8-7-2003

Image Compression Using Cascaded Neural Networks

Chigozie Obiegbu
University of New Orleans

IMAGE COMPRESSION USING CASCADED NEURAL NETWORKS

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
The Department of Electrical Engineering

by

Chigozie Obiegbu

B.Eng., Federal University of Technology Owerri, 1997

August 2003

ACKNOWLEDGMENTS

The completion of this thesis has involved an enormous amount of help from a number of people. First and foremost of these is Dr. Dimitrios Charalampidis, my thesis advisor, for providing the original ideas, suggestions, and motivation for the work. He freely bestowed his time, guidance, brilliance, and wisdom considerably beyond the call of duty; and was a model of professorial responsibility, professionalism, and commitment. What I have learned from him cannot be quantified. Special thanks to other members of my thesis committee, Dr. Juliette Ioup and Dr. Terry Riemer for enriching my experience in academia by sharing with me their intellectual curiosity, professional insight and integrity, and personal warmth and understanding through their courses. I would especially like to thank Dr. Juliette Ioup for carefully reading through the entire draft of my thesis and offering several helpful editorial comments.

If anyone has had to be patient with me in the course of writing this thesis, it is my special friend, Melissa Bias. Her companionship has helped me to put forth my full effort, and to maintain my sanity. Sincere appreciation goes to Vijay Kura, a fellow graduate student and a good friend, for lending some of his exquisite programming skills at the beginning stages. And last, but most of all, to my parents, James and Mmachukwu Obiegbu, I owe everything; they sustain me in all that I do and it is to them that this work is dedicated with love; and in loving memory of my dearest cousin, Uchenna Ebeledike.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Images are forming an increasingly large part of modern communications, bringing the need for efficient and effective compression. Many techniques developed for this purpose include transform coding, vector quantization and neural networks. In this thesis, a new neural network method is used to achieve image compression. This work extends the use of 2-layer neural networks to a combination of cascaded networks with one node in the hidden layer. A redistribution of the gray levels in the training phase is implemented in a random fashion to make the minimization of the mean square error applicable to a broad range of images. The computational complexity of this approach is analyzed in terms of overall number of weights and overall convergence. Image quality is measured objectively, using peak signal-to-noise ratio and subjectively, using perception. The effects of different image contents and compression ratios are assessed. Results show the performance superiority of cascaded neural networks compared to that of fixed-architecture training paradigms especially at high compression ratios. The proposed new method is implemented in MATLAB. The results obtained, such as compression ratio and computing time of the compressed images, are presented.

CHAPTER 1

Introduction

Computer images are extremely data intensive and hence require large amounts of memory for storage. As a result, the transmission of still images from one machine to another can be very time consuming. For this reason still image compression is subject of an intense worldwide research effort [1]-[8]. By using data compression techniques, it is possible to remove some of the redundant information contained in images, requiring less storage space and less time to transmit. The objective of digital image compression techniques is the minimization of the number of bits required to represent an image, while maintaining an acceptable image quality. Another issue in image compression and decompression is the processing speed, especially in real-time applications. It is often desirable to be able to carry out compression and decompression in real-time without reducing image quality.

Numerous *lossy* image compression techniques have been developed in the past years. The transform-based coding techniques have proved to be the most effective in obtaining large compression ratios while retaining good visual quality. In low noise environments, where the bit error rate is less than 10^{-6}, the JPEG [3]-[4] picture compression algorithm, which employs cosine-transforms, has been found to obtain excellent results in many digital image compression applications. However, an increasing number of applications are required for use in high noise environments [12] - [14], e.g.,

the transmission of a compressed picture over a mobile or cordless telephone. In these applications, the bit error rates due to noise on the transmission channel may be as high as 10^{-2} [12], [14]. At these error rates, JPEG and similar compression algorithms, which rely on entropy coding, are not suitable [12].

Recently, neural networks have proved to be useful in image compression because of their parallel architecture and flexibility [15]-[20]. They do not use entropy coding and are therefore intrinsically robust, making them an attractive choice for high noise environments. Of course a price must be paid for this robustness. In this case, the price is a reduction in decompressed image quality for the same compression efficiency. However, as shall be seen in chapter 4, the reduction in image quality is not excessive.

Although the parallel-structure neural networks are robust, they suffer from several drawbacks. The main drawbacks are:

1) high computational complexity;
2) moderate reconstructed picture quality;
3) variable bit-rate;

In this thesis the first two drawbacks are tackled in a proposed neural network (NN) method that uses a cascade of feedforward networks with one node at the hidden layer. The third drawback is not directly tackled to avoid increasing the computational complexity of the system. However, provision is made for the rare occasions when the compression efficiency is much lower than expected. Compared to the current parallel-structure NNs, the proposed NN has a lower computational complexity, faster convergence, and higher compression efficiency.

The chapters of this thesis are organized as follows: Chapter 2 briefly reviews three of the major approaches to image compression, namely predictive coding, transform coding and vector quantization. In particular, the JPEG technique is described. Chapter 3 discusses NN technology, including various architectures and algorithms. These architectures include Backpropagation, Radial-basis function, General Regression, Modular, Probabilistic, and Learning Vector Quantization. The single-structure NN consisting of two layers and the parallel-structure NN are embedded in the discussion. In Chapter 4, the single-structure NN as an image compression tool is implemented, and the proposed NN method using a cascade of feedforward networks is presented. A background on this approach is given followed by its application to image compression. Chapter 5 presents experimental results. The error metrics used in computing the results are described. Graphical and pictorial results are fully illustrated. Performance comparisons are made with JPEG and single-structure/parallel-structure NNs. Issues relating to a comprehensive evaluation of the image compression techniques presented herein are discussed in Chapter 6. The practicality and usefulness of the new method is mentioned.

The thesis concludes by suggesting certain modifications to the new NN method to achieve even better results. Specific remarks on the development of the code used in the new NN method assists in highlighting the usefulness of the completed work.

CHAPTER 2

Techniques for Image Compression

Image compression methods are categorized as lossless and lossy. Lossless methods preserve all original information without changing it. Lossy methods reduce information to attain better compression ratio. Lossless compression has been found to be adequate when low compression ratios are acceptable. Significantly substantial compression ratios can only be achieved with lossy compression schemes, which will be the main focus of this thesis. Due to the extensive breadth of this field, it is impossible to list all of the currently available image compression techniques. However, existing research fall into one of three major categories: vector quantization, predictive coding, or transform coding.

2.1 Vector Quantization

Quantization refers to the process of approximating a continuous set of values in the image data with a finite (preferably small) set of values. The input to a quantizer is the original data, and the output is always one among a finite number of levels. The quantizer is a function whose set of output values is discrete and usually finite. The concept of quantizing data can be extended from scalar or one-dimensional data to vector data of arbitrary dimension.

Vector Quantization (VQ) [9] – [11] uses a codebook containing pixel patterns with corresponding index for each of them. The main idea of VQ in image coding is then to represent arrays of pixels by an index in the codebook. In this way, compression is achieved since the size of the index is usually a small fraction of that of the block of pixels. The codebook is used to quantize the incoming vectors and is analogous to the quantization levels in a scalar quantizer. A good codebook can reduce the overall distortion of the reconstructed image, and is determinative to the performance of the VQ process.

At the encoder, the incoming image is partitioned into blocks of sub-images. These blocks have the dimension equal to entries in codebook, so comparison can be done easily between them. An input vector X_n consisting of blocks of pixels is quantized. This is done by encoding X_n into a binary index i_n which points to an entry in the codebook. This i_n then serves as an index for the output reproduction vector or codeword. The standard approach to calculate the codebook is by way of the Linde, Buzo, and Gray (LBG) algorithm [9]. Finally, the concatenation of all i_n represents the compressed image. However, the choice of index i_n will be different when using different mapping rules. These mapping rules often depend on minimizing a predefined distortion measure $d(X_n,Y_{in})$, where X_n and Y_{in} are the vectors from the image and from the codebook respectively. A reliable assessment criterion based on the properties of the human visual system has not yet been defined; hence Euclidean metrics are adopted as a default distortion measure. Besides the LGB algorithm, other iterative approaches relate to neural network models and are based on *Kohonen Self-Organizing Maps* (SOMs) [21].

The main advantages of VQ are the simplicity of its idea and the possible efficient implementation of the decoder. Moreover, VQ is theoretically an efficient method for image compression, and superior performance will be gained for large vectors. However, in order to use large vectors, VQ encoding becomes complex, which requires many computational resources (e.g., memory, computations per pixel) in order to efficiently construct and search a codebook. For instance, while the LGB algorithm converges to a local minimum, it is not guaranteed to reach the global minimum. In addition, the algorithm is very sensitive to the initial codebook. Furthermore, the algorithm is slow since it requires, on each iteration, an exhaustive search through the entire codebook. More research on reducing this complexity must be done in order to make VQ a practical image compression method with superior quality.

2.2 Predictive Coding

Predictive image coding algorithms [5] are used primarily to exploit correlation between adjacent pixels. They predict the value of a given pixel based on the values of the surrounding pixels. Due to the correlation property among adjacent pixels in an image, the use of predictor can reduce the amount of information bits required to represent the image. This can accomplished through the use of predictive coding or differential pulse-code modulation (DPCM).

The predictor uses past samples $x(n-1), x(n-2), \ldots, x(n-p)$, or in the case of images, neighboring pixels, to calculate an estimate, $\hat{x}(n)$, of the current sample. It the difference between the true value and the estimate, namely $e(n) = x(n) - \hat{x}(n)$, which is used for storage transmission. As the accuracy of the predictor increases, the variance of

the difference decreases, resulting in higher predictive gain and therefore a higher compression ratio.

The problem of course is how to design the predictor. One approach is to use a statistical model of the data to derive a function which relates the values of the neighboring pixels to that of the current one in an optimal manner. An autoregressive model (AR) is one such model which has been successfully applied to images. For a pth order causal AR process, the nth value $x(n)$ is related to the previous p values in the following manner:

$$x(n) = \sum_{j=1}^{p} w_j x(n-j) + e_n, \qquad (2.1)$$

where $\{w_j\}$ is a set of AR coefficients, and $\{e_n\}$ is a set of zero-mean independent and identically distributed random variables. In this case, the predicted value is a linear sum of neighboring samples (pixels) as shown by

$$\hat{x}(n) = \sum_{j=1}^{p} w_j x(n-j). \qquad (2.2)$$

Equation (2.2) is the basis of linear predictive coding. To minimize the mean squared error $E[(\hat{x}-x)^2]$, the following relationship must be satisfied:

$$Rw = d, \qquad (2.3)$$

where $[\mathrm{R}]_{ij} = E[x(i)x(j)]$ is $ijth$ element of the autocovariance matrix R and $d_j = E[\hat{x}(n)x(j)]$ is the jth element of the cross covariance vector d. Knowing R and d, the unknown coefficient vector w can be computed, and the AR model (i.e., predictor) is thereby determined.

2.3 Transform Coding

Another approach to image compression is the use of transformations that operate on an image to produce a set of coefficients [5]. A subset of these coefficients is chosen and quantized for transmission across a channel or for storage. The goal of this technique is to choose a transformation for which such a subset of coefficients is adequate to reconstruct an image with minimum discernible distortion.

A simple and powerful class of transform coding techniques is linear block transform coding. An image is subdivided into non-overlapping blocks of $n \times n$ pixels which can be considered as N-dimensional vectors x with $N = n \times n$. A linear transformation, which can be written as an $M \times N$-dimensional matrix W with $M \leq N$, is performed on each block, with the M rows of W, w_i being the basis vectors of the transformation. The resulting M-dimensional coefficient vector y is calculated as

$$y = Wx. \qquad (2.4)$$

If the basis vectors w_i are orthogonal, that is,

$$w_i^T w_j = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}, \qquad (2.5)$$

then the inverse transformation is given by the transpose of the forward transformation matrix resulting in the reconstructed vector:

$$\hat{x} = W^T y. \qquad (2.6)$$

The optimal linear transformation for minimizing the mean squared error is the Karhunen-Loeve transformation (KLT) [18]. The transformation matrix W consists of M rows of the eigenvectors corresponding to the M largest eigenvalues of the sample autocovariance matrix

$$\Sigma = E[xx^T]. \qquad (2.7)$$

The KLT also produces uncorrelated coefficients and therefore results in the most efficient coding of the data since the redundancy due to the high degree of correlation between neighboring pixels is removed. The KLT is related to principal component analysis (PCA) [23], since the basis vectors are also the M principal components of the data. Because the KLT is an orthogonal transformation, its inverse is simply its transpose.

A number of practical difficulties exist when trying to implement the above approach. The calculation of the estimate of the covariance of an image may be unwieldy and may require a large amount of memory. In addition, the solution for the eigenvectors and eigenvalues is computationally intensive. Finally, the calculation of the forward and inverse transforms is of order $O(MN)$ for each image block. Due to these difficulties, fixed-basis transforms such as the discrete cosine transform (DCT) [22], which can be computed in order, $O(N \log N)$, are typically used when implementing block transform schemes.

2.3.1 Discrete Cosine Transform

The currently accepted standard for lossy still image compression was developed by the Joint Photographic Experts Group (JPEG) [3], [4], which adopted the linear block transform coding approach for its standard using the DCT as the transformation [22]. The JPEG specification defines a minimal subset of the standard, called baseline JPEG, which all JPEG-aware applications are required to support. This baseline uses an encoding scheme based on the DCT to achieve compression.

Figure 1 describes the baseline JPEG process. The compression scheme is divided into the following stages:

1. Apply a DCT to blocks of pixels, thus removing redundant image data.
2. Quantize each block of DCT coefficients using weighting functions optimized for the human eye.
3. Encode the resulting coefficients (image data) using a Huffman variable word-length algorithm to remove redundancies in the coefficients.

The image is first subdivided into 8 x 8 blocks of pixels. As each 8 x 8 block or sub-image is encountered, its 64 pixels are level shifted by subtracting the quantity 2^{n-1}, where 2^n is the maximum number of gray levels.

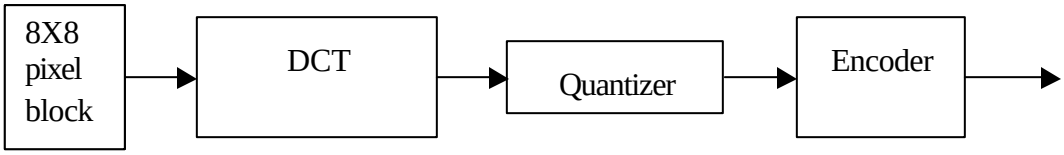

Figure 1.0: Block diagram of JPEG compression.

The 2-D discrete cosine transform of the block is then computed. The DCT helps separate the image into parts (or spectral sub-bands) of differing importance with respect to the image's visual quality. The DCT is similar to the discrete Fourier transform: it transforms a signal or image from the spatial domain to the spatial frequency domain. With an input image, A, the output image, B is:

$$B(u,v) = \frac{1}{4}C(u)C(v)\sum_{i=0}^{N_1-1}\sum_{j=0}^{N_2-1}A(i,j)\cos\left[\frac{pu}{2N_1}(2i+1)\right]\cos\left[\frac{pv}{2N_2}(2j+1)\right]. \qquad (2.8)$$

where $C(u)$, $C(v) = 1/\sqrt{2}$ for $u,v = 0$ and 1 otherwise

The input image is N_2 pixels wide by N_1 pixels high; $A(i, j)$ is the intensity of the pixel in row i and column j. $B(u, v)$ is the DCT coefficient in row u and column v of the DCT matrix. The DCT input is an 8 by 8 array of integers. This array contains each pixel's gray scale level; 8 bit pixels have levels from 0 to 255. The output array of DCT coefficients contains integers; these can range from -1024 to +1023. For most images, much of the signal energy lies at low frequencies; these appear in the upper left corner of the DCT. The lower right values represent higher frequencies, and are often small - small enough to be neglected with little visible distortion. A quantizer rounds off the DCT coefficients according to a *quantization matrix*. This matrix is the 8 by 8 matrix of step sizes (sometimes called *quantums*) - one element for each DCT coefficient. It is usually symmetric. Step sizes will be small in the upper left (low frequencies), and large in the lower right (high frequencies); a step size of 1 is the most precise. The quantizer divides the DCT coefficient by its corresponding quantum, and then rounds to the nearest integer. Large quantums drive small coefficients down to zero. The result: many high frequency coefficients become zero, and therefore easier to code. The low frequency coefficients undergo only minor adjustment. This step causes the lossy nature of JPEG, but allows for large compression ratios.

After quantization, it is not unusual for more than half of the DCT coefficients to equal zero. JPEG incorporates run-length coding to take advantage of this. For each non-zero DCT coefficient, JPEG records the number of zeros that preceded the number, the number of bits needed to represent the number's amplitude, and the amplitude itself. To consolidate the runs of zeros, JPEG processes DCT coefficients in the zigzag pattern shown in figure 2.

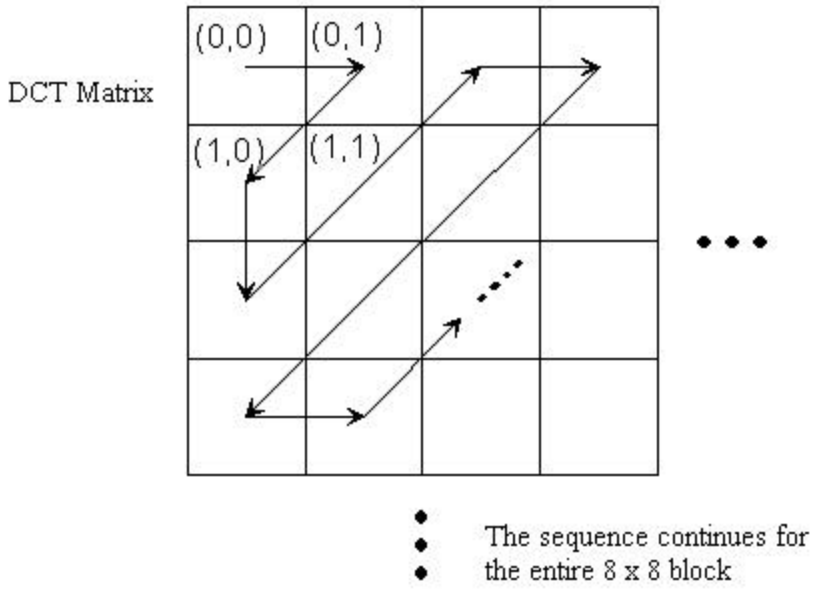

Figure 2. Zig-zag Sequence for Binary Encoding

The number of previous zeros and the bits needed for the current number's amplitude form a pair. Each pair has its own code word, assigned through a variable length code (for example Huffman, Shannon-Fano or Arithmetic coding). JPEG outputs the code word of the pair, and then the codeword for the coefficient's amplitude (also from a variable length code). After each block, JPEG writes a unique end-of-block sequence to the output stream, and moves to the next block. When finished with all blocks, JPEG writes the end-of-file marker. At this point, the JPEG data stream is ready to be transmitted across a communications channel or encapsulated inside an image file format.

