2002

Extensions and evaluations of adaptive processing of structured information using artifical neural networks

Markus Hagenbuchner
University of Wollongong, markus@uow.edu.au

Extensions and Evaluations of Adaptive Processing of Structured Information using Artificial Neural Networks

A thesis submitted in partial fulfilment of the requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Markus Hagenbuchner
markus@artificial-neural.net

Faculty of Informatics

2002

This work was partially funded by ARC large grant A49804194 awarded to A.C. Tsoi, and M. Gori. The author also acknowledges partial financial support from MURST grant 9903244848 and MM09308497 awarded to A. Sperduti, and a grand from Autostrade/Italy awarded to M. Gori which supported extended visits to Pisa, and Siena respectively.

Abstract

The application of Artificial Neural Networks has traditionally been restricted to fixed size data and data sequences. However, there are a large number of applications which are more appropriately represented in the form of graphs. Such applications include learning problems from the area of molecular chemistry, software engineering, artificial intelligence, image and document processing, and numerous others. The inability of conventional Artificial Neural Networks to encode this kind of data has motivated for research in this field.

Early successes with multilayer perceptron (MLP) based and auto-associative models for adaptive processing of graph models demonstrated that classic models of artificial neural networks can be extended so as to allow the encoding of graph structured information. This thesis continues this line of research and proposes further neural architectures for this kind of domains. Most noteworthy is the development of Self Organising Map (SOM) based models which extend the capabilities of standard SOM to allow the mapping of graph structures onto an n-dimensional map. This ability and the efficiency of this model is illustrated quantitatively on a range of experiments. A further extension of this model allows the inclusion of supervision to the learning process. It is demonstrated that this extension has a significant benefit on the network performance. It is also shown that the proposed models are a superset which includes the standard SOM algorithm as a special case, and that the extension of the models does not impose additional cost to the learning process. Hence, the proposed models represent a more general form of SOM capable of processing graphs, data sequences, and also fixed size input vectors.

The recent introduction of a Cascade Correlation (CC) based model for problems which are governed by graph based domains motivated a further modification of the CC model and the MLP based model. These modifications give respectively two models which are of similar network architecture. Since CC and MLP based models build and train the network in a very different manner, it is interesting to compare their performances with similar architectures. It is shown that each of the two new models feature advantages as well as disadvantages but that the combination of the underlying ideas may help to eliminate some of the disadvantages. Again, these models include their traditional counterparts as a special case, and hence, mark these model to be a more general form.

The above mentioned models have been analysed individually on practical examples, and the findings are then compared with one another. The result of these comparisons allows the drawing of conclusions stating the pros and cons for each model. In order to allow such comparison, a benchmark problem is deployed. The learning problem defined by this benchmark is a classification task. The flexibility and other advantages of this benchmark problem are illustrated, contributing to its acceptance by the research community. Experimental findings are confirmed through the application to a real world problem, viz., the logo recognition problem.

Contribution of this thesis

The contributions of this thesis are multifold. This is in great parts due to the fact that research in the area of *Adaptive Processing of Data Structures* is very new, which is in particular true when deploying Artificial Neural Networks for the processing of graph structured information. There were few models available when this project started. Most noteworthy are a MLP based architecture known as recursive multilayer perceptron (RMLP) introduced by [33, 86], a cascade correlation based model known as recursive cascade correlation (RCC) introduced by [9], and a type of auto-associate memory known as labelled recursive auto-associative memory (LRAAM) introduced by [80]. A more complete list of literature available at that time is available in Appendix C. At that stage it was not known whether it is possible to extend other popular neural models, e.g., the Self Organising Map (SOM), to the processing of graph structured information. In addition, only a limited set of practical experience existed with the existing models and a comparison of the models had not yet been addressed which was mainly due to the lack of a benchmark problem which would have allowed such a comparison.

The contributions of this thesis can be categorised into three sections as follows:

1. The introduction of four novel neural network models capable of processing graph structured information. The first model, which we called a SOM-SD (Self-Organising Map for Structured Data) extends the capabilities of Self-Organizing Maps [58]. This is done by adding information about the structure of the input data to the network input. While Self-Organizing Maps traditionally are trained in an unsupervised fashion, this thesis also proposed a novel model, called supervised SOM-SD (sSOM-SD) which incorporates a supervisor signal into the learning process if such a signal exists. It is demonstrated that this model which is capable of mapping graph structured information onto a Self-Organizing Map in a supervised fashion is capable of handling incomplete or missing target information.

   The third and fourth of the new models are respectively a modification of the existing RMLP and RCC models. The thesis proposes a novel model, called extended RMLP model to incorporate additional network parameters (additional connections) into a RMLP architecture to make it more similar to the RCC architecture. In turn, the modification of the RCC model suggests the removal of some network parameters (removal of connections) so that the resulting network architecture is similar to the RMLP architecture. This is called a reduced RCC model. These are particularly interesting modifications since these two models, extended RMLP, and reduced RCC, build up the architecture in a very different manner, and hence, it is interesting to learn how these two models compare if the underlying architecture is similar.

2. A comparison of adaptive models which are capable of processing graph structured information is not possible without a generally agreed benchmark problem. This thesis introduces the first benchmark problem, called a policeman problem, for this area. The benchmark problem is aimed at providing an arbitrarily sized set of data for a classification learning task. The benchmark allows the construction of a policeman with various features, e.g., raised or lowered left arm, whether wearing a hat, etc. This benchmark problem is also extended to give the extended policeman problem, which in addition, allows incorporation of other entities, e.g., house, boat. The advantages of these proposed benchmark problems are multifold. Firstly, it is created through a formal language, viz. an attributed

plex language. Hence, a potentially infinite large dataset can be distributed easily by simply providing a relatively small set of formal productions. Secondly, since the dataset is produced by a grammar and not by random distributions, it resembles training tasks which are more closely related to real world learning problems. Thirdly, the interpreter for the attributed plex language which is provided with the dataset produces pairs of outputs which give both an image and a graph representing this image. Using images instead of graphs makes it easier for a human observer to understand the relationship between any two graphs, and help to evaluate models to which this dataset is applied.

These benchmark problems have been presented to the research community and has already been accepted, and is currently being applied by a number of researchers, some of which are active in the area of statistical models.

3. This thesis is furthermore applying the extended policeman benchmark problem to nearly all known neural network models which are capable of processing graph structured information. The experiments aim at the practical evaluation of the various models. The models have to demonstrate the ability to efficiently encode graph structured information. Also, the behaviour of the models during training is observed. Both performance and behaviour of the models are then compared with each other and conclusions are drawn.

   Of particular interest of these experiments is the comparison between the new models, viz., extended RMLP, and reduced RCC, introduced in this thesis, and models, viz., RMLP, and RCC, that have already existed. In addition, the models are also applied to a real world learning problem, viz., logo recognition problem, which allows the observation of whether the proposed benchmark problem sufficiently represents real world cases.

The following list of publications were a direct result of research done on this project[1].

- M. Hagenbuchner, A.C. Tsoi, A. Sperduti, "A Self-Organizing Map for Adaptive Processing of Structured Data", Submitted to IEEE Transactions on Neural Networks in November 2001. (From materials contained in Chapter 4 and Chapter 5)
- Z. Wang, M. Hagenbuchner, A.C. Tsoi, S.Y. Cho, Z. Chi, "Image Classification with Structured Self-Organization Map", Accepted for publication in IJCNN 2002. (Main results not contained in this thesis. But the concept is mentioned in Chapter 4)
- M. Hagenbuchner, M. Gori, A.C. Tsoi, H. Bunke, C. Irniger, "Using attributed Plex Grammars for the generation of Image and Graph Databases", Accepted for publication in Mario Vento, editor, *Special issue PRL-Graph-based Representations*, 2002. (From materials contained in Appendix A)
- M. Hagenbuchner, A.C. Tsoi, M. Gori, H. Bunke, C. Irninger, "Generation of Image Databases using Attributed Plex Grammars", in 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, pages 200-209, 2001. (From materials contained in Appendix A)
- M. Hagenbuchner, A.C. Tsoi and A. Sperduti. "A Supervised Self-Organizing Map for Structured Data". In N.Allinson, H.Yin, L.Allinson

[1]The list of publications is sorted by date.

and J.Slack, editors, *Advances in Self-Organising Maps*, Pages 21-28, Springer, 2001 (From materials contained in Chapter 5)

- M. Hagenbuchner, A. Micheli, and A.C. Tsoi, "Building MLP Networks by Construction", in Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (IEEE Computer Society Press), pages 549-554, volume 4, 2000. (Preliminary results presented. Detailed experimental results (not yet published) are contained in Chapter 7)
- Hagenbuchner, Markus and Tsoi, Ah Chung. The Traffic Policeman Benchmark. In Michel Verleysen, editor, *European Symposium on Artificial Neural Networks*, D-Facto, pages 63-68, April 1999. (Preliminary results presented. Further experiments are contained in Appendix A)

The following publication was an indirect outcome of research done for this thesis:

- H.A. Kestler, S. Simon, A. Baune, M. Hagenbuchner, F. Schwenker, and G. Palm, "A Hierarchical Neural Object Classifier for Subsymbolic-Symbolic Coupling", in Mustererkennung, volume 21, Springer, 1999. (The material described in this paper uses a superset of utilities developed for the policemen benchmark. Basic concepts are presented in Appendix A)

Glossary

ANN Artificial Neural Networks aim at emulating the behaviour of neurons or neural assemblies in the brain.

BPTS Backpropagation through structure, a gradient based updating rule for recursive network models.

CC Cascade correlation is a neural network model that dynamically adjusts the number of parameters, e.g., hidden layer neurons and their associated connections.

DOAG Directed ordered acyclic graph.

Leaf node is a node in a graph which has no outgoing links. This is sometimes called a frontier node.

LRAAM Labelling Recursive Auto-Associative Memory.

LVQ Learning vector quantization.

MLP Multilayer Perceptron is a neural network model based on artificial neurons that are arranged in layers.

MSE Mean Squared Error.

NAPE An N-attachment point entity is a terminal or non-terminal symbol in a plex grammar.

Plex Is a type of grammar where each terminal or non-terminal symbol can feature attachment points which can be used to interconnect those symbols.

QSAR Quantitative Structure-Activity Relationship models attempt to predict the activity of chemical compounds by looking at the molecule structure.

QSPR Quantitative Structure-Property Relationship models attempt to predict the property of chemical compounds by looking at the molecule structure.

RCC Recurrent Cascade correlation is similar to CC but aims at the encoding of structured information.

RMLP Recursive multilayer perceptron is similar to MLP but aims at the encoding of structured information.

Root node is a node in a graph which has no incoming links.

RPROP Resilient propagation, an updating method for artificial neural network models.

SOM Self Organizing Map, a neural network model where neurons are arranged on an n-dimensional grid, with $n = 2$ most commonly. This is used often used for the projection of high dimensional data to one with lower dimensions, with grid points being represented by neurons.

SOM-SD Self Organizing Map for Structured Data. Similar to SOM but for the encoding of structured data.

sSOM-SD Supervised Self Organizing Map for Structured Data. This is a SOM-SD which includes supervision to the learning process.

SSE Summed Squared Error.

Tree A tree is a particular type of acyclic connected graphs where each node has at most one parent.

VQ Vector quantization.

Notation

The following uniform notation is used throughout this thesis. Scalars and constants are indicated by lowercase script letters e.g., c. Parameters for dynamic processes are stated as lowercase Greek letters such as α. Vectors are denoted by lowercase bold letters, e.g., $\mathbf{v}$. Sets and matrices are denoted by upper case letters, e.g., S. Sometimes, in order to avoid confusion, we use uppercase bold letters e.g., $\mathbf{M}$ to denote matrices. Calligraphic letters e.g., $\mathcal{G}$ are used for representing graphs. Domains are indicated by bold calligraphic letters e.g., $\boldsymbol{\mathcal{I}}$. Lowercase script letters are used to access elements of a vector or matrix. As an example, in order to access the i-th element of a vector $\mathbf{v}$ we use $\mathbf{v}_i$. Letters when used in combination with brackets such as in $f(x, y)$ denote functions. A few examples are given below:

$n = \|\mathbf{x}\|$	n is the dimension of vector $\mathbf{x}$
$\mathbf{x} = (x_1, \ldots, x_n)$	Vector $\mathbf{x}$ consisting of n elements.
$F(\mathbf{x})$	A function taking a vector as argument.
$\mathbf{C} = \mathbf{AI}$	$\mathbf{C}$ is the result of a matrix multiplication.
$S = \{0,1,2\}$	A set with three elements.
$\mathbf{m}_i = \alpha\mathbf{m}_i$	Recursive update of the i-th element of vector $\mathbf{m}$
$\alpha(t)$	The parameter α depends on time t.

In the Appendix, uppercase letters are also used to specify nonterminal symbols of a grammar, lowercase letters refer to terminal symbols, and lowercase Greek letters refer to strings of mixed terminals or nonterminals.

Hardware Environment

This thesis will address results from an extended set of experiments on various neural models. Hardware that was available for these experiments is as follows:

Hardware Type	OP	Number CPU [2]	CPU Speed[3]	Year available 1	2	3
PC	Linux	1	0.2	√	√	
PC	Linux	2	0.4	√	√	√
PC	Linux	3	0.45		√	√
PC	Linux	1	0.85			√
PC	Linux	1	0.93			√
SGI	IRIX	7	1.3			√
SGI	IRIX	64	1.5			√
PC	Linux	5	2			√

Throughout the thesis, when referring to CPU times, we actually refer to relative times required by a Pentium type CPU of speed 1GHz. Note that times and the speed indices are approximate values. The actual speed of a system has many other dependencies such as the speed of storage devices such as hard-discs and memory, the load of the machine, the efficiency of the operating system, and to lesser extend, the speed of display devices, network speed, and many others.

[2]Total number of CPUs of same type and speed.

[3]Relative to a Intel Pentium CPU with 1GHz

CERTIFICATION

I, Markus Hagenbuchner, declare that this thesis, submitted in partial fulfillment of the requirements for the award of Doctor of Philosophy, in the Faculty of Informatics, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

Markus Hagenbuchner

July 21, 2002

Acknowledgments

I first wish to thank Prof. Ah Chung Tsoi for being my supervisor and guide during the last 3.5 years of studies. Ah Chung devoted countless hours for discussions about this research topic and experimental procedures despite of his extremely high workload. Whenever I encountered obstacles, Ah Chung was there and cleared the way. Without Ah Chung's support, both morally and financially, the studies would have hardly been possible. I am deeply impressed by Ah Chung's endurance and friendliness he has directed towards me during the past few years. And I am looking forward working with Ah Chung in the future.

But there were many others whose support and encouragement brought me through this time. By following the order of appearance, not importance, I would like to thank Dr. Andreas Küchler for having put me on this track. Andreas had put me in contact with Ah Chung when he became aware that Ah Chung is making available funds for a Ph.D. project connected with a very interesting research topic. Andreas has also given me substantial support during the early weeks and months of my studies despite of being very busy with the writing of his own thesis at that time. Also early during my studies was Prof. Paolo Frasconi whose continuous efforts for the first three months gave me the right introduction into this research field, and a momentum that brought me right to the end of this project. I also had the pleasure of working with Prof. Marco Gori for three months towards the end of my first year. Those three months where probably the most challenging of them all. Much work was done and many ideas that ended up in this thesis were born during that time. Much appreciated was the support of A. Prof. Marco Maggini during the same time who patiently answered my questions and helped iron out mistakes I did during that time. The work with Marco Gori and Marco Maggini had given me yet another boost for the studies, and had resulted in a number of publications. A special acknowledgment goes to Prof. Alessandro Sperduti with whom I was given the privilege of working with at the end of the second year. Alessandro's ideas and constant encouragement which is lasting to this day have added substantially to the quality of this thesis, and also produced publications as a result.

I do not forget the fellow research students Michelangelo Diligenti, Alessio Micheli, and Diego Sona who, like me, enjoyed the research in this area. We have spent many hours discussing the materials, shared experience and knowledge, and enjoyed a glass of red wine when the pressure was greatest.

And of course, despite of having been on the receiving end of so much support from so many people, life would be empty without the support of a partner. My darling Mandy Liu has been with me before and during the last 3.5 years and will remain in my life for ever. Mandy's support and understanding contributed substantially to the success of these studies. Though sometimes I have difficulties understanding why Mandy remains on the side of a selfish person like me for such a long time.

My apologies to all those not mentioned by name in this section. Many have played an essential role in my life as a student during the last few years. I am grateful for having been accepted by so many as a fellow researcher and friend. I will never forget this!

Contents

List of Figures

Chapter 1

Introduction and Motivation

1.1 Introduction

This thesis is in the area of the application of artificial neural network techniques to processing of data structures. In this chapter, we will give a number of motivating examples indicating why a data structure is a convenient representation of a large number of artificial and practical examples. In chapter 2, we will give more detailed account of how to represent some of these practical examples using data structures.

This chapter gives a number of examples, the aim of these is to motivate and demonstrate that it is more economical to use structural information where such information exists to represent a problem. It will be demonstrated that there are many challenging real-world problems in fields as diverse as image processing, molecular chemistry, document and logo analysis, characterization of relationships among a number of web pages in the world wide web (WWW), handwritten character recognition, software engineering, and numerous others. In these areas, we find that the representation of data is more appropriately modeled by data structures [1] [86] , e.g., lists, trees, graphs [37] than atomic entities such as vectors as employed in more traditional data representation methods.

In the following sections, we will give a description of some of these artificial and practical examples. In Section 1.2, we will give an example of how an image can be represented using a tree structure; in Section 1.3, we will illustrate how a document, e.g., an envelope can be represented using a tree structure; in Section 1.4, we will illustrate how the surfing behaviour of web pages can be more conveniently considered using a data structure; in Section 1.5, we will discuss a class of molecular chemistry modeling problems which had been posed and studied in terms of structured data representations; in Section 1.6, we will describe a simple example in representing a program using structured information; in Section 1.7 we will illustrate how data structure can be applied to online handwritten character discriminations; in Section 1.8, we will show that a simple robot navigation problem can be reformulated in terms of a tree structure.

[1] In this thesis, we will use the terms "structured data", "structured domains", "structured information", "data structures" synonymously.

1.2 Image Processing

In image processing, a central issue is how to understand a particular given scene [53, 78]. Representing an image by its pixels requires significant memory storage. For example, if the image consists of $1,000 \times 1,000$ pixels, then the memory required to store it is 1 Mbits, assuming that it is a black and white image, i.e., it has 2 levels of gray only. The amount of memory will increase significantly if the image is a colour one. Representing images using pixels does not take into account any possible relationships among the objects in the image, assuming that the image contains a number of objects. On the other hand, if we pre-process the image so as to extract some primitives (objects), the image could be represented using a much smaller amount of memory. This fact is self evident in that if an image is represented using pixels, no relationship is assumed among any pixels or groups of pixels within the image. On the other hand, if a number of pixels is grouped together to form a primitive (in the form of an object), then we can reduce memory storage as we can depict the relationships among these objects [2]. In addition, the relationships among the objects become more transparent. We will illustrate this idea using the scene as shown in Figure 1.1.

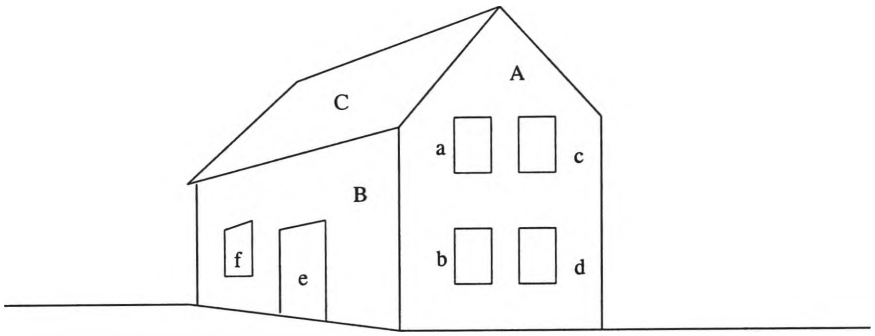


Figure 1.1: A scene showing some details of a house.

This scene has been pre-processed using, say, an edge detector [53], so that most of the non–essential details, e.g., shading, textures, have been eliminated. Each extracted part is labelled. Such a scene can be represented by a tree structure as shown in Figure 1.2. In this tree, it is noted that there is a root node, which we labelled "Scene". This is the starting point of the analysis. This node has a number of leaves, corresponding to the scene. For example, we have a leaf labelled "C" representing the roof; a leaf labelled "A" representing one side of the house, facing the reader; and another leaf labelled "B" representing another side of the house. Each of these sides has their own details. For example, side "A" consists of four windows, labelled "a", "b", "c", and "d". These are represented as children of the node labelled as "A". Node "A" is said to be the parent of the children nodes "a", "b", "c", and "d". Thus representing the pre-processed image using a tree model is more economical on the data storage as we only need to store the objects "Scene", "A", "B", "C", "a", "b", "c", "d", etc, and their relationship as expressed

[2]This statement begs the question: how can we "recognise" an object in an image in the first instance. Here we assume that the set of object recognition software has been applied to the image to recognise the embedded objects first, and that we also have some software to work out how these objects are related to one another. In doing so, we have placed some a priori assumptions on the image, e.g., the existence of objects, the type of objects that we expect, etc. An example of such object extraction algorithm as well as the underlying assumptions is shown in Appendix A.

in the tree. Secondly, it also clearly shows the relationship of individual objects with one another. For example, it shows that the side "B" consists of two objects, viz., "e" and "f". Furthermore, such a tree representation can be noise tolerant. If, at some stage, the pre-processor had failed to extract certain features (such as a window or door) due to the presence of noise in the image, then the extracted tree would still maintain great similarities to a noiseless representation of the image, as only some of the leaves are missing, while the overall tree structure of the image would still be intact.

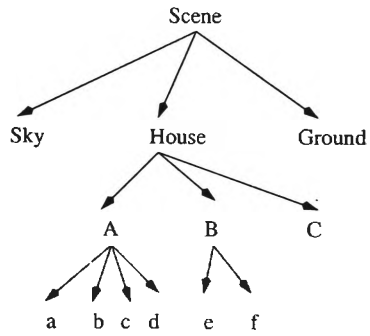


Figure 1.2: Representation of the house scene using a tree

One can imagine that there are many situations when a scene is more appropriately represented using a tree model, after suitable pre-processing. A question that is often asked is: given a number of scenes depicting different sceneries, can we recognise the differences among the scenes. If we can represent these scenes using tree models, an equivalent question to ask is: can we distinguish one class of trees from another. If we are given a number of trees, can we separate them into categories. Answers to such questions would be useful in scene analysis.

1.3 Document Processing

Document processing has emerged as one of the major practical problems related to office automation. It is claimed that everyday in a large enterprise, many documents are created. For example, in a large insurance company, everyday, there will be many forms filled in. These forms contain information which would be useful to the insurance company. In addition, there are many internal documents being created concerning these forms, e.g., accounts paid, transaction details, claims details, payment details. Depending on the company, these documents can be stored in paper form, or at least some of these documents will be stored in electronic form. Daily, these forms will accumulate. If they are in paper form, it would be quite time consuming to retrieve the information contained on a particular form. If they are stored electronically, again, often it is quite difficult to search for and retrieve a particular document with prescribed contents, especially if the description of the contents is "fuzzy", i.e., the user is not too certain of the exact content of the document. In other words, it is quite difficult to retrieve the exact document if the content of the documents cannot be categorised easily (in database terminology, to index the contents of the documents), as this would render it difficult to use modern database techniques to retrieve the particular document containing the information.

For example, every country has tax returns by citizens or businesses. A typical US Federal tax return form for single or joint partners with no dependents is shown in Figure 1.3 (left). This form is different from one which is used by employers filling

in their quarterly returns as shown in Figure 1.3 (right). The US Tax Department has an automated process which scans in the information contained on these forms directly to an automated tax processing machine. Once the information is scanned in, it can be processed using software, which in simple cases, resulted in automatic assessment of tax returns [3]. In complicated cases, or cases where the software cannot resolve the content, the tax return form would be routed to a human assessor. It is claimed that using such automated processes, many thousands of tax returns can be processed per hour.

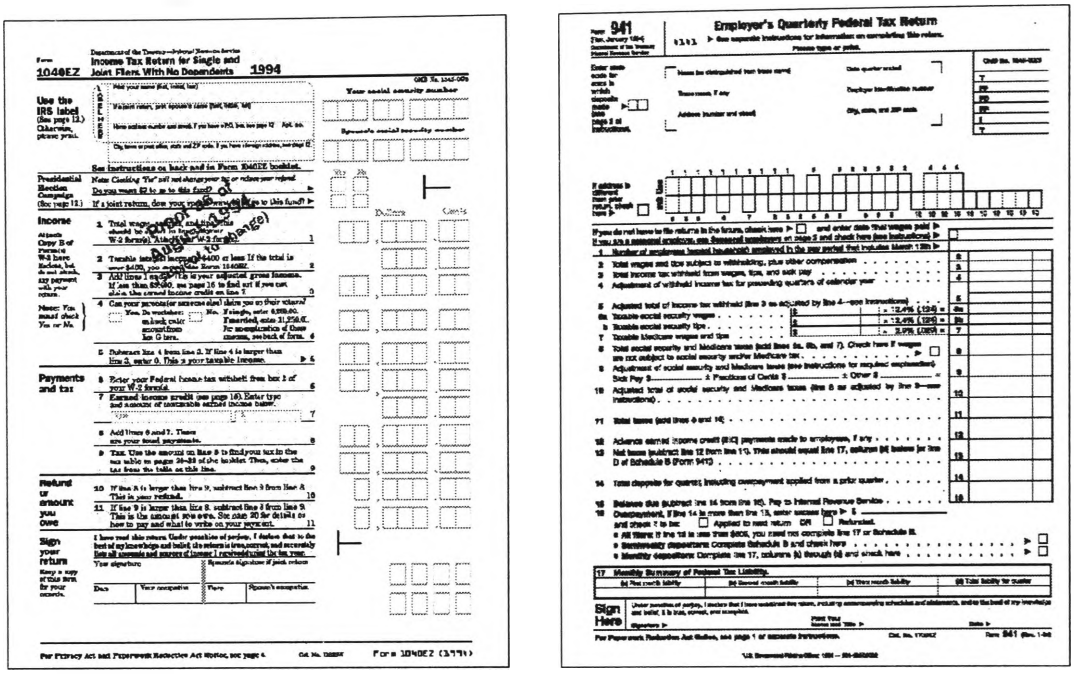


Figure 1.3: A typical US Federal Tax return form for single and joint partners without any dependents (left), and a typical US Federal Tax return form for business (right)

Document processing is a significant part of this automatic tax assessment process [88, 89]. As a preliminary step, it would be useful if we can automatically distinguish these two types of forms into separate piles. It is noted that the two forms are distinct. The scanned in image can be pre–processed to extract areas which contain information. These areas together with the details can be represented by a tree model. Once they are represented as tree models, then the question is: can they be distinguished from one another. In other words, can we recognise the differences between the two tree models. This question, for example, has been addressed in [3].

As another example of document processing, there have been much recent interest in machine recognition of mail envelopes in automatic mail sorting applications [87, 96]. One of the issues involved in automatically recognising mail envelopes lies in the ability to recognise the form of the envelope and to locate where the address block is. Figure 1.4 shows a typical example of an envelope. It is noted that it consists of a number of separate blocks, e.g., the logo block, the address block, the

[3]Note that one of the main reasons why such forms can be automatically processed lies in the careful design of the form, whereby there are redundant information coded into the questions that a person needs to provide in order to overcome some of the ambiguities caused by the inherent inaccuracies of the scanning process.

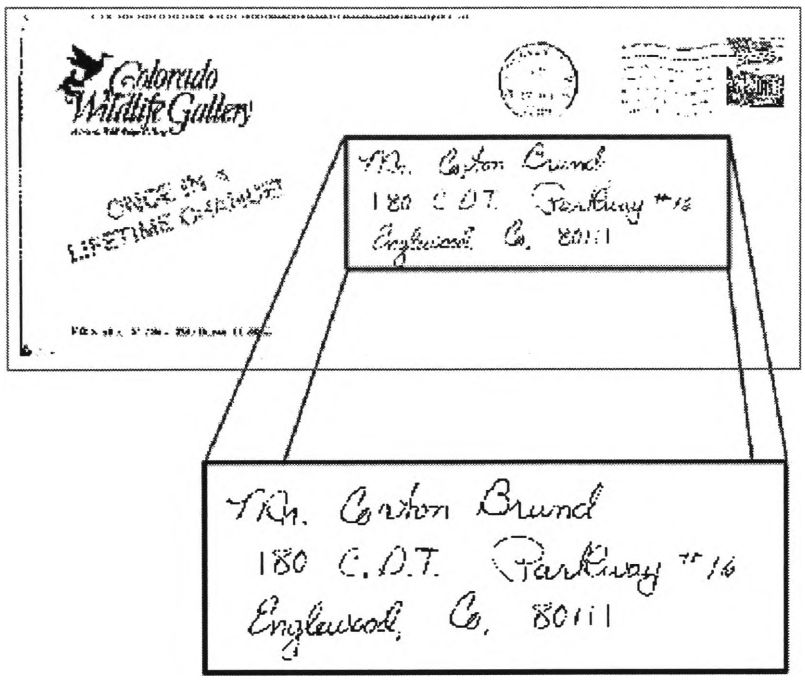


Figure 1.4: A typical handwritten mail envelope

stamp block, etc. This can be represented as a tree as shown in Figure 1.5. The tree model makes it transparent on how the various blocks are related to one another.

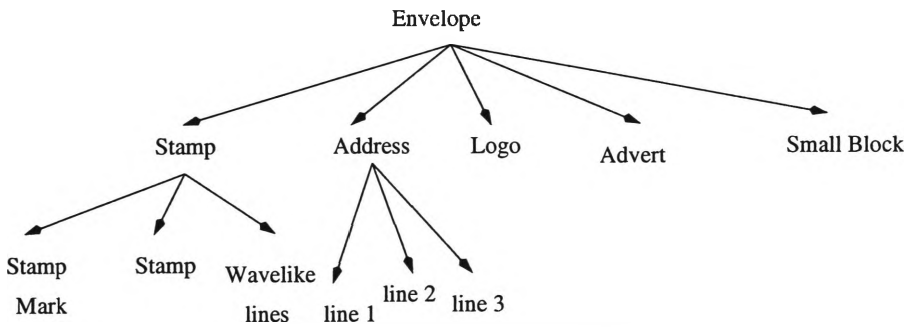


Figure 1.5: A tree representation of the mail envelope shown in figure 1.4

Again, one of the questions which one would wish to ask is: given many envelopes, can we "categorise" them into different categories. A tree representation of the envelops would be very useful in providing the first step in this recognition problem.

A sub–problem of the document/envelope recognition problem involves the problem of logo recognition [19, 28]. The issue is: given a large number of company logos, is it possible to recognise them individually. It is trivial to observe that each logo can be represented in terms of a tree representation, with each part giving the relationships among the components of the logo. Each company would have a different logo. Hence, each tree representing the logo is different.

1.4 Internet Behaviour

The previously discussed examples are all static, in the sense that the problems do not change with time. For example, the logo recognition problem: over the life span of a project, it is unlikely that a company would change its logo. Similarly, the address recognition problem in the document processing application would not change once it is written. On the hand, there are problems which are inherently time varying, i.e., the system behaviour varies with time. One such problem is the behaviour of users on the internet. When a person is searching for information on the internet, often the person would follow link after link in search of relevant information. If the person searches for the information today, there is no guarantee that the person will follow the same path if the person searches for the same information some time later. This is partly because the information contained in web pages is known to be changing. Indeed, it is claimed that on an average, the information on web pages changes once every 30 days. On the other hand, there are often multiple references to the same information contained from different pages. Hence a user can access the same information using different routes. A typical user behaviour is shown in Figure 1.6. The user first accesses document 1. Then the user pursues a link to page 2 which subsequently leads to page 3 or page 4, depending on what the user is doing. Pages 3 and 4 respectively have their own linkages. Also it is possible for the user to jump from page 1 to page 7 as indicated in the figure.

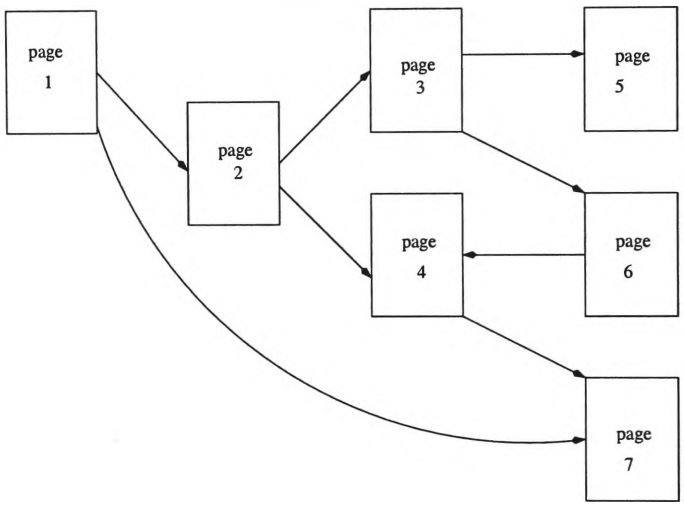


Figure 1.6: A diagram illustrating the behaviour of an internet user

This figure illustrates a time varying behaviour of the user. At one instant, the user could go through to the information contained on page 7 via pages 2, and 4. On the other hand, at other time instants, the user could go directly to page 7 from page 1. Thus, the behaviour of the user depends on what the user is looking for at a particular instant.

It is simple to observe that tree representation is ideally suited to represent the information that describes such user behaviour in this situation.

1.5 Molecular Chemistry

A central issue in modern drug design is how to correlate the biological activities to physical and chemical properties of biologically active compounds. There are

already well developed models for this [47, 103], for example, development of models which can exploit a wide variety of molecular properties, including structural descriptors such as topological indices. The importance of Quantitative Structure Activity Relationship (QSAR) analysis is that it enables the design of new drugs on the basis of known structure-activity relationships supplied by the QSAR analysis. Quantitative Structure-Property Relationship (QSPR) can be considered as a generalization of QSAR concept. It assumes that general properties, e.g., physical properties of the compounds can be related to their chemical and morphological structures. An example of a structure for a chemical compound is as shown in Figure 1.7.

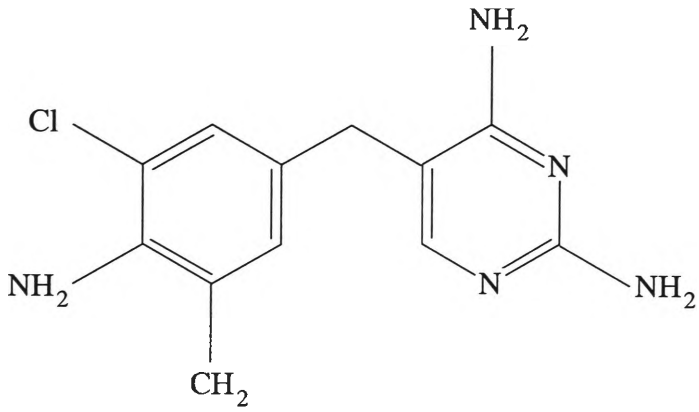


Figure 1.7: A diagram showing a chemical compound

Neural network methods, e.g., multilayer perceptrons have been applied to QSAR and QSPR analyses [17, 31]. However, they require task specific representations, e.g., physico–chemical parameters, topological indices, and vertical graph codes, as features. On the other hand, there are inherent structural contents in the QSAR and QSPR. Hence, it would be useful if these structural properties can be used specifically in the encoding of the neural network. In other words, it would be useful if we can devise a neural network architecture which explicitly takes into account the chemical structural properties [9, 10]. A method has been devised to convert the chemical structure into a tree structure, thus permitting analysis of the QSAR or QSPR using neural networks [8]. This method is rather involved, relying on some specific properties of chemical structures to avoid the possibility of having to deal with cyclic graphs explicitly. Hence, we will not include it here but refer the interested reader to the reference [8] for further details.

1.6 Software Engineering

A major issue in software engineering is the quality of the software. One way in which software quality can be measured is by using what is commonly called a software metric [62, 68]. A software metric attempts to measure the complexity of a given software package or program. Thus, if the metric indicates that the software is complex, then implicitly the software concerned would be more prone to error, and difficult to maintain. On the other hand, if the metric shows that the software concerned is less complex, then implicitly it is easier to maintain, and less error prone.

One of the earliest software metrics devised is called the McCabe metric [62] which

attempts to measure the complexity of the software concerned. This method works by counting the number of juncture points, and decision points. To make the method less dependent on a particular programming language, it is common to transform the program into an intermediate representation. A suitable intermediate representation of programming languages is given by dependency graphs. In dependency graphs, statements are represented by nodes while directed edges are used to represent the statement ordering implied by the dependencies in a source program. There are various different kinds of dependency graphs, e.g., control dependency graphs, data dependency graphs, control flow graphs, and instance dependency graphs [68].

It is a common belief that most procedural languages can be represented as a flow graph consisting of a number of basic elements, e.g., decision node, junction node, "begin", and "end" node. Hence, once a source program is transformed into a flow graph, it is relatively simple to compute the McCabe metric. The flow graph itself can be represented in a graph format showing the skeleton structure of the program source code. It is trivial to see that the flow graph can be represented in terms of a graph. In this case, there are cyclic paths in the graph (representing "do loops" and "if .. then .. else" structures). An example of a flow graph is displayed in Figure 1.8.

```
program name(list)
var
...
begin
if T1 then
        a
else
        b;
c;
while T2 do
        begin
        d;
        if T3 then
        while T4 do
                e
        end;
f
end
```

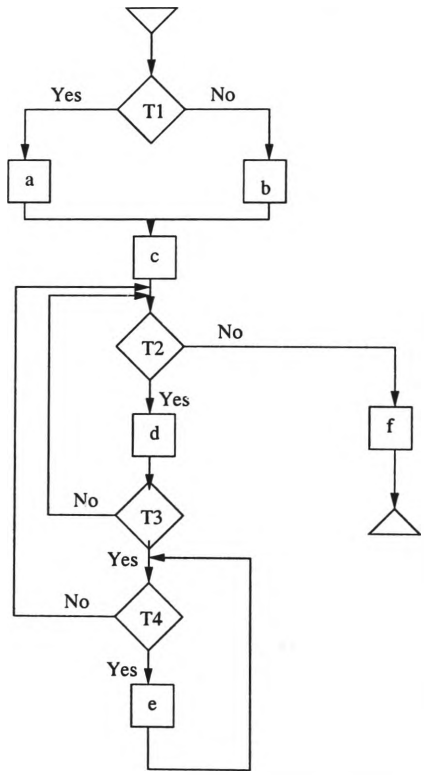


Figure 1.8: A flow graph and the corresponding program code of a small portion of a software package

Software engineering includes the reuse of software components as another important problem. There, the task is to find parts of existing software which can be useful for the implementation of new software products. This problem can be solved by developing a role model for a given task [69]. For example, the role model may express how objects of a software product interact. Hence, a role model can be expressed in the form of a graph. A more comprehensive example of this application is given in Section 2.6.

1.7 Handwritten Character Recognition

Handwritten Chinese character recognition is an important problem which could lead to significant advances in information processing of non–alphabetic languages [16], e.g., Japanese, Korean. The written language uses an ideographic concept, i.e, each character represents a pictorial concept which was evolved from earlier pictorial representation of the concepts conveyed by the character. Hence, there is significant structural information contained in the character.

There are two types of character recognition, viz., off–line and online. In the off–line character recognition case, the character is presented as it is, while in the online situation, information concerning the formation of the character is also available.

There have been much research in handwritten Chinese character recognition using syntactical and semantic approaches [16]. These approaches intend to capture the information contained in the handwritten character, and to determine what the character is.

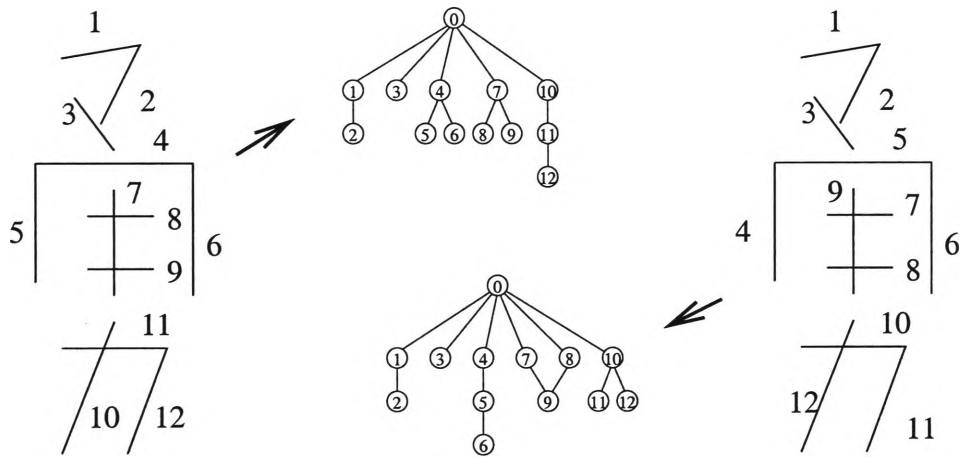


Figure 1.9: A Chinese character showing two alternative sequences of strokes and a possible graphical representation.

The online handwritten character recognition has the added information concerning the formation of the character [101]. This may assist in the recognition of the character, as it gives more information. On the other hand, because there are various ways in which the same character can be formed, thus, this may introduce some confusion. For example, Figure 1.9 shows two different ways of writing the same character. Both are equally valid. It depends on how the person learned to write that particular character. It would be important to ensure that no matter how the person writes the character that if it is the same character being written then it should lead to the same conclusion, i.e., the same character is being recognised. This is especially important since there are people who learned how to write Chinese as an adult from books, e.g., learning Chinese as a second language. Hence the sequence of strokes could be quite different from native Chinese who learned how to write characters using a particular sequence of strokes.

It is possible to represent the different ways of writing Chinese characters using graph structures. For example, in Figure 1.9, the sequence of strokes involved in writing the character on the left hand side can be represented as a graph involving nodes and links among the nodes representing the sequence of strokes. Nodes are connected if the strokes are connected, strokes drawn first build the parent of

connected strokes. In a similar manner, the strokes involved in writing the character on the right hand side of Figure 1.9 can also be represented in terms of a graph. This example demonstrates that there are cases where the extracted graph is disconnected (i.e. a graph without a common supersource). This problem can be overcome simply by introducing a new node and connecting it to the root nodes of all graphs extracted from a single pattern. In the given example, node numbered 0 builds the new root node which connects the extracted structures.
It is quite obvious that the two graphs are different. However, these two graphs represent the same character. Hence, we can put them into the same category. It is possible to imagine that we can go through all the common Chinese characters, enumerate possible ways in which each character can be formed, and represent them each in graphs. A question to ask is: can we distinguish the various ways in which a character is written from other characters. Such question can be formulated in terms of whether it is possible to recognise graphs which are of different structures, but nevertheless belong to the same category from a set of given graphs which might contain other categories.

1.8 Robot Navigation

There have been much interest in autonomous robot navigation. A particular interesting problem is to assume that the robot has no inbuilt plan of its environments. The problem is: if the robot is navigating in a Manhattan environment with similar landmarks, how could the robot determine if it has visited the same location before. This is illustrated in Figure 1.10. The robot is assumed to be able to move left, right, or forward. The robot needs to learn about the environment as it proceeds. A tree representing the robot movement at some of the junctions is illustrated in Figure 1.11.

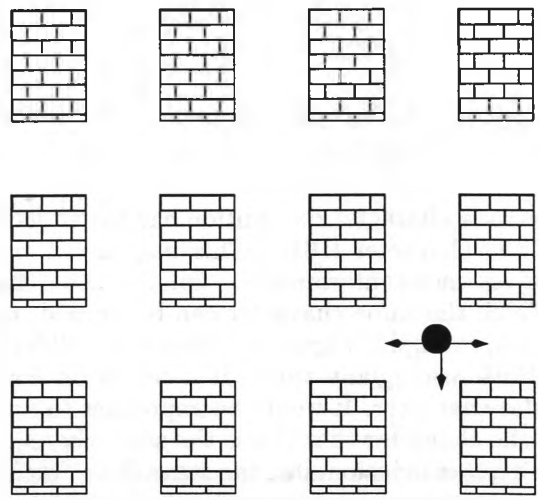


Figure 1.10: A robot navigating in a Manhattan environment with no explicit landmarks

The problem of whether the robot can recognise if it has visited the same place before can be formulated in terms of the recognition of the pattern in the graph representation. Hence, the task would be to recognise cycles in a graph since this these are produced by a robot having visited the same place. However, the general handling of cyclic graphs is known to be difficult. This thesis does not attempt to

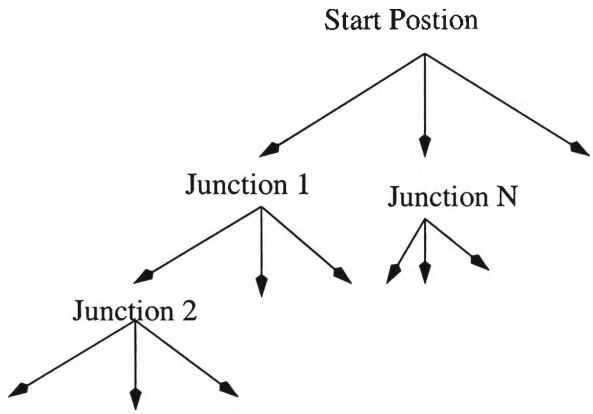


Figure 1.11: A possible tree representation of the robot navigating in a Manhattan environment with no explicit landmarks

address models capable of handling cyclic graphs [4].

1.9 Conclusions

In this chapter, we have considered a number of motivating examples of why graph representation of problems would be advantageous. For example, it is illustrated that using a graph structure, an image might be more economically represented than a pixel map representation. It is illustrated that graph structure underlies some problems associated with document processing, e.g., mail envelope routing, logo recognition. It is suggested that graph structure may be one way of modelling the behaviour of a user surfing the internet. It is indicated that graph structure might be one way in which QPAR and QPSR of chemical activities can be represented. It is illustrated that a flow graph in evaluation of the McCabe metric in the determination of the complexity of software packages may be a candidate for graph representation. In handwritten recognition of online Chinese character recognition, it is illustrated that a graph structure might be a simple way to represent the various ways of writing a particular Chinese character thus facilitating the processing of information. In robot navigation under a Manhattan environment, it is illustrated that a graph structure may be a convenient way to facilitate the recognition of whether the robot has visited the same place before.

In this chapter we have refrained from giving detailed information on exactly how a graph is obtained from a given problem. Our main aim here is to show the plenthora of artificial and practical examples in which graph structure may be a convenient way to represent the underlying information. This is best done without the cluttering of details of how to obtain a graph structure from a given problem. In the next Chapter, we lay the theoretical foundation to represent such problems using data structures, and will give details on how to obtain a graph structure from a given problem, illustrated by a number of problems.

[4]The models described in this thesis are capable of handling certain types of cyclic graphs. However, we will not focus on this property.

Chapter 2

Data structures

2.1 Introduction

In Chapter 1, we have illustrated the idea of graph representation of the information underlying problems with a number of examples from a large variety of artificial and practical problems. We did not give any detailed information on how this can be done. It is found that the methods involved in obtaining a graph representation differ with the type of data in consideration. Hence, there is no uniform graph extraction mechanism that can be applied to various type of data. In this Chapter, we will give a number of suggestions on how to obtain graph structures from a given problem.

The structure of this chapter is as follows: a review of some relevant concepts in data structures is given in Section 2.2. Section 2.3 illustrates the extraction of a graph structure using an image; and Section 2.4, illustrates the concept using document processing. In Section 2.5, more details are given on how to obtain a graph structure from a molecule while avoiding the issue of having to deal with cyclic graphs; and finally, Section 2.6 presents a concept using a software engineering example.

2.2 Fundamental concepts of Data Structures

In this section, we will give some relevant concepts in data structures. A distinguishing feature in data structures as compared to the common non–structured data representation is the possibility of representing variable data length. In multilayer perceptron applications, it is necessary that we nominate to use a set of fixed length data structure, even though the data structure may be variable. Because of this restriction, often we need to augment the variable length data structure by padding with zeros. Dependent on the problem, this may result in an oversized network (as it requires a larger network to represent the additional zeros in the fixed length data format), and possibly longer training time. On the other hand, using data structures to represent variable length data, this artifact is not necessary [29].

Data structures are commonly referred to as diagrams or *graphs*. A graph consists of a set of vertices or *nodes* which are connected by an *edge* to indicate a relationship between the nodes. Formally, a graph $\mathcal{G} = (V, E)$ is a finite nonempty set of vertices V together with a set of edges E. Elements in E consist of two element subsets indicating which element in V is connected with which other element in V. For

example, the tuple $(v, w) \in E$ indicates that node $v \in V$ is connected with node $w \in V$. For convenience, the set of vertices V from a graph $\mathcal{G}$ is denoted as $vert(\mathcal{G})$, and the set of edges E belonging to graph $\mathcal{G}$ is denoted as $egd(\mathcal{G})$.

An edge is considered to be directed if a tuple $(v, w) \in E$ is seen as an ordered pair rather than a set. In this case, a natural direction from the first vertex of the pair to the second can be associated with the edge. Every undirected graph can be represented as a directed graph by adding to every tuple $(v, w) \in E$ the reverse link $(w, v) \in E$ [37]. In this thesis, we consider directed graphs only. Hence, when referring to graphs, we implicitly mean directed graphs.

An example of a simple directed graph with four nodes $V = \{a, b, c, d\}$, and six edges $V = \{(a, b), (a, c), (b, c), (b, d), (c, d)\}$ is as shown in Figure 2.1. Here, a graph is graphically represented by using circles to represent nodes and arrows for the directed edges.

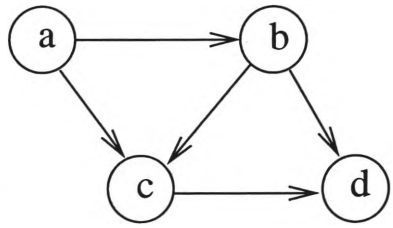


Figure 2.1: An example of a simple directed graph. Each node in the graph is identified by a unique symbol.

An incoming edge to a node 'a' is called an *inlink* of node 'a' and the total number of inlinks to a node is referred to as the *indegree* of this node. Similarly, outgoing edges are called *outlink* and the total number of outlinks of a node is called *outdegree*. The maximum number of outlinks at any node of a graph or a set of graphs is called the *maximum out-degree*.

A graph is called *rooted* if there is at least one node with zero inlinks, i.e., a node without any incoming links. Nodes with this property are called *roots* or *super-sources*. Nodes with zero outlinks are referred to as *terminal* nodes or *leaf* or *frontier* nodes. Nodes that have both inlinks and outlinks are called *intermediate* nodes. An intermediate node is said to be located at level l if the shortest sequence of edges leaving from this node to a terminal node of the same graph is l [37]. Thus, in Figure 2.1 the node a is the root of the graph, node d is the terminal node of this graph. All other nodes are intermediate nodes, where both nodes b and c are located at level 1. The maximum out-degree of this graph is 2. The maximum in-degree of this graph is also 2.

The source of an inlink is called the *parent* of a node $v \in V$ and is sometimes denoted as $pa[v]$. The destination of an outlink of a node $v \in V$ is called *child* or *offspring* and is denoted as $ch[v]$. The k-th child of v is denoted as $\mathrm{ch}_k[v]$. Using the graph depicted in Figure 2.1 as an example, node a is the parent of nodes b and c. Nodes b and c are respectively parents of node d. In addition, node b is also a parent of node c. As a consequence, node d is a child of both nodes b and c, whereas b and c are both the children of node a. Node c is also a child of node b.

A graph is said to contain a *cycle* if there is a sequence of edges leading from one node in the graph back to the same node. Graphs not containing any cycles are called *acyclic*. A collection of cyclic graphs is given in Figure 2.2.

An *ordered* graph is a graph $\mathcal{G}$ with a vertex set $\mathrm{vert}(\mathcal{G})$ and an edge set $\mathrm{egd}(\mathcal{G})$,

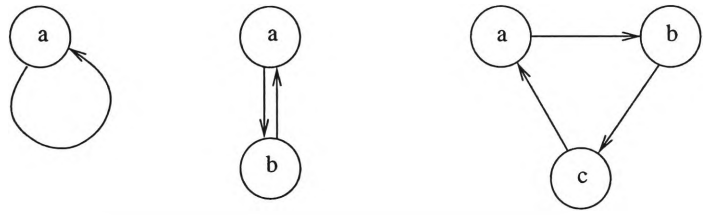


Figure 2.2: An collection of simple cyclic graphs. Note that none of the nodes in these graphs is either a root or a terminal node.

where for each $v \in \text{vert}(\mathcal{G})$, a total order on the edges leaving from v is defined. A superclass to ordered graphs is the set of positional graphs. A *positional* graph is a graph $\mathcal{G}$ with a vertex set $\text{vert}(\mathcal{G})$ and an edge set $\text{egd}(\mathcal{G})$, where for each vertex v a bijection $P : \text{egd}(\mathcal{G}) \rightarrow \mathbb{N}$ is defined on the edges leaving from v. Nodes that have an associated label (symbolic or numeric) are said to be *labelled nodes.* Graphs consisting of labelled nodes are referred to as *labelled graphs.* An example of a labelled graph is shown in Figure 2.3.

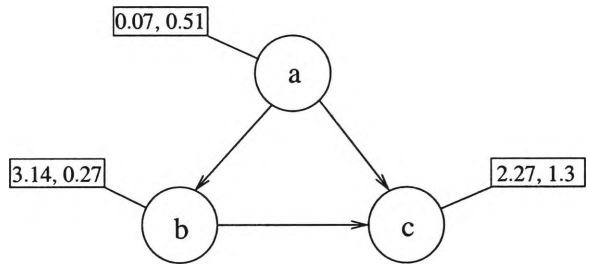


Figure 2.3: An example of a simple directed acyclic labelled graph. Labels are real valued vectors. The dimension of the data label is the same for all nodes.

A *subgraph* of a graph $\mathcal{G}$ is any graph $\mathcal{H}$ such that $V(\mathcal{H}) \subseteq V(\mathcal{G})$ and $E(\mathcal{H}) \subseteq E(\mathcal{G})$, and it is said that $\mathcal{G}$ contains $\mathcal{H}$.

In this thesis, we consider the use of labelled directed ordered acyclic graphs (DOAGs) which does not necessarily impose a restriction. Every undirected graph can be represented by a directed graph. In addition, the consideration of ordered graphs allows us to distinguish graphs such as those shown in Figure 2.4.

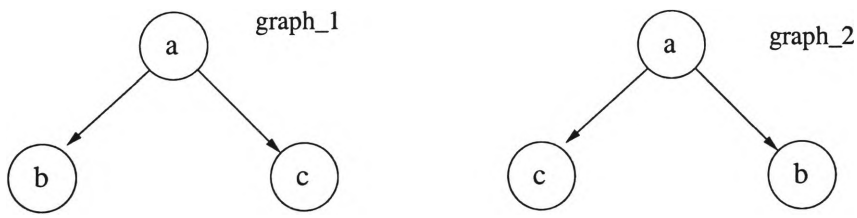


Figure 2.4: Two simple graphs which can be distinguished when considering ordered graphs.

Furthermore, the class of labelled graphs contain the class of unlabelled graphs as a subset. The use of data labels is often useful. For example, labels can hold attributes associated with an object which is represented by a node in the graph. The use of acyclic graphs imposes a restriction to the domain of directed graphs.

This restriction is necessary since nodes interacting through a loop define a infinite recursive problem, a problem which is hard to solve. However, it is possible in many cases to overcome this restriction by replacing nodes involved in a loop by a single node. This, for example, is commonly done in the area of molecular chemistry.

A *tree* is a particular type of DAG where the graph is connected, i.e. there is exactly one root node for each graph. Contrary to a physical tree, a node can have more than one parent. Sometimes we refer to tree data structures in order to simplify the notation.

The following sections illustrate on a number of real world applications how a graph representation is obtained from raw data.

2.3 Application to Image processing

Consider an example from the area of image processing. The task is to find a mechanism that is able to efficiently represent a given image in the form of a graph structure. One way of achieving this is by pre-processing a given image using common filter mechanisms e.g., low-pass filters and high-pass filters. The high pass filter will retain the external shape of the object by accentuating the edges, while the low pass filter will smooth out the interiors of an object [53]. An example of such a mechanism is shown in Figure 2.5.

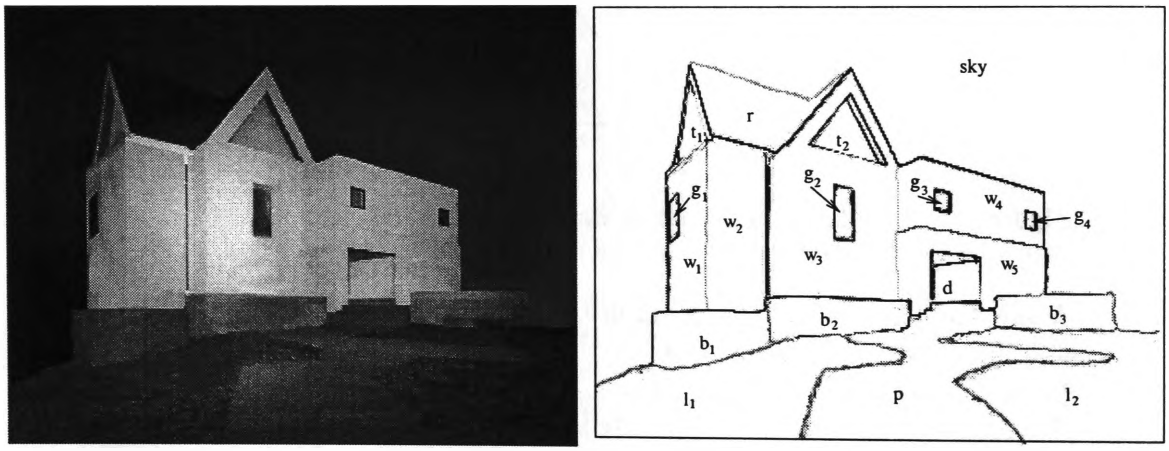


Figure 2.5: A photograph of a house (left), and the same image after pre-processing with low-pass and high-pass filters (right). Regions that are enclosed by edges are marked with unique symbols.

Here, the image of a house has been pre-processed using a low-pass filter for the removal of noise and small unwanted features (as these will be smoothed), and a high-pass filter for edge detection. The resulting image is then processed further to detect areas that are enclosed by edges. Each of the enclosed areas is marked by a unique symbol to allow the buildup of a graph representation. For example, Figure 2.6 displays a graph structure obtained from the image shown in Figure 2.5.

Each area in the image produces a node in the graph where relationships between the areas are used to build up the structure of the graph. This effectively represents structural relationships between elements in the original image (For example. g_2 is inside the area enclosed by w_3). A numeric [1] data label is attached to each node

[1]The labels are numeric in this particular case as they represent the features of the

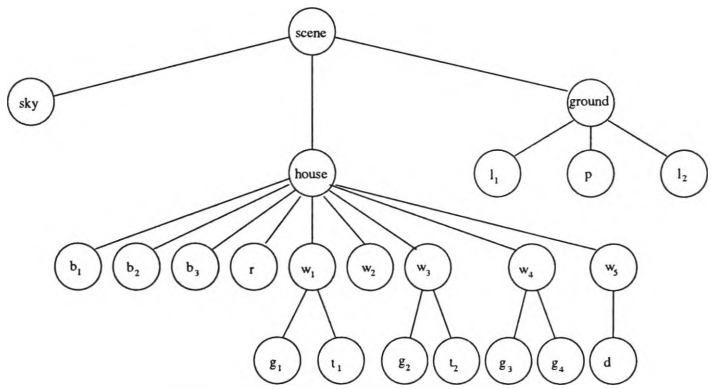


Figure 2.6: The graph representation of the house shown in Figure 2.5. Note that data labels which are associated with each node in the graph are not shown here.

in the graph holding information about the area represented by the node. This information can include shape, colour, size, location, and other details about this area. These are often referred to as features. Since every node exhibits the same number of features, the dimension of the data label is constant over all nodes in the graph.

This filter mechanism unfortunately has some drawbacks. First, it is computational expensive so that the processing of very large image database becomes quickly infeasible. Secondly, it can produce unreliable results on images with low contrast. In this case, edges in the image are very weak and it becomes difficult to reliably detect areas enclosed by edges. For example, parts of an edge may be missing due to a low contrast in the original image. Thirdly, graphs produced by this mechanism tend to be rather wide and flat. This effectively reduces the variety of graphs extracted from a large database of images, making the task of differentiating between graphs produced by different images a more difficult task.

An alternative approach based on the quadtree algorithm [53] has been applied to image processing tasks with more success [97]. Quadtree recursively decomposes an image into four equally sized rectangles as illustrated in Figure 2.7. The recursion continues until a stopping condition has been encountered. Common stopping conditions are based on reaching a certain depth in the subdivision process of the image, or an image segment featuring a small variety of colours, indicating that the segment does not contain significant features so that there is little variation in the colour of the segment.

There are a number of advantages associated with the application of the quadtree algorithm. First, it is simple to obtain a graph representation. It is convenient to elect the initial area (the entire image) as a root node for the graph. The offsprings are formed by attaching the areas formed by the decomposition of the original image into four segments. Selected segments are further sub-divided into four segments, as shown in Figure 2.7 and so on. Thus, the graph structure is built recursively with the application of the quadtree algorithm. The result is a simple type of graphs known as *trees*. The second advantage of using a quadtree representation is that the graph produced nodes that have either an out-degree of zero (for terminal nodes), or an out-degree of four (for all other nodes). This allows the implementation of

object, e.g., the colour, the size, the location, etc. In general, labels can be either numeric in value, or logical in value. Logical values are often used to indicate if a feature is present or not.

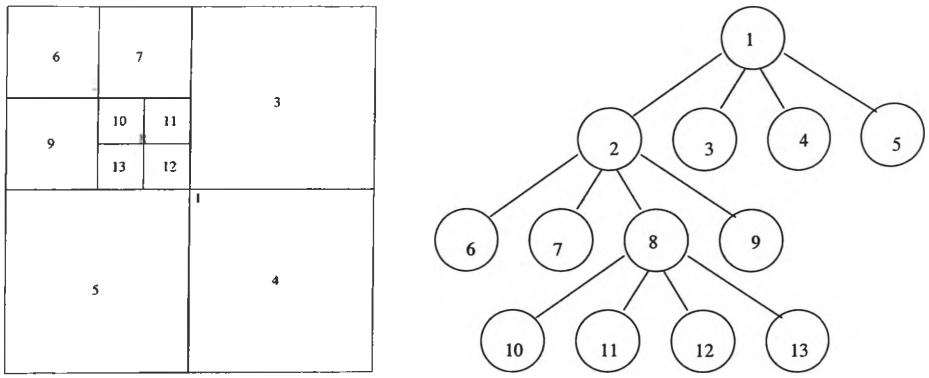


Figure 2.7: The quadtree algorithm. An image is recursively segmented into four equally sized parts (left). From this, a corresponding graph structure can be obtained (left).

"optimized" algorithms for the processing of such graphs taking advantages of these special properties.

Graphs produced through the application of the quadtree algorithm tend to feature a large number of nodes, because each time a segment is subdivided, four sub-segments are created, no matter whether these sub-segments will be useful or not. This may slow down an algorithm for processing graphs. However, having graphs with a relatively large number of nodes minimizes the risk of producing identical (or very similar) graphs from different images.

Again, each node in the graph receives a data label containing information about the area represented by this node. For example, in [97], each data label is a six-dimensional vector containing visual features consisting of four colour attributes, and two texture attributes. The colour attributes include the number of quantizised colours, and the percentage of the three most dominant colours in RGB (red, green and blue, the three primary colours) format. The average pixel value and the corresponding standard deviation are used to characterize the texture attributes.

There are many other techniques for representing images as graphs. Some of the better known ones are contour trees [104] and region adjacency graphs [18, 74]. This indicates that there is no graph extraction algorithm which is generally applicable to all types of images. Hence, the suitability of a graph extraction mechanism depends on the undelying task.

2.4 Document Processing

Logo recognition can be seen as a special case of image processing. Similarly, documents can be seen as images with particular properties. Since documents typically feature a geometric layout, the task of document processing is simpler than the processing of general images. A successful approach to the processing and classification of documents relies on a graph extraction method called *XY-tree* algorithm [65]. The idea of the XY tree algorithm is simple. A document is scanned in as a gray scaled image. The image content is then segmented into areas that could be separated through a straight horizontal or vertical line without crossing significant amount of text or graphic content. These segments are recursively segmented into smaller areas until no further line can be drawn without crossing a significant amount of text or graphic material. An example of applying this idea is shown in

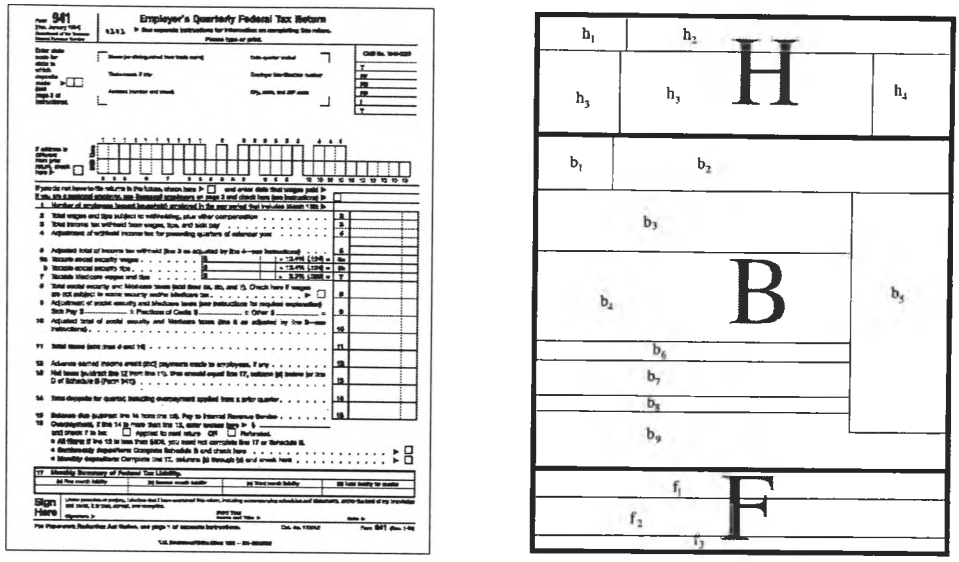


Figure 2.8: A document segmented into blocks of text, tables, and graphics using the XY-tree algorithm. The original document is shown on the left, the segmented result is shown on the right. The segmentation process has been stopped at the second level to avoid cluttering.

Figure 2.8. Here, the first step of the segmentation process produced three blocks identified by H, B, and F. These blocks are then divided into further blocks denoted by lower case identifiers. From this, a graph representation is obtained by representing the entire document as the root of the graph. The root's offsprings are formed by the result of the first iteration of the segmentation process. The graph structure is built up recursively in a similar manner to the quadtree algorithm presented in Section 2.3. Figure 2.9 depicts the graph representation of the document shown in Figure 2.8.

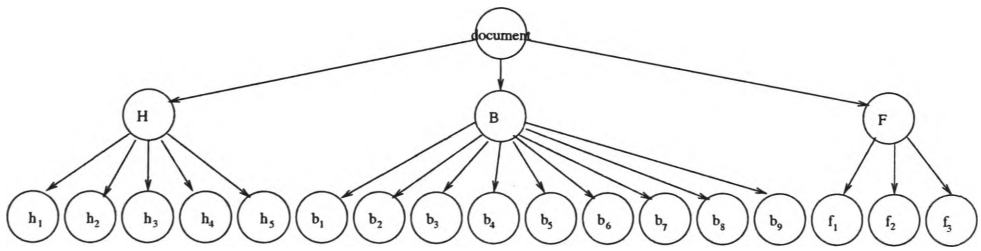


Figure 2.9: The graph representation of the document shown in Figure 2.8. Note that data labels are not shown here.

Dependent on the document types, the features used in the data labels can be quite different. For example, in the case which requires text recognition as part of the application, one can process the segments by first ignoring those which contain graphic materials. The segments which contain text materials can then be processed. If we assume that the words of the total vocabulary of the text be represented in a vector, then the occurrence of a word or not can be represented in this vector by using 0 or 1. If we assume that there is a "stop" list which contain words which should not be included in the vocabulary, e.g., the words "the", "a", "and" etc. Then, the words occurring in a text segment can be represented by 0 and 1 of a feature vector. Thus,

in this case, the data label could include this feature vector. In addition, it may include information on the size of the segment, the fonts used in the segment etc.

