OPTIMAL USE OF COMPUTING EQUIPMENT IN AN AUTOMATED INDUSTRIAL INSPECTION CONTEXT

Matthew James Jubb,
B.Sc. Hons (Dunelm)

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE COUNCIL OF THE UNIVERSITY OF DURHAM FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (PH.D.).

MAY 1995

Contents

Abstract

This thesis deals with *automatic defect detection.* The objective was to develop the techniques required by a small manufacturing business to make cost-efficient use of inspection technology.

In our work on inspection *techniques* we discuss image acquisition and the choice between custom and general-purpose processing hardware. We examine the classes of general-purpose computer available and study popular operating systems in detail.

We highlight the advantages of a hybrid system interconnected via a local area network and develop a sophisticated suite of image-processing software based on it.

We quantitatively study the performance of elements of the TCP/IP networking protocol suite and comment on appropriate protocol selection for parallel distributed applications. We implement our own distributed application based on these findings.

In our work on inspection *algorithms* we investigate the potential uses of iterated function series and Fourier transform operators when preprocessing images of defects in aluminium plate acquired using a linescan camera.

We employ a multi-layer perceptron neural network trained by backpropagation as a classifier. We examine the effect on the training process of the number of nodes in the *hidden* layer and the ability of the network to identify faults in images of aluminium plate. We investigate techniques for introducing positional independence into the network's behaviour. We analyse the pattern of weights

induced in the network after training in order to gain insight into the logic of its internal representation.

We conclude that the backpropagation training process is sufficiently computationally intensive so as to present a real barrier to further development in practical neural network techniques and seek ways to achieve a speed-up. We consider the training process as a search problem and arrive at a process involving multiple, parallel search "vectors" and aspects of genetic algorithms. We implement the system as the mentioned distributed application and comment on its performance.

List of Figures

Declaration

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous appplication for a degree, and that all sources of information have been duly acknowledged.

M. J. Jubb, May 1995

Acknowledgements

I would like to acknowledge and give thanks to the following people, without whose support and help this work and the production of this thesis would not have been possible.

To my parents, Roderic and Jennifer, who supported me steadfastly throughout.

To my supervisor, Alan Purvis, for his guidance and wisdom.

To my colleague and friend Nick Bailey, who schooled me in the One True Way of computing.

To Ruth and Vanessa, faces reproduced here as image-processing source material.

To all my friends in Durham but especially to Chris Drury and the Poplar People for looking after me in my continuation year.

Publications

Analysis of Network Protocol Performance in the Context of Multi-Workstation Parallel Distributed Applications,

Matthew J. Jubb and *Alan Purvis*,

Microprocessing and Microprogramming, 1994 Vol. 40, No. 10-12, pp 807-810.

Parallel Distributed Backpropagation Training on a Large Workstation Cluster,

Matthew J. Jubb and *Alan Purvis*,

Submitted to Euromicro, September '95, to be held in Como, Italy.

Chapter 1

Introduction

1.1 The Promise of Image Processing

Image processing is the field of study which involves the manipulation of images. Photography can be thought of as a particular kind of image processing, as can the analogue electronic processing which takes place inside a domestic television set. However, the term is more commonly used today to describe manipulation of pictures using a computer; image processing implies *digital* image processing. Aspects of a broad range of other scientific disciplines are involved, these include, for example, electronics, optics, computer science and mathematics. For this reason, perhaps, the topic as a whole is at present evolving very quickly, as developments in all of these fields are made.

Image processing is a field which has shown great promise and yet has hitherto largely failed to live up to the layman's expectations. This may be because images have an inherent subjective appeal which causes the subject to receive a disproportionate amount of attention and expectation. Since the beginning of the automation revolution, it has been widely anticipated that automatic systems would appear with human-like faculties, and although the field of robotics is now well-developed and finds regular industrial application, the only successful automatic vision systems to date work exclusively with tightly-constrained scenes. Thus the realisation of the general-purpose labour-saving robot, like the general automatic vision system, remains far off.

1.2 Motivation for this Work

Development towards real-world image processing applications appears now to be accelerating, driven largely by advances in hardware which have made high-powered, large memory computers available cheaply, and specialised boards for video capture, video processing and graphics are also widely available at modest cost. This is beginning to transform the immediate prospects for the development of realistic, "integrated" vision systems by academic groups.

Machine vision technology may now be sufficiently mature to turn many current research techniques into practical systems, for the benefit of manufacturing, inspection, surveillance, visually-guided control and other commercially important applications. However, the take-up of technology in many of these areas has been slow, in part because until there are clear demonstrations of integrated performance, few industries are prepared to invest the funds necessary to break new ground. Two additional factors contribute to the impasse:-

- A lack of incentive for academic researchers to take their work beyond theoretical developments and to demonstrate and quantify the performance of algorithms in the context of specific applications.

- Successful applications of machine vision in niche markets are often not taken up more widely because their industrial exploiters, unwilling to release technical information and marketing objectives, fail to define tractable needs requiring further research.

We have therefore taken the opportunity within an academic research project to contribute to the body of practical as well as theoretical image-processing knowledge.

1.3 Synthesis, Enhancement and Recognition

A thorough introduction to image processing theory lies outside the scope of this thesis, however, [1] and [2] do give a detailed consideration. We shall instead briefly comment on what we perceive to be the three main subdivisions of image processing, in order to illustrate where our own work fits in.

Image *synthesis* relates to the artificial construction of images. The computer-generated graphics display produces a familiar, brightly-coloured, slightly blocky image, however, with the expenditure of sufficient resources computer image synthesis becomes a powerful tool in graphic design and film-making, for example. For moving pictures, the rate at which even simple frames are produced is constrained by processing power to be many times slower than realtime, however, when the images are stored and played back at a realistic speed, the results can be quite breathtaking.

Possibly the most significant technique of interest here is ray-tracing, which involves the modelling of a scene with objects, surfaces and light sources, and the simulation of the interaction of light with the items within the scene in order to construct a simulated view or image. Whilst the earliest essays in ray-tracing involved objects designed to make the simulation task straightforward, such as totally-reflecting spheres and totally absorptive surfaces, for example, more recent work has shown that a sophisticated model, including textured and partially-transmissive surfaces, for example, can yield extremely lifelike results.

Image enhancement has perhaps the largest body of associated established theoretical knowledge. Until the recent emergence of image recognition as a popular area of study, the greater part of all research into image processing dealt with image enhancement. As evidence of this, Castleman's introductory text [1] published in 1979 deals mostly with enhancement techniques. Here the paradigm is almost a parallel of photography - a real-world image is input, processed in some way and then output or displayed.

Enhancement techniques include filtering and convolution operators, transforms and morphology, for example. The usual aim is to enhance an image acquired under non-ideal circumstances in order that it is more palatable for human consumption. Image compression can be regarded as a subset of image enhancement - here the aim is to find operators which will reduce an image's storage requirement, reducing its storage or transmission costs, and yet allow it to be re-expanded and presented in an acceptable way afterward.

Image recognition is still largely an unknown and emerging field. Although the proper paradigm seems to be well-established, and will be discussed in more detail, there are few established guides to selection of appropriate techniques. For example, as Castleman says in his chapter devoted to image recognition, entitled "Measurement and Classification":-

> *There are few analytical means to guide the selection of features. Frequently intuition guides the listing of potentially-useful features.*

There is, however, enormous interest in the use of image recognition systems for a wide variety of tasks which are achieved with varying rates of success, for example, automatic traffic monitoring [3, 4], barcode reading, postcode reading and automatic inspection and defect detection. These last two areas are the focus for our work, and a more specific introduction to these is given in chapter 2.

1.4 Image-Processing Hardware

1.4.1 Acquisition

Almost all image-processing applications make use of the same hardware subsystems, and we would next like to discuss these.

With the exception of systems which are *output-only*, for example the ray-tracing simulators described, some means is always required of *acquiring* data from the natural world and thereby converting a physical image into some kind

of logical representation. There is still some choice as to exactly what kind of *transducer* may best be employed for this purpose, for example, laser and infrared systems are discussed in chapter 2, although the outputs from these can only be regarded as images in the loosest sense.

More typically, image-processing systems require a *camera* as input transducer. Previously cameras themselves possessed a number of subsystems with some scope for flexibility in the choice of hardware for each. The basic principle is that light reflected by some portion of a natural scene is detected by some kind of photosensitive device which converts the illumination level into an electrical representation. A movable aperture plate or mirror can be used to change the portion of image being "viewed" and thereby "scan" the image. Typical photosensitive devices included photomultiplier tubes, photovoltaic cells and photodiodes, for example.

More modern cameras made use of electronic image tubes such as the *plumbicon* and *silicon diode vidicon*. Here the image is focussed optically on a *target* covered by a photoconductive material which converts light intensities to corresponding densities of charge. The target is scanned by an electron beam steered by deflection plates as in an oscilloscope; variations in beam current correspond to variations in illumination at the corresponding target position.

Most recently of all, advances in silicon integration technology have made the charge-coupled detector camera the most favoured of all camera systems. This takes the form of an array of solid-state photosensitive devices, one per pixel[1], which again converts light intensities into build-ups of charge which can be read electronically. Advantages over the tube cameras include cost of manufacture and robustness.

With either a tube or CCD camera, some additional electronic circuitry is required in order to turn the camera's output into a digital representation. In the case of a tube, a purely analogue signal is output, and an analogue to digital

[1]Short for picture element, the smallest subdivision of a digital image

converter needs to be used which will define both the spatial and intensity resolutions of the digital image. In the case of the CCD, the intensity is still read in an analogue form, and therefore needs to be digitised, but the spatial resolution is fixed by the layout of the CCD array.

1.4.2 Storage

Some specialised digital storage is usually required by an image processing system, which needs to be able to store an entire digital image, and to receive this at the digitisation rate of the analogue to digital converter. Such a store is often called a *frame buffer*, and often has the capability of being scanned at the frame rate in order to regenerate a display of the digitised image.

1.4.3 Processing

The most problematic aspect of the processing stage is the volume of data involved. Images are of course two-dimensional, and this means that the processing problem is an order of magnitude more computationally-intensive than for audio signal processing, for example. Consider as an illustration an audio compact disc player and a digital video mixer. Both are systems which involve digital signal processing of an originally analogue signal, yet the audio system costs less than £100, whereas the video system costs many tens or even hundreds of thousands of pounds.

Consequently, if video processing is to be carried out in real time, that is, at a video frame rate, then processing hardware must typically be specially designed with a particular algorithm or family of algorithms in mind. Such hardware will typically need a high-bandwidth connection to the frame buffer. The Imaging Technology system which we have used in our work on interfacing in chapter 4 is a good example - the dedicated processing card here is known as an *arithmetic logic unit*, and can perform simple operations such as thresholding, convolution and image addition/subtraction, for example. This type of approach is, however,

rather limited in its application to novel algorithms, although there is some interest in combining the advantages of general-applicability and high-performance using specialised hardware containing field-programmable gate arrays [5].

General-purpose Digital Processors

Where complex image processing algorithms are in use, such as the artificial intelligence techniques we will describe, it is more usual for a conventional general-purpose digital computer to be employed. Here image data needs to be passed over the computer's internal *bus*, typically this is not sufficiently high in bandwidth for processing at frame rate to be achieved in any but the most sophisticated of systems. *Off-line* processing is therefore the norm. It is, however, worthwhile to consider complex algorithms which could in future be employed in real-time as computer capabilities evolve, and this has been our approach in our work on algorithms in chapters 7 and 8.

The falling costs of silicon integration and the consequent rapid uptake of information technology in a wide variety of applications mean that there is currently a plethora of different kinds of general-purpose processors available, and a choice made between these would seem to be far from straightforward. In chapter 3 we have therefore discussed the evolution of the popular classes of device including the IBM PC-clone, the Unix workstation and the mainframe, highlighting the relevance of data networking issues, since these are particularly important for a data-intensive field of computation such as image-processing. Chapter 4 follows up our considerations of selection of a particular type of processing engine and the low-level performance of networking protocols with an examination of the various *operating systems* available for use on these processing engines, since the operating system facilities have a large bearing on the ease with which heterogenous hardware can be integrated together. This is of importance since a typical image-processing system is at present often a hybrid of various cameras, dedicated processors and general-purpose computers.

More Specialised Microprocessors

Microprocessors which are more specialised in that they are dedicated to signal processing whilst not being algorithm-specific are becoming increasingly popular as both research and implementation platforms. Common examples at the time of writing include the Texas Instruments TMS320C40, the Intel i860 and the Inmos T800 and T9000 Transputers, these last having particular application in parallel processing. Whereas digital processing systems in mass-produced products were at one time implemented using semi-custom or full-custom designs, developments in silicon technology have meant that the required performance for many designs can now be obtained using a more general-purpose processor. These devices are typically more demanding in terms of software development effort, requiring use of less flexible languages such as OCCAM or of native assembler code, but as processing power increases with time it would be expected that the use of higher-level languages such as C becomes prevalent, with useful consequences for the ease of use of such systems.

1.5 Summary

Now that we have introduced the general topic of image-processing, together with the issues surrounding the choice of items of hardware which go together to make up any practical image-processing system, we will in chapter 2 introduce and discuss the particular subset of the topic which is relevant to the problem in view. This is automatic defect detection, a specialised image recognition problem.

Chapter 2

Industrial Quality Control using Automated Inspection

2.1 Introduction

In this chapter we introduce and discuss the motivation for and characteristics of industrial quality control through automatic inspection. With the support of literature review items we discuss the relative merits of the main sensor techniques available before moving on to examine a range of contemporary systems which usefully illustrate the context of our own work. Finally we look more specifically at some web process inspection applications and introduce our own particular vision problem, which falls into this category.

2.2 Vision Systems and Quality Control

Quality control is today widely held to be an important part, or sometimes *the* most important part of any industrial manufacturing process, or indeed any type of business process at all. To borrow from signal-processing phraseology, quality control can be regarded as a kind of "feedback" activity, comparing the result produced at the end of the process with the *desired* result, and modifying the process parameters or starting conditions until they are the same. Quality control has such significance attached to it because it is the means of checking that what is happening in practice is what was originally planned and accounted for.

In a manufacturing context, quality control is the means by which the path can be found between two undesirable extremes, one of alienating customers by producing goods of inadequate quality, the other of escalating production costs and hence reducing competitiveness through over-specification manufacture. Successful quality control automation can enhance profitability by increasing speed and hence production throughput, also by giving a more objective and repeatable metric of quality than is possible using manual inspection.

Quality control in manufacturing was equally important before the advent of the present generation of automatic inspection systems, although in early times it was possibly not isolated as being worthy of study in its own right. Huang et al.[6] give a useful overview and comparison of early inspection techniques. Arguably the most straightforward from an implementation point of view are *manual* techniques, these require only the training of a human operator to check the part or material in question according to specified criteria. As might be expected the characteristics of manual labouring found in other industrial applications are equally evident here:- the sophistication of human faculties is such that a very varied and flexible set of inspection methods can be undertaken, however, Huang observes that the key disadvantages of manual inspection are its low speed, leading to production bottle-necks, and also the fact that the monotonous nature of the work leads to boredom and fatigue in the operators, thereby reducing performance, although performance at peak can be very high. *Contact* techniques may be usefully employed for very low-grade inspection, for example, to ensure that a part is approximately the right shape, in the right orientation and has no gross defects present. These techniques are typically not very sophisticated or accurate, and suffer from the additional disadvantage that the sensors will tend to wear out in use.

Huang defines machine vision as:-

> *... the application of devices for optical, non-contact sensing to automatically receiving and interpreting an image of a real scene in order*

to obtain information and/or control machines or processes.

It should be noted, however, that such an optical sensing system does not *necessarily* involve the use of a camera. In their work on automated inspection of coated papers, Ivonen et al. [7] compare and contrast the main alternatives in the context of web[1] inspection, these being:-

1. Laser-based systems. Typically laser light is directed at a quickly-rotating polygonal mirror, thereby creating a laser "spot" which moves rapidly over the web. Optical fibres collect light transmitted through the web and pass this to photomultiplier tubes. Strengths of this kind of approach are the high sensitivity which is typically obtained, allowing detection of very subtle defects. However, weaknesses include the strong dependence of sensitivity on process speed, the fact that performance is readily degraded by vibrations and dirt, the large amount of space typically required for installation, and the frequent, complicated recalibration required which results in high maintenance costs.

2. Infra-red systems. Usually these involve many discrete infra-red light source and detector elements, typically light-emitting diodes and phototransistors or photodiodes. Advantages here are relatively low cost, high sensitivity and sensitivity/speed independence, reduced need for recalibration, high reliability and robustness, and the fact that such a system can often be fitted into tight manufacturing configurations. Ivonen suggests that the main disadvantages lie with difficulties in detecting features parallel to the direction of process movement.

3. Video camera or CCD[2] systems. In the coated paper context in view, these

[1] A continuous sheet of product - e.g. paper, aluminium, shoe leather etc.
[2] Charge-coupled detector

are based on fluorescent or incandescent light and CCD elements which detect light penetrating through the web, although reflected light can alternatively be used if the transmissivity of the product is low. Strengths here include medium to high sensitivity. Disadvantages again include speed/sensitivity dependence, the requirement of lots of space for installation, plus frequent lamp replacement.

Ivonen's summary of detection methods is by no means exhaustive, for example, ultrasonic techniques might be used to detect flaws deep inside a solid metal component. Although strictly speaking this would be a contact technique, requiring a transducer mounted on the component surface, the procedure of transmitting ultrasonic radiation into the metal, and subsequently detecting that reflected by a potential flaw has more in common with a vision technique. Clearly the exact nature of the manufacturing process will ultimately determine which inspection technique proves to be the most suitable.

2.2.1 Charge-coupled Detector Systems

Ivonen has a somewhat mixed view as to the merits of CCD camera systems, however, it should be added that CCD detectors, together with their associated digitisation equipment are now falling in price due to the mass manufacture caused by demand for their use in a multiplicity of applications, to such an extent as to greatly enhance the cost effectiveness of CCD-based systems. In spite of their low cost these can give a very attractive specification in terms of sensitivity, signal to noise ratio, reliability and ease of setting-up when compared to other acquisition methods. Correspondingly, the cost of the hardware required to convert analogue video signals provided by a camera into the digital domain normally used for processing is also diminishing, as digital video techniques find an increasingly diverse range of applications in consumer electronics. It should also be noted that a laser-based system is difficult to use in scattered-light mode, and is therefore not generally the method of choice where the material is opaque.

Furthermore, infra-red based systems offer limited resolution due to the minimum size of available discrete elements, rendering them unsuitable for use where defects are very small, unless complex optics are also provided.

Huang is enthusiastic about CCD camera systems, asserting that vision inspection is becoming more popular in industrial applications mainly due to ongoing improvements in microprocessors, lighting systems and camera technology. However, he suggests that vision systems are not appreciated to their full potential in industry, mainly due to high development costs.

In many industrial settings, therefore, a video-based inspection system may be the method of choice. Clearly this kind of system will only be of interest when images of the product surface can give useful quality control information, but it should be noted that CCD detectors also lend themselves well to inspection using infra-red and ultra-violet wavelengths. Beneath-the-surface inspection may also be possible if images are acquired indirectly, for example, by digitisation of an X-ray film.

The scale at which digital video inspection is carried out can be varied through appropriate specification of the camera's optical system. For example, flaws in a magnetic disk head only visible through a microscope may be detected and analysed just as easily as gross flaws, several inches wide, in sheet steel plate.

2.3 Typical Applications

Contemporary machine vision applications are conceptually very straightforward, and we should like to present some examples in order to set the context for our own work on vision system development.

2.3.1 Inspection of Toothpicks

In [6], various alternative contemporary techniques are described for vision inspection of toothpicks. These include area calculation, through identifying pixels

lying on the part boundary. These can be determined by observing which pixels have both "black" (background area) and "white" (toothpick area) neighbours, and this technique goes some way to solving the frequently-occurring image-processing problem of incomplete boundaries, although it does rely on lighting and acquisition conditions being such that that the image can effectively be split into two intensity levels, corresponding to "toothpick" and "non-toothpick". The resulting calculated area can be compared with that of a known "good" part, subject to expected tolerances.

Alternatively, boundary following can be directly used to obtain a measure of the toothpick image's perimeter, this being again compared with that expected for a "good" part, or a histogram analysis could be conducted on a toothpick image acquired with a greater number of grey levels, the frequency of occurrence in the part of each again being checked.

2.3.2 Inspection of Magnetic Disk Heads

Although conceptually very simple, automatic inspection has hitherto been a notoriously and perhaps surprisingly difficult engineering problem. As humans we are all intimately acquainted with the characteristics of our own biological vision systems, and after a cursory inspection it does seem reasonable that one should be able to take steps towards replicating its performance artificially. The profundity of practical difficulties which occur when we attempt to implement machine vision in practice can therefore be rather unexpected. In [8], Sanz gives his impression of the problem:-

> *There is a big gap between successful demonstration of a method on carefully-controlled data and a functioning system in the manufacturing environment.*

The system which Sanz describes is a further typical example of a contemporary machine vision implementation, and is again based on CCD elements

as detectors. Standard averaging operators are used to improve the signal to noise ratio at the expense of a longer total acquisition time, and pixel values are scaled according to their position, in order to compensate for non-uniform illumination. Two images per part are acquired using light-field and dark-field microscopy respectively, and identification of the part's extents is made using maximum-likelihood parametric curve-fitting, in effect this constitutes use of the Hough transform. Feature extraction is then performed, and extracted features are parameterised by area, size of defect-enclosing box, perimeter and relative location of centroid. Sanz adopts an unusual design of classifier, saying that:-

> *Conventional classifier design based on statistical training is not feasible, because of the reduced number of defect samples that are usually available in the initial phase of almost any inspection task.*

Instead, Sanz implements a rule-based classifier, based on rules originally drawn up to aid humans in carrying out the inspection manually. For example, a scratch (defined as being below a defined area to perimeter ratio) over a certain length will cause the part in question to be deemed a quality control failure.

This is an unusual orientation of circumstances, since in general a reliable classification ruleset cannot easily be directly determined, as complex relationships between the parameters involved may be required. The purpose of statistical training, moreover, is to determine a suitable ruleset from a series of examples, and this is therefore not required in Sanz' case.

2.3.3 Human Skin Inspection

In [9], White et al. detail a machine vision system designed to quantify and assess lesions on human skin. The purpose is to provide the means for an objective assessment of the patient's skin condition to be determined. As treatment continues this is difficult to do by eye, as it requires comparison between the patient and photographs of the condition in an earlier stage. The vision system's objective

metric of the skin condition should allow a doctor to determine much more easily whether the patient is responding to a particular treatment.

The authors concentrate on the discussion of simple pre-processing operators which are used in an attempt to isolate lesions from the normal skin background. The simplest of all is a straightforward thresholding operator, since the lesions tend to be much darker than the surrounding area. The main drawback here is the sensitivity to variations in illumination, which can rarely be made uniform for a contoured subject, and it is therefore difficult to set a threshold which can successfully isolate lesions over the whole extent of the image. Furthermore, "false alarms" will be generated by pores and hairs which are also darker than the background.

The authors suggest that this illumination sensitivity can be offset by a *background correction* or normalisation process. A local first derivative filter may also be of use, an example is the Sobel filter which tends to enhance contrast boundaries. However, the problem here is that most of the features thereby extracted will have broken boundaries, leading to a requirement for some algorithm to reconnect them, such as the Hough transform. Furthermore, intensity gradients of non-defect features may be comparable to, or in excess of those of defects, and the problem of setting a suitable threshold therefore remains.

A logical progression is to a second-order derivative filter, such as the Laplacian of Gaussian, often written *LoG*. This is very computationally expensive, however, the LoG is insensitive to global and scale changes in illumination, but furthermore if zero-crossings are used, no threshold is needed to discriminate the feature boundaries. Such filters are, however, very sensitive to noise, so a great deal of noise-suppression prefiltering would be required.

The authors also comment on *morphological* operators, which involve iterative "building-up" or "paring away" of pixel regions with the object of producing a fully-connected pixel boundary corresponding to a contrast edge. White et al. comment that these are interesting, yet do not solve the above problems convincingly. They go on to develop the concept of a brightness "pit" as their basis

for feature segmentation, with parameters such as area, perimeter length, angle of conicity, volume and mean-squared distance from centroid being extracted in order to enable classification. Two statistical classifiers are compared, these being Fischler's Multivariate Discriminant Analysis (MDA), and K'th Nearest Neighbour Analysis (KNN). The first of these is of particular interest since it is ready-implemented in the commercial statistics package, SPSS.

2.3.4 Focus on Preprocessing Operators

Some researchers in the field of machine vision inspection have been concerned with preprocessing operators as a study in their own right. For example, in [10], Muhamad and Deravi concentrate on parameters which can conveniently be extracted from an image as a data reduction step by means of a spatial grey-level dependence matrix. The parameters themselves include energy, entropy and inertia.

Transforms may also be of use as preprocessing operators, although by definition an invertible transform cannot bring about any data reduction. It can be the case, however, that certain image characteristics are more clearly evident in the transform domain. For example, the discrete Fourier transform (DFT) may be used to reveal frequency domain characteristics - these may give a useful characterisation of texture, for example, which often has a recognisable frequency-domain signature which is relatively difficult to observe in the spatial domain. The DFT may also be used to conduct filtering operations more efficiently, for example in the case where the same image needs to be filtered in various different ways. This could justify the computational expense of the transform when compared with conducting many spatial-domain convolution operations. We conduct our own investigation into the usefulness of the DFT in the context of our particular defect detection problem in chapter 6.

In [11], Mihovilovic et al. present an information theoretic paper on a novel development from Gabor known as the "chirplet" transform. The "wavelet"

transform is an intermediary step; to arrive in the wavelet transform domain a time vs. frequency plot is dissected into time-invariant cells with an aspect ratio depending on frequency. The signal powers present in each of these cells can then be considered the output coefficients of the transform. In the "chirplet" transform, the cell shape is allowed to vary with time. A fundamental property of frequency-domain transforms is that good frequency resolution implies poor temporal or spatial resolution, and vice versa, so this property of the "chirplet" transform is useful in that it allows a representation to be made anywhere between these two extremes. For example, in speech analysis, brief, wideband consonants would be better presented with high temporal sharpness, whereas sustained vowel sounds would be better represented if sharpness were relaxed in favour of higher frequency-resolution. There are clearly also useful machine vision analogies.

Other authors have concentrated exclusively on single preprocessing operators which have particularly interesting properties. Examples are Waite in [12], who focusses on fractal transform operators as applied to general image-processing, and Jacquin in [13], who explores the same family of techniques, but whose motivation for data reduction is the desire to achieve efficient data *compression.*

2.3.5 Timber Inspection

In [14], Cho et al. report on a system for inspecting timber. This material seems to provide a particularly rich set of defect types, for example, wormholes, knots, cracks and so forth. Automatic inspection of timber and timber products appears to be of particular commercial interest, and is covered extensively in the literature, since an ability to grade the material quickly and objectively can have a big impact on profitability. In chapter 5, related work in [15] by Brzakovic et al. is considered in detail, since it exemplifies well the layout of components which we assert to be typical of the generic machine vision application. However, Cho's work is also worthy of mention here since the authors have concentrated on studying the *classification* stage of the process, reaching interesting conclusions

concerning the rule-based and statistical classifiers already mentioned, comparing these also with the neural network classifiers which are one of the focal points of our work. These are:-

- Rule-based classifiers: the advantages here are that no training set is required; in general the ruleset is derived from a human expert with experience of conducting the inspection manually. Where required it is possible to use the training data to "tune" the subjectively established decision parameters, although this is not necessarily straightforward.

- Statistical classifiers: K'th Nearest Neighbour (KNN) is the particular method under scrutiny here. Advantages are that this can outperform conventional parametric classifiers when the actual distribution of data is different from the assumed distribution (KNN makes no assumptions about the distribution of the data). However, it is difficult to find an optimal value of k which produces the best performance for a training set with a finite number of samples.

- Neural network classifiers: a multi-layer perceptron (MLP) trained by back-propagation is considered. The advantages are that the speed of classification is faster than with KNN, once the training phase has been completed. The authors assert that the MLP approximates the optimal discriminant function defined by Bayesian theory, which suggests that the behaviour of the MLP becomes asymptotic to that of KNN as training progresses. Selection of the number of hidden neurons is a particular problem, however. The authors also suggest that the multi-layer perceptron has an ideal architecture for *parallelising* the classification process with a view to speeding up the implementation. However, we should say that we have yet to encounter networks which are of such a size that classification takes any significant amount of time, although if a large throughput of classified data were required then parallelisation might be useful, most likely in the form of multiple copies of the trained network. The *training* process *is* however

very compute-intensive, and we present our work concerned with the parallelisation of this in chapter 8. With the parameters of the timber inspection application in mind, the authors select neural network classifiers as being their most favoured method.

Huang et al. in [6] also favour the use of neural network classifiers, saying that these are of particular industrial interest due to their special characteristics, these being learning by example, fault tolerance and pattern recognition. Huang expects the integration of neural networks and vision systems to improve overall vision system performance.

2.4 Automated Inspection of Web Products

We should next like to discuss *web product* inspection, a particular subset of the family of vision problems from which we have presented applications, since the original motivation for our own work arose in a web inspection context.

2.4.1 Typical Defects in Photosensitised Aluminium Plate

Our early work was aimed at the detection of surface defects in the photo-sensitive coating of a particular aluminium sheet product. After rolling and coating the material is cut to the customer's required size and ultimately used for lithographic paper printing.

Two types of defect were of particular interest to the sheet manufacturer:-

- Roughly-circular voids up to one centimetre across. These are caused by variations in the photo-sensitive coating's composition, leading to air bubbles which are later squashed flat by a roller. An image cross-section showing an example of such a defect is shown as figure 1.

- Scratches which may be many tens of centimetres in length running parallel to the manufacturing process' direction of travel caused by small particles

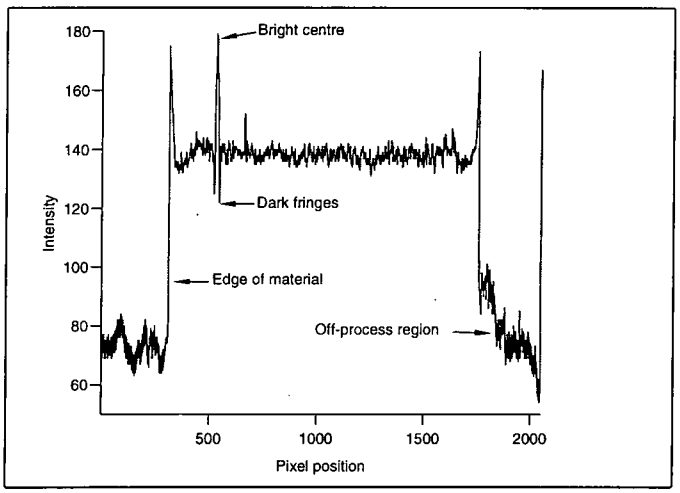


Figure 1: Image cross-section showing typical "void" defect

of grit in one of the roller mechanisms. A visually-similar type of fault may also occur running perpendicular to the process direction, this is caused by roller vibration.

The first class of defect can in general be detected by straightforward convolution filtering followed by a threshold operator, since these faults are typified by an outer fringe and a centre which are, respectively, much darker and lighter than the surrounding material.

The nature of the production process gives a slightly textured matt finish to the aluminium's photo-sensitive coating, and this makes detection of scratches very difficult or impossible using filtering techniques, since the defect's RMS power is less than that of the mottled background. The human eye, however, has no difficulty in picking out the fault because of its consistent positional correlation throughout the length or breadth of the material.

2.4.2 General Web Inspection Problems

A wide variety of materials are produced as a continuous strip or web - including metals such as steel and aluminium, paper, glass, textile fabrics etc. These are routinely inspected during manufacture to detect and identify defects, and, as is the case with more general problems, inspection using a human operative leaves

much to be desired. Automatic systems are already in industrial use for certain simple inspection tasks where a high equipment capital cost is acceptable.

In principle, many more web materials could be inspected automatically using machine vision, but unless the surface being inspected is smooth, monochrome and patternless, the signal processing required may be very difficult to achieve in real time. For the most difficult surfaces, suitable processing methods are often only just starting to be theorised.

Commercial considerations are also important - to be viable, instrumentation must be cost-effective. Many existing solutions which work are much too expensive for widespread application. It is necessary to provide new methods, and special-purpose hardware to implement these methods.

There is already a substantial academic literature on web inspection. The signal generated by a defect can be regarded as a "target" (radar analogy) or a "message" (communications analogy), and methods well-developed in these fields for detecting signals accompanied by noise can be exploited to provide detection. However, at some stage the analogy breaks down, and novel methods have to be introduced accordingly.

CCD linescan cameras are usually used for sensing - they offer high resolution and faster acquisition than framescan devices, without a need to freeze motion of the strip. Laser scanners are used when the highest speeds and resolutions are necessary, but they are very expensive and suffer from other problems as described.

2.4.3 Web Inspection Case Studies

In [16], Cho et al. themselves review a number of contemporary vision applications and suggest that the most successful systems tend at present to be custom-designed, since there are often advantages to be gained in terms of efficiency and cost when the inspection problem is tightly constrained. However, Cho's view is that their inherent inflexible and nonversatile structures prevent them from

being easily adapted, and that general-purpose inspection systems will become more attractive as computing power becomes cheaper.

With this in mind, Cho describes a knowledge-based machine vision system for industrial web inspection which he asserts is applicable to a broad spectrum of different inspection tasks.

Cho says that the problems peculiar to web process inspection are as follows:-

- The material flow is continuous, and it is therefore difficult to stockpile the material with a view to feeding it through the inspection system at a rate slower than that at which it is produced. This means that a truly effective inspection system needs to operate in realtime, at the same speed as the manufacturing process.
- The flow rate is high, typically 2 to 20 linear feet per second. This means that the requirement for realtime inspection above becomes highly demanding.
- A high spatial resolution is required to detect defects.
- Defects make up a small proportion of the total surface area.
- The same defect may manifest itself in many visually dissimilar ways, complicating the process of classification.

The system described by Cho et al. is divided into two logical modules, the first deals with feature segmentation, the second with classification and recognition.

The segmentation module attempts to extract potential defects from the background area. Its purpose is to reduce the volume of processing required in the higher-level phase - this is typically more sophisticated and time-consuming, and so there are economies to be gained through conducting this only on regions which first-order considerations suggest may contain a defect. Cho discusses three categories of segmentation technique:-

1. Edge detection. These are likely to produce false edges in the presence of noise, also the resulting edges are likely to be incomplete, requiring the use of some kind of algorithm to fill in the gaps.

2. Texture operators. One example is the fractal dimension, which can be used to derive a one-dimensional metric of *self-similarity* from a two-dimensional binary pattern. In Cho's view these are very computationally complex, too much so to be of use in current realtime industrial inspection applications. We give further consideration to the fractal dimension in chapter 6.

3. Histogram-based thresholding (pixel intensity profiling). This is the authors' preferred method.

Segmented regions are then passed on to the classification or recognition module. The authors' preferred technique here is based on neural networks, involving a three-layer feed-forward multi-layer perceptron trained by backpropagation. It is claimed that this is slightly superior in eventual performance than the K'th Nearest Neighbour (KNN) statistical classifier, and that the neural approach also requires less development effort.

The network described has 10 input, 10 hidden and 5 output neurons. The input neurons receive a feature vector comprising area, average grey level, elongatedness, compactness, contrast, variance, absolute central moment, edge density and two others which give a measure of the extent to which defects are touching the edge of the frame - certain timber defects are more likely to occur at the edges. Activation of the output neurons corresponds to five classes, these being clear wood and four defect types.

Olsson et al. in [17] also have some interesting conclusions about automatic web inspection. They are concerned particularly with the inspection of sheet steel coated with chromium dioxide and with tin.

They suggest that methods of inspecting such surfaces can be divided into

three types, these being intensity-based (thresholding), texture-based (comparison with a statistical model of the surface) and scattering-related, this last involving measuring and comparing the intensities of light scattered at various angles to the material surface.

Olsson et al. describe a web inspection system which uses a rotating decagonal mirror to sweep a laser beam across the process in 200ms. Detection is by means of a CCD device which is split into numerous sectors, allowing a degree of scattering to be measured. One feature vector component is produced and fed to the classification system for each of these sectors.

The described classification is by means of Kohonen's Learning Vector Quantisation (LVQ2) algorithm, the main advantage of this cited as being the high speed of the training phase. However, the authors' view is that more sophisticated neural networks could give better results.

2.5 Summary and Conclusions

Although some variance is evident, there seems to be something of a concensus among the authors we have reviewed here and others as to the most satisfactory paradigm to adopt when addressing vision inspection problems, namely that the whole process can be split into a number of subtasks, each of which has a number of applicable techniques. In logical order we will refer to these as:-

- Data acquisition.
- Feature segmentation.
- Feature parameterisation.
- Classification of parameter vectors.

We shall now briefly discuss each of the vision subtasks in turn.

2.5.1 Data Acquisition

If it has been decided that a vision inspection system is appropriate to a particular quality control task, due to advantages over manual and contact techniques, the system component which logically comes first is the data acquisition stage, which is concerned with converting a real world scene into some kind of machine representation. This stage includes the acquisition transducer, which as we have discussed may be based on a range of technologies including lasers, charge-coupled detectors and infra-red transducers. It seems to be clear that CCDs are becoming increasingly favoured for use here.

Previously, inspection systems have usually been tailored to a particular application, but it is likely that the reduced development costs associated with a more general system may make these more attractive in future as they become easier to engineer due to increases in available computer power and advances in processing algorithms. The use of CCDs would be appropriate in this context, since their setting-up is much less dependent on the exact manufacturing process mechanics than that of a laser-based system, for example.

Under the heading of data acquisition we will also mention any preprocessing which may be required in order to compensate for any physical factors complicating the acquisition process such as non-uniform illumination, for example, or noise generated by the CCDs themselves. Under some circumstances this can be quite complex, as with the skin inspection example in [9], where the subject is contoured and difficult to fix with respect to the camera. In other circumstances a simple positional offset may be all that is required.

2.5.2 Feature Segmentation

Having acquired a machine representation of the inspection scene, the next step in the process is that of feature segmentation, which is the task of identifying which image areas are likely to contain features of interest, and isolating each potential feature. The motivation for this is that the higher-level processing

which is subsequently required is highly compute-intensive, and therefore the volume of data fed into it needs to be reduced by some simpler technique in order to keep the scale of the computation problem manageable. Consider, for example, that a multilayer perceptron neural network could be designed which directly received a value from each of the thousands of pixel positions making up the inspection image, thereby eliminating the segmentation and parameterisation stages. However, the logistics of constructing a suitably-representative training set to allow such a network to make useful deductions from the image, not to mention the computational problem of *training* the network on such a set would render this a rather fruitless approach.

The techniques which we have mentioned in a feature segmentation context have included edge detection, texture, intensity-profiling, morphology and filtering operators, and there is no concensus as to the most appropriate among these since this seems to be largely application-dependent. However, there is a wealth of existing theory relating to such operators as they are equally relevant to the more developed field of image enhancement. It appears however, that in general none of these operators performs such that the desired inspection can be directly performed on the result, eliminating the need for parameterisation and classification stages, except in very carefully controlled conditions. It can be said, therefore, that the provision of feature segmentation and parameterisation stages makes the overall system more *robust* and better-able to deal with the inconsistencies that are characteristic of real-world data.

2.5.3 Feature Parameterisation

Having isolated certain portions of the image likely to contain a feature of interest, further data reduction is required since direct high-level processing of even a reduced volume of pixel data is still currently an unmanageable task. Therefore the image portions are generally *parameterised* to produce a *feature vector* which typically contains just a few floating-point values which are characteristic of the

image section. Parameters discussed have included mean distance from centroid and average intensity, for example. The key task here would seem to be to reduce the data being presented to the classification stage to the bare minimum required to differentiate the expected features from one another and from normal material, in order to make the higher-level processing as straightforward as possible. *Orthogonality* of parameters within the feature vector would therefore seem to be important, in other words, the parameters should be selected such that each characterises a unique quality of the image section not represented in any of the others.

2.5.4 Classification of Parameter Vectors

Having isolated a series of parameter vectors, the final stage in the process is to classify these as corresponding to a recognised defect type, or as normal material. The techniques mentioned here have included rule-based, statistical and neural network systems. In effect any of these will constitute the *artificial intelligence* component of the machine vision process, and theory relating to this subsystem is perhaps the less well-established and most debated of all. It appears that in many cases a rule-based approach is impractical, and whilst the functionalities of statistical and neural systems are broadly comparable, since both perform what is in effect pattern-matching, neural systems do seem to have some advantages, although there is no clear concensus as to what these are. They may include fault tolerance, ability to learn by example without a requirement for initial human analysis of the classification problem and classification speed. Neural systems are the least-understood of all the available classifiers, and show some promise in terms of performance. We conclude therefore that they are the most worthy of further research. The large computational effort involved in training appears to be a particular barrier to development in this area, also there appears to be only limited understanding of the internal representations formed by the network after training. We use these problems as the basis for our experimental work in

chapters 7 and 8.

2.6 Summary

Having introduced the particular defect detection application in view, and having examined a cross-section of contemporary approaches to similar problems, we will in the next chapter review the available types of general-purpose computer and recommend an appropriate selection. As we have already mentioned, data communications are of particular interest here since the applications in view are highly data-intensive. For this reason we have, in addition, reviewed contemporary data communications techniques and entitled the chapter accordingly.

Chapter 3

Data Communications

3.1 Introduction to Networking

3.1.1 Computers and Telecommunications

If one uses the term in its most general sense, then *computing engines* of one kind or another have been with mankind for what might be regarded as a surprisingly long time. The forms of technology applicable to computation have changed from Age to Age, and consequently the devices themselves have correspondingly changed their appearance and architecture. The Ancient Greeks, who established the basis for much of modern Mathematics, were extremely adept at working metals, and they employed these skills, together with their understanding of mechanics and geometry, to construct simple mechanical calculating devices. In the Modern Age, however, the doctrine of electronics, unknown to the Ancients, has come to the fore and it is his acquired skill in the relevant materials technology, rather than any significant developments in philosophy or novel modes of thought, which has enabled Modern Man to produce the fantastically complex computer systems in use in the world today.

The development of the thermionic valve by Fleming et al. shortly before the Great War pointed the way to the first electronic computing systems. A good example of an early such computer is the Colossos device in use by the Allied powers to facilitate cracking of the German "Enigma" code system towards the end of the second World War. A machine of this complexity required a great

many switching elements. This meant that the system was physically very large, the heater coils in each valve together generated a great deal of heat, the system as a whole consumed a great deal of electrical power and yet required a small army of technicians to maintain, since the lifetime of each tube was measurable in months or weeks. These factors meant that there was no question of any use being made of the system by anybody anywhere except inside the rather sizeable room which housed it.

Pioneering work by Shockley, Brittain and Bardeen made the first practical silicon devices available in the early 1950s. The technology associated with combining many such transistors onto a single piece of silicon was responsible for an order of magnitude increase in the switch density, and hence power and sophistication of computing engines of the time. Magnetic core store memory made bulk fast-access information storage relatively cheap and practical, and this allowed such systems to be used for the first time on a wide range of bulk information-handling problems such as company payrolling and accounting, for example. These *mainframe* computers of the 60s and early 70s still had a large power requirement and generated great quantities of heat, and in general input/output made use of *peripherals* which could be most conveniently located in the same large, air-conditioned room, since each would typically require its own hard-wired connection to the processor. Examples of such peripherals included paper tape stations, card punch/readers and magnetic tape, disk or drum drives.

