning was first formulated as a solution to the problem of classifying sparse data sets. Traditional classification solutions work best on large data sets where the statistics of large numbers is a necessary factor in finding a solution. It is the aim of support vector machine learning to classify data sets where the number of training data is small and where traditional use of statistics of large numbers cannot guarantee an optimal solution. For example consider figure 1.5. Here two decision boundaries on the same data are seen.

Both decision boundary's correctly classify the data with zero error. However, subjectively, the decision boundary on the figure on the right is a better choice. The reason that this decision boundary is a better choice is because this decision boundary classifies the two classes with the maximum possible distance (the margin) between the nearest points of each class. The goal of SVM learning applied to a classification problem is to find the maximum margin decision boundary. This is termed a maximal margin classifier. The formulation of SVM is based on the structural risk minimisation (SRM) principle whereas the empirical risk minimisation (ERM) approach is commonly used in statistical learning (for example the classical batch learning for radial basis networks). This approach to SVMs gives SVMs a greater potential to generalise.

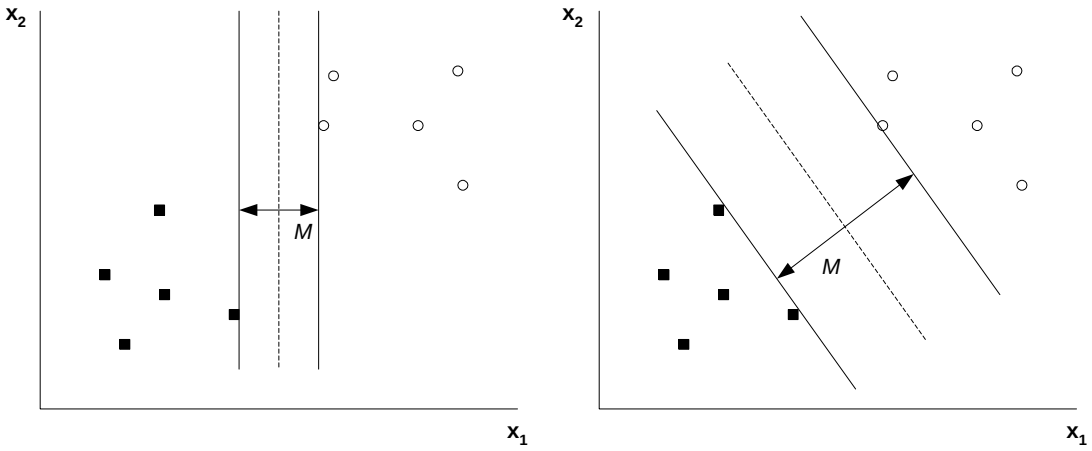

Figure 1.5: LEFT: Plot of two classes showing a small margin between classes defined by the decision boundary. RIGHT: The same plot but with the largest possible distance between opposite classes. SVM maximises the margin M.

Support vector machines have been applied to a variety of practical problems. A recent application is detailed in [14] where support vector machines were used to detect polyps in the colon. Polyps in the colon is the second leading cause of death in the United States. Although computer aided detection can detect a large number of cases, it does so at the expense of many false positives. In [14] a technique based on support vector machines was used to distinguish

the differentiating characteristics of polyps and healthy tissue using a three-dimensional pattern processing approach.

Another use for support vector machines was proposed in [15]. Here SVMs were applied to the problem of constructing an adaptive multiuser detector (MUD) for direct sequence code division multiple access (DS-CDMA). DS-CDMA is a communication technique that allows multiple users to transmit at the same carrier frequency in an uncoordinated manner. The problem is that multiuser interference is generated which can seriously degrade reception. Support vectors are applied here to classify the signals to minimise the multiuser interference.

The two examples of application of support vector machines to real problems are classification problems. SVMs can also be applied to regression (i.e. function estimation) problems. An input parameter to a SVM is a so-called ϵ-insensitivity zone (alternatively a tolerance or error). The goal of the SVM is to produce an output which is within this ϵ-insensitivity zone. Figure 1.6 illustrates a 1-dimensional regression problem with the geometrical interpretation of the ϵ-insensitivity zone. Here it is seen that the SVM has generalised the input signal and selected 9 of the 41 input points to use as centres for Gaussian kernel functions in a radial basis network. By defining the ϵ-insensitivity we can interpret the action of the SVM as 'compressing' the input data from 41 points to 9 points. It is this property of SVMs that is exploited in this thesis.

1.5 Research Objectives and Contributions

The main research objective is to investigate the sparse properties of kernel machines (SVMs) with application to image compression.

This thesis presents results of original work in the application of SVMs to the image compression problem. At the time of writing, although neural networks have been applied to the problem of image compression, there have not been any reports of SVM application to image compression. Algorithms exploiting sparse

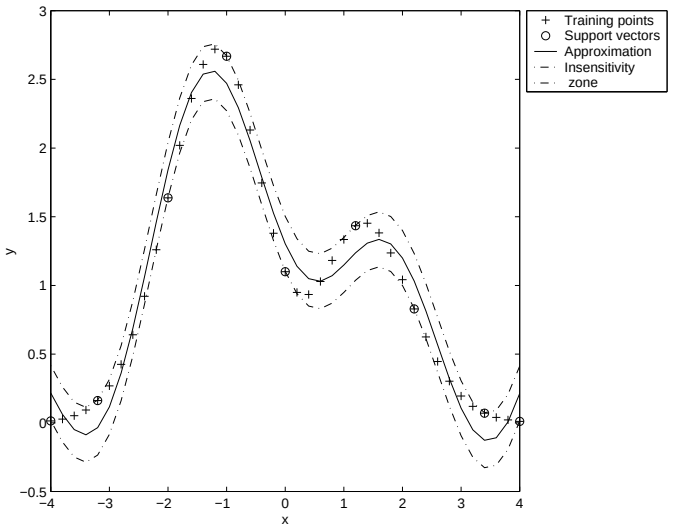

Figure 1.6: Example output of a 1-dimensional SVM. The training data comprises 41 points. 9 points were chosen by the SVM as support vectors so the 'compression' is 41:9.

properties of SVM learning are used to compress images in both the spatial and frequency domains. Compression is achieved in the frequency domain by combining SVM learning with the Discrete Cosine Transform (DCT). A novel approach is taken by the introduction of the 'inversion number'. The inversion number enables dramatic improvement of compression in the frequency domain and has resulted in the filing of a world-wide patent.

Also presented is the LOMON algorithm (LOcal MOdelling Networks) and combination of LOMON with SVM learning

1.6 Organisation of this Thesis

This thesis is organised as follows. The following chapter introduces an algorithm named LOMON (LOcal MOdelling Network) which employs a divide and conquer strategy to compress images. LOMON does not actually use SVMs but does use radial basis function networks. Chapter 3 discuses support vector machines and develops the theory used in compression algorithms detailed in later

chapters. Chapter 4 uses SVMs to model the colour surface presenting results in comparison to the baseline JPEG compression standard. LOMON is combined with SVM surface modelling in chapter 5. Following this is chapter 6 which applies SVMs to modelling an image in the frequency domain. Specifically we use SVMs to compress Discrete Cosine Transform (DCT) coefficients. Chapters 7 and 8 continue this work in the frequency domain by modelling the DCT coefficients in two dimensional space. The thesis is concluded with summary and discussion in chapter 9

Chapter 2

LOMON: LOcal MOdelling Network

Although not directly related to support vector machines this chapter presents some preliminary work on neural networks applied to an image compression problem. This work was first presented in a paper by the author in [16].

The LOMON (Local MOdelling Network) method directly models the colour surface. LOMON is an algorithm which uses a 'divide and conquer' strategy to take advantage of spatial redundancy in an image. The algorithm is given in figure 2.1, and a flow diagram is illustrated in figure 2.2.

The algorithm starts by creating two identical images from the original image and subdivides one image into two equal size vertical blocks and the other into two equal size horizontal images. A Gaussian is placed at the centre of each block and the weight and biases computed. The image with the lowest RMS error is chosen and the other image discarded. The cycle is repeated by creating two identical images from the new image. This time the block with the highest absolute error is selected to divide horizontally and vertically. These images are compared and the one with the lowest RMS error is selected. This process is repeated by selecting the block with the highest absolute error to divide until

the largest absolute error has fallen below a pre-defined level.

This method is similar to the LOLIMOT models as presented in [17]. The main difference between LOMON and LOLIMOT is that LOLIMOT operates on the whole input space with a single neural network, while LOMON treats each block with a separate neural network. The reason for this is that for realistic image sizes, the number of computations required to train on the whole image with a single neural network would require many hours to complete training. Thus the length of time to compress an image makes LOLIMOT infeasible in image compression in the form presented in [17].

```
current_block = block with largest error

Repeat

    (image_error_vert, left_block, right_block) =
        split_vert(current_block)
    (image_error_horz, top_block, bottom_block) =
        split_horz(current_block)

    remove current block from block list

    if image_error_vert < image_error_horz then
        add left_block, right_block to block list
    else
        add top_regio, bottom_block to block list

    current_block = block with largest error
    get max_image_error

Until max_image_error < 20
```

Figure 2.1: The LOMON algorithm where $\epsilon =$ `max_image_error` is the maximum error in a pixel.

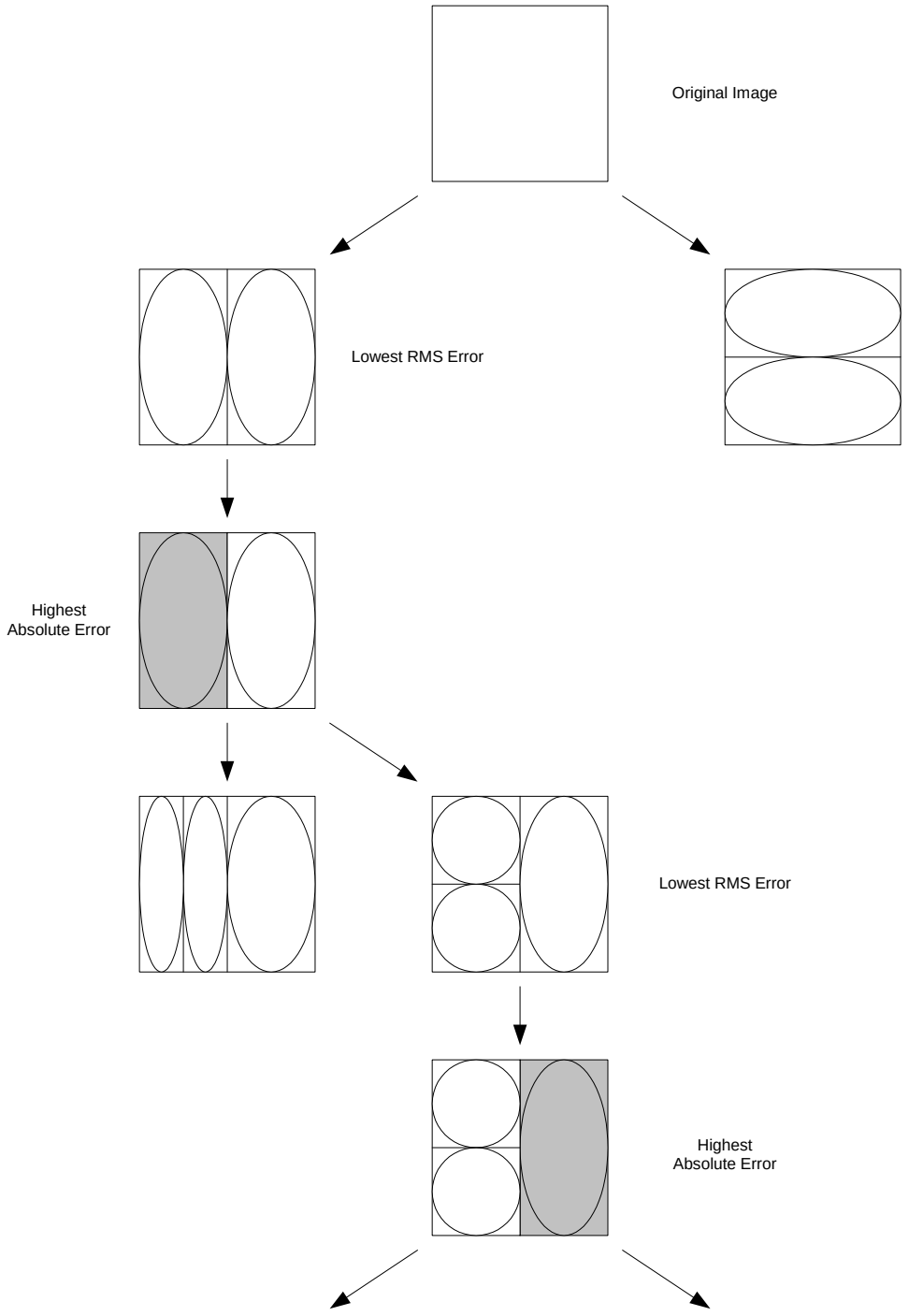

Figure 2.2: The LOMON strategy. Create two images from the original image. Subdivide one image into vertical blocks, the other into horizontal blocks and place a Gaussian at the centre of each block. Find the image with the lowest RMS error. In this image find the block with the largest absolute error (error in colour between a pixel in the compressed image and the original image). Create two images by dividing this block into vertical and horizontal blocks and repeat until the largest absolute error has fallen below a pre-defined limit ϵ.

2.1 Training

For each block we want to solve:

$$\mathbf{y} = \mathbf{G}\mathbf{w} \tag{2.1}$$

where $\mathbf{y}$ is the desired output (training data), $\mathbf{G}$ is the so-called 'RBF design' (aka Gramm) matrix, and $\mathbf{w}$ are the weights to be found. $\mathbf{G}$ is defined by:

$$\mathbf{G} = \begin{pmatrix} G(x_1, c_1) & \cdots & G(x_1, c_m) \\ \vdots & & \vdots \\ G(x_n, c_1) & \cdots & G(x_n, c_m) \end{pmatrix} \tag{2.2}$$

where $G(x, c)$ is the gaussian kernel function given by:

$$G(x, c, \lambda) = \exp\left[-\frac{1}{2}\left(\frac{(x - c)}{\lambda} \right)^2 \right] \tag{2.3}$$

where

x is the spatial coordinate

c is the centres of the Gaussians

λ is the Gaussian width (variance of Gaussian or shape parameter) (2.4)

The 2-dimensional Gaussian is given by:

$$G(x_1, x_2, c_1, c_2, \lambda_1, \lambda_2) = \exp\left[-\frac{1}{2}\left(\frac{(x_1 - c_1)}{\lambda_1} \right)^2 \right] \times \exp\left[-\frac{1}{2}\left(\frac{(x_2 - c_2)}{\lambda_2} \right)^2 \right] \tag{2.5}$$

The input ($\mathbf{X}$) is a vector the colour surface $z = f(x, y)$ is mapped into a

single column of numbers. We do this by setting:

$$\mathbf{y} = \begin{pmatrix} f(1,1) \\ f(1,2) \\ \vdots \\ f(1,n_x) \\ f(2,1) \\ f(2,2) \\ \vdots \\ f(2,n_x) \\ \vdots \\ f(n_y,n_x) \end{pmatrix} \tag{2.6}$$

where n_x is the image dimension along the x-axis (width), n_y is the image dimension along the y-axis (height) and $f(x,y)$ is the colour of the pixel at position (x,y).

Since only a single Gaussian is placed in each a block, $\mathbf{G}$ is a column vector given by:

$$\mathbf{G} = \begin{pmatrix} g(x_1,c) \\ g(x_2,c) \\ \vdots \\ g(x_n,c) \end{pmatrix} \tag{2.7}$$

Solving (2.1) for $\mathbf{w}$:

$$\mathbf{w} = \mathbf{G}^{+}\mathbf{y} \tag{2.8}$$

where $\mathbf{G}^{+}$ is the pseudo-inverse given by

$$\mathbf{G}^{+} = (\mathbf{G}^{T}\mathbf{G})^{-1}\mathbf{G}^{T} \tag{2.9}$$

and $\mathbf{G}^{T}$ is the transpose of $\mathbf{G}$.

In the LOMON algorithm ϵ is defined as the maximum error in an individual pixel (`Max_Image_Error` in Figure 2.1). i.e.

$$\epsilon = \max \sqrt{(\text{training output} - \text{input})^2} \tag{2.10}$$

Figure 2.3 is a plot of ϵ for each iteration of LOMON for the 256×256 Lena image. It can seen that even after around 25,000 iterations the error is still large. Note also that the error plotted in 2.3 can be larger than the maximum pixel level (255). In practice the value of the pixel would be limited to 255, however for clarity the raw error has been left in figure 2.3.

The plot of the image in Figure 2.4 after 20,000 iterations provides some clue as to why the LOMON network is taking so long to train on the image. The white 'spots' in top figure in Figure 2.4 are individual, small blocks. The algorithm has concentrated on the parts of the image where the change between the colour of adjacent pixels is largest (i.e. edges within the image) resulting in LOMON dividing the image into many small blocks. It appears that each block consists of only a few pixels. If the algorithm were left to run, the number of blocks would approach the number of pixels defeating the purpose of an image compression algorithm. The bottom figure in Figure 2.4 is a rotated view. Here it clear that the blocks are generally very small and that the algorithm has concentrated on areas of high detail.

We conclude that the gaussian with the only degree of freedom being a single weight is not adequate to model a larger block of pixels. In response to this we modify the training algorithm with the addition of bias terms. Geometrically, a bias will shift the Gaussian bell up or down.

To find the biases $\mathbf{G}$ in equation (2.7) is augmented with a column of 1's. This is the equivalent of a kernel function that does not vary with location. Thus the 'weight' corresponding to this kernel function can be treated as a 'bias' or offset as it will be the same for all pixels in the region (unlike the Gaussian whose value is a function of location). Thus:

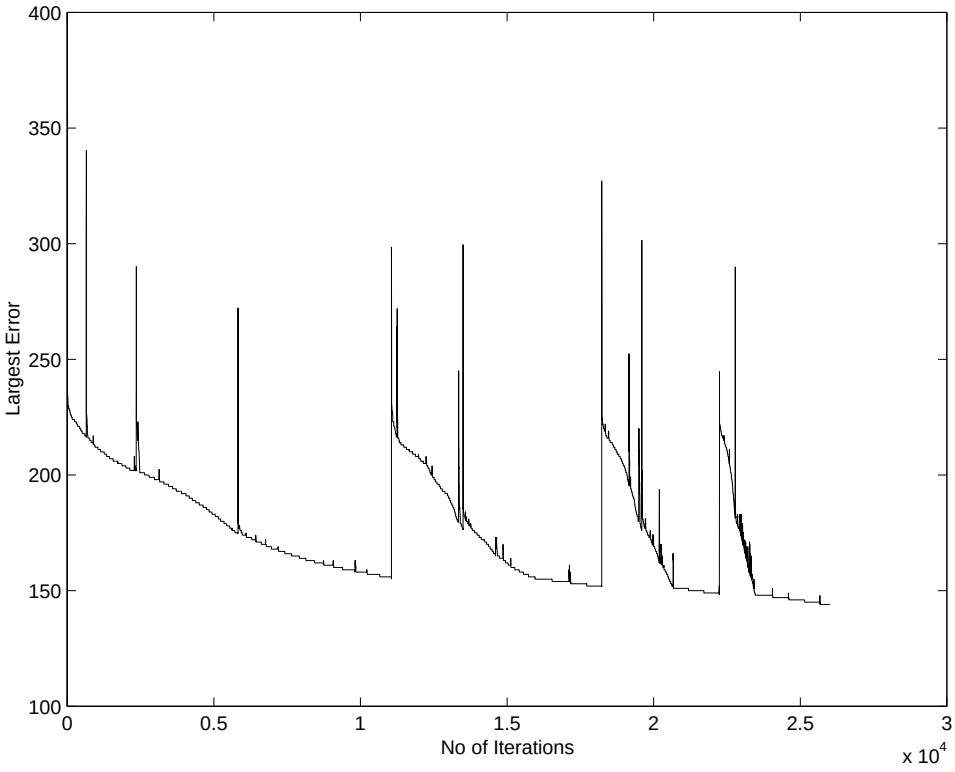

Figure 2.3: Largest pixel error ϵ during LOMON training for approx 26000 iterations on the 256×256 Lena image. It can be seen that the algorithm will take a long time to reach an acceptable error (say around 20).

$$\mathbf{G} = \begin{pmatrix} g(x_1, c) & 1 \\ g(x_2, c) & 1 \\ \vdots & \\ g(x_n, c) & 1 \end{pmatrix} \tag{2.11}$$

$\mathbf{G}$ now has dimensions $n \times 2$. The solution to (2.8) is now a column vector with two terms:

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \tag{2.12}$$

where w_1 is the weight and w_2 is the bias. The solution is found as before:

$$\mathbf{w} = \mathbf{G}^{+}\mathbf{y} \tag{2.13}$$

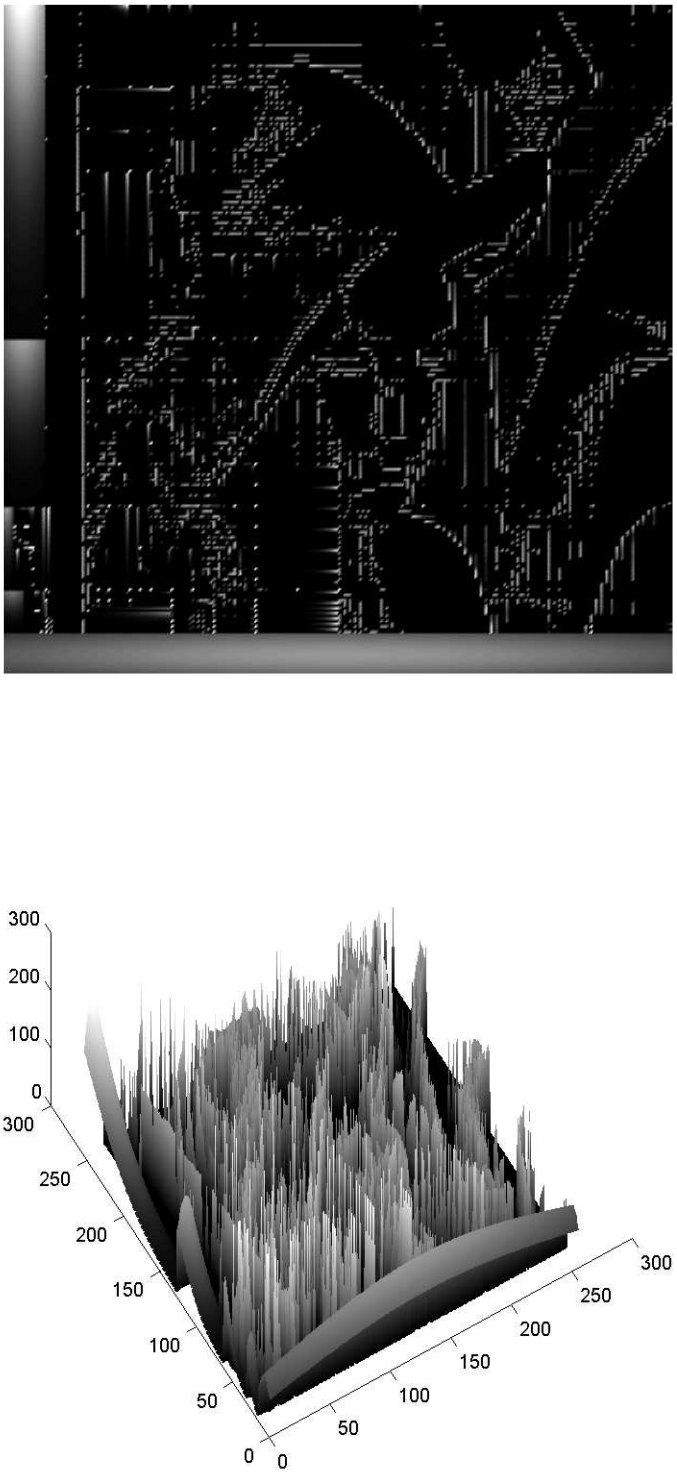

Figure 2.4: TOP: LOMON trained on Lena image. Here it can be seen that the algorithm has concentrated on areas of high contrast (i.e. edges). Each white 'spot' is a small block. The black portion of the image has not yet been processed by the LOMON algorithm. BOTTOM: The figure has been rotated. It is clear that the algorithm has concentrated on the areas of the image for which the colour change between adjacent pixels is greatest.

2.2 Results

Figure 2.5 is a plot of the largest error versus iteration. The spikes on the error curve are due to the largest error being larger than for the previous iteration when block is separated into two. It can be clearly seen that an image with an acceptable error is obtained much sooner with the addition of the bias term. We point out that the addition of the bias term has the consequence that an extra number must be transmitted for each block in a compression scheme. However the additional parameter has reduced the number of blocks by approximately an order of magnitude so the number of parameters required to be transmitted is still far fewer than if no bias is used.

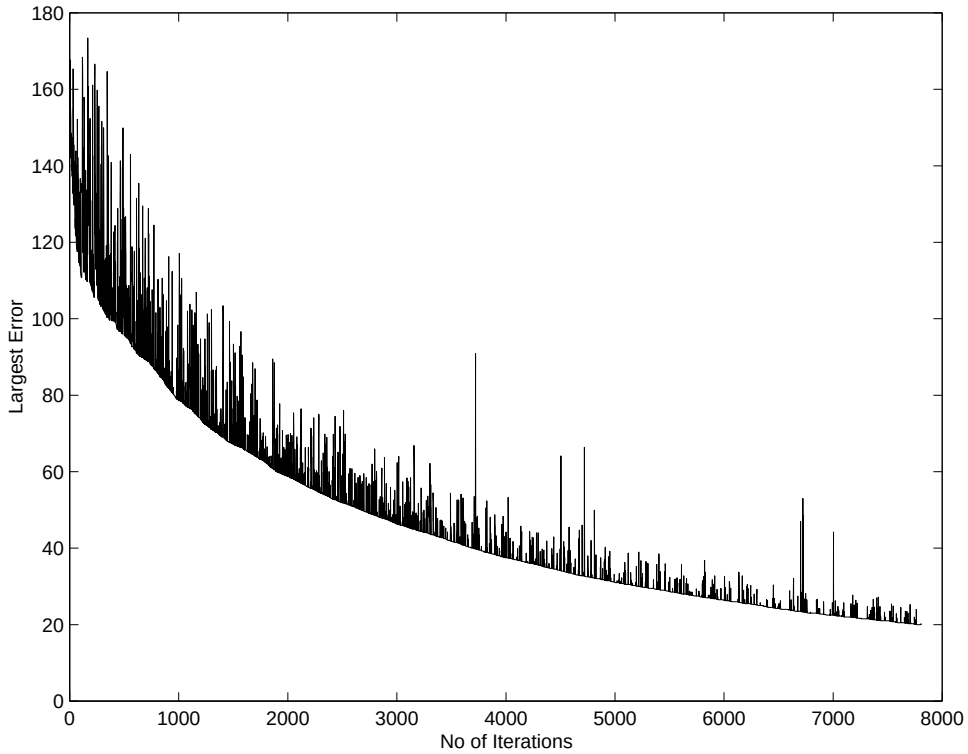

Figure 2.5: Largest pixel error during LOMON (with bias term) training for approx 7800 iterations.

Figures 2.6 and 2.7 show the reconstructed image on the 256×256 Lena image and the 400×400 Claudia image for various errors ϵ and constant shape

parameter λ. In both figures 'blocking' becomes evident when the error ϵ is greater than 10 and as ϵ is increased, compression is increased but image quality deteriorates. JPEG images have been included in the figures for comparison. For the Lena image it is clear that the JPEG algorithm performs far better than LOMON, however for the Claudia image JPEG is better when the compression is low and subjectively about equal when the compression is high. We note that the JPEG algorithm was far quicker to compress than LOMON, with LOMON requiring a few minutes to compress a single image compared with JPEG which required less than a second.

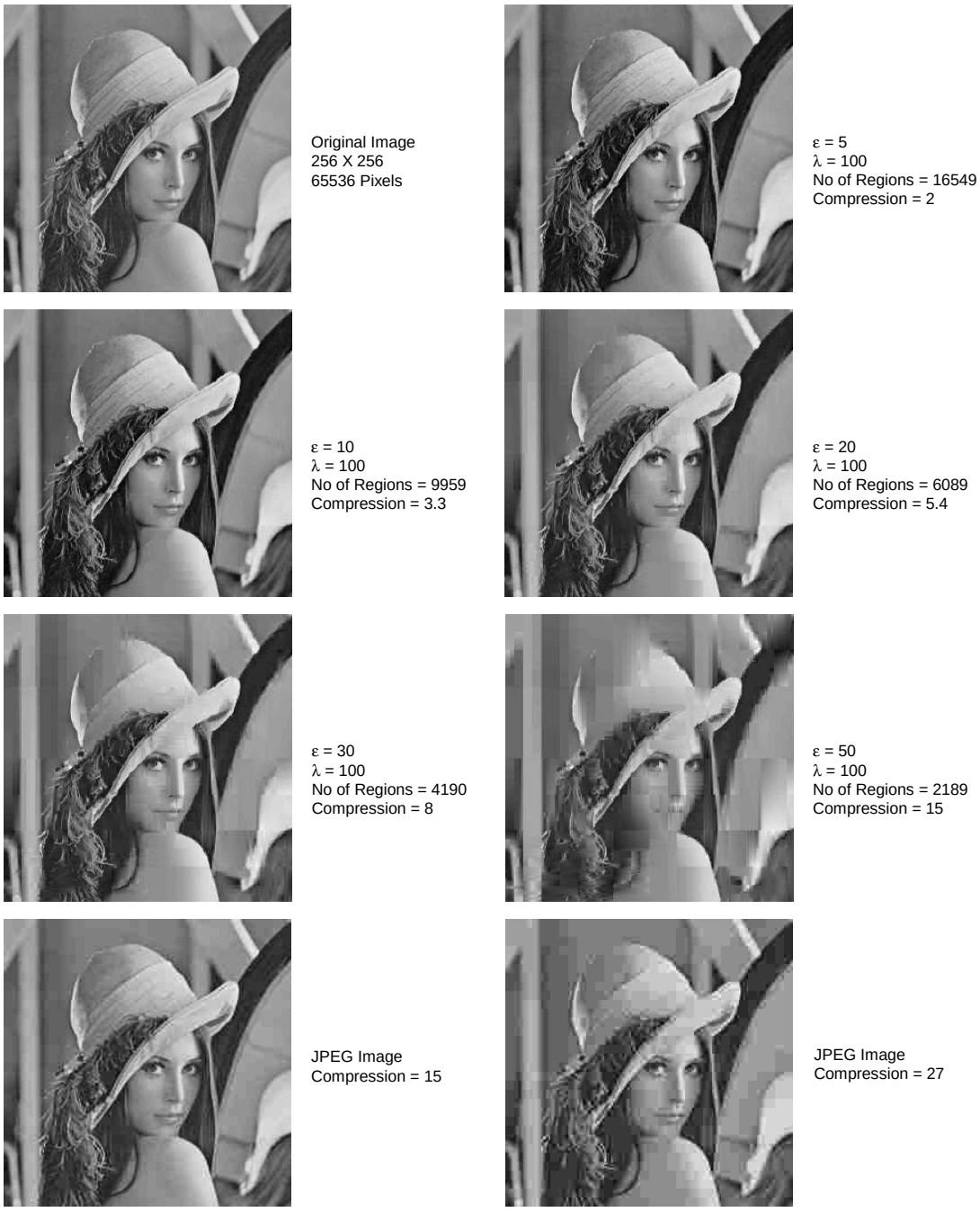

Figure 2.6: 256×256 Lena image trained using LOMON for different values of largest-error ϵ and constant shape parameter $\lambda = 100$. The compression has been estimated on two bytes required to encode each block.

Figure 2.7: 400×400 Claudia image trained using LOMON for different values of largest-error ϵ and constant shape parameter $\lambda = 100$. The compression has been estimated on two bytes required to encode each block.