2.5 Molecular chemistry

Another interesting example is from the area of molecular chemistry. In [8], a task is described in which the properties, e.g., the boiling point, of a particular group of molecules, the benzodiazepines, are to be predicted from structural properties of the chemical molecules. A method suitable for representing molecular structures was studied in [8, 9]. The method works by selecting a particular ring structure of the molecule to be the root of a graph. This ring structure is present in all benzodiazepines and hence is a suitable starting point for building up a graph structure. Offsprings are formed by atoms or structures directly attached to this ring structure. The graph is built up recursively as illustrated in Figure 2.10. The

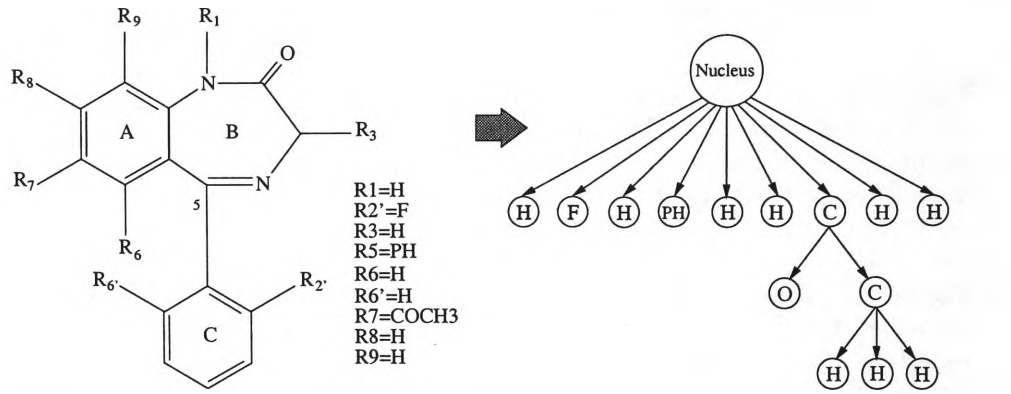


Figure 2.10: Generating a graph representation from a molecule can be performed by collating ring structures, and by respecting the connectivity between atoms in the structure.

method shown in Figure 2.10 represents ring structures by a single node. This is to avoid the generation of cyclic graphs which can cause problems with many graph processing methods.

The label associated with each node in the graph is a unique numeric representation of the chemical element (or compound) represented by the node. An example: The data label associated with the nodes shown in Figure 2.10 may be 1 if a node represents hydrogen (H), 2 if a node represents fluoride (F), 3 for nodes representing the compound PH, and so on.

2.6 Software engineering

Another group of challenging real world problems comes from the area of software engineering. Tasks can be the measurement of software quality as described in [62, 68], or the effective reuse of software components, and many other tasks. In [20] a model is described that represents object oriented code by a graph. These structures are then used for training a connectionist network (neural networks), where the goal is to find a representation of object-oriented specifications to be used in analogical mapping. The hope is that through analogical reasoning, object oriented specifications can be found that are suitable for reuse in other environments. The framework set by [20] includes a schema for representing object oriented specifica-

tions in terms of directed graphs. The example given in Figure 2.11 illustrates how an object oriented role model is represented by a graph.

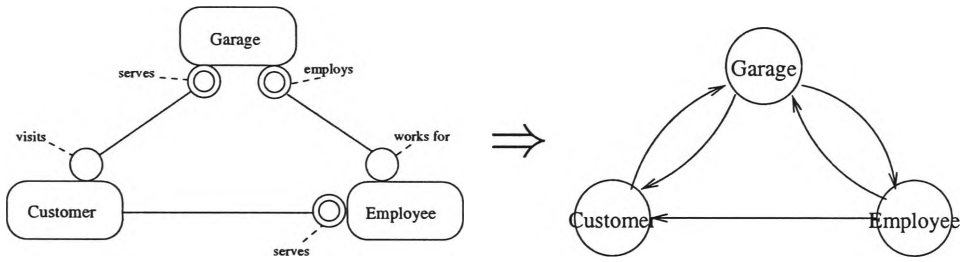


Figure 2.11: An object oriented role model (left), and a graph representation (right). The notation is explained in the text.

Here, the model represents an object by a super-ellipse, where collaborating objects are connected by a solid line. This line effectively represents a message path between objects where semantic relationships are represented by small circles, called ports. A port defines the knowledge an object has about the remote object. If an object knows exactly about one collaborator, then this is denoted by a small circle. A double circle indicates that the object knows about any number of the collaborator; a missing circle indicates that the object does not know the collaborator. A graph representation is obtained by creating a node for each object in the role model. Nodes are connected by directed links if there is a semantic relationship defined between the nodes. Note that the role model carries more information than the graph representation. This is because no distinction is made between the relationship which *knows about one collaborator* and which *knows about any number of collaborators.* This limitation may contribute to the limited success achieved by [20] when applying this method to a neural network known as LRAAM (Labelling Recursive Auto-Associative Memory) [83].

The label associated with each node in a graph is a composition of a unique binary node identifier and binary values to indicate the state of outlinks. Missing outlinks are encoded as 0, otherwise the value 1 is used to indicate an existing outlink. Hence, if there are n binary values to represent the node ID, and m binary values to represent the outlinks (where m must be greater or equal to the maximum out-degree of any node in the data set), then the dimension of the data label $\mathbf{l}$ is: $|\mathbf{l}| = n + m$. For example: If $m = n = 2$, then the data label associated with the nodes shown in Figure 2.11 is given as follows:

$$\begin{aligned} \mathbf{l}_{\text{Garage}} &= (0,0,1,1) \\ \mathbf{l}_{\text{Customer}} &= (0,1,1,0) \\ \mathbf{l}_{\text{Employee}} &= (1,0,1,1) \end{aligned}$$

2.7 Conclusion

In this Chapter, we have considered a number of practical examples which can be represented in terms of graph structures. We have indicated in each example how a graph structure can be obtained. In addition, we showed how features can be extracted and arranged in data labels.

The techniques for extracting graph structures and their associated data labels are specific to the practical examples we have indicated in this Chapter. While it is difficult to generalize these techniques and give a universal graph and data label extraction method as each practical problem may require different graph and data

label extraction methods, nevertheless the examples given in this Chapter serve as prototypes of such process. By considering a particular problem at hand, and by assuming some a priori knowledge on the practical situation, it is normally not too difficult to extract an underlying graph structure and its associated data labels, if a graph structure is a convenient way to represent the problem.

There are a number of interesting questions associated with the extraction of a graph structure and its associated data labels:

- Given a particular practical problem, is it always possible to find a graph structure representation? We know that the answer to this question is false. For example, there are images which simply cannot be represented in terms of a graph structure, e.g., random visual patterns, smooth textural patterns. However, a different question to ask is: what characteristics of a problem would allow it to be represented by a graph structure. We do not know the answer to this question. It is suspected that this question is hard to answer.
- Given a particular problem, and given that we know a graph structure can be extracted, is it always possible to extract a unique graph representation? We do not know the answer to this question. However, it appears from our own experience there are different ways of extracting a graph structure. Sometimes the graph structures produced visibly different (though there might exist some underlying equivalence). Hence there is a high possibility that the answer to this question might be false.
- Given a practical problem and given that we know we can extract a graph structure from this problem, is it possible to extract a set of unique data labels. For simplicity, we assume that we can extract a unique graph representation from this practical problem. We know the answer to this question is false. We know that there are different ways to extract data labels dependent on the aim of data processing. However, another question to ask is: under the same assumptions given previously, if it is for a particular aim, can we still obtain a unique set of data labels. We suspect the answer to this question is still false, as we know we can either add more features or less features to the data labels. But here is an interesting question: would the result of computation be different. In other words, if we are given a set of graphs representing various practical situations. If we extract two different set of data labels for the same graph, one more elaborate, e.g., in the case of image processing, including size, colour, location, while the second set of data labels is less elaborate, including, e.g., only size. If the aim of data processing is to recognise different categories of images from this given set of graphs, the question is: would the results be different if we use the set of more elaborate data labels, and if we use the set of less elaborate data labels. Again we do not know the answer to this question. We suspect that the answer would depend on the particular problem at hand.

In this thesis, we will not answer any of these interesting questions as we suspect that these questions are not answerable, at least in the loose informal manner which we defined them here. Instead, we propose to develop algorithms which can be used to process given graph structures. Implicitly we assume that through preliminary experimentations, we have found that the set of data labels used would be good enough for the problem at hand. Naturally, in this thesis, we will only consider problems which can be represented in terms of graph structures.

Chapter 3

Self Organizing Maps

3.1 Introduction

This chapter gives an introduction to vector quantization (VQ) and Kohonen's self organizing map (SOM) model [1]. A SOM typically performs a mapping from a high dimensional continuous input space to a low dimensional display space. It is common to discretise the display space using a layer of sensorial units. Such units are commonly referred to as *neurons* in recognition of their observed properties which are similar to those observed in biological neurons. Biological neurons demonstrate self-organizing properties in that different areas of the brain serves different purposes. This segmentation of biological neurons takes place during early stages of the brain's development and seems to be the result of some learning process. In this chapter, when referring to neurons, we actually mean models of artificial neurons. Note that these neural models differ from those used in describing artificial neurons in other neural architectures such as Multilayer Perceptron and Cascade Correlation networks (in terms of their learning behaviour and their architectures). In the literature, the neurons in a SOM are always arranged as a q-dimensional layer where often $q = 2$, and the layer is commonly called a *map*. Hence, one important property of the SOM model is that it reduces the dimensionality of input data through some mapping process to one with a lower dimension. In addition, the SOM model exhibits a topology preserving mechanism that has the effect of mapping related (similar) data in the high dimensional input space onto topological close areas of the display space. In other words, if two points are topologically close in the high dimensional space, then they will be remain topologically close in the low dimensional space.

In practice, the SOM model has become a widely applied mechanism which is used to either detect and visualize properties of high dimensional input data, or as a pre-processing step to filter out essential properties of a data set, and hence reduce the complexity of processing that data by reducing its dimensionality.

This chapter introduces the SOM-model and illustrates the learning algorithm in some detail. The theory introduced in this chapter forms the basis of an extended

[1] In this thesis, we abuse the terminology slightly by referring synonymously to a "self organizing map model" and a "self organizing map" method or technique. In strict terms, self organizing map is a technique to visualise a set of high dimensional data. However, in common with normal usage, many people refer to it as a self organizing map method or a self organizing map model.

model of the SOM. The extended SOM-model, which is addressed in Chapter 4, enhances the capabilities of SOM by allowing the SOM concept to map more complex data structures such as DOAGs. A further advancement of the SOM model is introduced in Chapter 5 where a supervised SOM model for the structured domain is introduced. As will be observed, the extended models contain the standard model as a special case, and hence are backward compatible in the sense that the same data applied to the standard SOM can be processed by the SOM-SD algorithm. The common feature of all the models described respectively in Chapter 3, Chapter 4 and Chapter 5 is that input data, either flat structures or graphs, are mapped onto a fixed dimensional display space. This is a particular interesting property when considering the processing of graph structured information.

The standard SOM model is a direct outcome of work in the area of "vector quantization" and "learning vector quantization". The SOM model as introduced by T. Kohonen was a logical next step in the evolution of methods for vector quantization.

Vector quantization (VQ) and self organizing maps (SOMs) are popular and well studied methods for quantizing sets of input vectors [59]. These methods are typically trained in an unsupervised fashion though some supervised ones do exist [55].

This chapter is structured as follows: The basis for understanding of self organizing maps is laid through Section 3.2 which introduces the topic of vector quantization, and Section 3.3 which addresses learning vector quantization. The description of the SOM model is given in Section 3.4, and a detailed overview to the SOM learning algorithm is presented in Section 3.4.1. Finally, Section 3.5 gives a summary of this chapter and motivates for Chapter 4 which will address a SOM model for graphs.

3.2 Vector quantization

Vector quantization (VQ) is a technique whereby the input space is subdivided into a number of distinct regions [61]. Each region is represented by an n-dimensional reference vector $\mathbf{m}_i \in I\!R^n, i = 1, \ldots, k$. The reference vectors together form a *codebook*, hence, reference vectors are also called codebook vectors [2]. Given a new input vector $\mathbf{u} \in I\!R^n$, a vector quantiser determines the region in which the vector lies. Then, the quantiser outputs the codebook vector that represents this region. This is performed through the computation of the closest Euclidean distance between the input $\mathbf{u}$ and all codebook entries. The competitive learning rule shown in (Equation 3.1) is then applied.

$$\Delta \mathbf{m}_i = \begin{cases} \alpha(t)(\mathbf{u} - \mathbf{m}_i) & \text{if } \mathbf{m}_i \text{ is the closest codebook to } \mathbf{u} \\ 0 & \text{else} \end{cases} \tag{3.1}$$

where α is a learning rate, typically much smaller than 1. This algorithm is applied recursively over the entire set of training data until a convergence criterion is met. Often, α decreases monotonically with the number of iterations. Indeed the formal theory for the convergence of this algorithm requires that α must decrease at least at a rate inversely proportional to the number of iterations. Effectively, vector quantization is an unsupervised learning scheme which can find clusters in the input space, and can be seen as a method that compresses information by representing the training set through a relatively small collection of codebook entries.

[2]Note that the codebook vectors may be obtained using a random initialisation process, or from a set of training data set.

The supervised version of vector quantization is called *Learning Vector Quantization* or LVQ, and is addressed in the following section.

3.3 Learning Vector Quantization

Learning vector quantization (LVQ) is a group of supervised vector quantization methods introduced by Kohonen in 1986 [54, 56]. The goal of LVQ is to have the neural network "discover" an underlying structure in the data by finding how the data is clustered. Classes are represented by a relatively small number of codebook vectors, properly placed within each class zone such that the decision boundaries are approximated by the nearest-neighbour rule. Unlike the normal k-nearest-neighbour classification, the original samples are not used as codebook vectors, but they are used to tune the codebook vectors. LVQ aims for the optimal placement of these codebook vectors.

A classifier based on LVQ employs k codebook vectors $\mathbf{m}_i \in I\!R^n, i = 1, \ldots, k$. Each codebook vector is labelled with an integer representing the class. Usually, several codebook vectors are assigned to each class. An input vector $\mathbf{u}$ is said to belong to the same class as the nearest codebook vector $\mathbf{m}_c$. This is illustrated in (Equation 3.2):

$$c = \arg\min_i \|\mathbf{x} - \mathbf{m}_i\| \tag{3.2}$$

There are three different updating rules for LVQ, namely LVQ1, LVQ2, and LVQ3. The simplest version of LVQ, commonly referred to as LVQ1, adapts only the closest codebook entry as shown in (Equation 3.3):

$$\Delta\mathbf{m}_i = \begin{cases} \alpha(t)(\mathbf{u} - \mathbf{m}_i) & \text{if } \mathbf{x} \text{ and } \mathbf{m}_i \text{ belong to the same class.} \\ -\alpha(t)(\mathbf{u} - \mathbf{m}_i) & \text{else} \end{cases} \tag{3.3}$$

All other codebook entries $\mathbf{m}_i, i \neq c$ remain unchanged. The learning rate α is normally defined to be $0 \leq \alpha \leq 1$ and decreases monotonically with the number of iterations until zero. Other versions of LVQ also adapt the second closest codebook in a similar manner. There is no VQ method which adapts all codebook entries in its learning rule. This is due to the existence of undesired side effects which would arise out of such an approach.

A method that extends the ideas expressed in VQ arranges the codebook entries onto a two-dimensional plane allowing the updating of all codebooks in a learning step. This method has become known as a *Self Organizing Kohonen Map* or SOM, and is addressed in the following section.

3.4 The classic Self Organizing Map

Kohonen's [58, 57] self-organizing (feature) map (SOM) has become one of the most well known form of unsupervised learning. It expands the ideas expressed in Section 3.2 and Section 3.3 in that it defines a *neighbourhood* relation on the codebook entries. In addition, SOM updates all codebook entries, not just the closest codebooks as in VQ or LVQ.

SOM was developed to help identify clusters in multidimensional datasets. The

SOM does this by effectively packing the dataset onto a two-dimensional plane [3]. The result is that data points that were "similar" to each other in the original multidimensional data space are then mapped onto nearby areas of the two dimensional output space. SOMs combine competitive learning with dimensionality reduction by smoothing the clusters with respect to an a priori grid. The SOM algorithm involves a trade-off between the accuracy of the quantization and the smoothness of the topological mapping. The SOM is called a topology-preserving map because there is a topological structure imposed on the nodes in the network. A topological map is simply a mapping that preserves neighbourhood relations.

3.4.1 The SOM learning algorithm

The basic idea of SOM is simple. The SOM defines a mapping from high dimensional input data space onto a regular two-dimensional array of neurons. Every neuron i of the map is associated with an n-dimensional codebook vector $\mathbf{m}_i = (m_{i1}, \ldots, m_{in})^T$. The neurons of the map are connected to adjacent neurons by a neighbourhood relation, which defines the topology, or the structure, of the map. The most common topologies in use are rectangular and hexagonal [59].

Adjacent neurons belong to the neighbourhood N_i of the neuron i. Neurons belonging to N_i are updated according to a neighbourhood function $f(.)$. Most often, $f(.)$ is a *Gaussian-bell* or a *Mexican-hat* function. In the basic SOM algorithm, the topology and the number of neurons remain fixed from the beginning. The number of neurons determines the granularity of the mapping, which has an effect on the accuracy and generalization of the SOM [59].

During the training phase, the SOM forms an elastic cover that is shaped by input data. The algorithm controls the cover so that it strives to approximate the density of the data. The reference vectors in the codebook drift to the areas where the density of the input data is high. Eventually, only few codebook vectors lie in areas where the input data is sparse.

The training algorithm of the weights associated with each neuron in the q-dimensional lattice can be trained [59] using a three step process as follows:

Step 1 Competitive step. One sample vector u is randomly drawn from the input data set and its similarity to the codebook vectors is computed. There are two possible matching mechanisms:

1. Similarity. In this case, we evaluate the dot product between the weight of the neuron and the input, i.e.,

$$s_i = \mathbf{u}^T \mathbf{m}_i \tag{3.4}$$

where $\mathbf{u}$ denotes the input vector and $\mathbf{m}_i$ denotes the weight vector associated with neuron i. The winning neuron, denoted by the neuron r, satisfies the relationship:

$$r = \arg\min_i \mathbf{u}^T \mathbf{m}_i \tag{3.5}$$

2. Minimum Euclidean distance. In this case, we measure the Euclidean distance $||\mathbf{u} - \mathbf{m}_i||$ between the input and the weights of the neurons.

[3] For simplicity reasons, we restrict ourself to two-dimensional Kohonen maps. In theory the input data can be mapped into a q-dimensional maps, where $q \in \mathbb{N}^+$ are possible.

The winning neuron is obtained through [4]:

$$r = \arg\min_i ||\mathbf{u} - \mathbf{m}_i|| \tag{3.6}$$

Step 2 Cooperative step. After the best matching unit $\mathbf{m}_r$ has been found, the codebook vectors are updated. $\mathbf{m}_r$ itself as well as its topological neighbours are moved closer to the input vector in the input space i.e. the input vector attracts them. The magnitude of the attraction is governed by the learning rate α and by a neighbourhood function $f(\Delta_{ir})$, where Δ_{ir} is the topological distance between $\mathbf{m}_r$ and $\mathbf{m}_i$. As the learning proceeds and new input vectors are given to the map, the learning rate gradually decreases to zero according to a specified learning rate function type. Along with the learning rate, the neighbourhood radius decreases as well [5]. Again we have two cases:

1. Similarity. In this case, the updating algorithm is given by:

$$\Delta\mathbf{m}_i = \begin{cases} \frac{\mathbf{m}_i + \eta\mathbf{u}}{\|\mathbf{m}_i + \eta\mathbf{u}\|} & \text{if } i \in N_r \\ 0 & \text{if } i \notin N_r \end{cases} \tag{3.7}$$

   where N_r denotes the neighbourhood of the winning neuron r, $\mathbf{m}_i$ the weight vector, and η is a learning coefficient. The neighbourhood function can be either rectangular or hexagonal as long as it forms a lattice.

2. Euclidean distance. In this case, the updating algorithm is given by:

$$\Delta\mathbf{m}_i = \alpha(t) f(\Delta_{ir})(\mathbf{m}_i - \mathbf{u}) \tag{3.8}$$

   where α is a learning coefficient, and instead of a neighbourhood region, we have used a neighbourhood function $f(.)$, which controls the amount which the weights of the neighbouring neurons are updated. The neighbourhood function $f(.)$ can take the form of a Gaussian function:

$$f(\Delta_{ir}) = \exp\left(-\frac{\|\mathbf{l}_i - \mathbf{l}_r\|^2}{2\sigma^2}\right) \tag{3.9}$$

   where σ is the spread, and $\mathbf{l}_r$ is the location of the winning neuron, and $\mathbf{l}_i$ is the location of the i-th neuron in the lattice. Other neighbourhood functions are possible.

Step 3 Recursion. The steps 1 and 2 together constitute a single training step and they are repeated until the training ends. The number of training steps must be fixed prior to training the SOM because the rate of convergence in the neighbourhood function and the learning rate is calculated accordingly.

After the training is over, the map should be topologically ordered. This means that n topologically close (using some distance measure e.g. Euclidean) input data vectors map to n adjacent map neurons or even to the same single neuron.

[4] Note that minimizing the Euclidean distance is equivalent to maximizing inner product ($\arg\max_i \langle u, m_i \rangle$) if u and m_i have unit length.

[5] Generally, the neighbourhood radius in SOMs never decreases to zero. Otherwise, if the neighbourhood size becomes zero, the algorithm reduces to vector quantization (VQ) and no longer has topological ordering properties (Kohonen 1995 [59], p. 111).

After the SOM has been trained, it is important to know whether it has properly adapted itself to the training data. Because it is obvious that one optimal map for the given input data must exist, several map quality measures have been proposed. Usually, the quality of the SOM is evaluated based on the mapping precision and the topology preservation.

The mapping precision measure describes how accurately the neurons respond to the given data set. For example, if the reference vector of the best matching unit calculated for a given testing vector x_i is exactly the same as x_i then the error in precision is 0. Normally, the number of data vectors exceeds the number of neurons and the precision error is thus always different from 0. A common measure that calculates the precision of the mapping is the average quantization error E_q over the entire data set [59]:

$$E_q = \frac{1}{N}\sum_{j=1}^{N}\|\mathbf{x}_j - \mathbf{m}_r\| \tag{3.10}$$

The topology preservation measure describes how well the SOM preserves the topology of the studied data set. Unlike the mapping precision measure, it considers the structure of the map. For a strangely twisted map, the topographic error is large even if the mapping precision error is small. A simple method for calculating the topographic error is [59]:

$$E_t = \frac{1}{N}\sum_{k=1}^{N} g(\mathbf{x}_k), \tag{3.11}$$

where $g(\mathbf{x}_k)$ is 1 if the first and second best matching units of $\mathbf{x}_k$ are not next to each other. Otherwise $g(\mathbf{x}_k)$ is 0.

The SOM can be applied to tasks where large amounts of unclassified data is available. Important practical applications of SOMs are in exploratory data analysis, pattern recognition, speech analysis, robotics, industrial and medical diagnostics, instrumentation and control.

3.5 Conclusions

The Self Organizing Map model allows to find clusters in high dimensional data sets by means of a topological preserving mapping process. This is performed without the use of a supervisor or a *teacher* signal. The SOM algorithm is relatively simple and grows in complexity linear with the number of input data and the number of neurons on the map.

There are a number of issues associated with SOM which include:

- Convergence of the SOM algorithm. There does not exist a convergence theorem for SOM, even though a convergence theorem exists for VQ, or LVQ [59]. The reason may be attributed to the fact that it is rather difficult to handle the neighbourhood function in convergence analysis.
- It appears quite difficult to formulate SOM from the minimisation of a cost function point of view. Even though there were various attempts in formulating the algorithm from an error function point of view, these were often

viewed as "un-natural" [50]. A major reason for the desire to formulate the algorithm in terms of an error function is that it would be much easier to consider the convergence of the algorithm from this point of view.

- A natural question to ask is: is it possible to map the continuous input space onto a continuous display space, rather than a discrete display space. This appears to be possible. However, the algorithm is far more complicated [93]. We will not consider this aspect in this thesis.
- Recently an interesting view of SOM is from a generative point of view [11]. This is known as generative topological map (GTM) [12, 94]. Instead of viewing the input from the point of view as illustrated in this chapter, the generative point of view considers the generation of the map from an input data set based on a priori model. The problem then becomes the estimation of the parameters of the model from the input data set. In other words, the generative topological map can be considered as the inverse of the SOM as considered in this chapter. We will not consider this point of view, as currently it is not clear how to extend our work to be presented in Chapter 4 to this formulation.

However, SOM is limited to the processing of fixed sized input vectors. This is because the methods employed do not allow the consideration of variable sized inputs such as data sequences or richer structured data. An approach which relaxes this limitation is described in the following chapter.

Chapter 4

SOM for structured information

4.1 Introduction

This section introduces a self organizing map which can process structured data (SOM-SD). A feature of the proposed method is that structures e.g., DOAGs, can be processed in an unsupervised fashion. Moreover, there are two different retrieval possible methods, demonstrating the flexibility of SOM-SD. As will be demonstrated, a SOM-SD model defines a general mechanism which includes standard SOMs as a special case. The behaviour and performance of the SOM-SD model is investigated through its applications to benchmark problems. It will be found that the SOM-SD model provides an efficient mechanism for tasks involving graphs.

As far as we are aware, the concepts and ideas presented in this chapter are novel. The work has been inspired by the original work of Kohonen [59].

The structure of this chapter is as follows: Section 4.2 demonstrates various ways in which data can be represented. This helps with the understanding on how to process graph structured information using a SOM. In Section 4.3 an alternative approach of presenting a SOM architecture is displayed. It is assumed that a SOM consists of two tightly connected maps, one for the mapping of data label, the other for the mapping of the underlying structure of the data. As will be observed, it is trivial to merge the two maps into just one. The training mechanism for SOM-SD is illustrated using a step-by-step example in Section 4.3.1. A more formal approach for describing the learning algorithm is taken in Section 4.3.2. Section 4.3.3 defines measurements for the network error, viz. the quantization error. Retrieval mechanisms which are available for SOM-SD are given in Section 4.3.4. Tips for speeding up network training are given in Section 4.3.5, and similarities with other neural network models for structured domains are shown in Section 4.3.6. A comprehensive set of experiments in Section 4.4 is used to illustrate the capabilities of the SOM-SD model. Experiments were conducted on a simple learning task (Section 4.4.1), a more complex data set (Section 4.4.2), and a large, relatively difficult learning task (Section 4.4.3). A short note on the long term dependency problem is given in Section 4.4.4. Finally, Section 4.5 summarizes the findings and draws some conclusions.

4.2 Data presentation

There are numerous ways to display and present graph structured data. In this section we will give an overview of some common methods for graph representations. To do this, we restrict ourselves to labelled directed acyclic graphs [1]. This section addresses 'external' data representation which is the structure of data as presented to the network. The 'internal' representation as encoded by the network is addressed in Section 4.3.1.

A common method for presenting graphs is shown in Figure 4.1. Such a representation allows a simple analysis of the structure by a human observer. However, for machine learning, such a representation is not suitable.

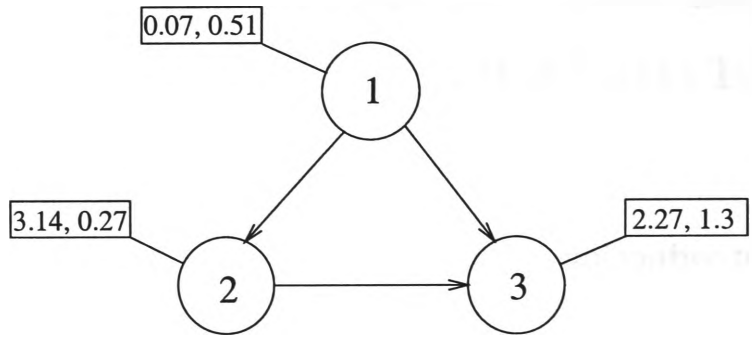


Figure 4.1: An example of a labelled directed graph. Each node in the graph is identified by a unique symbol (here numbers). Associated with each node is a 2-dimensional real valued data label. The node numbered 1 is called *root* or *super-source*, node 3 is the *terminal* node of this graph. All other nodes are *intermediate* nodes. The maximum out-degree (the maximum number of children at any node) of this graph is 2.

In machine learning, a data representation such as the one defined in Section 2.2 can be chosen. Thus, a graph representation can be given as follows: let V be a finite non-empty set of nodes, each representing a node in the graph which is associated with a p-dimensional data label. Further, let $E \subseteq V \times V$, where $(v_1, v_2) \in E$ if and only if there is a connection between the nodes v_1 and v_2. Then the pair (V, E) is a directed graph representation of the data structure. With this, the graph shown in Figure 4.1 can be written as: $V = \{1, 2, 3\}$, and $E = \{(1, 2), (1, 3), (2, 3)\}$.

Such a representation is easily transformed into a tabular form such as demonstrated in Table 4.1. It can be observed that this representation effectively transforms a graph structure into a set of vectors. These vectors can be made constant in size if the maximum out-degree and the maximum data label dimension are known. For nodes with missing children or smaller data label size, padding with a suitable constant value can be deployed. Fixed sized vectors are useful to serve as an input for an artificial neural network. Traditional neural models use fixed size data labels as input where relationships between the data is not considered. In the case of graphs, relationships between the data labels are well defined, and hence, can be used to assist network training. In practical terms, graphs are processed in a bottom-up fashion (from the terminal nodes towards the root). This is necessary in order to make available information about the children when the parent node is processed.

[1] The SOM-SD described in this thesis allows the processing of some special classes of directed cyclic graphs. We will not describe it as our main aim is the processing of acyclic graphs.

Node id	Data label	Children
1	(0.07, 0.51)	2,3
2	(3.14, 0.27)	3
3	(2.27, 1.30)	

Table 4.1: This table represents a directed labelled graph in a tabular form. The structure, content, and maximum out-degree of this graph are identical to the graph displayed in Figure 4.1.

In contrast with many other neural models, SOM-SD is unable to receive feedback from other neurons in a conventional sence since there are no weighted links between the neurons. An alternative method of passing information from one neuron to another needs to be found. SOM-SD maps data onto a two-dimensional map so that data can be represented at a particular "location" on the map. This suggests that for a recursive processing of data structures such as the processing of graphs and sequences, is to include some information about the mapping of offsprings into the feature vector of parent nodes. One way of achieving this is by adding the spatial location, i.e., the location of the winning neuron of the offsprings to the set of features of the parent node. As a result, the network needs to know where the winning neurons for all children nodes are located on the map when processing the parent node. Thus, the network utilizes not only the data label of a node as input but also include some information about the best matching codebook vector for each offspring. This forces the SOM-SD network to process the data in a bottom-up fashion.

In practice, for each node in the graph, the network input will be vectors which consist of:

(a) the p-dimensional data label $\mathbf{l}$,

(b) the coordinates $\mathbf{c}$ of the winning neuron for each child,

The vector $\mathbf{c}$ is qo-dimensional, where o the maximum out-degree of any graph in the data set, and q is the dimension of the map. Without loss of generality of the model described, we set q to be 2. Hence, $\mathbf{c}$ consists of o tuples, each tuple being a 2 dimensional vector representing the x-y coordinates of the winning neuron of a child node or an offspring node. Offsprings, which essentially are sub-graphs, are represented at a particular position within the map. Hence, the tuples in $\mathbf{c}$ contain the coordinates of codebook vectors which were associated with the offsprings of the current node. This dictates a bottom-up processing of data where terminal nodes are processed first. Once it is known where the children are represented on the map, we can update the vector component $\mathbf{c}$ of the parent node accordingly.

4.3 The SOM-SD architecture

The previous section defined a suitable input for a SOM-SD network and suggested already a way of processing graph structured data. In this section, the network architecture will be described.

Analogous to the classical SOM, the SOM-SD network is a two dimensional map. But since we have two components in the input data, the vectors $\mathbf{l}$ and $\mathbf{c}$, there are two SOM layers (illustrated in Figure 4.2). The first layer is denoted as M^l which processes the labels $\mathbf{l}$, and the second layer is M^c which processes the location $\mathbf{c}$

(the coordinates) of the offsprings. These two layers are strongly linked in that a neuron i consists of two codebook vectors $\mathbf{m}_i^l \in M^l$, and $\mathbf{m}_i^c \in M^c$. Thus, each of the two codebook vectors belonging to a neuron i is located at a congruent position within the corresponding layer.

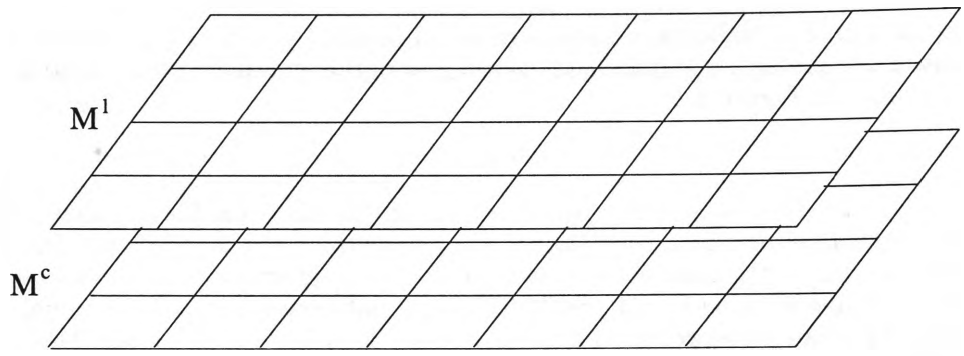


Figure 4.2: The SOM-SD architecture. Two layers of codebook vectors, one for each component of the input, are present. The two layers are strongly linked together in that a neuron i consists of two codebook vectors, one from each layer, located at congruent positions.

Alternatively, it is possible to combine the two layers of a SOM-SD network into a single layer requiring the concatenation of the two input vector component into a single vector. During learning, the similarity measure (e.g. the Euclidean distance) has to weight the input elements $\mathbf{l}$, and $\mathbf{c}$. This weighting is necessary to balance the influence of the elements to the training algorithm. For example: $\mathbf{l}$ may be high dimensional with elements larger than the elements in $\mathbf{c}$. The network would be unable to learn relevant information provided through $\mathbf{c}$. This is overcome by weighting the network input components. This will become clearer in Section 4.3.1. when the training algorithm is given.

A network input vector $\mathbf{x}$ is built through the concatenation of $\mathbf{l}$ and $\mathbf{c}$ so that $\mathbf{x} = [\mathbf{l}^T, \mathbf{c}^T]^T$. As a result, $\mathbf{x}$ is a $n = p + 2o$ dimensional vector. The codebook vectors $\mathbf{m}$ are of the same dimension.

It is more convenient to utilize a single layer of neurons since it simplifies the notation and the learning algorithm. In fact, a single layer SOM-SD network features a network architecture which is identical to the original SOM. Hence, in the following, when addressing SOM-SD networks, we implicitly refer to a SOM-SD model featuring a single layer.

4.3.1 Training the SOM-SD

The neural network architecture of a SOM-SD is identical to Kohonen's original self-organising map, and also the input data are fixed sized vectors. However, data need to be processed differently since we need to take into account that the input vector is a composition of data label, and structural information. While the data label remains constant during learning, the structural information for all except the terminal nodes changes. This is because a terminal node needs to be mapped before a parent node can be processed. The parent node requires the mapping of all its offsprings in order to initialize $\mathbf{c}$. The mapping of the terminal node will change during training, and hence will influence the network input for the parent node. The following example illustrates this concept.

Let us assume that we wish to process the data graph shown in Figure 4.1. For

each node, we generate the following initial data vectors:

$$\begin{array}{ll} (0.07, 0.51, c_{11}^1, c_{12}^1, c_{21}^1, c_{22}^1) & \texttt{root node} \\ (3.14, 0.27, c_{11}^2, c_{12}^2, c_{21}^2, c_{22}^2) & \texttt{node number 2} \\ (2.27, 1.30, c_{11}^3, c_{12}^3, c_{21}^3, c_{22}^3) & \texttt{leaf node} \end{array}$$

The first two elements are initialized with the data label of the node. The tuples (c_{i1}^k, c_{i2}^k) are the coordinates of the i-th child of node k. For each missing offspring we initialize (c_{i1}^k, c_{i2}^k) with $(-1, -1)$. This tuple marks the data vector with some illegal elements [2]. In particular, terminal nodes will always have all coordinates initialized with $(-1, -1)$. All other coordinates will be initialized with proper values later during the training phase. As a result, the initial set of data vectors can be re-written as follows:

$$\begin{array}{ll} (0.07, 0.51, c_{11}^1, c_{12}^1, c_{21}^1, c_{22}^1) & \texttt{root node} \\ (3.14, 0.27, c_{11}^2, c_{12}^2, -1, -1) & \texttt{node numbered 2} \\ (2.27, 1.30, -1, -1, -1, -1) & \texttt{leaf node} \end{array}$$

Note that the vector associated with node numbered 2 considers the leaf node to be its first child. The vector would look different if the leaf node was considered to be the second offspring. In this case, the vector associated with node numbered 2 would be $(3.14, 0.27, -1, -1, c_{21}^2, c_{22}^2)$. This shows that the proposed model handles ordered graphs as compared to unordered graphs for which the order of children does not matter.

Training proceeds in a bottom-up manner. Hence, the first node presented to the network is the terminal node. A distance measure (e.g. the Euclidean distance) is used to determine the best matching codebook entry. Let us assume that we have a 7×4 Kohonen map as shown in Figure 4.3. Assume further, that for the terminal node the best matching codebook entry was associated with a neuron located at $(5, 2)$. Then we have a situation as shown in Figure 4.3. Essentially, the codebook entry located at $(5, 2)$ is assumed to be most similar to the vector associated with the terminal node, and hence, is a representation of this node.

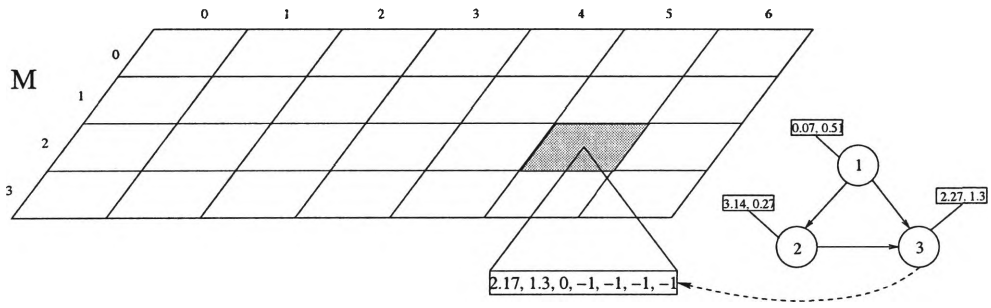


Figure 4.3: A sample SOM-SD architecture, a 7×4 map of 7-dimensional codebook vectors. Gray field highlights the best matching neuron for the given terminal node.

Since the structural representation of the sole offspring of node numbered 2 is available, we are able to proceed with processing the input graph. This is performed

[2]The tuple $(-1, -1)$ is illegal since **c** holds coordinates which can only be positive or zero. These illegal elements are used to mark missing children. Note that any other illegal combination can be used for the initialization.

by taking the data vector associated with node 2 and initializing (c_{11}^2, c_{12}^2) with the values $(5, 2)$ (the information about where the offspring is located). The remaining $(-1, -1)$ tuple remains unchanged since this node has no other offspring. As a result, we have:

$$(3.14, 0.27, 5, 2, -1, -1)$$

We can now obtain the best matching codebook entry. Let us assume that the best match was found at $(3, 1)$. We then obtain a situation as displayed in Figure 4.4. The codebook entry at $(3, 1)$ represents the sub-graph consisting of node number 2 and the terminal node.

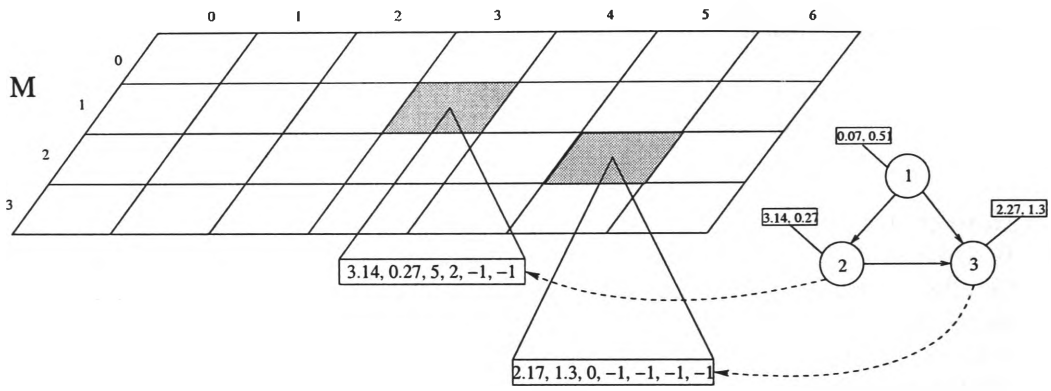


Figure 4.4: A sample SOM-SD architecture, a 7×4 map of 7-dimensional codebook vectors. The gray fields highlight the best matching codebook entry for the nodes processed so far.

Since the root node has only those nodes as offsprings that have already been processed, we can proceed further. Similarly as demonstrated at node 2, the input data vector associated with the root node is updated resulting in an input vector:

$$(0.07, 0.51, 3, 1, 5, 2)$$

which is presented to the network and the closest matching codebook entry is obtained. A situation such as the one shown in Figure 4.5 may arise.

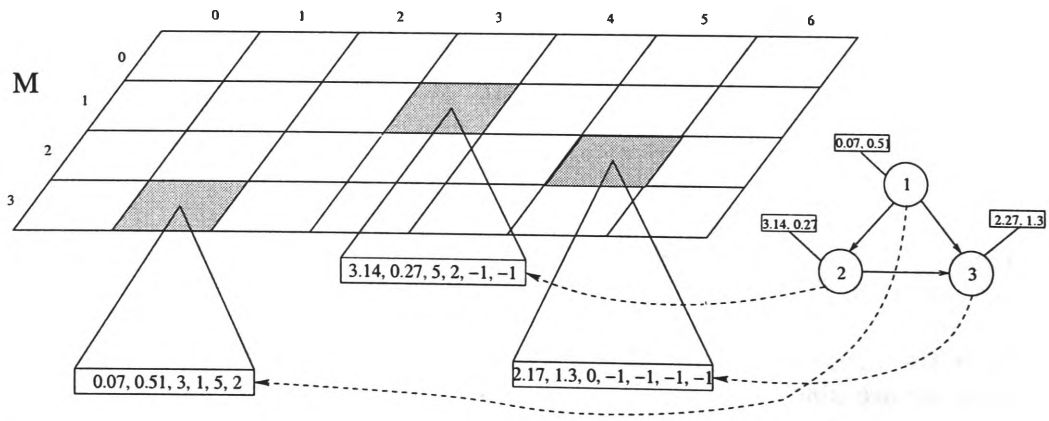


Figure 4.5: The sample SOM-SD network architecture after the mapping of all three nodes from the input graph.

In this last step, the root node may have found its best match at the location $(1,3)$. Then, this winner neuron will be a representation of the graph structure on the map, and we have completed the forward-step for this graph. The network is then updated by some updating rule which essentially will modify the winning codebook entries to become more similar to the input vectors. The introduction of a suitable updating mechanism is made in Section 4.3.2.

Other graphs from a data set can be processed in the same manner. We have completed an *iteration* after having presented all graphs in the data set exactly once. Typically, many iterations are performed during network training.

This example suppresses an important detail. Input data are composed of mutually different vector components. These components may differ in magnitude and dimension. Hence, for the similarity measure we need to weigh the influence of these components in order to balance the impact on the similarity value. The following section illustrates this, and formalizes the learning algorithm in more detail.

4.3.2 The SOM-SD learning algorithm

Training a SOM-SD is performed in a similar manner to the classical approach [59]. The difference is that for computing the similarity, the similarity measure needs to be weighted. Also, some components (the vector $\mathbf{c}$) of the input vector need to be updated at every training step. The following training algorithm illustrates this:

Step 1 A node j is chosen from the data set. When choosing a node special care has to be taken that the children of that node have already been processed. Hence, at the beginning the terminal nodes of a graph are processed first, the root node is considered last. Then, vector $\mathbf{x}_j$ is presented to the network. The winning neuron r is obtained by finding the most similar codebook entry $\mathbf{m}_r$. This can be achieved, e.g., by using the Euclidean distance as follows:

$$r = \arg\min_i \|(\mathbf{x}_j - \mathbf{m}_i)\mathbf{\Lambda}\| \tag{4.1}$$

where $\mathbf{\Lambda}$ is a $n \times n$ dimensional diagonal matrix. Its diagonal elements $\lambda_{11} \cdots \lambda_{pp}$ are set to μ_1, all remaining diagonal elements are set to μ_2. The constant μ_1 influences the contribution of the data label component to the Euclidean distance, and μ_2 controls the influence of the children's coordinates to the Euclidean distance.

Step 2 After the best matching neuron has been found, the winning codebook vector and its neighbours are updated.

$$\Delta\mathbf{m}_i = \alpha(t) f(\Delta_{ir})(\mathbf{m}_i - \mathbf{x}_j) \tag{4.2}$$

where α is a learning rate which gradually decreases to zero, $f(.)$ is the neighbourhood function depending on Δ_{ir} which is the topological distance between neuron i and neuron r.

Step 3 The coordinates of the winning neuron are passed on to the parent node which in turn updates its vector $\mathbf{c}$ accordingly [3].

[3]This step neglects the potential changes in the state of the desendants due to the weight change in step 2. This approximation is required to speed up the training process. The approximation works best when processing all the nodes from one graph before moving to the next graph.

Step 4 Cycles of Steps 1 to 3 are executed repeatedly until a given number of training iterations is performed, or when the mapping precision has reached a given threshold.

Note that step 2 requires that all vector elements in $\mathbf{x}$ are real values. During training, the vector components $\mathbf{c}$ are treated as reals. During retrieval, the components in $\mathbf{c}$ are converted back to natural numbers by using a common rounding function.

Gonzales et.al. [35] describes problems when applying SOM to non-stationary data. In our case, the weight change in training step 2 may produce non-stationary input data as the desendant vector component $\mathbf{c}$ may change at successive iterations. In practice, however, this issue was not found to cause any particular problem. We contribute this to a.) the fact that $\mathbf{c}$ contains cardinal values and hence, small weight changes do not have an immediate effect on the values in $\mathbf{c}$, and b.) choosing a small initial learning rate $\alpha(t)$ decreasing linearly with the number of iterations to zero ensures network convergence.

The optimal choice of the weight values μ_1 and μ_2 depends on the dimension of the data label $\mathbf{l}$, the magnitude of its elements, the dimension of the coordinate vector $\mathbf{c}$, and the magnitude of its elements. The Euclidean distance in Equation 4.1 is computed as follows:

$$d = \sqrt{\mu_1 \sum_{i=1}^{p} (\mathbf{l}_i - \mathbf{m}_i)^2 + \mu_2 \sum_{j=1}^{2o} (\mathbf{c}_j - \mathbf{m}_{n+j})^2} \tag{4.3}$$

Hence, it becomes clear that the sole role of μ_1 and μ_2 is to balance the influence of the two terms to the behaviour of the learning algorithm. Ideally, the influence of the data label and the coordinate vector on the final result is equal. One way of obtaining the pair of weight values is through the following equation:

$$\mu_1 = \frac{n}{m} \frac{\sum_{j=1}^{m} \phi(|\mathbf{l}_j|)}{\sum_{i=1}^{n} \phi(\mathbf{c}_i)} \mu_2 \tag{4.4}$$

where $\phi(|\mathbf{l}_i|)$ is the average absolute value of the i-th element of all data labels in the data set. Similarly, $\phi(\mathbf{c}_i)$ is the average of the i-th element of all coordinates. The data label $\mathbf{l}$ is available for all nodes in the data set, however, the coordinate vector becomes only available when training has started. Hence, $\phi(\mathbf{c}_i)$ needs to be approximated by assuming that the mapping of nodes is at random. As a result, the values in $\phi(\mathbf{c}_i)$ are simply half the dimension of the map.

Alternatively, good values for μ_1 and μ_2 can be obtained through Equation 4.5. This equation takes the distribution of the input data into account and produces more appropriate weight values.

$$\frac{\mu_1}{\mu_2} = \frac{n}{m} \frac{\sum_{j=1}^{m} (\phi(|\mathbf{l}_j|) - \sigma(|\mathbf{l}_j|))}{\sum_{i=1}^{n} (\phi(\mathbf{c}_i) - \sigma(\mathbf{c}_i))} \tag{4.5}$$

where $\sigma(\mathbf{v}_i)$ is the standard deviation of the i-th vector element of a vector $\mathbf{v}$. It has been found that Equation 4.5 predicts the weight values μ_1 and μ_2 more accurately since it suppresses the negative influence of rare extreme values, and hence will be used throughout this thesis. In order to obtain unique value pairs, we can make the assumption that $\mu_1 + \mu_2 = 1$.

4.3.3 Quantization error

During and after network training, it is important to know how well the network has learned to represent the input data. The quantization error as displayed in Equation 3.10 gives a good indication of the overall network performance in the case of standard SOM. However, when working in a graph structured domain then Equation 3.10 does not take into account the structural relationships between input vectors. Finding a perfectly matching codebook entry for a node $\mathbf{x}_j$ means that the quantization error for this node is zero. But this does not tell us how well the structure rooted by $\mathbf{x}_j$ has been mapped. Since the network is to learn graph structures and not single nodes, a measure has to be found which describes the accuracy of the mapping of graphs. An enhancement of Equation 3.10 adds the weighted quantization error of all offsprings to the quantization error of the current parent node.

$$e_j = \frac{1}{k+1}\left(d_j + \sum_{i=1}^{k}(q_i^{-1}e)\right) \tag{4.6}$$

where d_j is the error (e.g. Euclidean distance) associated with node $\mathbf{x}_j$, k is the number of children of node $\mathbf{x}_j$, and $q_i^{-1}e$ is the quantization error associated with the i-th child. Here, q^{-1} is a device denoting the availability of e from a child. The network's overall quantization error is computed as follows:

$$E_q = \frac{1}{N}\sum_{j=1}^{N} e_j \tag{4.7}$$

4.3.4 Retrieval with SOM-SD

Information, i.e., DOAGs stored in the output lattice can be retrieved. For example, if we are given a DOAG structure which is noisy or distorted, we can work through the mapping procedures to find a winning neuron **j**. We can then retrieve DOAGs by looking at which graphs from the training set activated the same neuron. Hence, this mechanism allows to retrieve graphs which are similar to a given graph. There are two ways in which the associated tree structure can be retrieved [41].

- **Retrieval by association** – Traditionally, retrieval in SOMs is performed by association. During training, input patterns are associated with the winning neurons. During a retrieval, or validation phase the best match with respect to a test pattern responds with the data (from the training set) that were linked to this neuron. Similarly, this can be done with SOM-SD. Here, every neuron is associated with the (sub-)structure that was activating the neuron. Note that not all neurons on the map are winning neurons for a pattern in the test set. Hence, it is possible that, during retrieval, a neuron is activated which does not have a link to a pattern from the training set. Commonly, this is avoided by finding the best matching neuron that has such a link. This method is memory intensive, as each winning neuron in the output layer would need to have an associated graph structure encoded.
- **Retrieval through recursive decoding** – Implicit in each input vector is an associated immediate subgraph structure of the previous step. For example, if we assume the weights of the winning neuron to be $[\mathbf{u}\ \mathbf{c}_1\ \mathbf{c}_2\ \ldots\ \mathbf{c}_c]$.

The vector $\mathbf{c}_i$ denotes the location of the winning neuron for child i. Thus it is possible to go to location $\mathbf{c}_i$, and consider the composite vector there. We can obtain the tree structure recursively using this step. Recursion stops when an illegal coordinate value is found in $\mathbf{c}_i$. Note that $\mathbf{c}_i$ is real valued whereas coordinate values are discrete. Hence, rounding is necessary to obtain the actual coordinate value.

4.3.5 Implementation issues

Typically a SOM-SD network is larger than traditional SOMs. This is because each input pattern is a graph which produces a set of input vectors for the network so that the input data set tends to become quite large (especially for very wide, or deep data structures). Fortunately, the complexity of the training algorithm for SOM-SD is the same as for SOM in that it grows linearly with the number of neurons in the network and the dimension of the data. However, the network has to be given more freedom to allow a suitable mapping according to data label and structure. With the increase of network dimension, training time increases as well. Hence, it becomes desirable to look for ways of optimizing the training algorithm.

Optimizations

Network training can be accelerated in a number of ways. An idea which has been suggested for traditional SOMs takes into account that the map becomes somewhat smoothly ordered after only a few iterations through the algorithm. So, searching the entire map for a particular feature vector's winning neuron is not necessary; the new winning neuron is likely to be close to the old winning neuron from the previous iteration. Hence, only nearby neurons need to be examined to obtain the new best match. This effectively speeds up training by a factor of possibly up to 10 or greater, depending on how large the map is.

A SOM-SD specific optimization has already been implicitly applied in the training algorithm stated in Section 4.3.2. By the very fact that in step 2, not only the best matching codebook vector is updated but also its neighbours, we would have to recalculate the correct position of all other nodes that are offsprings before the parent node can be processed. However, this change of position (of offsprings) can be assumed to be small for small learning rates. Since the learning rate decreases to zero during the training procedure, the error can eventually be neglected. Moreover, the error introduced by not updating the position of all offsprings is partially compensated in the next training iteration. By not retrieving the exact position of all offsprings before processing the parent node, the algorithm can become significantly faster depending on the out-degree of the input graphs.

Parallelization

The SOM and SOM-SD training algorithms are perfectly suited for parallel implementation. Hence, training times can be reduced multifold by processing the SOM or SOM-SD in parallel. A parallel implementation is performed most effectively by splitting the network into s pieces, where s is the number of processes (slaves or threads) running in parallel. Each process is given the entire training data set, and is assigned to one of the network pieces. This approach reduces the communication overhead during training to an absolute minimum. At each iteration, the slave processes report the local best matching neuron to the master process. The master process then determines the global best match and sends this information back to the slaves, which in turn update their part of the network. At the end of

the training session, the slaves send back their portion of the trained map. The master collects those pieces to rebuild the entire map.

4.3.6 Similarities with other methods

Structural elements of an input graph (the nodes of the graph) are given to the network as vectors of constant dimension. These input vectors are mapped onto codebook entries that are of constant dimension and are located in a fixed dimensional network. Hence, an input graph and its sub-structures are encoded and represented internally as fixed size vectors. Surprisingly, this is a common feature that is shared with all known neural network models which are capable of encoding graph structures. For example, the LRAAM model [81] encodes and represents a labelled graph and all its sub-structures by a single hidden layer, where the number of hidden neurons is constant. Hence, LRAAM represents structural information by means of a fixed size vector in a way that is more restrictive than with SOM-SD. Two other architectures known as Recursive Multi-Layer Perceptron (RMLP) networks and Recursive Cascade Correlation (RCC) networks (see Chapter 6 and Chapter 7 in this thesis) also encode input graphs recursively by processing terminal nodes first and proceed towards the root of the graph. Each node, and hence each sub-structure, is given to the network as a fixed size representation in a manner similar to SOM-SD. Graphs, and sub-structures are then encoded and represented by a layer of so-called state neurons or a layer of output neurons. These layers consist of a fixed number of neurons and hence, once again graphical information is represented by means of fixed sized vectors. Both, RCC and RMLP are described in greater detail in Chapter 6 and Chapter 7.

Similar to Kohonen's original SOM, the SOM-SD algorithm does not optimize an energy function [59, 22]. The SOM-SD algorithm involves a trade-off between the accuracy of the quantization and the smoothness of the topological mapping, but there is no explicit combination of these two properties into an energy function. Hence SOM-SD is not simply an information-compression method like most other unsupervised learning networks. Neither does SOM-SD have a clear interpretation as a density estimation method. Convergence of Kohonen's SOM original algorithm is allegedly demonstrated by Yin and Allinson [102], but their proof assumes the neighbourhood size becomes zero, in which case the algorithm reduces to VQ and no longer has topological ordering properties ([59], p. 111). Thus, there is no definite answer to the question of what SOM and SOM-SD learns.

4.4 Experimental results for SOM-SD

This section reports the results of applying the unsupervised SOM for structured data (SOM-SD) on three benchmark problems. These three benchmark problems are described in detail in Appendix A. The benchmark problem used in this section is a composition of artificial learning tasks. The use of an artificial learning problem allows the evaluation and comparison of the model in a controlled manner. A real world learning problem is applied later in Chapter 8.

A brief description of the benchmark problem is as follows:

- The first data set is referred to as dataset-1. It consists of 500 DOAGs belonging to two different classes which are linearly separable [4]. Hence, this data set represents an easy learning problem.

[4]The classes are separable by the data label associated with one of the leaf nodes. Further details on this property are available in Appendix A.

- The second data set combines 1250 patterns from 3 different domains producing a set of 3750 graphs each for the training and test set respectively. The learning task is considered to be more difficult than that of dataset-1 since patterns belonging to different domains are to be distinguished where some graphs belonging to different classes may feature a similar structure. This dataset is referred to as dataset-2.
- The third data set features the same set of graphs as dataset-2 with the difference that 12 classes are defined over the graphs. In this data set, some patterns can only be distinguished by information provided by the data label, while other data require structural information in order to be distinguished.

During training, the class membership of the data is ignored. The information of class memberships is used during evaluation to allow the visualization of how the graphs were encoded.

The aim of this section is to illustrate the general behaviour of SOM-SD, and its ability to encode structural information e.g., directed acyclic graphs. Results from the application of SOM-SD to the first two data sets are described briefly. Results obtained from the third data set are presented in greater detail.

Unless stated otherwise, all tests described in this section use networks with hexagonal topology and a Gaussian neighbourhood function. The displayed results reflect typical results obtained from random initial states. Experiments conducted on particular learning parameters have been started from identical initial conditions, and results are generally shown in a single plot.

Results obtained in this section will serve as a benchmark against which other methods such as the supervised SOM-SD, recursive MLP (multilayer perceptron) and recursive CC (cascade correlation), and their associated extended models will be measured [5]. These comparisons are given in Chapter 5 and Chapter 9 respectively. In order to allow a comparison between the various models described in this thesis, we chose to select the classification result as a performance meassure. Hence, unless stated otherwise the performance of a model refers to the classification performance.

4.4.1 Results using dataset-1

This section describes results as obtained when training the SOM-SD on a basic policemen benchmark problem (Dataset 1) which consists of a collection of patterns belonging to two linearly separable classes[6]. Since the classes are separable by the data label associated with one of the leaf nodes SOM-SD has to demonstrate that it is capable of encoding information encoded in data labels. The structure of the data does not contribute to the class definition. The information which distinguished the classes is whether the policeman has a raised left arm or a lowered left arm. Training is performed in an unsupervised fashion, i.e., the class information is unknown. However, in order to evaluate the model's performance, we use the class information during the evaluation process.

For the experiments we choose a neural network featuring roughly 1/3 as many neurons as nodes in the data set. As a result, a neural network featuring 45 neurons horizontally, and 40 neurons vertically is generated. Codebook vectors associated with the neurons are initialized by random values from within a valid range of values

[5]The description of supervised SOM-SD, recursive multilayer perceptron, and recursive cascade correlation techniques will be considered in later chapters of this thesis.

[6]For details on the policeman benchmark, please see Appendix A

obtained by scanning the training set. The state of the initial network is as shown in Figure 4.6. Displayed in the Figure are the neurons that were activated by root nodes belonging to graphs in the training data set. The mapping of nodes other than root nodes is not shown to avoid cluttering. It can be observed that at this initial state, the data is distributed randomly across the map. Neurons involved in the mapping of multiple nodes belonging to a different class are made visible by overlapping symbols (i.e. a plus inside a diamond shaped symbol). Neurons not involved in the mapping of any node produce empty spaces in this figure. As can be observed, at this initial stage, most neurons are not involved in the mapping process. In fact, only 12.6% of neurons are involved in the mapping.

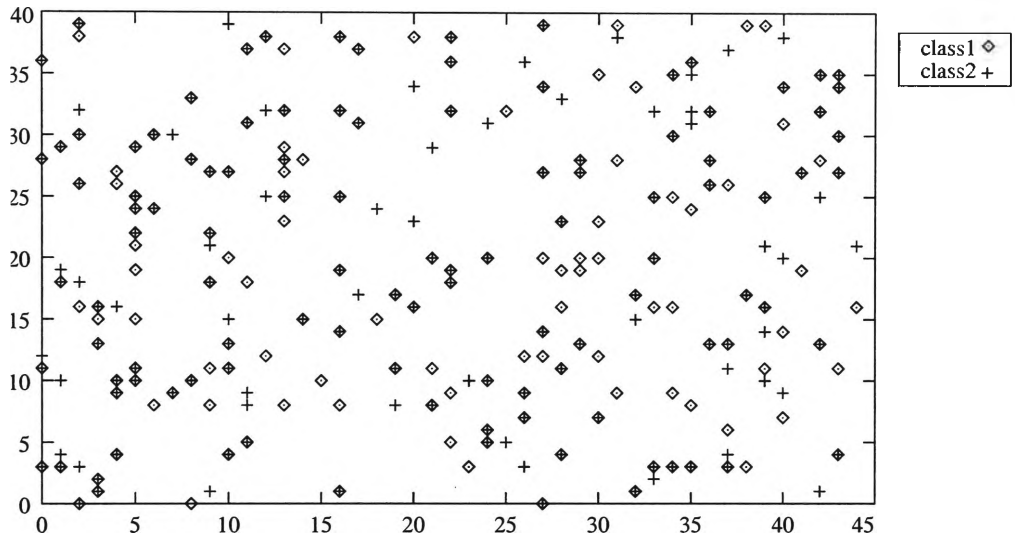


Figure 4.6: Mapping of the root nodes on a randomly initialized SOM-SD network of size 45×40. Diamond shaped symbols represent the location of neurons activated by nodes belonging to class 1 (policemen with lowered left arm). Neurons marked by pluses were activated by class 2 (policemen with raised left arm).

The neural network was then trained for just 50 iterations where the weighting value μ_2 is varied from within the range $[0 \ldots 140]\mu_1$. This illustrates the importance of structural information to the learning process. All other parameters were left at $\alpha(0) = 0.08$ and $\sigma = 40$. The result is as shown in Figure 4.7 which gives the network performance in relation to the weight value μ_2.

In Figure 4.7 it can be observed that the network performance is poor if the weight value μ_2 is chosen too small. This illustrates that structural information significantly assists the learning process. The network performance increases smoothly with increasing μ_2 reaching a performance level of 100% with $\mu_2 = 90\mu_1$. Increasing μ_2 to values beyond $90\mu_1$ produces unstable network performances. Performance levels drop significantly for μ_2 larger than $100\mu_1$.

Observations

- Firstly, this result has demonstrates that SOM-SD is able to encode graph structured information and to discriminate linearly separable data when appropriate values of μ_2 are chosen.
- Secondly, the importance of structural information whose influence is controlled by the weight value μ_2, to the learning process is demonstrated.

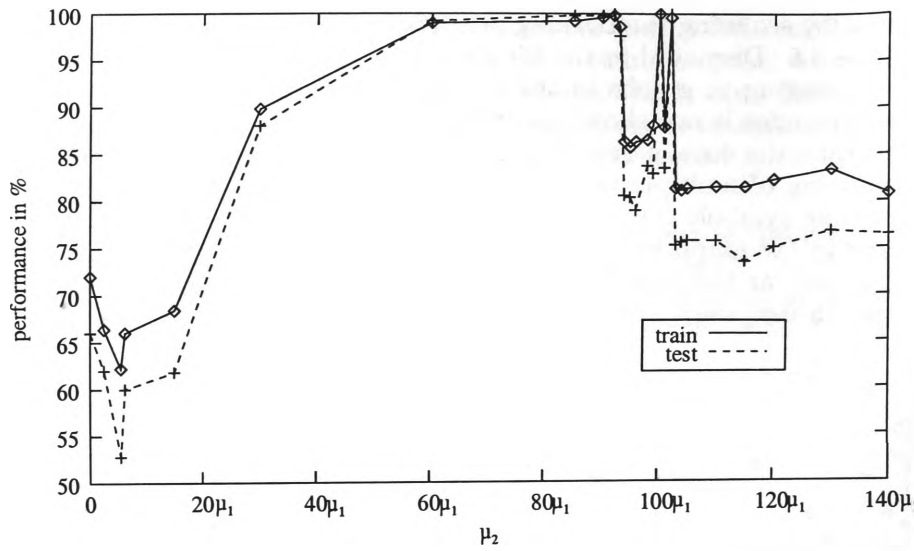


Figure 4.7: Training a network of size 45×40 for 50 iterations. Varying the weight value μ_2 produced networks with varying performance. Pluses and diamond shaped symbols indicate measuring points. Lines are a linear interpolation between measuring points.

As mentioned earlier, the definition of the two classes are independent of the underlying structure of the patterns. Nevertheless, due to the recursive nature of the learning process, it is important to pass the encoding of offsprings accurately to parent nodes. This is particularly true in the present case where vital information is presented in the leaf nodes. The optimal weight values suggested by Equation 4.5 are for the given task $\mu_2 = 6.22\mu_1$ but best results are obtained when choosing μ_2 approximately 10 times larger than this value. This demonstrates the importance of structural information to the learning process. It is to be noted that information provided through $\mathbf{c}$ is in fact knowledge about features encoded in the children nodes. Since the algorithm is recursive, this knowledge can only be passed on to the root of the graph in an accurate manner if the influence of $\mathbf{c}$ to the mapping process is sufficiently strong. Note also that the network was trained in an unsupervised fashion. Hence, the network was not given information about which features are involved to produce the two classes.

The following result will illustrate how the input graphs were encoded in the network. For this we mark neurons that were activated by a root node with a symbol. A plus shaped symbol is used to reflect neurons activated by root nodes belonging to class '1', diamond shaped symbols give the location of neurons involved in the mapping of root nodes that belong to class '2'. The result is given in Figure 4.8.

It is observed that during training, the mapping of root nodes has drifted to the right and formed clusters. The two clusters that are clearly separated are encircled in Figure 4.8. Apparently, these clusters were not formed by the two classes defined over the data set but by some other means. Root nodes belonging to the two classes have formed sub-clusters within the main clusters. From Appendix A we know that dataset-1 consists of 500 graphs where a graph can only have one of two possible structures: (1) one structure features a root node with two offsprings, (2) another structure has only one offspring. The graphs are identical in structure otherwise. An example of such pattern is given in Figure 4.9.

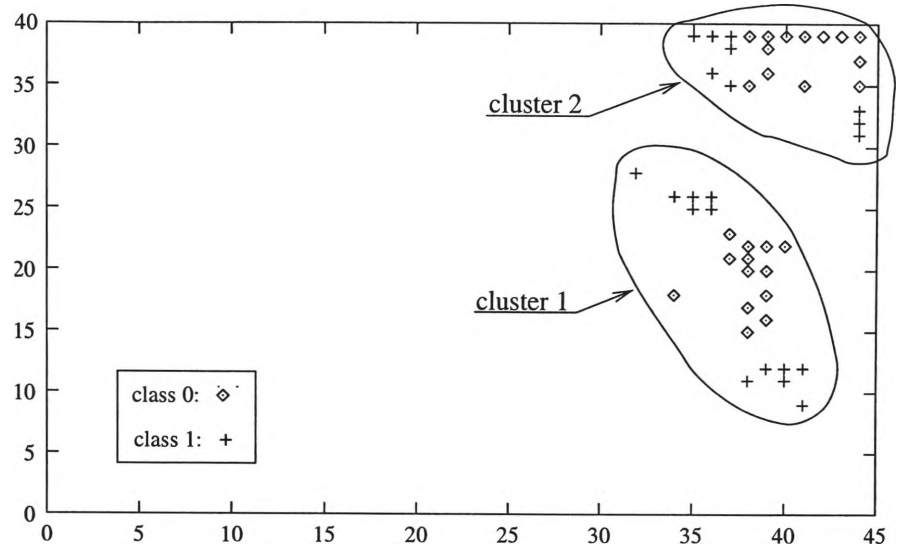


Figure 4.8: The mapping of root nodes after training a network of size 45×40 for 50 iterations with $\alpha(0)$ set to 0.08 and μ_2 set to $95\mu_1$.

The two classes in this set are formed by information presented by the data label attached to one of the leaf nodes of the graph. The structure does not contribute to the class definition. Investigating how the clusters in Figure 4.8 were formed, we found that all patterns represented in cluster 1 featured a root node with two offsprings, whereas in cluster 2 we found only patterns whose root node had a single offspring. Since the network was trained in an unsupervised fashion, it appears that SOM-SD considered the structural difference between the patterns as more significant than the difference presented by the data label. This property can be attributed to the influence of μ_2 which was set to 90 times larger than μ_1, and hence weighting structural information considerably stronger than the data label. In addition, a good network performance was only observed with such large μ_2 even so the class memberships as considered during the evaluation do not have a dependency on structural information but on information provided by a data label.