JPEG is not always an ideal compression solution. There are several reasons:

- Does not fit every compression need. Images containing large areas of a single color do not compress very well.
- Can be rather slow when it is implemented only in software.

- Hard to implement.
- Not supported by very many file formats.

Recently, novel approaches have been introduced based on pyramidal structures [24], wavelet transforms [25], and fractal transforms [26]. These and some other new techniques [27] inspired by the representation of visual information in the brain can achieve high compression ratios with good visual quality but are nevertheless computationally intensive.

With this brief review of conventional image compression techniques at hand, various types of neural networks and their architectures will be reviewed, and their role as an image compression tool considered.

CHAPTER 3

Artificial Neural Network Technology- an Overview

Artificial Neural networks are software or hardware systems that try to simulate the human brain functionality. From the beginning of their presence in science, Neural Networks (NNs) are being investigated with two different scientific approaches. First, the biological aspect explores NNs as simplified simulations of the human brain and uses them to test hypotheses about human brain functioning. The second approach treats NNs as technological systems for complex information processing. This thesis is focused on the second approach by which NNs are evaluated according to their efficiency to deal with complex problems, especially in the areas of association, classification and prediction, but specifically in the area of image processing.

The reasons why NNs often outperform classical statistical methods lie in their abilities to analyze incomplete, noisy data, to deal with problems that have no clear-cut solution and to learn on historical data. Because of those advantages, they have shown remarkable success in areas such as image transmission over high-noise environments. NNs, however, do have disadvantages. One such is the lack of tests of statistical significance of NN models and parameters estimated [32], [35]. Furthermore, there are no established paradigms for deciding which architecture is the best for certain problems and data types. This problem is partly investigated in this thesis. Despite those disadvantages, many research results show that neural networks can solve almost all problems more

efficiently than traditional modeling and statistical methods. It is mathematically proven (using the Ston-Weierstrass, Hahn-Banach and other theorems and corollaries [36]) that two-layer neural networks having arbitrarily squashing transfer functions are capable of approximating any nonlinear function.

3.1 Basic Principles of Learning in Neural Networks

NNs consist of one or more layers or groups of processing elements called neurons. The term neuron denotes a basic unit of a neural network model intended for data processing. Neurons are connected into a network in a way that the output of each neuron represents the input for one or more other neurons. The connection between neurons can be either one-directional or bi-directional, and according to its intensity the connection can either be excitatory or inhibitory. Neurons are grouped into layers. There are two main types of layers: hidden and output layers. Some authors refer to the inputs as another layer, but this will not be the case in this thesis. The hidden layer receives input data. Here, the information is processed and sent to the output layer neurons, where the network output is compared to the desired output and the network error is computed. The error information then flows backward through the network and the values of connection weights between the neurons are adjusted using the error term. The process is repeated in the network for the number of iterations necessary to achieve the output closest to the desired (actual) output. Finally, the network output is presented to the user. Neural network learning is basically the process by which the system arrives at the values of connection weights between neurons. The connection weight is the strength of the connection between two neurons. If, for example, neuron j is connected to neuron i,

w_{ji} denotes the connection weight from neuron j to neuron i (w_{ij} is the weight of the reverse connection from neuron i to neuron j). If neuron i is connected to neurons called $1,2,...,n$, their weights are stored in the variables w_{1i}, w_{2i}, w_{ni}. A neuron receives as many inputs as there are input connections to that neuron and produces a single output to other neurons according to a transfer function.

The process of neural network design consists of four phases:

1. arranging neurons in various layers,
2. determining the type of connections between neurons (inter-layer and intra-layer connections),
3. determining the way neuron receives input and produce output, and
4. determining the learning rule for adjusting the connection weights.

The result of NN design is the NN architecture. According to the above design processes, the criteria to distinguish NN architectures are as follows:

- number of layers,
- type of connection between neurons,
- connection between input and output data,
- input and transfer functions,
- type of learning,
- certainty of firing,
- temporal characteristics, and
- learning time.

3.1.1 Type of Connection between Neurons

Connections in the network can be realized between two layers (inter-layer connections) and between neurons in one layer (intra-layer connections) [28]. Inter-layer connections can be classified as: fully connected - each neuron in the first layer is connected to each neuron in the second layer; partially connected - each neuron in the first layer should not necessarily be connected to every neuron in the second layer; feed-forward - connection between neurons is one-directional, neurons in the first layer send their output to the neurons in the second layer, but they do not receive any feedback; bi-directional - there is a feedback when the neurons from the second layer send their output back to the neurons in the first layer; hierarchical - neurons in one layer are connected only to the neurons of the next neighbor layer; resonance - two-directional connection where neurons continue to send information between layers until a certain condition is satisfied.

Examples of some well known NN architectures with inter-layer connections:

- Perceptron (developed by Frank Rosenblat, 1957) - first NN, two-layered, fully connected,
- ADALINE (developed by Bernard Widrow, Marcian E. Hoff, 1962) - two-layered, fully connected,
- Backpropagation (developed by Paul Werbos, 1974, extended by Rumelhart, Hinton, Williams, 1986) - first NN with one or more hidden layers, connection between hidden layers is hierarchical,
- ART (Adaptive Resonance Theory) (designed by Steven Grosberg, 1976) -resonance connection, three-layered network,

- Feedforward Counterpropagation (designed by Robert Hecht-Nielsen, 1987) -structure similar to Backpropagation network, three-layered, but non-hierarchical. There is also a connection between neurons in one layer.

Connections between neurons in one layer (intra-layer) can be:

a) Recurrent - neurons in one layer are fully or partially connected. The connection is realized in a way that neurons communicate their outputs with each other after they receive their inputs from another layer. The communication continues until neurons reach a stable condition. When the stable condition is reached, neurons are allowed to send their output to the next layer.

b) On-center/off-surround - in this connection a neuron in one layer has an excitatory connection toward itself and toward the neighbor neurons, but an inhibitory connection toward other neurons in the layer.

Some of the intra-layer networks with recurrent connection are:

- Hopfield's network (designed by John Hopfield, 1982) - two-layered, fully-connected, neurons of output layer are mutually connected with recurrent intra-layer connection,
- Recurrent Backpropagation network (designed by David Rumelhart, Geoffrey Hinton, Ronald Williams, 1986) - recurrent intra-layer connection, but one-layered, where part of the neurons receive inputs, and the other part is fully connected with recurrent intra-layer connection,

and some of the networks with on-center/off-surround connection are:

- ART1, ART2, ART3 (designed by Steven Grosberg, 1960) - resonance on-center/off-surround connection,

- Kohonen's self-organizing network (created by Teuvo Kohonen, 1982),
- Counterpropagation networks,
- Competitive learning networks.

Details on the above architectures are discussed later in the text.

3.1.2 Connection between Input and Output Data

NNs can also be distinguished according to the connection between input and output that can be:

1) autoassociative - input vector is the same as output (common in pattern recognition problems, where the objective is to obtain the same data in output as they are in input),

2) heteroassociative - output vector differs from the input vector.

Autoassociative networks [17], [36], [37] are used in pattern recognition, signal processing, noise filtering and similar problems that aim to recognize the patterns of input data.

3.1.3 Input and Transfer Functions

In order to understand the main types of NN architectures that will be explored below, the basic principles of NN functioning will be described through the equations of input and output of neurons, transfer functions and learning rules.

Input (Summation) Functions

When a neuron receives input from the previous layer, the value of its input is computed according to an input function, usually called a "summation" function. The

simplest summation function for the neuron i is determined by multiplying the output sent by the neuron j to the neuron i (denoted as $output_j$) with the connection weight between neurons i and j, then summarizing those multiplications for all j neurons connected to neuron i, as given by:

$$input_i = \sum_{j=1}^{n} \left(w_{ji} \cdot output_j \right), \qquad (3.1)$$

where n is the number of neurons in the layer that sends its output received by the neuron i. In other words, $input_i$ of a neuron i is the sum of all weighted outputs that arrive into that neuron. Besides this standard network input, there are two additional specific types of inputs in a network: external input and bias. For the former, neuron i receives input from the external environment. For the latter, a bias value is used for neuron activation control in some networks. Input values can be normalized to an interval (usually [0,1] or [-1,1]) to avoid the extreme influence of high-valued inputs. Therefore, normalization is recommended in most neural networks (it is obligatory in Kohonen's network) [16], [21]. Details about data normalization used in this thesis will be explained later in the text.

Output (Transfer) Functions

After receiving the input according to the summation function presented in formula (3.1), the output of a neuron is computed and sent to the other neurons it is connected to (usually to the next layer neurons). The output of a neuron is computed according to a transfer function, which may be a linear or a nonlinear function of its input. A particular transfer function is chosen to satisfy some specification of the problem

that the neuron is attempting to solve. Several of the most frequently used transfer functions are the step function, signum function, sigmoid function, hyperbolic-tangent function, linear function, and threshold linear function. The output of each transfer function is computed according to a set formula.

Only two of the above-mentioned transfer functions are used in this thesis, namely, the hyperbolic-tangent and the linear functions. The former has the form:

$$output_i = \frac{e^u - e^{-u}}{e^u + e^{-u}} \qquad (3.2)$$

where $u = g \cdot input_i$. g =1/T is the gain of the function, where T is the threshold. The gain determines the skewness of the function around 0. The function has continuous values in the interval [-1,1]. The hyperbolic-tangent function is commonly used in multilayer networks that are trained using the backpropagation algorithm, in part because this function is differentiable. The graph is shown in the following the figure below:

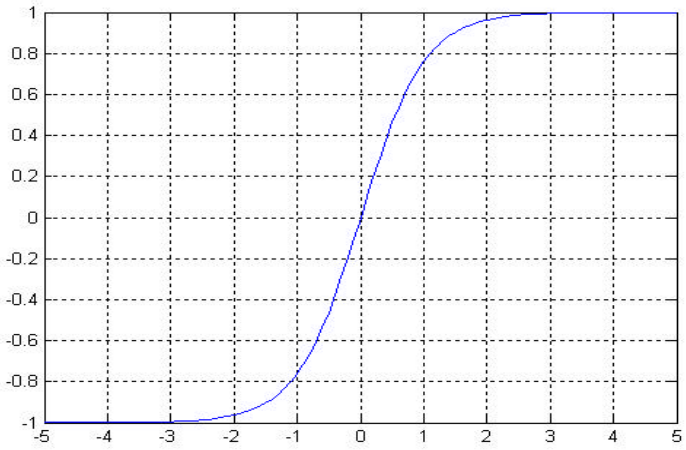

Figure 3.1: Graph of hyperbolic tangent function

Because of its ability to map values into positive as well as negative regions, this function is used throughout Matlab implementation in this thesis.

A linear function has the form:

$$output_i = g \cdot input_i \qquad (3.3)$$

It should be pointed out that Matlab names the hyperbolic tangent *tansig*, which has the same shape, and the linear function *purelin*. Figure 3.2 below depicts the overall picture of the input, transfer function, and output of a typical multiple-input neuron.

Choice of the appropriate transfer function is made in the network design phase, still allowing the change of threshold value (T) and gain (g). The best transfer function is usually obtained by experimenting on a particular problem.

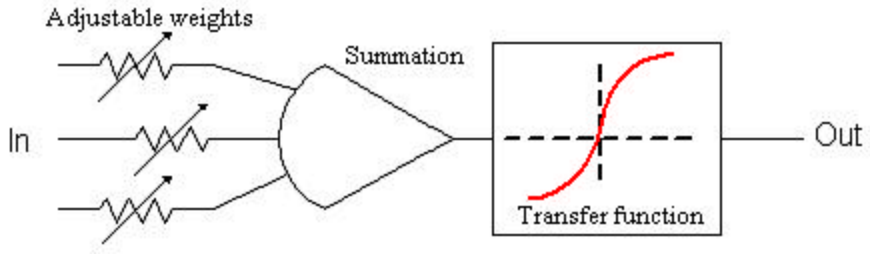

Figure 3.2 Multiple-input neuron

3.1.4 Type of Learning

"Learning" is the process of calculating the weights among neurons in a network [29]. NNs can be designed by supervised or unsupervised learning. In supervised learning, the network is presented with a set of input and desired output patterns. The resulting (actual) outputs are compared with the desired outputs and their differences are used to adjust the network weights. In unsupervised learning, the network is presented

with only input patterns. Without desired responses, the network has no knowledge about whether or not its resulting outputs are correct. As a result, the network has to self-organize (cluster) the data into similar classes by adjusting its weights so that the clustering improves. This type of learning is commonly used for pattern recognition problems and clustering. Kohonen's self-organizing network is based on unsupervised learning.

Every NN goes through three operative phases:

1) learning (training) phase - network learns on the training sample, the weights are being adjusted in order to minimize the objective function (for example the RMS or root mean square error),
2) testing phase - network is tested on the testing sample while the weights are fixed,
3) operative (recall) phase - NN is applied to the new cases with unknown results (weights are also fixed).

Learning Rules

A learning rule represents the formula that is used in NN to adjust the connection weights among neurons. Among various learning rules developed so far, four of them are most commonly used: Delta rule, Generalized Delta rule, Delta-Bar-Delta and Extended Delta-Bar-Delta rules, and Kohonen's rule.

1) Delta rule

Delta rule is also well known as Widrow/Hoff's rule [29], or the rule of least mean squares, because it aims to minimize the objective function by determining the weight

values. The aim is to minimize the sum of square error, where error is defined as the difference between the computed and the desired output of a neuron, for the given input data. The Delta rule equation is:

$$\Delta w_{ji} = h \cdot y_{cj} \cdot e_i , \qquad (3.4)$$

where Δw_{ji} is the adjustment of the connection weight from neuron j to neuron i computed by:

$$\Delta w_{ji} = w_{ji}^{new} - w_{ji}^{old} , \qquad (3.5)$$

y_{cj} is the output value computed in the neuron j; e_i is the raw error computed by:

$$e_i = y_{ci} - y_{di} , \qquad (3.6)$$

h is the learning coefficient, and y_{di} is the desired (actual) output that is used to compute the error.

The raw error in formula (3.6) is very rarely backpropagated; more often other error forms are used. In a classical Backpropagation NN, the error is backpropagated through the network using the gradient descent algorithm described in section 3.2.1. The gradient component of the global error E backpropagated into a connection k is:

$$d_k = \frac{\partial E_k}{\partial w_k} , \qquad (3.7)$$

which enables localization in a sense that each particular connection in the network is adjusted. Since Delta rule (or its variations) is commonly used in supervised networks, it is necessary to mention the main problem that can occur in backpropagating the error, i.e., the local minima. The local minima problem occurs when the minimum error of the function is found only for the local area and learning is stopped without reaching the

global minimum. Since the problem is mainly apparent in the Backpropagation algorithm, it will be discussed in detail later in the text together with suggested solutions.

2) Generalized Delta rule

Generalized delta rule is obtained by adding a derivation of input neurons into the Delta rule equation such that weight adjustment is computed according to the formula:

$$\Delta w_{ji} = h \cdot y_{cj} \cdot e_i \cdot f'(I_i), \qquad (3.8)$$

where $f'(I_i)$ is the derivative of the input I_i into neuron i. This rule is appropriate to be used with non-linear transfer functions.

3) Delta-Bar-Delta and Extended Delta-Bar-Delta rules

As can be seen from the previous section, the learning coefficient is an important parameter for the speed and efficiency of NN learning, and is typically determined as a single learning rate for all connections in the network. The Delta-Bar-Delta (DBD) learning rule was developed in 1988 by Jacobs [30] in order to improve the convergence speed of the classical Delta rule. It is a heuristic approach of localizing the learning coefficient in a way that each connection in the network has its own learning rate. Those rates change continuously as the learning progresses. Dynamic weight adjustment in the DBD rule is done according to the Saridis heuristic approach. The learning rate of a connection in the network is increased if the sign of the weight for that connection is the same for a number of time steps (or over the region of relatively low curvature). On the other hand, when the sign of the weight is changed for a certain number of time steps, the rate for that connection is decreased. Thus, Delta rule equation (3.4) is modified so that the learning rate is different for each connection *k*:

$$\Delta w_{ji(k)} = \eta_k \cdot y_{cj} \cdot e_i . \qquad (3.9)$$

Weight increments are conducted linearly, while decrements are conducted geometrically. Despite its advantages over the classical Delta rule, Delta-Bar-Delta has some limitations, such as lack of a momentum term in the learning equation and large "jumps" that can skip important regions of the error surface due to the linear increments of the learning rates. This cannot be prevented by slow geometrical decrements.

In order to overcome these shortcomings, Extended-Delta-Bar-Delta rule (EDBD), proposed by Minai and Williams [30] introduces a momentum term α_k, which also varies with time. The momentum term is used to prevent the network weights from saturation (see details in section 3.2.1), and the EDBD rule enables local dynamic adjustment of this parameter, such that the learning equation becomes:

$$\Delta w^t_{ji(k)} = \eta_k \cdot y_{cj} \cdot e_i + \alpha_k \Delta w^{t-1}_{ji(k)} , \qquad (3.10)$$

where α_k is the momentum of the connection k in the network and t is the time point in which the weights of the connection k are adjusted. Both the learning rates and the momentum term are adjusted exponentially, not linearly or geometrically as in DBD. The magnitudes of the exponential functions are the weighted gradient components δ_k (equation 3.7), which makes a larger increase in the areas of a small error curvature, and a smaller one in the areas of large curvature, thereby preventing the big "jumps" present in the DBD rule.