2.2.1 Effect of the Gaussian Shape Parameter

The Gaussian function formula given in equation 2.3 has a shape parameter, λ. This parameter sets the width of the Gaussian bell curve. For the value of the shape parameter we have two choices: we can set λ to be a pre-defined single value for all Gaussians; or we can allow λ to have a different value for each Gaussian. For the second choice we link the value of λ to the physical dimensions of the block that the Gaussian is placed. As the colour surface is a 2-dimensional surface, the Gaussian is two dimensional requiring that λ is defined along two axes (ie the width of the Gaussian along the 'width' or x-axis and the width of the Gaussian along the 'height' or y-axis). We link the Gaussian shape parameter λ to the physical dimension of the block in which it is placed by setting

$$\lambda_x = N \times w \tag{2.14a}$$

$$\lambda_y = N \times h \tag{2.14b}$$

where λ_x and λ_y are the shape parameter in the $x-$ and $y-$ axes, w and h are width and height of the block in which the Gaussian is placed and N is a pre-defined integer.

Figure 2.8 shows a plot of the number of blocks generated by the LOMON algorithm versus the Gaussian shape parameter λ for both fixed and variable λ. From inspection of Figure 2.8 it appears that the number of blocks decreases until a limit is reached. Beyond this limit the number of blocks stays constant. This occurs for both the fixed and variable λ cases. Figure 2.8 also shows that the variable shape parameter λ (i.e. λ as defined in equation (2.14a)) produces a slightly less number of blocks (thus, increased compression) than for fixed λ.

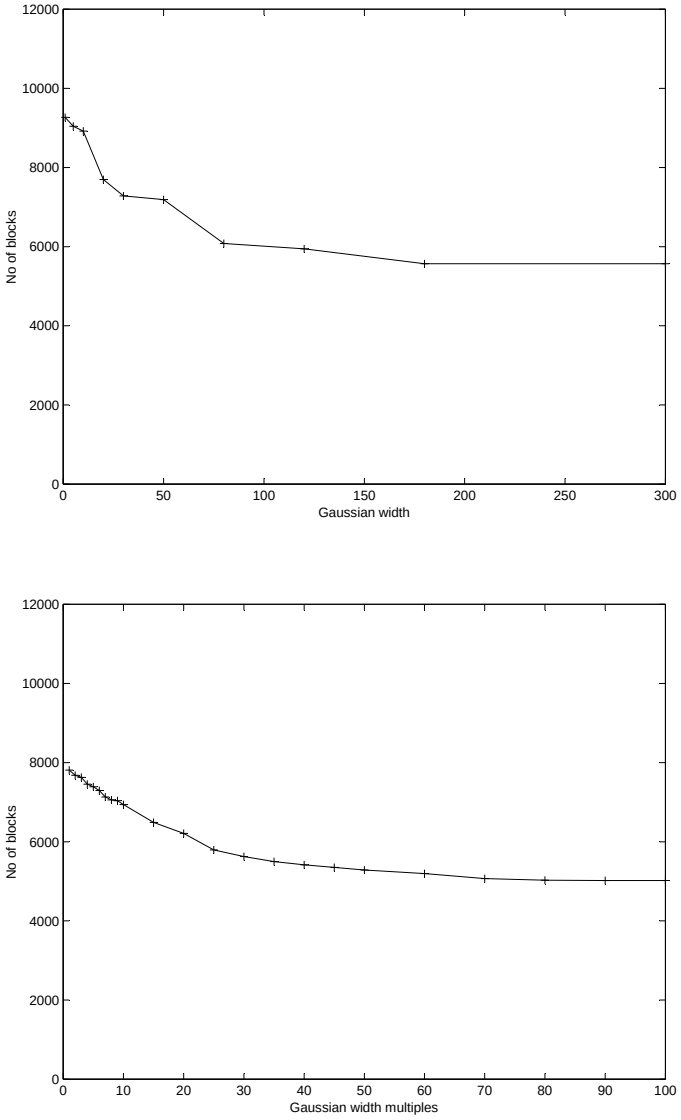

Figure 2.8: Number of blocks versus Gaussian shape parameter λ (Gaussian width). TOP fixed width. BOTTOM width is a multiple of block dimensions.

2.3 LOMON With Multiple Gaussians

The LOMON algorithm is modified so that multiple Gaussians are placed on each block. Figure 2.9 shows the case where two Gaussians are placed when a block is split horizontally and vertically. Note that the resulting Gaussians are in the same physical position independent of whether the split is vertical

or horizontal. Figure 2.10 shows the recovered image when shape parameter $\lambda = 50$ and maximum error $\epsilon = 50$. The blocking effect is clearly visible in this recovered image. Also shown in Figure 2.10 is the recovered image rotated so that the recovered Lena image is visible as a 3-dimensional colour surface. It can be seen from this figure that there is less 'activity' in the high detail areas although sharp edges have generally been preserved. Figure 2.11 shows how the blocks are divided by the LOMON algorithm and where the Gaussians are placed. Each block has two Gaussians, one marked by a '+' and the other by a 'o'. The image is recognisable as the Lena image as the algorithm has placed many small blocks in areas on the image with high detail. Also note in this figure that there are vertical blocks (i.e. the vertical length is longer than the horizontal length) that contain two Gaussians along the horizontal axis. This is particularly visible in the top left of the image. This occurs because the algorithm has found that this gives a lower error than if the Gaussians had been placed along the vertical axis.

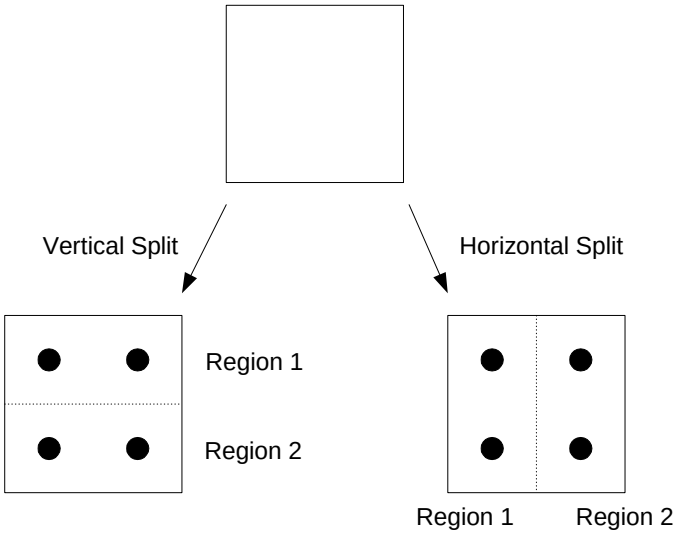

Figure 2.9: Definition of splitting in LOMON for two Gaussians

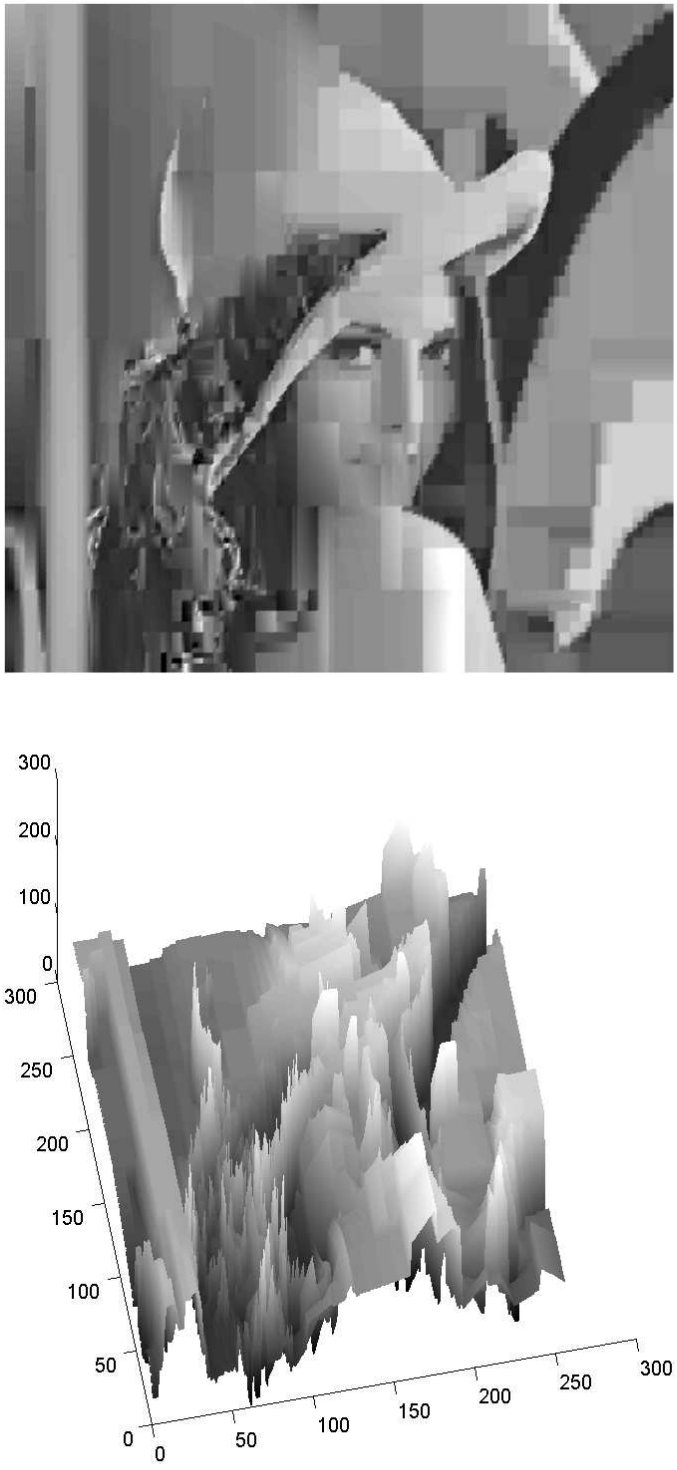

Figure 2.10: TOP: 256 × 256 Lena image trained using LOMON using two Gaussians per block. BOTTOM: The rotated recovered image. Here error $\epsilon = 50$ and shape parameter $\lambda = 50$, number of blocks = 2234.

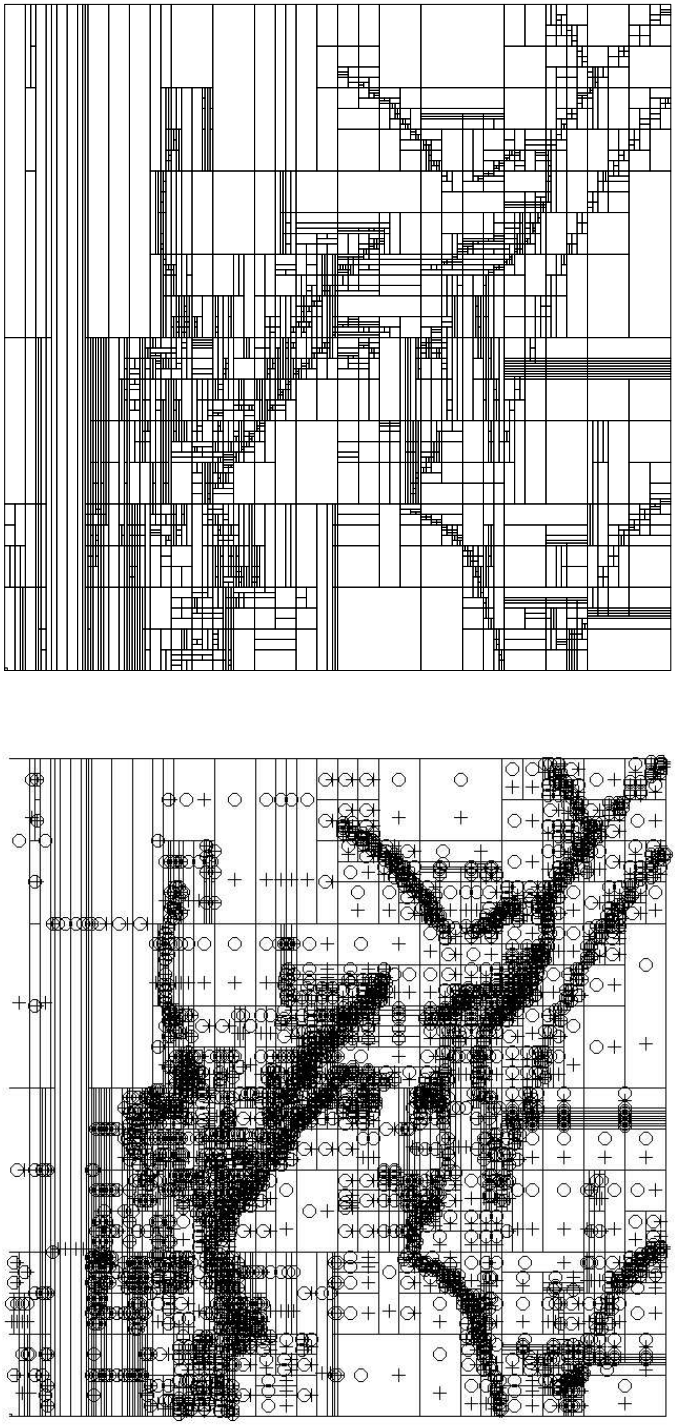

Figure 2.11: TOP: 256×256 Lena image showing blocks generated by LOMON (two Gaussians per block) for the image shown in Figure 2.10. BOTTOM: Same image as top showing the placement of Gaussians. The two Gaussians on each block are indicated by a '+' and a 'o'

The top plot in Figure 2.12 shows a plot of number of blocks generated by the LOMON algorithm versus fixed Gaussian width when two Gaussians are placed in each block as defined in Figure 2.9. The figure shows the optimal Gaussian width is around 200 for the 256×256 Lena image and 120 for the 400×400 Claudia image. This difference indicates that the optimal value for λ is an empirical parameter related to the content of the image. The bottom plot in Figure 2.12 is a plot of a variable Gaussian width. In this plot the width is an integral number of dimensions of the block in which the Gaussian in placed. From inspection of the plot in Figures 2.12 the better option for the two-Gaussian LOMON algorithm is a fixed λ rather than one that is allowed to vary with block size for both images. Figure 2.13 is a plot of the Lena and Claudia images using optimal values for λ for both the fixed and variable cases. The blocking effect is clearly visible in both the Lena and Claudia images. It is also seen that the compression is higher for the 400×400 Claudia image than for the 256×256 Lena image. A possible reason for this is that if both images were to be divided into 8×8 blocks, the smaller Lena image generally contains more detail in each 8×8 block than the larger Claudia image (i.e. the information per block is generally higher in the Lena image than for the Claudia image). Although the absolute number of blocks produced by the LOMON algorithm is higher for the Claudia image, the average number of pixels per block is higher resulting in a higher compression compared with the Lena image.

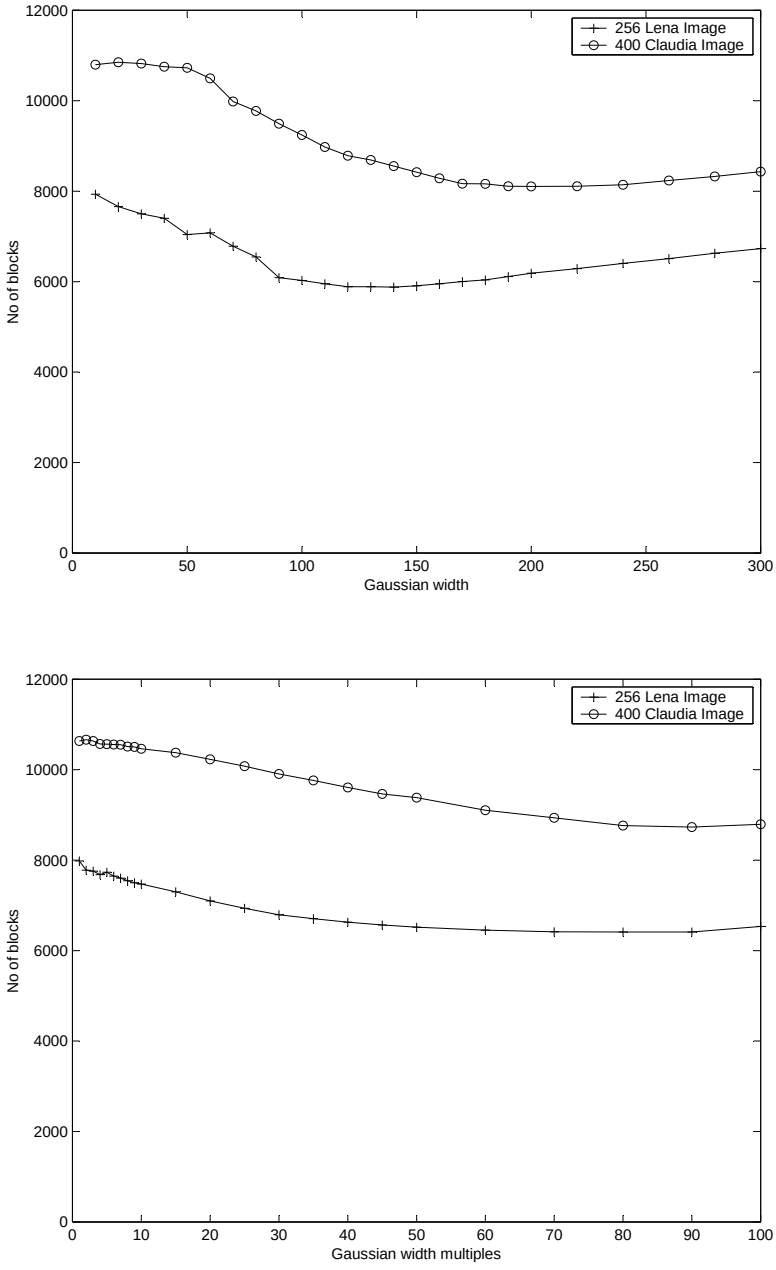

Figure 2.12: TOP: Number of blocks versus Gaussian width for two Gaussians using the 256×256 Lena image and the 400×400 Claudia image. Here ϵ is set to 20. BOTTOM: Variable Gaussian width versus number of blocks for two Gaussians using the 256×256 Lena image and the 400×400 Claudia image with $\epsilon = 20$. The width is an integral number of dimensions of the block which it is placed.

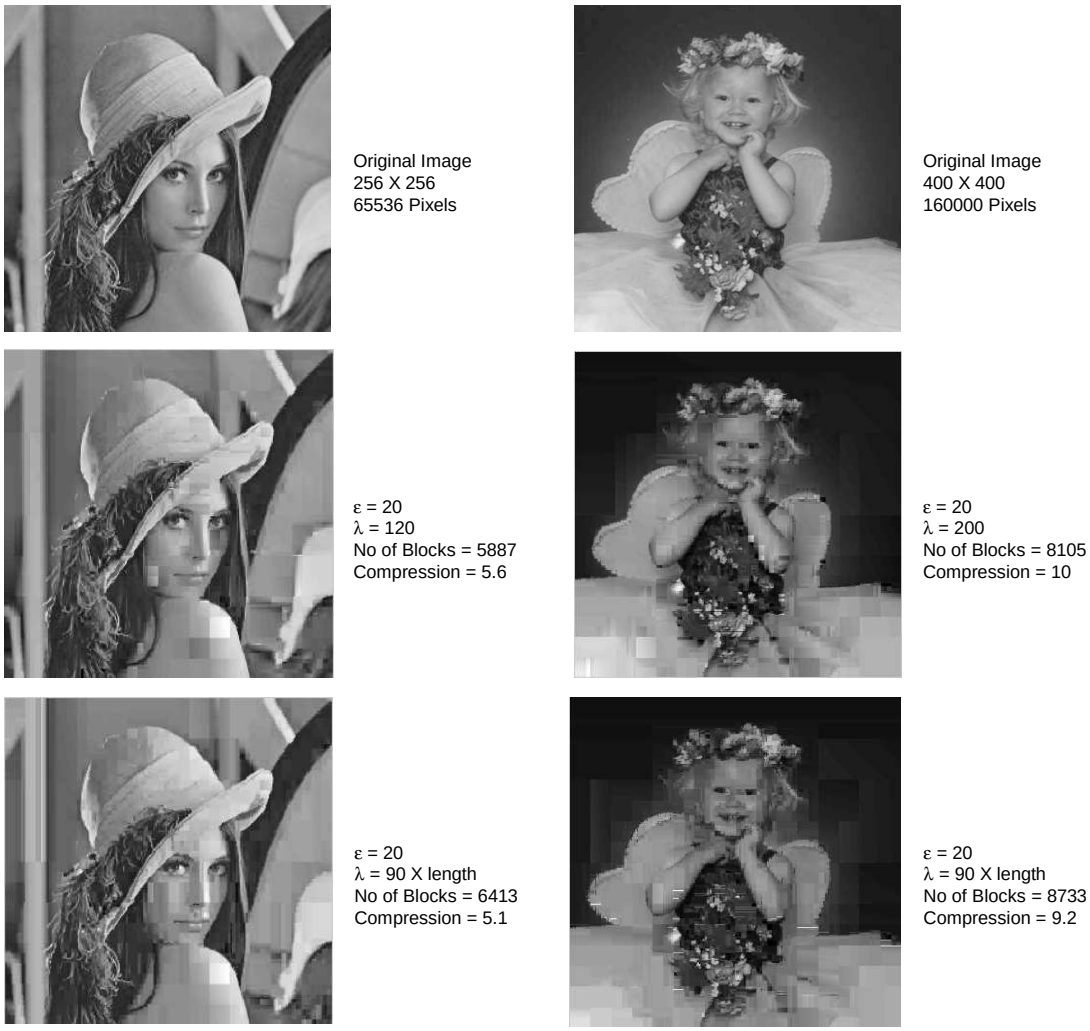

Figure 2.13: LOMON results for two Gaussians per block for both fixed and variable λ. Compression is estimated using an average of two bytes per block.

2.3.1 Four-Gaussian LOMON

The LOMON algorithm is now extended so that four Gaussians are placed in each block. The Gaussians are placed as defined by LOMON splitting in Figure 2.14. Note that unlike the two Gaussian separation (Figure 2.9), the layout pattern of the Gaussians in the vertical and horizontal blocks is different. Figure 2.15 shows the recovered image when shape parameter $\lambda = 50$ and maximum error $\epsilon = 50$. As in the two Gaussian case the blocks are clearly visible. Also shown

in Figure 2.15 is the recovered image rotated to show the 3-dimensional colour surface. As before it can be seen that the sharp edges in the image are preserved while individual features in high detail areas is lost. It may be expected that by increasing the number of Gaussians per block that the number of blocks would be reduced. However the four Gaussian LOMON algorithm produced 2719 blocks as compared with 2234 blocks for the two Gaussian LOMON algorithm for the same shape parameter and maximum error. Figure 2.16 shows how the blocks are divided by the LOMON algorithm and where the Gaussians are placed. The figure shows that the general position and size of the blocks is similar to that produced by the two Gaussian LOMON algorithm.

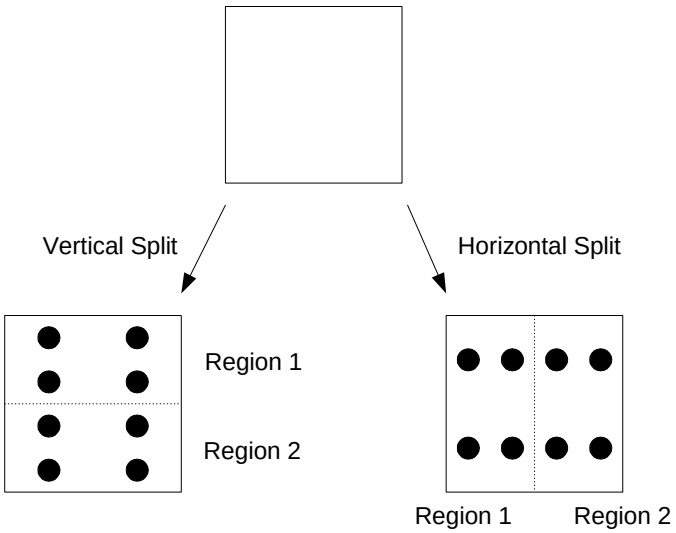

Figure 2.14: Definition of splitting in LOMON into four Gaussians per block

Figure 2.17 shows plots of the number of blocks versus Gaussian width for the case where shape parameter λ is fixed and for the case where λ is linked to the size of the block (i.e. variable λ) using the 256×256 Lena and 400×400 Claudia images as test images. Compare Figure 2.17 where 4 Gaussians are placed in each block with Figure 2.12 where 2 Gaussians are placed in each block. There is little effect on the number of blocks by increasing the number of Gaussians. In fact the number of blocks is slightly increased by increasing the number of

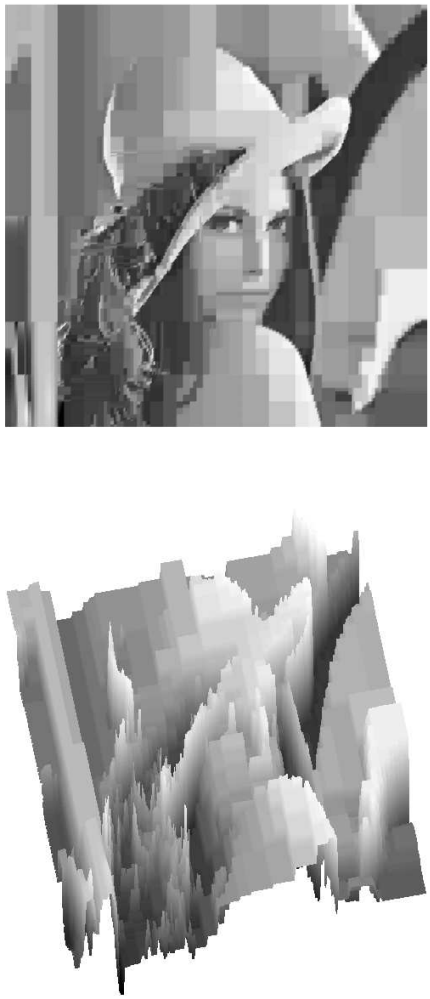

Figure 2.15: TOP: 256 × 256 Lena image trained using LOMON using four Gaussians per block. BOTTOM: The rotated recovered image. Here $\epsilon = 50$, $\lambda = 10$, No. of blocks = 2719.

Gaussians.

A possible explanation for this is that high detail areas have naturally small blocks. Increasing the number of blocks in these areas gives a greater edge length of the block close to a Gaussian when four Gaussians are placed in a block compared with two Gaussians in the same block. The result is that an edge colour will be at a higher value due to it's proximity to a Gaussian. The higher colour value may be greater than the maximum error ϵ requiring a split. If the Gaussian were further from the edge (as in the two Gaussian case) the

edge will have a lower colour value which may be within the specified ϵ.

2.4 Parameters Required to be Transmitted

In the LOMON scheme presented here, if the image were to be transmitted then we would need to transmit the following information:

1. The height and width of each block
2. The weight of each Gaussian
3. The bias parameter for each block

If these parameters could be coded (using for example Huffman coding detailed in chapter 6) so that on average, each block occupied two bytes, then where LOMON divides the image into 7808 blocks, 15616 bytes are required. This is a compression rate of $65536/15616 \approx 4.2$. This is a far lower compression ratio than an image compressed using the JPEG algorithm [8]. Working backward, if we wanted LOMON to produce the same compression rate as the bottom left JPEG image in Figure 2.6 (compression of 15:1), then the parameters need to be encoded in 4369 bytes. This requires LOMON divide a 256×256 image into no more than 2184 blocks. The results from the preceding sections indicate that, while LOMON can divide the image into 2184 blocks, the quality of the image will be far lower than the same compression rate as JPEG.

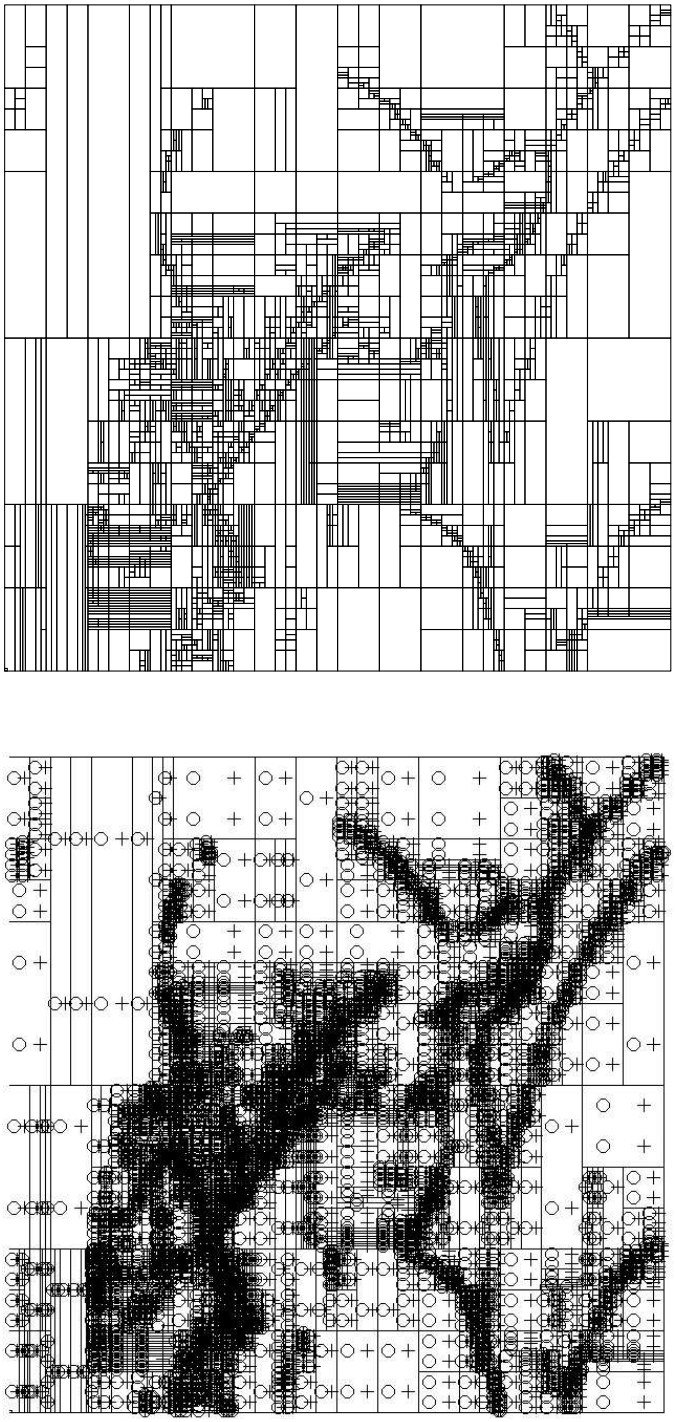

Figure 2.16: TOP: 256×256 Lena image showing blocks generated by LOMON (four Gaussians per block) for the image shown in Figure 2.15. BOTTOM: Same image as top showing the placement of Gaussians.