The reason for such behaviour becomes clear when considering that geographical location of the mapping of offsprings is passed on to the parent node. Since this process is recursive, it needs to be assured that this information is passed on correctly as the algorithm proceeds towards the root of the graph. The accuracy with which vital information about offsprings is passed on is controlled by μ_2. In the present case, vital information is available in the leaf nodes (i.e. the state of the left arm). A strong focus needs to be placed on μ_2 so as to ensure that these features have an effect on the mapping of the root node. However, choosing μ_2 too large causes SOM-SD to neglect information provided by data labels. Since it is the data label associated with a leaf node in dataset-1 which controls the class membership of patterns, there is a trade-off in the choice of μ_2. It is clear that the choice of μ_2 is application dependent. In the case of dataset-1, the best choice for μ_2 is $90\mu_1$.

From Figure 4.8 we find where root nodes are mapped, and how clusters among those nodes are built. In this figure, large areas are left blank. It can only be assumed that these areas are involved in the mapping of nodes other than root nodes. This is shown in Figure 4.10.

It is observed that the blank area which were visible in Figure 4.8 is indeed consumed

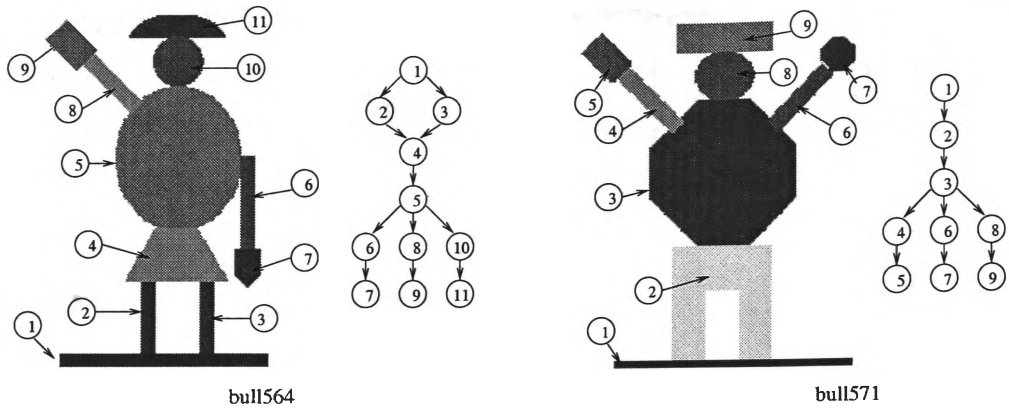


Figure 4.9: Two policemen figures and the graph representation. Dataset-1 consists of 500 graphs that have either one of these structures. Other structures are not present in dataset-1. Graphs in this data set differ otherwise only by the label attached to each node in the graph. Two classes are defined over the data set. Class 1 is a collection of graphs representing policemen with a raised left arm. All other graphs belong to class 2. The information about the location of the left arm is encoded in the data label associated with node 6 and node 7 (the nodes representing the left arm and hand) respectively.

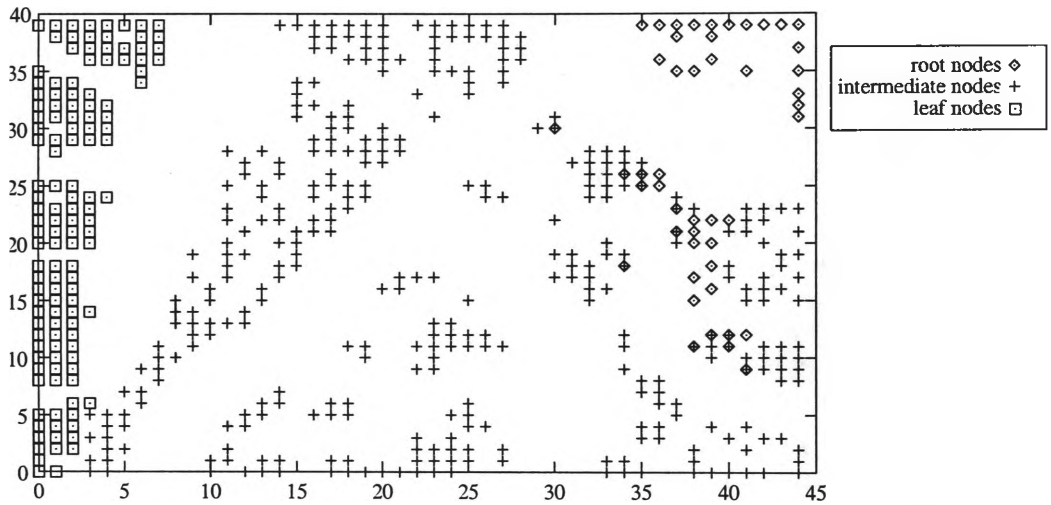


Figure 4.10: A network trained on dataset-1 for 50 iterations. The mapping of root nodes, intermediate nodes, and leaf nodes are shown in this Figure.

by neurons mapping nodes other than root nodes. Furthermore, it can be observed that intermediate nodes are mapped closer to areas where root nodes are found, and that leaf nodes are mapped furthest away from root nodes. This is an expected result because input vectors formed by leaf nodes differ greatly from input vectors formed by the root node since leaf nodes do not feature any offsprings.

A further observation is that relative large areas of the trained SOM-SD map are not marked by any symbols and hence, are areas that are not involved in the mapping of any node from the data set. In fact it is found that only 28.88% of all neurons are activated by a node from the training set. This demonstrates two aspects of the algorithm: firstly, the map leaves room to generalize "fuzzy" or inaccurate data.

This is confirmed by Figure 4.7 which shows that the generalization performance as obtained from the test set matches the performance achieved on the training set. Secondly, the map has demonstrated its ability to efficiently compress information. For example, the 5095 nodes given through dataset-1 are mapped onto just 512 neurons. This implies that SOM-SD may be efficient for data mining tasks; an issue which will be considered further in Section 4.4.3.

This experiment has demonstrated that SOM-SD is capable of encoding graph structured input data. The algorithm has demonstrated its ability to 'learn' the structure of a graph as it could be observed from the two clusters in Figure 4.8. The algorithm has furthermore demonstrated that information made available by the data label is learned. This was shown by the performance graph in Figure 4.7.

In the next section, the SOM-SD algorithm is applied to dataset-2. This data set does not necessarily represent a more difficult learning problem than dataset-1. However, SOM-SD will have to demonstrate its capability to encode a large amount of graphs, and its ability to handle graphs from different domains which may share common features.

4.4.2 Results using dataset-2

This section discusses the behaviour of a SOM-SD network when applied to dataset-2. This data set consists of a total of 3750 graphs, or 29864 nodes, from three different domains where some graphs from one domain can feature a similar graph or same structure as some graphs from another domain. Hence, the network needs to demonstrate its ability to cluster input data according to data label associated with each node in the graph. Training is performed in an unsupervised fashion, i.e., the class information is not available to the training algorithm. However, we encode the plots according to the original class membership of the data. The visualization of these classes gives implications on how well the SOM-SD network encodes structure and the data label.

We trained a network which had about 1/16 as many neurons as nodes in the data set. For the given learning problem, we generated a 45×40 output map, and initialized it with random values.

For this experiment we made a more exhausting search for suitable values of μ_2 and the number of training iterations. Weight values were chosen from within the range $[0 \ldots 200]\mu_1$ and the number of training iterations ranged from just 1 to 350. The initial learning parameter $\alpha(0)$ was set to 0.08, the initial neighbourhood spread σ was 40. During training, the learning rate gradually decreased to zero while the neighbourhood radius decreased to 1. The result shown in Figure 4.11 gives the network performance depending on training iterations and μ_2.

From this figure, we find that the network performance is always better than 90% if trained for more than 100 iterations independent of the value of μ_2. The best performance is achieved when training for 300 iterations or more with μ_2 set to values between $1\mu_1$ and $2\mu_1$. This shows that this learning task requires considerably less focus on structural information when compared to the results obtained from dataset-1.

The network trained for 300 iterations with $\mu_2 = 2\mu_1$ is used for further evaluations. The mapping of root nodes on a network trained with these parameters is shown in Figure 4.12.

It can be observed that during training, the mapping of root nodes has drifted into clusters, and that the SOM-SD has found an ordering that discriminates the

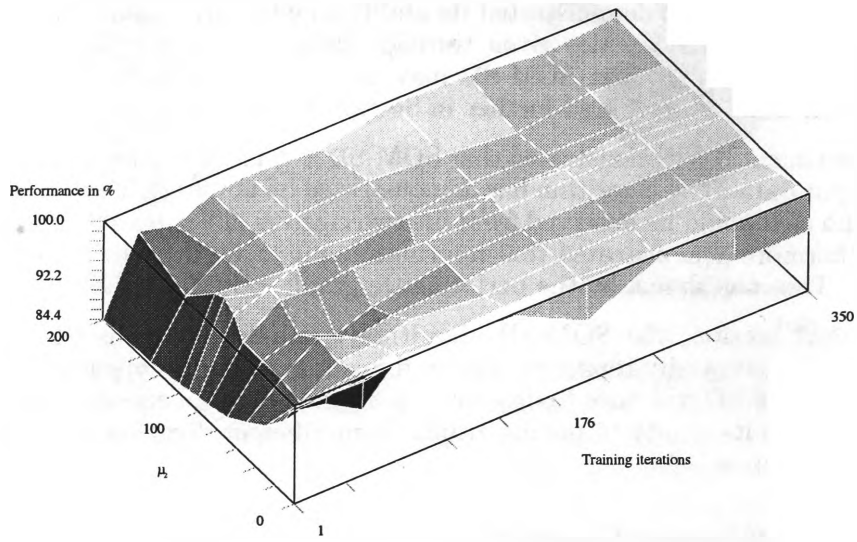


Figure 4.11: Network performance obtained when training the network with μ_2 as indicated for a given number of iterations. Intersection points of the grid give the actual sample points. Lines are a linear interpolation between two sample points, and the surface is an linear interpolation of its four corners. A white surface indicates a performance level of 100%. Darker areas represent configurations with poorer network performance.

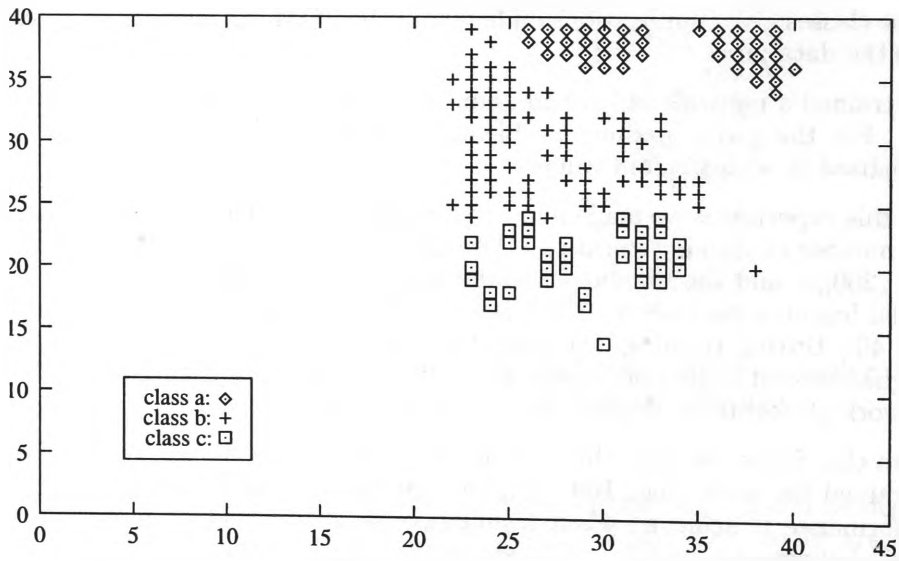


Figure 4.12: Mapping of the root nodes after training the network for a total of 300 iterations.

classes 'a' (policemen), 'b' (ships), and 'c' (houses). This demonstrates an important property: The SOM-SD network is well capable of encoding structural information.

From Figure 4.12 we can only assume that the empty space on the map is consumed

by all other except the root nodes. This assumption can be confirmed by considering Figure 4.13.

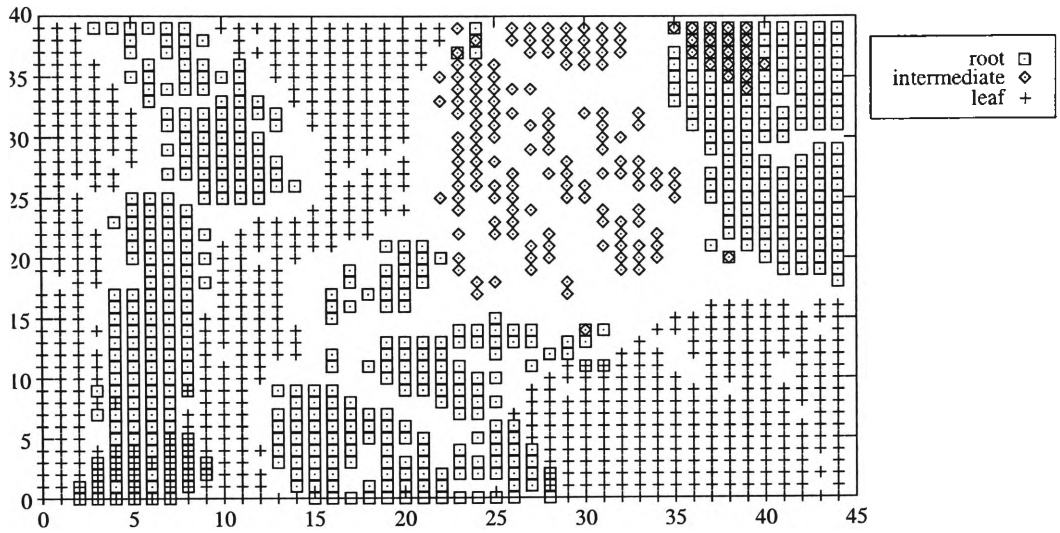


Figure 4.13: Neurons as activated by nodes from the training set. Boxes represent neurons activated by root nodes, the diamonds are neurons activated by intermediate nodes. Codebook entries that matched leaf nodes best are shown by plusses.

In 4.13 we display neurons whose codebook vector best matched any node from the input data set. Neurons are coded depending on the type of nodes. The diamond shaped symbols indicate neurons that were activated by the root nodes independent of the class to which they belong. Plus symbols give the location of neurons that were associated with leaf nodes, and the squares are neurons activated by nodes other than leaf or root nodes.

Figure 4.13 shows that the network is mapping nodes according to their location within the data graph. When comparing this figure with Figure 4.10 from dataset-1 we find that there are now considerably less blank areas. This is the result of training the network on a data set that is approximately 2.5 times larger than dataset-1. In fact, the network shown in Figure 4.13 displays just 27.89% neurons that were not activated by any node in the training set. Nevertheless, the 29864 nodes found a mapping on just 1298 active neurons. Hence, the compression is even greater than the one observed with dataset-1. Furthermore, in this figure (and in most other test runs) we did not find many neurons that were activated by two different types of nodes (e.g. intermediate and root node).

This experiment has demonstrated that SOM-SD is capable of dealing with large data sets even if patterns are retrieved from different domains. In this case, SOM-SD is able to find a discrimination between patterns from a different domain if the weight value μ_2 is chosen appropriately. In the present case, μ_2 is roughly twice the value of μ_1, and hence, considerably smaller than the μ_2 that was chosen for dataset-1. This demonstrates that the weight values μ_1 and μ_2 are an effective mechanism for controlling the focus of the training algorithm on either the data label component or the structural component of input vectors.

The aim of the following section is to evaluate the SOM-SD in greater detail. For this experiment, dataset-3 is used which features an identical set of graphs as those found in dataset-2. The only difference is that 12 classes are defined over this third data set. While SOM-SD is trained in an unsupervised fashion, it has to demonstrate

that it is able to map patterns in a range of conditions. This is most effectively evaluated by utilizing class membership information during the evaluation process.

4.4.3 Results using dataset-3

This section trains networks on dataset-3 where class membership information was not available during training. However, to allow the evaluation of the network performance, we labelled the neurons by a symbolic class label after training the network. The visualization of these classes gives indications on how well the network encodes the set of graph structures and their associated data labels.

Here we present a network for with about one third as many neurons as nodes in the data set. We generated a 114×87 output map, and initialized it with random values. Figure 4.14 shows the state of the initial network.

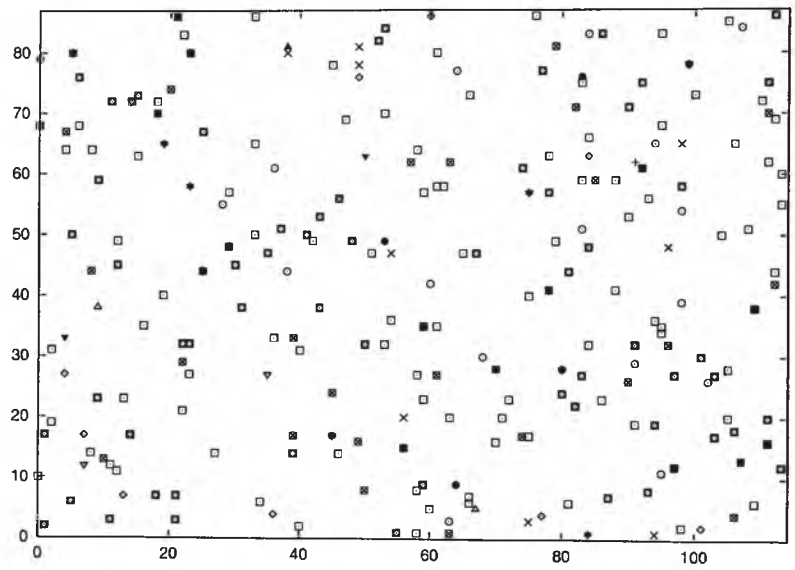


Figure 4.14: Mapping of the root nodes on a randomly initialized network of size $114 \times 87 \widehat{=} 9,918$ neurons (ratio 1.31:1). To avoid the cluttering of symbols, the mapping of nodes other than root nodes is not shown here.

The locations of neurons that were activated by root nodes only are displayed. It can be observed that, at this initial state, the data is distributed randomly across the map. Note that there are less neurons marked than the number of root nodes in the data set. This is because some neurons were activated by several root nodes. In the case where root nodes belonging to different classes activating the same neuron, this is made visible through overlapping symbols (e.g., a cross within a square).

In a first attempt, the network was trained for a total of 350 iterations. The initial learning parameter $\alpha(0)$ was set to 0.08, the initial neighbourhood spread σ was 60. During training, in order to assure convergence, the learning rate gradually decreased to zero while the neighbourhood radius decreased linearly to 1. The input vector components in $\mathbf{c}$ and $\mathbf{l}$ were weighted by $\mu_2 = 1.9\mu_1$ [7] . The resulting mapping of the root nodes after training is shown in Figure 4.15. In Figure 4.15, a unique symbol is used to indicate the location of neurons that were activated by

[7] According to Equation 4.5 this value pair balances the vector components in $\mathbf{x}$ best.

root nodes belonging to a particular class. The property and description of the 12 classes is given in Appendix A and is summarized by Table A.2.1.

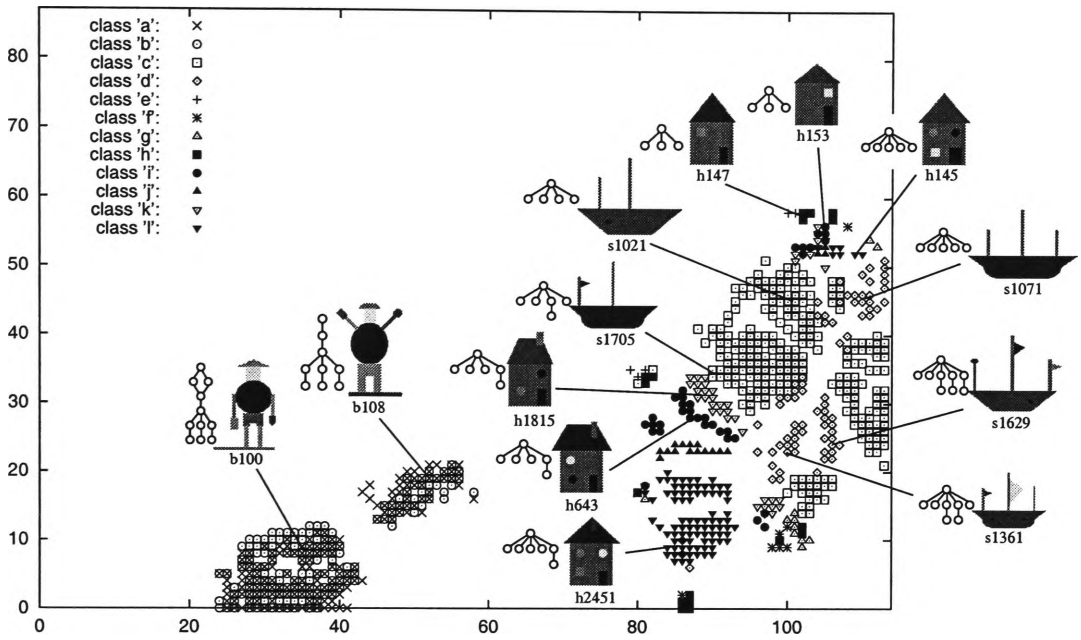


Figure 4.15: Mapping of the root nodes after training a network of size 114×87 for a total of 350 iterations. A typical sample has been retrieved for many sub-clusters and are displayed accordingly. The graph representation of a sample is displayed to its left, and its identity is plotted underneath.

It can be observed that during the training process, the mapping of root nodes has drifted into clusters, and that the network has found an ordering that discriminates the classes "house", "ship", and "policeman", i.e., the network is able to encode structural information. A further observation is that policeman patterns are represented in two distinct clusters but not according to the two policemen classes defined earlier. Upon closer inspection we found that the cluster in the upper right hand corner is formed by all patterns featuring policemen wearing long pants (no legs visible) such as shown in pattern b108, whereas the other cluster next to it holds all policemen that show two legs (wear a skirt or short pants) such as the pattern identified as b100. The difference between these two instances is that graphs obtained from policemen wearing long pants have a root node with a single offspring, where the single offspring is the long pants. All other graphs featured two offsprings, one for each visible leg. Hence, the network seems to have difficulties to finely encode the information given in the data label. For example, policemen with a raised arm and policemen with a lowered arm, which can only be discriminated by looking at the label attached to the node representing the arm are not mapped onto two distinct clusters. This observation mirrors the findings made earlier with dataset-1 and dataset-2 and is an issue which will be considered further later in this section.

Another interesting observation is that patterns belonging to the domain "ship" are mapped onto a relatively large region whereas the patterns representing "house" are mapped into two regions, one of which is just above the region that mapped the patterns representing "ship", the other is below. We found that the patterns of "house" mapped into the lower region featured a chimney so that the associated graph is of depth two. Patterns of "house" mapped into the upper region did not

feature a chimney so that the associated graphs were of depth one. Interestingly, the same was true for "ship". Patterns of "ship" mapped closer to the upper end of the region did not feature flags so that the graphs representing those pattern of "ship" were also of depth one. Patterns of "ship" mapped closer to the lower end of the region always featured some flags and thus their associated graphs were of depth 2. Similarly, we found that patterns of "house" and "ship" were mapped closer together featuring the same out-degree such as illustrated by patterns h1815 and s1705, or patterns s1071 and h145.

In addition, it is found that most classes were mapped onto distinct clusters with little overlap between the clusters. Since some clusters represented patterns with similar structures such as the clusters associated with patterns h1815 and h643, the network has demonstrated that it is able to distinguish graphs by considering information stored in the data labels.

There is very little overlap between clusters formed by different classes with the exception for the graphs produced from "policemen" patterns. This finding suggests that it is probably more advisable to train the network with a modified set of weights μ_i so that the focus on the data label is strengthened. Experiments shown later in this section will find that the opposite is true. Nevertheless, the SOM-SD has produced an impressive first results given that the general performance of this network was near 92.03% on the training data, and 91.52% on the validation data set.

From Figure 4.15 we can assume that the empty space on the map is consumed by all other but root nodes. This assumption can be confirmed by considering Figure 4.16. In this figure, we display neurons whose codebook vector best matched any node from the input data set. Activated neurons are marked depending on the type of node that activated the neuron. The plus shaped symbols are the neurons that were activated by the root nodes independently of the class to which they belong. Square shaped symbols indicate neurons that were associated with intermediate nodes, and neurons marked by crosses were activated by at least one leaf node.

It can be observed that there are areas for which neurons were not activated by any node in the data set. It is found that 3894 neurons (39.26%) are unused. We will find later, that the large number of unused neurons contribute to a good generalization performance. Of the 6025 neurons activated, 785 neurons are used by root nodes, 2022 by intermediate nodes, and 3227 by leaf nodes. 10 neurons were activated by at least one root and one intermediate node (overlap). There was no overlap between intermediate and leaf nodes or between leaf nodes and roots. In fact, in almost all other experiments we found only few neurons that were activated by two different types of nodes (e.g., intermediate and root nodes).

At this point, we know where the root and leaf nodes are located and that root nodes are ordered mainly according to a structural criterion. But what about intermediate nodes? One could expect that intermediate nodes located closer to a leaf node of a graph are also mapped to nearby codebook entries representing leaf nodes. Also, intermediate nodes nearer to the root may be expected to be mapped near codebook entries representing root nodes. Hence, one would assume that the mapping of intermediate nodes drifts from clusters representing leaf nodes towards clusters holding root nodes depending on the relative position of the intermediate node within the graph. This assumption is confirmed through the sequence of plots shown in Figure 4.17. This plot shows nodes obtained from policeman images according to their relative position within the graph. We considered plotting nodes associated with "policeman" patterns since they featured the deepest graph structure, and hence are best suited to demonstrate the ordering of nodes on the map.

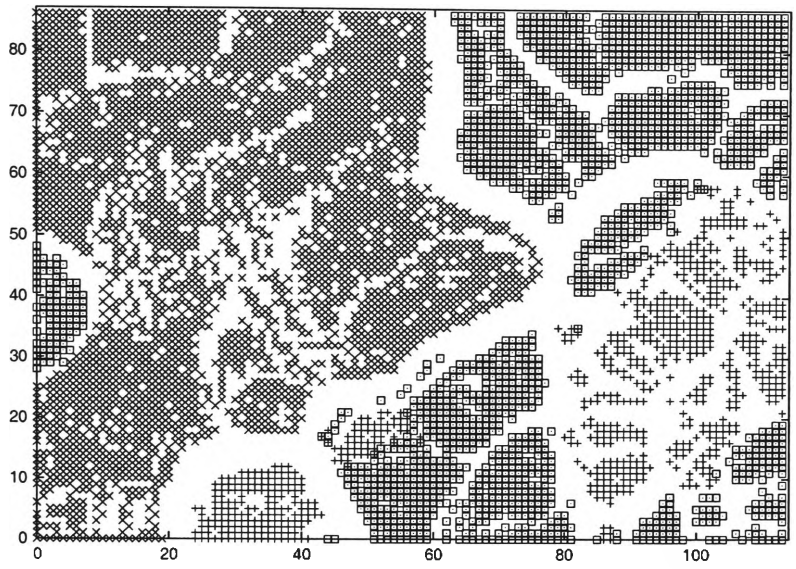


Figure 4.16: Neurons activated by nodes from the training set. Plus shaped symbols mark the location of neurons that represent root nodes, the squares are neurons activated by intermediate nodes. Codebook entries that matched leaf nodes best are shown as crosses.

Nodes from other domains are not plotted to avoid cluttering of the data displayed. In Figure 4.17 it is observed that the mapping of nodes drifts starting from the left edge for leaf nodes over to the upper edge and right edge for intermediate nodes to the lower edge for root nodes. It is also observed that the clusters often are not connected. This is because the space between the clusters is consumed by neurons which were activated by data from other classes which are not shown in this figure. We attribute this observation to a particular property of the data: Leaf nodes do not feature any offsprings and hence, the vector component $\mathbf{c}$ differs most significantly to nodes having a large number of offsprings which is often the property of a root node.

The network displayed in Figure 4.15 showed that the network was not always able to build distinct clusters according to information stored in the data label. The reason for this can be twofold. One reason might be that the network has been chosen too small (only $9,918$ neurons compared with $29,808$ different input nodes) so that there was not enough space to map the data according to both the structure and the data label. Another reason might be the wrong choice of weighting parameters μ_i, $i = 1, 2$. The following two experiments are designed to bring more light to illuminate this behaviour. First, networks of different sizes were trained. The initial neighbourhood radius was set to the maximum extension of a map while all other learning parameters remained unchanged.

The result shown in Figure 4.18 is that the network performance can be increased significantly if more neurons are used. However, a performance level of 100% is never achieved. In addition, the level of performance increase is getting weaker as the network grows. As a result, a network of size 114×87 seems an appropriate choice when considering that the network performance increases only slightly for networks larger than this but at the cost of increased computational requirements. Also, when choosing large networks, we lose the advantage of SOM to perform data compression on large amount of data. It is interesting to note that even for

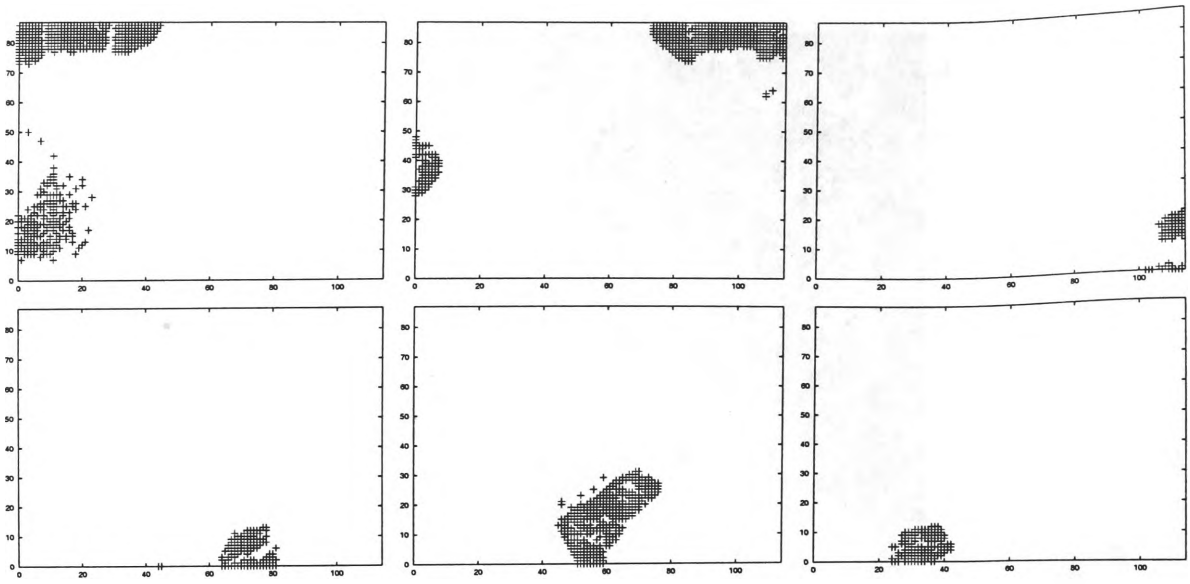


Figure 4.17: Nodes from "policeman" patterns sorted by the distance to the leaf nodes. The upper left plot shows neurons as activated by the leaf nodes (depth 0), the next plot to the right are nodes at depth 1, and so on. The lower right plot shows the root nodes located at depth 5. Note that the network has been trained on the entire data set and is the same as those shown in Figure 4.15. Nodes belonging to a class other than policemen are not displayed here.

small networks not all neurons are activated by nodes in the training or test data set. Only for extremely small networks that featured just one neuron for every 300 nodes in the data set that all neurons were utilized.

The second set of experiments is to determine the influence of the weighting parameters μ_1 and μ_2 on the network performance. For this, the ratio μ_1/μ_2 was varied within $[0; \infty]$. The neighbourhood spread σ was set to 114, and other training parameters are left the same as the initial experiment and the network size remained at 114×87. The result is as shown in Figure 4.19.

As stated earlier, the two weighting values balanced the vector components best for $\mu_2 = 1.9\mu_1$. However, best results are obtained when emphasizing the influence of structural information (μ_2) compared with that of the data label. The experiment shows that with μ_2 chosen 100 times larger than μ_1, the network performs best. This result is not surprising when considering that vital information is often given in nodes far away from the root node (i.e. only the data label associated with leaf nodes distinguishes between policemen with a raised or a lowered arm). Thus, it is essential, that information derived from a node's offsprings is passed to parent nodes accurately. This can be achieved by choosing large value for μ_2.

An additional finding was that when choosing the optimal configuration only 4234 neurons are activated by nodes in the training set. Hence, only 42.69% of neurons on the map are used as compared to 51.07% when having $\mu_2 = 1.9\mu_1$. This shows that the representation of the nodes is more compact, and explains why the generalization performance on the validation set is best.

This experiment has shown another very interesting result. Note that we trained a network with μ_2 set to zero. In this case the algorithm reduces itself to the standard SOM. Another network was trained with μ_1 set to zero in which case only structural

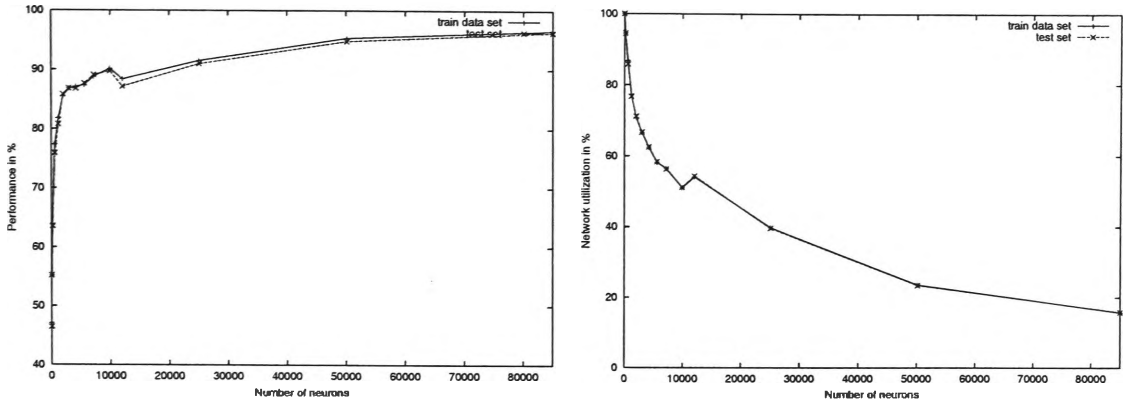


Figure 4.18: Performance of the SOM-SD when varying the size of the network. The left hand plot illustrates the overall performance, the right hand plot gives the amount of neurons activated by at least one node.

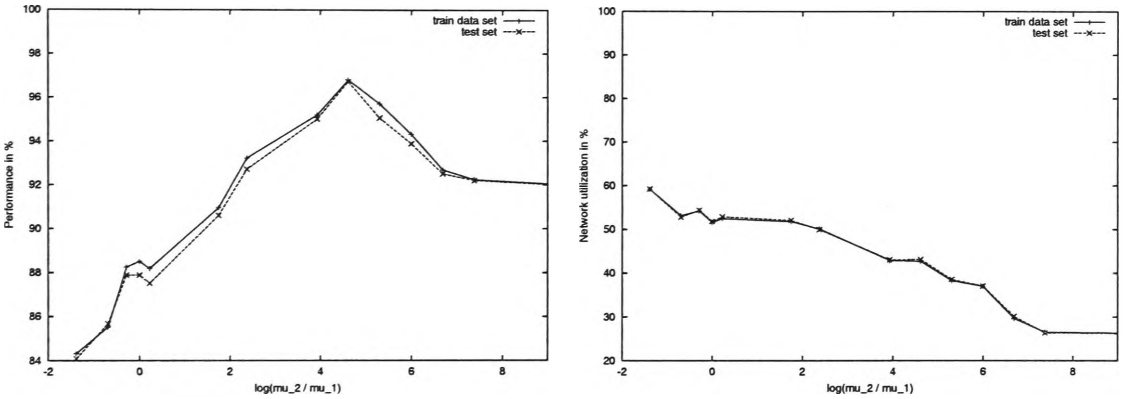


Figure 4.19: Performance of the SOM-SD when varying μ_0 and μ_1. The network performance is shown on the left; the right hand plot illustrates the network utilization in percent.

information is encoded. It was found, that information encoded in the data label contributes more strongly to a good network performance ($\sim$ 73% with $\mu_2 = 0$) than pure structural information ($\sim$ 67% with $\mu_1 = 0$). Also, when considering the network usage rates, it is found that data compression is growing with smaller μ_1/μ_2 ratio. With Figure 4.20 we illustrate the mapping of root nodes when choosing $\mu_2 = 0$ (left plot), and $\mu_1 = 0$ (right plot). With $\mu_2 = 0$, the network has particular difficulties in distinguishing the classes derived from within each domain. The three domains are well distinguished. This shows that classes within each domain can be distinguished only when considering the structural information as well. This result is not s surprise since the data labels in the root nodes are not too different from each other. In contrast, the right hand plot in Figure 4.20 shows the mapping of root nodes after training the network with $\mu_1 = 0$. Here the reasons for the network having difficulties in distinguishing the classes is caused by the sole consideration of the topology of the graphs as the graphs contain relatively few nodes and hence, the dataset does not show a great variability. This observation may pinpoint reasons for poor network performance. This shows that μ_2 controls the "focus" of the SOM-SD on features while μ_1 affects the "separation" of features. However, it has become evident that well performing SOM-SD network can only be achieved through the combination of both numerical information presented in the data label

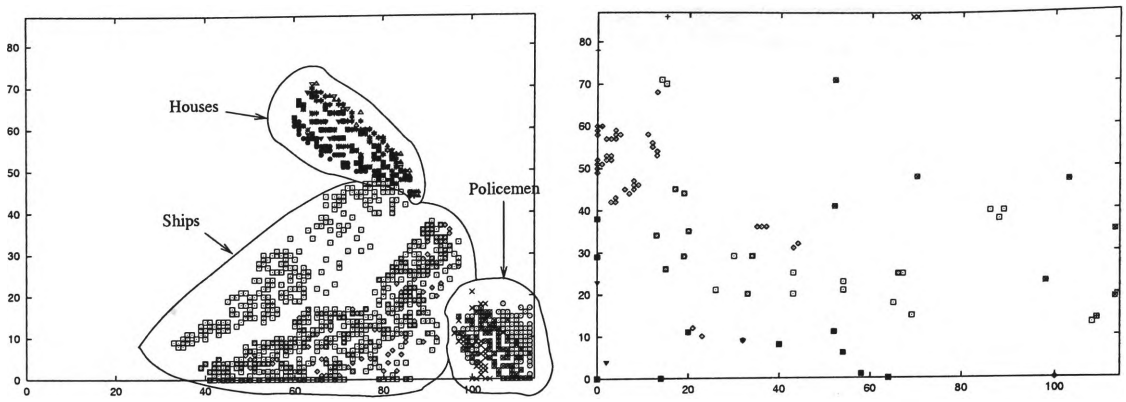


Figure 4.20: Neurons as activated by root nodes when setting $\mu_2 = 0$ (left), and $\mu_1 = 0$ (right).

and structural information respectively. However it is not possible to accurately predict the best combination of values for μ_1 and μ_2.

The mapping of nodes for the optimal set of μ_i is illustrated in Figure 4.21. The left hand plot shows the mapping of nodes depending on their class memberships. It shows that there is very little overlap between nodes from different classes. The interesting observation is that the training algorithm has placed a considerable focus on structural information even when structural information itself does not help to separate the classes, as shown in this experiment for the "policeman" classes. Here, the focus on the structure assures that vital information encoded in the data label associated with nodes away from a root node are passed back to the root node accurately. The plot on the right hand side shows the mapping of nodes depending on the type of the node.

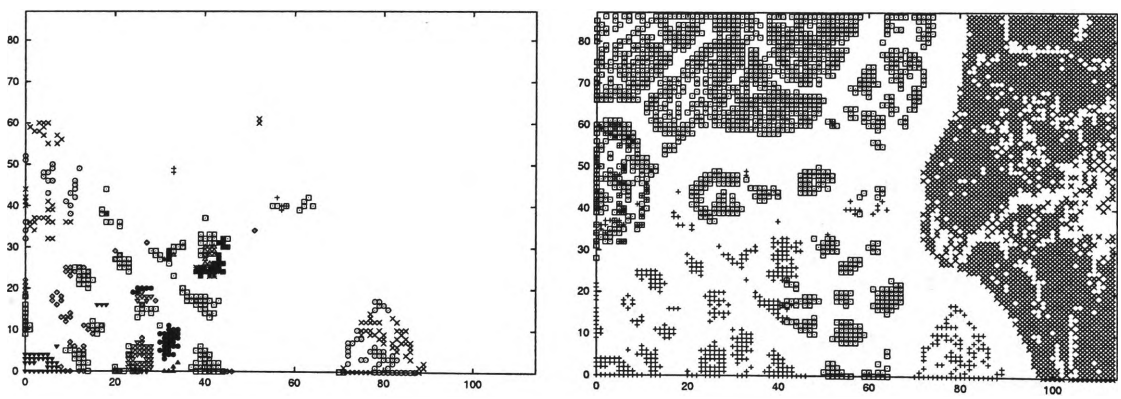


Figure 4.21: The mapping of nodes when $\mu_2 \approx 100\mu_1$. This setting produced a well performing network.

In this figure, a plus shaped symbol corresponds to a root node, squares represent the mapping of intermediate nodes, and crosses show the location on which leaf nodes are mapped. It can be observed that relatively large areas (48.73% of the map) are left blank. This resembles neurons which were not activated by any node in the training set, and contributes to a good generalization performance.

It has been demonstrated that SOM-SD's generalization performance is generally

very good in that the performance on validation data is typically less than 1% behind the performance achieved on the training data. This gives no indications as whether this is due to data of low complexity, the sheer size of the data set, or a strength of the SOM-SD model. The following experiment varies the size of the data set to give a better insight. The quantity of training data is gradually reduced until only 61 graphs (1/64 of the original set) are left. The network is then tested against the test set where no pattern in the training set is present in the validation set. Training parameters and the size of the network are the same as the first experiment described in this section with the exception of σ which was set to 114. The result is illustrated in Figure 4.22.

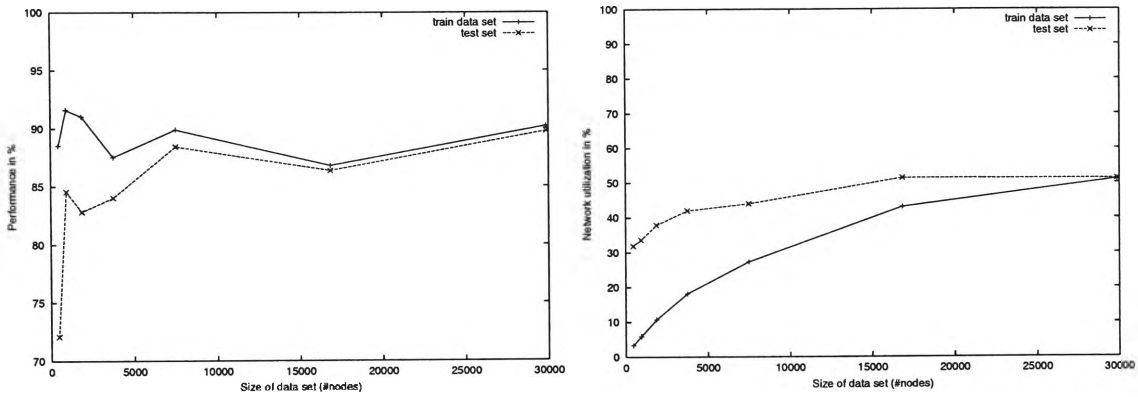


Figure 4.22: Network performance vs. size of the training set. The network performance is shown on the left hand plot. The right hand plot illustrates the amount of neurons activated by at least one node.

It is found that the network performance is always around the 90% mark on the training data independent of how many patterns are present during training. In contrast, generalization performance decreases as soon as the training set consists of less than $\sim$ 16000 nodes. Also illustrated is the network utilization rate where the amount of neurons actually activated by any node from the training (test) set is plotted against the size of the data set. Lesser number of neurons is activated as the size of the training set is reduced. Interestingly, network utilization rates also decrease for the test set when the training set gets smaller despite the test set remaining static in size. From this observation. it appears that a reduction of the training set size has an effect on SOM-SD's ability to generalize since fewer neurons are involved in the generalization process.

In the following, we consider the influence of the number of training iterations on the performance of a SOM-SD network. One would assume that due to the large size of the training data, only a relatively small number of iterations is required to achieve good results. To verify this, we trained some networks for just 50 iterations while others with as many as for 500 iterations. The result of these experiments is as shown in Figure 4.23.

It is found that the network performance increases steadily with the number of training iterations. However, training the network for more than 350 iterations did not help to improve the performance much further. Instead, the generalization performance was observed to decrease slightly which perhaps is caused by overfitting. Also shown in this figure is the quantization error as obtained at the last training iteration. The quantization error shown is the total sum of quantization errors as defined by Equation 4.7. It is observed that while the network performance is at

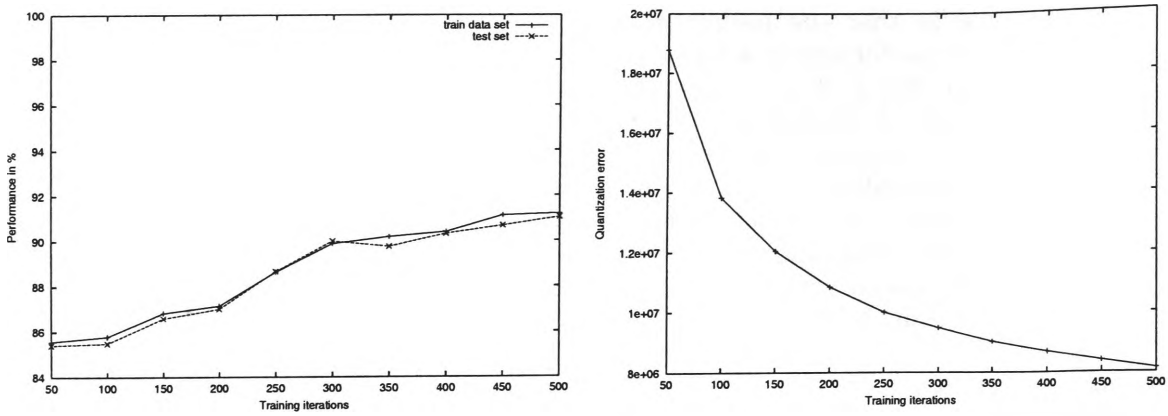


Figure 4.23: Network performance versus the total number of iterations trained. The left hand plot shows the network performance while the right hand plot gives the quantization error.

about 85% when training only 50 iterations, the quantization error is more than twice as large when trained for 500 iterations where the increase in performance is just about 6%. Thus, the quantization error does not necessarily reflect how the network performs on a given task.

So far, most experiments were based on an initial learning parameter set to 0.08 and a neighbourhood radius set to the maximum dimension of the SOM. During training, the learning parameter decreases linearly to zero while the neighbourhood radius decreases linearly to 1. Traditionally, the initial neighbourhood radius is chosen large so as to reduce the chances of falling into a local minima situation [59]. To verify whether this is true for SOM-SD, we conducted the following experiments. Networks were trained with different initial radius, then the network performance was evaluated. From the results displayed in Figure 4.24 it can be observed that SOM-SD's performance actually increases with smaller initial radius. This is in contrast to observations typically made on traditional SOM. An explanation to this behavior is that SOM-SD clusters input data into areas depending on the types of nodes (root, intermediate at different levels, leaf). This clustering has been observed to be established very early during training (within the first or second iteration). From then on, those clusters change their locations only little whereas major updates are happening only within each cluster. Thus, it is beneficial for the network to have a small neighbourhood function so that training can be focused on smaller areas of the network. Evidently, large radius does have a rather disturbing effect. This is supported by considering the plot showing the minimum quantization error as obtained after training. The quantization error actually increases with large initial radius.

The last parameter to be considered for experiments is the initial learning parameter. So far, all experiments were conducted based on an initial learning rate starting at 0.08 which decreased linearly with the training iterations down to zero. Kohonen described that in the standard SOM [57], too large and too small learning rates can lead to poor network performances. There is no clear guideline which suggests good learning rates to any given learning problem. Hence, the following experiments aim to find the optimal learning rate for the benchmark problem under consideration. Networks of size 114×87 were trained with a range of different initial learning parameters $\alpha(0)$. All other parameters were left the same as the first experiment described in this section. Results are illustrated in Figure 4.25.

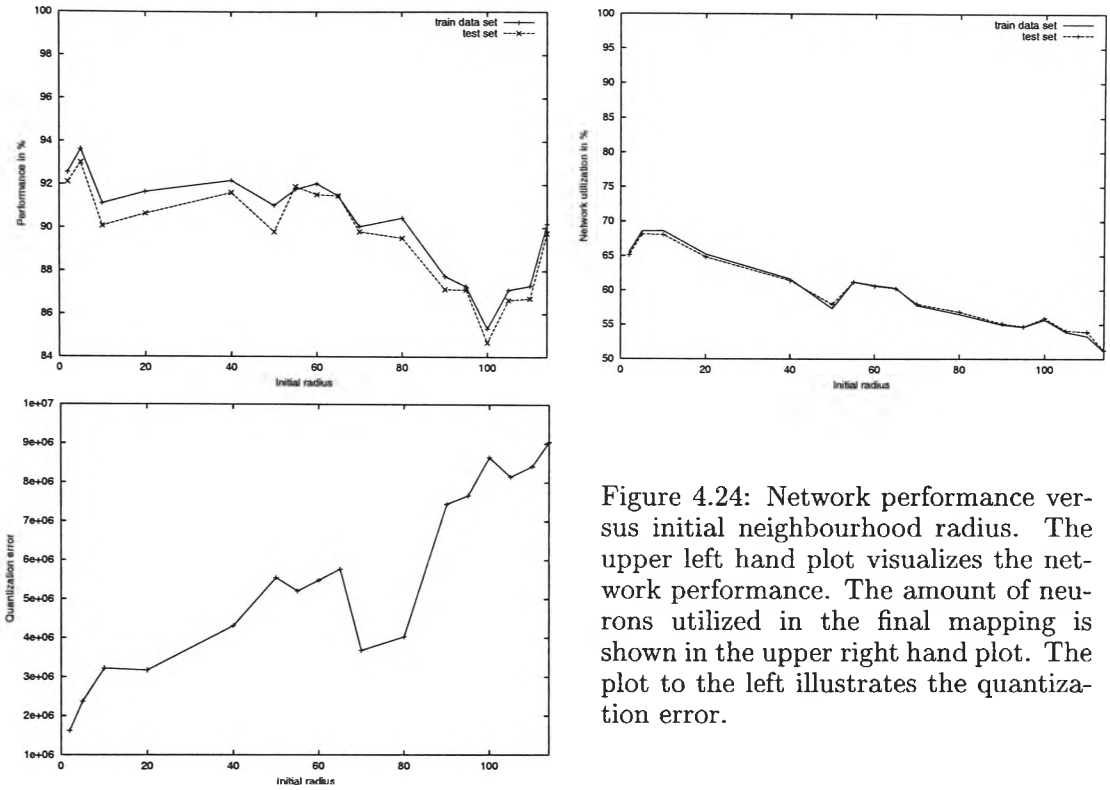


Figure 4.24: Network performance versus initial neighbourhood radius. The upper left hand plot visualizes the network performance. The amount of neurons utilized in the final mapping is shown in the upper right hand plot. The plot to the left illustrates the quantization error.

It is found, that the SOM-SD can be trained using a wide range of initial learning rates. Any initial learning rate larger than 0.32 and smaller than 1.5 can produce good results. Learning rates larger than 2 produced data overflow errors, rates smaller than 0.32 were insufficient to make the SOM-SD converge to high performance levels. The ability of this model to accept a large range of $\alpha(0)$ together with the observation that a local minimum situation was never observed in any of the 106 training sessions conducted on this experiment indicates that this model may be more robust than the original SOM. This finding is supported by [97] where a SOM-SD is applied to a real world learning task.

Summary of the experimental results:

By putting all results described in this section together, we find that for the given learning problem, and for a network of size 114×87 (number of neurons approximately 1/3 the number of nodes in the training set), that the best set of parameters is $\sigma \approx 60$, $\alpha(0) \in [0.32; 1.5]$, $\mu_2 \approx 100\mu_1$, and the number of training iterations is greater 450.

A network trained with this set of parameters produced the following results: 97.63% performance on the training set, 97.33% on the validation set. 54.19% network utilization on the training set, and 54.40% utilization on the test set. The quantization error was 90.9×10^6. This indeed is the best result; no better performance was observed by any other experiment conducted so far. Nevertheless, there is still an error of about 2.5%.

It was observed that the greatest contributors to the network error are patterns belonging to classes that are represented by a small number of samples. Also some "ship" patterns and "policeman" patterns contributed to the total error. It was

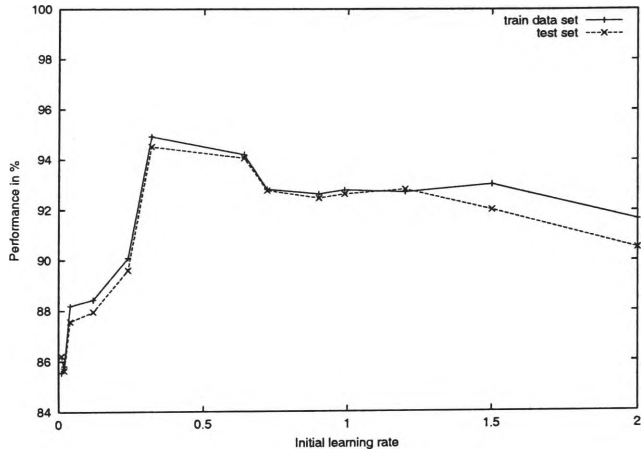


Figure 4.25: The plot gives an overview of the network performance dependent on the initial learning parameter.

found that these misclassifications were caused by the fact that the patterns actually shared an identical structural representation with some patterns from another class. Evidently, information provided by the data label was insufficient to produce a better mapping.

Note that an optimal set of training parameters can only be found through trial-and-error. The parameters suggested here are the best as far as the learning task provided with dataset-3 is concerned. Other learning problems may benefit from a different set of training parameters. However, by showing this set of parameters, it is hoped that this gives a good indication of what to expect in other practical problems. Perhaps this set of parameters could be used as an initial set of parameters for other practical problems. In addition, by considering various aspects of the implementation of SOM for this problem, it is observed that there are many aspects to the practical application of such techniques which need to be considered in practice.

4.4.4 Long term dependency problem

Gradient based learning techniques such as the back-propagation algorithm [73] suffer from a phenomenon known as *long term dependency problem* [5]. In the area of learning graph structured information, this implies that the network loses information about the nodes processed first when recursing through the graph. As a result, gradient based methods have a limited ability to learn deep structures.

SOM-SD training does not rely on gradient based information during training and hence does not suffer from long term dependency problem. As a result, using a self organizing map for the mapping of (potentially deep) graph structures does not impose a problem.

An experiment has been crafted to confirm this. A set consisting of two input graphs was generated such that the graphs were identical up to a certain depth. This depth was varied from zero (the root nodes differ) to 16. A network was trained for each of the cases. The result is as shown in Figure 4.26. It is observed that the depth of a graph structure has little or no influence on the number of iterations required to distinguish the graphs. Note that this experiment does not take the influence of data labels into consideration. The weight value μ_2 was chosen

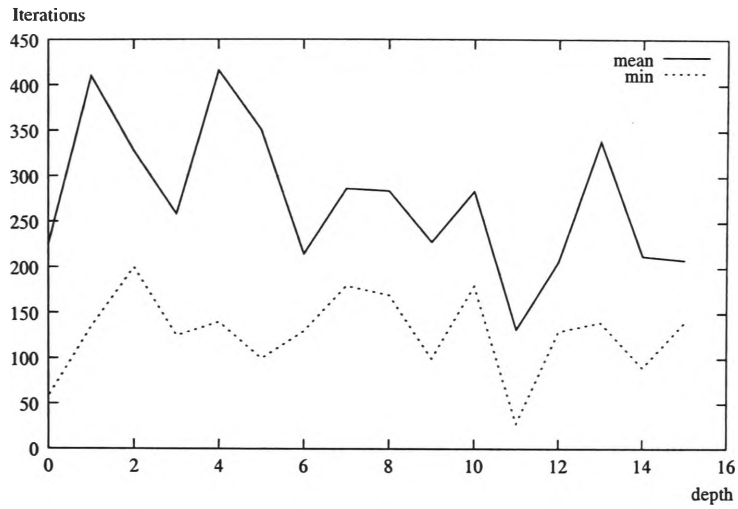


Figure 4.26: Learning deep structures. Shown in the diagram are the minimum and average number of iterations required to perfectly map two graphs that are different only at the indicated depth.

to be large ($\mu_2 = 100$) to assure that structural information is passed on to parent nodes accurately.

4.5 Conclusions

In this chapter, we described a new way of formulating unsupervised learning problem of graphs structured architectures using self organizing map techniques. The major innovation is to treat each leaf node as a self organizing map, and the coordinates of the winning neuron is transferred to the parent node. In other words, we assume that the information transferred between the children and the parent is conveyed by the coordinates of the winning neuron in the child node. The technique has been applied to three benchmark problems: dataset-1, dataset-2, and dataset-3. It is shown that the proposed technique is able to cluster data according to some prescribed classes.

An important finding is that the weight values μ_i provide an effective mechanism for controlling the influence of structural information to the learning process. Moreover, it was demonstrated that the quality of the mapping can benefit by including a strong emphasis on structural information.

It is trivial to observe that in the case where the maximum out-degree of the data is zero, the SOM-SD reduces to the standard SOM model. In addition, a SOM-SD model is capable of encoding data sequences by providing DOAGs with out-degree 1. Hence, the model introduced in this chapter extends the general form of SOM.

The SOM-SD suffers a number of deficiencies which also afflict the general SOM model. These include:

- There is no convergence proof of the SOM-SD training algorithm.
- We cannot formulate an energy function to derive the training algorithm.

In addition, the SOM-SD model has the following parameters which need to be

considered:

- The interaction between the structural information and the data labels. In this thesis, this relationship is embodied in the parameters μ_1 and μ_2 respectively. In general it is useful to assume that μ_2 in comparison to μ_1 is larger than suggested by equation Equation 4.5, though as shown in our experiments in this chapter, this rule may not hold in some cases.
- The number of neurons in the display space. In this thesis we have made the assumption that the number of neurons in the display space should be about $\frac{1}{3}$ of the total number of the nodes in the input graphs. In practice we do not know the number of nodes in the input graphs. Hence this parameter can be obtained by some trial and error process. One could, if desired, use evolutionary methods to determine this parameter.
- The shape of the initial neighbourhood whether it is a hexagonal shape or a rectangular shape.
- The initial learning rate.
- The stopping criterion, whether it should be the number of training steps, or whether the quantisation error should be smaller than a prescribed threshold.

It is noted that while we considered three datasets: dataset-1, dataset-2, and dataset-3, in effect, dataset-1 is logically a subset of dataset-2. Dataset-2 and dataset-3 are the same except that dataset-3 has 12 classes. Hence, from now on we will only consider dataset-3. Dataset-1 and dataset-2 are used only in this chapter, as one way to introduce the reader to the complexity of applying SOM-SD to graphical structures.

The SOM-SD model addressed in this section was trained in a unsupervised manner. In its present form, the model is unable to incorporate class information to assist the learning process in cases where such information exists. In the following chapter, the SOM-SD model is extended so as to allow supervised training.

Chapter 5

Supervised SOM for structured information

5.1 Introduction

This chapter presents an extension of the unsupervised SOM for graph structured data to incorporate a teacher signal when processing nodes for which a (symbolic) target label exists. The idea is to assign codebook vectors to the same class as the node that was mapped at this location. If several nodes from different classes are mapped at the same location, then the class of the codebook entry is obtained through majority voting. Training proceeds in a similar manner as the unsupervised case with the difference that codebook entries are *rejected* if they belong to a different class as the input vector. The advantage of this method is as follows: by adding supervision to the learning process, the network performance is improved. With the method proposed in this chapter, the system can handle problems which feature incomplete or missing class labels.

The structure of this chapter is as follows: in Section 5.2, some theoretical background for supervised self organising map will be introduced. In Section 5.3 a training algorithm for the supervised SOM-SD model will be given. In Section 5.4 some experimental results will be given in applying the supervised SOM-SD to a benchmark problem, viz., dataset-3. In Section 5.5 some conclusions will be drawn.

5.2 Theoretical background

Kohonen describes in [59] a mechanism for training a SOM supervised. The idea is to produce input vectors through the concatenation of the (numeric) target vector with the data label, and then to proceed training in the usual manner. However, the extension of this idea to the class of SOM-SD did not produce a well performing model. A supervised SOM-SD model which has been developed with considerably more success is presented in this chapter [1].

Given a self organizing map $\mathbf{M}$ with k neurons. Each neuron is associated with a codebook entry $\mathbf{m} \in \mathbb{R}^n$. The best matching neuron $\mathbf{m}_r$ for an input node $\mathbf{x}$ is obtained in a similar manner as the unsupervised case (see Chapter 4). The i-th

[1] As far as we are aware, the training algorithm presented in this chapter is novel.

element of the j-th codebook vector $\mathbf{m}_j$ is updated as follows:

$$\Delta m_{ij} = \begin{cases} -\epsilon\ \alpha(t)\ f(\Delta_{jr})h(x_i, m_{ij}) & \text{if } \mathbf{x} \text{ and } \mathbf{m}_j \text{ are in different classes.} \\ \alpha(t)\ f(\Delta_{jr})(x_i - m_{ij}) & \text{else.} \end{cases} \quad (5.1)$$

where $f(\Delta_{jr})$ is the neighbourhood function as before, α is the learning rate which decreases to zero in time, ϵ is a rejection rate which weights the influence of the rejection term $h(.)$. The purpose of the rejection term is to move $\mathbf{m}_j$ and its near neighbours away from $\mathbf{x}$. The effect is a reduction of the likelihood that an input node activates a codebook vector which is assigned to a foreign class in subsequent iterations. In order to improve efficiency, the rejection term should dictate stronger actions if a codebook entry is very similar to the input node and have a lesser influence on codebook vectors which are already very different to $\mathbf{x}$. Initially, we shall define the rejection term as follows:

$$h(x_i, m_{ij}) = \frac{-1}{(x_i - m_{ij})} \exp^{\frac{-(x_i - m_{ij})^2}{2\sigma_i^2}} \quad (5.2)$$

where σ_i is the standard deviation defined as follows:

$$\sigma_i = \sqrt{\frac{\sum_{l=1}^{q}(x_{li} - \overline{x_i})^2}{q}} \quad (5.3)$$

where q is the total number of nodes in the training set, and $\overline{x_i} = 1/q \sum_{l=1}^{q} x_{li}$.

σ_i can be approximated by a constant when assuming that the mapping of nodes is random. This approximation significantly reduces the computational complexity of Equation (5.2). Another way of reducing the computational demand is by imitating the behaviour of (Equation 5.2) by the following (Equation 5.4):

$$h(x_i, m_{ij}) = -\varepsilon\ \mathrm{sgn}(x_i - m_{ij}) \left(1 - \frac{o_i}{|x_i - m_{ij}|}\right), \quad i = 1, \ldots, n \quad (5.4)$$

where ε is a normalization value, sgn(.) is the signum function returning the sign of its argument, and o_i is the maximum absolute distance between any two nodes in the data set and is defined as $o_i = |\max(x_i - x_j)|, \quad i = 0, \ldots, q \quad j = 0, \ldots, q$.

The use of (Equation 5.4) is a preferred option because the rejection term is applied to the innermost loop (i.e. it is applied to every element of every codebook vector in the map and for every node of all graphs in the data set.). Hence, (Equation 5.4) facilitates the reduction of the computational demand of the rejection term.

However, when applied in practice, it has been found that the application of (Equation 5.2) and (Equation 5.4) produce unstable networks. This can be attributed to the influence of the term $1/(\mathbf{x} - \mathbf{m})$. If $\mathbf{x}$ and $\mathbf{m}$ are very similar [2] then the update step becomes very large. As a result, it has been found that network training can produce data overflow errors even when the rejection rate ϵ has been chosen to be

[2] $\mathbf{x} - \mathbf{m}$ cannot be zero because the updating rule in (Equation 5.1) allows this to happen only asymptotically. However, $\mathbf{x} - \mathbf{m}$ can become smaller than a computer's floating point precision and hence cause a division by zero problem.

very small. One way of overcoming this is to limit the rejection rate for the cases when $|\mathbf{x} - \mathbf{m}|$ reaches a certain threshold as suggested by (Equation 5.5)

$$h(x_i, m_{ij}) = \begin{cases} -\varepsilon \operatorname{sgn}(x_i - m_{ij}) \left(1 - \frac{o_i}{|x_i - m_{ij}|}\right) & \text{if } |x_i - m_{ij}| > \Delta. \\ \kappa & \text{else.} \end{cases} \tag{5.5}$$

where Δ is a threshold and κ is a small constant. While this approach eliminates the problem described above, it adds two more parameters to the system that need to be adjusted manually to achieve optimal performance. This can be a difficult and time consuming task since the choice of both Δ and κ are dependent on the learning task.

An alternative formulation of the rejection term in (Equation 5.6) which achieves similar results without having the burden of extra parameters is as follows:

$$h(x_i, m_{ij}) = \operatorname{sgn}(x_i - m_{ij})(o_i - |x_i - m_{ij}|) \tag{5.6}$$

It is useful to note that (Equation 5.6) is not an approximation of (Equation 5.2) but has the benefit of simplicity, and eliminates the possibility of runaway values or overflow errors. The goal of (Equation 5.6) is to reject codebook vectors belonging to a different class such that the distance between codebook entry and the input vector is the same distance as the maximum distance between the vectors in the data set. The disadvantage of this approach is that (Equation 5.6) does not take into account the distribution of the input data. Hence, the rejection rate may be unreasonably high for some of the vector elements. By using the standard deviation σ instead of the maximum distance value o we obtain an improved rejection term as defined by (Equation 5.7).

$$h(x_i, m_{ij}) = \operatorname{sgn}(x_i - m_{ij})(\sigma_i - |x_i - m_{ij}|) \tag{5.7}$$

The standard deviation is as defined by Equation 5.3. In practice, the application of (Equation 5.7) has produced the best results and will be used throughout this chapter.

5.3 Training algorithm

Training a supervised SOM-SD network is very similar to the training of an unsupervised SOM-SD network. The difference between the two approaches is that codebook vectors are assigned to a class during training, and the use of a rejection term for codebook entries that do not belong to the same class as the input vector. Training a SOM-SD network in a supervised fashion is as follows:

Step 1 A node j from the training set is chosen and presented to the network. When choosing a node special care has to be taken that the children of that node have already been processed. Hence, at the beginning of this process terminal nodes for each graph are processed first, the root node is considered last. Then, vector $\mathbf{x}_j$ is presented to the network [3]. A winning neuron r is obtained by finding the most similar codebook entry $\mathbf{m}_r$. This can be achieved, e.g., by using the Euclidean distance as follows:

[3]Note that $\mathbf{x}_j$ is a concatenation of $\mathbf{l}_j$ and $\mathbf{c}_j$ similar to the unsupervised case.

$$r = \arg\min_i ||(\mathbf{x}_j - \mathbf{m}_i)\mathbf{\Lambda}|| \tag{5.8}$$

where $\mathbf{\Lambda}$ is a $n \times n$ dimensional diagonal matrix. Its diagonal elements $\lambda_{11} \cdots \lambda_{pp}$ are assigned to be μ_1, all remaining diagonal elements are set to μ_2. The winning neuron is assigned to the same class as the node. Step 1 is repeated until all nodes in the training set have been considered exactly once. Codebook vectors that were activated by nodes belonging to different classes are assigned to the class which activated this neuron most frequently. Note that this step does not involve any training. It is solely used to initialize neurons with class labels. Neurons that were not activated by any node from the training set are assigned to the class *unknown*.

Step 2 A node j is chosen from the data set in the same way as step 1. Hence, when choosing a node special care has to be taken that the children of that node have already been processed. Then, vector $\mathbf{x}_j$ is presented to the network and the winning neuron r is obtained by finding the most similar codebook entry $\mathbf{m}_r$, e.g., by using Equation 5.8.