At this point, one development played a particularly significant part in the emergence of *data communications* as a field of technological study in its own right, this being the introduction of the *microcomputer* or *microprocessor*. Further enhancement in materials science had made it possible to increase component density to a level where it became feasible to fabricate a complete processor on a single piece of silicon, and indeed the microprocessor's most striking feature is its high degree of sophistication and complexity, given its small size. However, arguably the most significant features from a practical point of view come about as a secondary result, these being namely the micro's tiny power consumption,

high reliability and, above all else, relatively *low cost.* The micro enabled small amounts of computing power to be cheaply incorporated into portable equipment, and this allowed the concept of a data *terminal* to be realised, such a device being in effect a computer in its own right, but one whose functionality was oriented solely around communication with a central *host* machine which would carry out all the required application processing.

Equipped with many such terminals, the usefulness of a mainframe computer increased dramatically. No longer would a *job* need to be punched by a programmer onto cards or tape, sent to the computer room, fed into the machine in *batch mode* whose results would likewise be punched onto cards or tape and returned, giving a turnaround typically of the order of hours or even days. The terminal allowed *interactive* use of the mainframe by means of *time-sharing* between users, giving a virtually instant turnaround and enabling a whole new range of computer applications for which this interactivity was an essential component. In such a system the concept of a *data communications channel* first begins to emerge, that is, a medium which transfers information in some standard symbolic form between two "intelligent" entities, in this case the mainframe and terminal. Superficially the result appears to be the same as before - the terminal is simply another type of peripheral to be located inside the computer room and hard-wired to the processor. However, the presence of a pseudo-intelligence at each end of the link makes it possible for the terminal to be made physically remote, communication taking place either via dedicated pilots, or through the telephone system or other network.

More recent developments have brought various types of *personal* computer into the marketplace. These machines are almost invariably based upon a single microprocessor, and are characterised by their comparatively low cost and their *single-user* mode of operation. Such systems are particularly useful for applications which are input/output intensive, yet relatively undemanding in terms of their processor utilisation, for example, word processing and data visualisation. Personal computers can also be used in the rôle of data terminal to a more

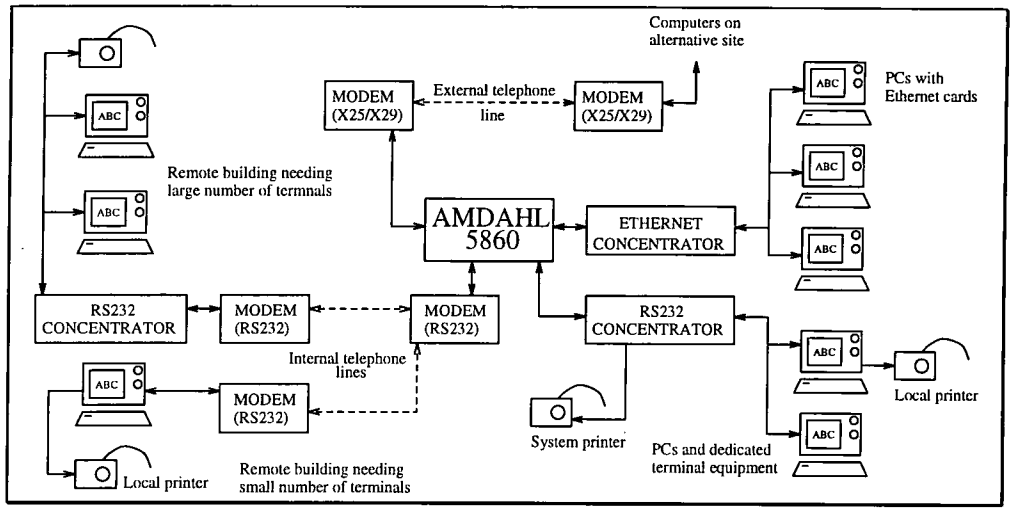


Figure 2: Schematic diagram of star-shaped data network

powerful time-sharing computer, but have greatly enhanced capabilities for manipulating data *offline* when compared with the earlier terminal devices. In the type of system which has many such PCs accessing the more powerful processing and software resources of the central mainframe, the telecommunications devices involved become increasingly complex. Convenience dictates that facilities for remote access to the time-sharing machine be provided, since users may be spread throughout many buildings over a wide area. Such facilities may be said to constitute a *data network*, a means of electronic communication between user and resource, that is, between client and host.

The mainframe may be connected to other, similar systems elsewhere to allow network services such as electronic mail, however, in other respects the layout is essentially a *star-shaped* network. The University of Durham operated such a system until Summer 1990, when facilities were modernised, and it is illustrative to examine this as an example - a schematic diagram is shown as figure 2. Where facilities for only a small number of simultaneous connections are required in a remote location, these can be provided by making use of existing low-bandwidth internal telephone pilots. Where facilities for a large number of simultaneous connections within one building are required, it may be more economic to use a *concentrator* in that building which shares its high-bandwidth link to the host or

hosts between many users.

The last five years have seen the emergence of yet another class of computer system, the *workstation*, which possesses some features normally associated with both of the pre-existing classes, the mainframe and the PC. Examples of these similarities are, respectively, the workstation's ability to serve multiple users at the same time (time-sharing) and its moderate cost, which in some cases allows the workstation to be installed on a one-per-desk basis. As with the PC, a workstation is typically oriented around a microprocessor, although with higher-specification models it is not uncommon for many micros to be employed in parallel inside a single unit. Further developments in microprocessor technology have here allowed the computing power of what might previously have been regarded as a supercomputer to be compressed into a convenient desk-sized package with a reasonably low power requirement and little need for maintenance.

Networking is a particularly important concept in the context of workstation systems which are typically organised into *clusters* of machines, interconnected by links which, compared with the older star-shaped, principally text-only arrangement just described, are relatively high in bandwidth. Such links may be based, for example, at the physical level on 10 Mbps[1] Ethernet or token ring schemes. Whereas in the previous example all communication involved the mainframe as one end-point, the newer type of network is oriented around providing for connections between *any* two nodes. This is frequently termed a *peer-to-peer* data link.

The increased available bandwidth and capability for arbitrary connection topology in such modern networks means that system facilities which are either inherently communication-intensive, or else require *routing* of data to arbitrary nodes, can be provided. Examples are electronic mail, network file systems, remote printing, remote task execution, network-oriented window environments and network graphics applications.

This personal, one-per-desk networked computer solution is also attractive in

[1] 10^6 bits per second

a number of other ways. For example, the fact that processing power is *distributed* rather than *centralised* may reflect more closely the management structure of the organisation in which the cluster exists. This can be an advantage, since it may make it possible for a user, or group of users, to take responsibility for the maintenance and configuration of the machines that he, she or they use on a day to day basis. By contrast, the previously described star-shaped mainframe network required a central administrative body to look after it, which attempted to provide, in terms of facilities, all things to all people. At the same time, high-bandwidth network links allow a good balance to be struck in the latter configuration between self-sufficiency and co-operation, since magnetic disk storage, printers and software can conveniently be shared.

It might be considered that the workstation cluster type of system is in fact so desirable as to ultimately mean the end of the line for the mainframe, with its associated air-conditioning and cooling systems, three-phase power supply and team of operators. Indeed, it is instructive to consider, in this age of miniaturisation, exactly what it is about the mainframe style of machine which necessitates such environmental support. The nub of the problem is that, in order to construct a time-sharing computer which can provide many tens of users simultaneously with a significant processing resource, very fast circuitry is required at the gate level. The earliest, discrete silicon logic made use of DTL[2], and it was the reduction in power dissipation as heat per gate brought about by the introduction of TTL[3] which made the first small-scale integrated silicon packages possible. Further breakthroughs such as LSTTL[4] enabled the gate propagation delay/power dissipation product to be further reduced, facilitating greater and greater scales of integration. Subsequently CMOS[5] fabrication technology allowed gates to be

[2]Diode-transitor logic, in which a diode built-in potential is used to affect the base-bias conditions, and hence conductivity, of the gate transistor

[3]Transistor-transistor logic

[4]Low-power Schottky transistor-transistor logic

[5]Composite metal-oxide semiconductor

built in silicon using FETs[6] rather than BJTs[7], the advantage being that, when in a quiescent[8] state, such gates consume only the infinitesimal power attributable to current leakage through the FETs' insulating gates.

However, it now seems that for the forseeable future at least, the state of the Art in computing power will always require a great deal of "environmental support" that cannot be catered for in a convenient desk-top package.

Time-shared computing facilities still make economic sense for many kinds of organisation. In general it is desirable to have the computer solve any given problem as quickly as possible. However, the rate at which any research group or department can:-

1. Perceive applicable problems
2. Devise systematic solution techniques for these problems
3. Write implementations of these techniques using a programming language appropriate to the machine involved

is likely to be many times slower than the rate at which the computer can run the implementations produced in (3). Therefore it makes sense for many such groups or departments to pool resources in order to provide a shared facility. The consequent increased affordable cost makes available to all a more powerful resource than could be justified by each individual contributant; and so everybody involved gets their problems processed much more quickly as a result of the co-operation.

Technological advances over the years have brought increasingly powerful computing resources within the reach of many organisations. It is clear that improvements in processing speed alone can make for great time savings where the computation involved is processor-intensive. However, it can be seen that the

[6]Field-effect transistor
[7]Bipolar junction transistor
[8]Non-switching

performance upgrade may be compromised for general problems if these additionally make significant demands on the system's communications capabilities, unless those capabilities are also expanded.

Networking itself has evolved into a highly-complex field of expertise, and we should now like to provide an introduction to the subject, since this is vital to an understanding of our work on multi-processor parallel distributed applications in chapter 8. We first discuss the *Internet*, a global data network which has been largely responsible for the fostering of the TCP/IP protocols which we shall later employ.

3.1.2 The Internet

The Internet is a collection of smaller networks which have largely evolved independently, and which are now interconnected. These include the American ARPAnet[9], NSFnet[10], sections of the British JANET[11], a number of military networks and various other local networks at University and research institutions elsewhere around the world. Users can send messages from nodes on any of these to any other, except where there are security or other policy restrictions on access. The University of Durham has been connected to the Internet quite recently.

Standard facilities are supported by the Internet, each site has a subset of these available depending on requirements and hardware. These include:-

- File transfer protocol (FTP). This allows a user on any machine to get files from, or send files to, any other machine. Security is handled by requiring the user to specify a username and password for the remote computer. Access is then granted to all files which that user ID would normally have access to if logged on directly. This is not quite the same as a "network file system" as described below - FTP is a utility which is run every time access is required to a file on a remote system. The user *copies* the file to

[9] Advanced Research Project Agency
[10] National Science Foundation
[11] Joint Academic Network

his/her local system, and then works with the local copy. It is functionally very similar to Kermit, which transferred files using the X25 protocol over a serial connection.

- Network terminal protocol (TELNET). Using this a user on any machine in the network may log in to any other. A remote session is started by invoking telnet and specifying the name of the remote computer. Generally the connection behaves very much like a dialup connection, in that the host will usually prompt for a user ID and password.

- Electronic mail (SMTP[12]). This allows the user to send and receive messages to and from specific users at remote sites. In general a message is prepared using a package on the local machine. When the command to send is given, this package will *spool* the message onto a local queue. At some appropriate time, a local process will connect to a remote "mail-hub" system and transmit the queue of messages. This system will subsequently contact a delivery process on each machine to which messages have been addressed, and transmit the appropriate data. Finally, each individual delivery process will place the messages in the users' "mailboxes" which are files on the target machines.

- Network-oriented Window Systems. Until recently, high-performance graphics programs had to execute on a computer which had a bit-mapped graphics display directly attached to it. This meant in effect that processor- intensive software, which required the use of a time-shared mainframe computer, could only produce graphics in a *batch-oriented* mode, since input/output from such machines had hitherto been very low-bandwidth. In other words, the program would be left to run for some time, and eventually a page or pages of graphics output would be created. Interactive graphics and windowing environments were previously only possible using non-sharable

[12]Simple Mail Transfer Protocol

low-performance devices such as PCs. Network window systems allow a program to use a display on a different computer. Full-scale systems[13] provide an interface that lets the user distribute jobs to the systems that are best suited to handle them, but still gives a single graphically-based user interface.

- Name Servers. In large installations, there are a number of different collections of names that have to be managed. This includes users and their passwords, names and network addresses for computers and so on. It becomes very tedious to keep a local copy of this information up to date on every single machine on the network. Thus the databases are kept on a small number of systems. Other systems access the data over the network.

- Network File Systems (NFS). This allows a computer to access files on another host in a more closely integrated fashion than does FTP. A network file system gives the illusion that disks or other devices from a remote machine are directly connected locally. This capability is useful for several different purposes. Large disks can be attached just to a few computers, but others can still be given access to the space. Apart from the obvious economic benefits, this allows people working on several computers to share common files. Some manufacturers offer "diskless" workstations, which have no local storage at all, relying totally on NFS access from a remote server.

- Talk. Here a connection may be made between two users on different machines. What each of them type is transferred to the screen of the other, and so the connection behaves a little like a telephone call.

Other commonly-used protocols tend to be facilities for getting information of some kind from a remote system. Some examples are:-

[13]The most widely-implemented window system is X.

- RDATE - obtain time of day and date. This is useful for network- connected PCs which may execute RDATE while booting. This may be desirable because the PC may not maintain the time and date while switched off, or else this time-keeping may be battery-backed and unreliable.

- RUSERS - display which users are logged in to remote machines. This works in two modes. Either a particular remote system is interrogated, or else a broadcast message is sent - all machines receiving it will respond with the appropriate information. In general, inter-site gateways are programmed not to propagate RUSERS broadcast messages. Thus only machines local to that which sourced the request tend to reply.

- FINGER - display information about a specific user on a remote machine.

This widespread, international network is based on a system of protocols which are perhaps most accurately referred to as the "Internet Protocol Suite". TCP[14] and IP[15] are two of the protocols in this suite. Because these are the best known, it has become common to use the term TCP/IP to describe the whole family. This leads to some problems, for example, Sun Microsystems' NFS and PC-NFS are sometimes said to be based on TCP/IP. In fact an alternative protocol, UDP[16], is used instead of TCP. However, the habit has become well-established.

3.1.3 Network Protocols: TCP/IP

TCP/IP is a layered set of protocols. They take care of the low-level data manipulation required for facilities such as those above. Consider the situation of sending mail as an example. Firstly, there is a protocol for mail (SMTP). This defines a set of commands which one machine can send to another, which specify who the sender of the message is, who it is being addressed to, and the text of the message. However, this protocol assumes that there is a way to communicate

[14]Transport Control Protocol
[15]Internet Protocol
[16]User Datagram Protocol

reliably between the two computers. Like many other application protocols, mail simply defines a set of commands and messages to be sent. It is designed to be used together with TCP and IP.

TCP is responsible for making sure that the data gets through to the other end. It breaks data up into "datagrams" which are a convenient size for transmission. It keeps track of what has been sent, and retransmits anything that did not get through. It passes data for transmission to the IP protocol below it. It passes data which it has received from the IP protocol to the application layer above it. TCP is a *connection-oriented* protocol. That is, functions are provided for establishing a connection and closing it down again. Data transmission is reliable and guaranteed to be correctly sequenced with no repeats.

IP is responsible for the actual routing of datagrams. It is a *connectionless* protocol. That is, the routing of each datagram is considered separately. There is no concept of a connection existing between the two machines at the IP level. Data transmission at the IP level is unreliable. A datagram may arrive before another datagram which was sent previously. Multiple copies of datagrams may be received.

Beneath IP is a physical protocol layer. The medium may be Ethernet, fibre-optic cable or UHF satellite link, for example. The protocol could be X25 in any of these cases. Note that there is a distinction here between "datagram" and "packet" which often seem almost interchangeable. A packet is a quantity of data at the physical level, and often there are efficiency advantages associated with sending one datagram per packet, so the distinction vanishes. However, when TCP/IP is used on top of X25, the interface breaks datagrams up into 128-byte packets, and reassembles them at the other end before handing them back to IP.

TCP/IP is based on the "catenet model". This model assumes that there are a large number of independent networks connected together by gateways. The user should be able to access computers or other resources on any of these networks. Datagrams will often pass through a dozen different networks before

getting to their final destination. The routing needed to accomplish this should be completely invisible to the user. All he/she needs to know in order to access another system is an *Internet address.* This looks like 129.234.200.116. It is in fact a 32-bit number, but it is normally written as four decimal numbers, each representing eight bits of the address.

3.1.4 Sockets

A socket is an endpoint of communication. It is the point of interface between an application program and the underlying transport protocols such as TCP/IP.

A socket manages the flow of data between an application, or process, running on one machine, and other processes running on machines somewhere else on the network. It is a logical descriptor which may be treated very much like a file handle by the application, once it has been created and has made successful contact with another socket owned by the remote process. By use of the protocols described, a socket data stream provides sequenced, reliable full duplex communication.

A machine connected to the Internet may be uniquely specified by use of the appropriate 32-bit address. However, many processes using socket-type communication may be running concurrently, and any one process may have as many individual connections open as required. Therefore it is necessary for a socket to be associated with an *address* unique among all sockets open on that machine.

Addresses may be assigned:-

- Explicitly. The application may request that a specific address be given to a socket which it has created. A disadvantage here is that the request will be denied if the address specified is already in use by another process.
- By the operating system. The request is sure to succeed - the address will be the next available one in sequence.

Sockets may be opened:-

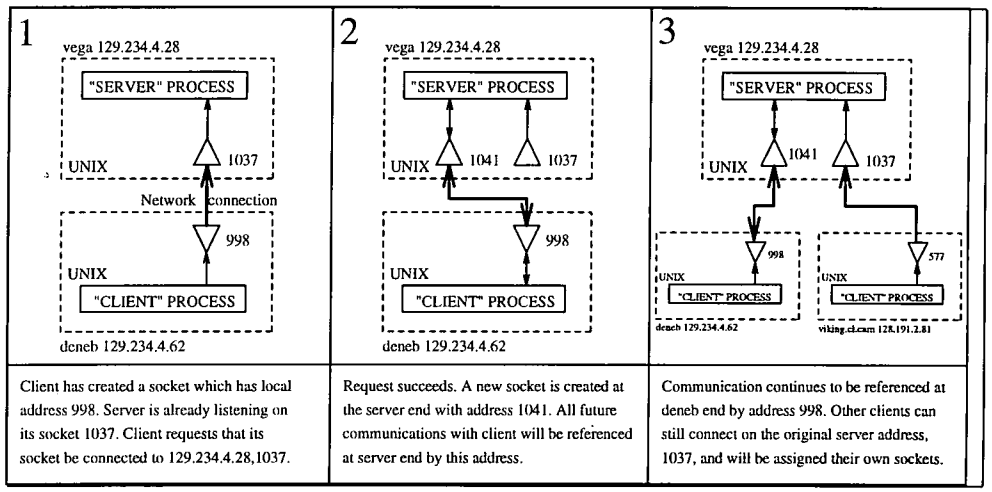


Figure 3: Socket behaviour during connection requests

- In "listening" mode. Here the endpoint waits passively after creation for a connection request. This is often the behaviour required of "servers" which are making some resource available to their "clients" upon request. When a connection is made, a new socket is generally automatically created with a new address, in order that the original "listening" socket may be used to accept further connections. This procedure is invisible as far as the "client" application is concerned. An illustration of this procedure is shown as figure 3.

- In "connection-seeking" mode. After creation, the socket attempts to connect to a remote socket determined by an Internet address plus socket address, both of which must be provided by the application. The remote endpoint must be a "listening" socket.

It can be seen that connections can only be established between pairs of one "listening" socket and one "connection-seeking" socket. However, after communications are established, the distinction between the two vanishes, and either end can then take the initiative to send data which will be buffered at the remote end awaiting a "read" instruction from the application.

Problems associated with socket-based communication are:-

- The "connection-seeking" application has to have knowledge of the *socket*

address of the remote endpoint. System programs achieve this by using well-defined, reserved addresses associated with their function. For example, a *TELNET* program will usually attempt to connect to the specified machine on socket number 23. A *FTP* program will connect on socket 21. A Unix machine which is available for remote login will listen on socket 23 for connections. Many different users may log in in this way since a new socket is created for each at the "server" end as described.

User programs may also use this procedure, and the required address may be hard-coded into both "server" and "client" applications. However, there is a risk that the "server" may fail to obtain this address when it starts up, if it is already in use. Because non-explicit address allocation is sequential, all non-reserved addresses will be allocated in this way from time to time. The "server" may instead use non-explicit address allocation, in which case it is guaranteed a problem-free start-up. However, the problem of communicating the address to the "client" remains, this must be achieved by some other means in this case.

- Sockets can only transmit *opaque data*. That is, only strings of eight-bit bytes may be communicated. There is no conception of what constitutes a floating point number or a long integer, for example. This may not be a problem if the two machines at either end of the link are similar in architecture and have applications built by the same compiler, since the internal representation of such numbers will be the same at each end. However, if the architectures are different, for example in the case of a PC communicating with a Sun SPARC workstation, or if different compilers have been employed, then the byte-order of the internal representation may be different, or it may even consist of a different number of bytes. Socket-based communication gives no help with this problem. Bytes are faithfully delivered, but fitting them into program variables is left entirely to the programmer and his/her application.

3.1.5 Remote Procedure Calls

Both of the problems outlined concerning sockets may be solved by resorting to a higher-level protocol which uses specialised procedure calls to hide some of the details of the underlying network. This is called the *Remote Procedure Call* or RPC library.

With RPC, the client makes a procedure call that sends service request packets to the server as necessary. When these packets arrive, the server initiates a dispatch routine, which performs the requested service and sends back a reply. The procedure call then returns to the client. The client application does not need to know about the existence of the underlying network, or how that network functions - it simply calls a procedure, just as it would call *malloc()*.

Data is passed from client application to server application in the form of parameters to the remote procedure call. After completion, the call will generally return a user-definable data structure to the client. This provides for communication in the reverse direction.

The client's RPC call needs an IP address[17] which identifies the machine running the server process. At the level beneath the RPC protocol, socket-based communication is still taking place. However, a socket address is not needed to identify the desired endpoint on the remote machine. Instead, a 36-bit *program number* is used, together with a version number and a procedure number.

The program number specifies a group of related remote procedures, each of which has a different procedure number. Each program also has a version number so that a new program number does not have to be assigned when a minor change is made.

Internet facilities for acquiring information about remote systems such as those listed above tend to be RPC-based. To allow an application to make use of information gleaned from the RUSERS command, for example, which finds the names of users logged in on a specific machine, it would be necessary to obtain

[17]Internet address

the appropriate program, version and procedure numbers from a reference source.

It can be seen that RPC-based communication differs from the socket-based in two important respects:-

- A client can identify the desired server process using an address which is unique to that process. This address is a combination of the program, version and procedure numbers as outlined. In this way, the task of selecting a suitable address for a server application, which was a problem when using socket-based IPC[18], is made straightforward.

  This is made possible by a network daemon[19] process which has two parts to its functionality, these are called the *rpcbinder* and the *portmapper*. Server applications which want to receive remote procedure calls first register with the rpcbinder, which stores the program, version and procedure numbers in a map. A logical port (socket) is then allocated to the application. Client applications query the portmapper running on the machine which is host to the desired server process, quoting the program, version and procedure numbers, and receiving in return a socket address at which the server may directly be contacted. High-level RPC calls encapsulate this query as part of an RPC message-creation call, making contact with the portmapper transparent. An illustration of this procedure is given as figure 4.

  The rpcbinder knows the address of all RPC server programs that register with it in advance. If a program does not register, it cannot receive RPC messages. Likewise, the socket address of the rpcbinder must be known to all applications that want to register with it. On NFS networks, this address is port 111 on every server machine.

- RPC calls can handle the transmission of arbitrary data structures in both

[18] Inter-process communication

[19] A background task which carries out some kind of system administration function

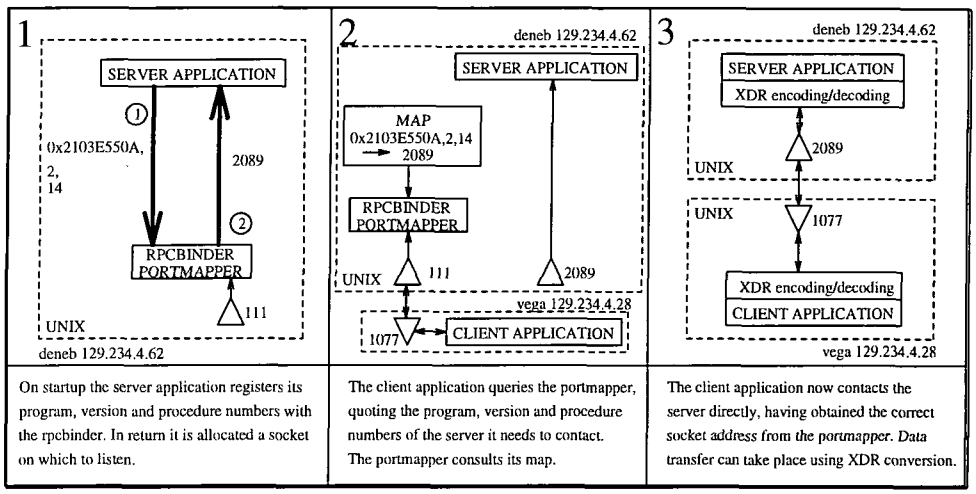


Figure 4: Procedure for typical RPC call

the client to server and vice-versa directions, regardless of different machines' byte orders or structure layout conventions. This is possible because the structures are always converted to a network standard called XDR[20] before they are sent. Once received by the target process, the XDR-encoded data is converted again to the appropriate format for the target machine. This activity is transparent as far as the programmer is concerned.

The additional functionality described means that RPC communication is free from both of the problems mentioned concerning socket-based communication. However, some performance may be sacrificed:-

- A client application must establish two network connections for every RPC call made, firstly with the portmapper, to determine a suitable address on which to contact the server process, and secondly with the server process itself. The client side cannot "remember" the address from one call to the next - indeed, there is no reason why the address may not change inbetween times. Extra waiting time is therefore introduced both because of the mechanics of establishing a second connection, and because it may take the portmapper some time to service the map look-up request if there are many clients competing for its attention. For systems in which there is extensive

[20]External data representation

"dialogue" between server and client, necessitating the issue of many RPC calls, this additional overhead may mean that RPC communication gives unacceptably poor performance.

- Data structures passing between the client and server are always converted to the XDR format. Where both client and server applications run on machines of similar architectures, it may be the case that the internal representation of variables is exactly the same at both ends. In this case, the time spent at each end converting the data to and from the XDR format is wasted. This may be a large unnecessary overhead if the volume of data being transmitted is large.

It can therefore be concluded that in general, RPC calls, by virtue of their hidden functionality, offer a relatively simple and problem-free interface to the programmer. However, in certain circumstances acceptable performance may only be obtained using socket-based communication.

We now conduct a detailed performance analysis of four distinct member protocols from the Internet TCP/IP family, these being the user datagram protocol (UDP), transmission control protocol (TCP), the remote procedure call system (RPC) and data transmission aspects of the network file system (NFS). We discuss the trends in computer development which have led to the widespread use of distributed workstation environments interconnected by local area networks, the motivation for implementing parallel distributed programs on such systems and the impact that protocol- selection can have on the ultimate efficiency of such an application.

3.2 Comparison with Previous Work

We have referenced various publications which have dealt with multi-workstation parallel distributed applications from the standpoint of particular computational

problems and/or specific task allocation paradigms. We have addressed ourselves to the issues of task organisation such that the quantity of communication required between nodes can be optimised. The novel aspect of our own work therefore lies in our focus on optimisation of the communication mechanisms themselves. This has not yet been dealt with in detail in the context outlined.

3.3 Trends in Computing Hardware Solutions

Recent years have seen a decline in popularity of the mainframe and minicomputer systems previously favoured for processor-intensive applications. This is due in part to the emergence of a new class of computer, the workstation, which combines the significant processing resource typical of these systems with the graphical interface capabilities of a desktop personal computer. Magee et al. [18] suggest that the main advantages of a distributed workstation-based resource over a centralised system are improved value for money and predictability of response time, as well as the provision of improved input/output facilities. However, a counter-trend is also observable; the reduction in cost and enhanced reliability of high-bandwidth telecommunications mean that centralised facilities still make economic sense for many organisations. Nonetheless, workstation clusters are now commonplace in many research, and other, institutions.

3.4 Application Speed-up Through Parallelisation

In many cases the individual workstations in such a facility spend much of their time idle, since they are used in the main for reading electronic mail, editing files and so forth. The machines are typically interconnected by a local area network system such as Ethernet, and it therefore seems reasonable that one should be

able to recover this lost CPU resource by putting it to work on a highly compute-intensive application.

In effect this requires the *parallelisation* of the application in view. Programming even *dedicated* parallel architectures is notoriously difficult, but parallel workstations introduce the extra difficulties of heterogeneity and a constantly-changing load situation due to tasks run by other users [19]. Nonetheless, many attempts have been made to speed up processor-intensive applications in this way, examples include molecular dynamics simulation[20], solution of partial differential equations[21] and sparse matrix factorisation[22]. Ready-written software libraries[21] are available both as freeware and as commercial products. These aim to allow the scientific programmer to write parallel applications without having to learn specialist network programming techniques. As an alternative, some researchers have achieved communication between workstation hosts through the simple expedient of reading and writing shared files in a network file system (NFS).

Performance and efficiency of the parallelisation process are key issues in all these cases, a frequently-quoted metric being effective processing power vs. number of hosts employed. If this is defined as the speed-up, $S(n)$, with n the number of hosts, then Amdahl's law[23] may be used to express a relationship with W_s, the quantity of work in the application which must be performed sequentially, and W_p, the quantity of work which is amenable to parallelisation, thus:-

$$S(n) = \frac{W_s + W_p}{W_s + \frac{W_p}{n}} \tag{1}$$

This consideration dictates that optimum performance will be achieved in the form of *linear* speed-up where *all* of the application processing can be parallelised:-

$$\lim_{W_s \to 0} S(n) = n \tag{2}$$

[21] Titles available include Parallel Virtual Machine (PVM), Linda and P4.

However, in order to accurately model the specific case of workstations interconnected by local area networks, this simple analysis needs to be extended to include the extra processing required by networking protocols, which we will call $P_{proto}(n)$. For simplicity, transmission time is assumed to be a component of this term; this is permissible since computation $\propto$ time for a system of constant computational power. Furthermore, there is in some cases a reduction in total processing which can be brought about through distribution of the application on many nodes. For example, a highly memory-intensive task running on a machine with limited physical memory will necessitate extra processing in the form of virtual memory swapping. When distributed over many nodes, however, the quantity of physical memory present in the system as a whole is increased, which can lead to a *superlinear* speed-up characteristic. We will denote this performance *bonus* as $P_{distrib}(n)$.

$$\rightarrow S(n) = \frac{W_s + W_p}{W_s + P_{proto}(n) - P_{distrib}(n) + \frac{W_p}{n}} \tag{3}$$

The form of $P_{distrib}(n)$ and the value of W_s are highly application-specific, and in our first-order consideration of the general distributed-application case we will assume:-

$$W_s = 0, P_{distrib}(n) = 0 \forall n \tag{4}$$

For parallelisation to be worthwhile, we require the speed-up to increase as n increases, thus:-

$$S(n+1) > S(n) \forall n \tag{5}$$

$$\rightarrow P_{proto}(n) + \frac{W_p}{n} > P_{proto}(n+1) + \frac{W_p}{n+1} \tag{6}$$

From equation 6 it is clear that the protocol overhead characteristic is of crucial importance to the ultimate success of the distribution process. In other words, if the extra processing necessitated by network protocols on moving from n

to $n+1$ hosts exceeds the saving made through dividing the application workload, then the transition is not worthwhile.

Furthermore, if the speed-up characteristic *is* sublinear, as is the case for most practical implementations, then a distributed application will consume a greater *total* quantity of resources than would the same task executed on a single node. Thus there must be a trade-off between enhanced overall execution speed and the increase in total CPU time required to achieve it.

We will therefore examine the performance of four distinct communication protocols, these measures are of interest since they are directly linked to $P_{proto}(n)$ for any distributed application making use of them. The four include the NFS protocol mentioned above, as well as the Remote Procedure Call (RPC) system commonly used by ready-written networking tools.

3.5 The Internet Protocols

The interconnection of former military, research and commercial networks known compositely as the *Internet* is the largest peer-to-peer data network in the world, consequently the suite of protocols on which it is based is the most significant of those currently in use. The member protocols which we have examined can be outlined as follows:-

3.5.1 User Datagram Protocol (UDP)

UDP [24] is a protocol concerned with the routing of data packets using the minimum overhead required to transfer data onto a physical medium, whilst maintaining a convenient standard interface to the application in the form of 16-bit socket addresses. UDP does not deliver data reliably, that is, the specification allows for the protocol to simply discard data under certain circumstances. It does however incorporate a 16-bit checksum, badly-checksummed data being dropped as described, therefore any data which *is* delivered is guaranteed not to

be corrupted. Datagrams may be duplicated or delivered out-of-sequence. UDP will not fragment data, therefore the application has responsibility for dividing data up into blocks suitably-sized for the physical medium, as well as initiating retransmission of lost data if required. Within our own computer system, based on SunOS 4.1.2 and Ethernet, the maximum UDP block length is 9,000 bytes.

3.5.2 Transmission Control Protocol (TCP)

TCP [25] is the *reliable* equivalent of UDP. It uses the same style of 16-bit socket addresses to differentiate logical communication end-points on the same host, although the UDP and TCP address spaces are distinct. Communication is guaranteed free of corruption, in-sequence and reliable, since TCP incorporates its own error-checking and retransmission generation procedures. Very few assumptions are made about the reliability of protocols and hardware underneath the TCP layer. TCP will automatically fragment data into convenient packets as appropriate. There is no limit in principle to the amount of data that can be buffered for transmission in a single operation, however, in practice there is an adjustable buffer-size limit known as the *high water mark*. Data in excess of this limit will be refused until space becomes available.

3.5.3 Remote Procedure Calls (RPC)

RPC is an additional, higher-level protocol layer which can be used in conjunction with either TCP or UDP. Its use overcomes two problems:-

1. The 16-bit address space of TCP and UDP is, in practice, rather restrictive. Although well-established protocols such as *telnet* or *ftp*[22] have standard, reserved addresses, there is no mechanism for reserving addresses for user programs. Applications must therefore either use dynamically-allocated addresses, in which case some alternative means must be found to communicate the address to the remote host, or else seek to use a fixed address,

[22] File transfer protocol

accepting that it may be already in use elsewhere on the system.

2. TCP and UDP transmit streams of *untyped* bytes. This can be problematic, for example, if floating-point numbers are being transmitted between different architectures, each with a different standard for storing them.

RPCs employ a 32-bit address space, which means that fixed addresses can reasonably be allocated to every RPC program. A central authority exists to guarantee the uniqueness of addresses allocated to applications registered with it. However, this expanded address space means that a *portmapper* process is required on each machine using RPCs, the function of which is to convert the 32-bit RPC address into a dynamically-allocated 16-bit TCP or UDP address. Clearly this will increase the set-up latency for any RPC-mediated communication.

The RPC protocol converts all data to a network-standard format known as External Data Representation (XDR) before transmission, converting back to native format at the remote end. Although this means that an application using RPCs will not have to carry out type-conversion, the procedure is not efficient in terms of CPU resource consumption. In the worse case, where two identical hosts are communicating, no type-conversion is necessary, although the RPC protocol will in any case carry one out at each end. In the better case, where two unmatched hosts communicate, a single type-conversion in each direction would be optimal.

3.5.4 Network File System (NFS)

The NFS protocol allows filesystems to be conveniently shared between hosts in a networked cluster. From the application programmer's point of view, a very straightforward method of implementing communication between hosts in a distributed-processing scheme is by use of shared files. However, this type of communication is extremely heavy in terms of protocol overhead, since it incorporates all of the inefficiencies of RPC, together with latency due to disk access

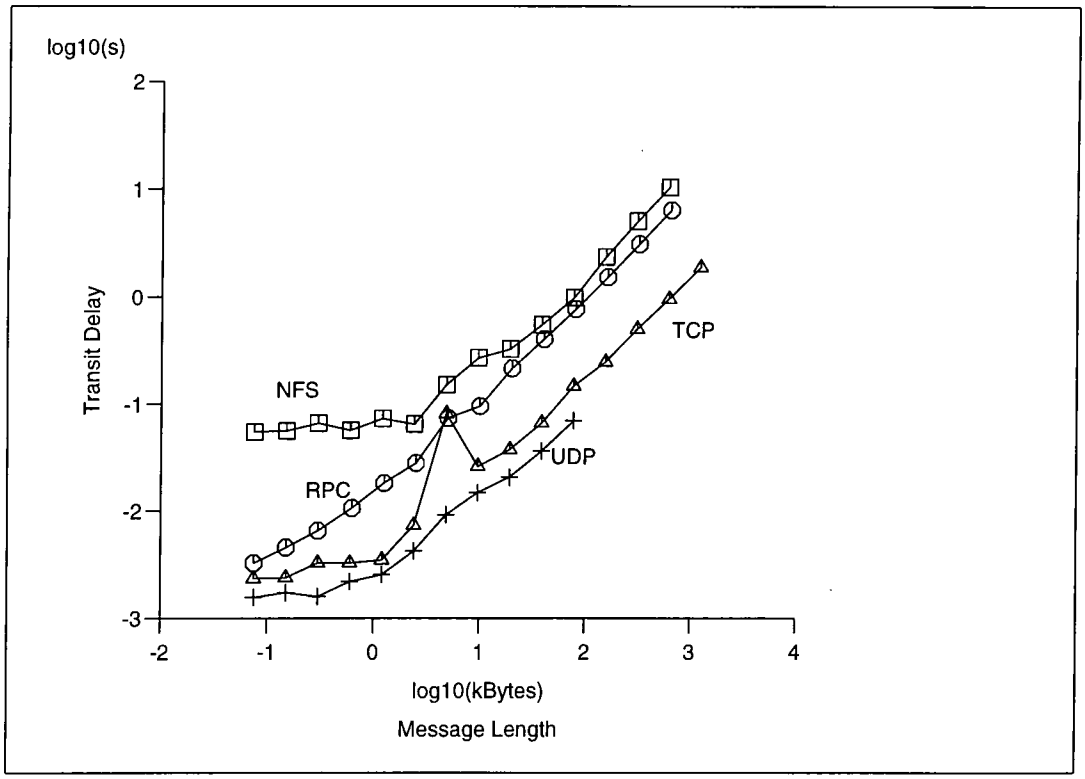


Figure 5: Analysis of protocol performance with log/log scale

contention and limited disk bandwidth, although these may be alleviated to some extent by intelligent disk-cacheing.

3.6 Experimental

The performance of each of the protocols described was investigated by measuring propagation delay as a function of message size. The results are shown as figures 5 and 6.

The experiments were conducted using a pair of Sun Microsystems IPC workstations running SunOS 4.1.2, interconnected using 10 MBit/s Ethernet. To avoid difficulties of clock synchronisation, all measurements were made by timing the double transit of data from host A to B and back again, this figure being divided by 2 to give an average time for a single transit.

The network interconnection used for the experiments was shared with intra-departmental traffic, although isolated from the campus backbone and the wider

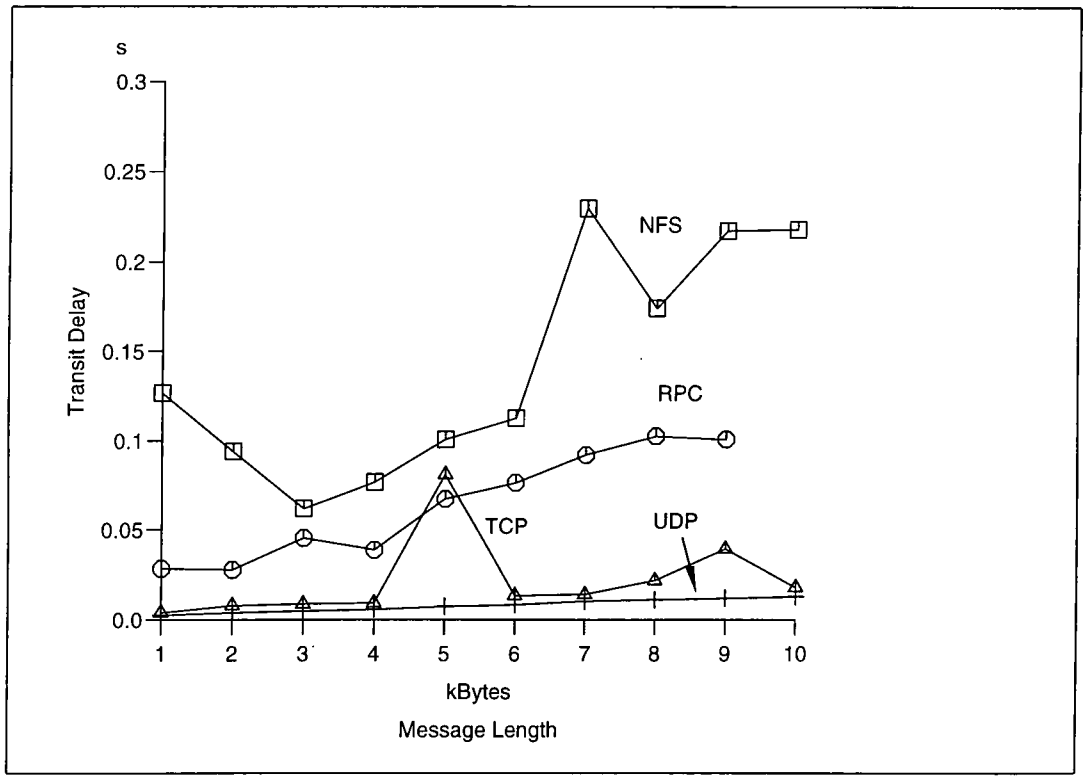


Figure 6: Analysis of protocol performance with linear scale

Internet by a bridge/router. Each data point shown represents the average of 10 trials made at random times through the course of a working day. The data therefore give a realistic impression of how the protocols can be expected to perform in a normal working scenario. This statistical treatment is required since the systems involved are inherently non-deterministic, as can be seen from figure 7, which shows the transit delay for a fixed-length TCP message sent at various times of day. A well-defined minimum latency can be observed here, which is often not achieved due to collision retries, or alternatively due to delays caused by the host B slave process being swapped-out at the point in time where a network message is received.