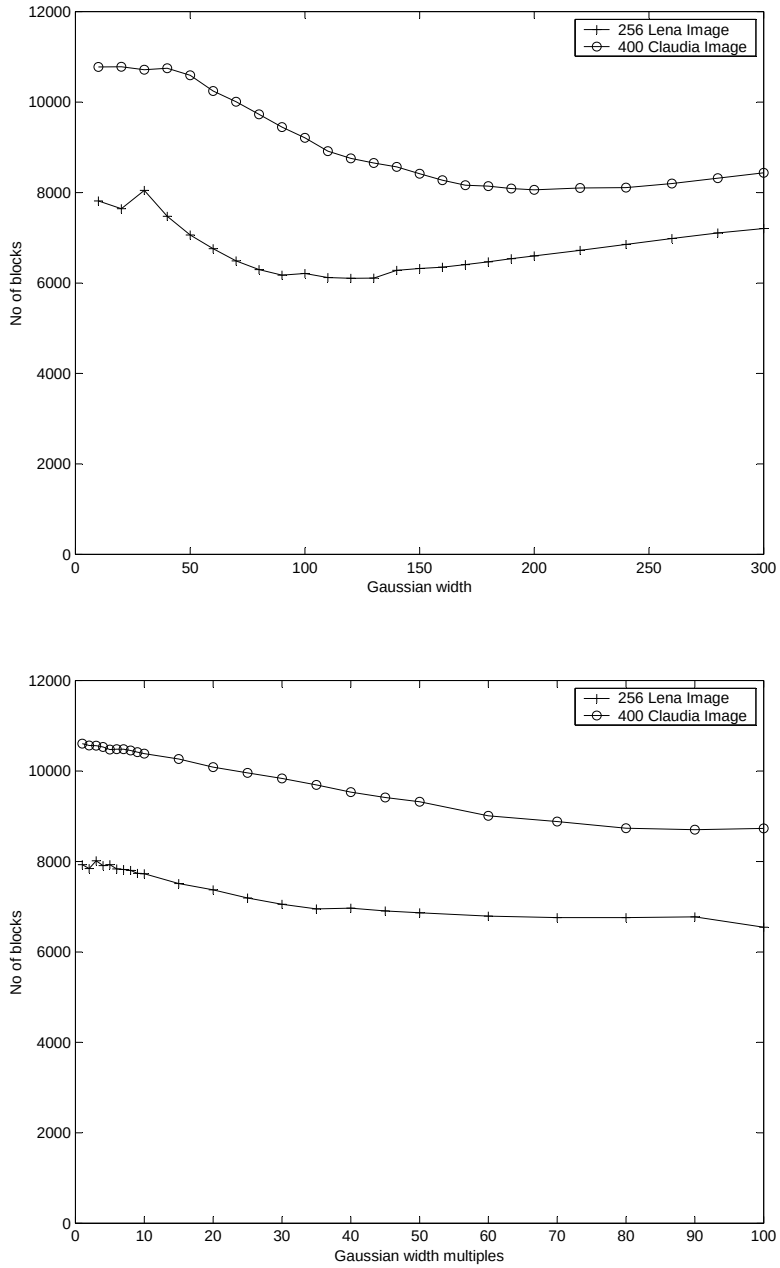

Figure 2.17: TOP: Number of blocks versus Gaussian width four Gaussian LOMON using the 256×256 Lena image and the 400×400 Claudia image. Here $\epsilon = 20$. BOTTOM: Variable Gaussian width versus number of blocks for four Gaussians using the 256×256 Lena image and the 400×400 Claudia image with $\epsilon = 20$. The width is an integral number of dimensions of the block in which it is placed.

2.5 Discussion

In this chapter a new algorithm (LOMON) has been presented utilising a divide and conquer strategy to approximate the colour surface of an image. The method operates in the spatial domain meaning that it works directly on the image colour surface. The image is divided into blocks into which Gaussians are placed. A Gaussian only influences the area of the block in which it is placed. The algorithm has been implemented so that it can place one, two or four Gaussians in a block (the number of Gaussians is predetermined so that an identical number of Gaussians are placed in each block). Increasing the number of Gaussians had a negligible effect on the number of blocks produced by LOMON and the compression was actually decreased (for the same image quality) as more weights are required to be transmitted in order to recover the image. With the addition of a bias term the compression and time taken to train the network of the image within a pre-defined tolerance (error) is much improved.

As a footnote a variation of LOMON was attempted where the Gaussians influence every pixel in the image, not just the pixels located in the block that the Gaussian is placed. Although mathematically valid this approach turned out to be impractical due to the exponentially increasing number of computations required at each iteration. For example to find the inverse of a 2000×2000 matrix required many hours of computation, and this is just for a single iteration. After the first few iterations it became obvious that a single image would require weeks to compute and thus this approach was abandoned. Also tried was altering the selection criteria of the block to be split. In the algorithm shown in Figure 2.1 the criteria for selecting the block to be split is the block with the highest error in an individual pixel. A variation was tried where the block selected was the block with the highest RMS error. This had virtually no effect on the number of blocks created by the algorithm,

The algorithm does require several minutes to compress even a moderately

sized image (256×256 pixels) and compression and image quality are lower than the JPEG algorithm. It is concluded that although the LOMON algorithm has novelty value, it's application to image compression is of limited use.

Chapter 3

Support Vector Machine Learning

This chapter provides an introduction to support vector machine (SVM) learning and develops theory relevant to the compression algorithms presented in following chapters. The purpose of this chapter is to develop a theoretical basis for the support vector machine learning applied in following chapters without being too concerned with detailed mathematical theorems and proofs.

For example the explanation of support vectors normally begins with a detailed account on 'risk' or error as it is the aim of SVM learning to minimise risk. A detailed discussion of risk is not directly relevant to the application of SVM in this thesis although it is briefly touched upon in the discussion on SVM applied to regression problems. For more a more detailed and comprehensive account of support vector machines the interested reader is referred to [18, 19, 1, 20].

3.1 Introduction

A Support Vector Machine may be viewed as a method of training a radial basis neural network. In the classic least-squares batch-training method for

a radial basis function (RBF) network there is the problem of how to assign the centres of the Gaussian bells. The SVM solves this problem as it takes a maximum allowable error in the output as a training parameter. In the case of 1-dimensional regression (see Figure 3.1) this error can be geometrically interpreted as a tube or insensitivity zone centred on the training data. The SVM learning process selects the minimum number of training points to use as the centres of the Gaussian kernel functions in an RBF network in order for the function to be approximated within the insensitivity zone. These selected training points are the support vectors. Figure 3.1 shows the insensitivity zone drawn around the resulting function. When the penalty parameter, C (introduced below), is infinite, the support vectors will always lie at the edge of the zone as is seen in Figure 3.1. Support vector machines were initially developed for classification problems. It is beneficial to explain support vector machine theory using binary classification to illustrate the principles developed by the theory. Support vector techniques can be applied to regression problems, i.e. functional approximation problems [21, 22]. It is this application of support vector techniques that is exploited in this thesis.

3.2 Classification by SVM

First consider the case of the separable binary classification problem. The separable binary classification problems requires that a line be drawn between the two classes that correctly divides the two classes with zero error. Consider the two classification lines shown in Figure 3.2. The Figure on the right has a more logical classification boundary (or decision boundary) than the figure on the left although both classification boundaries correctly classify all points. Why? The classification boundary on the right has a margin (physical distance between opposite points) that is the maximum possible length. This is defined as the *maximal margin*. If a human were to draw a line that best separated the two

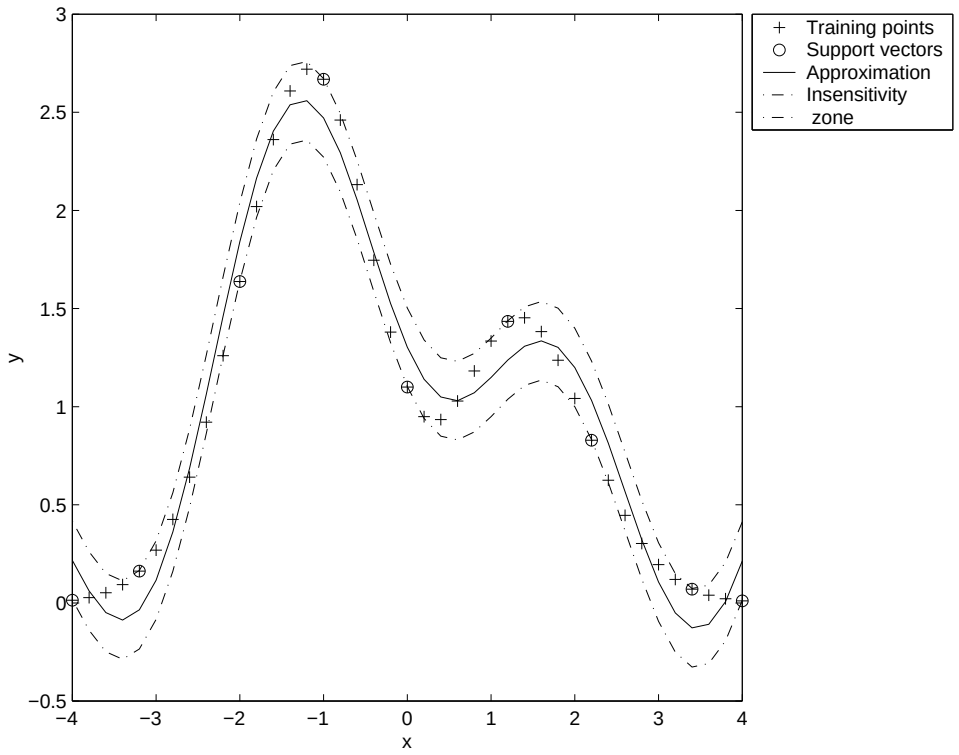

Figure 3.1: Example output of a 1-dimensional SVM. The training data comprises 41 points. 9 points were chosen by the SVM as support vectors so the 'compression' is 41:9. Note that the support vectors lie on the edge of the insensitivity zone surrounding the approximated function when the penalty parameter, C, is infinite.

classes it would be somewhere near the maximal margin boundary shown on the right plot of Figure 3.2. A traditional (e.g. back-propagation) neural network will start with the decision boundary set to some random (likely incorrect) orientation and train until the error is zero, perhaps stopping when the decision boundary is something like the decision boundary shown on the left plot of Figure 3.2. Although the plot on the left of Figure 3.2 does correctly classify each point with zero error, it is not the optimal solution. The search for a maximal separating margin decision boundary is the aim of support vector machine learning.

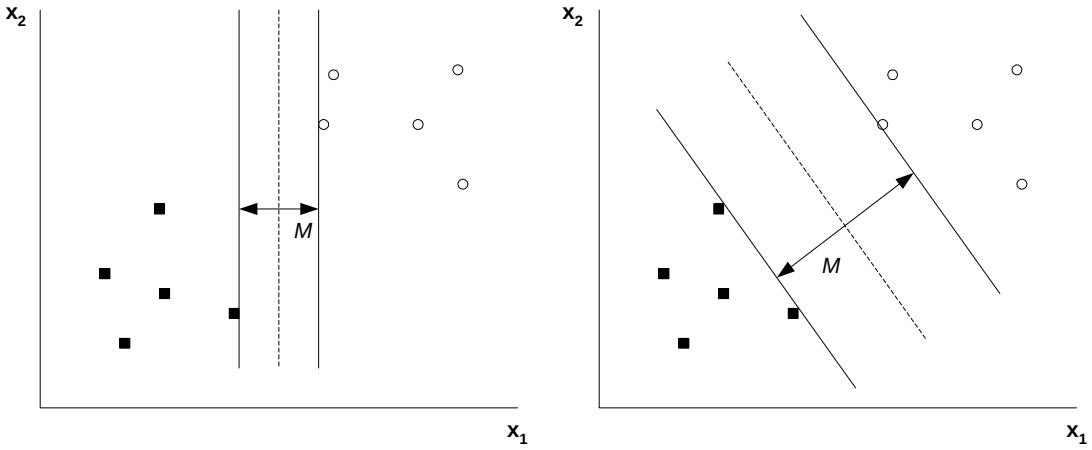

Figure 3.2: LEFT: Plot of two classes showing small margin between classes defined by the decision boundary. RIGHT: The same plot but with a decision boundary with the largest possible distance between opposite classes. This is the maximal margin.

3.2.1 Optimal Canonical Separating Hyperplane

The optimal canonical separating hyperplane (OCSH) is best illustrated for the 1-dimensional case. Consider the diagram in Figure 3.3. In this figure we have six points belonging to two classes. The decision boundary is the intersection between the decision function and the input space. In this particular example the decision boundary is a point. The decision function defines the boundary between the classes. The decision function shown passes through the two nearest points of each class. This is the optimal separating canonical hyperplane. The values of the points are obtained from the indicator function I_f. In this particular example $I_f(\mathbf{x}, \mathbf{w}, b) = \text{sign}(d(\mathbf{x}, \mathbf{w}, b))$ where $d(\mathbf{x}, \mathbf{w}, b)$ is the decision function (or boundary) and $d(\mathbf{x}, \mathbf{w}, b) = \mathbf{w}^T\mathbf{x} + b$. The two points that define the OCSH are the *support vectors*. This leads to a mathematical definition for the OCSH. The support vectors satisfy:

$$y_i[\mathbf{w}^T\mathbf{x}_i + b] \equiv 1, i = 1, 2, \ldots, N_{SV} \tag{3.1}$$

where N_{SV} is the number of support vectors. The OCSH must also satisfy the inequalities for all other training points:

$$y_i[\mathbf{w}^T\mathbf{x}_i + b] > 1, i = 1, 2, \ldots, l - N_{SV} \tag{3.2}$$

where l is the total number of training data points. This means that the absolute value of OCSH for non support vectors is > 1. This definition of support vectors and the inequalities for the OCSH are verified from inspection of Figure 3.3.

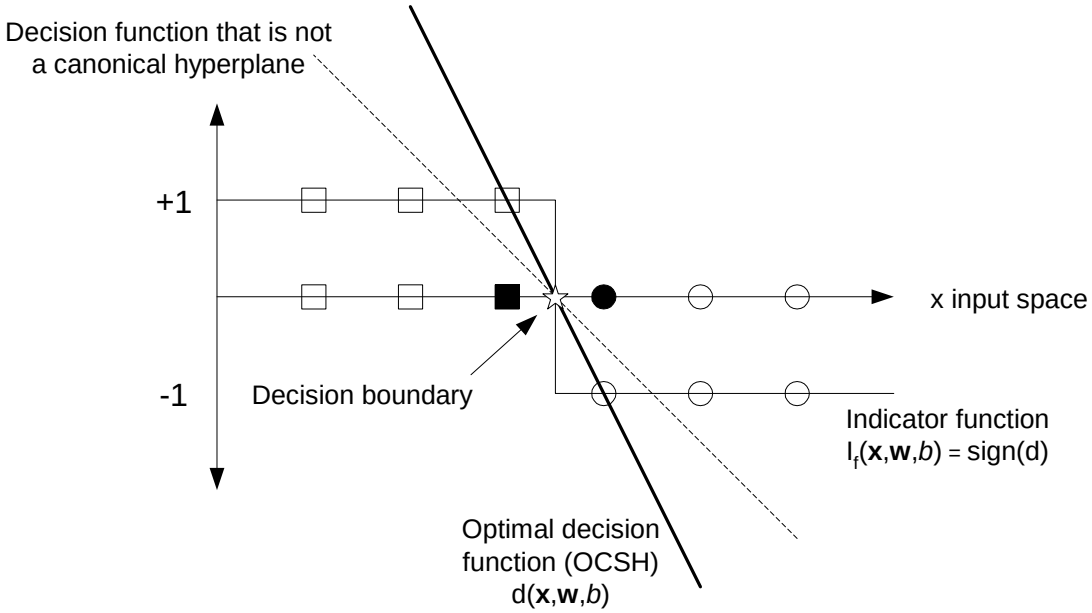

Figure 3.3: Optimal canonical separating hyperplane in 1-dimension. The OCSH passes through two support vectors, one from each class (adapted from [1]).

Figure 3.4 illustrates the maximal margin with the mathematical interpretation given above with the support vectors indicated with circles around the points. In this two class classification problem there are three support vectors. In fact, three is the minimum number of points that are needed to define an optimal separating hyperplane for the separable binary classification problem.

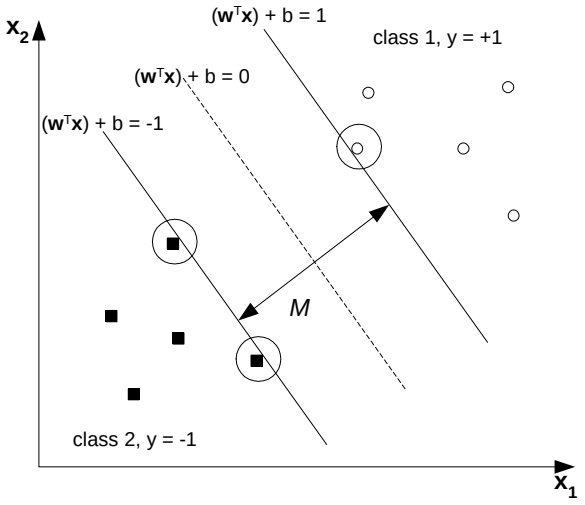

Figure 3.4: Illustration of the maximal margin and training points chosen as support vectors. The support vectors are indicated with circles around the points.

3.2.2 The Maximal Margin

Figure 3.5 shows a maximal margin and two (of three) support vectors. The length of the margin M is

$$M = (\mathbf{x}_1 - \mathbf{x}_2)_{\mathbf{w}} \tag{3.3}$$

where the subscript $\mathbf{w}$ indicates the projection of $(\mathbf{x}_1 - \mathbf{x}_2)$ onto the weights vector $\mathbf{w}$ so that $(\mathbf{x}_1 - \mathbf{x}_2)_{\mathbf{w}}$ is in the direction of the weights vector. From Figure 3.5 we have:

$$D_1 = \|\mathbf{x}_1\| \cos(\alpha) \tag{3.4a}$$

$$D_2 = \|\mathbf{x}_2\| \cos(\beta) \tag{3.4b}$$

$$M = D_1 - D_2 \tag{3.4c}$$

The angles α and β are given from the geometry of Figure 3.5 as:

$$\cos(\alpha) = \frac{\mathbf{x}_1^T \mathbf{w}}{\|\mathbf{x}_1\| \|\mathbf{w}\|} \tag{3.5a}$$

$$\cos(\beta) = \frac{\mathbf{x}_2^T \mathbf{w}}{\|\mathbf{x}_2\| \|\mathbf{w}\|} \tag{3.5b}$$

substituting (3.5) into (3.4) yields

$$M = \frac{\mathbf{x}_1^T\mathbf{w} - \mathbf{x}_2^T\mathbf{w}}{\|\mathbf{w}\|} \tag{3.6}$$

From Figures 3.4 and 3.5 $\mathbf{x}_1$ and $\mathbf{x}_2$ are support vectors. As they are support vectors they satisfy equation (3.1) ($y_i|\mathbf{w}^T\mathbf{x}_j + b| = 1$) so we have

$$\mathbf{x}_1^T\mathbf{w} + b = 1 \tag{3.7a}$$

$$\mathbf{x}_2^T\mathbf{w} + b = -1 \tag{3.7b}$$

Substitution of (3.7) into (3.6) yields a final result for M:

$$M = \frac{2}{\|\mathbf{w}\|} \tag{3.8}$$

Here we see that to maximise the margin M we must minimise the norm of the weight vector $\|\mathbf{w}\|$. The goal of the support vector machine is to calculate

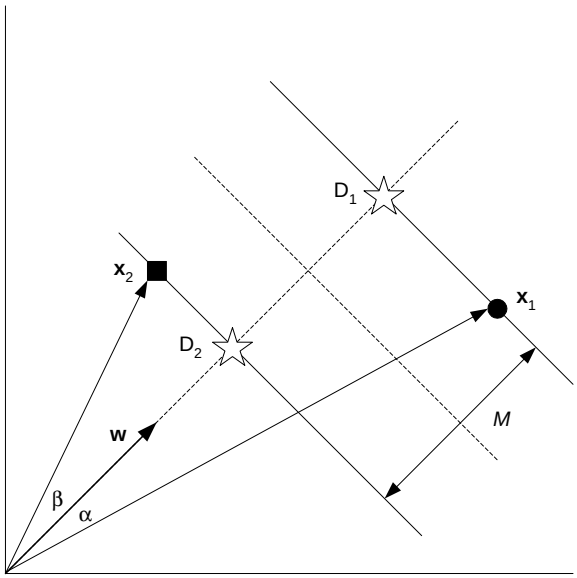

Figure 3.5: Parameters used in the computation of the maximal margin

the weight vector so that the distance between opposite classes is maximised. Using (3.8) and the mathematical definitions of support vectors and the optimal canonical separating hyperplane (equations (3.1) and (3.2)), and taking into

account that minimisation of $\|\mathbf{w}\|$ is equal to the minimisation of $\sum w_i^2$, produces a standard non-linear optimisation problem:

$$\min z = \sum w_i^2 = \mathbf{w}^T\mathbf{w} \qquad i = 1, 2, \ldots, l \tag{3.9a}$$

subject to

$$y_i[\mathbf{w}^T\mathbf{x}_i + b] \geq 1 \tag{3.9b}$$

The setting above is a standard quadratic programming problem.

3.2.3 Solution by Lagrangian Multipliers

Equation (3.9) is an optimisation problem of the type that is normally solved by forming the Lagrangian [23]. The Lagrangian is written:

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{l} \alpha_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1] \tag{3.10}$$

where the α_i are the Lagrange multipliers. This is now an optimisation problem without constraints where the objective is to minimise the Lagrangian $L(\mathbf{w}, b, \boldsymbol{\alpha})$. The problem is normally solved by transforming the Lagrangian into the dual.

For an optimisation problem of the form

$$\min f(x_1,x_2,\ldots,x_n) \tag{3.11a}$$

subject to

$$g_1(x_1,x_2,\ldots,x_n) \geq 1 \tag{3.11b}$$

$$g_2(x_1,x_2,\ldots,x_n) \geq 1 \tag{3.11c}$$

$$\vdots \tag{3.11d}$$

$$g_l(x_1,x_2,\ldots,x_n) \geq 1 \tag{3.11e}$$

then for $\bar{x} = (\bar{x_1}, \bar{x_2}, \ldots, \bar{x_n})$ to be a solution the following conditions must be met [23]:

$$\frac{\partial f(\bar{x})}{\partial x_j} + \sum_{i=1}^{l} \frac{\partial g_i(\bar{x})}{\partial x_j} = 0 \tag{3.12a}$$

$$\bar{\lambda}_i[b_i - g_i(\bar{x})] = 0 \tag{3.12b}$$

$$\bar{\lambda}_i \geq 0 \tag{3.12c}$$

These are the Karush-Kuhn-Tucker (KKT) conditions. Equation (3.12a) means that the partial differential of the Lagrangian L with respect to the two primal variables $\mathbf{w}_o$ and $\mathbf{b}_o$ must be zero. Thus in order for $(\mathbf{w}_o, b_o)$ to be an optimal solution of (3.10) we require:

$$\frac{\partial L}{\partial \mathbf{w}_o} = 0 \tag{3.13a}$$

$$\frac{\partial L}{\partial b_o} = 0 \tag{3.13b}$$

Application of the KKT conditions to (3.9) yields:

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}}\left[\frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{l} \alpha_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1]\right] = 0 \tag{3.14a}$$

$$= \mathbf{w} - \sum_{i=1}^{l} \alpha_i y_i \mathbf{x}_i = 0 \tag{3.14b}$$

therefore

$$\mathbf{w}_0 = \sum_{i=1}^{l} \alpha_i y_i \mathbf{x}_i \tag{3.14c}$$

Where $\mathbf{w}_o$ is the optimal value of $\mathbf{w}$ Similarly

$$\frac{\partial L}{\partial b} = \frac{\partial}{\partial b}\left[\frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{l} \alpha_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1]\right] = 0 \tag{3.15a}$$

$$= -\sum_{i=1}^{l} \alpha_i y_i = 0 \tag{3.15b}$$

therefore

$$\sum_{i=1}^{l} \alpha_i y_i = 0 \tag{3.15c}$$

Substitution of (3.14c) into (3.10) and taking (3.15c) into account:

$$L(\mathbf{w}_0, b_0, \boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w}_0^T\mathbf{w}_0 - \sum_{i=1}^{l} \alpha_i[y_i(\mathbf{w} + 0^T\mathbf{x}_i + b_0) - 1] \tag{3.16a}$$

$$= \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T\mathbf{x}_j - \sum_{i=1}^{l} \alpha_i y_i \mathbf{w}^T x_i - \sum_{i=1}^{l} \alpha_i y_i b_0 - \sum_{i=1}^{l} \alpha_i \tag{3.16b}$$

$$= \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T\mathbf{x}_j - \sum_{i=1}^{l} \alpha_i \tag{3.16c}$$

In matrix notation this is written

$$\min L(\boldsymbol{\alpha}) = 0.5\boldsymbol{\alpha}^T\mathbf{H}\boldsymbol{\alpha} - \mathbf{f}^T\boldsymbol{\alpha} \tag{3.17a}$$

subject to

$$\mathbf{y}^T\boldsymbol{\alpha} = 0 \tag{3.17b}$$

$$\boldsymbol{\alpha} \geq 0 \tag{3.17c}$$

where $\mathbf{H}$ is the Hessian matrix:

$$H_{ij} = y_i y_j \mathbf{x}_i^T\mathbf{x}_j \tag{3.18}$$

and $\mathbf{f}$ is a unit vector:

$$\mathbf{f} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \tag{3.19}$$

Equation (3.17a) is a standard quadratic optimisation problem. The primal variables $\mathbf{w}_0$ and b can be obtained from the solution $\boldsymbol{\alpha}_0$ to this problem from

(3.14c) and (3.13a):

$$\mathbf{w}_0 = \sum_{i=1}^{l} \alpha_{0i} y_i \mathbf{x}_i \tag{3.20a}$$

$$b_o = \frac{1}{N_{SV}} \left[\sum_{j=1}^{N_{SV}} \left(\frac{1}{y_j} - \mathbf{x}_j^T \mathbf{w}_0 \right) \right] \tag{3.20b}$$

3.2.4 Non-separable Classification

Equation (3.17a) holds for the separable case where the data can be separated without error. We now examine the non-separable case. Consider Figure 3.6. There is no line that can be drawn between the two classes that separates the data without misclassifying some data points. Stated mathematically there is no pair $(\mathbf{w}, b)$ such that

$$\mathbf{w}^T \mathbf{w} \leq = A^2 \tag{3.21a}$$

subject to

$$y_i[\mathbf{w}^T \mathbf{x}_i + b)] \geq 1 \tag{3.21b}$$

$$A > 0 \tag{3.21c}$$

where A is any non-zero value, holds true. Now the aim is to find the hyperplane that makes the smallest number of errors. Non-negative 'slack' variables $\xi_1, \xi_2, \ldots, \xi_l$ are introduced. These measure the deviation of the data from the maximal margin, thus it is desirable that the ξ_i be as small as possible. The geometric interpretation of the ξ_i is shown in Figure 3.7. The optimisation problem is now:

$$\min f(\mathbf{w}, \xi) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^{l} \xi_i \tag{3.22a}$$

subject to

$$y_i((\mathbf{w}^T \mathbf{x}_i) + b) \geq 1 - \xi_i \tag{3.22b}$$

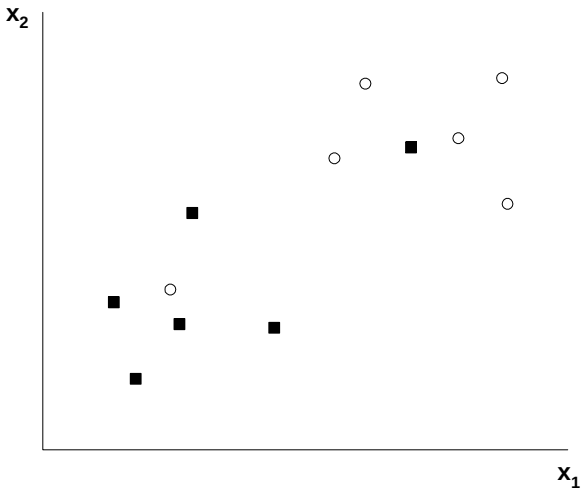

Figure 3.6: Non-separable binary classification. There is no line that can be drawn which separates the data without error.

Here C is a design parameter called the penalty parameter. The penalty parameter controls the magnitude of the ξ_i. An increase in C penalises larger errors (large ξ_i). However this can be achieved only by increasing the weight vector norm $\|\mathbf{w}\|$ (that we want to minimise). At the same time an increase in $\|\mathbf{w}\|$ does not guarantee smaller ξ_i. Forming the Lagrangian of (3.22a):

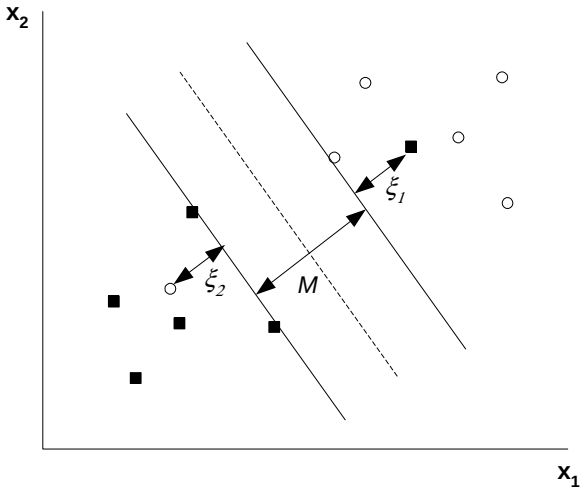

Figure 3.7: Non-separable binary classification showing the geometrical interpretation of ξ_i.

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i - \sum_{i=1}^{l}\alpha_i\Big[y_i((\mathbf{w}T\mathbf{x}_i) + b) - 1 + \xi_i\Big] \tag{3.23}$$

finding the partial derivatives with respect to $\mathbf{w}$:

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}}\left[\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i - \sum_{i=1}^{l}\alpha_i y_i \mathbf{w}^T\mathbf{x}_i - \sum_{i=1}^{l}\alpha_i y_i b + \sum_{i=1}^{l}\alpha_i - \sum_{i=1}^{l}\alpha_i\xi_i\right] \tag{3.24a}$$

$$= \mathbf{w} - \sum_{i=1}^{l}\alpha_i y_i \mathbf{x}_i \tag{3.24b}$$

rearranging for $\mathbf{w}$:

$$\mathbf{w} = \sum_{i=1}^{l}\alpha_i y_i \mathbf{x}_i \tag{3.24c}$$

$$\tag{3.24d}$$

Finding the partial derivatives with respect to b:

$$\frac{\partial L}{\partial b} = \frac{\partial}{\partial b}\left[\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i - \sum_{i=1}^{l}\alpha_i y_i \mathbf{w}^T\mathbf{x}_i - \sum_{i=1}^{l}\alpha_i y_i b + \sum_{i=1}^{l}\alpha_i - \sum_{i=1}^{l}\alpha_i\xi_i\right] \tag{3.25a}$$

$$= -\sum_{i=1}^{l}\alpha_i y_i \tag{3.25b}$$

or

$$\sum_{i=1}^{l}\alpha_i y_i = 0 \tag{3.25c}$$

Finding the partial derivatives with respect to ξ_i:

$$\frac{\partial L}{\partial \xi_i} = \frac{\partial}{\partial \xi_i}\left[\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i - \sum_{i=1}^{l}\alpha_i y_i \mathbf{w}^T\mathbf{x}_i - \sum_{i=1}^{l}\alpha_i y_i b + \sum_{i=1}^{l}\alpha_i - \sum_{i=1}^{l}\alpha_i\xi_i\right] \tag{3.26a}$$

$$= C - \sum_{i=1}^{l}\alpha_i \tag{3.26b}$$

or

$$\sum_{i=1}^{l} \alpha_i = 0 \tag{3.26c}$$

Substitution of (3.26c) into (3.23) and taking (3.25c) and (3.26c) into account:

$$\min \Big[-\frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j^T + \sum_{i=1}^{l} \alpha_i \Big] \tag{3.27a}$$

subject to

$$0 \leq \alpha_i \leq C \tag{3.27b}$$

$$\sum_{i=1}^{l} \alpha_i y_i = 0 \tag{3.27c}$$

We can see that optimisation problem (3.27a) is identical to (3.17a) with the exception of slightly different constraints. The equation can also be solved for the α_i using quadratic programming.

3.3 Function Approximation by SVM

Here we discuss the theory behind the application of SVM learning to regression (i.e. function approximation) problems. Regression is an extension of the non-separable classification such that each data point can be thought of as being in its own class. It is this application of SVM learning to regression problems is used in this thesis.