Step 3 After the best matching neuron has been found, the winning codebook vector and its neighbours are updated so that codebook entries belonging to the same class as the input vector are moved closer to vector $\mathbf{x}$, and those of other classes are moved away from it.

$$\Delta\mathbf{m}_i = \begin{cases} -\epsilon\alpha(t)\ f(\Delta_{jr})h(\mathbf{x}_j, \mathbf{m}_i) & \text{if } \mathbf{x} \text{ and } \mathbf{m} \text{ are in different classes.} \\ \alpha(t)\ f(\Delta_{jr})(\mathbf{x}_j - \mathbf{m}_i) & \text{else.} \end{cases} \tag{5.9}$$

Note that if either $\mathbf{m}$ or $\mathbf{x}$ belong to an unknown class that updating is performed as if they belong to the same class.

Step 4 The coordinates of the winning neuron are passed on to the parent node which in turn updates its vector $\mathbf{c}$ accordingly.

Step 5 Steps 1 to 4 are executed until a given number of training iterations are performed, or when the mapping precision has reached a given threshold.

It is not absolutely necessary to execute step 1 for every iteration. It is sufficient to initialize the class membership of codebook vectors when training starts. Then, a change of class membership can be detected during the execution of step 2. Hence, training can be performed by recursively running steps 2 to 4.

5.4 Experimental results for supervised SOM-SD

From Section 4.4.3 we know that the best result obtained when training SOM-SD in an unsupervised fashion was 96.8% (train), and 96.72% (test). This result shall serve as a benchmark for the experiments described in this chapter. Such a comparison is valid because we aimed our previous experiments on the finding of an optimal set of parameters which produce the best classification result [4]. In addition, from the experiments conducted by the unsupervised method, we already have a very good idea about useful values for the parameters μ_1, μ_2, σ, α, and the number

[4] One may argue that it is obvious that a supervised algorithm will outperform an unsupervised algorithm. However, this is not always true as we found e.g. in Kohonen's original idea [57] of padding class information to the input vector actually produces results which are worse than the model trained in an unsupervised mode.

of training iterations. In this section, we will use a network of size 114×87 and the same set of parameters that produced the best results for unsupervised trained network. Thus, for the first experiments we will use the following parameters: $\sigma = 60$, $\alpha(0) = 0.32$, $\mu_2 = 100\mu_1$, and the number of training iterations is 450. Supervised trained SOM-SD networks do have one additional learning parameter ϵ which controls the rejection rate for codebook entries that were found to represent a different class as the current input node. We restrict the set of experiments given in this section to the finding a value for ϵ that produces the best performing network. At the same time we will monitor the influence of ϵ on the network's performance. A more comprehensive set of experiments with supervised SOM-SD is given in Section 5.4.1.

Networks were trained with ϵ varying from 0 to 10. Choosing values for ϵ greater than 10 produced unstable networks in that data overflow errors started to occur. Figure 5.1 shows the behaviour of the network performance with the choice of ϵ:

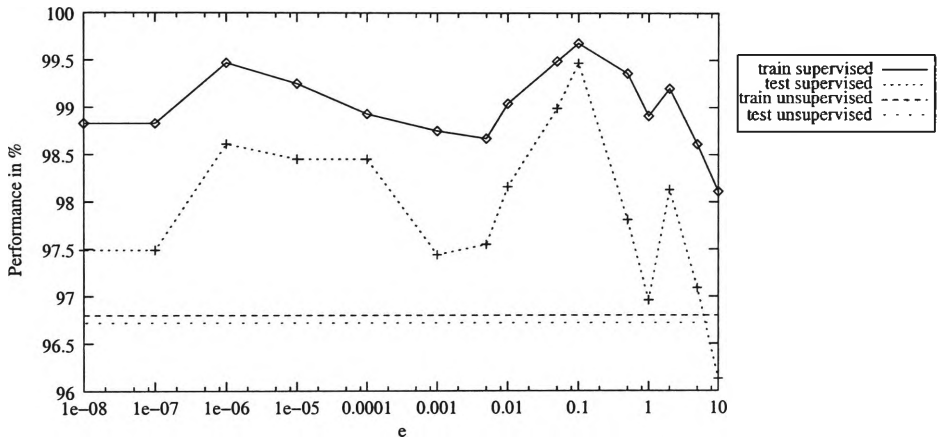


Figure 5.1: Training a network in a supervised fashion. Shown in the Figure is the network performance dependence on the rejection rate ϵ. Sample points are indicated by pluses and diamond shaped symbols. Lines are a linear interpolation between sample points. In comparison, the two horizontal lines are the best performance achieved by the unsupervised training method.

It is found that the network performance exceeds the performance of the unsupervised trained networks for any value of ϵ smaller than 10. Note that even networks trained with ϵ equal to zero outperform the unsupervised SOM-SD. In this case, codebook vectors that are associated with a class other than the current input node are not updated. This apparently has a rather beneficial effect on the performance. The performance is increased by setting ϵ to be a small positive value. Choosing ϵ to be 0.1 produced the best performing network. Here, the performance is 99.68% on the training set and 99.47% on the test set. It is noted that the results shown in Figure 5.1 are typically achieved by the proposed algorithm on this problem. Best and worst case performances are not shown because the training time requirements of the learning task did not allow us to conduct a sufficiently large number of experiments for each instance of ϵ using different initial network conditions.

The mapping of root nodes after having trained a network supervised with $\epsilon = 0.1$ is shown in Figure 5.2. It can be observed that the general arrangement of the root nodes does not significantly differ compared with the mapping obtained when training in an unsupervised fashion. The difference is mainly a better ordering of patterns within sub-clusters so that nodes belonging to the same class form a

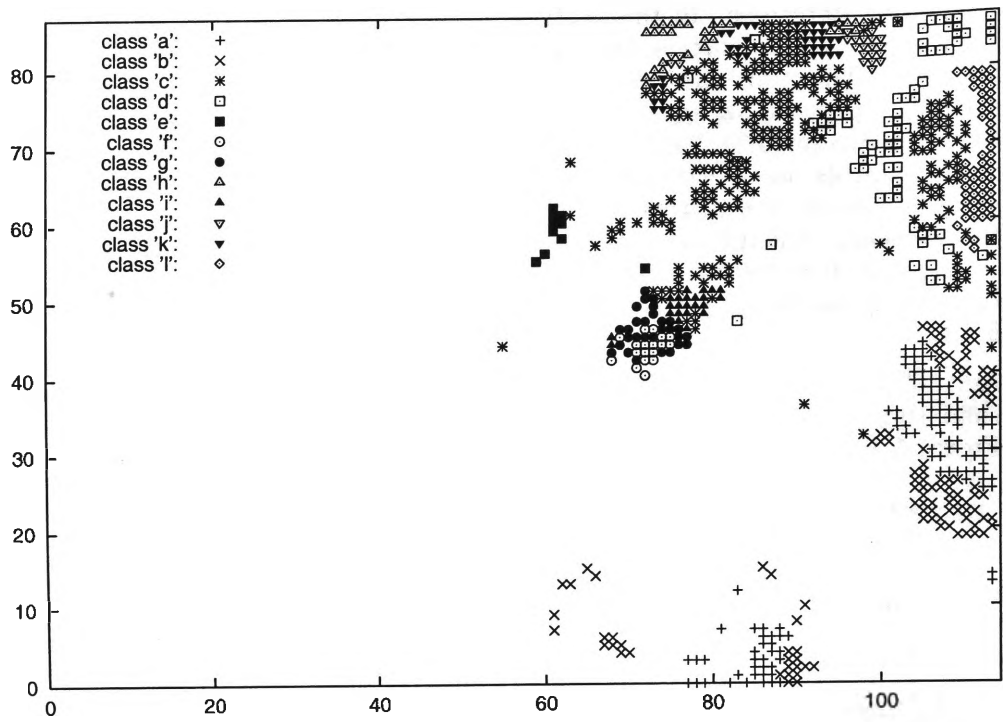


Figure 5.2: Mapping of root nodes after training a network in a supervised fashion with ϵ set to 0.1. Neurons are marked according to the class membership in which they belong.

cluster. It is observed that clusters belonging to different classes but in the same domain are often located very close to each other. There is very little overlap of clusters formed by different classes.

In Figure 5.3 we show the mapping of all nodes from the data set. It is observed that the network utilization rate is higher compared with the one obtained when training the network in an unsupervised fashion. In fact, 55.93% of the neurons on the map are involved in the mapping of at least one node from the training set [5]. Hence, it appears that the incorporation of supervision to the learning process results in a more efficient use of neurons.

Section 6.2 addresses another supervised neural model which is able to process graph structured information. The model described in Section 6.2 is based on an MLP type architecture and training is performed using a gradient decent method. A comparison of the supervised SOM-SD model with the unsupervised counterpart is given in Section 5.4.2. The following subsection gives a more comprehensive evaluation of the supervised SOM-SD model.

5.4.1 Supervised SOM-SD on dataset-3

This section presents a more comprehensive set of results as obtained when training a SOM-SD in a supervised manner on dataset-3 (see Appendix A). The experiments conducted in this section were kept as similar as possible to those described for the unsupervised SOM-SD method in Section 4.4.3. This will allow a fair comparison

[5]The large size of symbols used in Figure 5.3 may produce the false impression of a higher usage rate.

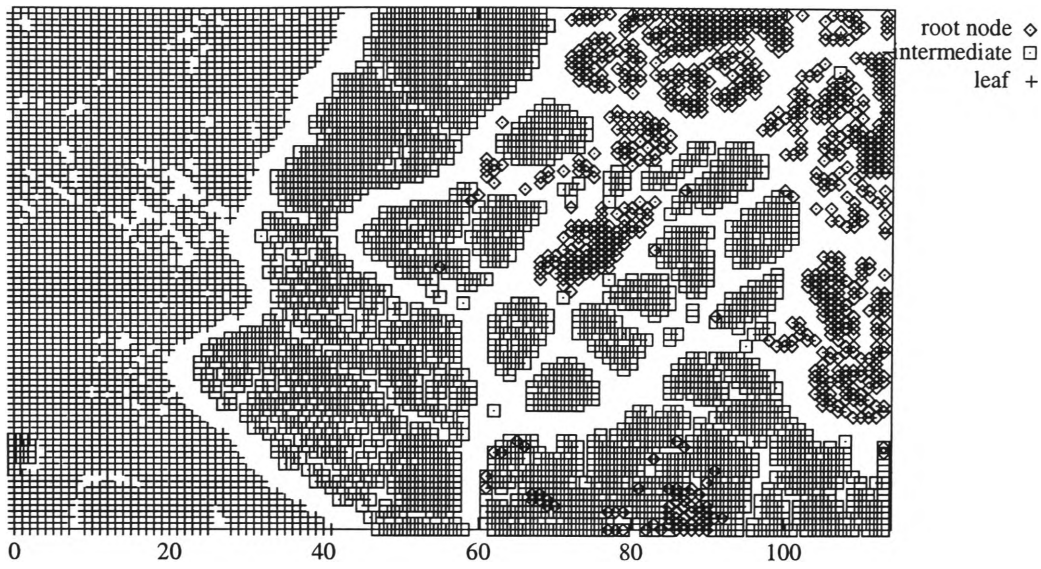


Figure 5.3: Mapping of all 29864 nodes in the training set. Neurons are marked according to the type of nodes that activated the neuron. Note that there is very little overlap between the different types of nodes.

between the two models later in this chapter. As a result, a network with hexagonal topology and a Gaussian neighbourhood function was chosen, and the rejection rate ϵ remained at 0.1 for all experiments. All other parameters, including the extension of the network were varied as described by the individual experiments.

Findings obtained from unsupervised trained SOM-SD networks demonstrated that the performance of the network increases with the size of the display map, but that the performance increase weakens as the display map grows beyond a certain size. This is an expected result because an increase of a SOM network means a liberation of the mapping conditions. The following experiment repeats this experiment on the supervised SOM-SD model.

Networks of various sizes were trained on dataset-3, a data-set featuring 29864 (sub-)graphs. The number of neurons in a network ranged from just 10 (a network with 2×5 neurons) to 84992 neurons (a network of size 332×256). All other parameters remained fixed at $\mu_2 = 1.9\mu_1$, $\alpha(0) = 0.08$, $\sigma = 114$, and the number of iterations was 350. Typical results obtained are shown in Figure 5.4

It is observed that the performance of the network increases relatively smoothly with the number of neurons. Furthermore, it is found that the performance of the network featuring approximately 10000 neurons gives the best "training time to performance" ratio because the increase of network performance is only minor for those larger than 10000 neurons.

Despite having trained the SOM-SD network in a supervised manner, it was not possible to achieve a performance level of 100% even for the largest network considered. This implies that the task of mapping graphs from this set of data onto a two-dimensional map cannot be done without compromise. In addition, it is found that the generalization performance can degrade when choosing large networks. This observation is quite typical for most type of neural network models. The problem here is called *overfitting*. Overfitting occurs when a network is trained for too long or with too many parameters so that the network is allowed to *focus* too much

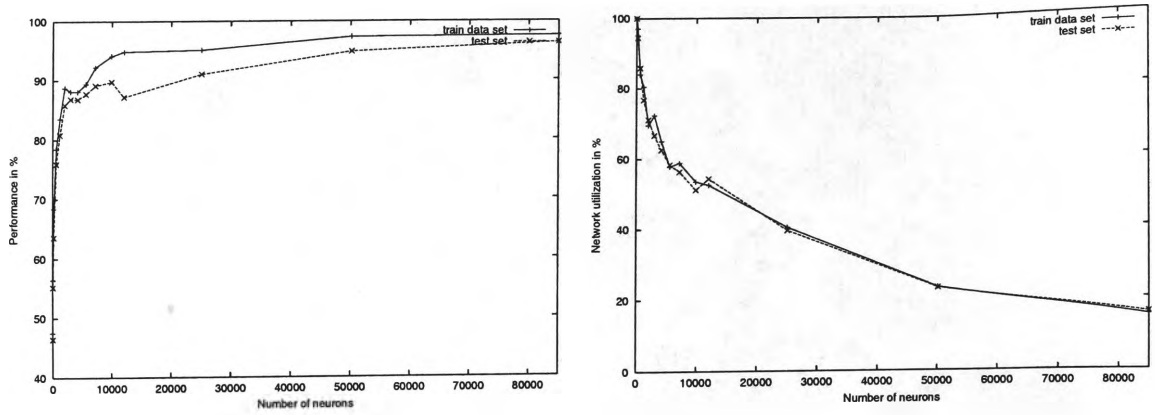


Figure 5.4: Performance of the SOM-SD trained in a supervised manner when varying the size of the network. The left hand plot illustrates the overall performance, the right hand plot gives the amount of neurons activated by at least one node.

on the training set and reducing its ability to generalize over unseen data.

Also shown in Figure 5.4 are the network utilization rates. These rates indicate how many neurons are involved in the mapping of nodes of the data set. It is found that the utilization rate decreases monotonically from 100% for the smallest network to less than 16% for the largest network. Networks with more than 25000 neurons feature more unused neurons than neurons that were activated at least once. The ratio between network use on the training set and network use on the test set gives some idea as to how well the network generalizes. From Figure 5.4 it is found that the network usage rate is nearly identical for both the training set and the test set on networks featuring more than 15000 neurons. This indicates that generalization does not occur and supports an earlier observation about the overfitting problem. Hence, the finding of this experiment is that it may not be beneficial to choose arbitrarily large networks. The appropriate sized network is application dependent but it is generally a good idea to choose a network featuring less neurons than (sub-)graphs in the data set. In the present case, an appropriate size for the network is approximately 114×87, a network featuring about $1/3$ the number of neurons as compared to the number of nodes in the training set.

The next experiment focuses on the influence of the weight values μ_1 and μ_2 on the performance of a SOM-SD network when trained in a supervised mode on dataset-3. This experiment utilizes networks of size 114×87 which were trained with the parameters $\alpha(0) = 0.08$, $\sigma = 114$, and the number of iterations was 350. The weight values ranged from $(\mu_1, \mu_2) = (0, 1)$ to $(\mu_1, \mu_2) = (1, 0)$. The result is shown in Figure 5.5.

It is observed that the performance of the worst performing network is already near 90% and that the performance increases with a stronger emphasis on μ_2. The best performances are achieved when choosing μ_2 between 50 to 1600 times larger than μ_1. In these cases, performance levels can exceed 98%. In cases where the focus on μ_2 increases to over $1600\mu_1$, the network performance starts decreasing and this becomes 67.07% when $\mu_1 = 0$.

An interesting observation is that the choice of μ_1 and μ_2 appears to have no effect on the generalization performance of the network. The performance achieved by using the test set is always about 1.5% below the performance obtained by the training set.

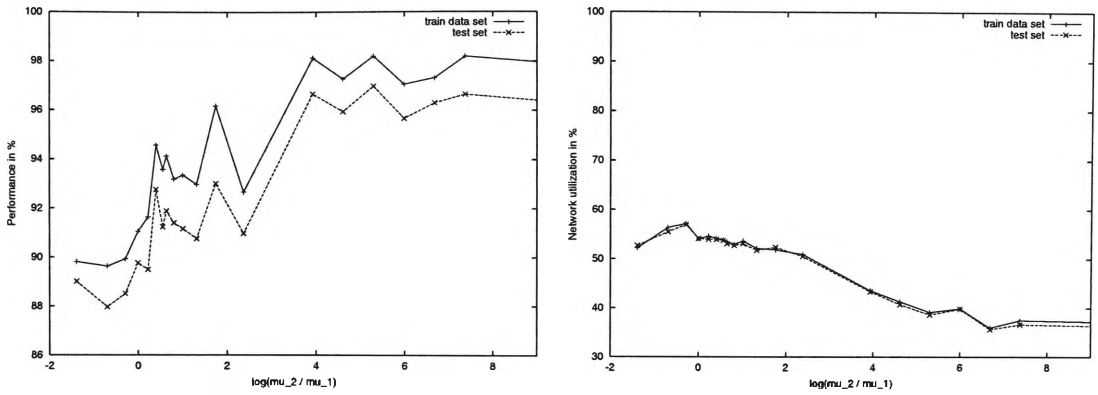


Figure 5.5: Performance of the SOM-SD model trained in supervised mode with varying μ_0 and μ_1. The network performance is shown on the left; the right hand plot illustrates the network utilization in percent.

A further interesting observation is made by investigating the network utilization rates. These rates decrease with larger μ_2 from a peak at 55% to a low of 36%. This shows that the data compression ratio obtained during the mapping process can be controlled through the choice of the weight values; a stronger focus on μ_2 results in a higher compression. A higher compression ratio has a similar effect as choosing a larger network in that the algorithm is given more room for the mapping of the data. The result is an improvement in network performance. However, μ_2 cannot be chosen arbitrarily large since the algorithm starts losing vital information provided by the data label. Hence, the network performance starts decreasing for very large μ_2.

Dataset-3 features a total of 29864 graphs and subgraphs. An experiment has been crafted to determine whether the size of the dataset is appropriate for the underlying learning task. A SOM-SD network of dimension 114×87 was trained in a supervised manner on subsets of the training set. The number of the training data ranged from just 61 subgraphs (1/64 of dataset-3) to the full 29864 subgraphs in the training set of dataset-3. Training parameters were identical to those used when conducting this experiment on the unsupervised SOM-SD method (see Chapter 4). The result of the experiments is as shown in Figure 5.6.

It is observed that the network performs above the 90% mark whenever the data set features more than 1/8-th of the size of dataset-3. The performance falls sharply with smaller training sets. In addition, the generalization performance was observed to be the best when using just 1/8-th of the size of data set-3. Also shown in Figure 5.6 are the network utilization rates. The network utilization rate decreases with the size of the training set. This can be attributed to the fact that the compression ratio decreases with smaller training sets since the size of the network remains constant. However, the decrease of the utilization rate is not proportional to the size of the training set because the fewer input data are given a higher degree of freedom to spread over the network. Interestingly, the network utilization rate as obtained by the full test set (for all the experiments) follows sharply the rates obtained from the training set. An observation which cannot be not fully explained yet.

The question of how many training patterns are required for a successfully performing SOM-SD trained in supervised mode is application dependent. However, it can be stated that due to the incorporation of a supervisor signal into the learning process, a higher compression ratio can be achieved compared with that obtainable by

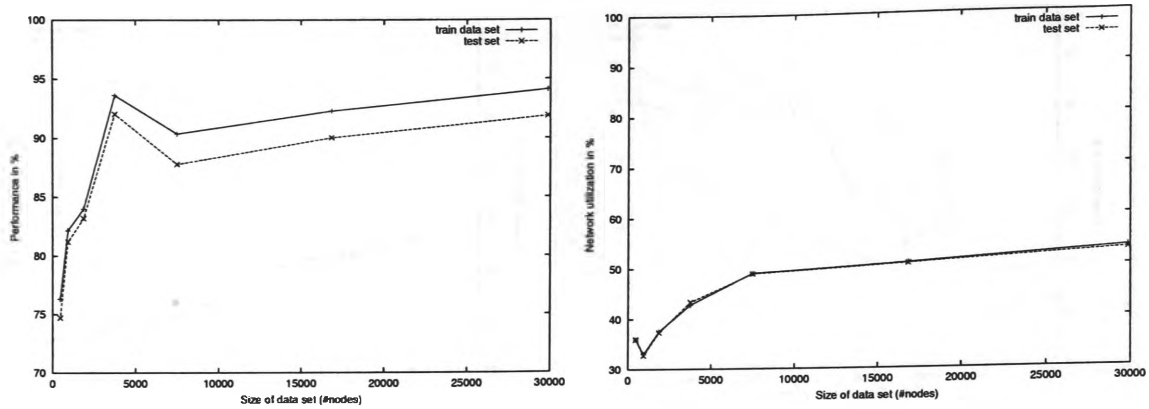


Figure 5.6: Network performance versus size of the training set. The network performance of a SOM-SD network trained in supervised mode is shown on the left hand plot. The right hand plot illustrates the amount of neurons activated by at least one node.

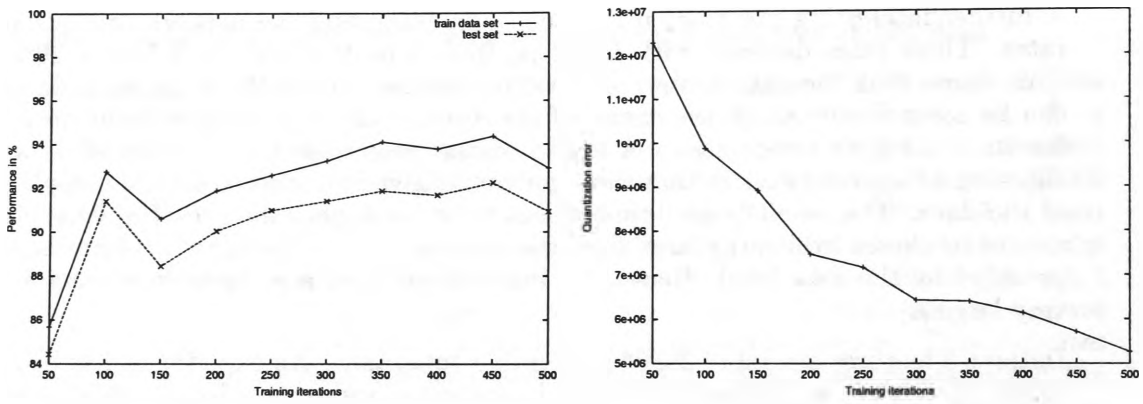


Figure 5.7: Network performance versus the total number of iterations trained. The left hand plot shows the network performance when trained in supervised mode, the right hand plot gives the quantization error.

unsupervised trained networks. It is interesting to observe that the generalization performance may decrease if the training set is chosen too large, an observation which is also often made with other supervised neural models such as MLP based architectures.

Next, the influence of the number of training iterations on the network performance is investigated. For this experiment a network of size 114×87 is trained on dataset-3 for a given number of iterations. All other parameters remained fixed at $\mu_2 = 1.9\mu_1$, $\sigma = 114$, and $\alpha(0) = 0.08$. The results are as shown in Figure 5.7.

It is found that the network is unable to perform satisfactory if trained for less than 100 iterations. Training the network for more than 200 iterations produced stable and well performing networks. However, it is also observed that the increase of network performance becomes less significant for larger number of training iterations, at the same time, the generalization performance tends to decrease. This is an indication that overfitting occurs if the network is trained for too long. This result agrees with observations typically made on other supervised trained neural models.

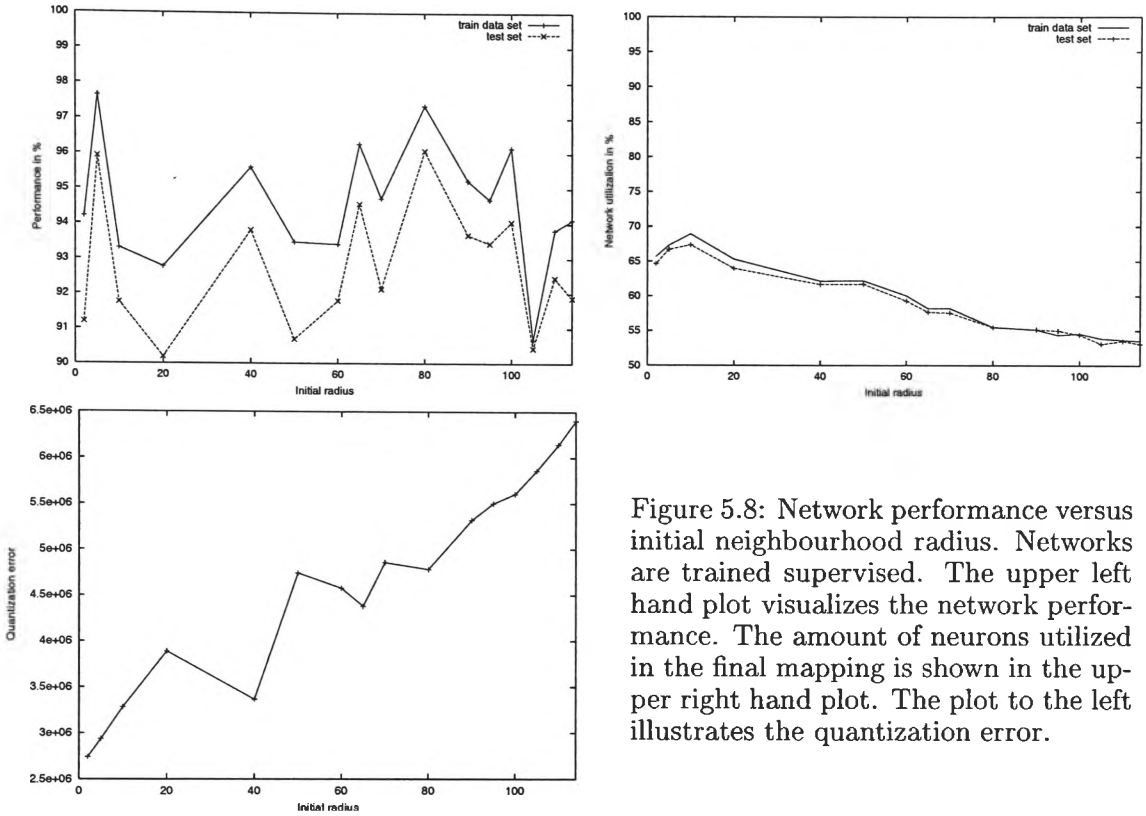


Figure 5.8: Network performance versus initial neighbourhood radius. Networks are trained supervised. The upper left hand plot visualizes the network performance. The amount of neurons utilized in the final mapping is shown in the upper right hand plot. The plot to the left illustrates the quantization error.

The number of training iterations required for a given learning task is application dependent. However, it can be stated that self-organizing maps tend to require more iterations for smaller datasets.

Also shown in Figure 5.7 is the quantization error obtained when training a supervised SOM-SD network for a given number of iterations. It is observed that the quantization error decreases significantly with the number of training iterations. Comparing this with the increase of the network performance, it is found that a decrease of the quantization error is by no means proportional to the performance gain. Hence, judging the network performance by looking at the quantization error alone is not an appropriate approach.

In Section 4.4.3 an experiment was crafted to determine the influence of the initial neighbourhood spread to the final network performance. The following experiment repeats this efforts on a SOM-SD network trained in supervised mode. A network of size 114×87 was trained for 350 iterations using $\mu_2 = 1.9\mu_1$ and $\alpha(0) = 0.08$. The initial neighbourhood spread was varied as indicated in Figure 5.8 which gives the results of the experiments.

It is observed that the network performance exceeds 90% for any $\sigma(0)$ values chosen in these experiments. However, the performance level fluctuates between 90.67% and 97.65% without a clear indication to a dependency on $\sigma(0)$. By looking at the network utilization rates, we find that the rates decrease relatively smoothly with larger $\sigma(0)$ which indicates that the network compresses the information more efficiently. However, at the same time the quantisation error increases. This demonstrates that the network performance is governed more strongly by other parameter rather than $\sigma(0)$, and that an optimal choice for $\sigma(0)$ can only be found through trial-and-error.

The following experiment targets the remaining free adjustable parameter $\alpha(0)$. Networks with dimension 114 × 87 were trained on dataset-3 using $\mu_2 = 1.9\mu_1$, $\sigma(0) = 114$, and the number of training iterations are 350. Results are shown in Figure 5.9.

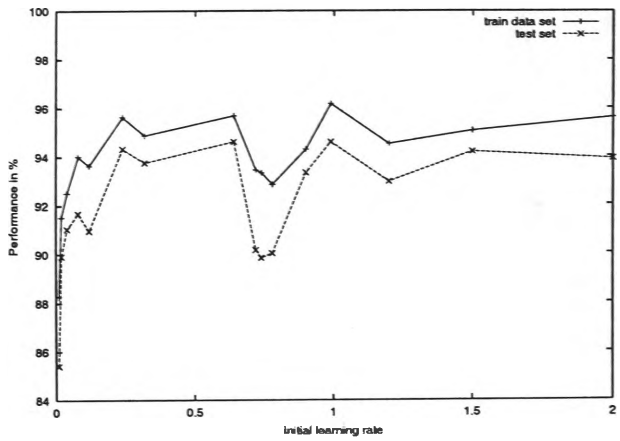


Figure 5.9: Network performance versus the initial learning parameter. The plot to the right gives an overview over the network performance.

It is observed that the network performs above the 90% level for all but the smallest initial learning rates. Good performances can be achieved for a large range of $\alpha(0)$ but it was observed that when choosing $\alpha(0) > 2$ that networks suffered from data overflow errors. Hence, a supervised trained SOM-SD network demonstrates robustness to the initial learning rate. However, in order to obtain optimal results, trial-and-error approaches need to be applied. In the present case, the best result was obtained when choosing $\alpha(0) = 0.99$.

In addition, no evidence was found for effects of $\alpha(0)$ on the generalization performance. The network always performed around 1.5% less well on the test set when compared with the training set.

Summarizing the results conducted in this section, it is possible to define a set of parameters that are best suited for the given data set. Hence, the most appropriate network configuration would feature an architecture with 114 × 87 neurons which is to be trained for 350 to 450 iterations using $\mu_2 \approx 1600\mu_1$, $\sigma(0) \approx 80$, $\epsilon \approx 0.1$, and $\alpha(0) \approx 0.99$. Networks trained with this set of parameters performed at 99.72 (training set) and 99.53 (test set).

This concludes the experiments conducted on the supervised SOM-SD model. A comparison of the results with the unsupervised SOM-SD is given in the following section.

5.4.2 Comparison with unsupervised SOM-SD

This section compares the performance of the supervised SOM-SD network with the unsupervised version of SOM-SD which was described in Chapter 4. Overall, the performance of a SOM-SD network trained in supervised mode exceeds the performances obtained from an unsupervised SOM-SD network significantly in all observed circumstances. Nevertheless, none of the 184 experiments conducted were able to achieve a 100% performance level. We find that a supervised SOM-SD network was only able to achieve 99.72% under optimal conditions. The following gives a more detailed comparison and analysis:

Network size Experiments revealed that for both models the best suitable network architecture for the given learning problem features approximately 10000 neurons which is about 1/3-rd of the size of the training set. While the performance of a supervised trained SOM-SD network exceeds that of the unsupervised counterpart, it is noted that the generalization performance is weaker for the supervised model. The unsupervised trained network demonstrated that the performance on the test set is always less that 1% below the performance achieved on the training set. This value increased to typically 1% to 2% for the supervised method. In addition, it is found that the network utilization rate is much the same for both models.

Vector weight values The supervised model demonstrated great robustness to the choice of μ_1 and μ_2, performing well for any values between $\mu_2 = 200\mu_1$ and $\mu_2 = 1600\mu_1$. This is in contrast to the unsupervised model which performed best when $\mu_2 = 100\mu_1$. This demonstrates that the introduction of supervision to the learning process dominates the influence of the vector-weight values. Nevertheless, it is observed that the generalization performance obtained with the supervised model is not as good as with the unsupervised method. Statements concerning generalization made previously also apply for this scenario. Also, network utilization rates appeared to be similar for both models.

Size of the training set The behavior of the two models is remarkably different as soon as the size of the training set is reduced. While the unsupervised model requires at least 50% of the samples in dataset-3, the supervised model is able to achieve good results with just 12.5% of training data. Evidently, due to the incorporation of supervision to the learning process, fewer training samples are needed. The supervisor guides the network into the "right" direction while the unsupervised model has to derive desired features from the samples alone.

Number of iterations The unsupervised SOM-SD network requires considerably more training iterations as the supervised method. While the supervised model is able to achieve good results with as few as 100 iterations, the unsupervised model requires at least 4 times as many. This observation is consistent with the previous observation on the size of the data set. Hence, the supervisor signal governs the learning process into the "right" direction whereas the unsupervised model is controlled only through statistical information encoded in the training set. Consistent with this are the observed quantization error rates which are considerably lower for the supervised method.

Neighbourhood radius The supervised SOM-SD network has demonstrated great robustness on the choice of $\sigma(0)$. Nearly any value of $\sigma(0)$ produces good results though fluctuations are also observed. In contrast, unsupervised trained SOM-SD networks perform best with small $\sigma(0)$. A smaller neighbourhood radius means that the updating rule has a lesser effect on areas away from a winning neuron, and hence is not able to affect information learned and encoded in distant areas of the map. In contrast, the presence of a supervisor during the learning process assures that negative effects on distant areas are minimized so that it is possible to achieve good results even with large $\sigma(0)$. In addition, the quantization error in the supervised model is at least a magnitude smaller when compared with results from the unsupervised model.

Learning rate A learning rate has to be chosen carefully when training a SOM-SD network without supervision. In the present case, the best value for $\alpha(0)$ is

0.4. In comparison, the supervised SOM-SD network gives superior results on a large range of learning parameters. Any value from within the range [0.02 : 2] enabled the network to perform well. However, both networks failed to be trained successfully due to data overflow errors when $\alpha(0)$ is set to values larger than 2.

In summary, the incorporation of supervision to the learning process produces robust and well performing networks even when trained with fewer training samples and for fewer iterations. Hence, training a SOM-SD in supervised mode should be the preferred method if a supervisor signal is available.

5.5 Conclusions

This chapter introduced a new method that allows the training of self organizing maps on graphs in a supervised fashion. It was demonstrated that through the incorporation of class membership information into the learning process the performance of the network is increased considerably. The proposed method appears to be very stable and is not very sensitive to the rejection rate. In addition, the supervised trained SOM-SD model has demonstrated great robustness to initial conditions for μ_2, $\alpha(0)$ and $\sigma(0)$ rendering this model considerably more robust to learning parameters when compared with the unsupervised counterpart. This improvement comes at almost no additional computational cost. A simple `if` statement in the neuron update algorithm is all that is different between the training algorithms of supervised and the unsupervised SOM-SD models.

Additional benefits of this method are that the complexity of the training algorithm is not greater than the one for unsupervised trained networks, and that the algorithm is capable of handling missing or incomplete class information. In the case where data have no class label available, the training algorithm reduces to the unsupervised mechanism. Hence, the supervised learning mechanism generalizes the method introduced in the unsupervised SOM-SD chapter.

Note that there is no convergence theorem for this training algorithm introduced in this chapter. This deficiency also afflicts the general SOM model.

Chapter 6

Multilayer Perceptron Networks on graphs

6.1 Introduction to supervised learning with MLP

Multilayer Perceptrons (MLPs) are perhaps the most well known form of artificial neural networks [49]. MLP networks gained considerable fame from the fact that even the simplest of its structures (i.e. a single hidden layer multilayer perceptron network) has universal approximation property [52]. MLP networks consist of a collection of artificial *neurons* arranged in layers and are massively interconnected by "communication" lines. The artificial neurons in an MLP network are simple models of brain cells, trying to emulate the behaviour of their natural counterpart. Essentially, an artificial neuron is a simple device which produces an output by passing the sum of all its inputs through a memoryless nonlinearity. Inputs are received from other artificial neurons in the network through the "communication" lines associated with constant weights. These communication lines are often referred to as synapses and the associated weights are called synaptic weights or simply weights [48]. The output of an artificial neuron is transmitted to other artificial neurons in the network via those weights.

Artificial neurons must not be misunderstood to be purely theoretical or a software implementation. It is an abstract mathematical model of the biological counterpart intending on capturing the most important aspect as far as pattern recognition is concerned. It does not pretend to be an accurate model, e.g., of the living nature of the biological neuron. A network of artificial neurons could be considered as a hardware device designed based on a theoretical abstract model. It could be implemented using software *simulation* so that its behaviour with various parameters or inputs can be studied. Software simulations allow a fast, flexible, and affordable mechanism for evaluating new neural network models. This is because of the fact that the mathematical model of a network of artificial neurons could be very difficult to analyse, and very few qualitative or quantitative results exist for a network of artificial neurons connected together in an arbitrary manner. In these situations, software simulation is probably one of the quickest way to gain some insight into the behaviour of the network under various working conditions.

There are three different types of layers in an MLP network [48]. The output of an artificial neuron is transmitted to other artificial neurons in the network via

those weights. The *output layer* is a collection of artificial neurons which produce the output of the network. Hence, neurons located in this layer are called *output layer neurons* or in short *output neurons*. Dependent on the problem, the output neurons can either have a linear transfer function (in regression type problems) or a memoryless nonlinearity (in classification type problems). Another layer is the *input layer* which takes the input of the network. This layer does not actually contain artificial neurons but rather sensory devices called *input nodes*. For simplicity, input nodes are sometimes seen as simple neurons with a linear transfer function. The third type are layers that are located inbetween the input layer and the output layer. Such layers are called *hidden layers*, and artificial neurons located within a hidden layer are referred to as *hidden layer neurons* or in short *hidden neurons* [1]. The hidden layer neurons have memoryless nonlinearity.

Typically, the dimension of the input layer and the dimension of the output layer is dictated by input/output data pairs provided with a training set used for training a network. As is well known, a major difficulty in the deployment of multilayer perceptrons is that the number of hidden layer neurons is obtained by trial and error. There are no known practical methods which can provide an estimate of the number of hidden layer neurons [49] [2].

On the other hand, there exist a number of constructive methods which can be deployed to design a feedforward neural network [3]. Some methods result in feedforward neural networks which do not have the usual multilayer perceptron architecture. One of the earliest constructive method is the cascade correlation [23]. This method is a step by step constructive method, adding hidden layer neurons, one at a time, until the training data set is suitably modelled, i.e., until a suitably chosen error criterion is minimised. However, it results in an architecture, which is of the feedforward type, but not of the multilayer perceptron type.

Generally, MLP networks are trained by minimising a cost function. Because of the nonlinear nature of the optimisation problem, often one means of obtaining a solution is to apply some kind of gradient descent techniques. This is often referred to in neural network literature as a *learning rule* or an *updating rule*. A learning rule adjusts the strength of the synaptic weights in the network and thus, influences how information is transmitted between the neurons.

MLP networks are typically trained in a *supervised* fashion. Supervised training means that for every given network input vector, a target (numeric or symbolic vector) is given. The target represents the desired network output. Hence, the primary purpose of the learning rule is to adjust the network weights so that the desired net-

[1] Henceforth, there will not be any danger of confusion whether we are referring to a biological neuron, or an artificial neuron. Hence, we will use the term "neuron" to denote an artificial neuron.

[2] There are various information criteria based methods, e.g., Bayesian information criterion [76], Akaike Information criterion [2]. However, these information criteria based methods have not proved to be useful in the determination of the number of hidden layer neurons required for a particular problem.

[3] In this chapter, we make a difference between multilayer perceptron, and feedforward neural network architectures. We use the term 'multilayer perceptron' to denote an architecture which has at least one hidden layer. There are no interconnections among the hidden layer neurons. On the other hand, we use the term 'feedforward neural network' to denote an architecture, which has at least one hidden layer; the hidden layer neurons are permitted to have zero time delay interconnections with other hidden layer neurons, and/or the output neurons. However, we do not allow any connections from the hidden neurons to the input neurons. This distinction will be made clear in Section 6.2.2 and in Chapter 7 respectively.

work output meets the target value in the best possible way, e.g., in minimising an accumulated squared error function; the error is defined as the difference between the target and the actual output from the network. The difficulty of the learning task is that it has to update the weights for a possibly large collection of different input/output data pairs. To produce a good network, training is generally performed in small steps (controlled by a learning rate) for many iterations. The weight updating mechanism performs modification of the weights based on the error so as to adjust the weights in a direction that reduces that error. The procedure of reporting the error to the network (the output error would be modified as it makes its journey through the multilayer perceptron) and updating of the weights is called an *error back-propagation* algorithm or in short *backpropagation*. Hence, MLP networks are sometimes referred to as *backpropagation networks* by some researchers.

MLP networks have been well studied [23, 51, 63, 64]. There exist a number of updating algorithms, most of them either attempt to improve the performance of a simple gradient descent type algorithm, or attempt to give a faster convergence.

Traditionally, MLP networks are restricted to the processing of fixed size data structures such as vectors. Some extended models exist which allow the processing of data sequences [21, 99, 100]. Such networks are called *time-delay* neural networks or *recurrent* neural networks. Often, recurrent neural network process data by splitting up sequences into equally sized, possibly overlapping portions. These portions are then presented to the network one by one where the activation of the network from processing the previous portion is used as an additional network input when processing the next portion.

However, the processing of data structures such as graphs has not been possible until recently. In this chapter we will describe a method that has become known as *recursive MLP* or in short *RMLP*. RMLPs are networks that encode graph structured information in a recursive manner. Training of an RMLP network is performed through BPTS which is an acronym for *backpropagation through structure*. The updating rule can be considered as the extension of the backprop updating algorithm in the MLP networks to the processing of data structures. In this chapter, when referring to BPTS we mean a neural network architecture updated through the BPTS updating rule.

The structure of the chapter is as follows: in Section 6.2, we will describe and derive the BPTS algorithm. In Section 6.2.1, we will give a brief overview of the representational models for data structures, with special emphasis on tree models. In Section 6.2.2, we will describe a data structure representational model based on the multilayer perceptron for each node, together with its training algorithms. This is a slight generalization of the models introduced in [29]. In Section 6.3, we will present results of experiments on MLP networks for data structures. A summary of findings made in this chapter is given in Section 6.4. Constructive models are addressed in Chapter 7. In Chapter 7 we will also introduce an extended MLP based neural network model for data structures which features greater similarities with a cascade correlation type network.

6.2 Back-propagation through structure (BPTS)

As indicated in Section 6.1 earlier, data structures can be represented using multilayer perceptrons with fixed input vector lengths [14, 86]. Recent activities made efforts to reformulate the problem into one which does not require a fixed length input vector [29, 86]. In essence, this methods employs a divide and conquer like approach in that it builds a neural network model for each node in the data struc-

ture. The novelty of the method lies on the fact that the model for each node has the same parameters. This is akin to weight sharing, except that it is arrived from quite different perspective and considerations. Once such a model is furnished, the unknown parameters can be learned minimizing a cost criterion.

In [86], a particular neural network model is employed to model each node in the data structure, viz., a multilayer perceptron. This section will give a rigorous mathematical formulation of the MLP algorithm for representing the DOAG data structures. Then a training algorithm for RMLP type networks is derived. We start with a mathematical formulation of a representation model for a special class of data structures, viz., a graph. This is a slight extension of the work presented by [29].

6.2.1 Data Structure Models

Consider a general tree model [4], with M nodes. Each node has a maximum of c out degrees, i.e., each node can only have a maximum of c children. Each node also has a label $\mathbf{u}$, which is assumed to be an m-dimensional vector. We will denote the labels by m-dimensional vectors: $\mathbf{u}^j$, $j = 1, 2, \ldots, M$. The label may consist of real values, or integer values. It is assumed that m will be the maximum dimension of the labels at any node. If there are less than m-dimensional labels, say, at node j, the vector $\mathbf{u}^j$ will be padded with zeros. It is further assumed that the output of the tree is obtained at the root node, i.e., $\mathbf{y}$, which is a p-dimensional vector. The structure of the model will be given by the structure of the tree. Note that the definition of the tree only as a set of nodes $\mathbf{u}$ neglects the topology of the tree. However, such a definition greatly simplifies the notation used later in this section. The underlying topology of the data will be respected in the text so that there is no reason for confusion.

Thus the question of tree structure representation is as follows: given a training set $\mathbf{u}^j(i), \mathbf{t}(i); j = 1, 2, \ldots, M_i; i = 1, 2, \ldots, N$, can we find a tree neural network architecture which will minimize the error criterion:

$$J = \frac{1}{2} \sum_{i=1}^{N} tr\left((\mathbf{t}(i) - \mathbf{y}(i))(\mathbf{t}(i) - \mathbf{y}(i))^T\right) \tag{6.1}$$

where $\mathbf{y}$ is the output of the tree. $tr(\cdot)$ denotes the trace operator which in this case is the sum of the diagonal values of the matrix. The superscript T denotes the transpose of a vector or a matrix. Note that in (Equation 6.1), we have used the least square error criterion. In general, we may use other types of cost criteria, e.g., a maximum absolute error criterion.

Furthermore, here we assume that the output data is only available at the root node. This is in recognition of the special nature of the problem formulation. In general, it is possible to assume that output data are also available at the intermediate nodes. But this situation is rare in data structures, as we do not normally assume that output data are available at the intermediate nodes in this type of problems.

The tree architecture dictates a hierarchical approach, as one needs to process the data in a systematic hierarchical fashion. In this case, we need to process the

[4] A tree data structure differs slightly from a DOAG data structure in that a tree is a connected DOAG. The use of a tree model in this section helps to simplify the notation. The capabilities of the neural model under consideration are not affected by this restriction.

data from the bottom up, i.e., from the frontier nodes first towards the root node. Frontier nodes are defined as ones which do not have any children.

A device which has been found to be extremely useful in formulating this type of problems is the q operator, alternatively known as the *shift operator*. This is defined as follows: $q^{-1}x(k) \triangleq x(k-1)$, where k denotes the k-th stage in which the data is being processed. For example, $q^{-1}x(k)$ denotes the data which is available from the processing performed at the previous stage. Note that this operator does not involve any computations at all. It is merely used as a device to denote the data which was processed in the previous stage, and is available in the current stage for further processing. The significance of this notation will become clear in both Section 6.2.2, and later in Chapter 7.

As the structure of the tree is dictated by the nodes of the tree, the data structure model depends on possible mathematical models used to represent each node. There are many models available. For example, [86] employed a multilayer perceptron model, while in [29], this is generalized to a hidden Markov model.

A fundamental assumption which we place on the problem is: the data is structurally invariant [86]. By this we mean the data does not change in time, or with respect to the structure. From this assumption, we can assume that all nodes have the same neural network model, all having the same weights. This assumption significantly reduces the number of parameters to be estimated in the model. Without further mention, this will be the assumption which we will place on the problems at hand.

6.2.2 Multilayer Perceptrons

The architecture of a classic Multilayer perceptron (MLP) is given as follows:

$$\mathbf{y} = F_p(C\mathbf{x}) \tag{6.2}$$
$$\mathbf{x} = F_n(B\mathbf{u}) \tag{6.3}$$

where $\mathbf{u}$ is an m-dimensional input vector, $\mathbf{x}$ is an n-dimensional vector denoting the outputs of the hidden layer neurons, or units, and $\mathbf{y}$ is an p-dimensional output vector. B and C are matrices of appropriate dimensions, denoting the weights connecting the input to the hidden layer neurons, and from the hidden layer neurons to the outputs respectively. $F_n(\cdot)$ is an n-dimensional vector:

$$F_n(\cdot) = \begin{bmatrix} f(\cdot) \\ f(\cdot) \\ \vdots \\ f(\cdot) \end{bmatrix} \tag{6.4}$$

$f(\cdot)$ is a nonlinear function. This can be a sigmoidal function, or a hyperbolic tangent function. $F_p(\cdot)$ is a p-dimensional vector, defined similarly to $F_n(\cdot)$.

We will refer to this architecture as the 'classic' MLP architecture. Note that we have abused the notation here slightly in that we include the bias in the model. Each hidden layer neuron and each output neuron is assumed to include a bias term. This is commonly handled by assuming an extra node which has a constant output [49]. In order not to clutter the notation too much, we include this in the model, rather than denoting it explicitly.

It has been found by [77], that it is more beneficial to include a direct feedthrough of the input to the output. Hence we can use a more extended architecture as follows:

$$\mathbf{y} = F_p\left(C\mathbf{x} + D\mathbf{u}\right) \tag{6.5}$$
$$\mathbf{x} = F_n\left(B\mathbf{u}\right) \tag{6.6}$$

We will refer to this as the MLP architecture. The total number of parameters is $nm + np + mp$. A modular representation of this model is illustrated by Figure 6.1.

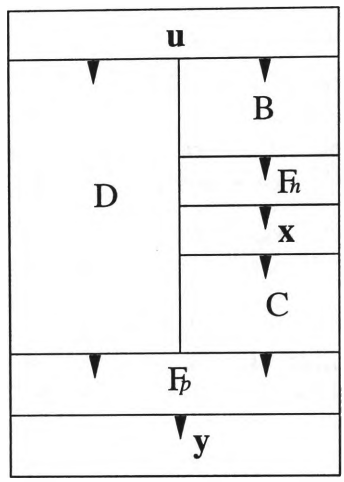


Figure 6.1: A modular representation of the MLP architecture. Arrows indicate the flow of information.

The classic MLP architecture has been applied to many problems, and found to be successful in modelling the underlying data. As indicated in Section 6.1, it has also been applied to data structures. But it was found that the classic MLP architecture requires that the input vectors all to be of the same fixed length. This has been found to be an impediment, as data structures giving the same output classifications may vary in length. This would necessitate the input vector to be the maximum of the set of input vectors of individual data structures. For those data structures with a shorter input vector, zero padding is employed. This technique of using the classic MLP architecture for data structure modelling has been found to be cumbersome, as well as adding to the complexity of the problem.

Instead, as indicated in Section 6.1, a suggestion to model each node in the data structure using a MLP has been introduced [86]. This opens the way to a more efficient modelling of data structures. We will describe how such technique can be deployed below.

In the tree architecture, we will use the MLP representation in each node. We assume that there are a maximum of c children in the tree, hence, any node in the tree incorporating a multilayer perceptron can be represented as follows:

$$\mathbf{y} = F_p\left(C\mathbf{x} + D\mathbf{u}\right) \tag{6.7}$$
$$\mathbf{x} = F_n\left(B\mathbf{u} + Aq^{-1}\mathbf{z}\right) \tag{6.8}$$

where $\mathbf{y}$, and $\mathbf{u}$ are respectively the p-dimensional output and the m-dimensional label (input) to the node. The labels can be either integer or real values. If we

denote the elements of the label (input) vector as u_i, $i = 1, 2, \ldots, m$. In each node, it is not necessarily true that all the elements u_i need to be present. If the element is not present then it is replaced by a value of 0. This device allows us to represent the labels in each node by using the same symbol. $\mathbf{x}$ denotes the n-dimensional output of the hidden layer neurons. Now because each node has a maximum of c children, the hidden layer neurons are influenced by the data which were processed by the children of the current node. This forms effectively an extra set of inputs to the hidden layer neurons. But these inputs from the children are dependent on the weights in the MLP, as each node is modelled by the same MLP architecture. Hence, the inputs to the parent node from the children nodes depend on the same model as the parent node. This is the main reason why such a model is called a "recursive neuron" in [86]. In order to denote that the data is available from the children, we use the q operator described in Section 6.2.1 to denote this. $\mathbf{z}$ may be the output of the children, or the outputs of the hidden layer neurons of the children, as will be explained below.

Now each hidden layer neuron is influenced by the output from the previous stage, i.e., from the children. Each child could make available to the parent node two types of outputs, viz., the output $\mathbf{y}$, or the outputs of the hidden layer neurons $\mathbf{x}$. If the available child's output is $\mathbf{y}$, this is known as a two layer network, and if the available child's output is $\mathbf{x}$, then this is known as a one layer network [86]. We find this nomenclature quite confusing, as it does not clearly indicate the situation. As a result, we will call the one when the child makes available to the parent node the hidden layer neuron outputs as a 'state MLP tree architecture'. The one when the child makes available its output to the parent, we will call it an 'output MLP tree architecture'. This nomenclature, we feel, is more appropriate.

Thus, in the state MLP tree architecture, each node is described by the following model:

$$\mathbf{x} = F_n\left(B\mathbf{u} + Aq^{-1}\mathbf{x}\right) \tag{6.9}$$

where $Aq^{-1}\mathbf{x}$ is a shorthand form to represent the following:

$$Aq^{-1}\mathbf{x} \triangleq [\ A_1 \quad A_2 \quad \ldots \quad A_c\ ] \begin{bmatrix} q_1^{-1}\mathbf{x}_1 \\ q_2^{-1}\mathbf{x}_2 \\ \vdots \\ q_c^{-1}\mathbf{x}_c \end{bmatrix} \tag{6.10}$$

where $\mathbf{x}_i$ denotes the n-dimensional vector of the hidden layer neurons of the i-th child. A_i denotes the $n \times n$ matrix depicting the weights connecting the hidden layer neurons of the i-th child to the hidden layer neurons of the current node. Note that as the hidden layer neurons in the children are different, hence we have used the notation q_i^{-1} to denote the relationship between the i-th child and the current node. A modular representation of a state MLP tree network architecture is illustrated by Figure 6.2.

Similarly in the output MLP tree architecture each node is described by the following model:

$$\mathbf{y} = F_p\left(C\mathbf{x} + D\mathbf{u}\right) \tag{6.11}$$

$$\mathbf{x} = F_n\left(B\mathbf{u} + Aq^{-1}\mathbf{y}\right) \tag{6.12}$$

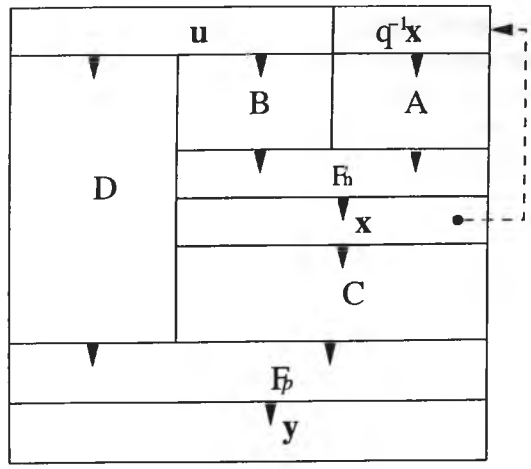


Figure 6.2: A modular representation of a state MLP architecture. Arrows indicate the flow of information.

where again $Aq^{-1}\mathbf{y}$ is a shorthand form of the following:

$$Aq^{-1}\mathbf{y} \triangleq [\ A_1 \quad A_2 \quad \ldots \quad A_c\ ] \begin{bmatrix} q_1^{-1}\mathbf{y}_1 \\ q_2^{-1}\mathbf{y}_2 \\ \vdots \\ q_c^{-1}\mathbf{y}_c \end{bmatrix} \tag{6.13}$$

where $\mathbf{y}_i$ is the p-dimensional output vector of the i-th child. A_i is a $n \times p$ matrix, denoting the weights connecting the outputs from the i-th child to the hidden layer neurons of the current node.

Note that in the case of the output MLP architecture, we assume that the root node has an output layer [5], i.e.,

$$\mathbf{y}^R = F_p\left(G\mathbf{x} + H\mathbf{u}\right) \tag{6.14}$$

where $\mathbf{y}^R$ is a r-dimensional output vector denoting the output of the root node. G and H are respectively $r \times n$ and $r \times m$ matrices. This device is required as the output MLP tree architecture has only hidden layer neuron outputs at each node. Thus, we include an output stage for the root node, so that the root node output is r-dimensional rather than an p-dimensional vector. A modular representation of an output RMLP architecture is as shown in Figure 6.3. It is noted that recursion is performed via the output layer $\mathbf{y}$ whereas a state MLP tree architecture uses the hidden layer $\mathbf{x}$ as the recursive layer. In order to avoid confusion we call the recursive layer a *state* layer because it recursively passes the state of the network back when processing a parent node.

The number of network parameters available for each of the two models is summarized in Table 6.1. Hence, dependent on whether it is a state MLP tree architecture, or an output MLP tree architecture the number of parameters are different.

[5] This equation is the same as the one shown in (Eq. 6.8). Here we use different symbols to emphasise the fact that it is only at the output of the entire network, rather than at each node, as is the case of the 'state MLP' model described earlier in this section.

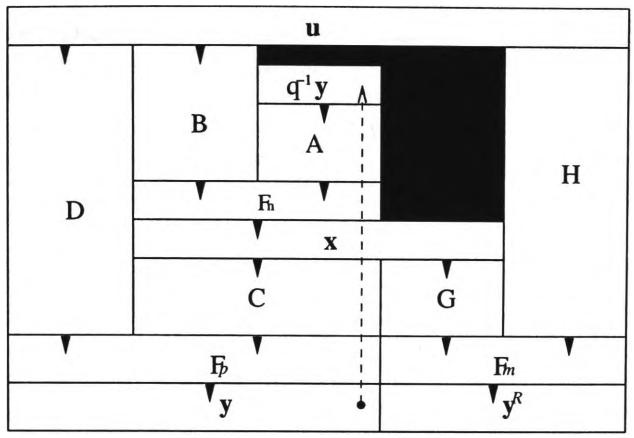


Figure 6.3: A modular representation of an output MLP architecture. Arrows indicate the flow of information.

Architecture	Number of parameters in each node
State MLP	$pm + pn + nm + cn^2$
Output MLP	$pm + pn + nm + cnp$

Table 6.1: A table showing the number of parameters in each node in the MLP tree architectures.

We have assumed that all the nodes are governed by the same model, with the same parameters. In the case of the output MLP tree architecture, we need to add the extra number of parameters due to the output at the root node, i.e., we need to add $rn+rm$ parameters. These will need to be added to the total number of parameters which are required to be learned from the set of training data.

To avoid the cluttering of symbols, we note that the root node output of the state MLP tree architecture is essentially the same as the output MLP tree architecture for each node. Hence, we will abuse the notation by denoting the output of the state MLP tree architecture by the same symbols, i.e., we denote $G = C$ and $H = D$. There is no risk of confusion as in the state MLP case, the matrices C and D are only active at the root node, while in the output MLP case, C and D are active in all the nodes.

Remarks:

1. In the above formulation we have assumed that the output of the root node lies within a range; if we use a sigmoidal nonlinear activation function, this range will be between 0 and 1. This is most convenient for classification studies.
2. We can modify the above formulation for regression type of problem, if we relax the output of the root node to be linear, i.e.,

$$\mathbf{y}^R = C\mathbf{x} + D\mathbf{u} \tag{6.15}$$

This simple device will allow the tree architectures to be deployed for regression type problems.

3. It was shown by [45] that the state MLP architecture has a universal approximation property. However, as common with universal approximation theorems, [45] did not give any results on how to determine the required number of state neurons for a particular problem. In practice, the number of required state neurons is obtained by a trial and error process, as will be indicated in this chapter.

6.2.3 Training algorithms

We can train the parameters of the MLP ((Eq. 6.7), and (Eq. 6.8)) using a batch update algorithm, minimising the error criterion J.

$$J = \frac{1}{2}\sum_{i=1}^{N} tr\left(\left(\mathbf{t}(i) - \mathbf{y}(i)\right)\left(\mathbf{t}(i) - \mathbf{y}(i)\right)^T\right) \tag{6.16}$$

where $\mathbf{t}(i)$ are the target values. The weight updating algorithm is given as follows:

$$C(i+1) = C(i) - \eta\frac{\partial J}{\partial C} \tag{6.17}$$

$$D(i+1) = D(i) - \eta\frac{\partial J}{\partial D} \tag{6.18}$$

$$B(i+1) = B(i) - \eta\frac{\partial J}{\partial B} \tag{6.19}$$

$$A(i+1) = A(i) - \eta\frac{\partial J}{\partial A} \tag{6.20}$$

where η is a global learning constant, and $\frac{\partial J}{\partial C}$ needs to be evaluated. This can be evaluated element by element, due to the recursive nature of the processing steps. Let the ij-th element of the matrix C be c_{ij}.

$$\frac{\partial J}{\partial c_{ij}} = -\sum_{k=1}^{N} \mathbf{e}(k)^T \Lambda(\zeta)\left(C\frac{\partial \mathbf{x}}{\partial c_{ij}} + Q_{ij}\mathbf{x}(k)\right) \tag{6.21}$$

where $\mathbf{e}(i) \triangleq \mathbf{t}(i) - \mathbf{y}(i)$. $\Lambda(\zeta)$ is a $p \times p$ diagonal matrix, defined as follows:

$$\Lambda(\zeta) = \begin{bmatrix} f'(\zeta_1) & 0 & 0 & \dots & 0 \\ 0 & f'(\zeta_2) & 0 & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & f'(\zeta_p) \end{bmatrix} \tag{6.22}$$

where $f'(\cdot)$ denotes the derivative of the nonlinear activation function $f(\cdot)$ of the neuron. $\zeta = C\mathbf{x} + D\mathbf{u}$, a p-dimensional vector, and ζ_i is the i-th element of the vector ζ. Q_{ij} is a $p \times n$ matrix with all elements 0 except a value of 1 in the i-th row and j-th column. Note that we can define $\delta(i) = C^T\Lambda(\zeta)\mathbf{e}(i)$, the backprop error. Thus, (Eq. 6.21) can be written more conveniently as:

$$\frac{\partial J}{\partial c_{ij}} = -\sum_{i=1}^{N}\left(\delta(i)^T\frac{\partial \mathbf{x}}{\partial c_{ij}} + \mathbf{e}(i)^T\Lambda(\zeta)Q_{ij}\mathbf{x}(i)\right) \tag{6.23}$$

The second term within the parenthesis is the common term which is associated with the backprop algorithm. The first term is new. This term arises as x depends on C, as the value of x depends on hidden layer neuron outputs from children of the root node. The quantity $\frac{\partial \mathbf{x}}{\partial c_{ij}}$ can be evaluated from the following:

$$\frac{\partial \mathbf{x}}{\partial c_{ij}} = \Lambda(\xi) A q^{-1} \frac{\partial \mathbf{x}}{\partial c_{ij}} \tag{6.24}$$

where $\xi = B\mathbf{u} + Aq^{-1}\mathbf{x}$. It is noted that the right hand side of (Eq. 6.24) depends on $q^{-1}\mathbf{x}$, i.e., the outputs of the children of the current node. But the output of the children nodes are themselves dependent on their children. Thus the evaluation of this derivative is dependent on the tree structure. Hence we cannot continue with the formal derivation without referring to a specific tree structure. It suffices to say that the evaluation [6] of $q^{-1}\frac{\partial \mathbf{x}}{\partial c_{ij}}$ depends on the tree structure; and needs to be evaluated first before we can evaluate $\frac{\partial \mathbf{x}}{\partial c_{ij}}$ on the left hand side. The term $q^{-1}\frac{\partial \mathbf{x}}{\partial c_{ij}}$ in turn needs to be evaluated dependent on their children. Thus it is observed that the evaluation of $\frac{\partial \mathbf{x}}{\partial c_{ij}}$ in (Eq. 6.24) depends on the tree structure. It must commence from the frontier nodes first, compute this value, and then forward them to their parent nodes. The parent nodes evaluate this quantity based on information from the children nodes, and in turn forward them to their parent nodes, and so on. This process goes on until it reaches the root node, in which stage it will be evaluated, before the value of c_{ij} can be updated. This *forward* evaluation of the quantity $\frac{\partial \mathbf{x}}{\partial c_{ij}}$ until the frontier nodes, and then the *backward* evaluation towards the root node is a characteristic feature of the updating algorithms for this type of architectures. This algorithm is an extension of the RTRL algorithm. Its behaviour is the same[7] as the BPTS algorithm as obtained by Kuechler [60][8].

In a similar manner we can compute the derivative with respect to the elements of D. Let d_{ij} be the ij-th element of D. Then,

$$\frac{\partial J}{\partial d_{ij}} = -\sum_{i=1}^{N} \mathbf{e}(i)^T \Lambda(\zeta) \left(C \frac{\partial \mathbf{x}}{\partial d_{ij}} + Q_{ij}\mathbf{u} \right) \tag{6.25}$$

where Q_{ij} in this case [9] is a $p \times m$ matrix with all elements 0, except a value of 1 in the ij-th position. Again, the second term in (Eq. 6.25) is the common term associated with the backprop algorithm. The first term is the one which is associated with the fact that we are dealing with a tree structure. The value of $\frac{\partial \mathbf{x}}{\partial d_{ij}}$ can be evaluated from the following:

$$\frac{\partial \mathbf{x}}{\partial d_{ij}} = \Lambda(\xi) A q^{-1} \frac{\partial \mathbf{x}}{\partial d_{ij}} \tag{6.26}$$

[6]Here in order to differentiate the two $\frac{\partial \mathbf{x}}{\partial c_{ij}}$ on both sides of (Eq. 6.24), we use $q^{-1}\frac{\partial \mathbf{x}}{\partial c_{ij}}$ to denote the quantity on the right hand side of (Eq. 6.24).

[7]In view of their equivalent behaviour from henceforth we abuse the notation and refer to this as BPTS algorithm.

[8]Recurrent neural networks are known to have similar behaviour. From our experiments with recursive neural networks we find this is true as well though there is no formal proof of their equivalence.

[9]Here we have abused the notation by denoting this matrix using the same symbol Q_{ij}, even though their dimensions are different.

where $\xi = B\mathbf{u} + Aq^{-1}\mathbf{x}$. The evaluation of $q^{-1}\frac{\partial \mathbf{x}}{\partial d_{ij}}$ depends on the tree structure.

Let a_{ij} denote the ij-th element of the matrix A. Then we have:

$$\frac{\partial J}{\partial a_{ij}} = -\sum_{i=1}^{N} \delta(i)^T \frac{\partial \mathbf{x}}{\partial a_{ij}} \tag{6.27}$$

and

$$\frac{\partial \mathbf{x}}{\partial a_{ij}} = \Lambda(\xi)\left(Aq^{-1}\frac{\partial \mathbf{x}}{\partial a_{ij}} + Q_{ij}q^{-1}\mathbf{x}\right) \tag{6.28}$$

where Q_{ij} is a $n \times n$ matrix in this case. The quantity $q^{-1}\frac{\partial \mathbf{x}}{\partial a_{ij}}$ again can be evaluated from the architecture.

Let b_{ij} be the ij-th element of the matrix B. Then we have

$$\frac{\partial J}{\partial b_{ij}} = -\sum_{i=1}^{N} \delta(i)^T C \frac{\partial \mathbf{x}}{\partial b_{ij}} \tag{6.29}$$

and

$$\frac{\partial \mathbf{x}}{\partial b_{ij}} = \Lambda(\xi)\left(Aq^{-1}\frac{\partial \mathbf{x}}{\partial b_{ij}} + Q_{ij}\mathbf{u}\right) \tag{6.30}$$

where Q_{ij} in this case is a $n \times m$ matrix with all elements zero except a value of 1 in the ij-th position. $q^{-1}\frac{\partial \mathbf{x}}{\partial b_{ij}}$ can be evaluated from the tree structure.

The unknown parameters A, B, C and D can be updated accordingly. These updating algorithms are jointly called the backprop through structure (BPTS) algorithm.

In the above formulation of the updating rules, we used the simple gradient update. However, in practice it is known that the simple gradient update rules are slow. There are a number of ways to speed up the updating process by incorporating second order information on the curvature of the error criterion without actually evaluating the Hessian matrix. Quickprop is one of the more successfully updating mechanisms incorporating second order information. Two further algorithms, viz., incorporation of a momentum term, and the Rprop algorithm are considered.

Incorporation of a momentum term

The definition of a learning algorithm which incorporates a momentum term can be made quite simply as follows [48, 49]:

$$\Delta w_{ij}(k+1) = -\eta\frac{\partial J}{\partial w_{ij}} + \alpha\Delta w_{ij}(k) \tag{6.31}$$

where α is commonly called the momentum term. $w_{ij}(k)$ is the value of w_{ij} at the k-th instant. w_{ij} denotes any weight in the network; in this case, it is used to denote the connection between neuron i and neuron j in the network. The addition of this term incorporates some second order information into the training algorithm. We will refer to this algorithm as the momentum BPTS algorithm.

Quickprop

Quickprop [24] is a method which attempts to speed up learning by using information about the curvature of the error surface. Normally, this would require the computation of the second order derivative. Quickprop avoids this by assuming that the error surface is locally quadratic. During learning, it attempts to jump directly into the minimum of the assumed parabola (the local approximation of the error surface). Quickprop does this by computing the derivatives of each weight using regular backpropagation. A direct jump into the error minimum is attempted through the following algorithm:

$$\Delta w_{ij}(k+1) = \frac{S(k+1)}{S(k) - S(k+1)} \Delta w_{ij}(k) \tag{6.32}$$

where

$\Delta w_{ij}(k+1)$	weight change of the connection between neuron i and j
$\Delta w_{ij}(k)$	previous weight change
$S(k+1)$	Partial derivative of the error function at w_{ij}
$S(k)$	The previous partial derivative at w_{ij}.

Rprop

Introduced by Riedmiller in 1993, RProp [10] is a local adaptive learning scheme performing supervised batch mode learning. For a detailed discussion see [71, 70]. The idea is to eliminate possibly harmful influence of the size of the gradient. As a consequence, only the sign of the derivative is used to indicate the direction of the weight step. The weight step is exclusively determined by a locally dynamic learning rate as follows:

$$\Delta w_{ij} = \begin{cases} -\Delta_{ij}^{(k)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(k)} > 0 \\ +\Delta_{ij}^{(k)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(k)} < 0 \\ 0 & , \text{ else} \end{cases} \tag{6.33}$$

where $\frac{\partial E}{\partial w_{ij}}^{(k)}$ denotes the summed gradient over all patterns. The dynamic learning rate is adapted by:

$$\Delta_{ij}^{(k)} = \begin{cases} \eta^+ * \Delta_{ij}^{(k-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(k-1)} * \frac{\partial E}{\partial w_{ij}}^{(k)} > 0 \\ \eta^- * \Delta_{ij}^{(k-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(k-1)} * \frac{\partial E}{\partial w_{ij}}^{(k)} < 0 \\ \Delta_{ij}^{(k-1)} & , \text{ else} \end{cases} \tag{6.34}$$

where $0 < \eta^- < 1 < \eta^+$. Often, η^+ and η^- are set to fixed values e.g. $\eta^- := 0.5$ and $\eta^+ := 1.2$ are often a good choice.

A modification of this adaptation schema is to use a global learning rate which is adjusted as follows:

[10] **R**esilient back**prop**agation

$$\Delta^{(k)} = \begin{cases} \eta^+ * \Delta^{(k-1)} & , \text{ if } \delta_e^{(k-1)} \leq \delta_e^{(k)} \\ \eta^- * \Delta^{(k-1)} & , \text{ if } \delta_e^{(k-1)} > \delta_e^{(k)} \end{cases} \tag{6.35}$$

where δ_e is the summed absolute network error. η^+ and η^- are not fixed. Values are chosen which will lead to a smaller $\delta_e^{(k)}$. This modification has been found to have similar effects as standard RProp, and it reduces the computational expenses.

In practice, RProp was found to have the most efficient behaviour of all updating mechanisms listed here. It will be interesting to observe how RProp performs on recursive models. This will be investigated in this chapter.

Note that the classic BPTS training algorithm uses a static learning rate. However, M. Maggini suggests that the use of an adaptive learning rate in BPTS produces networks which converge faster. This was supported by initial experiments which demonstrated benefits of using an adaptive learning rate where the learning rate η is adapted by a similar mechanism as described in Equation 6.35. Hence, for the experiments described in this chapter we use an adaptive learning rate η for BPTS, BPTS with momentum, and RProp.

6.3 Experimental results on RMLP

In this section, we will present some computational experience which we have had with the BPTS algorithm. At this stage, we will concentrate on using only one class of problems, viz., the extended traffic policeman benchmark problem which we will refer to as dataset-3. The benchmark problem described in Section A actually provides three data sets. The reason why we restrict the experiments in this section to dataset-3 is that this dataset is a combination and extension of the other two data sets, dataset-1, and dataset-2 respectively. Hence, the careful evaluation of experimental results will allow us to draw conclusions on how the network models would have performed on the other two data sets as well. A short description of dataset-3 is as follows: the extended traffic policeman benchmark problem features a collection of 7500 graphs split equally to create a train and a test set. There are 12 classes defined over the patterns. Some classes have more data than others, i.e., each class does not have equal number of training data. Some classes can be distinguished through information encoded by the labels while other classes require structural information in order to be distinguished. The learning task is to process and to classify the graphs. The ideal results will be: train the network on the training data and when test on the test data it will correctly classify all the graphs into their respective classes. The dataset has been created artificially using our own software as described in Appendix A. The reason why we chose this one, instead of another problem is that this problem has a well controlled environment within which we can experiment with the method. Dataset-3 was used in the evaluation of self organising maps in Chapter 4. Hence it will be possible to compare the performance of supervised SOM-SD with that of the RMLP. Furthermore, experimental results obtained from the application of RMLP to a real world problem, viz., the logo recognition problem will be addressed later in Chapter 8.

Dataset-3 is a collection of graphs which feature a 12-dimensional binary target vector, and a 2-dimensional data label. Hence, for all the networks discussed in this section, the dimensionality of the output layer $\mathbf{y}$ (or $\mathbf{y}^R$ in the case of output MLP architecture) remains fixed at 12 and the dimension of the input layer $\mathbf{u}$ is 2. The maximum out-degree of any graph in the dataset is six.

Unless stated otherwise, the format of the plots is as follows: Sample points are indicated as points represented by various symbols. Lines connecting sample points are a linear interpolation. The vertical axis is usually used to give the network performance expressed in percent. A value of 100% indicates that the network was successful in classifying all patterns correctly. Similarly, a performance level of 0% means that no pattern was classified correctly. Since the dataset features 12 classes, it is expected that the network performance is at least $1/12 * 100\% = 8.3\overline{3}\%$. Such performance charts are unable to express how the networks perform on a class by class basis. Confusion matrices represented by various shades of gray are used to visualise how a network performed on individual classes. Darker fields in a confusion matrix represent a high classification rate (e.g. black $\hat{=}$100%), whereas lighter areas represent lower percentages (e.g. white $\hat{=}$0%). The scale of the gray colors used is linear.

The experiments are split into several sections. A performance evaluation of state MLP networks is given in Section 6.3.1 whereas Section 6.3.2 gives the performance of output MLP networks. A comparison between the two models: state MLP model and output MLP model is given in Section 6.3.3. Extensions to these two models will be discussed later in Chapter 7.

6.3.1 Experiments with state MLP networks

This section evaluates the performance of a state MLP architecture when applied to dataset-3. With state MLP networks, the only variable layer of neurons is the state layer $\mathbf{x}$. The dimensionality of the input layer $\mathbf{u}$ and the output layer $\mathbf{y}$ is controlled by the given data. The first experiment investigates the influence of the dimension of $\mathbf{x}$ on the network performance.

Varying the Number of State Neurons

We performed this experiment by using networks with a varying number of state neurons and trained for 400 iterations with an adaptive learning rate which was initialized with $\eta(0) = 0.04$. The result of this experiment is shown in Figure 6.4.

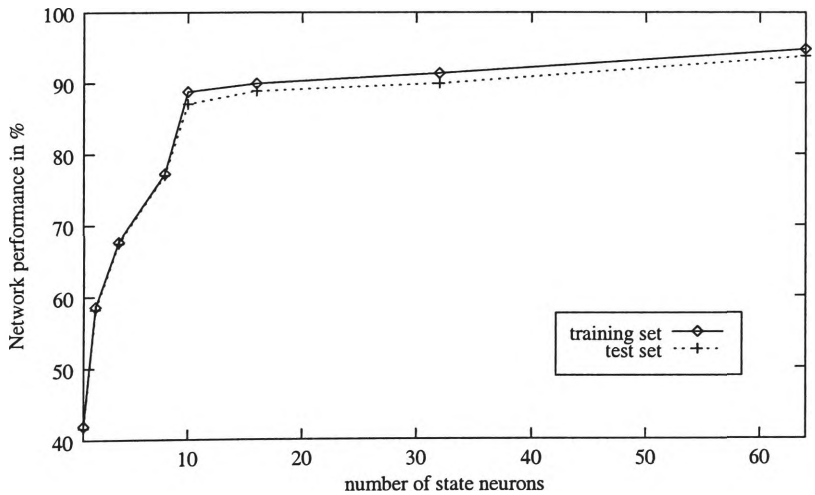


Figure 6.4: A diagram showing the recognition of the samples using a state MLP network trained using the BPTS updating rule. The number of state neurons are plotted against the network performance.

The finding of this first experiment is that at least 10 state neurons are required to obtain reasonable classification results on the given data set and that increasing the number of state neurons beyond 10 does not increase the network performance significantly. Also illustrated in Figure 6.4 is the generalization performance which is never more than 2% below the performance achieved by the training set.

Two properties are observed:

- The network does not appear to be overtrained
- The number of training patterns in the train set sufficiently represent possible instances of the learning problem.

Overtraining or too few training patterns often cause a poor generalization performance of a network which is not observed in this case. The performance levels observed in Figure 6.4 were obtained when updating the network weights with the BPTS training rule with dynamic learning rate. A comparison of performances achieved when training network with the RPROP learning rule or the BPTS with momentum learning rule is given in Figure 6.5.

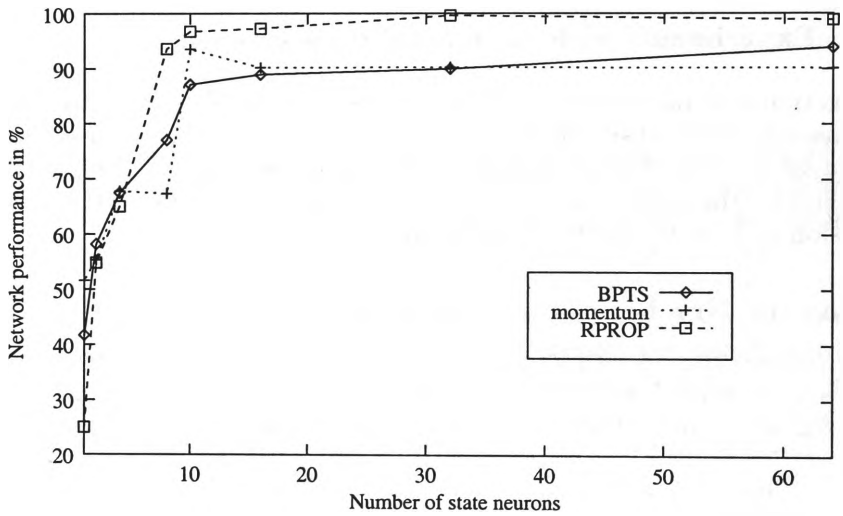


Figure 6.5: A diagram showing the recognition of the samples using a state MLP network featuring 10 state neurons. Three training rules: BPTS with dynamic learning rate, BPTS with RPROP learning rule, and BPTS with momentum learning rule, are considered.

Figure 6.5 demonstrates that the network performance can increase significantly when choosing "improved" learning rules instead of standard BPTS. Using the RPROP rule produced the best and also very stable results. A common property for all updating rules is that the network needs to feature at least 10 neurons in $\mathbf{x}$ to produce reasonably good results. The best result was obtained when training a network featuring 32 state neurons using RPROP updating; a performance level of 99.55, with only 17 patterns classified incorrectly, was achieved.

Table 6.2 presents the confusion matrix obtained from networks featuring 10 hidden neurons and trained for 400 iterations using the BPTS [11] and RProp learning rule.

[11] Henceforth, we will call a training algorithm BPTS to denote the BPTS training algorithm with a dynamic learning rate.

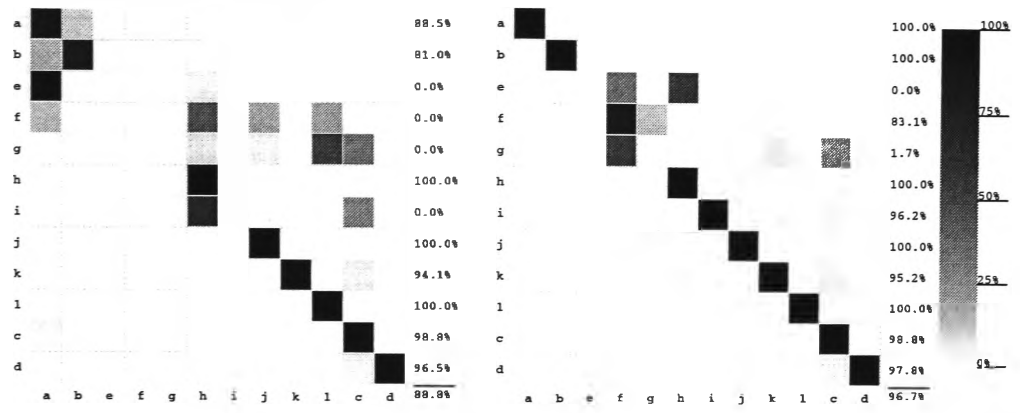


Table 6.2: Confusion matrix as obtained from training a state MLP featuring 10 hidden neurons with BPTS (left) and RProp (right) for 400 iterations.

It is observed that a network trained using BPTS performs unsatisfactory on the classes `f`, `g`, and `i` whereas an RProp trained network has the same problems with classifying patterns from the classes `e` to `g` correctly. The reason for this behaviour can be found in the properties of the training set. Given that there are 3750 graphs in the training set which would mean an average of 312 patterns for each class. However, the number of sample patterns in dataset-3 is not balanced. There are classes which feature up to 645 samples (class `a`) while other classes are represented poorly with as few as 28 samples. As it turns out, the classes `e`, `f`, `g`, and `i` are represented respectively with only 28, 59, 58, and 53 samples. This finding demonstrates that this recursive MLP model suffers from the same problem as the classic MLP model. For the classic MLP, it is observed that the network may perform poorly if training samples are not represented in approximate equal numbers. This is because in those cases, the learning rule adjusts the network weights more often in the direction of the class which is represented by a larger number of samples. In practice, one way to overcome this is by repeating the presentation of training data belonging to classes for which only few samples a number of times. Another way to avoid this is to use some a priori information, e.g., prior probability of the occurrence of samples in various classes, and use such information in the learning process.

Number of Training Iterations

Each of the past experiments executed 400 training iterations. Whether this is an appropriate value is investigated in the next experiment. Networks featuring 10 state neurons were trained for up to 2000 iterations. The performance of the network was probed at the end of every iteration. The result is as shown in Figure 6.6.

It is found that the application of the BPTS learning rule causes the network to converge slowest. Using BPTS updating rule requires at least 700 training iterations on the current dataset. In addition, the network performance increases very slowly when training for longer than 700 iteration which indicates the presence of a local minima or the learning landscape is almost flat. In comparison, the incorporation of a momentum term can speed up network conversion significantly. 400 training iterations suffice when using this weight adaptation mechanism. The RPROP updating rule converged fastest out of the three learning rules and produced networks that outperformed networks trained by using other rules at any stage during training. This finding confirms that 400 training iterations suffice for the current simulation

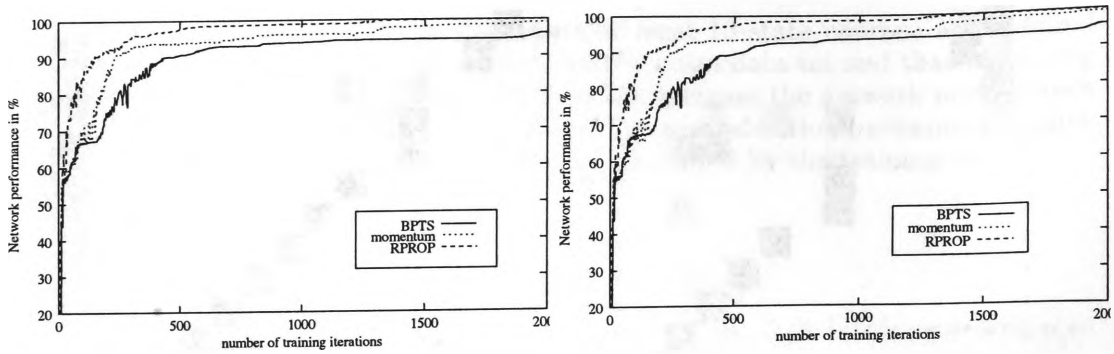


Figure 6.6: A diagram showing the recognition of the samples using a state MLP network featuring 10 state neurons. The number of training iterations are plotted against the network performance. Three learning rules: BPTS, BPTS with RPROP, and BPTS with momentum term, are used. The left plot shows the performance as obtained on the train set, the right plot is from applying the test set.

environment if the updating rule is either RPROP or BPTS with a momentum term. If network performance is critical and training time is not crucial, then training 1000 or more iterations can assure a slightly improved network performance. In addition to this finding, it is observed that the generalization performance (right plot of Figure 6.6) does also increase with the number of training iterations. This confirms an earlier observation which stated that the cardinality of the training set (the number of training samples) is sufficiently large to represent most instances of the learning problem, and thus, helps avoiding the problem with overtraining.

All networks are trained using adaptive learning rates which were initialized with $\eta(0) = 0.04$. During training, the learning rates are adjusted dynamically using the rule given by Equation 6.35. Figure 6.7 demonstrates how the learning rate changes during training.

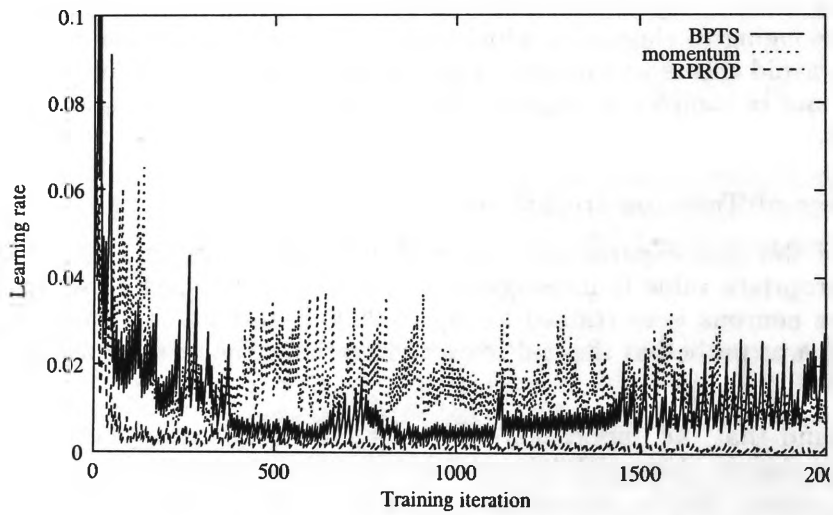


Figure 6.7: A diagram showing the adaptive learning rate values when training state MLP networks featuring 10 state neurons for 2000 iterations. Again three learning rules: BPTS, BPTS with RPROP, and BPTS with momentum term are used.

It is found that the development of the learning rate is different for different updating rules. In particular, learning with a momentum term causes the learning

rate to fluctuate considerably stronger than when learning with other rules. One reason for this fluctuation of the learning rate can be attributed to the influence of the momentum term. From the weight updating rule described in Equation 6.31 we find that the momentum term increases with every training step into the same 'direction'. Hence, at some stage, the size of the momentum term could increase to a value which is of no benefit to the training process. At this stage, an adjustment of the updating 'direction' is performed which has as an effect a significant adjustment of the learning rate causing the fluctuations of the learning rate as observed. Standard BPTS updating does not feature a momentum term and hence, behaves more smoothly. In comparison, the RPROP updating rule proves to be very efficient in making the network converge quickly. The efficiency of this updating mechanism is reflected by a relatively smooth learning rate which converges to relatively early in the training session.

Next, the development of the mean square error (MSE) during a training session is investigated. Figure 6.8 compares the MSE obtained when training networks with either BPTS, BPTS with momentum, or RPROP.

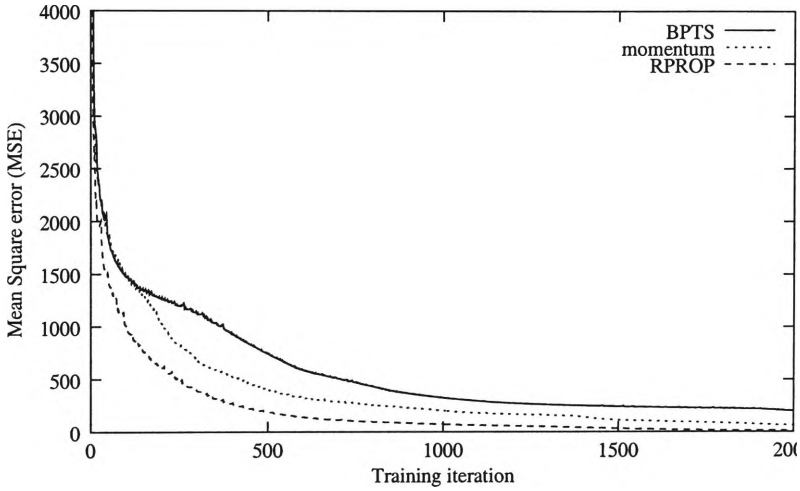


Figure 6.8: A diagram showing the mean square error when training state MLP networks featuring 10 state neurons for 2000 iterations. Three learning rules are used: BPTS, BPTS with RPROP and BPTS with momentum term.

Figure 6.8 confirms earlier findings which state that RPROP is the most efficient updating mechanism. However, this plot suggests that the number of training iterations is best chosen to be above 750 iterations whereas earlier findings showed that the classification performance is good after only 400 iterations (for the case of RPROP learning). Hence, the observation of the MSE alone does not lead to an optimal choice of learning parameters. In addition, it can be observed that the MSE curve for BPTS follows closely the curve of BPTS with momentum for the first 200 iterations. Only then, the incorporation of the momentum term to the updating rule starts to show benefits. The behaviour of the MSE curves support an earlier finding which suggested the presence of a flat learning landscape. A flat landscape is more easily overcome by a learning rule which features a momentum, or a learning rule which does not rely on the steepness of the gradient. The RProp updating rule demonstrated a good performance because it avoids the harmful influence of the size of the gradient. Quickprop, the fourth learning rule considered for the experiments

[12] failed to produce any reasonable results. The network error remained at a very high level independent of the parameters chosen for the network training sessions. Quickprop's assumption of a local parabola is clearly incorrect for the learning task at hand. As a result, we removed results obtained by utilizing this updating mechanism from the figures in order to improve the quality of the plots.

Size of the Training Set

The next experiment is to investigate the influence of the size of the training set to the overall network performance. The following experiment was conducted: Networks featuring 10 state neurons were trained for 400 iterations using the BPTS updating rule [13]. The size of the training set was reduced gradually from a total of 3750 graphs down to just 1% of this number. The result of this experiment is shown in Figure 6.9.

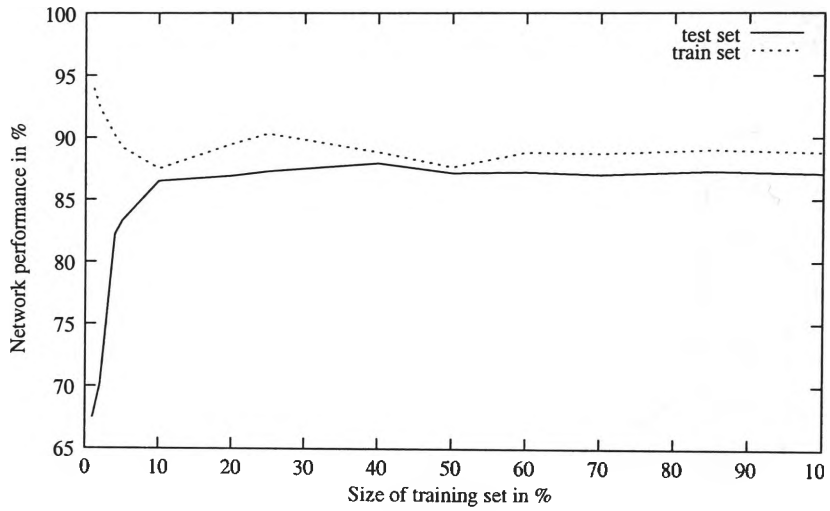


Figure 6.9: A diagram showing the recognition of the samples using a state MLP network featuring 10 state neurons. The number of training patterns are plotted against the network performance.

The experiments illustrate that just 10% or 375 of the graphs in the training set are sufficient to produce networks that generalize well. It is also shown that the network performance remains stable when using more than 375 data. This finding explains why the generalization performance was very good in the previous experiments.

Since there is no harm in using any number of training patterns between 375 and the full set of 3750 graphs, we continue to utilize the entire dataset for future experiments. This is done to allow a fair comparison with other methods described in this thesis. As will be found, there are models which may require substantially more than 350 graphs in the training set to perform satisfactory.

Summary

Summarising the findings of the experiments, it is found that state MLP networks are well able to encode graph structured data. A condition for obtaining a good

[12] Not shown in this thesis as it failed too often on many other tasks as well.

[13] We chose to use BPTS learning rule because it performs the worst out of the three learning rules considered in this chapter. Hence, if the results work for the BPTS learning rule, it will work on the other two learning rules.

network performance is that sufficient network parameters (network weights) exist to allow the encoding of information provided by the data set. In the case of state MLP networks, the number of network weights is controlled by the number of state neurons in layer $\mathbf{x}$. As with standard MLP networks, no general rule can be given on how many network weights and training iterations are required for any given learning problem. This can only be obtained through trial and error. Similarly, this is true for the cardinality of the training set. For the current case we find that a state MLP network will perform well on dataset-3 if it features at least 10 state neurons trained by the RPROP learning rule for 400 or more iterations using at least 375 graphs for training. The network performed stable under these conditions.

Similar experiments were conducted on the output MLP network model. Results are given in the following section. A comparison of the performance between state MLP and output MLP is given in Section 6.3.3.

6.3.2 Experiments with output MLP networks

This section discusses the performance of networks which feature the output MLP architecture when applied to dataset-3. Experiments are conducted in a way which allow for a fair comparison of their performance with other models such as those obtained by the state MLP in Section 6.3.3.

The main difference between the state MLP and output MLP architectures is that recursion is performed over layer $\mathbf{y}$ instead of layer $\mathbf{x}$ and that there is an additional output layer $\mathbf{y}^R$. Hence, the layer $\mathbf{x}$ acts like a hidden layer in the output MLP networks and layer $\mathbf{y}$ as the 'state' layer [14]. From experiments conducted in the previous section it was found that an appropriate choice of state neurons for the given learning problem is 10. We shall use this number of state neurons as a starting point for the experiments in this section.

Number of Hidden Layer Neurons

The first experiment is to investigate the influence of the dimension of the hidden layer $\mathbf{x}$ to the network performance. Networks with the number of hidden neurons ranging from 1 to 64 were trained for 400 iterations using the BPTS updating rule. The initial learning rate $\eta(0)$ was 0.04 and network weights were initialized with random values from within the range $[-1:1]$. The result is shown in Figure 6.10.

It is somewhat surprising to find that the network performed best with 32 hidden layer neurons and that the network performance was unstable when using less than 16 hidden layer neurons. Reasons for this relatively poor classification performance may be due to a number of reasons: It can range from a local minima problem, inappropriate learning parameters or insufficient training iterations to network architecture that simply does not work well. However, a local minima problem is highly unlikely to occur in this case since it was not observed with the state MLP network which can be seen as a substructure of this output MLP network. It is also unlikely to be caused by a failure of the architectural design for the same reason. Inappropriate learning parameters are also very unlikely to cause these problems because of the use of an adaptive learning rate, and the lack of other free training parameters that may have influenced the outcome. The most likely reason for the lack of good performance is an insufficient number of training iterations or the

[14] The nomenclature may appear a bit clumsy here as we use the word "state" to refer to the output of the node model. This can be confused with the "state" MLP architectures. However, this can be overcome by more clearly indicating the term, which is what we have done in this section.

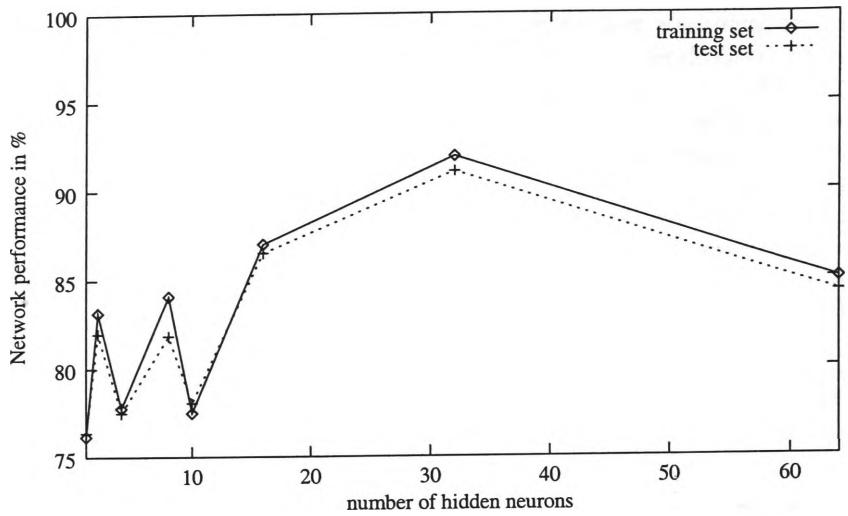


Figure 6.10: A diagram showing the recognition of the samples using an output MLP network trained through the BPTS updating rule. The number of state neurons is plotted against the network performance.

learning rule converging too slowly. This idea is supported by the very fact that observations on the results in the state MLP network experiments suggest at least 400 training iterations would be required for a good performance. In the case of the output MLP architecture, an additional layer of network weights need to be updated. The gradient based updating method has the effect that changes in network weights are smaller the further away a weight is from the output layer (this is a commonly known problem and an effect referred to as the long term dependency problem). Hence, it is well expected that 400 training iterations might not be sufficient for this model. Hence the observed behaviour shown in Figure 6.10 can be explained by the fact that a training session being stopped too soon. To investigate this line of reasoning, we conducted two experiments. First, the influence of the learning rule to the classification performance is investigated. Secondly, the number of training iterations are increased gradually to explore the influence of this parameter to the final outcome.

Learning Algorithm

The influence of the type of updating mechanisms to the network performance is investigated by the following experiment: Different weight updating mechanisms such as BPTS with momentum and RPROP are attempted. The result is presented in Figure 6.11. Again, it was found that the network performance depends significantly on the updating rule chosen.

It is found that the incorporation of a momentum term into the BPTS updating rule produces networks outperforming those trained by BPTS only. However, both updating rules show similar behaviour in that they feature a performance peak at around 32 hidden layer neurons, and unstable behaviour when choosing less than 16 hidden layer neurons. In comparison, the utilization of RPROP produces a very different behaviour. Networks trained with RPROP outperformed all other updating mechanisms significantly. In addition, RPROP produced good results with just 8 hidden layer neurons and demonstrated a stable behaviour over the entire range, and no significant decrease of performance is observed when choosing more than the optimum of 8 hidden layer neurons. This is in contrast to performance

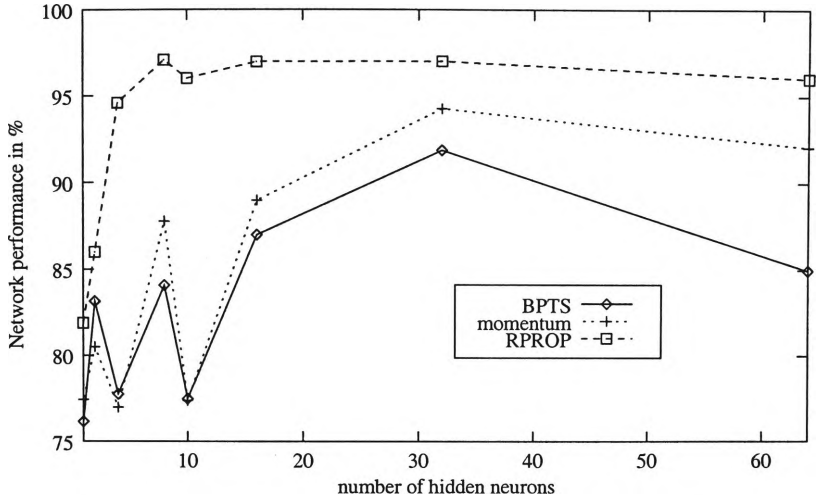


Figure 6.11: A diagram showing the recognition of the samples using an output MLP network featuring 10 state neurons. The updating rule is plotted against the network performance.

obtained from BPTS based updating rules.

Number of Training Iterations

As will be shown next, the behaviour of BPTS based updating rules can be explained by the following experiment which investigates the influence of the number of training iterations to the network performance. For this experiment, networks with various training rules were trained for up to 2000 iterations where the performance was measured at every iteration. The result is as shown in Figure 6.12.

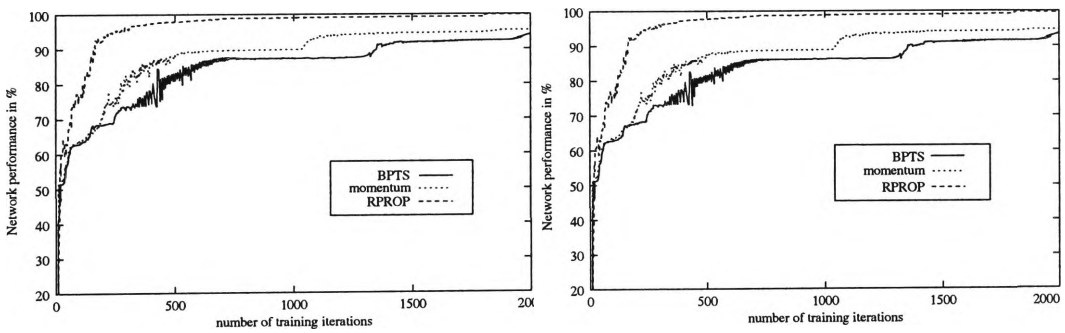


Figure 6.12: A diagram showing the recognition of the samples using an output MLP network featuring 10 state neurons. The number of training iterations are plotted against the network performance. The results obtained with the training set (left) and the test set (right) are shown. Three learning rules are used: BPTS, BPTS with momentum term, and RPROP.

The networks used for this experiment featured 8 hidden layer neurons and 10 'state' layer neurons. It is found that more than 1400 training iterations are required when updating the network weights with BPTS. Similarly, BPTS with momentum updating require about 1000 training iterations for convergence while RPROP has converged after just about 400 iterations. This explains the unexpected behaviour of earlier experiments which were executed with just 400 iterations. At this stage,

neither BPTS nor BPTS with momentum were able to adjust the network weights sufficiently well which resulted in unstable and/or less than optimal network performances.

The classification performance is investigated further by considering the confusion matrix given in Table 6.3. The confusion matrices obtained after training an output MLP network for 1500 iterations using BPTS updating (left matrix), and after training the network for 400 iterations using RPROP updating (right matrix) are shown.

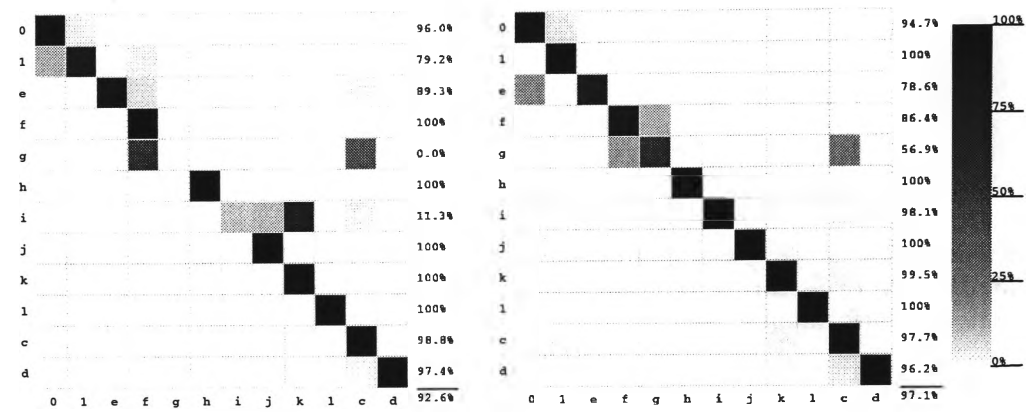


Table 6.3: Confusion matrix obtained from training an output MLP featuring 8 hidden nodes and 10 'state' neurons with BPTS (left) for 1500 iterations, and RProp (right) for 400 iterations.

As can be observed clearly, the performance of the network trained through BPTS is significantly worse when compared to the network which was updated through the RProp rule even though the BPTS trained network was updated nearly four times as many iterations. In addition, the network trained using BPTS also have considerable problems with correctly classifying patterns belonging to the classes g and i. The problem with classifying patterns from class g is also present for the network trained using RProp. This is a repetition of a problem described earlier. It was found that some classes are represented by fewer training patters. Amongst those classes were the classes g and i. Hence, the output MLP network exhibited the same problems with classes which are poorly represented. It appears that the problem is less severe when applying the RProp learning rule.

Size of the Training Set

The next set of experiments investigates the minimum size of a training set such that an output MLP network performs well on the given benchmark problem. The experiment was conducted as follows: Training sets were formed by randomly selecting graphs from dataset-3, where the size of the training set was varied between 1% and the full size of dataset-3. Output MLP networks featuring 8 hidden and 10 state layer neurons were trained on those datasets for 1500 iterations using the BPTS updating mechanism. The result of this experiment is shown in Figure 6.13.

It is found that only 375 of the 3750 graphs in dataset-3 are sufficient to produce a well performing output MLP network. Choosing more than the minimum of 375 training patterns had little effect on the network performance. The network was unable to generalize well when the training set consisted of less than 375 graphs.

Summary

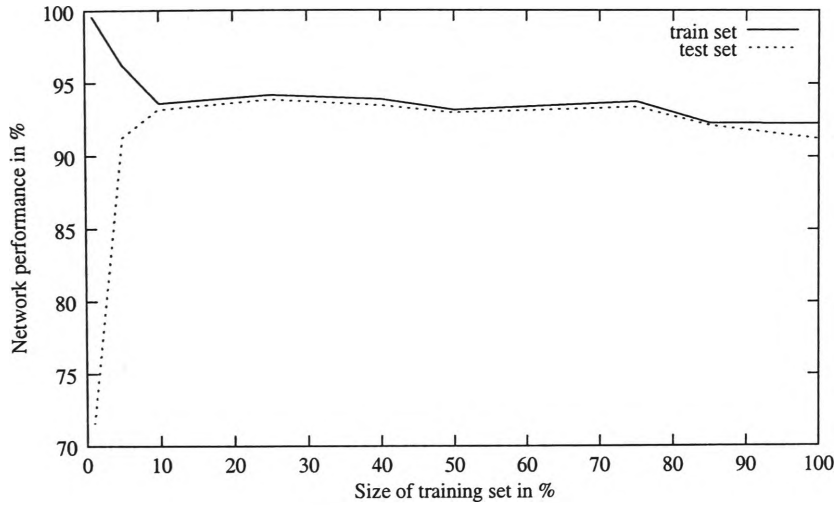


Figure 6.13: A diagram showing the recognition of the samples using an output MLP network featuring 8 hidden and 10 state neurons. The number of training patterns are plotted against the network performance.

Overall, the best performing network is produced when applying the RPROP learning rule for 400 or more iterations. Where network performance needs to be maximised and training time is not an issue, then training a network for substantially more than the minimum of 400 iterations can help to improve the generalization performance. The best performance was registered to be at 99.68% where only 12 graphs are classified incorrectly.

A comparison of the findings from this section with the findings from experiments conducted on state MLP networks is given in the following section.

6.3.3 Comparisons between state MLP and output MLP networks

Experiments conducted in Section 6.3.1 on state MLP models and in Section 6.3.2 on output MLP models demonstrated that both models are well able to encode and classify graph structured information. Moreover, both models behave similarly during training and produced best results when applying the RPROP learning rule. The performance of a state MLP architecture on dataset-3 was generally very good. A classification rate of 99.55% was achieved with a good set of initial parameters. In comparison, the output MLP architecture produced 99.68% classification rates which is marginally better than the performance of the state MLP network. It may seem that the given learning task is not sufficiently complex to allow the detection of more significant advantages of a output MLP structure over a state MLP structure. This assumption is confirmed by experiments conducted on a real world problem as addressed in Chapter 8. In Chapter 8 the output MLP architecture does demonstrate some advantages over a state MLP architecture.

A more detailed comparison and analysis is as follows:

Updating rule It was observed that the RProp learning rule is the most efficient and stable mechanism for updating weights in a MLP network. In contrast, the BPTS updating mechanism made MLP networks converge considerably slower and have problems with the error surfaces which feature flat regions. The incorporation of a momentum term or the use of RProp are effective

mechanisms for overcoming such flat regions.

Number of training iterations The number of training iterations required differs greatly with the choice of network architecture and updating rule. An output MLP architecture is essentially a state MLP architecture with an additional hidden layer. It is well known that gradient based weight updating mechanisms that rely on the size of a gradient have problems with deep network structures since the gradient tends to become smaller the deeper a weight is located inside a network. As a result, an output MLP network requires considerable more training iterations than a state MLP network. RProp does not rely on the size of the gradient, so a problem with small gradients is not observed. Both network types considered in this chapter required a similar number of iterations when trained by RProp.

Performance Both network architectures performed well on the given learning task. It was observed that the output MLP network has a slight advantage over an state MLP network in terms of performance. This observation becomes particularly interesting when considering that an output MLP with 8 hidden and 10 state neurons featured a total of 672 network weights [15] which is less than the number of weights in a state MLP network which has 740 weights when the number of neurons in the hidden layer is 10. Hence, the efficiency of a RMLP network can increase by the introduction of an additional layer.

Size of the training set Both network architectures have an equal demand on the training set. It was demonstrated that the networks can produce good results even if the training set is relatively small. However, this can also be observed as a side effect of a problem generally observed with MLP type networks: Classes that are represented only by few samples are often not classified well. The network may demonstrate a good performance globally, but the performance for some classes represented by few samples may be very poor. As a result, the reduction of the size of the training set has no effect until too few training samples are left for even those classes that were originally represented by a large number of samples.

Both types of network are a generalization of the 'classic' MLP and recurrent MLP type networks. This can easily be observed by considering a learning problem which consists of a set of graphs featuring only a single (root) node. In this case, the state MLP architecture reduces to a standard MLP architecture with a hidden layer. Using single node data reduces the output MLP architecture to a standard MLP architecture with a second hidden layer where the second layer is not used. We find from this observation that the two architectures are in fact very similar, and hence, demonstrate quite similar behaviours. Moreover, recurrent MLP models are also special cases of the state MLP model if the input data are sequences of nodes (i.e. graphs with maximum out-degree of one). Hence, both state MLP and output MLP models can be seen as a more general form of MLP architectures.

6.4 Conclusions

This chapter presented MLP type architectures providing effective means for tasks requiring the encoding of graph structured information in a supervised fashion.

[15] The networks considered for the experiments given in this chapter all featured an input dimension of 2 and an output dimension of 12. Note, that for the experiments we chose to set all weights in matrix D and H (direct connection of the input with the output) to be zero.

The models discussed are able to encode graphs as well as data sequences and fixed size input vectors. It was demonstrated that the use of the RProp updating rule produces networks that perform best in a relatively short time of training. Both network models have demonstrated a similar behaviour when applied to a benchmark problem. Nevertheless, the output MLP architecture, which features an additional hidden layer may have a slight advantage over the state MLP architecture which does not feature such a layer.

However, the two models also revealed a number of problems:

- It was demonstrated that patterns that belong to classes that are insufficiently represented in the training set are classified poorly. While this is a serious problem, it can be overcome quite easily in most cases by presenting data which are given in small numbers more often to the network.
- A second problem is that RMLP type networks are to date unable to encode graphs in an unsupervised fashion. This problem has been recognized, and research into solving this issue is presented as a challenge for future research.
- Another problem with this type of networks is that the number of hidden (state) layer neurons can only be obtained through trial and error. In practical terms, time and expertise is required to find an appropriate MLP architecture for a given learning problem. A model which aims at solving this problem is given in the following chapter.

Chapter 7

Cascade Correlations

Cascade Correlation (CC) is a method which builds a feedforward neural network through construction [23]. With Cascade Correlation we do not have the problem of finding the number of hidden neurons required for a given learning task in a trial and error approach. Originally, CC type networks were unable to process data other than fixed sized vectors and sequences. In [8, 82], a version of cascade correlation, called recursive cascade correlation (RCC), is applied to processing data structures with considerable success.

In this chapter, we will first describe the training algorithm of a cascade correlation, before extending it to the situation of data structures. This is necessary because, as far as we are aware, there is not yet any useful exposition of cascade correlation in the format which we seek. Secondly the formulation would be useful as a precursor for applying cascade correlation on data structures.

As far as we are aware, there has never been any comparison made between the cascade correlation, and the multilayer perceptron applied to data structure problems. In this chapter, we wish to consider the following questions:

1. What are the differences in performance between cascade correlation neural networks, and multilayer perceptron as applied to data structures.
2. Comparing the two architectures, are there other related architectures which might be useful to data structure problems.
3. How do they perform on a series of practical problems.

This chapter gives answers to these questions and defines two new architectures which allow a more thorough comparison of architectures based on MLP and CC.

The structure of this chapter is as follows: In Section 7.1, we will give a thorough description of the cascade correlation (CC) neural network architecture [1]. Section 7.2 extends the theory employed in CC to derive a data structure representational model known as a RCC (recursive cascade correlation) model and gives the associated training algorithm. Experiments with RCC are presented in Section 7.3. Based on the materials presented in Section 6.2.2, and Section 7.2, we will give a number

[1] Here we will abuse the terminology by calling a neural network model resulting from a cascade correlation constructive training algorithm as a cascade correlation neural network model.

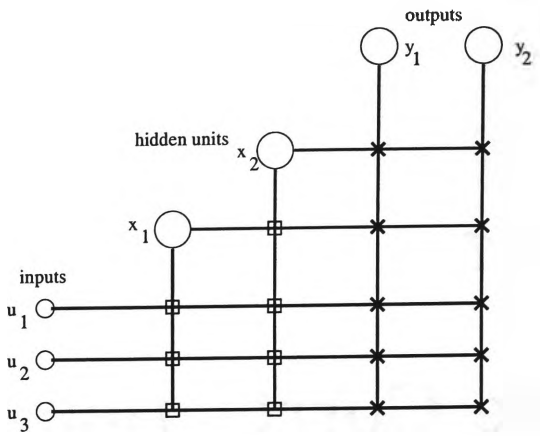


Figure 7.1: A diagram showing a cascade correlation neural network architecture with two hidden units, three input units, and two output units. The hidden units have nonlinear activation functions. The output units either have nonlinear activations, for classification problems; or linear activations, for regression problems. The input units all have linear activation functions. The square symbol denotes that the weights of the unit, once obtained will be frozen. The cross symbol denotes weights which are required to be trained.

of extended neural network architectures which can be used for data structure modelling in Section 7.4. It is proposed to reduce an RCC architecture in Section 7.4.1 and to extend the RMLP architecture in Section 7.4.2. Experimental results with extended RMLP are given in Section 7.5 whereas experimental results with reduced RCC neural models are given in Section 7.6. Finally, a summary and conclusions are drawn in Section 7.7.

7.1 Introduction to Cascade Correlations

The cascade correlation neural network architecture, as introduced by Fahlman [23], shown in Figure 7.1, can be described by the following model [2] :

$$\mathbf{y} = F_p\left(C\mathbf{x} + D\mathbf{u}\right) \tag{7.1}$$

$$\mathbf{x} = F_n\left(A\mathbf{x} + B\mathbf{u}\right) \tag{7.2}$$

where $\mathbf{u}$ is a m-dimensional input, $\mathbf{x}$ is an n-dimensional vector denoting the hidden layer neuron activations, and $\mathbf{y}$ is a p-dimensional output. A, B, C, and D are respectively $n \times n$, $n \times m$, $p \times n$ and $p \times m$ matrices.

Note that A has a special structure: it is a lower triangular matrix with zero diagonal elements. Note that while this has a similar form to the recurrent neural networks [91], in actual fact, it is not. With the lower triangular structure, it indicates that successive elements of $\mathbf{x}$ are influenced by the predecessor values of $\mathbf{x}$. In other words, x_i depends on x_j, $j = 1, 2, \ldots, i-1$, where x_i is the i-th element in the vector $\mathbf{x}$.

It is noted that the cascade correlation architecture has $nm + np + pm + \frac{1}{2}n(n-1) = p(m+n) + nm + \frac{1}{2}n(n-1)$ parameters. Thus, the cascade correlation architecture

[2] The exposition given in this section, though not containing any new results, is new. It forms the background for the derivation of training algorithm in Section 7.2.

has more parameters than the MLP with the same number of hidden layer neurons. Specifically, it has $\frac{1}{2}n(n-1)$ parameters more than the corresponding MLP case. These extra parameters are contributed by the matrix A, i.e., the additional connections from the preceding neurons to the succeeding neurons in the hidden layer.

It is known that the MLP architecture is a universal approximator [52], i.e., it can approximate a given input-output set of data arbitrarily well, provided that a sufficient number of hidden layer neurons is used. It is curious what effect would the additional parameters $\frac{1}{2}n(n-1)$ have on the performance of the cascade correlation architecture compared with the MLP architecture. This will be one of the questions we wish to answer in this chapter.

It is also known that the cascade correlation architecture is not a universal approximator. However, it is known that the architecture is good in practice in classification as well as in regression problems. Hence, it can be anticipated that the cascade correlation architecture as applied to the structured data domain will not have a universal approximator property.

7.1.1 Training algorithm

The training of a cascade correlation architecture is by construction, i.e., it builds the architecture step by step, by adding one hidden layer neuron one at a time. Since it is a feedforward neural network framework, it has to solve the issue of how to train hidden layer neurons without information concerning the hidden layer neuron behaviours. In this instance, Fahlman [23] used two ideas:

(a) Use a number of candidate units [3]. Candidate units are units which can be used as intermediate devices in the training process. Their use is associated with the correlation measure described in (b).

(b) Use the correlation between the output of the hidden layer neuron and the error, as obtained from a **previous** iteration as a measure of how well the candidate units perform in modelling the residual errors. The correlation gives a criterion in the choice of the candidate units. The candidate unit with the highest correlation will be selected to be the hidden layer neuron. The other candidate units are discarded.

We will consider the training of a cascade correlation neural network step by step as follows:

Step 1 Given a training data set $T = \{\mathbf{t}(i), \mathbf{u}(i), i = 1, 2, \ldots, N\}$. $\mathbf{t}$ and $\mathbf{u}$ are respectively p-dimensional vector denoting the target values and m-dimensional vector denoting the inputs. At the beginning of the training process, we assume a cost criterion:

$$J = \frac{1}{2}\sum_{i=1}^{N} tr\left((\mathbf{t}(i) - \mathbf{y}(i))\,(\mathbf{t}(i) - \mathbf{y}(i))^T\right) \tag{7.3}$$

where $tr(\cdot)$ is the trace operator of a matrix and the superscript T denotes the transpose of a vector or a matrix. Note that the initial architecture is an input-output architecture:

[3] Candidate units differ from one another in the initial conditions only. Otherwise they are identical.

$$\mathbf{y} = F_p(D_0\mathbf{u}) \tag{7.4}$$

where D_0 is a $p \times m$ matrix. The unknown matrix D_0 can be trained by minimising the cost function J. This can be obtained by differentiating the cost function J with respect to the elements of matrix D_0. It can be easily proven that the training algorithm is given by the following equations:

$$D_0(i+1) = D_0(i) - \eta \frac{\partial J}{\partial D_0} \tag{7.5}$$

where $D_0(i)$ is the i-th iterate of the matrix D_0, η is a learning constant, and

$$\frac{\partial J}{\partial D_0} = -\sum_{i=1}^{N} \Lambda(\zeta)\mathbf{e}(i)\mathbf{u}(i)^T \tag{7.6}$$

where $\mathbf{e}(i) = \mathbf{t}(i) - \mathbf{y}(i)$ is a p-dimensional vector denoting the error for the i-th training vector. $\Lambda(\zeta)$ is a $p \times p$ diagonal matrix, of the following form:

$$\Lambda(\zeta) = \begin{bmatrix} f'(\zeta_1) & 0 & 0 & \dots & 0 \\ 0 & f'(\zeta_2) & 0 & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & f'(\zeta_p) \end{bmatrix} \tag{7.7}$$

where $f'(\cdot)$ denotes the derivative of the nonlinear function $f(\cdot)$. $\zeta = D_0\mathbf{u}$. ζ_i denotes the i-th element in the p-dimensional vector ζ.

Note that this is a batch training algorithm for a multilayer perceptron without any hidden layers. Hence, the computation should be very fast.

We store the error vector $\mathbf{e}(i)$ for the current computational step. This will be used in Step 2. Furthermore, we denote the average of this vector by $\bar{\mathbf{e}}$. In other words,

$$\bar{\mathbf{e}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{e}(i) \tag{7.8}$$

Step 2 In the second step, there are two sub–steps:

1. Train candidate units. Select the one which is most correlated with the previous error as the hidden unit.
2. Compute the weights from the hidden unit to the output.

We use a number of candidate units. The difference between the candidate units lies in the difference in initial conditions. These units all have direct connections to the input vector $\mathbf{u}$. Hence, the architecture of each candidate unit is as follows:

$$x = f\left(\mathbf{b}_1^T\mathbf{u}\right) \tag{7.9}$$

where $\mathbf{b}_1$ is a m-dimensional vector. Note that x is a scalar, denoting the output of the candidate unit.

Since we do not have any output to train this unit, we cannot use J. Instead we use the following correlation measure:

$$J_c = \sum_{j=1}^{p} \left| \sum_{i=1}^{N} (x(i) - \overline{x(i)})(e_j(i) - \overline{e_j(i)}) \right| \tag{7.10}$$

where $x(i)$ is the output of x in response to the i-th input, and $e_j(i)$ is the j-th error in response to the i-th training input in the **previous** cycle, i.e., in step 1. $\overline{x}$ denotes the average value of x over the training data set. (Eq. 7.10) can be written more compactly in the following form:

$$J_c = \sum_{j=1}^{p} \left| tr\left((\mathbf{e}_j^N - \overline{\mathbf{e}_j^N})(\mathbf{x}^N - \overline{\mathbf{x}^N})^T \right) \right| \tag{7.11}$$

where $\mathbf{x}^N = \begin{bmatrix} x(1) \\ x(2) \\ \vdots \\ x(N) \end{bmatrix}$, $\overline{\mathbf{x}^N} = \begin{bmatrix} \overline{x} \\ \overline{x} \\ \vdots \\ \overline{x} \end{bmatrix}$. $\mathbf{e}_j^N$, and $\overline{\mathbf{e}_j^N}$, $j = 1, 2, \ldots, p$ are defined similarly.

$\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x(i)$, the average of the output of the candidate unit. $\overline{e}$ is defined similarly.

This is a simple one layer network, and can be trained by maximising the value of J_c.

$$\mathbf{b}_1(i+1) = \mathbf{b}_1(i) + \eta_c \frac{\partial J_c}{\partial \mathbf{b}_1} \tag{7.12}$$

where η_c is a learning constant. Note that in general, $\eta_c \neq \eta$. Furthermore,

$$\frac{\partial J_c}{\partial \mathbf{b}_1^T} = \sum_{j=1}^{p} \text{sign}(tr\,(Z(j))) \left(\Lambda(\zeta^N)(\mathbf{e}_i^N - \overline{\mathbf{e}^N})(\mathbf{u}^N)^T \right) \tag{7.13}$$

where $Z(j) = (\mathbf{e}_j^N - \overline{\mathbf{e}_j^N})(\mathbf{x}^N - \overline{\mathbf{x}^N})^T$ and $\text{sign}(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0 \end{cases}$. $\Lambda(\zeta^N)$ is a $N \times N$ diagonal matrix defined as follows:

$$\Lambda(\zeta^N) = \begin{bmatrix} f'(\zeta(1)) & 0 & 0 & \ldots & 0 \\ 0 & f'(\zeta(2)) & 0 & \ldots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \ldots & f'(\zeta(N)) \end{bmatrix} \tag{7.14}$$

$\zeta = \mathbf{b}_1^T \mathbf{u}$, a scalar variable. $\zeta(i)$ denotes the i-th instant of the scalar ζ.

$\mathbf{u}^N$ is a Nm vector defined as follows:

$$(\mathbf{u}^N)^T = [\ u_1(1) \ \ldots \ u_1(N) \ \ u_2(1) \ \ldots \ u_2(N) \ \ldots \ u_m(1) \ \ldots \ u_m(N)\ ]^T$$

We train a number of candidate units; each candidate unit differs from the other only in the initial conditions. We find the best suitable candidate unit by selecting the one with the largest correlation measure. This will be the first hidden unit. All the other candidate units are discarded. Once this is chosen, we fix the weight of the vector $\mathbf{b}_1$. We denote the output of the first hidden unit by x_1.

Once $\mathbf{b}_1$ is fixed, then we can train the weights of the following architecture:

$$\mathbf{y} = F_p\left(D_1\mathbf{u} + \mathbf{c}_1 x_1\right) \tag{7.15}$$

where $\mathbf{c}_1$ is a p-dimensional vector. Note that $D_1 \neq D_0$.

Since the incoming weights to x_1 are fixed, x_1 can be computed for each input training pattern. Hence it is simple to treat x_1 as though it is an input. Thus, we can use the single layer training algorithm to find the unknown matrix D_1, and $\mathbf{c}_1$.

$$\left[\begin{array}{cc} D_1(i+1) & \mathbf{c}_1(i) \end{array}\right] = \left[\begin{array}{cc} D_1(i) & \mathbf{c}_1(i) \end{array}\right] - \eta \frac{\partial J}{\partial \left[\begin{array}{cc} D_1 & \mathbf{c}_1 \end{array}\right]} \tag{7.16}$$

where

$$\frac{\partial J}{\partial \left[\begin{array}{cc} D_1 & \mathbf{c}_1 \end{array}\right]} = \sum_{i=1}^{N} \Lambda(\zeta)\mathbf{e}(i) \left[\begin{array}{cc} \mathbf{u}^T(i) & x_1(i) \end{array}\right] \tag{7.17}$$

where $\zeta = D_1\mathbf{u} + \mathbf{c}_1 x_1$, a p-dimensional vector. Once D_1 and $\mathbf{c}_1$ are found, we can compute the corresponding output errors $\mathbf{e}$. Again we store the error $\mathbf{e}(i) = \mathbf{t}(i) - \mathbf{y}(i)$ for use in Step 3.

Step 3 Now the way ahead is clear. We can use the error sequence $\mathbf{e}$ obtained in Step 2 to train another set of candidate units, each candidate unit has the following architecture:

$$x = f\left(\mathbf{b}_2^T\mathbf{u} + a_1 x_1\right) \tag{7.18}$$

where $\mathbf{b}_2$ is a m-dimensional vector, and a_1 is an unknown constant.

Now, x_1 is known, and hence it is the same as an input. Thus again we can find the unknown vector $\mathbf{b}_2$, and the unknown constant a_1 by maximising the correlation measure (see (Eq. 7.10) and (Eq. 7.11)).

The unknown vector $\mathbf{b}_2$ and constant a_1 can be found by maximising the following correlation measure.

$$\left[\begin{array}{cc} \frac{\partial J_c}{\partial \mathbf{b}_2^T} & \frac{\partial J_c}{\partial a_1} \end{array}\right] = \sum_{i=1}^{p} \text{sign}(z(j))\Lambda(\xi^N)(\mathbf{e}_i^N - \overline{\mathbf{e}^N}) \left[\begin{array}{cc} (\mathbf{u}^N)^T & (\mathbf{x}^N)^T \end{array}\right] \tag{7.19}$$

where $\xi = \mathbf{b}_2^T\mathbf{u} + a_1 x_1$.

We choose the candidate unit with the largest correlation measure and discard all the other candidate units. Once that is chosen, we fix the incoming weights $\mathbf{b}_2$, and a_1 to the neuron. This will be the second hidden layer neuron. Denote this by x_2.

The output will then be given by:

$$\begin{aligned} \mathbf{y} &= F_p\left(D_2\mathbf{u} + \mathbf{c}_1 x_1 + \mathbf{c}_2 x_2\right) \\ &= F_p\left(D_2\mathbf{u} + \begin{bmatrix} \mathbf{c}_1 & \mathbf{c}_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) \\ &= F_p\left(D_2\mathbf{u} + C\mathbf{x}\right) \end{aligned} \tag{7.20}$$

where $C = [\ \mathbf{c}_1 \quad \mathbf{c}_2\ ]$, $p \times 2$ matrix.

Once again the unknown matrix D_2 and unknown vectors $\mathbf{c}_i$, $i = 1, 2$ can be found by minimising the cost criterion J.

$$\begin{bmatrix} \frac{\partial J}{\partial \mathbf{b}_2} & \frac{\partial J}{\partial c_1} & \frac{\partial J}{\partial c_2} \end{bmatrix} = \sum_{i=1}^{N} \Lambda(\zeta)\mathbf{e}(i) \begin{bmatrix} \mathbf{u}^T(i) & x_1(i) & x_2(i) \end{bmatrix} \tag{7.21}$$

This is again a single layer training algorithm, as x_i, $i = 1, 2$ can be considered as inputs.

Step 4 We can follow the same procedures as carried out in Step 3.

Using this method recursively, we can find the cascade correlation neural network architecture which can represent the given set of input-output training data.

Here in this section, we have formulated the classification problem. Hence, the stopping criterion used would be either when all the outputs are correctly classified, or that there is no change in the classification error.

The formulation can be modified easily for regression problems. In this case, instead of the output equation:

$$\mathbf{y} = F_p\left(C\mathbf{x} + D\mathbf{u}\right) \tag{7.22}$$

we assume a linear output neuron, i.e.,

$$\mathbf{y} = C\mathbf{x} + D\mathbf{u} \tag{7.23}$$

The training algorithm can be modified suitably for this case to constructively find an architecture which can be used for regression analysis.

7.2 Cascade Correlation applied to Tree Data Structures

From the observation made in the MLP tree architecture section, we can treat the inputs from the children as extra inputs to the current node, and the exposition of the cascade correlation in the previous section, it is quite simple to modify the approach so that it can be applied to model tree architectures [4].

Each node in the tree architecture can be modelled as follows:

$$\mathbf{x} = F_n\left(B\mathbf{u} + Aq^{-1}\mathbf{x}\right) \tag{7.24}$$

[4]The training algorithm presented here is similar to the ones given in [8]. However, the way how it is derived is new.

The output of the root node is given by

$$\mathbf{y} = F_p\left(C\mathbf{x} + D\mathbf{u}\right) \tag{7.25}$$

where $Aq^{-1}\mathbf{x}$ is a shorthand form for denoting the inputs from the children to the current node. Note that this model is similar to the state MLP tree architecture. The matrix A is constructed in a step by step manner.

The recursive cascade correlation algorithm can be stated as follows:

Step 1 Since we are dealing with a tree architecture, it does not make sense to build first a direct input-output model as in the classic cascade correlation situation. Hence in the first step, we just assume that the outputs to be zero. The error sequence in this case becomes $\mathbf{t}(i)$, $i = 1, 2, \ldots, N$. This error sequence will be used in Step 2 in the determination of the correlation measure.

Step 2 We assume that the candidate unit in each node can be described by the following model:

$$x = f\left(\mathbf{b}^T\mathbf{u} + \mathbf{a}q^{-1}\mathbf{x}\right) \tag{7.26}$$

where $\mathbf{a}q^{-1}\mathbf{x}$ is a shorthand form denoting:

$$\mathbf{a}q^{-1}\mathbf{x} = \begin{bmatrix} a_1 & a_2 & \ldots a_c \end{bmatrix} \begin{bmatrix} q_1^{-1}x_1 \\ q_2^{-1}x_2 \\ \vdots \\ q_c^{-1}x_c \end{bmatrix} \tag{7.27}$$

where a_i are unknown constants. x_i corresponds to the hidden layer neuron in the i-th child, and q_i^{-1} is a device to denote the input from the i-th child to the current node.

Since we do not have the output available at the intermediate nodes, and hence we use the correlation measure:

$$J_c = \sum_{j=1}^{p} \left| \sum_{i=1}^{N} (x(i) - \overline{x(i)})(e_j(i) - \overline{e_j}) \right| \tag{7.28}$$

This correlation criterion can be maximised with respect to the unknown parameters $\mathbf{b}$, and $\mathbf{a}$. The updating rules are as follows:

$$\mathbf{b}(i+1) = \mathbf{b}(i) + \eta_c \frac{\partial J_c}{\partial \mathbf{b}} \tag{7.29}$$

and

$$\mathbf{a}(i+1) = \mathbf{a}(i) + \eta_c \frac{\partial J_c}{\partial \mathbf{a}} \tag{7.30}$$

The partial derivatives of the correlation measure with respect to the parameters can be obtained as follows: let the k-th element of $\mathbf{b}$ be denoted by b_k. Then,

$$\frac{\partial J_c}{\partial b_k} = \sum_{j=1}^{p} \text{sign}(z_j) \sum_{i=1}^{N} (e_j(i) - \overline{e_j}) \frac{\partial x(i)}{\partial b_k} \tag{7.31}$$

and we have

$$\frac{\partial x(i)}{\partial b_k} = f'(\zeta) \left(Q_k \mathbf{u}(i) + \mathbf{a} q^{-1} \frac{\partial x}{\partial b_k} \right) \tag{7.32}$$

where $\zeta = \mathbf{b}^T \mathbf{u} + \mathbf{a} q^{-1} \mathbf{x}$. Q_k is a m-dimensional vector with all elements zero except a value of 1 in the k-th position. The derivative $q^{-1} \frac{\partial x}{\partial b_k}$ is dependent on the tree structure. $z_j = \sum_{i=1}^{N} (x(i) - \overline{x(i)})(e_j(i) - \overline{e_j})$. The derivatives of $\overline{x(i)}$ with respect to b_k are obtained accordingly. Then, in a similar manner, we have:

$$\frac{\partial J_c}{\partial a_k} = \sum_{j=1}^{p} \text{sign}(z_j) \sum_{i=1}^{N} (e_j(i) - \overline{e_j}) \frac{\partial x(i)}{\partial a_k} \tag{7.33}$$

and

$$\frac{\partial x(i)}{\partial a_k} = f'(\zeta) \left(\mathbf{a} q^{-1} \frac{\partial \mathbf{x}}{\partial a_k} + Q_k q^{-1} \mathbf{x} \right) \tag{7.34}$$

Q_k is a n-dimensional vector with all elements 0 except a value of 1 in the k-th position. Again the evaluation of the derivative $q^{-1} \frac{\partial \mathbf{x}}{\partial a_k}$ is dependent on the tree structure, and the derivatives of $\overline{x(i)}$ with respect to a_k are obtained accordingly.

The candidate unit with the maximum correlation will be chosen as the hidden unit and the other candidate units are discarded. Once the unit is chosen, the parameters **b** and **a** are frozen.

Then we will determine the values of the output by minimising the error criterion. The output model is given by:

$$\mathbf{y} = F_p(\mathbf{c}x + D\mathbf{u}) \tag{7.35}$$

The unknowns **c**, a p-dimensional vector, and the $p \times m$ matrix D can be determined by using the error criterion. Now x is fixed once the weights **b** and **a** are frozen. Hence x can be considered as an extra input in the single layer network. Thus, the unknown can be trained readily.

We then compute the error sequence $\mathbf{e}(i) = \mathbf{t}(i) - \mathbf{y}(i)$, for $i = 1, 2, \ldots, N$. This will be used in step 3.

Step 3 This again can be decomposed into two sub-steps, one for finding a suitable candidate unit so that the number of hidden units is increased by one; and then find the weights for the output layer.

We can use the following model for the candidate unit:

$$x = f\left(\mathbf{b}^T \mathbf{u} + a_1 x_1 + \mathbf{r}^T q^{-1} \mathbf{x}_1 + \mathbf{s}^T q^{-1} \mathbf{x}\right) \tag{7.36}$$

where x_1 is the output of the hidden unit chosen in Step 2; a_1 is the corresponding weight connecting this to the candidate unit. $\mathbf{x}_1$ are the output of the children from the hidden unit as determined in Step 2. $\mathbf{r}$ is a vector for weights connecting the candidate unit with the first hidden unit (as determined in Step 2) of each child node. $\mathbf{s}^T q^{-1}\mathbf{x}$ is a shorthand form denoting the following:

$$\mathbf{s}^T q^{-1}\mathbf{x} = \begin{bmatrix} s_1 & s_2 & \cdots & s_c \end{bmatrix} \begin{bmatrix} q_1^{-1}x_1 \\ q_2^{-1}x_2 \\ \vdots \\ q_c^{-1}x_c \end{bmatrix} \tag{7.37}$$

where x_k denotes the output of the candidate unit in the k-th child. The unknown parameters a_1, $\mathbf{r}$, $\mathbf{s}$ and $\mathbf{b}$ can be found by maximising the correlation measure (Eq. 7.28).

The weight a_1 can be updated as follows:

$$\frac{\partial J_c}{\partial a_1} = \sum_{j=1}^{p} \text{sign}(\xi) \sum_{i=1}^{N} \left((e_j(i) - \overline{e_j}) \frac{\partial x}{\partial a_1} \right) \tag{7.38}$$

$$= \sum_{j=1}^{p} \text{sign}(\xi) \left((e_j(i) - \overline{e_j}) f'(\xi) x_1 \right) \tag{7.39}$$

Let r_k denote the k-th element in the c-dimensional vector $\mathbf{r}$. Then we have

$$\frac{\partial J_c}{\partial r_k} = \sum_{j=1}^{p} \text{sign}(\xi) \sum_{i=1}^{N} \left((e_j(i) - \overline{e_j}) \frac{\partial x}{\partial r_k} \right) \tag{7.40}$$

We then have:

$$\frac{\partial x}{\partial r_k} = f'(\xi) \left(Q_k q^{-1}\mathbf{x}_1 + \mathbf{s}^T q^{-1} \frac{\partial \mathbf{x}}{\partial r_k} \right) \tag{7.41}$$

where Q_k is a c-dimensional vector with all elements 0 except a value of 1 in the k-th position. The evaluation of this quantity is dependent on the tree structure. In a similar manner, let s_k be the k-th element in the vector $\mathbf{s}$, we can obtain

$$\frac{\partial J_c}{\partial s_k} = \sum_{j=1}^{p} \text{sign}(z_j) \sum_{i=1}^{N} (e_j(i) - \overline{e_j}) \frac{\partial x}{\partial s_k} \tag{7.42}$$

and

$$\frac{\partial x}{\partial s_k} = f'(\xi) \left(Q_k q^{-1}\mathbf{x} + \mathbf{s}^T q^{-1} \frac{\partial \mathbf{x}}{\partial s_k} \right) \tag{7.43}$$

Let b_k be the k-th element in the vector $\mathbf{b}$, then we have

$$\frac{\partial J_c}{\partial b_k} = \sum_{j=1}^{p} \text{sign}(z_j) \sum_{i=1}^{N} (e_j(i) - \overline{e_j}) \frac{\partial x}{\partial b_k} \tag{7.44}$$

and

$$\frac{\partial x}{\partial b_k} = f'(\xi) \left(Q_k^T \mathbf{u} + \mathbf{s}^T q^{-1} \frac{\partial \mathbf{x}}{\partial b_k} \right) \tag{7.45}$$

Thus using these equations, the unknown parameters can be updated. Then we choose the candidate unit which maximises the correlation measure. We then freeze the weights which are associated with this unit. We denote the output of the second hidden unit as x_2.

Then, we use the following output model:

$$\mathbf{y} = F_p \left(D\mathbf{u} + \mathbf{c}_1 x_1 + \mathbf{c}_2 x_2 \right) \tag{7.46}$$

Since both x_1 and x_2 can be evaluated, they can be treated as though they are additional inputs for the output layer. Thus the unknown weights D, $\mathbf{c}_1$, and $\mathbf{c}_2$ can be obtained by minimising the error criterion.

The way ahead is clear from here. We can progressively add hidden units, one at a time, until the error criterion is small.

The resulting architecture from this constructive process will be called a recursive cascade correlation (RCC) model.

It is noted that the RCC architecture has $nm + np + \frac{1}{2}n(n-1) + \frac{1}{2}cn(n+1)$ parameters as compared to $nm + np + cn^2$ parameters in an RMLP network. Thus, depending on the outdegree of the graphs and the number of hidden neurons, the cascade correlation architecture has a different number of parameters to that of the RMLP. Specifically, the difference is $\frac{n}{2}(n - cn + c - 1)$, and hence, the number of parameters in an RCC is greater than the number of parameters in an RMLP if $c < \frac{n-1}{1-n}$. Because n is never negative and because values for c must be cardinal, it is noted that in practical terms, RCC always features less parameters than a state MLP network architecture that has the same number of hidden layer neurons unless c is zero. In the case where $c = 0$, the RCC reduces to CC and features more parameters than an MLP network.

The RCC model has been tried on a benchmark problem. Many of the experimental findings are given in Section 7.3.

7.3 Experimental results with RCC

This section reports some experience in applying the RCC architecture to the benchmark problem dataset-3 which is described in Appendix A. The task of this benchmark problem is to classify all 3750 graphs in the training/test set correctly. 12 classes are defined over the dataset. The dimension of the data label $\mathbf{u}$ is 2, the dimension of the binary target vector $\mathbf{t}$ is 12, and the maximum out-degree c is 6.

Experimenting with RCC is easier than with the corresponding RMLP architecture. This is because the algorithm is able to find an appropriate number of hidden nodes by itself. With RMLP, an appropriate choice of the number of hidden nodes can

only be found through a trial and error process. As a result, the training of an RCC network requires only few parameters. The only parameters which we can play with are (1) the type of updating rule, and (2) the number of candidate units. However, preliminary training sessions with RCC indicated that the training of an RCC architecture requires considerably more time than the training of a corresponding RMLP network. A great contributor to the training time requirements is the complexity of the training algorithm which is found to be greater than that of the RMLP algorithm. This difference between these two models will be discussed further in Section 7.6.2. Other reasons for the increased training time requirements are that the output layer has to be re-trained every time a new hidden layer is added to the network. In addition, before a hidden neuron can be added, n candidate units have to be trained. Only the best candidate is chosen whereas the others are discarded. Training an RCC network can take magnitudes longer than the training of RMLP networks so that the utilization of a relatively large set of data such as dataset-3 quickly exceeds reasonable training times. In response to this, we restricted our experiments to training sessions with the RProp learning rule since RProp has been found to be the most efficient updating mechanism for this type of networks. Also, we chose to set the size of the pool of candidate neurons for the hidden layer evaluations to be 15. A larger pool did not appear to have a significant effect on the final network performance. However, it was observed that when choosing a pool size of 8 or smaller, the networks obtained performed poorly.

The first experiment is to demonstrate how the network performance evolves as hidden layer neurons are added to the system. This is shown if Figure 7.2.

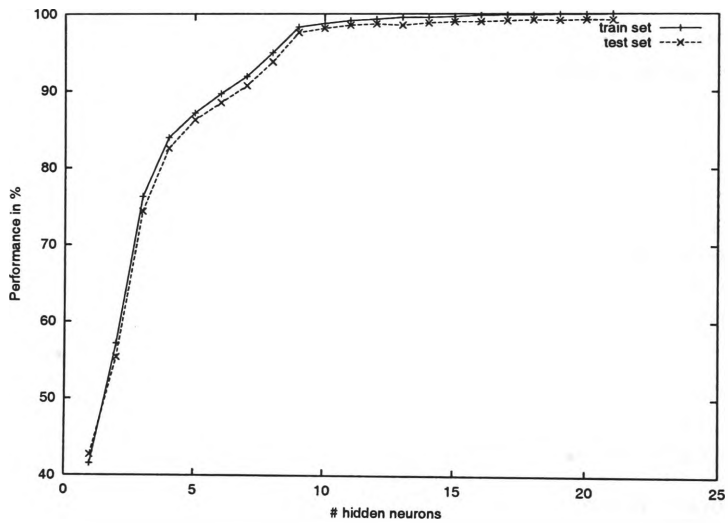


Figure 7.2: Training RCC on the benchmark problem: dataset-3. The number of hidden neurons is plotted against the network performance.

Figure 7.2 gives the percentage of correctly classified patterns from the training/test set in relation to the number of hidden layer neurons added to the RCC network. Two important findings are associated with this result:

- RCC provides an effective mechanism for the encoding of graph structured information.
- The algorithm found that at least 9 hidden layer neurons are needed for a good performance on the given learning learning task.

This is an interesting result because earlier findings with state MLP networks suggested that 10 hidden neurons are required for this learning task. An RCC with 9 hidden nodes has 306 network parameters less than a state MLP with 10 hidden nodes. An explanation for this observation is found in Section 7.6.2 where an appropriately modified version of a state MLP network is compared with RCC, and also in Section 7.6.1 where the effects of a modified RCC architecture is investigated. Note that training the RCC was interrupted after 21 hidden neurons were added because training times exceeded reasonable levels (more than 20 days of clock time). At that stage, the RCC with 21 hidden layer neurons classified only one training pattern and 30 test patterns incorrectly.

In Figure 7.3 we present the confusion matrices as obtained from a RCC network featuring 10 neurons in the hidden layer. We use an RCC with 10 hidden layer neurons to allow a direct comparison with earlier findings from RMLP network experiments. It is found that the classification performance is reasonable well balanced over the different classes. This is a positive result when considering that some classes are represented only by relatively few samples while other classes have a large number of samples. In comparison with findings made with a state MLP architecture previously in Figure 6.2, it is found that the overall performance on an RCC architecture is slightly better despite the fact that the RCC architecture is featuring fewer network parameters. The most significant difference is that RCC has no apparent problems with classes that are represented by only few training patterns. State MLP networks had considerable difficulties with such cases. This may be attributed to the fact that the RCC models have direct input output connections, while the state MLP models, at least in the experiments which we have conducted, do not have direct input output connections [5]. It is known that a multilayer perceptron with direct input output connections can outperform ones without direct input output connections.

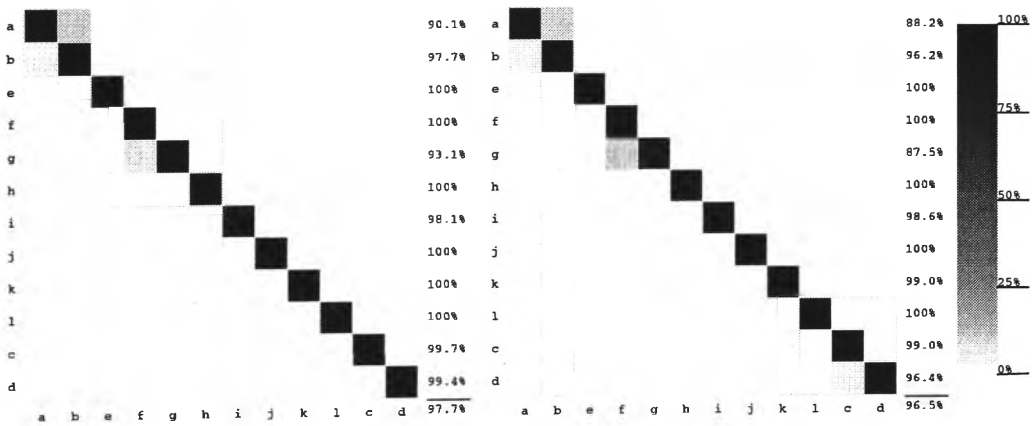


Figure 7.3: Confusion matrix as obtained from a RCC with 10 hidden neurons. The table on the left hand side refers to the training set, the table on the right hand side refers to the test set.

The following experiment is to investigate the impact of training an RCC network on a reduced set of training data. Preliminary experiments showed that RCC generalizes well on this benchmark problem. The question we would like to ask is: how many training pattern are required by the RCC algorithm in order to obtain a good

[5] In the models given in Chapter 6, we have made provisions that it is possible to include direct input output connections. However, the programming of such inclusion is far more complex.

generalization performance. The result of this experiment is shown in Figure 7.4.

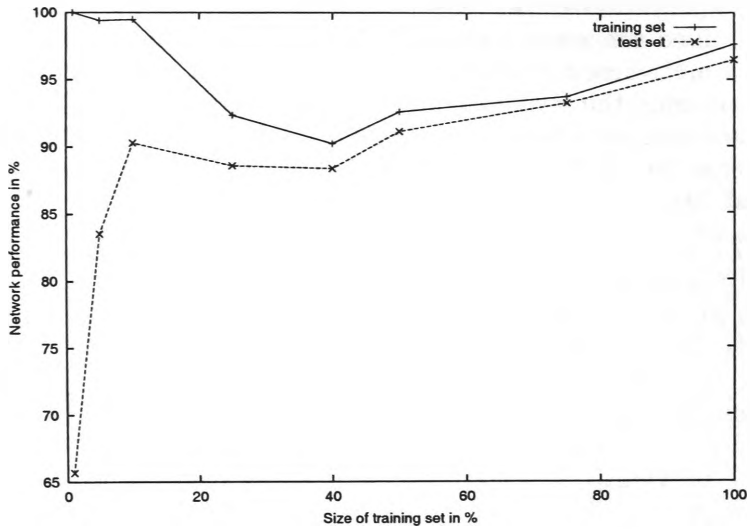


Figure 7.4: Network performance of a RCC model featuring 10 hidden nodes plotted against the number of training patterns.

The interesting observation is that RCC requires at least half of the training samples in dataset-3 to generalize well. This is in contrast to earlier findings with state MLP networks which performed well with about 10% of training patterns from dataset-3. An explanation of this particular property can be surmised intuitively as follows: in the MLP type of networks, the supervisor signal is present at the output of the network. This manifests itself in the formulation of the error signal which is then used in the update of the network weights. On the other hand, for the RCC type of models the determination of the hidden layer neurons is achieved through a correlation measure. This is not as strong as the error measure in the MLP type models. Hence, it is surmised that it will take the network much more training samples to achieve similar performance as the MLP type models. In addition, choosing more than the 50% of training data from dataset-3 helps to improve the general performance of the RCC network.

This section made attempts to compare the performance of RCC networks with that of the state RMLP networks that feature the same number of hidden neurons. However, such a comparison is not sufficiently expressive because the network architectures differ despite the many similarities between these two models. One major difference is the connections between hidden layer neurons which are not present in the case of state MLP models. The second difference is that recursive connections (the matrix A) is not fully connecting hidden layer neurons with all states from the processing of children nodes in the case of RCC. Hence, the number of network weights is different for the two models even if the number of hidden layer neurons is the same. Moreover, the interconnections between hidden nodes have the effect that nodes added earlier during the training process act like hidden layer neurons at different levels. This too can have a significant impact on experimental results. As a consequence, we suggest two slightly modified architectures for RCC and state MLP models which allow a better comparison of the two models, and help to clarify what impacts the additional or missing links among the hidden layer neurons in a network model can have. This is addressed in the following section.

7.4 Extended classes of architecture

This section proposes a modification of the MLP and CC type networks respectively. These modifications are first explained on classic models before applying the underlying ideas to the more general data structure models [6].

From a comparison between the MLP, and the cascade correlation neural network architectures, it is clear that we can formulate other neural network architectures within these classes. The only difference between the cascade correlation neural network architecture and the MLP is the $\frac{1}{2}n(n-1)$ weights from the preceding hidden units to the succeeding units in the case of the cascade correlation architecture. Based on this observation, it is possible to formulate two novel neural network architectures:

7.4.1 Reduced cascade correlation

This architecture is obtained by disallowing the connections from the previous hidden layer units to the succeeding hidden layer units. In other words, we have the following architecture:

$$\mathbf{y} = F_p(C\mathbf{x} + D\mathbf{u}) \tag{7.47}$$

$$\mathbf{x} = F_n(B\mathbf{u}) \tag{7.48}$$

This will be equivalent to the multilayer perceptron architecture, except that the training algorithm is a step by step constructive training algorithm, in the sense that the hidden units are found one by one, instead of being the usual backprop training algorithm, in which all the hidden units are present, as in the case of the multilayer perceptron situation.

The step by step training algorithm can be described as follows: Given a training data set $T = \{\mathbf{u}(i), \mathbf{t}(i); i = 1, 2, \ldots, N\}$, we wish to find a MLP architecture which will represent this set of training data.

Step 1 We form a direct input-output model as follows:

$$\mathbf{y} = F_p(D_0\mathbf{u}) \tag{7.49}$$

where D_0 is a $p \times m$ matrix, which can be obtained by minimising the error criterion. Store the error sequence $\mathbf{e}(i) = \mathbf{t}(i) - \mathbf{y}(i)$ for use in Step 2.

Step 2 We use candidate units, each candidate unit is modelled as follows:

$$x = f(\mathbf{b}_1\mathbf{u}) \tag{7.50}$$

where $\mathbf{b}_1$ is a m-vector. The unknown parameter $\mathbf{b}_1$ can be trained by maximising the correlation measure between the previous error sequence $\mathbf{e}(i)$, and the outputs of the candidate unit.

The candidate unit with the largest correlation measure will be selected as the hidden unit and the other candidate units are discarded. The corresponding weight $\mathbf{b}_1$ is frozen. Let us denote the output of the hidden unit by x_1.

[6] The ideas presented in this section are new.

Then, we can use this hidden unit together with the inputs to form the following output model:

$$\mathbf{y} = F_p\left(D_1\mathbf{u} + \mathbf{c}_1 x_1\right) \tag{7.51}$$

where $\mathbf{c}_1$ is a p-dimensional vector. The unknowns D_1 and $\mathbf{c}_1$ can be determined by minimising the error criterion.

Step 3 The way ahead is clear as we can determine the next hidden unit using the correlation measure, and so on until a satisfactory network is obtained giving an acceptable error rate.

For tree architectures applied to structured data domain, we have the following candidate unit model:

$$x = f\left(\mathbf{b}^T\mathbf{u} + \mathbf{s}^T q^{-1}\mathbf{x}\right) \tag{7.52}$$

In this case, the training algorithm used is the step by step method, i.e., the hidden layer units are obtained one at a time. Note that the candidate model is the same as the one used for the state MLP model. Since the training algorithm is quite similar to the one used for the cascade correlation method for data structures, specifically we do not include any parameter $a_i x_i$, as a result, the training algorithm can be obtained by dropping certain terms associated with $a_i x_i$ in the RCC training algorithm. The details of the training algorithm is omitted here.

7.4.2 Extended multilayer perceptron

It is possible to extend the multilayer perceptron to one which allows the preceding hidden layer neurons to influence the succeeding ones, i.e.,

$$\mathbf{y} = F_p\left(C\mathbf{x} + D\mathbf{u}\right) \tag{7.53}$$

$$\mathbf{x} = F_n\left(H\mathbf{x} + B\mathbf{u}\right) \tag{7.54}$$

The $n \times n$ matrix H has a very special structure, viz., it is lower diagonal, with zero elements in the diagonal. Thus even though the form of the architecture appears the same as a recurrent neural network, in actual fact, it is different.

The training algorithm of this architecture can be determined by minimising the error criterion J. Note that the updating equations for matrices B, C and D are the same as those obtained in a MLP. As for the updating equation for matrix H, we need to proceed as follows: consider an element h_{ij} in the matrix H. We can differentiate the error criterion with respect to this parameter as follows:

$$\frac{\partial J}{\partial h_{ij}} = -\sum_{\ell=1}^{N} \delta^T(\ell)\frac{\partial \mathbf{x}}{\partial h_{ij}} \tag{7.55}$$

where $\delta(\ell) = C^T\Lambda(\zeta)\mathbf{e}(\ell)$, and $\zeta = C\mathbf{x} + D\mathbf{u}$.

Now, $\frac{\partial \mathbf{x}}{\partial a_{ij}}$ can be obtained by formally differentiating $\mathbf{x}$ with respect to a_{ij} as follows:

$$\frac{\partial \mathbf{x}}{\partial h_{ij}} = \Lambda(\xi)\left(H\frac{\partial \mathbf{x}}{\partial h_{ij}} + Q_{ij}\mathbf{x}\right) \tag{7.56}$$

where Q_{ij} is a $n \times n$ matrix with all elements 0, except an element of 1 in the i-th row and j-th column position. $\xi = H\mathbf{x} + B\mathbf{u}$. (Eq. 7.56) can be solved as follows:

$$\frac{\partial \mathbf{x}}{\partial h_{ij}} = (I - \Lambda(\xi)H)^{-1} Q_{ij}\mathbf{x} \tag{7.57}$$

The matrix inversion of $I - \Lambda(\xi)H$ is particularly simple, as H is lower triangular with 0 in the diagonal elements. Thus, $I - \Lambda(\xi)H$ is a lower triangular matrix with all diagonal elements equal to 1. Thus, (Eq. 7.57) can be solved for the values of $\frac{\partial \mathbf{x}}{\partial h_{ij}}$. Once this is obtained, the value of $\frac{\partial J}{\partial h_{ij}}$ can be obtained, and hence the elements of H can be updated.

Note that the fact we can determine the unknown matrix H in this simple manner is due to the fact that we only have feedforward links in the hidden layer neurons, i.e., x_j depends on x_i, $i = 1, 2, \ldots, j-1$. If it is a full recurrent neural network, i.e., when H has no special structure, then, while the solution is still possible, as there exist many algorithms for inverting $I - \Lambda(\mathbf{x})H$, the nice insight into the operation of the architecture is lost.

Thus, it can be observed from this that the cascade correlation architecture occupies a place somewhere in between the fully recurrent neural network (with no delays in the interconnections among the hidden layer neurons), and the MLP. It is hence an interesting architecture to explore, even though we know that the MLP architecture by itself is already a universal approximator.

For tree architectures for structured data domain, each node of the extended model can be represented by the following model:

$$\mathbf{x} = F_n\left(H\mathbf{x} + B\mathbf{u} + Aq^{-1}\mathbf{x}\right) \tag{7.58}$$

where H and B are the same as those indicated previously. A is a $n \times nc$ matrix denoting the connections from the children nodes to the current node. In addition for the root node, we have

$$\mathbf{y} = F_p\left(C\mathbf{x} + D\mathbf{u}\right) \tag{7.59}$$

The training algorithm for this model is a combination of the previously described models.

If we let h_{ij} denote the ij-th element of the matrix H, and if the error criterion is a least squared error criterion, then, we have

$$\frac{\partial J}{\partial h_{ij}} = -\sum_{i=1}^{N} \mathbf{e}(i)^T \frac{\partial \mathbf{y}}{\partial h_{ij}} \tag{7.60}$$

and

$$\frac{\partial \mathbf{y}}{\partial h_{ij}} = \Lambda(\zeta)\left(C\frac{\partial \mathbf{x}}{\partial h_{ij}}\right) \tag{7.61}$$

and

$$\frac{\partial \mathbf{x}}{\partial h_{ij}} = \Lambda(\xi)\left(H\frac{\partial \mathbf{x}}{\partial h_{ij}} + Q_{ij}\mathbf{x} + Aq^{-1}\frac{\partial \mathbf{x}}{\partial h_{ij}}\right) \tag{7.62}$$

where $\zeta = C\mathbf{x} + D\mathbf{u}$, $\xi = H\mathbf{x} + B\mathbf{u} + Aq^{-1}\mathbf{x}$. Q_{ij} is an $n \times n$ matrix with all elements 0 except for a value of 1 at the ij-th position. Then, we have

$$\frac{\partial \mathbf{x}}{\partial h_{ij}} = (I - \Lambda(\xi)H)^{-1}\left(Q_{ij}\mathbf{x} + Aq^{-1}\frac{\partial \mathbf{x}}{\partial h_{ij}}\right) \tag{7.63}$$

The inversion of the matrix $I - \Lambda(\xi)H$ is particularly simple, as H is a lower triangular matrix with diagonal elements being 0.

A similar set of updating equation can be found for the other unknown weights. They will be omitted here.

It is noted that both, the extended cascade correlation model and the reduced MLP model can be used to model data structures. An extended state MLP model is of an architecture which is more similar to an RCC network architecture, whereas the reduced RCC model features an architecture which is similar to the one found in RMLP models. Note that while the two models have become more similar, the number of parameters is still different, i.e. a reduced RCC features less weights than an (extended) state RMLP network for all cases where the outdegree is greater or equal to 1.

With these two new models at hand, we are able to conduct quantitative evaluations. The application of the extended RMLP architectures to a benchmark problem, dataset-3, is addressed in Section 7.5 whereas applying the reduced RCC model to dataset-3 is examined in Section 7.6. Particularly interesting are the comparisons made between the those models respectively in Section 7.5.3, Section 7.5.4, and Section 7.6.1, Section 7.6.2.

7.5 Experimental results on extended RMLP

This section reports our experience applying extended RMLP models to a benchmark problem: dataset-3, the Extended Policemen Benchmark problem which is defined in Appendix A. The dataset is an artificial learning problem consisting of 3750 graphs for the training set and 3750 graphs in the test set. The graphs belong to 12 different classes where each class is associated with a 12-dimensional target vector. A data label is associated with each node in the graph, where the dimension of this label is 2. As a consequence, all networks considered for these experiments commonly featured 2 input nodes in the input layer $\mathbf{u}$, and the output layer $\mathbf{y}$ (or $\mathbf{y}^R$ in the case of extended output RMLP architecture) is of dimension 12. All networks were trained using adaptive learning rates where the initial learning rate $\eta(0)$ was set to 0.04.

Section 7.5.1 addresses practical findings obtained from extended state MLP networks, whereas the extended output MLP architecture is investigated in Section 7.5.2. A comparison of the two models is made in Section 7.5.3. A further comparison is made in Section 7.5.4 where the extended models are compared with the RMLP architectures. Experiments and comparisons with the reduced RCC model are addressed in Section 7.6.

7.5.1 Experimental results on extended state MLP

This section repeats the experiments made on state MLP networks (Section 6.3.1) on its extended counterpart. This will allow a comparison of the two models which will be made in Section 7.5.4.

The first set of experiments focuses on finding the number of state layer neurons required to produce a well performing network for dataset-3. Since the dimension of the input layer **u** and the dimension of the output layer **y** are controlled by the dataset, the only layer with which we can experiment with is the state layer **x**. A set of experiments was crafted so as to find the influence of the cardinality of state neurons on the network performance. Networks featuring a single state neuron were trained and the performance was recorded. Training was performed with the BPTS updating rule which was applied for 400 training iterations. The experiment was then repeated where the number of state neurons increased gradually up to 64. The result is summarized in Figure 7.5.

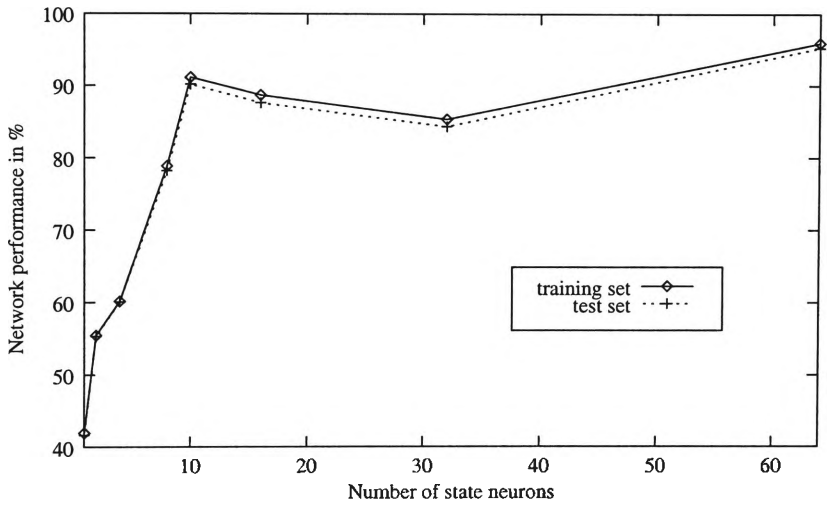


Figure 7.5: A diagram showing the recognition of the samples using an extended state MLP network trained using the BPTS updating rule. The number of state neurons are plotted against the network performance.

The results clearly indicate that an extended state MLP network requires at least 10 state neurons to encode the graphs in dataset-3 efficiently. Increasing the number of state neurons does not appear to result in any significant performance improvements. In addition, it is found that the generalization performance as obtained from the test set follows closely the performance achieved on the training set even if the number of state neurons is large. This indicates that the training set is sufficiently large to efficiently represent most instances of the learning problem. A good training set also helps to control the effect of overtraining which is also no apparent problem in this experiment.

Figure 7.5 gave results obtained when updating the network weights with the BPTS updating rule. It is an interesting question to ask: how other updating rules such as RPROP and BPTS with momentum influence the performance of an extended state MLP network. As a result, previous experiments were repeated with the application of various learning rules. The result is summarized in Figure 7.6.

It is observed that all updating rules require 10 state neurons in an extended state MLP network to produce good classification results on dataset-3. Moreover, the

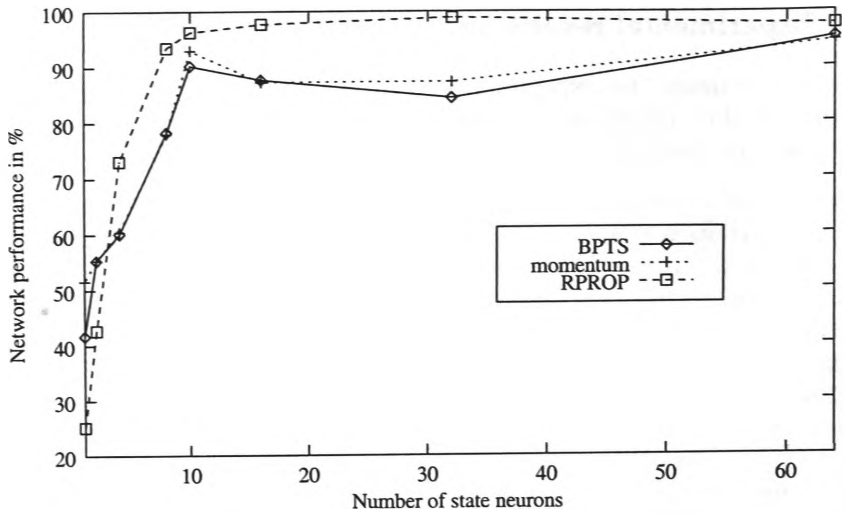


Figure 7.6: A diagram showing the recognition of the samples using an extended state MLP network featuring 10 state neurons. For each updating rule, the number of state neurons is plotted against the network performance. Three learning rules are compared: BPTS, BPTS with momentum term, and the RProp learning rule.

learning rules have in common that the network performance is not improved significantly if the number of state neurons is increased over 10. Nevertheless, it is found that the incorporation of a momentum term into the BPTS learning process may help to improve the quality of a trained network. But it is RPROP which produces the best results. This indicates that RPROP is the most efficient mechanism for finding a minimum and fast convergence.

The efficiency of the learning algorithms is investigated by the following experiment: Extended state MLP networks featuring 10 neurons in layer x were trained for up to 2000 iterations using three different learning algorithms: BPTS, BPTS with momentum, and RPROP. The network performance was measured at the end of every iteration. The result of this experiment is shown in Figure 7.7.

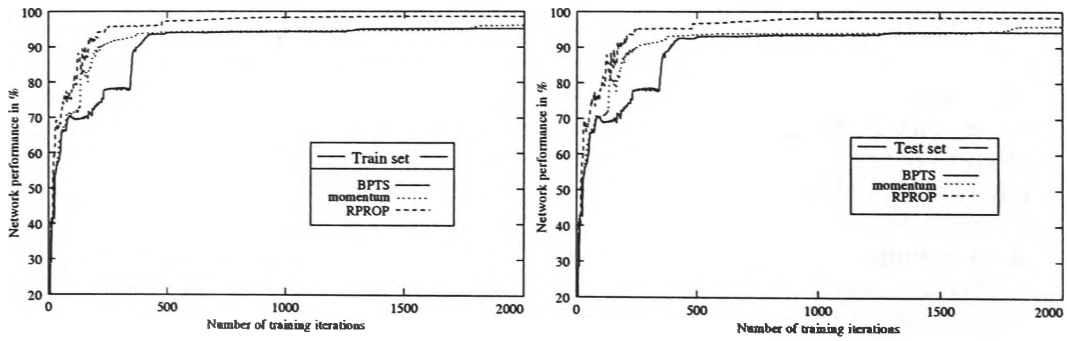


Figure 7.7: A diagram showing the recognition of the samples using an extended state MLP network featuring 10 state neurons. The number of training iterations are plotted against the network performance. The left hand plot shows the performance obtained on the train set, while the right hand plot shows the results by applying the trained model on the test set.

The finding shown in Figure 7.7 confirms that the RPROP learning rule converges fastest and produces best performing networks. An interesting observation is ob-

tained when considering the BPTS based updating rules between the iterations 330 and 370. During this interval of iterations 330 and 370, the performance curve of BPTS is near horizontal whereas BPTS with momentum does not exhibit such a feature. This is a strong indication of the presence of a plateau which hampers the BPTS model from converging quickly. In contrast, the momentum term has the expected effect moving seamlessly over areas with small gradients.

The confusion matrices visualised in Table 7.1 reveal further interesting behaviour of the algorithms.

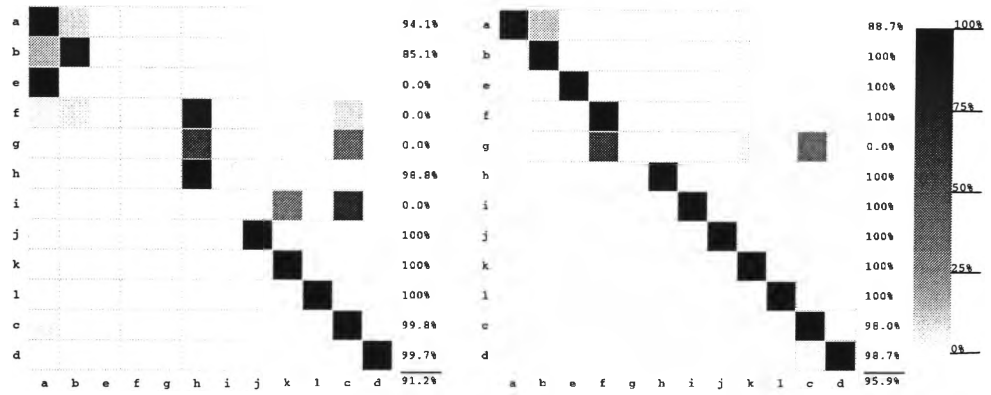


Table 7.1: Confusion matrix as obtained from training an extended state MLP featuring 8 hidden nodes with BPTS (left) and RProp (right) for 400 iterations.

In Table 7.1, the network trained through BPTS (the confusion matrix visualised and shown on the left hand side) failed to classify patterns from $\frac{1}{3}$rd of all classes in the data set despite demonstrating an overall performance of over 91%. This is due to the fact, that the classes `e`, `f`, `g`, and `i` are the ones which are represented by only few samples. In comparison, the same network trained using RProp (the confusion matrix shown in the right hand side) did not demonstrate such a poor behaviour. Yet, even the RProp trained network had considerable trouble classifying patterns from class `g`. As a result, it can be stated that an extended state MLP network requires a well balanced training set for which all classes are represented by a similar number of training samples. This problem of unbalanced number of training samples among the classes can be countered by presenting patterns belonging to classes for which only few samples exist more often.

The experiments given in this section have shown that the extended state MLP network is able to efficiently encode graph based information and that RPROP is the most efficient learning rule out of the three training rules attempted. A comparison of this extended model with standard RMLP model is given in Section 7.5.4. Next, we will consider the application of the extended output RMLP architecture to dataset-3.

7.5.2 Experimental results on extended output MLP networks

The experiments in this section have been constructed to allow a direct and fair comparison of this extended model with the standard output MLP network considered in Chapter 6. Hence, the first set of experiments focuses on the ability of an extended output MLP network to encode graph based representations given by dataset-3. We recall that the layer $\mathbf{x}$ in the output MLP architecture is not used for recursion but merely acts as a hidden layer between the input layer $\mathbf{u}$ and the

recursive layer **x**. Hence, for the output MLP case, it is layer **y** that recursively passes the state of children nodes to parent nodes so that we call layer **y** the 'state' layer consisting of state neurons and layer **x** a 'hidden' layer. The extended RMLP model adds links to layer **x** through additional links within this layer. There are no additional links in the state layer **y**.

From preliminary experiments we found that a good number of state neurons for the given learning task is 10 so we will restrict the following experiments to finding a reasonable number of hidden layer neurons. Thus Figure 7.8 gives the results of experiments which plot the number of hidden layer neurons in an extended output MLP network against its classification performance. Networks trained for this experiment utilized the BPTS updating rule for 400 training iterations.

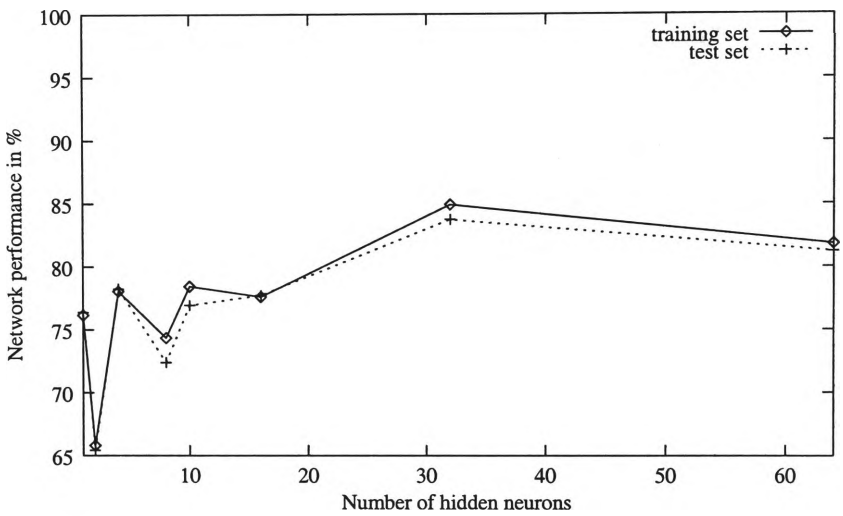


Figure 7.8: A diagram showing the recognition of the samples using an extended output MLP network trained using the BPTS updating rule. The number of hidden neurons is plotted against the network performance.

While it is not intended to compare this result with the results obtained from other architectures at this stage, we find that the performance is rather poor in particular when using less than 30 hidden layer neurons. In addition, the network performance appears to become unstable when using less than 16 hidden layer neurons. However, the generalization performance never drops more than 2% below the classification performance obtained from the training set. This is an expected result since the same problem was observed on the non-extended model in Chapter 6. In Chapter 6 the problem was traced to an insufficient number of training iterations for the BPTS updating rule. Nevertheless, we will first compare the efficiency of various learning rules before investigating the requirements in terms of training iterations.

In Figure 7.9 we present results obtained from applying various learning rules to the extended output MLP networks featuring a number of hidden layer neurons from 1 to 64. The number of training iterations used for these experiments remained at 400.

It is observed that the RPROP learning rule produces networks that significantly outperform those networks that were trained by BPTS based learning rules by 10 – 15%. In addition, RPROP trained networks produce good results already with just 4 neurons in the hidden layer whereas BPTS trained networks have a performance peak with 32 hidden layer neurons. Incorporating a momentum term into the BPTS learning rule can bring an advantage if the number of hidden layer neurons is not too

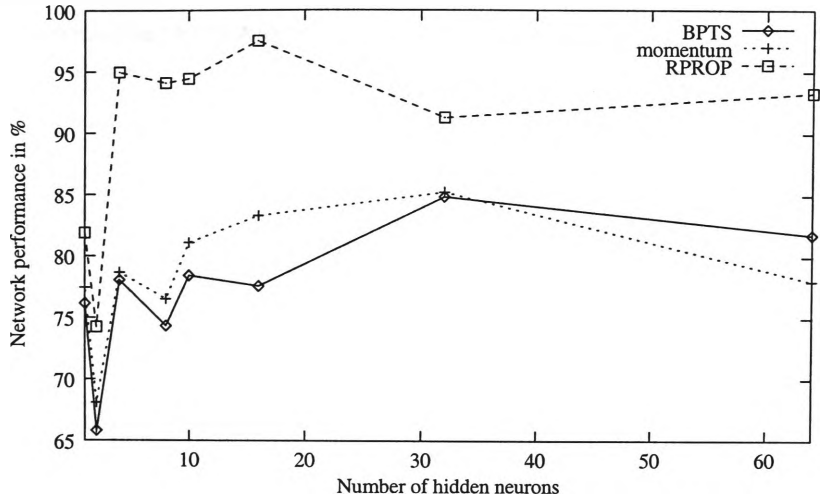


Figure 7.9: A diagram showing the recognition of the samples using an extended output MLP network featuring 10 state neurons. The performance of each updating rule as a function of the number of hidden layer neurons is plotted against the network performance.

large. This finding supports the suspicion that the number of training iterations is insufficient for *slow* updating mechanisms to converge. The next experiment measures the network performance after every iteration when training networks featuring 8 hidden layer neurons for up to 2000 iterations. The result is as shown by Figure 7.10.

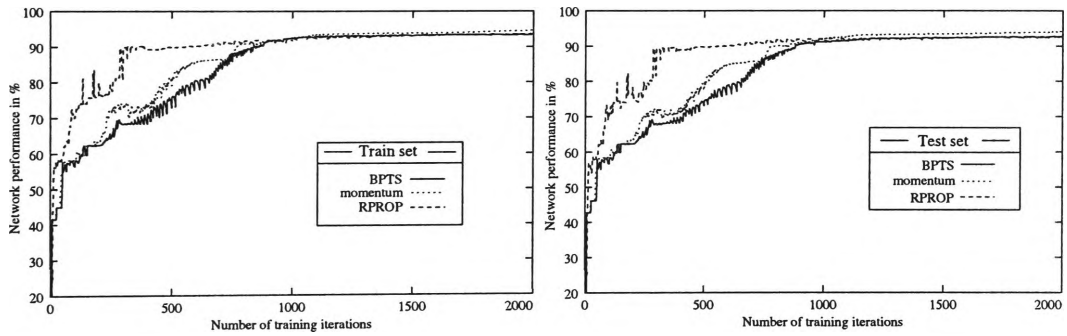


Figure 7.10: A diagram showing the recognition of the samples using an extended output MLP network featuring 8 hidden layer neurons. The number of training iterations are plotted against the network performance. Three learning rules are investigated: BPTS, BPTS with momentum term, and RProp learning rule.

This set of experiments confirms that BPTS based updating mechanisms require substantially more than 400 training iterations for the given learning task. From the experiment it is found that both BPTS and BPTS with momentum term require about 900 training iterations to converge whereas RPROP has converged after just about 350 iterations. It is interesting to observe that the generalization performance obtained by applying any of the three updating mechanisms is nearly identical if training is performed for at least 900 iterations. This differs with the observation made on the training set where RPROP trained networks outperform other trained networks at any stage during a training session.

Table 7.2 shows the performance of extended output MLP networks for each class by visualising the confusion matrices.

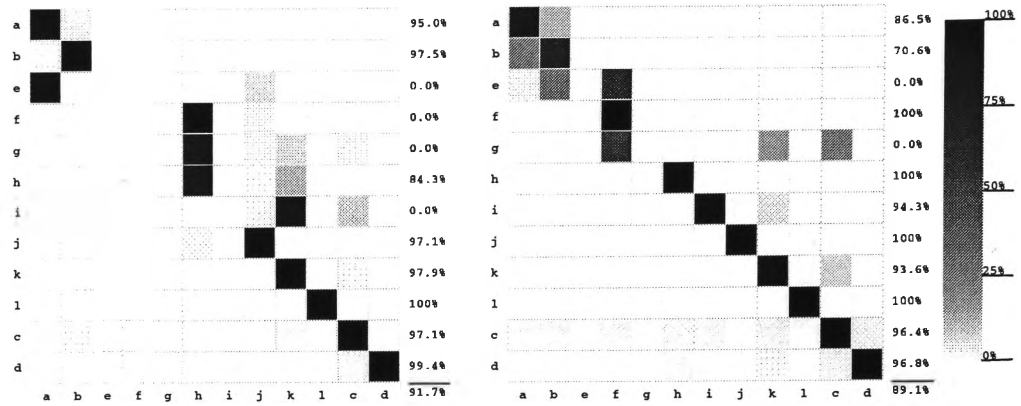


Table 7.2: Confusion matrix obtained by training an extended output MLP model featuring 8 hidden nodes with BPTS (left) for 900 iterations, and RProp (right) for 400 iterations.

It is found, that despite achieving an overall performance level of around 90%, a network updated using the BPTS algorithm (left confusion matrix) has considerable problems classifying certain classes. We recall that extended state RMLP networks demonstrated great difficulties with classifying patterns from classes which are not represented well in the input data. As this becomes clear from observing Table 7.2, this situation also applies to the extended output MLP case. This effect is visible to a lesser extend for networks updated through RProp (right confusion matrix). As indicated in the confusion matrix on the right of Table 7.2, we also observed a relative poor classification performance for patterns belonging to class `b`. As mentioned in the description of dataset-3 in Appendix A, the classes `a` and `b` can only be distinguished by information provided with the data label attached to a leaf of the graphs. Hence, the essential information is located deep inside the graph structure. This property makes it difficult to learn and it is expected that more training iterations are required for the network to fully adapt. We were able to confirm this by training a network with RProp for 900 iterations. Then, also the patterns from class `b` were classified well with a classification rate of 97.3%.

The experiments in this section have demonstrated two important properties of the proposed model:

- The extended output MLP networks are well able to encode graph based data similarly to the non– extended counterpart.
- The extended model requires more training iterations for convergence than the non–extended model when the updating rule depends on gradient information. This is caused by the additional links between the neurons in the hidden layer which are updated only slowly because the BPTS algorithm suffers from a long term dependency problem for neurons located deeper in a network structure. RPROP is considerably less sensitive to this problem.

The performances of the extended state MLP and the extended output MLP models are compared in the following section. A further comparison between the extended models with their standard counterparts will be performed in Section 7.5.4.

7.5.3 Comparing extended state MLP with extended output MLP networks

From an architectural point of view, the two architectures differ by the role of layer $\mathbf{x}$ and layer $\mathbf{y}$ and the presence of layer $\mathbf{y}^R$. In the case of a state MLP architecture, the layer $\mathbf{x}$ is to store the activity of neurons as obtained when processing the children of a node from an input graph. The activation is then utilized as an additional input to the network when processing the parent node. Since the layer $\mathbf{x}$ is applied recursively as we traverse through the graph, and since $\mathbf{x}$ is used as a device for storing intermediate states, we call the layer $\mathbf{x}$ a *state* or a *state recursive* layer. Layer $\mathbf{y}$ is simply called the output layer. This is in contrast to output MLP networks where recursion is exercised over layer $\mathbf{y}$ whereas a new layer $\mathbf{y}^R$ is the new output layer. As a result, in the case of output MLP networks we refer to $\mathbf{y}$ as the *state recursive* layer and to $\mathbf{y}^R$ as the output layer. The layer $\mathbf{x}$ is hidden in between the input and output layer so that this layer is referred to as the *hidden* layer. As a result, a significant difference between the extended state MLP and the extended output MLP network is that the extension takes place in the recursive state layer if the network is of an state MLP architecture whereas it is the hidden layer that is extended when considering the output MLP case.

From a practical point of view, the difference between the two models is the number training iterations that are required for training the network on a given dataset. An extended output MLP network requires significantly more training iterations than an extended state MLP network; in particular, if the learning rule depends on the magnitude of the gradient e.g., in backpropagation based methods. Backpropagation has a known problem with deep network structures because the gradient gets smaller the further a weight is away from the output. By adding links between neurons in layer $\mathbf{x}$, we are actually increasing the depth of the neurons in such a way that the first hidden neuron is located deepest. Since the gradient could be small for these weights, it takes more training iterations to adjust them. Hence, training time is increased. In the case where there are 10 neurons in the state layer, and 8 neurons in the hidden layer, we find that nearly double as many training iterations are required if the standard BPTS learning rule is applied. In contrast, RPROP does not take the magnitude of the gradient into account in determining the weight changes; it only considers its sign. As a result, RPROP is considerable less affected by the small gradients and does not suffer from the long term dependency problem that badly. It is for this reason that RPROP trained networks converge considerably faster than backpropagation based learning rules. The difference between the two updating mechanisms is the greater the deeper a network structure is, or the more neurons are in layer $\mathbf{x}$.

From a performance point of view we find that the difference between the extended state MLP network and the extended output MLP is negligible. No obvious performance improvement was observed by using the extended output MLP over the extended state MLP architecture. The opposite seems to be true when considering the generalization performance. It is found that the generalization performance of the extended output MLP networks is not as good as for extended state MLP networks. The reason for this behaviour may be explained by the fact that an extended output MLP network with 8 hidden layer neurons and 10 state neurons features 85 network weights less than an extended state MLP network with the same number of state neurons.

7.5.4 Comparing extended RMLP with standard RMLP networks

There is an important difference between the two extended models. For the extended state MLP case, there are interconnecting links between neurons in the state layer whereas for the extended output MLP case, the links are between neurons in the hidden layer. By adding these links, we effectively are introducing additional layers of hidden neurons which are not fully connected, and hence, the extended state MLP architecture can be seen as an architecture located between the standard state MLP architecture and a standard output MLP architecture.

Since the introduction of links between neurons in $\mathbf{x}$ effectively introduces different layers of neurons, it becomes clear that learning rules which rely on the magnitude of the gradient require substantially more training iterations than the network models without those extensions. Since layer $\mathbf{x}$ renders state MLP networks to be universal approximators, it becomes evident that the incorporation of additional hidden layers does not contribute to further improvements of the network performance. The opposite can be true as some training algorithms require more time to adjust the network weights. From this point of view, a state MLP network architecture is sufficiently suited for the encoding of graph structured information. Using extended models or the output MLP model does not add significantly to the performance of the network. This finding is confirmed later in this thesis when the various models are applied to a real world learning problem, see Chapter 8.

While this appears to be a negative result on the extended output MLP architecture, this at least satisfies our curiosity, in that we can conclude, based on experiments carried in this chapter, the extended output MLP architecture is not beneficial to consider.

7.6 Experiments with Reduced RCC

We repeat the experiments conducted on RCC in Section 7.3 on the reduced RCC model. This procedure allows us to directly compare the behaviour of the two RCC models in Section 7.6.1. A comparison between the reduced RCC model and the MLP model is given in Section 7.6.2. As a consequence, we utilize the extended policemen benchmark, dataset-3 and utilize the RProp updating rule for training this type of architectures. As before, we use 15 pool candidate units. Figure 7.11 shows the network performance as obtained after the i-th hidden neuron was added.

It is found that a reduced RCC network requires at least 9 hidden nodes to classify the patterns provided by the dataset-3 well. The addition of more than 9 hidden neurons does not help to significantly improve the performance. Instead, it appears that the generalization performance weakens with larger number of hidden neurons. But this effect is weak in that the degradation of the generalization accuracy is slight. Training was interrupted after the 25-th hidden neuron was added because training times exceeded reasonable levels (24 days clock time).

The classification performance is investigated further as shown in Table 7.12. In Table 7.12 the confusion matrices are shown. It is observed that all but class a are classified well. Nearly 20% of the patterns in class a are confused as class b. From the description of dataset-3 in Appendix A we know that the classes a and b are linearly separable, and that both classes are represented well by training samples. Hence, this is unlikely the cause of the observed problem. The two classes are differentiated by information provided with the data label of one of the leaf nodes. Hence, vital information leading to a successful classification of these patterns is located deep inside the structure. Given the recursive behaviour of the RCC algorithm, it appears

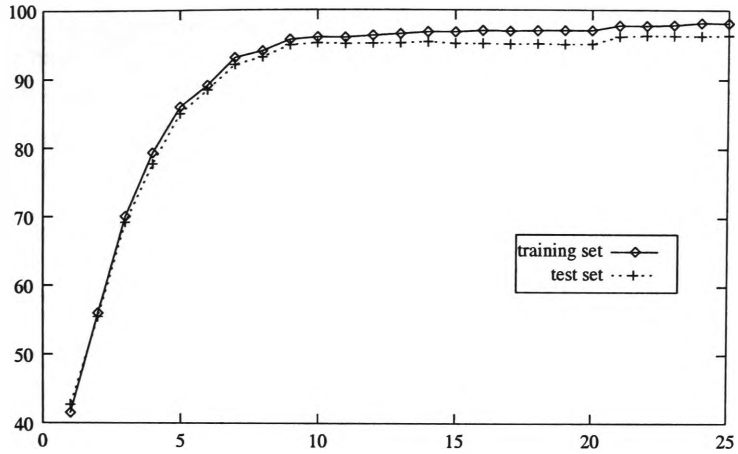


Figure 7.11: Training a reduced RCC on the benchmark problem: dataset-3. The number of hidden neurons is plotted against the network performance.

that the algorithm suffers from a long term dependency problem in that information provided by nodes deep inside a graph structure is inaccurately propagated forward to the root of the graph. Alternatively, this can be seen as that the error signal obtained at the root of a graph is inaccurately propagated back to deep nodes inside the structure.

Such issues of long term dependency can be overcome to some extent in the MLP type of architectures using the RProp learning mechanism. However, in the case of CC type of architectures, there does not appear to be any discussions on RProp type learning mechanisms. This can be a topic for future challenge in research in this area.

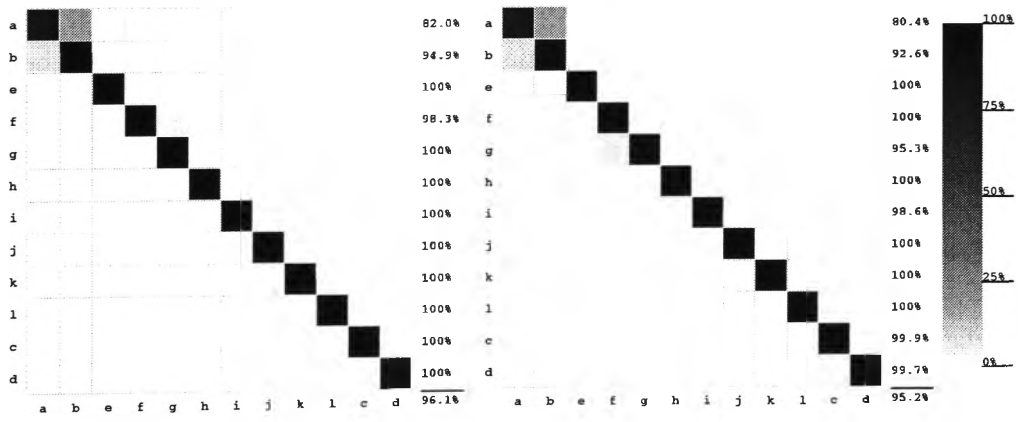


Figure 7.12: Confusion matrix as obtained from a reduced RCC with 10 hidden neurons. The left figure refers to the training set, the right figure to the test set.

The second experiment considers the effect of having a reduced training set available. This experiment is interesting since it was found in the past that different network models have different demands on the number of training data. The result of such an experiment is given in Figure 7.13. The network considered for this experiment featured 10 hidden nodes. This was done so as to allow a direct comparison with other models later in this chapter.

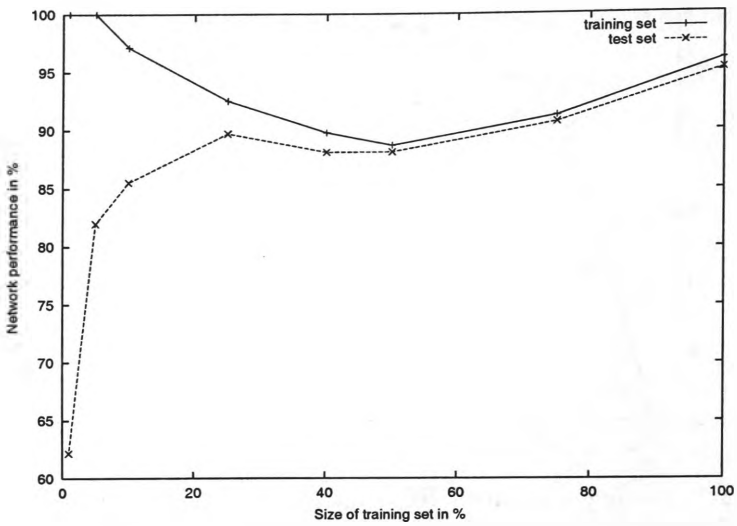


Figure 7.13: Network performance of a RCC featuring 10 hidden nodes plotted against the number of training patterns.

There are two interesting findings associated with Figure 7.13.

- It is demonstrated that the reduced RCC algorithm requires at least 50% of the data in the training set in order to generalize well. This is the same as observed in the RCC situation.
- The general network performance increases with a training set larger than 1875 training patterns (50% of the training set). Moreover, it is observed that the classification performance on the training set increases when the number of graphs in the training set becomes smaller than 1875 patterns. Eventually, when the training set is very small, the classification level reaches 100%. This observation is easily explained. By reducing the size of the training set beyond a certain point, the complexity of the learning problem is also reduced. However, the number of training patterns do not sufficiently represent the learning problem at hand so that the generalization performance is poor.

7.6.1 Comparing reduced RCC with RCC

The sole difference between a RCC architecture and a reduced RCC architecture is the $1/2n(n-1)$ connections between neurons in the hidden layer. It is interesting to compare the two models since it allows us to determine the influence of these connections on the behaviour and performance of the network.

In practical terms, the removal of the $1/2n(n-1)$ connections has only a small impact on the complexity of the training algorithm. Since the complexity for RCC is found to be:

$$O(Nn^2c^2P) \tag{7.64}$$

where P is the size of the pool of candidate units, N is the number of nodes in the training set, n is the number of hidden neurons, and c is the maximum out-degree of the data in the training set. Hence, the influence of the $1/2n(n-1)$ additional

connections in RCC to the complexity is negligible. This was confirmed by the observation that while conducting the experiments, training of the two network models took a similar amount of time.

The performance of the two models is considerable. An RCC (reduced RCC) featuring 10 hidden neurons classifies 97.7% (96.1%) of the training data correctly, and 96.5% (95.2%) of the test data are correctly classified. With this observation, both models have demonstrated to be successful in encoding graph structured information and they are both able to generalize well over the domain. A reduced-RCC performs slightly weaker than an RCC model which features the connections between neurons in the hidden layer. This is an expected result since a reduced RCC network model features less parameters which can be used for a given learning task. However, the impact is negligible.

Both architectures have the same demand on the quality of the training set. The performance can be poor if an insufficient number of training patterns is supplied. This generates a condition on the training set which needs to cover most of the problem space in order that it can generalize well. It was shown, that both models require a relatively high number of training samples in order to generalize well.

Thus far, the connections between hidden layer neurons have not demonstrated any significant impact on the network performance. The difference between the two models becomes visible when considering more than the minimum number of 9 or 10 hidden layer neurons. It appears that the adding of neurons beyond the lower limit helps to classify patterns which were more difficult to classify than others. Both models performed on a very similar level when the number of hidden neurons was 10. However, when doubling the number of hidden neurons we find that RCC is able to classify all but 2 patterns correctly whereas a reduced RCC is still misclassifying 110 patterns. Even the best observed performance of reduced RCC did not come closer to the performance of an RCC. Hence, it can be stated that the connections between hidden layer neurons help to classify difficult cases.

This is an interesting result, in that it ascertains the importance of the connections among the hidden layer neurons in the CC type of models.

In the following section, a comparison of the reduced RCC model is made with the state RMLP model.

7.6.2 Comparing reduced RCC with RMLP models

This section compares the reduced RCC model with the state MLP model. This is a valid comparison because the two architectures are very similar. The only difference is that RCC does not fully connect hidden layer neurons (matrix A) with the states from the children nodes. The matrix A in a RMLP architecture has cn^2 parameters whereas the same matrix in RCC features $\frac{1}{2}cn(n+1)$ parameters. Thus, the difference is an additional $cn(\frac{n-1}{2})$ parameters in an RMLP network.

When conducting experiments of these two architectures we found that training took considerable longer for the RCC model. We found that while the computational complexity of the standard Cascade Correlation model is lower compared to standard MLP networks, it is the opposite case for the recursive models. It has been found, that the state MLP tree model training algorithm has a computational complexity:

$$O(Nn^2c) \tag{7.65}$$

where N is the total number of nodes in all graphs from the training set, n is the number of state neurons, and c the out-degree. In comparison, the computational complexity for recursive CC is:

$$O(Nn^2c^2P) \tag{7.66}$$

where P is the size of the pool of candidate units. This difference can be dramatic especially when the out-degree of the training patterns is large. In the case of the benchmark problem: dataset-3 which featured an out-degree of 6, and since we utilized a pool of candidates of 15, training took about 540 times longer as compared to training a state MLP network. This is a clear disadvantage of the RCC algorithm. However, RCC has a greater potential of parallelizing the algorithm so that the algorithm can run on multi-processor computer systems. For example, it is possible to simultaneously train candidate units on P processing units.

In practice, both models, the reduced RCC and the state MLP model produced very good results on the given benchmark problem. A reduced RCC (state MLP) network featuring 10 hidden layer neurons classified 96.1% (96.7%) graphs from the training set correctly, and 95.2% (96.3%) test patterns are classified correctly. The following behaviour was observed very often: the two models performed on the training set on a near equal level, while the state MLP network often demonstrated a slightly better generalization performance. This may be attributed to the additional $cn(\frac{n-1}{2})$ parameters in an RMLP network.

More significant is the difference in the demand on the training data. It was demonstrated that a (reduced) RCC network requires a good portion of the data provided by dataset-3 in order to perform well. This is in contrast to findings with a state MLP network which demonstrated a good performance with just 10% of data from dataset-3. Again, the RMLP model seems to have a better generalization abilities than the RCC model. We surmise that this is due to the training algorithms applied to the RCC model. In the RCC model, only the best candidate unit is chosen and added to the network. The weights to this unit are then frozen. Hence, during the early stages of the training process, hidden units that focus strongest on the data provided are added which can lead to overfitting. This effect can be reduced by presenting a sufficiently large set of training data. In comparison, there are no fixed weights in an RMLP network model. Hence, during training, there is a higher degree of interaction between the neurons in the network which renders the problem of overfitting less likely to occur.

On the other hand, RCC has shown its ability to classify patterns well even if some classes are represented by only a few samples. This is in contrast with the observations made on a state MLP network which clearly demonstrated problems with classes that were not represented well. In these cases, a state MLP network required considerably more training iterations in order to properly encode the data that are not available frequently. In practice, such a problem is easily overcome by simply repeating the presentation of patterns belonging to classes for which only few samples exist more often.

The greatest advantage of the RCC algorithm over the RMLP architectures is that it dynamically adds hidden layer neurons as needed. With RMLP type of networks, the number of hidden layer neurons required for a given learning task can only be obtained through a trial and error process. This can be a time consuming task and requires the expertise of a specialist. The complexity of the RCC algorithm has a serious impact on training times for training tasks particularly where graphs feature a large out-degree. It becomes quickly infeasible to employ this type of

architecture if the out-degree is large. An advantage can be gained by combining the two algorithms: it is observed that the performance curve asymptotically approaches a maximum as hidden neurons are added to an RCC network architecture. It should be possible through a curve fitting process to predict the minimum number of hidden neurons required after the first few hidden neurons are added. This predicted number could then be used for the creation of an RMLP network. Such a system could work autonomous and feature training times that are located somewhere between those required for an RCC and a state RMLP network.

A summary and analysis of experimental findings made in this chapter is given by the following section.

7.6.3 Comparison Analysis

It has been shown in Section 6.2.2 and Section 7 respectively that state MLP tree architectures and recursive cascade correlation networks are of similar architecture. Also, experimental results have shown that both approaches produce comparable results. Nevertheless, these models differ in some important points which are summarized in the following table:

RMLP	RCC
• Is an universal approximator [43, 46].	• Not an universal approximator [32].
• Network dimension is static.	• Detection of network dimension.
• Lower computational complexity.	• Training a single unit for each step.
• Fewer training iterations required.	• Output layer is re-trained after each added hidden layer neuron.
• Might get caught in local minimum.	• Better stability when using a pool of candidates.
• Good generalization performance.	• Overfitting can occur more easily.

The fundamental difference between state MLP architectures and recursive Cascade Correlation models is that in Cascade Correlation the number of hidden layer neurons required for a given learning problem is determined automatically. In contrast, for state MLP networks the number of hidden nodes needs to be chosen before network training is conducted. This is often a difficult task especially if there is no information about the complexity of the learning task given. When choosing too few hidden layer neurons, the state MLP network will be unable to solve a given problem satisfactorily. Selecting too many hidden layer neurons will result in long training times and increases the risk that overfitting may occur. Thus, the ability of CC to determine the number of hidden layer neurons required for any given learning task is a significant advantage. Generally, RCC may find a very efficient solution without any prior knowledge on the problem. As a result, RCC is a useful instrument to make the first approach to a new problem and to have an idea about the number of hidden units required for the problem under analysis.

Furthermore, it has been shown in [43, 46] that recursive MLP networks are universal approximators and that Recursive CC is not a universal approximator [32][7]. It must be noted that this limitation holds when representing certain cycles in some finite-state automata (where it must recognize arbitrarily long input sequences) [8].

[7] In [32] Recurrent Cascade Correlation were shown not to be universal approximators for certain types of sequences. This is similar to classic CC networks which are not universal approximators also shown in [32].

[8] In this case the number of added hidden units grows linearly with the longest string

For most of the practical tasks used in data structures processing, these limitations do not hold since there are only finite structures.

A further observation is, that recursive MLP networks do not always converge. This is especially the case when the number of hidden nodes is too small for a given learning problem. In comparison, RCC utilizes a pool of candidates. In doing so, the risk of falling into a local minima is reduced. Our experiments have shown that if the pool is chosen sufficiently large, RCC always converges to a solution, and is likely to converge with the smallest number of hidden layer neurons required for the given learning task.

While the computational complexity of standard Cascade Correlation is lower compared to the standard MLP networks, it is the opposite case for recursive models. It is found that the RCC models have a computational complexity which depends on the size of the pool of candidate units, and the squared maximum out-degree (Eq. 7.66) while the RMLP type of architectures has a computational complexity which depends on the maximum out-degree (Eq. 7.65). This difference can be significant especially when the out-degree of the training patterns is large and when the pool of candidate units is large.

Experimental results have also shown that RMLP networks are often better in generalizing the given training data. We concluded that this is a side effect of the fact that for the same number of hidden layer units a RMLP network has $cn(\frac{n-1}{2})$ more parameters than RCC.

7.7 Conclusions

When conducting these experiments, the finding was that RCC may be a useful tool for predicting the minimum number of hidden layer units required for a state RMLP network. RMLP models have demonstrated that by using a number slightly larger than the minimum number of hidden neurons the generalization performance is improved as well as the network becomes more stable in that it converges to low error levels. However, the problem is the computational complexity associated with RCC's training algorithm. In cases where the data set features a large out-degree, and no device is available for a parallel execution of the RCC algorithm, training times can quickly exceed reasonable levels. This is in contrast with the classic CC which had been found to be of a lower computational complexity than the MLP models. Similarly, CC network architectures feature more network parameters than MLP networks whereas the opposite is true if the models are extended to the domain of graphs. An RCC network features less parameters whenever the out-degree of the data is equal to 1 or greater.

Furthermore, the experiments in this chapter confirm the fact that RCC is not a universal approximator, but this does not have a significant negative effect on real world problems. It has been demonstrated that a less powerful model such as RCC is able to obtain efficient results in specific applications.

It is interesting to note that the cascade correlation algorithm allows to constructively build a MLP from "scratch". In other words, one does not need to know the size of the hidden layer at all. One only needs to use the cascade correlation algorithm and it will automatically find an equivalent MLP architecture. This observation is an important one, in that so far, everyone takes it for granted that it is not possible to infer automatically a MLP structure. One either finds a cascade correlation architecture which is not exactly the same as a MLP architecture, or a

length of the training set examples.

MLP architecture which one does not know how many hidden layer neuron to use. The observation made here makes this clear. One can use the cascade correlation training algorithm, and one can obtain the MLP architecture, without knowing the number of hidden layer neurons a priori.

By the very fact that it is possible to obtain a MLP architecture using a cascade correlation algorithm, it is observed that the reduced cascade correlation architecture is a universal approximator. This again is an interesting observation. Whether the recursive cascade correlation architecture is a universal approximator or not has not been discussed in the literature. However, from the observations made here, it is readily recognised that the reduced cascade correlation neural network may be a universal approximator as they appear to have the same type of generalization capability as the recursive MLP architectures. Thus, by inference, the recursive cascade correlation neural network architecture may be a universal approximator even though we cannot prove it yet. This may be one reason why it has been successful in practical applications. This, again, is in contrast with the non-recursive model where it was found that the cascade correlation model is not an universal approximator.

It is also interesting to note that it is not possible to use the cascade correlation algorithm to obtain the classic MLP architecture, at least not in its present form without modifications. Basically, in the classic MLP architecture, there is no direct feed through from the input to the output, i.e., matrix $D = 0$. As observed previously, the availability of the error from the immediate preceding step is crucial in the training of the hidden layer neurons. Without this, we cannot formulate the correlation measure. Secondly it appears that the influence of the input directly on the output is crucial to the cascade correlation training algorithm.

Fortunately, there is some theoretical justification in this formulation, i.e., with a direct feed through from the input to the output. It was shown [77] that the MLP architecture which includes a direct feed through from the input to the output performs better than the one without this direct feed through. Hence, one can safely make use of the extended MLP architecture.

The extended MLP architecture is also an interesting one, in that it is half way between a fully recurrent neural network in the sense of Pineda [66], and the classic MLP. This architecture was not reported before, at least not in the form discussed here. This could open up some interesting questions.

This also raises an interesting question: What are the functions of these links as represented by the matrix A? From a universal approximator point of view, these links are superfluous, i.e., they are not required to ensure a universal approximator property. But, their presence may facilitate easier learning, as they allow a more nonlinear mapping between the input and output, without creating an additional layer, as common in practical application of MLP. In other words, traditionally, the procedures of using MLP are as follows: postulate a fixed number of hidden layer neurons. Train the MLP architecture accordingly. If the results are inadequate, then select another number of hidden layer neurons, and try again. If after trying a number of times using different values for the number of hidden layer neurons, and still not succeeding to find an adequate MLP architecture, then one would use a two layer MLP architecture in an attempt to find a better input-output mapping. The idea of using a lower triangular matrix A with zero diagonal elements has not been suggested before. Hence, this may be the next architecture which one may attempt without having to use a two hidden layer MLP architecture.

Despite of the introduction of extended architectures to the case of tree recursive

models rendering RCC and RMLP architectures more similar, we did not discuss modifications which produce identical architectures. Future research may consider a modified extended state MLP architecture, where the matrix A is modified so as not to fully connect all the hidden layer neurons with all hidden layer neurons from the children in the way it is found in RCC. In other words, all elements to the right of the diagonal of matrix A are zero. This would produce a state RMLP which is identical to an RCC network architecture. It would be interesting to compare this model with RCC on the benchmark problem. This is presented as a challenge for future research in this area.

Both, RMLP and RCC are updated by gradient based mechanism. The problem with this approach is that the gradient becomes smaller the further we proceed into the network structure. It is well known that MLP network architectures featuring many hidden layers suffer from a long term dependency problem because many updating rules depend on the magnitude of the gradient.

In the case of graph recursive models, the problem appears due to the application of a graph recursive algorithm. Every node in the graph is encoded by the same MLP neural network so that recursive MLP networks of this type can be seen as MLP networks with as many hidden layers as the deepest node in a graph where each hidden layer shares the same parameters. This has been addressed by [33] where RMLP networks are called *folding architectures* demonstrating that recursive MLP networks can suffer from a long term dependency problem if graphs are deep in structure. The problem can be eased through the application of RProp updating. However, preliminary experiments indicate that even with RProp updating, that a graph depth of more than 12 leads to networks which do not converge or require an extraordinary large number of training iterations in a high precision computing environment.

It was demonstrated that (reduced) RCC performs at similar levels as (extended) state MLP models despite the fact that it features fewer parameters and has been considered as not being a universal approximator. That this is not always the case is found in the following section where various recursive neural network models are applied to a difficult real world classification task, viz., the logo recognition problem.

Chapter 8

Application to a real world problem

8.1 Introduction

Until now, the models introduced and described in this thesis have been applied to artificial learning problems only. This was done to allow an evaluation and comparison in a well controlled manner. This chapter applies the various proposed models in this thesis to a real world learning problem, viz. the logo recognition problem. One of the main aims of doing this is to demonstrate that the models behave in a similar fashion for the real world problems when compared with the artificial learning problems. In addition, it will be shown that the new models can also perform well when applied to a real world problem.

The logo recognition problem is a problem from the area of image recognition. The task is to recognize and classify company logos [28]. A dataset consisting of 39 different classes of logos were available in the form of digital images [1]. There are 300 different samples available for each of the 39 classes, producing a total set of 11700 images. A graph representation was extracted from each of the images by following the procedures described in Appendix B. The result is a training set consisting of 5850 graphs featuring a total of 55547 nodes, and a validation data set with 5850 graphs featuring 55654 nodes in total. The problem and the dataset are described in greater detail in Appendix B.

This chapter does not intent to give an exhaustive evaluation of the performance of the proposed models in this thesis. Instead, only final results obtained from a typical training session are presented. Typical results were obtained by removing the best and the worst configurations in the experiments.

The structure of this chapter is as follows: the application of the logo recognition problem to SOM based architectures is addressed in Section 8.2. The application of RMLP based architectures to this classification task is given in Section 8.3, and RCC based architectures are applied in Section 8.4. A summary of the findings and some conclusions are given in Section 8.5.

[1]The logo dataset was provided by the Document Processing Group, Center for Automation Research, University of Maryland.

8.2 Application of SOM-SD method to the logo recognition problem

A selection of Self Organising Maps were trained on the Logo recognition problem. The size of this data set did not permit the execution of an exhausting set of experiments within a reasonable amount of time. As a consequence, experience gained from experiments on the benchmark problems was used to determine a suitable set of training parameters.

All maps addressed in this section feature $156 \times 119 = 18564$ neurons, which makes the number of neurons approximately 1/3 of the total number of nodes in the training set. The neurons were connected by a hexagonal neighbourhood and were trained for 350 iterations with an initial neighbourhood radius of 40, and an initial learning rate of 0.08. The best values for the (μ_1, μ_2) pair as suggested by Equation 4.5 were $\mu_1 = 1531\mu_2$. However, from experimental findings made in Section 4.4.3 and Section 5.4.1 it is known that an increase of value for μ_2 by two magnitudes has beneficial effects on the network performance. Hence, for the experiments on this logo recognition task, we choose $\mu_1 = 15\mu_2$.

First, the SOM-SD networks are trained in an unsupervised fashion. This is addressed in Section 8.2.1. Section 8.2.2 presents findings from SOM-SD networks trained in a supervised fashion.

8.2.1 Unsupervised SOM-SD

This section gives the results obtained when training a SOM-SD network in an unsupervised fashion on the logo data set. The training parameters were as specified above. The result obtained is given in the confusion matrix presented in Table 8.1. It is found that the overall performance of the network reaches 88.44% correct classification on the training set and 77.3% on the test test. Note that training was performed without any class information. The statistics shown in Table 8.1 utilized class relationships of the input data after the completion of the training session and are used to provide a better insight into the quality of the results obtained. This class information is not used elsewhere in the training nor in the validation process.

From the confusion matrices in Table 8.1 it is found that most patterns are classified correctly and that the generalization performance for each individual class is within acceptable limits. The generalization performance is about 11% below the level achieved by the training data set. Common causes for a reduced generalization performance include: overtraining, a network chosen too small, a training set which does not resemble a sufficient coverage of instances of the learning problem at hand. The answer to what might have influenced the generalization performance in this experiment will be given in Section 8.5 after observations were made on other architectures. Nevertheless, it is demonstrated that a SOM-SD network when trained unsupervised with a preselected set of learning parameters produces reasonable results on this dataset.

Next, this result is compared to a SOM-SD network which is trained in a supervised manner on the same dataset using the same set of parameters.

8.2.2 Supervised SOM-SD

This section gives results obtained when training the SOM-SD network in Section 8.2.1 in a supervised manner. Again, we use the same set of initial parameters for the training session. In Chapter 5 it was found that the supervised SOM-SD

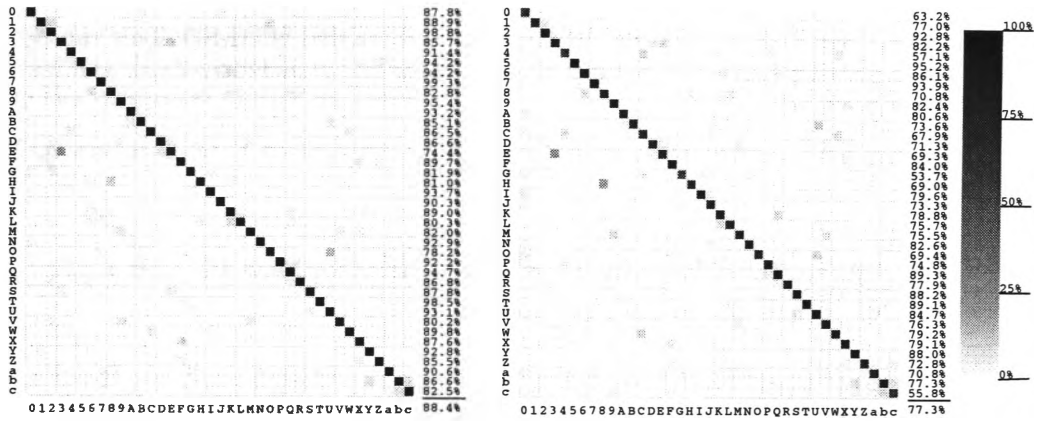


Table 8.1: Performance comparison between the training and the testing data set for the logo recognition problem. The confusion matrix is visualised as a gray coded matrix. Class labels are shown to the left and below the matrix, individual and the total performance rating is shown to the bottom right of a matrix. The left hand side matrix reflects results obtained from the training data set, while the right hand side matrix shows the results obtained from the testing data set.

is robust with respect to the choice of the rejection rate ϵ but that small positive values can lead to marginally better results. As a result, the rejection rate for this experiments is chosen to be $\epsilon = 0.1$. The results of the training session after 350 training iterations are shown in Table 8.2.

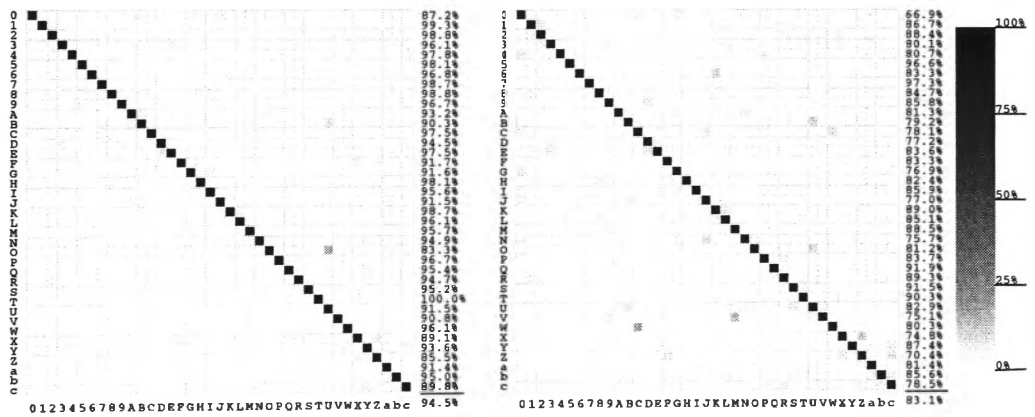


Table 8.2: Confusion matrix obtained from a supervised SOM-SD network on the logo recognition problem. The confusion matrix to the left reflects the result obtained on the training data set, while the confusion matrix to the right is obtained from the testing data set.

It is found that the incorporation of a teacher signal to the learning process led to a significant improvement of the network performance. When comparing these results with the unsupervised trained SOM-SD network it is found that the performance on the training data set has improved by 6.1% to 94.5%, and improved by 5.8% to 83.1% on the test set. Thus, the incorporation of a supervisor signal to the learning process has resulted in a significant performance improvement on the training data set. However, the generalization performance on the testing data set is still about 10% behind the classification performance of the training data set; an observation

which has also been made with the unsupervised trained SOM-SD. This can be seen as an indication that the training data set does not sufficiently cover the learning problem at hand.

Next, the logo recognition learning problem is applied to various recursive MLP models.

8.3 Application of the RMLP architectures to the logo recognition problem

This section applies RMLP networks to the logo recognition problem. Results from four different RMLP architectures: the output MLP model, the state MLP model, the extended output MLP model, and the extended state MLP model architectures are addressed individually and their performances are compared with each other. Since all RMLP architectures addressed in this section are trained in a supervised manner, a comparison can be made with the findings from training a supervised SOM-SD network. In addition, the extended state MLP architecture is identical to the RCC architecture so that a comparison between these two models can be made. Results obtained from the RCC model are addressed later in this chapter.

The finding of appropriate learning parameters and network sizes is somewhat more difficult as compared to the SOM-SD architectures. The reason for this is that SOM-SD behave rather robust within a range of reasonable parameters so that an appropriate choice for the learning parameters is found easily. While the initial network weights and the initial learning rate for RMLP based networks contribute little to the final network performance due to the use of the RPROP updating rule and adaptive learning rates, the appropriate size of the network has to be found through trial and error.

All networks addressed in this section have been trained for 1600 iterations using the RPROP weight updating rule and adaptive learning rates with $\eta(0) = 0.04$. The input and output dimensions are controlled by the given data set. All networks feature 12 input nodes and 6 output neurons since the dataset features 12-dimensional data labels and 36 classes which are encoded by 6 binary values. The maximum out-degree of this dataset is 7. The number of hidden and state neurons have been varied until a good network performance was achieved but were kept as small as possible to keep training times minimal.

8.3.1 Standard RMLP architectures

Two architectures based on the standard RMLP architecture are applied to the logo recognition learning problem, and experimental findings are given. First, an investigation of the state MLP networks is considered. The findings are then compared with those obtained by the output MLP networks.

State MLP models

The difficulty with RMLP based networks is that an appropriate number of network parameters which is controlled through the number of state neurons in the network has to be found. Since there is no general rule that may lead to an appropriate choice of the number of state neurons for a given learning task, a trial and error approach has to be applied. For the logo recognition problem, we trained the state MLP networks featuring a number of state neurons in layer $\mathbf{x}$ varying between 7 to 60. It was found that choosing 22 state neurons produced networks with the

best performance/time ratio. Results obtained when training a state MLP model featuring 22 state neurons with the above mentioned training parameters are given by the confusion matrix in Table 8.3.

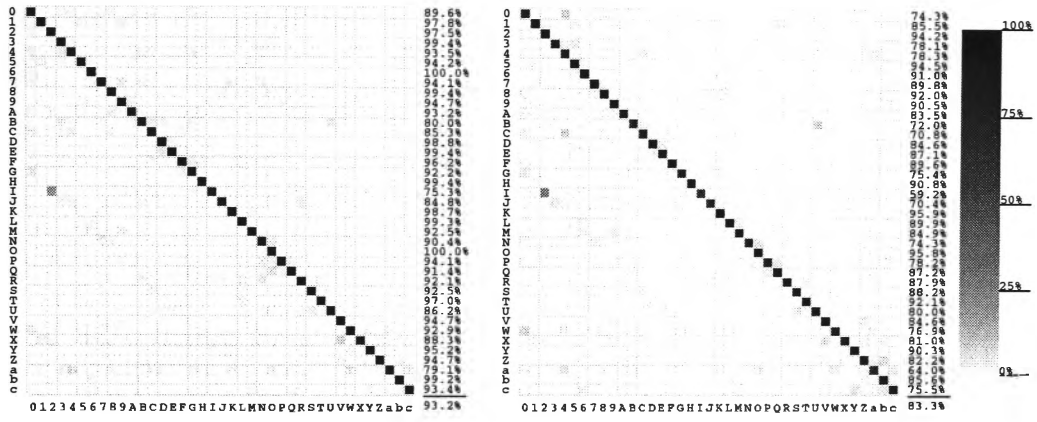


Table 8.3: Confusion matrix obtained when training a state MLP network featuring 22 state neurons for 1600 iterations using the RPROP learning rule. The matrix to the left gives the results obtained from the training set, while the matrix to the right is the results obtained when validating the network with the test set.

From Table 8.3 we find that the state MLP network classified 93.2% of the logos in the training set correctly and exhibited a generalization performance of about 10% below that figure. Furthermore, Table 8.3 appears to illustrate a significant misclassification occurring to patterns belonging to class I which were misclassified as class 2 patterns in 33% of the cases. However, this is not a particular problem with this network architecture nor a problem with the dataset. Locally significant misclassifications were not always observed, and this seemed to occur sporadicly on random locations. Typically the occurrence of misclassified patterns were distributed equally.

An interesting comparison of these findings can be made with the performance of an output MLP network because those resemble a state MLP network with an additional hidden layer. This will be considered later in this chapter.

Output MLP architectures

The output MLP architecture features recursive state neurons in layer **y** as opposed to the state MLP case where layer **y** serves as the output layer. As a result, output MLP based architectures feature state neurons in layer **y**, and neurons in layer **x** are called hidden layer neurons. The difficulty with output MLP networks is that the finding of an appropriate number of network weights is even more difficult than that of the case of state MLP networks. This is because there is an additional layer of neurons whose dimensionality is flexible. In order to reduce the complexity of finding an optimal number for the dimension of **x** and **y**, we use the findings obtained from state MLP networks which stated that 22 state neurons is a good choice. Hence, we restricted our experiments to the finding of a good number of hidden layer neurons and found that the best performance/time ratio is obtained when using 20 hidden layer neurons. Other training parameters remained the same as stated earlier for the state MLP architecture case. The performance of this output MLP network is given in Table 8.4.

It is found that an output MLP network featuring 20 hidden layer neurons and

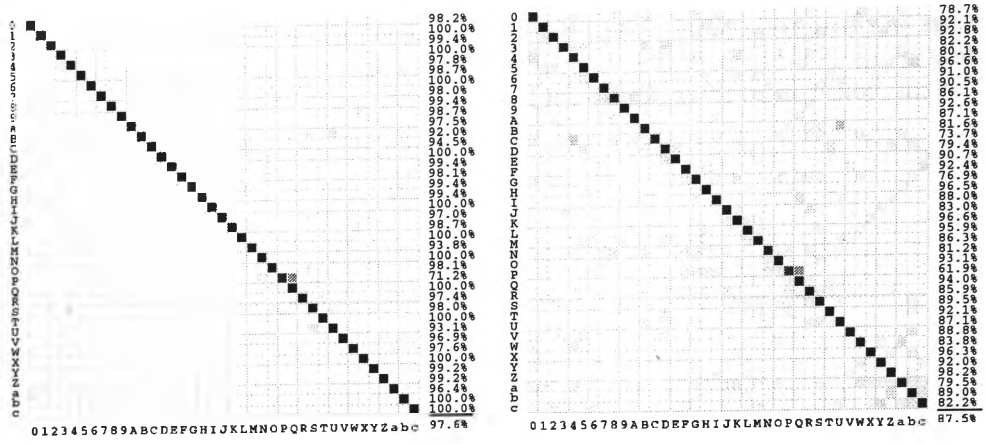


Table 8.4: Confusion matrices obtained when training an output MLP network featuring 20 hidden layer neurons and 22 state neurons. The confusion matrix obtained from the training set is shown on the left, and that obtained from the test set is shown on the right.

22 state neurons significantly outperforms a state MLP network. A performance boost of 5% over state MLP network performances is observed. This is a very interesting finding because an output MLP network of this size is featuring 3924 network weights which is only 140 weights more than the state MLP network which features 22 neurons in the recursive state layer. Hence, this performance boost cannot be attributed to the increase on network parameters. It indeed demonstrates that sufficiently complex learning tasks will benefit from an additional hidden layer. However, the generalization performance is still about 10% worse than the training performance.

The inclusion of a hidden layer to the network architecture produces networks with an improved performance in particular if the given learning task is complex. Hence, it will be interesting to compare these findings with results obtained from extended RMLP models because the extension is effectively introducing hidden layer neurons of various depths.

8.3.2 Extended RMLP architectures

In standard RMLP architectures, there are no connections between neurons in layer $\mathbf{x}$. However, from the analysis of a RCC architecture, we find that it may be beneficial to have those neurons partially connected. As a result of this observation, RMLP architectures are extended by featuring interconnections between neurons in $\mathbf{x}$ so that every neuron $x_i \in \mathbf{x}$ has an additional incoming link from all its predecessors neurons $x_0, \ldots, x_{i-1}$. The effect of such an extension is that neurons in $\mathbf{x}$ are moved deeper into the structure of the network architecture, and hence, produces hidden layer neurons at various levels. Then, neuron x_0 is located deepest and has the distance n, where n is the dimension of vector $\mathbf{x}$, to the network output layer.

From the finding in the preceding section, we find that the introduction of a hidden layer to the network architecture can result in a significant boost in the network performance, it will be most interesting to investigate whether the extended models will also exhibit this phenomenon.

Extended state MLP architectures

In order to allow a fair comparison of an extended state MLP network with a standard RMLP network, we use a network featuring the same number of state neurons as used in the previous experiments, and trained it with the same set of initial parameters. Hence, the results shown in Table 8.5 refer to an extended state MLP network featuring 22 state neurons and this architecture was trained for 1600 iterations using the RPROP learning rule.

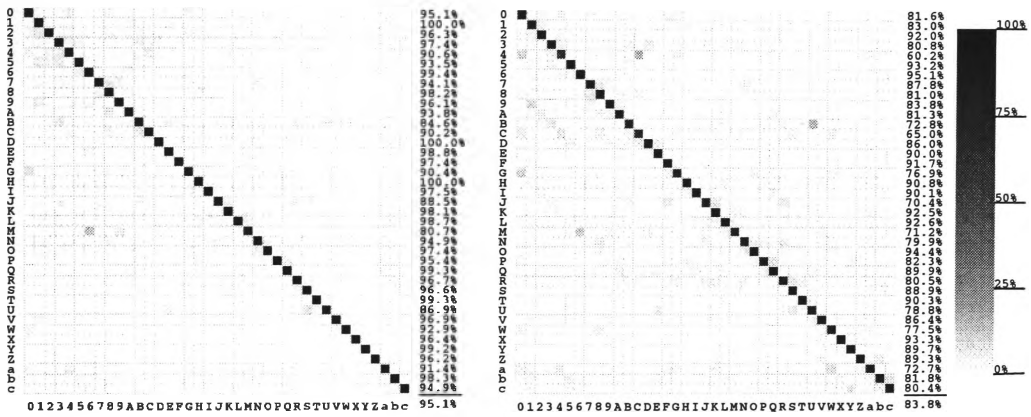


Table 8.5: Confusion matrices obtained from the training and testing of an extended state MLP architecture featuring 22 state layer neurons. The confusion matrix on the left is obtained from the training set. The confusion matrix on the right is obtained from the test set.

It is found that the incorporation of the additional 190 forward links between neurons in the state layer has resulted in a slight improvement of the network performance whereas the ability to generalize has not improved significantly. This result reflects two properties:

- The network can perform better through the introduction of hidden layer neurons.
- Neurons located deep in a network structure do little or not contribute to the network performance. This is mainly due to the gradient based learning method which has problems updating weights which are located far away from the output layer.

Hence, it can be expected that the inclusion of links between neurons in $\mathbf{x}$ to an output MLP architecture will not help to improve its performance significantly.

Extended Output MLP architectures

The experiment with an output MLP network addressed earlier in this chapter has been repeated on the output MLP model where neurons in the hidden layer $\mathbf{x}$ are partially interconnected. The results of this experiment is given in Table 8.6

The confusion matrix gives a slight performance improvement over the training result obtained from the standard output MLP architecture. However, it also exhibits a slight decrease of the corresponding generalization performance. The difference of performance between the two models is very small. Furthermore, time constraints did not allow us to conduct a more comprehensive set of experiments to confirm a practical difference between the two models. However, it can be assumed that the

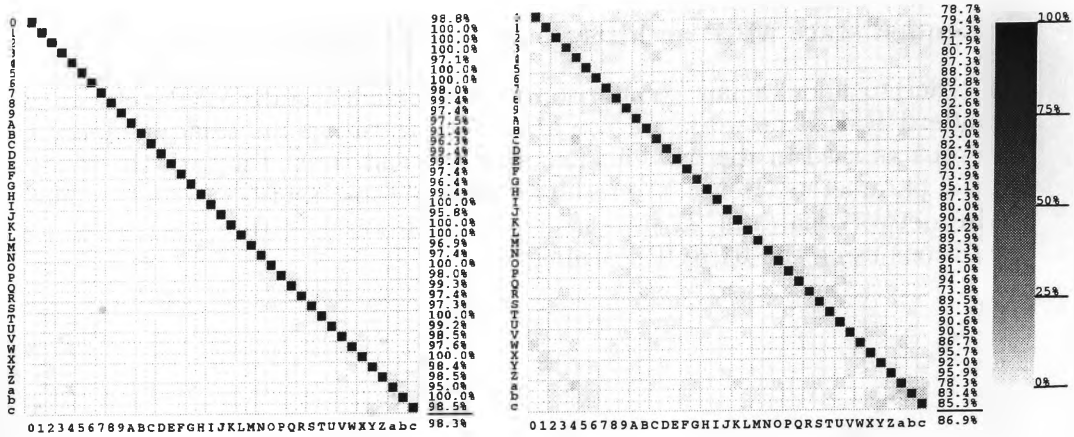


Table 8.6: The confusion matrices obtained from an extended output MLP network featuring 20 hidden layer neurons and 22 state neurons. The matrix to the left gives the result obtained from the training data set, while the matrix to the right gives the results obtained from the test set.

inclusion of additional links between neurons in $\mathbf{x}$ does not help to influence the performance of an output MLP network architecture significantly.

The similarity of the extended state MLP network architecture with the MLP network architecture motivates us to compare the findings of this section with experiments conducted on RCC based models.

8.4 Application of RCC models to the logo recognition problem

The most significant advantage of the Recursive Cascade Correlation model is that hidden layer neurons are added by a dynamic process as needed. Findings from Chapter 7 suggest that the RCC model might be a useful tool for predicting the minimum number of hidden layer neurons required for an equivalent RMLP. The disadvantage of the RCC model is that the computational complexity grows quadratically with the out-degree and the number of hidden layer neurons. This section applies the RCC algorithm to the logo recognition problem, where the dataset features an out-degree of 7. In addition, the expected number of hidden layer neurons required for this learning problem is 22 because state MLP networks demonstrated a good performance with 22 hidden layer neurons. Experiments shown in this section are restricted to the observation of the impact of the training computational complexity, and the observation of network performances for networks featuring a similar number of hidden layer neurons or a similar number of network parameters.

Networks considered for the experiments commonly featured a 12-dimensional input layer, a 6 dimensional output layer, a pool for candidate units of size 12, and an initial learning rate of 0.8.

Section 8.4.1 addresses some experimental findings obtained from the application of RCC to the logo recognition problem. The reduced RCC model that does not feature connections between hidden layer neurons is addressed in Section 8.4.2.

8.4.1 Standard RCC architectures

An RCC network was trained with a set of parameters stated previously and the results obtained after 22 hidden layer neurons were added is displayed in the form of confusion matrices in Table 8.7.

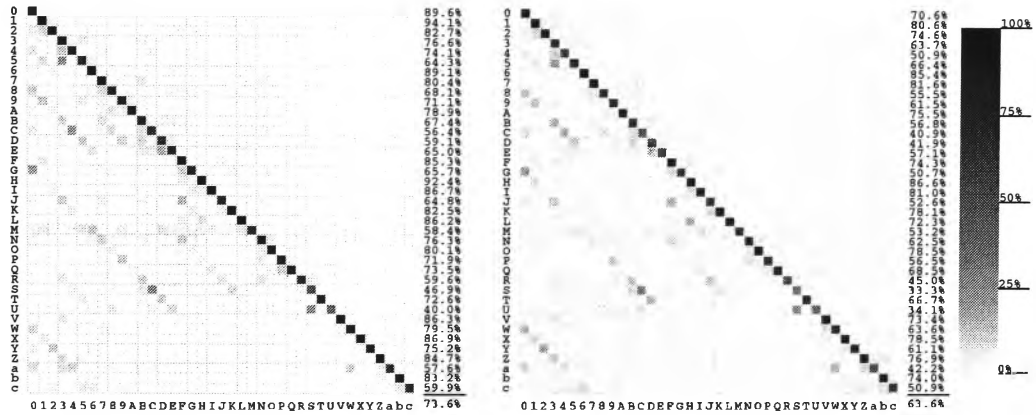


Table 8.7: The confusion matrices obtained from an RCC network featuring 22 hidden layer neurons. The matrix to the left gives the results obtained from the training data set, the matrix to the right gives the results obtained from the test set.

It is observed that the overall performance of the RCC architecture on the training data set is 73.3% whereas the performance on the test set is 63.6%. Hence, the RCC has shown a performance which is considerably worse than the performance achieved by an RMLP network. This result is somewhat surprising. However, it might be attributed to the fact that an RCC network model features less parameters as compared to a state MLP network architecture. For networks featuring 22 hidden layer neurons, a state MLP architecture features a total of 3784 network weights whereas the RCC architecture has only 2398 weights. This is a significant difference of number of weights and might well be the cause for the poor performance of the RCC architecture. From the observation that a state MLP network architecture has $mn + pn + cn^2$ weights whereas the RCC architecture has $mn + np + \frac{1}{2}cn(n+1) + \frac{1}{2}n(n-1)$ weights, we can conclude that a RCC architecture with 28 hidden layer neurons will feature almost the same number of weights as a state MLP model which has 22 hidden layer neurons. Unfortunately, we were unable to produce an RCC model with 28 hidden layer neurons due to intensive training time requirements by the RCC algorithm. We had to interrupt the training session after 3 weeks of continuous computation after which 27 hidden layer neurons were added. At this stage, the adding of the 28-th hidden layer neuron would have taken nearly a further 4 days. However, we did not have the option of training for more than 3 weeks. Figure 8.1 gives a performance curve obtained when plotting the number of hidden layer neurons against the overall network performance.

From this curve it is possible to predict that the RCC architecture with 28 hidden neurons would have performed at about 85% on the training set and at about 73% on the test set. This performance would be slightly worse than the performance achieved with a state MLP model featuring 22 hidden neurons. Note that the experiment with the RCC model could only be executed once due to excessive training time requirements. It can be assumed that by choosing a larger size for the pool of candidate units, and different initial network parameters, that the performance

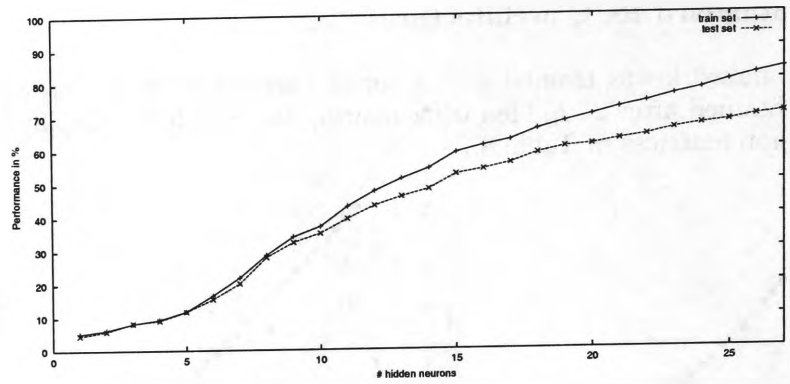


Figure 8.1: Training an RCC model with up to 27 hidden layer neurons on the logo recognition problem.

of the RCC model matches the performance of a state MLP model as long as the total number of network parameters is similar.

8.4.2 Reduced RCC architectures

The experiment described in Section 8.4.1 has been repeated on the reduced RCC model. The results using the visualisation of confusion matrices are shown in Table 8.8.

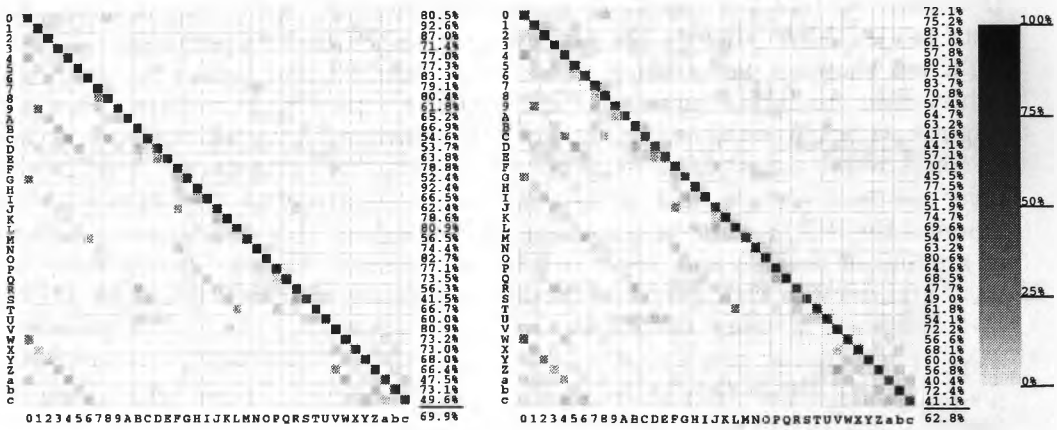


Table 8.8: The confusion matrices obtained from a reduced RCC network featuring 22 hidden layer neurons. The matrix to the left gives the results obtained from the training data set, while the matrix on the right gives the results obtained from the test set.

The results obtained from a reduced RCC model featuring 22 hidden layer neurons are shown in Table 8.8. It is found that the performance is significantly worse compared to a state MLP architecture and also worse than those obtained for a RCC architecture. As before, this result can be attributed to the fact that a reduced RCC model features considerably less parameters than a state MLP model. A reduced RCC network architecture would have to feature 30 hidden layer neurons in order to have a similar number of network parameters as an equivalent state MLP model. The training of this architecture is computationally intensive. Unfortunately, the training session had to be interrupted after 3 weeks of computation. At this stage

the network had 27 hidden layer neurons. It was predicted that the addition of a further 3 hidden layer neurons would have required another 12 days of computation. Again, we were not given the option of exceeding 3 weeks training time. Figure 8.2 gives the performances obtained after the adding of the n-th hidden layer neuron.

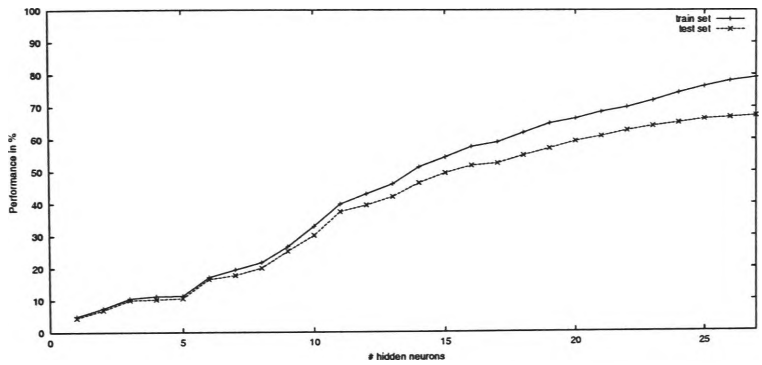


Figure 8.2: Training a reduced RCC model on the logo problem.

It is observed that the performance curve starts to level beyond 25 hidden layer neurons. This is particularly the case for the curve which is associated with the test set. Hence, it can be predicted that a reduced RCC model with 30 hidden layer neurons would probably not be able to match the performance of a state MLP model. In particular, the generalization performance can be expected to be well below the generalization performance achieved with a state MLP model with similar number of network weights. This supports earlier findings given in Chapter 7 which states that the generalization performance of an RCC model may not be as good as for RMLP based models.

8.5 Conclusions

The findings of this chapter are multifold and are as follows:

- All models addressed in this chapter produced reasonable results on this real world learning problem. This demonstrates that the models are a reliable and effective method for the encoding graph structured information.
- Training times vary considerably with the neural network model under consideration. This can be attributed to the computational complexity of the algorithms. The complexity of the SOM-SD (supervised and unsupervised) algorithm grows linearly with the number of neurons, the out-degree, and the number of training samples. However, SOM-SDs typically feature considerably more neurons than other neural network models so that training a SOM-SD model can take considerable time as well. The RMLP training algorithm grows in complexity quadratically with the number of hidden layer neurons and has a linear dependency on other factors. In contrast, the RCC algorithm grows quadratically with respect to the number of hidden layer neurons and the out-degree. A consequence of the different complexities is reflected by the following table which gives training times [2] typically required for the logo recognition learning problem.

[2]Training times are approximate times relative to a Pentium-1GHz type CPU.

Algorithm	Complexity	Time required
SOM-SD	$O(Nnc)$	7 days
RMLP	$O(Nn^2c)$	2 days
RCC	$O(Nn^2c^2P)$	> 20 days

where N is the total number of nodes in the training set, c is the out-degree, n is the number of (hidden) neurons, and P is the size of the candidate pool for the RCC model. These complexities hold when execution is performed sequentially. All neural network models have in common that they can be implemented on a massively parallel environment so that execution can be performed in parallel. A parallel execution of these algorithms would ideally reduce the complexity to linearly dependent on the network parameters.

- The number of network parameters can differ significantly between the state MLP model and the RCC model particularly when the number of hidden layer neurons grows. It was found that the network performance depends more strongly on the total number of network parameters than on the number of neurons in the hidden layer. Hence, an RCC network requires more hidden layer neurons in order to produce a similar performance. Since an RMLP network architecture features $mn_1 + pn_1 + n_1^2$ network weights, and an RCC model has $mn_2 + pn_2 + \frac{1}{2}cn_2(n_2+1) + \frac{1}{2}n_2(n_2-1)$ weights, it can be estimated how many more hidden layer neurons that an RCC model will require in order that it will have approximately the same number of weights as an equivalent state MLP model. Alternatively, it can be estimated how many hidden neurons an RMLP network requires given an RCC model which dynamically obtained the number of hidden layer neurons.
- The data set used for the experiments featured the same number of training samples for each class. Hence, we did not observe problems as with the benchmark learning task where patterns from poorly represented classes were not classified correctly.
- All models showed that the generalization performance is about 10% below the performance achieved by the training data set. This is a good indication of a training data set which does not fully cover the learning problem at hand. In other words, the test set features many patterns which are not represented by patterns in the training data set. This shows that a well chosen and well sized training data set where it is practicable [3] is a requirement for obtaining a good network performance. In practice, there are a number of ways to artificially increase the cardinality of the training data set and improve the coverage of the training set. For example, in some practical situations, the quality of the given training data set can be enhanced by adding intentionally distorted training patterns to the training set.

[3]Note that in most practical problems, the size as well as the features are given. They cannot be chosen at will by the user. If the user can choose the number of training patterns as well as where they are located, this is called "active learning" in the literature.

Chapter 9

Conclusions

9.1 Introduction

The extensions of many popular neural network models into the domain of graph structures have produced some more general forms of neural network models. Each of the models addressed in this thesis is capable of encoding graph structured information as well as other common data structures e.g., sequences and fixed size input vectors. The thesis has demonstrated that the proposed models are able to efficiently encode graph structured data, and hence, make it possible to adaptively process the data contained in such domain of applications.

However, limitations do still exist. For example, currently, none of the proposed models is able to accept more general cases of graphs e.g., cyclic graphs and undirected graphs. While there are methods for handling certain types of cyclic graphs [9], it is a very difficult task to handle cycles in graphs in general. A new set of neural network models needs to be introduced for this extended set of tasks. A motivation for this kind of research is likely to come from the area of statistics where models based on Markov models were applied successfully to cyclic and undirected graphs.

The outcome of research performed in this thesis can certainly be rated as very successful. We now have tools at hand which can efficiently process graph structured information in either a supervised or an unsupervised fashion. This opens the door for the application of neural network models to areas where the application was not possible previously. This is particularly true for the areas of molecular chemistry and software engineering. But many other areas also will benefit from these new models because it simplifies the task of feature extraction from the raw data.

The capabilities and the efficiency of the new models have been investigated quantitatively through a relatively large set of experiments. The experiments were executed within a general framework by the utilization of a benchmark problem, viz., the extended policemen benchmark. Consequently, this allowed us to compare the new models with each other and to conclude advantages and disadvantages for each model. For example, the supervised SOM-SD model has demonstrated some flexibility since it is capable of handling incomplete and missing target information. SOM-SD is one of a number of known graph recursive methods which supports unsupervised training. However, the drawback of SOM-SD models is that it performs a mapping from a high dimensional input space to a low dimensional display map so

that the performance levels are typically below those achieved through RMLP and RCC based models. Both, RMLP and RCC models support an arbitrary number of hidden and state layer neurons and are more flexible to the choice of the output dimension. However, RMLP and RCC based architectures can only be trained in a supervised fashion at present.

The structure of this chapter is as follows: Section 9.2, we will give a brief summary of the major findings of this thesis. Section 9.3, we will give a brief description of some of the problems and issues emerged from the investigations in this thesis. Section 9.4, we will give an indication of some of the open questions associated with this research area. Both the issues indicated in Section 9.3 and Section 9.4 present challenges for future research.

9.2 A brief summary of the major findings of this thesis

This thesis discussed the properties, performance and behaviour of eight different recursive neural network models for processing structured information. By doing so, we found that each of the models featured advantages as well as disadvantages. These are summarized as follows:

Model	Advantages	Disadvantages
SOM-SD	• The complexity of the algorithm is linear. • Able to learn from unclassified data. • Able to encode graphs, data sequences, and fixed sized vectors. • Good generalization abilities. • Able to process deep graph structures.	• Large number of neurons required typically. • Unable to utilize class information if it exists. • Unable to encode certain cyclic graphs. • Network size is static and needs to be predefined. • Requires many training samples. • Mapping to a discrete output space can affect the performance.
sSOM-SD	• Same as for SOM-SD except that it is able to incorporate class information into the learning process. This enables sSOM-SD to outperform SOM-SD significantly. • Able to handle missing class information. • Fewer training samples are required. • Insensitive to initial learning parameters. • Requires fewer training iterations than SOM-SD.	• Same as for SOM-SD.

continued on next page

state MLP	• Is a universal approximator. • Typically fastest in training (as the number of neurons is much smaller than for SOM) • Able to encode graphs, data sequences, and fixed sized vectors. • Requires relatively few training data to be trained successfully.	• The complexity of the algorithm grows quadratically with the number of hidden layer neurons. • Can be trained only on classified data. • Can suffer from a long-term dependency problem on deep graph structures. • Determines the number of state neurons by a trial and error process. • Has considerable problems encoding patterns which are poorly represented in the data set.
output MLP	• Is a universal approximator. • More efficient use of network parameters. • Requires relatively few training data to be trained successfully. • Able to encode graphs, data sequences, and fixed sized vectors.	• Same as for state MLP.
extended state MLP	• Same as for state MLP. • Architecture is similar to an RCC.	• Same as for state MLP.
ext. output MLP	• Same as for output MLP.	• Same as for output MLP.
RCC	• Dynamically adds hidden layer neurons. • Less likely to fall into local minima due to use of a pool of candidate units. • Able to encode graphs, data sequences, and fixed sized vectors.	• The complexity of the algorithm grows quadratically with the number of hidden layer neurons and the out-degree. • Generalization performance is not as good as for RMLP networks. • Requires more training samples than a RMLP. • Not a universal approximator. • Can suffer from a long-term dependency problem on deep graph structures.
reduced RCC	• Same as for RCC • Architecture similar to state RMLP network.	• Same as for RCC.

All models have in common that they reduce to 'classic' counterparts if the input graph is a single labelled node, and reduce to recurrent models if the maximum out-degree of the input graphs is 1. A SOM-SD provides an efficient mechanism to encode and cluster graph structured information in an unsupervised manner. RMLP models provide effective supervised mechanisms for the encoding of graphs, and RCC determines the size of the network through a dynamic process. Hence, this thesis has covered a wide range of models which provide unsupervised as well as supervised approaches, static and dynamic architectures.

This thesis applied all of the above mentioned models to a benchmark problem, viz. the policemen benchmark problem which consists of a number of data sets, and a real world learning problem, viz., logo recognition. It was found that the various models performed differently on the given data sets. Table 9.2 summarizes the performance rates obtained when conducting the experiments. Unless stated otherwise, the table gives the *best* performances achieved with those models as opposed to *typical* results reported in earlier chapters.

	Learning problem							
	dataset-1		dataset-2		dataset-3		Logo[a]	
Algorithm	train	test	train	test	train	test	train	test
SOM-SD[b]	100%	100%	100%	99.53%	97.63%	97.33%	88.4%	77.3%
sSOM-SD[b]	100%	100%	100%	99.86%	99.72%	99.53%	94.5%	83.1%
state MLP[c]	100%	100%	NT	NT	99.55%	99.73%	93.2%	83.3%
output MLP[d]	100%	100%	NT	NT	99.68%	99.49%	97.6%	87.5%
ext. state MLP[c]	100%	100%	NT	NT	98.8%	98.48%	95.1%	83.8%
ext. out. MLP[d]	100%	100%	NT	NT	93.39%	92.75%	98.3%	86.9%
RCC[e]	100%	99.8%	NT	NT	99.12%	98.53%	*time!*	*time!*
reduced RCC[f]	100%	99.6%	NT	NT	99.52%	98.96%	*time!*	*time!*

Table 9.2: Best results obtained for the benchmark datasets, dataset-1, dataset-2, dataset-3, and the Logo data set, when learned by the neural network models shown. NT means "Not tried", and *time!* means that training could not be completed within a reasonable time, otherwise classification rates are reported.

[a] Typical results are displayed.
[b] Number of neurons is approx. $\frac{1}{3}$ the number of nodes in the training set
[c] 10 state neurons for the benchmark problems, 22 for the logo recognition problem.
[d] 10 state, 8 hidden neurons (benchmark problem). 22 state, 20 hidden neurons (logos).
[e] 22 hidden neurons (benchmark), 28 hidden neurons (logo)
[f] 22 hidden neurons (benchmark), 30 hidden neurons (logo)

A natural question which arose from such comparison of results is: in practice, what is a good model for the practical engineer who is faced with the issue of having to solve a practical classification problem in the structured data domain. Here the answer can be multifold: it depends on what data is available to the practical engineer.

- If only unlabelled data is available, then a good model to use is the SOM-SD model with about $\frac{1}{3}$ of the total number of nodes in the training data set.
- If we are given also some class labels, then we can use the supervised SOM-SD model, again with about $\frac{1}{3}$ of the total number of nodes in the training data set.
- If we are given a set of input output pairs, and that there are sufficient number of them, then we may contemplate the use of recursive cascade correlation technique to determine the approximate number of state neurons required. Also from the RCC model, it gives us some information concerning the quality of the data set (whether the given data set covers the entire space spanned, i.e., if the given data set is rich enough to allow us to find good generalization capability of the trained model). This is conditioned on the fact that fast computational resource is available, especially when the out-degree and/or the number of state neurons is large. If there is

not much computational resources available, then one resorts to using the RMLP type of models. In this case, the number of state neurons will need to be determined through a trial and error process. It would be prudent to commence with a state MLP model. If such a model does not give "good" results, then one might consider using an output MLP model. The judgment of whether a failure to obtain good results using a state MLP model is due to the lack of richness of the state model, or the lack of richness in the underlying given data set is made from the experience of practical engineer in encountering this type of problems.

- At this stage, the extended MLP models, and the reduced CC models serve only theoretical purposes. They do not appear to perform well in comparison with other models.

In most cases, the training parameter set which we have obtained in this thesis may serve as a guide in the initial setting of the training parameters. We do not have a theory that it will work in all cases. All we can say is that this set of training parameters might be a good place to start, in view of the lack of any other information on the data set at hand.

Thus, it is observed that handling structured data domain is quite similar to the situations when a practical engineer has to tackle when dealing with problems with fixed sized data, or for sequences in that the engineer needs to experiment with various approaches, armed by some of the experience which we report in this thesis on the proposed models given in this thesis as well other models proposed by others for tackling structured domains. One may lament: this was the situation with using multilayer perceptron around the early 1990's. Surely we must have moved from that position by now [1]. We agree with this sentiment. Indeed, we see this thesis as one which is very much in the tradition of experimental computer science in that we investigate the behaviour of proposed neural network models through extensive experiments. From such extensive experiments, we hope to obtain some insight into the complexity of the behaviour of the models by iteratively interpreting the results, and carrying out further experiments to test our insights, as typified in the approaches carried out in this thesis. Through such experimentation, we hope to build up sufficient insight into the behaviour of the models as to give us some confidence in using these models in practice. Hopefully when we have built up sufficient insight into the behaviour of the proposed models, we might be able to use some analytical means to analyse the behaviour of the models, and thus provide some theoretical insight into their behaviours. Such theoretical insight would give a practical engineer some confidence in deploying the models in practice, knowing their limitations, as well as their strengths.

Comparing the benchmark problem: logo recognition with the extended policeman benchmark problem, we find that the extended policeman benchmark problem is a more difficult learning task. This is because the benchmark problem given by dataset-3 features classes with overlapping properties, and classes that are represented by different numbers of samples. In particular, the experiments revealed that RMLP based models have considerable problems with the classification of patterns which are poorly represented in the training set. The dataset for logo classification did not feature such a property. The reason why the models did not perform and

[1] Yes. Indeed, we have moved a long way since those early days of multilayer perceptrons, thanks to the research efforts in the past decade in clarifying the capabilities of multilayer perceptron models. However, we feel that the investigation into adaptive processing of data structures at the present moment is exactly like multilayer perceptrons in the early 1990's in that there is very little available theoretical results.

generalize on the logo recognition task as good as on the benchmark problem is quite simply: the size and quality of the training set. The training patterns did not sufficiently represent the learning problem at hand in the logo recognition problem. This property was also observed when reducing the size of the benchmark problem. The experiments also revealed that some models such as SOM-SD and RCC require substantially more training patterns than others e.g., RMLP and sSOM-SD. These observations quite clearly justify the use of benchmark problems in evaluating the performance of new proposed algorithms. The size, quality, and property of the data set, and the difficulty of the learning task defined by the extended policeman benchmark problem render it to be a valid tool for the evaluation of statistical and recursive neural network models. The extended policeman benchmark problem includes most of the problems that are encountered in a real world environment.

9.3 Issues and problems which arose from the investigations of this thesis

This thesis presented and introduced a wide range of neural network models which are capable of encoding graph structured information. These models were then evaluated quantitatively on a range of learning tasks. Advantages and disadvantages of the models were found and described. However, the work carried out in this project was often not a straight forward approach. Some of the issues and problems that arose from the investigations of this thesis are given as follows:

- Work in the area of recursive neural models is relatively new so that there was little literature available which could be used as a basis for this research. However, it was possible to generalize existing work and to use known material as a motivation for the introduction of new novel recursive models.
- Attempts were made to introduce an unsupervised mechanism for RMLP based models by using a modified energy function. However, these attempts were unsuccessful and are not reported in this thesis.
- Attempts were made to produce a supervised SOM-SD model which uses Kohonen's original idea by attaching the target vector to the input vector [59]. These attempts were also unsuccessful and are not described in this thesis. An alternative way of introducing supervision to the SOM-SD algorithm was developed with considerably more success. This was reported in Chapter 5.
- Attempts were made to introduce a dynamic SOM-SD model which adds neurons as needed. However, this task could not be completed in time and is not described in this thesis.
- Most models proposed in this thesis were not tested on regression learning problems due to the lack of sufficiently large data sets which could be applied to all models [2]. Hence, the application to regression learning problems is not described in this thesis.
- Some models such as RCC and reduced RCC had high computational demand due to a quadratic computational complexity of the learning algorithm. A thorough practical evaluation of those models was extremely time

[2]We have evaluated the performance of the CC type of models on a molecular chemistry problem. However, this problem has few training samples, thus rendering it impractical to be used in the evaluation of the SOM-SD type models. As a result we decided not to report the results in this thesis.

consuming and in some cases not possible with the computing resources available to us.

- A parallel implementation of the SOM-SD algorithm was attempted. However, the results are not reported in this thesis in view of its "singleton" status, as we have not considered the parallel implementation of other proposed algorithms. We seriously contemplated the parallelisation of the cascade correlation algorithm due to its inordinate long computational time, especially, the time required to obtain further additional neurons beyond a certain point. However, this was met with considerable difficulties.
- Many of the models described in this thesis have a large number of training parameters which could be tuned and their effects could be described. It was not possible to address the effect of all training parameters without exceeding reasonable space limits in this thesis as well as the time and computational resources it takes to fine tune these parameters. Hence, this thesis restricts itself to the evaluation of the more important parameters.
- An underlying theme in this thesis is: the experience and the results of extensive experimentation on some benchmark problems would give us a better chance of tackling real world problems. This is demonstrated in using the set of training parameters which we had obtained in the extended policeman benchmark problem and applied them to the logo recognition problem. It appears that the performance on the logo recognition problem is quite reasonable.

  One may ask: how transferable is such knowledge to other un-tested real world problem. The answer is: the extensive experimentation in this thesis on various models on the extended policeman benchmark problem has heightened our awareness of the type of issues which need to be tackled in a practical real world problem. The direct transfer of the set of training parameters might be fortuitous in the case of the logo recognition problem. But the guidelines obtained, e.g., using about $\frac{1}{3}$ of the total number of neurons in the training data set in the case of a SOM-SD model appears to be a good rule of thumb for most practical cases. As to the transferability of other training parameters to other real world problems this is yet to be tested. So far, the main challenge is to find a suitable practical real world problem which has sufficiently large amount of data which we can use to evaluate the proposed algorithms.
- In the logo recognition problem, with all the proposed models, we could have experimented on a number of training parameters so that its performance can be considerably enhanced. However, due to the computing resources available to us, this is proving to be very time consuming. In addition, our main aim in working with the logo recognition problem is not to obtain the best possible performance, but instead, such application confirms for us that the proposed algorithms when applied on a real world problem give reasonable results.

This list illustrates that much work is still to be done in this area. Some of the open problems which are interesting to study are summarized by the following section.

9.4 Open Problems

There are still a number of open problems in adaptive processing of data structures. Some of these issues are discussed as follows:

1. Minimum number of parameters [2, 72] – As indicated previously, the tree model which we have used includes more parameters than necessary. We have assumed that there are always c children at each node. Hence an interesting question to ask is: could we use a smaller number of parameters.

   This question is related to the problem of minimum encoding a given data structure. In other words, we are asking the question: is there an equivalent formulation of some form of information criterion for the tree structure. While it is intuitively clear that some such formulation could exist, as far as we are aware, there are no published results in this area.

   Note that we have previously considered some ideas on regularisation and pruning. These ideas can be used to reduce the total number of parameters in the tree model (in the case of RMLP type models). It is useful to note that since all nodes use the same MLP model, it is unlikely that there are many parameters which can be trimmed. Nevertheless, an interesting question to ask: how would the generalization capability of the models be affected if we prune some of the parameters, using for example, a brain damage type of algorithms.

2. Long term dependency [5]. Another interesting question is: could the tree model exhibit some kind of long term dependency problem. In MLP training, it is known that if there are too many hidden layers, then because of the fact that the backprop error is multiplied by the derivative of the sigmoidal function which is between 0 and 1, it is plausible that the product of the derivatives and the gradient for very deep layers could become very small, thus, the parameters are not updated.

   In the case of tree models, it is not known whether the same phenomenon occurs or not. If we have a very deep tree, this could conceivably happen. However, because each node uses the same MLP model, it is envisaged that the long term dependency problem is not severe. One could imagine that even though the parameters are not updated in a particular branch due to long term dependency, however, they could still be updated due to the contributions from shorter branches. However, todate, we do not know of a formal theory which proves this formally.

   From the work carried out in this thesis, it is observed that long term dependency might occur in the extended models. This manifested itself in the training of these extended models. We have given some heuristic explanation of the reasons for some of the observations on these extended models. However, it is not too clear how to analyse the situation formally. One way in which this can be performed is through the unfolding of the extended MLP architecture, in a manner very similar to the "folding architecture" proposed in [75]. However, as far as we are aware, no one has performed such analysis yet.

3. Parameter dependency. In the models introduced here, we have assumed that the parameters stay constant. However, it is possible to formulate a tree model in which the parameters are allowed to vary. This type of models would be useful for internet modelling purposes. In this case, if we have a large tree, there could be many parameters which need to be estimated, i.e., we may face an explosion of parameters.

   On the other hand, it may be possible to adapt the idea of "alternative time delay operators" as discussed in [4] which is a kind of linear filter on the evolution of time history of the system. In this case, it might reduce the total number of parameters, while at the same time allowing the system to model some time variations.

4. Cyclic and undirected graphs. In this thesis, we have considered only a very simple tree model. However, it is possible to have a more complicated graph model, whereby there are cyclic paths in the graph. In this case, we need to solve a set of algebraic–difference equations. The algebraic part can be solved by using some kind of numerical relaxation methods. However, todate there is no computational experience with this type of models.
5. Extraction of structures. As shown previously, we can use a different structure in the training of the data, than assuming that the true structure is known a priori. Hence, it is interesting to ask the question: is it possible to extract the structure from a trained network. By this, we wish to extract a structure with which the data can be "comfortably" classified [29].

   A particular problem in this area is: given a set of input output training samples, can we extract the underlying structure which generate these samples. If we can do so, then this would allow us to have an automatic method for inferring structures from the data.
6. VC dimensions and learnability. Recently there have been much interest in the question of learnability, i.e., is the given set of patterns learnable. Associated with this is the concept of Vapnik– Chervonenkis (VC) dimension [95]. Basically the idea is: if the VC dimension is finite, then the problem is learnable. On the other hand, if the VC dimension is infinite, then the problem would be hard to learn. Here, the idea of easy or hard to learn is related in the sense of probability. There are some preliminary work in this area [45], where it is shown that even for simple tree structures the VC dimension could be infinite.

The problems indicated in this section are challenging, in that as far as we are aware, there are no published results in these areas. Some of the problems might be very difficult to solve, e.g., the extraction of structures from a set of input output data, while some of the problems may be relatively easy to solve, e.g., the long term dependency problem, at least as it relates to the extended models. All these problems present themselves to be challenges for future research. Solving some of these problems will lead to further maturation of this area, and give confidence to the practical engineers in using these proposed models.

Appendix A

Benchmark problems

A.1 Introduction

Despite the fact that graph based methods are gaining more and more popularity in different scientific areas, it has to be considered that the choice of an appropriate algorithm for a given application is still the most crucial task. A large database of graphs which makes the task of comparing the performance of different graph processing algorithms possible was not available until the simultaneous introduction of two large database of graphs. The database of graphs for isomorphism benchmarking [26] provides 18,200 randomly generated graphs which are categorized into five different classes "Randomly Connected Graphs", "Regular Meshes", "Irregular Meshes", "Bounded Valence Graphs", and "Irregular Bounded Valence Graphs". The size of the graphs ranges from few dozens to about 1000 nodes. The dataset is made available on a CD-ROM. The second database of graphs is called "The Policemen Benchmark" [42, 39, 40]. The policemen benchmark provides pairs of artificially generated images and associated graph structures. The datasets are of arbitrary size and are produced from attributed plex grammars. The benchmark problem is distributed over the internet so that it can be made widely available to the research community [38].

We decided to utilize the Policemen Benchmark Problem for the evaluation of the models described in this thesis for the following reasons:

(1) The graphs are not generated by a random distribution, and hence allows evaluations in a controlled manner.

(2) The size of the dataset is arbitrary and the width (out-degree) and depth of the graphs are also arbitrary, and hence allows the evaluation of models on particular types of graphs.

(3) Pairs of graphs with associated images allow us to more easily visualize experimental results. For example, it is easier for a reader to detect similarities between two images than between two graphs.

(4) The dataset is produced by an attributed plex grammar which requires very little storage space (typically less than a few kilo-bytes).

(5) The Policemen Benchmark is receiving support from the community of Adaptive Processing of Graph Structures and is expected to develop into a standard dataset for benchmarking purposes. In comparison, the properties

of the dataset for isomorphism benchmarking makes it more suitable to the community of Graph Matching Algorithms. The application of the isomorphism in the area of Adaptive Processing of Graph Structures is limited due to its static properties and the fact that it is generated through random distributions.

The Policemen Benchmark is part of the contributions of this thesis and hence, will be explained in greater detail in the following sections. Section A.1.1 to section A.1.4 give an introduction to relevant grammars and to methods employed for the policemen benchmark. Section A.2 describes the procedures that lead to the generation of the benchmark problem and describes its properties. The datasets used in this thesis are described in section A.2.1. Finally, a resumeé is drawn in section A.3

A.1.1 Syntactical/Structural Pattern Recognition

A basic idea underlying a syntactical/semantic pattern recognition approach is the explicit use of structure for pattern recognition [13]. Using a structure, it is possible to recursively build up a complex pattern starting from elementary components, often known as terminals (or primitives), and expressing the relationship among the various parts of a pattern using linkages among these components. There are normally two issues associated with syntactical pattern recognition, viz., pattern representation, and pattern recognition. In pattern representation, a "grammar" is introduced which gives means whereby a complex pattern can be built up from components. In pattern recognition, given a pattern, the question to ask: is this pattern described by the given grammar. This often involves "parsing" a given pattern. Patterns can be described in many different formats. In this paper, we will only consider patterns which can be described by "strings", i.e., a concatenation of alphabets. The study of syntactical/semantic pattern recognition is under-pinned by the theory of formal grammars, and languages, which have been developed in the past few decades [34, 1].

Formal grammars operate on finite sets of symbols, known commonly as alphabets. An alphabet A is a finite set of symbols. A *word* x over A is a sequence of symbols

$$x = a_1 \dots a_n$$

where $a_i \in A$, $i = 1, 2, \dots, n$. It is possible to define an *empty* word, which is denoted by the symbol ϵ as a sequence with no symbols. The length of a word, denoted by $|x|$ is equal to the number of symbols contained in the word. The set of all words over an alphabet is denoted by A^*. A^* can be finite or infinite. The set of words over an alphabet excluding the empty word is denoted by A^+, i.e., $A^+ = A - \{\epsilon\}$. The concatenation of two words $x = a_1 a_2 \dots a_n$, and $y = b_1 b_2 \dots b_m$ is given by $xy = a_1 \dots a_n b_1 \dots b_m$.

Definition 1 *A formal grammar is a four tuple*

G	=	$\{N, T, P, S\}$, *where*
N		*is a finite set of non terminal symbols*
T		*is a finite set of terminal symbols*
P		*is a set of productions*
S	$\in$	$N \cup T$ *is the set of initial symbols.*

Further, $N \cap T = 0$. *The vocabulary is given by* $V = N \cup T$. *Each production* $p \in P$ *is of the form* $\alpha \rightarrow \beta$, *where* α *and* β *are called the left hand and right hand side component list respectively, and* $\alpha \in V^+$, *and* $\beta \in V^*$.

The grammar allows the production of words from the terminals, and non terminals using production rules. The production rule $\alpha \rightarrow \beta$ means that one substitutes the left hand side occurrence of α by the right hand side β, and obtain a new word. This can be formally defined as follows:

Definition 2 *Let G be a grammar, and $\alpha \rightarrow \beta$ a production in P. Any word $v = x\alpha y$ with $x, y \in V^*$ can be derived into the word $w = x\beta y$. We write this as $v \rightarrow w$. If we have $v = v_0, v_1, \ldots, v_n = w$, such that $v_i \rightarrow v_{i+1}$; $i = 0, 1, \ldots, n-1$, then we write it as $v \overset{*}{\rightarrow} w$.*

Definition 3 *The language generated by a grammar G is given by*

$$L(G) = \{x | x \in T^*,\ S \overset{*}{\rightarrow} x\}$$

An element of $x \in L(G)$ is obtained from the set of initial symbols S, by applying the production rules recursively until a word is obtained which contains only symbols from the terminal set. Note that even though the set of initial symbols and terminal symbols may be finite, the set of words generated from a set of production rules can be infinite.

For example, in a particular grammar $G = \{\{S, A, B\}, \{a, b, c\}, P, \{S\}\}$ with the following productions:

$$P = \{S \rightarrow cAb, A \rightarrow aBa, B \rightarrow aBa, B \rightarrow ab\}$$

To form the word *caacbaab*, we can use the following:

$$S \rightarrow cAb \rightarrow caBab \rightarrow caaBaab \rightarrow caacbaab$$

Conceptually, there can be many different types of grammars, and associated with them, there are many different types of languages. One way in which the various languages can be classified is to use what is commonly known as the Chomsky's hierarchy [1].

Definition 4 *Formal languages are classified according to the Chomsky's hierarchy into:*

1. *Regular languages, or type 3 language if and only if any production rule is of the form $A \rightarrow aB$, or $A \rightarrow a$ where $A, B \in N; a \in T$.*
2. *Context free or type 2 language if and only if any production rule is of the form $A \rightarrow z$, where $A \in N$; $z \in V^+$.*
3. *Context sensitive or type 1 language if and only if any production rule is of the form $xAy \rightarrow xzy$, where $x, y \in V^+$; $A \in N$; $z \in V^+$.*
4. *Unrestricted, or type 0 language if there are no restrictions on the production rules.*

In addition, a language of type i cannot be generated by a language of type $i+1$.

The grammar gives a representation of the underlying behaviour of the entity. It represents the entity by using sets of symbols, viz., the initial symbol set S, and the terminal symbol set T. The set of non terminal symbols is N. The production rules

operate on the initial symbol set S, and derive the set of acceptable words x. By using different sets of production rules, ranging from no restraints, to well defined constraints, words can be obtained which have different properties.

Parsing is used for recognition. It tackles the problem of deciding if a given string belongs to a grammar. In other words, it asks the question: given a string, can we decide if such a string can be generated by a given grammar with a given set of initial symbols, terminal symbols, and non terminal symbols.

Definition 5 *Let $G = \{N, T, P, S\}$ be a context free grammar. A derivation tree is a tree where:*

1. *Each node is labelled with a symbol $z \in V$ such that*

   each leaf is labelled with a symbol $a \in T$
   each non-leaf is labelled with a symbol $A \in N$
   the root is labelled with the initial symbol S

2. *If there exists a node with label $A \in N$ such that its successor nodes are labelled with $x_1, x_2, \ldots, x_n \in V$, then there exists a production $A \rightarrow x_1 \ldots x_n \in P$.*

The derivation tree for the example in generating the word *caacbaab* is given in Figure A.1.

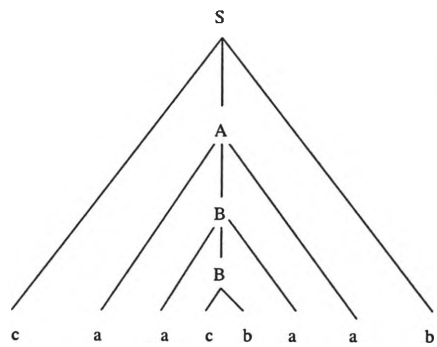


Figure A.1: A derivation tree for generating the word caacbaab

Definition 6 *Let $G = \{N, T, P, S\}$ be a context free grammar. A word $x \in L(G)$ is called unambiguous if there exists only one derivation tree of x with respect to G. It is called ambiguous if there exists more than one derivation tree.*

For any grammar G and any word $x \in T^*$, there exists a derivation tree according to G, if and only if $x \in L(G)$. So, the task of deciding if $x \in L(G)$ is equivalent to the construction of a derivation tree. This is a parsing problem. There are in general two main types of parsing, viz., top down parsing and bottom up parsing. In top down parsing, the derivation tree is constructed from root towards the leaves while in bottom up parsing, the derivation tree is built from the bottom, i.e., the initial symbols towards the root by using the production rules.

The above mentioned grammars and languages are deterministic in nature, i.e., once the set of initial symbols, terminal symbols, and non terminal symbols are defined, and the set of production rules is given, the set of words can be derived. The representational power of these grammars can be shown. For example, it can be shown that type 0 languages, i.e., unrestricted grammars is Turing [92] equivalent.

The set of regular languages can be shown to represent linear systems, while the set of context free languages can be used to represent bilinear systems.

On the other hand, often, it is more desirable to represent underlying entities which require a richer structure. There are a number of approaches, including the following:

1. Probabilistic grammars. Instead of having a deterministic grammar, we wish to introduce probability concepts into the production rules.
2. Attributed grammars. This allows one to attach attributes to the description of the vocabulary. In a more general case, we could use plex grammars, which allows us to specify how different components can be fused together to form a more complex pattern.

These will be considered in the following subsections.

A.1.2 Grammars with probability

We will only consider context free grammars for simplicity. Let the set of non terminals of a grammar G be: $N = \{A_1, \ldots, A_m\}$. Let the set of productions be

$$\begin{aligned} A_1 &\rightarrow X_{11}, \ldots, A_1 \rightarrow X_{1,n1} \\ &\vdots \\ A_m &\rightarrow X_{m,1}, \ldots, A_m \rightarrow X_{m,nm} \end{aligned}$$

Definition 7 *A stochastic context free grammar is a four tuple* $G^* = \{N, T, P^*, S\}$ *where:*

1. *N, T, and S are respectively the set of nonterminal symbols, terminal symbols, and initial symbols.*
2. *P^* is a finite set of productions of the form $p_{ij} : A_i \rightarrow X_{ij}$ with $A_i \in N$, $X_{ij} \in V^+$, $0 < p_{ij} \leq 1$, $\sum_{j=1}^{n_i} p_{ij} = 1$, $i = 1, 2, \ldots, m$*

Definition 8 *The* characteristic *grammar $G = \{N, T, P, S\}$ of a stochastic grammar G^* is obtained by deleting the numbers p_{ij} from each production in P^*.*

Definition 9 *Let $G^s = \{N, T, P^*, S\}$ be a stochastic grammar. The language generated by G^s is defined by $L(G^s) = \{|x, p(x|G^s)||x \in T^*, S \stackrel{*}{\rightarrow} x\}$.*

Stochastic grammar can be obtained from the characteristic grammar if one associates multiple possibilities with each production rule. These multiple possibilities are controlled by a probability. Hence the probability associated with the same left hand side must add up to 1.

Parsing of a probabilistic grammar can be obtained by using Bayes' rule, and maximum likelihood decision [13].

A problem with probabilistic grammar is that one is required to prescribe the probability involved in the production rules. To obtain these probabilities would require substantial amount of data. As a result, it is not convenient to use these concepts if one has only limited amount of data.

On the other hand, not all complex objects can be generated by stochastic grammars. In some cases, it is more convenient to build them up from primitives, using a

more powerful means of joining the primitives together other than by using concatenation alone. This is one of the main impetus for introducing the idea of attributed grammar, or attributed plex grammar as in the following subsections.

A.1.3 Attributed Grammars

In the representation of structural properties, often we require means to describe the pattern, e.g., lengths, orientations. These cannot be incorporated in the deterministic grammar considered previously. One way in which this information can be incorporated is to use the concept of attributed grammars. The idea is to augment each grammar symbol $Y \in V$ by a vector of attribute values: $m(Y) = (x_1, \ldots, x_k)$, where an attribute α is a function $\alpha : Y \rightarrow D_Y$ mapping a symbol $Y \in V$ into a domain D_Y of numerical values. An attribute vector can be interpreted as a numerical feature vector.

For example, in the traffic policeman case, we can describe each of the terminals by a set of attributes. In this case, we could incorporate the colour, the size of the terminals as a way to describe them. For example,

$$\text{terminal} = \begin{cases} \text{colour} \\ \text{xscale} \\ \text{yscale} \end{cases}$$

where *xscale* denotes the scale along the x–axis, while *yscale* denotes the scale along the y–axis. *terminal* denotes the terminal object, which can be "Hat", "Face" etc.

A.1.4 Plex grammars

Plex grammar is a general form of attributed grammar. It allows us to attach structures together to form a complex structure. This was introduced by Feder [25]. The basic idea is to introduce symbols with an arbitrary number of attachment points. Complex structures are formed by joining an attachment point of one object with an attachment point of another object.

A symbol with n attaching points is called an n-attaching point entity or NAPE. Structures obtained by joining NAPEs are called plex structures. Formally, a NAPE is represented by an identifier and a list of its attaching points, $L(n_1, \ldots, n_m)$, where L is the identifier, and $n_1, \ldots, n_m$ are the attaching points. A plex structure is described by three components:

1. A list of NAPEs
2. A list of internal connections between NAPEs
3. A list of attaching points where the structure can be joined with other NAPEs or plex structures.

NAPEs are called terminal if they represent basic primitives as defined through the grammar, NAPEs are called non-terminal if they are formed by joining other (terminal or nonterminal) NAPEs. As an illustrative example: Two terminal NAPEs are shown in Figure A.2. In NAPE "ver", there are three points identifies as, 1, 2, and 3 which can be used for connections to other NAPEs, or other plex structures. The NAPE "hor", on the other hand, has only two points, 1 and 2 which can be used to connect to other NAPEs.

The number and location of connecting points in a terminal NAPE is arbitrary. It is open to the user to specify. Connecting points of nonterminal NAPEs are specified

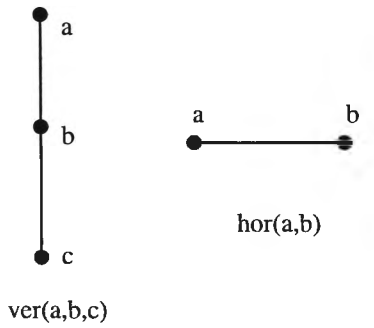


Figure A.2: An example of NAPEs

through 'inheritance' of connecting points given in its elements. These elements in turn can be either nonterminal or terminal NAPEs. Hence, the plex structure offers a very powerful way in which complex structures can be built up from simple structures.

A string grammar generating a set of plex structures is called a plex grammar. In such a grammar, terminals correspond to objects with user specified connecting points. Similarly, the set of nonterminal NAPEs is a set of symbols used to describe intermediate plex structures. Thus, the plex grammar can be defined in a very similar manner to the context free grammars. Plex grammars are formally defined through the six touple

$$G = (N, T, P, S, I, i_0)$$

where

N is a finite nonempty set of NAPEs called *nonterminal objects.*
T is a finite nonempty set of NAPEs called the *terminal objects.*
P is a finite set of productions;
$S \in N \cup T$ is a starting NAPE.
I is a finite set of identifiers;
$i_0 \in I$ is a null identifier.

It is required that $T \cap N = \emptyset$. For convenience we also assume $I \cap (T \cup N) = \emptyset$ and, as before, $V = N \cup T$. The symbols in I are used to identify the attaching points of NAPEs. No two attaching points of the same NAPE have the same identifier. The null identifier i_0 serves as a place marker and is not associated with any attaching point of any NAPE. Connections of NAPEs can only be made through specified attaching points where the null identifier is not permitted to serve as a connecting point.

The productions in P are defined as follows:

$$\text{NAPE}(p_1, \cdots, p_n) \rightarrow (\text{NAPE}_1, \cdots, \text{NAPE}_m)(\mathbf{c}_1, \cdots, \mathbf{c}_m)(\mathbf{d}_1, \cdots, \mathbf{d}_n)$$

where $p_i \in I$ denotes to an external connection point identifiers associated with the nonterminal NAPE, the list $(\mathbf{d}_1, \cdots, \mathbf{d}_n)$ defines the connecting points stated in $(p_1, \cdots, p_n)$ by listing fields of connecting points that are to be 'inherited' from the list of NAPES stated on right side, and $(\mathbf{c}_1, \cdots, \mathbf{c}_m)$ is a list of fields describing how the NAPES on the right hand side interconnect to form the nonterminal NAPE.

Example. Given the NAPEs "hor" and "ver" as shown in Figure A.2. These NAPEs can be used to create the letters H, F, and L using the following grammar.

$$G = (\{Letter, Help\}, \{ver, hor\}, P, \{Letter\}, \{0, a, b, c\}, 0)$$

where

$$P = \left\{ \begin{array}{lcl} Help(a,b,c) & \rightarrow & (ver,hor)(ba)(a0,0b,c0) \\ Letter() & \rightarrow & (ver,hor)(ca)() \\ Letter() & \rightarrow & (Help,hor)(aa)() \\ Letter() & \rightarrow & (Help,ver)(bb)() \end{array} \right\}$$

The rule $Letter() \rightarrow (ver, hor)(ca)()$ is interpreted as follows: there are two terminals, viz., *ver* and *hor*. The point c on terminal *ver* is joined with the point a of the terminal *hor*. Hence this produces the letter L. Since this is the end result of what we wish to achieve, there are no further points required in order to connect with other NAPEs or plex structure, hence there are no specified external connecting points. The interactions described by this production is visualized in figure A.3.

Figure A.3: The production $Letter() \longrightarrow (ver,\ hor)(ca)()$

In a similar manner, the rule $Help(a, b, c) \rightarrow (ver, hor)(ba)(a0, 0b, c0)$ is interpreted as follows: *Help* is a non terminal symbol. It is formed from the two terminal NAPEs, *ver* and *hor* by joining point b on *ver* with point a on *hor*. There are three points which can be connected with the outside world. These are the points $a0, 0b, c0$. By convention, $a0$ is interpreted as the point a on the NAPE *ver*. The 0 denotes that NAPE *hor* is not involved in this external connecting point. $0b$ is the connecting point b on *hor*, while $c0$ corresponds to the connecting point c on *ver*. It is noteworthy to mention, that the list in the Extended Connection Point List may be unordered. Thus, the production $Help(c, a, b) \rightarrow (ver, hor)(ba)(c0, a0, 0b)$ is legal and will produce an identical output. This production is displayed in Figure A.4.

Figure A.4: The production $Help(a, b, c) \longrightarrow (ver,\ hor)(ba)(a0, 0b, c0)$

The rule $Letter() \rightarrow (Help, hor)(aa)()$ can be interpreted as follows: it is formed by joining the point a on the non terminal *Help* together with point a of the NAPE *hor*. Thus, this forms the letter F. Note that point a of the non terminal *Help* corresponds to the point $a0$ when we form the non terminal *Help*, i.e., it corresponds to the point a as specified in the output of the formation of *Help*. The

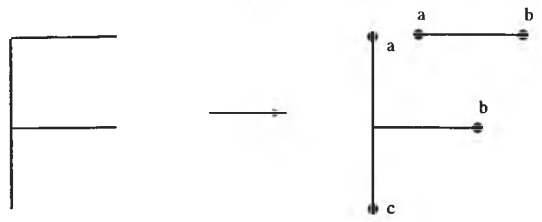


Figure A.5: The production $Letter() \longrightarrow (Help,\ hor)(aa)()$

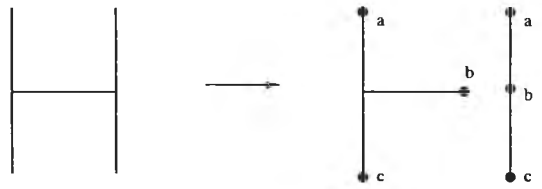


Figure A.6: The production $Letter() \longrightarrow (Help,\ ver)(bb)()$

component interconnection that takes place in this rule is illustrated in figure A.5. The remaining production that forms the letter H is displayed in Figure A.6.

Thus, it is observed that the plex grammar is a very powerful means of building complex objects. However, the classic approach in plex grammars does not use static objects to restrict size or orientation. As a result, plex grammars are unable to produce structures with a particular shape, dimension, or orientation. This, however, is required in many applications including the policeman benchmark. To overcome this limitation, objects are defined to be strictly static in size and orientation [42]. In this approach, the only way to change the size or angle of an object is through additional attributes.

A.2 Generation of the Traffic Policeman benchmark

In this section, we will give more details in how one may build up very complex objects from simple objects. In this aspect, we will produce a grammar to generate a (arbitrary) large set of images from a small set of terminal symbols. The images created shall feature policemen standing on a pedestal giving directions to traffic. This dataset shall be called *the policemen dataset* and will be used as the basis for the *policemen benchmark*. The basics utilized for the generation of policemen patterns will be used towards the end of this section so as to build up other images, such as images showing ships and houses.

Let us consider the following attributed plex grammar:

$$G = (N,\ T,\ P,\ S,\ I,\ i_0)$$

with $I = \{-, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u\}$ being the set of identifiers for attaching points, the null identifier is chosen to be $i_0 = -$. The set of terminal NAPEs shall consist of just two elements named `triangle` and `farc`. Default size, shape, and connecting points of the terminal NAPEs are as shown in Figure A.7.

The starting NAPE S, the list of nonterminal NAPEs N, and the set of plex productions P is obtained as we proceed with this example.

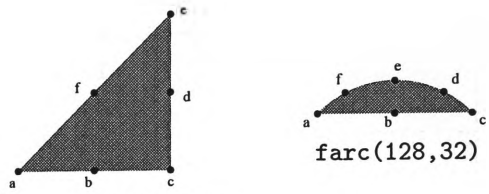


Figure A.7: The two terminal NAPEs used for the policemen benchmark. Shown in the diagram are the shape, default size, and the attachment points of the atoms. The default size is given in brackets. For example, the horizontal extension of `farc` is 128, the vertical extension is 32

In a first step we use the terminal NAPEs to build up other basic geometric objects such as a square and a circle. A square can be built by using two triangles, one of which is rotated by 180 degrees, and connecting them via the connecting point `f`. The production of a square is as shown in Production (1) and is illustrated in Figure A.8.

```
square(a,b,c,d,e,f,g,h,i) → (triangle, triangle={ROTATE=180})
                (ff)(a-,b-,c-,d-,e-,-b,-c,-d,f-)
```
(1)

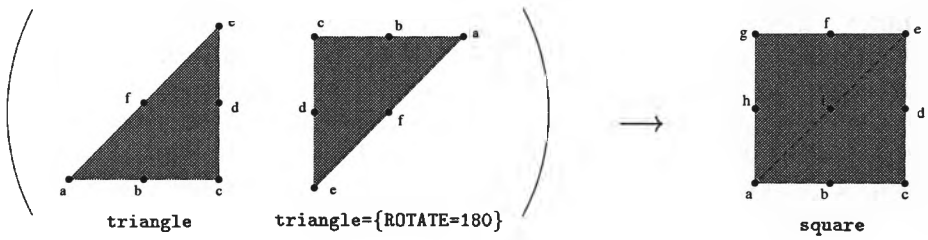


Figure A.8: Production of a square from two triangles.

Note that the term `ROTATE` in Production (1) specifies an attribute for the second `triangle`. The result of this attribute is the rotation of an object by the number of degrees as indicated. The non-terminal NAPE `square` features 9 attachment points which are derived from both triangles. The size of the `square` is derived from its elements. Since neither of its two elements is modified in size by the attributes, the square inherits its size from default values assigned to the `triangles`. Hence, the `square` is has the extension 128 by 128.

Another frequently needed geometric figure is the circle. The circle can be created by attaching appropriately rotated disk segments to either side of a square. Hence, a circle consists of four copies of the terminal `farc` and one non-terminal `square`. These are combined as specified by Production (2).

```
circle(a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u) → (square,
farc, farc={ROTATE=180}, farc={ROTATE=-90}, farc={ROTATE=90})
(fb---,b-b--,h--b-,d---b)(--e--,--f--,--a--,----d,----e,----f,
 ----a,-d---,-e---,-f---,-a---,---d-,---e-,---f-,---a-,f----,
                  h----,i----,d----,b----)
```
(2)

The production of the non-terminal `circle` demonstrates the power of attributed plex grammars. This combination of simple terminal and non-terminal objects has

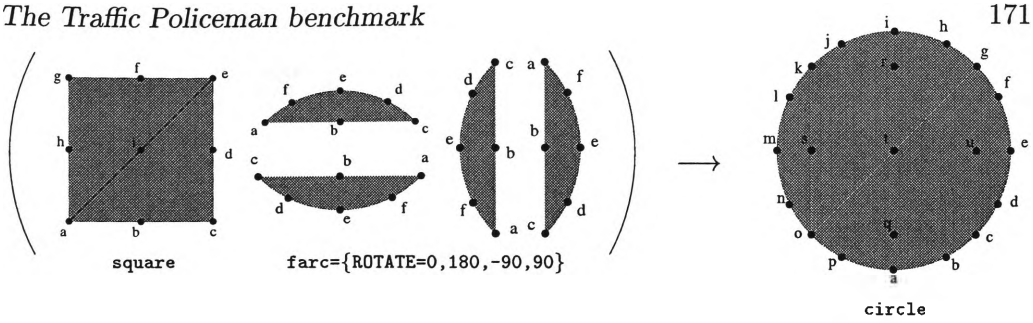


Figure A.9: Production of a circle from a square and four circle segments.

resulted in a `circle` featuring 21 attachment points.

Attributes can be assigned to terminal or non-terminal objects so as to alter the shape of the object such as illustrated by Production (3) which produces a non-terminal called `leg` from a non-terminal `square`.

```
leg(a,b) → (square={XSCALE=0.2})()(b,f)        (3)
```

The attribute `XSCALE` scales an object horizontally by a factor as indicated. In case of Production (3), the non-terminal `square` is shrunk horizontally to just 20% of its original extension. Hence, the non-terminal `leg` is a horizontally distorted version of a square. At the same time, `leg` inherits just two attaching points from `square`. Similarly, the production shown in Production (4) distorts the size of a terminal symbol and reduces the number of attachment points in order to form a new non-terminal symbol named `hat2`. The attribute `COLOR` is used to alter the default color of `farc`.

```
hat2(a) → (farc={SCALE=1.1|1.2|1.25, COLOR=green|red|blue})()(b)   (4)
```

The `COLOR` attribute in Production (4) lists three alternative colors which are separated by a '|' (a vertical line called *slash*). This notation indicates that the default value of `farc` is changed to either one of these colors. Hence, Production (4) creates a non-terminal `hat2` which is either green, red, or blue in color. Similarly, the `SCALE` attribute specifies that the default size of `farc` is increased equally in horizontal and vertical directions by either 10%, 20%, or 25%. A parameter is chosen at random since no probability is assigned to either one of the options. This example demonstrates that a non-terminal NAPE can have many attributes. If a production specifies a range of attributes such as `SCALE=1.1 - 1.25`, then the number of attributes a NAPE can have is infinite. Hence, it is possible to generate an arbitrarily large dataset of distinct images from a small set of terminal symbols using relatively simple attributed plex grammar.

Thus far, our attributed plex grammar has been extended by four non-terminal objects `square`, `circle`, `leg`, and `hat2`. The application of the very same mechanism can be employed to produce the set of non-terminal objects shown in Figure A.10.

This set of non-terminal NAPEs gives body parts of a traffic policeman which can be used to build up a database of images featuring traffic policemen. The productions of the grammar is extended so as to connect the elements shown in Figure A.10. The policeman as shown in A.11 is formed by the grammar [1]

[1]The example assumes that the set of objects shown in Figure A.10 has already been made available through some process (e.g. by another attributed plex grammar).

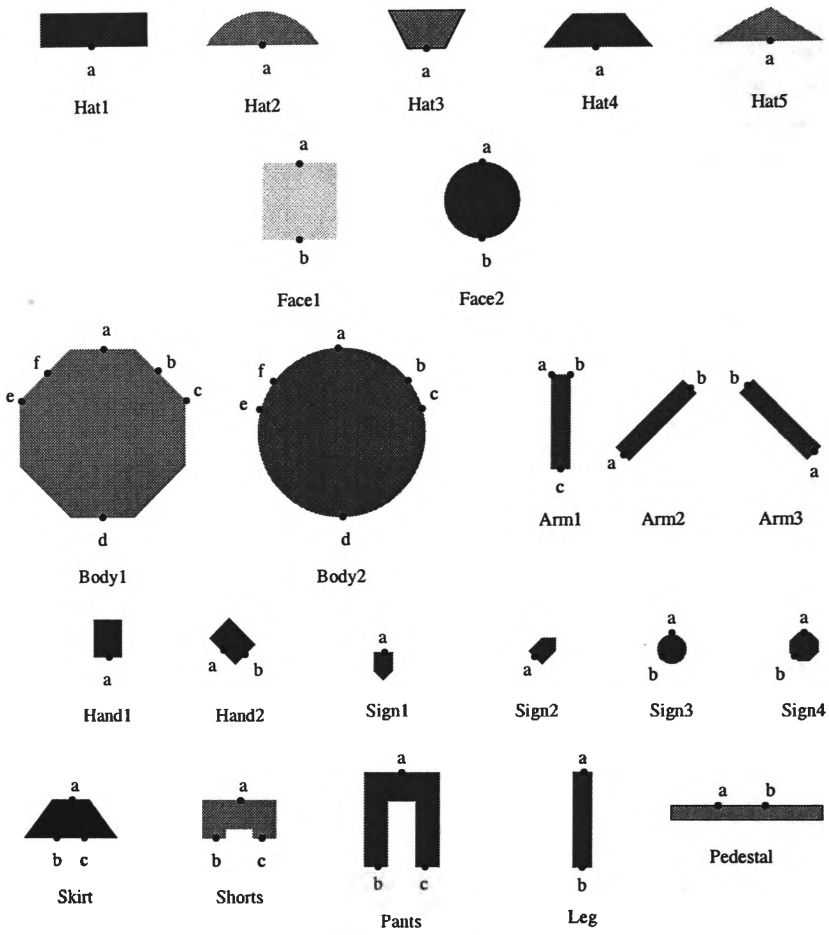


Figure A.10: A set of non-terminal symbols used to produce policemen images. Each non-terminal may vary in size, color, and other by attributes given in he grammar.

$$
\begin{aligned}
N &= (\{Policeman, Head, MBody, Lower, Legs, Rarm, Larm, square, \\
&\quad \{Hat3, Face1, Body1, Skirt, Leg, Arm1, Hand1, Sign1, circle\}, \\
S &= \{Policeman\}, \\
I &= \{0, a, b, c, d, e, f\}, \\
i_0 &= 0
\end{aligned}
$$

where the list P of productions in G is given through:

$$
\begin{aligned}
\text{Policeman} &\rightarrow (\text{Head, MBody, Lower})(aa0, 0ba)() && \text{(A.1)} \\
\text{Head}(a) &\rightarrow (\text{Face1, Hat2})(aa)(b0) && \text{(A.2)} \\
\text{MBody}(a,b) &\rightarrow (\text{Larm, Body1, Rarm})(be0, 0ca)(0a0, 0d0) && \text{(A.3)} \\
\text{Lower}(a) &\rightarrow (\text{Skirt, Legs})(21)(10) && \text{(A.4)} \\
\text{Legs}(a,b) &\rightarrow (\text{Leg, Leg, pedestal})(b0a, 0bb)(a00, 0a0) && \text{(A.5)} \\
\text{Rarm}(a,b) &\rightarrow (\text{Arm1, Hand1})(ca)(a0, b0) && \text{(A.6)} \\
\text{Larm}(a,b) &\rightarrow (\text{Arm1, Sign1})(ca)(a0, b0) && \text{(A.7)}
\end{aligned}
$$

The rules can be explained by using the following diagrams. For example, the rule $Head(1) \rightarrow (Face1, Hat2)(aa)(b0)$ is the linking of the terminals $Hat3$ and $Face1$

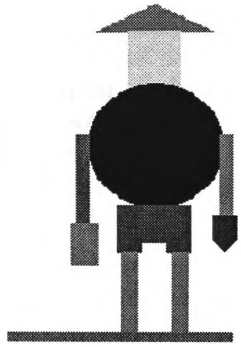


Figure A.11: A policeman generated by the productions A.1 to A.7

together. This is illustrated in Figure A.12. In this case, there is one external connection point of the Head. It is labelled 'a' on the left hand side of the rule. This point corresponds to point 'b' on the terminal *Face*1, as shown on the right hand side of the rule. Point 'a' from the terminal *Hat*3 is joined with point 'b' of the terminal *Face*1.

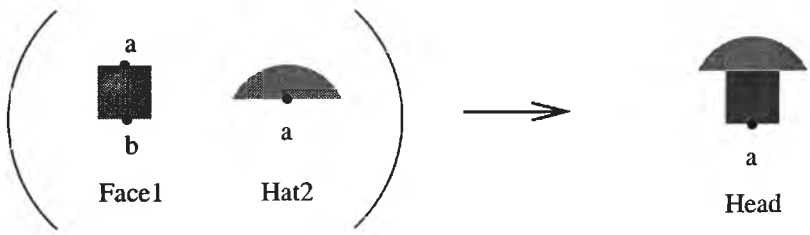


Figure A.12: The production Head(a) → (Face1 Hat2)(aa)(b0)

As another example, we can form the main body of the policeman by joining the head, together with the body as shown in Figure A.13. This is more complicated. There are 2 points of the non terminal *MBody* which can be connected with other NAPEs. These correspond to points a and d of the terminal *Body*1. Point 'a' of the terminal *Head*, the result of the formation from a previous construction, is connected with point 'a' of the terminal *Body*1 to form the non terminal *MBody*.

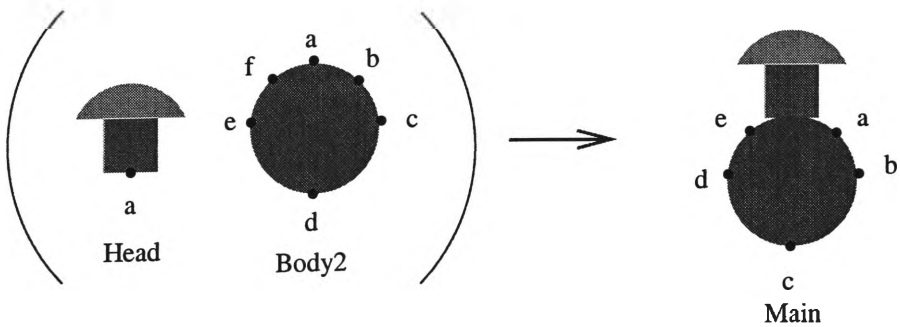


Figure A.13: The production of Main(a, b, c, d, e) → (Head, Body2)(aa)(0b, 0c, 0d, 0e, 0f)

A.2.1 The datasets

The evaluation and comparison of the methods described in this thesis is performed on three different sets of graphs which were extracted from the policeman benchmark problem. The three sets can be seen as a counterpart to the Monks problem [90] used to evaluate conventional adaptive methods, e.g., multilayer perceptrons, cascade correlations.

Dataset-1: The first set which we will refer to as *Dataset-1*, or the Stop-and-Go problem, consists of a set of 1000 graphs belonging to two linearly separable classes. Instances from both classes were equally represented. The learning task is to discriminate between the two classes "stop" and "go", each denoting whether the traffic policeman is signaling a "go" concept or a "stop" concept. To make it more interesting, we have varied the shape of the policeman, e.g., the policeman has an extraordinary long left arm, the policeman has extraordinary long pants, viz., has long legs etc. Approximately half of the examples indicating the concept of "go", and the rest indicating the concept of "stop". These examples are generated by using the attributed plex grammar as indicated previously. A sample of these two instances is given in figure A.14, where the policeman to the left represents a "go" instance (lowered right arm), and the policeman to the right represents the instance "stop" (raised right arm).

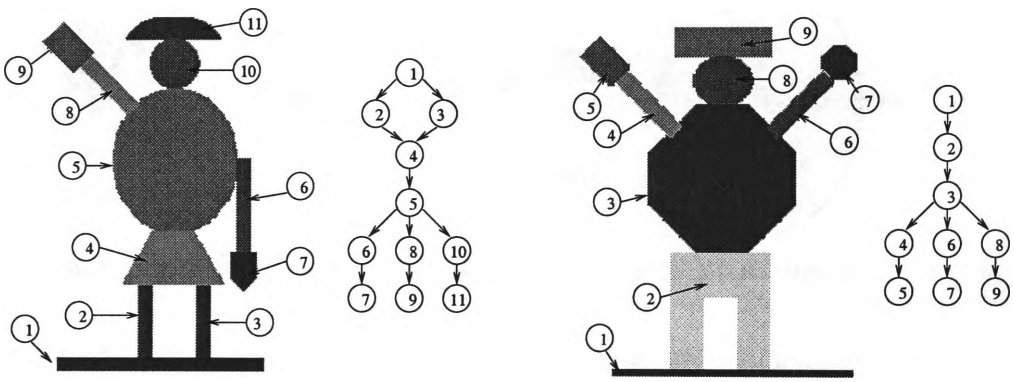


Figure A.14: A selection of artificially generated traffic policemen images and extracted graph structure. Data labels are not shown. Nodes are numbered so to indicate which element they represent.

For the training task, we assume that we do not know anything about the attributed plex grammar with which these examples are generated. Instead, we assume to have only the image of the policeman available. We only assume that the images given are policemen, with the rudimentary concepts that the policemen might have a hat, or no hat, have a body, two arms, hands, signs, legs, skirt, or pants, and the pedestal. We do not assume to know their shapes at all, nor their presence. Hence, the first task which we are required to do is to extract features from these images.

Since we assume that the image is a policeman composed of different parts, we decide to make use of this a priori knowledge in our feature extraction. Since the policeman must stand on a pedestal, this will be the root of the graph with which we will extract the features [2]. We use a simple contour following technique to

[2]Note that it does not matter which node is used as the root. But in this case, we find it convenient to use the pedestal as the root, as we know that all policemen will stand on a pedestal.

determine the shape of the legs of the policeman. Once we have determined the shape of his legs, we compute the center of mass of each of the legs. This results in two features, the x and y coordinates of the legs. We then continue to use the contour following technique and determine if the policeman is wearing a skirt or a pair of pants. Once again, we compute the center of mass of the located object. We continue this process until we finish with the policeman figure.

The learning task has been simplified further by restricting each policeman to consist of at most 9 elements. This has been accomplished by giving each policeman long pants, replacing pants or skirt and the two legs. Since we have nine objects (pedestal, long pants, two arms, two hands or one hand and one sign, body, head, hat), there are altogether 18 features, being the x and y coordinates of the center of mass of these 9 objects. These 18 features will be used as the input features to our learning algorithm. Note that in extracting the features in this manner, we have "ignored" much of the inherent information which comes with the image, e.g., the shape of the components. We could have easily incorporated the knowledge about the shape of the components as features. However, we have decided not to do so as this additional information was not required in this simple experiment.

Figure A.15 and A.16 respectively show the projection of the 18 dimensional feature vectors of the two classes "stop" and "go" onto a two dimensional space. These two diagrams show that when we project the feature space from 18 dimensional space to 2 dimensional space, the two classes appear to be intertwined. In this projection, each data point represents the center of mass of each policeman element. Note that in these plots, the origin has been set to the lower left corner of the image rather than to the upper left corner conventionally used on digital images.

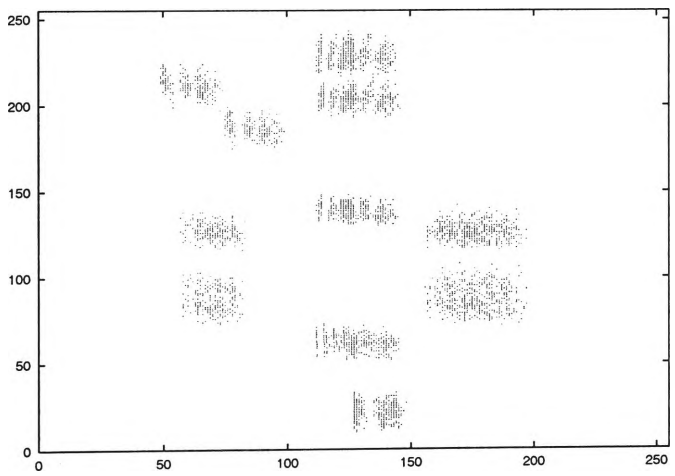


Figure A.15: A diagram showing the projection of the 18 dimensional feature space to two dimensional space for the class "go"

Dataset-2: A second dataset extends the stop-and-go problem and defines a more difficult learning task. Here, the dataset consists of patterns from three different domains. Graphs are extracted from the domains Policemen, Houses, and Ships. Samples are as illustrated in figure A.14 (Policemen), figure A.17 (Houses), and figure A.18 (Ships). Four classes are defined where two of the classes are from dataset-1, the other two classes are defined depending on whether a graph belongs to the domain ships or to the domain houses. Some graphs belonging to one class may feature a structural representation that is identical to graphs from another

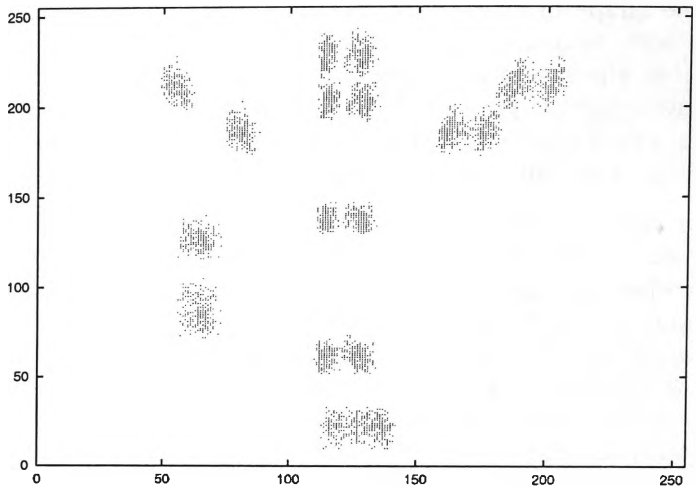


Figure A.16: A diagram showing the projection of the 18 dimensional feature space to two dimensional space for the class "stop"

class e.g., the house numbered 648 in figure A.17 and the ship numbered 1034 in figure A.18. This dataset consists of a total of 7000 graphs. The learning problem is referred to as the *domain learning problem.*

Dataset-3: The third dataset extends dataset-2 by defining 12 classes over the 7000 graphs in dataset-2. The classes are defined so that some classes can only be distinguished through features in the data label, while other instances require structural information in order to be discriminated. Structural representations had been obtained by a scan line algorithm where images are canned from bottom to top, and from left to right. The first colored object found constitutes the root node. Objects directly connected from the offsprings. Applied recursively, all objects are considered and a graph representation is obtained. Each node in the graph receives a two dimensional label stating the $\{x, y\}$ coordinate of the center of gravity of the corresponding object. Examples are illustrated in Figure A.14, Figure A.17, and Figure A.18.

The result is a dataset that provides directed acyclic graphs with the following properties:

Data set	Outdegr. Max.	Depth Min	Depth Max	Num. nodes Min	Num. nodes Max	Num. classes
Policemen	3	4	5	9	11	2
Houses	5	2	3	4	7	8
Ships	6	1	2	3	13	2

Hence, the policemen patterns produce deep narrow graphs, ships and houses have a flat and wide data structure. Some graph structures produced by the ships and houses are identical in structure such as `house648` in Figure A.17 and `ship1034` in Figure A.18. There is no graph structure derived from policemen images that is contained in the domain houses or ships. Thus, some patterns can be distinguished only through features encoded in the labels. For example, when considering policemen, the location of the arm is not encoded in the graph structure, but in the data label, while the graph structure is not affected.

The maximum outdegree of all nodes in the data set is 6. As a result (with the

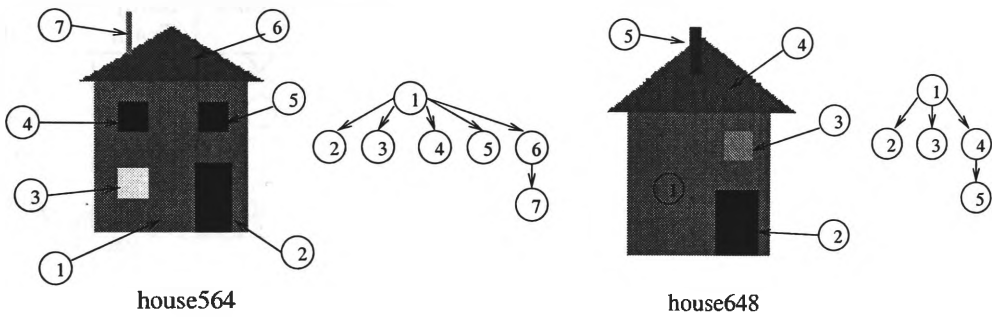


Figure A.17: Artificially generated houses and associated graph structure. Data labels are not shown. Nodes are numbered to indicate which element is represented.

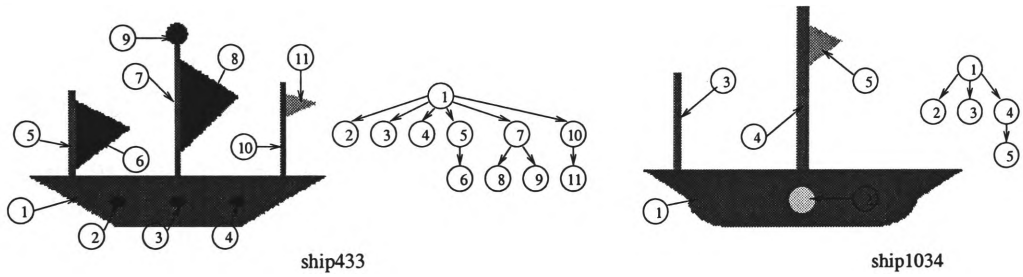


Figure A.18: Artificially generated images of ships. Data labels are not shown.

dimension of the data label $\mathbf{u} = 2$) every input vector $\boldsymbol{v}$ is of dimension $2+2\times6$. The training (test) set contained 3,750 (3,750) graphs with a total of 29,864 (29,887) nodes. For the purpose of testing the network performance, 12 classes are defined as shown in Figure A.2.1. The property of these classes is such that some classes can only be distinguished by considering information provided through data labels such as class 'a' and class 'b'. Some classes require structural information in order to be distinguished such as the classes 'h' and 'k'.

A.3 Conclusions

The attributed plex grammar provides a powerful mechanism for creating potentially infinite sets of images and graphs of arbitrary size. Hence, large datasets of images and graphs can be generated by a relatively small set of attributed plex productions. In addition, the plex language has no restriction for what terminal symbols to use. For example, instead of using basic geometric objects, one could use more complex elements such as a face, eyes, arms, etc. of real people, e.g., obtained from photographs. Such objects can then be appropriately combined by a plex grammar and modified (in color, orientation, etc.) by associated attributes. Hence, it is a relatively simple task to create large sets of artificially generated images which are very similar to real world images.

Another advantage of this approach is that images and associated graph representations are generated by the same procedure. Hence, we are given the option to compare either images or graphs for evaluation purposes.

However, an attributed plex grammar also features some limitations. Firstly, the connecting points are static in number and location. If at some stage during a

Class	Description	Symbol	Samples in set	
			Train	Test
'a'	Policemen with raised left arm		645	670
'b'	Policemen with lowered left arm		605	580
'c'	Ships featuring two masts		937	944
'd'	Ships with three masts		313	306
'e'	Houses without windows		28	21
'f'	Houses with 1 window in (LL) corner		59	59
'g'	Houses with 1 window in (UR) corner		58	64
'h'	Houses with 1 window in (UL) corner		172	195
'i'	Houses with 2 windows in (LL) and (UL)		53	71
'j'	Houses with 2 windows in (UL)) and (UR)		170	173
'k'	Houses with 2 windows in (LL) and (UR)		188	194
'l'	Houses with all three windows		522	473
		Total:	3750	3750

Table A.1: Definition of the 12 classes and their symbols as used later in Fig. 4.14 to Fig. 4.21. (LL) means "lower left", (UL) "upper left", and (UR) "upper right".

production, two objects are to be joined at a point for which no connecting point is defined, then it is not possible to introduce a new connecting point unless affected terminal symbols are re-defined and productions re-written. A second restriction is for alternative attributes or productions as follows:

$$\text{colored\_square} \rightarrow (\text{square=RED}|\text{GREEN})()()$$

which states that the nonterminal `colored_square` is either a red or green square. The likelihood for `colored_square` to be red is the same as for being green. An attributed plex language does not provide an effective mechanism to control the likelihood of certain properties of a non-terminal object effectively. This can only be done through the incorporation of a probability into plex productions.

The policemen benchmark provides at its current stage already a very useful mechanism for evaluating and comparing adaptive systems which are able to process graph structured information. Nevertheless, the limitations of the mechanism employed has stimulating further development in this area. Future versions of this utility are expected to incorporate features of a stochastic grammars as well as intelligent procedures for adding connecting points dynamically.

Appendix B

A real world dataset

B.1 Introduction

There are many reasons for using artificial learning problems. The most common reasons for using artificially generated data are the evaluation of new models in a well controlled manner, and the generalization of learning problems which may be too specific when using real world data. However, using artificial learning data involves the risk of focusing on problems which may not be typical in a real world scenario, or even worse, may not even occur in real world. As a result, real world learning problems are normally applied to verify the results obtained from using artificial data.

This section describes a dataset obtained from a real world problem. We continue to use a learning problem from the area of image recognition since it allows us to produce a better comparison with the artificial learning problem described in Appendix A. In addition, image recognition produces challenging problems which are in many cases ideal candidates for the application to adaptive processes for data structures. And of course, image recognition allows us to illustrate the problem using pictures which help the reader to understand the problem better than when using an abstract learning problem.

The problem described in this section addresses the problem of logo recognition. The learning task is described in the following section.

B.2 Logo recognition

Logo recognition is the central task for some applications in document processing. Thus, the recognition of a company logo printed on the envelope or the header of a document supports the effort to classify the document. For example, the task may be for a machine to automatically sort incoming mails so that mails sharing certain properties are put on the same stack (e.g. one stack to hold orders, another stack to receive application forms, etc.).

A data set of company logos has been made available by the Document Processing Group, Center for Automation Research, University of Maryland. The data set consists of digital gray-scaled images belonging to 39 different instances of logos. This set includes patterns with pure text logos, pure graphic logos, and mixed text and graphic logos. Samples of the 39 instances of logos are shown in Figure B.1.

Figure B.1: The original data set of logos. 39 different instances of logos define the 39 classes for the learning problem. The images are scaled to feature the same horizontal extension.

Since the original data set contains only one sample for each of the 39 logo classes, distortions have been added to simulate the case where patterns are obtained from photocopying or facsimile transmission. Images were distorted by adding up to 50% of random impulsive noise and by rotation up to seven degrees. As a result, images could vary in quality as shown by the example given in Figure B.2.

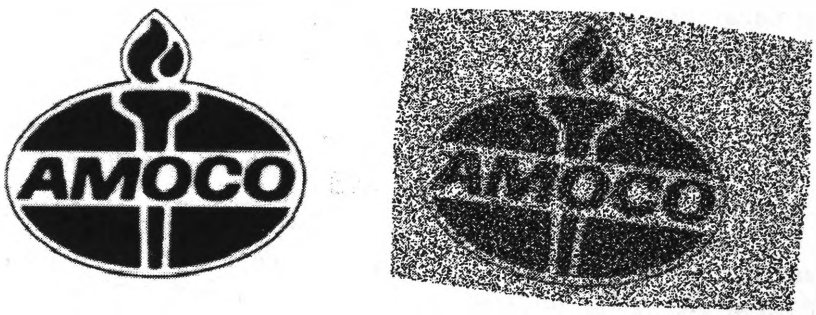
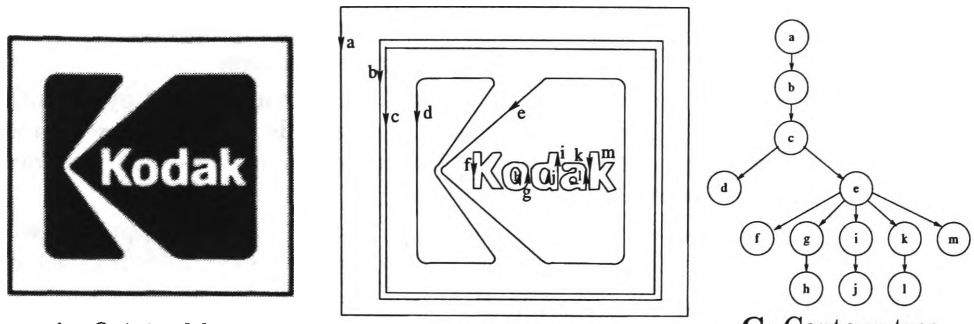


Figure B.2: A logo in original condition (left) and with noise added (right).

A total of 299 randomly distorted versions are added to the original image so that 300 images are available for each of the 39 classes. Hence, the complete set consists of a total of 11700 images, half of which are to be used for testing purposes, the rest for training.



A: Original logo **B:** Contour **C:** Contour-tree

Figure B.3: Graph extraction from a logo using the contour tree algorithm. (**A**) the original logo without noise. (**B**) Contours detected after processing the original image. (**C**) a DOAG obtained from the original logo.

In [28] suggests a method for graph extraction from company logos. The method suggested by [28] is based on a contour-tree algorithm because company logos are made up of simple geometrical elements which are often nested. The algorithm consists of 4 components which are applied in the following sequence:

Step 1: A Gaussian low pass filter is applied to minimize the effect of distorted pixel values. This effectively removes impulsive noise for the greatest part but may also remove small (wanted) features of a logo. Hence, the filtered image can feature spurious elements that were introduced by severe impulsive nose, or may show missing or incomplete elements which were originally part of the logo.

Step 2: A contour following algorithm is applied which effectively detects elements of the logo. Figure B.3 shows the contours detected by following the transition among white and black pixels. When a transition is found, the corresponding contour is obtained by following the transition counterclockwise until the starting point is met again. Thus, all contours are closed. In addition, the contour detection algorithm is insensitive to rotation of the original image.

Step 3: A graph representation is obtained by using the contour tree algorithm [15]. This algorithm associates a node with a contour. The root of the graph corresponds to the most external contour of the image, or the outer bounding box if such a contour does not exist. Offsprings of a node indicate that a given contour is located inside a contour represented by a parent node. An example of a graph produced by this algorithm is given in Figure B.3. There, the root node is created by the bounding box of the image. The root node is linked to as many sub-trees as the contours which reside inside. The complete graph is built by applying this algorithm recursively.

Step 4: In this step, features are extracted from each image contour. These features produce a data label which is then attached to the node that represents this element. The feature-label is a numerical vector which describes some geometrical properties of the contour. There are 12 parameters computed for each contour:

(1) The area consumed by the contour, which is the number of pixels surrounded by the contour. This value is normalized with respect to the maximum value among all contours.

(2) The outer boundary of the contour in pixels normalized with respect to the largest boundary found in the picture.

(3) The number of pixels found inside the area enclosed by a contour that feature a black color value. The value is normalized with respect to the maximum number of black pixels found in any of the contours of the picture.

(4) The minimum distance in pixels between the image barycenter and the contour. This value is normalized with respect to half of the diagonal of the image bounding box.

(5) The angle between a horizontal line and the line drawn through the image barycenter and the contour barycenter. The angle is quantized in order to reduce the sensitivity to rotation in the image. Eight possible values are coded as real numbers in $[0, 1]$ with a uniform sampling.

(6 and 7) The maximum curvature angle for convex sides, and the maximum curvature angle for concave sides.

(8 and 9) The number of points for which the curvature angle exceeds a given threshold for convex (1st value) and concave regions (2nd value).

(10) The smallest distance in pixels between the contour and other contours.

(11 and 12) The smallest distance between the contour and the two contours with the largest outer boundary.

These four steps produce a graph presentation for each logo, where each node represents a contour of the logo. Associated witch each node in the graph is a 12-dimensional vector describing properties of the contour such as the area and perimeter of the contour, the distance between the center of mass of the whole image and that of the contour, the maximum curvature for convex and concave vertex, the number of concave and convex vertices, etc. In addition, each tree structure is assigned a symbolic class label indicating the class membership of the original logo. In order to increase the flexibility of this dataset, a unique six-dimensional binary vector is generated for each of the 39 classes and is associated with the corresponding graph. This allows the application to methods requiring numerical class labels.

The result of this operation is a dataset featuring a total of 11700 graphs, 5850 of which are chosen randomly to build the training data set, the remaining 5850 graphs build a set for validation purposes. The graphs in the training set features a total of 55547 sub-trees (nodes), the validation set collects 55654 sub-tree structures. The learning task is to correctly classify the given patterns.

B.3 Conclusions

The dataset provided with this logo recognition problem marks a difficult learning task since the patterns are distorted with up to 50% impulsive noise. The feature and graph extraction mechanism is insensitive to rotation so that the rotation of logos is not expected to have a significant impact on the quality of the data.

Models that are to process this dataset have to demonstrate the ability to encode and generalize over a large number of noisy graph structures.

Appendix C

Literature Review

C.1 Introduction

In this appendix, we will give a brief literature review of relevant literature which influenced the development of this thesis. We have refrained from performing an extensive review of the literature in the body of the thesis because (1) there is not much prior literature in this area, as it is a relatively new area of research, and (2) we do not wish to break up the flow of the presentation of the work by referring to other work, which might not be appropriate. Hence in the body of the thesis, we referred to prior work only when we feel that it will enhance the presentation of the materials.

The structure of this appendix is as follows: in Section C.2, we give a description of the major ideas in processing of data structures prior to 1996. Then in Section C.3 we will review some of the underlying currents of ideas in between 1996 and 1998. In Section C.4, we will review the major breakthrough in this area with the landmark publication of the work by Frasconi, Gori, and Sperduti in 1998. In Section C.5 we will review some of the interesting work built upon the paper by Frasconi, Gori and Sperduti.

C.2 Literature prior to 1996

Probably the first discussion of adaptive processing of data structure problems in the open literature was by Pollack [67]. He considered the problem of how to process a data structure using a multilayer perceptron. As is well known, in data structure representations, the input data might be of varying lengths. In MLP, however, the input data must be of equal lengths. Pollack overcome this problem by padding the input data vector with zeros where the data length is not of the maximum length.

This approach was quite successful in that it was shown that it is capable of processing data structures. However, there are a number of issues related to the application of this idea:

- The resulting MLP architecture appeared to be quite complex. This may be related to the padding of zeros in the input vectors.
- The training of the MLP architecture took quite considerable time. Again this can be related to the fact that the architecture proposed by Pollack

requires substantial number of hidden layer neurons in order to represent the given data set.

At about the same time, Sperduti was studying a special type of MLP architecture, viz., a recursive auto-associative memory [84]. An auto-associate memory is a special type of MLP architecture in that it has equal number of inputs and outputs, and the number of hidden layer neurons is often smaller than the number of inputs or outputs. A recursive auto-associative memory is one which allows a feedback from the output to the input. Sperduti studied this architecture, and attempted to analyse and understand its behaviour. Sperduti extended this model slightly by allowing the inclusion of a label, which he called a labelled recursive auto-associative memory (LRAAM) [81, 85]. This architecture is capable of handling data structures. However, it is not too clear from Sperduti's work what are the limitations of this model [1].

These were the two major ideas that were proposed in that period.

C.3 Literature between 1996 and 1998

A breakthrough in the consideration of adaptive processing of data structures came in 1996 with the publication of a little known paper by Goller and Küchler [2], In this paper, Goller and Küchler proposed a way to unfold a data structure, e.g., a graph, into a cascading architecture. This unfolding of the architecture is very similar to one of the main methods used in the derivation of the training algorithm of a recurrent neural network. In a recurrent neural network, one way to derive a training algorithm is to unfold the recurrent connection so that it becomes a cascaded connection of MLPs. Once the recurrent neural network is unfolded into a cascaded MLP architecture, standard training algorithm derivations can be used to derive a training algorithm for the resulting architecture. This training algorithm is commonly called backprop through time [98, 4], The idea of unfolding the data structure is similar. By unfolding the graph structure, it is possible to use techniques in the derivation of training algorithms for MLPs to this case. This resulted in a training algorithm which Goller and Küchler called backprop through structure (BPTS) [60].

In the meantime, Sperduti continued to work on the problem of adaptive processing of data structures, and derived a framework for supervised learning approach [86]. This approach uses a MLP model for each node of the graph. The learning algorithm is similar to the one proposed by Goller and Küchler.

C.4 Breakthrough in this research area

The breakthrough came in 1998 with the publication of the paper by Frasconi, Gori and Sperduti [29]. In this paper, the authors described a general framework for the adaptive processing of data structures. The major contribution of the paper were as follows:

- Introduction of a common framework for the consideration of data structures.

[1] Sperduti himself analysed the model later, and found some conditions under which the architecture is stable [79].

[2] Even the authors themselves did not realise the power of their method until much later.

- By analogy with time invariance in linear systems, the authors introduced the idea of structural invariance. Using this structural invariance, the authors were able to deduce that the models in each node in the graph must be the same, thus reducing the total number of parameters required to describe the structure drastically. This idea while inherent in the work of Goller and Küchler, and Sperduti and Starita, was first made explicit in this paper.
- The paper also discussed the issue of cyclic graphs and their associated processing problems.

This paper was significant in that it gave a framework with which we can view adaptive processing of data structures. This paper opened up a new sub field: adaptive processing of data structures of which this thesis is part of a continuing push towards further understanding of the properties of these models and the application of such models to practice.

C.5 Literature appeared since 1998

Since the publication of the paper by Frasconi, Gori and Sperduti, there have been some activities in this new sub-field. The major contributions relevant to this thesis are as follows:

- Generalisation of the framework to more general models, in this case, the hidden Markov models [27, 30]. This is an interesting piece of work in that it was shown the basic model in the graph nodes can be considered to be a hidden Markov model.
- The capabilities of a graphical model. Here there were various work, e.g., if the neuron in the model is linear [6], if the neuron is binary [36].
- A relaxation of the constraint on the ordering of the children at any node in the graph [7]. This was achieved by using an appropriate weight sharing mechanism, that guarantees the independence of the network output with respect to the permutations of the arcs.
- A universal approximation theorem for such graph models [45].
- Some preliminary investigation of the Vapnik-Chervonenkis (VC) dimension of the graph models [44]. The VC dimension is related to the learnability of a model. If the VC dimension is infinite, it means that the model is difficult to learn. If the VC dimension is finite it implies that the model is learnable.

It should be noted that as far as we are aware, no one has considered the extension of unsupervised learning to this type of models. In particular, it appears that no one has considered the possibility of extending the self organising map concept to this type of models.

Bibliography

[1] A.V. Aho and J.D. Ullman. *Foundations of Computer Science.* pcss. csp, 1992.

[2] H. Akaike. A new look at the statistical model identification. In *IEEE Trans. Auto. Cont.*, volume 19, pages 716–723, 1974.

[3] E. Appiani, F. Cesarini, A.M. Colla, M. Diligenti, M. Gori, and G. Soda. Automatic document classification and indexing in high-volume applications. In *IJDAR*, volume 4(2), pages 69–83, 2001.

[4] A. D. Back and A. C. Tsoi. FIR and IIR synapses, a new neural network architecture for time series modelling. *Neural Computation*, 3(3):337–350, 1991.

[5] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994. Special Issue on Recurrent Neural Networks.

[6] M. Bianchini, M. Gori, and F. Scarselli. Computation capabilities of linear recursive networks. In *Proceedings of KES 2000*, pages 462–465, Brighton (UK), 2000.

[7] M. Bianchini, M. Gori, and F. Scarselli. Processing directed acyclic graphs with recursive neural networks. In *IEEE Transactions on Neural Networks*, volume 12, no. 6, pages 1464–1470, 2001.

[8] A.M. Bianucci, A. Micheli, A. Sperduti, and A. Starita. Quantitative structure-activity relationships of benzodiazepines by recursive cascade correlation. In *IEEE Processing of IJCNN'98- IEEE World Congress on Computational Intelligence*, pages 117–122, Anchorage, Alaska, May 1998.

[9] A.M. Bianucci, A. Micheli, A. Sperduti, and A. Starita. Application of recursive cascade-correlation networks for structures to chemistry. *Applied Intelligence Journal, Special Issue on "Neural Networks and Structured Knowledge"*, (Vol. 12, Knowledge Extraction and Applications):115–145, 2000.

[10] A.M. Bianucci, A. Micheli, A. Sperduti, and A. Starita. Analysis of the internal representations developed by neural networks for structures applied to quantitative structure-activity relationship studies of benzodiazepines. *J. Chem. Inf. Comput. Sci.*, 41(1):202–218, 2001.

[11] C.M. Bishop, M. Svensen, and C Williams. Developments of the generative topographic mapping. *Neurocomputing*, 21(1):203–224, 1998.

[12] C.M. Bishop, M. Svensn, and C.K.I. Williams. GTM: The generative topographic mapping. In *Neural Computation*, pages 215–234, January 1998.

[13] H. Bunke. String grammars for syntactic pattern recognition. In H. Bunke and A. Sanfeliu, editors, *Syntactic and Structural Pattern Recognition, Theory and Applications*, pages pp29–55. Ed. World Scientific, Singapore, 1990.

[14] F. Burbello, S. B. Pollard, J. Porrill, and J. E. W. Mayhew. Retrieval of high-level data structures from stereo vision data. *Lecture Notes in Computer Science*, 549:312–??, 1991.

[15] H. Carr. Efficient generation of 3-d contour trees. Master's thesis, University of British Columbia, 2000.

[16] F.H. Cheng, W.H. Hsu, and M.Y. Chen. Recognition of handwritten chinese characters by modified hough transform techniques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(4):429–439, 1989.

[17] D. Cherqaoui and J. Villemin. Use of a neural network to determine the boiling point of alkanes. In *Journal of the Chemical Society Faraday Transactions*, pages 97–102, 1994.

[18] P. Colantoni and B. Laget. Color image segmentation using region adjacency graphs. In *Proceedings of the sixth conference on Image Processing and its Applications*, volume 443 of *IEE*, pages 698–702, Dublin, July 1997.

[19] M. Diligenti, M. Gori, M. Maggini, and E. Martinelli. Adaptive graphical pattern recognition for the classification of company logos. In *Pattern Recognition*, volume 34(10), pages 2049–2061, 2001.

[20] B. Ellingsen. Connectionist-based analogical mapping of objectoriented specifications: A representational scheme, 1997.

[21] J.L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[22] E. Erwin, K. Obermayer, and K. Schulten. Self-organizing maps: Ordering, convergence properties and energy functions. *Biological Cybernetics*, 67:47–55, 1992.

[23] S. E. Fahlman. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 524–532, San Mateo, CA, 1990. Morgan Kaufmann Publishers.

[24] S.E. Fahlman. Faster-learning variations on back-propagation: An empirical study. In T.J. Sejnowski G.E. Hinton and D.S. Touretzky, editors, *Connectionist Models Summer School*, San Mateo, CA, 1988. Morgan Kaufmann.

[25] T. Feder. Plex languages. *Info. Sciences 3*, pages 225–241, 1971.

[26] P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In J.M. Jolion, W.G. Kropatsch, and M. Vento, editors, *Workshop on Graph-based Representations in Pattern Recognition*, pages 176–187. IAPR, 2001.

[27] P. Frasconi, F. Costa, and G. Soda. A topological transformation for hidden recursive models. In *European Symposium on Artificial Neural Networks ESANN*, 1999.

[28] P. Frasconi, E. Francesconi, M. Gori, S. Marinai, J.Q. Sheng, G. Soda, and Sperduti A. Logo recognition by recursive neural networks. In R. Kasturi and LNCS K. Tombre, editors, *Second International Workshop on Graphics Recognition, GREC'97*, pages 104–117. Springer-Verlag, 1997.

[29] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing in data structures. *IEEE Transactions on Neural Networks*, Vol 9(5):768–785, 1998.

[30] P. Frasconi, G. Soda, and A. Vullo. Hidden markov models for text categorization in multi-page documents. In *Journal of Intelligent Information Systems, (special issue on Automated Text Categorization*, number 2/3 in 18, pages 195–217, 2002.

[31] J. Gasteiger and J. Zupan. Neural networks in chemistry. *Angewandte Chemie, International Edition in English*, 1993.

[32] C. L. Giles, D. Chen, G. Z. Sun, H. H. Chen, Y. C. Lee, and M. W. Goudreau. Constructive learning of recurrent neural networks: Limitations of recurrent casade correlation and a simple solution. *IEEE Transactions on Neural Networks*, 6(4):829–836, 1995.

[33] C. Goller and A. Küchler. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN'96)*, pages 347–352, 1996.

[34] R.C. Gonzales and M.G. Thomason. *Syntactic Pattern Recognition.* Addison-Wesley, 1978.

[35] A.I. Gonzalez, M. Graña, A. D'Anjou, F.X. Albizuri, and M. Cottrell. A sensitivity analysis of the self organizing map as an adaptive one-pass non-stationary clustering algorithm: the case of color quantization of image sequences. In *Neural Processing Letters*, volume 6, pages 77–89, 1997.

[36] M. Gori, A. Küchler, and G. Soda. On the implementation of frontier-to-root tree automata in recursive neural networks. In *IEEE Transactions on Neural Networks*, volume 10(6), pages 1305–1314, November 1999.

[37] R. Gould. *Graph Theory.* The Benjamin/Cummings Publishing Company Inc., 1988.

[38] M. Hagenbuchner. *The Traffic Policemen Benchmark Software Package.* http://artificial-neural.net, 1999.

[39] M. Hagenbuchner, M. Gori, A.C. Tsoi, H. Bunke, and C. Irniger. Generation of image databases using attributed plex grammars. In J.M. Jolion, W.G. Kropatsch, and M. Vento, editors, *Workshop on Graph-based Representations in Pattern Recognition*, pages 200–209. IAPR, 2001.

[40] M. Hagenbuchner, M. Gori, A.C. Tsoi, H. Bunke, and C. Irniger. Using attributed plex grammars for the generation of image and graph databases. In Mario Vento, editor, *Special issue PRL-Graph-based Representations*, 2002.

[41] M. Hagenbuchner, A. Sperduti, and A.C. Tsoi. A self-organizing map for adaptive processing of structured data. *IEEE Transactions on Neural Networks*, Submitted in November 2001.

[42] M. Hagenbuchner and A.C. Tsoi. The traffic policeman benchmark. In Michel Verleysen, editor, *European Symposium on Artificial Neural Networks*, ISBN 2-9600049-9-X, pages 63–68. D-Facto, April 1999.

[43] B. Hammer. On the approximation capability of recurrent neural networks. In *Proc. of the International ICSC/IFAC Symposium on Neural Computation*, pages 512–518, 1998.

[44] B. Hammer. On the learnability of recursive data. *Mathematics of Control Signals and Systems*, 12:62–79, 1999.

[45] B. Hammer. Generalization ability of folding networks. *IEEE Trans Data and Knowledge Engineering*, 13(2):196–206, 2001.

[46] B. Hammer and V. Sperschneider. Neural networks can approximate mappings on structured objects. In *Second International Conference on Computational Intelligence and Neuroscience*, March 1997.

[47] C. Hansh and A. Leo. *Exploring QSAR*, chapter 3. ACS Professional Reference Book, ACS, Washington, D.C., 1995.

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

[49] S. Haykin. *Neural Networks, A Comprehensive Foundation.* Macmillan College Publishing Company, Inc., 866 Third Avenue, New York, New York 10022, 1994.

[50] T. Heskes. Energy functions for self-organizing maps. In S. Oja, E. & Kaski, editor, *Kohonen Maps*, pages 303–316. Elsevier, Amsterdam, 1999.

[51] J. J. Hopfield and D. W. Tank. Computing with neural circuits: A model. *Science*, 233:625–633, 1986.

[52] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[53] R. Jain, K. Rangachar, and G.S. Brian. *Computer Vision.* mcGraw Hill, Inc., 1995.

[54] T. Kohonen. Learning vector quantization for pattern recognition. Technical Report TKK-F-A601, Helsinki University of Technology, 1986.

[55] T. Kohonen. The 'neural' phonetic typewriter. *Computer*, 21(3), 1988.

[56] T. Kohonen. Improved versions of Learning Vector Quantization. In *Proc. IJCNN-90, International Joint Conference on Neural Networks, San Diego*, volume I, pages 545–550, Piscataway, NJ, 1990. IEEE Service Center.

[57] T. Kohonen. *Self-Organisation and Associative Memory.* Springer, 3rd edition, 1990.

[58] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78:1464–1480, 1990.

[59] T. Kohonen. *Self-Organizing Maps*, volume 30 of *Springer Series in Information Sciences.* Springer, Berlin, Heidelberg, 1995.

[60] A. Küchler and C. Goller. Learning task-dependent distributed structure representation by backpropagation through structure. In *IEEE International Conference on Neural Networks*, pages 347–352, 1996.

[61] Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantization design. In *IEEE Trans. on Communication*, 1980. COM-28-84-95.

[62] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[63] W. Pitts W. S. McCulloch. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[64] M. L. Minsky. *Neural-Analog Networks and the Brain-Model Problem.* PhD thesis, Princeton University, 1954.

[65] G. Nagy and S. Seth. Hierarchical representation of optically scanned documents. *Seventh international conference on pattern recognition*, pages 347–349, 1984.

[66] F. J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232, 1987.

[67] J. B. Pollack. Recursive distributed representations. In *Artificial Intelligence*, number 46 in 1, pages 77–105, 1990.

[68] R.E. Prather. An axiomatic theory of software complexity research. *The Computer Journal*, 27:340–346, 1984.

[69] T. Reenskaug, P. Wold, and O.A. Lehne. *Working with Objects. The OOram Software Engineering Method.* Manning Publications Co., 1996.

[70] M. Riedmiller. Untersuchungen zu konvergenz und generalisierungsverhalten überwachter lernverfahren mit dem snns. *Proceedings of the SNNS*, 1993.

[71] M. Riedmiller and Braun H. A direct adaptive method for faster backpropagation learning: The RProp algorithm. In *In Proceedings of the IEEE International Conference on Neural Networks*, pages 586–591, 1993.

[72] J. Rissanen. A universal prior for integers and estimation by minimum description length. In *Ann. Statist.*, volume 11, pages 416–431, 1983.

[73] D. Rumelhart, G. Hinton, and J. McClelland. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1 of *Computational Models of Cognition and Perception*, pages 318–362. MIT Press, Cambridge, MA, 1986.

[74] K. Saarinen. Color image segmentation by a watershed algorithm and region adjacency graph processing. In *International Conference on Image Processing*, pages 1021–1025, 1994.

[75] S. Schulz, A. Küchler, and C. Goller. Some experiments on the applicability of folding architecture networks to guide theorem proving. In D.D. II, editor, *Proceedings of the 10th FLAIRS Conference, Daytona Beach*, pages 377–381. Florida AI Research Society, 1997.

[76] G. Schwarz. Estimate the dimension of a model. In *The Anals of Statistics*, volume 6, pages 461–464, 1978.

[77] H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. In *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, pages 440–449, New York, N.Y., 1992. ACM.

[78] M. Sonka, V. Hlavac, and R. Boyle. *Image Processing, Analysis and Machine Vision.* Chapman and Hall Computing, 1993.

[79] A. Sperduti. On some stability properties of the LRAAM model. Technical Report TR-93-031, International Computer Science Institute, Berkeley, CA, June 1993.

[80] A. Sperduti. Encoding of Labeled Graphs by Labeling RAAM. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 1125–1132. San Mateo, CA: Morgan Kaufmann, 1994.

[81] A. Sperduti. Labeling RAAM. In *Connection Science*, number 4 in 6, pages 429–459, 1994.

[82] A. Sperduti, D. Majidi, and A. Starita. Extended cascade-correlation for syntactic and structural pattern recognition. In P. Perner, P. Wang, and A. Rosenfeld, editors, *Advances in Structural and Syntactical Pattern Recognition*, volume 1121 of *Lecture notes in Computer Science*, pages 90–99. Springer-Verlag, 1996.

[83] A. Sperduti and A. Starita. An example of neural code: Neural trees implemented by LRAAMs. In N .C. Steele R. F. Albrecht, C. R. Reeves, editor, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 33–39, Innsbruck, Austria, February 1993. Springer.

[84] A. Sperduti and A. Starita. A general learning framework for the RAAM family. In M. Marinaro and R. Tagliaferri, editors, *7th Italian Workshop on Neural Networks (WIRN), Vietri sul Mare*, pages 136–141. World Scientific, 1995.

[85] A. Sperduti and A. Starita. A memory model based on LRAAM for associative access of structures. In *Proceedings of IEEE International Conference on Neural Networks*, volume 1, pages 543–548, June 2-6 1996.

[86] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, Vol. 8(No. 3):714–735, 1997.

[87] S. N. Srihari, E. Cohen, J.J Hull, and L. Kuan. A system to locate and recognize zip codes in handwritten addresses. In *International Journal of Research and Engineering - Postal Applications*, pages 37–56, 1989.

[88] S.N. Srihari, Y.C. Shin, V. Ramanaprasad, and D.S. Lee. Name and address block reader system for tax form processing. In *ICDAR95*, pages 5–10, 1995.

[89] S.N. Srihari, Y.C. Shin, V. Ramanaprasad, and D.S. Lee. A system to read names and addresses on tax forms. *PIEEE*, 84(7):1038–1049, July 1996.

[90] S.B. Thrun and et al. The MONK's problems, a performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, Carnegie Mellon University, December 1991.

[91] A.C. Tsoi. Recurrent neural network architectures: An overview. In C.L. Giles and Marco Gori, editors, *Adaptive Processing of Sequences and Data Structures*, pages 1–26. Springer, Berlin, 1998.

[92] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. Lond. Math. Soc.*, pages 230–265, 1936-7. correction ibid. 43, pp 544-546 (1937). Reprinted with some annotations in The Undecidable, ed. Martin Davis, Raven, New York (1965).

[93] M.M. Van Hulle. Faithful representations and topological maps: From distortion- to information-based self-organization. New York: John Wiley, 2000.

[94] M.M. Van Hulle. *Self-organizing maps: Theory, design, and application.* Tokyo: Kaibundo, 2001.

[95] V.N. Vapnik. *The nature of statistical learning theory.* Springer Verlag, Heidelberg, DE, 1995.

[96] C.H. Wang and S. N. Srihari. Object recognition in structured and random environments: Locating address blocks on mail pieces. In *Proc. of AAAI-86*, pages 1133–1137, Philadelphia, PA, 1986.

[97] Z. Wang, M. Hagenbuchner, A.C. Tsoi, S.Y. Cho, and Z. Chi. Image classification with structured self-organiation map. In *IJCNN*, 2002.

[98] P. Werbos. Backpropagation through time: what it does and how to do it. In *Proceedings in IEEE special issue on neural networks*, volume 2, pages 1550–1560, 1992.

[99] R.J. Williams and J. Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. In *Neural Computation*, volume 2(4), pages 490–501, 1990.

[100] R.J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. In *Neural Computation*, volume 1(2), pages 270–280, 1989.

[101] P.K. Wong and C. Chan. Off-line handwritten chinese character recognition as a compound bayes decision problem. In *Pattern Analysis and Machine Intelligence*, volume 20 No. 9, September 1998.

[102] H. Yin and N.M. Allinson. On the distribution and convergence of feature space in self-organizing maps. *Neural Computation*, 7:1178–1187, 1995.

[103] J. Zupan and M. Novic. Optimization of structure representation for qsar studies. *Anal. Chim. Acta*, 3:243–250, 1999.

[104] Michael J. Zyda. A decomposable algorithm for contour surface display generation. In *ACM Transactions on Graphics*, volume 7, Number 2, April 1988.

Betta Book Binding
M & D Morrisey 4261 2998
26 Fields Street
Kanahooka NSW 2530