Protocol-specific experimental details are as follows:-

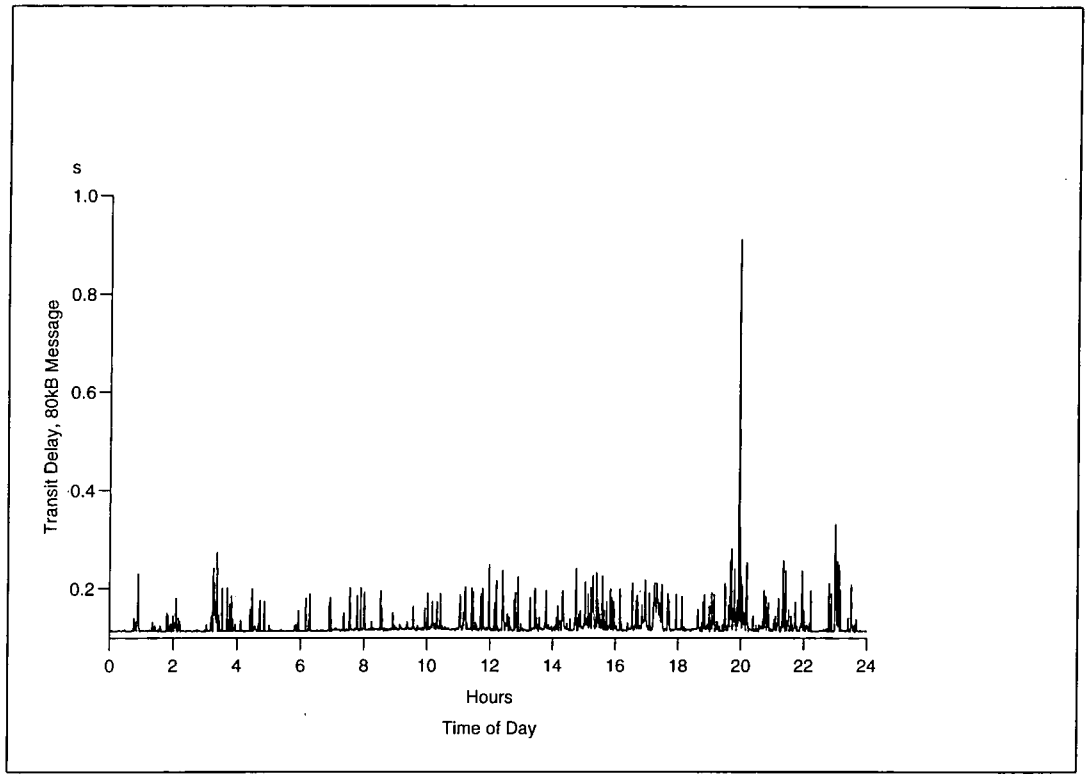


Figure 7: Ethernet transmission delay for 80kB TCP message

3.6.1 UDP Experimental Details

Timings for UDP transmissions were taken from the point where the application on host A queues the data to the point where the correct quantity of data is received by host A having been relayed by host B. An integrity check is subsequently made on the data before accepting the time as valid, but the processing required for this is not included in the measurement. The justification for this is that the packet duplication or out-of-sequence reception allowed by UDP is generally caused by network topology anomalies, for example, transitory duplicate routes. Clearly these may not occur where all transponding nodes are located on a linear Ethernet segment with no intervening routers.

Where data throughput is such that send or receive buffer sizes are exceeded, UDP is permitted to simply drop the overflowing packets as described. In our experiment this meant that contiguous messages longer than 76.8 kB could not be sent by UDP.

3.6.2 TCP Experimental Details

Unlike UDP, TCP is a *connection-oriented* protocol which means that a logical data connection has to be *established* before any data can be transferred. This small set-up overhead has not been included in measurements, this reflects the fact that such setting-up needs to be carried out only once by a distributed application and therefore does not have a real bearing on its performance. No checking of data integrity was carried out after transfer, since TCP guarantees this.

3.6.3 RPC Experimental Details

Measurements relating to the RPC protocol were carried out by registering an RPC service on host B, using host A to make calls to it. The data structures transmitted in each direction, the RPC *parameters* and *return values* were identical, consisting of a variably-sized array of integers. The structure definition was compiled-in and so recompilation of the experimental software was required in order to change the message size. The RPCs were configured to use TCP for transport since the use of UDP would have imposed a restriction of one datagram per call.

3.6.4 NFS Experimental Details

Two files shared by NFS were used for communication between the two hosts, one corresponding to transmission in each direction. Considerable effort was expended in making the NFS measurement application as efficient as possible in terms of communication performance, since considerably more flexibility is available to the programmer in this case compared with those previous. In the final assessment the option selected required the programs on host A and B to each request from the system a mandatory exclusive access lock on one of the shared files. The effect of such a lock is to cause other processes attempting access to *block* until such time as the lock is removed by the application holding it. In

effect this allows an interrupt-driven response to new data arriving in a shared file, and thus eliminates the inefficiencies and waste associated with polling such a file.

3.7 Analysis

Figure 5 shows timings for messages varying in size between 75 bytes and 1.2 Mbytes sent using the four different protocols.

It can be seen that both TCP and UDP have a timing characteristic which is independent of size for messages smaller than 9,000 bytes. The UDP application has control of data fragmentation and in our case is designed to make packets as large as possible in order to achieve maximum efficiency. Messages of the described size can be sent as a *single* packet, and we can therefore conclude that for UDP, Ethernet transmission time is insignificant compared with the processing time required to assemble the various packet headers, where packet size is small. By comparison with UDP we can say that TCP is adopting a similar fragmentation strategy even though this is not known *a priori* from the application specification. However, the fixed delay for TCP is somewhat longer, this is due to the extra integrity-checking performed.

The timing for TCP corresponding to a message size of 4.8 kBytes is almost one order of magnitude greater than that suggested by the trend of the surrounding data. This occurs since it is at this point that our measuring processes need to extend their data segments in order to accommodate the extra buffer space required by the increasing message length. This is carried out automatically by the protocol library functions, however, attention is required from the system memory management daemon, producing the delay. This feature could be eliminated by requesting from the system a sufficiently large data segment at application start-up time, however, this is not necessarily good practice since other active processes on the machine could themselves be delayed for want of physical memory in this case.

The characteristic for RPC remains almost linear over a very wide range of message sizes. This suggests that the processing required to perform data type-conversion to XDR and back to native format is a large overhead compared with that required to assemble and transmit packets. The reduction in gradient for small packet sizes is due to dominance of the overhead in exchanging address details with the RPC server (host B) machine's portmapper as processing $\propto$ message size is reduced. A one-time additional delay is again observed for a message size of 4.8 kBytes, again this is caused by data segment extension, expected since we selected TCP for our RPC transport protocol.

The results for NFS are somewhat noisier than for other protocols, since a greater range of factors can potentially cause delay, these include contention for disk access and waits for attention from the network file lock daemon which administers the file locks used for the interrupt-driven file access described. This is particularly well shown on the linear plot, figure 6. The characteristic has a size-independent region for small message sizes extending up to 2.4 kBytes at which point the processing $\propto$ message size begins to dominate. This reflects the comparatively heavyweight nature of the set-up processing. For large message sizes, the characteristic shows a larger amount of processing per unit data transmitted than is the case for RPC, this reflects the additional delay caused by limited disk bandwidth.

In the case of maximum performance disparity between the worst, NFS, and best, UDP, of these protocols, it can be seen that the difference corresponds approximately to one order of magnitude.

3.8 Conclusion

We conclude that the difference between performance of worst and best protocols is extremely significant, and that the impact of protocol selection on overall

distributed-application performance is correspondingly quite profound if the application is in any way communication-intensive. However, when choosing a communication technique from among those examined, one must balance against this the general rule that low-level, high-performance protocols are in general more difficult for applications programmers to use, requiring more specialist knowledge and possibly more processing at the application level in the form of type-conversion (UDP and TCP) and/or integrity checking (UDP) if required.

Chapter 4

Operating Systems

4.1 Introduction

We have already discussed the history of the computing engine with particular relevance to its hardware development in chapter 1, and to methods used for intercommunication in chapter 3. The emergence of the *operating system* as an essential component of the computing device would appear toward the end of either account, and it is the development thereof that we shall next examine, in order to make an informed judgement about the most appropriate operating system to use for a general machine vision problem, of which the one in view is a specific example.

The most basic high-level function of any operating system is to enhance the interaction of the user with the hardware. Early computers such as the Colossos system mentioned in chapter 3, although huge in terms of energy consumption and physical size, were conceptually sufficiently simple to make it feasible for users to design programs at the lowest level of symbolic instruction present in the machine, the *machine code* as it is known. By modern standards such software was typically modest in its functionality, limited as it was by the hardware performance of the target machine, and the task of *assembling* these instructions was therefore not unreasonably laborious. Today assembly code is still frequently written by the application designer, most often to illustrate to students the lowest level of computer operation as well as the advantages of using a high-level language, but also where performance is at a premium, especially if the system in

view is to be mass-produced - the microcontroller at the heart of a digital mobile telephone, for example, which runs signal-processing algorithms the efficiency of which is crucial to the performance of the telephone as a whole. However, the key problem with machine language or assembly code is that it is for general applications rather inefficient, in that the programmer will need to implement the same sub-tasks over and over again. By introducing a higher level of symbolic representation, in which each symbol represents a commonly-used sub-task, the programmer's productivity is boosted by an order of magnitude, and this therefore is the motivation for the development of the high-level language. As hardware capabilities grew and correspondingly more was demanded from the applications software, increasingly the basic machine instructions were written not by a human, but by the *back end* of a high-level language compiler.

Just as high-level languages enable commonly-requested tasks *within* the application to be implemented with a smaller amount of human effort, so an *operating system* makes the process of *using* and *developing* the application more efficient by providing high-level functions for the commonly-requested tasks involved. At first this meant a peripheral manager, application loader and possibly rudimentary file manipulation, but the functionality of the facilities now offered to support today's highly-sophisticated applications is considerably more extensive, as we shall see. The idea of this software *re-use* occurs again and again as the technology evolves, although it tends to do so at higher and higher levels of symbology.

4.2 Microsoft DOS

The first operating system which we have examined in our quest for the optimal machine vision environment is Microsoft's Disk Operating System, or MS-DOS as it is also commonly known. Through an unusual set of circumstances this has today become the most widespread operating system in the world, since it is the most popular operating system in use on the most widespread class of

hardware, the personal computer or PC clone. Microsoft's partnership with the IBM corporation, forged at a time when the former was a small and rather highly-specialised software house, the latter conducting the majority of their business in the mainframe market, led to the emergence of the original IBM PC fitted with an IBM-written BIOS[1] on a ROM chip and Microsoft's DOS, providing facilities typical of an early operating system as described in our introduction, supplied on floppy disk.

The considerable initial popularity of this product was most evident in the business environment, perhaps most importantly because it made available for the first time a general-purpose microcomputer packaged together with the peripherals essential for its use as a tool of office automation - keyboard, display unit and disk drive. Although more specialised desktop machines, for example, those aimed at desktop publishing or payrolling had already been available for some time, much of the appeal of the original IBM PC lay in the fact that it was not designed with any such specific application in mind, and in its relatively low cost. This meant that the PC market expanded quickly beyond a critical size at which point the PC's popularity became self-fulfilling, that is, the driving force behind it became the popularity itself.

The software industry perceived a huge emerging market for PC applications, and as a result poured resources into developing what eventually became, and in fact still is, the widest range of application software for any single microprocessor-based machine available. This in turn fuelled the popularity even further - here was a low-cost, general-purpose machine complete with a ready-made suite of software to enable it to perform a quite astonishingly wide range of tasks.

Although very cheap compared with its rivals at the time of its first emergence, the expanding PC market was the province of a single manufacturer, IBM, throughout the early 80s and thus competition was not effective in keeping a check

[1]Built-in operating system, which controls hardware at the lowest level, providing an application interface in the form of standard interrupt calls

on equipment prices in that sector. It was the very *idea* of a low-cost, general-purpose computer that so many people had found so appealing, and there was therefore very little about the personal computer that could be patented or copyrighted by IBM. One example of material that *does* fall into this category is the BIOS, a standard piece of software provided by IBM on a ROM chip. It was impossible for other hardware manufacturers to simply duplicate this without being in breach of IBM's copyright, however, it was a fairly simple matter to duplicate the functionality of the original BIOS using completely different code, in other words to *reimplement* the BIOS, and this was the approach adopted by many rival manufacturers. The other lynch-pin to *cloning* the PC was arguably the availability of the microprocessor, the 8086, but since this was the property of Intel, the semiconductors giant, rather than of IBM, it was easy for rivals to obtain a supply. Intel's prime motivation towards profitability lay in making the use of their microprocessor series more widespread, and it therefore had no interest in maintaining IBM's monopoly. Microsoft, the company that had provided the operating system software had similar aims.

In the mid-80s, therefore, a variety of PC *clones* emerged - compatible with the original IBM variety, capable of running all the pre-existing applications software, but manufactured by other companies and frequently selling for a drastically-reduced price. This in turn increased further the PC-compatible's popularity - the falling prices brought about through competition were an obvious incentive, but also, those people who had resisted the encroachment of the PC through not wanting to become reliant on a single organisation for supply and support no longer had any basis for their fears and embraced the PC with open arms.

The fortunes of IBM are but a sideline in our discussion, yet it is relevant to add that their subsequent strategy reduced them from being a market dominator to the status of a relatively minor player today. Rather than pricing their product to compete with the newcomers, IBM instead based its marketing strategy heavily on the Corporation's long-standing reputation as a quality supplier of computer equipment, whilst maintaining the relatively high price of the original

PC. Although this approach had served IBM well in the mainframe part of their business, the fickle nature of the PC market, due arguably to the reduced requirement for after-sales support, meant that the majority of PC consumers were only too happy to switch to the new low-cost clones. In this way, the PC's hardware specification was taken forever out of the hands of a single company and left to evolve according to the concensus of the additional manufacturers now involved.

Microsoft's DOS, perhaps surprisingly, survived the cloning process intact, and it continued to be shipped as the standard operating system on practically every clone. Although many attempts to enhance it were made by other software houses, none of these guaranteed to run 100% of the enormous range of application software already written for DOS, and since this was the PC clone's main attraction, the enhancements, although in many cases technically worthy, were insufficiently enticing to warrant the adoption of a *non-standard* operating system by the majority of computer consumers. Examples of "enhanced DOS" products include Locomotive Software's DOS Plus, shipped as an extra with Amstrad's PC-clone from 1986, as well as, more notably, IBM's first release of OS/2. This left Microsoft in the enviable position of being able to collect royalties on the mass-distribution of MS-DOS, a piece of software which, although modest in terms of functionality and sophistication, was by now an essential part of the PC. The huge array of PC software titles was largely built upon it, relying on it to provide a consistent interface with the rather changeable hardware of a variety of different manufacturers. In a sense, MS-DOS *became* the definition of the PC standard, as the key to success for both hardware and application software developers was now the DOS-compatibility of their products.

Although this standardising effect was crucial to the PC-clone's success, it also gave rise to the single most important difficulty behind using such a system in a machine-vision application, namely, the problems of memory map fragmentation and addressability.

The original IBM PC's memory map had a 64 kbyte segment reserved for the BIOS ROM, followed by a further 64 kbytes of fitted RAM usable by programs.

There then followed 512 kbytes of unused addresses before the display RAM. At the time 64 kbytes was regarded as being a huge amount of memory for a microcomputer, and the unused address space above it more than adequate to provide for any expansion that might have been required over the machine's design lifetime. As a result MS-DOS was designed to load programs only into the contiguous address space present below the screen memory, in other words, no capability was included to cope with the hardware memory map's *fragmentation.* This happened not due to the technical difficulty associated with this feature, which was arguably negligible, but because there was apparently no prospect of it ever being required by users. The original Intel 8086 processor being used in these machines had after all only 20 address lines, and consequently could address only 1 MByte without the use of some kind of memory paging or switching. The extra memory made available to users, had the extra operating system functionality been added, therefore, would have been the space between the top of screen memory and the 1 MByte addressing limit, at best 380 kbytes, with a colour graphics adapter (CGA) display fitted. So this enhancement was apparently not worthwhile.

Subsequent developments turned this lack of foresight into a serious problem for PC users and application developers. From the beginning MS-DOS was a "single-threaded" operating system, in other words, it was not designed to support multi-threading or multi-tasking of programs. However, the advantages of multi-tasking were obvious and in demand by users - a clock present on the screen whilst the user continues word- processing or programming, for example, or perhaps a background printing process to avoid having the PC "locked up" whilst information is downloaded to a printer with a limited buffer. Developers therefore sought to provide, through programming ingenuity, what was absent in terms of operating system support, and made applications like these possible by writing them as TSRs, or *terminate and stay resident* programs.

TSRs were made possible by exploiting an MS-DOS feature which allowed a program, upon exit, to call a particular operating system interrupt which would

return control to MS-DOS whilst leaving the memory occupied by the program marked as such, and thus unavailable for reallocation. In other words, such programs would literally terminate and stay resident in memory. This facility was originally designed for use by *device drivers*, small pieces of code designed to be loaded at boot time and to remain in memory, providing an application interface to a particular item of hardware. Execution of such code would typically be called by a hardware interrupt from the relevant device, or from an operating system software interrupt. The main program's execution would then be stopped, the context saved by the TSR module, that is, contents of processor registers and a return address stored in order that the TSR may manipulate registers without corrupting the main program's data.

Use of this TSR mechanism makes a primitive form of multi-tasking possible. A TSR can be regularly *polled*, that is, periodically re-executed, by attaching its callback to an interrupt generated by a hardware clock. In this way, a host of applications including the on-screen realtime clock mentioned can be implemented.

TSRs remain in memory at all times whilst in use, and there is nothing to prevent a user from *chaining* TSRs, that is, installing multiple TSRs in an execution chain triggered by a single clock interrupt. The normal method for registering a TSR interrupt callback is to "loop" its execution inbetween the interrupt itself and the previous callback routine - in other words, to overwrite the interrupt service vector with the TSR's start address, and to cause the TSR to finish its execution with a jump to the previous value of the service vector. Therefore the chaining of TSRs can be achieved *transparently*, since no knowledge is required during the installation procedure of the current execution chain structure. Clearly there is danger here in that an erroneously- coded TSR may cause the whole machine to "hang", or stop responding to input, in a way which is very difficult to analyse since it may only occur when coresident with certain other TSRs - at other times there may through coincidence be a "recovery" execution path which allows the machine to continue executing normally despite the presence of the error.

Assuming that TSRs can be properly installed, the fact that many of them can

be simultaneously loaded means that shortage of memory can become a serious problem. The 640 kbyte memory addressing limit would otherwise constrain only the maximum size of main program executable that can be run, however, with increasingly-complex and larger TSRs and device drivers loaded, the memory left for the main application is gradually whittled away. In some circumstances, therefore, it is necessary for the user or application developer to "juggle" memory by unloading TSRs in order to release memory to the main application.

The Intel 8086 was the first processor to be used in the PC, and whilst this prevailed the operating system memory limitation could not be said to be a particularly serious limitation, since the size of the address map was in any case also limited by the processor addressing architecture as we have described. However, the introduction of the Intel 80286, with an extra 4 addressing lines, meant that the hardware limitation was effectively removed - up to 16 Mbytes could in theory now be addressed. Simultaneously, more advanced display graphics standards came into use, such as EGA (enhanced graphics adapter) and subsequently VGA (video graphics array). Users became rapidly used to the more complex displays made possible by the higher resolution and enhanced colour palette of the new standard display hardware. Applications grew in complexity, fuelled by the increased performance of the 80286, similarly more and more multi-tasking functionality was demanded, resulting in ever more memory being given over to the permanently-resident TSRs.

It would have been quite technically straightforward to design a new version of MS-DOS able to take advantage of the extra physical memory which could now be fitted and addressed by the new processor. However, the demand from the marketplace for *backwards compatibility* with pre-existing software written for the earlier 8086-based PCs meant that the scope for doing this was limited. The 80286's instruction set was a superset of that of the 8086, just as the 8086 supported all of the instructions of Intel's earlier processor, the 8080, whilst adding new ones of its own. This design philosophy has in fact been continued by Intel throughout the introduction of newer processors in the series, the 80386, 80486

and most recently at the time of writing, the Intel Pentium; all of these support the instructions of the original 8080. However, the reverse is clearly not true - the 8086 *is* in a sense compatible with the newer processors, in that the same machine code instructions can be executed, but *only* if that code is written exclusively using 8086 instructions, and herein lies the problem. To rewrite MS-DOS to take advantage of the extra memory address space would have required internal use of 80286-only instructions, which would deny the backwards-compatibility so much in demand. The 80286 and later processors did in fact have two distinct modes of operation: *real* mode, in which the processor's address space behaved like that of the 8086, using 8086-compatible instructions to access it, and *protected* mode, using new instructions to access the full hardware memory map.

A compromise was reached to partially alleviate this problem, and this was the *extended* memory feature supported by MS-DOS versions 5 and later. This enabled permanently memory-resident code such as TSRs, device drivers and DOS itself to be loaded into the area of memory between the top of the display buffer and the 1 Mbyte real mode hardware address boundary. Although normal applications could still not be loaded into this space, more room was freed for them in the lower portion, now termed "conventional" memory.

At the time that 80386-based PCs entered the market, physical memory had been reduced in price such that it became cost-effective to fit these machines with several megabytes thereof. A consistent interface for applications to address this memory within the DOS environment was established in the form of the LIM[2] *expanded* memory manager standard, but special techniques had to be used by applications accessing this, and the space could in any case only be used for program data. The *conventional* memory, limited in size, retained its special significance since it was the only place where executable code could be loaded.

[2]Lotus, Intel, Microsoft

4.2.1 Experimental: Network Handling using TSRs

Unfortunately, the development of commercially-available compilers appears to have lagged some way behind those in use by professional application developers. At the time of our work C compilers which could address the upper, *expanded* memory range with the aid of an expanded memory manager were only just beginning to become available - the majority were still constrained to use only conventional plus extended memory for program data, that is, memory up to the 1 Mbyte boundary. Furthermore, we were influenced to use compilers produced by either Borland or Microsoft, since the manufacturers of our machine vision hardware supplied the equipment with compiled libraries of standard routines to make application interfacing to the hardware more convenient, Two copies of these libraries were supplied, one in each of the relevant proprietary formats, and to renounce these compilers would therefore also have meant abandoning these libraries, necessitating the extra difficulty of addressing the hardware at the register rather than the functional level. On the other hand, maintaining the libraries' availability meant working with the described memory constraint, which was problematic since image-processing manipulations are particularly memory-intensive.

Our initial approach was therefore to use the 80286 PC, with associated machine vision hardware, as the front end to a more powerful host processor. We sought to do this by using the Ethernet TCP/IP local area network for communication between the PC and host, and we envisaged this host as being a Sun Microsystems Unix IPC workstation, also connected to the network. In the proposed set-up image data would be acquired and stored in the dedicated machine vision system, which would also perform any required rudimentary processing such as contrast stretching or convolution. The remote Unix machine would perform any operations lying outside the capability of the machine vision hardware, which, whilst purpose-designed and high-speed, was limited in functionality. Data would be received from the machine vision system, firstly via the host PC and

subsequently by Ethernet, processed and returned for display.

In order to make use of network connectivity from within our application, compiled using Microsoft's V5.1 C compiler, we made use of another compiled library, this being Sun Microsystems' PC-NFS programmers' toolkit. This provided both a socket-level and a remote procedure call interface to the network. In order to make use of these facilities, a TCP/IP device driver known as the RTM[3] was required to be loaded in memory as a TSR at all times; this consumed approximately 128 kbytes of conventional memory. The RTM's callback is attached to the Ethernet device hardware interrupt; upon execution it does the required TCP/IP processing upon incoming and outgoing data, thus presenting an interface spanning the physical, data link, network, transport and session layers of the OSI networking model. Applications using the session layer interface can be configured to receive software interrupts from the RTM when data is available for reading via a socket or remote procedure call. An overview of sockets and remote procedure calls, together with a discussion of various issues relating specifically to them is given in chapter 3.

In general it is desirable for networking activities to occur *asynchronously* with the operation of the rest of the computer. The arguably undesirable alternatives here are for host processing to stop pending the arrival of data from the network, or the host can *poll* the network device periodically, since this typically has the ability to temporarily buffer data as it arrives.

The most efficient option is for the arrival of network data to *interrupt* the main flow of execution. In order to investigate how this might best be achieved, we designed a TSR *messaging* application, and this engendered our described understanding of the issues affecting the use of MS-DOS as an operating system for the support of machine vision applications.

The functionality of the messaging application can be briefly explained as follows:- having loaded the TSR "server" module, which is self-installing, control is returned to the command-line prompt, and the user is then free to run other

[3]Resident transport module

programs as required. Another user using a remote host connected to the network uses a "client" module to send messages which will then cause the "server" user's main application to be interrupted by the TSR, which will display the message on the screen before passing control back to the main application.

The "server" module callback is attached to a software interrupt generated by the PC-NFS toolkit RTM as described, and for this reason it is necessary to load the RTM first. Upon installation the messaging application communicates with the RTM via library function calls and registers a listening socket, a logical endpoint of communication such that the RTM will generate the required software interrupt when data is addressed to it from a remote host.

The "client" module is a more straightforward DOS program which uses the PC-NFS library to originate messages destined for the "server" module's listening socket. With no material changes to the source it was also possible to compile the "client" module on a network-connected Unix machine, and thereby to send messages inbetween the two different platforms.

4.2.2 Discussion

The messaging application worked well in practice, although there were various initial problems which detracted from its usability. For example, it was quite possible to load *two* or more instantiations of the messaging TSR which would both attempt to bind the same socket address. Although only one of these would succeed, thus maintaining messaging functionality, this loading of multiple copies consumes more of the valuable conventional memory than is strictly necessary.

To remedy this problem, the messaging TSR was modified to write a disk file with the start address of its installation. Upon loading, this file would be opened and the contents of the specified address examined in order to ascertain whether a copy had already been loaded, installation being aborted if this was found to be the case. The presence or absence of the disk file alone could not be relied upon as an indicator of the TSR's presence, since no reliable mechanism was available

for deleting the file upon powering down the machine.

As originally coded, no way of *unloading* the messaging TSR was available, short of rebooting the system. Although it was originally envisaged that the TSR would never need to be unloaded, the use of programs with large memory requirements, typically compilers, required that the memory occupied by the messaging TSR and the RTM be freed. To meet this requirement, the most straightforward approach was to cause the messaging TSR to *unload itself* after its next execution. To this end a third utility was designed which would obtain the TSR's installation address from the described disk file, apply an offset to obtain the address of a variable dedicated to the purpose, and change this variable to a pre-set value. The messaging TSR itself was modified to check this variable and, if appropriate, to extract itself from the interrupt chain before making a standard function call to MS-DOS which would return control and delete the TSR image.

The most significant hurdle to the messaging application's use was the large amount of memory taken up by the RTM, since this dramatically reduced the proportion of standard applications which could coexist. Furthermore, certain applications could not coexist despite the availability of sufficient memory, this was generally due to the main application's own use of software interrupts already employed by the RTM. Notably the Microsoft Windows environment fell into this category.

4.2.3 Interim Conclusion

Although we have demonstrated that it is possible to use a PC-clone running MS-DOS in the described rôle, that is, as front end to a more powerful host processor, connections being by TCP/IP local area network, there are a variety of problems with the asynchronous network operation which the PC is required to perform, which seriously impede development of complex applications along these lines. The most significant we encountered is the lack of testability of such an application, and this means that the bugs which are inevitably introduced during

any programming process are extremely difficult to analyse. This testability problem is brought about through two factors - firstly, the organisation of code required to bring about this primitive form of multi-tasking is inherently complex in terms of linking interrupt vector chains, saving execution contexts and so forth, and a simple mistake in such a critical activity is likely to "crash" the machine into an irretrievable state, leaving little or no clue as to the cause of the problem. This is compounded by the second factor, this being that MS-DOS implements no scheme of memory *reservation*, which is usually an important component of a multi-tasking operating system. Such a scheme typically allows execution threads to access or modify only specifically-allocated areas of memory, returning an error interrupt if attempts are made to access elsewhere. In our messaging application, however, there is nothing to prevent any instruction from modifying any portion of memory. Consequently a straightforward bug caused by erroneous manipulation of a pointer may have a wide variety of effects including an instant execution halt, for example, if the program counter stack is corrupted. Unfortunately, the effects are also likely to manifest themselves in a much more complex way, perhaps overwriting an essential system variable such that the system crash does not actually occur until control passes back to the operating system.

As a result we updated our view of the best choice of operating system to use for a general low-cost machine vision problem. It appeared that, whilst competitively-priced imaging hardware is most readily available in a form compatible with the PC-clone architecture, the difficulties associated with designing the kind of complex application described, although not insurmountable, are prohibitively expensive in terms of the extra resources which must be expended in order to overcome them.

4.3 Unix

We next turned our attention to machines running the Unix operating system, the particular systems in view were Sun Microsystems IPC workstations, these being

low-end machines based on a Texas Instruments SPARC[4] processor, and also a multi-processor server machine, the SPARCCenter 2000, identical in functionality but greater in terms of processor throughput and memory resources.

Originally we had envisaged our use of this environment as being simply a processing "workhorse", free of the memory and testability limitations of the PC-clone, and we therefore coded trial applications along these lines. Again we used the PC-NFS programmers' toolkit, this time to write straightforward non-TSR applications which would load a line or block of image data from the dedicated image-processing system into PC host memory, transmit it via the socket interface to the SPARCCenter Unix machine, which would then perform the one-dimensional or two-dimensional (as required) fast Fourier transform using a commercial library from the Numerical Algorithms Group, returning the results back to the PC and thence to the machine vision system frame buffer for display. Compared with our asynchronous messaging application this was comparatively straightforward to achieve, although it was here that we first encountered the importance of the compute/communicate time ratio which is explored in more detail in chapter 3. In short, the speed advantages of using a more powerful remote processor may be offset or even completely negated, depending on the amount of communication overhead involved.

It is possible to use the described Unix machines by means of a text-only terminal connection, and indeed this was the access method of choice with the earliest such systems. However, the more modern systems in view support a range of sophisticated input/output facilities based around the X windowing system. Upon closer examination it was also apparent that easily-testable asynchronous networking was very straightforward to achieve within the Unix environment. This is due to a number of factors - one of the most important is that Unix implements the memory *reservation* system so unfortunately absent from MS-DOS. Under this scheme general processes which do not possess appropriate override privileges can only access the program and data segments owned by them, and

[4]Scalable processor architecture

attempts to access elsewhere result in an error interrupt. Therefore coding errors involving memory access, typically through pointer manipulation, can be exposed almost immediately. Also, the asynchronous networking support, which we previously sought to achieve under MS-DOS through loading device drivers and directly registering interrupt service routines, is present as part of the operating system kernel, greatly reducing the amount of application development required.

We therefore reviewed our proposed use of the Unix systems - it appeared that the console of the Unix workstation itself was ideal for use as a "front end" to our proposed machine vision cluster, and that the PC-clone with attached dedicated vision hardware was far more suited to the rôle of "hardware server" slave. At this stage it was still envisioned that the PC-NFS programmer's toolkit would be used to allow the Unix machine, now fulfilling the rôles both of system front end and computation engine to communicate with the PC, thereby obtaining access to the imaging hardware.

In pursuit of this goal we investigated the Unix environment and sought to build within it a general-purpose machine vision application suite.

4.3.1 Experimental: Xdefect X-Windows Application

At the present time, X-Windows has become the standard display subsystem for Unix workstations from all manufacturers. As in Unix itself, its roots lie with the academic institutions of the United States - X began as an application project at the Massachusetts Institute of Technology. Its original purpose was to attempt to unify the many proprietary Unix windowing environments present when graphics workstations were first introduced, producing a standard which would make compliant machines intercompatible at the application graphics level, even where the machines have different architectures, are built by different manufacturers and are running different operating systems or networking protocols.

X is a windowing system with built-in networking support. Although it is often used where the client application is controlling a window on a graphics device physically attached to the machine where it is running, leaving this networking support largely redundant, its presence does mean that a window can be displayed by a client on a *remote* workstation, and this is a transparent process as far as the application programmer is concerned. Note that the terms "client" and "server" are somewhat counter-intuitively assigned in the context of X-Windows - the display "server" is the most important subsystem involved in X, and each machine has a single server instantiation which provides an interface between network requests for screen access and the frame buffer which directly controls what is displayed on the screen. The display "server" is so named because it is "serving" out the display resources to the application "clients" which consume and compete for them. The other, lesser X subsystem is the window manager, which allows the user to manipulate client windows with resizes or iconification actions, for example. These operations are conducted without any correspondence with any of the clients. However, it is possible to run an X display without the use of a window manager, in which case the user loses the ability to control the layout of windows unless the clients themselves provide this functionality.

The protocol which defines procedures for communication between window manager, application client and display server is part of the X standard devised by MIT - this is the minimum standardisation required to ensure compatibility between compliant implementations. However, the X standard falls short of defining shapes of buttons or other widgets for example - this enabled manufacturers with existing proprietary windowing systems to design new, X-compliant versions without alienating the existing customer base through altering the "look and feel" to which they were used. Exact details of window appearance *are* specified, however, when a particular X widget *toolset* is selected. Of those available, including X Intrinsics and Motif, we selected the Sun Microsystems XView toolset, since the package is available free of charge, thus helping us to achieve our objective of a low-cost solution. Sun Microsystems have since withdrawn support for XView

and joined other manufacturers in adopting Motif as the toolset standard. However, this has no implications for the maintainability of already-existing XView applications such as our own - providing these are statically linked with the toolset they can be run on any X-Windows platform.

In common with most windowing systems, use of X requires "event-driven" programming. The reason for this can be explained thus:- the conventional non-windowing programming style puts the machine effectively "in control" of the flow of interaction with the user. In other words, the system will typically prompt for input, whether this is in the form of a menu or straightforward text/numerical entry. The user has no means of controlling the flow of execution unless the opportunity for him/her to do so is explicitly offered. An exception to this rule is that in these situations the user typically has the ability to "interrupt" with a special key combination, this may halt execution completely or may cause the application to produce a menu of interrupt options, for example.

A windowing system, on the other hand, puts the user more directly in control of the execution flow. The application designer makes available a range of controls which make take the form, for example, of buttons, sliders, pull-down menus and so forth - collectively these are often termed "widgets". At every stage the user can choose to manipulate any of the controls available. In fact, the default activity for a windowing application is to do nothing, pending an "event" from the user - this is in effect a software interrupt. We have found that the development of this kind of application takes much more design effort, since the program must be able to handle any event at any time that the user chooses to generate it. The application therefore needs to be extremely robust. On the positive side, however, we have found that windowing applications tend to be more intuitive and easier-to-learn than their traditional counterparts.

Figure 8 shows the top-level window, or *root frame*, of the Xdefect application; this is all that is visible upon the application's first invocation. Each of the four icons shown represents a sub-window, the buttons below will cause them to be *raised* or *lowered* as required by the user. In fact the sub-windows are constructed

Figure 8: Xdefect application top-level control panel

during initialisation, and details thereof are communicated to the display server which maintains their image in memory before being instructed to reveal, or raise them by the client. The sub-windows therefore appear with a minimum of delay when the appropriate button is pressed.

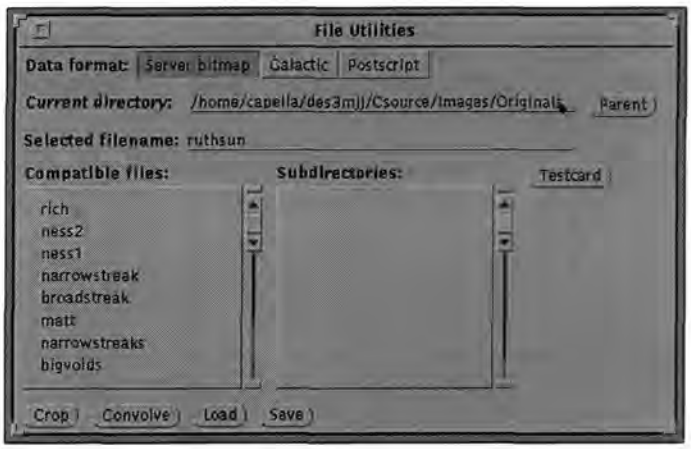


Figure 9: File utilities menu

The sub-windows represented by the icons are as follows. Figure 9 shows the file utilities menu, which is used to save and restore images to disk on the Unix system. The user can traverse the directory hierarchy using the "parent" button to move up to the level above the one current, and choosing the desired subdirectory from the list presented to move to one below. The "crop" and "convolve" buttons are handles for functionality, the implementation of which was not finished, and if these buttons are pressed by the user, no action will be initiated since they have null callback functions. This is an example of a feature which we have found to be a particularly useful aspect of windows programming - modularised development is very straightforward to achieve, since code in callbacks for specific widgets can be modified and enhanced with little need for interaction

with the rest of the software.

Figure 10: Showing significance of differing server memory formats - compare original in figure 14

The "data format" selector is of particular importance. Although X allows standard graphics calls to operate correctly irrespective of the underlying hardware architecture, the overhead of the protocol is such that the time delay for certain display operations can seriously detract from the application usability. For example, the time taken for the Xdefect application to load a 512 by 512 pixel by 8 bit image into the display server would be many tens of seconds if the pixels were set one by one using the XPutPixel function, although this would be guaranteed to show the correct results on any X-compliant display. Instead, our application can directly save and restore the server memory which contains the image, represented in the native server format. Although these operations are considerably faster, they introduce potential non-portability into the application, since server memory is organised differently on different machines. Figure 10 demonstrates this effect. The image shown was originally acquired and converted to server memory format using a Sun Microsystems IPC workstation. The server memory was then saved directly to disk and restored onto a copy of the same application, this time running on a Hewlett-Packard 805 workstation. It can be seen that the information has not undergone a positional transformation, but is now displayed with a corrupted colourmap.

For this reason, we included in the application facilities for saving and restoring images in a portable format. For this purpose we selected a Postscript[5] bitmap dump. Whilst a full Postscript interpreter would have been outside the scope of, and inappropriate to, such an application, the application was able to load files in this particular format, a subset of the full Postscript language. This file format had the added attraction of being laser-printer compatible, allowing us to generate the various application screen dumps presented here.

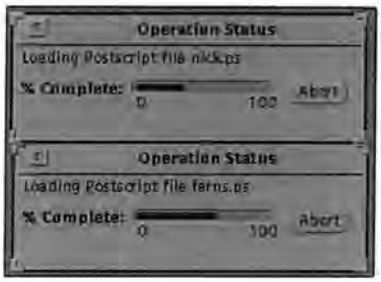


Figure 11: Status indicators showing progress of parallel-decoding operation

Converting to and from the Postscript language is a highly CPU-intensive process taking many tens of seconds for a typical image. This caused us to discover another interesting feature of X-Windows programming - as we have mentioned, the default activity for the application is to do nothing pending an event from the user, which will then generate a callback to a handler function. However, no further events can normally be processed until the handler function terminates, although they will be queued for future handling. In the specific case of our encoding/decoding of Postscript files, this means that the user would normally lose control of the application whilst these operations continue.

For this reason, we designed the callbacks for events connected with the described encoding/decoding to *spawn* a separate thread of execution, or *process*, which would handle the CPU-intensive part of the task, leaving the main execution thread to resume processing of user events, thus allowing the user to retain control at all times. The secondary or *child* process was designed to communicate periodically with the main process using a Unix *pipe*, giving progress information

[5]Postscript is a trademark of Adobe

in terms of percentage of job completed. The main process' end of the pipe was registered as a generator of window events with the XView *notifier*, this enabled the status indicators shown in figure 11 to be updated as progress is made. The process-spawning procedure was designed to allow an arbitrary number of pictures to be simultaneously decoded, although naturally the CPU resources of the host would be split between these jobs. Thus there are two status windows shown in the figure.

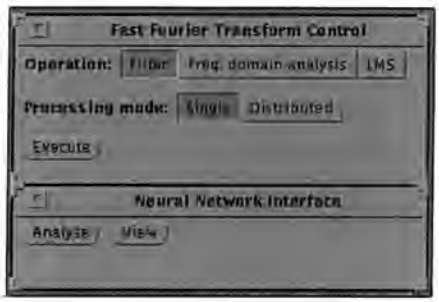


Figure 12: Interface to frequency-domain functions and neural network subsystem

Topmost in figure 12 is the simple frame which controls frequency-domain oriented functions. The NAG library's fast Fourier transform implementation is again used to transform images between the spatial and frequency domains. The available operations include a straightforward transform of an image into the frequency domain, the modulus (power) of the result being displayed as an image with zero frequency at the centre of the image, brightness corresponding to power according to an exponential scale. Figure 13 shows progressive iterations of an image low-pass *filtering* operation, which is achieved by transforming the image to the frequency domain, multiplying by a kernel which sets all power at frequencies higher than a threshold to zero, and then operating the reverse Fourier transform. As this operation is repeated, it can be seen that high spatial powers are indeed lost from the image, resulting, for example, in visual "echoes" of strong contrast boundaries.

Bottom-most in figure 12 is the frame containing the two buttons used to interface with the neural network backpropagation subsystem described in detail in chapter 7.

Figure 13: Frequency-domain filtering operation showing progressive loss of high spatial frequencies

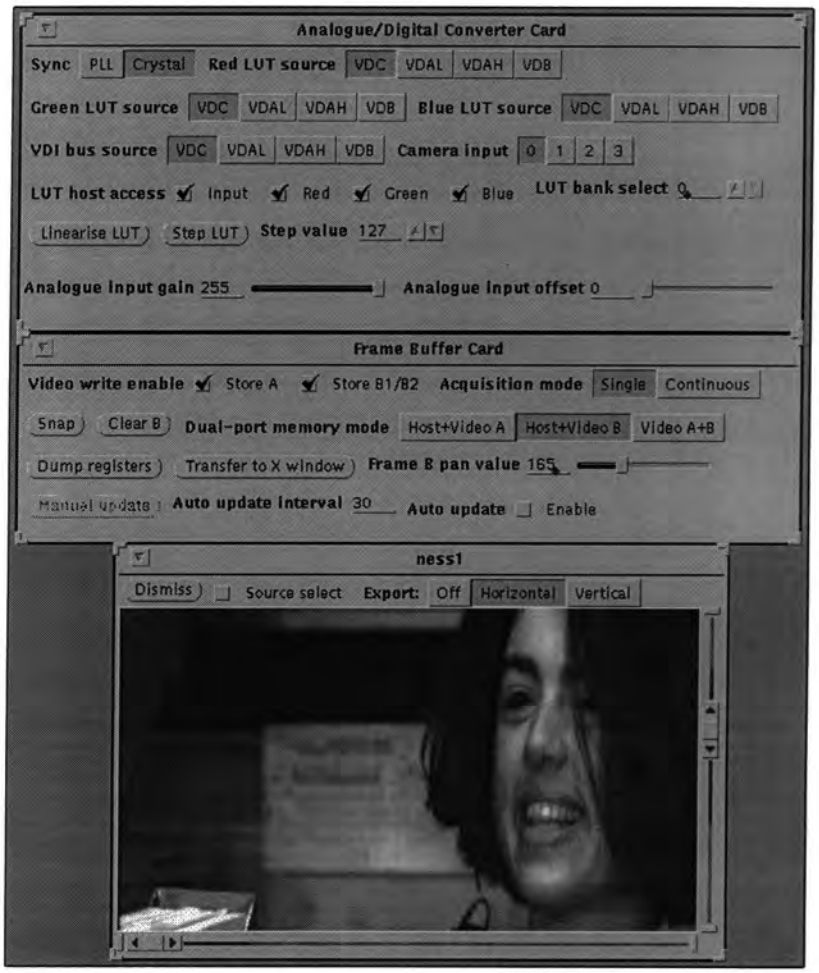


Figure 14: Interface to machine vision system via networked PC, including acquired image

Figure 14 shows the interface to the Imaging Technology dedicated machine vision hardware. This operates by communicating over Ethernet with the host PC which then addresses the machine vision system as instructed over the PC bus. The two frames shown control respectively the system's analogue / digital converter card and frame buffer card. Output from the system can be viewed on a directly connected analogue monitor. Alternatively, the "Transfer to X Window" button can be used to move an acquired or processed image to the Unix system for display and subsequent manipulation there. Bottom-most in figure 14 is an image which has been acquired from camera and then transferred in this way.