We are now approximating functions of the form

$$f(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{N} w_i \phi_i(\mathbf{x}) \tag{3.28}$$

where the functions $\phi_i(\mathbf{x})$ are termed kernel functions (alternatively features or basis functions) and N is the number of support vectors. In the case of regression

we use Vapnik's linear loss function with ϵ-insensitivity zone as a measure of the error of approximation:

$$\text{error } = |y - f(\mathbf{x}, \mathbf{w})| = \begin{cases} 0 \text{ if } |y - f(\mathbf{x}, \mathbf{w})| \leq \epsilon, \\ |y - f(\mathbf{x}, \mathbf{w})| - \epsilon, \text{ otherwise.} \end{cases} \tag{3.29}$$

Thus, the loss is equal to 0 if the difference between the predicted $f(\mathbf{x}, \mathbf{w})$ and the measured value is less than ϵ. Vapnik's ϵ-insensitivity loss function [19] defines an ϵ tube such that if the predicted value is within the tube the error is zero. For all other predicted points outside the tube, the error equals the magnitude of the difference between the predicted value and the radius ϵ of the tube. Figure 3.8 illustrates the geometrical interpretation of the ϵ tube along with all relevant mathematical objects to a regression problem. Note that for $\epsilon = 0$, Vapnik's loss function equals a least modulus (a.k.a. Huber's robust loss) function:

$$\text{error } = |y - f(\mathbf{x}, \mathbf{w})| \tag{3.30}$$

The total 'risk' or error is given by:

$$R_{emp}(\mathbf{w}, b) = \frac{1}{l} \sum_{i=1}^{l} |y_i - f(\mathbf{x}_i, \mathbf{w})|_\epsilon \tag{3.31a}$$

$$= \frac{1}{l} \sum_{i=1}^{l} |y_i - (\mathbf{w}^T \mathbf{x}_i + b)|_\epsilon \tag{3.31b}$$

$$= \frac{1}{l} \sum_{i=1}^{l} |y_i - \mathbf{w}^T \mathbf{x}_i - b|_\epsilon \tag{3.31c}$$

where $|*|_\epsilon$ is the Vapnik loss function given in (3.29). The goal is now to minimise

$$\min R = \frac{1}{2} \|\mathbf{w}\|^2 + C\Big(\sum_{i=1}^{l} |y_i - \mathbf{w}^T \mathbf{x}_i - b|_\epsilon \Big) \tag{3.32}$$

from inspection of Figure 3.8 and from the definition of (ξ_i, ξ_i^*), for data outside the insensitivity tube ϵ:

$$|y - f(\mathbf{x}, \mathbf{w})| - \epsilon = \xi \text{ for data above } \epsilon \text{ tube} \tag{3.33a}$$

$$|y - f(\mathbf{x}, \mathbf{w})| - \epsilon = \xi^* \text{ for data below } \epsilon \text{ tube} \tag{3.33b}$$

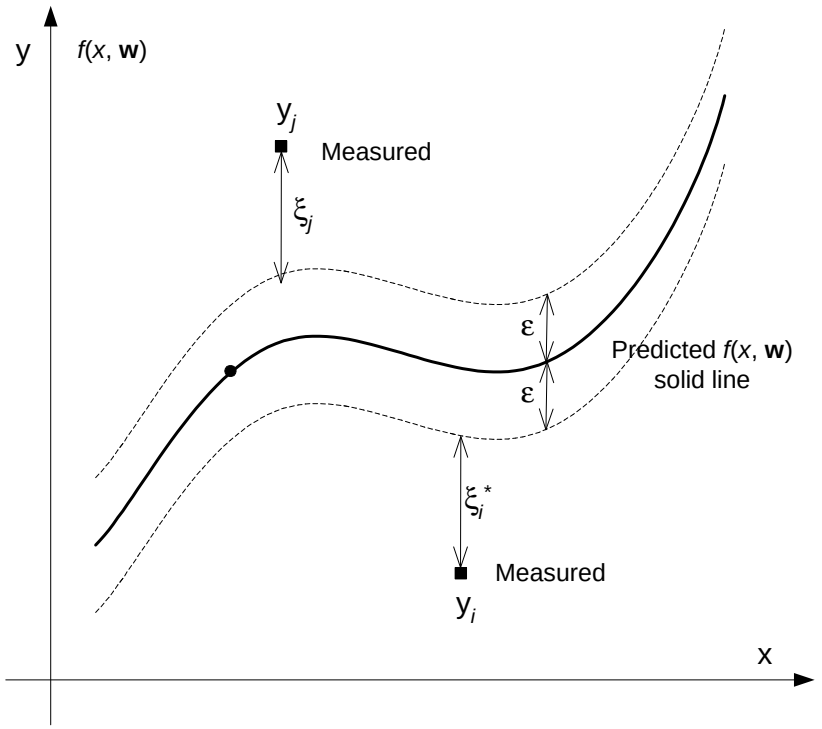

Figure 3.8: The parameters used in (1-dimensional) support vector regression (from [1]).

Equation (3.33a) is substituted into (3.32) so our optimisation problem is now to find w which minimises the 'risk' or error given by:

$$R_{\mathbf{w},\xi,\xi^*} = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i + \sum_{i=1}^{l}\xi_i^* \tag{3.34a}$$

subject to

$$y_i - (\mathbf{w}^T\mathbf{x}_i + b) \leq \epsilon + \xi_i \tag{3.34b}$$

$$(\mathbf{w}^T\mathbf{x} + b) - y_i \leq \epsilon + \xi_i^* \tag{3.34c}$$

$$\xi_i \geq 0 \tag{3.34d}$$

$$\xi_i^* \geq 0 \tag{3.34e}$$

where ξ_i and ξ_i^* are slack variables shown in Figure 3.8 for measurements 'above' and 'below' an ϵ-tube respectively and $\mathbf{x}$ is a Gaussian kernel. Forming the

Lagrangian of (3.34a):

$$\min L(\mathbf{w}, b, \xi, \xi^*, \alpha, \alpha^*, \beta, \beta^*) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i + C\sum_{i=1}^{l}\xi_i^* - \sum_{i=1}^{l}\alpha_i(y_i - \mathbf{w}^T\mathbf{x}_i - b + \epsilon + \xi_i^*) - \sum_{i=1}^{l}\alpha_i^*(\mathbf{w}^T\mathbf{x} + b - y_i + \epsilon + \xi_i^*) - \sum_{i=1}^{l}\beta_i\xi_i - \sum_{i=1}^{l}\beta_i^*\xi_i^* \quad (3.35)$$

where the β components arise due to the constraints $\xi \geq 0$ and $\xi^* \geq 0$. Finding the partial derivative with respect to $\mathbf{w}$:

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}}\left[\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i + C\sum_{i=1}^{l}\xi_i^* - \sum_{i=1}^{l}\alpha_i y_i + \sum_{i=1}^{l}\alpha_i\mathbf{w}^T\mathbf{x}_i + \sum_{i=1}^{l}\alpha_i b - \sum_{i=1}^{l}\alpha_i\epsilon - \sum_{i=1}^{l}\alpha_i\xi_i - \sum_{i=1}^{l}\alpha_i^*\mathbf{w}^T\mathbf{x}_i - \sum_{i=1}^{l}\alpha_i^* b + \sum_{i=1}^{l}\alpha_i^* y_i - \sum_{i=1}^{l}\alpha_i^*\epsilon - \sum_{i=1}^{l}\alpha_i^*\xi_i^* - \sum_{i=1}^{l}\beta_i\xi_i - \sum_{i=1}^{l}\beta_i^*\xi_i^*\right] = 0 \quad (3.36a)$$

$$= \mathbf{w} + \sum_{i=1}^{l}\alpha_i\mathbf{x}_i - \sum_{i=1}^{l}\alpha_i^*\mathbf{x}_i = 0 \quad (3.36b)$$

solving for $\mathbf{w}$:

$$\mathbf{w} = \sum_{i=1}^{l}(\alpha_i - \alpha_i^*)\mathbf{x}_i \quad (3.36c)$$

Finding the partial derivative with respect to b:

$$\frac{\partial L}{\partial b} = \frac{\partial}{\partial b}\left[\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i + C\sum_{i=1}^{l}\xi_i^* - \sum_{i=1}^{l}\alpha_i y_i + \sum_{i=1}^{l}\alpha_i\mathbf{w}^T\mathbf{x}_i + \sum_{i=1}^{l}\alpha_i b - \sum_{i=1}^{l}\alpha_i\epsilon - \sum_{i=1}^{l}\alpha_i\xi_i - \sum_{i=1}^{l}\alpha_i^*\mathbf{w}^T\mathbf{x}_i - \sum_{i=1}^{l}\alpha_i^* b + \sum_{i=1}^{l}\alpha_i^* y_i - \sum_{i=1}^{l}\alpha_i^*\epsilon - \sum_{i=1}^{l}\alpha_i^*\xi_i^* - \sum_{i=1}^{l}\beta_i\xi_i - \sum_{i=1}^{l}\beta_i^*\xi_i^*\right] = 0 \quad (3.37a)$$

$$= \sum_{i=1}^{l} \alpha_i - \sum_{i=1}^{l} \alpha_i^* = 0 \tag{3.37b}$$

rearranging:

$$\sum_{i=1}^{l} \alpha_i = \sum_{i=1}^{l} \alpha_i^* \tag{3.37c}$$

Similarly finding the partial derivative with respect to ξ and ξ^* yields the results:

$$\sum_{i=1}^{l} \beta_i + \sum_{i=1}^{l} \alpha_i \leq C \tag{3.38a}$$

$$\sum_{i=1}^{l} \beta_i^* + \sum_{i=1}^{l} \alpha_i^* \leq C \tag{3.38b}$$

Substitution of 3.36c into 3.36a and taking 3.37c and 3.38a into account yields:

$$L(\alpha, \alpha^*) = -\frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} (\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j) \mathbf{x}_i^T \mathbf{x}_j + \sum_{i=1}^{l} y_i(\alpha_i^* - \alpha_i) - \sum_{i=1}^{l} \epsilon_i(\alpha_i^* + \alpha_i) \tag{3.39a}$$

subject to

$$\sum_{i=1}^{l} \alpha_i = \sum_{i=1}^{l} \alpha_i^* \tag{3.39b}$$

$$0 \leq \alpha_i^* < C \tag{3.39c}$$

$$0 \leq \alpha_i < C \tag{3.39d}$$

In matrix notation equation (3.39a) is written:

$$\min L(\boldsymbol{\alpha}) = 0.5 \boldsymbol{\alpha}^T \mathbf{H} \boldsymbol{\alpha} - \mathbf{f}^T \boldsymbol{\alpha} \tag{3.40}$$

where

$$\mathbf{H} = [\mathbf{x}^T\mathbf{x} + 1] \tag{3.41a}$$

$$\mathbf{f} = \begin{pmatrix} \epsilon - y_1 \\ \epsilon - y_2 \\ \vdots \\ \epsilon - y_l \\ \epsilon + y_1 \\ \epsilon + y_2 \\ \vdots \\ \epsilon + y_l \end{pmatrix}^T \tag{3.41b}$$

Equations (3.41a) hold for the linear case where

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T\mathbf{x} + b \tag{3.42}$$

Our final goal is to solve non-linear regression problems. i.e problems of the type

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T G(\mathbf{x}) + b \tag{3.43}$$

where $G(\mathbf{x})$ is a non-linear mapping that maps input space $\mathbf{x}$ to feature space $G(*)$. The mapping $G(\mathbf{x})$ is normally the RBF design matrix given by:

$$\mathbf{G} = \begin{pmatrix} G(x_1, c_1) & \cdots & G(x_1, c_l) \\ \vdots & & \vdots \\ G(x_l, c_1) & \cdots & G(x_l, c_l) \end{pmatrix} \tag{3.44}$$

where $G(x, c)$ is the kernel function. Typically a Gaussian kernel function is used, given by (in 1-dimension):

$$G(x, c, \lambda) = \exp\left[-\frac{1}{2}\left(\frac{(x - c)}{\lambda}\right)^2 \right] \tag{3.45}$$

where

$$\begin{aligned} &x \text{ is the spacial coordinate} \\ &c \text{ is the centre of the Gaussians} \\ &\lambda \text{ is the Gaussian width (or shape parameter)} \end{aligned} \tag{3.46}$$

To solve non-linear regression problems the only change required is to the Hessian in (3.39a). The Hessian for non-linear regression problems is given as [1] :

$$\mathbf{H} = \begin{pmatrix} G(\mathbf{x}) & -G(\mathbf{x}) \\ -G(\mathbf{x}) & G(\mathbf{x}) \end{pmatrix} \tag{3.47}$$

where $G(\mathbf{x})$ is defined above. The weights vector w is found from

$$\mathbf{w} = \boldsymbol{\alpha}^* - \boldsymbol{\alpha} \tag{3.48}$$

and the bias b given by

$$b = \frac{1}{l} \sum_{i=1}^{l} (y_i - G(\mathbf{x}_i)) \tag{3.49}$$

Note that when a positive definite kernel (such as Gaussian or complete polynomial) is used the bias, b, equals zero. Also note that in (3.44) only the columns corresponding to the chosen support vectors (the ones having $\boldsymbol{\alpha}$'s bigger than zero) will be used.

3.4 Summary

In this chapter details on support vector machines have been presented, developing the theory from binary classification problems to non-linear regression problems. It is the application of support vector machines to solving non-linear regression problems that is used in image compression algorithms detailed in later chapters.

Chapter 4

SVM Modelled Colour Surface

Here we detail the most obvious technique which is to attempt to model the image's colour surface directly. Direct modelling of a colour surface using SVM learning was first presented in a paper by the author in [16]. The basic concept of using neural networks in image compression schemes revolves around constructing a network that can be trained on an image, then storing (or transmitting) the network parameters such that the number of data required to reconstruct the network (and hence the image) is much less than the number of data in the original raw image.

In directly modelling the colour surface spacial redundancy within the image is being exploited. This means that in parts of the image (e.g. background or large objects with little features) adjacent pixels have similar colour values. A neural network could potentially model these pixels.

4.1 SVM Model

In previous uses of neural networks applied to the image compression problem, the network topology is always predefined before compression (i.e. the number of Gaussians is set and fixed before learning starts). A different approach is

taken here as the learning process includes defining the best network topology for the image block being processed.

A SVM achieves this by choosing the minimum number of training points required to reproduce the learned function within a pre-defined error. For each training point chosen by the SVM there is a corresponding neuron. Hence the number of neurons is identical to the number of training points chosen. However only the support vectors will have non-zero weights. In this way those neurons whose weight is zero are ignored as they do not contribute to the outcome of the recovered image when the network is reconstructed.

A SVM network is identical to a Radial Basis Network (RBF) as illustrated in Figure 4.1. The inputs to the network are the physical location of the pixels (i.e. the x- and y-coordinates. The output of the network is the colour value of the pixel. Thus the training data set is of the form $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), , (\mathbf{x}_l, y_l)$ where $\mathbf{x}_i$ is the location of the pixel and y_i is the colour value. In modelling the colour surface with a SVM a function is being modelled of the form:

$$f(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{N} w_i \phi_i(\mathbf{x}) \tag{4.1}$$

where the functions $\phi_i(\mathbf{x})$ are the kernels or basis functions and $\mathbf{w}$ is the weights vector to be found.

4.1.1 Implementation

In the implementation Gaussian functions are used as the kernel functions. The Gaussians are of the form:

$$G(x, c) = e^{-\frac{(x-c)}{\lambda}^2} \tag{4.2}$$

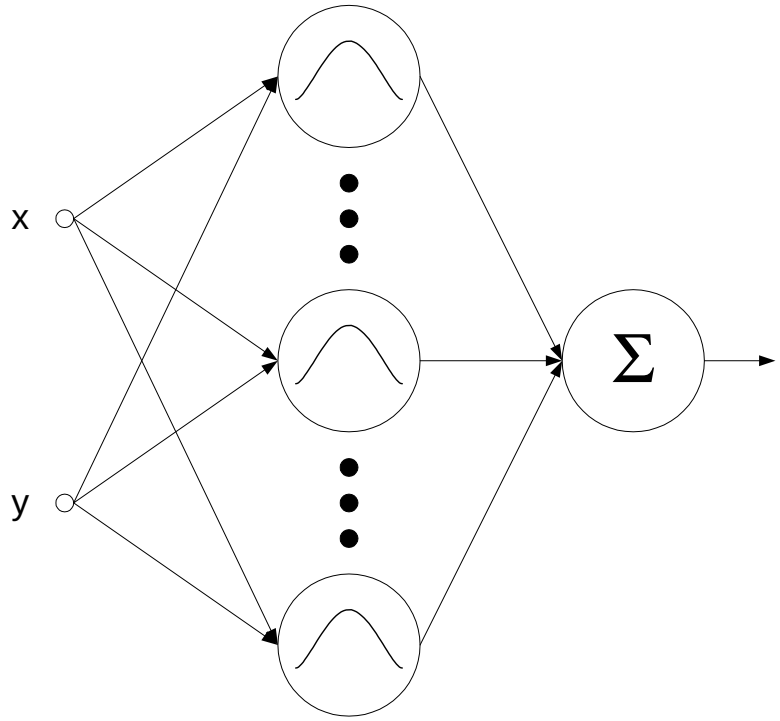

Figure 4.1: The RBF network used in a SVM learning algorithm. Shown is a 2-dimensional input vector and a network with m support vectors, resulting in m weights to be learned.

where

$$\begin{aligned} &\mathbf{X} = (x_1, \ldots, x_n) \text{ is the input vector} \\ &c_i \text{ are the centres of the Gaussians} \\ &\lambda \text{ is the Gaussian width (or shape parameter)} \end{aligned} \tag{4.3}$$

The function to be modelled is now of the form:

$$f(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{N} w_i e^{-\frac{1}{2}\frac{(\mathbf{x}-c_i)}{\lambda_i}^2} \tag{4.4}$$

The image is first subdivided into 8×8 blocks (64 pixels per block). Each block is processed independently of the other blocks. In a 256×256 image there are 1024 blocks thus 1024 networks need to be trained and their parameters stored. There are only three parameters which affect the compression which must be defined before learning can begin. These are the maximum allowed error

ϵ, termed the insensitivity zone in SVM terminology, the penalty parameter C and the Gaussian shape parameter λ. Here we set penalty parameter $C = \infty$.

A SVM is trained on each block using the (x, y) coordinate of each pixel as the input and the colour value of the corresponding pixel as the desired output. SVM learning is a batch training method so all inputs and outputs are processed simultaneously. As there are 64 pixels in each block and each pixel lies in 2-dimensional space our input to the support vector learning machine is a 64×2 matrix. Each pixel has a single colour value therefore the desired output is a 64×1 column vector. To restate, the input is:

$$\mathbf{X} = \Big(\mathbf{x}_1\mathbf{x}_2\Big) = \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ \vdots & \\ 8 & 1 \\ 1 & 2 \\ 2 & 2 \\ 3 & 2 \\ \vdots & \\ 7 & 8 \\ 8 & 8 \end{pmatrix} \tag{4.5}$$

and the desired output (training data) is

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{63} \\ y_{64} \end{pmatrix} \tag{4.6}$$

where the y_k are the colour values of the pixels. We note that for each block

the input matrix is identical. Thus in the implementation of the algorithm the input matrix only need to be computed once.

The actual implementation was written in MATLAB. In the implementation a non-linear optimisation problem is setup using the parameters described above. The 'RBF' design matrix (for more detail see chapter 3) is

$$\mathbf{G} = \begin{pmatrix} G(x_1, c_1) & \cdots & G(x_1, c_n) \\ \vdots & & \vdots \\ G(x_n, c_1) & \cdots & G(x_n, c_n) \end{pmatrix} \tag{4.7}$$

The non-linear optimisation problem is formulated as:

$$\min \left\{ -0.5\boldsymbol{\alpha}^T \mathbf{H} \boldsymbol{\alpha} + \mathbf{f}^T \boldsymbol{\alpha} \right\} \tag{4.8a}$$

where α_i are the Lagrange multipliers to be found, $\mathbf{H}$ is given by

$$\mathbf{H} = \begin{pmatrix} G & -G \\ -G & G \end{pmatrix} \tag{4.9}$$

and $\mathbf{f}$ is given by

$$\mathbf{f} = \begin{pmatrix} \epsilon - y_1 \\ \epsilon - y_2 \\ \vdots \\ \epsilon - y_{63} \\ \epsilon - y_{64} \\ \epsilon + y_1 \\ \epsilon + y_2 \\ \vdots \\ \epsilon + y_{63} \\ \epsilon + y_{64} \end{pmatrix} \tag{4.10}$$

We define $\mathbf{H}$ and $\mathbf{f}$ in this way as there are two constraints for each training data point: One constraint forcing the output to be $\leq y_i + \epsilon$ and a second constraint forcing the output to be $\geq y_i - \epsilon$. Defining the inputs in this way

results in the output as a 128×1 column vector. The weights are calculated from the output to the optimisation problem as discussed in chapter 3:

$$\mathbf{w} = \begin{pmatrix} \alpha_1 - \alpha_{65} \\ \alpha_2 - \alpha_{66} \\ \alpha_3 - \alpha_{67} \\ \vdots \\ \alpha_{63} - \alpha_{127} \\ \alpha_{64} - \alpha_{128} \end{pmatrix} \tag{4.11}$$

thus the final weight vector $\mathbf{w}$ is a 64×1 column vector.

This non-linear optimisation problem can be solved using quadratic programming. The MATLAB function 'QP' can be used for solving problems of this type. This will return a value for α from which we can compute the weights. The support vectors are indicated in $\mathbf{w}$ as having a non-zero weight value. That is, $\mathbf{w}$ will always be a 64×1 column vector but only the support vectors will have non-zero values.

4.1.2 Non-Negative Least Squares (NNLS)

The quadratic programming based algorithm is computationally intensive often requiring a few hours to run for each image. The bottle-neck in processing is solving the non-linear optimisation problem using quadratic programming. It is now shown that the problem can be transformed into a least squares optimisation problem and solved using the Non-Negative Least Squares (NNLS) algorithm as documented in [24]. The advantage of NNLS over quadratic programming is that far less computations are required to find a solution resulting in faster compression time. Using NNLS compared with QP reduced the time from a few hours to less than a minute for most images. MATLAB was used for the implementation of NNLS. Further gains in performance would be expected if the compression algorithm was optimised for speed and implemented in a compiled

language (such as C++ or Delphi) or implemented in a DSP (Digital Signal Processor).

4.1.3 Transformation of the QP problem to NNLS

When penalty parameter $C = \infty$ and positive definite kernels (e.g. Gaussian) are used the bias $b = 0$.

The original optimisation problem was

$$\max \left\{ L(\boldsymbol{\alpha}) = 0.5\boldsymbol{\alpha}^T \mathbf{H}\boldsymbol{\alpha} - \mathbf{f}^T\boldsymbol{\alpha} \right\} \tag{4.12a}$$

subject to

$$\mathbf{y}^T\boldsymbol{\alpha} = 0 \tag{4.12b}$$

$$C \geq \boldsymbol{\alpha} \geq 0 \tag{4.12c}$$

where $\mathbf{H}$ and $\mathbf{f}$ are given in (4.9) and (4.10). Working with penalty parameter $C = \infty$ and bias $b = 0$, the equality constraints (4.12b) and (4.12c) do not apply and the problem to solve is:

$$\max \left\{ L(\boldsymbol{\alpha}) = 0.5\boldsymbol{\alpha}^T \mathbf{H}\boldsymbol{\alpha} - \mathbf{f}^T\boldsymbol{\alpha} \right\} \tag{4.13a}$$

subject to

$$\boldsymbol{\alpha} \geq 0 \tag{4.13b}$$

This is a typical NNLS problem.

The quadratic programming optimisation problem in equation (4.13a) can

be transformed into a NNLS optimisation problems by setting $\mathbf{w} = \mathbf{F}\mathbf{x} - \mathbf{g}$:

$$\min \frac{1}{2}\|\mathbf{w}\|^2 = \frac{1}{2}\|\mathbf{F}\mathbf{x} - \mathbf{g}\|^2 \tag{4.14a}$$
$$= \frac{1}{2}(\mathbf{F}\mathbf{x} - \mathbf{g})^T(\mathbf{F}\mathbf{x} - \mathbf{g}) \tag{4.14b}$$
$$= \frac{1}{2}(\mathbf{F}^T\mathbf{x}^T - \mathbf{g}^T)(\mathbf{F}\mathbf{x} - \mathbf{g}) \tag{4.14c}$$
$$= \frac{1}{2}\left[\mathbf{x}^T\mathbf{x}\mathbf{F}^T\mathbf{F} - \mathbf{x}^T\mathbf{F}^T\mathbf{g} - \mathbf{g}^T\mathbf{F}\mathbf{x} + \mathbf{g}^T\mathbf{g}\right] \tag{4.14d}$$
$$= \frac{1}{2}\left[\mathbf{x}^T\mathbf{x}\mathbf{F}^T\mathbf{F} - (\mathbf{g}^T\mathbf{F}\mathbf{x})^T - \mathbf{g}^T\mathbf{F}\mathbf{x} + \mathbf{g}^T\mathbf{g}\right] \tag{4.14e}$$

noting that $\mathbf{g}^T\mathbf{g}$ is constant and will not affect the outcome of the minimisation problem:

$$= \min \frac{1}{2}\left[\mathbf{x}^T\mathbf{F}^T\mathbf{F}\mathbf{x} - 2\mathbf{g}^T\mathbf{F}\mathbf{x}\right] \tag{4.14f}$$
$$= \min \frac{1}{2}\mathbf{x}^T\mathbf{B}\mathbf{x} + \mathbf{a}^T\mathbf{x} \tag{4.14g}$$

Where

$$\mathbf{B} = \mathbf{F}^T\mathbf{F} \tag{4.14h}$$
$$\mathbf{a} = -\mathbf{g}\mathbf{F}^T \tag{4.14i}$$

Equation (4.14g) has the same form as (4.12a) hence we have shown that for suitable choices of $\mathbf{F}$ and $\mathbf{g}$ the optimisation problem is now linear:

$$\min \mathbf{w} = \frac{1}{2}[\mathbf{F}\mathbf{x} + \mathbf{g}] \tag{4.15}$$

Substitution of (4.14h) and (4.14i) into (4.12a) yields

$$\mathbf{B} = \mathbf{H} \text{ and } \mathbf{a} = -\mathbf{f} \tag{4.16}$$

so our optimisation problem is now:

$$\min \frac{1}{2}\left[\mathbf{H}\boldsymbol{\alpha} - \mathbf{f}\right] \tag{4.17a}$$

subject to

$$\boldsymbol{\alpha} \geq 0 \tag{4.17b}$$

Equation (4.17) is a least squares problem with linear inequality constraints. It can be solved using the non-negative least squares (NNLS) algorithm as first described in [24]. The advantage of this approach is that NNLS is much faster (by orders of magnitude) than quadratic programming. Finding $\boldsymbol{\alpha}$ using the NNLS over quadratic programming reduced the time required to compress an image from a couple of hours to 1-2 minutes.

Note that equations (4.17) are strictly valid only for positive definite kernels (bias $b = 0$) and for penalty parameter $C = \infty$.

4.2 Results

Figure 4.2 shows plots of the Lena image after SVM learning. It can be seen that the SVM learning has introduced unwanted 'artifacts' into the image. These are visible as bands of darker colour between blocks. Examination of the 3-dimensional surface of one block (Figure 4.3) reveals what is happening. It can be seen that the SVM is tending to place Gaussians in the centre of the block so that while the error is still less than the predefined ϵ it is a non-random error which is easily visible to the human eye. One reason why the SVM is doing this is that the average colour value of each block is > 0. In order to reduce this artifact we can introduce a 'bias' term. The bias is the average colour of the block and is computed from:

$$b = \frac{\sum_{i=1}^{64} y_i}{64} \tag{4.18}$$

where b is the bias (average colour) for a block of 64 pixels. This value is computed for each block and is subtracted from each pixel colour value in the block before SVM learning. In the reconstruction of the block it must be added

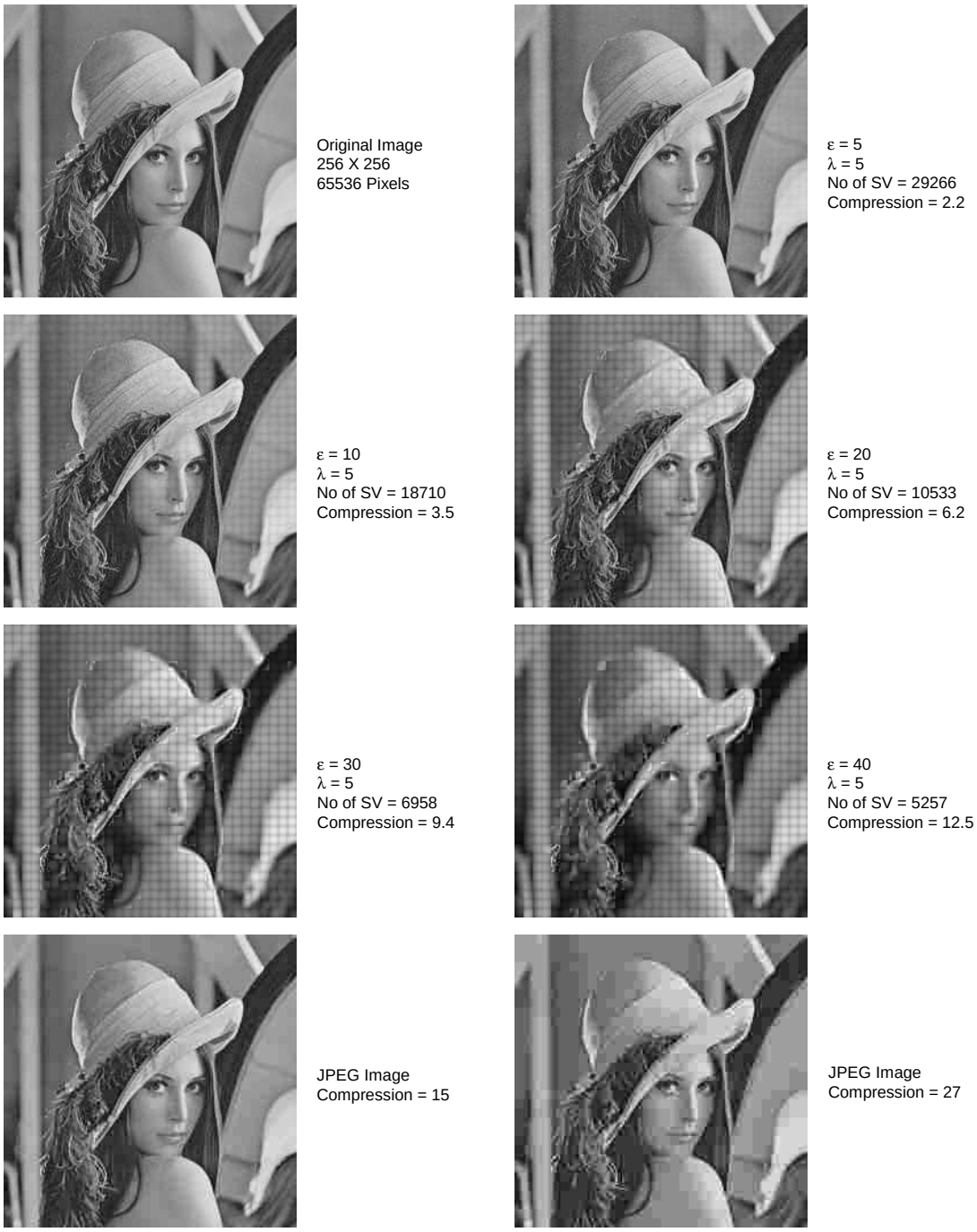

Figure 4.2: 256×256 Lena image with various values of error ϵ with no bias component for each block. The compression given is estimated based on an average of 8 bits (1 byte) per support vector using a Huffman encoding scheme. The bottom two images are compressed using JPEG for comparison.