The above learning rules use the desired (real) output to compute the error, thus they learn supervised. If the desired output is not known, one of the unsupervised learning rules should be used, such as the following Kohonen's rule.

4) Kohonen's rule

Since Kohonen's network does not learn on known outputs, the weights are adjusted using the input into the neuron i:

$$\Delta w_{ji} = h \cdot extinput_i - w_{ji}, \qquad (3.11)$$

where $extinput_i$ is the input that neuron i receives from the external environment. Kohonen's rule is used in Kohonen's self-organizing network. Details concerning learning equations are given in section 3.3.2.

3.1.5 Other Parameters for NN Architecture Design

According to the number of layers, NN architectures can be one-layered (with the output layer only) or multi-layered (with one or more hidden layers additionally). The number of necessary hidden layers should be experimentally determined. It is to be expected that more hidden layers should be used for approximating a very complex non-linear function, although it is proven that two-layered NNs can approximate any non-linear function as mentioned earlier.

NNs can be divided into:

a) deterministic networks - when a neuron reaches a certain activation level, it sends impulses to other neurons (it "fires"),

b) stochastic networks - firing is not certain and it is performed according to probabilistic distribution (for example, the Boltzman machine).

Architecture	Characteristics			
	Type of connection		Learning	
	Inter-layers	Intra-layers	Type	Equation
Two-layered				
Perceptron	fully connected	-	supervised	$\Delta w_{ji} = h(y_{di} - y_{ci})$
ADALINE/ MADALINE	fully connected	-	unsupervised	$\frac{1}{n}\sum_{i=1}^{n}(y_{di} - y_{ci})^2$
Kohonen's	fully connected	on-center/ off-surround	unsupervised	$\Delta w_{ji} = h \cdot extinput_i - w_{ji}$
Hopefield's	fully connected	Recurrent cross-bar	unsupervised	$E = -\frac{1}{2}\sum_{j \neq i}^{n}\sum_{i=1}^{n} w_{ji} y_j y_i$
Multi-layered				
Backpropagation	hierarchical	recurrent	supervised	$\Delta w_{ik} = h \cdot y_i \cdot e_k$
Recurrent Backpropagation	fully connected	recurrent cross-bar	unsupervised	$\Delta w_{ji} = 2(y_{di}^t - y_{ci}^t) r_{ji}^t,$
Radial-Basis	fully connected	on-center/ off-surround	Unsupervised phase + supervised	$\Delta w_{ik} = h \cdot y_i \cdot e_k$ $r_{ji}^t = f'(input_i^t)(y_{cj}^{t-1} + \sum w_{ki}) r_{ki}^{t-1}$
Probabilistic	fully connected, non-hierarchical	on-center/ off-surround	Unsupervised phase + supervised	$\Delta w_{ik} = h \cdot y_i \cdot e_k$
Learning Vector Quantization	fully connected	on-center/ off-surround	Unsupervised phase + supervised	$\Delta w_{ik} = h \cdot y_i \cdot e_k$
Counter-propagation	fully connected, non-hierarchical	recurrent cross-bar	supervised	$\Delta w_{ji} = h \cdot extinput_i - w_{ji}$ between 1st and 2nd layer
ART networks	Resonance fully connected	on-center/ off-surround	unsupervised	

Table 1. Neural Network Architectures

NNs can also be classified as

a) static networks (receive inputs in one pass),

b) dynamic networks (receive inputs in time intervals, they are also called spatio-temporal networks).

NN learning can be:

a) batch learning - network learns only in the learning phase, in other phases weights are fixed,

b) on-line learning - network also adjusts its weights in the recall phase.

Table 1 [31] above shows a brief overview of well-known NN architectures according to the above parameters for architecture design. Further text presents detailed description of various NN architectures.

3.2 Backpropagation Network

Back-propagation (BP) [15], [19], [37] is a multi-layer neural network using sigmoidal activation functions. Originally developed by Paul Werbosin in 1974, extended by Rumelhart, Hinton, and Williams in 1986, this was the first network with more than one hidden layer. Its role was primarily to solve the "credit assignment" problem imposed by the Perceptron network, which is the problem of assigning the adjustments of parameters or connection weights. The suggested solution was to localize the error by computing it at the output layer and backpropagating the error to each hidden layer such that weights of connections are adjusted until the input layer is reached.

The classical Backpropagation algorithm involves error optimization using a deterministic gradient descent algorithm, which will be described in detail. However, recent research includes some other deterministic (second order methods) [32], [33] for error optimization, such as conjugate gradient and the Levenberg-Marqardt algorithm that tries to overcome the main disadvantage of the steepest descent method, i.e., the danger

of local minima. In this thesis implementation, second order methods will not be used, but some parameter adjustments to avoid the main shortcomings of the classical Backpropagation algorithm will be implemented.

Architecture of the network

The network is made up of an input layer, at least one hidden layer, and an output layer. Nodes in each layer are fully connected to those in the layers above and below. Each connection is associated with a synaptic weight. Typical backpropagation architecture is presented in Figure 3.3 (for clarity reasons only 1 hidden layer is shown):

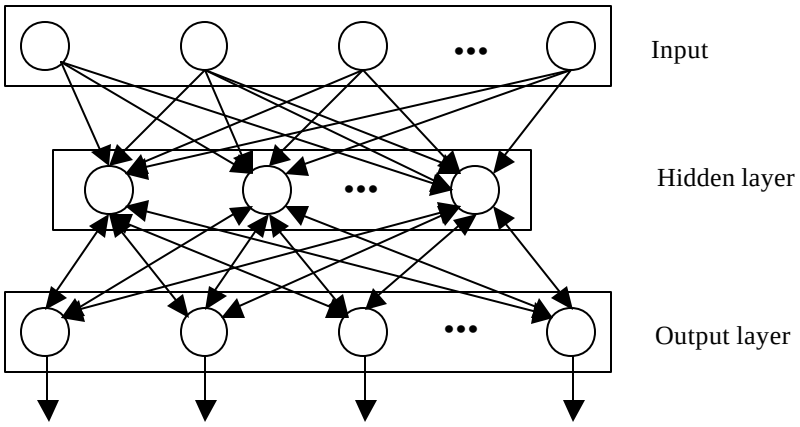

Figure 3.3: Architecture of Backpropagation Neural Network

Data flow through the network can be briefly described in few steps:

1) from the input to the hidden layer: the input layer loads data from input vector X, and sends them to the first hidden layer,

2) in the hidden layer: units in the hidden layer receive the weighted input and transfer it to the next hidden or to the output layer using one of the transfer functions,

3) as information propagates through the network, all the summed inputs and output states are computed in each processing unit,

4) in the output layer: for each processing unit, the scaled local error is computed and used to determine the weight increment or decrement,

5) Backpropagation from the output back to the hidden layers: the scaled local error and weight increments or decrements are computed for each layer backwards, starting from the output layer and ending at the first hidden layer, and the weights are updated.

Computation in the network

When the input layer sends data to the first hidden layer, each hidden unit in the hidden layer receives weighted input from the input layer (initial weights are set randomly) according to the formula [15]:

$$I_j^{[s]} = \sum_i w_{ji}^{[s]} \cdot x_i^{[s-1]} , \qquad (3.11)$$

where $I_j^{[s]}$ is the input to neuron j in layer s, $w_{ji}^{[s]}$ is the connection weight from neuron j to neuron i in layer s, and $x_i^{[s-1]}$ is the output of the neuron i in layer $s-1$. Units in the hidden layer transfer those inputs according to the formula:

$$x_j^{[s]} = f\left(\sum_i w_{ji}^{[s]} \cdot x_i^{[s-1]}\right) = f\left(I_j^s\right), \qquad (3.12)$$

where $x_j^{[s]}$ is the output of the neuron j in layer s, and f is the transfer function (sigmoid, hyperbolic tangent, or any other function). If there is more than one hidden layer, the above transfer function is used through all hidden layers until the output layer is

reached. At the output layer, the network output is compared to the desired (real) output, and the global error E is determined as:

$$E = \frac{1}{2}\sum_k (d_k - x_k)^2, \qquad (3.13)$$

where d_k is the desired (real) output, x_k is the output of the network, and k is the index for the component of the output, i.e., the number of output units. Each output unit has its own local error e whose raw form is $(d_k - x_k)$, but what is backpropagated through the networks is the scaled error in the form of a gradient component:

$$e_k^{(x)} = -\partial E / \partial I_k^{(x)} = -\partial E / \partial x_k \cdot \partial x_k / \partial I_k = (d_k - x_k) \cdot f'(I_k) \quad . \qquad (3.14)$$

The objective of the Backpropagation learning process is to minimize the above global error by backpropagating it into the connections through the networks backwards until the input layer is reached. By modifying the weights, each connection in the network is corrected in order to achieve a smaller global error. The process of incrementing or decrementing the weights (learning) is done by using a gradient descent rule:

$$\Delta w_{ji}^{[s]} = -h \cdot (\partial E / \partial w_{ji}^{[s]}), \qquad (3.15)$$

where h is the learning coefficient. To compute partial derivations in the above equation we can use (3.14), which gives:

$$\partial E / \partial w_{ji}^{[s]} = (\partial E / \partial I_j^{[s]}) \cdot (\partial I_j^{[s]} / \partial w_{ji}^{[s]}) = -e_j^{[s]} \cdot x_i^{[s-1]}. \quad (3.16)$$

When the above result is included in formula (3.15), the weight adjustment is

$$\Delta w_{ji}^{[s]} = h \cdot e_j^{[s]} \cdot x_i^{[s-1]}, \qquad (3.17)$$

which leads to the main problem of setting the appropriate learning rate.

There are two mutually conflicting guidelines for determining h. The first guideline is to keep h low because it determines the area in which the error surface is locally linear. If the network aims to predict high curvatures, that area should be very small. However, a very low learning coefficient means very slow learning. In order to resolve this conflict, the previous delta weights in time $(t-1)$ are added in equation (3.17), so that the current weight adjustment is:

$$\Delta w_{ji}^{[s]} = h \cdot e_j^{[s]} \cdot x_i^{[s-1]} + a \cdot \Delta w_{ji}^{t-1[s]}, \qquad (3.18)$$

where a is the momentum term which makes learning faster when the learning coefficient is low. Learning can be accelerated also if the weights are not adjusted for each training vector but cumulatively, where the number of training vectors after which the weights are adjusted is called the epoch. An epoch that is not very large can improve the convergence speed, but a large epoch can make the computation of the error more complex and therefore decreases its benefit. Another problem that can occur in Backpropagation is that some processing units will stop to learn if their incoming weights become large. In such case the summation values become large and the weights are saturated (value 0 or 1) leading the derivation to zero and the scaled error to zero. Such saturation can be prevented by adding a small bias value (or F' offset) to the derivative of the sigmoid transfer function.

Improvements of Standard Backpropagation Network Learning Rules

Since one of the main disadvantages of Backpropagation is its slow learning, much effort has been expanded in improving the learning rules and other parameters. Some of the achievements are [19], [38]:

- Delta-Bar-Delta (DBD) rule - a learning rule that uses past values of the gradient to find the local curvature of the error and allocates a different learning coefficient to each connection in the network,
- Extended Delta-Bar-Delta (EDBD) rule - besides using a different learning rate for each connection, it uses a different momentum term for each connection (equations are described in section 3.1.4),
- QuickProp and MaxProp - learning rules that use quadratic estimation heuristics to determine the direction and step size for the weight changes,
- Resilient Backpropagation (Rprop) - A local adaptive learning scheme that eliminates the harmful effect of having a small slope at the extreme ends of the sigmoid "squashing" transfer functions.

Because of its advantages in dynamic and local adjustments of learning rates to the topology of the error function, the Rprop is used in the implementation, and will be further discussed below. Backpropagation neural network allows the usage of a different error function, such as quadratic and cubic, but they will not be discussed here since they are not included in the implementation.

Resilient Backpropagation (Rprop)

Multilayer networks typically use sigmoid transfer functions in the hidden layers. Sigmoid functions are characterized by the fact that their slope must approach zero as the input gets large. This causes a problem when using steepest descent to train a multilayer network with sigmoid functions, since the gradient can have a very small magnitude; and therefore, cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the Rprop [38] training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative is used to determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor delt_inc whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor delt_dec whenever the derivative with respect that weight changes sign from the previous iteration. If the derivative is zero, then the update value remains the same. Whenever the weights are oscillating the weight change will be reduced. If the weight continues to change in the same direction for several iterations, then the magnitude of the weight change will be increased.

Dealing with Local Minima and Overtraining

Two of probably the most famous problems with Backpropagation are local minima and overtraining. Because of the way that error is backpropagated through the

network (gradient descent optimization), learning can stick in a local minimum and minimize the error only locally. There are a number of solutions for this problem. Some of them are deterministic and use second order equations to compute the error, while others are stochastic, and rely on random numbers rather than on equations. One of the stochastic methods for avoiding local minima is simulated annealing.

Overtraining is the universal problem for all types of NN algorithms. It occurs when the network learns the training sample perfectly, but is not able to generalize on the test sample. One of the main still unanswered questions on how long it takes to learn can be approached in the following ways [33]:

- cross validation using the validation sample to determine when to stop learning. The training will continue as long as the error on the validation sample improves. When it does not improve, the training will stop. Such iterative procedure is usually called the "Save best" procedure, which alternatively trains and tests the network until the performance of the network does not improve for n number of iterations. After the best network is selected, it is tested on a new test sample to determine its generalization ability (since this method is used in our experiments, it is described in detail in section),
- adding bias and random error in parameter estimates,
- jackknifing,
- bootstrapping, and others.

Input Parameters to Build the Network

1) *number of input, hidden, and output layer units*

The number of hidden units can be statically set to a fixed number or dynamically optimized during the learning phase of the NN. In this implementation, one node is used and another node added if the goal is not met (this is discussed in detail in section 4).

2) *learning coefficients*

Learning coefficients can be set:

- statically and globally for the whole network in a way that coefficients do not change during the learning process,
- statically and locally by setting a different learning rate for each hidden layer or connection,
- dynamically and globally by changing the global learning rate while the learning process improves,
- dynamically and locally by assigning a different learning rate to each connection in the network and changing them during the learning process.

3) *Bias* (*F'Offset)*

As explained in the previous section, this parameter prevents the network from saturating the weights.

4) *learning rule*

This is a procedure for modifying the weights and biases of the network.

5) *transfer function*

The choice of a transfer function is made according to its ability to map into positive as well as negative regions. This transfer function is often used in image compression.

6) *bipolar inputs*

Input values are scaled between -1 and 1. Because positive and negative values in the input variables are desired, this option is used in the thesis implementation.

7) *MinMax table*

Inputs to the network are preprocessed using the so-called Minmax table created from the training data. Such a table consists of minimum $m_i (i = 1,...,n)$ and maximum $M_i (i = 1,...,n)$ values for each of the n variables in the network, where n is the sum of the number of input variables I and the number of desired output variables D. Those values together with the network range parameters (specified in the I/O set of parameters) are used to scale each input and output variable according to the formula:

$$s_i = \frac{(R_i - r_i) \cdot x_i + (M_i \cdot r_i - m_i \cdot R_i)}{(M_i - m_i)}, \qquad (3.19)$$

where s_i is the scaled new value for the variable i, R_i is the upper limit of the network range for inputs (or outputs), and r_i is the lower limit of the network range for inputs (or outputs). Such a scaling process is necessary because of the output range of transfer functions used in the networks. For example, the hyperbolic tangent function has the output range of [-1,1], and therefore the inputs to and outputs of the network need to be mapped into the same range. Upon completion of the learning process, output values of the network are rescaled, so that original real values are presented to the user.

8) *epoch*

An epoch is a presentation of a set of training (input and/or target) vectors to a network and the calculation of new weights and biases. Training vectors can be presented one at a time or all together in a batch.

3.3. Radial-Basis Function Network

A Radial-Basis function network (RBFN), proposed by M.J.D. Powel [34], is a general-purpose network which can be used in the same situations as a Backpropagation network for prediction as well as for classification problems. Since it uses a radially symmetric and radially bounded transfer functions in its hidden layer, it is a general form of probabilistic and general regression networks. It overcomes some disadvantages of Backpropagation such as slow training time and the local minima problem, but requires more computation in the recall phase in order to perform function approximation or classification.

Computation in the Network

Any network using radially symmetric hidden units belongs to the class of Radial-Basis Function networks. A pattern of hidden units is radially symmetric [30], if it: (a) has a "center", i.e. an input vector stored in the weight vector between the input and the hidden layer, (b) has a distance measure which determines the distance of each input vector from the center, (c) has a transfer function which maps the output of the distance function.

Such a general definition also includes General Regression networks, Probabilistic, Counter-propagation, and other similar networks. The most common distance measure used is the Euclidean distance, while a Gaussian is the usual transfer function (or kernel) in the hidden layer. The output of this hidden layer is the same for all inputs within a fixed radial distance from the center, i.e., for the inputs that are radially symmetric. The performance of RBFN "depends on the number and position of the radial-basis functions,

their shape, and the method used for determining the associative weight matrix W [35]. Some existing strategies for training RBFNs can be classified as follows:

1) RBFNs with a fixed number of centers selected randomly from the training data,
2) RBFNs with unsupervised procedures for selecting a fixed number of Radial-Basis Function centers,
3) RBFNs with supervised procedures for selecting a fixed number of Radial-Basis Function centers.

The above strategies all have the same disadvantage: the number of centers must be determined in advance. To overcome this shortcoming, several authors suggested algorithms, such as the growing cell structure (GCS) proposed by Fritzke, distribution of radial-basis functions with space-filling curves proposed by Whitehead and Choate, dynamic decay adjustment (DDA) algorithm proposed by Berthold and Diamond, and merging two prototypes at each adaptation cycle. All the above algorithms involve either cascade or pruning principles [39].

The focus below will be on the RBFN algorithm proposed by Moody and Darken [30], which uses Euclidean distance and a Gaussian transfer function in the hidden layer. The input to the hidden units is computed according to the formula [37]:

$$I_k = \| X - c_k \| = \sqrt{\sum_{i=1}^{N} (X_i - c_{ki})^2}, \qquad (3.20)$$

where c is the center. The output is computed using a Gaussian transfer function:

$$f(x) = j(\|x - c\|) = e^{\left(\frac{I_k^2}{s_k^2}\right)}, \qquad (3.21)$$

where the center c is determined by a clustering algorithm and by the nearest neighbor technique.

Architecture of the Network

The RBF learning algorithm can be briefly described as follows:

- training starts in the hidden layer with an unsupervised learning algorithm in order to determine the center,
- training continues in the output layer with a supervised learning algorithm in order to compute the error,
- simultaneous application of a supervised learning algorithm to the hidden and output layers to fine-tune the network.

A common RBFN architecture is shown in the figure below.

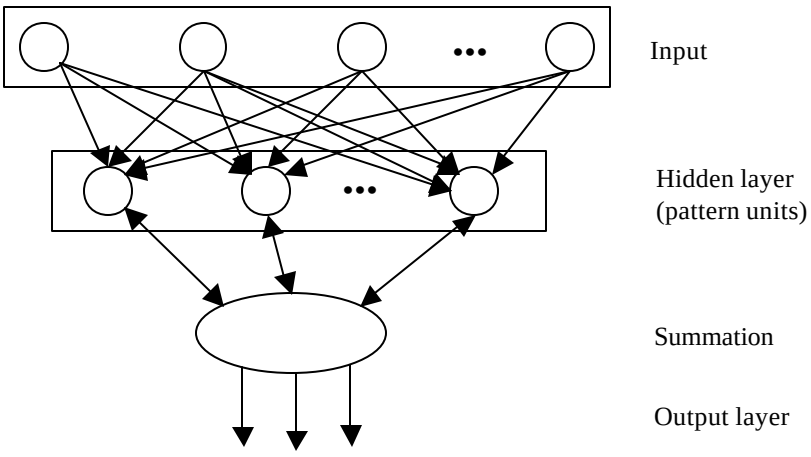

Figure 3.4: Architecture of RBFN

Learning through the architecture can be described in the following steps:

1) from the input to the hidden layer: Clustering phase. In this phase the incoming weights to the prototype layer learn to become the centers of clusters of input vectors using a dynamic algorithm.

2) in the hidden layer: The radii of the Gaussian functions at the cluster centers are computed using a 2-nearest neighbor technique. The radius of a given Gaussian is set to the average distance to the two nearest cluster centers.

3) in the output layer: Error is computed at the output layer using one of the learning rules. It is also possible to include one additional hidden layer to improve learning.

Application of the Network

Karayiannis and Weigun [35] give a brief overview of the previous usage of a Radial-Basis network that starts with Broomhead and Lowe year who first implemented this network and showed how it models nonlinear relationships. The ability of a RBFN with one hidden layer to approximate any nonlinear function is proved by Park and Sandberg. Then Michelli showed how this network could produce an interpolating surface, which passes through all the pairs of the training set.

Advantages of a RBFN can be briefly summarized as follows:

- fast training,
- better decision boundaries than Backpropagation when used for classification and decision problems,

- hidden unit can be interpreted as a density function for the input vectors and thus measures the probability that a new vector is a member of the same distribution as others in the input space.

Disadvantages:

- despite fast learning, it can be slower than Backpropagation in the recall phase,
- since the initial learning phase of a Radial-Basis Function network is the unsupervised clustering phase, some discriminatory information could be lost in this phase,
- it is difficult to determine the optimal number of prototype units [35]. The authors who propose several ways to overcome this disadvantage: a Growing Radial-Basis (GRBF) network that starts with a small number of prototypes at each growing cycle and grows in the training process by splitting of the prototypes in each cycle. They also suggest two criteria to determine which prototype to split, and test different hybrid learning schemes for incorporating existing learning schemes into RBFN, such as unsupervised learning for clustering, learning vector quantization, and linear neural networks, with very satisfactory results. The authors also propose a supervised learning scheme based on minimization of the localized class-conditional variance.

Input Parameters to Build the Network

The RBFN uses the same input parameters as the Backpropagation network

3.4 General Regression Network

According to Specht [40], the General Regression Neural Network (GRNN) is a generalized form of the Probabilistic network, which is primarily designed for

classification problems. GRNN can be used for system modeling and prediction, with the special ability to deal with sparse and nonstationary data. Its disadvantages, such as memory intensiveness and time-intensiveness in the recall phase, are not limiting factors for today's fast computers.

Computation in the Network

GRNN is designed to perform a nonlinear regression analysis. If $f(x,z)$ is the probability density function of the vector random variable x (input vector) and its scalar random variable z (measurement), then the computation in GRNN consists of calculating the conditional mean $E(z \mid x)$ of the output vector, given by [37]:

$$E(z \mid x) = \frac{\int_{-\infty}^{\infty} z f(x,z)\,dz}{\int_{-\infty}^{\infty} f(x,z)\,dz}. \qquad (3.22)$$

The joint probability density function (pdf) $f(x,z)$ is required to compute the above conditional mean. GRNN approximates the pdf function from the training vectors using Parzen window estimation, a nonparametric technique that approximates a density function by constructing it out of many simple parametric pdfs [40]. Parzen windows are Gaussian with a constant diagonal covariance matrix:

$$\hat{f}_p(x \mid z) = \frac{1}{(2p\,s^2)^{(N+1)/2}} \cdot \frac{1}{P} \sum_{i=1}^{P} \left(e^{\frac{-D_i^2}{2s^2}} \cdot e^{\frac{-(z-x_i)^2}{2s^2}} \right), \qquad (3.23)$$

where P is the number of sample points x_i, N is the dimension of the vector of sample points x_i, s is a smoothing constant, and D_i is the Euclidean distance between x and x_i computed by:

$$D_i = \|x - x_i\| = \sqrt{\sum_{i=1} (x - x_i)^2}, \tag{3.24}$$

where N is the number of input units to the network, s is the width parameter which satisfies the following asymptotic behavior as the number of Parzen windows P becomes large:

$$P \xrightarrow{\lim} \infty (P s^N P) = \infty \text{ and} \tag{3.25}$$

$$P \xrightarrow{\lim} \infty (P s^N P) = 0 \text{ or when } s = \frac{S}{P^{(E/N)}}, 0 \le E < 1, \tag{3.26}$$

where S is the scale and, N is the number of input units. When estimated pdfs are inserted into equation (3.22), the following formula for computing each component z_j is obtained

$$z_j(x) = \frac{\sum_{i=1}^{P} z_j^i e^{-\frac{D_i^2}{2s^2}}}{\sum_{i=1}^{P} e^{-\frac{D_i^2}{2s^2}}}. \tag{3.27}$$

Since computation of Parzen the estimation is time consuming when the sample is large, a clustering procedure is often incorporated in GRNN. According to this procedure, for any given sample x_i, instead of computing a new Gaussian kernel $e^{-\frac{D_i^2}{2s^2}}$ at center x for each point, the distance of that sample to the closest center of a previously established

kernel is found, and the old closest kernel is reused. Such an approach transforms the equation (3.27) for z_j, into:

$$\hat{z}_j = \frac{\sum_{i=1}^{P} A_i e^{\left(\frac{-D_i^2}{2s^2}\right)}}{\sum_{i=1}^{P} B_i e^{\left(\frac{-D_i^2}{2s^2}\right)}}, j = 1,...,M, \qquad (3.28)$$

where

$$A_i \equiv A_i(k) = A_i(k-1) + z_j \text{ and } B_i \equiv B_i(k) = B_i(k-1) + 1. \quad (3.29)$$

Architecture of the Network

The network consists of the input layer, the pattern layer and the output layer (see Figure 3.5). There is also an additional summation/division layer whose function will be explained later. The process of network learning is conducted as follows:

1) *from the input layer to the pattern layer*: training vector X is distributed from the input layer to the pattern layer, and the connection weights from the input layer to the k^{th} unit in the pattern layer store the center X_i of the k^{th} Gaussian kernel.

2) *in the pattern layer*: The summation function for the k^{th} pattern unit computes the Euclidean distance D_k between the input vector and the stored center X_i and transforms

it through the exponential function $e^{-\frac{D_i^2}{2s^2}}$. Then B coefficients are set as connection weights from the pattern layer to the first unit in the summation/division layer, and A coefficients are set as the weights to the remaining units in the summation/division layer.

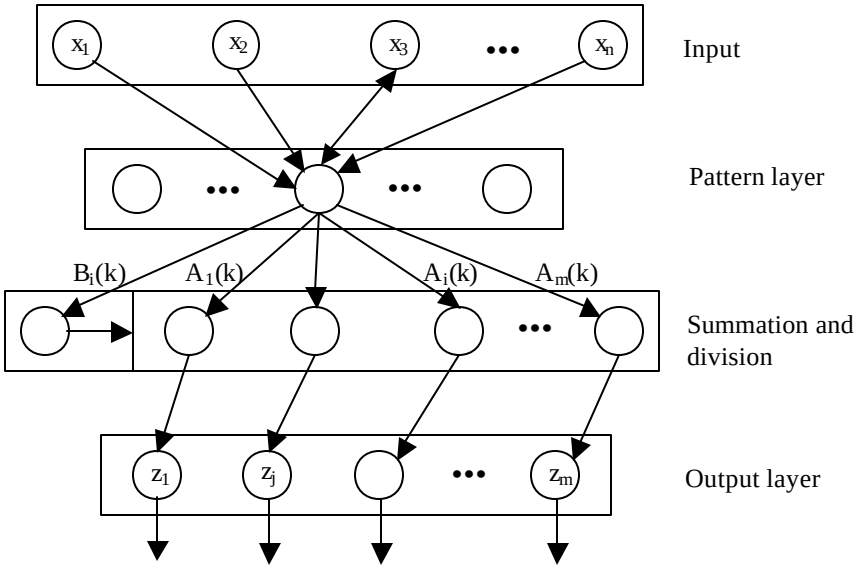

Figure 3.5: Architecture of GRNN

3) *in the summation/division layer*: the summation function of this layer (which is the standard weighted sum function) computes the denominator of equation (3.27) for the first unit (j), then the numerator for each next unit $(j+1)$. To compute the output $\hat{z}_j(x)$, the summation of the numerator is divided by the summation of denominator and such output is forwarded to the output layer (note that the first unit of the summation layer does not generate the output).

4) *in the output layer*: output layer receives inputs from the summation/division layer, outputs the estimated conditional means, and computes the error on the basis of the real output from the environment.

Application of the Network

Because of its generality, GRNN can be used in various problems such as prediction, plant process modeling and control, general mapping problems, or for other problems where nonlinear relationships exist among inputs and output [37]. One of the main advantages of GRNN is the ability to deal with nonstationary data (time series data whose statistical properties change over time). This ability is obtained by modifying the computation of the B and A coefficients in equation (3.27) such that a time constant is introduced in terms of number of training vectors, as an indication of how fast the time series changes its characteristics.

It can be concluded from the above description of GRNN that it is especially adaptable to (a) nonstationary and sparse data and (b) stationary but noisy data.

Input Parameters to Build the Network

1) number of input, pattern and output layer units.

2) summation function in the pattern unit (Euclidean, City Block, or Projection).

3) t - time constant

t is defined in terms of training vectors, and should be adjusted according to the degree of nonstationarity present in the data. A smaller time constant will cause the network to forget the previous cases faster.

4) q - reset factor

This factor is divided by the number of pattern units, and used as the comparing value for resetting the B coefficients.

5) radius of influence

This is a clustering mechanism for determining the limit of the Euclidean distance by which an input vector will be assigned to a cluster. The input vector will be assigned to a cluster if the cluster center is the nearest center to the input vector, or if the cluster center is closer than the radius of influence. If the input vector does not satisfy the above conditions, a new center is computed for the vector.

6) sigma scale (S) and sigma exponent (E)

S and E values are used in computing the Parzen window width in formula (3.26).

3.5. Modular Network

Proposed by Jacobs, Jordan, Nowlan and Hinton (1991), this network is a system of many separate networks (usually Backpropagation). Each of them learns to handle a subset of the complete set of training cases. It is therefore able to improve the performance of Backpropagation when the training set can be naturally divided into subsets that correspond to distinct subtasks.

Computation in the Network

This network consists of several networks called "local experts" connected by a gating network that allocates each case to one of the local experts. The output of the local expert is compared to the actual output, and the weights are changed locally only for that expert and for the gating network. In that way the gating network "encourages" a particular local expert to specialize in similar cases. Other experts specialize in other cases. Decisions of the gating network are made stochastically. While a previous paper

suggested that the final output of the whole system is a linear combination of the outputs of the local experts, Jacobs et al. [41] use a stochastic selector and compute the error according to the formula:

$$E^c = \left\langle \left\| d^c - o_i^c \right\| \right\rangle = \sum_i p_i^c \left\| d^c - o_i^c \right\|^2, \qquad (3.30)$$

where o_i^c is the output vector of expert i in case c, p_i^c is the proportional contribution of expert i on the combined output vector, and d^c is the desired output vector in case c. In such a process each local expert produces the whole output, and the goal of one local expert is not directly affected by the weights of the other local experts. Although some indirect coupling can occur if the gating network alters the responsibilities from one local expert to another, still the sign of the local expert error remains uninfluenced. The number of local experts in the network is determined in advance, based on the assumption of the number of subsets or local regions in the input space of the sample. Each local expert is a feedforward network and all experts have the same number of input and output units. Local experts as well as the gating network receive the same input. Of course, their output differs. Output of the gating network is the probability [41]:

$$p_j = \frac{e^{(x_j)}}{\sum_i e^{(x_i)}}, \qquad (3.31)$$

where x_j is the total weighted input received by output unit j of the gating network, and p_j is the probability that the switch will select the output from local expert j. This output is normalized to sum to 1. The output of the local experts y_i is then corrected by the probability (3.31), and the final output of the network is

$$y = \sum_{i=1}^{N} y_i \cdot p_i . \qquad (3.32)$$

Unlike Backpropagation where the objective function is to minimize a global error function E, a Modular network tries to maximize the following objective function J:

$$J = \ln\left(\sum_{i=1}^{N} p_i e^{-\frac{(d-y_i)^T (d-y_i)}{2}} \right). \qquad (3.33)$$

The error that is backpropagated for the k^{th} local expert is $\frac{\partial J}{\partial I_k}$ and for the gating network $\frac{\partial J}{\partial G_k}$, where I_k is the input to the k^{th} local expert output node and G_k is the input to the gating network output node. According to the above learning process, if an expert gives a smaller less error than the weighted average of the errors of all the experts, its responsibility for that case will be increased, and vice versa. The error is backpropagated and the weights are updated according to the chosen learning rule.

Architecture of the Network

The figure below represents the architecture of the Modular neural network. For clarity reasons, the architecture in figure 3.6 consists of two local experts marked as LE1and LE2. Each local expert has only one output neuron. The gating network and local experts have the same number of input neurons, but the number of output neurons in the gating network is the number of local experts, i.e., two in our example presented in the figure.

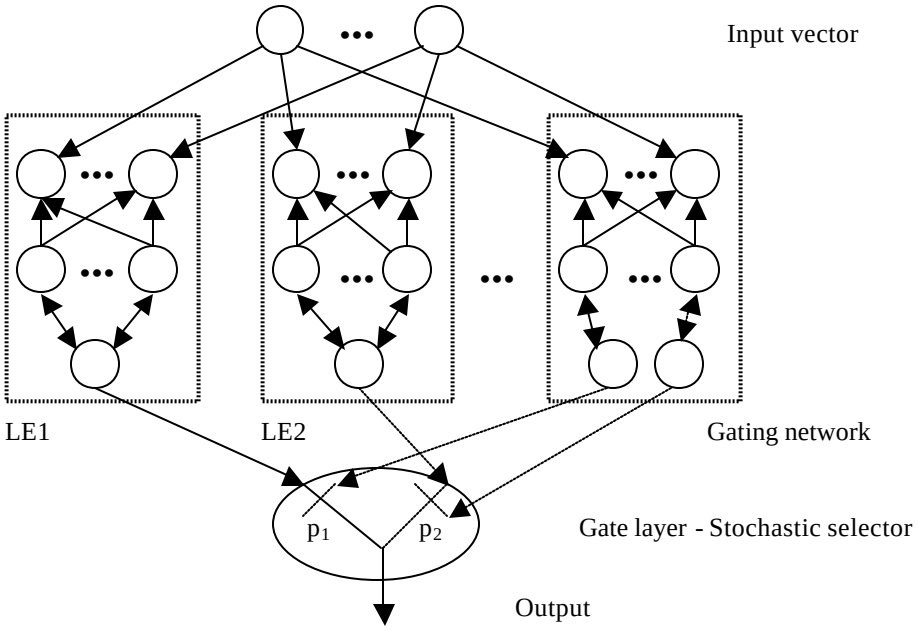

Figure 3.6: Architecture of Modular Neural Network

Learning is conducted as follows:

1) *from the input layer to the local experts and to the gating network*: The same training vector X is distributed from the input layer to each local expert and to the gating network. Each expert is a feedforward (usually Backpropagation) network. Output of the local experts depends on the feedforward architecture incorporated in the expert. Output of the gating network is computed as described above.

2) *in the gate layer*: The gating network sends output to an intermediate layer (called a gate) where the probabilities sent by the gating network are used to correct the local expert outputs.

3) *in the output layer*: final output to the user is the output of the local expert with the highest probability. The error is computed according to the formula 3.30 and backpropagated to the local experts and to the gating network.

Application of the Network

A Modular network can be applied in most cases where Backpropagation is used, especially in problems with different regions in the input space. One of the illustrations of such problems is the absolute value function:

$$y = \begin{cases} x \; if \;\; x \geq 0 \\ -x \; if \;\; x < 0 \end{cases}, \qquad (3.34)$$

where output y is computed by a different function for different subsets of x. Jacobs et al. [41] applied a Modular network on a speaker independent, four-class vowel discrimination problem and compared the performance of a Modular network using 4 and 8 local experts with three-layered Backpropagation networks. The comparison showed that the performance of tested networks is the same, although the Modular network reaches the error criterion significantly faster than Backpropagation. A Modular network is also a way to make competitive learning associative, in a sense that a local expert whose output vector specifies the mean of a multidimensional Gaussian distribution replaces each hidden unit in the competitive network. A Modular approach also enables the usage of more complex architectures when each local expert is designed as one Modular network.

Input Parameters to Build the Network

1) number of input, hidden and output layer units for local experts (LE)

Each local expert has the same initial structure since it is not known in advance which part of the input space will be allocated to each LE.

2) number of hidden and output units in gating network

Hidden units in the gating network can be determined heuristically or optimized in the learning phase. The number of output units in the gating network determines the number of local experts (LEs) in the network.

3) learning coefficient

4) learning rule

The EDBD learning rule is used as described in the Backpropagation section.

5) other parameters described in the Backpropagation section.

3.6 Probabilistic Network

A Probabilistic neural network (PNN), one of the stochastic-based networks, is built on a decades old statistical algorithm developed by Meister [36], although Donald Specht proposed the complete neural network algorithm in 1988. It uses nonparametric estimation methods for classification and therefore does not suffer from local minima problems, as do feedforward networks. According to Kartalopoulos [34], Probabilistic NN is able to approximate the optimum boundaries between categories; therefore it can be used as a classifier, with the assumption that the training data are a true representative sample. The Architecture of the Probabilistic neural network is built upon Bayes'

classifier using the Parzen window estimator to estimate the probability distributions of the class samples [37].

Computation in the Network

In general, the classification problem can be stated as sampling the m-component multivariate random vector $X = [x_i, ..., x_m]$, where the samples are indexed by k, $k = 1, ..., K$ [33]. The probability that a sample will be drawn from a population k is h_k, and the cost of misclassifying the sample will be c_k. The population from which the training samples are taken is known, and the aim is to create an algorithm for classifying unknown samples with the expected misclassification cost less than or equal to any other. Such algorithm is *Bayes optimal*. If the probability density functions for all populations k are known, then the Bayes optimal decision rule is: classify X into population i if

$$h_i c_i f_i(X) > h_j c_i f_j(X) \; \forall \; j, i. \qquad (3.35)$$

Since the probability density functions are usually not known in practice, it is often assumed that they are members of a normal distribution. The training set is then used to estimate the parameters of the distribution. However, it is more appropriate to use a nonparametric estimation method such as Parzen windows, also used in GRNN. In order to classify unknown samples, most common classifiers separate the unknown from each known member of the training set using the Euclidean or other distance. The unknown member is then classified into the population of its nearest neighbor. The Parzen windows technique goes one step further in a way that it takes into account more distant neighbors. Parzen's technique estimates a "sphere-of-influence" function for separating an unknown point from the known training sample point. Such a function has a higher value

if the distance is close and converges to zero if the distance becomes large. Taking the sum of this function for all known training set members and classifying the unknown point into the population with the largest sum is the main idea of the probabilistic algorithm. Parzen's estimated density function is

$$g(x) = \frac{1}{ns} \sum_{i=0}^{n-1} W\left(\frac{x - x_i}{s} \right), \qquad (3.36)$$

where n is the sample size, s is the scaling parameter that controls the width of the area of influence of the distance, and W is the weighting function. Since a Gaussian is usually used for weighting, the probability density function takes the form:

$$g(x) = \frac{1}{ns^{p}(2p)^{p/2}} \sum_{i=0}^{n-1} e^{\frac{\|x - x_i\|^2}{2s^2}}. \qquad (3.37)$$

Although the value of s is an important smoothing parameter in the Probabilistic network since it affects the estimation error, there is no mathematical way of determining it. A too small value of s gives the same effect as the nearest neighbor technique, and too large does not give clear separation of classes so classification cannot be made. A large value gives a flat curved surface, while a small value results in narrow peaks.

Architecture of the Network

The Probabilistic neural network consists of the input layer, the pattern layer, the summation layer, and the output layer. A simplified architecture is shown in the figure below.

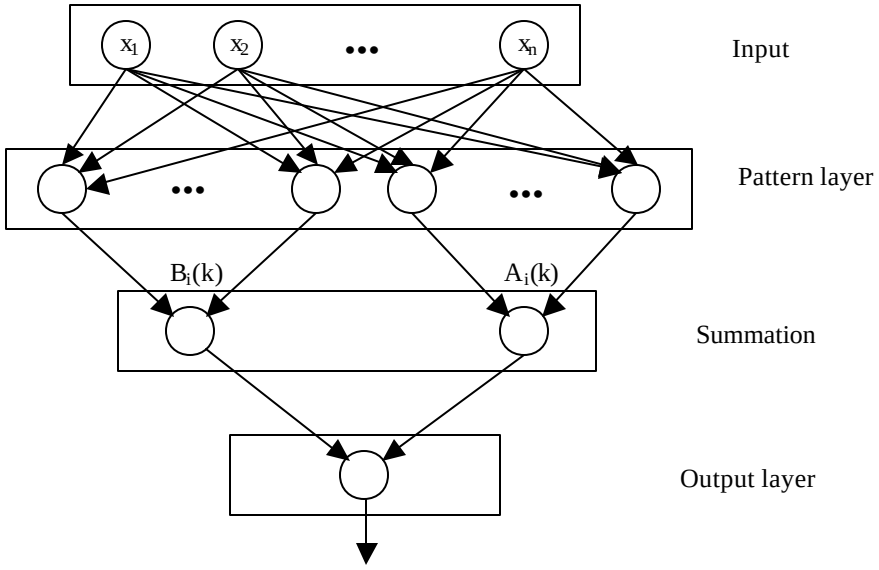

Figure 3.7: Architecture of Probabilistic Neural Network

Learning in PNN is not an iterative process. Only one pass through the training sample is needed for the network to learn. The flow of information through the layers is the following:

1) *from the input layer to the pattern layer*: Training vector X is distributed from the input layer to the pattern layer,

2) *in the pattern layer*: The pattern layer consists of K units, one for each training vector X. Input in the pattern layer unit j is computed according to the formula:

$$I_j = (x - w_i)^T (x - w_i).) \qquad (3.38)$$

The pattern layer units are not fully but selectively connected to the summation units, depending on the classes they represent. Output from the pattern unit j is performed according to the activation function:

$$f(x, w_j) = e^{\frac{\sum_{i=1}^{k} (x_i - w_{ij})^2}{2s^2}}. \qquad (3.39)$$

3) *in the summation layer*: The number of units in the summation layer is equal to the number of classes. Each summation unit receives input from the pattern unit of the same class. Output of the summation units is the estimation of the class probability density function according to formula (3.39).

4) *in the output layer*: Each unit in the output layer receives inputs from each summation unit and produces a binary output signal, which is a product of the summation unit's output and the weight coefficient.

Application of the Network

The probabilistic network is exclusively designed for classification problems. Although in some cases it can be adjusted for autoassociation, it is primarily a classifier. Successful usage of this algorithm is observed in vector cardiogram interpretation, radar/target identification, hull-to-emitter correlation on radar hits, for example. It is also a very fast training algorithm compared to feedforward algorithms, especially in the case when it is optimized in an extensive jackknifing process. Another advantage of this network is that its results can be interpreted as Bayesian posterior probabilities, thus suitable for confidence estimates. Disadvantages of Probabilistic network are in the necessity of having a representative training set and in memory requirements, since the whole training set is processed while classifying each unknown case.

Since it takes the sum of the density function, the Probabilistic network is especially suitable for problems with sparse data and data outliers, since outliers will have no strong effect on decisions.

Input Parameters to Build the Network

In order to build a PNN, the following parameters should be determined:

1) number of input, pattern, and output units

Pattern units are set to the number of training vectors according to the computation procedure described before. If the training sample is very large, it is recommended to use a smaller number of pattern units, and then to set the radius of influence to a positive value. The number of output units is set to the number of classes.

2) summation function in the pattern unit: Euclidean, City Block, or Projection

3) radius of influence

This parameter is used for a clustering procedure in the same way as in a GRNN with the aim to make learning faster by finding the closest already computed Parzen window for a new training vector.

4) output mode for the output layer

Possible output modes are probabilistic, competitive, and normalized. The probabilistic mode directly outputs the values produced by the density function. The competitive output produces 1 for the winning unit and 0 for the others. The output values in the normalized mode are all positive and normalized such that they sum to 1.

5) sigma scale (S) and sigma exponent (E)

In order to find the correct asymptotic behavior for the Parzen window width when the pattern units become very large, NeuralWare [30] implements an exponential decay of. Parameters S and E are used to compute according to the formula (3.26).

3.7 Learning Vector Quantization Network

Vector quantization (VQ) is one of the compression methods usually used when large quantities of data must be divided into a number of classes in order to increase the processing efficiency. As was previously discussed in section 2.1, it is the process of mapping input vectors x_i of dimension n into a finite number of classes, represented by a codeword or a prototype vector w_j ($j = 1,..., m$), where $m < n$. Mapping is performed using one of the nearest neighbor techniques such as Euclidean distance or a cost function [37]. VQ is a method of unsupervised learning, but has a special supervised form called Learning Vector Quantization (LVQ), originally proposed by Teuvo Kohonen. Such supervised versions of VQ will be discussed below.

Computation in the Network

Learning in LVQ a takes place in the hidden Kohonen layer. This network differs from a VQ in its usage of known (desired, true) output classifications t for each input vector x. If $C(x)$ is a class of x computed by the network and t is the true class, then the learning in LVQ is performed as follows [37]:

1) After receiving an input vector, a Kohonen layer finds the appropriate class in an unsupervised way, finding the distance d_i:

$$d_i = \|w_c - x\| = \min_i \|w_i - x\|. \qquad (3.40)$$

2) Computed class $C(x)$ is compared to the true class t and the weights are adjusted in the following way:

$$\begin{aligned} \Delta w_c &= a(x - w_c) \quad if \;\; C(x) = t \\ \Delta w_c &= -g(x - w_c) \quad if \;\; C(x) \neq t \\ \Delta w_i &= 0 \quad \forall\, i \neq c. \end{aligned} \qquad (3.41)$$

In other words, if the computed class is equal to the true class, then the weight vector is shifted toward the input vector. It is shifted away from the input vector if the computed and true classes do not match. Basic LVQ suffers from one important shortcoming: it can happen that the initial unit for learning is chosen far from the true class. Then this unit will be shifted toward the true class, while the others will do nothing. To avoid such situation, a penalizing factor is introduced in the form of a distance bias proportional to the difference between the winning frequency of a unit and the average unit winning frequency (proposed by DeSieno, 1998). This version of LVQ is called LVQ1 (with conscience). In order to improve learning in LVQ, other variants are developed by Teuvo Kohonen [21], called LVQ2, LVQ2.1, LVQ3, Extended LVQ and others.

Learning starts with the basic LVQ where Kohonen units compute the Euclidean distance according to the formula (3.40). The weights of the winning unit are adjusted according to (3.41). Then LVQ1 takes control by adding a bias b_i to the distance d_i of the units in the correct class:

$$d_i' = d_i + b_i. \qquad (3.42)$$

Using biased distances, LVQ1 computes the in-class winner, but also finds the global winner using unbiased distances. Weights are adjusted such that the in-class winner is moved toward the training vector (correct class) according to the equations:

$$\begin{aligned} \Delta w_c &= a(x - w_c) \quad \textit{if the in-class winner is equal to the global winner}, \\ \Delta w_c &= b(x - w_c) \quad \textit{if the in-class winner is not equal to the global winner}. \end{aligned} \tag{3.43}$$

If the global winner is not in the correct class, it is shifted away from it according to the formula:

$$\Delta w_c = -g(x - w_c). \tag{3.44}$$

Then the new biased distance is computed such that the estimated maximum distance $d_{i\max}$ is corrected using the winning frequency p_i for that unit and a constant h (conscience factor), which increases with learning:

$$b_i = h \cdot d_{i\max} \cdot (1 - Np_i), \tag{3.45}$$

where N is the number of units in the Kohonen layer per class. The initial value for p_i is 1/N. It is updated according to:

$$\begin{aligned} p_i &= (1 - j)\, p_i \quad \textit{if } i \textit{ is not the in-class winner, or} \\ p_i &= (1 - j)\, p_i + j \quad \textit{if } i \textit{ is the in-class winner}. \end{aligned} \tag{3.46}$$

Control is then taken by LVQ2 in order to refine that is solution is found by LVQ1. It is focused on the cases where the winning unit is in the wrong class, but the second best unit is in the correct class. Thus, the weights in LVQ2 are adjusted if one of the following occurs: if computed and true classes do not match, or the closest prototype weight vector is the correct class, or if the input vector is close to the binding hyperplane separating the two closest prototype weight vectors [37]. If one of these three situations occurs when LVQ2 takes control, then the winning unit (which is in the wrong class) is

moved away from the correct class, while the second best unit is moved closer to the input vector according to formulas:

$$\begin{aligned} w_1' &= w_1 - \alpha(x - w_1) \\ w_2' &= w_2 - \alpha(x - w_2) \end{aligned} \quad \text{if } x \text{ is near to } \frac{(w_1 + w_2)}{2}. \qquad (3.47)$$

Architecture of the Network

The network consists of three layers: the input, the Kohonen and the output layer. The summarized learning process through the layers is:

1) *from the input layer to the Kohonen layer*: inputs are transmitted from the input layer to the Kohonen layer through full connection.

2) *in the Kohonen layer*: the Kohonen layer learns using LVQ technique to compute distances, then LVQ1 to adjust winning distances by a penalizing factor, and last LVQ2 to improve learning in finding the second best units that are in correct classes.

3) *in the output layer*: the computed class $C(x)$ is compared to the correct class t, and weights in the Kohonen layer are adjusted according to the LVQ1 and LVQ2 equations.

Application of the Network

LVQ is applicable to any type of classification problem where the output classes are known. Its wide usage is noted in speech recognition, image processing, or other problems where data compression is needed and the probability distribution of the pattern sample is not known [37]. In cases where correct classes are not known, unsupervised networks such as Kohonen's Self-organizing maps (SOM) or regularization clustering techniques are recommended.

Input Parameters to Build the Network

The following input parameters must be set in order to build the LVQ network:

1) number of input, Kohonen, and output units:

There are no strict rules for choosing the number of Kohonen units. The number should be divisible by the number of output, units i.e., classes. It can also be set as a percentage of the number of training vectors.

2) number of learning iterations in the LVQ1 and LVQ2 phases:

The number of iterations in the LVQ1 phase is obligatory, while the LVQ2 phase is optional. The number of iterations can be determined heuristically or it can be set from the training file.

3) initial learning rate for LVQ1 and LVQ2

This parameter is used for initializing rates that are reduced over each learning phase.

4) LVQ2 width parameter

This is the parameter which determines the width of the hyperplane corridor in LVQ2 learning.

5) In-class winner always learns

This option enables LVQ1 learning with conscience.

6) conscience factor

This is the constant that is used to compute the bias in the LVQ1 learning.

7) frequency estimation

This is the p_i parameter that is also used to compute bias for penalizing winning units. It can also be set from the training sample as the inverse of the number of training vectors.

3.8 The Cascade Architecture Neural Network

To solve real-world problems with neural networks, the use of highly structured networks of a rather large size is usually required. A practical issue that arises in this context is that of minimizing the size of the network and yet maintaining good performance. A neural network with minimum size is less likely to learn the idiosyncrasies or noise in the training data, and may thus generalize better to new data. This design objective may be achieved in one of the two ways:

- *Network growing,* which starts off with a small multiplayer perceptron, small for accomplishing the task at hand, and then add a new layer of hidden neurons only when the design specification is not met.
- *Network pruning,* which starts of with a large multiplayer perceptron with an adequate performance for the problem at hand, and then prune it by weakening or eliminating certain synaptic weights in a selective and orderly fashion.

The *cascade-correlation learning architecture* [39] is an example of the network-growing approach. The rest of this section presents a detailed description of the cascade-correlation architecture, its algorithm and mathematical background, and its application to image processing.

Cascade-Correlation

Cascade-Correlation (CC) is an algorithm developed by Scott Fahlman [39]. There are two types of CC- Pruned CC and Recurrent CC. Both will be briefly discussed but the results obtained in the implementation are from using the standard algorithm, albeit with a few modifications. Strictly speaking, CC is a kind of meta-algorithm, in which other algorithms such as back propagation (BP) are embedded. The procedure begins with a minimal network that has some inputs and one or more output nodes as indicated by input/output considerations, but no hidden nodes. The LMS algorithm, for example, may be used to train the network. The hidden neurons are added to the network one by one, thereby obtaining a multilayer structure. Each new hidden neuron receives a synaptic connection from each of the input nodes and also from each preexisting hidden neuron. When a new hidden neuron is added, the synaptic weights on the input side of that neuron are frozen; only the synaptic weights on the output side are trained repeatedly.

Expanding on the above description, learning in CC takes place as repeating two-phase steps. The first involves the embedded standard learning algorithm, which in our case is Backpropagation (BP). During this phase the ideal activity as specified in the training pattern is compared with the actual activity and the weights of all trainable connections adjusted to bring these into correspondence. This is repeated until learning ceases or a predefined number of cycles have been exceeded.

The second phase involves the creation of a pool of 'candidate' units. Each candidate unit is connected with all input units and all existing hidden units. It is this which leads to the cascading architecture, as each new unit is connected to all preceding

units. There are no connections from these candidate units to the output units. The links leading to each candidate unit are trained with the selected standard learning algorithm (BP) to maximize the correlation between the residual error of the network and the activation of the candidate units. Training is stopped if the correlation ceases to improve or a predefined number of cycles is exceeded. The final step of the second phase is the inclusion, as a hidden unit, of the candidate unit whose correlation was highest. This involves freezing all incoming weights (no further modifications will be made) and creating randomly initialized connections from the selected unit to the output units. This new hidden unit represents, as a consequence of its frozen input connections, a permanent feature detector. The weights from this new unit and the output units will undergo training. Because the outgoing connections of this new unit are subject to modification its relevance to the final behavior of the trained network is not fixed. These two phases are repeated until either the training pattern has been learned to a predefined level of acceptance or a preset maximum number of hidden units has been added, whichever occurs first. Figure 3.8 illustrates the inclusion of the first two hidden units into a network undergoing training. The same steps are taken no matter how many hidden units are already included in a given network.

Mathematical Background

The training of the output units tries to minimize the sum-squared error E:

$$E = \sum_{p} \frac{1}{2} \sum_{k} (d_{p,k} - y_{p,k})^2, \qquad (3.48)$$

where $d_{p,k}$ is the desired (real) output and $y_{p,k}$ is the observed output of the output unit k for a pattern p. The error E is minimized by gradient decent using

$$e_{p,k} = (d_{p,k} - y_{p,k}) \cdot f'_p(net_k), \qquad (3.49)$$

$$\partial E / \partial w_{i,k} = \sum_p e_{p,k} I_{i,p}, \qquad (3.50)$$

where f'_p is the derivative of an activation function of the output unit k and $I_{i,p}$ is the value of an input unit or a hidden unit i for a pattern p. $w_{i,k}$ denominates the connection between an input or hidden unit i and an output unit k.

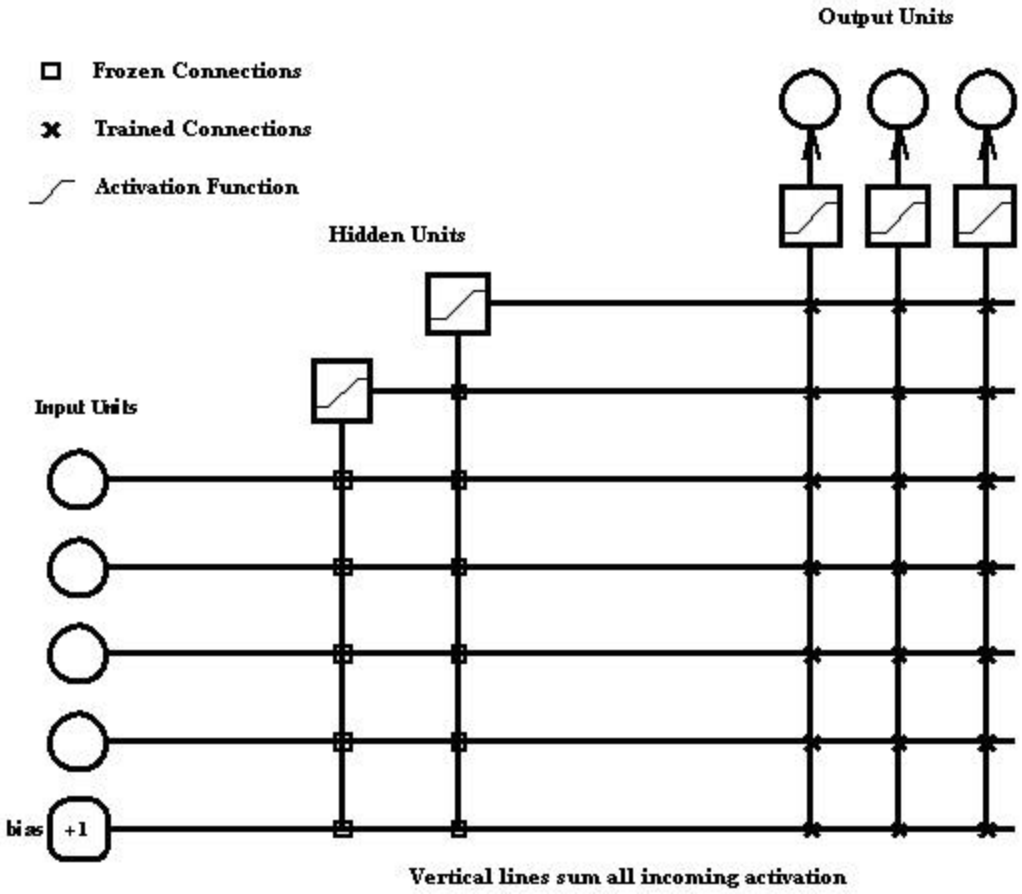

Figure 3.8: A neural net trained with cascade-correlation after 2 hidden units have been added. The vertical lines show unit inputs and horizontal lines show unit output. Square connections indicate frozen links. Cross connections indicate trainable links.

After the training phase the candidate units are adapted, so that the correlation C between the value $y_{p,k}$ of a candidate unit and the residual error $e_{p,k}$ of an output unit becomes maximal. Fahlman with gives the correlation:

$$\begin{aligned} C &= \sum_k \left| \sum_p \left(y_{p,k} - \bar{y}_k \right) \left(e_{p,k} - \bar{e}_k \right) \right| \\ &= \sum_k \left| \sum_p \left(y_{p,k} e_{p,k} - \bar{e}_k \right) \sum_p y_{p,k} \right| \\ &= \sum_k \left| \sum_p y_{p,k} \left(e_{p,k} - \bar{e}_k \right) \right|, \end{aligned} \tag{3.51}$$

where $\bar{y}_k$ is the average activation of a candidate unit and $\bar{e}_k$ is the average error of an output unit over all patterns p. The maximization of C proceeds by gradient ascent using

$$\begin{aligned} d_p &= \sum_k s_k \left(e_{p,k} - \bar{e}_k \right) f_p', \\ \frac{\partial C}{\partial w_i} &= \sum_p d_p I_{p,i}. \end{aligned} \tag{3.52}$$

where s_k is the sign of the correlation between the candidate unit's output and the residual error at output k.

In summary the standard CC algorithm is realized in the following way:

1. Start with a minimal network consisting only of an input and an output layer, both fully connected.
2. Train all the connections ending at an output unit with a usual learning algorithm until the error of the net no longer decreases.

3. Generate the so-called candidate units. Every candidate unit is connected with all input units and with all existing hidden units. Between the pool of candidate units and the output units there are no weights.
4. Try to maximize the correlation between the activation of the candidate units and the residual error of the net by training all the links leading to a candidate unit. Learning takes place with an ordinary learning algorithm. The training is stopped when the correlation scores no longer improves.
5. Choose the candidate unit with the maximum correlation; freeze its incoming weights and add it to the net. To change the candidate unit into a hidden unit, generate links between the selected unit and all the output units. Since the weights leading to the new hidden unit are frozen, a new permanent feature detector is obtained. Loop back to step 2.

The above five steps are repeated until the overall error of the net falls below the given tolerance value. The two variants of the standard CC are briefly discussed below:

Pruned Cascade Correlation

In the standard Cascade Correlation the Network is fully connected. Pruned Cascade Correlation [39] is a variant of the standard algorithm, which removes unnecessary links by applying selection criteria or a holdout set. This in turn decreases the training time required by the network.

Examples of the selection criteria are:

- Schwarz's Bayesian criterion (SBC)
- Akaikes information criterion (AIC)

- Conservative mean square error of prediction (CMSEP)

The *SBC*, the default criterion, is more conservative compared to the *AIC*. Thus, pruning via the *SBC* will produce smaller networks than pruning via the *AIC*. Both *SBC* and *AIC* are selection criteria for *linear* models, whereas the *CMSEP* does not rely on any statistical theory, but happens to work pretty well in an application. These selection criteria for linear model can sometimes directly be applied to nonlinear models, if the sample size is large.

Recurrent Cascade Correlation

Recurrent Cascade-Correlation (RCC) is a recurrent version of Cascade-Correlation and can be used to train recurrent neural networks (RNN) [40]. Recurrent Networks are used to represent time implicitly rather than explicitly. One of the most commonly known architectures of RNNs is the Elman model [42], which assumes that the network operates in discrete time steps. In the Elman model the outputs of the network's hidden units at a time t are fed back for use as additional network inputs at time $t+1$. Context units are used in the Elman model to store the output of the hidden units.

To use the standard CC with the Elman model (and recurrent models in general), only one change is needed to the algorithm. The hidden units' values are no longer fed back to all other hidden units. Instead, every hidden unit has links to all input units and also has one self-recurrent link. This self-recurrent link is trained along with the candidate units other input weights to maximize the correlation when the candidate unit is added to the active network as a hidden unit. The recurrent link is frozen along with all other links.

CHAPTER 4

Image Compression Using Neural Networks

After discussion of some of the various types of neural networks and their architectures, this chapter considers their role as an image compression tool. First the image compression problem is described, then the implementation of the single and parallel structure NNs with regard to image compression is presented, after which the new NN method is presented in detail. Its implementation is fully presented and its performance evaluated. Evaluation metrics for comparing the compression schemes are also described.

The Image Compression Problem

A still image I is described by a function $f : Z \times Z \rightarrow \{0,1,...,2^k - 1\}$, where Z is the set of natural numbers, and k is the maximum number of bits to be used to represent the gray level of each pixel. In other words, f is a mapping from discrete spatial coordinates (x,y) to gray level values. Thus, $M \times N \times k$ bits are required to store an $M \times N$ digital image. As was mentioned in Chapter 1, the aim of lossy image compression is to develop a scheme to encode the original image I into the fewest number of bits such that the image I' reconstructed from this reduced representation through the decoding process is as similar to the original image as possible, *i.e.*, the

problem is to design a compress and a decompress block so that $I \approx I'$ and $|I_c| << |I|$ where $|\ |$ denotes the size in bits. The above scenario is depicted in Figure 4.0.

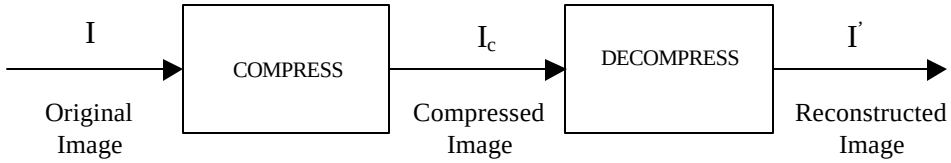

Figure 4.0: Image Compression Block Diagram

Evaluation metrics

In order to compare the quality achieved by lossy compression schemes, a metric is needed to quantify the quality of the reconstruction. The metric used to compare the neural network and JPEG image compression schemes is the peak signal-to-noise ratio (PSNR). Assuming that the original and reconstructed images are represented by functions $f(x,y)$ and $g(x,y)$ of the pixel plane position (x,y), respectively, the *PSNR* is defined by:

$$PSNR = 10 \log_{10} \frac{(2^k - 1)^2}{MSE} \qquad (4.1)$$

where k is the number of bits representing a pixel, and the mean square error (MSE) is:

$$MSE = \frac{1}{MN} \sum_{y=1}^{M} \sum_{x=1}^{N} \left[I(x,y) - \hat{I}(x,y) \right]^2 \qquad (4.2)$$

MSE is the cumulative squared error between the compressed and the original image, whereas the PSNR is a measure of the peak error. This metric does not adequately evaluate the visual quality of the decompressed image, but it is very simple to compute and relates to usual signal metrics when the original image I is viewed as the signal and

$\hat{I}$ is viewed as the same signal corrupted by some "noise" representing the deformation due to lossy compression.

4.1 Single-structure NN image compression implementation

Different approaches have been implemented for the single-structure NN. In this implementation, the backpropagation algorithm is employed and *Rprop* is used to speed the training process. All the algorithms described in this chapter are implemented in MATLAB.

4.1.1 Pre-processing

The image is split into J blocks $\{\boldsymbol{B}_j,\ j = 1, 2, \ldots, J\}$ of size M x M pixels. The pixel values in each block are rearranged to form a M^2 length pattern $\boldsymbol{C}_j = \{C_{1,j}, C_{2,j}, \ldots, C_{M^2,j}\}$, where $j = 1, 2, \ldots, J$ ($C_{i,j}$ is the i^{th} element of the j^{th} pattern). It is common practice to normalize the input patterns using the transformation $\boldsymbol{P}_j = f(\boldsymbol{C}_j) = (\boldsymbol{C}_j - m_{\boldsymbol{C}j})/\sigma_{\boldsymbol{C}j}$ where $m_{\boldsymbol{C}j}$ and $\sigma_{\boldsymbol{C}j}$ are, respectively, the average and standard deviation of $\boldsymbol{C}_j$. The reason for using normalized pixel values is because NNs operate more efficiently when the input is limited to a range of [0,1]. Patterns $\boldsymbol{P}_j$ are used in the training phase of the NN as both inputs and outputs.

4.1.2 Training

The network is trained to minimize the mean square error between output and input values, thus maximizing the peak signal-to-noise ratio (PSNR), with the proviso

that the image PSNR is measured for quantized values in [0,255] while the NN learning uses the corresponding real-valued network parameters.

The NN consists of the input and two layers: the hidden and output layers with number of nodes M^2, H, and M^2, respectively. Refer to Section 3.1.5 for a complete description of NN architectures. The NN acts as a coder/decoder. The coder consists of the input-to-hidden layer weights $v_{i,k}$ $\{i = 1,\ldots,M^2$ and $k = 1,\ldots,H\}$, and the decoder consists of the hidden-to-output layer weights $w_{k,m}$, $\{k = 1\ldots\text{H}$ and $m = 1,\ldots, M^2\}$. In the definitions above, $v_{i,k}$ represents the weight from the i^{th} input node to the k^{th} hidden node, and $w_{k,m}$ represents the weight from the k^{th} hidden node to the m^{th} output node. Compression is achieved due to the transformation of patterns $\boldsymbol{P}_j$, through the $v_{i,k}$ weights, by setting the number of hidden nodes H smaller than the input pattern length M^2 ($H<M^2$). Actually, even though $H<M^2$, no real compression has occurred because unlike the M^2 original inputs which are 8-bit pixel values, the outputs of the hidden layer are real-valued (between –1 and 1), which requires an astronomic number of bits to transmit. True image compression occurs when the hidden layer outputs are quantized before transmission. Thus the hidden layer neuron values are quantized to 8 bits to obtain an image that truly corresponds to a given compression ratio. The encoding/decoding processes are shown in Figure 4.1 below:

The network is trained with the resilient backpropagation algorithm (Rprop). As was mentioned in section 3.2, this allows for faster training. The training parameters, namely, the learning rate, epochs, and minimum gradient are set at 0.001, 1000, and 1×10^{-6} respectively. The hyperbolic tangent and liner transfer functions are used for the hidden and output layers respectively. The bias and layer weights are initialized.

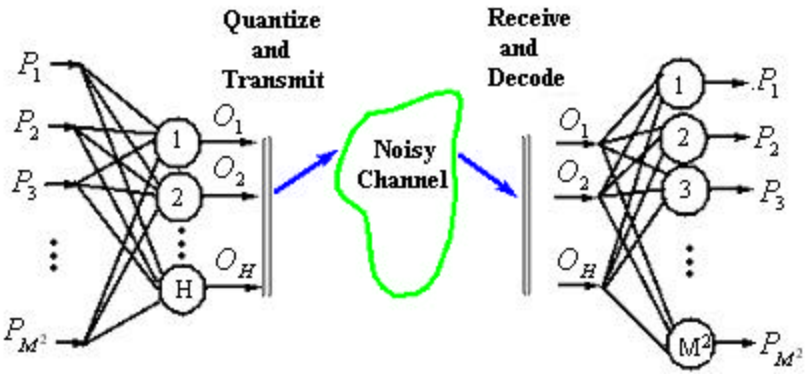

Figure 4.1: Single-structure neural network image compression/decompression scheme.

4.1.3 Simulation

After training, the network is simulated with the input and target matrices. This is done by multiplying the hidden layer weight matrix $w_{k,m}$ by the output of the pre-processor, adding the bias, and then applying the output layer transfer function to the result (Equation 3.1.2). This result becomes the output from the hidden layer, which otherwise could not have been directly obtained by using the Matlab "sim" function. The same process is repeated to obtain the output from the output layer, with the input to it being the output from the hidden layer. It must be pointed out that the weights and biases being referred to are the initial choices.

4.1.4 Post-processing

The next step is to reconstruct the simulated image. The coding product associated to the j^{th} pattern $\boldsymbol{P}_j$ is the hidden layer output $\mathbf{O}_j = \{O_{1,j}, O_{2,j}, \ldots, O_{H,j}\}$. The set $\{\mathbf{O}_j, m_{Cj}, \sigma_{Cj}\}$ together with weights $w_{k,m}$ is sufficient for reconstructing an approximation $\hat{\boldsymbol{C}}_j$ of the original pattern $\boldsymbol{C}_j$ in the decoding phase. Considering the overhead $\{m_{Cj}, \sigma_{Cj}\}$ due to the normalization function f, the compression ratio (CR) achieved is $M^2:(H+2)$.

4.2 Parallel-structure NN image compression implementation

The parallel-structure NN can be viewed as a single-structure NN but with multiple hidden layers. In that light, it is implemented in much the same way as the single-structure NN. The details presented in the previous section will thus apply here as well, albeit with a different mode of presentation.

Four networks $\{NET_k, k = 1,2,3,4\}$ with different number of hidden nodes (4, 8, 12, and 16) are used. Each NN is trained similarly to the NN in section 4.1. The goal is to achieve CR = 8:1. The coding procedure consists of two phases. In the first phase, each pattern is associated with a NET_k. As with all NNs, the larger the number of hidden nodes to which a pattern is assigned, the smaller the associated error $e_j^2 = (\hat{\boldsymbol{P}}_j - \boldsymbol{P}_j)\ (\hat{\boldsymbol{P}}_j - \boldsymbol{P}_j)^T$ between the original $\boldsymbol{P}_j$ and estimated patterns $\hat{\boldsymbol{P}}_j$. On the other hand, a large number of hidden nodes results in low CR. Initially, patterns are assigned to NNs so that the CR is as close as possible to the predefined value 8:1.

The second phase is an iterative procedure. At each successive iteration, the goal is to reduce the total error $E^2 = \sum e_j^2$ without changing the CR. Let $de_{j,k}^2$ be the error reduction caused due to reassigning pattern $\boldsymbol{P}_j$ from NET_k. to NET_{k+1}. Then the goal is achieved by reassigning a pair of patterns. Pattern $\boldsymbol{P}_{j1}$ is reassigned from NET_{k1} to NET_{k1+1} if the reassignment causes a maximum error decrease $de_{j1,k1}^2 = max_{j,k}(de_{j,k}^2)$, and pattern $\boldsymbol{P}_{j2}$ is reassigned from NET_{k2} to NET_{k2-1} if this results in a minimum error increase $de_{j2,k2}^2 = min_{j,k}(de_{j,k-1}^2)$. Iterations continue as long as the error E^2 decreases.

This parallel architecture has the advantage of providing better image quality for a given CR than the methods described in chapter 3 in terms of error E^2, including both single and parallel structures presented in Carrato et al. [43]. Furthermore, coding is

faster than previous parallel architectures. Nevertheless, training is still significantly slow due to the use of multiple networks. Moreover, the total number of weights is large. Thus, training cannot be part of the coding process; otherwise, it cannot be performed in real-time. Thus, the compression quality depends on the training data and their similarity to the test images.

4.3 Proposed Cascade Architecture

This section introduces an image compression method based on a cascade of adaptive units that provides better image quality than previous NN-based techniques. It borrows some of the idea of Cascade Correlation in that hidden units are added to the network one at a time. Each unit is equivalent to a feed-forward NN with a single node at the hidden layer. The proposed method exhibits a fast training phase that is included in the encoding process. Thus, it achieves independence of the training data, which makes it appropriate for general image compression applications. The implementation of the new architecture is described next.

4.3.1 Encoding process

The inputs $\boldsymbol{C}_j = [C_{1,j}, C_{2,j}, \ldots C_{M^2,j}]$ are defined as in section 4.1.1. The normalization function is

$$\boldsymbol{P}_j = f(\boldsymbol{C}_j) = (\boldsymbol{C}_j - \mathrm{m}_{\boldsymbol{C}j}) \qquad (4.3)$$

Patterns $\boldsymbol{P}_j$ are used as inputs/outputs to train the first unit. The output of the first unit's hidden node corresponding to the j^{th} pattern is denoted as $O_{j,k}$, $k = 1$. The first unit's estimated weight vector is denoted as

$$\mathbf{W}_k = [W_{1,k}, W_{2,k}, \ldots W_{M^2,k}],\ k = 1, \qquad (4.4)$$

and is equivalent to the weight vector connecting the hidden node to the output layer. Element $W_{i,k}$ is the i^{th} weight associated to the k^{th} unit. It is shown below that the weight vector connecting the input to the hidden node is not required in the implementation. The output $O_{j,1}$ and the weight vector $\mathbf{W}_1$ are required in the decoding process.

The first unit's error patterns are denoted as

$$\mathbf{e}_{j,1} = [e_{1,j,1}, e_{2,j,1}, \ldots e_{M^2,j,1}], \qquad (4.5)$$

They are defined as the difference between the original $\boldsymbol{P}_j$ and estimated patterns $\hat{\boldsymbol{P}}_j$ at the output of the first unit, after training is complete. The element $e_{i,j,k}$ is the i^{th} element of the j^{th} error pattern at unit k. If the set of error patterns at the output of unit k is defined as $\Re_k$ then only a subset of these error patterns is used as input/output to train the next unit. It should be pointed out that this is where this new method differs from most NN-based image compression methods. Instead of training with the entire patterns it makes sense to train with only patterns that do not meet a threshold. This enables faster training time and convergence. The subset $\Re_k^s$ consists of S error patterns $\mathbf{e}_{j,k}^s$, $j = 1, \ldots, S$, whose square sum is larger than a specified threshold Q: $\Re_k^s = \{\mathbf{e}_{j,k}^s \in \Re_k,\ \mathbf{e}_{j,k}^s\ \mathbf{e}_{j,k}^{s\,T} > Q\}$. Again, the second unit's hidden node outputs, denoted as $O_{j,2}$, and the weight vector $\mathbf{W}_2$ are stored to be used in the decoding process.

Similarly, the second unit's error patterns, $\mathbf{e}_{2,j}$, are defined as the difference between $\mathbf{e}_{j,1}^s$ and the estimated error patterns $\hat{\mathbf{e}}_{j,1}^s$ at the second unit's output. Again, only a subset of the new error patterns $\mathbf{e}_{j,2}$ is used as input/output to train the third unit. The procedure of adding/training units is repeated for as long as the CR does not exceed a specific target. There is an additional overhead per block indicating how many units

encode that block. It is important to note that since only a subset of error patterns trains each unit, the number of outputs $O_{j,k}$ per unit k is variable. This allows assignment of image-blocks with larger estimation error to more units, while keeping the same CR.

The threshold Q is based on the first unit's error patterns $\mathbf{e}_{j,1}$ and the CR. More specifically,

$$Q = a\left[\frac{1}{M^2}\sum_{i=1}^{M^2} \underset{j}{std}(e_{i,j,1})\right] CR. \qquad (4.6)$$

The justification for the threshold definition in equation (4.6) is as follows. A small threshold indicates an expectation for a small coding error. First, the threshold Q is proportional to the desired CR. A small CR promises a small coding error; therefore, the threshold can be set low. Second, the threshold is proportional to average standard deviation (ASD) of the error patterns between the original and the encoded images (using only the first unit). The ASD (content of the square bracket in equation (4.6)) is a "similarity" measure between the error patterns. A small ASD indicates that the error patterns may be relatively similar throughout the image-blocks. As a result, the additional adaptive units are expected to be able to produce a small coding error, thus the threshold can be set low. Finally, parameter a is a constant which was fixed and equal to 1.2 in all implementations.

The advantage of the cascade technique over existing parallel NN architectures is that the total number of weights is significantly smaller and that the number of hidden node outputs is variable. Furthermore, the training time requirements are low due to the units' low computational complexity. The algorithm converges in 4-5 iterations by repeatedly applying the simple set of equations:

$$O_{j,k} = \mathbf{W}_k \, \boldsymbol{T}_{j,k}^{T} / \mathbf{W}_k \mathbf{W}_k^{T}, \qquad (4.7)$$

$$\mathbf{W}_k = \sum_j O_{j,k} \, \boldsymbol{T}_{j,k}^{T} / \sum_j O_{j,k}^2, \qquad (4.8)$$

where $\boldsymbol{T}_{j,k}$ is equal to $\boldsymbol{P}_j$ if $k = 1$, and $\mathbf{e}^{s}_{j,k-1}$ otherwise.

Equation (4.7) gives the unit's optimum hidden layer outputs $O_{j,k}$ given the unit's weights. This is the result of minimizing the sum of square errors χ^2 between input and output patterns

$$\chi^2_{j,k} = \sum_j (\boldsymbol{T}_{j,k} - \hat{\boldsymbol{T}}_{j,k}) \, (\boldsymbol{T}_{j,k} - \hat{\boldsymbol{T}}_{j,k})^{T}, \qquad (4.9)$$

where $\hat{\boldsymbol{T}}_{j,k} = \mathbf{W}_k \, \boldsymbol{T}_{j,k}^{T}$, for unit k with respect to $O_{j,k}$. Similarly, equation (4.8) gives the optimum unit's weights given the hidden layer outputs $O_{j,k}$. This is the result of minimizing χ^2 with respect to $\mathbf{W}_k$. The conditions from which equations (4.7) and (4.8) are derived are shown below:

$$\frac{\partial c^{2}_{j,k}}{\partial O_{j,k}} = 0, \text{ and } \frac{\partial c^{2}_{j,k}}{\partial W_k} = 0 \qquad (4.10)$$

Since only the weights emanating from the hidden layer $\mathbf{W}_k$ and the hidden layer outputs $O_{j,k}$ are needed in the decoding process, it is imperative to obtain the optimum set $\{\mathbf{W}_k, O_{j,k}\}$ for each unit. The algorithm based on equations (4.7) and (4.8) directly attempts to find optimum values for both $\mathbf{W}_k$ and $O_{j,k}$.

4.3.2 Decoding process

Each block is decoded using the set $\{O_{j,k}, \mathbf{W}_k\}$, considering all units k used to encode that block. For instance, the first unit produces an estimate of patterns $\hat{\boldsymbol{P}}_j = O_{j,1}\mathbf{W}_1^{T}$ and the second unit an estimate of the first unit's error patterns $\hat{\mathbf{e}}^{s}_{j,1} = O_{j,2}\mathbf{W}_2^{T}$. The decoded block is obtained from summing of all those estimates and the block

average m_{Cj}. Figure 4.2(a) and 4.2(b) presents the proposed encoding and decoding schemes.

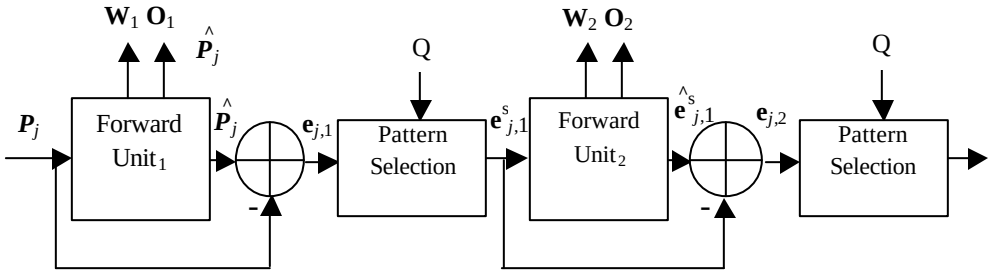

Figure 4.2(a): Encoding scheme for proposed cascade architecture.

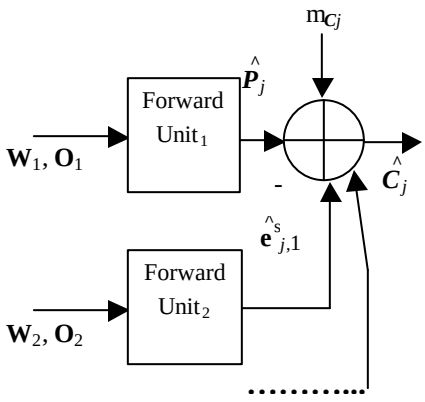

Figure 4.2(b): Decoding scheme for proposed cascade architecture.

The image compression results for the methods described above are presented next.

CHAPTER 5

Results

The image compression techniques presented in chapter 4 are tested on four different test images. This chapter presents and compares the results obtained from these tests for the single-structure, parallel-structure, and proposed cascade architectures. Also these results are compared with those of JPEG compression. The comparisons are made in terms of Peak Signal to Noise Ratio (PSNR) and computational complexity. Graphical and visual comparisons are also presented.

5.1 Comparisons in terms of PSNR

Table 1 presents the results obtained by compressing and decompressing four different test images (Lena, Baboon, and Peppers) with the single-structure NN and the proposed cascade method. The single-structure NN is both trained and tested on the same image to avoid dependence of the results on training data. Although this is impractical for real applications due to high training time requirements, it is useful for comparison purposes. Table 2 shows that the proposed cascade architecture gives higher PSNR for all tested images and for three different compression ratios (4:1, 8:1, 16:1).

The parallel-structure architecture resulted in PSNR = 33.8 dB (for CR = 8:1) for "Peppers" when trained with "Lena", and 33.0 dB when trained with "Baboon". The PSNR for the proposed technique is 35.2 dB for the same CR. The dependence of the parallel-structure NN on the training data now becomes apparent. Similarly when image

"Lena" was compressed by training the parallel architecture with "Peppers" the PSNR was 34.2 dB while for the cascade architecture it was 35.9 dB.

CR	Lena		Baboon		Peppers	
	Single-Structure	Cascade	Single-Structure	Cascade	Single-Structure	Cascade
16:1	28.93	31.36	21.74	21.99	28.28	30.07
8:1	31.87	35.88	23.04	23.42	30.57	35.21
4:1	35.19	38.96	25.07	25.46	33.06	37.35

Table 2: Comparison between single structure and cascade architectures in terms of PSNR (dB).

Quality Factor	CR	JPEG		
		Lena	Baboon	Peppers
34.5	16:1	34.24	33.16	33.87
74.9	8:1	38.19	38.79	37.51
92.0	4:1	42.89	43.02	42.47

Table 3: PSNR (dB) for JPEG algorithm.

Table 3 gives the PSNRs obtained using the JPEG algorithm. The quality factor determines the degree of compression and thus the compression rate. Strictly speaking, it is not possible to make a direct comparison between JPEG and the proposed cascade method or, in fact, between JPEG and single/parallel-structure NNs. The reason for this is has already been described in section 2.3.1 but will be re-iterated here: In JPEG, all quantized coefficients, which are transmitted, are entropy coded using a sophisticated differential pulse code modulation and run-length Huffman code. Without the entropy coding, JPEG gives substantially lower PSNRs or, equivalently, a much lower CR for the

same PSNR [34]. Nevertheless, it can be clearly seen that for the Lena image the proposed method, with no entropy coding, gives PSNRs that are only between 3 and 4 dB below the corresponding PSNRs obtained with JPEG. These results are further presented graphically in the next figures.

Figure 5.1 plots the PSNR values versus the CR for the reconstructed Lena image using the single-structure NN algorithm, the proposed cascaded method, and standard JPEG. As is clearly seen, the proposed method outperforms the single-structure NN. It is always better by 3dB. It seems, however, that JPEG compression performs better than the proposed method, which resulted in 32.8 dB, 36.7 dB, and 39.0 dB at compression ratios 16:1, 8:1, and 4:1, respectively. JPEG resulted in 34.3 dB, 39.5 dB, and 42.9 dB at the same compression ratios.

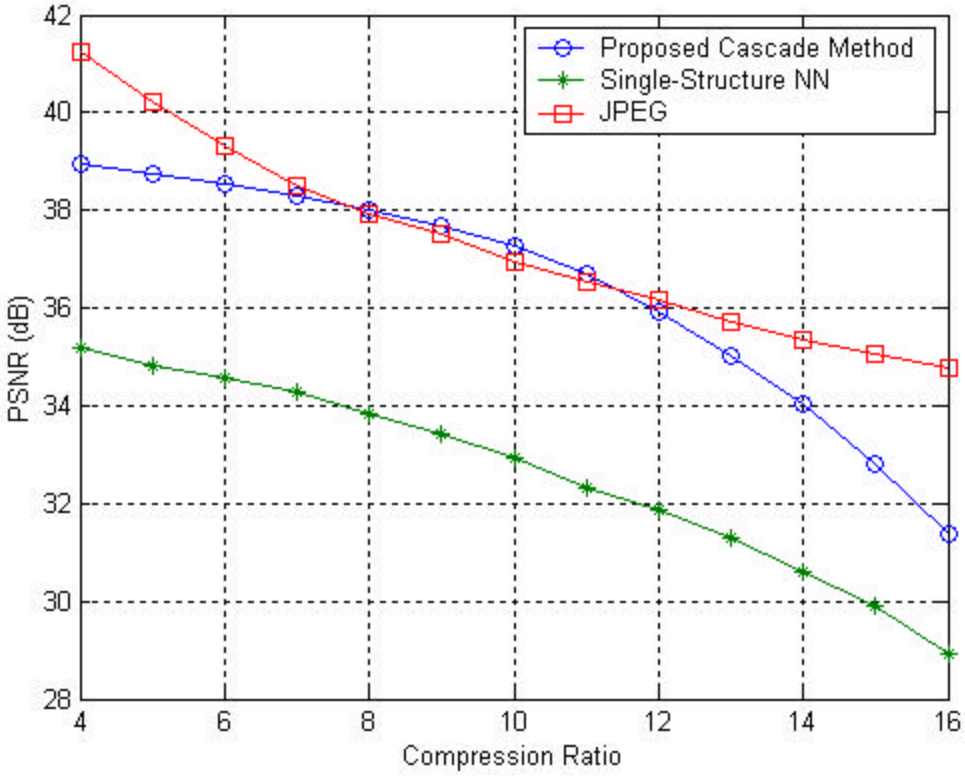

Figure 5.1: PSNR values for the reconstructed Lena image at different compression ratios.

In similar fashion Figures 5.2 and 5.3 plot the PSNR values for the reconstructed Peppers and Baboon images, respectively.

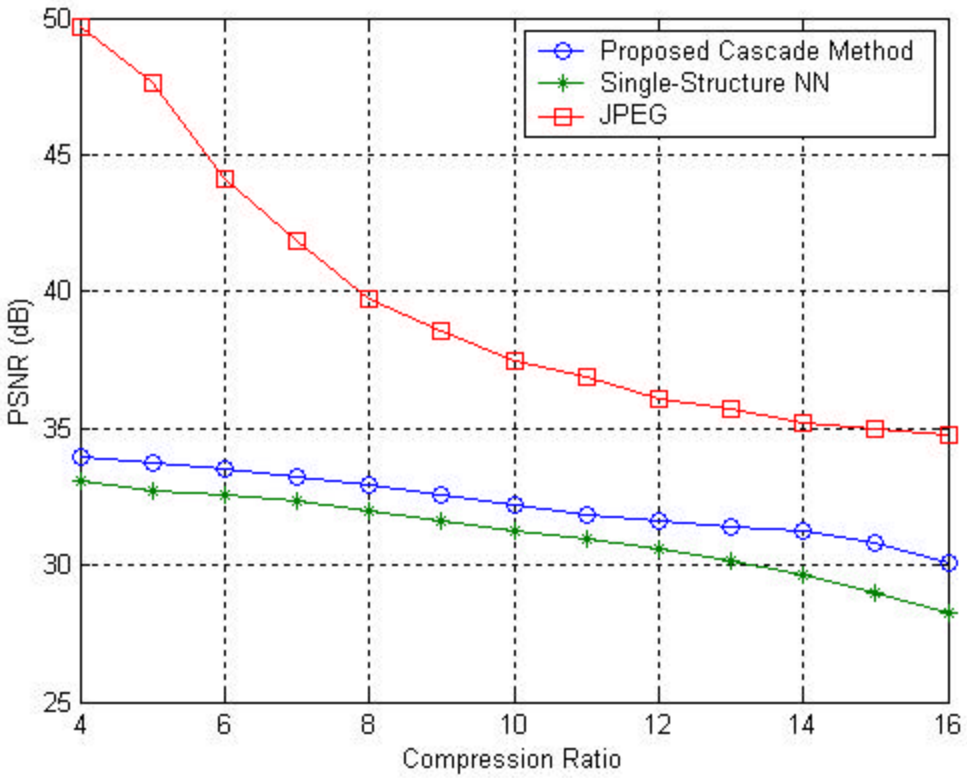

Figure 5.2: PSNR values for the reconstructed Peppers image at different compression ratios.

The JPEG approach yields a PSNR of 5dB higher on the average than the other two methods. However, the proposed cascaded approach does have certain advantages over the JPEG approach. They are:

1) The decompressed NN output can be recompressed further by up to 30% with a lossless compression scheme with minimal loss in quality. JPEG recompression rate is much smaller in comparison (less than 5%).
2) The coding/decoding time required for the cascade method is also much smaller. Once the off-line training is complete, the compressing and decompressing

process is significantly faster with the cascaded method. For instance, at 16:1 CR, it takes less than 2 seconds to compute, while at 4:1 CR the computing time is less than 1 second.

3) The parallel processing capability of NNs makes them superior to JPEG in terms of hardware implementation.

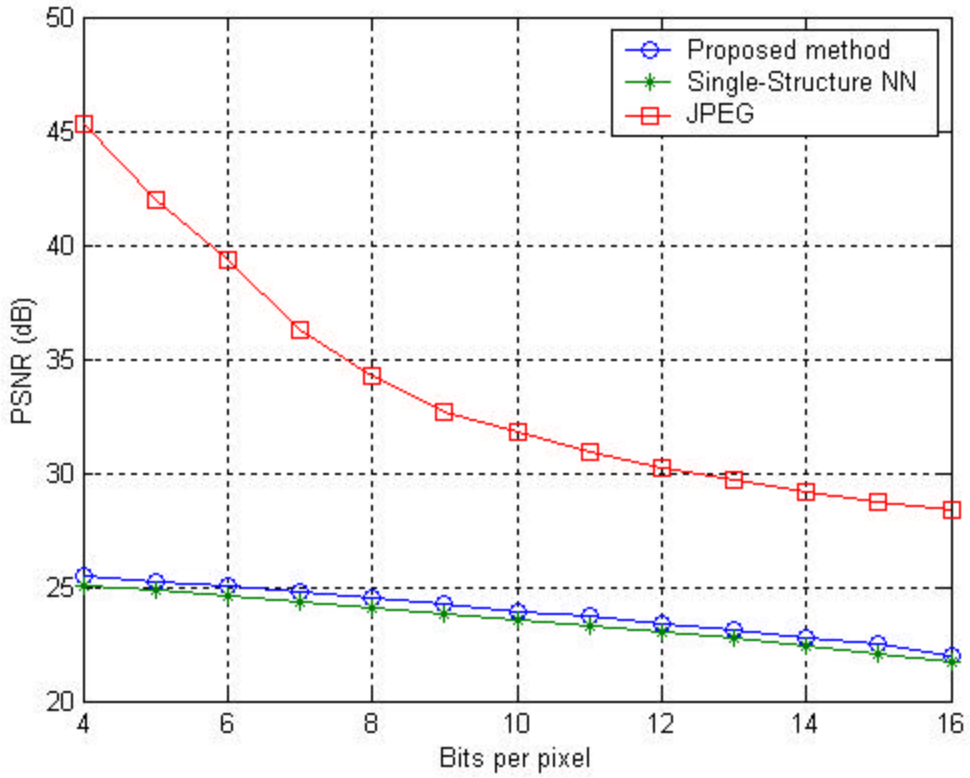

Figure 5.3: PSNR values for the reconstructed Baboon image at different compression ratios.

An example that presents a visual comparison of the compression results is shown in Figure 5.4. Figure 5.4(a) presents the original Lena image. Figures 5.4(b), 5.4(c), and 5.4(d) present the compressed image using, respectively, the single NN architecture, the proposed algorithm, and JPEG at CR 8:1. Although compression has apparently affected all compressed images, the proposed algorithm presents the best visual result, especially

around lines and edges. Notice that although the JPEG algorithm is better in terms of the PSNR, its performance is poor visually especially at high CRs, as is clearly seen when Figures 5.4(c) and 5.4(d) are compared.

Figure 5.4: (a) Original Lena image and Reconstructed Lena image at 8:1 compression ratio using the (b) Single-structure NN, (c) Proposed cascaded method, and (d) JPEG algorithms.

5.2 Comparisons in terms of Computational Complexity

Generally, NN techniques exclude training from the encoding process. Network(s) are trained with images other than the ones to be encoded. This subsection compares the parallel-structure and the proposed method in terms of required operations. As mentioned previously, the proposed method includes training in the encoding process.

The total number of hidden nodes in the parallel architecture is 40. In the assignment process, all patterns have to be presented to all four networks. This requires a total of 40 (nodes) $\times$ 64 (pattern size) $\times$ 4096 (blocks) $\times$ 2 (3 layer NNs) $\approx 21 \times 10^6$ multiplications and $40 \times 64 \times 4096 = 10.5 \times 10^6$ additions for a 512×512 image. Additionally, there are $3 \times 40 \times 64 \times 4096 = 31.5 \times 10^6$ operations required for the calculation of the error patterns. Hence, a total of 63×10^6 operations are required for encoding. This number does not include the iterative optimization algorithm which is also part of encoding. On the other hand, the number of operations for the cascade architecture is variable due to the adaptive nature of the technique. The number of iterations per unit is set to 4. It was found that for "Lena" (size 512×512), and for CR of 8:1 the number of multiplications and additions were 13×10^6 and 13×10^6 respectively, for presenting the patterns to the architecture. Furthermore, there were 19.5×10^6 operations required for the error pattern calculation. Hence, a total of 45.5×10^6 operations were required for encoding. Similarly for "Baboon" (size 512×512), a total of 53.2×10^6 operations were required for encoding. Therefore, although the proposed algorithm includes training in the encoding process, it requires fewer operations than the parallel NN technique.

CHAPTER 6

Discussion and Conclusions

Selected still images were compressed with various compression techniques. This thesis proposed a new cascade adaptive algorithm for image compression. The algorithm is based on the idea of neural network (NN)-based image compression. The major advantage of this algorithm is that it requires a small number of training parameters and has a fast training phase. Therefore, training can be incorporated into the encoding process. The proposed architecture exhibits smaller coding error than previous single structure and parallel structure NN architectures. Although the training phase is included in the encoding process, it exhibits faster encoding than parallel-structure and single-structure NN techniques. The new algorithm does not use any entropy coding. This makes it simple to implement and very robust against transmission errors usually encountered in high-noise environments, but it does limit the compression performance. The proposed technique has overall demonstrated the ability to compress still images at compression ratios of up to 16:1, and in fact can go higher than 32:1, while still maintaining a relatively high-quality output. It has also been shown that this output can be further recompressed with minimal loss in image quality, and compared to JPEG compression this is a major advantage. It is, however fair to mention that a major change from the JPEG algorithm used here is that wavelets replace the Discrete Cosine

Transform as the means of transform coding. Wavelet transforms are currently being employed in JPEG 2000. Among many things it will address are:

- Low bit-rate compression performance,
- Lossless and lossy compression in a single codestream, and
- Transmission in noisy environment where bit-error is high.

Many further improvements of the proposed cascade method can be thought of and some are certainly worth further work. In particular the following observations can be used to design networks with enhanced compression capabilities:

- The quantized hidden layer weights could be encoded by applying some form of lossless compression e.g., run-length Huffman code.
- An adaptive quantization scheme could be employed to minimize quantization errors and enhance image quality.
- The threshold detection parameter is a very important factor that can alter the results if not carefully controlled or considered. It is possible to obtain better results if this parameter was further optimized.
- Employ genetic algorithms [44] to:
    - configure the problem (image compression) for the network,
    - configure the network architecture, and
    - train the network.
- All the work described in this thesis needs to be extended to colour images.
- Currently, learning of the weights of each W_i, O_i pair is obtained using least mean squares and the PSNR used as a performance criterion is essentially

equivalent to a quadratic cost function. Other cost metrics (such as *LAB*-type measures) could be used to carry out learning for colour images.

In addition to the general scheme described above, some other enhancements related to the non-linearity of the input-output amplitude mapping of the compression/decompression scheme could be examined. It is expected to obtain further quality improvement with appropriate compensation of non-linearity. This compensation can also be part of the learning scheme. Moreover, the adaptive selection of the level of compression to be used at the transmitter side can be improved by making use of the state of the transmission medium - specifically of the network being used. This would be particularly relevant if we are dealing with an ATM (Asynchronous Transfer Mode) network. The adaptive decision can be based on feedback about network state - such as current load on the network - as well as PSNR and/or visual quality metrics. For example, in case of little load on the network, small compression ratios can be favored, thus increasing visual quality. Similarly, in case of a heavily loaded network, visual quality can be sacrificed while transmiting with maximal compression. This adaptive decision can also be learned.

With some of the improvements described above, it is expected that compression ratios as high as 250:1 can be achieved for gray-scale images, and still higher levels for colour, with quality levels of the order of SNR = 40 dB for gray-scale images, and acceptable *LAB*-type measures and PSNR levels for colour images. With the wide variety of approaches to image compression, and in particular neural networks, there is a compelling need for a comprehensive evaluation of the proposed technique. Such an

evaluation would require training and testing on a common set of images. The results and derivations obtained, while comparatively excellent, constitute the basis for developing additional results or equations for image compression.

REFERENCES

[1] S. Carrato, "Neural Networks for Image Compression," in *Neural Networks: Advances and Applications*, E. Gelembe, Ed. Amsterdam, The Netherlands: North-Holland, 1992.

[2] J. Jiang, "Neural-Networks Image Compression- A Survey," *Signal Processing: Image Compression*, to be published.

[3] W. B. Pennebaker and J. L. Mitchell, "JPEG Still Image Data Compression Standard. New York": Van Nostrand Reinhold, 1993.

[4] G. K. Wallace, "Overview of the JPEG (ISO/CCITT) still image compression standard," in *Proc. SPIE*, vol. 1244, PP. 220-233, Feb. 1990.

[5] M. Rabbani and P. W. Jones, "Digital Image Compression Techniques", vol. TT7 of Tutorial Texts Series. Bellingham, WA, SPIE Optical Eng. Press, 1991.

[6] R. C. Gonzalez and R. E. Woods, "Digital Image Processing", Reading, MA: Addison-Wesley, 1992.

[7] H. B. Mitchell, N. Zilverberg, and M. Avraham, "A Comparison of Different Block Truncation Coding Algorithms for Image Compression," *Signal Processing: Image Commun*., vol. 6, pp. 77-82, 1994.

[8] N. Efrati, H. Liciztin, and H. B. Mitchell, "Classified Block Truncation Coding-Vector Quantization: An Edge Sensitive Image Compression Algorithm," *Signal Processing: Image Commun*., vol. 3, pp. 275-283, 1991.

[9] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun*., vol. C-28, pp. 84-95, Jan. 1980.

[10] N. M. Nasrabadi and R. A. King, "Image coding using vector quantization: A review," *IEEE Trans. Commun*., vol. 36, pp. 957-971, Aug. 1988.

[11] R. M. Gray, "Vector quantization," *IEEE ASSP Magazine*, vol. 1, pp. 4-29, 1984.

[12] P. P. Gandhi, W. E. Darlington, and H. L. Dyckman, "Error-resilient image compression based on JPEG," in *Proc. SPIE, Still-Image Compression II*, San Jose, CA, Jan. 29-30, 1996, vol. 2669, pp. 106-123.

[13] S. Lin and D. J. Costello Jr., "Error Control Coding". Englewood Cliffs, NJ: Prentice-Hall, 1983.

[14] M. Morimoto, M. Okada, and S. Komaki, "Robust mobile transmission using hierarchical QAM," *Signal Processing: Image Commun*., vol. 12, pp. 127-134, 1998.

[15] G. W. Cottrell, P. Munro, and D. Zipser, "Image compression by backpropagation: An example of extensional programming," in *Models of Cognition: A Review of Cognition Science*, N. E. Sharkey, Ed. Exeter, U.K.: Intellect Books, 1989, pp. 208-240.

[16] Y. Benbenisti, D. Kornreich, H. B. Mitchell, and P. Schaefer, "Normalization techniques in neural network image compression," *Signal Processing: Image Commun*., vol. 10, pp. 269-278, 1997.

[17] A. Basso and M. Kunt, "Autoassociative neural networks for image compression," *European Trans. Telecommun*., vol. 3, pp. 593-598, 1992.

[18] H. M. Abbas and M. M. Fahmy, "Neural model for Karhunen-Loeve transform with application to adaptive image compression," *Proc. Inst. Elect. Eng*., vol. 140, pt. I, pp. 135-143, 1993.

[19] S. K. Kenue, "Modified backpropagation neural-network applications to image compression," in *Proc. SPIE, Applicat. Artificial Neural Networks III*, Orlando, FL, Apr. 21-24, 1992, vol. 1709, pp. 394-407.

[20] S. Carrato, S. Marsi, "Parallel Structure Based on Neural Networks for Image Compression," *Electronics Letters*, Vol.28, No.12, pp. 1152-1153, June 1992.

[21] T. Kohonen, "Self-organization and Associative Memory". Springer Verlag: 3rd Ed., 1989.

[22] K. R. Rao and P. Yip, "Discrete Cosine Transform: Algorithms, Advantages, and Applications". New York: Academic, 1990.

[23] E. Oja, "Principal components, minor components, and linear neural networks", *Neural Networks*, Vol.5, pp. 927-935, 1992.

[24] E. H. Adelson, E. Simoncelli, "Orthogonal pyramid transforms for image coding", *Visual Communications and Image Processing II*, Proc. SPIE, Vol.845, pp.50-58, 1987.

[25] W. Zettler, J. Huffman, D. C. P. Linden, "Application of Compactly Supported Wavelets to Image Compression", *Image Processing Algorithms and Techniques, Proc. SPIE*, Vol.1244, pp.150-160, 1990.

[26] A. E. Jacquin, "Image Coding Based on a Fractal Theory of Iterated Contractive Image Transformations", Vol.1, No.1, p.18-30, January 1992.

[27] M. Kunt, M. Benard, R. Leonardi, "Recent Results in High-Compression Image Coding", *IEEE Transactions on Circuits and Systems*, Vol.CAS-34, No.1, pp. 1306-1336, 1987.

[28] F. Zahedi, "Intelligent Systems for Business, Expert Systems With Neural Networks". Wodsworth Publishing Inc., 1993.

[29] B. Widrow and S. D. Sterns, "Adaptive Signal Processing", New York: Prentice-Hall, 1985.

[30] NeuralWare, Neural Computing: A Technology Handbook for NeuralWorks Professional II/Plus and NeuralWorks Explorer, NeuralWare, Aspen Technology, Inc., 1998.

[31] M. Zekic, "Neural Networks for Time Series Prediction in Finance and Investing", in *Proceedings of the 6th International Conference on Operational Research Society*, Zagreb, pp. 215-220, 1996.

[32] A. J. Shepherd, "Second-Order Methods for Neural Networks", Springer-Verlag, 1997.

[33] T. Masters, "Advanced Algorithms for Neural Networks, A C++ Sourcebook". John Wiley & Sons, 1995.

[34] S. V. Kartalopoulos, "Understanding Neural Networks and Fuzzy Logic, Basic Concepts and Application", IEEE Press, 1996.

[35] N. B. Karayiannis, G. M. Weigun, "Growing Radial Basis Neural Networks: Merging Supervised and Unsupervised Learning with Network Growth Techniques", *IEEE Transactions on Neural Networks*, Vol. 8, No. 6, pp. 1492-1505, 1997.

[36] T. Masters, "Practical Neural Network Recipes in C++", Academic Press, 1993.

[37] D. W. Patterson, "Artificial Neural Networks", Prentice Hall, 1995.

[38] M. Riedmiller, and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *Proceedings of the IEEE International Conference on Neural Networks*, 1993.

[39] S.E. Fahlman, and C. Lebiere, "The cascade-correlation learning architecture", in *Advances in Neural Information Processing Systems 2* (D.D. Touretzky, ed.), pp. 524-532, San Mateo, CA: Morgan Kaufmann, 1990.

[40] D. F. Specht, "A General Regression Neural Network", *IEEE Transactions on Neural Networks*, Vol. 2, No. 6, pp. 568-576, 1991.

[41] R. A. Jacobs, M. I. Jordan, S. Nowlan, G. E. Hinton, "Adaptive Mixtures of Local Experts, Neural Computation", No. 3, pp. 79-87, 1991.

[42] J. L. Elman, "Finding structure in time". *Cognitive Science*, 14: 179-211, 1990.

[43] S. Carrato, S. Marsi, "Parallel Structure Based on Neural Networks for Image Compression," *Electronics Letters*, Vol.28, No.12, pp. 1152-1153, June 1992.

[44] D. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, 1988.

APPENDIX

MATLAB Codes

Functions used for Single-structure NN compression:

- *singlestruct* - compresses test image using single-structure algorithm.
- *mynewff* - creates a feedforward NN.
- *mysim* - simulates the network and returns the output.

Functions used for Proposed cascade architecture compression:

- *cascade* - compresses test image using cascaded architecture algorithm.
- *estimate* - creates and trains the network.

Functions common to both Single-structure and Cascade architecture algorithms:

- *image_to_blocks* - breaks up the image into z × z blocks and changes its dimensionality.
- *reconstruct* - performs inverse of "image_to_blocks" operation.

Function used for JPEG compression:

- *jpeg* - compresses test image using JPEG algorithm.

Script which executes all three compression techniques:

- *run_all*.

Test Images

Image1.gif, Image2.gif, and Image3.gif contain the test images Lena, Baboon, and Peppers, respectively.

VITA

Chigozie Obiegbu was born in 1973 in PortHarcourt, Nigeria. He graduated from Marist Brothers' Juniorate in 1990. In the fall of the following year he began studies in electrical/electronic engineering at the Federal University of Technology, Owerri, and graduated with a Bachelor of Engineering degree in August 1997. During the next three years he worked as a full time employee with two different companies. In the spring of 2001 he came to the University of New Orleans to pursue graduate studies in electrical engineering and physics. He worked as a teaching assistant in the department of electrical engineering and is currently working as a teaching assistant in the department of Physics.