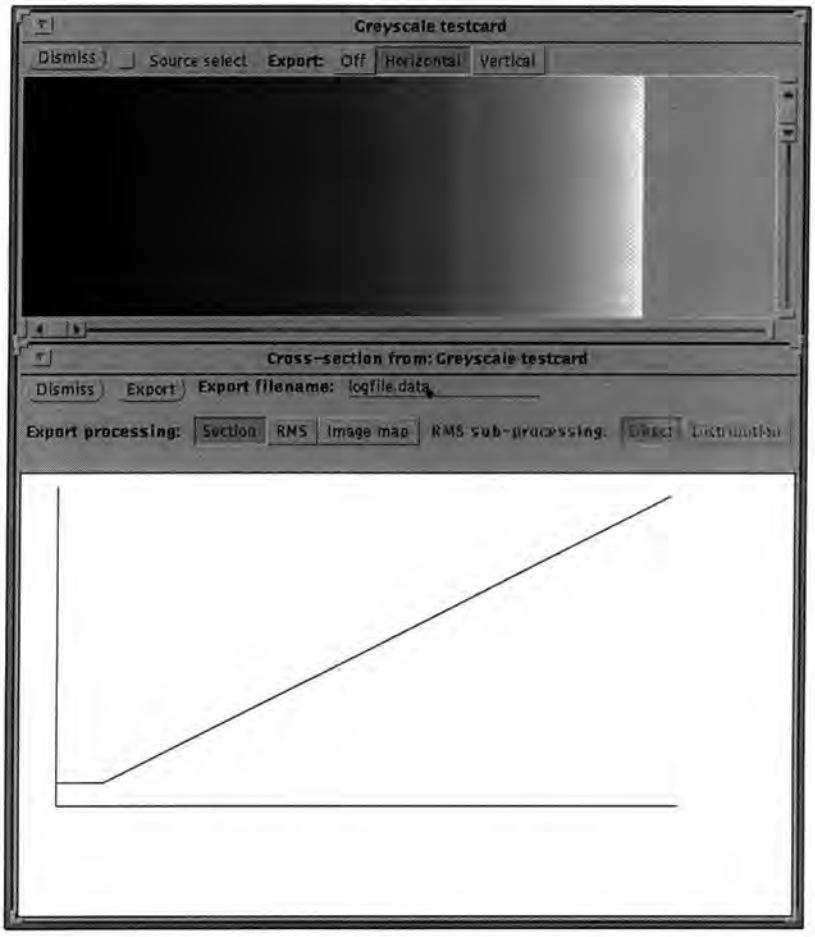


Figure 15: Greyscale testcard with corresponding cross-sectional view

Figure 15 shows the Xdefect application's facility to show a graphical cross-section of image intensity. The controls on each image display window can be used to select a vertical or horizontal section, and the mouse used to determine

the portion of image which is to be profiled. The "export processing" options were used to generate the image data samples used in chapter 7's experimental.

4.4 Linux

In parallel with our development of the Unix Xdefect application, we investigated a third operating system in order to assess its relevance to our search for the optimal low-cost general machine vision environment. Known as Linux, this is an alternative operating system for the PC-clone architecture, running on machines with the 80386 processor and above.

As with Unix, the origins of this operating system lie predominantly with academic institutions - the project was started as a hobby interest by a research student with the University of Helsinki in Finland, Linus Torvalds. An experienced user of Minix, an early PC Unix, Torvalds set out to design a better system for himself, overcoming many limitations characteristic of Minix, beginning by writing a kernel from first principles. As the project developed it took on board input from many parties who gave of their efforts without monetary reward, the aim being to provide a fully-functional PC Unix for use by all, free of charge. The most significant contributions have been from MIT's Free Software Foundation (also known as the Gnu project), who have specialised in compiler and utilities support, and from the various individuals with specialist skills who sought to reimplement TCP/IP. As Linux passed a critical level of functionality which included a fully-working POSIX application interface with kernel networking support, it became possible for the standard Berkeley freeware networking application code to be included, providing features common to most modern Unix implementations such as DNS[6], telnet and NFS[7], without additional development effort. Furthermore, Linux also supports the freeware PC X-Windows server implementation known as X386, in common with other PC Unixes such as

[6]Domain name service
[7]Network file system

Mach, SCO and BSDI.

4.4.1 Experimental: Itex System Interface

Because Linux is a free and fully-functional Unix, the source code for its kernel and subsystems is readily available, and users are encouraged to develop their own enhancements to the system and make the fruits of the work involved available to others. Therefore the writing of a Unix device driver, which would under usual circumstances require highly-specialised skills and detailed knowledge of the proprietary hardware involved, becomes relatively straightforward under Linux, particularly since source examples can easily be obtained.

Although at this stage we already possessed the means to produce the PC application which would allow the Unix Xdefect suite to interface with the Imaging Technology hardware, using the PC-NFS programmers' toolkit as described, it appeared that the Linux route might provide a better overall solution within the parameters of our search. The immediately-occurring reasons for this were as follows:- the complete Linux package was available entirely free, whereas the PC-NFS toolkit, providing much more limited networking support, had cost in the region of 500 pounds. Secondly, much more flexibility was available with Linux in that the full multi-tasking support available made it straightforward to continue using the PC for other applications, such as word-processing, or network file-serving, at the same time as running the hardware interface. This could only be achieved to a limited extent using MS-DOS/PC-NFS and by using the error-prone and difficult TSR techniques described. The use was limited in the sense that DOS applications making their own direct use of software interrupts could not be run at the same time as the Itex hardware interface.

We therefore examined the two options available for writing the Imaging Technology system device driver, these being to write it as a *system* mode or *user* more driver.

System mode device drivers are most commonly employed for the majority

of peripherals. In effect these constitute part of the operating system, requiring recompilation thereof when any changes to the driver are made. These are the more difficult of the two types to write and interface with an application, but provide a more sophisticated set of facilities. The application interface is generally made by means of a special device identifier. Usually these are kept in a single directory for convenience, */dev*, although this is not a requirement. These special files possess many attributes of a normal file, for example they have all the usual access control features and may be opened, read and written in the same way. However, when an application makes such accesses, control is in fact passed to the associated device driver functions as determined by the special file's major and minor device numbers.

Additional complications of system mode device drivers include the necessity to copy data between reserved kernel memory and user memory when transferring data to and from the application using the driver, since system mode driver variables must generally be located in kernel memory which is inaccessible to user applications.

We found that a user mode device driver is much more straightforward to code and can in fact be built-in to the application if required, removing the need for application and device driver to be implemented as two separate modules, communicating via a special device file. We coded our user mode driver using inline assembler instructions in a C program running within the Linux environment; these used *in* and *out* processor instructions to manipulate the Imaging Technology system's registers under C program control. A particular problem was Linux's bitmap I/O protection mask - each process has associated with it such a mask, consisting of one bit per hardware port address between 0 and 0x3FF. A set bit indicates that the process is permitted to access the corresponding port, and vice-versa. The 80386 (and later) processors do in fact support this feature in hardware - the Linux kernel keeps copies of the bitmap for each process and loads them into the processor according to which process in running, the access control functionality therefore being in hardware.

The Imaging Technology system is based on a VME bus, a converter card being used to interface this to the PC bus. The analogue/digital converter, frame buffer and arithmetic logic unit are implemented on modular cards, each with a bus connection and a different VME bus address - in our case these were set to 0x300, 0x1300 and 0x2300 respectively. The VME/PC bus converter repeats access requests from PC to VME bus, with direct address translation, but in order to isolate the VME bus from spurious traffic on the PC bus, it does so only when it is selected by the presence of 0x300, its own address, on the least significant 12 lines.

A problem occurs because conventional PC peripherals use port addresses only within the range 0 to 0x3FF - this is required for compatibility with processors earlier than the 80386, since these were not capable of addressing ports outside this range, and the described hardware bitmap access control reflects this. In order to allow our user mode driver to make the required accesses to the vision subsystems at addresses 0x1300 and 0x2300, therefore, it was necessary to set the processor in *supervisor* or *override* mode, to disable the access control and allow access to any port. This action clearly has security implications for a potentially multi-user system, and can therefore only be achieved through a Linux kernel-level function which requires superuser privilege for access, and it was therefore necessary for our application to run with the SETUID root flag set. In general this would not be necessary for a driver accessing only ports inside the conventional range, since the process access control bitmap could be appropriately set. Our finding is that the mechanisms for doing this are unduly complex for the application in view, however, and that the simplicity of running the driver as root makes this the solution of choice for a system where security and inter-application protection are not key issues.

4.5 Conclusions

We have investigated the most important and relevant computer architectures and operating systems in order to determine the optimum configuration for a general-purpose low-cost machine vision system. Our final solution involves dedicated machine vision hardware directly interfaced to a PC-clone running the Linux operating system. The prime motivation for this is the large value for money obtainable due to the mass-production of both PC and PC-bus-compatible peripherals, since both are in great demand due to the commercial factors explained. The Linux system is preferred over MS-DOS with networking toolkit - this is because the operating system's origins in academia as freeware mean that it is lower in cost, that internals can readily be analysed as source, speeding understanding and application development, and that far more is available in terms of multi-tasking and networking support, again reducing the developer's outlay of effort. Of the two modes of device driver outlined, we find that the simplicity of use of the user mode justifies its selection for the problem in view. The enhanced feature set of the system mode might be justified should more complex hardware interfacing be required, for example where many processes are contending for access to the hardware, or where DMA and hardware interrupts need to be supported.

Our solution's front end and general processing facilities are provided by a low-end Unix workstation with an X-Windows graphics display. We find that a window-oriented package makes the user interface far more intuitive and easier to learn, although it also increases the programming effort. However, the operating system support and availability of highly-functional windows programming libraries at reasonable cost makes use of the Unix machine in this way a greatly more attractive option when compared with the described TSR-related techniques on an MS-DOS PC.

Now that we have succeeded in finding an optimal combination of low-cost hardware, software and communications techniques, we need to return to the study of the basic image-processing algorithms, and it is to this that we turn our

attention in chapter 5.

Chapter 5

Neural Network Overview

5.1 Why Neural Networks?

5.1.1 *TEXIS*, an Illustrative Vision Problem Example

Vision System Paradigm

As has been discussed, is can be helpful to express the general automatic inspection problem in terms of a number of sub-processes, as follows:-

- Data acquisition.
- Preprocessing of acquired data.
- Artificial intelligence element.
- Output processing.

It can be seen that when this paradigm is adopted, the task to be performed by the artificial intelligence element is essentially that of a data classifier. That is, the AI engine must analyse input data vectors and assign them to the most suitable *class* based on a ruleset which may be either explicitly specified, or, in the case of a supervised neural network, learnt by example using a *training set* consisting of data which exemplifies clearly the rule which is to be inferred.

Description

The work of Brzakovic et al.[15] deals with a typical inspection problem which illustrates well the rôle of the various functional units described, involving the construction of an automatic expert system named by the authors as *TEXIS*, capable of inspecting parquet[1] samples for artefacts such as *cracks*, *streaks*, *knots* and *holes*. The salient features of this system will now be described, since this will bring the issues relating to the selection of AI element type into focus.

Firstly, a digital image of the sample is obtained using a CCD camera. Secondly, the data therein is *preprocessed* in order to reduce its volume to a more manageable level - the key aim of this activity is to discard the large quantities of unimportant information present in the image, whilst retaining data which describes, as concisely as possible, the features of interest. This step is also known as *feature segmentation*, and in this case is a two-stage process. Firstly the parquet samples, which may be arbitrarily oriented in the image, must be isolated from the background, the steps involved here being as follows:-

- Extraction of edge pixels.
- Grouping of edge pixels that constitute individual edges, and computation of the coefficients of corresponding lines.
- Grouping the parallel and perpendicular edges that bound individual samples.

Secondly the samples, once isolated, must be partitioned into *defects* and *generic texture*. This is carried out using a variety of traditional image-processing functions, for example the *Marr-Hildreth* operator, which involves filtering by convolution, followed by a tracking of the appropriate zero-crossings in the convolved result, considering eight-neighbour connectivity. Ideally this will produce

[1] A floor covering of pieces of hardwood fitted in a decorative pattern

the outline of an image feature, but in practice such an outline is likely to be incomplete, and a subsequent morphological operator may be required to "thicken" the outline until it becomes completely interconnected.

Numerical parameters must now be extracted from the outline shape such that a feature vector, suitable for classification processing, may be constructed. In this case the measure of *compactness*, C, is employed, defined as:-

$$C = \frac{P^2}{A} \tag{7}$$

Here P denotes the outline perimeter and A its area. The feature vector therefore has only one element, and this allows a straightforward Bayesian classifier to be employed. This assumes that all defect classes have an equal likelihood of occurrence, and that each class w_i is characterised by a normal conditional probability density:-

$$p(x/w_i) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-1/2(\frac{x-m_i}{\sigma_i})^2} \tag{8}$$

For each class w_i the mean, m_i and standard deviation σi in equation (8) are obtained from a training process in which C is measured for various representative defects interactively identified by human operators.

In the recognition phase, that is, after training has been completed, the classifier assigns defects of measured compactness C to a class w_i if $p(C/w_i) > p(C/w_j), j = 1, 2, \ldots, L$ where L denotes the number of classes.

The probability densities p defined in (8) will tend to overlap for different classes, and for certain values of C the difference between the largest and second-largest p will be of insufficient magnitude to make a reliable classification. In this case further processing is carried out, specifically, a defect may be classified as a *crack* rather than a mineral *streak* if its average width is less than a certain characteristic width W_{max}. Further, a measure of defect *texture* may be used to distinguish cosmetic defects such as *knots* and *streaks*, which have a generic wood texture, from defects generated by outside forces such as worm *holes* and *cracks*,

which have no texture.

In this case the texture descriptor, M_{tex} is defined by:-

$$M_{tex} = \frac{f_{count}}{A} \tag{9}$$

$$f_t(x,y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} [f(x,y) - f(x+i, y+j)]^2 \tag{10}$$

Here f_{count} is the number of pixels in the defect area for which the *intensity typicality* value, f_t is greater than a certain threshold, T_{thresh}. Equation (10) defines f_t for a pixel at position x, y in terms of the summed differences between the intensity f of that pixel and those of its neighbours. m and n delimit the defect boundary.

Finally, worm *holes* and *knots* can be distinguished simply by their area, since these are found to be, respectively, smaller and larger than a certain characteristic area A_{max}.

Performance

The *TEXIS* system learnt to correctly classify its training set of 100 defects, chosen by human inspectors as typical representatives of the four defect classes, with 100% accuracy. When using the system to inspect previously unseen features, the correct recognition rate was found to be 81%. The authors suggest that the resolution of the camera was the limiting factor in the system's performance.

Discussion

The *TEXIS* system is typical of contemporary approaches to automated inspection in that the artificial intelligence element, in this case a Bayesian classifer supplemented by various parametric rules, as required, is almost trivial in nature, and by far the greater part of engineering effort has been poured into the preprocessing phase, in this case defect outline extraction. The acquisition phase

also presents a multitude of problems, here the chief difficulty is that of ensuring uniform illumination, or else compensating for the non-uniformity.

Although the system works well in its intended application, a serious limitation is that, as the authors say:-

> ... *parameters and threshold values are determined experimentally, and are generally a function of the material being inspected, and the digitisation conditions.*

Such shortcomings are highly typical for systems of this type, and it is evident that this is having a dramatic effect on the take-up by manufacturing industry of automated inspection systems, perhaps surprisingly so, since this is a technology which at first sight appears to have obvious cost benefits. The problems mean that, hitherto, the offerings of machine vision companies have been closer to services than to products. This has limited market growth by making it dependent on the availability of suitably-trained machine vision engineers, and by increasing the cost to the end user of a solution to his problem.

There is, therefore, a very clear motivation for working towards a new generation of automatic inspection equipment, in which the necessity for a skilled instrumentation engineer on the factory floor has been removed. One might reasonably envisage for the future the availability of a range of generic off-the-shelf inspection *products*. Having selected a unit from the range, specified appropriately to the vision problem in terms of sophistication and processing power, customers would be able to use non-specialist personnel to configure the system to their needs using a straightforward *training* process, at greatly reduced development and installation costs. For this an operator would need to give the system *feedback* about its classification decisions until acceptable performance is achieved.

This, therefore, is the ultimate goal when we examine artificial neural networks in the machine vision context.

5.2 Introduction and Background

Hertz et al. in [26] give a rigorous mathematical introduction to the topic of neural computation including details of the multi-layer perceptron backpropagation algorithm which will not be reproduced here.

Forsyth et al. in [27] give a useful perspective on neural networks in the wider context of machine learning in general. We draw upon the expertise of both these sets of authors in order to illustrate the theoretical background to our neural network implementation presented in chapter 7.

5.2.1 Machine Learning

Many experts researching the field of artificial intelligence have hitherto constructed "intelligent" systems which do not possess the capability to learn. Consider as an example a master-level chess-playing program. Although chess algorithms are today highly-refined, and capable of performing at the level of a strong club player[2], they are in general completely *deterministic*. In other words, the algorithm cannot learn by itself to do better. More recently, however, there has been a resurgence in machine learning driven by the developments in the field of *expert systems*. Such a system, designed perhaps for equipment fault or medical diagnosis, requires for its function a high-quality knowledge base, which is difficult to artificially construct. Machine learning offers a way around this problem; in effect the system produces its own knowledge.

5.2.2 Black Box Techniques

Neural networks have been the subject of much interest largely because of their apparent ability to learn by example. Neural networks are, however, part of a class of algorithms which can be called "black box" techniques. The characterising feature here is that the user is in general not concerned with what happens *inside*

[2]Rated performance of GNUChess 4.0, available from the Free Software Foundation

the algorithm whilst it is learning, but rather with the inputs and outputs.

Behavioural scientists favour the black box approach in biology, simply because it is very difficult to investigate representations inside the brain of an animal or other biological system whilst it is still alive. In the case of an artificial system, internal representations can generally be accessed much more easily, but it may nonetheless still not be profitable to do so.

Typically artificial systems in the black box category have a mathematical bias, and partly as a result of this, the knowledge gained during the training phase tends to be opaque. Even a mathematically-sophisticated person cannot inspect the system's internal representation and say what the system has learned. Forsyth et al. in [27] say that such a system has a "write-only" knowledge base. This is because mathematical theory can as yet make only faltering steps towards the reverse transformation from the internal representation back into the "real world" domain of knowledge.

5.2.3 The McCulloch-Pitts Neuron

Designers of machine learning systems have drawn inspiration from the domain of biology and natural organisms. The human brain is clearly a highly-effective learning system, and researchers have therefore felt that there may therefore be merit in reproducing artificially such features of its internal structure as are presently understood.

The McCulloch-Pitts neuron appears in almost all types of neural network and is modelled after a crude representation of a biological neuron. Its essential features include a series of input connections from other neurons; in the special case of an *input* neuron there may be only one input value. For each input connection the neuron stores an associated weight, and its computational function is to multiply each input value by the associated weight, sum the results and transform this value using a non-linear function to produce an output value. In the case of an *output* neuron this is considered an output from the network,

otherwise a neuron's output value is typically fed as input to another neuron.

Within this general framework there is much scope for variations of neuron design to suit particular networks. For example, the particular non-linear function employed is often a matter of implementation convenience, although in general the chosen function must saturate to a finite value for very large positive and negative values of input. In the case of a Hopfield network, a simple step-function is often used, with the result that the neurons have a discrete output with two values.

The usefulness of the biological analogy is much debated, but it can be seen that the elements of the McCulloch-Pitts neuron can be interpreted as representing very approximately the functions of a biological neuron which "fires" as the dendrites of connected neurons communicate a threshold level of electrochemical messenger compound. At best, however, this is a first-order representation - there is evidence to suggest that there are many higher-order effects which influence the firing of a biological neuron. For example, a neuron which has just fired has a characteristic "relaxation time" which must elapse before it can fire again. This is dependent on the levels of messenger compound present in the neuron, and is therefore a complex function of the number and rate of times the neuron has recently fired. Although there have been attempts to model this sort of behaviour artificially, the artificial networks which at present yield the most promising results have diverged somewhat from their biological analogies. For example, as we shall go on to report, multi-layer perceptrons trained by backpropagation have recently been the subject of much successful study, however, there is no evidence to suggest that any process resembling backpropagation actually takes place in biological systems.

5.2.4 The Hopfield Network

The Hopfield network is one of the most structurally straightforward and yet potentially most versatile of all neural networks. It comprises a number of

McCulloch-Pitts neurons which generally have binary outputs, that is, the outputs are constrained to take on one of two possible values, as described. The Hopfield net is typically *fully-interconnected*, that is, each neuron has inputs from each other neuron. There are therefore no well-defined inputs and outputs to the network.

The Hopfield network has some surprising and interesting properties. As the network is "iterated", that is, the output value at each node is re-evaluated based on the new input values, the outputs may, depending on the starting values and the network weights, either progress through a cyclical series, exhibit more chaotic patterns or remain invariant.

One of the most useful applications of the Hopfield network is as a *distributed memory store*. It is possible to select weights for the network so as to "program in" certain stable output states. Furthermore, when iterated the network will tend to converge to the nearest[3] stable output state to the starting condition. Thus the network may be able to reconstruct the full pattern with which it has been programmed when presented with a partially corrupted version. This feature has some application in pattern recognition tasks.

Theory relating to the pattern-matching capabilities of Hopfield networks is well-developed and is based on analogies with Physics theory relating to magnetic materials. The analog of a Hopfield network neuron is a magnetic dipole element which will "flip" according to the coupling conditions of its neighbours.

It is useful to mention the Hopfield network here since it illustrates well one of the most fundamental problems with neural networks. Potentially such a network can perform arbitrary computation tasks - it is fully-connected and yet connection weights may be set to zero, and therefore any logical network structure can be assumed, including that of the perceptron, for example, which we shall shortly describe. The reader can imagine how a binary shift register might be designed using a Hopfield network, and indeed there is some evidence to suggest

[3]The relevant distance metric here is not entirely straightforward, however, it approximates the pattern with the most similar outputs or "bits".

that motor control neurons in certain insects follow such a pattern in order to produce the repeating sequence which causes a centipede's legs to walk in their characteristic manner. One might further imagine control inputs to the Hopfield sequence generator which cause the sequence to stop, start and run backwards, for example. These could be the outputs from some kind of perceptron-like learning engine, also fashioned within the confines of the same Hopfield network.

In summary, it would appear that powerful arbitrary functions with both time-invariant and time-dependent characteristics could be constructed simply by specifying a certain pattern of weights in a Hopfield network. However, the problem is that the theory relating to such networks is not yet able to support such a design procedure. Indeed, the theoretical understanding of the most useful applications currently available with Hopfield comes about only through analogy with another branch of Science.

5.2.5 The Perceptron Compared with Statistical Classification

The perceptron is a type of neural network which is particularly applicable to the task of *classification*, in other words, of deciding to which *class* a particular input belongs. A classifier is an important subcomponent of any machine vision system as we have discussed.

In order to put the features of the neural perceptron into context, we should first examine exactly what is meant by a "classifier". Consider one of the most straightforward conventional techniques for achieving the same thing - the "nearest neighbour" classifier.

Nearest neighbour classification operates using a training set as its basis, just as with the backpropagation-trained multi-layer perceptron. Each item in the set is typically a feature vector, which has already been assigned by some other means to a known class. Each defines a point in a multi-dimensional space which has as many dimensions as there are features in the vector. A new case is classified by

measuring the distance between the point in space represented by that case and each of the examples in the training set - it is assigned the class of the nearest one.

The concept of "distance" in this context requires the selection of a suitable metric. Often this is the straight-line, or Euclidean metric, but others may also be used.

The main limitation of the nearest neighbour classifier is that it is highly-susceptible to the presence of "rogue", or unrepresentative examples in the training set. However, it illustrates well the idea of a decision "surface" in multi-dimensional space. For example, nearest neighbour classification with two examples in the training set implements a linear decision boundary perpendicular to a line joining the points representing the two examples.

The perceptron consists of a single "layer" of neurons, not counting the input layer as is conventional. Each neuron receives a copy of all the network inputs and maintains its own set of weights which it uses to scale the inputs before summing and non-linearising. The weights are adjusted using an error-correcting learning algorithm based on the distance between the actual and expected network outputs.

The perceptron behaves as a trainable classifier. Given a set of examples and associated classes, the perceptron will, after training, arrive at a decision boundary which will differentiate the classes.

When Rosenblatt first proposed the perceptron [28] in 1958 it was as a simple theoretical model of neurological systems in biology. However, it was the subject of much interest in the field of artificial neural networks for some time, since the perceptron can be taught a variety of useful classification tasks. In 1969 Minsky and Papert published their famous book [29] on perceptrons which largely killed this interest as they showed the perceptron to be incapable of solving classification problems requiring *non-linear* decision boundaries, such as the well-known exclusive-or problem.

5.2.6 The Multi-Layer Perceptron and Backpropagation

The principal limitation of the simple perceptron, which is that it can form only linear decision boundaries or *discriminants*, may be overcome by extending the perceptron such that it possesses more than one layer. This means that there is one or more *hidden* layer intervening between the input and output of the network. Potentially such a network is a much more powerful classifier, since it can approximate almost any decision surface where there are sufficient nodes in the hidden layer [30]. *Training* such a network was a particular problem, however, until the development of the *backpropagation* training algorithm in the late 1980s which revived interest in the use of neural networks both as general learning engines and as classifiers in particular. Backpropagation training is, however, considerably more computationally-intensive than the earlier algorithm used for training the simple perceptron, so much so that one can regard it as a search problem in its own right. It would appear that the volume of computation required to achieve the optimum combination of weights is so large as to present a serious problem to application development with the multi-layer perceptron. It is with this in mind that we have sought to parallelise the training algorithm using multiple processors in chapter 8.

5.3 Previous Neural Inspection Systems - MLP Applications

We should now like to review a selection of existing vision systems which have made use of neural networks at the classification stage in order to get an impression of the current concensus regarding their characteristics, performance and suitability; this will serve to put our own work, detailed in chapter 7, into context and illustrate its motivation.

5.3.1 Toothpick Inspection

In [6], Huang et al. used a ready-implemented neural network[4] as a classifier in the toothpick quality control system already mentioned in chapter 2. In particular, a multi-layer perceptron trained by backpropagation was used, configured with 130 input nodes, 8 hidden nodes and 2 output nodes. The output took the form of a simple pass/fail decision. 128 of the input elements were used to provide information representing the image of the part to the network, the remaining 2 being used to "supervise" the network during the training phase, although Huang does not tell us the exact nature of this representation (parameterisation) or supervision.

Huang's conclusion is that, when compared with conventional image-processing algorithms, in particular, classifiers, the neural approach required more development time. However, his findings are that the neural networks outperformed the conventional approach in terms of accuracy.

Huang also suggests that the multi-layer perceptron stores an item of knowledge in a distributed way across many memory units, or nodes. Therefore, he says, the system may have redundancy which allows it to sustain partial destruction. Thus the neural network may be a naturally fault-tolerant system. At present this feature would seem to be of purely academic interest, since typical useful networks possess only a few tens of nodes and are generally simulated on vector processors which have other, reliable means of error correction. However, it may be in the future that this becomes important when complex neural networks are implemented in high-density hardware.

5.3.2 Human Face Recognition

In [31], Evans et al. use a multi-layer perceptron trained by backpropagation as the classifier in an application designed to identify human facial features as a first step towards face identity recognition. Although the authors state that the

[4]Neuralworks II by Neuralware

perceptron is used as the first component in their vision system, they go on to say that the network inputs are "convolved" over a subsampled image at various resolutions as a data reduction step. According to our adopted paradigm, therefore, this amounts to a data preprocessing step. At publication this work was still at an early stage and the system's performance was therefore rather limited. However, the authors are of the opinion that much more can potentially be achieved through tuning and refining the preprocessing and neural classification,

5.4 Discussion and Summary

Artificial neural networks can form a useful element of machine vision systems. Their ability to learn *higher-level* representations from supplied input examples is well-known, and there are numerous reports of the power of neural networks in matching and exceeding the performance capabilities of traditional approaches, for example, Kendall and Hall in [32] constructed a multi-layer perceptron which performed edge extraction on an image with results which compared favourably to more traditional operators such as the Laplacian.

Our primary aim in investigating the capabilities of neural networks is to obviate, as far as possible, the need for human thought in the solution of machine vision problems. Although favourable reports of neural network performance such as [32] do permeate the literature, it is rather dangerous to attach too much significance to comparisons between the two since the outcome naturally depends rather critically on the effectiveness with which each technique is implemented. Clearly a poorly-designed neural classifier is likely to compare unfavourably with a well-designed statistical classifier, for example.

It seems likely that, at the current level of development, neural solutions are more generally found to rank only second-best to "bespoke" vision solutions produced by means of human intelligence. However, in chapter 2 it was concluded that more generally-applicable vision systems would bring benefits as an increasing level of computer power and an enhanced theoretical understanding made

these possible, therefore it seems that neural networks are worth persuing in the name of general applicability, even if there is no prospect of achieving more accuracy than is possible with conventional techniques.

Known disadvantages of the MLP include:-

- The large computational effort required for training. Using desktop PC or workstation equipment, this is typically of the order of hours rather than minutes, for a single network configuration. This means that inspection systems using the described neural approach will take a long time to reconfigure themselves for a new inspection task, or a change in parameters on an already-running task.

- The difficulty in deducing the structure of an MLP which is capable of training in a satisfactory way. Currently there is no established analytical means, for example, of deducing the number of nodes which should comprise the MLP's hidden layer or layers, and therefore empirical methods must normally be used to determine this. If there are two few hidden nodes, it becomes impossible for the network to train, that is, to produce the desired results from the training example input vectors. Too many, and the network will effectively learn the training data "by rote", that is, without inferring the desired generalised rules which allow correct interpretation also of other data. The overall effect is to multiply further the computational burden of training, since many trials will be required to select the optimum configuration.

Chapter 6

Image Preprocessing Considerations

6.1 Introduction

As we have discussed, data preprocessing in order to distil the key features of the defects is vital if a neural network is to make progress towards a defect classification solution. However, a compromise needs to be found between two extremes, one of doing *too little* preprocessing, in which case the network has a very difficult and unconfined problem to solve, and the other of reducing the data to such an extent that possible fast solutions and key factors for recognition have been pre-processed out. It was important to get a feel for the way in which the defect signals might respond to preprocessing techniques in order to ascertain how this compromise might best be made. We will therefore now present our investigations into Iterated Function Series and the Fourier Transform, two promising families of preprocessing techniques.

6.2 Iterated Function Series

A popular approach to general machine image recognition has been, in the past, to analyse the performance of the human visual perception system with a view to imitating its function. In [33] Giles gave consideration to this kind of idea and particularly to the 'visual images' or 'mental pictures' humans use when

imagining objects or scenes that are not currently available for scrutiny.

> *The suggestion is that such visual images are spatial representations in short-term memory that are not simply retrieved, but are in some way constructed from more fundamental representations in long-term memory using conceptual knowledge. [33]*

In other words, it might seem that the human visual perception process gradually 'builds up' in the mind an isomorphic representation of the object to be recognised, that is, "One in which the laws and relationships governing the real world objects are inherent in the data structures and operations of the representation." [34], in order to compare it with the real thing,

Giles dealt in detail with a branch of mathematics known as iterated function series (IFS) in the context of general machine recognition. It appeared from the outset of his work that there could be merit in using the IFS to construct a representation of an image which would, in effect, be a machine equivalent of the human 'mental pictures', in the sense that the representation could be "built up" gradually into increasing levels of detail.

In order to discuss the potential underlying utility of the IFS, it will be necessary to present a brief review of the mathematics on which it is based. A mathematically rigorous treatment is given in [35] and [36]. However, as an incentive to the reader to bear with us through this section, we shall first explain the iterated function series' most interesting features in respect of the context in view.

The IFS is usually regarded as a subset of the relatively novel branch of Mathematics which deals with fractal geometry, and in fact the IFS can be used to construct various well-known fractal patterns such as the fern and the Sierpinski triangle. The reason for its potential relevance to image recognition is that, as with many fractal construction techniques, a small amount of data can specify a highly-complex pattern which is usually built-up over many iterations. It would seem that this property would also make the IFS useful in image compression

applications, and indeed Barnsley went on from his work in [35] to propose novel image compression methods using fractal techniques [37].

We therefore explored the use of the IFS as an image recognition preprocessing operator, that is, one which can reduce the volume of data which needs to be processed by the classification stage whilst retaining the essential components of the image information which allow the classification to be made.

If a suitable method of automatically deriving the IFS from an image could be found, then two possibilities for an image recognition system might be opened:-

- The IFS could be used simply to *compress* the volume of data needed to specify an image in the stored library of the recognition system. It seems that any approach to image recognition will require quite a sizeable library of images with which to compare the view currently being seen, and that in general the storage requirement for this is a severe problem. Therefore the IFS affine transformation coefficients could be stored instead of simple image bit-maps, and the attractors regenerated by the application of the iteration procedure only when required.

- Comparison between the real image and stored images could be carried out *in the IFS coefficient space.* That is, the procedure for deriving IFS coefficients could be applied to the image to be recognised, these coefficients then being directly compared with coefficients in the stored library. It can be seen that if this type of approach were to be viable, a very fast and powerful recognition technique might be devised, since comparisons between a few tens of floating-point numbers can be carried out very much faster than comparisons between large bit-map images.

In general an IFS consists of one or more *affine transformations* of the form:

$$\begin{bmatrix} x^1 \\ y^1 \end{bmatrix} = W_n \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

Where n denotes the reference number of the transformation, (x, y) and (x^1, y^1) are points in the two-dimensional Euclidean space R^2.

It can be seen that any combination of rotation, scaling and translation may be incorporated into an affine transformation by suitable selection of the constants a to f. One requirement for the IFS to function correctly is that each of the transformations which comprise it must be *contractive*, that is, for any set of points in R^2 to which it is applied, the new set of points produced must be *closer together*. Apart from this condition, however, a to f may be arbitrarily chosen as required.

To 'build' the IFS equivalent of the human 'mental picture', an arbitrary starting point is assigned to (x, y). Next, a transformation is selected *at random* from those which comprise the IFS, but according to a predetermined table of probabilities. For example, if the IFS contained four affine transformations, we might have a probability array P as follows:

$$P = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.4 \\ 0.4 \end{bmatrix}$$

Where p_n is the probability of the transformation W_n being selected.

The chosen transformation is then applied to the point (x, y) to yield a new point (x^1, y^1) as shown.

This procedure is *iterated* many times, producing a set of points in R^2. This set is known as the *attractor* of the IFS, and when plotted gives a 2D image which is *always the same*, even though the transformation selection procedure is random. A proviso must be made, however - the first ten or so points generated by the IFS are defined *not* to belong to the set in R^2 which constitutes the attractor.

n	a	b	c	d	e	f	p_n
1	0	0	0	0.16	0	0	0.01
2	0.2	-0.26	0.23	0.22	0	1.6	0.07
3	-0.15	0.28	0.26	0.24	0	0.44	0.07
4	0.85	0.04	-0.04	0.85	0	1.6	0.85

Table 1: IFS parameters for generating the 'fern'

6.2.1 Experimental

An application was coded using Microsoft C on an 80286-based PC to construct patterns using the IFS based on coefficients input by the user. This was used to generate the patterns shown in figure 16.

Our main concern regarding the IFS was that the mapping of parameter space onto image space might prove to be itself *chaotic*, for example, the production of a certain pattern by the IFS might depend very sensitively on the accuracy of IFS coefficient values - a slight alteration to one of the coefficients would cause the production of a completely different image. If this were to be true, it would detract seriously from the possibilities of classification of image features based on the IFS coefficients which construct them.

In order to investigate this potential problem, we compared the resulting images when a single IFS parameter is gradually varied. Table 1 shows a set of IFS parameters which produces the 'fern'-like attractor shown as the bottom left image in figure 16. The remaining eight images show how the attractor changes when one parameter in one of the IFS transformations is gradually altered, in this case the value for e in W_3. Each successive image was generated with an increase of 0.2 for this parameter, so the top right image has a value of 1.6 for e in W_3.

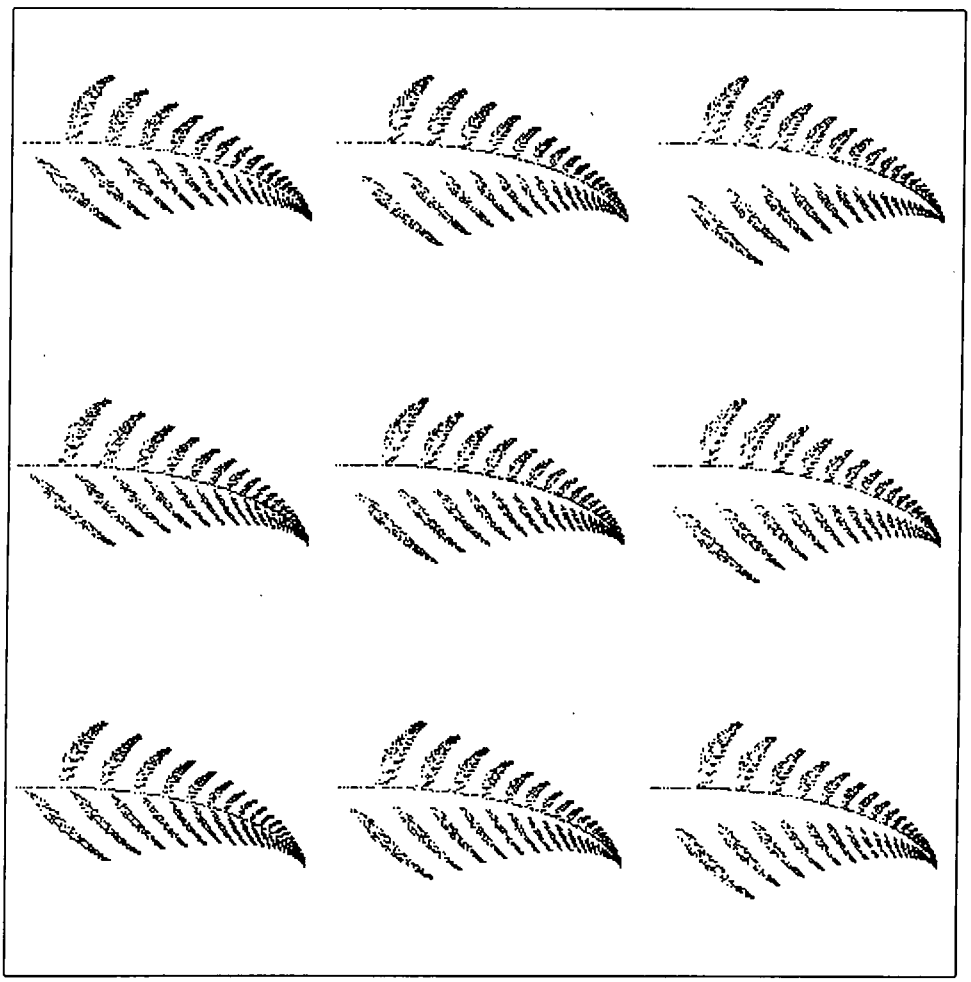


Figure 16: A set of IFS attractors, one parameter being varied

6.2.2 Discussion

Figure 16 shows that in this case the reconstructed ferns vary smoothly and continuously as the parameter e in W_3 is changed, and further investigation demonstrated that this was generally the case, and our concerns about the potentially chaotic nature of the IFS parameter space proved to be mainly unfounded. A remaining difficulty, however, is the task of making the reverse transformation from an arbitrary image shape to a set of IFS coefficients.

Giles in [33] sought a reliable technique for doing this, using as his starting point proofs which state that an IFS exists which will reconstruct an arbitrary shape as its attractor, and the "Collage Theorem" proposed by Barnsley in [37]. Here the procedure is to find a set of transformations which will shrink distances and cause the target image to be approximated by the union of the affine transformations of that image.

Among the methods investigated by Giles in [33] were "boundary matching" techniques. However he concluded that:-

> *The problem of finding a full two-dimensional collage cannot be avoided by this approach, and that it is necessary to develop an algorithm for directly obtaining full collages.*

However, as the fern results suggest above, the iterated function system is *robust*, that is, the attractor solution space is non-chaotic, and so small perturbations in the code will not result in unacceptable damage to the image. Therefore it might seem that an adaptive algorithm might be used to seek out a suitable IFS for an arbitrary image, iteratively selecting guesses and coming closer to the optimum solution.

In parallel with our own work, progress has been made in using the IFS as a basis for fractal compression of general, colour images. Jacquin in [13] proposed a practical technique for finding the set of contractive transformations under which an arbitrary image remains invariant. Iterations of this transformation set upon any starting image will converge to the stored image, and thus the stored image is

defined by the transformation parameters. Use of the IFS appears to be attractive as a compression technique since it can achieve arbitrary compression ratios, and hence reproduction qualities, depending on the accuracy with which the transformations are devised. Furthermore, the property of gradual convergence to the stored image during many iterations can be useful for certain applications where it is desirable to achieve a rough, low-resolution approximation to the stored image early in the decompression process, with additional processing being used to gradually refine the image and achieve maximum resolution. The origin of the data compression comes about through the IFS' ability to define parts of an image in terms of other parts, and consequently the *compressibility* of an image, defined by the picture quality or error obtained at a certain compression ratio, will vary as the degree of self-similarity in the image varies.

A remaining problem, however, is that of *uniqueness* - although we have discovered that the iterated function system is robust, there is no evidence to suggest that there is a one-to-one mapping between image and transform coefficient space. Indeed, it appears intuitively that many sets of transforms might be found under which a given image remains invariant, and it would seem therefore that the IFS is not necessarily *unique* in the sense that a particular image pattern does not always map to the same position in IFS parameter space.

6.3 Fourier Transform

We next investigated the usefulness of the Fourier Transform as an image preprocessing operator. As we have already mentioned in chapter 2, reversible transforms are potentially useful if defects are more easily detected in the transform domain, even though, by definition, no data reduction is brought about. We made use of the remote processing application whose development is described in section 4.2.1 to investigate whether the Fourier Transform can be of use in detection of the aluminium plate defects in view.

Figure 17 shows an *average* one-dimensional transform result for the "bigvoids"

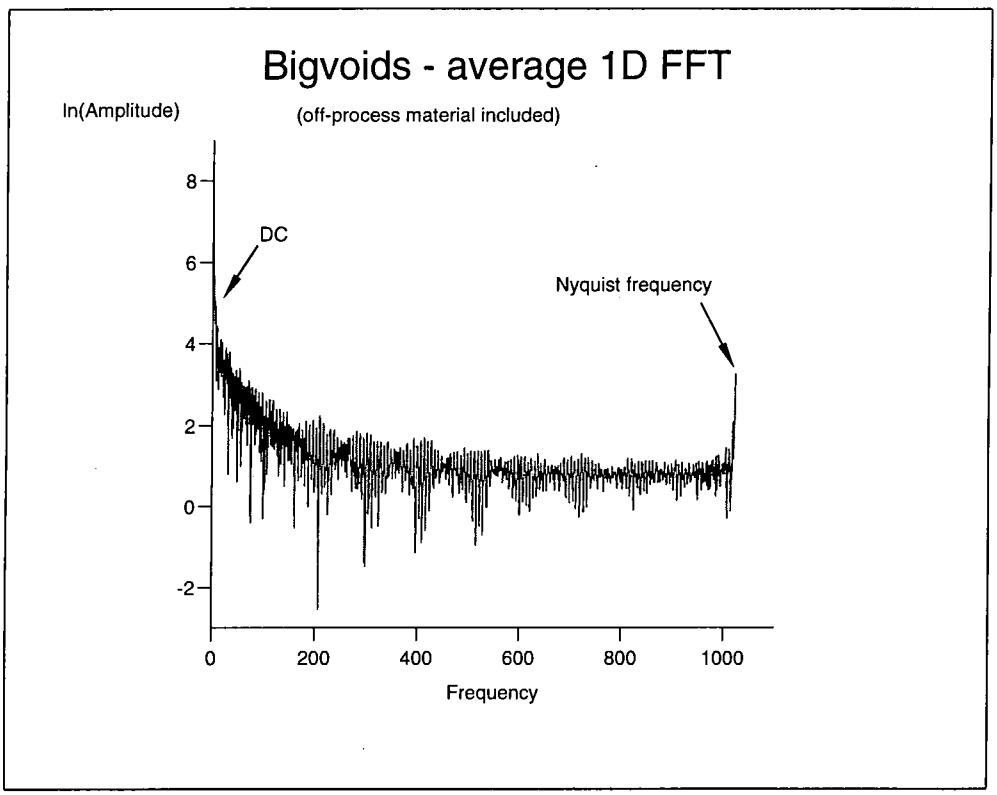


Figure 17: Average over one-dimensional FFT for all rows in "bigvoids" image, showing amplitude

image, the most significant section of which is shown in image form in figure 32 (page 133), and in full as a three-dimensional profile in figure 47 (page 150). The discrete version of the Fourier Transform known as the Fast Fourier Transform (FFT) was used to produce these results. The FFT is a complex-to-complex transform, that is, a series of complex numbers in the spatial or time domain is converted to a series of complex numbers in the transform or frequency domain. Our image data in the spatial domain is, however, not complex, we therefore feed it into the FFT setting all the imaginary coefficients to zero.

The FFT output under these circumstances is, nonetheless, complex, and is best represented in *polar* rather than *rectangular* form. The plot shown in figure 17 shows one half of this data - the *magnitude* or *amplitude* of the series.

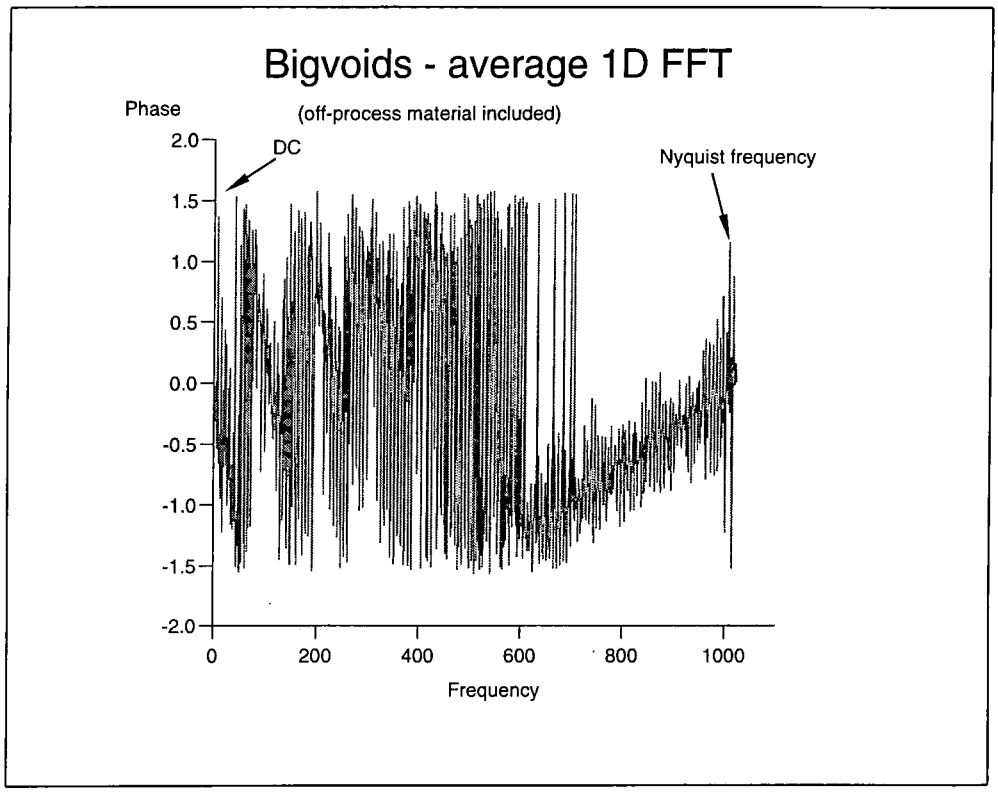


Figure 18: Average over one-dimensional FFT for all rows in "bigvoids" image, showing phase

Figure 18 shows the opposite half - the *phase* of the series. In both cases data is only plotted up to the Nyquist limit on the frequency axis, which is in fact only half of the data returned by the transform. However, for real-only

(non-imaginary) time- or spatial-domain data, both amplitude and phase data are mirrored about the Nyquist frequency and for clarity this mirroring is not reproduced here.

The amplitude plot of figure 17 shows a *ringing* effect which is attributable to the presence of the two contrast boundaries between off-process and on-process material. This can also be seen as a phase *ramp* towards the higher-frequency end of the phase plot in figure 18. This trend is superimposed on $\frac{1}{f}$ noise which originates in the random texture of the material as well as in the digitisation process itself.

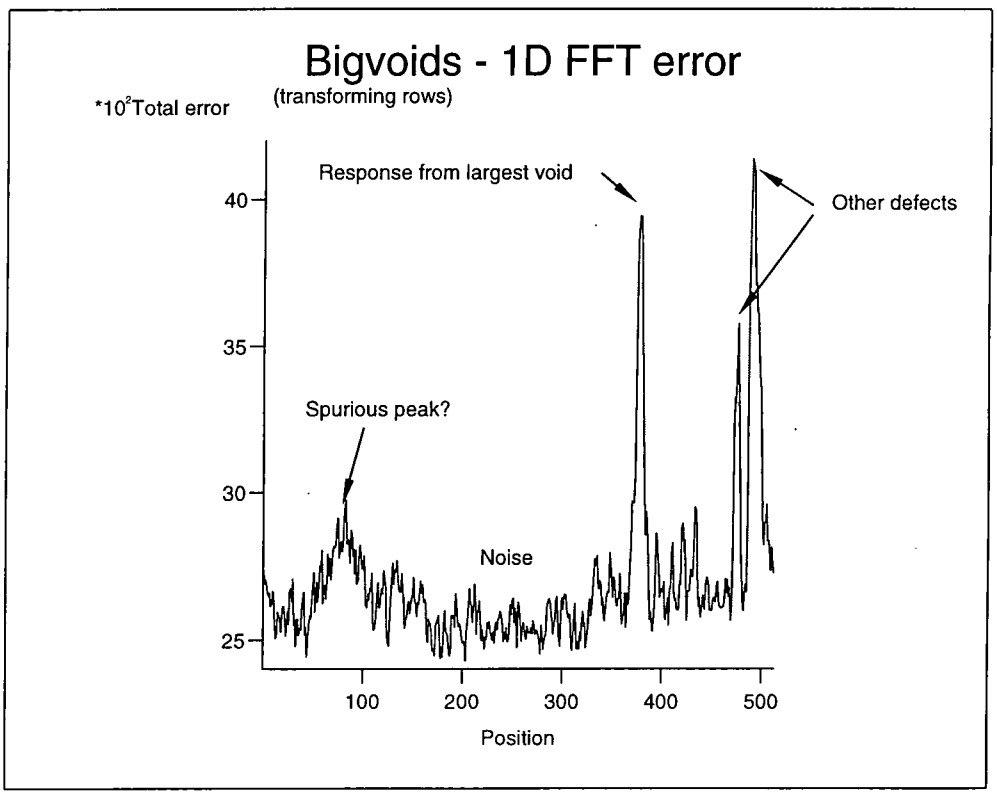


Figure 19: Detection performance based on line-by-line deviation from FFT average in figure 17

We sought to detect defects by summing differences between elements of series produced by frequency-transforming individual horizontal cross-sections of the "bigvoids" image and the corresponding elements in the average series shown in figures 17 and 18. This was done on a one-dimensional basis, that is, magnitudes only were subtracted, and no consideration was given to the phase information.

The sum of these differences produced a single one-dimensional error value corresponding to each horizontal slice, and these are plotted with respect to vertical position in the image in figure 19. It can be seen that this technique is very effective and produces a response with a signal to noise ratio much higher than that obtainable through simple convolution filtering and thresholding.

Figure 19 also shows a smaller peak which we have labelled as "spurious" in the context of detection of the "void" defects. This corresponds to a slightly altered texture at this position caused by a roller "chatter mark", the second type of relevant defect described in section 2.4.1. It is therefore apparent that this one-dimensional FFT technique can detect both sorts of anomaly fairly successfully.

Conducting a one-dimensional FFT on each row of the image is a computationally-intensive process, and in order to examine whether the transform itself is a valuable part of the detection scheme devised, we performed a comparison with a scheme in which pixel values are used directly in the place of the transform coefficients.

Figure 20 shows the average raw pixel values taken over each horizontal slice in the image, including off-process material. For comparison with a single horizontal slice including a "void" defect, see figure 1 (page 21).

Figure 21 shows the results when the same summation of differences between individual slices and the average slice is applied to the raw pixel data. It can be seen that again there is a clear response to the presence of defects and the "chatter mark" peak also appears.

6.4 Conclusions

It appears that the signal to noise ratio achieved by frequency transforming pixel data before summing differences is not significantly improved by the frequency transforming process, as a comparison between figures 19 and 21 shows. We therefore conclude that the extra computational expense of the FFT is not justified in this context.

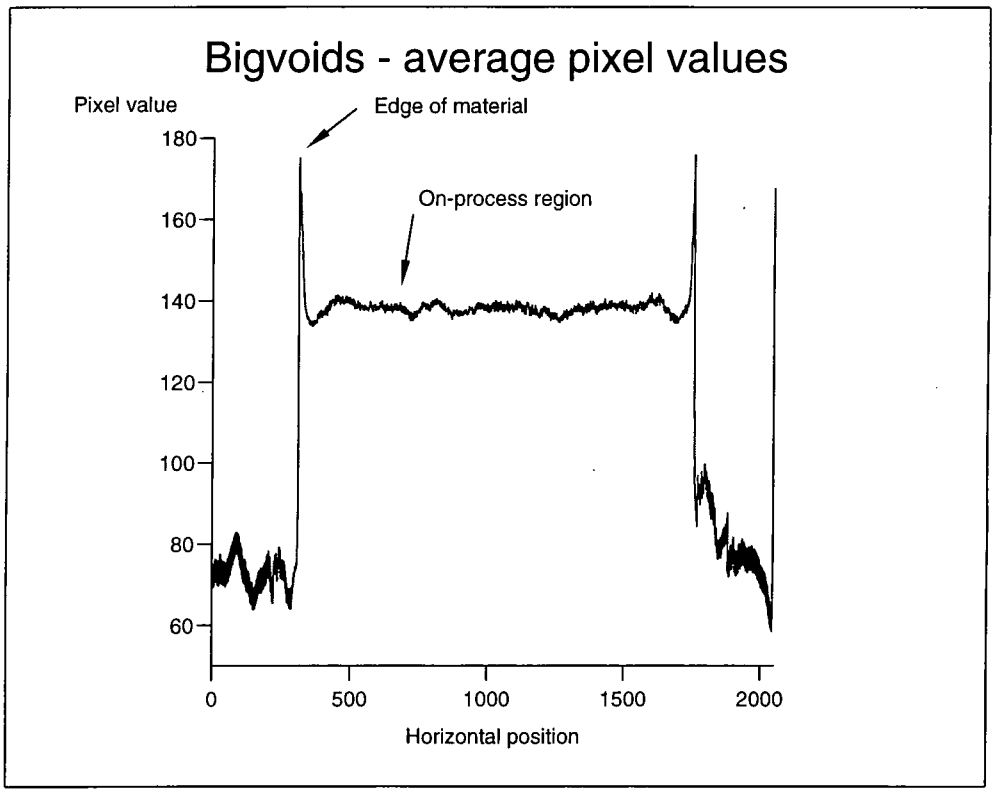


Figure 20: Average raw pixel values over all horizontal cross-sections in "bigvoids" image

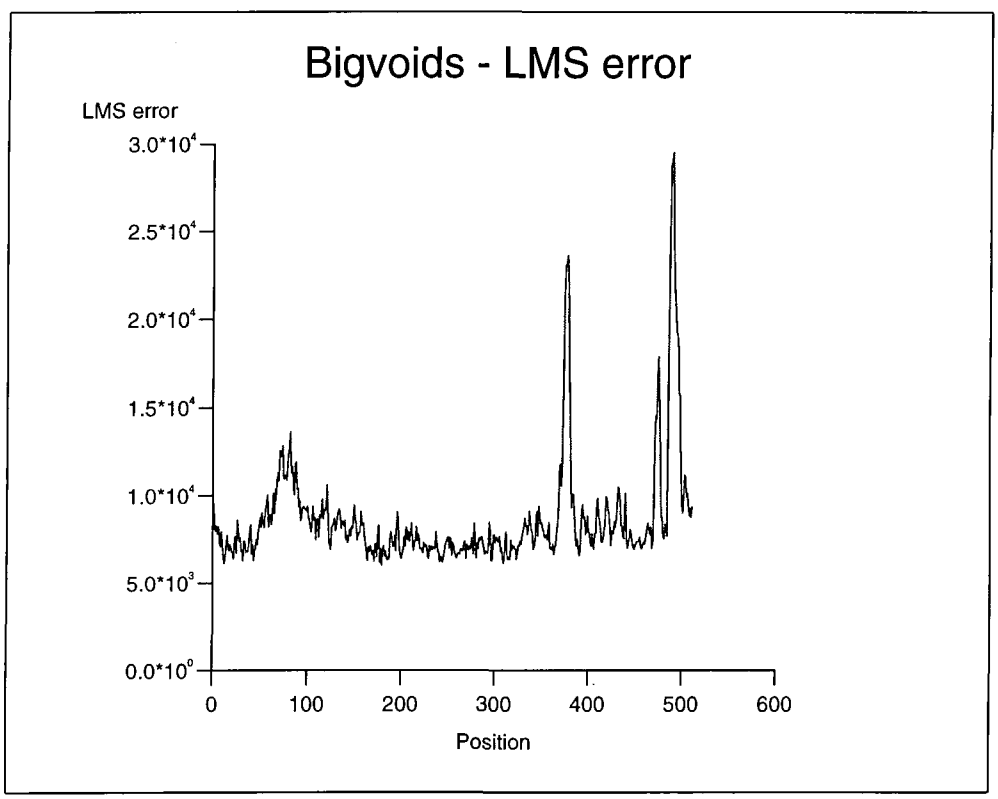


Figure 21: Detection performance based on line-by-line LMS deviation from average in figure 20

The IFS, on the other hand, although now well-established as a useful compression technique, seems to suffer the drawback of *non-uniqueness* which prevents it from being directly useful as a preprocessing operator in a recognition application. However, a related operator known as the *fractal dimension* [35] *is* of potential use, since this is a *metric* of the degree of self-similarity present in an image or portion thereof, representing this as a one-dimensional value. It can be seen this might be usefully used to provide a single feature vector element as input to a classification stage; intuitively there would seem to be a high degree of orthogonality between the fractal dimension of an image portion and the output of more conventional operators such as RMS power, for example. However, the fact that the fractal dimension *is* one-dimensional for a two-dimensional image means that it might be used as a component, but not as the exclusive basis for, classification. An interesting development here is the work by Arduini et al. [38] on "multifractals" - here a two-dimensional *function* is derived from a texture, although the functions produced by Arduini's work appear closely related, and so true two-dimensionality cannot be said to have been achieved, although textures which have identical fractal dimensions can be differentiated using Arduini's technique. This could be a useful direction for further work.

For the moment, however, we leave the IFS and fractal geometry, although there are clearly avenues still to be explored, and the Fourier Transform, since its additional computational expense does not appear to be justified, and in chapter 7 we use only very basic preprocessing in order to focus our study on the behaviour of the neural classification engine itself.

Chapter 7

Neural Network Experimental

7.1 Experimental

7.1.1 Backpropagation Simulator

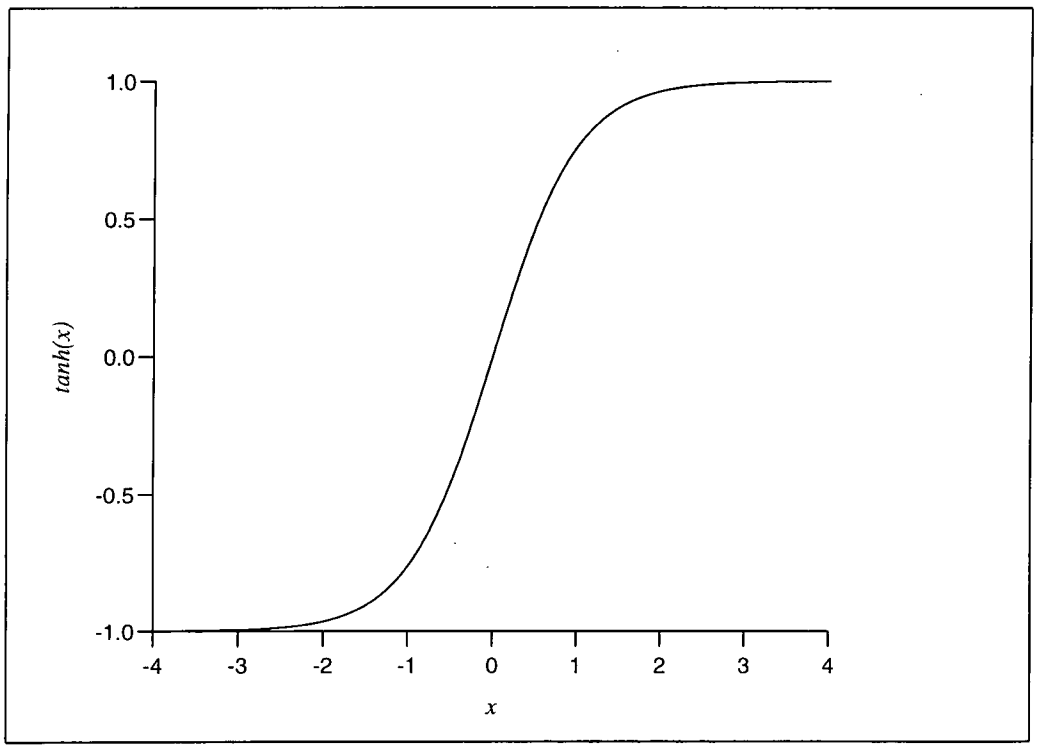


Figure 22: Non-linearity transfer function $tanh(x)$

A multilayer perceptron using the well-known backpropagation training algorithm was simulated using 'C' and the Gnu Project freeware compiler. At each node

in the output and hidden layers, the network employed identical McCulloch-Pitts continuously-valued neurons utilising the hyperbolic tangent nonlinearity function whose characteristic is illustrated in figure 22. This is currently popular for practical simulations, since this function's derivative may be expressed in terms of itself, thus:-

$$tanh'(x) = 1 - tanh^2(x) \tag{11}$$

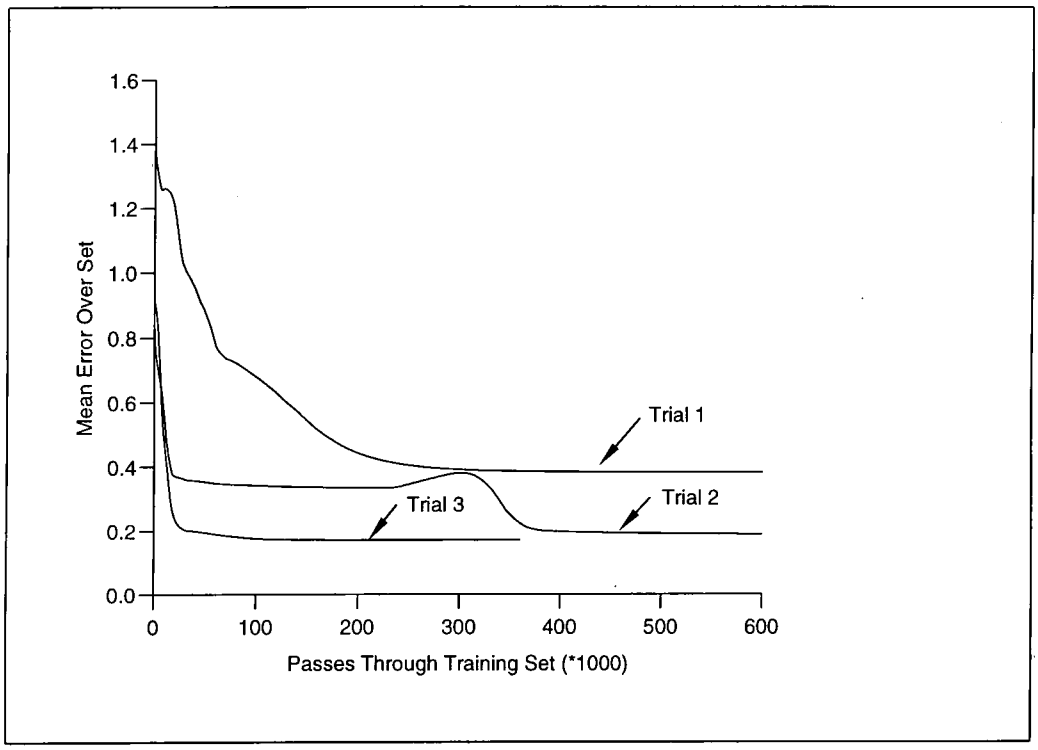


Figure 23: Training performance for MLP trained by backpropagation using training set of 10 random input/output relations, training parameter η=0.001. Network has 5 input, 3 hidden and 5 output nodes.

Our simulation ran initially on a Sun Microsystems IPC workstation with a 25 MHz Texas Instruments Sparc processor. With typically 98% CPU availability for the task and maximum compiler optimisation, training required of the order of tens of hours. Figures 23 and 24 show training performance for various trials during our software testing and validation phase. In both cases the ordinate variable represents the mean error over the whole training set, that is, the sum over the set of the differences between expected and obtained results at the output

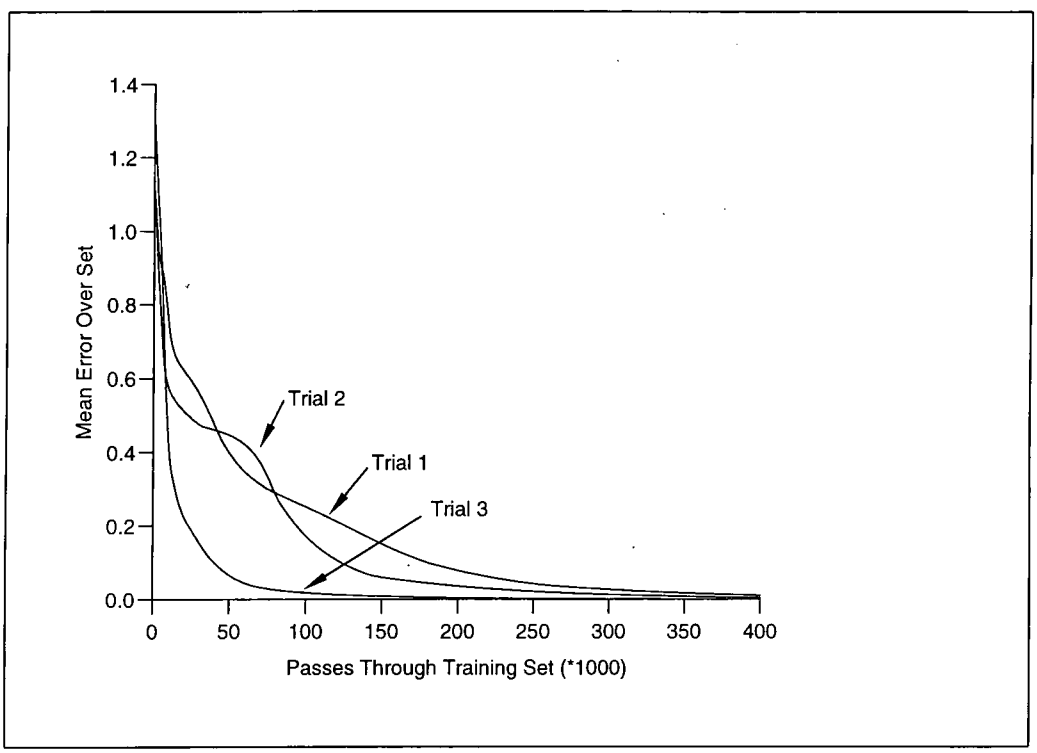


Figure 24: Training performance for MLP trained by backpropagation using training set of 10 random input/output relations, training parameter η=0.001. Network has 5 input, 7 hidden and 5 output nodes.

node for each training vector. In all trials there there are five input and five output nodes, the two figures highlighting the effect of increasing the number of hidden nodes from three to seven. The training input/output vectors are randomly generated in each case.

It can be seen that the mean error evolves differently for each trial, even where network structure and training data are the same. This is due to the fact that network weights are initialised to random values at the start of the training phase, in other words, the search for a solution starts at a random position in weight-space.

Figure 23 shows that the network is unable to learn a random training set successfully when only three nodes are present in the hidden layer. This is as is expected from basic information theory—since the network in question is a *feedforward* network, the values at the output nodes can be expressed exclusively in terms of those at the hidden nodes. If training had been successful, this would have implied that for each vector in the training set, the five floating-point output values could be inferred from *three* values in the hidden layer. This is tantamount to *data compression*, and indeed the multilayer perceptron with backpropagation training has been used in this application. In this case, however, the training data is random and there are therefore no underlying trends or correlations which may be exploited to achieve compression.

7.1.2 Single "Void" Detection using Simple RMS Processing

Description

Our next objective was to further demonstrate the validity of our backpropagation training implementation and we therefore chose a simple problem which would show this clearly. In this first experiment, each training vector consisted of 204 floating-point values, each representing the root mean square power of a

contiguous block of ten eight-bit pixels, these being obtained from a full horizontal cross-section of an image, each being 2,048 pixels in width. Compared with presenting the raw image data, this preprocessing had the effect of constraining the training problem to a much more manageable size.

The network comprised:-

- An output layer consisting of a single neuron, from whose output a straightforward defect / no defect decision can be inferred. It was anticipated that more nodes would be added to this layer in order to make classification as well as detection possible in future experiments.

- A hidden layer consisting of seven neurons. This was found empirically to be the minimum number which allowed the example data to be trained into the network. Care was taken not to overpopulate the hidden layer in order to avoid overfitting of the training data and hence poor generalisation performance.

- An input layer comprising 204 fixed-value nodes. Strictly these are not neurons since there are no associated input weights or non-linear operators, the values being simply fixed to that of the applied input vector. Current trends are to discount the input "layer" and to term a network such as this a "two-layer" network.

Figure 25 shows the layout of the described network.

Training data consisted of two sets, each consisting of ten example parameter vectors. The ten "defect" vectors were obtained from different parts of one particular void example. A cross-section of raw intensity data showing a typical void with characteristic bright centre and dark fringe is shown in figure 26. The result of RMS preprocessing on this data is shown as figure 27. The ten "non-defect" vectors were obtained from randomly-selected portions of good material. An example of such a cross-section after RMS processing is shown as figure 28.

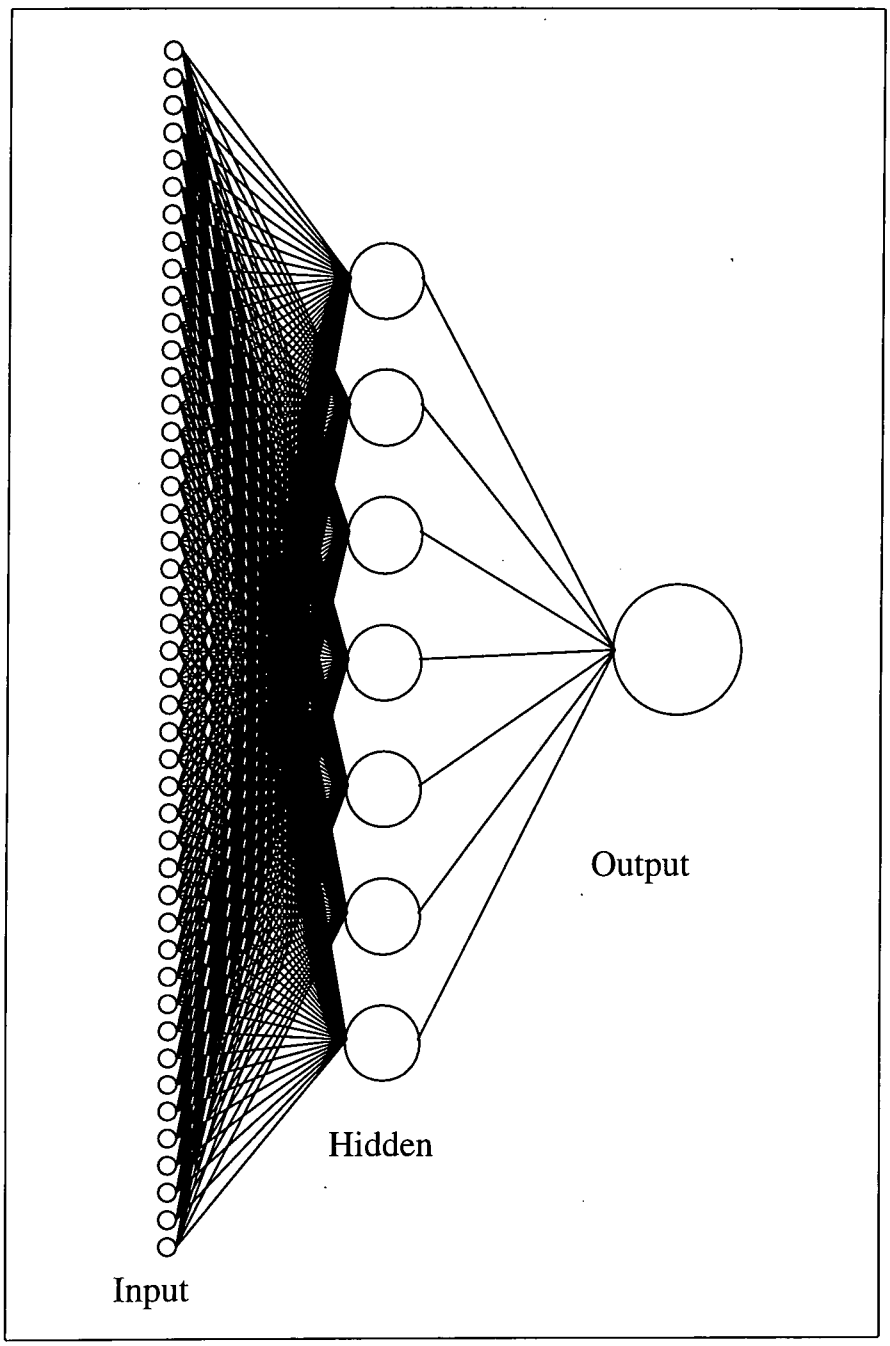


Figure 25: Topography of the Three-Layer MLP employed in RMS-processing experiments.

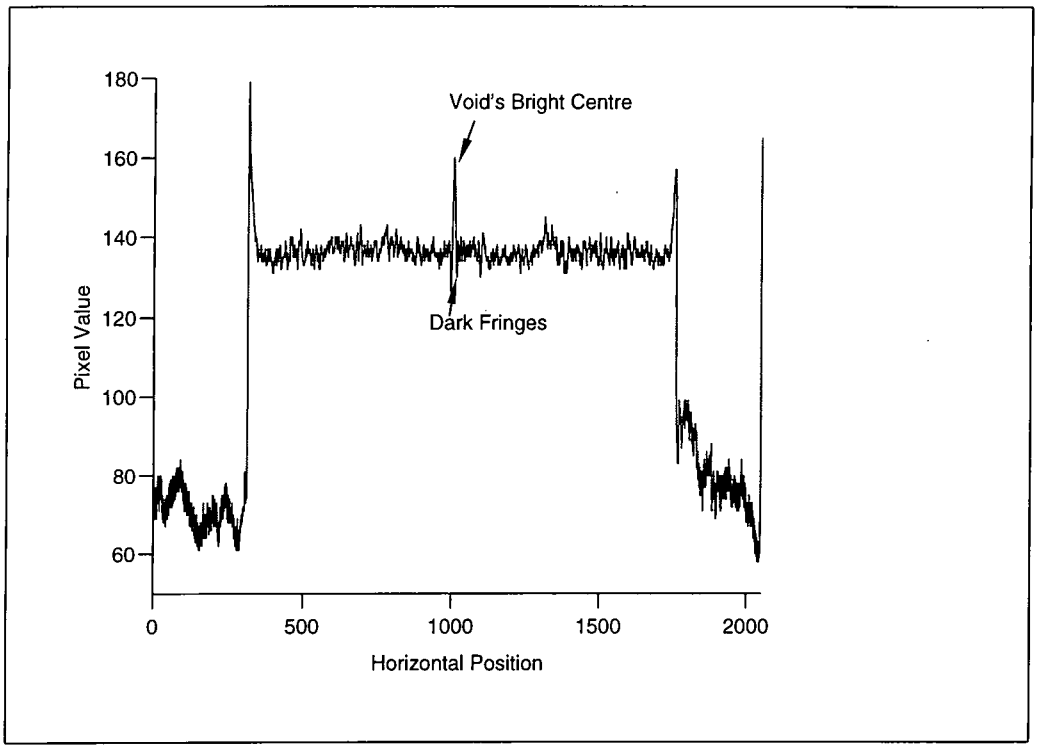


Figure 26: Raw data cross-section showing typical "void" defect on sheet aluminium surface.

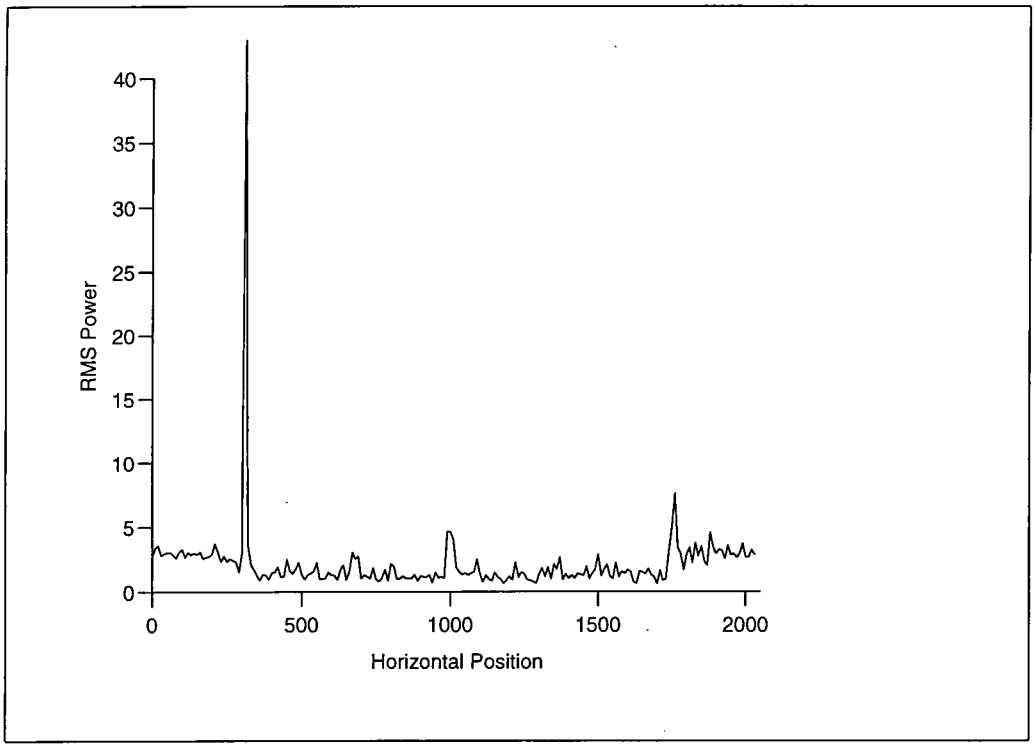


Figure 27: Raw data from figure 26 after RMS processing, reducing 2048 pixels to 204 floating-point power representations.

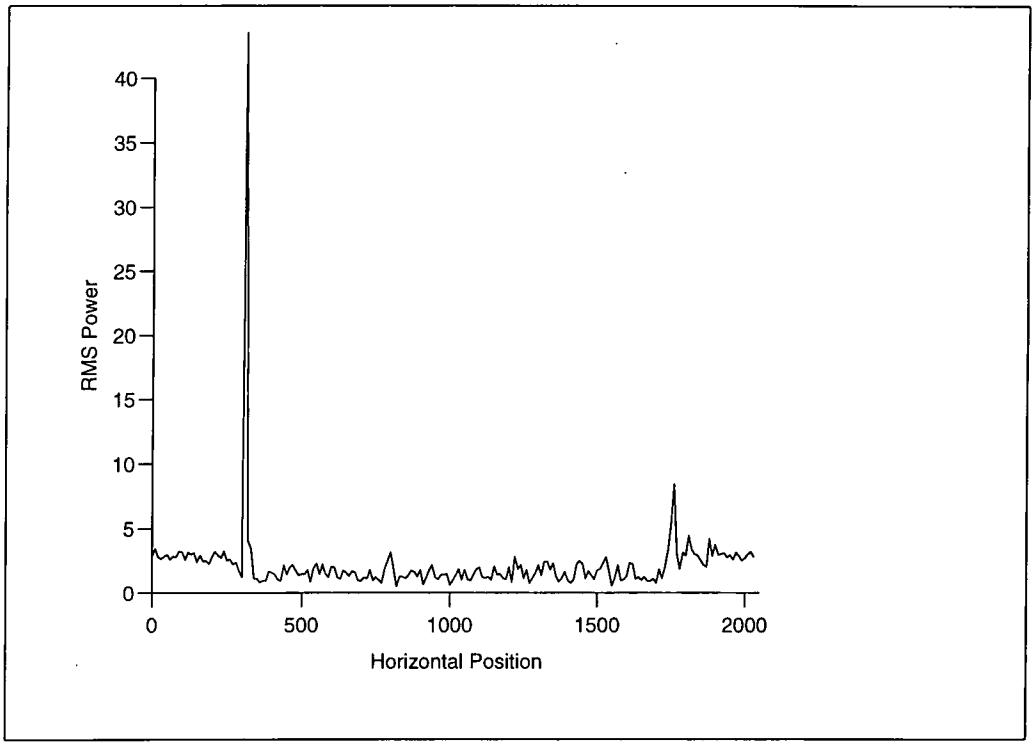


Figure 28: As figure 27, "void" cross-section replaced with data from "good" material.

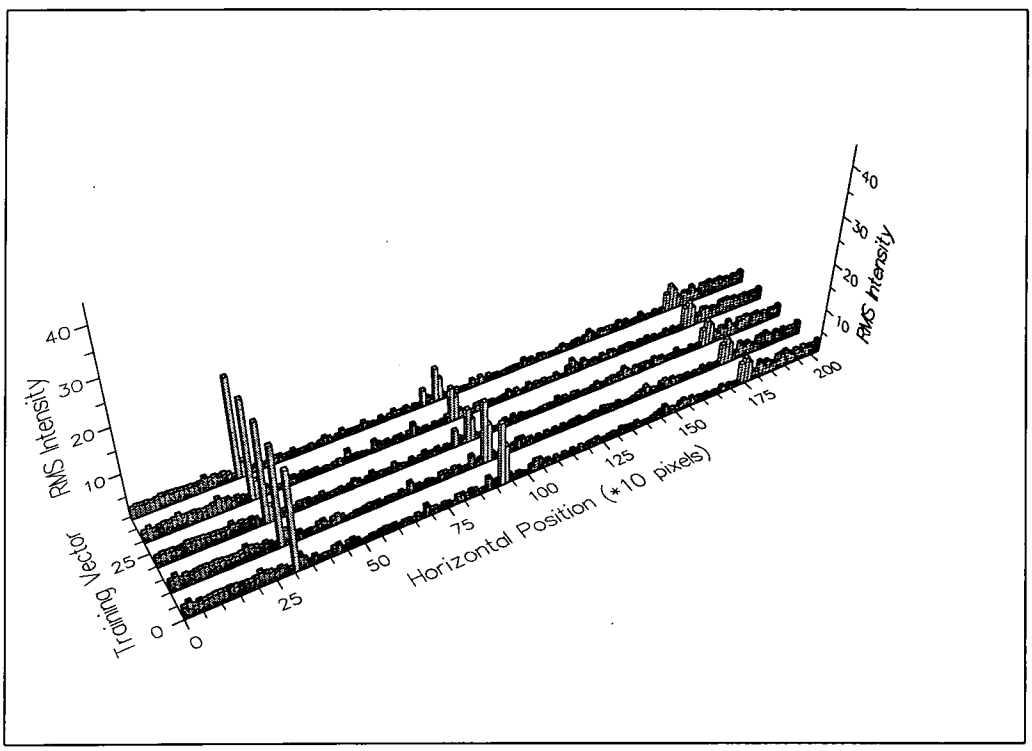


Figure 29: First half of the training set used, consisting of 5 randomly-obtained RMS-processed cross-sections taken across the same "void" defect.

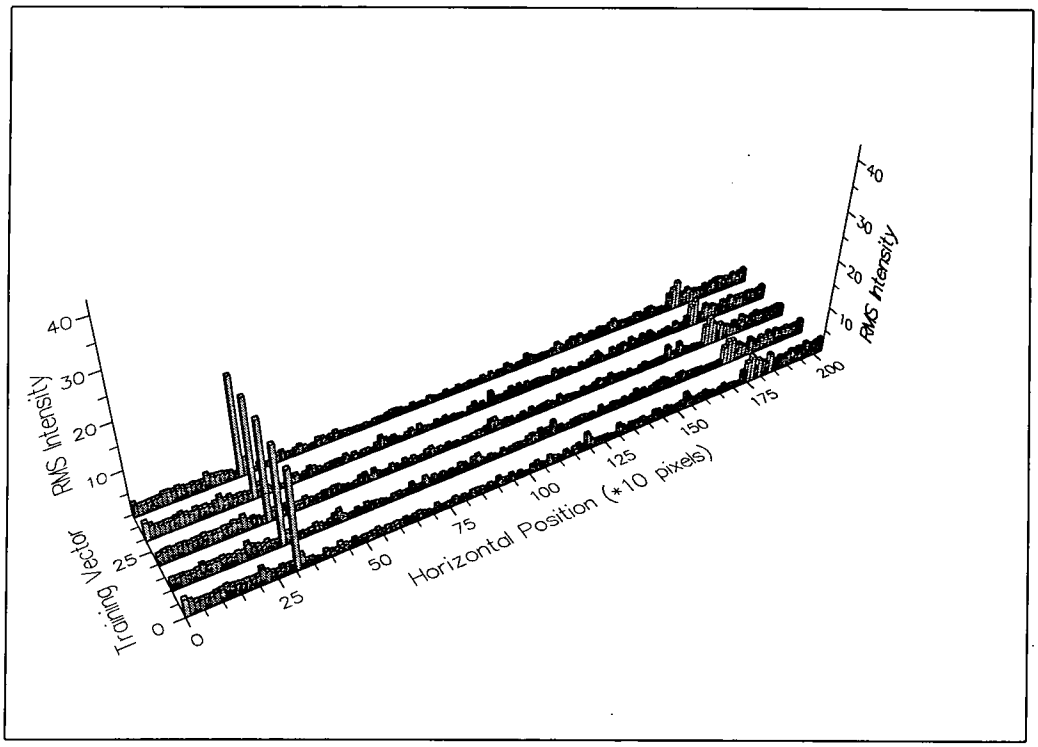


Figure 30: Second half of the training set used, consisting of 5 randomly-obtained RMS-processed cross-sections taken from "good" material.

The actual "defect" and "non-defect" training sets used are shown as figures 29 and 30 respectively.

Basic batch backpropagation training was applied, with the desired result at the output neuron set respectively to +1 and -1. No attempt was made at this stage to speed up the training process by means of algorithmic refinements such as the addition of momentum or gradient terms, or through ongoing adjustment of the stability variable η; this was fixed at 0.001.

Results and Discussion

Figure 31 contains two traces which show the evolution of the mean error for the two halves of the training set as the training process is iterated. It can be seen that many tens of thousands of passes through the training set are required before a mean error close to zero is achieved.

Figure 32 shows a portion of the original image alongside a representation of

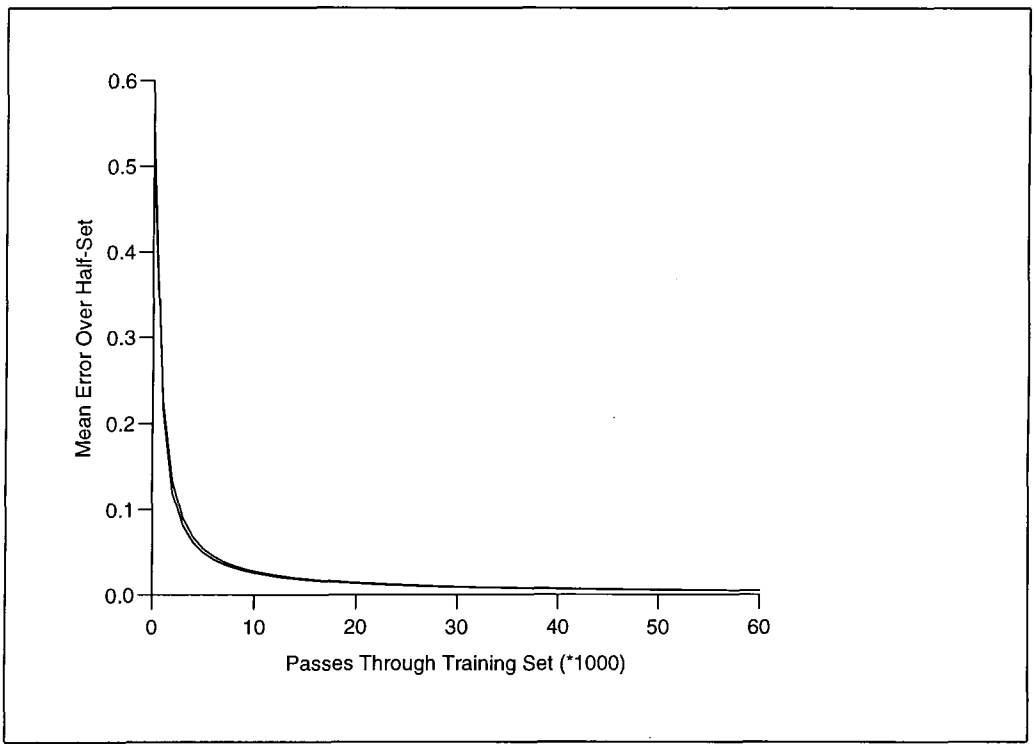


Figure 31: Training performance for MLP trained by backpropagation using training set consisting of two subsets each of five training examples, the first consisting of RMS-processed data taken from randomly-chosen cross-sections across the same "void" (see figure 27), the second from randomly-chosen "good" material (figure 28). Training parameter η=0.001. Network has 204 input, 7 hidden and 1 output node(s). The two training subsets corresponding to defect/non-defect data were associated with values of +1 and -1 respectively at the output node.

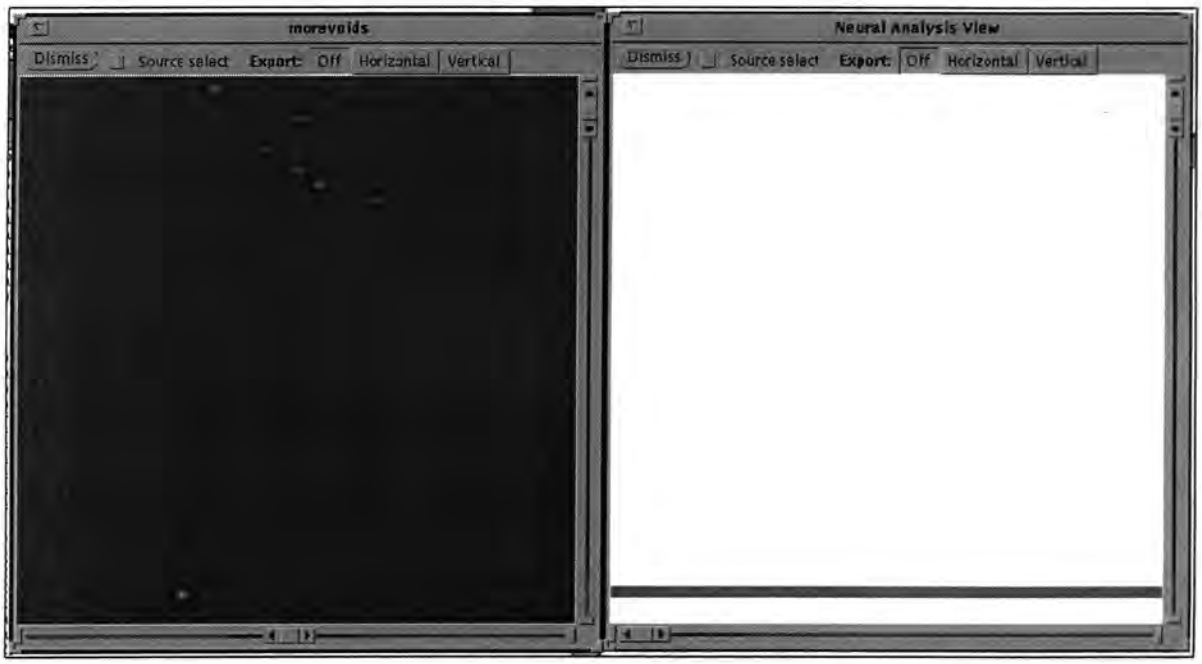


Figure 32: Neural network analysis of fresh data after training on samples from void in bottom left

the trained network's corresponding line-by-line analysis. To obtain this, each horizontal line of the image was preprocessed using the RMS operator in exactly the same way as the training set data. The power values thus produced were then assigned to the input layer nodes and propagated forward using the weights found by training such that a floating-point result is obtained at the single output neuron. The output values, that is, the network's defect / no-defect decisions on each horizontal strip are represented here as a continuous line of colour in the "Neural Analysis View" window, laterally adjacent to the appropriate section of image. Although the output node's $tanh(x)$ non-linearity function permits in theory a continuous range of decision values from -1 (no defect) to +1 (defect), in this case all output values were found to lie within 1% of one of these extremes. All the results have therefore been represented with one of only two colours—white corresponds to a "no defect" decision, grey to a "defect" decision.

The "morevoids" window shows a 512 pixel deep by 512 wide section taken from the 512 by 2,048 pixel original, and is therefore one quarter of its size. In order to produce the results shown, the network has therefore included unshown

data in its processing. However, the unshown portions of the image contain no significant features and it is therefore safe to assume that the network's decision-making is oriented around the features visible in figure 32.

The printed reproduction of the original image section shows *false contouring*, these are the sharp contrast boundaries responsible for the "mottled" effect evident in the background. The contours are false in the sense that they are not really present in the original data—they are artefacts of the laser printing process used, which has only 16 print densities available with which to represent an 8-bit greyscale image. Additionally, the image has been contrast-stretched in order to make the voids easier to see. This makes the false contouring effect more pronounced.

The network's analysis shows a "defect" decision over a narrow band corresponding to the position of the void used in the training set, with a "no defect" decision elsewhere.

It is evident that, although the training phase has allowed the network to successfully learn the required response for each vector in the training set, the performance on fresh data is unsatisfactory, since the collection of rather marked voids in the top right of the image section has been labelled "no defect". The network has therefore largely failed to extract any information from the training set which would allow it to solve the problem more generally. However, a limited amount of *generalisation* has taken place, since the network has successfully labelled cross-sections from the training void in the bottom left, which were *not* in the training set, as being defective.

Since each element of the input vector corresponds to the power level at a specific horizontal position in the image, it seems reasonable to conclude that the trained network has simply learnt to identify defects using a simple rule based on position—in effect, that significant RMS power in a specific location should give rise to a "defect" classification, and that high power elsewhere, for example that caused by the interface between the bright material and its carrier, should be disregarded.

7.1.3 Verifying the "Position-Sensitive" Hypothesis

In order to verify our hypothesis we sought to repeat the results using a different training set, network topography remaining unchanged.

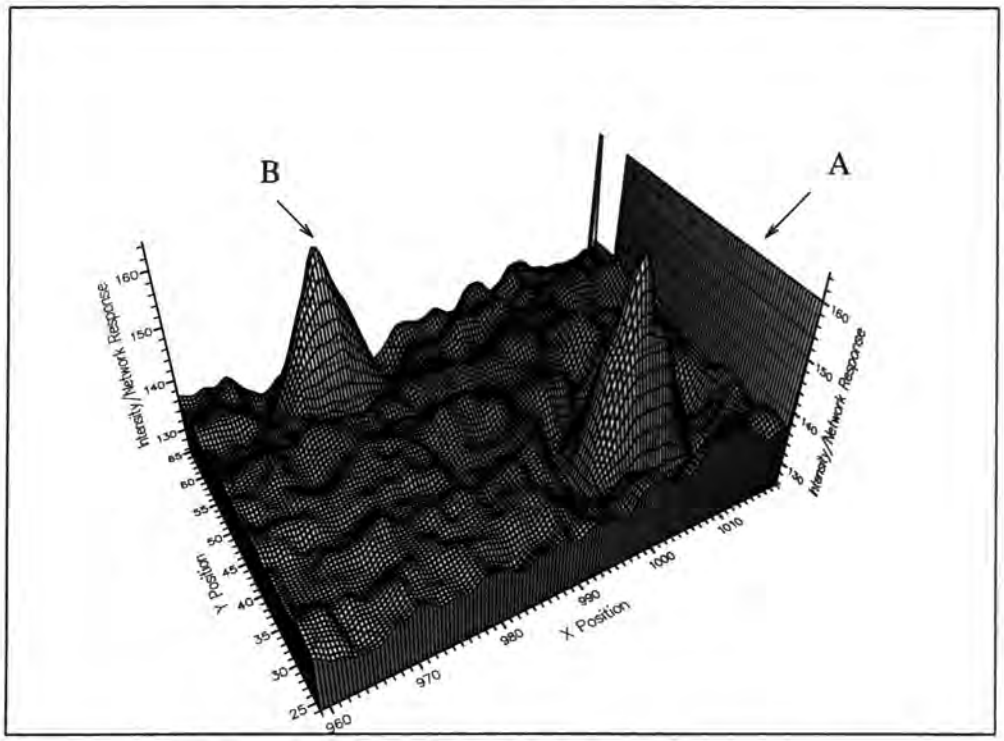


Figure 33: Indeterminate results when attempting to confirm the "position-sensitive" hypothesis

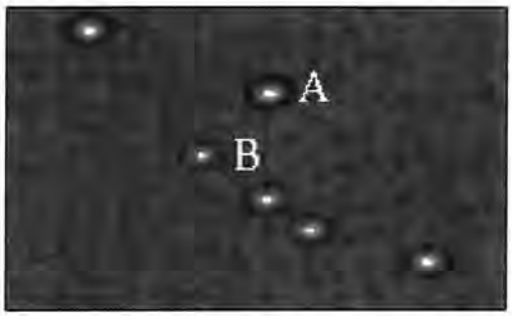


Figure 34: Close-up of voids cluster, those used in training set labelled A and B.

Figure 33 shows two voids from figure 32 represented as a three-dimensional surface plot, the Z variable corresponding to intensity. Figure 34 shows a close-up of the voids cluster with those in question labelled A and B to enable identification. Note that these two diagrams are inversions of each other about a central

horizontal mirror line, since, due to different plotting conventions in the two graphical environments, the points nearest the origin in figures 33 and 34 are in the bottom left and top left respectively.

The strip of data in figure 33 at the maximum limit of the "X Position" axis represents the verdict of the neural network corresponding to the entire 2,048-pixel horizontal row of data at that Y Position, preprocessed by the RMS operator as before. This gives a much clearer visualisation of the original data and results than is possible using a laser-printed greyscale representation. For viewing convenience, the "defect" and "non-defect" results are represented on the Z-axis as values of 160 and 140 respectively.

In this experiment a training set with five examples each of "defect" and "non-defect" data cross-sections was again used, in this case the former being randomly-chosen slices across the defect labelled A in figure 33, the latter being data obtained at random from elsewhere in the image.

It can be seen that the network correctly classifies the entire extent of void A as being defective, whilst the central portion of void B is labelled as non-defective. This lends further credence to our hypothesis that the network is simply learning to regard certain horizontal positions in the image as being particularly significant, however, the result is not conclusive since the network fails to correctly classify the material *inbetween* voids A and B as being non-defective.

Our interpretation of this eventuality was that the the network's poor generalisation performance might be attributable to detection by the network of other, incidental trends in the training data, causing it to respond accurately to training data using some *alternative* means to distinguish "defect" data from "non-defect" data, that is, without detecting the high RMS powers due to the voids as expected.

We sought to test this extended hypothesis by repeating the experiment, all conditions and conventions being maintained constant, with the exception that the "non-defect" training set data was in this case obtained from cross-sections *inbetween* the voids A and B. The new results are shown as figure 35.

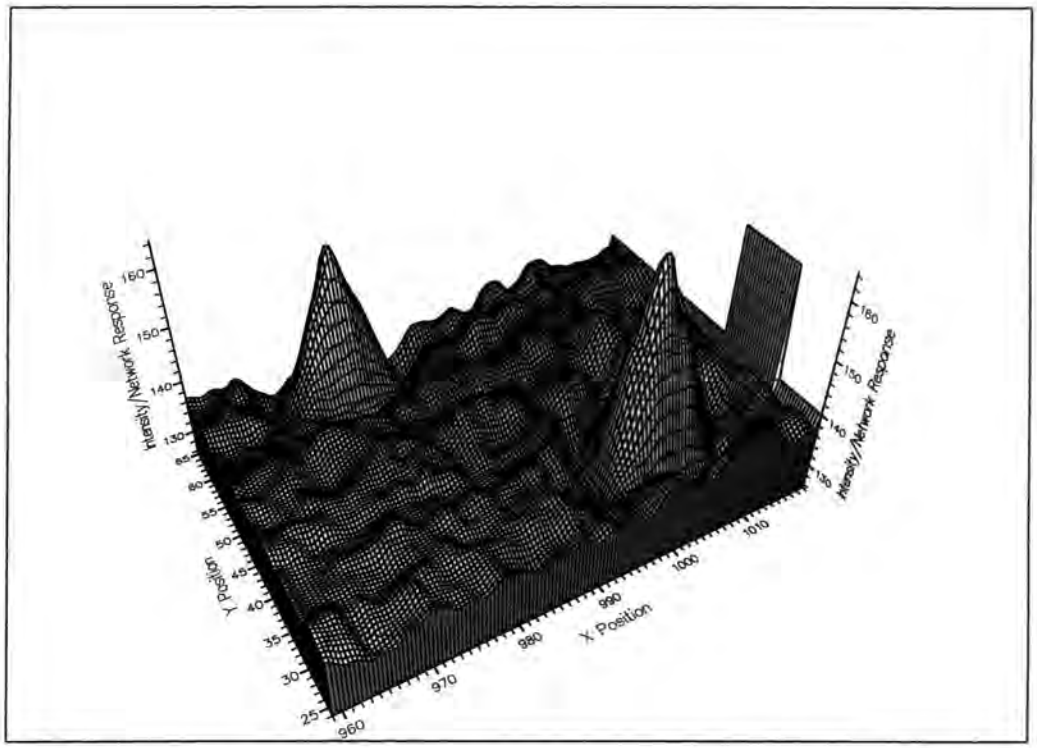


Figure 35: "Position-sensitive" experiment repeated, "non-defect" data obtained from cross-sections *inbetween* voids A and B.

It can be seen that the network now correctly classifies the central portion of void A as being defective and the inbetween region as non-defective. In keeping with our "position-sensitive" hypothesis, void B is also labelled as non-defective, and we attribute this to the fact that no training data indicated a defect at a horizontally-coincident position to this feature.

As a final step towards verification, we further iterated the experiment, in this new case the "defect" training data was obtained from random cross-sections across *both* voids A and B, "non-defect" training data from the inbetween region and other conditions remaining constant. Results are shown as figure 36.

It can be seen that the network now correctly classifies *both* voids as well as the "non-defect" inbetween material. However, other voids present in the image are still not reliably detected, and we are therefore confident in our view that in this experiment the network is learning to recognise positional significance rather than any intrinsic properties of the voids themselves.

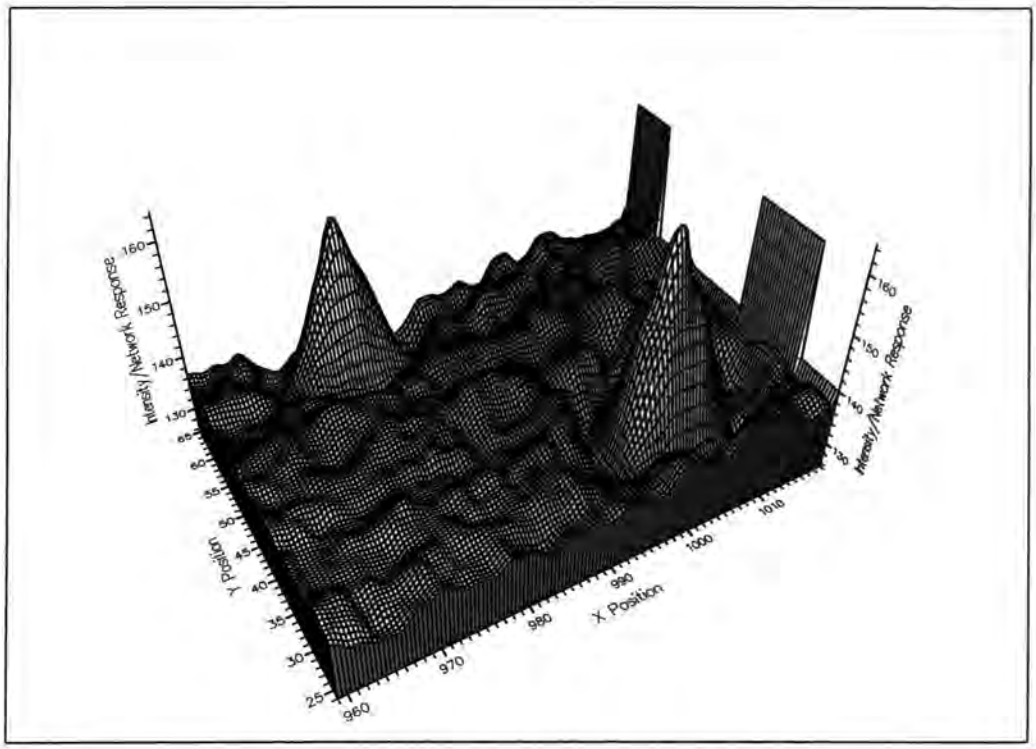


Figure 36: "Position-sensitive" experiment repeated, "defect" data obtained from *both* voids A and B.

7.1.4 Overcoming Position-Sensitivity using a Histogram Operator

The "Position-Sensitive" effect described, that is, the tendency of the network to detect only defects in specific locations would clearly be an unwanted feature in an automatic inspection system, and we therefore next sought to investigate techniques which might potentially remove this. It appeared that one straightforward method would be to apply a *histogram* operator to the RMS-processed data used in previous experiments.

Figures 37 and 38 illustrate the results when a histogram operator is used to process the RMS data shown in figures 29 and 30 respectively. The RMS power is now represented on the abscissa, with the number of instances of that power, the *population*, being represented on the vertical axis. The histogrammed data is discretised such that the number of data values is unchanged after the operation, allowing a network structure to be used identical to that employed in the earlier

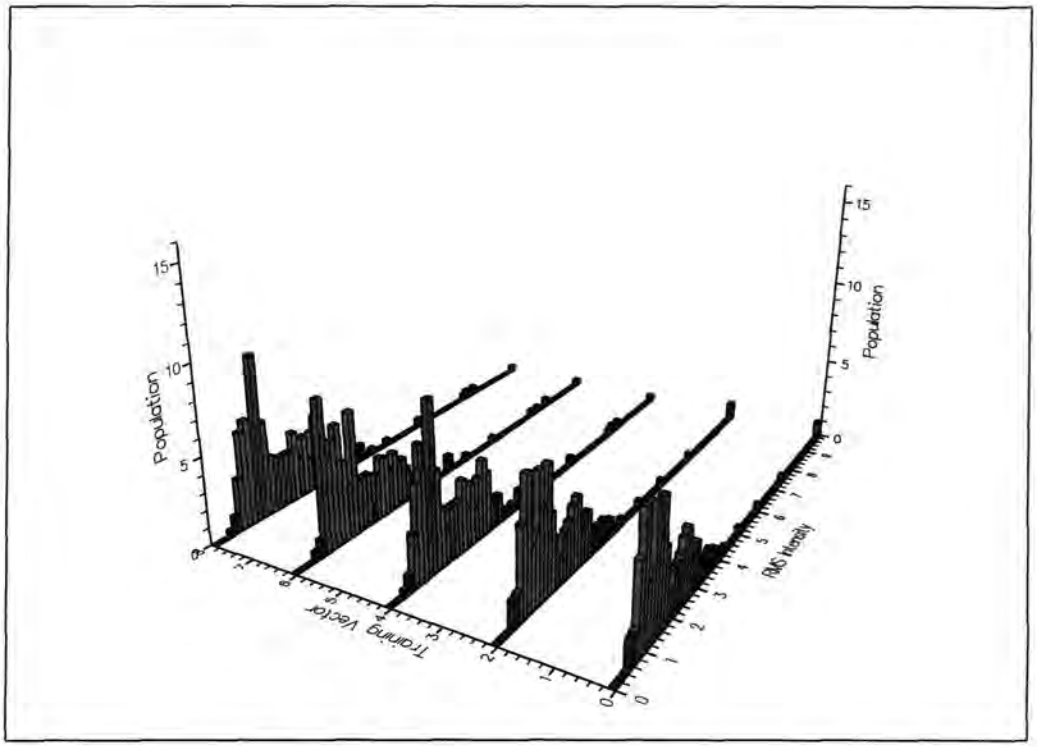


Figure 37: Data from figure 29, preprocessed by a *histogram* operator - defect present.

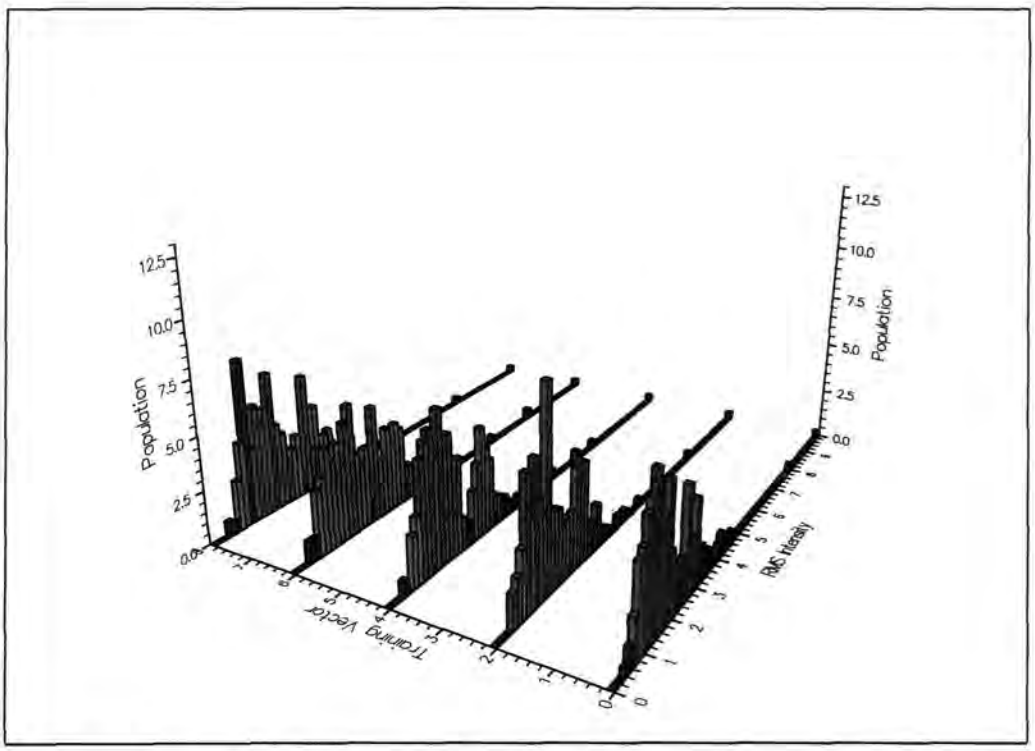


Figure 38: Data from figure 30, preprocessed by a *histogram* operator - defect absent.

experiments, that is, with 204, 7 and 1 node(s) in the "input", hidden and output layer respectively. The discretisation process is linear, that is, all histogram *bins* are equal in extent. The histogram process has been hard-limited at an RMS power of 10 units, and the few instances of RMS powers higher than this value have been incorporated into the "highest power" histogram bin, since it was felt that the population would otherwise be mostly confined to a small number of low-valued bins, thus reducing resolution over what is intuitively the most significant portion of the data.

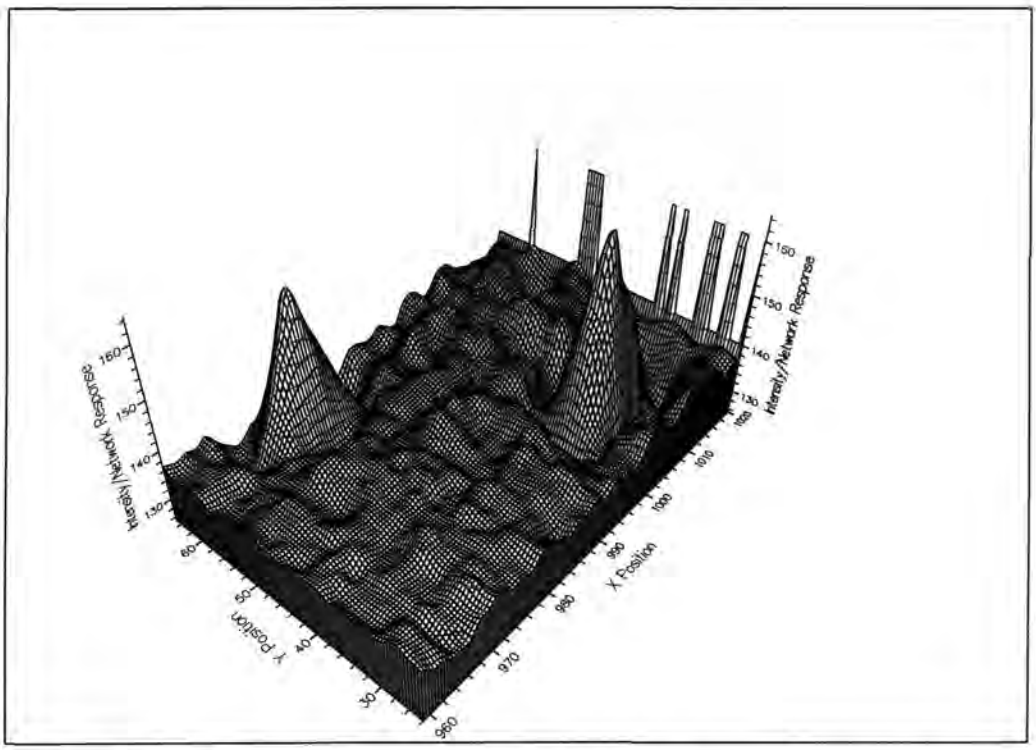


Figure 39: Network results using histogram data from figures 37 and 38 as training set.

Figure 39 shows the results of the network's analysis of the image when data from figures 37 and 38 is used to train the network. Again, training is successful, that is, the mean error over the training set is reduced to a very small amount attributable to floating-point inaccuracies in the implementation. In order to obtain the analysis shown, each line of image data is preprocessed in exactly the same way as that in the training set, that is, firstly with the RMS operator as before and secondly with the described histogram operator.

It can be seen that the network output is now completely indeterminate and we can therefore conclude that we have failed to eliminate the "position-sensitive" effect using the histogram-processing technique. Furthermore, it is clear that although the network has identified some feature or trend which enables it to distinguish data in the "defect" half of the training set from its counterparts in the "non-defect" half-set, as is indicated by the low mean error after training, the network's failure to categorise accurately the defects illustrated in figure 39 in a more general way shows that the scheme identified by the network is based on something other than that desired, namely the presence or absence of voids.

An anecdotal item is a useful analogical reference here. Rumour has it that researchers working on a US Military project had attempted to train a neural network to differentiate, with the aid of suitable preprocessing, images containing heavy artillery from those of similar scenes with no such equipment present. The system successfully learnt to differentiate the images in the training set, but, as with the experiment in view, was far from successful when analysing images which had not previously been "seen" by the network. Further investigation revealed that, by coincidence, pictures with artillery had all been acquired on sunny days, whereas those without were obtained on cloudy days. In fact the network had learnt to differentiate the weather conditions.

It would appear that in that case, as is the case here, careful conditioning of the training data is vital in order to ensure that the network learns the desired "rule" rather than some other *parasitic* rule which may be present in the data quite by accident.

7.1.5 Overcoming Position-Sensitivity using a Synthetic Training Set

Training Set Description

We next tried to overcome the described "Position-Sensitive" effect and to further understand the neural network behaviour using an alternative technique, which

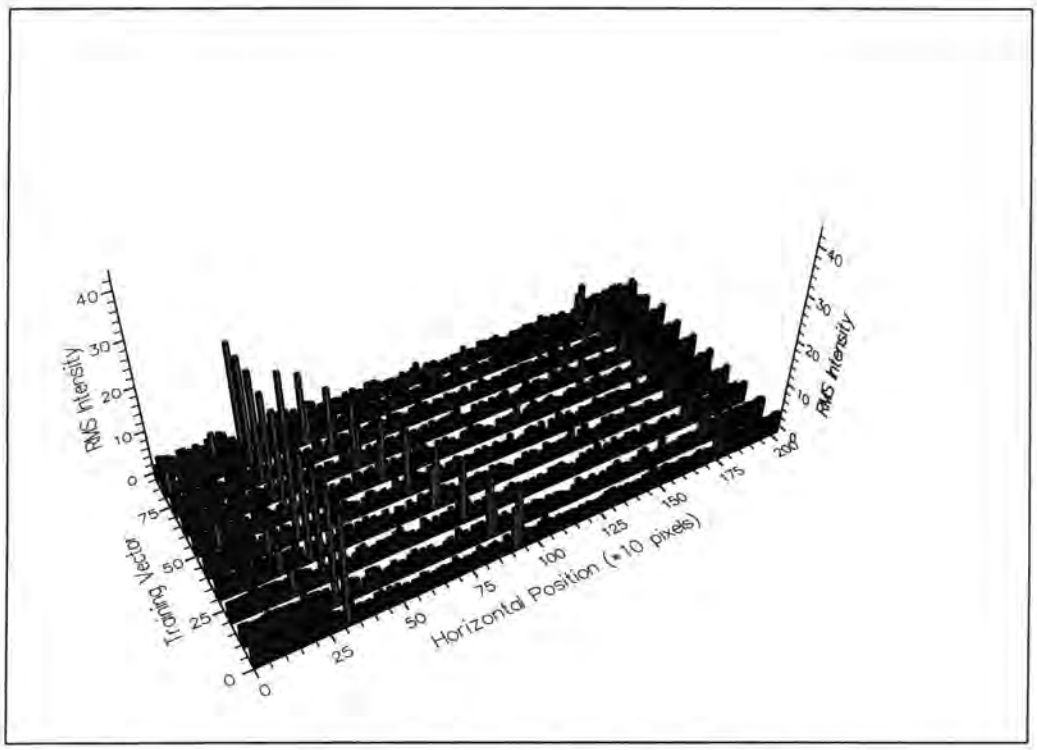


Figure 40: Synthetic void experiment: first half of training set, corresponding to presence of defect

involved the use of a *synthetic* training set. Strictly speaking only half of the set, the "defect" half, has been synthetically generated, and this is shown in figure 40. Here only *one* strip of data has been directly obtained from the image, that is, the strip nearest to the origin and in fact this is a cross-section through the void previously labelled as A in figure 34. The other strips in this half set have been obtained by subtracting the RMS powers of an image strip *not* containing a defect from the origin strip, leaving RMS power due exclusively to the void, plus some noise. This data vector is then positionally rotated by various amounts and summed with the data previously subtracted in order to obtain the vectors shown, which as a result have the RMS peak due to the void present in various positions.

Figure 41 shows the second half of the training set, that is, that which corresponds to "no defect". As in previous experiments, these vectors were randomly obtained from elsewhere in the original image.

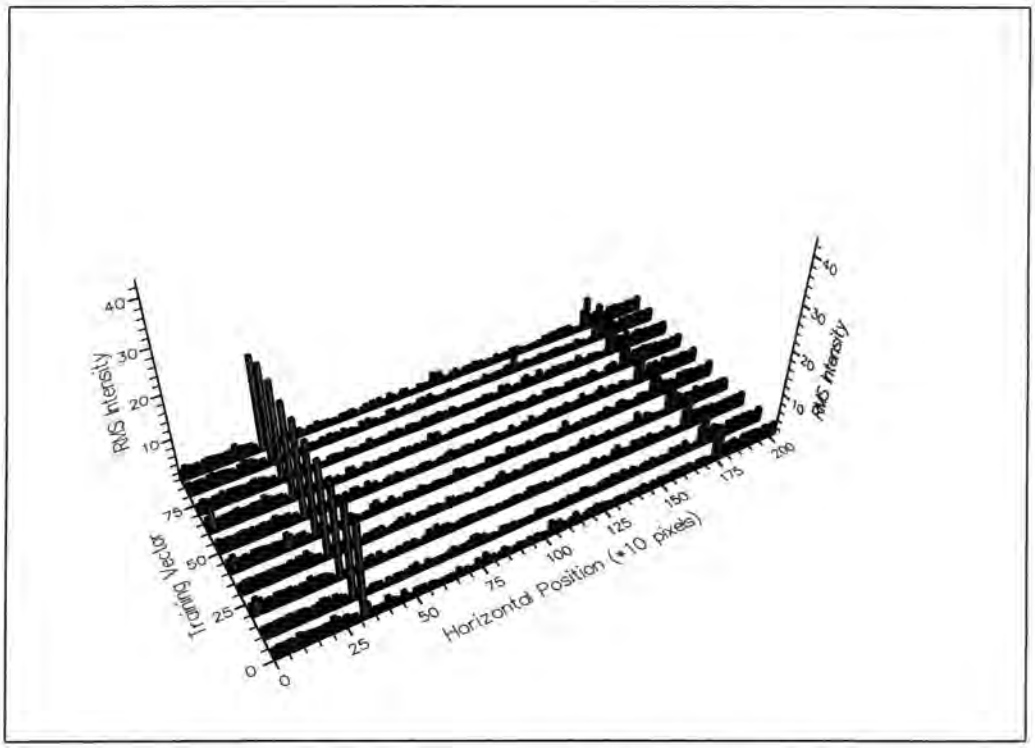


Figure 41: Synthetic void experiment: second half of training set, corresponding to absence of defect

As previously, training with this set was successful and a near-zero mean error over the set was achieved.

Improved Network Result Visualisation

In order to conveniently visualise the response of the network to each of the complete collection of horizontal slices through the original image, we sought an operator for each slice which would yield a single floating-point value giving a good indication of whether a void defect was present at some point in that slice.

Initially we attempted unsuccessfully to use various operators for this purpose including the standard deviation (figure 42) and maximum consecutive pixels with no turning point over both entire horizontal slices and over on-process material slices only (figures 43 and 44 respectively. Eventually a simple peak-to-peak value was found to be most successful, calculated over on-process material only in order to prevent the peak-to-peak power due to voids being swamped by that due to

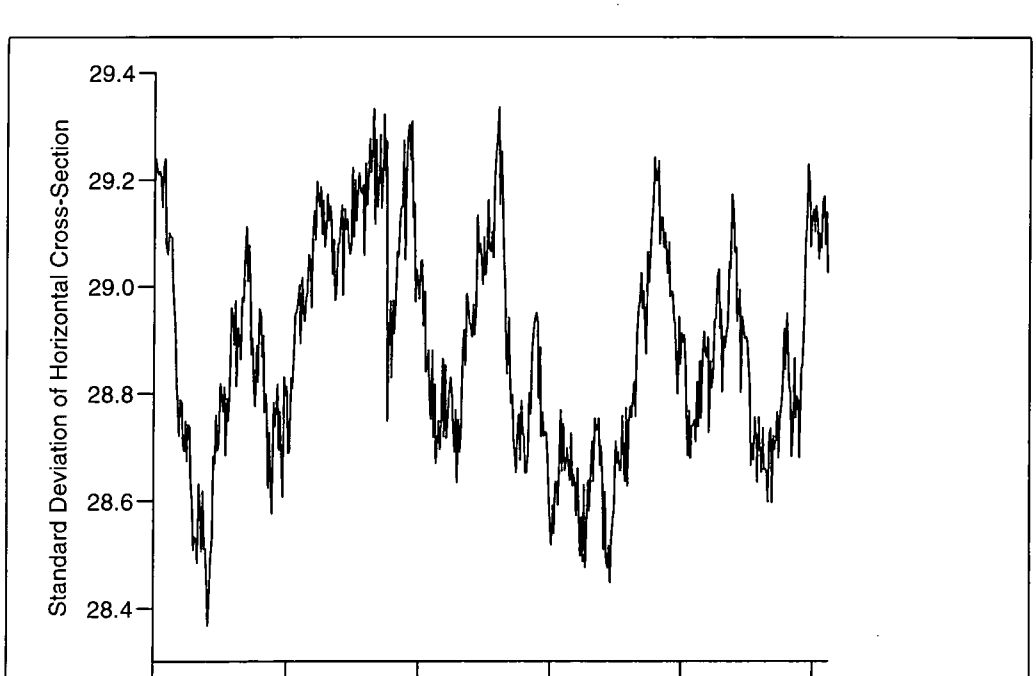


Figure 42: Standard deviation of horizontal cross-sections taken from "morevoids" with respect to vertical position in image

the on/off-process contrast boundary.

We were therefore able to show graphically in figure 45 the presence of defects over the whole image together with the corresponding neural network analysis after training using the synthetic training set.

Analysis of Neural Network Weights

In the past, attempts have been made to understand the rule that has been inferred by neural networks such as the one in view by examination of the weights after training. For example, in [39], Feng, Houkes et al. examined in detail the weights of a multi-layer perceptron which had learnt to distinguish short and fat bars in an image from long and thin examples. However, the trained network's reasoning process is in general notoriously resistant to analysis.

Nonetheless, we decided to examine the network's weight values after training had been successfully completed. Figure 46 contains 7 graphs which each show

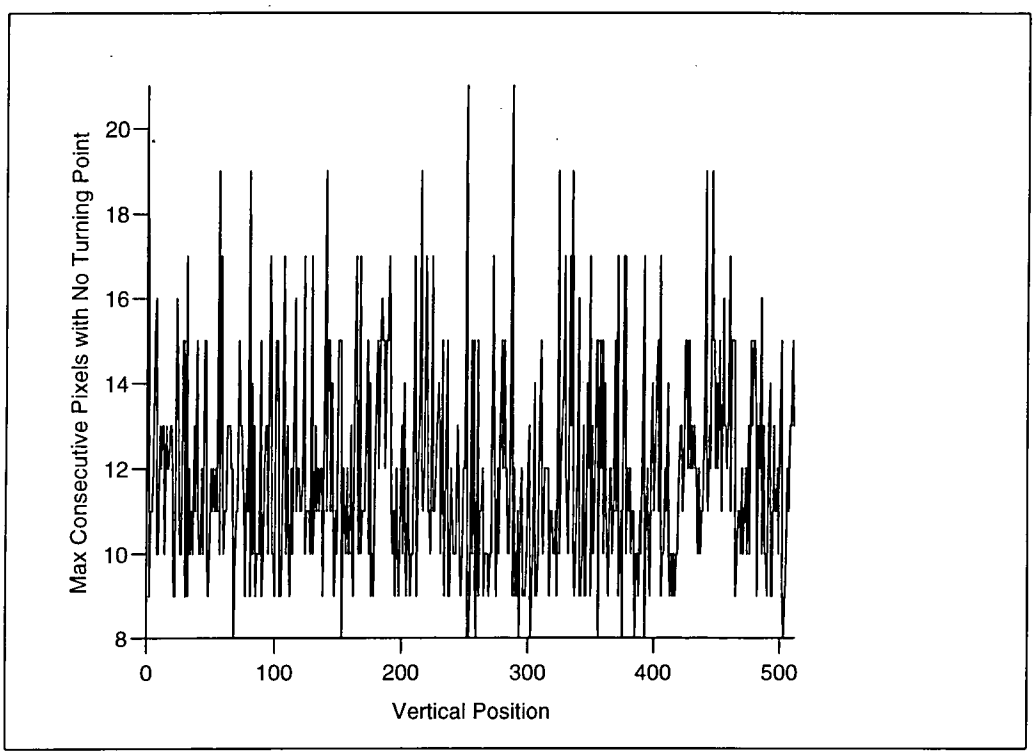


Figure 43: Maximum number of consecutive pixels with no turning point in a full 2048-pixel horizontal cross-section taken from "morevoids", with respect to vertical position in image

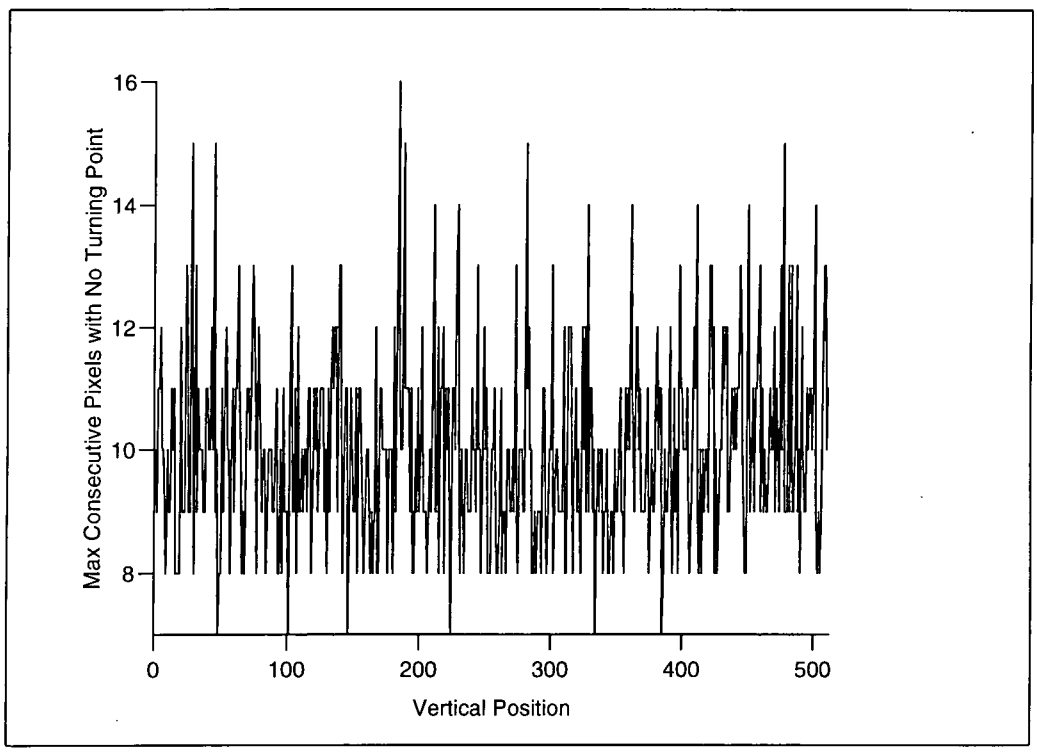


Figure 44: Maximum number of consecutive pixels with no turning point, in a horizontal cross-section of on-process material only taken from "morevoids", with respect to vertical position in image

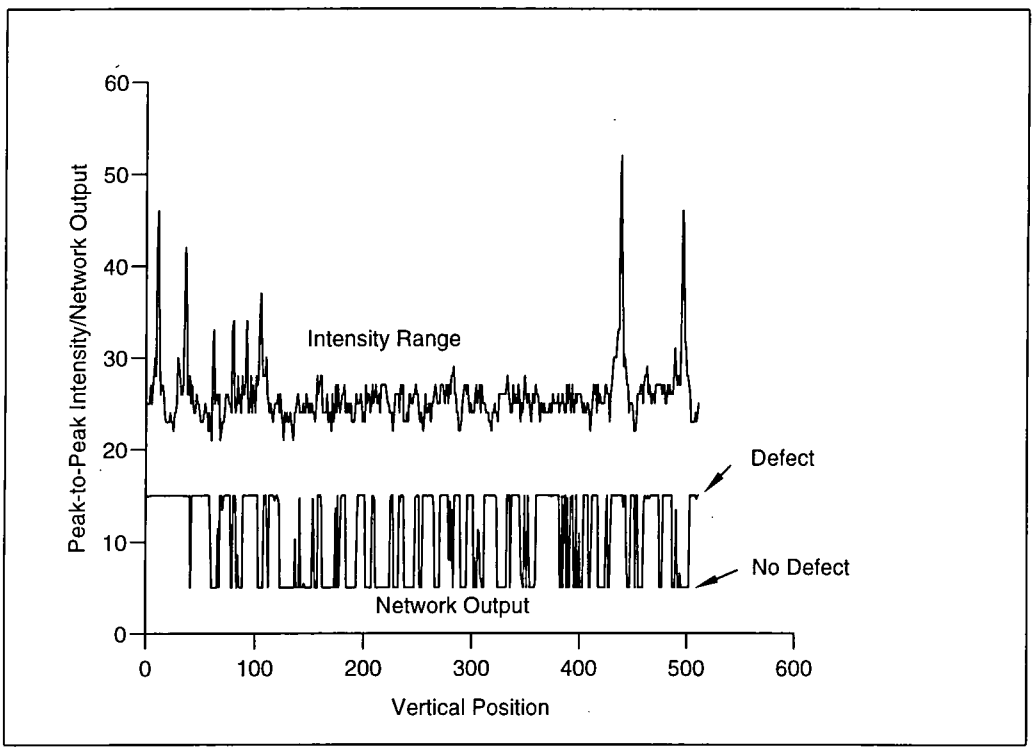
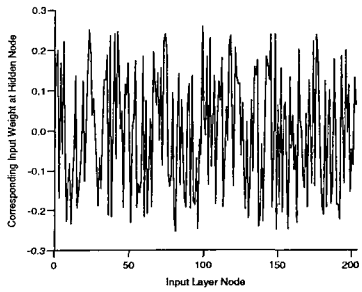
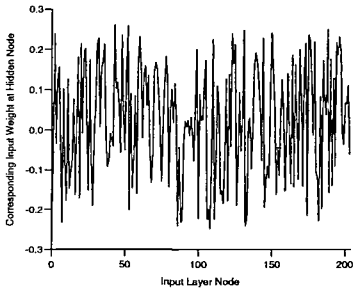


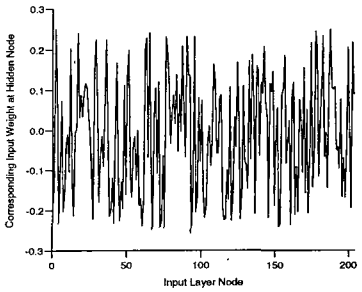
Figure 45: Peak-to-peak intensity range over horizontal cross-sections from "morevoids" with respect to vertical position in image, showing corresponding neural network output



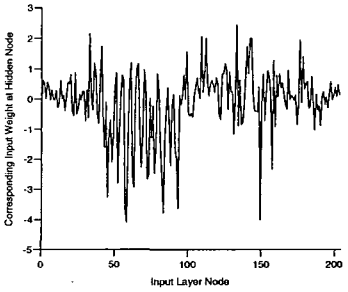
Hidden node 0



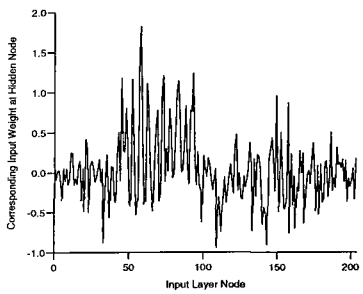
Hidden node 1



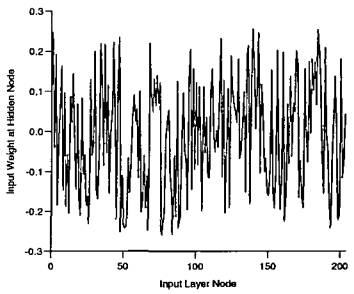
Hidden node 2



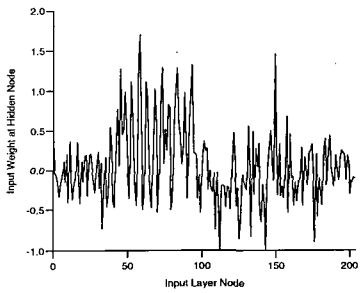
Hidden node 3



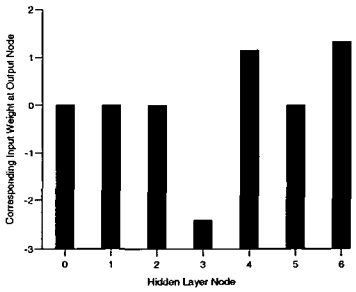
Hidden node 4



Hidden node 5



Hidden node 6



Output node

Figure 46: Network weights after successful training on "synthetic void" sets (figures 40 and 41).

the values of all 204 input weights which are applied to data from the input layer before summing and non-linearising at the appropriate hidden node. The final bar graph in the bottom right shows the weights applied to hidden node values before summing and non-linearising at the output node.

If one bears in mind that a positive value at the output node has been defined to represent a "defect" decision, and vice-versa, it can be seen that the network's decision-making methodology is surprisingly straightforward. The input weights to hidden nodes 4 and 6 have evolved such that they take on large positive values when the input node with which they are associated is positionally coincident with one of the voids in the synthetic "defect" half of the training set. The weights associated with the links between hidden nodes 4, 6 and the output node are also large and positive, and thus the mechanism by which a large RMS power at one of the significant positions brings about a positive value at the output node becomes evident.

It should also be noted that the input weights to node 3 have adopted a similar although inverted pattern, however, the corresponding hidden to output weight is negative and so the net contribution is the same, that is, high RMS powers in the synthetic defect positions will tend to cause a positive "defect" result at the output node.

Hidden nodes 3, 4 and 6 have input weight patterns which also show peaks having the effect of *desensitising* the network to high RMS powers, These are peaks in the opposite sense to those positionally coincident with the synthetic voids, that is, for hidden nodes 4 and 6 they are negative-going, whereas for hidden node 3 they are positive-going. For example, such peaks of non-sensitivity occur coincident with the high power values which correspond to the off/on-process boundary. This is logical, since such boundaries occur in both halves of the training set.

The input weight patterns at hidden nodes 0, 1, 2 and 5 do not at first sight have any discernable correlation either with the training vectors or with each other. However, it can be seen that the corresponding hidden to output weights

are close to zero.

The neural network output shown in figure 45 incorrectly identifies much of the image, which is in fact defect-free, as being defective. We attribute this to the fact that the "defect" vectors in the training set (figure 40) have a higher average background RMS power level than those in the "non-defect" half of the set (figure 41). This has resulted in a DC bias in the input weights for nodes 3, 4 and 6. Again, the network's decision-making methodology has latched onto a feature present purely by coincidence as well as the one desired.

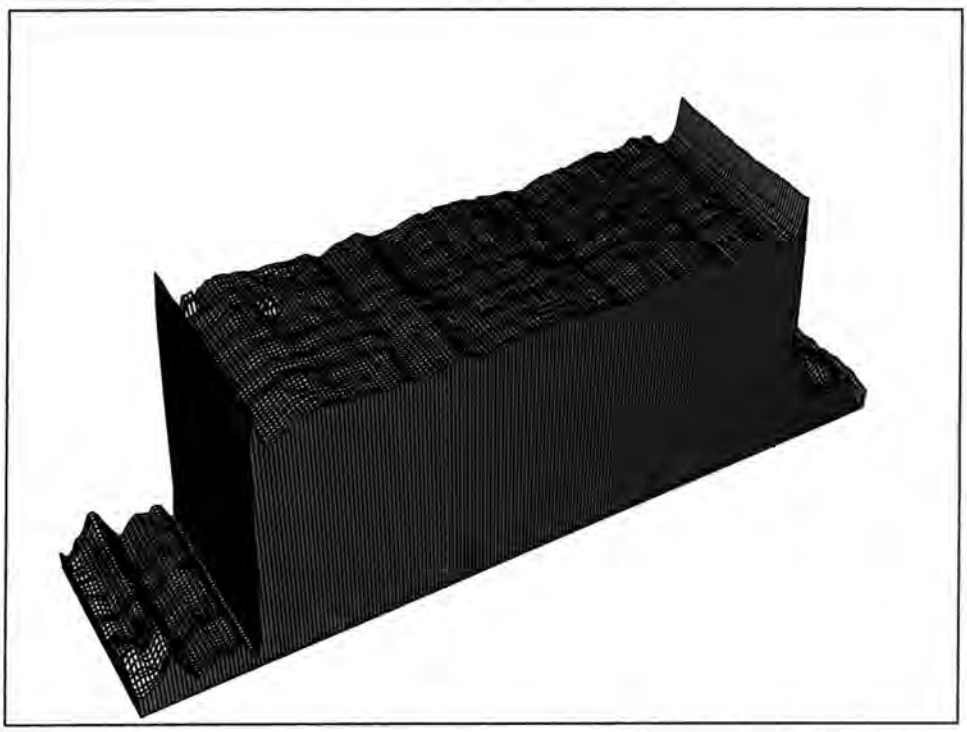
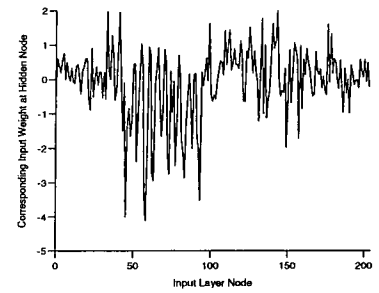


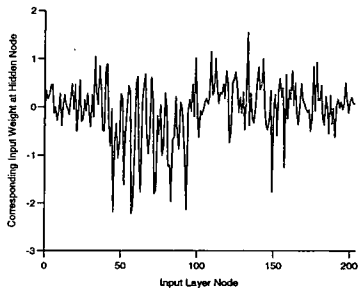
Figure 47: Subsampled surface representation of complete 2048 by 512 pixel linescanned image

7.2 Conclusions

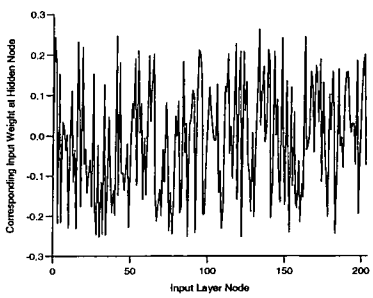
Progress was made towards understanding the behaviour of the neural networks employed and towards achieving a configuration which could accurately classify the defects in view. The most important conclusion to draw from these experiments is that both the conditioning of the training set and the format of the



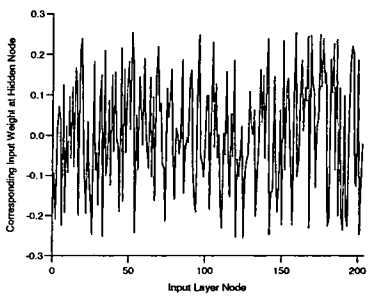
Hidden node 0



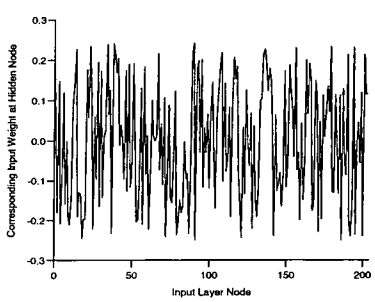
Hidden node 1



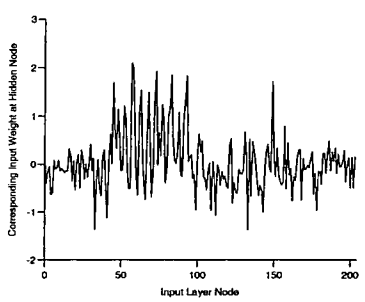
Hidden node 2



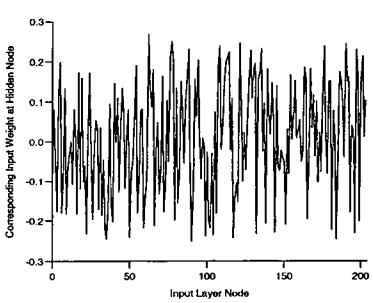
Hidden node 3



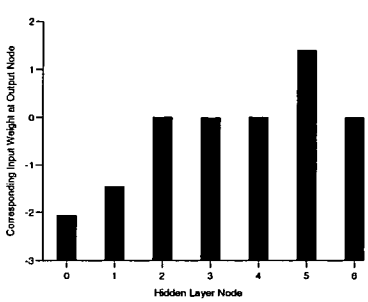
Hidden node 4



Hidden node 5



Hidden node 6



Output node

Figure 48: Second trial: network weights after successful training on "synthetic void" sets (figures 40 and 41).

data preprocessing is very important to network performance. In particular, the network is likely to make use of any trend of data present in the training set which will allow it to distinguish between the two halves of the set, and the set must therefore be carefully formulated to ensure that only the desired trend is present. Careful choice of preprocessing will make this process more straightforward.

Each of the experiments presented here required a significant amount of processing time in the network training phase, and it was felt that this was a serious inhibiting factor to future progress. We therefore turned our attention to the possibility of speeding up the training operation in chapter 8.

Chapter 8

Algorithmic Development

8.1 Introduction

8.1.1 Parallel Processing Speedup

It is widely recognised that computational efficiency can be significantly improved through the use of parallel processing. Exploitation of parallel techniques has allowed the development of more affordable supercomputers, since cost per MFlop[1] is significantly lower for an array or network of medium-power processors than for a single, very high performance processor. The computational task under consideration must, however, have elements which are amenable to parallelisation. In many cases these are present at the machine code level, that is, at the point where the task has been translated into a series of instructions which can be directly interpreted by the microprocessor. Most members of this instruction set will incorporate identical subtasks, such as instruction fetch, execution and effective address calculation. If each such subtask is handled by a distinct unit of hardware within the processor, then many instructions can in effect be handled at once. This is the basis of pipelining.

The computational task may also be parallelised at a higher level in order to achieve speed-up. In this case, however, the programmer needs to be concerned directly with determining which sub-elements of the task may be run concurrently. For example, if subtask B uses as initial data the results of subtask A,

[1] A standard measure of floating-point operation performance

then simultaneous execution will not be possible. In contrast, pipelining schemes in popular microprocessors operate transparently as far as the programmer is concerned.

Our approach is to include high-level parallelisation in our implementation of backpropagation training. The platform comprises many workstations which are based on pipelined microprocessors, and so the two schemes described effectively operate simultaneously in this case.

8.1.2 Previous Parallelisation Work

We should next like to review a cross-section of the existing literature on software parallelisation in order to illustrate the context and relevance of our own work.

Researchers have previously sought to parallelise a wide range of applications; the most straightforward of these to work with are tasks which involve an algorithm iterated many times over different data. For example, Müller-Plathe's work in [20] involved a molecular dynamics algorithm, here the same dynamic modelling equations are operated for each molecule in the system. The simplest way to parallelise such a system is to divide the molecules among processors, although inter-process communication will be required to compute inter-molecule interactions. Alternatively, algorithms which have less inherent parallelism may still be approached, although this is typically more demanding, for example, Rothberg et al. in [22] sought to improve the performance of sparse matrix factorisation on a multi-processor workstation - here, though, the parallelisation is organised transparently by the operating system, and the optimisation work was oriented around implementing the factorisation algorithm so as to maximise the number of "hits" of the involved Silicon Graphics machine's split-level memory cache. This constitutes custom-hardware programming.

The issue of coarse-grained versus fine-grained parallelisation appears to be very important. Tightly-coupled systems, for example, the multiple Intel 80386 processor system described by Horiguchi et al. in [46] can handle both types of

parallelism, although development of such systems is rather specialised. A corresponding choice has to be made between *message passing* and *shared memory* modes of inter-process communication. Tightly-coupled systems, typically multiple processors sharing the same memory bus communicate most effectively using shared memory techniques; here multiple processors access information which stays in a fixed location. Where processors have distinct memory busses, it is usually necessary to employ the message passing technique; here information is read from one processor's memory, written over some kind of communications channel and thence written to the other processor's memory. It should be mentioned, however, that even with tightly-coupled processors the message passing model is sometimes used, since this makes it easier to *synchronise* the communicating processes. This may be of particular importance where the volume of information to be communicated is small, since shared memory systems will in any case require some kind of additional message passing in order to achieve this synchronisation, unless the shared memory segment is to be polled.

The coarseness or fineness of parallelisation grains is determined by the volume of interprocess communication required and the quantity of resources required to conduct each task unit, as we have discussed in chapter 3.

There is considerable interest in "network compilers" which automatically translate a sequential program into a set of distributed programs and control their execution; Shi and Blathras in [50] make use of "Linda", one of a family of such compilers presently available in order to achieve parallel implementations of fractal plotting and liquid crystal dynamics simulation. Research has also focussed on the merits of one such compiler with respect to another, and on techniques for getting the best performance. However, our view is that any technique which takes away from the implementor control over the exact nature of the parallel implementation can only yield solutions which are sub-optimal when compared with those custom-designed with a particular application in mind. There is naturally also interest in techniques for designing such compilers. King in [47] goes one stage further and proposes various operating system design techniques

for a multiprocessor architecture. He makes an interesting distinction between "lightweight", that is, low-overhead multi-processing, and more "heavyweight" systems such as a Unix scheduler. Clearly these are relative terms, in the context of more loosely-coupled multiprocessor systems both of the arrangements described by King would be considered "lightweight". King emphasises the importance of conformance with API[2] standards to encourage the uptake of such systems; those mentioned include IEEE Posix P1003.1, which provides a definition of operating system services and C language binding, P1003.4, realtime extensions and P1003.8, networking interfaces.

The choice between realtime and non-realtime operating systems is significant for implementors of distributed applications, and Williams [48] gives a useful consideration of the relevant issues.

At the time of writing, most Unix kernels in popular use such as SunOs 4.2 are non-realtime. This means that executing processes are not pre-emptable by realtime interrupts, in other words, once a particular process starts to execute, it cannot be stopped in favour of another irrespective of priority until an interrupt associated *with that process* occurs, typically a memory page wait or end of time slice.

Realtime kernels, on the other hand, generally have areas where a higher-priority process can pre-empt a running process without endangering data structures. In the most modern kernels, there are few areas which cannot be pre-empted.

The disadvantage of non-realtime Unix systems is that non-pre-emptability makes it impossible to calculate worst-case completion times for any task. In other words, the system is non-deterministic. The nature of Ethernet CSMA/CD[3] communications makes the situation more chaotic since this system cannot guarantee a maximum transmission time for any message.

It would seem that the use of realtime kernels together with a deterministic

[2]Application Programming Interface
[3]Collision-sensing multiple access/carrier detect

networking protocol such as token ring is required for a performance-critical distributed application which requires guaranteed worst-case completion times for certain tasks.

The use of workstation clusters for the implementation of distributed parallel applications is not in itself a new idea. Magee et al. in [18] report on their work using the travelling salesman problem with a simple "supervisor/worker" paradigm on such a cluster. Here loads are balanced in the sense that a "worker" received a new task as soon as it has finished the old one, thus all nodes are in theory permanently occupied. Magee et al. do give a cursory consideration of parallel speed-up efficiency, although this is not held to be of particular importance since the resource being used is effectively "free". Data transmission times are held to be negligible in this consideration, however, which is in our view rather unrealistic.

Griebel et al. in [21] make use of a workstation cluster in order to speed up the sparse-grid preconditioning or colution of partial differential equations. Again a simple "supervisor/worker" arrangement is used, with one node controlling the allocation of tasks and collation of all results. The simple structure is deemed suitable because the computational task is large compared with the amount of communication required for this application. Communications are therefore achieved very straightforwardly using shared NFS files although this technique is rather inefficient, as we have discussed in chapter 3. The authors suggest that a "sub-mastering" system might be of use for finer-grained tasks.

Cap et al. in [19] give the most detailed consideration we have found of the use of workstation clusters for parallel distributed applications. The authors suggest that in general, the programming of parallel architectures is difficult, but the use of parallel workstations introduces the extra difficulties of heterogeneity and a constantly-changing load situation. Much of their report concentrates on load-balancing - the authors note that there is no obvious way of determining the extent to which a CPU will be "slowed" by a subtask of a particular size, since this is a function of the capability of the CPU itself. Therefore the initial loading

strategy is likely to be sub-optimal - subsequent balancing strategies may help.

The authors implement a heat conduction application and conclude that for workstations numbering up to 20, the speed-up is "near linear". However, in our view this is due to a combination of sublinear effects, caused by the overhead of the parallelisation process such as communication time, for example, and superlinear effects, for example due to the greater total quantity of memory available in the system as the number of nodes is increased. For a greater number than 30 workstations, speed-up is severely degraded due to "saturation" of the communications bandwidth - CSMA/CD networks are highly inefficient for close-to-capacity loads. However, the exact number of workstations at which this happens is clearly dependent on the volume of communication involved.

8.1.3 Motivation for This Work

The work which we will now describe focusses on the speed-up of the backpropagation training algorithm, which, as we have already concluded, is computationally-intensive to the point of hindering further developments in the understanding of practical neural network systems. Since there is no single algorithm which is iterated independently on many data objects, we have of necessity introduced an "indirect" means of parallelisation, which is the introduction of multiple search "workers". Although a considerable speed-up is achieved it is clear that even with a completely efficient implementation, two search workers will not double the search speed. Therefore it is meaningless to discuss the time to solution as being a metric of implementation efficiency; for this reason we present our results in terms of "normalised iterations".

Our implementation uses a heterogenous workstation cluster. We have made our implementation as efficient as possible in terms of communications according to our findings in chapter 3. We have designed our application using the lowest practicable level of network programming since we believe that efficiency is

lost through the use of the network "compilers" adopted by many of our contemporaries. The system produced through a hybrid of ideas in neural network applications, algorithms and implementation is novel and has highly-desirable properties.

8.2 Distributed Processing - Backpropagation Training

8.2.1 Basis for Parallelisation

In section 7.1.2 we detail the training of a multi-layer perceptron network by means of backpropagation such that "voids" in an image taken from sheet aluminium can be identified from horizontal slices through the pixel data which are then preprocessed using a histogram operator.

This training process can be regarded as being essentially a search problem, that is, a search through the neural *weight space*, which consists of a continuum of positions, the co-ordinates of each being given by the corresponding complete set of network weights. It is helpful to envisage this space as a multi-dimensional surface, each point thereon having an associated *height* representing the network's performance at that point, this being expressed in terms of *mean error*, that is, the mean difference between desired and observed behaviour at the output node when the training input vectors are applied. The search *goal* is therefore the surface position corresponding to a weight vector yielding a minimum mean error, this minimum ideally being zero. It can be shown [26] that training by basic backpropagation is analogous to a deterministic *gradient descent* search strategy. Note that section 7.1.1 describes training sessions which evolve differently for each trial, even when the search problem and parameters are the same. However, this is due to the fact that network weights are initialised to small random values each time, and thus in each case searching begins from a different position in the weight space.

The basic backpropagation training procedure is highly processor-intensive, and our original single-node implementation took of the order of tens of hours to arrive at a weight vector with an acceptably low mean error performance, as described in section 7.1.1. In order for the technique to be a useful tool in dealing with the types of vision problems there outlined, it therefore appeared that the enormous and lengthy nature of the processing task required for retraining would present just as great an obstacle to exploitation in practical systems, as would the difficulties in formulating schemes for preprocessing the data and determining the network topology.

We therefore sought to speed-up the training process, and it is evident that the possible approaches are twofold:-

- The search algorithm itself can be made more efficient. For example, the addition of a *momentum* term can give a substantial upgrade in performance, since the search will traverse more quickly through regions of the weight space which have a relatively steep gradient, yet sufficiently slowly through shallow regions so as not to "overshoot" minima. *Simulated annealing* is another example of this type of approach, although here the emphasis is on preventing the search from being trapped in *local* minima.

- Multiple searches can be performed in parallel. In other words, many workers would "wander" through the same weight space in search of the problem's solution, that is, the global optimum.

Progress in speeding up neural network training algorithms is regularly reported in the literature, and it is certainly the case that a great deal of sophistication can be added to single-search algorithms when compared with those used in chapter 7. However, frequently such improvements make use of particular features of the problem in view, and there is thus a price to be paid in loss of *general applicability.* The driving force behind this thesis, however, has been to develop views of the best direction for the solution of future vision problems in general,

whilst maintaining an awareness of the constraints imposed by the equipment which can realistically be expected to be available. We have therefore chosen to focus on the latter of the two, that is, the multiple-search approach.

It appeared that an efficient speed-up might well be obtained through use of multiple search workers, providing that the overhead of coordinating the actions of each did not outweigh the benefit thereby accruing, and that a parallel scheme could make use of a greater total computing resource than the single method. One should of course bear in mind that in many cases problems yield faster results when a parallel search is applied, even for single-processor implementations - this is the basis for the family of search strategies known as *genetic algorithms.*

8.2.2 Specification

We sought to investigate the practical speed-up possibilities by repeating the single "void" training procedure described in section 7.1.2, this time making use of "spare" CPU time on many remote workstations.

At the outset of this stage of practical work it appeared that the following features would be required from the software implementation:-

- The entire system should be controllable from a single workstation node, that is, without any requirement for the user to communicate manually with worker nodes, since the process of starting individual slave applications will be very laborious and time-consuming if there are many such nodes.

- Tasks running on slave nodes should have a low scheduling priority assigned so as to minimise performance impact on other users. In other words, only CPU time which would otherwise be wasted should be used by the application.

- Inter-node communication should use a relatively low-level[4], low-overhead protocol such as TCP, so as to minimise the application's effect on network

[4]Low-level in comparison with remote procedure calls, for example

load.

- Memory allocation should be *dynamic*, that is, memory will only be requested from the system when required and released as soon as it is no longer needed, in order to minimise memory contention with other users' applications.

- Overall design of software should be as robust and fault-tolerant as possible. This is of particular importance, for example, since many instantiations of the slave program will run in parallel, thereby multiplying the chances of a single failure.

8.2.3 Implementation Details

Hardware Description

The distributed application was intended to make use of workstations administered by the University of Durham Computing Service and geographically distributed around the University of Durham Science Site. Physical interconnection was by 10MB/s Ethernet throughout, although in some places this protocol is operated over a fibre-optic backbone rather than the more usual 50 ohm coaxial cable. The network connecting the workstations in question is however nonlinear in that there are various intervening bridge/routers which are designed to isolate intra-departmental traffic. Although this project uses exclusively the TCP/IP suite of protocols for inter-node communication, the physical network layer is in fact shared between this and several other protocols such as Novell Netware and Appletalk.

The workstations as a group are heterogenous in architecture, consisting mostly of model 730/750 Hewlett-Packard and Sun Microsystems IPC machines, although there are a variety of other, more powerful HP and Sun systems which are used for filesystem service and time-shared processing.

Automatic Start-up

The automatic start-up facility was coded as a separate, virtually stand-alone piece of software. The design object of this program was to relieve the user of the need to manually log-in to remote workstations and start the slave application on each.

The start-up application makes use of the Unix *rsh*[5] protocol, as this is supported by all of the slave nodes in question. This consists of *rshd*, the remote shell *daemon* or *server* process which runs on each machine as part of the operating system. *Rshd* is invoked indirectly by *inetd*[6] when a network message is received from the *calling* or *client* application *rsh*. This message gives details of the calling UID[7], calling hostname and of the command or program that is being requested to run on the remote machine. On this basis *rshd* performs authentication, and if this is successful invokes a new process as required.

Authentication provided the first practical obstacle to coding. The main purpose of the remote shell protocol is to allow users to remotely run programs on machines on which they also have accounts - the authentication mechanism allows the user to specify which foreign host/UID combinations he/she owns and which should therefore be "trusted", that is, allowed to run tasks on the local host with that user's normal access rights and privileges. There is no facility for password transmission under *rsh* and thus it is essential that the authentication mechanism be made to work.

The problem lies in the nature of the files which specify trusted host/UID combinations. For both Hewlett-Packard and Sun hosts these are two-fold:-

- A file with system-wide applicability */etc/hosts*. This lists remote hosts on which all matching UIDs should be trusted. Typically all such hosts are under a single administrative control.

- A file specific to a single user *.rhosts*. This resides in that user's home

[5]Remote shell
[6]Internet services daemon
[7]User identifier

directory and may specify any trusted host/UID combination *for that user*, even where the host is not under the same administrative control.

Originally it was envisaged that a Sun workstation *capella* situated physically and administratively in our Engineering department laboratory would be used to automatically start slave processes on a variety of remote workstations which are in several different physical locations as described, but all under the administrative control of the University Computer Centre. Since our own workstation is not implicitly trusted by any of these machines it was necessary to make use of the *.rhosts* mechanism to allow remote shell calls to them. This caused difficulties since the authentication implementation details for Sun and HP machines differ slightly in that there are slightly different expected formats for *.rhosts*. Unfortunately, all the slave workstations share a single network filesystem and thus it is not possible to have a separate file for each architecture.

In practice it was found that the easiest solution was simply to use a Computer Centre workstation to run the automatic start-up, since each of these implicitly trusts all others and there is thus no need for any *.rhosts* entries.

Rsh is a public protocol, and it would therefore in theory be quite straightforward to design the auto-start application such that it would interact directly with the remote *rshd*. However, software re-use is a key method of saving design effort and so it was considered that a faster time to solution would be obtained through use of the *rsh* client, which is normally invoked on the command line.

In general a remote shell command will have a latency of several seconds, this being the time required for the remote *inetd* to service the request, invoke *rshd* which performs authentication and starts the process appropriately, redirecting output back to the originating command. When used as intended this is seldom an inconvenience, however, since the program in view is required to perform many such calls, one per slave host, the latency for each when summed over all hosts would be quite significant. Another consideration is that the most convenient method for a *C* program to interface with a command-line application is by

means of one of the *execnn()* family of calls which transform the calling core image to that of the desired application. However, there can be no return from a successful *execnn()* call since the calling core image is lost, and it would therefore be impossible to call *rsh* serially as described.

Both of these difficulties, these being the set-up latency and core image destruction mean that the extra complication involved in designing the automatic start facility as a *multi-threaded* implementation is warranted in this case. The coded application starts as a single thread which reads a configuration file giving details of hostnames and corresponding executable pathnames. For each such host/executable combination a *child* process is *spawned*, that is, the current core image is completely replicated as a separate thread. Each child then performs an *execnn()* to call *rsh* with the appropriate parameters and terminates after successful completion.

A specific executable pathname is specified host-by-host in the configuration file since varying executable formats are required by the different host architectures. Also, the child process' realtime interval timer is set to interrupt after 60 seconds as a safety device - if for some reason the *rsh* call hangs then the child will in any event terminate after this time, thus ensuring that the system process table is not littered with defunct entries.

Slave Processing

The slave processing application is a second, distinct piece of software. The same slave source code is compiled for both Sun and HP architectures, the corresponding executable being stored in a separate file for each - each host being directed to run the appropriate binary by the automatic start system as described.

Each instantiation of the slave program communicates with the master application only - as implemented there is no inter-slave consultation, and a functional diagram of the whole system might therefore resemble a "star-shape". As is suggested by the master/slave analogy, each slave instantiation's activities are commanded by the master - the default slave status is idle and awaiting instructions.

A command from the master application will typically consist of information which is processed by the slave, the results are then communicated back to the master before the slave returns to its idle state.

Master/slave communication takes place by means of a TCP[8] socket link which is connection-oriented, that is, the link must be explicitly established at the start of processing. TCP performs error-checking, and there is therefore no need for the application to handle this since the link is reliable and sequenced.

After being initiated by the automatic start-up system, each slave application opens a fixed-address socket and *listens* for a connection from the master. Thereafter each application can treat the connection much like an ordinary file descriptor. In order to ensure maximum reliability, a new process is spawned by the slave to deal with each connection request, and thus each slave could in theory communicate with and perform processing requests for many masters simultaneously, although in practice there is only ever one such. However, this system is preferable in order to ensure that the slave is always ready to reset and restart when requested to do so by the master - there can be no question of the slave refusing to connect in the event that it is "busy" - for example if the master with which it is communicating fails and does not terminate cleanly for some reason.

The slave application's first process resets a realtime interval timer to interrupt after ten minutes, each time a connection is received from a potential master. When this interrupt is received, the original process will immediately terminate, but leaving intact any children that have been spawned and which are still communicating with a master. The rationale behind this is to allow a certain amount of time in which "teething problems" in starting the distributed processing system can be dealt with, for example, machines which start their slave application unexpectedly slowly such that connection is refused when the master is first invoked - in this case the user can choose to abort the master application and attempt to re-issue connection requests when all of the slaves are ready. Slave child processes, which do the actual application processing, will

[8]Transport control protocol

terminate automatically as soon as the link to the master is lost, and there is therefore no desirability in setting an interval timer. By these means, the slave applications will always terminate cleanly after use without the need for a user to log in to each individual machine and manually kill them.

Each slave has a complete representation of the neural network, that is, neurons, training set with expected output values and weight values. Each has an implementation of the backpropagation training algorithm and is therefore capable, given a starting weight *vector*[9] of finding a new weight vector which is closer to the solution, that is, a vector for which the mean error is reduced.

The cycle of master/slave interaction is as follows:-

1. The master downloads a copy of the training set and a weight vector to the slave.

2. The slave performs backpropagation training for a predetermined period, after which it calculates the *mean error* for the new weight vector which has been determined as a result, and this error value is then communicated back to the master.

3. The slave then idles, awaiting instruction from the master. Before resuming gradient descent the slave may be told either to upload its current vector/position to the master, to scrap the current vector and download new data from the master, or to proceed directly without further communication. The cycle then repeats from (2).

Master Processing

The master application is a further, distinct piece of software. It is designed to be the centre of communication and control, and is in contact simultaneously with all of the slave processors, to each of which the master opens a separate TCP communications channel.

[9] A vector comprising all the weights which specify the network

Initially the master reads hostnames from the configuration file originally used by the automatic start-up application to start the slave servers and attempts to establish communication with each. Should connection be refused, typically because there is no slave program ready and waiting to receive it, operation will continue with that host omitted from further consideration, providing of course that there is at least one slave node with which a link can be established. The system as a whole is therefore tolerant of nodes which are temporarily unavailable for some reason.

Next an initial weight vector with small, random values is determined and downloaded to each slave along with the training set data. All connected slaves then begin processing, meanwhile the master idles and awaits results. Each slave goes through an identical number of iterations of the backpropagation algorithm and reports back as soon as these are complete, however, the results will arrive in no particular order from the point of view of the master, since the execution speed of a particular slave will depend on the processing capability of the host concerned as well as its current load from other applications.

The master is therefore designed to process results asynchronously - after dispatching work it makes a *select()* call which *blocks* until data from one of the slaves is received over the network. Whilst blocked the application is essentially idling pending a network interrupt, consuming no CPU resources and potentially swapped out to disk. Results are processed in the order they are received, and in our initial implementation no more work is dispatched until a full set of mean error values has been acquired. Consequently the slave workers which finish their batch of work most quickly will idle until the slowest workers are also ready for more.

When results from all the slave workers are available, the master ranks the performance of each node according to the value of the mean error which corresponds to the position, or weight vector, at which that processor has arrived. Those with lowest mean error have achieved solutions which are closer to the global optimum and are therefore ranked more highly.

The command next sent to each slave worker can be either to *proceed* with training using its own current position, or to *reload* a new weight vector from the master, thus abandoning its previous work. The decision over which to do is made probabilistically on the basis of assigned ranking - those nodes which have achieved a better mean error will have a greater chance of being instructed to continue, whereas those with a poorer mean error will be more likely to be directed to obtain a new weight vector from the master. In our initial implementation the probability of reload P_r is determined from the rank r thus:-

$$P_r = \frac{1}{10} \times (r-1), r \in \{1, 2, 3 \ldots n\} \tag{12}$$

Here n is the number of slave nodes. It can be seen that the most successful slave worker will therefore *always* be directed to proceed from its current position, since the probability of reload will be zero. This is important, since our implementation relies on there *always* being at least one worker continuing at every iteration, since one continuing position is used by the master to derive weight vectors with which to reload the nodes that *do not* continue. In general this "source" position is chosen at random with an even probability distribution from all those hosts which are directed to resume, although this can in some circumstances amount to a choice of one.

All of the weight vectors for downloading to slaves are obtained from a single weight vector uploaded from a continuing slave, derived by means of a *mutation* operator. In our initial implementation, mutations are applied on a weight-by-weight basis to obtain new weights W_n from original weights W_o with individual probabilities P as follows:-

- $P = 0.5$: Weight remains unchanged.
- $P = 0.4$: Weight is subject to a slight modification thus:-

$$W_n = W_o + x \tag{13}$$

Here x is a random variable linearly distributed over the range {-0.05,0.05}.

- $P = 0.1$: Weight is completely changed:-

$$W_n = y \tag{14}$$

Here y is a random variable linearly distributed over the range {-1,1}.

8.2.4 Initial Implementation - Results and Discussion

First Evaluation

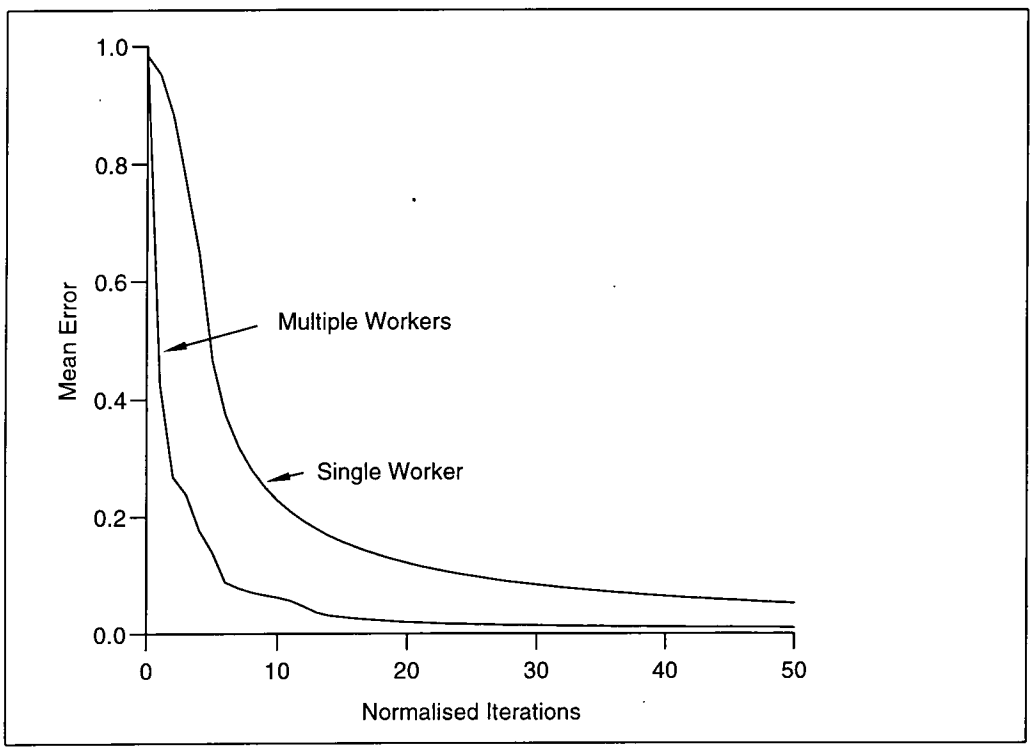


Figure 49: Comparison of training performance between single-node implementation and that with 19 hosts or slave workers.

Figure 49 shows the improved convergence efficiency of the multi-worker scheme described when compared with the previous, single-node implementation. Both applications start from the same initial weight vector which has small, pseudo-randomly generated values for each weight. Identical random number generator algorithms are used together with a predefined "seed" in order to achieve this.

We can therefore deem the comparison to be realistic in the sense that the two competitors are not starting from wildly different parts of the search space. The abscissa of figure 49 represents "normalised iterations", which further helps to allow the two to be fairly compared - it should be noted that in the parallel case these are the number of iterations executed *by each slave.* In fact, a single unit on this axis represents 20 passes through the training set.

It is clear that the initial implementation of this multiple search scheme is therefore a good deal less efficient in terms of total CPU resource consumed, since it consumes in this case 19 times the processor time, plus a small extra quantity due to the overhead involved in network communication, ranking of slave performance etc., without giving a 19 times upgrade in the rate of convergence. In other words, one might say that a law of diminishing returns is in effect - a small performance benefit is available, but at the expense of a disproportionately large extra outlay of resources.

However, this initial experiment does show well that the parallelisation scheme, that is, the very idea of having multiple workers "wandering" around the search space simultaneously is indeed a valid approach to speeding up the training process, and although it does at first sight appear costly in terms of total processor time, it should be borne in mind that this time would otherwise be wasted through underutilisation and in a sense can therefore be considered a "free" resource. The mutation operator, although its initial form was devised on a rather ad-hoc basis, is clearly shown to have at least some beneficial effect on the overall solution speed and consequently it seems that the idea is worth optimising and persuing further.

Mutation and Genetic Search Aspects

The fact that new weight vectors are obtained by *mutating* copies of existing vectors selected on a probabilistic basis according to their relative success or fitness means that there are clear genetic algorithm overtones in our initial parallel-search implementation.

A pure genetic algorithm uses exclusively mutation and recombination of the existing *populus* of potential solutions to produce the next generation - the only processing additionally required is to evaluate each new individual's fitness to enable selection for the *next* generation to be made. If constructed carefully the genetic search should be more efficient than a purely random system, and yet will have a reduced tendency to become stuck in local minima when compared with a deterministic search strategy such as gradient descent. This is so because the genetic method combines elements of both random and deterministic techniques, although the success with which it does so depends rather critically on the nature of the selection and mutation operators. Specifically, the selection mechanism must ensure that the aspects of a current individual solution's make-up can in general be transferred intact to the next generation - this is the algorithm's "gradient descent" aspect. The mutation operator must allow random variations to be occasionally introduced in order that it may be possible to explore beyond local minima, yet this must not be so violent that existing individuals with a high degree of fitness are not destroyed.

The usefulness of the *recombination* aspect of genetic solution generation relies on the principle that it is possible somehow to combine two successful individuals to form a third, higher "fitness" solution in the next generation. Again, the suitability of this approach to finding the global optimum is sensitively dependent on the recombination mechanism or choice thereof available.

The setting of suitable parameters for a successful genetic search algorithm is therefore rather heuristic in nature, and indeed our approach to developing an initial mutation operator reflects this, in that it is based on intuition rather than a solid theoretical basis.

Our search algorithm might, therefore, be regarded as a hybrid between a genetic algorithm and a gradient descent strategy. We have taken from the former the idea of a mutation operator, but in effect replaced the genetic random recombination idea with a more deterministic technique.

Conclusions

Although the distributed search technique does indeed bring a clear improvement to the rate of convergence of the backpropagation training algorithm, the processing required for our initial implementation is at least 19 times that of the single-worker version. The speed-up, however, as can be seen from figure 49, is a factor considerably less than this, and is therefore quite substantially sublinear. Furthermore, figure 49 gives no information about the *sensitivity* of the speed-up to the quantity of resources employed. We therefore sought to investigate this by repeating the experiment with a larger number of hosts.

8.2.5 Initial Scheme Extended to 44 Workers

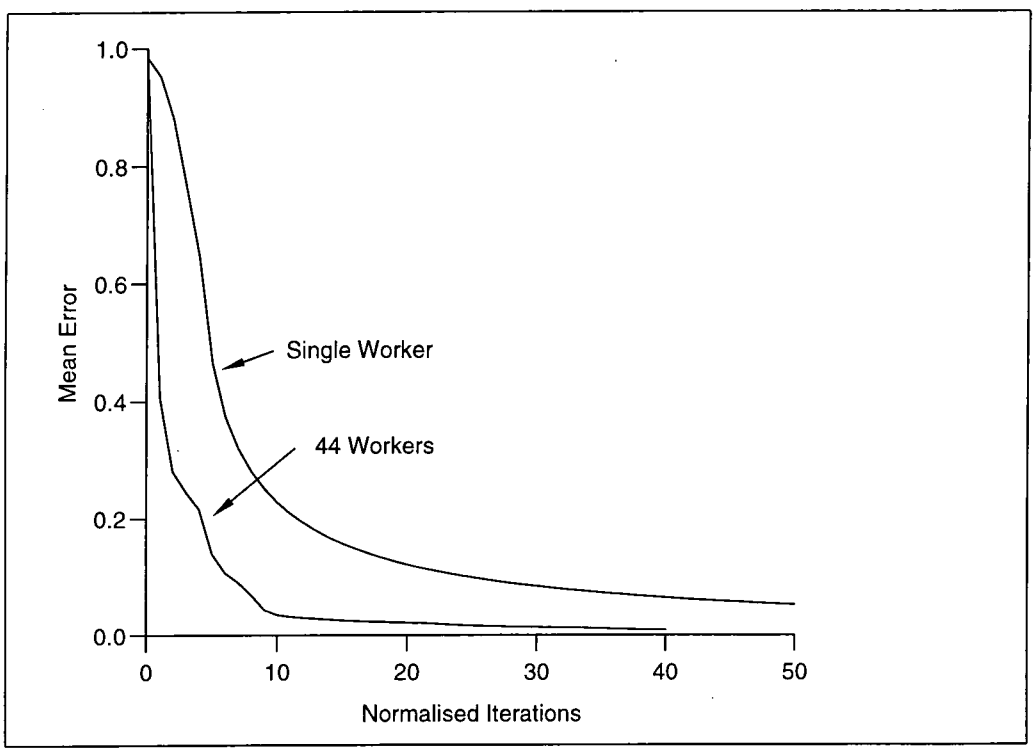


Figure 50: Comparison of training performance between single-node implementation and that with *44* hosts or slave workers.

Figure 50 shows results gathered in an identical manner when the experiment is repeated using 44 rather than 19 hosts. Comparison with figure 49 shows that the beneficial effect from the additional 25 hosts whilst maintaining other algorithm

parameters as before is rather marginal. Our hypothesis at this stage was that the speed-up accruing to the use of parallelisation through mutation is most marked during the early stages of the training process. We attributed this to the fact that the mutation operator, which remains constant throughout, is likely to corrupt a search worker's weight vector to such an extent that the newly-formed vector is unlikely to fare well in the ranking process described in section 8.2.3. Since those weight vectors which are ranked in this way less highly are *more* likely to have their search vector *reloaded* once again, that is, with a new, *mutated* version of a more successful vector, which is itself likely to give a poor mean error after a short period of evolution by backpropagation, then a repetitive cycle of *reloads* occurs.

Correspondingly, according to our suggestion, there should be other workers that undergo repetitive cycles of *continue* instructions. In the later stages of the training algorithm these are searching with weight vectors which give mean errors close to the best which has yet been found, are therefore ranked highly, are unlikely to be subject to the mutation operator which is at this stage allegedly damaging and are therefore likely to continue to be ranked highly in future iterations.

In order to test the veracity of this idea we next analysed the average number of "consecutive reloads" experienced by workers at each stage in the training operation, and figure 51 shows this for the 44-worker experiment just described. These results were obtained by averaging at each iteration the number of consecutive reloads which had just been experienced by hosts *ending* the cycle *at that iteration* with a *continue* instruction, a *reload* instruction having been received at the iteration immediately previous.

Although the data are somewhat noisy due to averaging over a small sample, it can be seen that there is a clear upward trend between the start of the training process and the point approximately 10 normalised iterations into it. This would appear to confirm our outlined hypothesis, namely that the mutation operator is only of real benefit up to this point.

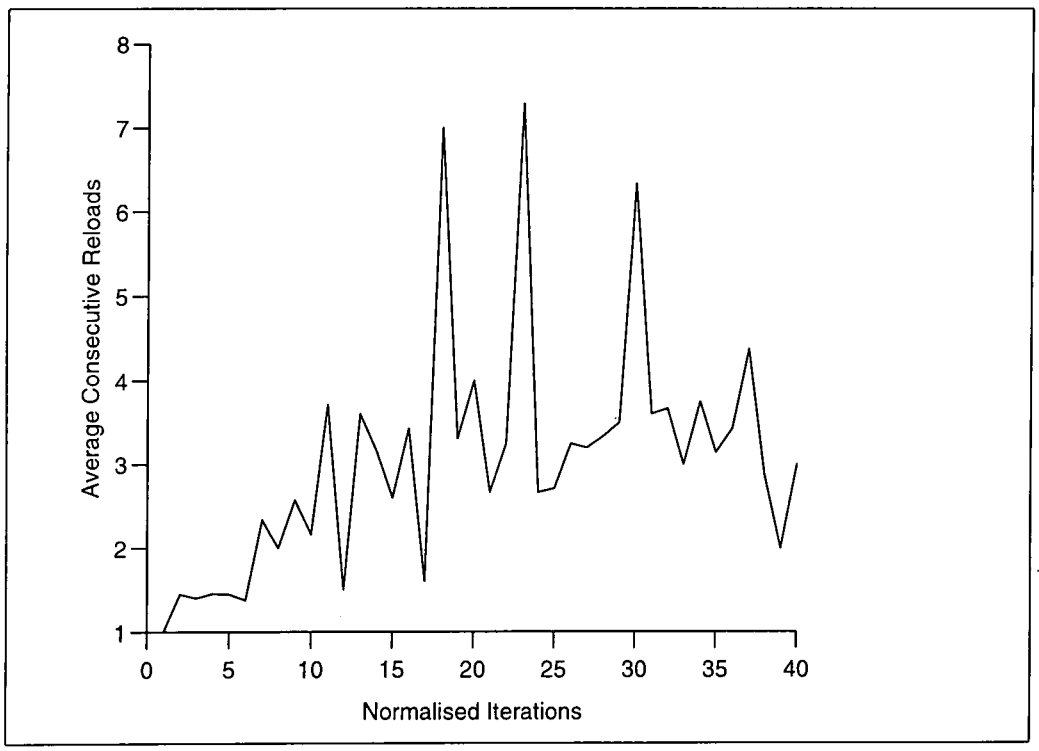


Figure 51: Number of consecutive reloads for 44-worker scheme, averaged over all hosts and plotted against iterations.

8.2.6 Processor Utilisation

8.2.7 Scheduling and Load-balancing

In order to achieve a good degree of a parallel speed-up, the problem of how exactly to partition the work among the CPU resources available needs to be addressed. The optimum size of work "packet" needs to be determined - too small, and network protocol overhead will become excessive. Too large, and the degree of parallelisation achieved will be sub-optimal. Also a strategy must be established for allocating these packets efficiently to remote hosts, which may vary widely in their processing capacity, load from other users and their network data propagation delay. Our application uses the "fixed" allocation scheme below, but there are alternatives:-

1. Fixed allocation. This has the advantage of being easy to arrange, here, the same work distribution is used irrespective of the load on the hosts

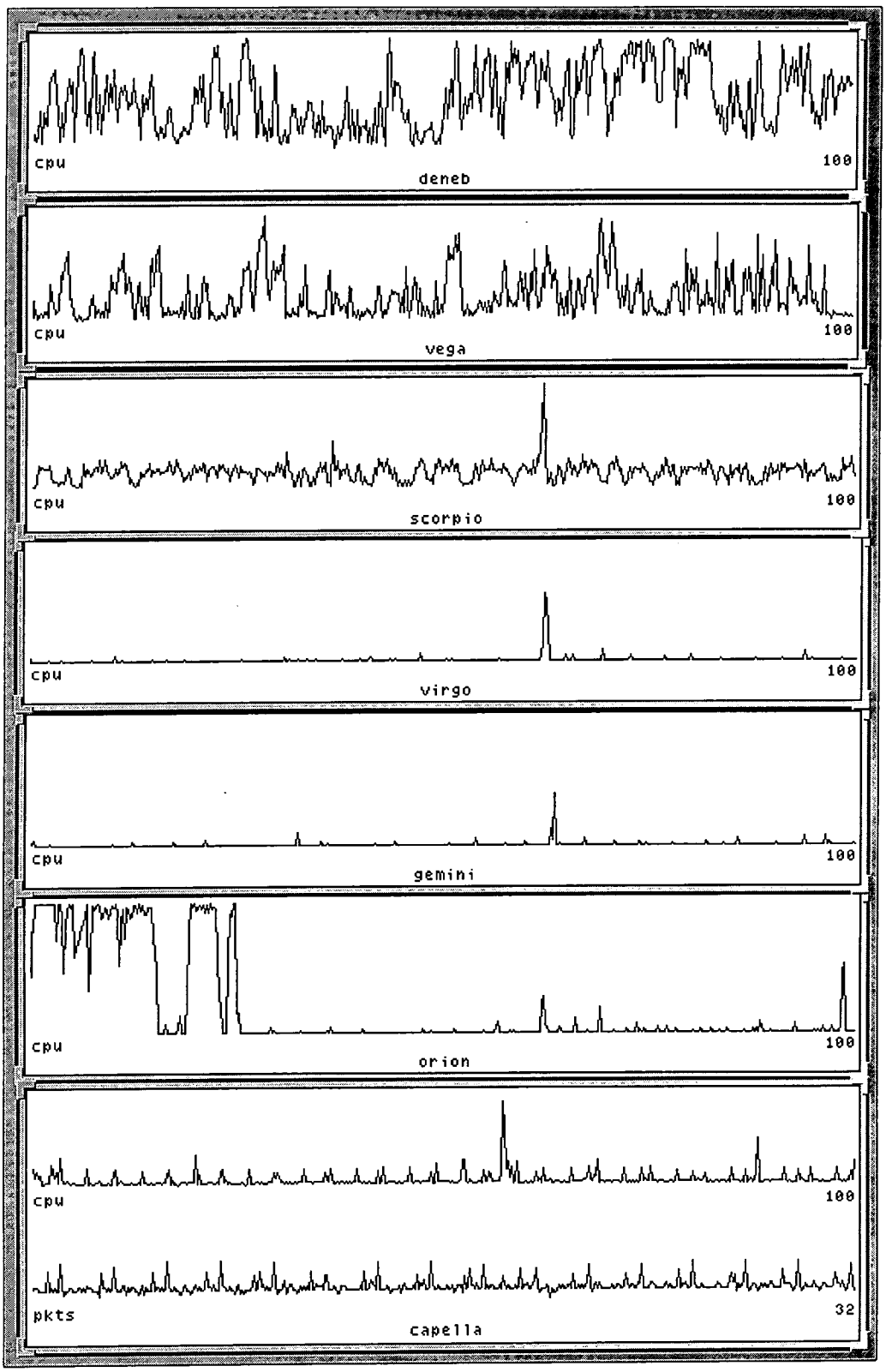


Figure 52: Processor utilisation for 6 of the 44 slave hosts during background application activity. CPU load and Ethernet traffic are shown for *capella*, the controlling node.

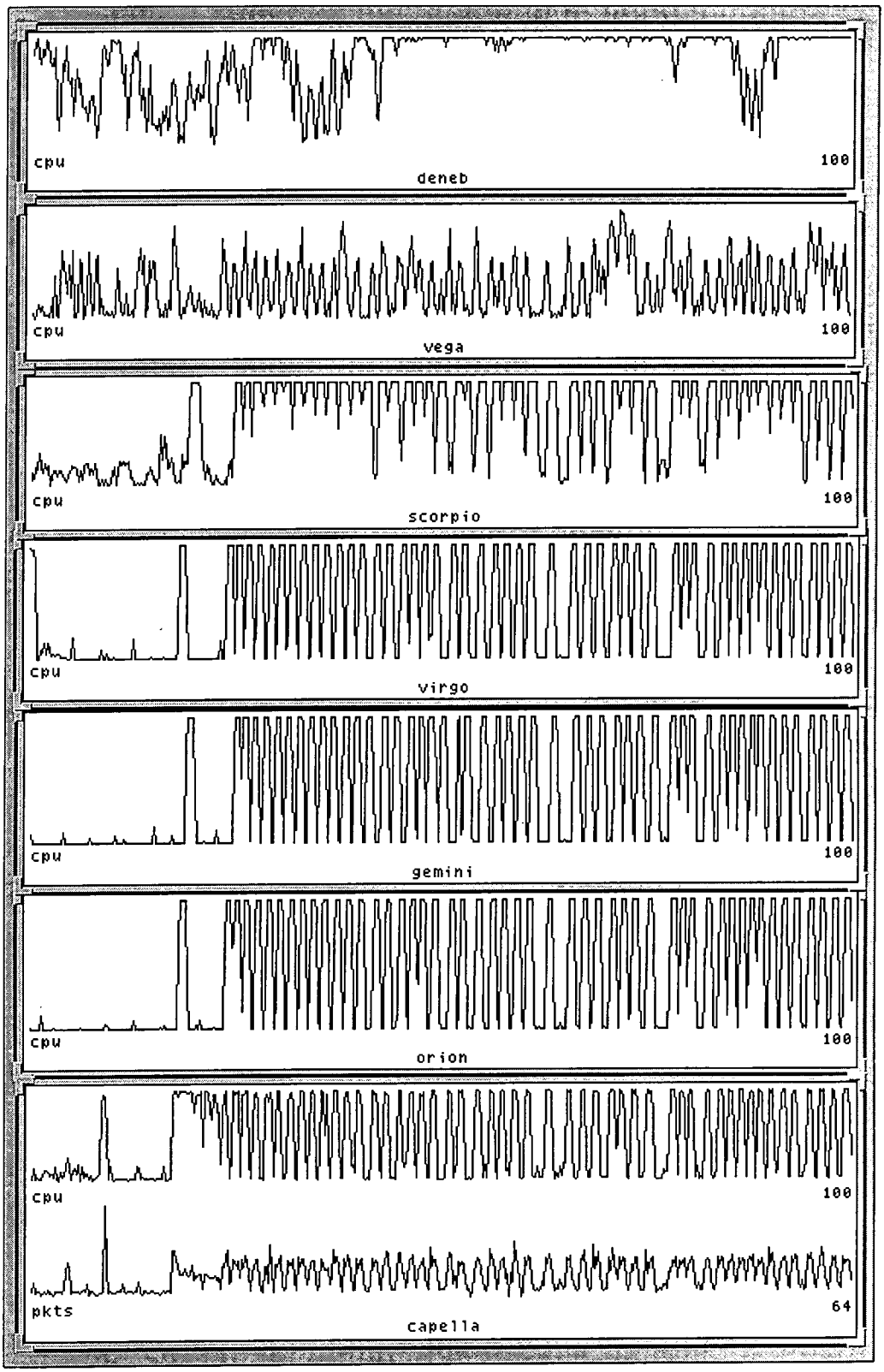


Figure 53: As figure 52, these results taken whilst distributed application is running.

concerned. This might be satisfactory for an environment in which the application user is prepared to allow the slave processes to compete for run-time on the remote hosts on an even basis with other users. However, in a University system such as our own, it is in general necessary to lower the scheduling priority ("niceness") of the slave tasks such that they consume only spare CPU time, that is, time in which the processor would otherwise idle, in order that other users do not experience adverse effects. The disadvantage of this method is that the operation may be held up by a single heavily-loaded host, while other hosts finish their portion of the task and stand inactive.

2. Benchmark allocation. Here, the task scheduler dispatches small test packets of work to all available servers. The order in which the results arrive is logged by the scheduler, allowing the real job to be apportioned between the hosts which were quickest to respond. The advantage here is that the response order is likely to give an accurate indication of which machines will give the best performance at the time the task is issued, taking account automatically of variables such as machine capacity, load and so on. The disadvantage is that unnecessary work is generated for the system. This can be offset, for example, by using a portion of the real task to test the remote hosts' response, in order that the test results can be usefully retained. However, the possibility that some hosts may be very slow indeed to finish the test activity should be considered, requiring their job portion to be reallocated.

3. Algorithmic allocation. With this scheme operational variables are gathered from each potential slave host. An appropriate algorithm is then used to derive the likely performance for each, and the task is then partitioned and allocated on this basis. Information fed to the algorithm could include - machine type, number of runnable processes, number of context switches

and swaps per second, and so on. The advantage is that adaptive load-balancing behaviour is achieved without creating extra load for the system. The disadvantage is that an algorithm which reflects observed performances well is hard to determine, since the systems involved are quite complex.

Figures 53 and 52 give representations of CPU utilisation for some of the workstation nodes involved in the distributed application when respectively it is and is not running. In addition, an indication of the volume of network traffic being processed by the controlling node, *capella*, is also given. The "pulse" effect of the application can be clearly seen; this is due to the fact that all machines wait for the slowest node to finish at each iteration. It is also evident that machines such as *vega* are more powerful than those such as *gemini*, since the CPU peaks are narrower for the former, indicating that the same task is taking less time. Machines such as *deneb* are saturated with work from other applications, and peaks are therefore not visible.

Chapter 9

Conclusions

9.1 Chapter 1 - Introduction

In chapter 1 we decide that the field of image processing as a whole is evolving very quickly, since it comprises aspects of so many different scientific disciplines. Unfortunately, image processing applications are still not as widespread as might be anticipated given their potential, and we conclude that this is partly because both academic researchers and industrialists in the field have hitherto been largely constrained in the scope of their work, the former to theoretical development of algorithms, the latter to practical solutions to highly-specific problems.

Of the three main subdivisions of image processing, namely synthesis, enhancement and recognition, we suggest that image recognition is the least theoretically well-established, and that there are few analytical tools available to guide the designers of recognition systems. However, this area is also the most potentially rewarding in terms of possible applications.

We suggest that the essential elements of an arbitrary image-processing system are the *acquisition*, *storage* and *processing* subsystems. The volume of the processing problem is such that *on-line* processing can typically only be achieved using specialised hardware and fairly straightforward algorithms in terms of computational expense. However, *off-line* processing is frequently used on general-purpose digital processors. A "middle route" also exists, in the form of microprocessors which are optimised for signal-processing, but which are not algorithm-specific.

9.2 Chapter 2 - Industrial Quality Control using Automated Inspection

In chapter 2 we conclude that vision inspection is a particularly useful element of a manufacturing quality control process, since it does not, like manual inspection, lead to eventual loss of inspection performance through operator boredom and fatigue. Of the various types of acquisition transducer available, it appears that charge-coupled detector systems are becoming increasingly popular due to their falling cost and attractive technical specification when compared with alternative systems such as those based on lasers and infrâ-red diodes.

There is great enthusiasm among the authors of the papers reviewed in this section as to the potential applications of CCD-based vision inspection systems, however, it is felt that this potential is as yet not fully exploited in industry, due to high development costs.

In reviewing a cross-section of contemporary inspection applications, we find that there is in general a need to "parameterise" image features before any recognition or classification can take place. However, before this can be done the features must themselves be isolated from the image background. There is a difficulty here in developing reliable techniques - boundary following can be used, although the boundaries thus produced are likely to be incomplete, and some sort of higher-level algorithm, for example maximum-likelihood parametric curve fitting, or the Hough transform, needs to be used in order to fill in the gaps. This is a good example of a more general finding which other authors have observed, which is that, whilst it is straightforward to develop algorithms and techniques which work well for carefully-conditioned and well-behaved data, it is still a large step from here to a fully-functional and robust inspection system which works well on real data. Alternatively, filtering operators can be used on the data in order to segment background from features, although broken boundaries are again a likely problem.

Many different metrics have been used to produce *feature vectors* for classification, including feature area, perimeter, entropy and mean-squared distance from centroid. Although few formal analytical tools are at present available to guide the formulation of preprocessing algorithms, the key appears to be to ensure that sufficient information is present in the feature vectors so as to enable different features to be differentiated, whilst not overloading the feature vector with redundant information, thereby impeding the classification process. For this reason, *orthogonality* of the individual metrics is important. Transforms may be of use at the parameterisation stage if the defect signal can be more easily detected in the transform domain.

There are three types of classifier presently in popular use, these are the *rule-based*, *statistical* and *neural network* varieties. Of these the rule-based classifier is the most straightforward - the classification rules are pre-set manually and may, for example, be based on rules originally specified to guide manual inspectors. Statistical and neural network classifiers derive a rule-set based on analysis of a *training set*, and consequently rule-based classifiers may be of particular use where no training set is available. Of the statistical classifiers, many make assumptions about the distribution of the underlying data, and the choice between them may therefore be made on the basis of the appropriateness of these assumptions. K'th Nearest Neighbour is a particularly significant technique, since it makes no such assumptions and therefore tends to outperform techniques which involve assumptions that are invalid. Neural classifiers, based on the multi-layer perceptron trained by backpropagation, are also known to give good results. However, these are not in general easily amenable to analysis which would enable the rule-set derived by the network to be extracted from it, whereas this is straightforward for classifiers of the statistical type. It is difficult to compare the performance of neural versus statistical classifiers as types, since well-designed implementations of one will tend to outperform bad implementations of the other.

Particular problems relating to automatic *web* process inspection, a sub-set of the general inspection problem have been found to include the fact that the

material flow is generally continuous, leading to a requirement for inspection in *realtime*. Also, the defects are small, and make up a small proportion of the total surface area. This means that a high spatial resolution is generally required to detect them.

9.3 Chapter 3 - Introduction to Networking

In chapter 3 we suggest that the recent explosion of computing and information technology is due for the most part to developments in materials technology rather than in philosophy or logic. We discuss the relevance of networking technology to three important classes of general-purpose computer, the PC, the workstation and the mainframe and find that, whereas networking began with mainframes as a simple extension of the usual I/O facilities to enable remote use from terminals situated elsewhere in the building, the high-bandwidth peer-to-peer networks which are now used to connect PCs and workstations open up a very much wider range of possibilities, including distributed application processing. Although we mention that the cost-effectiveness of the computing made available by the workstation class is such as to seriously encroach into the mainframe sector of the market, centralised computing continues to have some attraction partly because of the availability of cheap high-bandwidth telecommunications, since there are cost benefits to be gained through maintaining only *one* air-conditioned room, team of technicians and so forth for some kinds of organisation.

In our consideration of protocols from the TCP/IP family we suggest that the possible methods of achieving communication between different processing nodes in a parallel distributed application include, from lightest-weight to heaviest-weight of protocol, socket-based connections using UDP or TCP, remote procedure calls or use of shared files in a network file system. We examine the implementation difficulties which are likely to be encountered when making use of these techniques, and conclude that these become progressively more serious for the lighter-weight techniques. However, these, more problematic protocols

are also more efficient in terms of communications throughput. Given that communications protocol efficiency is a key variable in determining the optimum size for a distributed application cluster, we conclude that a trade-off needs to be made when designing such an application, based on the compute/communicate ratio demanded by that application. Where the volume of inter-node communication is small, for example, the application may be quickly implemented by using the more straightforward, higher-level although less efficient protocols, without adversely affecting performance to a great extent.

9.4 Chapter 4 - Operating Systems

In chapter 4 we examine the evolution of operating systems and the considerations which arise when choosing among those available in the design stage of a machine vision application. We suggest that the operating system has grown in importance as the complexity of the computer hardware has increased, and that the most important function of the operating system is to increase ease of use of the hardware by providing a simpler interface to it, thereby reducing application development time. This is done by *re-using* the low-level routines which perform commonly-required tasks. Although applications are sometimes still written using the lowest-level machine language, we suggest that this is done only for performance-critical components, or in an educational context.

From our considerations of Microsoft's DOS as a potential candidate operating system on which to base a machine vision application, we conclude that this system's design for backward compatibility with the original IBM PC gives rise to a segmented memory architecture which causes serious problems for memory-intensive applications, since no standard interface is available for accessing memory above the 1 MB boundary. Although support for access to the extra memory was available through the use of specialised compilers, use of these would have been mutually exclusive of use of the compiled libraries supplied by the hardware manufacturer. Our experimental work on network programming under MS-DOS

demonstrated that it was possible to achieve asynchronous networking with the use of "terminate and stay resident" application code. However, the lack of operating system support in terms of memory reservation makes such development very troublesome and time-consuming, particularly since the use of TSRs requires manipulation of chains of software interrupts. In addition, the described memory segmentation problem is compounded since the use of a TSR reduces the amount of memory available for the conventional part of the application.

Our considerations of Unix as a candidate operating system led us to revise our original estimation that a Unix system would be best employed as the processing "workhorse" of a machine vision distributed application, since the provision of freely-available libraries makes the development of a powerful windows-based graphical front-end very straightforward. Unix provides the features whose absence in MS-DOS causes such difficulty - a memory model which is linear, as far as the application is concerned, as well as reservation protection which speeds the debugging phase of development. Therefore complex applications are ideally suited to a Unix machine. However, peripheral hardware which is compatible with typical workstation architectures is in general considerably more expensive than the PC-bus equivalents. Therefore our optimal solution in a value-for-money sense involves a PC which interfaces to a Unix workstation hosting the main application via TCP/IP over Ethernet. We found Linux, a freeware implementation of Unix, to be the best operating system for use on this PC, since the "open" nature of this package makes it straightforward to design Unix internals such as device drivers. Of the two methods of device-driver writing considered, we found that the extra complexity of a system-mode driver was not warranted, although this method might usefully be employed where DMA or hardware interrupts are in use by the peripheral, or where contention between processes for access to the peripheral needs to be mediated. A user-mode device-driver was found to be adequate for the application in view.

9.5 Chapter 5 - Neural Network Overview

In chapter 5, we formalise the breakdown of a general automated inspection application into the data acquisition, preprocessing, artificial intelligence and output processing stages. In examining an inspection system designed to tackle a problem closely-related to the one in view, we conclude that one of the most serious limitations is that algorithm parameters and threshold values need to be determined through experience and experimentation in using the system on real-world data. We suggest that this is an important factor limiting the uptake of industrial vision inspection systems, and that this might be encouraged if a more generally-applicable system could be designed. With this in mind we examine neural network approaches to classification, since the adaptive properties of these are well-known.

In our review of neural network theory we explain that, whilst the idea of a neural network is biologically inspired, the accuracy of the analogy between biological and artificial neural systems is very limited. We also suggest that one of the most significant drawbacks to the use of neural networks is the fact that they construct a "write-only" knowledge base.

We describe the characteristics of the Hopfield network and suggest that this is one of the most potentially versatile of all networks, since it possesses the ability to adopt an arbitrary structure. Unfortunately, much theoretical development needs to be made before these properties can be exploited to their full potential. Existing theory relating to the use of the Hopfield network as a distributed memory store is based by analogy on that relating to magnetic materials in Physics.

We explain the development of perceptron networks and the relevance of the backpropagation algorithm to the training of multi-layer perceptron networks. Although this has been shown to be a successful training technique, we point out that there is no evidence of any backpropagation-like processes at work in any known biological systems. We suggest that the large volume of computation

required for successful training is a serious barrier to neural application development. Other problems include the lack of analytical tools which would allow the appropriate network structure for a given problem to be directly determined; a trial and error procedure is usually adopted at present.

In reviewing contemporary neural applications, we find that there is a good deal of variance of opinion relating to the use of neural networks, but there is a concensus that data preprocessing is of crucial importance to network performance, also that neural classifiers show great potential which can be realised through tuning of the preprocessing and training stages. There is also a general view that a neural network solution is always "second best" to a custom-designed solution, however, the general applicability and self-teaching of the neural approach is what makes this worth following.

9.6 Chapter 6 - Image Preprocessing Considerations

In chapter 6 we explain that the essence of preprocessor design is to strike a compromise between too little preprocessing, which leaves the following stage with a classification problem which is too difficult to solve, and doing so much preprocessing that the key features of data are removed.

Our investigations into the Iterated Function Series as a preprocessing operator are partly biologically inspired, since the IFS will "build up" its attractor over many iterations in the same way that humans are thought to "build up" mental representations of real world objects.

Our experiments suggest that the IFS is a robust system in the sense that small changes in defining parameters will not result in chaotic changes in the attractor. However, we conclude that there is no evidence to suggest that a particular image pattern has a single, unique mapping to one position in IFS coefficient space. Consequently we suggest that recognition on the basis of IFS coefficients alone

is unlikely to prove fruitful. We put forward the fractal dimension, a metric of texture self-similarity theoretically-related to the IFS, as being a useful feature parameter because of its likely orthogonality with conventional metrics.

Our experimentation with the Fast Fourier Transform as a preprocessing operator show that detection is effective in the transform domain. However, when compared with a similar detection technique in the spatial domain, we find that detection is not significantly aided by the use of the transform, and that its computational expense is therefore not justified.

9.7 Chapter 7 - Neural Network Experimental

In chapter 7 we successfully test our implementation of a multi-layer perceptron trained by backpropagation by showing the convergence properties for random training data using networks with a varying number of hidden nodes. The network cannot be successfully trained on random data where there are fewer hidden than input and output nodes, a result consistent with information theory, and this finding contributes to the validation of our implementation. The pattern of weights evolves differently for each trial since each weight is initialised to a small, random value.

The network configuration was altered, in order that real image data of plate aluminium defects could be used for training and analysis. The network was successfully trained on a set of horizontal image cross-sections, half containing defects, half containing normal material. The use of the trained network on fresh data meets with limited success, and our hypothesis that the network has learnt to identify defects purely on the basis of position is consistent with the results of further experiments.

In order to eliminate this "position-sensitive" effect we add a further preprocessing stage to the RMS operator and present to the network a distribution of populations of RMS powers rather than the RMS powers themselves. The detection results in this case are inconclusive, and we attribute this to the fact that the

trained network attaches a greater significance to the general distribution shape than to the high power bins which contain the defect information.

We further seek to eliminate the "position-sensitive" effect through the use of a "synthetic" training set, which contains defects in a range of horizontal positions. Although the effectiveness of the network in responding to the presence of defects is now improved, spurious defect responses are now generated for normal material. We attribute this to the fact that the training set exemplars of normal material are insufficiently general, and the fact that the exemplars of defective material have a higher average background RMS power level.

Our efforts to examine the knowledge induced by the training process are quite successful, and we see that the network has become sensitised to high RMS powers in positions coincident with the defects in the synthetic training set, and desensitised to high RMS powers in positions where *both* halves of the training set possess them.

Although some progress was made towards understanding the neural network behaviour, the training time for each experiment proved to be, as predicted, a serious inhibiting factor towards further progress. We therefore sought to speed up the training procedure. This was the motivation for the work in chapter 8.

9.8 Chapter 8 - Algorithmic Development

In chapter 8 we suggest that parallel processing is an efficient way to tackle very computationally-demanding applications providing that these are amenable to parallel decomposition, since the cost per unit processing speed is lower for an array of medium-power processors than for a single supercomputer.

Through reviewing contemporary parallel distributed applications we conclude that there is a spectrum of such applications ranging from coarse-grained to fine-grained in the nature of their parallelism. Coarse-grained applications require only limited communication between nodes, whereas the functionality of the nodes themselves is complex; for fine-grained applications the converse

is true. Different architectures are appropriate for these different applications; fine-grained applications for example are best-suited to tightly-coupled microprocessors sharing memory over a single bus.

We implement a parallelised backpropagation training algorithm using a cluster of Unix workstations interconnected by TCP/IP over a local area network. The coupling between nodes here is relatively loose in bandwidth terms, whilst the nodes are relatively powerful in processing and resource terms, and we therefore design our application to be coarse-grained in nature.

The backpropagation algorithm is parallelised by expressing it as a search problem which has a search vector travelling over a surface in coefficient or weight space in search of a global optimum. The parallel algorithm has many search vectors, one per node, which independently traverse the surface. Periodically the nodes communicate the "fitness" of the solution which each has found to a controlling node which then ranks each node's results in terms of performance. On a probabilistic basis the nodes which perform more poorly are instructed to abandon their current search positions and resume searching from a new position derived from that reached by a better-performing node. The derivation here involves genetic-like mutation and thus enables the search to avoid local optima without being subject to the inefficiency of a purely random search.

Our initial trial with 19 workstations, one slave search worker per workstation, shows that a considerable speed-up is achieved and that the basis of the parallel search algorithm is therefore sound. The speed-up is sublinear, however it is clearly unreasonable to expect linear speed-ups from such an algorithm and this does not reflect badly on the networking and processing configuration. Although the mutation operator used was designed on a heuristic basis, in the absence of appropriate analytical tools, it is shown to have a beneficial effect on search performance.

In extending the scheme to 44 workstations we conclude that the extra benefit achieved is marginal. We hypothesise that this is at least partly due to the fact that the mutations introduced are sufficiently vigorous that, once the search is

quite well-advanced, mutation generally produces a worse-fitting search vector than the one from which the vector is derived. This hypothesis is borne out by investigation of the number of consecutive worker "reloads" plotted against search iterations.

Examination of the CPU utilisation of a selection of hosts involved in the distributed application shows that a good deal of inefficiency is present, since all workers must wait for the slowest worker to complete at each iteration. A useful direction for further work would therefore be to implement the mentioned benchmark or algorithmic schemes in order to improve the networking/processing efficiency, as well as to tune the mutation parameters so as to achieve better algorithmic efficiency.

9.9 Final Conclusion

In this thesis we have identified the need for general-purpose image inspection solutions, and have made progress towards these along a number of avenues. Since it appears that such a system is likely to take the form of a heterogenous mixture of architectures, our work on network analysis and hardware interfacing is of value. The proper paradigm for a general-purpose image-processing system is now well-established, and there is a clear need for efficient image preprocessing, to which our considerations of the Iterated Function Series and the Fast Fourier Transform are relevant. Although neural networks are not proven as the method of choice for future classifier design, there appears to be considerable untapped potential in this area, and this therefore seems to be a good direction for future work. Finally, parallel algorithm design using local area networks is shown to be a low-cost method of achieving the provision of supercomputer-like resources, and the particular application in view, backpropagation training, is successfully speeded up, thereby reducing the development time of future neural systems. Furthermore, if the highlighted implementation inefficiencies were tackled in future work, it seems likely that even better performance would be yielded.

Bibliography

[1] Kenneth Castleman. *Digital Image Processing.* Prentice-Hall, 1979.

[2] Paul Wintz Rafael Gonzalez. *Digital Image Processing.* Addison Wesley, 1987.

[3] M. Kilger. Video-based traffic monitoring. In *4th International Conference on Image-Processing and its Applications, Maastricht, Netherlands.* Siemens AG, Germany, 1992.

[4] E. I. Dagless A. T. Ali. Automatic traffic monitoring using a transputer image-processing system. In *Second International Conference on the Application of Transputers.* IOS Press, 1990.

[5] B. J. Hosticka J. Moeschen. Programmable hardware architecture for real-time pattern recognition systems. In *4th International Conference on Image-Processing and its Applications, Maastricht, Netherlands.* Fraunhofer-Institute of Microelectronic Circuits and Systems, Germany, 1992.

[6] M. C. Liu C. N. Huang, C. C. Lim. Comparison of image-processing algorithms and neural networks in machine vision inspection. *Computers and Industrial Engineering*, 23(1-4):105–108, 1992.

[7] F. Martel P. I. Ivonen. Defect detection and online analysis of coated papers. *Appita*, 44(5):305–306, 1991.

[8] J. L. C. Sanz. Machine vision algorithms for automated inspection of thin-film disk heads. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(6):830–848, November 1988.

[9] R. A. Schowengerdt R. G. White, D. A. Perednia. Automated feature detection in digital images of skin. *Computer Methods and Programs in Biomedicine*, 34(1):41–60, 1991.

[10] F. Deravi A. K. Muhamad. Cooccurence-based features for automatic texture classification. *Neural and Stochastic Methods in Image and Signal Processing*, 1766(74):489–496, 1992.

[11] R. N. Bracewell D. Mihovilovic. Adaptive chirplet representation of signals on time-frequency plane. *Electronics Letters*, 27(13):1159–1161, 1991.

[12] J. Waite. A review of iterated function system theory for image compression. *IEE Colloquium, The Application of Fractal Techniques in Image-Processing*, December 1990.

[13] A. E. Jacquin. A novel fractal block-coding technique for digital images. *IEEE ICASSP Proceedings*, pages 2225–28, 1990.

[14] P. A. Araman T. H. Cho, R. W. Conners. A comparison of rule-based, k-nearest neighbour and neural net classifiers for automated industrial inspection. In *Proceedings of the IEEE/ACM International Conference on Developing and Managing Expert System Programs*, pages 202–209. The Spatial Data Analysis Laboratory, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, 1991.

[15] N. Sufi D. Brzakovic, H. Beck. An approach to defect detection in materials characterised by complex textures. *Pattern Recognition*, 23(1-2):99–107, 1990.

[16] R. W. Conners T. H. Cho. A neural network approach to machine vision systems for automated industrial inspection. In *Proceedings of the Neural*

Networks International Joint Conference, volume 2, pages 205–210. Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, 1991.

[17] S. Gruber J. Olsson. Web process inspection using neural classification of scattering light. In *Proceedings of the Industrial Electronics, Control, Instrumentation and Automation International Conference*, pages 1443–1448. Department of Electrical Engineering and Applied Physics, Case Western Reserve University, Cleveland, Ohio, USA, IEEE, 1992.

[18] S. C. Cheung J. N. Magee. Parallel algorithm design for workstation clusters. *Software: Practice and Experience*, 21(3):235–250, 1991.

[19] V. Strumpen C. H. Cap. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19(11):1221–1234, 1993.

[20] F. Mueller-Plathe. Parallelising a molecular dynamics algorithm on a multiprocessor workstation. *Computer Physics Communications*, 61(3):285–293, 1990.

[21] W. Huber M. Griebel. The combination technique for parallel sparse-grid-preconditioning or -solution of pde's on workstation networks. *Lecture Notes in Computer Science*, 634:217–228, 1992.

[22] A. Gupta E. Rothberg. Techniques for improving the performance of sparse matrix factorisation on multiprocessor workstations. In *Supercomputing*, pages 232–241. Department of Computer Science, Stanford University, California, USA, 1990.

[23] G. Amdahl. Validity of the single-processor approach to achieving very large scale computing capabilities. In *Proceedings of the AFIPS Computing Conference*, pages 483–485, 1967.

[24] J. Postel. *RFC 768: User Datagram Protocol.* Network Information Center, SRI International, Menlo Park, CA, USA, 1980.

[25] J. Postel. *RFC 793: Transmission Control Protocol - DARPA Internet Program Protocol Specification.* Network Information Center, SRI International, Menlo Park, CA, USA, 1981.

[26] Richard G. Palmer John Hertz, Anders Krogh. *Introduction to the Theory of Neural Computation.* Addison Wesley, 1992.

[27] Roy Rada Richard Forsyth. *Machine Learning, Applications in Expert Systems and Information Retrieval.* Ellis Horwood, 1986.

[28] Frank Rosenblatt. The perceptron: a probabilisitc model for information storage and organisation in the brain. *Psychological Review*, 65, 1958.

[29] Seymour Papert Marvin Minsky. *Perceptrons.* MIT Press, 1969.

[30] Casimir A. Kulikowski Sholom M. Weiss. *Computer Systems That Learn.* Morgan Kaufmann, 1991.

[31] C. C. Hand M. R. Evans, S. W. Ellacott. A multi-resolution neural network classifier for machine vision. In *Proceedings of the Neural Networks International Joint Conference*, volume 3, pages 2295–2599. Information Technology Research Institute, Brighton Polytechnic, UK, 1991.

[32] T. J. Hall G. D. Kendall. Performing fundamental image-processing operations using quantised neural networks. In *4th International Conference on Image-Processing and its Applications, Maastricht, Netherlands.* King's College London, UK, 1992.

[33] P. A. Giles. *Iterated Function Systems and Shape Representation.* PhD thesis, University of Durham, 1991.

[34] M. A. Fischler. On the representation of natural scenes. *Computer Vision Systems*, 1978.

[35] Michael F. Barnsley. *Fractals Everywhere.* Addison Wesley, 1988.

[36] Kenneth Falconer. *Fractal Geometry.* Wiley, 1993.

[37] A. D. Sloan M. F. Barnsley. A better way to compress images. *Byte Magazine*, January 1988.

[38] D. D. Giusto F. Arduini, S. Fioravanti. Multifractals and texture classification. In *4th International Conference on Image-Processing and its Applications, Maastricht, Netherlands.* University of Genoa, Italy, 1992.

[39] Z Houkes T. Feng. Internal measuring models in trained neural networks for parameter estimation from images. In *4th International Conference on Image-Processing and its Applications, Maastricht, Netherlands.* Ocean University of Qingdao, P R China, 1992.

[40] M. K. Molloy J. J. Devai, T. C. Kerrigan. Simulation of the lan behaviour in the distributed hp-ux environment. In *Proceedings of the 1990 Winter Simulation Conference*, 1990.

[41] F. Deravi A. K. Muhamad. Neural networks for texture classification. In *4th International Conference on Image-Processing and its Applications, Maastricht, Netherlands.* University of Wales, Swansea, UK, 1992.

[42] V. S. Sunderam G. A. Geist. *Network Based Concurrent Computing Using the PVM System.* Oak Ridge National Laboratory, Oak Ridge TN, USA.

[43] J. Postel. *RFC 791: Internet Protocol - DARPA Internet Program Protocol Specification.* Network Information Center, SRI International, Menlo Park, CA, USA, 1981.

[44] E. M. Ormsby J. W. Hamilton. Simulating hypercubes in unix. *Dr Dobb's Journal*, 17(12):72–76, 1992.

[45] M. Weber. Workstation clusters: One way to parallel computing. *International Journal of Modern Physics C: Physics and Computers*, 4(6):1307–1314, 1993.

[46] T. Nakada S. Horiguchi, A. Katahira. Parallel processing of incremental ray tracing on a multiprocessor workstation. In *Proceedings of the International Conference on Parallel Processing*, pages 192–196. Department of Information Science, Tokohu University, Sendai, Japan, 1991.

[47] P. J. King. Process-level parallelism in a unix environment. *Transputer Research and Applications*, 3(33):189–195, 1990.

[48] T. Williams. Realtime unix develops multiprocessing muscle. *Computer Design*, 30(5):26–30, 1991.

[49] E. B Fernandez R. Bealkowski. A heterogenous multiprocessor architecture for workstations. In *IEEE Proceedings of the Southeastcon*, pages 258–262. International Business Machines Corporation, Boca Raton, Florida, USA, 1991.

[50] K. Blathras Y. Shi. Parallelizing scatter and gather applications using heterogenous networked workstations. In *Proceedings of the 5th Siam Conference on Parallel Processing for Scientific Computing*, pages 588–595. Department of Computer and Information Sciences, Temple University, Philadelphia, USA, 1991.

[51] K. de Jong. Adaptive system design: a genetic approach. *IEEE Transactions on Systems, Man and Cybernetics*, 10(9):566–574, September 1980.

[52] C. L. Hendrick. An introduction to the internet protocols. Technical report, Computer Science Facilities Group, State University of New Jersey, 1987.