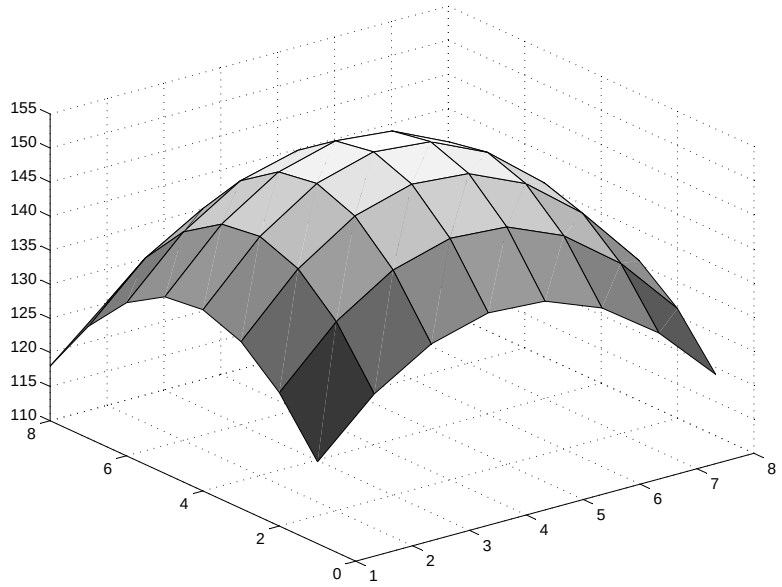

Figure 4.3: 8×8 block from the top left of the Lena image (with no bias) illustrating that the SVM is placing Gaussians at the centre of the block resulting in the maximum error being at the edges of the block. This is visible as dark bands between blocks in figure 4.2.

back to each pixel, hence by introducing the bias we also introduce an extra overhead component. There is one bias for each block, thus for a 256×256 image a further 1024 data elements need to be stored and for a 512×512 image there are an extra 4096 elements.

To see how the width of the Gaussian influences the compression, Figure 4.4 is a plot of Gaussian width versus number of support vectors for the 512×512 Lena image. The figure indicates that the optimal value for the Gaussian width is around 5 if bias is used, and about 60 if bias is not used. This figure also illustrates the difference in the number of support vectors when bias is used. The use of bias significantly reduces the number of support vectors required to reconstruct the image. Even when allowance is made for the extra overhead in storing the bias elements (the line labelled *With Bias (inc Bias)* in Figure 4.4) the total number of elements required to reconstruct the image is less than if no bias had been used. Figures 4.7 and 4.8 show the reconstructed image for

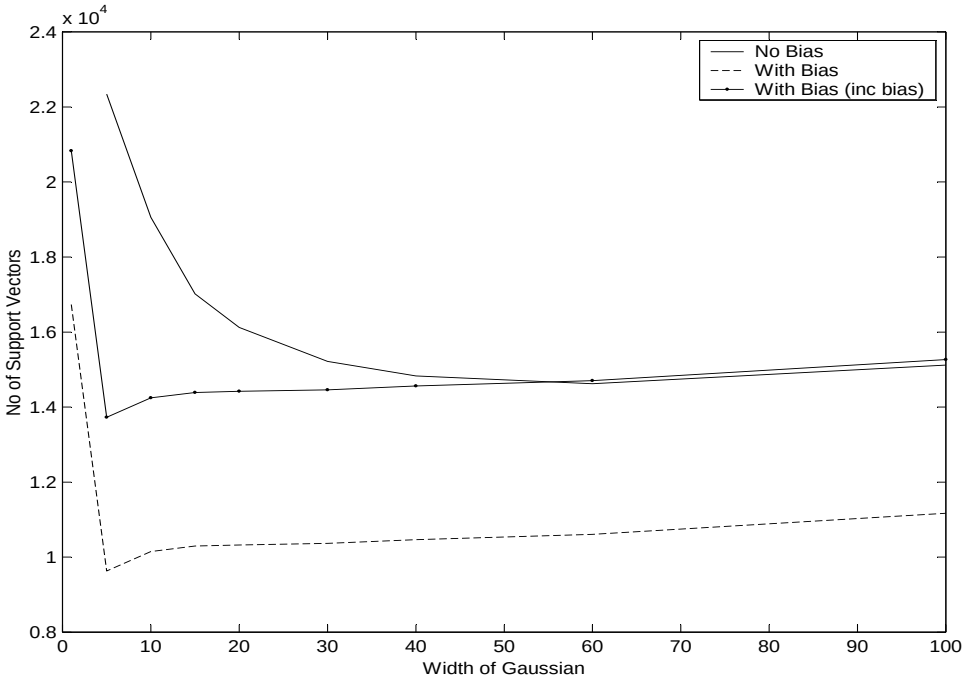

Figure 4.4: Plot of number of support vectors versus Gaussian width for the 512×512 Lena image with insensitivity $\epsilon = 20$. The line labelled *With Bias (inc Bias)* is a plot which includes the storage overhead for the bias components of each block. In this particular case an extra 4096 data elements are required to store the bias components.

the 256×256 and 512×512 Lena images respectively. Figure 4.9 shows the 400×400 Claudia Image. Images compressed using JPEG are shown in these figures for comparison. Inspection of the images in these three figures shows that JPEG outperforms the SVM surface modelling method in every case i.e. for the same compression ratio, JPEG produces a higher quality image. This is further visible in Figures 4.5 and 4.6 where the signal to noise ratio is plotted against compression. To objectively measure image quality, the signal to noise ratio (SNR) is used. The SNR is calculated using:

$$SNR = 10 \log \left[\frac{\sum_{ij} \text{Original_Image}_{ij}{}^2}{\sum_{ij} \left(\text{Original_Image}_{ij} - \text{Recovered_Image}_{ij} \right)^2} \right] \tag{4.19}$$

Figures 4.5 and 4.6 show that JPEG has better image quality for all compression ratios.

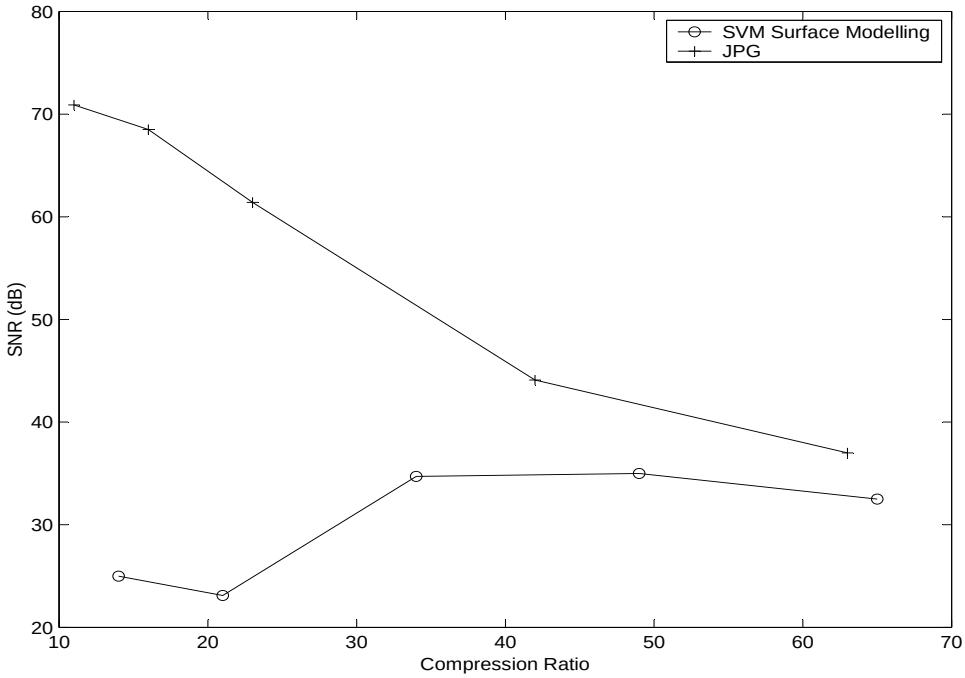

Figure 4.5: Plot of SNR versus compression ratio for the 400×400 Claudia image

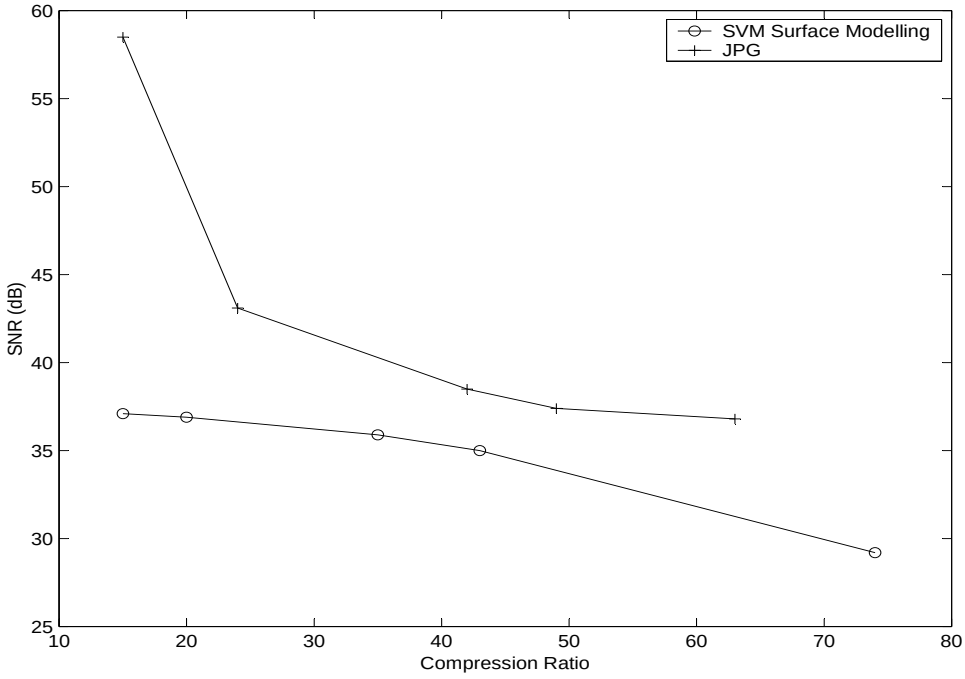

Figure 4.6: Plot of SNR versus compression ratio for the 512×512 Lena image

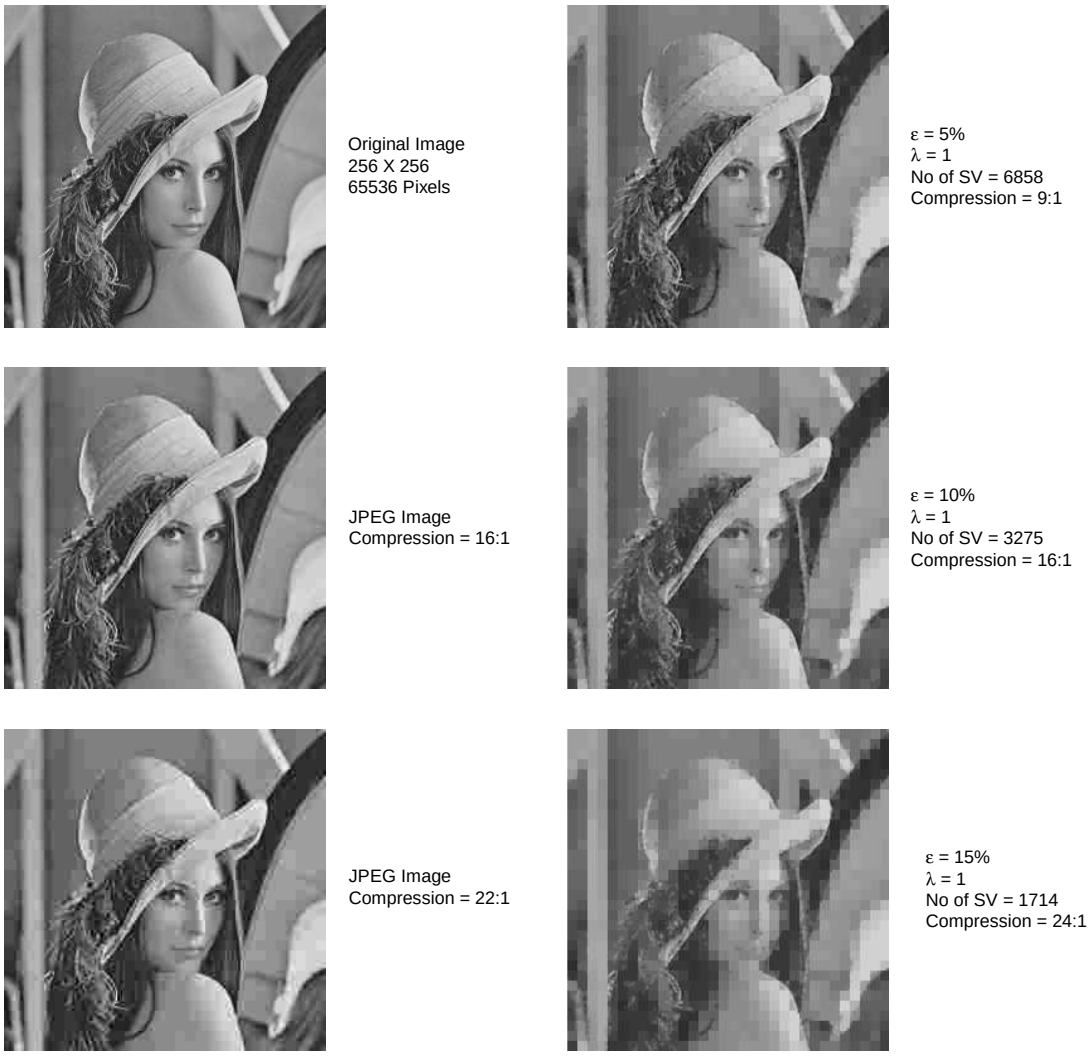

Figure 4.7: 256×256 Lena image with various values of error ϵ and with a bias component for each block. The JPEG images are shown for comparison. It can be seen that JPEG performs better than SVM surface modelling across all compression ratios.

4.3 Implementation

The compression of the SVM surface modelled images in Figures 4.7, 4.8 and 4.9 was computed from an actual binary file containing all information necessary to recover an approximated version of the original image. The compression is

Figure 4.8: 512×512 Lena image with various values of error ϵ and with a bias component for each block. Here it is seen that JPEG appears to perform better at lower compression ratios. At higher compression ratios, image quality is similar between JPEG and SVM surface modelling.

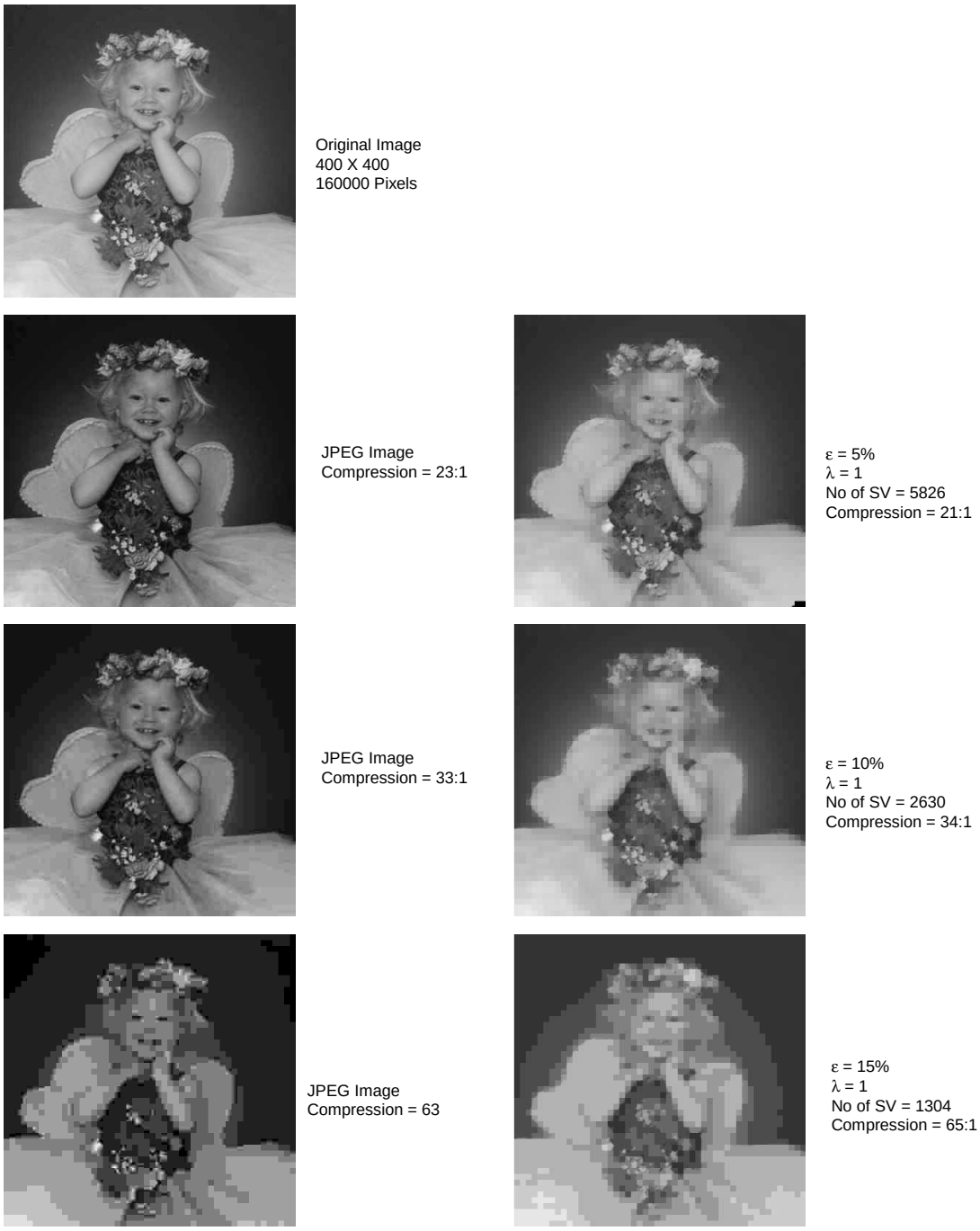

Figure 4.9: 400×400 Claudia image with various values of error ϵ and with a bias component for each block. The images on the left are compressed using JPEG for comparison. Image quality of the JPEG images appears better than SVM surface modelling at lower compression rates. At higher compression rates image quality is similar between JPEG and SVM surface modelling.

computed by

$$\text{compression} = \frac{\texttt{no_of_pixels}}{\texttt{file_size}} \tag{4.20}$$

where `file_size` is in bytes.

In order to recover each 8×8 block parameters are required to be stored so that the RBF network produced by SVM learning can be reconstructed. The parameters required to recover each network are:

1. The support vectors (i.e. the (x, y) position of each training point chosen as a support vector
2. The weights of each support vector
3. The bias for each block

4.3.1 Run Length and Huffman Encoding

The aim of some of the steps described above is to introduce as many zeros into the data as possible. This is no accident. This is done to make the data more compressible when Run Length Encoding (RLE) and Huffman encoding is applied. A combination of Huffman coding and RLE is used to create a binary image file.

Run length encoding is perhaps the simplest method of compression. For example, if an image contains a row of pixels with colours:

$$10\ 11\ 13\ 13\ 13\ 13\ 13\ 13\ 13\ 11\ 11\ 11\ 11\ 11\ 11 \tag{4.21}$$

the row of pixels could be stored as:

$$10\ 00\ 11\ 00\ 13\ 06\ 11\ 05 \tag{4.22}$$

where the first number is the pixel colour followed by the number of times it is repeated.

Huffman coding was invented in 1952 by D. A. Huffman [25]. It is a variable length entropy encoding scheme which makes use of the frequency of occurrence of symbols to be encoded. For example the letter e in the English language occurs far more frequently than the letter q. Therefore it makes sense to encode e with fewer data bits than q. A similar principal is followed in Morse code where e is represented by '.' and q is represented by '--.-'. The trick with variable length encoding is that each code must not be identical to the first part of a longer code. For example, for a set of three symbols (A, B, C) with the codes

$$A = 0$$

$$B = 110$$

$$C = 11110$$

it can be seen that the sequence

$$00011000110111100$$

is an encoding of

$$AAABAABCA \tag{4.23}$$

However there can be no code for D of 1100 because during the decoding phase it would not be possible to distinguish between ($D = 1100$) and ($B = 110$, $A = 0$).

Huffman coding is usually explained by developing a tree to which leaves (or nodes) are added based on the probability of each symbol's occurrence. Another approach is to generate the codes in two phases is taken here. Firstly generate the code lengths for each symbol, the secondly, generate the actual codes from the code lengths. In this implementation, an algorithm is used which essentially combines these two algorithms. The algorithm is detailed in Figure 4.10.

An example of Huffman encoding is shown in Figure 4.11. In this example a string of 21 symbols is Huffman encoded resulting in a string of 52 bits. An 'uncompressed' string of 21 symbols with 8 bits per symbol (i.e. 1 byte) would require 168 bits, thus the string has been 'compressed' approximately $3.2 : 1$.

An interesting feature of this algorithm is that in order to store the Huffman table (i.e. a list of codes and their corresponding values) only the number of codes of a given length and the decoded values need to be stored. This is illustrated in Figure 4.12. Here it is seen that the number of codes of each length are stored. The actual Huffman code is generated from these lengths. This reduces the amount of space required to store the table, resulting in increased compression compared with the case where the actual codes are stored. Huffman codes can become quite long (up to 20 bits in some of the images) and to store each code would require 20 bits per code.

```
PHASE 1:

    Create a list of nodes from the set of symbols with
    increasing probability. Probability here means the number
    of times the symbol has occurred.

    Repeat

        Identify the two nodes with the lowest probability.
        Call them Node_A and Node_B.

        Create a new node with
        probability := Node_A.probability + Node_B.probability.

        Mark Node_A and Node_B as 'Joined' so they cannot be
        selected again.

        Link the new node to Node_A and Node_B as the 'Parent'
        node. i.e. Node_A is the left child node and Node_B
        is the right child node to the new node.

    Until only one 'unjoined' node left. This is the root_node.

PHASE 2:

    Call GetListOfNodes( nil, root_node )

    Procedure GetListOfNodes( CurrentCode ; Node )
    begin

        If there is a node on the left then
            call GetListOfNodes( CurrentCode+'0' , Node.LeftNode );

        If there is a node of the right then
            call GetListOfNodes( CurrentCode+'1' , Node.RightNode );

        Create new symbol in the list of symbols with parameters
            Symbol.Symbol := Node.Symbol;
            Symbol.Code := CurrentCode;

    end;
```

Figure 4.10: Encoding algorithm to generate Huffman codes.

```
Input String: A MAN A PLAN A CANAL
Symbol A       = 01 [Occurrences = 7]
Symbol (space) = 00 [Occurrences = 7]
Symbol M       = 1110 [Occurrences = 1]
Symbol N       = 100 [Occurrences = 3]
Symbol P       = 1111 [Occurrences = 1]
Symbol L       = 110 [Occurrences = 2]
Symbol C       = 101 [Occurrences = 1]
No of codes: 7
No of bits: 52

Coded string:
0100111001100000100111111001100000100101011000111000

Total number of bits: 52
Number of symbols: 21
```

Figure 4.11: Huffman encoding example.

4.3.2 Encoding the Biases

The biases are encoded in a similar fashion to 'DC' discrete cosine coefficients in the JPEG algorithm (see Chapter 6 for an explanation of the discrete cosine transform). A bias sets the average pixel colour of its associated block of pixels. Because of this the human eye notices errors in the biases more readily than errors in higher frequency components, hence the biases must be preserved more accurately than the neural network weights from which the higher frequency components are recovered.

Biases are encoded as the difference between the current value and the previous value. Differences are encoded rather than the full value as the differences tend to be smaller than full values. A bias is encoded in two steps. The first step encodes the magnitude as a Huffman code, and the second step adds 'literal bits'. Literal bits are not encoded – they contain raw data. The number of literal bits is equal to the magnitude value. Table 4.3.2 lists the Huffman encoded magni-

```
=======Biases Huffman Table=======
Max code length: 5
No of codes of length 1 = 0  [Raw data: 0000000]
No of codes of length 2 = 3  [Raw data: 0000011]
No of codes of length 3 = 1  [Raw data: 0000001]
No of codes of length 4 = 1  [Raw data: 0000001]
No of codes of length 5 = 2  [Raw data: 0000010]

No of codes: 7

[Length= 2] Code: 00  -  Value = 0  [Raw data: {0}00000000]
[Length= 2] Code: 01  -  Value = 2  [Raw data: {0}00000010]
[Length= 2] Code: 10  -  Value = 1  [Raw data: {0}00000001]
[Length= 3] Code: 110  -  Value = 3  [Raw data: {0}00000011]
[Length= 4] Code: 1110  -  Value = 4  [Raw data: {0}00000100]
[Length= 5] Code: 11110  -  Value = 5  [Raw data: {0}00000101]
[Length= 5] Code: 11111  -  Value = 6  [Raw data: {0}00000110]
```

Figure 4.12: Binary storage of a Huffman table. Only the number of codes of a certain length and the corresponding decoded values are stored. The actual Huffman codes can be generated by knowing only how many codes of each length are required. This example is the Huffman table for the biases from an image of Claudia.

tude values (0–11) with the corresponding magnitude of the change in DC. For example to encode the number 58, the magnitude value is the Huffman code for 6, followed by 6 literal bits 011010 (the difference between 58 and 32 in binary).

4.3.3 Encoding Weights

Each 8×8 block has 63 weights. Where the weight is not a support vector it's value is set to zero. This means we do not have to explicitly send the position of the support vectors.

It was found that for shape parameter $\lambda \gtrsim 1$ the weights tend to have a larger dynamic range. For $\lambda \gtrsim 1.5$ weights often have a value of magnitude 10^9. It was also found than high magnitude weights like this needed to be

Table 4.1: Huffman encoding biases. Each bias is assigned a number in the 'Encoded Value' column which matches the 'Bias Value Range' of the bias.

Encoded Value	Bias Value Range
0	0
1	1,-1
2	3,2,-2,-3
3	7 ... 4,-4 ... -7
4	15 ... 8,-8 ... -15
5	31 ... 16,-16 ... -31
6	63 ... 32,-32 ... -63
7	127 ... 64,-64 ... -127
8	255 ... 128,-128 ... -255
9	511 ... 256,-256 ... -511
10	1023 ... 512,-512 ... -1023
11	2047 ... 1024,-1024 ... -2047

accurately transmitted to around nine significant figures. Generally no two (non-zero) weights had the same value. Encoding weights of this magnitude posed a problem as sending many nine digit numbers requires a large file resulting in low (even negative) compression. However when $\lambda \lesssim 1.1$ the value of an individual weight was generally in the region $-1 \ldots 1$ and needed to be encoded with an accuracy of one or two significant digits.

The first step in encoding the weights is to quantise them. That is decide how many quantisation levels to set (a parameter of the compression algorithm), define the value of each quantisation level and reassigning each weight to the closest matching quantisation level. The steps taken to quantise the weights are:

1. Find the maximum and minimum weight values. Call these `max` and `min`.
2. Find the difference, d, between quantisation levels by $d = \frac{\texttt{max}-\texttt{min}}{n}$ where n is the number of quantisation levels.
3. Set lowest quantisation level, $q_1 = \texttt{min}$.
4. Set remaining quantisation levels by $q_m = q_{m-1} + d$, until $q_n = \texttt{max}$.

5. Reassign each weight the value of the closest matching quantisation level q_m.

Care needs to be taken over the number of quantisation levels chosen. Figure 4.13 shows what happens to an image if too few quantisation levels are chosen. It can be seen that the image loses high frequency detail in addition to obvious distortions around edges. The steps of setting weights equal to zero for non-

20 Quantisation Levels 60 Quantisation Levels

Figure 4.13: Two images showing the effect of choosing too few quantisation levels. It can clearly seen that when too few quantisation levels are chosen, the recovered neural network is unable to correctly model edges in the image.

support vectors and the quantising step above introduces many zeros into the stream of weights to be stored. The quantised weights are encoded using a combination of run length and Huffman encoding. Two Huffman tables are used. One table encodes the actual value of the quantised weights and the other encodes the number of repeat zeros when a weight has a zero value. Figure 4.14 shows how these two table encode the weights. The figure shows one table encoding the actual value of the quantised weights and the other table encoding the number of repeat zeros after a weight with a zero value has been encountered. Note that Huffman codes from both tables are embedded in the final output

stream.

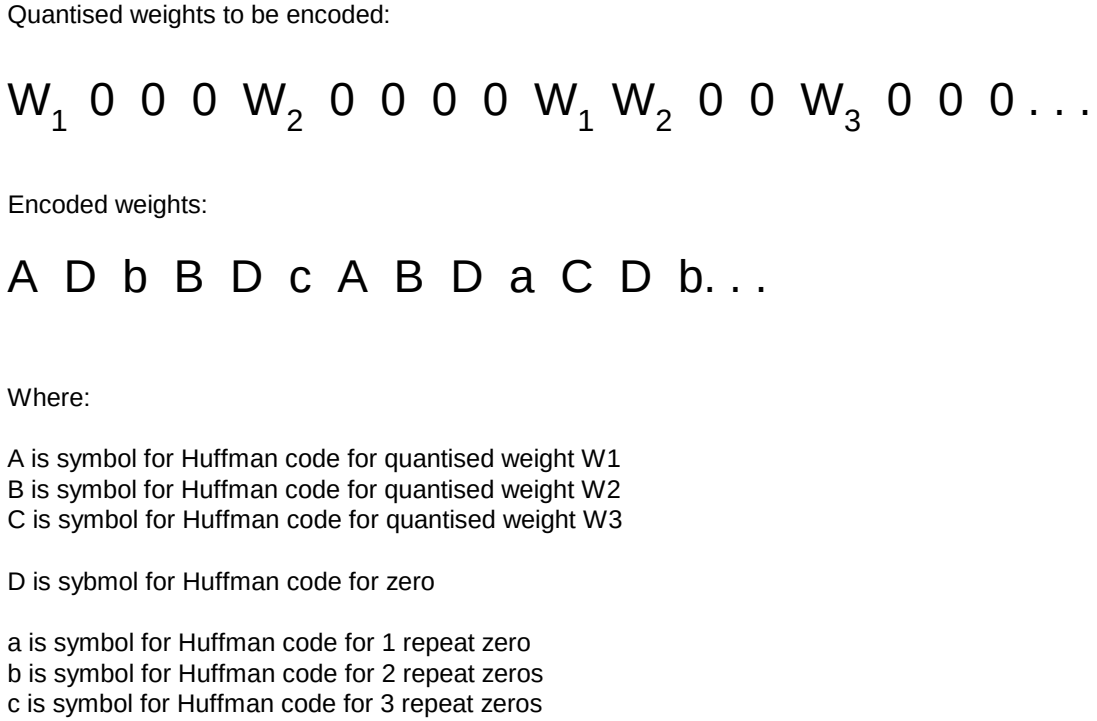

Figure 4.14: Illustration of how the weights are Huffman encoded using two Huffman tables. Note that in a real file there are generally far more zeros between weights than shown above.

4.4 Discussion

An algorithm has been detailed which uses SVMs to model the colour surface of an image. The SV optimisation problem was solved using NNLS to find the support vectors. The algorithm was implemented so that a binary file was produced containing all information (including Huffman tables) to recover an image. Vector quantisation was applied to the weights followed by a combination of Huffman and run length encoding.

Results show that while this algorithm is more useful when applied to image compression compared with LOMON and LOMON-SVM (discussed in the next Chapter), it performs poorly compared with JPEG.

Chapter 5

LOMON combined with SVM

5.1 Introduction

In this chapter the concept of the LOMON algorithm presenting in chapter 2 are combined with the application of SVMs to surface modelling discussed in chapter 4. The main difference between this algorithm and the previous SVM surface modelling algorithm in chapter 4 is that here the blocks are irregular in shape. The algorithm in this chapter attempts to group pixels together with similar values before applying SVM learning. Contrast this with the previous algorithm where the image is sub-divided into 8×8 blocks.

5.2 The LOMON-SVM Algorithm

In the previous chapter on applying SVMs to the modelling of a colour surface we worked with fixed 8×8 blocks of pixels. A variation of LOMON is used such that LOMON is used to define the block sizes to which SVM learning is then applied. There are two phases to the LOMON-SVM algorithm. The first phase uses the LOMON strategy to sub-divide the image into blocks such that in each block the error in an individual pixel is no greater than a predefined error ϵ. The

second phase is to apply SVM learning to each block.

The division of the image into blocks is illustrated graphically in Figure 5.1. The blocks are divided as follows:

1. Start with one selected block. This is the whole image.
2. Create two images by dividing the selected block into two. One image is created by a vertical separation, the other by a horizontal separation.
3. Find the average value of each block – call this b. Compute the RMS error for each block in the image by adding the sum of the squares of the error between each pixel value and b.
4. Select the image with the overall lowest RMS value.
5. For the two blocks created in step 2 find the block with the highest absolute error. That is to find the block containing the largest value of $(original_pixel_colour_i - b)^2$ where $original_pixel_colour_i$ is the value of an individual pixel in the original image.
6. If the largest absolute value is greater than a predefined error, ϵ then stop dividing, otherwise repeat step 2.

The second phase of the LOMON-SVM algorithm involves using SVMs as described in the previous chapter on SVM surface modelling. We are now using the regions found using LOMON to define the input training vector from the original image. Pseudo-code for implementation of both phases of the algorithm is illustrated in Figure 5.2. In the LOMON-SVM algorithm there are three user-definable parameters:

1. The maximum acceptable error ϵ for the phase I (LOMON).
2. The maximum acceptable error ϵ_{SVM} for the SVM learning phase.
3. The width of the Gaussian in the SVM learning phase.

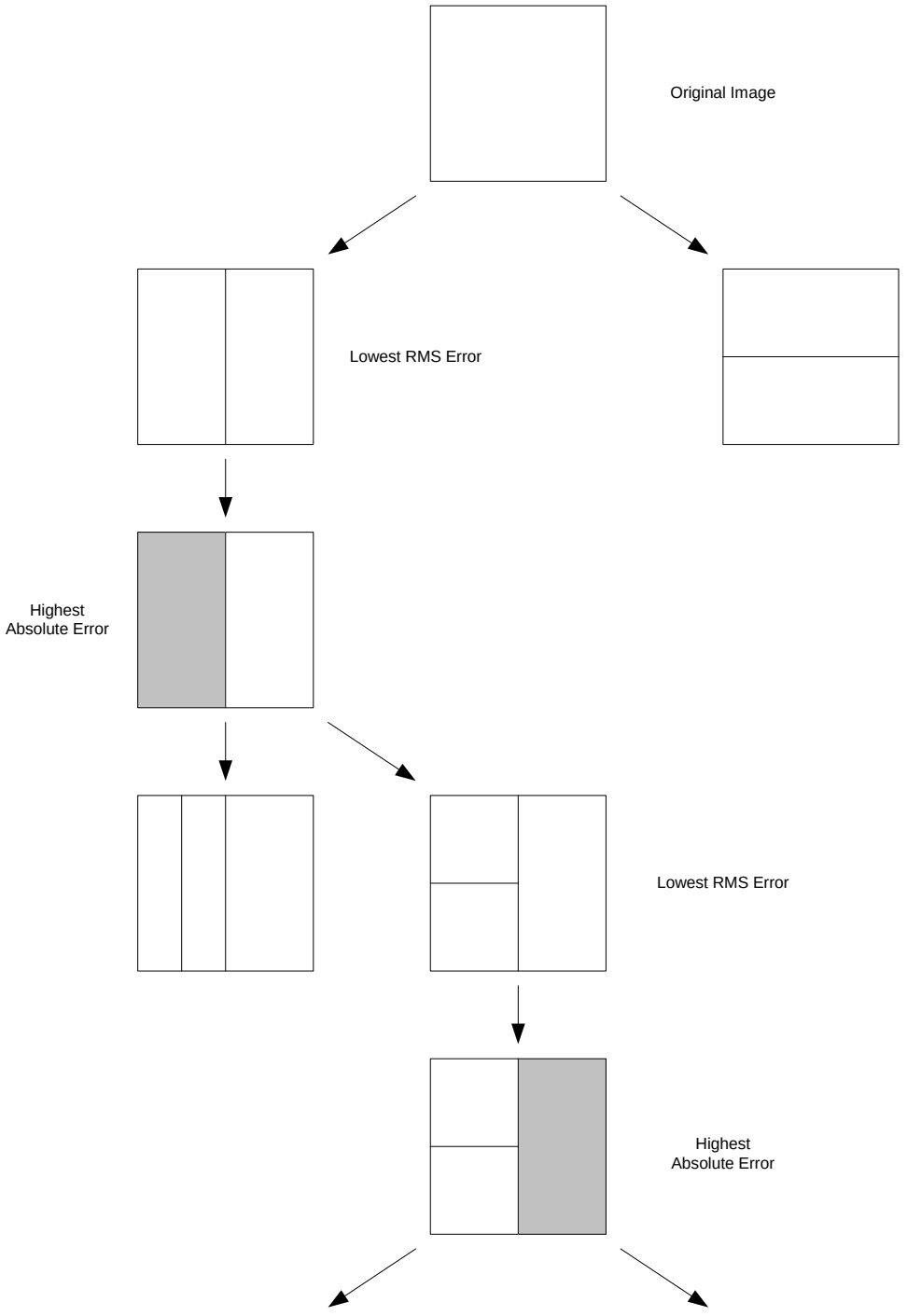

Figure 5.1: The modified LOMON strategy. Create two images from the original image. Subdivide one image into vertical regions, the other into horizontal regions. Find the image with the lowest RMS error. In this image find the region with the largest absolute error (error in colour between a pixel in the reconstructed image and the original image). Create two images by dividing this region into vertical and horizontal regions and repeat until the largest absolute error has fallen below a pre-defined limit.

```
PHASE I:
current_region = region with largest error

Repeat

    (image_error_vert, left_region, right_region) =
        split_vert(current_region)
    (image_error_horz, top_region, bottom_region) =
        split_horz(current_region)

    remove current region from region list

    if image_error_vert < image_error_horz then
        add left_region, right_region to region list
    else
        add top_region, bottom_region to region list

    current_region = region with largest error
    get max_image_error

Until max_image_error < e

PHASE II:
for all regions do begin
    set training_vector = original_image(region_x..region_x + x_width,
        region_y..region_y + y_width)
    train a SVM using training_vector as input
    store weights (support vectors) and biases for each region
end;
```

Figure 5.2: The LOMON algorithm where `max_image_error` is the maximum error in a pixel.

5.3 Results

Figure 5.3 shows results where the 256×256 Lena image has been compressed using an error of 60 for the LOMON part of the algorithm and a value of 20 for the SVM surface modelling with the width of the Gaussians λ set to 5. The number of blocks produced by LOMON is 2467 and the number of support vectors is 7206. Compare this when SVM surface modeling is used by divided the image into 8×8 blocks. This produced 8890 support vectors. Thus the number of support vectors has been reduced by 1684 or around 19% for the same error, $\epsilon = 20$. For this to be of benefit it is required that the storage space occupied by the 1684 support vectors be more that the storage space required to store the 2467 blocks. That is, without LOMON 8890 support vectors are required to model the surface in the SVM surface algorithm. With LOMON 7206 support vectors *and* 2476 blocks are required to model the surface using the LOMON-SVM algorithm. For the LOMON-SVM algorithm to be better than the SVM surface algorithm, the storage requirement of the 2476 blocks must be less than the storage requirements of the 1684 support vectors.

5.4 Discussion

Although moderate gains over the LOMON algorithm are achieved using this LOMON-SVM algorithm, it still suffers from the fact that the size and position of each block need to be stored. Examination of the values of the sizes and positions shows that entropy encoding (such as Huffman encoding described in chapter 4 would be of little benefit. There is also the drawback of speed. The divide and conquer strategy takes a few minutes to run and the SVM surface modelling can take a few minutes for each block for the larger blocks. This equates to a single image taking more than an hour to compress. The conclusion is that the LOMON-SVM algorithm is not suitable for the basis of an image

compression algorithm

Figure 5.3: Lena image using LOMON combined with SVM learning where e_1 is the error used during the LOMON part of the algorithm and e_2 is the ϵ error used during SVM learning. The bottom figure shows a previous result using SVM surface modelling with fixed block size (8×8 blocks).

Chapter 6

Compression in the Frequency Domain: The RKi-1 Algorithm

In this chapter the image is transformed into the frequency domain and apply SVM to the frequency components. The Discrete Cosine Transform (DCT) is used as it has properties which are exploited in SVM learning. The basic idea is to transform the image using the DCT, use SVM learning to compress the DCT coefficients and use Huffman coding to encode the data as a stream of bits. Portions of this work are presented in papers by the author [26, 27].

6.1 Introduction

Most image compression algorithms operate in the frequency domain. That is, the image is first processed through some frequency analysing function, further processing is applied onto the resulting coefficients and the results generally encoded using an entropy encoding scheme such as Huffman coding. The JPEG [8] image compression algorithm is an example of an algorithm of this type. The first step of the JPEG algorithm is to subdivide the image into 8×8 blocks then apply the DCT to each block. Next quantisation is applied to the resulting

DCT coefficients. This is simply dividing each element in the matrix of DCT coefficients by a corresponding element in a 'quantising matrix'. The effect of reduces the value of most coefficients, some of which vanish (i.e. their value becomes zero) when rounding is applied. Huffman coding is used to encode the coefficients. A more detailed description of JPEG is available in appendix 9.1.

Other researchers have used the DCT as a basis for neural network based compression algorithms. Recently Amerijckx et. al. [28] presented a compression scheme where the DCT was used followed by vector quantisation of the DCT coefficients by Kohonen map, differential coding by first order predictor and entropic coding of the differences. This method gave better performance than JPEG for compression ratios greater than 30:1.

The algorithm presented in this chapter also uses the discrete cosine transform. The DCT has properties which make it suitable to SVM learning. SVM learning is applied to the DCT coefficients. Before the SVM learning is applied the DCT coefficients are 'processed' in such a way as to make the trend of the DCT curve more suitable to generalisation by a neural network. This step forms the basis of a recently filed world-wide patent. As the DCT is fundamental to the algorithm a detailed description follows.

6.2 Discrete Cosine Transform

The DCT has properties making it the choice for a number of compression schemes. It is the basis for the JPEG compression scheme, and the 3-dimensional DCT is used in video compression algorithms. The DCT is a transform that maps a block of pixel colour values in the spatial domain to values in the frequency domain. The DCT can operate mathematically in any dimension, however an image is a two-dimensional surface so the 2-D DCT transform is used. The

2-dimensional DCT is given by:

$$T[i,j] = c(i,j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} V[x,y] \times \cos \frac{(2y+1)i\pi}{2N} \times \cos \frac{(2x+1)j\pi}{2N} \qquad (6.1)$$

where

$$c(i,j) = \frac{2}{N}, i \text{ and } j \neq 0$$
$$c(i,j) = \frac{1}{N}, i \text{ and } j = 0$$

The DCT is more efficient on smaller images. When the DCT is applied to large images, the rounding effects when floating point numbers are stored in a computer system result in the DCT coefficients being stored with insufficient accuracy. The result is a deterioration in image quality. As the size of the image is increased, the number of computations increases disproportionately. For these reasons an image is subdivided into 8×8 blocks. Where an image is not an integral number of 8×8 blocks, the image can be padded with white pixels (i.e. extra pixels are added so that the image can be divided into an integral number of 8×8 blocks. The 2-dimensional DCT is applied to each block so that an 8×8 matrix of DCT coefficients is produced for each block. This is termed the 'DCT Matrix'. The top left component of the DCT matrix is termed the 'DC' coefficient and can be interpreted as the component responsible for the average background colour of the block (analogous to a steady DC current in electrical engineering). The remaining 63 components of the DCT matrix are termed the 'AC' components as they are frequency components analogous to an electrical AC signal. The DC coefficient is often much higher in magnitude than the AC components in the DCT matrix.

The DCT is illustrated in Figure 6.1. Each component of the DCT matrix represents a frequency in the image (the DC component representing a frequency of 0). The further an AC component from the DC component the higher the frequency represented. The magnitude of higher frequency components tends

to diminish the higher the frequency represented. Higher frequency components are less visible to the eye, and it is this property which is exploited in JPEG and in our algorithm as these higher frequency components can be attenuated or removed with little noticeable effect on the quality of the image. The trend in the magnitude of the DCT coefficient is visible in Figure 6.1.

43	54	54	49	58	45	52	78
42	49	54	49	60	44	55	82
53	47	53	49	56	50	56	96
47	54	49	52	55	54	59	106
49	54	56	47	54	59	61	105
49	52	53	49	68	64	63	95
53	57	67	59	64	67	74	102
58	58	58	60	73	80	81	109

DCT →

← IDCT

488	-81	45	-43	29	-36	15	4
-44	20	-4	-9	5	-2	-6	4
10	5	-13	-6	6	5	-5	5
-11	7	-6	0	-6	-4	-2	3
5	-4	6	-1	-1	-3	-1	-7
2	2	1	-5	0	2	0	-2
-1	-1	0	9	4	-2	1	0
2	3	5	-4	-2	4	5	1

Figure 6.1: The DCT maps a block of pixel colour values to the frequency domain. Note that the magnitude of the coefficients generally increases the nearer they are to the top-left coefficient.

The original image block is recovered from the DCT coefficients by applying the inverse discrete cosine transform (IDCT), given by:

$$V[x,y] = \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} c(i,j)T[i,j]\cos\frac{(2y+1)i\pi}{2N} \times \cos\frac{(2x+1)j\pi}{2N} \tag{6.2}$$

where

$$c(i,j) = \frac{2}{N}, i \text{ and } j \neq 0$$
$$c(i,j) = \frac{1}{N}, i \text{ and } j = 0$$

6.2.1 Transformation of the DCT Matrix to 1-D

The elements of the matrix $T[i,j]$ in (6.1) are mapped using the zig-zag sequence shown in Figure 6.2 to produce a single row of numbers. That is a single row of

numbers is collected as the zig-zag trail is followed in the DCT matrix. This will produce a row of 64 numbers where the magnitude tends to decrease traveling down the row of numbers.

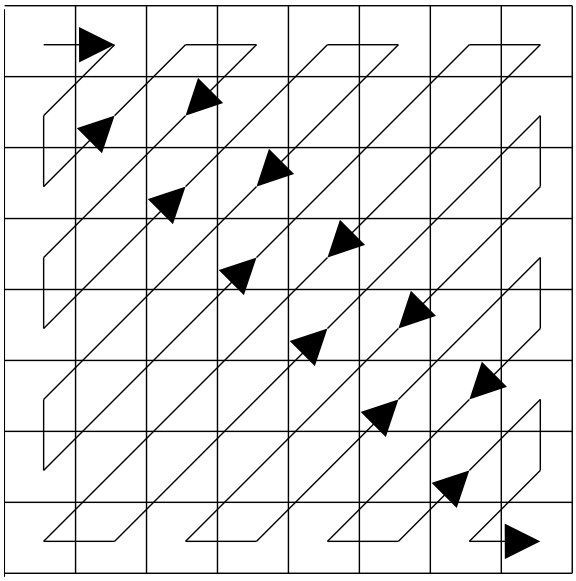

Figure 6.2: The zig-zag pattern applied to a block of DCT coefficients to produce a row of 64 coefficients. The importance of each coefficient is proportional to its position in the row.

6.3 Combining SVM with DCT

Here a new algorithm is detailed, for compressing the AC DCT coefficients after the discrete cosine transform has been applied to an image. The algorithm which we call the 'RKi-1' algorithm, also includes a preparation steps to improve the compression efficiency. An overview of the algorithm is shown in Figure 6.3 illustrating the steps described below.

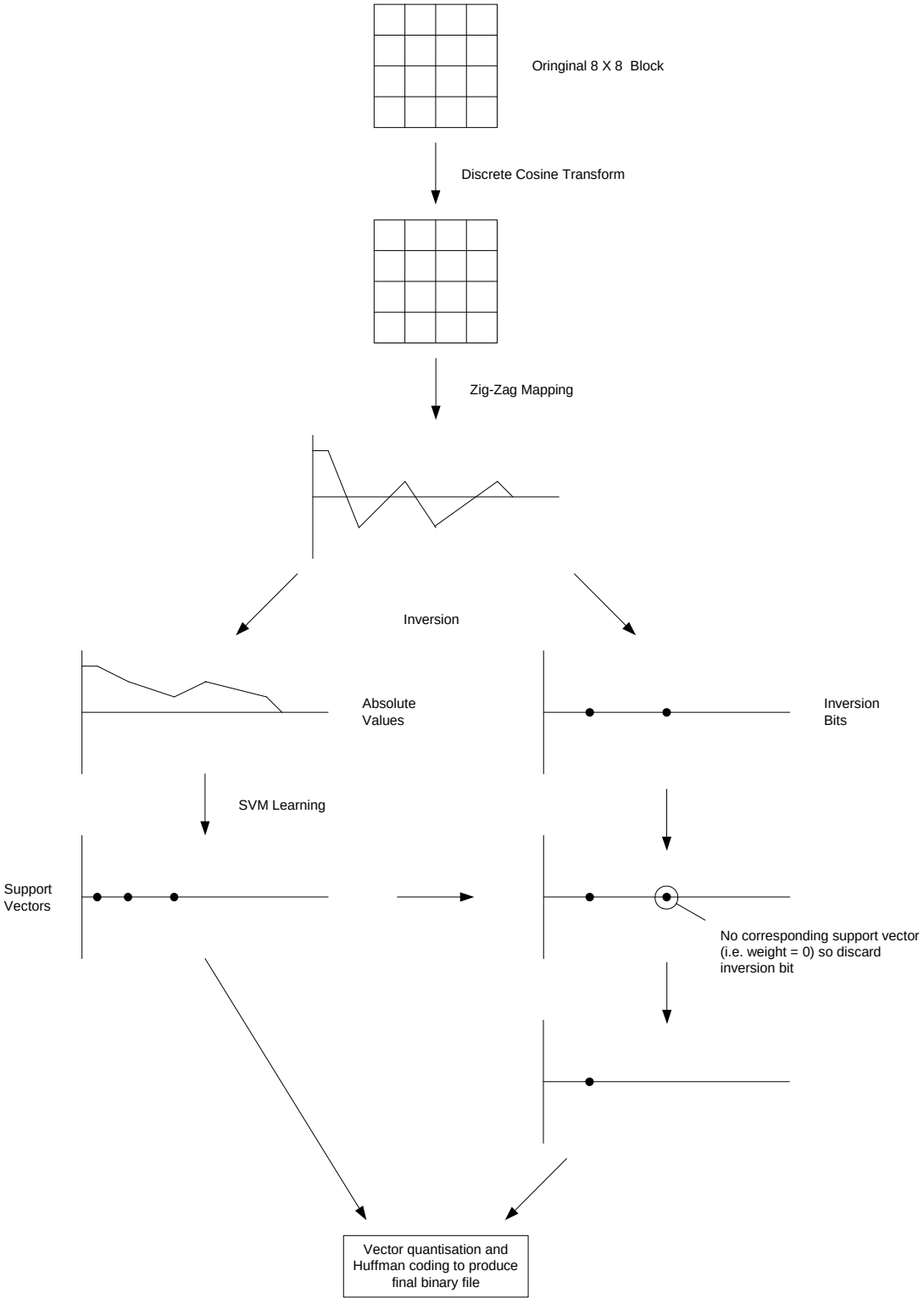

Figure 6.3: The RKi-1 Algorithm

Figure 6.4 (a) shows the 1-dimensional plot of the AC coefficients for the matrix shown in fig 6.1 after the zig-zag mapping. This is a plot of a single row of DCT coefficients after the zig-zag mapping has been applied to the block in Figure 6.1. The DC component has been removed as it is treated separately leaving the 63 AC coefficients.

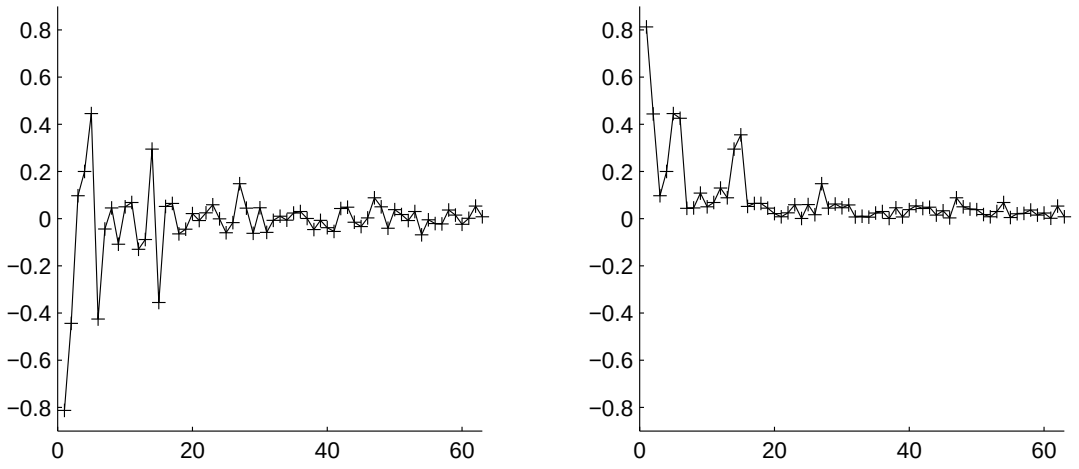

Figure 6.4: The graph on the left is a plot of a typical 8×8 block of DCT coefficients after the zig-zag mapping to a row of numbers. This plot is not easily generalised by a neural network. The plot on the right is a plot of the absolute magnitude of the same data. This plot, while still not ideal, is better generalised by a neural network.

The 1-dimensional row of DCT coefficients is used as the training data for a SVM. In the discussion of SVMs in Chapter 3 attention was drawn to the fact that a SVM will produce the minimum number of support vectors required to generalise the training data within a predefined error (the ϵ-tube). Thus it is expected that when the row of DCT coefficients are used as training data for the SVM, a lower number of support vectors will be required in order to recover the DCT coefficients within the predefined error. This is illustrated in Figure 6.5 (top) where the DCT coefficients in Figure 6.1 are used as input training data to a SVM. The bottom plot in Figure 6.5 shows the error when the output is

compared with the original.

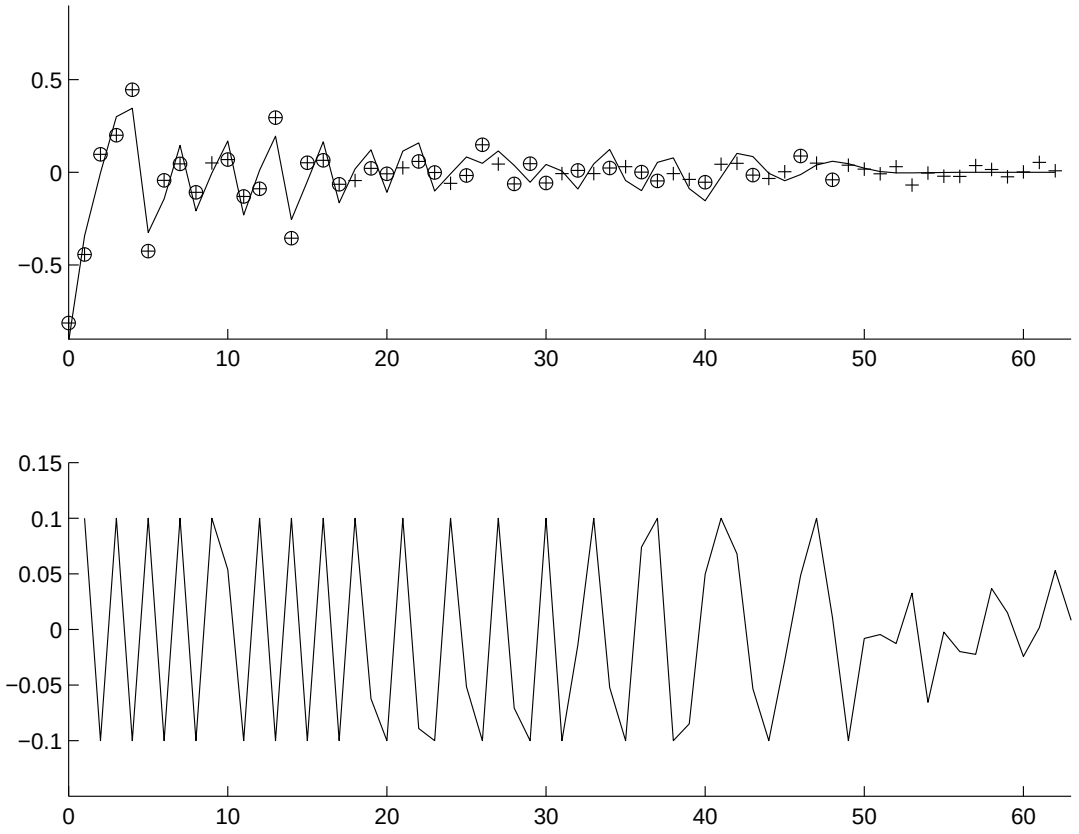

Figure 6.5: TOP: Real values of the row of DCT coefficients input to the SVM. BOTTOM: The error of the output subtracted from the desired input. The maximum allowable error is 0.1

In Figure 6.5 there are 63 input training points. With the error (ϵ-insensitivity) as 0.1, 34 training points were chosen by the SVM as support vectors. This can be loosely interpreted as compression of 63:34 or approximately 1.85 : 1.

Examination of the input data (ie the DCT coefficients) reveals that the magnitudes of the coefficients are generally decreasing traveling down the row of input data, however the sign (positive or negative) appears to be random. This has the consequence that two coefficients next to each other can be of similar magnitude but opposite sign causing a large swing in the input data.

The exercise is now repeated but using the absolute magnitudes of the DCT

coefficients in Figure 6.1. That is, the negative signs are ignored and all inputs are treated as positive. Figure 6.6 (top) shows the result. It can be seen that this input data is more suitable for generalisation by the SVM as less training points are chosen to be support vectors. In this example the SVM has chosen 10 support vectors representing a compression of 63:10 or 6.3 : 1. This is an increase in compression of around 3.5 times over the preceding example when the real values of the DCT coefficients were used as training data. Note also that when the real values were used as training data, the total accumulated error was 452, and when the magnitudes were used the total accumulated error reduced to 271. In this example, by ignoring the sign of the DCT coefficient when used as training data to the SVM, the number of support vectors has reduced by $\approx 70\%$ and the accumulated error has reduced by $\approx 40\%$. It is concluded that by simply using the magnitude of the input values and ignoring the sign, that the compression is substantially increased and the output is a closer match to the desired output (i.e. the error in the output is reduced). A comparison plot of the input data when both the real values and the absolute magnitudes is shown in Figure 6.4.

If the sign of each DCT coefficient is ignored when used as input data to the SVM, there is the problem of how to re-assign the signs when the DCT coefficients have been recovered. In order to recover the sign the 'Inversion Bit'is introduced.

6.3.1 The 'Inversion' bit

The 'inversion bit' indicates which of the recovered points on Figure 6.4 (b) should be inverted (i.e. multiplied by -1) so that they are negative – that is, positive points in Figure 6.4 (b) that were originally negative are made negative by multiplying by -1 if the inversion bit is set. The inversion bit is a single '0' or '1'. It is the sign of the corresponding input datum. Each input datum has

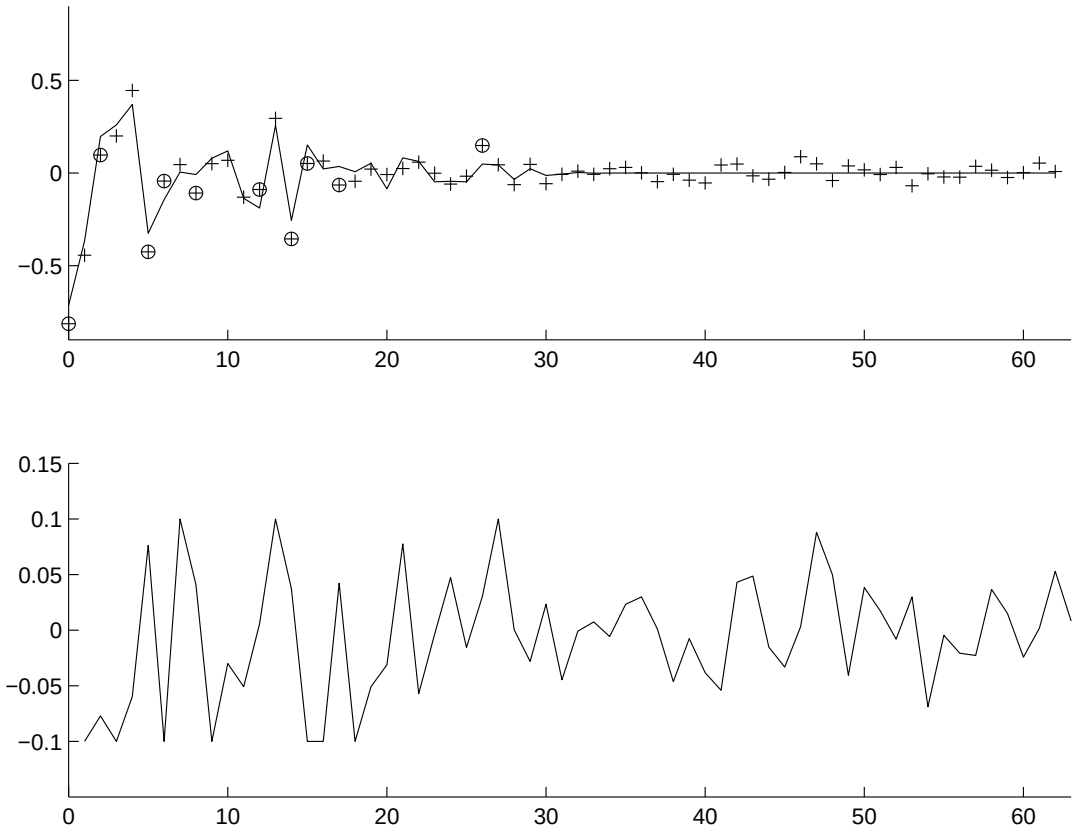

Figure 6.6: TOP: Absolute magnitude of the row of DCT coefficients input to the SVM. BOTTOM: The error of the output subtracted from the desired input. The maximum allowable error is 0.1

an inversion bit, so for the block shown in Figure 6.1 there are 63 inversion bits. Together these bits form what is termed the 'inversion number'. The inversion number for the matrix of DCT coefficients in Figure 6.1 is

110001101001101011101000010010110100011110011000100101011001000

After a block has been processed by the SVM, some the recovered DCT coefficients may have a magnitude lower than the maximum error defined for the SVM. If these components had an inversion bit of '1' this can be set to '0' as the sign of coefficients with small magnitude does not affect the final recovered image. Put another way, inversion bits for very small magnitude DCT coefficients do not contain significant information required for the recovery of the image. For

example if the ϵ-insensitivity is defined as 0.1, the desired value of a particular DCT coefficient is -0.06, and the recovered value 0.04 then the inversion bit is set to '0' for this coefficient (rather than '1') because a recovered value of 0.04 is within 0.1 of the original desired value. The importance of this property will be detailed in the implementation.

6.4 Implementation

The image is first sub-divided into 8×8 blocks. The 2-dimensional DCT is applied to each block to produce a matrix of DCT coefficients. The zig-zag mapping is applied to each matrix of DCT coefficients to obtain a single row of numbers for each original block of pixels. The first term of each row (the DC component) is separated so that only the AC terms are left. Not all the terms in the row of AC coefficients are needed since the higher order terms do not contribute significantly to the image. Only the first few values are taken (typically the first $10 - 16$ values). Exactly how many values are taken is a degree of freedom in the algorithm. Support vector machine learning is applied to the absolute values of each row of AC terms as described above and the inversion number for each block is generated. By following this method, for each original block the gaussian centres (i.e. the support vectors), the weights and the inversion number need to stored/transmitted to be able to recover the block.

The AC components are used as training data to a SVM. The support vector machine learning used is identical to the SVM learning detailed in the previous chapter on SVM modelling of the surface (Chapter 4). The Non Negative Least Squares (NNLS) algorithm detailed in [24] is used. NNLS typically requires only a few seconds to solve compared with a quadratic programming solution requiring a few minutes.

By way of example, the first 16 AC coefficients, training data and inversion number of the image block in Figure 6.1 is

Original Coeffs	-81	-44	10	20	45	-43	-4	5	-11	5	...
Training Data	81	44	10	20	45	43	4	5	11	5	...
Inversion No	1	1	0	0	0	1	1	0	1	0	...
					...	7	-13	-9	29	-36	5
					...	7	13	9	29	36	5
					...	0	1	1	0	1	0

A SVM trained on the data above with an error set to 10 and Gaussian width set to 1, selects the following training indices as support vectors with the associated weights:

Support Vectors	1	5	6	9	12	14	15
weights	71.29	23.44	18.29	0.60	1.95	5.91	21.97

In order to recover the image the DC coefficient, the support vectors, the weights and the inversion number are required. Thus if the image is to be transmitted then these are the components that must be encoded and sent.

After decoding, the recovered coefficients are

Original Coeffs	-81	-44	10	20	45	-43	-4	5	-11	5	...
Recovered Coeffs	-71	-44	13	17	35	-33	-14	3	-1	1	...
Error	10	0	3	3	10	10	10	2	10	4	...
					...	7	-13	-9	29	-36	5
					...	1	-3	-8	20	-26	14
					...	6	10	1	9	10	9

A surface plot of the original 8×8 surface is shown along side the recovered surface in Figure 6.7. In the actual implementation the pixels are normalised in the image between 0 and 1. This is because neural networks tend to be more efficient when the inputs and outputs are normalised [29]. Normalising the image produces weights that are lower in magnitude (compared with an un-normalised image) and are similar in value. This is an important consideration when encoding the data for storage/transmission described in the following section.

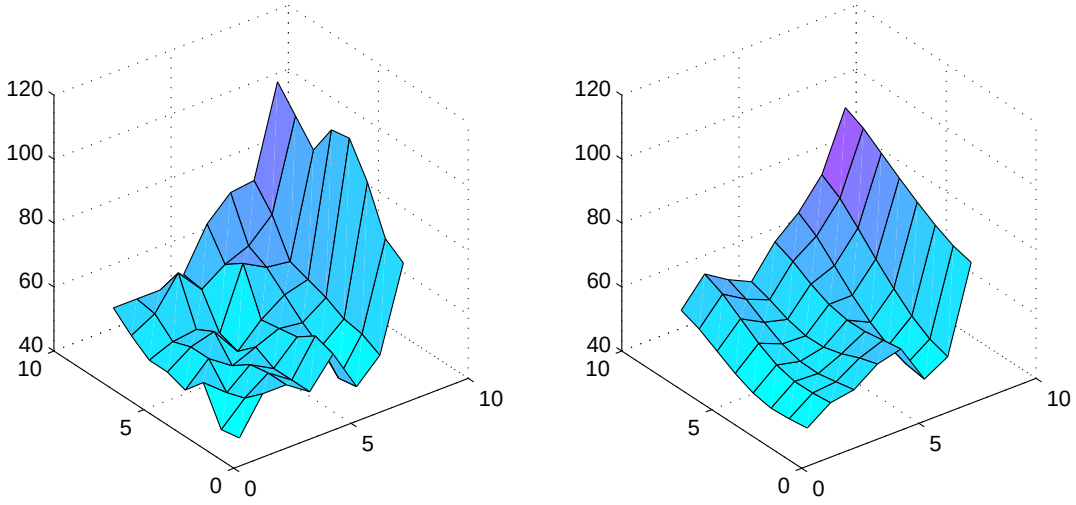

Figure 6.7: (a) Original block, (b) Recovered block

6.4.1 Encoding Data for Storage

For each block weights and support vectors are required to be stored. The support vectors are the gaussian centres. In our RKi-1 algorithm we combine the weights with the support vectors so that each block has the same number of weights as DCT coefficients. For example if only the first 16 DCT coefficients are chosen to be retained (discarding the remaining 47 DCT coefficients), then there are 16 weights for each block. Where a weight has no corresponding support vector the value of the weight is set to zero. That is, the only non-zero weights are weights for which a training point has been chosen to be a support vector by the support vector machine.

The next step is to quantise the weights. Quantising involves reassigning the value of the weight to one of limited number of values. To quantise the weights the maximum and minimum weight values (for the whole image) are found and the number of quantisation levels are pre-defined. The number of quantisation levels chosen is a degree of freedom in the algorithm. For example if the minimum weight is 0 and the maximum is 0.9 and we define 5 quantisation levels then the

weights can only take the values 0.09, 0.27, 0.45, 0.63, 0.81. Each weight in the image is reassigned to the closest value of these quantised values.

The inversion bits are now combined with the weights as follows. After quantisation, the minimum quantisation level is subtracted from each weight. This will ensure that all weights have a positive value. An arbitrary number is added to all weights (the same number is added to all numbers) making all weights positive and non-zero. To recover the weights both the minimum quantisation level and the arbitrary number must be stored. Each individual weight has an associated inversion bit. The inversion bit is combined with its corresponding weight to making the value of the weight negative if the inversion bit is '1', otherwise it is positive. Where the weight is not a support vector the inversion data is discarded. This introduces a small error when the image is decompressed, but significantly increases compression (see the results section).

The above steps introduce many 'zero' values into the weight data. By setting inversion bits from '1' to '0' when the associated DCT is less than the error ϵ many more zeros are introduced.

The compression part of the algorithm was implemented in MATLAB. The MATLAB routine produces an XML export file which is then read be a program written in Borland's Delphi. For prototyping purposes MATLAB is a better choice than Delphi for complex mathematical implementation (such as NNLS) whereas Delphi is better at manipulating binary and string data. An example XML export file is shown in Figure 6.8.

6.4.2 Encoding the DC Coefficients

DC coefficients are treated differently to AC coefficients. The DC component sets the average background colour of its associated block, thus any error in the DC coefficient will be much more noticeable than errors in the AC coefficients. The method of encoding the DC coefficients in RKi-1 is similar to JPEG. A

```
<SVM_IMAGE>
  <HEADER>
    <no_of_blocks>4096</no_of_blocks>
    <NoOfCoeffs>8</NoOfCoeffs>
    <height>512</height>
    <width>512</width>
    <max_pixel_level>25</max_pixel_level>
    <Sigma>0.5</Sigma>
  </HEADER>
  <BLOCK>
    <block_x>1</block_x>
    <block_y>1</block_y>
    <block_number>1</block_number>
    <DC_Coeff>-28.5000</DC_Coeff>
    <Sigma>0.5</Sigma>
    <Center>1</Center>
    <Center>3</Center>
    <Center>5</Center>
    <Center>8</Center>
    <Inversion>1</Inversion>
    <Inversion>1</Inversion>
    <Inversion>1</Inversion>
    <Inversion>0</Inversion>
    <Inversion>1</Inversion>
    <Inversion>1</Inversion>
    <Inversion>1</Inversion>
    <Inversion>0</Inversion>
    <weight>0.5036</weight>
    <weight>0.1127</weight>
    <weight>0.1382</weight>
    <weight>0.2013</weight>
  </BLOCK>
  <BLOCK>
    ...
  </BLOCK>
</SVM_IMAGE>
```

Figure 6.8: An example of an XML export file produced by the MATLAB routine. In this example only one out of 4096 blocks are shown.

Table 6.1: Huffman encoding DC values. Each DC coefficient is assigned a number in the 'Encoded Value' column which matches the 'DC Value Range' of the DC coefficient.

Encoded Value	DC Value Range
0	0
1	1,-1
2	3,2,-2,-3
3	7 . . . 4,-4 . . . -7
4	15 . . . 8,-8 . . . -15
5	31 . . . 16,-16 . . . -31
6	63 . . . 32,-32 . . . -63
7	127 . . . 64,-64 . . . -127
8	255 . . . 128,-128 . . . -255
9	511 . . . 256,-256 . . . -511
10	1023 . . . 512,-512 . . . -1023
11	2047 . . . 1024,-1024 . . . -2047

description on the encoding algorithm has been given in chapter 4, however it is repeated here for convenience.

DC coefficients are encoded as the difference between the current value and the previous value. Differences are encoded rather than the full value as the differences tend to be smaller than full values. A DC coefficient is encoded in two steps. The first step encodes the magnitude as a Huffman code, and the second step adds 'literal bits'. Literal bits are not encoded – they contain raw data. The number of literal bits is equal to the magnitude value. Table 6.4.2 lists the Huffman encoded magnitude values (0–11) with the corresponding magnitude of the change in DC. For example to encode the number 58, the magnitude value is the Huffman code for 6 (corresponding to the DC value range 63 . . . 32), followed by 6 literal bits 011010 (the difference between 58 and 32 in binary).

6.5 Results

The SVM learning component of the compression algorithm was implemented in MATLAB. The routine exports an XML file which is read by an encoding/decoding application developed in Borland's Delphi. The XML file contains DC components, weights, indexes of the support vectors and the inversion details. These are encoded to produce a binary file as described above. The application also decodes these binary files. Compression ratios are computed by the following formula:

$$\text{compression} = \frac{\text{image_height} \times \text{image_width} \times 256}{\text{Filesize in Bytes}} \tag{6.3}$$

where 256 is the number of gray-scale levels.

The routines were written for ease of implementation and as such are less than optimal. Even so, both the compression/coding and decoding routines only take a few seconds (on a 600 MHz Pentium III computer). The compression takes longer than the decompression and the higher the compression ratio, the quicker the decompression. The opposite is true for the JPEG algorithm. From a practical user perspective there is little difference in the decompression speed between JPEG and RKi-1 algorithm.

6.5.1 Performance

To objectively measure image quality, the signal to noise ratio (SNR) is used. The SNR is calculated using:

$$SNR = 10 \log \left[\frac{\sum_{ij} \text{Original_Image}_{ij}{}^2}{\sum_{ij} \left(\text{Original_Image}_{ij} - \text{Recovered_Image}_{ij} \right)^2} \right] \tag{6.4}$$

Results using the benchmark 512×512 'Lena' image are shown in Figure 6.9 compared with the baseline JPEG algorithm. Figure 6.9 shows that the JPEG algorithm performs better than the RKi-1 algorithm for compression ratios up to 22:1 (on this particular image). For compression ratios beyond this, the RKi-1

algorithm produces higher quality images for the same compression ratio. The baseline JPEG algorithm could not compress the 512×512 Lena image grater than 63:1. The RKi-1 algorithm achieved a compression ratio of 192:1 and still achieved better image quality than the image compressed using JPEG at 63:1. Similar results are shown in Figure 6.10 for the Claudia image where it can be seen that RKi-1 performs better than JPEG at higher compression ratios.

Decompressed images are shown in Figures 6.12 and 6.13 for subjective comparison of the RKi-1 algorithm compared with the baseline JPEG algorithm. For both RKi-1 and JPEG the blocking effect is clearly seen at higher compression ratios. It is clearly evident from inspection of these figures that RKi-1 performs better at higher compression ratios that JPEG.

In comparing the SNR results (Figures 6.9 and 6.10) with the corresponding visual images (Figures 6.12 and 6.13) the difference in subjective image quality is greater for higher compression ratios. For example the visual difference in image quality for the Claudia image at 23:1 (a SNR of about 46 dB for RKi) is small (and barely detectible by the human eye) yet the SNR difference is 15 dB. At a compression ratio of 63:1 (a SNR of about 42 dB for RKi) the difference in SNR is 5 dB yet the visual difference is far more noticeable and the JPEG image is significantly deteriorated. The conclusion is that above a SNR of 45 dB any difference in SNR is much less significant than when the SNR is below 45 dB.

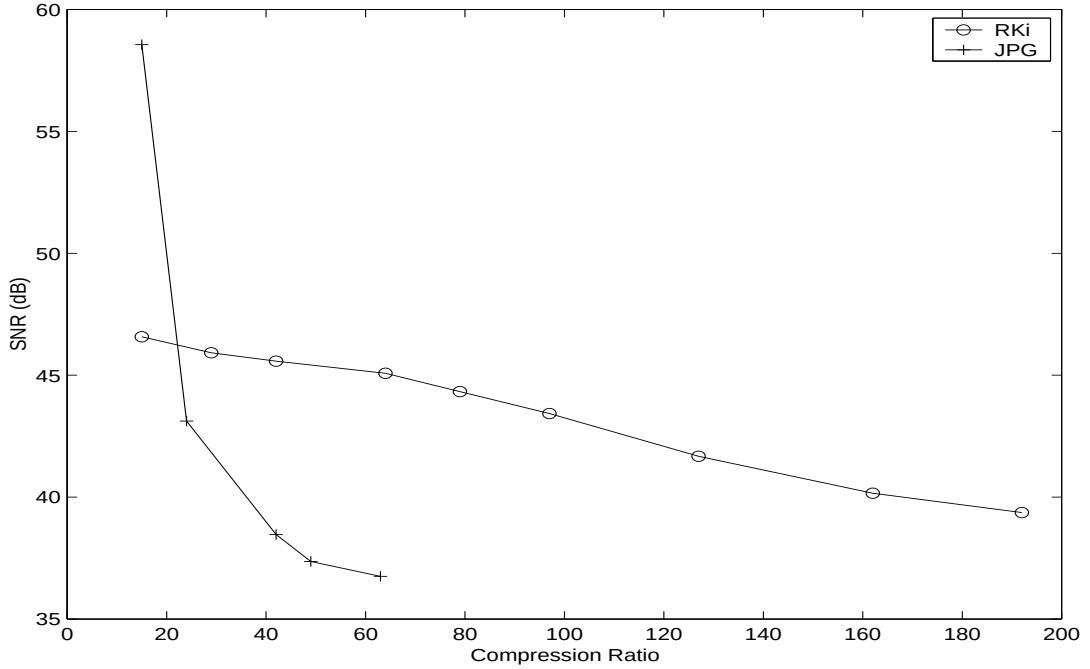

Figure 6.9: Plot of compression ratio versus SNR for the RKi-1 algorithm and baseline JPEG using the 512×512 Lena image

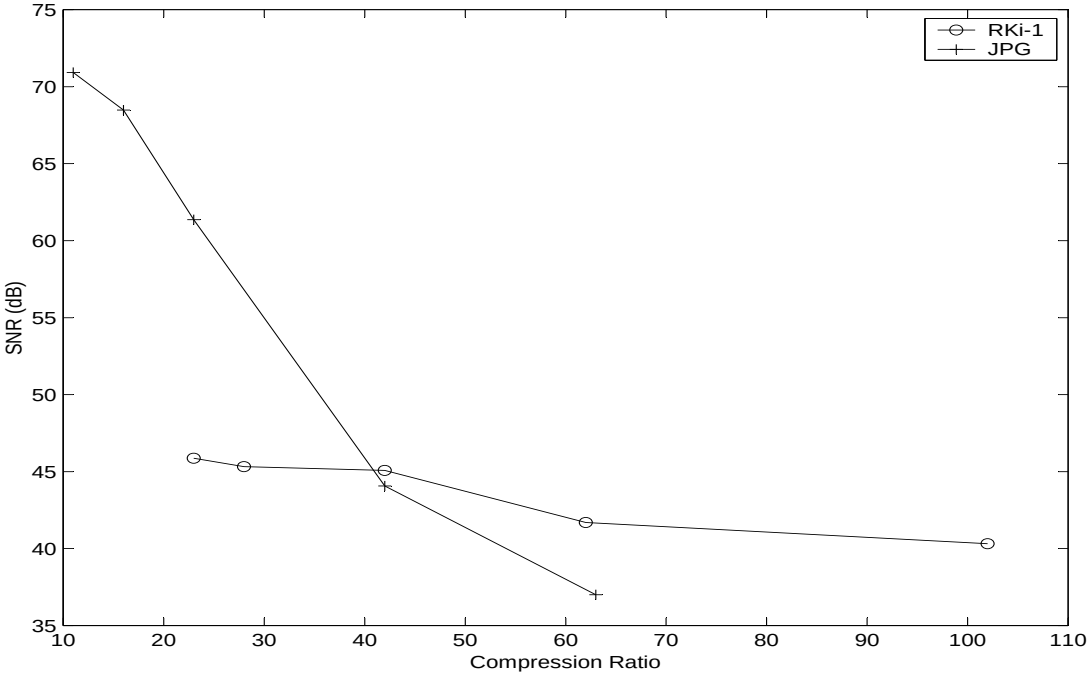

Figure 6.10: Plot of compression ratio versus SNR for the RKi-1 algorithm and baseline JPEG using the 400×400 Claudia image

All inversions. Compression 44:1 Reduced inversions. Compression 58:1

Figure 6.11: Effect of reducing the number of inversion bits for the 400×400 Claudia image. Little difference is observed between the two images however by reducing the number of inversion bits the compression has been increased from 44:1 to 58:1.

6.5.2 Effect of Reducing the Number of Inversion Bits

One of the steps in the RKi-1 algorithm involves reducing the number of inversion bits by setting an inversion bit equal to zero if there is no corresponding support vector. This step introduces errors, however the errors are generally minor and barely perceptible to the human eye. Figure 6.11 demonstrates this for the 400×400 Claudia image. Inspection of the two images in Figure 6.11 shows little visual difference between the two images however the difference in compression is 44:1 (all inversions) compared with 58:1 (reduced inversions).

Figure 6.12: Subjective comparison of the 512×512 Lena image compressed using both the RKi-1 and JPEG algorithms. Note that the JPEG algorithm would not compress beyond 63:1

Figure 6.13: Subjective comparison of the 400×400 Claudia image compressed using both the RKi and JPEG algorithms. Note that the JPEG algorithm would not compress beyond 63:1

6.6 Discussion

In this chapter a novel image compression algorithm has been presented which takes advantage of support vector machine learning. The algorithm exploits the trend of the DCT coefficients after the image has been transformed from the spacial domain to the frequency domain via the discrete cosine transform. Support vector machine learning is used to estimate the DCT coefficients within a predefined error, ϵ. The SVM is trained on the absolute magnitude of the DCT coefficients as these values require less support vectors to estimate the underlying function. The net result of the SVM learning is to compress the DCT coefficients much further than other methods such as JPEG. The algorithm also defines how the original values are recovered by the introduction of the 'inversion number'. The inversion number allows recovery of the original sign (i.e. positive or negative) of each DCT coefficient so that combined with the magnitude of the coefficient as estimated by the SVM, a close approximation to the original value of the DCT coefficient is obtained in order to reconstruct the image.

Results have been presented showing that the new method produces better image quality than the JPEG compression algorithm for compression ratios greater than around 20:1. It has also been shown that large compression ratios (192:1) are possible with the RKi-1 algorithm while still retaining reasonable image quality.

Chapter 7

The RKi-2 Compression Algorithm

This chapter presents the RKi-2 algorithm which is a logical extension of the RKi-1 algorithm presented in the previous chapter. The RKi-1 algorithm applied SVM learning to DCT coefficients after a 'zig-zag' mapping, mapping each 2-dimensional block of DCT coefficients into a 1-dimensional row of parameters. The RKi-2 algorithm applies SVM learning to each block of DCT coefficients in 2-dimensional space removing the need for zig-zag mapping.

7.1 2-Dimensional SVM

In the SVM modelled surface algorithm (Chapter 4) and RKi-1 detailed in the previous chapter 1-dimensional modelling was used. The main difference between 1-dimensional modelling and n-dimensional modelling is the number of spacial parameters to the kernel functions. In 1-dimension the Gaussian kernels are of the form:

$$G(x, y) = \exp(-\frac{(x - c)^2}{\lambda}) \tag{7.1}$$

In matrix notation this is expressed as:

$$\mathbf{G} = \exp(-\frac{\mathbf{x} - \mathbf{c}}{\lambda}) \tag{7.2}$$

Thus for the 2-dimensional case each component of **G** is expressed as:

$$G(x_i, y_j, c) = \sum_{m=1}^{M} \exp(-\frac{(x_i - c_m)^2}{\lambda}) \sum_{n=1}^{N} \exp(-\frac{(y_j - c_n)^2}{\lambda}) \tag{7.3}$$

where (x_i, y_j) is the spatial coordinate, (c_n, c_m) is the centre of a Gaussian, M is the number of Gaussians in the x-axis and N is the number of Gaussians in the y-axis.

In the implementation used in the RKi-2 algorithm a Gaussian is initially placed at each point, thus the size of the image is $M \times N$ pixels (the support vector machine will eliminate unnecessary Gaussians). The dimensions of **G** are the same as for the 1-dimensional case and all parameters required for encoding and storage of the compressed image are as developed in the RKi-1 algorithm.

The RKi-2 algorithm is illustrated in Figure 7.1. The figure shows the algorithm is similar to RKi-1 except that SVM learning is performed on a three dimensional input space.

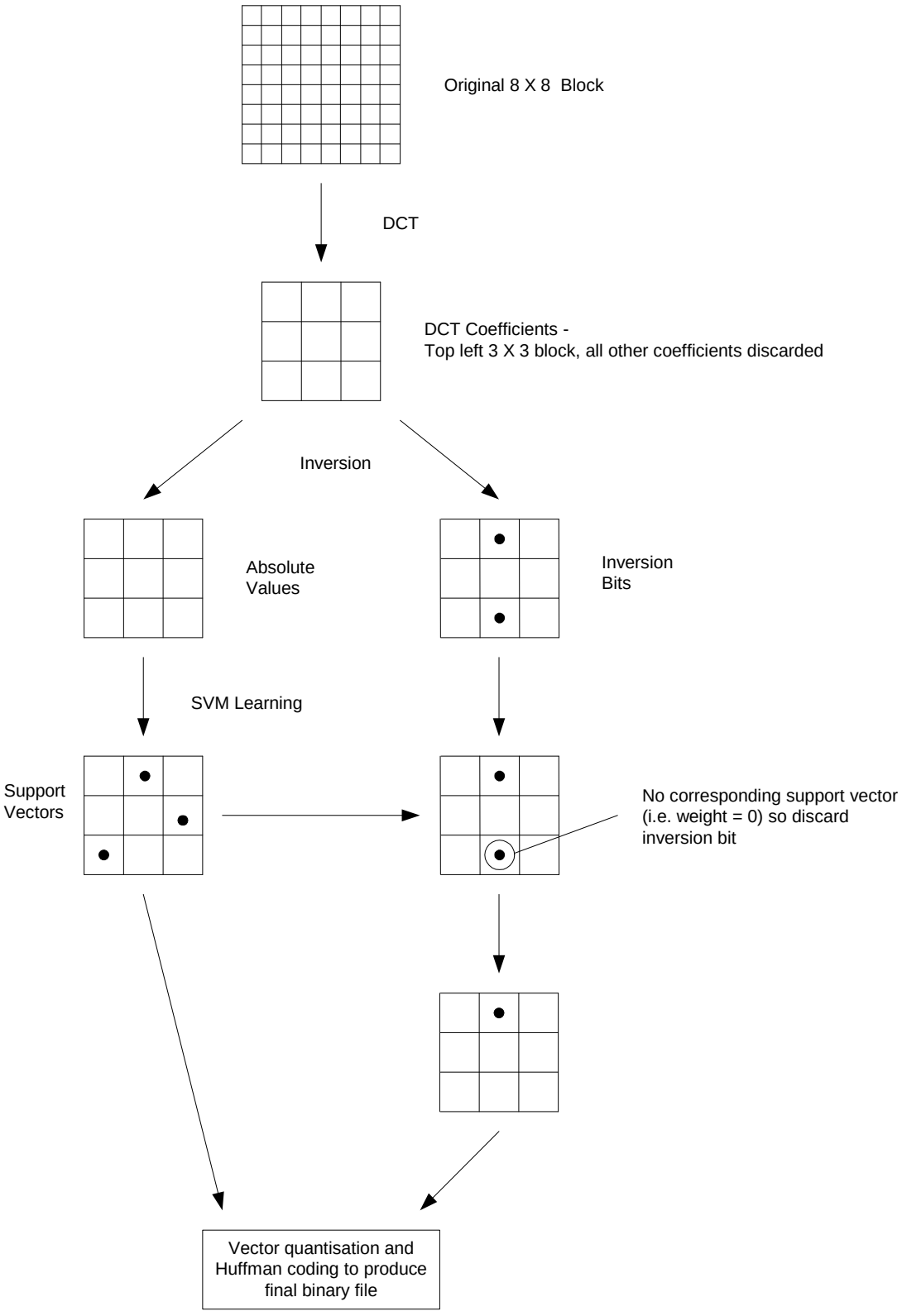

Figure 7.1: The RKi-2 Algorithm

7.2 Results

Subjective results for the 512×512 Lena image and 400×400 image are shown in Figures 7.4 and 7.5. The figures show that image quality is better than JPEG at high compression ratios, while at lower compression ratios JPEG produces better quality images. Comparing these figures with images produced by RKi-1 in Figures 6.12 and 6.13, it can be seen that RKi-1 performs better than RKi-2. A possible reason for this is discussed below. Figures 7.2 and 7.3 show plots of the signal to noise ratio for different compression ratios. These figures clearly show that for both images RKi-2 performs better at higher compression ratios than JPEG, but JPEG performs better at lower compression ratios.

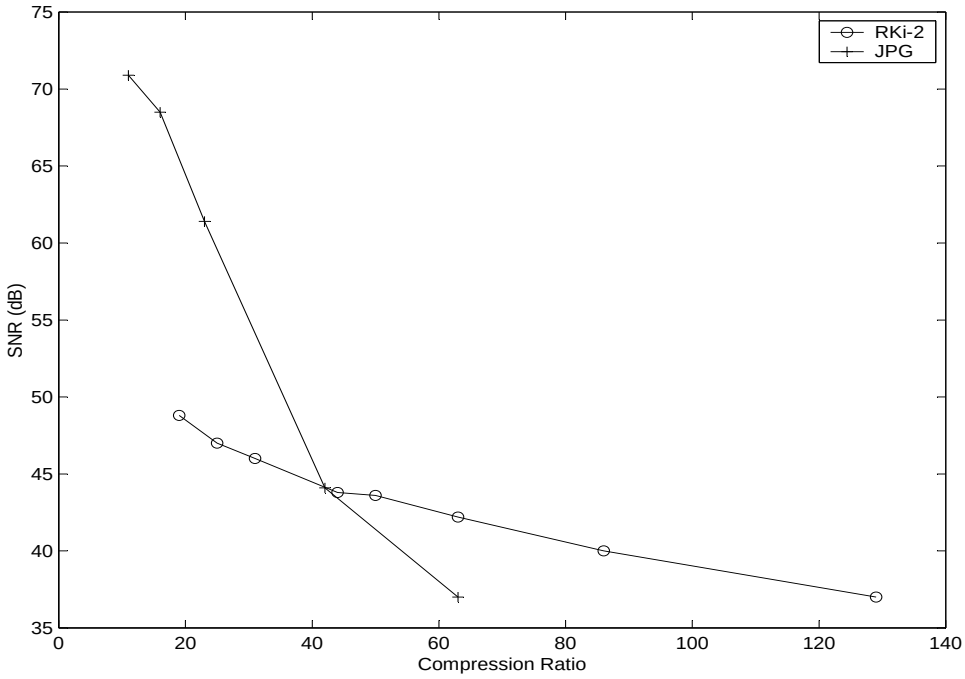

Figure 7.2: Plot of SnR versus compression ratio for the 400×400 Claudia image

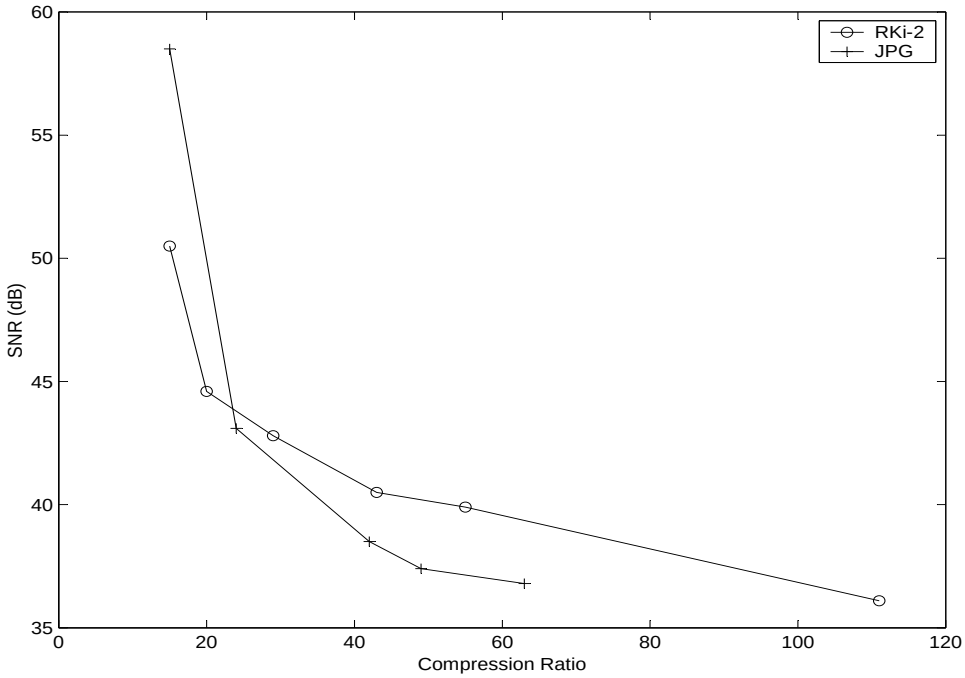

Figure 7.3: Plot of SnR versus compression ratio for the 512×512 Lena image

Figure 7.4: Subjective comparison of the 512×512 Lena image compressed using both the RKi-2 and JPEG algorithms. Note that the JPEG algorithm would not compress beyond 63:1

Figure 7.5: Subjective comparison of the 400×400 Claudia image compressed using both the RKi-2 and JPEG algorithms. Note that the JPEG algorithm would not compress beyond 63:1

7.3 Discussion

Compression rate is not as high as RKi-1 for the same image quality, even though training on a two dimensional surface may seem a more logical approach than training on a two dimensional surface mapped to a one dimensional line. This is a counter-intuitive result as it would be expected that modelling a 2-dimensional surface using a 2-dimensional neural network would produce better results than transforming the 2-dimensional surface to one dimension and using a 1-dimensional neural network.

A possible explanation for the poorer results is in encoding the step of the algorithm where the inversion number is discarded if the corresponding weight is zero. This step introduces many more 'zeros' which result in more efficient storage of the weights as the weight data are used to 'carry' the inversion data. However, this step does introduce errors in the recovered DCT coefficients. The effects are more noticeable in the 2-dimensional case than the 1-dimensional since an individual Gaussian in the 2-dimensional case has more influence on each recovered coefficient than the 1-dimensional case.

Figure 7.6 illustrates the influence on the value of recovered DCT coefficients for both 1-dimensional and 2-dimensional cases. It is seen in this figure that the recovered value an individual DCT coefficient in the 2-dimensional case is more likely to be a product of more than one Gaussian compared with the 1-dimensional case where the value of a recovered DCT is likely to be the product of a single Gaussian. The 2-dimensional case would also likely produce a result that more closely matches the original. Thus discarding the inversion bit would increase the error in the recovered DCT compared with the 1-dimensional case (where the result may be closer to zero).

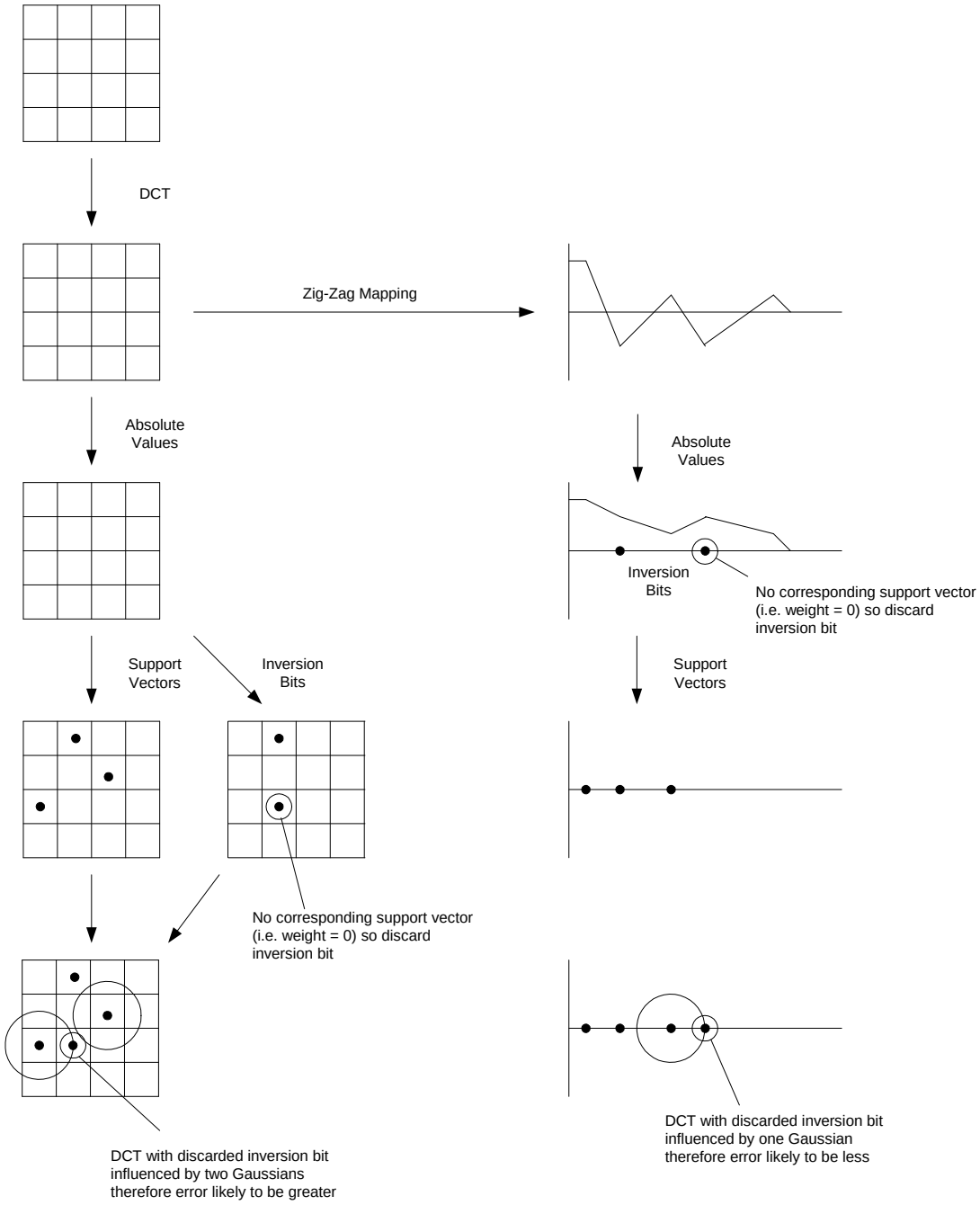

Figure 7.6: Influence of Gaussians over recovered DCT coefficients in RKi-1 and RKi-2.

Chapter 8

The RKi-3 Algorithm

The RKi-3 algorithm is a modification of the RKi-2 algorithm whereby the DCT coefficients are 'shuffled' prior to SVM learning. This is an attempt to place DCT coefficients with similar magnitude next to each other on the assumption that fewer support vectors will be required to model the 2-d DCT surface. Figure 8.1 shows how the coefficients are shuffled. The figure shows nine blocks containing 3×3 pixels. In each block all top-left coefficients are placed next to each other in the shuffled block. Likewise for the other coefficients.

The image is divided into 'super-blocks' of dimension $n^2 \times n^2$ where where n is the dimension of a block. In the actual implementation blocks of size 8×8 are used, thus the image is divided into super-blocks of 64×64. The mapping requires the image dimensions to be a multiple of n^2. If the image dimensions are not a multiple of 64 it is a simple matter to add extra 'white' pixels to pad out the dimensions.

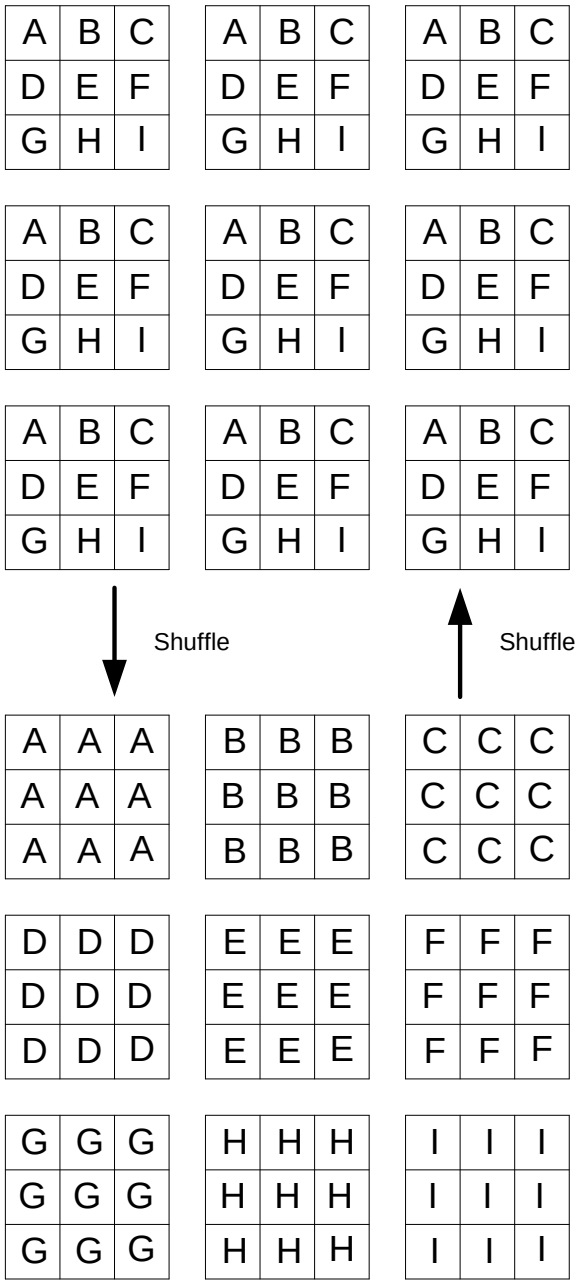

Figure 8.1: Shuffling DCT coefficients. Note that the same mapping applied to the shuffled coefficients which produced the shuffled coefficients will 'de-shuffle' them.

The shuffled coefficients are mapped using the equation

$$B(i,j) = A([([(i-1)\text{mod } n]+1)*n] + ([(i-1)\text{div } n) - (n-1)), \\ [([(j-1)\text{mod } n]+1)*n] + ([(j-1)\text{div } n) - (n-1))) \tag{8.1}$$

where n is the number of coefficients in a block, A is the original block of $n^2 \times n^2$ coefficients, B is the shuffled block of $n^2 \times n^2$ coefficients. The 'mod' operator in equation (8.1) returns the remainder obtained by dividing its operands and the 'div' operator, x div y is the value of x/y rounded in the direction of zero to the nearest integer.

The initial size of each super-block is 64×64. However as for RKi-1 and RKi-2 all coefficients do not need to be preserved. In RKi-1 and RKi-2 the top left coefficients (usually 8–16) were kept while all others were discarded. Similarly for RKi-3. If the first 9 coefficients are kept (discarding all others) then each block contains only 9 coefficients. After shuffling the result is 9 blocks of 8×8 pixels. This concept is illustrated in Figure 8.2 where the case when a block is 3×3 pixels has been used for clarity. This results with each block to be modelled containing 64 coefficients, however the number of blocks has been reduced. In this case where 9 coefficients per block have been used each super-block of 64×64 pixels has been reduced to 27×27 coefficients.

8.1 Results

Subjective results for the 512×512 Lena image and 384×384 Claudia image are shown in Figures 8.3 and 8.4. The figures show that image quality is better in the JPEG images than the RKi-3 images. Objective results are shown in Figures 8.5 and 8.6 where signal to noise ratio is plotted for various compression ratios. These figures show that JPEG produces better quality images in all cases where the compression ratio is less than 63 (this was the limit of compression for JPEG on these images). An interesting feature of Figures 8.5 and 8.6 is that SnR does

not change much over the range of compression ratios.

Comparing these figures with images produced by RKi-1 and RKi-2 shows RKi-3 producing the lowest quality images for a given compression ratio. This may be for a similar reason to that discussed above where more weights are influenced by a single Gaussian. In RKi-1 and RKi-2 a single block was modelled using the top-left DCT coefficients (usually 8–16 coefficients). All other coefficients were discarded. In RKi-3 DCT coefficients are shuffled so that a block now contains 64 coefficients. All 64 coefficients must be preserved in order to recover the image. This also means that an error in the value of a recovered weight will affect more DCT coefficients (64 per block) than for RKi-1 or RKi-2 (8–16 per block).

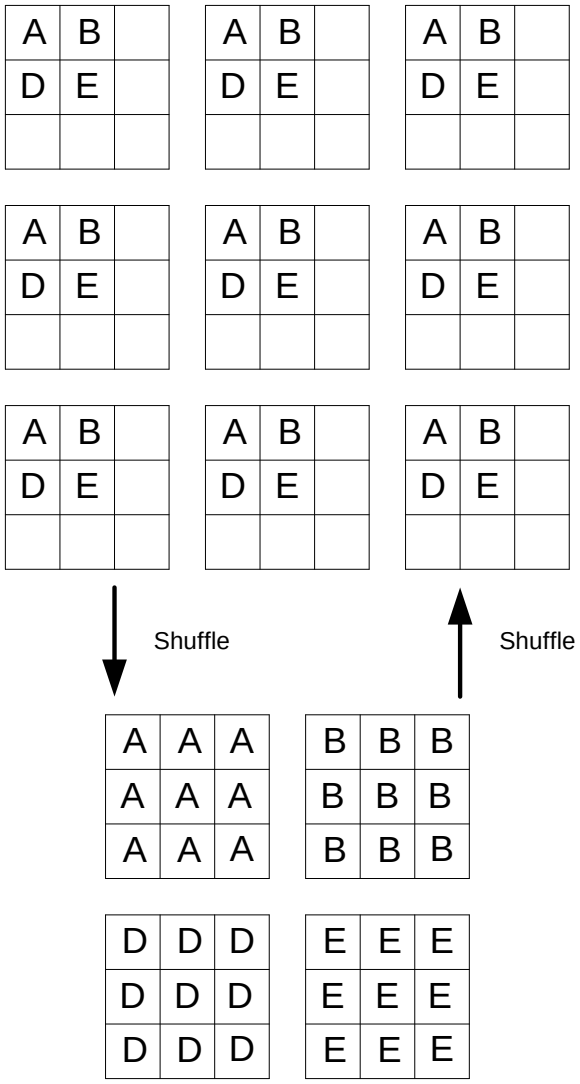

Figure 8.2: Reduction of a 'Super-Block' in RKi-3. Illustrated for block size of 3×3 for clarity (in the actual implementation blocks of size 8×8 are used). In this case the first 4 coefficients from each block are kept, discarding the rest resulting in the reduction from 9×9 coefficients to 6×6 coefficients in a 'super-block'.

Figure 8.3: Subjective comparison of the 512×512 Lena image compressed using both the RKi-3 and JPEG algorithms. Note that the JPEG algorithm would not compress beyond 63:1

Figure 8.4: Subjective comparison of the 384×384 Claudia image compressed using both the RKi-3 and JPEG algorithms. Note that the 384×384 image has been used rather than the 400×400 image as the image dimensions must be a multiple of 64 in the RKi-3 image.

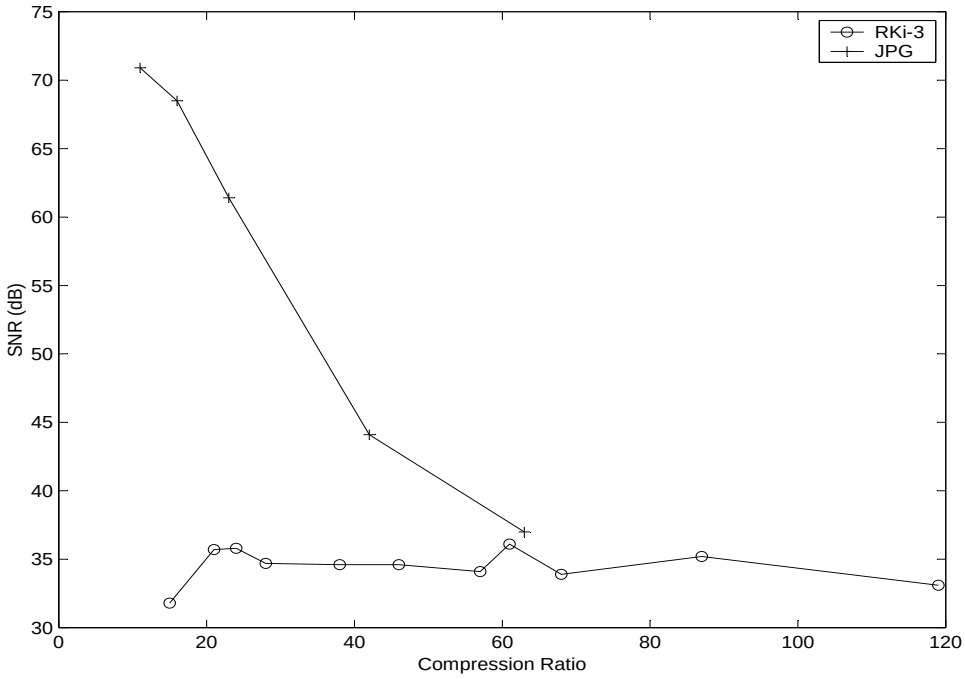

Figure 8.5: Plot of SnR versus compression ratio for the 384×384 Claudia image. Note that JPEG compression is superior to RKi-3.

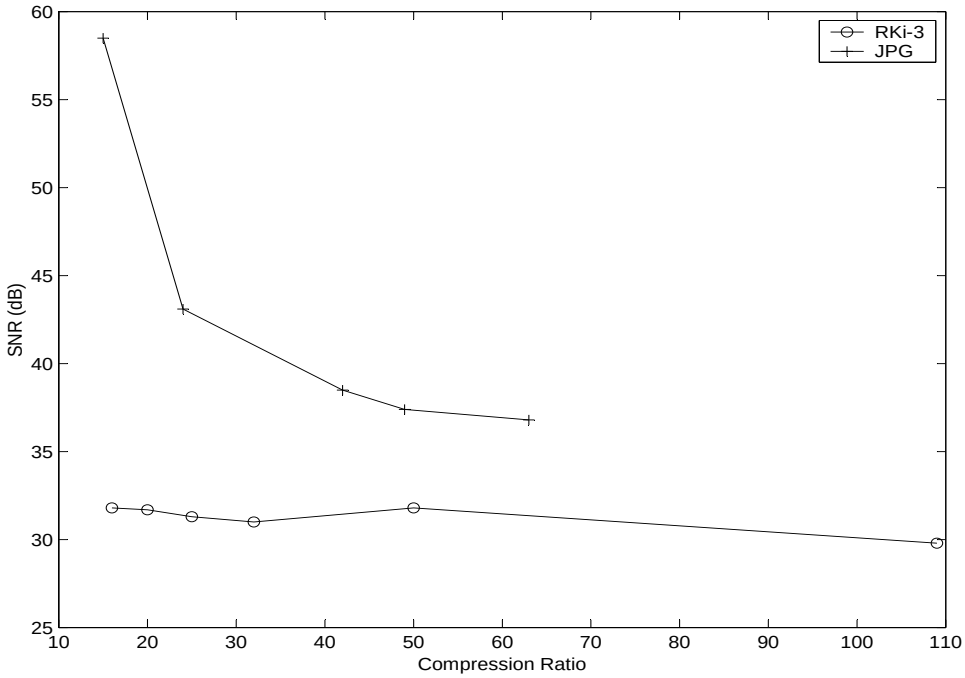

Figure 8.6: Plot of SnR versus compression ratio for the 512×512 Lena image. Note that JPEG compression is superior to RKi-3.

8.2 Discussion

In this chapter an algorithm was described which places coefficients of similar value next to each other in an attempt to reduce the number of support vectors required to model the 2-dimensional coefficient surface. Results were worse than for RKi-2 and RKi-1 as it appears that an error in a weight affects more Gaussians than for RKi-1 and RKi-2.

Chapter 9

Summary and Discussion

This thesis presents several novel algorithms utilising radial basis networks (LOMON) and support vector machines (LOMON-SVM, SVM-Surface modelling, RKi-1, RKi-2 and RKi-3). Each of the algorithms used support vector machines in an attempt to exploit the sparse properties of support vector machines by reducing the number of neurons required to model input training data. The parameters of the neural network were stored using Huffman coding and combinations of run-length and Huffman encoding.

The best performing algorithm was found to be RKi-1 where the 2-dimensional DCT coefficient surface was transformed to a 1-dimensional line and then approximated using SVM learning. This is counter-intuitive as it may be expected that the 2-dimensional surface would be better modelled by a SVM directly using the 2-dimensional surface as input. A theory on why 1-dimensional modelling is better than 2-dimensional is that more DCT coefficients are influenced by a single Gaussian in the 2-dimensional case than the 1-dimensional case. This has the effect that if there is an error in the weight (due to quantising, rounding, reduction of inversion bits etc) then more coefficients are affected resulting in more errors in the recovered image block.

Figures 9.1 and 9.2 show plots of signal to noise ratios for all viable algo-

rithms detailed in this thesis (SVM-surface modelling, RKi-1, RKi-2 and RKi-3) and JPEG for different compression ratios. Figure 9.1 of the 512×512 Lena image clearly shows that the RKi-1 algorithm is the better algorithm at higher compression ratios, followed by the RKi-2 algorithm. Both RKi-1 and RKi-2 perform much better than JPEG on this image. The same is true for the 400×400 Claudia image although this is not as clear as for the Lena image.

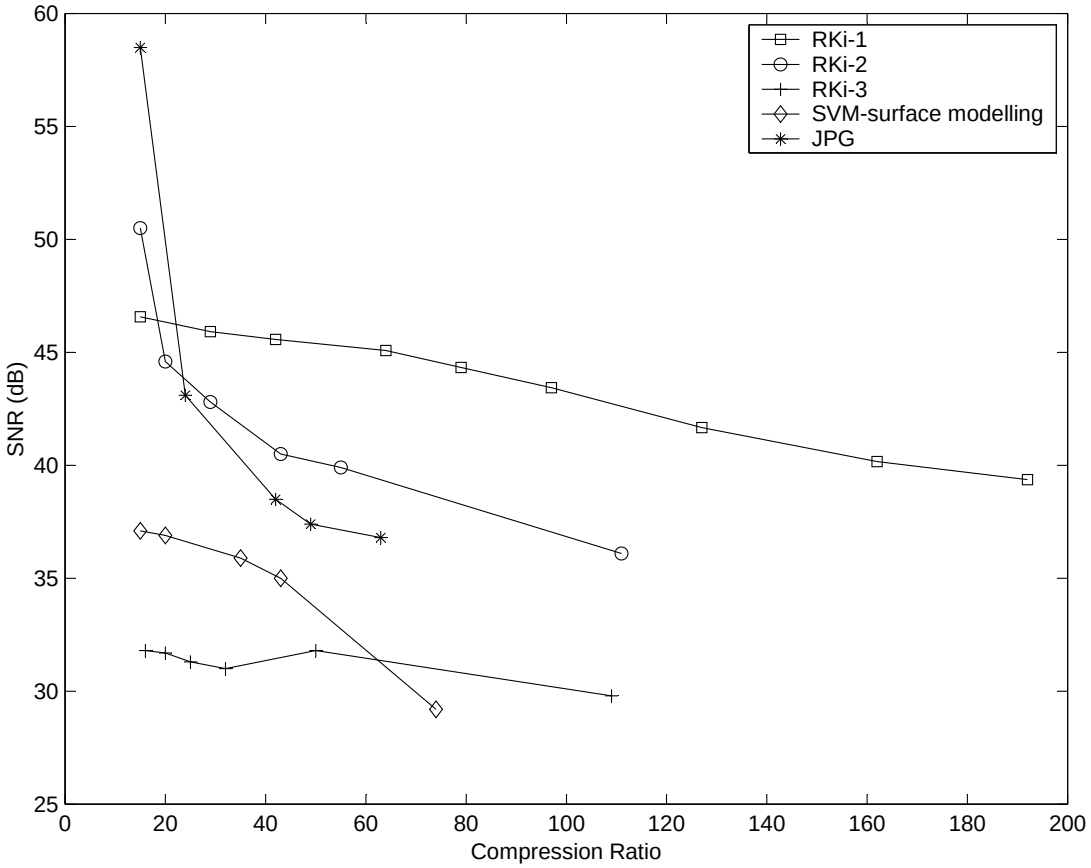

Figure 9.1: Plot of SnR versus compression ratio for the 512×512 Lena image for all algorithms.

9.1 Further Work

The next step in the development of the algorithms described in this thesis is an extension to colour images. It would be expected that compression ratios will

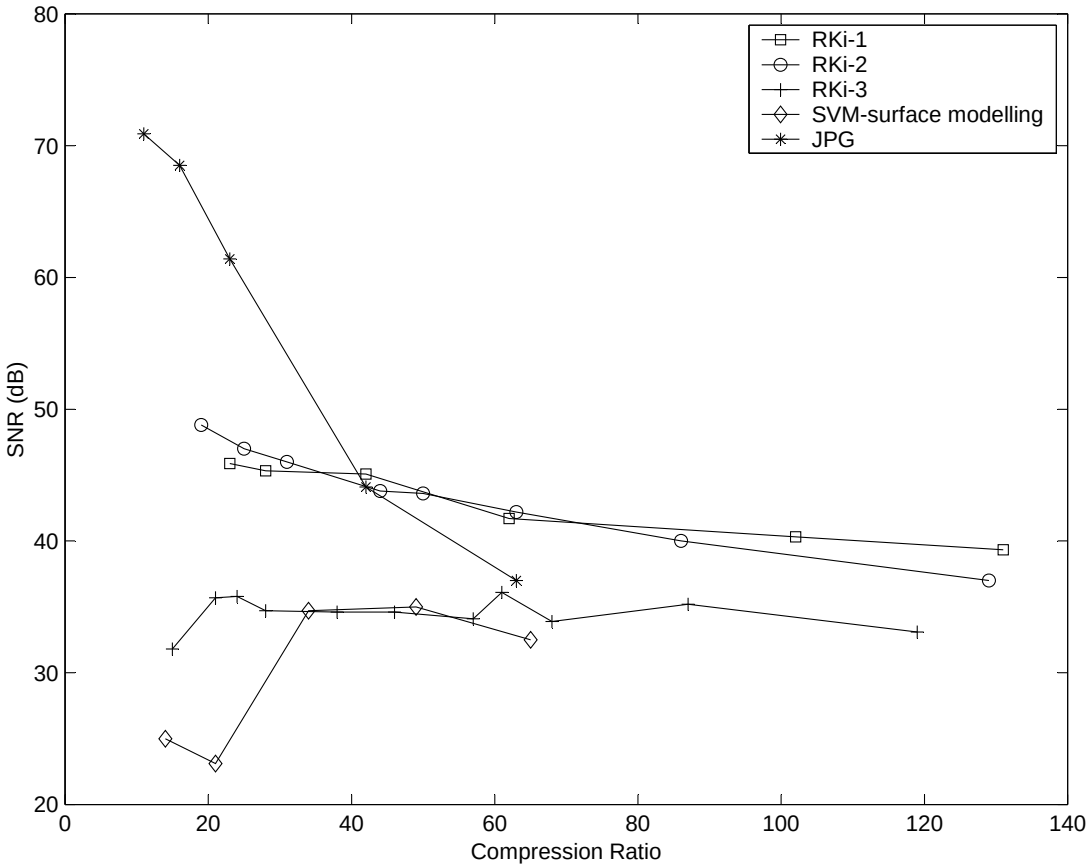

Figure 9.2: Plot of SnR versus compression ratio for the 400×400 Claudia image for all algorithms.

improve considerably if a YCbCr colour model is used as opposed to the RGB (Red Green Blue) colour model. In the YCbCr colour model images are divided into three components: Y - the luminance (i.e. the black and white image); and the two chrominance components Cb (the 'blueness') and Cr (the 'redness'). Most information in an image is contained in the luminance component (hence the reason black and white images were used in this thesis) thus this component can be compressed the least. The Cb component can be compressed further than the Y component as the eye does not notice 'blueness' as much as black and white. The Cr component can be compressed even further.

Another area for further work is the extension of the compression algorithms to moving images (i.e. video) and audio data. MPEG (Moving Pictures Expert

Group) is the video version of JPEG. As RKi-1 can achieve about three times the compression than JPEG at high compression ratios it would be expected that this algorithm would be a good candidate for the basis of a video compression algorithm. Additionally when an image is 'moving' individual frames are only visible for a fraction of a second. Any errors in an individual frame (i.e. image) are 'hidden' as the video progresses.

Appendix I
Overview of the JPEG Compression Standard

The JPEG (Joint Pictures Expert Group) compression standard was established in 1994 [8]. An updated version was released in 2002 [30]. The JPEG standard is large and comprehensive. One of the more unusual aspects of the original 1994 standard is that it did not define a file format. Without a standard file format programs would not be able to exchange images. A defacto standard JFIF (JPEG file interchange format) has emerged and has been widely adopted. Today the term 'JPEG image' refers to a JFIF file. The JPEG group has recently released a file format SPIFF (still picture interchange file format) which was intended to succeed JFIF , but so far it has not been widely adopted.

A brief overview of the 'base-line' JPEG algorithm applied to gray-scale images is described in this appendix.

JPEG starts by subdividing the image into 8×8 blocks. The 2-dimensional discrete cosine transform is applied to each block. After the DCT each block comprises 64 discrete cosine coefficients. The coefficients are comprised of one DC coefficient and 63 AC coefficients. The DC coefficients are separated and treated differently to the AC coefficients. Before the AC and DC coefficients are separated, the matrix of discrete cosine coefficients is divided by a quantisation

Table 9.1: The Y component quantisation table

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

matrix. The JPEG defines standard quantisation tables so these do not need to be transmitted (unless a custom table has been used). Table 9.1 shows the quantisation matrix for the Y (gray-scale) component of an image. The components of this table were found empirically. The components of the quantisation table are largest in the bottom-right corner. This will produce smaller AC coefficients for higher frequency components in the image.

The quantisation table may be multiplied by an arbitrary number. This number is a user defined parameter which defines the compression ratio. A larger number results in larger components in the quantisation table which in turn results in smaller discrete cosine coefficients.

The DC coefficients are Huffman encoded as described in section 4.3.2.

The AC coefficients are mapped into a row of number using the zig-zag mapping described in 6.2.1. The row of AC coefficients will tend to have non-zero coefficients at the beginning of the row while toward the end of the row, the coefficients are usually zero. Increasing the number by which the quantisation matrix is multiplied will further increase the number of zeros.

The row of AC coefficients is Huffman coded. Together with the Huffman encoded DC coefficients a binary file is produced. The above steps are carried out in reverse order to recover the image.

Appendix II
Contents of the Accompanying CD

A CD is located in the back cover of this thesis. This contains a working Windows application implementing the algorithms detailed in this thesis. To install, insert the CD and an installation window should automatically appear. If the installation window does not automatically appear, select **Run** from the Windows **Start** menu and type **d:\setup.exe** where **d** is the letter of your CD drive.

After installation a 'Pictures' folder will be installed in the program folder. This contains sample images compressed using SVM-surface modelling, RKi-1, RKi-2 and RKi-3.

References

[1] V. Kecman. *Learning and Soft Computing.* The MIT Press, Cambridge, MA, 2001.

[2] T. Hopper, C. Brislawn, and J. Bradley. Wsq gray-scale fingerprint image compression specification, revision 1.0. Technical report, Federal Bureau of Investigation, November 1992.

[3] Computer. 34(9), September 2001.

[4] L. P. Deutsch. *"DEFLATE Compressed Data Format Specification", version 1.3, RFC 1951.* 1996.

[5] L. P. Deutsch and J. Gailly. *"ZLIB Compressed Data Format Specification", version 3.3, RFC 1950.* 1996.

[6] Federal Bureau of Investigation. Wsq gray-scale fingerprint image compression specification. Technical Report IAFIS-IC-0110v2, Washington DC, February 1993.

[7] P. N. Topiwala (Editor). *Wavelet Image and Video Compression.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1998.

[8] JPEG (1994). *Digital Compression and Coding of Continuous-tone Still Images.* ISE/IEC IS 10918-1. American National Standards Institute, New York, NY, 1994.

[9] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, NY, 1993.

[10] CompuServe Inc. *Graphics Interchange Format (GIF) Specification*. CompuServe, Columbus, OH, 1987.

[11] CompuServe Inc. *Graphics Interchange Format (GIF) Specification, Version 89a*. CompuServe, Columbus, OH, 1989.

[12] *Playboy Magazine*, November 1972.

[13] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, 1995.

[14] S. B. Gokturk, C. Tomasi, B. Acar, C. F. Beaulieu, D. S. Paik, R. B. Jeffrey, J. Yee, and S. Napel. A statistical 3-d pattern processing method for computer-aided detection of polyps in ct colonography. *IEEE Transactions on Medical Imaging*, 20(12):1251–1260, December 2001.

[15] S. Chen, A. K. Samingan, and L. Hanzo. Support vector machine multiuser receiver for ds-cdma signals in multipath channels. *IEEE Transactions on Neural Networks*, 12(3), May 2001.

[16] J. Robinson and V. Kecman. The use of support vectors in image compression. *Proceedings of the* 2^{nd} *International Conference on Engineering Intelligent Systems, University of Paisley*, June 2000.

[17] O. Nelles, R. Hecker, and R. Isermann. Automatic model selection in local linear model trees (lolimot) for nonlinear system identification of a transport delay process, kitakyushu, fukuoka, japan. 11^{th} *IFAC Symposium on System Identification (SYSID '97)*, July 1997.

[18] V. N. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, 1998.

[19] V. N. Vapnik. *The Nature of Statistical Learning Theory.* Springer, 1995.

[20] N. Cristianini and J. Shawne-Taylor. *An Introduction to Support Vector Machines and other Kernal Based Learning Methods.* Cambridge University Press, 2000.

[21] H. Drucker, C. J. C. Burges, L. Kaufmann, A. Smola, and V. Vapnik. *Support Vector Regression Machines, 155–161.* Advances in Neural Information Processing Systems 9. MIT Press, Cambridge, MA, 1997.

[22] V. Vapnik, S. Golowich, and A. Smola. *Support Vector Method for Function Approximation, Regression Estimation, and Signal Processing*, volume 9 of *Advances in Neural Information Processing Systems 9.* MIT Press, Cambridge, MA, 1997.

[23] W. L. Winston. *Operations Research: Applications and Algorithms.* Duxbury Press and International Thomson Publishing, 1993.

[24] C. Lawson and R. Hanson. *Solving Least Square Problems.* Prentice-Hall, 1974.

[25] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(3):1098–1101, 1952.

[26] J. Robinson and V. Kecman. Combining support vector machine learning with the discrete cosine transform in image compression. *IEEE Transactions on Neural Networks*, 14(4):950–958, July 2003.

[27] J. Robinson and V. Kecman. Exploitation of sparse properties of kernel machines in image compression. *Proceedings of the Internation Conference on Neural Networks, Portland, Oregon*, July 2003.

[28] C. Amerijckx, M. Verleysen, P. Thissen, and J. Legat. Image compression by self-organized kohonen map. *IEEE Transactions on Neural Networks*, 9(3), May 1998.

[29] J. Jiang. Image compression with neural networks – a survey. *Signal Processing: Image Communication*, 14, 1999.

[30] D S. Taubman and M. W. Marcellin. *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers.