Selecting Feature Detectors for Artificial Neural Networks Using Genetic Algorithms

by Andrew D. Brown

A thesis submitted to the faculty of graduate studies in partial fulfilment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering
University of Manitoba

Winnipeg, Canada

December, 1996

© Andrew Denis Brown, 1996

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-16103-X

Canada

Name _____

Dissertation Abstracts International and *Masters Abstracts International* are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

SUBJECT TERM

ELECTRONICS AND ELECTRICAL ENGINEERING

SUBJECT CODE 0544

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Design and Decorative Arts 0389
Fine Arts 0357
Information Science 0723
Journalism 0391
Landscape Architecture 0390
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Educational Psychology 0525
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291
- Rhetoric and Composition 0681

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- Caribbean 0360
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Church 0330
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Fisheries and Aquaculture 0792
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Range Management 0777
- Soil Science 0481
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Animal Physiology 0433
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Plant Physiology 0817
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Dentistry 0567
- Education 0350
- Administration, Health Care 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Safety 0354
- Oncology 0992
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Rehabilitaion and Therapy 0382
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Condensed Matter 0611
- Electricity and Magnetism 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Environmental 0775
- Industrial 0546
- Marine and Ocean 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Cognitive 0633
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

SELECTING FEATURE DETECTORS FOR ARTIFICIAL NEURAL NETWORKS USING GENETIC ALGORITHMS

BY

ANDREW D. BROWN

A Thesis/Practicum submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Andrew D. Brown © 1996

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis/practicum, to the NATIONAL LIBRARY OF CANADA to microfilm this thesis/practicum and to lend or sell copies of the film, and to UNIVERSITY MICROFILMS INC. to publish an abstract of this thesis/practicum..

This reproduction or copy of this thesis has been made available by authority of the copyright owner solely for the purpose of private study and research, and may only be reproduced and copied as permitted by copyright laws or with express written authorization from the copyright owner.

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I also authorize the University of Manitoba to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Andrew D. Brown, 1996

Abstract

This thesis is concerned with evolutionary artificial neural networks (ANNs) which combine genetic algorithms (GAs) with local neural learning using gradient descent methods. It begins by reviewing previous work on the use of GAs for weight and structure optimization of ANNs. Initially, experiments of a previous study on the use of GAs to produce a single feature detector for a two-layer neural network are reproduced. Also, a more thorough comparison of this method with back-propagation (BP) using cross-validation is presented. The comparison shows that on the test problem – a font independent digit classification task – the GA method does not significantly outperform BP with cross-validation. A second method, which uses the GA to find multiple feature detectors which further reduce the dimensionality of the input patterns is then proposed. Rather than replicating feature detectors at all locations in the input field, the GA is required to select their positions; thus, the feature detection layer of the network is much more sparsely connected. In addition three methods of encoding the feature detectors in the chromosome are implemented. One encodes all of the features in a single chromosome, whereas the other two methods split the feature detectors among different chromosomes and combine them to form whole networks. Of these two methods, one selects chromosomes at random from the population and uses an averaging technique to evaluate their fitness; the other splits the population into smaller pools and selects one chromosome from each, allowing the sub-populations to specialize. A simple image classification problem which contains features local to portions of the input field is the basis for evaluation. These experiments show that the GA can correctly determine a set of feature detectors which the network uses to accurately classify the data set. Finally, a set of experiments is presented which test the generalization of these network training algorithms on a noisy, single-font digit recognition task. The experiments show that the three algorithms are comparable at promoting generalization to previously unseen data, but the multiple template version of the genetic algorithm produces the networks which most consistently and accurately classify the training and validation data with high confidence. However, for problems which are much larger than the one presented here, it would be necessary to partition the chromosome, otherwise the search space would grow too large for the GA to reasonably search. This is suboptimal, but the experiments here suggest that there is no appreciable deterioration of performance. Between the two algorithms which partition the chromosomes, there is a noticeable difference in the performance of the average network produced by each algorithm, but the best networks produced by each process are almost the same. This suggests that although the algorithm which subdivides the population was designed to promote cooperation between chromosomes, both algorithms can effectively promote chromosomes which cooperate to form a good network. The algorithm which subdivides the population is at least as good as the one which keeps all chromosomes in a single pool, and sometimes better. Furthermore, it is observed that the genetic algorithm, through its influence on the lower hidden layer feature set, can predispose or bias the network to perform better on validation data than on the data used to train the upper layer of the network. This is attributed to the evolutionary progress of the GA being guided by the network performance on the validation set.

Acknowledgements

I would like to thank Dr. Howard Card for many helpful discussions and suggestions throughout the course of this work. Also, thanks to Dean McNeill for his help with the problems which constantly crop up using the local computer network. Additional, thanks to Dean, Imran Khan and Alex McIlraith for many useful discussions.

I would also like to thank Stacey and my family for all their support over the years. My small successes would not have been possible without them.

Financial support for this work has been provided by the Natural Sciences and Engineering Research Council. The computer equipment used for this thesis was provided by the Canadian Microelectronics Corporation.

Contents

Abstract ii
Acknowledgements iii
Contents iv
List of Figures vi
List of Tables viii
Nomenclature ix

CHAPTER 1 *Introduction 1*

1.1 Motivation 1
1.2 Overview 4

CHAPTER 2 *Background 6*

2.1 A Brief Overview of Artificial Neural Networks 6
2.1.1 Taxonomy of Architectures 7
2.1.2 Learning Algorithms 9
2.1.3 Network Complexity 11
2.1.4 Feature Detectors 13
2.2 Evolutionary Optimization 13
2.2.1 The Genetic Algorithm 14
2.2.2 Schema Theorem 17
2.2.3 Discussion of the Canonical Genetic Algorithm 18
2.2.4 Real Number Genetic Algorithms 20
2.3 Combinations of Genetic Algorithms and Neural Networks 21
2.3.1 Weight Optimization 22
2.3.2 Architectural Optimization 29
2.3.3 Direct Encoding 30
2.3.4 Indirect Encoding 33
2.4 Genetic Algorithms for the Selection of Feature Detectors 38
2.4.1 Feature Selection 39
2.4.2 Other Uses of the GA to Find Feature Detectors 40

CHAPTER 3 *Experiments 41*

3.1 Introduction 41
3.2 Software 41
3.3 Experiments with Single Feature Detectors 45
3.3.1 Description of the Experiment 45
3.3.2 Single Feature Detector – Experimental Results 49
3.3.3 Back-Propagation Comparison – Experimental Results 53
3.3.4 Comments and Conclusions 55

3.4 Experiments with Multiple Feature Detectors 56
3.4.1 Description of the Experiments 56
3.4.2 Experimental Results 59
3.4.3 Comments and Conclusions 61
3.5 Cooperative-Competitive Fitness Evaluation 62
3.5.1 Description of the Experiment 62
3.5.2 Experimental Results 63
3.5.3 Comments and Conclusions 65
3.6 Comparison of Multiple Feature Algorithms 67
3.6.1 Experimental Results – Four Feature Detectors 67
3.6.2 Comments and Conclusions 70
3.7 Generalization of the Algorithms 72

CHAPTER 4 *Conclusions and Recommendations* 81

4.1 Discussion 81
4.2 Recommendations for Future Work 83

APPENDIX A *Back-propagation Experimental Results* 86

References 91

List of Figures

FIGURE 1.1. a) Feature selection and b) a feature detector **3**

FIGURE 2.1. A non-linear perceptron **7**

FIGURE 2.2. Multi-layer perceptron **8**

FIGURE 2.3. Roulette wheel selection. (Goldberg, 1989, p. 11) **15**

FIGURE 2.4. Single point crossover **16**

FIGURE 2.5. Examples of island model and fine-grained model GAs **19**

FIGURE 2.6. Mapping from unbounded-real to bounded-discrete domain **22**

FIGURE 2.7. Flow chart of GA for weight optimization **24**

FIGURE 2.8. Tortoise and hare behaviour (Schaffer et al., 1992) **25**

FIGURE 2.9. Combining the GA with local search (Belew et al., 1991) **26**

FIGURE 2.10. The combinations of genetic and gradient descent optimization: a) Non-Lamarckian and b) Lamarckian **27**

FIGURE 2.11. Network which computes XOR function. (Koza and Rice, 1992) **32**

FIGURE 2.12. Example of a node encoded in Moriarty and Miikkulainen's scheme **33**

FIGURE 2.13. Chromosome structure with area and projection specs. (After Harp and Samad, 1991) **34**

FIGURE 2.14. Network generated by Kitano's method (Kitano, 1990) **35**

FIGURE 2.15. a) The population minimum and average fitness and b) the number of non-linear polynomials in the population (Belew, 1993) **37**

FIGURE 2.16. Structure of the single feature detector network **39**

FIGURE 3.1. Input patterns for the digit recognition problem **46**

FIGURE 3.2. Structure of the chromosome **47**

FIGURE 3.3. Local selection and recombination **48**

FIGURE 3.4. Single Feature, Local Selection Experiment **50**

FIGURE 3.5. Single Feature, Roulette Wheel Selection Experiment **51**

FIGURE 3.6. Selectivity of Local and Roulette Wheel Selection **52**

FIGURE 3.7. Population Variance of Local and Roulette Wheel Selection **53**

FIGURE 3.8. Neural network with multiple local feature detectors **57**

FIGURE 3.9. Local feature input patterns **59**

FIGURE 3.10. Multiple template chromosome experiments for (a) 5 epochs of CG learning, (b) 10 epochs of CG learning, and (c) 15 epochs of CG learning **60**

FIGURE 3.11. Cooperative competitive algorithm with (a) 5 epochs of CG learning, (b) 10 epochs of CG learning, and (c) 15 epochs of CG learning **64**

FIGURE 3.12. Cooperative competitive algorithm using sub-populations with (a) 5 epochs of CG learning, (b) 10 epochs of CG learning, and (c) 15 epochs of CG learning **66**

FIGURE 3.13. Four feature detector experiment – Best and population average for a) most successful run of MTA, b) least successful run of MTA, c) most successful run of CCA, d) least successful run of CCA, e) most successful run of CCAS, and f) least successful run of CCAS **69**

FIGURE 3.14. Eight feature detector experiment – Best and population average for a) most successful run of MTA, b) least successful run of MTA, c) most successful run of CCA, d) least successful run of CCA, e) most successful run of CCAS, and f) least successful run of CCAS **71**

FIGURE 3.15. Example of Geneva character set with noise **72**

FIGURE 3.16. MTA Run #6 – a) generation average performance and b) performance of the best overall network on three data sets **76**

FIGURE 3.17. CCAS Run #9 – a) generation average performance and b) performance of the best overall network on three data sets **77**

FIGURE 3.18. CCA Run #2 – a) generation average performance and b) performance of the best overall network on three data sets **78**

FIGURE 3.19. Average performance of MTA, CCAS, and CCA **79**

List of Tables

TABLE 2.1. A population of chromosomes (Goldberg, 1989, p. 11) **15**
TABLE 3.1. Single feature experiment **52**
TABLE 3.2. Single layer BP experiment **57**
TABLE 3.3. Single layer BP experimental results **57**
TABLE 3.4. Two layer BP experiment parameters **58**
TABLE 3.5. Two layer BP experiment results **58**
TABLE 3.6. Multiple template experiment parameters **63**
TABLE 3.7. Cooperative-competitive experiment parameters **67**
TABLE 3.8. Cooperative-competitive algorithm with sub-populations experiment parameters **69**
TABLE 3.9. Four feature comparison – experiment parameters **71**
TABLE 3.10. Four feature comparison – best chromosome of each run **72**
TABLE 3.11. Eight feature comparison – experiment parameters **74**
TABLE 3.12. Eight feature comparison – best chromosome of each run **75**
TABLE 3.13. Parameters for the generalization experiment **79**
TABLE 3.14. MTA – Performance of the best networks of each run on three data sets **81**
TABLE 3.15. CCAS – Performance of the best networks of each run on three data sets **82**
TABLE 3.16. CCA – Performance of the best networks of each run on three data sets **83**
TABLE A.1. Multiple training runs, using cross-validation, of the single-layer back-propagation network **91**
TABLE A.2. Multiple training runs, using cross-validation, of the two-layer back-propagation network. **94**

Nomenclature

ANN . . . . . artificial neural network

BP . . . . . back-propagation

CG . . . . . conjugate-gradient

EA . . . . . evolutionary algorithms

EP . . . . . evolutionary programming

CNN . . . . . cellular neural network

GA . . . . . genetic algorithm

GP . . . . . genetic programming

MLP . . . . . multi-layer perceptron

MSE . . . . . mean squared error

CHAPTER 1

Introduction

1.1 Motivation

Artificial neural networks (ANNs) are highly parallel computational structures consisting of many simple processing units and a large number of weighted connections between those units. One class of ANN, the multi-layer perceptron (MLP), is capable of performing very general mappings from the input units of the network to the output units. The units in an MLP are arranged in layers and the signals flow through the network from input to output with no recurrences. There are two aspects of the MLP which determine the nature of the mappings which it is able to perform: the architecture and the weight parameters. The architecture of a network refers to the number of layers and the number of units in each layer, as well as, the type of neuron activation function. The size of the architecture determines the number of weights to be specified, and the weight parameters determine the mapping which the network performs, within the constraints of the architecture.

The weight parameters of a network are usually determined in a learning procedure by defining a cost or error function, computing the derivative of the error function with respect to each weight, and minimizing the cost using a gradient descent numerical optimization algorithm, such as the back-propagation or the conjugate-gradient line search algorithms. However, there are several identifiable problems with these algorithms. First, the network structure must be known in advance or be crafted by trial and error. Second, gradient descent optimization techniques are known to be susceptible to becoming trapped in local minima. A gradient descent algorithm simply searches from its starting point to the nearest local minimum. If the starting point for the algorithm is not within the basin of attraction of the global minimum, then it will not find it.

Recent research has focused attention on genetic algorithms (GAs), which are a stochastic optimization method loosely based on the theory of evolution and the process of natural selection, for determining network structure and parameters. They use a performance directed random search, rather than gradient information, to evaluate a population of points in the search

space. Over time the population settles in the regions of the search space which have the best performance. The advantage of these algorithms is that, unlike gradient descent algorithms, they are capable of coarse global optimization without getting stuck in local minima. Also, they do not depend on the analytic properties of the error surface. In fact, the objective function of the genetic algorithm can be discontinuous. For this reason objective functions other than the squared error of the output units can be used, and structural modifications to the ANN, which have a discontinuous relationship to the error function, can also be included.

Research into the weight optimization of ANNs by genetic algorithms has taken one of a variety of approaches. The first attempts used the genetic algorithm alone to optimize the network weights (Whitley and Hanson, 1989; Fogel et al. 1990; Porto et al. 1995). However, the genetic algorithm is not very capable of doing fine-tuning of network weights. Gradient descent algorithms, on the other hand, are good at local search, but they are heavily dependent on the point in the error space at which they start. For these reasons, combinations of the GA with a gradient-descent algorithm provide a better mix of global and local search. One method of accomplishing this is to use the genetic algorithm to optimize the network weight until an individual network is produced which meets some threshold of error, and then complete the optimization with a gradient descent algorithm (Kitano, 1990). In a second approach, each set of weights considered by the genetic algorithm is taken as the starting point for a gradient descent algorithm. Thus, each sample of the error space made by the GA is not a point, but a basin. The gradient descent algorithm searches for the lowest point of the basin, and this is used as the fitness of that sample. (Belew et al. 1991; McInerney and Dhawan, 1993). While this method is the most effective way of combining the GA and gradient descent algorithms, it suffers from the large computational effort required by embedding gradient descent inside the GA.

Besides weight optimization, the GA can also be used to find transformations of the data set which aid the neural network in performing classification tasks. This process is sometimes referred to as feature extraction. One method which is used is feature selection. Feature selection, as it is defined in the statistical pattern recognition literature, is the process of choosing a subset of the input variables, which provide the most information for the network to discriminate among the classes of patterns, while reducing the number of input variables which the network sees. Other than throwing some of the original variables away, it performs no other operations on the input data. One study has shown that the GA can successfully be used for this task (Brill et al., 1992). A related technique, often used in ANNs, is the use of local feature detectors (LeCun et al., 1989). These are implemented in neural networks by low layer hidden units which have weighted connections to small, localized patches of the input field. Both methods attempt to constrain the amount of data which the neural network processes, without reducing its discrimination ability. Note that feature detectors can also refer to units which look at the entire input image. Furthermore, there are other kinds of data set transformations, such as principal component analysis, which can be performed, but these two techniques are the ones to which the GA has been applied in the past.

If we consider these two techniques from a vector space point of view, then feature selection takes the original input vectors, and projects them into a lower dimensional space, spanned by a subset of the primitive variables of the data. By comparison, feature detectors project the

data vectors into a low dimensional subspace which is spanned by a linear combination of several input variables. Another difference between the two techniques is that feature detectors have a non-linearity, because they are implemented as a layer of neural network units. Figure 1.1 shows an example of the methods on a two-dimensional data set. Both attempt to project the data into a lower dimensional subspace which maximizes the discrimination of the two data classes, but feature selection (a) picks one of the axes of the original data, and a feature detector (b) projects the data onto a subspace which may be a linear combination of some of the original variables. The use of feature detectors is the more sophisticated of the two approaches.

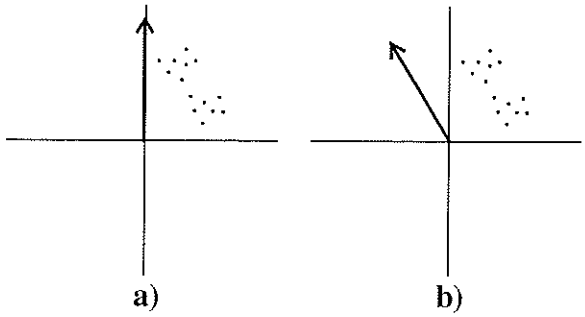


FIGURE 1.1. a) Feature selection and b) a feature detector

Earlier work by Dellaert and Vandewalle (1994) showed that the GA could be used to find a single, space invariant, feature detector for performing an optical character recognition task. This thesis describes work which uses the genetic algorithm to attempt to find *a set* of local feature detectors for a similar optical character recognition problem. There are a number of advantages to the methods proposed here. Whereas most work in combining genetic algorithms and gradient descent methods has trained the entire network by genetic algorithm and refined it by gradient descent, in our method each algorithm only optimizes a portion of the network. This reduces the number of parameters which each algorithm must optimize. In addition, the use of a small number of feature detectors greatly reduces the number of weights in the higher layers of the network, and speeds up the gradient descent algorithm. Finally, the top layer of the network, which performs the final classification step, is tuned by gradient descent, which is superior to the GA at local search. The lower layer, with the feature detectors, contains most of the weights. These weights are coarsely optimized using the GA, but the gradient descent algorithm will attempt to train the upper layer weights to perform the classification with whatever information is provided by the hidden units. Thus, the method still effectively combines the characteristics of global and local search of each of the algorithms.

A further aspect of this problem is the need to encode the feature detectors into a chromosome which can be manipulated by the GA. We look at several methods of doing this. All of them are based on the real number, as opposed to binary encoded, GA. The most direct method of encoding multiple feature detectors into a single chromosome is to assemble the weights of all the feature detectors into a single vector. A population of these are evolved and evaluated by constructing and training a neural network using the feature detectors prescribed in the chromosome. The success of the network at performing the classification task is taken to be the fitness value for the chromosome. A second way of accomplishing this, based on a concept from

Moriarty and Miikkulainen (1994), is to encode a single feature detector into each chromosome, and construct many neural networks by selecting chromosomes from the population at random. The fitness of each feature detector is the average of the classification performance of the networks in which the feature detector participated. A variation on this method is to split the population into small specialized sub-populations. A network is constructed by selecting one feature detector from each sub-population. The fitness is evaluated in the same way, using the average classification performance, but this method imposes more structure on the GA's population, and allows the sub-populations to define what role they will play in network functioning.

1.2 Overview

Chapter 2 gives a brief review of the theory of artificial neural networks, evolutionary algorithms and hybrids of the two. Section 2.1 describes the pertinent aspects of artificial neural networks, particularly multi-layer perceptrons, and some of their training algorithms. Section 2.2 describes the genetic algorithm and modifications to it such as real number representations. Section 2.3 reviews papers in the field of evolutionary artificial neural networks, including work which has been done using genetic algorithms to optimize neural network weights, and also uses of evolutionary algorithms for optimization of neural network architectures. Section 2.4 describes some work specifically related to this thesis, using genetic algorithms to determine feature detectors for neural networks, and the related topic of using them for feature selection for a genetic algorithm.

Chapter 3 describes a series of experiments with an evolving algorithm which uses the genetic algorithm to find a single feature detector, or a set of feature detectors which are usable by a neural network for performing an image classification problem. Section 3.2 documents the software which was written to perform the experiments. In section 3.3 some experiments are described reproducing the method of Dellaert and Vandewalle (1994), using a GA to determine a single first layer feature detector for a neural network. The algorithm is applied to a font independent recognition task. Some additional experiments are performed exploring more fully the selection operator which they incorporate into the GA, as well as a more thorough comparison of the algorithm which they propose with the standard back-propagation algorithm for the classification task. In section 3.4 multiple feature detectors and a simple way of encoding them into the chromosome are incorporated into the algorithm. The new methods are demonstrated on a simple problem. Section 3.5 presents a way of representing multiple feature detectors in the GA by breaking them up among different chromosomes, and a novel variation on this method is also described. The effectiveness of the method is demonstrated on the same classification problem used in section 3.4. Section 3.6 describes a set of more thorough comparison experiments of the three methods described in the two preceding sections. Finally, section 3.7 applies the three methods to a more realistic font classification problem, and assesses their ability to generalize from the training data sets to previously unseen verification data.

Chapter 4 summarizes the work and discusses the results. Some ideas for extensions to the algorithm are mentioned, as well as ideas about further experimental work which could be undertaken.

CHAPTER 2

Background

2.1 A Brief Overview of Artificial Neural Networks

An *artificial neural network* (ANN) is a model of computation which has emerged as a highly useful method of solving problems which were previously difficult to solve using traditional computing paradigms. The advantage of neural networks is that they have been shown to be capable of tasks such as pattern recognition which are performed easily by animal brains, but which are difficult to formulate as a series of sequential steps, as required for traditional computers. ANNs, which are loosely modelled on the operation of animal nervous systems, differ in two primary ways from the traditional digital computer designs. First, ANNs are not programmed to perform a task which has previously been decomposed and solved step by step, but instead are trained by examples to perform tasks for which the solution is not known a priori. Second, they are highly parallel. They distribute computations over many, mostly homogeneous, simple processing units, whereas traditional computers consist of a single or at most a few complex processors. While artificial neural networks have mostly been implemented by simulating them in software on serial digital computers, their inherent parallelism makes them highly attractive for implementation in hardware where the parallelism can be fully exploited.

ANNs are collections of simple processing elements which are connected together in such a way that they are capable of performing sophisticated computational tasks. The basic computational unit of an ANN is the perceptron, shown in Figure 2.1. The input to the perceptron is a set of signals, weighted by the strength of the corresponding connections, which are summed with a bias term:

$$x_j = \sum_i w_{ij} o_i - b_j \qquad \text{(EQ 2.1)}$$

As shown in the figure, the bias term is often treated as an extra weight, but connected to a unit with constant output of -1. The perceptron output

$$y_j = f(x_j) \quad \text{(EQ 2.2)}$$

is usually a non-linear function of the input. Three common forms for this transfer function are the signum, logistic, and the tanh functions. Perceptron transfer functions can also be linear.

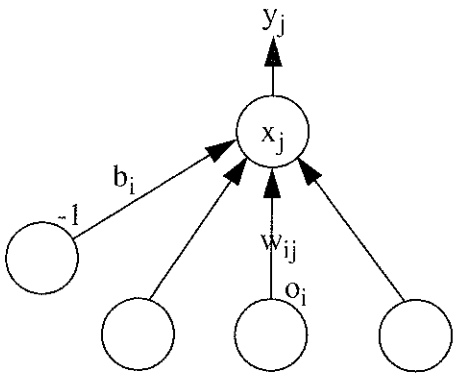


FIGURE 2.1. A non-linear perceptron

2.1.1 Taxonomy of Architectures

Neural networks consist of a large number of such units linked together into a graph. Usually one set of units has its inputs clamped to values determined by an external process. These are termed the *input* units. Similarly, another set of units is designated as the *output* units. These are the units providing the results of the computational task. Finally, the units which are neither input or output units are termed *hidden* units. There are two general categories of connectivity for neural networks: *feed-forward* and *recurrent*.

Feedforward networks, the simpler of the two, are typically arranged in layers. Their connections are uni-directional and they allow for no cycles in the connection graph. Recurrent networks allow both bi-directional connections and cycles. Lacking cyclic connectivity, the current state of a feed-forward neural network is strictly a function of the current inputs, while the current state of a recurrent network is a function of both the current input and previous inputs experienced by the network. This thesis will deal exclusively with feed-forward neural networks. While a single layer of feed-forward perceptrons has serious shortcomings in the range of functions it can compute (Minksy and Papert, 1988), it has been shown that a three layer feed-forward neural network with a sufficient number of sigmoidal hidden units can compute any arbitrary non-linear mapping (Hertz et al., 1991, p. 142). Such multi-layer feed-

forward neural networks are frequently referred to as *multi-layer perceptrons* (MLPs). An example is shown in Figure 2.2.

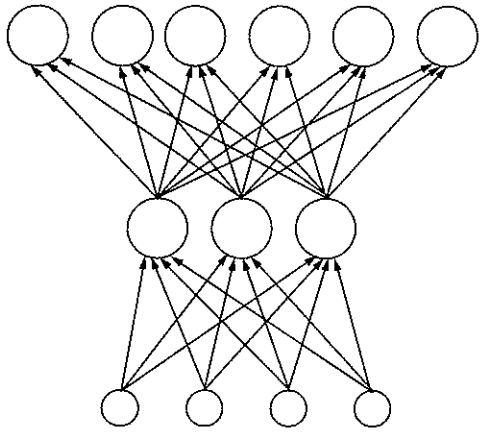


FIGURE 2.2. Multi-layer perceptron

A class of neural networks of interest are *cellular neural networks* (CNNs). Developed by Chua (Chua and Yang, 1988a; Chua and Yang, 1988b; Chua and Roska, 1993; Kozek et al., 1993), these networks emphasize local connectivity. The units are usually arranged in a two dimensional $M \times N$ grid, and each unit is connected only to a neighbourhood, N_r of units within a small radius, r. The neighbourhood of a unit $C(i,j)$ can be described mathematically as the set of units meeting the following constraints,

$$N_r(i,j) = \{C(k,l) \mid \max\{|k-i|, |l-j|\} \le r,\ 1 \le k \le M,\ 1 \le l \le N\} \qquad \text{(EQ 2.3)}$$

The output of a unit, y_{ij} is a non-linear or piecewise linear function of the internal activation, $x_{ij,}$ which can be described by the following equation,

$$x_{ij} = \sum_{k,\,l \in N_r(i,j)} A_{ij;kl} y_{kl} + \sum_{k,\,l \in N_r(i,j)} B_{ij;kl} u_{kl} + b_{ij} \qquad \text{(EQ 2.4)}$$

where y_{kl} is the output from unit kl, u_{kl} is an external input, and b_{ij} is the bias value for the unit. A and B are r x r symmetric matrices representing the weights over the neighbourhood. The weight matrices for these networks are referred to as *templates*. Typically the templates are space invariant, so that the behaviour of an entire network may be specified by its size, its templates and its biases. The neighbourhood of a unit has a symmetry property that, if unit C(k,l) is a neighbour of C(i,j) then the reverse is also true. This implies that CNNs are recurrent because there are cycles in the network graph. The equation defines the operation of a single layer CNN, but it can be extended to multiple layer CNNs, and if the recurrent connections are broken (i.e. $A=0$) then the CNN network becomes a feed-forward network.

The emphasis on local connectivity in CNNs comes from the need to reduce the cost of connections in hardware implementations of neural networks. If in a feed-forward network a layer of size M is fully connected to a layer of size N then the number of connections is *O(MxN)*. If, however, the connections are constrained to a fixed size neighbourhood then the connectivity grows as *O(M)*. Thus, designing neural networks while constraining the number of links used is of great benefit. The concept of local connectivity will come up again in the discussion of localized feature detectors.

2.1.2 Learning Algorithms

One of the critical tasks in an ANN is determining the network's connection weights. This is known as the learning or training algorithm. Learning algorithms can be broadly classified into one of two categories: *supervised* or *unsupervised*. Supervised networks rely on the presentation of a set of target or teaching patterns which specifies the desired network output for a set of corresponding input patterns. The training algorithm is then a method of minimizing the difference between the actual output signals of the network and the desired output signals based on some error measure. The most common error measure used is the squared error. The error for a given input pattern n is given by

$$E_n = \frac{1}{2}\sum_j (d_j - y_j)^2 \qquad \text{(EQ 2.5)}$$

where d_j is the desired value of output unit j and y_j is its actual value. The total error for the network is then found by summing E_n over all the entire pattern set:

$$E = \sum_n E_n \qquad \text{(EQ 2.6)}$$

(Bishop, 1995, p. 196). Thus, neural networks using supervised training algorithms are used for tasks in which the input and output patterns are known but the mapping from the input space to the output space is not. These tasks include *association* and *classification*. The second is technically a subset of the first but the differences are worth noting. Association occurs in associative memory type problems in which we would like the network to respond to a given stimulus with some arbitrary prescribed output pattern. If the network is recurrent and the input patterns are time-based then an associative network can be used for control and time series prediction tasks. In a classification task the network is typically constructed such that there is one output unit for each class in the problem. The desired outcome is for a single output neuron to be on for each input pattern indicating which class that input pattern belongs to. The difference in activation between the 'winning' unit and the other units gives us a measure of the confidence with which the network makes the classification. This thesis will be concerned primarily with classification tasks.

Unsupervised learning algorithms don't make use of a training signal, but instead try to minimize a cost function based on the network's internal representation of the data set. Unsupervised learning algorithms are used for tasks such as pattern *clustering*, in which the desired classification of the patterns in the data set is not known and the network is left to find the structure in the data set.

In supervised learning the minimization of the cost function is often performed through the use of a gradient descent search technique. The most common of these is the back-propagation algorithm (Rumelhart et al., 1986a). This is a local learning rule which, as its name suggests, propagates information about the derivative of the error measure at the output of the network backwards through the layers. The learning rule uses this error information to adjust the weights in a direction opposite to the direction of the gradient. In this way the learning rule seeks the minimum of the error function defined on the network's weights. In terms of gradient based optimization techniques it can be classified as a fixed-step, steepest descent algorithm (van Camp et al., 1992). If the entire pattern set to be learned is processed by the algorithm in a batch then the algorithm moves deterministically through the weight space seeking the nearest minimum. One of the problems with this approach is that there is no guarantee that it will find the global minimum of the error measure. Instead it will seek the minimum which is nearest in the weight space to the point where it starts when the network's weights are initialized. One method of countering this problem is to perform pattern by pattern updating of the weights, rather than batch updating. In this case the updating of the weights depends on the presentation of the training patterns to the network. If the patterns are presented in a random order then the updating of the weights will be *stochastic*. The advantage of this approach is that the learning algorithm may have a chance to escape the local minimum. However, local minima are still a large problem in gradient based learning algorithms (Haykin, 1994, p. 191).

A second gradient descent algorithm, which we make use of in this work, is the conjugate-gradient (CG) algorithm. This algorithm is different from the back-propagation algorithm in several ways. Conjugate-gradient uses line searches, rather than a fixed step size, to find the best position along a line in the weight space. A single line search may require several evaluations of the error function. For this reason, pattern by pattern updating of the weights is not appropriate. The direction of the first line search in the CG algorithm is selected in the same way as in the back-propagation algorithm. The derivative of the error function is evaluated at the starting point and the direction opposite the gradient direction is selected. Once the optimal weight parameters in this direction have been found, the next and all subsequent search directions are chosen to be a linear combination of the current gradient direction and the previous search direction. The goal of choosing the direction vector is to select the direction which provides the greatest prospect of minimizing the error function without undoing the optimization of previous steps. There are different versions of this algorithm, each with its own formula for selecting the next search direction. While the CG algorithm is more complex than back-propagation it generally requires fewer epochs of learning than the back-propagation algorithm. It also has the advantage that it has no adjustable learning parameters. One drawback is that the algorithm is dependent on the accuracy of the line search method (Haykin, 1994, p. 216).

One approach to alleviate the problem of local minima is *simulated annealing* (Kirkpatrick et al., 1983; Haykin, 1994, p. 317). This stochastic optimization technique is based on concepts from statistical mechanics, and makes the analogy between optimization and finding the minimum energy of a physical system. Simulated annealing does not rely at all on gradient information, but purely on a method of making random transitions from one search point in the weight space to another. These random jumps are evaluated using the same cost function as in gradient descent methods, but in this case they are either accepted or rejected probabilistically. The probability of a state being accepted is based on the change, ΔE, in the cost, or energy, of the system from one iteration to the next. If the change in energy is negative ($\Delta E \leq 0$), so that it reduces the error, then it is accepted. If the change is positive ($\Delta E > 0$), such that it actually increases the energy of the system, then the new state is accepted with probability,

$$P(\Delta E) = e^{\left(-\frac{\Delta E}{T}\right)} \qquad \text{(EQ 2.7)}$$

where T is a parameter analogous to the temperature of a physical system. For large values of T the algorithm will accept large increases in the cost function and for values of T approaching 0, only transitions which decrease the cost function will be accepted. If the algorithm starts with a large value of T which decreases slowly over time, the system will converge to a low energy state. If the temperature schedule decreases slowly enough, it can be shown that the algorithm is guaranteed to converge to the global minimum (Geman&Geman 1984; Haykin, 1994, p. 317). To put the algorithm in context, if gradient descent starts at an arbitrary point on the cost surface and moves deterministically downhill, then simulated annealing samples widely across the whole surface and gradually narrows the search to regions which have the lowest cost. The problem with simulated annealing is that to achieve the theoretical convergence above, the temperature schedule will be too slow to be practical (Haykin, 1994, p. 317). The success of the algorithm depends on its ability to sample a large enough portion of the search space to find the region which approaches the global minimum of the error function, and this process may take considerable time.

2.1.3 Network Complexity

Another serious problem in neural network design is selecting the complexity of the architecture. While the multi-layer perceptron is only one of many possible classes of neural network architectures which may be used, even within MLPs there is a great deal of freedom in selecting the number of layers to be used and the number of units in each layer necessary to solve a given problem.

One of the characteristics of a good neural network solution to a problem is it's ability to *generalize* from data which it has been trained on to data it has never seen before. If it is possible in a given task to train the neural network using every possible input pattern then the goal would be to reduce the error to zero. However, in most pattern recognition tasks this is not the case and instead we have only a small sample of the space of all the possible input patterns.

When we train the network in this case the error is referred to as the *apparent error*. The *true error* is the performance of the network on all possible patterns. In most cases the best that we can hope for is an estimate of the true error. In order to get an estimate of the true error the available data set is partitioned into a training and a test set. The network is trained on the training set and then the error measure is evaluated for the test set. The consistency of the error measure from the training set to the test set is an indication of the network's generalization ability. While generalization has a good deal to do with the number and variety of data points in the training set, it also strongly depends a lot on the structure of the network.

In an MLP the number of units in the hidden layer and the weights connected to those units correspond to the number of independent parameters that the network has available to it when it is modelling the training data. As in two dimensional polynomial curve fitting if a neural network has too many free parameters then it is possible to overfit the data points and this will detract from the ability of the network to generalize. The goal in constructing a neural network is to provide it with enough hidden units to reduce the error measure, but not so many hidden units that it 'memorizes' the training data and performs poorly with previously unseen data patterns. This is called finding the right *complexity fit*. A good discussion of this topic can be found in Bishop (Bishop 1996).

Two ways of improving the complexity fit of the network are network growing and network pruning. Cascade correlation (Fahlman and Lebiere 1990; Haykin, 1994, p. 205) is an algorithm for growing networks by adding one new hidden unit at a time and retraining only the weights connected to the new unit until the network achieves good performance and a good level of generalization. Methods which work in the other direction are weight decay (Hinton 1987; Haykin p207, 1994) and weight elimination (Weigend et al., 1991; Haykin, 1994,p. 208). Both of these methods add a term to the network's cost function which penalizes large weights. Thus the minimization method tends to drive weights toward zero unless they are contributing to improving the performance of the network.

Another factor when designing the architecture of a neural network is the number of features in the data patterns which are used to perform a pattern recognition task. The success of a pattern recognition task hinges on the predictive power of the features which comprise the data patterns. If there are not enough good features to determine which class a pattern belongs to then a transformation may need to be made which maximizes the discriminatory ability of the input patterns. One such method is *principal component analysis* (PCA) (Hertz et al., 1991, p. 204). At the other extreme there may be too many features, some with good predictive power and others with poor predictive power. If there are too many poorly predictive features in the input data they may act as noise which obscures the class of the input pattern and hinders the network's ability to classify. The process of removing variables from the data vectors which are not useful is called *feature selection* (Weiss, 1991, p. 72). This thesis deals with the application of image recognition, where the problem of feature selection is very important because the network is dealing with a large amount of low level image data, from which it must make its decisions.

2.1.4 Feature Detectors

In their work on using multi-layer perceptrons to solve the problem of handwritten digit recognition, LeCun, Boser, Denker et al. (1989) stress the importance of using localized feature detectors in visual pattern recognition tasks. This principle is built in to neural network architectures by connecting hidden layer units to a small neighbourhood of units in the receptive field. If one considers the activation rule of an artificial neuron (eq. 2.2) then we see that the activation roughly corresponds to a correlation (with non-linear transfer function) of the hidden units weights with the activation of the inputs to the unit. If there is a strong correlation (or anti-correlation) with the weights then the hidden unit will have a strong positive or negative activation. Low activation indicates that there is little correlation between the inputs and the weights. In this way a hidden unit can indicate the presence or absence of a feature at a particular location in the receptive field.

2.2 Evolutionary Optimization

Evolutionary algorithms are a class of stochastic optimization techniques which are inspired by the mechanism of natural selection found in biological evolution. Because they use random permutation to move through a search space, they are related to simulated annealing. The fundamental difference between simulated annealing and evolutionary algorithms is that the latter work on a population of search points in parallel, whereas simulated annealing evaluates only a single point at a time. Each point constitutes a solution in the search space which can be evaluated by a cost or *fitness* function which returns a scalar value. The method for selecting whether a search point is accepted for the next generation is based upon competition among the members of the population. In each iteration of the algorithm a new population is generated based on the previous one, but the least fit solutions are culled in a probabilistic way to ensure a population with ever increasing overall fitness. In this section I will give a brief introduction to the genetic algorithm and the possibilities for its application in the design and optimization of artificial neural networks.

Artificial evolutionary optimization encompasses a broad spectrum of techniques. Fogel (Fogel, 1995) divides these into two general categories: *genetic algorithms* (GAs) and *evolutionary programming* (EP). The primary differences are in the way that the two different algorithms encode the problem, and the way that they generate new solutions at each iteration of the algorithm. Evolutionary programming usually defines the problem as a vector of real numbers, with a simple interpretation in the problem domain. The method for generating new solutions from the old population of solutions is to apply some small perturbation on the members of the population. Genetic algorithms, on the other hand, encode the problem space in a binary string which mimics the structure of genes in biological organisms. The operators which generate new population members from the old are also based on the processes found in genetic replication. The most important of these is genetic recombination, which is mimicked by the GA's crossover operation. The genetic algorithm must also define a mapping from the abstract representational space of the chromosome to the domain space. The litera-

ture borrows two terms from biology to describe these two spaces. The behavioural domain is commonly referred to as the *phenotypic* space while the coding or informational domain is called the *genotypic* space. (Atmar, 1994) The advantage of using a binary encoding in the genetic algorithm is that the optimization algorithm doesn't require domain knowledge. The genetic operations of mutation and crossover operate independent of the information which they encode. However, if the problem being optimized is inherently real valued, then a real number encoding provides a more direct link between the chromosome and the problem solution which it encodes.

While the work described in this thesis uses a real coded GA, which borrows the idea of using real valued representations from EP, the following section describes the canonical, bit string genetic algorithm and the theory behind it.

2.2.1 The Genetic Algorithm

The genetic algorithm is attributed to Holland (1975). It operates on problems whose solution can be represented as a string of symbols. These are the steps in the algorithm as described in (Goldberg, 1989).

1. Encode the problem as a string using a small alphabet of symbols. These strings are referred to as *chromosomes*. Usually a binary alphabet is used, but the necessity of this will be discussed later. The string contains L *genes*, each of which determines a particular feature of the solution which is being encoded. The position of a gene is its *locus*, and the set of values which a gene may take on are called *alleles*. H is an example of a short, binary-encoded chromosome.

   ex. H = 0110010

2. Generate a random initial population of chromosomes.
3. Evaluate the fitness of each of the chromosomes in the population according to an objective function which represents the problem being optimized. Table 2.1 gives an example of some strings with their associated fitness values. The fitness is a positive scalar value which increases with the suitability of the chromosome. If the function being optimized does not fit these constraints then the value must be converted. For this reason we often speak of separate objective and fitness values. For instance, if the problem at hand is a minimization task then the fitness must be inversely related to the objective.

#	String	Fitness	% of total
1	01101	169	14.4
2	11000	576	49.2
Total		**1170**	**100.0**

TABLE 2.1. A population of chromosomes (Goldberg, 1989, p. 11)

3	01000	64	5.5
4	10011	361	30.9
Total		**1170**	**100.0**

TABLE 2.1. A population of chromosomes (Goldberg, 1989, p. 11)

4. A new population of strings is generated by selecting members of the previous population. One common method of doing this is through 'roulette wheel' selection. This method assigns each chromosome a probability of reproduction proportional to its fitness, relative to the other chromosomes in the population. By analogy to a roulette wheel, each chromosome receives a wedge of the roulette wheel, and chromosomes are selected by 'spinning' the wheel until enough have been selected to fill up the next generation of the population. Figure 2.3 shows the roulette wheel for the above example.

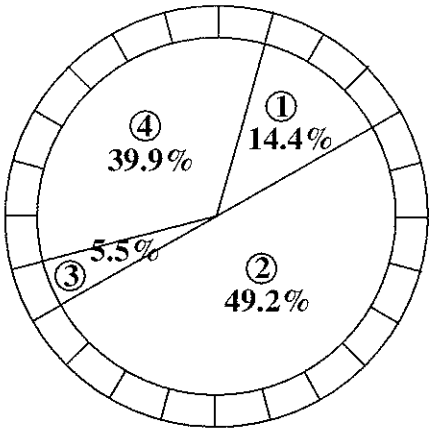


FIGURE 2.3. Roulette wheel selection. (Goldberg, 1989, p. 11)

5. Apply probabilistic operators on the chromosomes which modify them to create entirely new chromosomes. The two most common operators are mutation and crossover. Mutation is applied by iterating through the chromosome and flipping each bit with a probability called p_m. Crossover is based on the process of recombination. It is performed by first selecting a pair of chromosomes from the population. Next, one locus is selected within the chromosome with uniform probability, and with probability, p_c, the substrings defined by that locus crossed over as shown in Figure 2.4. Several variations on the crossover operator also exist, such as *two-point* crossover,

in which two loci are chosen to segment the parent chromosomes, or *uniform* crossover in which the value at each position in the offspring chromosome is randomly selected from one of the parents (Davis, 1991).

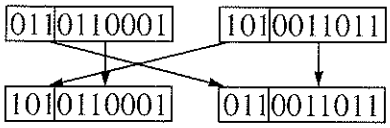


FIGURE 2.4. Single point crossover

6. The algorithm returns to Step 3 and repeats itself until some convergence criteria is met, or until a maximum number of generations has been reached.

There is a problem with the roulette wheel selection method which occurs later in the evolutionary process when the population has begun to converge. Many of the chromosomes in the population will have fitness values which are roughly equal. This means that these individuals will have almost equal probabilities of being selected to reproduce in the next generation and the progress of the algorithm stalls, because there is no selection pressure toward the best individuals. The solution which is usually taken to address this problem is to linearly scale the raw fitness values of the population to accentuate the differences between the individuals, and bias the selection process towards the ones which have the greatest fitness. Linear scaling as described by Goldberg (1989, p. 76) is performed by finding constants a and b such that the equation $ax+b$ maps the average of the population to the same average, and the best fitness in the population onto a value which is twice the average. This is done in such a way that no individual receives a negative fitness value. Thus the best individual in the population, even if its raw fitness is only slightly better than the average, will have twice as high a probability of reproducing as the average member of the population. This technique is used in the experiments which are presented in this paper.

Another way of selecting chromosomes is *rank-based selection* (Whitley, 1989). This deterministic scheme ranks the chromosomes in the population, and linearly assigns a probability of reproduction based on its rank. This way selection pressure is applied to the population based on the ordering of an individual's fitness relative to the other members of the population. It provides even selection pressure irrespective of how large or small the gap in fitness is between individuals. Another important point made by Whitley in this paper is that population diversity is important for effective search by a genetic algorithm. Large populations tend to improve the genetic diversity of the population, but adjusting the selection operator to have less preference for reproducing the most fit individuals can compensate for this. Thus, there is a trade off to be made between selective pressure and population size.

As an optimization algorithm the GA has several advantages over other methods. Since each evaluation of a sample point is independent of evaluations of other members of the population it is easy to implement the GA in parallel. Furthermore, the GA does not require any problem specific information to operate. The problem domain is treated as a black box which evaluates search points and returns the fitness values. Also, the genetic operators of mutation and cross-

over allow the GA to take widely varying samples of the search space. Gradient descent search methods, on the other hand, only sample a single point at a time, make small steps through a local area of the problem space, and require that the problem space have a continuous derivative. This makes the GA less susceptible to problems which contain many local maxima (or minima), and it is useful for optimizing parameters which do not have a continuously differentiable relationship to the performance measure. Conversely, the GA is generally not capable of finding the optimal solution to a problem, only near-optimal solutions, because the genetic operators do not have the ability to perform a thorough search of a local part of the problem domain to find the optimal point.

2.2.2 Schema Theorem

The concept of a schema was introduced by Holland (Holland, 1975) to provide a theoretical justification for the operation of the genetic algorithm. This analysis as described by Goldberg (Goldberg, 1989, p. 28) is based on a genetic algorithm with single-point crossover. The analysis begins by introducing an extended alphabet for the chromosome. If a binary alphabet is used then the extended alphabet is $V = \{0, 1, *\}$, with the addition of * as a 'don't-care' symbol. Thus, over a chromosome which is 8 bits long we may write a schema as follows:

$$H = *11*0***.$$

This schema can be found in any chromosome with a 1, 1, and 0 at positions 2, 3, and 5 respectively, and any other combinations of bits at the other positions in the chromosome. During the processing of the GA such schemata are reproduced, exchanged, created and destroyed. An analysis of the success of a particular schema at remaining in the population can be performed if we first have some definitions of properties. The first is the order, $o(H)$, of a schema. This is simply the number of symbols, not including don't cares, which are contained in the schema. The second property is the defining length of a schema, $\delta(H)$, which is the number of symbols between the first and the last symbol of a schema including the don't cares.

The following formula shows the expected number of instances of a particular schema which are carried from one generation to the next, taking into account only the process of roulette wheel selection:

$$m(H, t+1) = m(H, t)\frac{f(H)}{\hat{f}} \qquad \text{(EQ 2.8)}$$

where $f(H)$ is the average fitness of all the members of the population which contain the schema, and $\hat{f}$ is the average fitness of the population. Thus, if schema H helps the chromosomes which contain it to have better than average fitness, then the constant of growth is greater than 1 and the schema will receive exponentially-increasing samples in the population.

If the opposite is true then the schema, H, will receive a exponentially-decaying number of samples in the population.

Next, we take into consideration the possibility that a chromosome might be destroyed by mutation or crossover. An extra factor is added to the above equation describing the probability of a schema surviving intact from one generation to the next:

$$m(H, t+1) = m(H, t)\frac{f(H)}{\bar{f}}\left[1 - p_c\left(\frac{\delta(H)}{L-1}\right) - p_m o(H)\right], \quad \text{(EQ 2.9)}$$

where L is the overall length of the chromosome, p_c is the probability of crossover, and p_m is the probability of mutation. The second term of the survival factor gives the probability of the crossover operation occurring and the probability that the crossover point will fall within the schema's defining length. The third term gives the probability of a bit in the schema being changed by mutation. From this equation we can derive what is known as the Schema Theorem: that low order schemata, with short defining length, which raise the fitness of their respective chromosomes will receive exponentially increasing trials in the population. Also, Goldberg estimates that the number of schema which are processed far exceeds the number of function evaluations which are performed by the GA. He refers to this effect as *implicit parallelism.*

This gives rise to the concept of building blocks, a collection of such schema which the GA can sample and combine to create high-fitness chromosomes. This is extremely important in selecting an encoding from the problem domain to the chromosome. If a good solution to a problem cannot be built up out of smaller building blocks then it is unlikely that a GA will be able to find it.

Thus we have the important principles of the GA which are relevant to our discussion. It provides parallel, global search which is less susceptible to local maxima problems. It is capable of finding a set of near-optimal solutions. It does not require domain-specific information to do the search other than the ability to evaluate the fitness of a particular individual in the problem domain. Finally, good solutions to a problem must be decomposable into good solutions of a sub-problem, and this must be reflected in the mapping of solutions into chromosomes.

2.2.3 Discussion of the Canonical Genetic Algorithm

Fogel raises a number of issues and doubts concerning Holland's analysis of genetic algorithms (Fogel, 1994). One apparent contradiction with the schema theorem is the effectiveness of alternative crossover operators. While the schema theorem would indicate that single point crossover operates by recombining good building blocks, comparison of different crossover operators indicates that, on some problems, uniform crossover is at least as effective as one-point or two-point crossover despite the fact that it is highly disruptive to the schema contained in the chromosome.

Another problem with the canonical genetic algorithm is that roulette wheel selection does not guarantee the survival of the most fit chromosome from one generation to the next. Thus, there is no guarantee that the algorithm will converge to an optimal solution, or that it will converge at all, because it does not maintain the best solutions that it has discovered during the course of a run (Rudolph, 1994). A simple modification which has been proposed (Grefenstette, 1986), is to retain the best chromosome from one generation to the next. Known as *elitist* selection, this ensures that the genetic algorithm will asymptotically converge to an optimum (Fogel, 1995). Goldberg, on the other hand, argues that this is not a problem at all. He states that elitist selection compromises continuing global searching performance for better local search (Goldberg, 1989, p. 115).

The concept of *spaciality* has also been introduced to genetic algorithms. Spacial GAs view the population as being arranged in a one or two dimensional geometry, as opposed to a single pool. This mimics the effect which geographical isolation has on evolving species in biological populations. The primary purpose of spaciality is to make GAs more suitable for implementation on parallel computers (Maniezzo, 1994; Goldberg, 1989, p. 208; Belew et al., 1991). The genetic algorithm is naturally parallelizeable because evaluations of the fitness function are independent and they can be distributed among multiple processors in a parallel computer. However, the selection operator requires global communication of information about the population members. Spaciality reduces the amount of global information required by restricting competition to local sub-populations. Maniezzo has identified two models of parallel GAs. One is the *island model* which breaks the population up into sub-populations which compete among themselves and periodically exchange their best members with the neighbouring sub-populations. The other model is the *fine-grained model* where the individuals in the population are distributed on a grid or toroid and they compete only within a neighbourhood of a given radius. The island model supports explicit migration among sub-populations of the most fit individuals, while migration is implicit in the fine-grained model. Figure 2.5 illustrates how parallel genetic algorithms might be distributed through a 2-dimensional space.

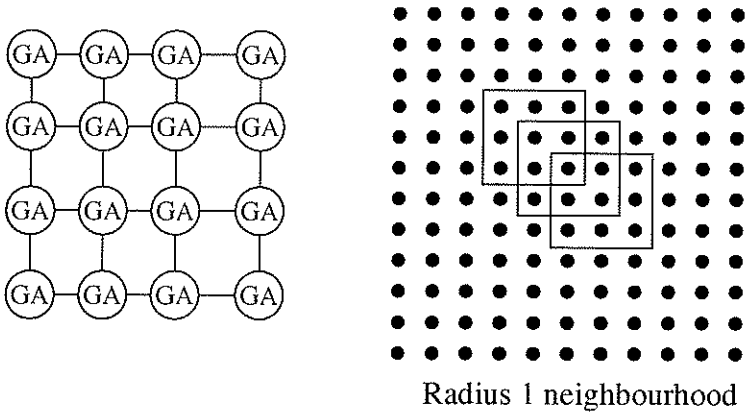


FIGURE 2.5. Examples of island model and fine-grained model GAs

Spaciality may also be effective at promoting population diversity and eliminating premature convergence by shielding weaker individuals from being dominated by extraordinarily fit members of the population (Dellaert and Vandewalle, 1994). This is related to a technique known as *niching*, which attempts to promote diversity by manipulating the fitness function to reward some individual in the population for finding local extrema in the fitness space as well as the global maximum or minimum (Goldberg, 1989, p. 185).

2.2.4 Real Number Genetic Algorithms

Another experimental dilemma that confronts the schema theorem is the success of genetic algorithms using non-binary representations. Holland argued that to maximize the implicit parallelism that he identified operating in the GA, it would be best to increase the number of schemata which exist in each chromosome. He showed that this occurs when the alphabet size is $|V| = 2$. If an optimization problem is naturally represented using a real vector then this must somehow be converted to a bit-string representation. This involves defining the range which a value will take, quantizing it and then representing it as a binary number using regular binary representation or a Gray code representation. However, these encoding problems tend to impose artificial effects of limited range and resolution on to real valued optimization problems.

Some researchers have reported greater success using modified versions of the genetic algorithm which use a vector of real numbers as a chromosome instead of a binary string (Davis, 1991; Michalewicz, 1996; Fogel, 1995). Davis describes a genetic algorithm which replaces the bit-string used in binary encoding with a vector of real valued numbers. While the genetic crossover operator can be used as usual, the mutation operator must be modified to operate in a real number domain. One way of doing this is to randomly modify each position in a chromosome with probability p_m, but instead of a bit flip mutation, the real number is replaced by a random number uniformly distributed over a selected range. In addition to these standard genetic operators Davis suggests a new *Gaussian creep* operator which, with a given probability, adds a Gaussian distributed random vector to a member of the population. The creep operator increases the ability of the genetic algorithm to do local search. Michalewicz (1996, p. 105), in particular, reports that optimization using real valued genetic algorithms is faster, more accurate and more consistent from run to run.

The success of researchers using real coded genetic algorithms appears to fly in the face of the schema theorem. The schema theorem indicates that low cardinality alphabets are superior, but real coding implies an extremely large cardinality alphabet. Goldberg (1991) has attempted to reconcile the two. He hypothesizes that in the early stages of evolution the process of selection serves to narrow the search to the alleles in the initial population which are most fit in a single dimension of the multi-dimensional fitness function. In essence, it takes fitness slices along each dimension and finds the member of the population containing the allele which maximally contributes to the fitness of the individual. Thus, selection reduces the search to a small number of values in each dimension which he refers to as the *virtual alpha-*

bet. The search will proceed using the basic alphabet discovered in the early stages of processing while mutation effects a form of local hill climbing.

One of the problems with the real number encoding in genetic algorithms is that the GA is no longer a 'black box' optimization algorithm. While GA's which use binary encoding are fairly standardized in the way that they use crossover and mutation, a real number genetic algorithm requires some injection of problem domain knowledge to operate. This is incorporated into the GA operators as the range of initial values of the chromosomes, the range of the random numbers used in the mutation operator, and the variance of the Gaussian distributed random vectors used in the creep operator. Unlike the parameters of the canonical genetic algorithm these values are problem dependent and may require investigation to determine good values. On the other hand, Davis (Davis 1991) believes that the use of problem specific knowledge is what contributes to the success of real number genetic algorithms when applied to real world optimization problems. He argues that combining a genetic algorithm with previously known optimization techniques is a good way to improve those techniques to create a hybrid algorithm which outperforms either one in isolation.

In this section we have reviewed some of the important issues regarding genetic algorithms. There remain many open questions within the area of genetic algorithms itself, and it is important to bear this in mind as we try to construct good algorithms which apply these evolutionary techniques to artificial neural networks.

2.3 Combinations of Genetic Algorithms and Neural Networks

After having reviewed some of the basic concepts of both genetic algorithms and ANNs we can now look at some of the areas where they might be combined to produce more effective systems. Schaffer et al. (1992), Balakrishnan and Honavar (1995), and Yao (1993) give good reviews of the work that has been done in this area. Yao in particular provides a framework for viewing the work which has been done in applying genetic algorithms to neural networks. Yao identifies three ways in which GAs have been used in combination with neural networks: searching the weight space of a neural network, optimizing the architecture, and searching for new learning rules of the genetic algorithm.

This section will give a broad overview of the work which has been done in applying genetic algorithms to optimize both the weights, and architectures of neural networks, in order to give the reader a context in which to place the experimental work described in this thesis.

2.3.1 Weight Optimization

There are two ways in which genetic algorithms have been used to optimize the weights of a neural network: as a replacement for the regular gradient descent learning procedures, or in combination with them in some complementary role. The motivation for this second method comes from the differing searching abilities of the genetic algorithm and the gradient based methods which are usually used to optimize the weights of neural networks. While gradient descent optimization is susceptible to trapping by local minima, the multi-point search of the genetic algorithm allows it to search large multi-modal spaces without getting caught in a local minima. In this section we will review examples of both approaches to weight optimization.

The first thing to consider when using genetic algorithms to optimize weights is how to represent the weights in the GA chromosome. Given that we are discussing feed-forward neural networks in which the architecture of the network is fixed, the chromosome which the genetic algorithm operates on is formed by concatenating all the weights of the network together. The problem then becomes one of how best to code each weight into the string. One way is to encode each weight as a binary number (Belew, et al., 1991). To do this the range of values that each weight in the chromosome is allowed to take and the number of bits used to represent each weight are fixed. Then a mapping can be specified from the bounded discrete domain defined by the binary representation to the real domain occupied by the weights of the neural network (Figure 2.6).

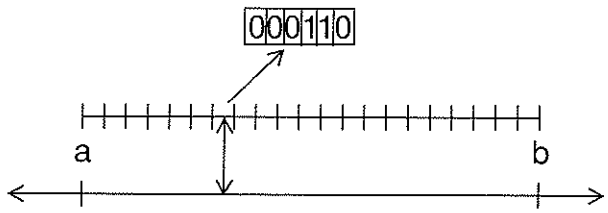


FIGURE 2.6. Mapping from unbounded-real to bounded-discrete domain

In addition the type of binary representation may vary. One possibility is a standard binary representation. However, some feel that this makes it impossible for the genetic mutation operator to do effective local search, because the Hamming distance between neighbouring numbers in a binary representation is frequently greater than one, and the probability of the exact combination of mutations occurring to move from one number to its neighbour is very low. For this reason other researchers advocate the use of a binary Gray code which has the property that adjacent numbers always have a Hamming distance of one.

Fixing the range and precision limits the solution space which the GA is allowed to explore as well as the accuracy of the solution. This creates the dilemma that these parameters must be known in advance and if they are not set correctly then it may not be possible for the genetic algorithm to find a satisfactory solution irrespective of the capabilities of the search technique. Maniezzo (1994) has proposed one solution, which is to make the encoding precision a part of

the chromosome. Thus the precision of the representation evolves along with the network weights.

Another approach, which is not subject to the limits of range and precision, is to use a real number genetic algorithm (McInerney and Dhawan, 1993; Kitano, 1990). This eliminates the problem of how best to map the weights of the neural network into the chromosome, but it also means that the genetic algorithm is no longer problem independent, because the appropriate range of values for the initial population and values for the mutation and creep operators must be found either through experimentation or heuristic rules.

A second representation problem is the order in which the weights are concatenated in the chromosome. If we adhere to the schema theorem, then it makes sense to arrange the chromosome in such a way that the genetic algorithm can take advantage of natural building blocks in the creation of the neural network. The hidden unit of the MLP is the most logical basic functional block, so it seems natural that the weights connected to each hidden unit in the network should be coded as a block on the chromosome. This gives the single point crossover operator the best chance of recombining the hidden units to find the best network for the problem. However, this raises another issue. The function of hidden units in a neural network is a property of its connectivity and not its position in the network diagram. Thus, the hidden units in a network can be permuted and the network will have equivalent functionality, but the chromosome representation of the network may be quite different. This effect is referred to as the *competing representations* problem (Schaffer, et al., 1992). It admits the possibility that two highly fit chromosomes with a good complement of hidden units may be selected from the population, and after crossover is applied a very unfit offspring is produced, because the arrangement of the hidden units in the strings are not the same. Belew et al. (1991) make the case that a good way of combating this problem is to use a small population of chromosomes so that a a single ordering of the hidden units will dominate the population, but this is essentially the same as saying that they are restricting genetic diversity.

Once a way of representing the neural network weights is settled the algorithm proceeds as shown in the flow chart in Figure 2.7. At each iteration of the algorithm the fitnesses of the chromosomes in the population are determined by decoding the network weights and thresholds represented by the values in the chromosome. The fitness of the network is generally computed by presenting the complete pattern set to the network and taking the sum of squared errors. Once again, this assumes that a feed-forward, supervised neural network is being used. If an unsupervised network is being used then a different fitness criteria will have to be used. Also, if the network is being designed for a specific application, then there might be a more appropriate performance measure. This is another advantage of using the genetic algorithm. It is very easy to add new criteria to a fitness function without having to calculate the derivative. For instance, Schaffer et al. (1990) compute the fitness function based on the error over both the training pattern set and an additional test set so that the GA selects for generalization ability in the network.

One of the first studies of the genetic algorithm for optimizing neural network weights did not incorporate any gradient search mechanism at all (Whitley, 1989; Whitley and Hanson, 1989).

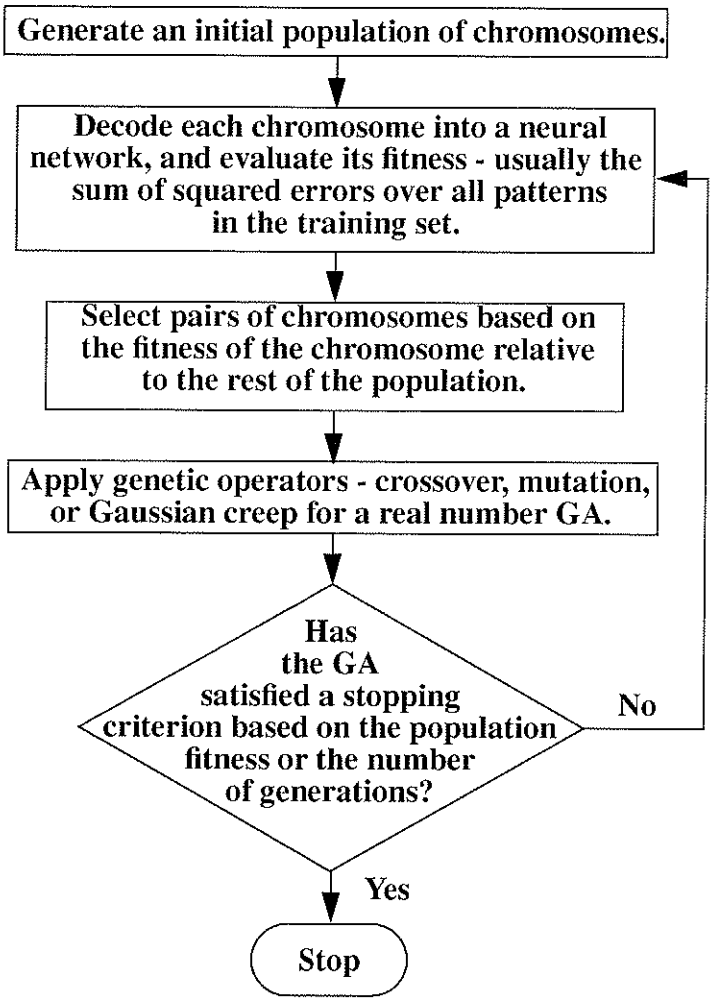


FIGURE 2.7. Flow chart of GA for weight optimization

This study represented the network weights as a binary string and used the squared error over the whole pattern set as the objective function of the algorithm. The problems were the XOR, 4-2-4 encoder and two versions of the 2-bit adder problem. They found that the GA could find a set of weights to solve these problems, and larger populations (200) could achieve lower error rates than smaller populations (50). The two versions of the 2-bit adder problem used two slightly different architectures. One version used four hidden layer units, while the other used only two hidden layer units, but allowed connections which jump from input to output. Back-propagation can find good weights for the first version, but it is known to become stuck in a local minimum when solving the second. The GA was able to find a set of weights which provided the correct output for both versions of the network, supporting the idea that the GA is less susceptible to getting trapped in local minima than gradient descent algorithms.

Evolutionary programming has also been shown to be capable of finding the weights for a neural network (Fogel et al. 1990). Porto et al. (1995) perform a comparison of back-propagation, simulated annealing and evolutionary programming for optimizing neural network

weights. Each algorithm is used in isolation to optimize a three layer neural network for a sonar signature recognition task. For purposes of comparison they used the mean squared error of the neural network on the training set as the performance measure, although it is well known that optimizing error on the training set does not necessarily produce the best classification performance (Bishop 1996). Performing twenty runs of each algorithm, and comparing the classification accuracy of the three algorithms on both a training set and an equal sized test set, showed that all three algorithms were capable of performing good classification. However, the stochastic algorithms provided more consistent results, and the back-propagation algorithm frequently became stuck at a local minima in the weight space. The evolutionary programming version of the algorithm used a population size of fifty. The fact that they were able to undertake a real world problem, with 238 training patterns, attests to the advantage of not using a hybridized evolutionary gradient descent algorithm in terms of speed. The trade-offs of this decision are further explored in the discussion of work done on hybrid methods (Belew et al., 1991; McInerney and Dhawan, 1993).

Other studies have made the point that the genetic algorithm by itself is only capable of finding suboptimal weights for a neural network, due to its poor performance at doing local search (Belew et al., 1991; Kitano, 1990; Schaffer et al., 1992). Thus in a comparison of accuracy with gradient descent algorithms the GA can find good solutions very quickly, but in the long run gradient descent algorithms tend to find the better solutions. Schaffer et al. (1992) refer to this as "tortoise and hare behaviour", and it is illustrated in Figure 2.8

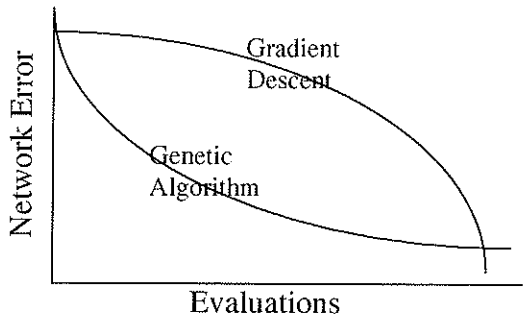


FIGURE 2.8. Tortoise and hare behaviour (Schaffer et al., 1992)

This complementary behaviour of genetic algorithms and gradient descent algorithms has suggested to many that the two should be combined, with the genetic algorithm performing the coarse, global optimization followed by a gradient descent algorithm to do the local search. Figure 2.9 illustrates how in a noisy, multi-modal search space the genetic algorithm

and a gradient descent based local search procedure might collaborate; the GA finding a group of near-optimal points and gradient descent finding the nearest local maxima.

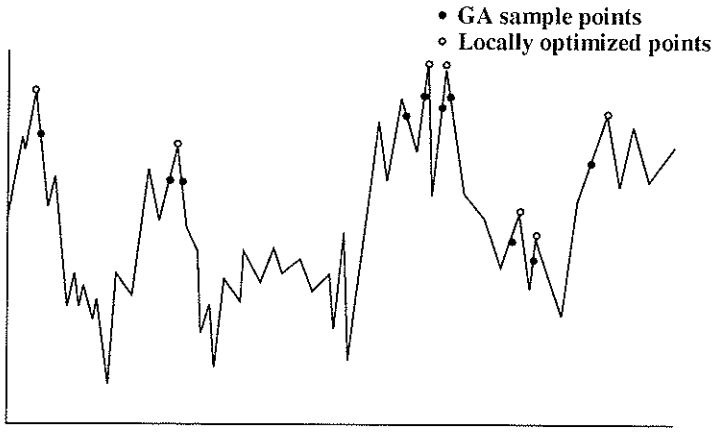


FIGURE 2.9. Combining the GA with local search (Belew et al., 1991)

There are several ways of implementing this combination of optimization algorithms. Kitano (Kitano, 1990) optimizes the weights of the neural network first using the genetic algorithm, until some error threshold is reached, followed by a phase of gradient descent optimization. At each generation of the GA the chromosome is interpreted as a neural network and the squared error of the network on the pattern set is used as the fitness of the chromosome. After a convergence criteria is met, the back-propagation algorithm is used to fine tune the weights discovered in the initial phase of training.

In Kitano's study the raw weights from the chromosome were used to measure the network's fitness, whereas in Belew et al. (1991), and McInerney and Dhawan (1993) some kind of gradient descent learning is performed prior to evaluating the fitness. In the second method the genetic algorithm can be thought of as doing a search, not for a good set of weights, but for a good set of starting weights for the conjugate-gradient algorithm. In both cases the weights encoded in the chromosome (the genotype) are converted into a neural network representation (the phenotype) and the fitness of the chromosome is taken to be the squared error after several epochs of gradient descent optimization. One can go a step further and take the weights of the trained neural network and re-code them into the chromosome (Figure 2.10b). This is the approach used by Belew et al. (1991), and McInerney and Dhawan (1993), and it is referred to as *Lamarckian* evolution, after an early theory of biological evolution which asserted that

learning during the lifetime of an organism can be genetically inherited by its offspring. Figure 2.10 illustrates the difference between the two approaches.

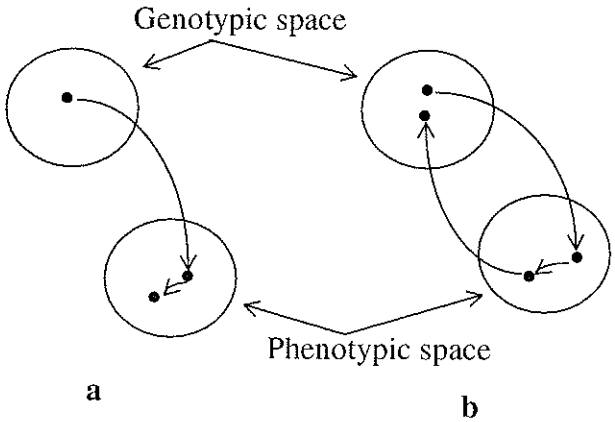


FIGURE 2.10. The combinations of genetic and gradient descent optimization: a) Non-Lamarckian and b) Lamarckian

In Kitano's study (1990) he compared the training speed of a neural network using the accelerated version of back-propagation, Quickprop, and the combined GA-BP algorithm mentioned above. The problems studied were the XOR problem, a number of sizes of the encoder problem ranging from 2-2-2 to 16-4-16, and the two spirals problem (Fahlman and Lebiere, 1990). It is unclear why standard BP was used with the GA and not Quickprop as this might skew the result slightly. His conclusions indicate that the GA-BP hybrid converged faster and more consistently than the GA by itself, but in general the Quickprop algorithm converged faster than the GA-BP algorithm. He does make the concession that the problems attempted, except for the two spirals problem, probably have simple error surfaces which are suited to gradient descent. A problem with a more complex error surface might be more favourable to the GA. In addition he observed that the GA-BP hybrid does not scale well to the larger problems. This probably occurs because of the increase in the size of the chromosome, and the concomitant increase in the dimensionality of the search space.

The other two reports mentioned above (Belew et al., 1991; McInerney and Dhawan, 1993) present a different conclusion. While the idea of taking a computationally intensive iterative algorithm like conjugate-gradient and wrapping it in an outer layer of evolution may sound excessive, Belew et al. make the argument that in real world problems neural networks using gradient descent algorithms are not trained in a single pass. If a gradient descent algorithm doesn't converge then multiple restarts with a different set of random initial weights are required to obtain a satisfactory solution. For this reason, they ran their experiments using a GA/conjugate-gradient hybrid (GA-CG) algorithm versus conjugate-gradient alone using random restarts. They ran these experiments using a number of different binary encodings of the network weights which allowed the chromosome to represent weights of varying ranges. In the GA-CG hybrid they ran the conjugate-gradient algorithm for 30 epochs. The problem under consideration was a six bit version of the symmetry problem and the fitness function

was the mean squared error (MSE) over the pattern set. The results indicate that the GA-CG hybrid consistently converges to a better solution than the CG algorithm with restarts, and the Lamarckian variant mentioned above outperforms the CA-CG algorithm without replacement of the weights. If the number of restarts is equivalent to the number of evaluations used in the evolutionary methods then the computational effort is the same. They also observed the effect that the networks in the GA-CG hybrid tended to converge in fewer iterations of the gradient descent algorithm, since they started from more favourable points in the weight space.

The second important result from this work involves a second set of experiments performed to evaluate the GA for its ability to tune the learning rate (η) and the momentum (α) of a back-propagation algorithm with momentum. They ran the algorithm several times allowing different ranges of representation and found that the GA consistently converged on the largest values. Given a large enough range of representation they found that the GA selected very large values $\eta = 2.5$, $\alpha = 0.33$ for these parameters. This result is natural if we consider that neural networks, running a fixed number of epochs of the BP algorithm and competing to achieve the lowest MSE tend to encourage high learning rates. While this may be appropriate for the small networks considered here, we should remember that the stability of the back-propagation algorithm depends strongly on the learning rate. If the learning rate is too high then the algorithm may not converge smoothly. The conclusion which they draw from this experiment is that networks trained in an evolutionary hybrid algorithm can sustain higher values of the learning rate for more rapid convergence.

The work reported by McInerney et al. (McInerney and Dhawan 1993) describes a similar set of experiments with the primary difference being that they used a real number genetic algorithm. In these experiments they implemented both methods of combining the genetic algorithm with gradient descent. In one experiment they used the same method as Kitano. The GA trains a network to within a threshold of the global minimum of the squared error and then they optimize it with BP. They refer to this algorithm as GA-BPthresh. In another experiment they used Lamarckian evolution which trains each network until it reaches a desired minimum squared error, or until the change in error from one iteration of BP to the next stops descending. The weights resulting from this process are returned to the chromosome. They call this variation of the method GA-BPbasic. They point out that in this method BP trains until it gets stuck in a local minimum and the genetic operators, in effect, help it to jump out of that minimum. Using the 3 bit parity and 4-2-4 encoder problems, they confirm the results reported by Belew et al. that the GA-BPbasic algorithm converges more reliably than BP even with very high values of the learning rate. They also find that the GA-basic algorithm trains faster and more reliably than the GA-BPthresh method. This casts doubt, to an extent, on the conclusions of Kitano, that BP is faster than a GA-BP hybrid, because the hybrid he used was less efficient than the one used in the other two studies.

The major conclusions which we can derive from these studies is that a GA-BP hybrid may be able to outperform BP alone, for two reasons. The GA-BP hybrid can use higher values of the learning rate to achieve faster, more robust convergence in the BP computations. Secondly, to ensure that BP has obtained a good solution, multiple runs with different starting weights may

be required. Thus comparing a GA-BP hybrid against a single run of BP is an unfair comparison in terms of computational requirements.

2.3.2 Architectural Optimization

Artificial neural network construction is a more natural application of the genetic algorithm to neural networks, because unlike the problem of weight optimization there are few systematic techniques for designing neural network architectures and no guarantee of finding the best solution. In addition, the optimal size or structure of a neural network cannot be easily represented as a differentiable cost function which is required for the gradient based techniques used in weight optimization. The problem of finding a successful architecture is related to the issue of complexity fitting discussed earlier. There exist some techniques for finding the optimal number of hidden nodes in a neural network (cascade correlation) or reducing the number of unnecessary weights (weight elimination), but these methods still require a number of assumptions on the part of the designer. The idea that a general optimizing technique such as a genetic algorithm could be used to find the appropriate architecture is very appealing.

If we consider the range of possible neural network architectures as a space and the optimality of the different architectures as a surface in the space, similar to the error surface which is under consideration when optimizing the weights of a neural network, then there are a number of reasons why the GA is superior at searching such a space compared to the heuristic techniques mentioned above. Yao (1993) lists five:

> "The surface is infinitely large since the number of possible nodes and connections is unbounded.
>
> The surface is non-differentiable since changes in the number of nodes or connections are discrete and can have a discontinuous effect on evolutionary artificial neural network (EANN) performance (optimality).
>
> The surface is complex and noisy since the mapping from the EANN architecture to EANN performance after training is indirect, strongly epistatic, and dependent on initial conditions.
>
> The surface is deceptive since EANN's with similar architectures may have dramatically different information processing abilities and performances.
>
> The surface is multi-modal since EANN's with quite different architectures can have very similar capabilities."

The challenge of applying genetic algorithms to the problem of constructing a good architecture is in finding a flexible and extensible way of representing the neural architecture in a string which can be operated on by the genetic algorithm. Koehn (1994) gives a good overview of some of the work which has been done on the problem of encoding. The problem of representation is fairly simple for weight optimization of a fixed size neural network architecture; the weights are simply concatenated into a vector. However, architectures whose size and

connectivity change, require a way of representing the varying number of nodes and connections. This might mean a string representation of variable size. Also, the network representation must be closed under the genetic operators. This means either a representation is used in which every possible string encoding can be interpreted as a valid network, or else the genetic operators must be modified such that only valid networks are produced.

If a representation cannot be created which is closed under the genetic operators, then a method must be found of handling invalid chromosomes. An invalid chromosome cannot just be given a fitness of zero, because then the genetic algorithm has no relative way to assess whether one chromosome is 'less invalid' than another. Goldberg (1989) suggests that constraints be implemented in genetic algorithms by adding a penalty term to the fitness function. This allows the genetic algorithm to differentiate which solutions are better, even if they are all invalid.

In mapping the architectures onto the chromosomes two approaches are generally taken. Either the architecture is directly encoded onto the chromosome as a connection list, or a more indirect method is used which specifies the general attributes of the connectivity as well as a rule for producing the actual connections (Yao, 1993). In this section I will explore several studies which have used both direct and indirect encoding methods.

2.3.3 Direct Encoding

In techniques which use a direct encoding a one to one mapping is defined between the connections and nodes in a network and the alleles in the chromosome or, as in some studies, no chromosome is used at all and the network is it's own representation.

The approach taken in two studies (Bornholdt and Graudenz, 1992; Fogel, 1993) is to forgo the use of a string representation altogether and just apply random modifications to the network structure itself. Fogel uses the evolutionary programming model to optimize the weights of a three layer neural network; i.e. he uses a real number encoding with no recombination and Gaussian creep as the only operator. The structure of the network is modified by adding or deleting a hidden node with probability $p = 0.05$, and all the weights connecting that node are initialized to zero. The method did not involve any neural learning procedure. Bornholdt and Graudenz don't even use a creep operator to modify their networks. The sole operator is a type of mutation which, with a specified probability, removes a number of nodes from the network, and with a different probability adds several new randomly connected nodes, with random connection weights. This mechanism alone, though crude, is enough to generate networks capable of computing some of the standard boolean functions such as OR and XOR. The rules of connectivity are extremely general, allowing for asymmetric, recurrent network designs. Because asymmetric networks have no guarantee of stability, the evaluation function gives each network a limited number of cycles to achieve a stable state, and the fitness function rewards networks which stabilize quickly. In other words, unstable asymmetric networks will not be propagated by the algorithm.

Another approach taken by some researchers can be described as connection based encoding (Koehn, 1994). In these experiments the chromosome encodes the connectivity of a fixed size architecture. Maniezzo uses a binary encoding which reserves space on the chromosome for every possible connection in a specified architecture. One bit is reserved to indicate the existence of a particular connection, and several bits are allocated for the weight of the connection. The number of bits used to represent the connection weights, called the granularity, is included in the chromosome and is optimized at the same time as the weights. Maniezzo varies the genetic algorithm slightly by assigning different mutation probabilities for connection bits, granularity bits, and weight bits. A second study by McDonell and Waagen (1993) also evolved network connectivity using the evolutionary programming approach. The chromosome is a real valued representation of the weights and the mutation operators are Gaussian creep, together with a new operator which adds and deletes connections in the network with a probability calculated from the mean variance of the activation of a node over the data set. A node which has above average activation variance has a high probability of making new connections and a low probability of having connections removed. This method of determining the probabilities is based on the assumption that nodes with high variance in their activations are performing a useful computation. While the algorithm only affects the connectivity, nodes can indirectly be purged from the network by removing all connections to them. The fitness function also penalizes large networks by adding the number of nodes and the number of connections into the network cost function.

Other studies have used schemes which encode both the nodes and the connections of a network. Koza (1992) makes the observation that a chromosome need not be restricted to the one dimensional kind used in the genetic algorithm if another geometry is better suited to the solutions being encoded. In the *genetic programming* (GP) paradigm, developed by Koza, binary tree shaped chromosomes are used to represent computer programs constructed from Lisp expressions. In (Koza and Rice, 1991) the technique is extended so that it can represent a feed-forward neural network architecture including both the nodes and their connections. The chromosome consists of a tree containing several different kinds of primitives: the labelled input variables, numbers representing connection weights, linear threshold units and four mathematical operators for combining the numbers. The terminal nodes of the tree are restricted to being input variables and random numbers. The operators which are used are the same as those used in genetic programming: mutation is the random replacement of a subtree with a new randomly generated tree, and recombination is performed by exchanging subtrees from two chromosomes. He also uses the concept of a *defined block*. This is an operator unique to genetic programming which encapsulates a subtree of a chromosome as a new primitive, thus protecting it from further mutation and allowing it to be used as a building block in other chromosomes. The use of defined blocks helps the method overcome the deficiency of trying to represent an artificial neural network, which is in general a graph structure, by a binary tree. For instance a single hidden unit in a neural network might have connections to several output units. The binary tree does not allow this structure, but it can be imitated through the use of defined blocks by reusing the same sub-tree at multiple points in the tree. In addition each tree has only a single output node. Thus neural networks with more than one output node must be represented by several trees concatenated into a list. Figure 2.11 shows a network which computes the two input XOR function. The P's are the units, W's are the

weights, D# represents an input variable and the addition and multiplication operators perform the expected operations. Koza also demonstrates the use of this method on the one-bit adder

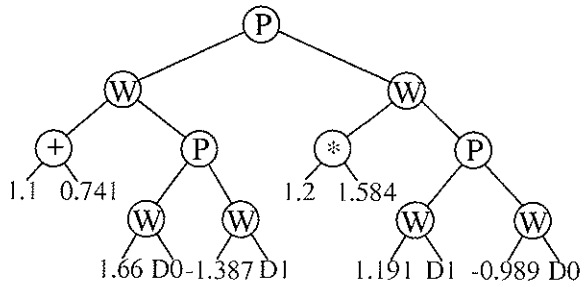
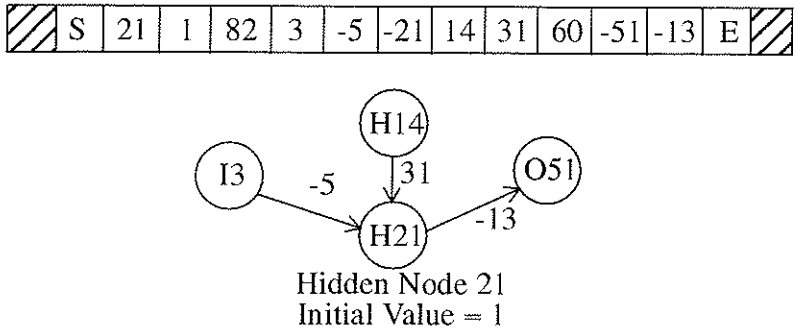


FIGURE 2.11. Network which computes XOR function. (Koza and Rice, 1992)

problem. While the tree based approach works successfully on these small problems, it is doubtful that it would be effective for more complex problems because the tree based chromosome must become unnecessarily complicated to represent even a simple fully connected graph.

A different approach to encoding the nodes and connection of a network is taken by Moriarty and Miikkulainen (1994). Their network encoding scheme is inspired by the presence of markers in DNA. They use an integer string chromosome, with markers delineating the ends of a node description. The chromosome is interpreted by scanning the string for begin and end markers, and the intervening digits are a node definition. Each node definition contains a label naming the node, an initial value for the node activation (this is required because each node is self-connected to create short term memory), and finally a variable number of keys, labels and weights representing connections to other nodes in the network. The chromosome length in this representation determines not the node count, but the number of connections in the network. The number of nodes is variable depending on the number of begin and end marker pairs, which is subject to genetic variation. In this scheme the input and output nodes were not represented in the chromosome, only hidden units. Connections are specified by two numbers, the key and the label. The sign of the key determines whether the connection is to an input/output unit or a hidden unit. If the key indicates an input or output unit, the sign of the label determines which. Figure 2.12 shows the interpretation of one node. The first two numbers after the start marker indicate the label and starting value of the node. The first connection is specified by 82,3,-5. The positive key indicates an input/output unit. The positive label indi-

cates input unit 3, and the weight is -5. They tested their networks on a game playing problem,



<*start*><*label*><*value*><key_0><$label_0$><w_0>...<key_n><$label_n$><w_n><*end*>

FIGURE 2.12. Example of a node encoded in Moriarty and Miikkulainen's scheme

the game of Othello, against an $\alpha - \beta$ game tree program. The networks created were evaluated directly; there was no other learning procedure. The input and output patterns consisted of board positions, and the fitness of the network was determined solely from its game-playing performance. The results show that the marker encoded GA was able to develop superior networks more quickly than a GA which optimized the weights for a fixed architecture network. Thus the mutability of the neural network architecture accelerated the learning process.

2.3.4 Indirect Encoding

Besides being subject to the problem of competing representations, the direct encoding of neural network structures suffers from a lack of scalability (Kitano, 1990). Since there is a one to one mapping between the nodes in the network and the positions on the chromosome the length of the chromosome grows with the size of the network. Longer chromosomes imply higher dimensionality search spaces for the genetic algorithm, and this makes the genetic algorithm less efficient at finding good structures.

To combat this problem, several researchers have attempted to decouple the size of the network representation from the size of the network by using more indirect representations, which define either general characteristics of the network structure or else a set of rules from which the network structure may be constructed. Here we will review a number of studies which have been done along these lines. The first study represents neural networks by specifying general connectivity characteristics and learning parameters of groups of neurons. The other studies all use some form of rule-base, encoded in the chromosome, which dictates how to construct a neural network.

The most straightforward of the indirect encoded models was proposed by Harp and Samad (1991). In this method the neural network is represented as groups of neurons called *areas*,

which are encoded in the chromosome as a set of general parameters such as the number of neurons, a numerical label, the learning rate of the bias weights, and a height and width specifying a two dimensional arrangement of the neurons. They use a variable length binary chromosome which is divided into sections, called area parameter specifications (APSs). The chromosome contains a list of APSs including: an input area, an output area and a variable number of hidden areas. Associated with each APS is a number of projection specification fields (PSFs). A PSF describes another area to which the associated area is connected using either the area's numerical label or a relative referencing method. It also specifies the density of connections to that area, the learning rate of the weights and the two dimensional geometry of the connections. A simplified diagram of a sample chromosome is shown in Figure 2.13. The algorithm uses the standard mutation operator, but the crossover operator is a slightly modified two-point crossover such that the chromosomes are only broken at the ends of area and projection specifications. This allows the crossover operator to handle the existence of variable length chromosomes by exchanging functionally interchangeable blocks. This coding scheme still allows the creation of invalid chromosomes. These are handled in one of two ways: chromosomes with severe problems are culled from the population, and chromosomes with less severe abnormalities are used, but the invalid portions of the chromosome are not expressed. The fitness of a network is evaluated by training it using back-propagation. The actual fitness formula contains a number of weighted terms including learning time, network size, and accuracy. This method was applied to several problems including XOR, a 4x8 digit recognition problem, and a sine function approximation. It was found that the GA consistently constructed networks which could be trained to perform the tasks faster than did networks constructed from a random search.

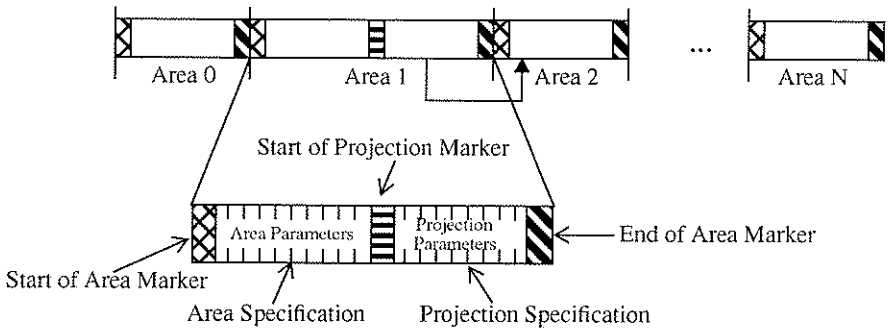


FIGURE 2.13. Chromosome structure with area and projection specs. (After Harp and Samad, 1991)

The other, more popular, approach to indirectly specifying neural network structures is the use of production rule grammars. Rather than specifying the neural architecture in the chromosome, the chromosome contains a number of replacement rules with left hand side (LHS) patterns and right hand side (RHS) patterns, which can be used to develop a network. If the LHS of a rule appears somewhere in the network, the RHS is substituted for it. The LHSs and RHSs consist of a set of symbols. Some symbols may be replaced and others, called terminal symbols, may not. The process begins with a single symbol, or axiom, and the replacement

process operates until either there are only terminal symbols left in the represenation or a specified number of iterations have occurred. What is left at the end should be a direct representation of a valid neural network. If the LHSs of a set rules depend only on the symbols being replaced, then the grammar is said to be context free. If, on the other hand, the rules depend also on the neighbouring symbols of the symbols being replaced then the grammar is called context sensitive.

Kitano (1990) uses a context free rewriting grammar to specify the connectivity matrix of an artificial neural network. Each chromosome contains a set of rewriting rules for generating the connectivity matrix. The set of symbols used is a collection of upper case letters and 16 lowercase letters. Each of the uppercase letters represents a non-terminal and can be replaced by a set of lowercase letters, while each of the lowercase letters is replaced by a fixed pattern of 2x2 bits. The bits are the terminal symbols of the grammar. The construction of a neural network begins with a specific uppercase letter known as the start symbol. The first rule appearing in the chromosome which specifies a rewriting rule for this symbol is invoked. This substitutes the axiom with a 2x2 matrix of more symbols. Likewise, each of these is replaced by a 2x2 matrix of symbols. The dimension of the matrix doubles at each iteration of the algorithm. The last replacement from lowercase letters to bits uses fixed mappings from the sixteen lowercase letters to the sixteen combinations of 2x2 bits. For a feedforward neural network only the upper triangle of the connectivity matrix is needed. Each transformation is encoded by 5 allele positions on the chromosome, the LHS and the four RHS symbols. The first position in the chromosome is fixed to be the starting symbol, S, so that there is guaranteed to be a rule for this symbol. Figure 2.14 shows the development of a network starting with the initial symbol, and ending with a final connectivity matrix. The network is evaluated

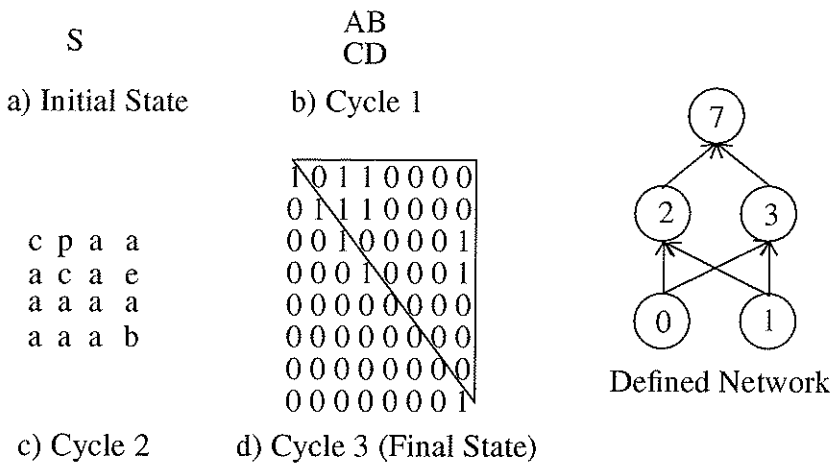


FIGURE 2.14. Network generated by Kitano's method (Kitano, 1990)

by training it on the problem set using back-propagation and random starting weights. In the second incarnation of this method (Kitano, 1992) the algorithm was extended so that the terminating symbols were no longer single bits but several bits which could serve as indices into a randomly generated weight lookup table. In this way the chromosome could specify, in addition to the network architecture, a distribution of the initial weights. Kitano also brings the

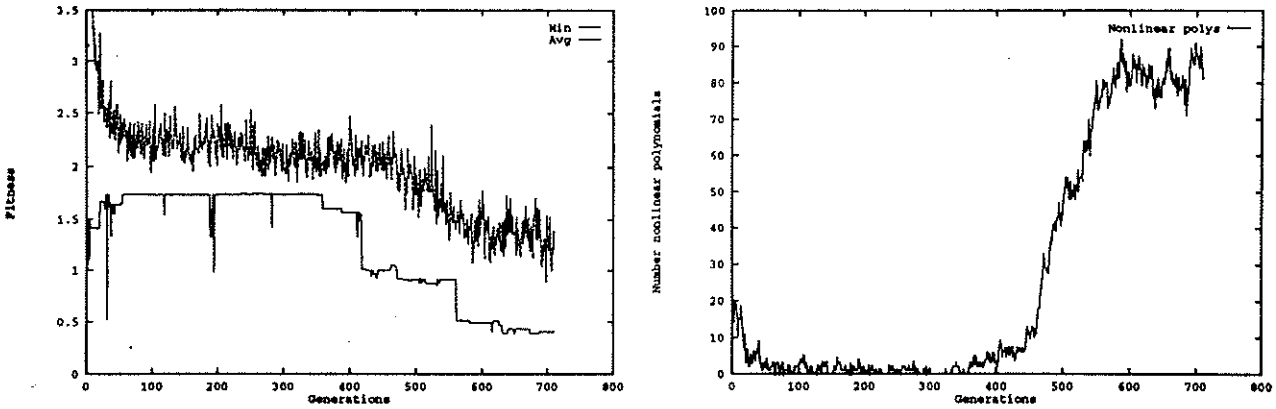
concept of global and local search to the optimization of architectures by suggesting that a network reduction phase which prunes disconnected and redundant nodes from the network would help to fine tune the development process.

Kitano ran experiments on the binary encoding problem of various sizes comparing performance of the grammar encoded neural network described here versus a fixed architecture encoding. The fitness was the total squared error after 10 epochs of back-propagation learning. The results show that the performance of the fixed sized architecture degrades with the increasing size of the problem, while performance of the grammar encoded network remains relatively constant. Kitano makes three points about the success of this encoding method. First, it allows unified substructures to be encapsulated and moved around by the genetic operators. Second, the weights co-evolve with the architecture. Third, new rules may be created which trigger on the same symbol as a previous rule, but the old rule is not destroyed. It still exists in the chromosome and may be useful at a later point in time. This is an effect which also occurs in natural genetics.

Two other studies use context sensitive grammar encodings for neural networks (Belew, 1993; Boers et al., 1993). Belew uses a grammar to encode polynomial networks[1] and takes advantage of the fact that a polynomial neural network can be represented on a two dimensional grid. This provides the context for the rules in the chromosome. Each node in the network can have up to four neighbours on the two dimensional grid and it also has an internal counter which counts down, limiting the number of rules which can be executed for each node. The LHSs of the production rules are patterns of neighbours and a node's internal clock state. The right hand side contains actions which the node can perform, such as spawning a new neighbour node, ticking down its counter or connecting to a neighbouring node. Network development begins with a single node and, at each step, every node in the network searches the list of rules in the chromosome looking for one that applies to its state until its internal clock reaches zero, halting development. The GA uses four specialized mutation operators: randomly change the elements of a production rule, randomly swap two rules in the chromosome (rules which appear earlier in the chromosome take precedence), trim a number of rules from the end of the chromosome, or duplicate a rule and insert it somewhere in the chromosome. Thus, rules compete for precedence in the rule base, and the rules which best contribute to the fitness of the generated neural network help the chromosome to survive. The method was applied to an experiment in time series prediction of data generated by the chaotic logistic equation. Since the data is generated by a second order polynomial a solution to the problem exists. The experimental results show that the GA population is initially dominated by first order approximations, which are overtaken by second order approximations. Figure 2.15a clearly shows the reduction in error after about 500 generations, and Figure 2.15b shows that at exactly the

1. Each layer of a polynomial neural network computes products, rather than sums, of neural activation in the previous layer. Each successive layer produces terms of higher order. Thus a three layer polynomial network produces a third degree polynomial as its output.

same point in the evolutionary process the number of non-linear polynomial networks in the population dramatically increases.



a) b)

FIGURE 2.15. a) The population minimum and average fitness and b) the number of non-linear polynomials in the population (Belew, 1993)

Boers et al. (1993) make use of a more traditional form of context sensitive grammar rules. The rules represent strings of symbols with left and right contexts. The symbols are the set {A-H, 0-5, [,]}. The letters represent nodes and the square brackets serve to encapsulate modules of nodes. The digits, called skips, indicate which modules are connected to which. A node or module followed by a number, *x*, indicates that it is fully connected to the node or module which is *x* skips to the right in the string. Starting from the axiom symbol, rules from the chromosome are invoked until no more rules apply or until 6 iterations have occurred. This prevents infinite loops. At the end, the string of symbols and skips can be converted to the nodes and connections of a neural network. A unique characteristic of this method is that the bit string is converted to rules by reading it twelve times. It is read starting at the beginning of the chromosome and then again starting at each of the first six bits. The same process is repeated reading the string backwards. This further compresses the representation of the neural network by reusing the information in the chromosome several times over. The method uses the regular genetic operators, and the fitness evaluation is the network error after an unspecified number of epochs of back-propagation learning, starting from random initial weights.

Another method proposed by Gruau (1992) is based on Koza's genetic programming paradigm (1991). Each tree shaped chromosome is a program specifying the development of the network. Network development begins with a single node with a 'read head' placed at the top of the program tree. The node reads the instructions at the vertices of the tree, executes them and descends the tree. The programs consist of a few primitive operations. There are two division operators which replace a single node with two new nodes in a parallel or sequential arrangement. (At this point, the program tree branches and each node continues reading the program from a different branch of the tree.) When nodes divide they are automatically fully connected in a feed-forward arrangement. There are also operators for setting values in the

network, such as connection weights and node thresholds. There is a wait operator which temporarily halts the program development. Another operator, which provides this representation scheme with much of its power, is the recursion operator. This operator causes a node to go back to the start of the tree and begin development over again. This allows certain substructures in the network to be repeated several times over. A global variable called the life parameter limits the number of times that recursion can occur. This recursion operation allows the generated neural networks to scale very well, on problems which have a natural repetitive structure to them. Gruau demonstrates this on the n input parity problem. In the early stages of the genetic algorithm, the networks are trained on the two input XOR problem. Once a network is discovered which solves this, increasing the value of the life parameter allows deeper recursion, and the GA quickly finds structures which solve larger versions of the problem. Since the chromosomes are tree shaped, as in genetic programming, the recombination and mutation operators used are also the same. The GA uses a chromosome population with a two dimensional geometry, and individuals in the population can only recombine with chromosomes which are nearby on the grid (Figure 2.5). The benefits of this method are those provided by spaciality in genetic algorithms (see section 2.3.3).

All of these studies, using indirect encoding of neural networks, report good results on the simple problems on which they were evaluated and they should be scalable to larger problems, but there is no experimental evidence yet to support this claim. The scope of the studies has likely been limited by the computational effort required to evaluate a neural network on a problem approaching real world complexity.

The experimental work presented in this thesis has more in common with direct encoding schemes than the indirect ones, although it does provide a mechanism for the GA to perform limited tuning of the network structure. The final section of this chapter introduces the application studied experimentally in this thesis, the use of GAs to evolve feature detectors for neural networks, and reviews some work which is directly related to this task.

2.4 Genetic Algorithms for the Selection of Feature Detectors

The experiments that are described in this thesis are an attempt to apply genetic algorithms to the problem of finding good hidden layer feature detectors for use in effective pattern recognition. The primary motivation for this research comes from the observation that extracting low level local features, which can be combined to form higher order features, is an effective method of performing this task (LeCun et al., 1989). While LeCun et al. employ constrained back-propagation to obtain the feature detectors in their model, work by Dellaert and Vandewalle (Dellaert and Vandewalle, 1994) suggests another approach. In their experiments they employ a genetic algorithm to find a local template for a two layer cellular neural network

type structure. Each hidden unit is connected to a small neighbourhood (5 x 5) of units in the 11x8 input layer, but there are no intra-layer connections as in a general CNN architecture. The hidden layer and the output layer are fully connected as illustrated in Figure 2.16.

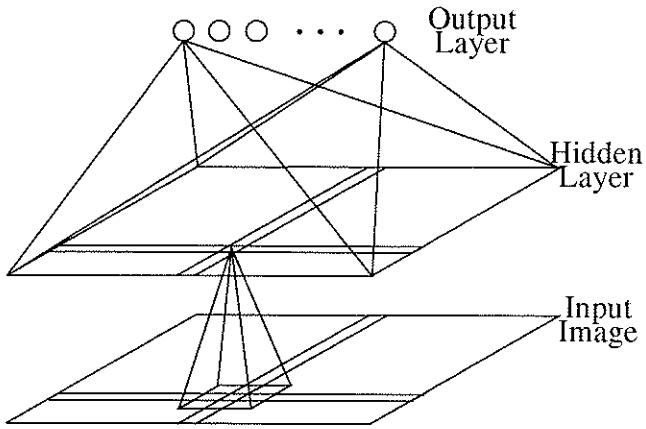


FIGURE 2.16. Structure of the single feature detector network

The problem these authors addressed was font independent recognition of digits. The patterns used were images of the digits from two fonts. A genetic algorithm was employed to find the local template for the hidden layer and BP was used at each iteration of the GA to train the weights of the output layer on the 10 digits of one font. They then evaluated the fitness of a feature detector based on the number of digits which the network could correctly classify in the second font. More of the details of this algorithm will be presented in the next chapter.

As mentioned in section 2.1.3 feature selection is the term used in the statistical literature to refer to the process of determining which inputs to a classification system provide the greatest discriminatory power among the pattern categories, and excluding those inputs which provide less information. The use of local feature detectors in image recognition is related to the process of feature selection, because both techniques are an attempt to reduce the dimensionality of the information which is being presented to the neural network by eliminating those parts of the input data which are not useful for classification. In the next two sections several studies will be described which are directly related to the problems of feature selection and character recognition.

2.4.1 Feature Selection

Two studies have successfully applied genetic algorithms to the task of selecting features from a data set intended for classification by a neural network (Brill et al., 1992; Flotzinger et al., 1994). The approach taken in both studies was to represent all the inputs to a neural network as single bits on a chromosome, and the string of bits in the chromosome then determines which inputs are used and which are ignored. The chromosome is evaluated by attempting to classify the pattern set using just the inputs specified. Because training and testing a neural

network on the pattern set is computationally expensive, in both cases they evaluate the predictive capability of a set of features by implementing a nearest neighbour classification. Brill et al. argue that while the neural network – in this case, a counter propagation network – is faster as an actual classifier, because it requires fewer computations, the nearest neighbour algorithm's error can be evaluated faster without the extensive training period. In addition they verify experimentally that a nearest neighbour classifier gives a reasonable approximation to the classification results of the counter propagation network. They concede, however, that using a different classifier, other than the targeted one, can introduce a bias. The results of these two experiments indicate that a genetic algorithm can be an effective tool for selecting which inputs in a pattern set are the most valuable for classification.

2.4.2 Other Uses of the GA to Find Feature Detectors

In (Koza, 1993) Koza uses the technique of genetic programming (Koza, 1992) to find feature detectors for the problem of character recognition. In this paper Koza uses a chromosome consisting of logical operators and directional motions for an imaginary turtle which traverses the image searching for on and off pixels. If a chromosome/program is found which directs the turtle to traverse the shape of a character, testing for the correct on and off pixels as it moves along then it is able to generate a final output indicating if it is a character, and if so, which character. In the experiments the fitness function was the ability of the chromosome/program to distinguish between two characters, I and L. Koza reported that using a pattern set of 4 x 6 images consisting of the I, L and 76 negative test cases the GA was able to find an individual capable of correctly classifying all positive and negative test cases.

A similar approach was taken in (Tamburino et al., 1995), except that the feature detectors in question were operators based on mathematical morphology. Each chromosome consisted of a variable number of these operators which are applied to an image, and the resulting output is fed into a single layer neural network or a nearest neighbour classifier. The chromosomes of the initial population consisted of a randomly constructed set of morphological operators. At each generation, the chromosomes were recombined using crossover, and mutated by randomly adding and deleting detectors from the chromosome. The test problem was recognizing digitized text, including capital and lower-case alphabetic characters, digits, and punctuation. These authors found that a population of only 16 chromosomes could find a set of detectors in less than 40 generations, capable of recognizing the patterns with 94.3% accuracy.

These two papers exemplify how the GA can be used to find non-neural type feature detectors which can effectively solve the problem of character recognition.

CHAPTER 3

Experiments

3.1 Introduction

This chapter describes experiments which were performed to expand and improve the hybrid genetic algorithm neural network technique described in (Dellaert and Vandewalle, 1994). An experiment is described which repeats the technique of that study to assess the merits of the method that they employed. In further experiments, some of the ideas are expanded to involve multiple feature detectors, and a cooperative-competitive method for evaluating the fitness of networks consisting of multiple templates. A final set of experiments are described which compare the different evaluation methods, and test their ability to generalize on a pattern classification task. The chapter begins with a description of the software which was written to carry out these experiments. The software description is primarily for the benefit of those using or adding to the simulations.

3.2 Software

The software for these experiments was written in Standard C and employed the Xerion 3.0 neural network simulator (vanCamp et al., 1992). Xerion consists of a set of simulators which implements several different models of neural networks. The simulator has a command line interface with a set of commands for constructing and deconstructing the links, units and layers of a neural network, as well as code which implements various gradient descent optimization algorithms for adjusting the network weights. In addition, there are X Window based visualization tools for viewing the values of network weights and activations, and graphing different quantities such as the error function over the course of a training session. The Xerion simulator also uses a run-time type system which scans C header files, and uses this information to implement two commands, `set` and `show`, which allow users to access and modify an

algorithm's data structures from the command line at run-time. Xerion also supports a help system. The user can obtain a list of available commands by typing `help`, and can obtain specific information about a given command by typing `help [command_name]`. All of the new commands added to the simulator for the GA/NN experiments are distinguished by the prefix `ga` at the start of the command name.

The code for the algorithms presented here is implemented as a set of new commands in the Xerion framework. The genetic algorithm code is based on the SGA-C implementation by Smith, Goldberg and Earickson (1994). This is a C implementation of a basic, binary string encoded genetic algorithm. This code was heavily modified to support a real coded genetic algorithm, two-point crossover, the real creep operation, and several different fitness functions. However, the main organizing loop of the GA remains mostly unchanged. A large data structure was added to Xerion, called `gaInfo` (of type `GARec` – defined in the header files), which stores the arrays of chromosomes for the current and previous generation, the best chromosome found throughout the run, and a variety of statistics of the evolving population. Many important parameters for the algorithms are stored in the `gaInfo` structure and these can be accessed using `set` and `show`. Another important data structure implemented in the simulator is the `Individual` structure which stores all the information related to a single chromosome, including the array of real numbers which constitute the chromosome. The old and new populations stored in `gaInfo` are arrays of `Individual` structs.

The SGA-C code was designed to be modular in the sense that new objective functions which interpret the chromosomes and return a real objective value can easily be added. In this case a series of objective functions have been added, which decode a first layer feature detector or set of feature detectors from a chromosome, that is then used to construct a two-layer neural network for the classification task. The objective function trains the neural network for a fixed number of epochs of a gradient descent optimization algorithm, employing either regular back-propagation or conjugate-gradient learning rules, and returns a measurement of the classification performance. Since there are several different models being tested, each with slightly different chromosome structure, all the routines which deal with the chromosomes are implemented as function pointers in the `gaInfo` structure. Thus, the main loop of the genetic algorithm can operate without specific knowledge of which model is being run. The function pointers are set at the command line by two commands: `gaSetObjective` and `gaSetSelection`. The former sets up the version of the objective function which is being used. This dictates the size and make-up of the chromosome. The three objective function options are multiple-template, cooperative-competitive and the cooperative-competitive with subpopulations objective functions. The `gaSetSelection` command determines which selection algorithm is being used: roulette wheel with scaling, the local selection algorithm, or the separate sub-population selection methods introduced for the last experiment. The selection method actually consists of a set of four function pointers internally. Each selection method implements it's own version of these four functions. There are two functions, `selectAllocMem` and `selectFreeMem`, for allocating and de-allocating memory needed by the selection function for bookkeeping purposes. The `preSelect` function converts objective values to fitness values prior to the selection process. In some cases this does nothing, but for the roulette wheel algorithm this function performs the linear scaling so that the fitness value

of a chromosome is actually different from the objective value returned by the objective function. Finally the function `select` is called by the main loop of the GA to select a single chromosome out of the population for reproduction. While the `gaSetObjective` and `gaSetSelection` commands are provided for setting the objective function and selection operator, other parameters of the GA are set by the user manipulating the `gaInfo` structure directly. This is done via Xerion's built in `set` command.

Since the type of objective function being used determines the structure of the chromosome, the `gaSetObjective` must be invoked before other commands to construct and initialize the population of chromosomes. `gaAddPop` takes as a parameter the number of chromosomes in the population as well as the width and height of templates being used. This information, in conjunction with the previously set objective function, is used to allocate memory for the chromosome structures and appropriate length arrays for the chromosome itself. Another command `gaInitPop` accepts as a parameter a mean and radius for randomly initializing the population. A third command which is integral to the GA functionality is `gaRun` which commences the main loop of the genetic algorithm. These three commands all change their functionality based on the setting of the objective function pointer.

As mentioned there are three objective functions in the code which determine the fitness value of a particular chromosome, one for each of the three algorithms implemented. These three routines all accept a pointer to an array of `Individual` structs containing the population. The routine accesses the current network pointer, and it destroys and reconnects all the links in the first layer of the network. It trains the network on the training example set, for the number of iterations specified in the `gaInfo` structure, using another routine called `gaMinimize`. Another function called `performance` counts the number of correct classifications of the network on the validation example set. The objective function repeats this process a fixed number of times. For the routine called `gaMultiTemplateObj` it constructs a number of networks equal to the size of the population. The other two objective functions, `gaCoCompObj` and `gaSubPopCoCompObj`, construct the number of networks specified by the `gaInfo.numSaneTrials` parameter. The performance of the current neural network can also be measured by invoking the `gaPerformance` command from the command line. This makes it easy for the user to interactively manipulate and evaluate the neural network.

The three genetic operators: creep, mutation and crossover, also change according to the objective function which is being used. For instance, if the multiple-template or cooperative-competitive algorithms are being used, then the chromosome positions which specify the template location are treated differently by the mutation and creep operators than the positions which specify weights – they have a much larger allowed range. For this reason these operators are also implemented as function pointers in the `gaInfo` structure. There is only a single instance of the crossover function, called `gaCrossover`, which implements two-point crossover. However, there are separate creep and mutation functions for each of the single template, multiple-template and cooperative-competitive algorithms.

Before the evolution begins, the data structures are allocated for a two-layer network which is altered by the fitness function of the GA in order to evaluate the chromosomes. Before the `gaRun` command is invoked, a routine called `gaBuildNet` automates the construction of a two-layer network. The hidden and output layers of the network are fully connected, but the input and hidden layers are left unconnected, as is the bias unit and the hidden layer. These connections are created by the various objective functions genetic algorithm code when they evaluate the chromosomes. The number of hidden layer units are specified by the `numTemplates` field of the `gaInfo` structure, the number of output layer units are specified by the `numOutputs` field, and the number of input units are specified by the `imageHeight` and `imageWidth` fields. These fields are set by the user of the software, but they must be equal to the size of the input patterns which are being used, or an error will result.

While cross-validation is a common neural network technique, it is not built into Xerion, so an additional command called `gaCrossValidateExp` was added to the simulator. This command trains the neural network on the training example set, but it periodically stops to check how well the network classifies the validation example set. This command allows the user to specify the maximum number of training iterations, and how frequently to stop training to check the classification performance. The performance measure used is not the squared error of the output units, but the same percentage classification measure used in all the genetic algorithm experiments. It records the best classification performance on the training set, the validation set, and the average performance on both of those sets.

Support was also added to the network for a confusion matrix. A confusion matrix contains a number of rows and columns equal to the number of classes of patterns in the data set. Every time a pattern is impressed on a neural network the matrix element in the row representing the correct class and the column representing the class actually indicated by the network is incremented. Thus, by viewing the confusion matrix one can easily see how many classifications the network is making correctly (indicated by the diagonal elements), how many are incorrect (indicated by the off-diagonal elements), and which classes the neural network is confusing. The command `gaAddConfusion` allocates memory for the confusion matrix, using the value in `gaInfo.numOutputUnits` to indicate the number of classes in the data set which the network is being trained for. The command `gaClearConfusion` resets all the elements to zero, and the command `gaPrintConfusion`, displays the confusion matrix, as well as the percentage correct broken down by class. The `performance` contains the code for updating the confusion matrix. Thus, the confusion matrix is incremented whenever the neural network is evaluated on an example set, either inside the `gaPerformance` command, or in the internal loop of the genetic algorithm.

3.3 Experiments with Single Feature Detectors

3.3.1 Description of the Experiment

The purpose of the first set of experiments was to reproduce the results demonstrated by Dellaert and Vandewalle in their paper describing the use of the GA to find local feature detectors for pattern recognition neural networks (1994). As mentioned in the previous chapter, the basic concept of their application of GAs to the training of neural networks was to evolve a 5x5 local feature detector which would make up the hidden layer of a two-layer MLP for classifying 11x8 bitmaps of the digits from two different fonts.

The hidden layer of the neural network contains the same number of units as the number of pixels in the input images, as depicted in Figure 2.16. The single feature detector is connected at every possible position in the receptive field. Each unit in the first layer calculates the correlation of the feature with the input image at that position, and then passes the result through a non-linear transfer function (eq. 2.1), (eq. 2.2). In this case the non-linear function is the logistic function:

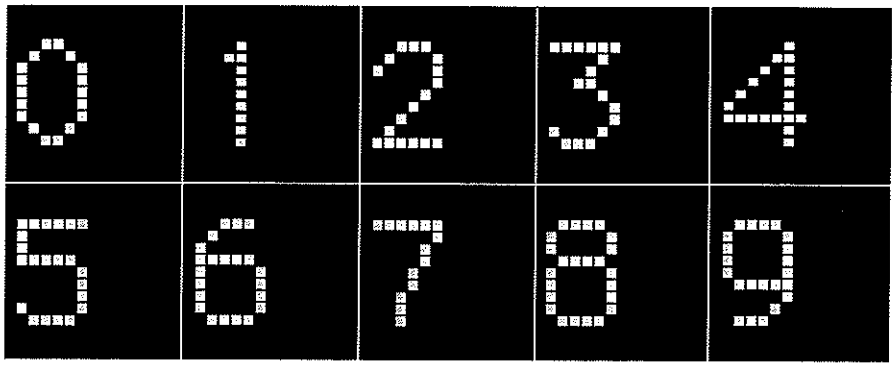
$$y = f(x) = (1 + e^{-x})^{-1} \quad \text{(EQ 3.1)}$$

The output layer of the network contains ten units, each of which is fully connected to the first layer. The pattern sets which are used for training and testing the network consist of bitmap images of the digits 0 through 9 from two different typefaces. The patterns in the training set are the digits of the 12 pt. Geneva font (Figure 3.1a) and the digits in the validation pattern set are taken from the Courier font (Figure 3.1b). Thus, there is one output unit for each class of input pattern. The desired behaviour of the classification network is that the presentation of a digit to the network will cause a strong activation of the corresponding output unit and very weak activation of all other output units. The degree to which a network achieves this desired behaviour is measured by the GA's fitness function.

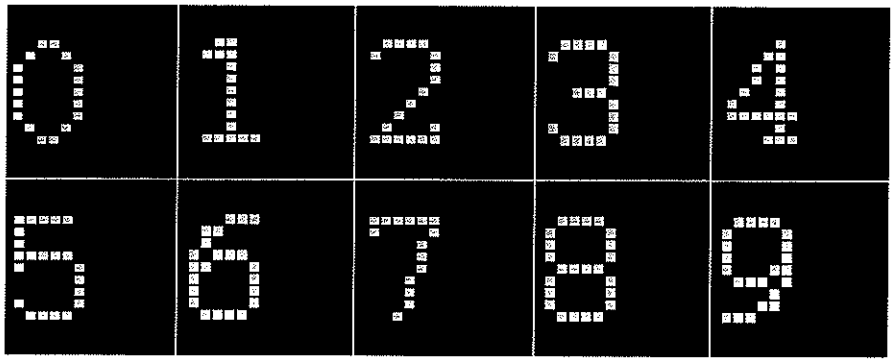
The first layer of weights is generated by a real number genetic algorithm. At the evaluation stage of the GA each chromosome in the population is decoded as a 5x5 feature detector. Every unit in the first layer of the network is connected to a 5x5 patch in the input image and the weights are fixed using the values in a given chromosome. During the second part of the evaluation the weights in the second layer of the neural network are randomized and then trained using the back-propagation algorithm. Assuming a logistic transfer function, for a given pattern, the learning rule for an output unit, i, can be written as follows (refer to Figure 2.1):

$$\delta_i = \alpha \cdot y_i \cdot (1 - y_i) \cdot (d_i - y_i) \quad \text{(EQ 3.2)}$$

$$\Delta w_{ij} = -\delta_i \cdot o_j \quad \text{(EQ 3.3)}$$



(a)



(b)

FIGURE 3.1. Input patterns for the digit recognition problem

$$\Delta b_i = -\delta_i \qquad \text{(EQ 3.4)}$$

where y is the actual output of the unit, d is the desired output of the unit, α is the learning rate, w_{ij} is the weight connecting the output, o_j, of hidden unit j to the output unit, and b_i is the bias term of the output unit. δ_i is an expression which is used for notational convenience. The term $y_i \cdot (1 - y_i)$ is the first derivative of the logistic function and $(d_i - y_i)$ is the error term. This learning rule results from the minimization procedure of the sum of squared error function defined in (eq. 2.5) and (eq. 2.6).

The network is trained for three entire passes of the training pattern set (30 presentations in all). After each presentation of a pattern the output layer weights are adjusted according to the learning rule. Since the GA fitness function depends only on the classification rate of the network, the actual value of the squared error is ignored for the purpose of determining the network's fitness. In the final stage of evaluation, a network is evaluated by testing its classification performance on the second set of digit patterns. The fitness formula is the classification percentage, plus a second term called the *selectivity* which measures the degree of confidence of the network's prediction:

$$P = \sum_{n=0}^{9} p_n \begin{cases} p_n = 10 + s_n, \max(y) = y_n \\ p_n = 0, \max(y) \neq y_n \end{cases} \quad \text{(EQ 3.5)}$$

$$s_n = y_n - \frac{1}{9}\sum_{i \neq n} y_i \quad \text{(EQ 3.6)}$$

The first of these equations iterates over the ten patterns in the validation set and computes the number correct scaled by 10. For each pattern which is correctly classified, a second term s_n is added, which measures how much greater the activation of the winning output unit was than the average of the losing output units. Multiple units which are strongly activated indicate that there is doubt in the network's classification. The selectivity formula measures this. The final result of these computations is used as the fitness of the neural network in classifying the validation set.

Figure 3.2 shows the structure of the chromosome. Each position on the chromosome con-

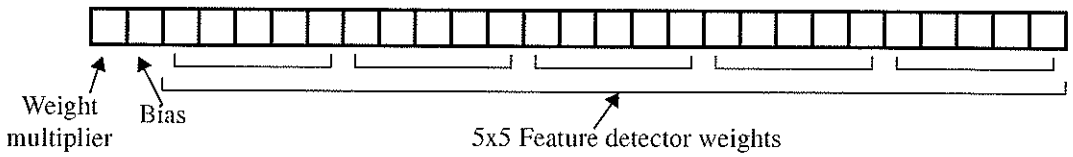


FIGURE 3.2. Structure of the chromosome

tains a real number, representing either a weight in the feature detector or the unit's bias value. The weight multiplier is a value which scales all of the weights and the bias. By including the weight multiplier in the chromosome, the GA has a way of very coarsely finding a range for the weights in the feature detector. The weights of the second layer of the network are in no way encoded in the chromosome. This differs from many of the studies discussed previously.

Dellaert and Vandewalle (1994) make use of a real number genetic algorithm. They use a two point crossover operator, and forgo the use of a real mutation operator, as did McInerney and Dhawan (McInerney and Dhawan 1993), in favour of a gaussian distributed creep operator. The other change which they make is to use a selection scheme which works on the basis of local competition, making use of the spacial GA concepts introduced in the last chapter. This selection method works by viewing the chromosomes as being arranged in a circular array. The array is segmented into small groups and the best chromosome from each group is selected to reproduce. Each group is replaced by recombining the winning chromosome with

that from an adjacent group with probability p_c. As depicted in figure Figure 3.3 the winning

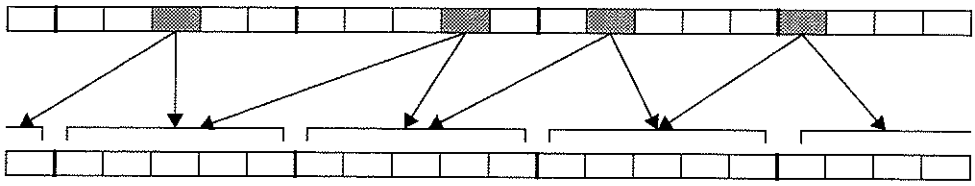


FIGURE 3.3. Local selection and recombination

chromosome in each group is recombined with the winning chromosome from one of the adjacent groups. The best chromosomes are shown in dark grey, and heavy lines separate the groups. The number and size of the groups is determined by the ratio of the population which is to survive. If 20% of the population is to survive to the next generation, then the population is divided into groups of five. Also, the choice of whether to use the left or right neighbours is arbitrary. The creep operator is applied to each chromosome after selection and crossover. The use of localized competition reduces the risk of premature convergence in a population. If the chromosomes were competing in a single pool, then extremely fit individuals might be able to dominate the population, causing the algorithm to converge. In this method, highly fit chromosomes can only recombine with chromosomes in the neighbouring group. They can spread throughout the population, but only by migrating from group to group over many generations.

Dellaert and Vandewalle performed an experiment using this algorithm. They ran the GA for 1000 generations, using a crossover probability of 0.2 and applying a gaussian creep operator with variance of 0.2 to all chromosomes. The GA quickly discovered a feature detector which was capable of correctly classifying all 10 patterns in the validation set. After this there were continual small improvements in the selectivity of the best feature detector. After 1000 generations the best chromosome in the population had a fitness of 104.1, which means it correctly classified all ten patterns in the validation set and had an average selectivity of 4.1. The best feature detector trained to a performance of 106.8 on the training set.

In addition, these authors report that they experimented with a single-layer neural network using the same classification problem, in order to test if the feature detector found by the GA improves the recognition ability of the network. Using the back-propagation learning rule, the network learned to classify all ten training patterns correctly, but could only correctly classify three out of ten validation set patterns. However, this experiment does not really certify the superior performance of the genetic algorithm on this problem, since the limitations of a single-layer neural network for non-linearly separable problems are well known (Minsky and Papert, 1988). A better comparison would be to experiment with a two-layer network, which has the same structure as the two-layer network as used in the GA program, including the local feature detector constraints. An experiment with such a network is presented in section 3.3.3.

There are several novel aspects to the techniques developed for this study which makes it of interest. The use of the weight multiplier in the chromosome is different from other studies that have used the genetic algorithm to optimize the weights of a neural network. In addition, the localized competition within the population is a unique variation on other methods, which have attempted to use the idea of positional separation to reduce the possibility of premature convergence.

Dellaert and Vandewalle have attempted to incorporate the concept of generalization into the fitness function, although care must be taken in drawing conclusions from this measurement. An error measure from a set of data which is applied to any classification scheme, and then fed back into the tuning of that classification scheme, no longer provides a good indication of that classification system's *true error*. The only way to assess the true error of a classification system is to apply data which has never been used in the training of the classification system. The present system gives an indication of the system's apparent error. Even though no derivative information from the validation pattern set is used by the neural learning rule, information about the classification rate is used by the GA. Hence, the search for feature detectors is being driven towards the correct classification of that particular data set.

Two experiments are presented below. The first is a reproduction of the experiment with the GA/BP hybrid algorithm used by Dellaert and Vandewalle (1994). The second set of experiments use one and two-layer neural networks using the BP algorithm alone for comparison purposes.

3.3.2 Single Feature Detector – Experimental Results

This section presents the results of experiments which were conducted to replicate the techniques described by Dellaert and Vandewalle. While the experimental results were not identical to those demonstrated in the earlier paper, for the most part they corroborate those findings. The genetic algorithm was able to find a feature detector within the first 150 generations which was incorporated in a neural network capable of training to 100% correct classification of the validation pattern set as shown in Figure 3.4. The best chromosome of the run had a performance of 104.022 in the 151st generation of the algorithm. Table 3.1. shows the

Parameter	Value
$p_{crossover}$	0.2
p_{creep}	1.0
Creep Variance	0.2
Proportion selected for next generation	0.2
Number of generations	300
Population Size	50
BP Iterations	30 on-line
Learning Rate	0.2

TABLE 3.1. Single feature experiment

GA parameters which were used. As in the original paper, the creep operator was applied to every chromosome in the population. The proportion selected for the next generation applies only to the local selection operator, which propagates a predetermined number of chromosomes to the next generation. This value is not relevant to the experiment using roulette wheel selection

There are two notable differences between the results of this experiment and the results of the earlier work. One is that the best feature detector was not found until later in the GA's run, whereas the best genome found in (Dellaert and Vandewalle, 1994) appeared in the first few

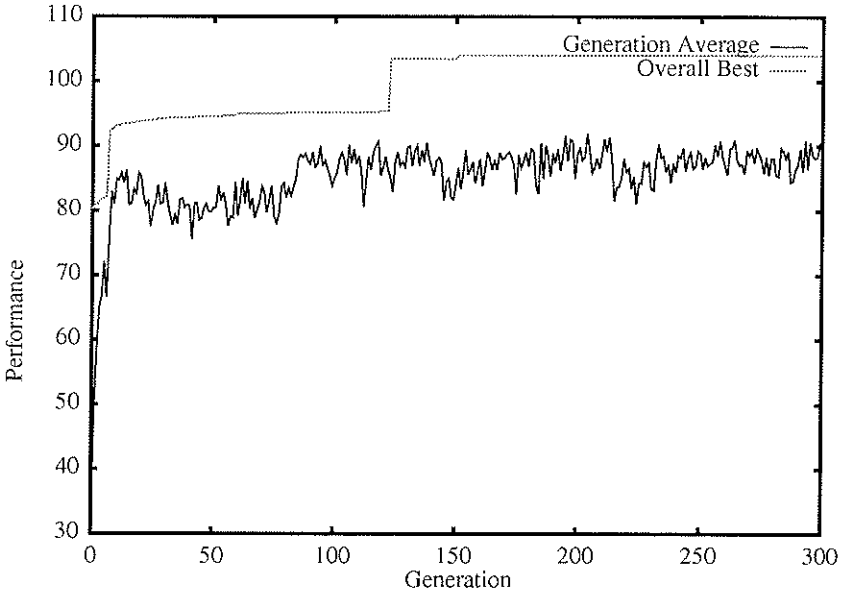


FIGURE 3.4. Single Feature, Local Selection Experiment

iterations of the genetic algorithm. In addition, the population average performance was slightly lower. In this experiment, the population average performance seems to fluctuate between 80% and 90% classification accuracy. In the Dellaert and Vandewalle experiment the average performance maintained a level around 90% accuracy throughout. While an attempt was made to keep all the variables in the experiment the same, there was a difference in the size of the patterns used in the experiments. The input patterns used in this experiment were larger (15x15 compared with 11x8), and the digits accounted for a smaller portion of the image area. Also, it is unclear from their report how often they applied the real number creep to the chromosomes in the population. For this experiment, the real number creep operator was applied to every member of the population, thus making the GA more like Evolutionary Programming, but it could also have been applied to each chromosome with some fixed probability. This might account for the observed differences in classification accuracy.

To verify the Dellaert and Vandewalle claims of superior performance of the local selection algorithm, over more standard selection methods, a second experiment was conducted using the same parameters, except that the local selection algorithm was replaced with roulette

wheel selection with linear scaling, used in many implementations of the GA (Goldberg, 1989). Figure 3.5 shows that this experiment did not converge as quickly, or as well, as the local selection algorithm. By only choosing the chromosome in each group with the highest fitness, the local selection operator is presumably putting greater selection pressure on the population than the roulette wheel selection algorithm, which allows poorly performing chromosomes to reproduce with some small probability.

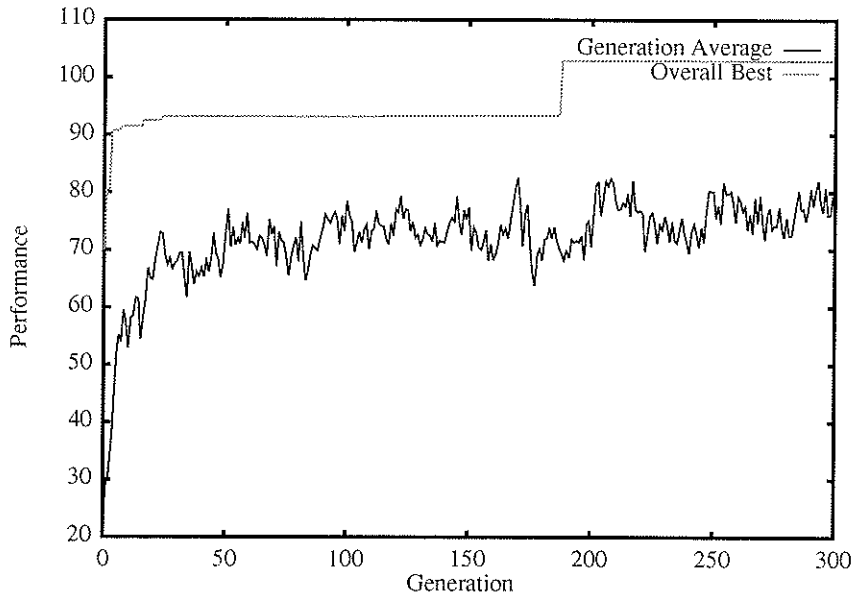


FIGURE 3.5. Single Feature, Roulette Wheel Selection Experiment

Figure 3.6 shows a plot of population-average selectivity versus the generation for both runs of the experiment. The graph clearly shows that the population average selectivity levels off after about 150 generations. The highest selectivity values reach 4.5 to 5 for the GA using the local selection operator, while the GA using roulette wheel selection does not force convergence of the selectivity measure to higher values. Once again this illustrates the differences between the two selection schemes. Because the local selection operator only chooses the highest fitness chromosome in a local region, it is sensitive to small differences in the performance values. The roulette wheel selection operator with linear scaling is only sensitive to small differences in the performance values when the population becomes extremely uniform, the scaling of the objective values biases the selection towards the chromosomes with higher selectivity. When the performance values are diverse, the selectivity measure will be dwarfed by the classification accuracy in the scaling process, and the roulette wheel selection operator will be insensitive to differences in selectivity.

The claim for the local selection technique was that it reduced the possibility of premature convergence. For this to occur we would expect that the population of the GA using the local selection operator would have greater diversity. One way of measuring the spread of the population is to calculate the variance at each iteration of the GA. This calculation is given by

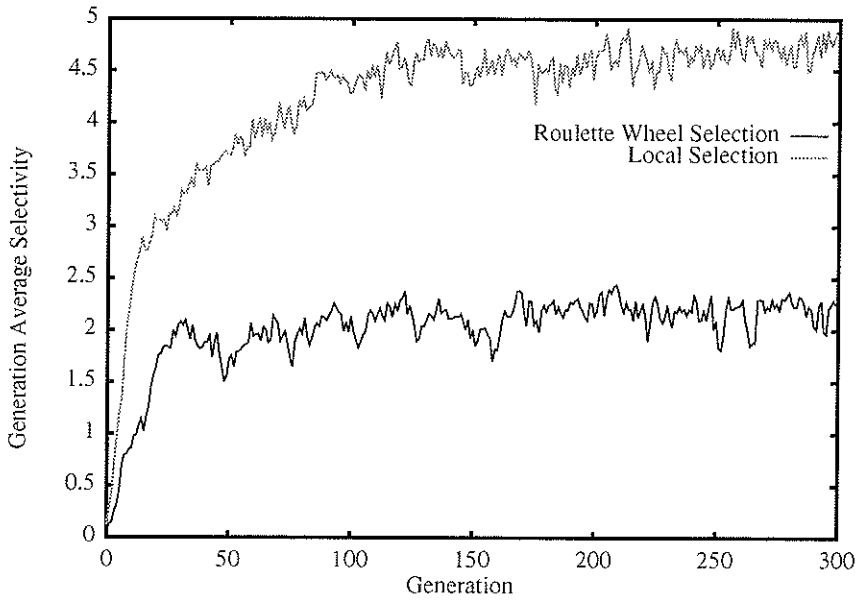


FIGURE 3.6. Selectivity of Local and Roulette Wheel Selection

$$s^2 = \frac{1}{N}\sum_{n=1}^{N} (x_n - \bar{x})^T \cdot (x_n - \bar{x}) \quad , \qquad \text{(EQ 3.7)}$$

where N is the number of chromosomes in the population, x_n is the n^{th}chromosome and $\bar{x}$ is the population average chromosome. Thus, s^2 is a scalar value which indicates the spread of the chromosomes in the population.

Figure 3.7 shows the graph of this statistic over the course of the experiment. This indicates that, contrary to our expectations, the roulette wheel selection operator gives the population a higher variance than the local selection operator. One possible explanation for this effect might be the way the local selection operator chooses the winning chromosome in each sub-group of the population. This tends to drive the population towards convergence, while at the same time protecting weaker chromosomes by the spatially-distributed competition in the population. Thus, there are two competing effects in the local selection operator. The high

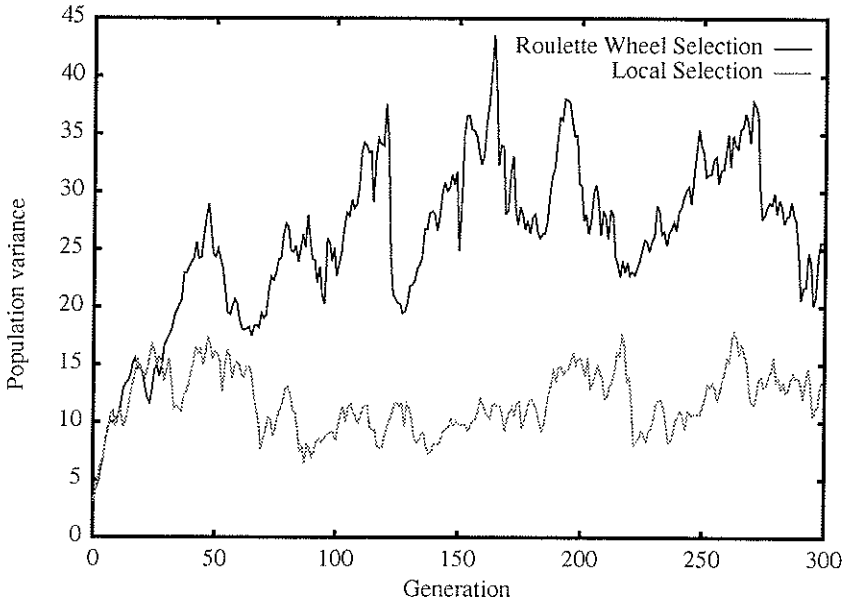


FIGURE 3.7. Population Variance of Local and Roulette Wheel Selection

diversity of the roulette wheel selection GA might also be indicative of the diluting effect of poor chromosomes which are selected to propagate from one generation to the next.

3.3.3 Back-Propagation Comparison – Experimental Results

While the genetic algorithm found a feature detector capable of correctly classifying the digit image patterns, there remains a question of how much of the classification success is due to the gradient descent learning process embedded in the evaluation of the chromosomes. In other words, could the back-propagation algorithm using the same neural network architecture train a network capable of correctly classifying both the training and the validation pattern set? To answer this question, two additional experiments were performed using a one layer, and a two layer architecture. In the single layer architecture, the output units are fully connected to the input field. The two layer architecture is the same as the one used in the GA/BP experiments (Figure 2.16). The hidden layer has a number of units equal to the number in the input field, and each unit is connected by the same set of weights to a 5x5 patch in the input field. The output units are fully connected to the hidden layer.

To make the comparison between the GA/BP method and back-propagation fair, the method of cross-validation is used. In training a neural network, we expect that the error on the training set will continually decrease. To ensure that the network is able to generalize from the training data to other data, of which the training set is representative, training is periodically stopped to compute the error of the network on a second validation data set. What typically occurs is that the error on the validation will decrease at the start, but as the network begins to overfit, or memorize, the training set, then the error on the validation set will begin to

increase. At this point training of the network should stop. In these experiments, we did not use the squared error to perform cross-validation, but used the classification performance measure (eq. 3.5) instead. The values which are reported for all experiments are the best performance values achieved by the network, for a given run, on the training and the validation sets.

The first experiment investigated whether the first layer containing the feature detectors is really necessary to solve the digit classification problem. To verify this an experiment was performed using a single-layer neural network. One of the problems that became immediately apparent in configuring this experiment is that the learning rate of 0.2 used in the previous experiment is too large for the single-layer net. For many values of the starting weights the learning can become unstable causing the weights to become large and the sigmoidal function of the output units to saturate in the flat regions. For this reason a lower learning rate of 0.05 was used. The following tables show the parameters, the maximum performance values and the average performance values from these experiments.

Parameter	Value
Initial Weight Range	±0.5
Learning Rate	0.05

TABLE 3.2. Single layer BP experiment

Maximum Training Performance	100.5601
Average Training Performance	80.5465
Maximum Validation Performance	63.7178
Average Validation Performance	36.1975

TABLE 3.3. Single layer BP experimental results

The complete results of this experiment are given in Table A.1 of Appendix A. The results clearly show that the single-layer network is not capable of solving the classification problem. On the training patterns it does not reach 100% accuracy in most of the trials, and it also does an inferior job of classifying the validation data. This is similar to an experiment reported in the Dellaert and Vandewalle (1994) paper. They report that a single layer network is not capable of solving this classification problem, which we have verified here. However, one cannot make the case that the GA/BP algorithm is superior, based on this, because it is not fair to compare a single layer classification network with a two layer classifier.

The second experiment investigated the performance of the BP algorithm using the two layer network with the same restricted network architecture as the one used in the GA/BP method. Because the weights to each hidden unit are constrained to be the same as the weights to every other hidden unit, the standard back-propagation algorithm cannot be used. In the normal back-propagation algorithm the change in each weight is proportional to the error derivative of that individual weight, and thus, each weight will change differently. In the technique of weight sharing (LeCun et al., 1989), the weight change is taken to be proportional to the sum of all the derivatives of the weights which are constrained to be equal. Thus, if the weights

start with the same initial value they are guaranteed to remain equal throughout the learning process.

The network was trained for 450 iterations of on-line learning on the training set, and every 10 iterations the network was evaluated on both the training set and the validation set using the performance measure. Table 3.4 shows the network parameters chosen for the experiment. After 450 iterations of on-line BP learning (i.e. 45 complete passes through the training set) the network was randomized and restarted. This was repeated 100 times, and Table 3.5 shows the best and average recorded performance of the network. The complete results of this experiment are given in Table A.2 in Appendix A. The number of training epochs for the neural net-

Parameter	Value
Initial Weight Range	± 0.5
Learning Rate	0.2

TABLE 3.4. Two layer BP experiment parameters

work with BP is significantly larger than the number of epochs used in the GA/BP hybrid method of training, which was presented the entire training set only 3 times. However, three hundred generations of a population of 50 amounts to 15, 000 times that the GA/BP hybrid network had to train a neural network, compared with only 100 times for this experiment.

Table 3.5 illustrates a problem commonly found with neural networks, and pattern classification systems in general. The network learns to perfectly classify the patterns in the training set, but is unable to classify the patterns in the validation set about which it has no information. The evolved neural network, on the other hand, was able to find a network which cor-

Maximum Training Performance	100.4917
Average Training Performance	100.4606
Maximum Validation Performance	90.4022
Average Validation Performance	72.5448

TABLE 3.5. Two layer BP experiment results

rectly classified both the training set and the validation set with 100% accuracy.

3.3.4 Comments and Conclusions

We can see from the experiments performed here that, while the genetic algorithm hybrid method did not significantly outperform the BP neural network on the digit classification task, there is a problem in training a network to generalize to the digit patterns in a different typeface. The problem of generalizing the classification of characters in different fonts is probably too difficult to be solved with such a small amount of data. Typically, in classification problems, one requires a large representative data set on which to train the classifier, but this problem uses a small set of training data which is only representative if the font chosen for

validation is similar to the font used for training. If the two typefaces are very different then it is unreasonable to expect the network to generalize, even thought the GA is selecting for networks that can generalize between the two.

A more reasonable approach might be to accumulate digit patterns from several different font sets and randomly partition the data into the training and validation data sets. Another approach is to use a single font, and corrupt it with noise so as to generate enough data patterns for the network to learn to generalize in spite of the random variations. However, the use of more data patterns will cause a corresponding increase in the run time of the algorithm, which is a problem considering the presently long run time of this method.

Unlike other ANN/GA hybrid algorithms described in the previous chapter, not all the weights of the network are specified in the chromosome. For a network of this size that would create very long genomes, increasing the size of the search space for the genetic algorithm. Instead, at each evaluation of a neural network, the chromosome specifies the weights of the first layer, and the initial weights of the second layer are randomized prior to optimization by gradient descent. Constructing the network in this manner shortens the chromosome, but it also introduces a stochastic component into the fitness function of the GA.

3.4 Experiments with Multiple Feature Detectors

3.4.1 Description of the Experiments

This set of experiments uses a modified version of the algorithm which addresses the limitations of using neural networks with only a single feature detector for pattern recognition tasks. The algorithm employs a longer chromosome, which concatenates together several feature detector templates. The construction of a neural network using these multiple-template chromosomes can be effected in a number of ways. The most direct way of incorporating multiple feature detectors would be to add an equal-sized hidden layer of the neural network model (shown in Figure 2.16) for each of the templates which is encoded in the chromosome. Each of these sets of hidden layer units would be fully connected to the units of the output layer. The problem with this approach is that it greatly increases the number of units and connections in the network, and that much of the information which is received by the output units becomes irrelevant. It would be preferable, rather than attempting to match the feature detector at every possible location in the receptive field, to position the feature detector at the point where it can achieve the most discrimination amongst the patterns in the training set. Figure 3.8 illustrates a network with three local feature detectors positioned at specific locations. This accomplishes the goal of reducing the number of units and connections in the network as compared to the other approach, and thereby reducing the dimensionality of the input to the output units of the network.

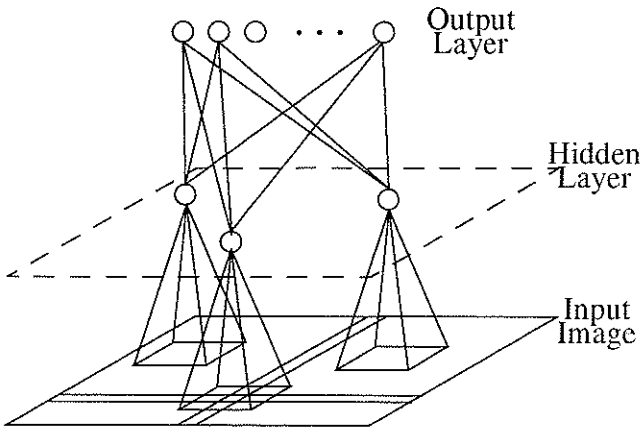


FIGURE 3.8. Neural network with multiple local feature detectors

Given that the best position of these feature detectors cannot be known in advance, their positioning is left as an optimization problem for the genetic algorithm. The genetic algorithm is ideally suited to this task, because the position of the template in the receptive field is a discrete quantity, and there is no differentiable function which relates the position of a feature detector to the error function of the network's output. Thus, a gradient-type optimization algorithm could not be used for this purpose.

For the purpose of specifying the x and y coordinates of the centre of the template in the receptive field, two additional fields are added to the chromosome for each of the feature detectors in the chromosome. The chromosome in the previous experiments had $5 \times 5 + 2 = 27$ fields in the chromosome including the bias and weight multiplier, whereas the multiple-template chromosome has $(5 \times 5 + 4 = 29) \times$ (the number of templates). This means that a chromosome containing a modest number of templates can become quite long, and the search space which must be addressed by the genetic algorithm grows exponentially with the length of the chromosome. This problem will be addressed in another set of experiments described later.

The addition of the x and y fields to the chromosome necessitates two other changes to the algorithm. First, these components will have a larger range of values than the weights, biases and multiplier values. Therefore, the chromosome initialization and gaussian creep operators are modified so that they treat the coordinate positions in the chromosome differently than the others. The random initial value and the creep variance for these values are scaled by the vertical and horizontal size of the input image respectively. Furthermore, the coordinate positions have constraints which cannot reasonably be exceeded. If an x or y coordinate for a particular feature detector template falls outside the bounds of the input image, the network will lose any useful information from that feature detector. This is acceptable as long as the number of chromosomes with invalid templates is in the minority and the decline in fitness value is severe enough that they will naturally be culled from the population. However, if there were a situation in which many of the chromosomes had enough feature detectors out of bounds so

that their fitnesses became uniformly zero then the GA would have no way to choose between them. Goldberg (1989) recommends that, when dealing with problems which contain constraints, the fitness function should be made to incorporate some information about the degree to which a solution violates the constraint. He recommends subtracting a penalty term from the fitness function which increases with the degree that a chromosome violates a constraint. Rather than subtract a penalty term from the fitness, we limit the range of fitness values which a chromosome containing invalid templates can have. The fitness function checks for the coordinates of any template which are out of bounds, and sums the square of the distances out of bounds. It then computes the fitness based on the inverse of this sum, such that a chromosome containing an invalid coordinate has a maximum fitness of 5, and this function decreases with the distance that the template is out of bounds.

$$f(x) = \frac{5}{1 + \sum_{j \in X} d_j} \qquad \text{(EQ 3.8)}$$

X is the set of templates within the chromosome which are out of bounds, and d_j is the square of the Euclidean distance from the edge of the input field to the centre of the templates' specified position. The value 5 was chosen so that chromosomes containing invalid templates can not compete with valid chromosomes, which can correctly classify as little as 5 percent of the input patterns.

A different set of artificial patterns was generated to specifically test the ability of the genetic algorithm to position the feature detectors in the appropriate location in the receptive field. Figure 3.9 shows the patterns which were used. The patterns are separated into four classes: an x shape, two diagonal lines, and an absence of any input at all. An example of each shape appears four times in the data set, once in each corner of the input field. The empty input pattern is repeated four times, so that it is equally represented.

There are four local patterns, but they are placed in the four corners of the input field, creating a total of sixteen patterns which the network must identify. The localized nature of the features in these patterns requires the genetic algorithm to properly position a limited number of local feature detectors so that the neural network has enough information in the hidden layer to correctly classify them. This experiment removes the element of testing the network's generalization ability, as the same pattern set was used for both validating and training. Thus, the problem tests only the ability of the GA to generate good feature detectors in useful positions.

The use of a new pattern set also precipitated a change in the way that the performance measure was made. In the previous experiment there were only ten patterns, so taking the number of patterns correctly classified and multiplying by ten generated a percentage value. The code was modified for this experiment to compute the actual percentage. Also, the selectivity calculation was modified to compute the average over the patterns correctly classified rather than the sum of the selectivities over all correctly-classified patterns. This results in smaller values

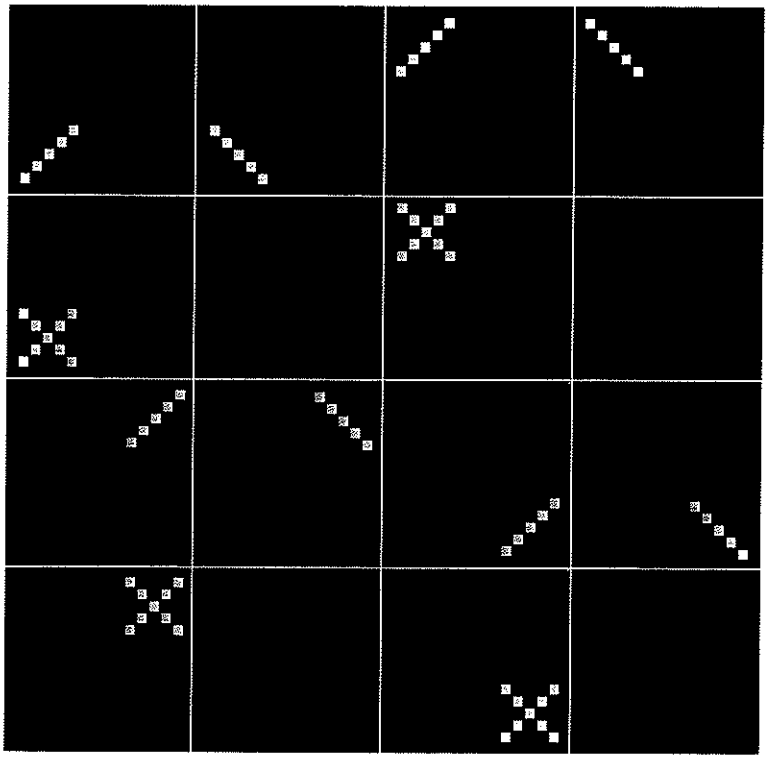


FIGURE 3.9. Local feature input patterns

for the selectivities, but eliminates the possibility of discontinuous jumps in the selectivity value when an extra pattern is correctly classified.

Furthermore, this set of experiments utilised the conjugate-gradient algorithm for the gradient descent learning method in lieu of back-propagation. The conjugate-gradient algorithm uses a line search rather than a fixed step size to move from point to point in the search space. This removes the dependency on the learning rate parameter, and the difficulty of setting it to a reasonable value which is valid for all the networks trained during a run.

3.4.2 Experimental Results

The experiments using multiple-template chromosomes were performed three times. Each experiment used a different number of epochs of conjugate-gradient learning. Also, since the local selection rule achieved the best performance in the first set of experiments using the sin-

gle template chromosomes, the local selection operator was used. Table 3.6 shows the parameters for the experiment.

Parameter	Value
$P_{crossover}$	0.2
P_{creep}	0.2
Creep Variance	0.2
Proportion selected for next generation	0.2
Population Size	80
Number of Templates	4

TABLE 3.6. Multiple template experiment parameters

The following graphs show the results from the three experiments.

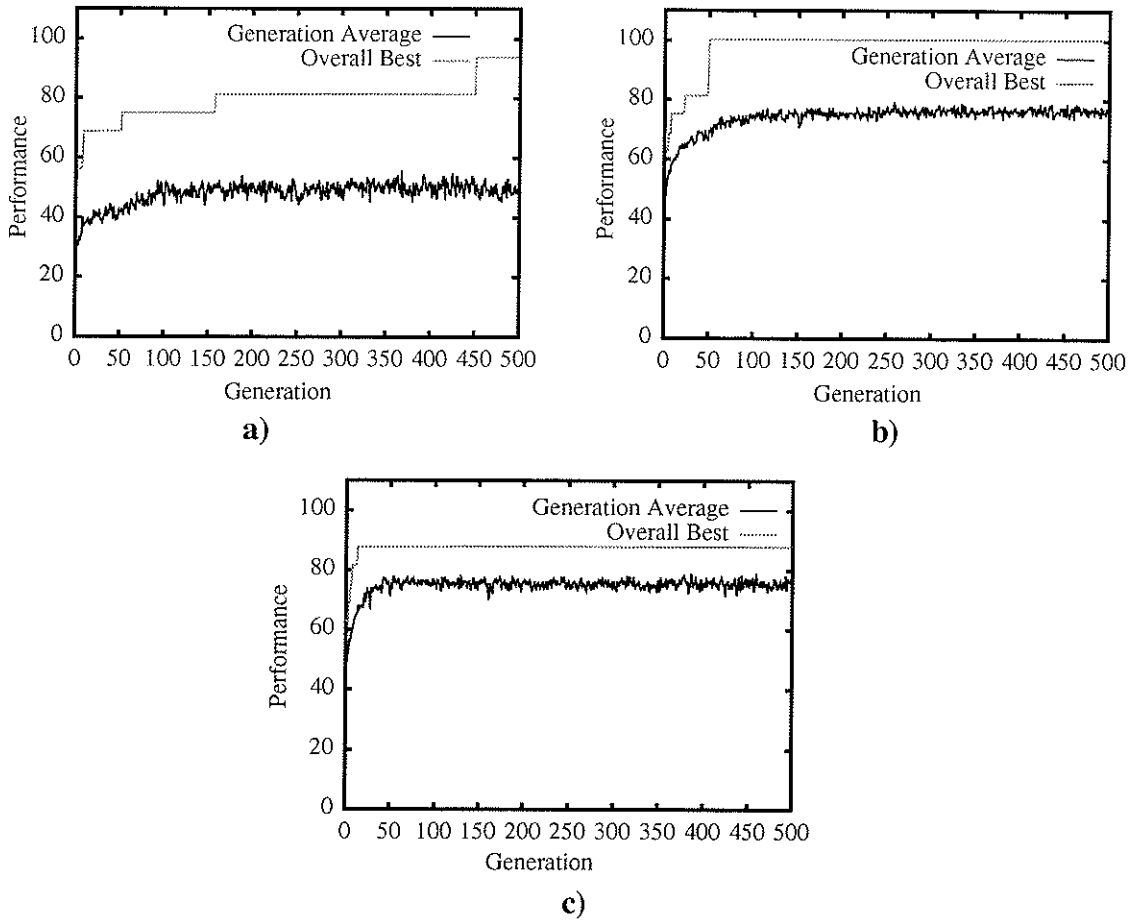


FIGURE 3.10. Multiple template chromosome experiments for (a) 5 epochs of CG learning, (b) 10 epochs of CG learning, and (c) 15 epochs of CG learning

There is a clear increase in average performance of the population from the experiment using 5 epochs of CG learning to 10 epochs, and a smaller improvement using 15 epochs of CG learning. This may indicate that there is limit to the number of iterations of neural learning required to train the output layer weights of this network. This is expected to be a problem-specific result. While the experiment with 10 epochs of CG learning found the chromosome with the best overall performance, 100.284, this is less indicative of the overall quality of the solutions produced by the two experiments than is the population average performance.

We can see that only the second run of the algorithm produced a solution which could correctly learn all the patterns in the training set. While this shows that it is possible for the algorithm to correctly position the feature detectors and, and to configure the feature detectors so that the CG algorithm can learn to make the correct classifications at the output layer, the low values of the population average performance are discouraging.

3.4.3 Comments and Conclusions

While the experiment showed mixed results in the performance of these multiple-template chromosomes, determined and positioned by the genetic algorithm, there are two factors which we must take into consideration. First, by reducing the size of the hidden layer from that of the input field to a few units equal to the number of templates in the chromosome, we have greatly reduced the amount of information available to the output layer units for making their decision. This is a very common technique for information compression and dimensionality reduction. A classic example of this is the binary 8-3-8 encoder network (Rumelhart et al., 1986). By doing this, we have placed more of the onus of the pattern recognition task on the feature detectors, increasing the importance that the feature detectors which are generated by the genetic algorithm are efficient at discriminating amongst the image patterns.

Secondly, using multiple features in each chromosome increases their length and with it the size of the search space for the genetic algorithm. While the run was performed with a slightly larger population than in the first set of experiments, 80 as opposed to 50, the size of the search space for the feature detectors increases exponentially with the number of positions in the chromosome. While the particular experiments here used only 4 feature detectors in a chromosome, it is conceivable that a more complex problem would require more feature detectors and hence an even longer chromosome. This presents a scaling problem for the GA and a method to reduce the size of the chromosomes is therefore considered in the following section.

3.5 Cooperative-Competitive Fitness Evaluation

3.5.1 Description of the Experiment

The experiments described in this section used a further modification of the algorithm to perform the image recognition task, which we will refer to as the cooperative-competitive chromosome. While still using multiple feature detectors to construct and evaluate neural networks, this version attempts to decrease the length of the chromosomes which are required to represent the multiple feature detectors. Moriarty and Miikkulainen (1994) describe a system called SANE (symbiotic adaptive neuro-evolution), in which the chromosomes encode a partial solution to a problem which can be combined with partial solutions in other chromosomes, to create a complete solution which is then evaluated. They assign a fitness to each chromosome by constructing many complete solutions, and assigning to each chromosome the average fitness of the complete solutions in which it took part.

We have applied this principle to our image recognition algorithm. Rather than representing many feature detectors, each chromosome contains a single feature detector template and its accompanying parameters. The algorithm constructs a neural network by selecting N chromosomes at random out of the population, where N is the number of feature detectors desired for the first layer of the neural network. The resulting network is the same as one constructed from the multiple-template chromosomes from the previous section, except that the weights for each hidden layer unit come from a different chromosome. In each generation, the algorithm constructs a sufficient number of networks so that each chromosome will have been selected several times, and the chromosome is assigned the average of the fitness values of the networks in which it participated. Thus, the fitness of an individual no longer depends on the solution which it alone contains, but on the potential for that partial solution to cooperate with other partial solutions. Moriarty and Miikkulainen (1994) make the point that this forces the chromosomes to specialize, thereby increasing population diversity.

One problem with this scheme which can easily be identified is that a chromosome which is a good detector for a particular feature may need chromosomes which complement it to construct a useful network. A network constructed from randomly chosen chromosomes might choose several chromosomes containing templates which actually detect the same feature. These chromosomes will receive low fitness values, despite the fact that they may very well be good for a specialized purpose. In the previous chapter we mentioned the problem of competing representations, where a set of functional blocks which are performing the same set of functions in two different chromosomes, do not appear in the same order. When the two chromosomes are crossed over then the correct complement of functional blocks does not appear in either chromosome. The multiple template encoding, presented in the last section, is susceptible to this problem, and the cooperative competitive algorithm has an analogous shortcoming.

To combat this problem a second, slightly different, algorithm was also studied. The population was divided into sub-populations equal to the number of templates used to construct a

neural network. During the population evaluation phase, rather than selecting chromosomes from the entire population, the algorithm selects one chromosome from each sub-population. This allows each sub-population to specialize on a template which detects a particular feature and ensures that the specialized chromosomes are combined with their correct complements. There are two options for adapting the selection operator to this algorithm. If the selection operation remains unchanged, then it will choose chromosomes from all parts of the population as candidates for crossover, and genetic material will be allowed to migrate across boundaries of sub-populations. The selection algorithm can also be modified so that only chromosomes within the same sub-population are selected for crossover. The second approach was taken here.

The other operations of the algorithm, recombination and genetic creep, remain unchanged. The selection operator, used in both the full population and the sub-population versions of this algorithm, was the same local selection scheme used in the other experiments presented in this chapter.

3.5.2 Experimental Results

Table 3.7 shows the parameters which were used for the first cooperative-competitive experiment. Once again, these are the same as the parameters used for the multiple-template experiment, and the experiment is repeated with 5, 10 and 15 epochs of conjugate gradient learning.

Parameter	Value
$P_{crossover}$	0.2
P_{creep}	0.2
Creep Variance	0.2
Proportion selected for next generation	0.2
Population Size	80
Number of Templates	4
Number of trials each generation	160

TABLE 3.7. Cooperative-competitive experiment parameters

The number of trials refers to the number of neural networks which the algorithm constructs each generation. In the multiple-template experiments the algorithm constructs, trains and evaluates one neural network for each chromosome in the population, for a total of 80 networks in each generation of the algorithm. Here the algorithm constructs twice as many networks. The computational cost is approximately doubled, because the overhead of running the GA is, in this case, dwarfed by the computational time of constructing, training and evaluating the neural networks. The reason that 160 trials were used was to reduce the probability that a chromosome would have a fitness score of 0 because it was not selected to participate in any trials. In this experiment 4 templates were used in each network. Thus, there were a total of

640 chromosomes selected to participate in the 160 networks. Each member of the population was selected with equal probability. If each chromosome has an equal probability of being selected, the expected number of times that each chromosome will be selected is

$$640 \times \frac{1}{80} = 8$$

A small percentage of chromosomes at each generation are invalid and are not selected. Thus, for valid chromosomes, the actual expected value is slightly higher. To reduce the amount of computation, the number of trials can be reduced, but this increases the likelihood of a chromosome remaining unselected and receiving a fitness score of 0 irrespective of its actual worth.

Figure 3.11 shows the performance of the standard cooperative-competitive algorithm.

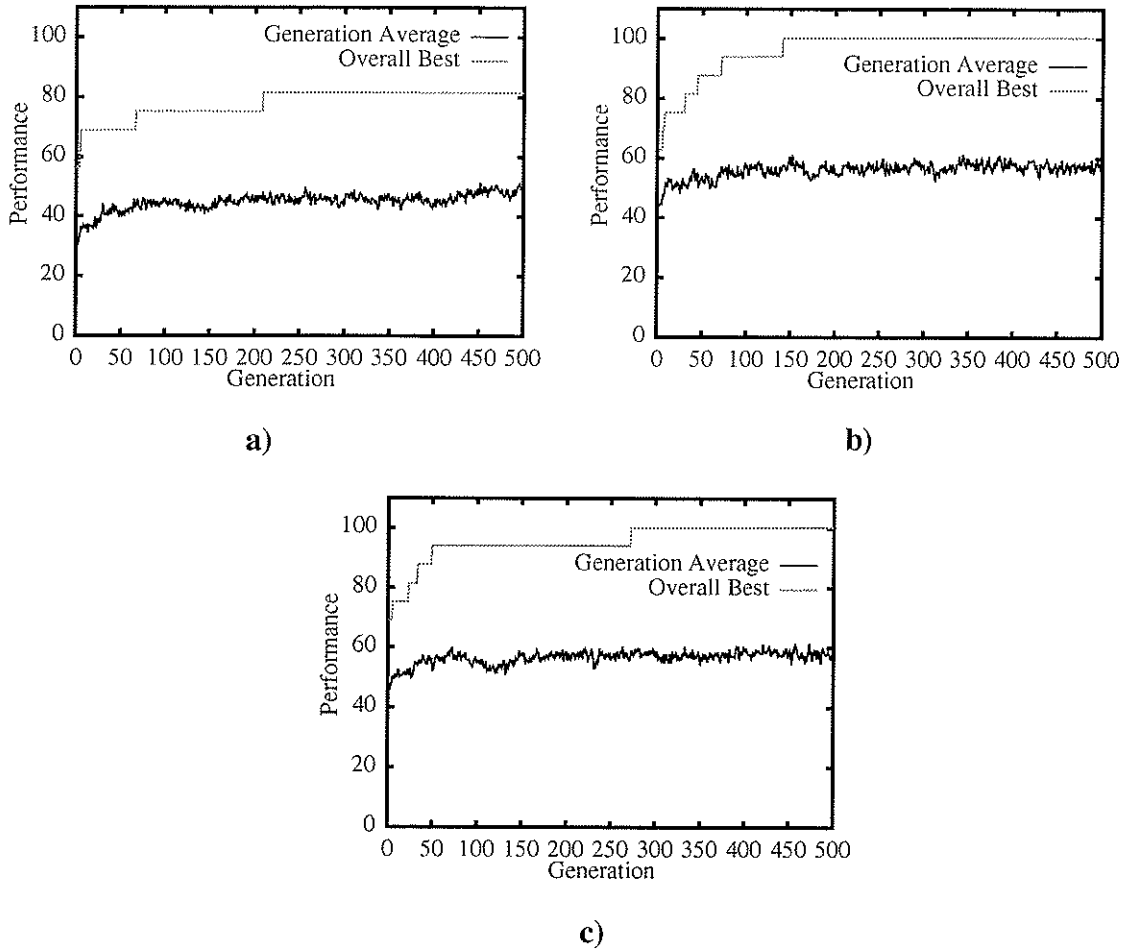


FIGURE 3.11. Cooperative competitive algorithm with (a) 5 epochs of CG learning, (b) 10 epochs of CG learning, and (c) 15 epochs of CG learning

There is a notable decrease in population average performance between the cooperative-competitive algorithm and the multiple-template algorithm shown in the last section. As reported by Moriarty and Miikkulainen (1994), this algorithm promotes chromosomes which work well with others, but it does not necessarily allow them to specialize into the sub-tasks which might be required to solve the problem. The multiple-template algorithm allows chromosomes to specialize into sub-tasks, because they co-evolve in the same chromosome and they are always guaranteed to be implemented together in the same network

For this reason, a second experiment was performed with the, above mentioned, modification that chromosomes for construction of a network are not picked from the entire population, but each one is selected from a sub-population. The parameters used in this experiment are shown in Table 3.8. Once again we repeat the experiment using 5, 10 and 15 epochs of conjugate gradient learning in the chromosome evaluation function. Since we are using 4 feature detectors

Parameter	Value
$P_{crossover}$	0.2
P_{creep}	0.2
Creep Variance	0.2
Proportion selected for next generation	0.25
Population Size	80
Number of Templates	4
Number of trials each generation	160

TABLE 3.8. Cooperative-competitive algorithm with sub-populations experiment parameters

in each network with a population of size 80, the population will be divided into 4 sub-populations of size 20. Also, since 25% of the population is permitted to reproduce in the next generation using the local selection operator, the chromosomes in each sub-population are divided into 5 groups of 4 and the best chromosome in each group of 4 is selected to mate with the winning chromosome from it's neighbouring group.

Figure 3.12 shows the results of the experiments performed with the modified cooperative-competitive algorithm.

3.5.3 Comments and Conclusions

The results from the experiment using the cooperative-competitive algorithm with sub-populations show performance which is comparable to that of the multiple template chromosome experiment, and a substantial improvement over the regular cooperative-competitive algorithm. The following section describes two more experiments which were performed to compare the relative performance of the three algorithms on the local feature image recognition problem. The absolute performance of these algorithms is somewhat discouraging. While the algorithms are able to discover a set of feature detectors which CG can use to perfectly clas-

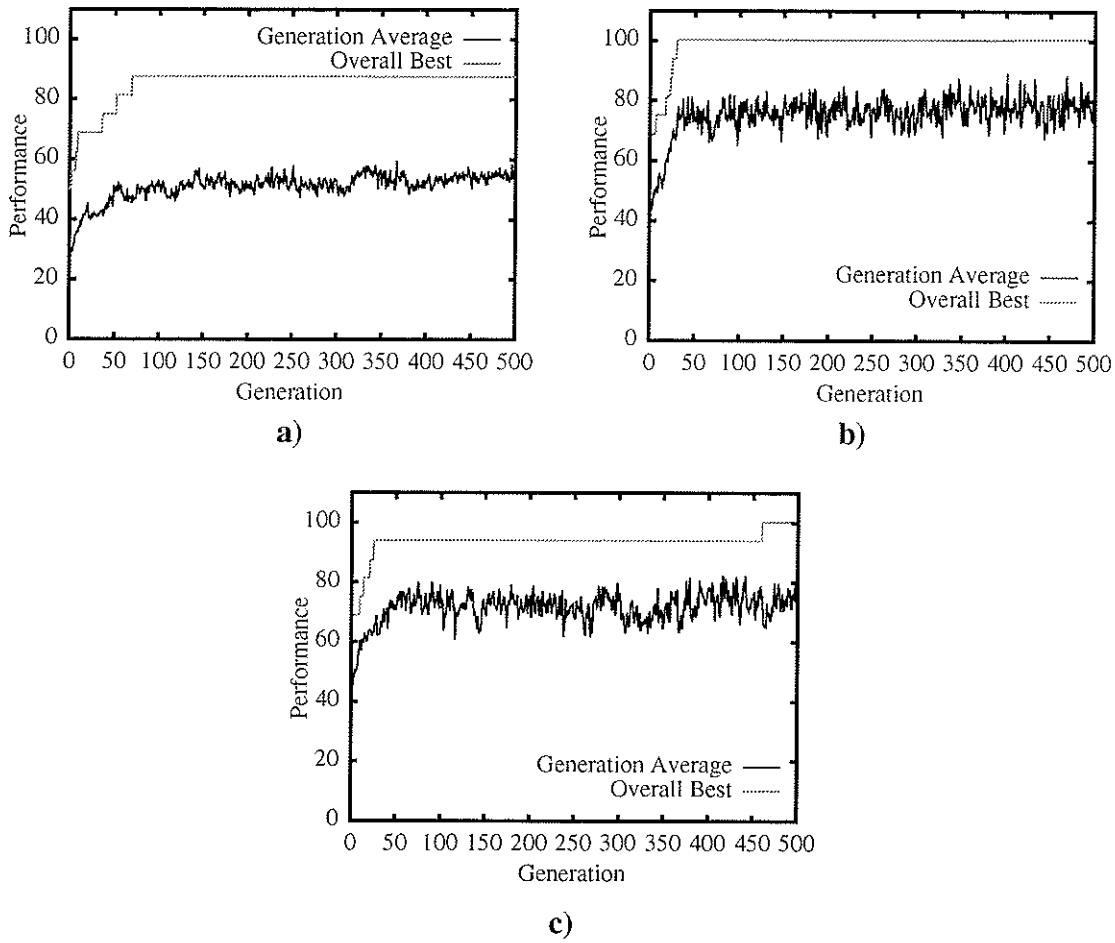


FIGURE 3.12. Cooperative competitive algorithm using sub-populations with (a) 5 epochs of CG learning, (b) 10 epochs of CG learning, and (c) 15 epochs of CG learning

sify the patterns in 15 epochs of learning, the average member of the population achieves performance values in the range of 70 to 80 percent. This suggests that four feature detectors may not be sufficient to solve this problem. The experiments in the next section attempt the problem with both four and eight feature detectors to see if there is a marked improvement in performance.

3.6 Comparison of Multiple Feature Algorithms

3.6.1 Experimental Results – Four Feature Detectors

To better gauge the relative effectiveness of the three algorithms described in the last two sections a broader experiment was performed. For brevity the three algorithms, multiple template, cooperative-competitive and the cooperative-competitive algorithm with subpopulations, will be referred to as the MTA, CCA, and CCAS algorithms. For this experiment each of the algorithms was run five times on the same local feature pattern set used in the previous two experiments. Once again, the same pattern set was used for both training and validating, so the fitness function does not measure the generalization ability of the network. The parameters which were used for the two experiments are shown in Table 3.9. The 'number of trials' parameter does not apply to the MTA experiment, only to the two cooperative competitive versions of the algorithm. For this experiment the cooperative-competitive algorithms performed 160 trials compared to 80 for the MTA. This was done to reduce the probability that a chromosome might not be selected to participate in a single trial. Because the cooperative-competitive algorithms calculate the fitness of a chromosome by averaging over the trials in which it participates, this will not affect the fitness values of the chromosomes. The overall best network, on the other hand, is simply the best of all the trials during the run. Thus, in comparing the best overall network generated by the three algorithms, there might be a bias toward the two cooperative-competitive algorithms. Simply because they train and test twice as many networks, they are more likely to generate a better network. This discrepancy was eliminated in the eight feature detector experiment.

Parameter	Value
$P_{crossover}$	0.8
P_{creep}	0.2
Creep Variance	0.2
Proportion selected for next generation	0.25
Population Size	80
Number of Feature Detectors	4
Epochs of CG learning	15
Number of trials each generation	160

TABLE 3.9. Four feature comparison – experiment parameters

For each experiment, the performance of the best overall network generated by each algorithm is maintained. The best performing networks found over each run of each algorithm are shown in Table 3.10. Each table entry shows the total performance value and the selectivity of the network on the data set.

	Best Run	Worst Run	Others		
MTA	100.369/	87.676/	87.710/	100.162/	93.864/
	0.369	0.176	0.210	0.162	0.114
CCA	100.328/	94.065/	100.263/	100.240/	100.288/
	0.328	0.315	0.263	0.240	0.288
CCAS	100.372/	100.303/	100.364/	100.371/	100.349/
	0.372	0.303	0.364	0.371	0.349

TABLE 3.10. Four feature comparison – best chromosome of each run

The graphs of the best and worst run of each algorithm are shown in Figure 3.13. This gives a clear visual indication of the average population performance and the best overall performance of the networks generated by these two algorithms.

We can see from Table 3.10 that all three algorithms are capable of producing a single network which can classify all of the patterns in the data set. However, CCAS consistently produces a perfectly performing classifier, where the other two algorithms do not. Clearly, the addition of subpopulations to the CCA algorithm is an improvement in terms of both the population average and the overall best networks generated. If we compare the graphs of the best runs of MTA and CCAS, the average network produced by MTA is more successful than the average produced by CCAS, even though the best overall network produced by CCAS is better.

To see if the absolute performance of the classification method could be improved, a second experiment was performed using eight feature detectors, as opposed to four. The genetic operators for this experiment were changed slightly. Where the previous algorithms used only crossover and creep, this experiment made use of a mutation operator which, with a small probability, changes the value of a position on the chromosome to a new random number uni-

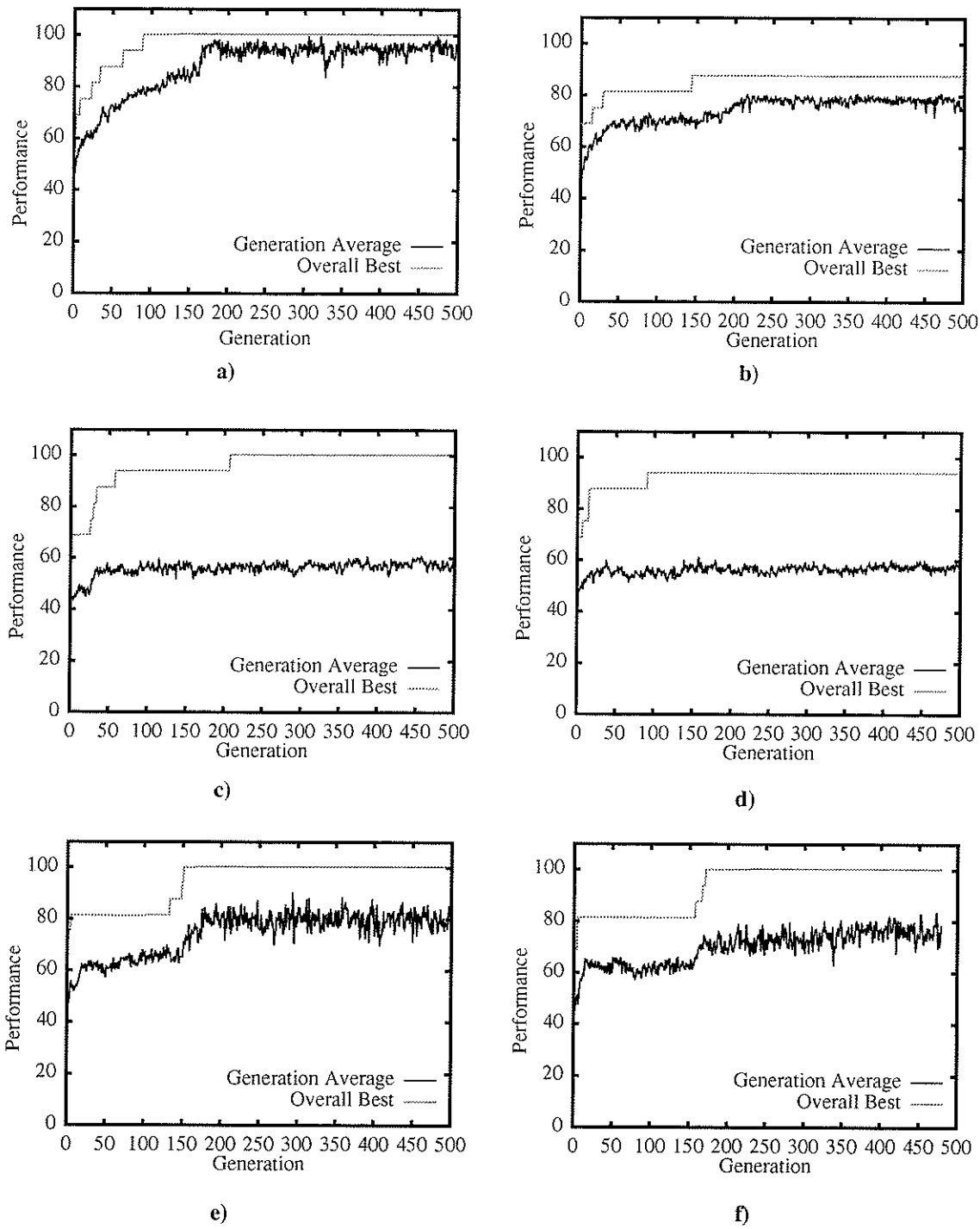


FIGURE 3.13. Four feature detector experiment – Best and population average for a) most successful run of MTA, b) least successful run of MTA, c) most successful run of CCA, d) least successful run of CCA, e) most successful run of CCAS, and f) least successful run of CCAS

formly distributed over some range. Table 3.11 shows the parameters which were used for this

Parameter	Value
$P_{crossover}$	0.6
P_{creep}	0.2
Creep Variance	0.1
$P_{mutation}$	0.01
Mutation range	±0.5
Proportion selected for next generation	0.25
Population Size	128
Number of Feature Detectors	8
Epochs of CG learning	15
Number of trials each generation	128

TABLE 3.11. Eight feature comparison – experiment parameters

experiment. The mutation radius refers to the range, centred on zero, of the uniformly distributed random number used to mutate the values of the chromosome. Like the creep operator the mutation radius is scaled up for those positions of the chromosome which contain the coordinates of the feature detectors in the 16x16 image field. Table 3.12 shows the total performance and the selectivity of the best networks which were generated by each run of each algorithm. The best and the worst results for each algorithm are shown in the first two columns of the table. The evolution of the best and worst runs of each of the algorithms is

	Best Run	Worst Run	Others		
MTA	100.977/	100.665/	100.953/	100.951/	100.910/
	0.9770	0.6651	.9530	0.9509	0.9095
CCA	100.607/	100.391/	100.471/	100.484/	100.518/
	0.6073	0.3905	0.4712	0.4837	0.5176
CCAS	100.972/	100.762/	100.931/	100.918/	00.771/
	0.9719	0.7618	0.9306	0.9184	0.7710

TABLE 3.12. Eight feature comparison – best chromosome of each run

depicted graphically in Figure 3.14.

3.6.2 Comments and Conclusions

By comparing the two experiments one immediately notices a qualitative difference in the graphs of the two experiments. The four feature experiments generally show a strong rising action at some point through the evolutionary process, particularly in the MTA and the CCAS runs. This suggest that the genetic algorithm is playing an important role in organizing the information which is seen by the second layer of the neural network. By comparison, in the eight feature experiments we do not see this rising action of the population fitness, suggesting that with eight features, even if they are not positioned ideally, or do not have an appropriate

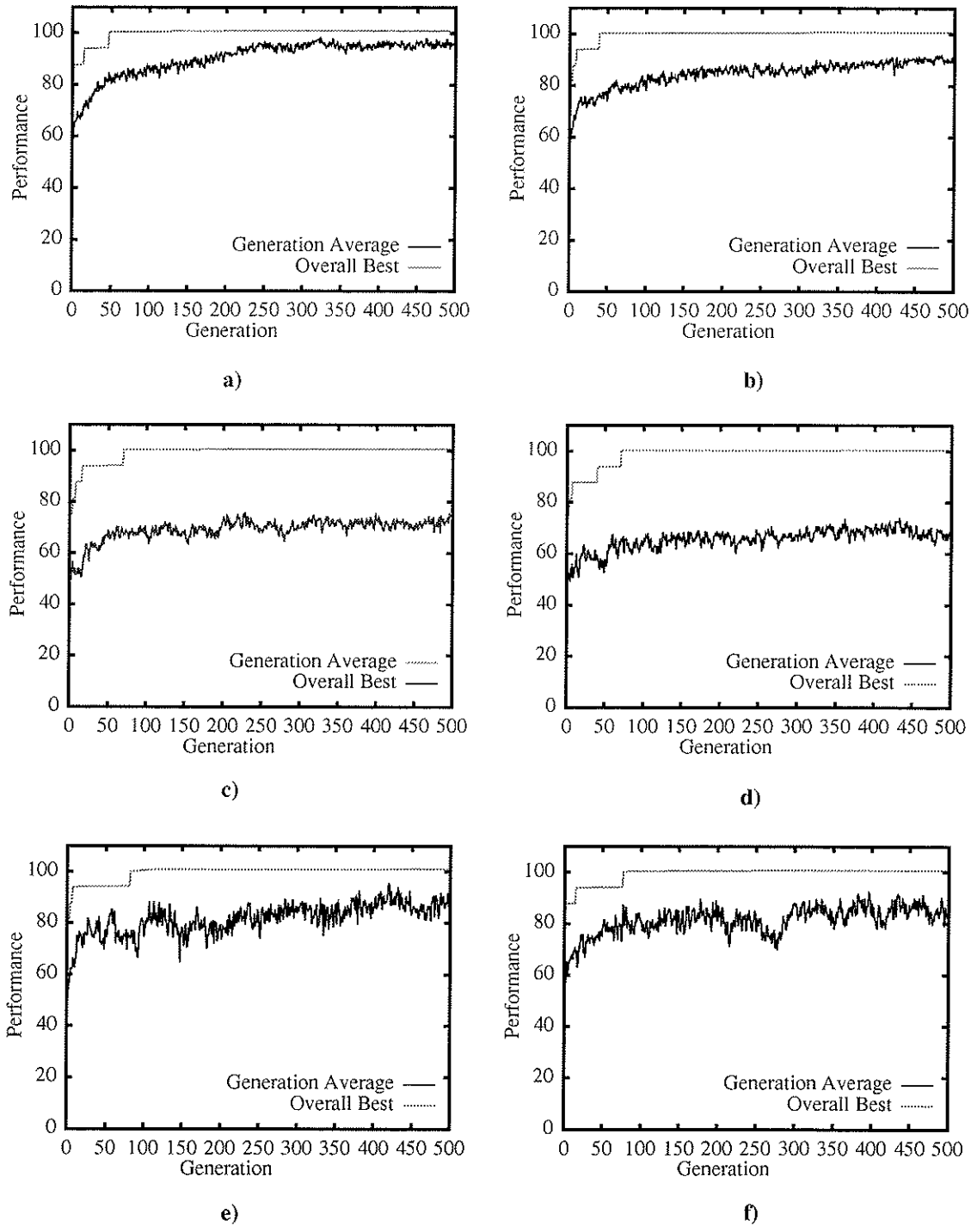


FIGURE 3.14. Eight feature detector experiment – Best and population average for a) most successful run of MTA, b) least successful run of MTA, c) most successful run of CCA, d) least successful run of CCA, e) most successful run of CCAS, and f) least successful run of CCAS

set of weights, there is still enough information for the conjugate gradient algorithm to dis-

criminate amongst the pattern classes. Thus, in these cases the GA is doing little to aid in the performance of the classification task.

These experiments attempted to provide more confidence in the preliminary results presented in earlier sections by running each of the three algorithms multiple times, and they corroborate those initial results. The MTA and CCAS produce networks which are roughly equal in classification ability, while the CCA does not fair as well. We can see from the graphs of the population average that the average network generated by CCAS is superior to that generated by CCA. This definitely supports our supposition that dividing the population up improves the ability of the algorithm to generate better whole networks. If we look at the best overall networks produced by each of the algorithms in the two experiments (Table 3.10 and Table 3.11), we can see that there does not seem to be much difference between the networks produced by MTA and CCAS. CCAS slightly outperforms MTA in the four feature experiment, but this is probably due to the unintentional bias introduced by allowing twice as many trials for the cooperative competitive algorithms. However, if we compare the best networks of CCA and CCAS we can see that CCAS produces the best performing classifiers in both experiments.

3.7 Generalization of the Algorithms

For the final set of experiments we wanted to test the ability of the three algorithms to general-

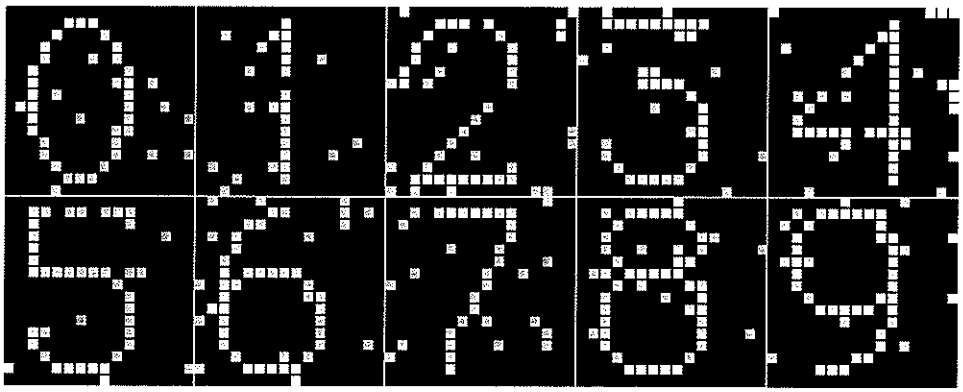


FIGURE 3.15. Example of Geneva character set with noise

ize from the data used for training and evaluating the neural networks to previously unseen data patterns. To do this a new data set was created. It is based on the digits of the Geneva font, except uniform point noise has been added to the images to generate enough unique yet similarly distributed data for the training, validation and testing sets. Figure 3.15 shows an example of a set of digits from the data set. The noise is created by flipping each pixel in the image with a probability of 5%. We generated 120 patterns and split them evenly among the three data sets. Thus, for each network evaluated by the three genetic algorithms, the weights of the second layer are trained for 10 epochs of the conjugate gradient algorithm using the 40 patterns in the training set. At the end of the training phase the fitness of the network is evalu-

ated by testing the performance on the 40 patterns of the validation set. The performance measure of the network on the validation and testing sets is also recorded, so that the generalization of the network can be measured.

Prior to conducting the experiments, a set of informal trials were performed, to ascertain a set of good values for the parameters of the algorithms, including the creep probability and variance, the probability of the real mutation operator, and a good range of values for the random starting weights of the second layer of the network. These experiments suggested that the algorithm converges better if the effect of the creep operator is reduced. Also, the inclusion of the mutation operator increases the performance over the runs when it is absent.

One of the points we have made about the use of random starting weights in the output layer for the conjugate gradient algorithm is that it probably increases the robustness of the feature detectors produced, because the best feature detectors which are passed down through the generations must consistently perform well in spite of the variations in the conjugate gradient training. Indications from some informal experiments using different ranges of the starting weights indicate that the algorithm performs better when using smaller than accepted values for the ranges of the initial weights. For instance, one commonly held rule of thumb is that the starting weights should be distributed uniformly over the range $\pm\frac{2.4}{\text{fan-in}}$ (Haykin, p. 162, 1995). For the output layer of our network the fan-in for each unit is 5, leading to a starting value range of ± 0.48. However, the experiments indicate that the algorithm achieves better population average classification performance with lower values than this. There could be two possible explanations. Either the conjugate gradient algorithm performs better with smaller starting weights, or there is less nondeterminism in the evaluation of the feature detectors and the genetic algorithm is taking advantage of this. For the following experiments we decided on a starting initial weight range of ± 0.2 for the second layer weights.

The parameters which were used for these experiments are shown in Table 3.13. One other

Parameter	Value
$P_{crossover}$	0.6
P_{creep}	0.05
Creep Variance	0.05
$P_{mutation}$	0.01
Mutation range	± 0.2
Proportion selected for next generation	0.25
Population Size	128
Number of Feature Detectors	4
Epochs of CG learning	10
Number of trials each generation	128

TABLE 3.13. Parameters for the generalization experiment

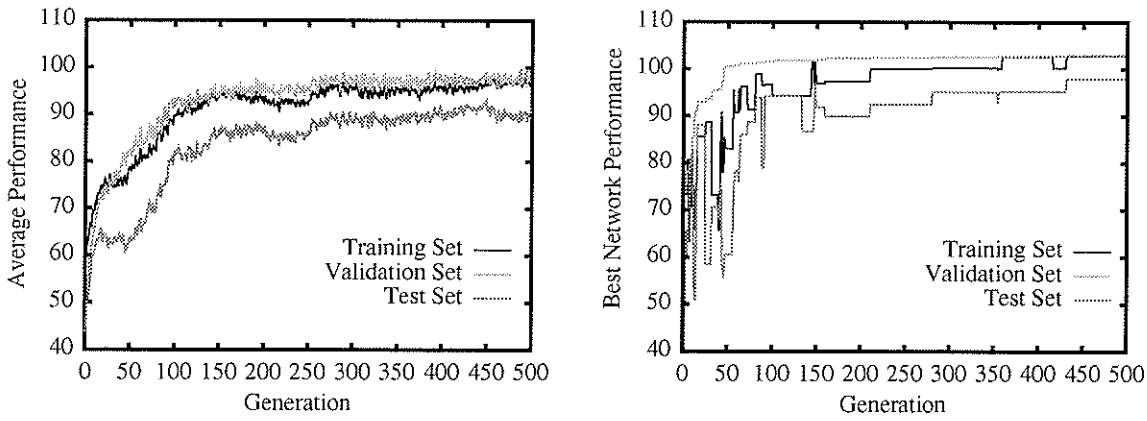
change was made to the algorithm. The selectivity part of the fitness formula was given a weight multiplier of 5. Thus, a network which classifies with very high confidence will have a selectivity component on the order of 2.5 points instead of 0.5. Given that the pattern sets each contain 40 patterns, this is the same number of points awarded a network for correctly classifying a single pattern. Therefore, the reader should understand that the performance scores of the networks are not exactly the same as the percentage correct.

Each algorithm was run ten times to see if there are any consistent trends in the operation of the three algorithms, and their performance on the data sets. While we were looking for an indication that one of these three algorithms would be superior at promoting generalization of the neural network to new pattern sets, the experiment did not provide such a clear result. The three tables (Table 3.14, Table 3.15 and Table 3.16) show the best networks, in terms of fitness, from each algorithm. Each column shows the performance of the network on the three data sets, and each row of the table represents one run of the algorithm. The last two rows of each table give the averages and the standard deviations. Since the GA selects for networks with high performance on the validation set, and since the best networks are chosen because they have a high performance on this data set, we see very little variation in validation set performance from one run to the next. The training set and test set performance shows much more variation, and this is reflected in the standard deviation of those performance values. For the MTA algorithm, the sixth run produced the network capable of best generalizing to the test data set. The graphs of the average performance by generation and the performance of the best overall network by generation from this run are shown in Figure 3.16. Similarly, graphs of the runs which produced the most successful generalizing neural networks from the CCAS and CCA are show in Figure 3.17 and Figure 3.18 respectively. In Figure 3.16 to Figure 3.18, the averaging is over chromosomes in a single run, rather than over separate runs as in Table 3.14 to Table 3.16.

The experiments suggest that all of the three algorithms are comparable at producing neural networks which can generalize from the training and validation sets to the test set. There is a very slight difference in the average performance on the test set of the best networks from each algorithm. MTA has the best average performance, 89.0303, CCAS is slightly lower at, 87.4513, and CCA has the lowest average test set performance at, 86.4506. However, the small differences in the average performance between the three algorithms is swamped by the large deviations from one run to the next. Thus, none of the three algorithms could be said to reliably produce networks which are better at generalization. Improvement of generalization performance appears to depend primarily on other issues such as the size and representativeness of the training and validation data sets.

	Training Set		Validation Set		Test Set	
	Performance	Selectivity	Performance	Selectivity	Performance	Selectivity
1	102.9490	0.5898	103.3160	0.6633	85.3595	0.5719
2	100.1040	0.5208	102.6350	0.5270	82.6339	0.5268
3	102.5750	0.5150	102.7050	0.5411	82.5918	0.5184
4	102.7500	0.5500	102.9110	0.5821	95.1578	0.5316
5	103.2340	0.6468	103.2190	0.6438	95.7094	0.6419
6	102.8400	0.5680	102.9840	0.5968	97.8958	0.5792
7	102.9870	0.5974	103.0310	0.6062	80.1824	0.5365
8	99.9657	0.4931	102.5410	0.5082	92.4609	0.4922
9	102.7690	0.5539	102.7810	0.5563	90.3101	0.5620
10	100.4610	0.5922	103.0650	0.6131	88.0013	0.6003
Average	102.0635	0.5627	102.9188	0.5838	89.0303	0.5561
Std. Dev.	1.3186	0.0460	0.2526	0.0505	6.2303	0.0441

TABLE 3.14. MTA – Performance of the best networks of each run on three data sets

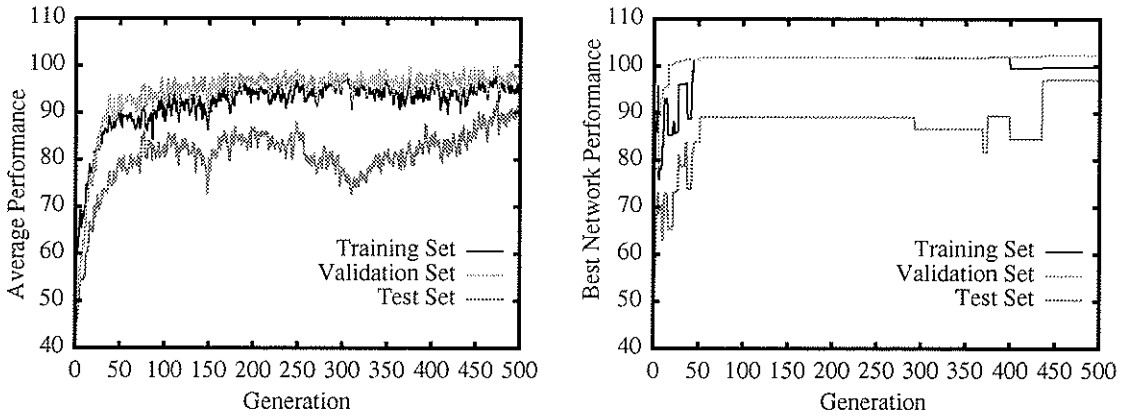


a) b)

FIGURE 3.16. MTA Run #6 – a) generation average performance and b) performance of the best overall network on three data sets

	Training Set		Validation Set		Test Set	
	Performance	Selectivity	Performance	Selectivity	Performance	Selectivity
1	84.2483	0.3497	101.8820	0.3764	91.9210	0.3842
2	99.4624	0.3925	101.9720	0.3945	84.3869	0.3774
3	102.2920	0.4585	102.3860	0.4773	92.2591	0.4518
4	97.0649	0.4130	102.1020	0.4204	84.5459	0.4092
5	99.2364	0.3473	101.7990	0.3598	81.8071	0.3614
6	96.6579	0.3316	101.7550	0.3510	81.7408	0.3482
7	99.6543	0.4309	102.1250	0.4250	89.4744	0.3949
8	99.8307	0.4661	102.3940	0.4787	87.1617	0.4323
9	99.8618	0.4724	102.3860	0.4773	97.0941	0.4188
10	94.1229	0.3246	101.6330	0.3266	84.1221	0.3244
Average	97.2432	0.3986	102.0434	0.4087	87.4513	0.3903
Std. Dev.	5.0852	0.0576	0.2809	0.0562	5.0951	0.0393

TABLE 3.15. CCAS – Performance of the best networks of each run on three data sets

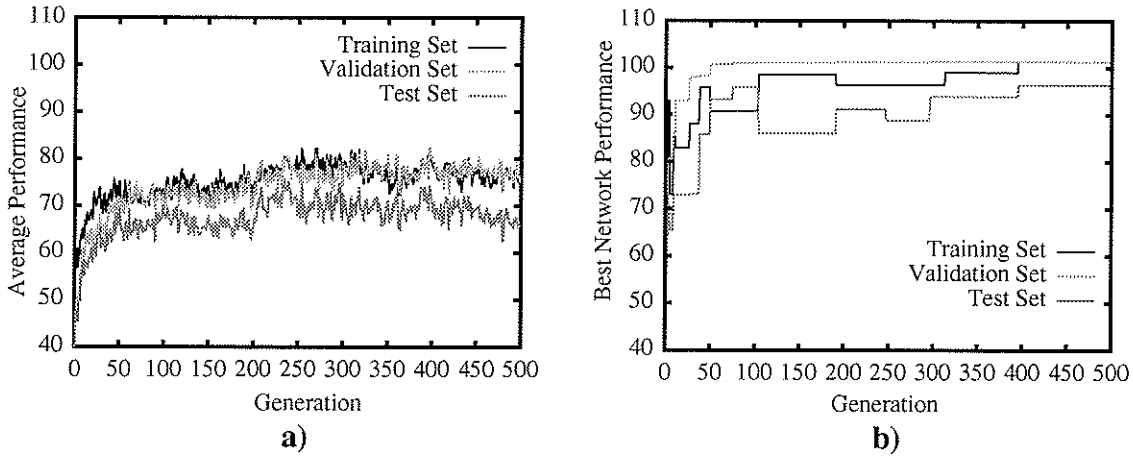


a) b)

FIGURE 3.17. CCAS Run #9 – a) generation average performance and b) performance of the best overall network on three data sets

	Training Set		Validation Set		Test Set	
	Performance	Selectivity	Performance	Selectivity	Performance	Selectivity
1	101.5760	0.3152	101.5900	0.3181	81.5831	0.3166
2	101.4020	0.2803	101.4040	0.2808	96.3792	0.2758
3	96.6159	0.3232	101.6180	0.3237	81.5638	0.3128
4	99.3458	0.3692	101.9020	0.3804	86.9118	0.3824
5	96.5920	0.3184	101.6880	0.3377	84.1142	0.3228
6	93.7586	0.2517	101.1210	0.2242	81.1254	0.2251
7	93.8643	0.2729	101.3960	0.2793	81.2698	0.2540
8	98.6588	0.2318	101.1870	0.2374	88.6826	0.2365
9	93.7700	0.2540	101.2430	0.2486	91.2374	0.2475
10	96.7323	0.3465	101.6650	0.3329	91.6391	0.3278
Average	97.2316	0.2963	101.4814	0.2963	86.4506	0.2901
Std. Dev.	2.9664	0.0449	0.2518	0.0504	5.3938	0.0502

TABLE 3.16. CCA – Performance of the best networks of each run on three data sets



a) b)

FIGURE 3.18. CCA Run #2 – a) generation average performance and b) performance of the best overall network on three data sets

One observation is that the best networks produced by MTA (Table 3.14) provide the highest, most consistent performance on the training data set as evidenced by the low standard deviation. Also MTA consistently has the highest selectivity values on all three data sets. This indicates that co-evolving the different parts of a network in the same chromosome allows them to better cooperate than separate network portions which are evolved in different chromosomes, given that scaling is not a problem. One would expect that the larger the network, and hence the chromosome, the more difficult it would be for the GA to thoroughly search the space to find a good network, but the size of the chromosome does not seem to be a factor in the classification task under consideration. On the other hand the performance on the training data of the best networks produced by the other two algorithms (Table 3.15 and Table 3.16) is not significantly lower than those produced by MTA. This suggests that splitting a problem up among several chromosomes is a viable alternative in cases where network size grows too large to be represented in a single chromosome. A comparison of the best networks produced by each algorithm, averaged over the 10 runs, is shown graphically in the bar chart of Figure 3.19. Once again the results above 100 are due to the selectivity term in the performance measure.

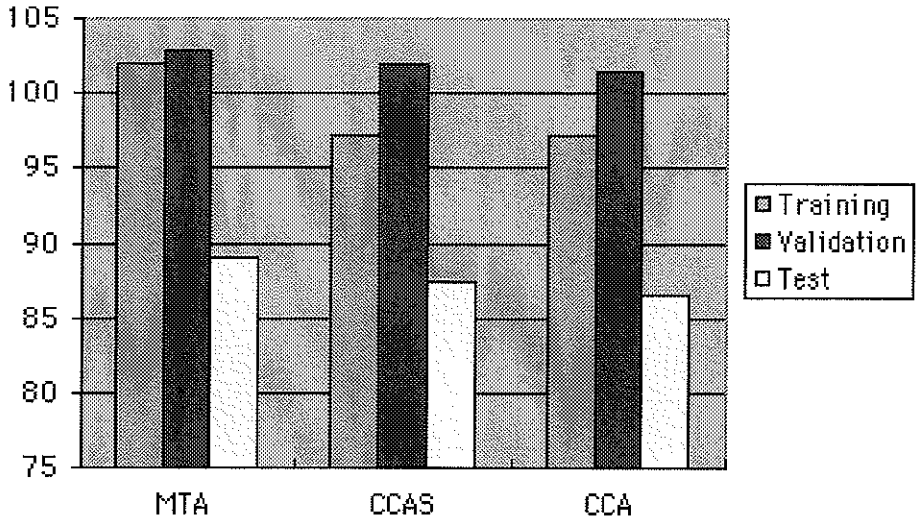


FIGURE 3.19. Average performance of MTA, CCAS, and CCA

Another interesting result is that the CCA has a very low generation average performance, yet it produces networks which classify the data sets almost as well as the best networks generated by the CCAS algorithm. The intention of subdividing the population was to allow the chromosomes to specialize their function within the network, *and* ensure that the chromosomes are always combined in the most complementary way. However, this result suggests that CCA is almost as good a method for evolving feature detectors which can cooperate to form good classifiers, but the correct combination of chromosomes to produce the highest performing networks will be hit upon less often. This reduces the average performance of the trial networks created by this algorithm.

One of the most interesting, and unanticipated, results of this experiment is the difference in performance of the networks on the training and the validation data sets. If one looks at the three graphs of the average performance of the trial networks produced by each algorithm over the course of evolution, one clearly sees a trend where they perform better on the validation set than on the training set. This is in spite of the fact that the conjugate gradient algorithm is attempting to optimize the second layer weights of each algorithm such that it minimizes the error on the training set. It suggests that the GA can arrange the weights of the first layer in such a way that it predisposes the network to perform better on the validation data in spite of the efforts of the conjugate gradient algorithm. This effect is most clear in Figure 3.16 and Figure 3.17, but it is an effect which is very consistent throughout all thirty runs of the three algorithms. This effect indicates that, to better promote generalization in the neural networks produced, both the training set and the validation set should be considered in the fitness function.

CHAPTER 4

Conclusions and Recommendations

4.1 Discussion

In this thesis we have taken ideas regarding combinations of neural networks and evolutionary algorithms, combined them and added some ideas of our own. The initial concept of using the genetic algorithm to determine a single feature detector for a neural network comes from Dellaert and Vandewalle (1994). Our methods make use of several techniques which they used, including a formula for evaluating the classification performance of a network, the weight multiplier term in the chromosome encoding, and their localized selection operator. We have reproduced their results and performed a more thorough comparison of the genetic algorithm method against a standard back-propagation network. To this work we have added the idea of evolving multiple, complementary feature detectors for performing the classification task. We have also incorporated the cooperative co-evolution concept of (Moriarty and Miikkulainen, 1994). While their study focuses on the problem of reinforcement learning for a control problem, we have adapted the method to pattern classification. Using the weights of a single unit in a neural network as the basis for a chromosome helps ensure that the genetic operation of crossover is performing useful manipulations to the components of the neural network. In addition, we have implemented a variation of the method, using sub-populations, which still places the genetic optimization at the level of the neural network hidden unit, but allows the sub-populations to specialize on a particular sub-task of the network's functioning. This is different from the algorithm, as originally implemented, which encourages very general feature detectors capable of being combined in any arrangement to produce a good classifier. The sub-population method seems like a more reasonable way to organize the network so as to avoid duplication and make the best use of the network's limited resources, and the simulations bear this out.

The experiments performed in this thesis have dealt with a particular problem of using GAs to produce a good set of feature detectors for a classification network. The first set of experiments with a single feature detector and a full sized feature map in the second layer indicated

that the genetic algorithm was probably not necessary to solve the font independent classification problem, and the back-propagation algorithm alone could do almost as well. Also, providing more feature detectors to the neural network to perform that classification problem did not significantly improve performance. For this reason, further experiments made use of a simpler network architecture, which presented a more difficult challenge to the genetic algorithm. Rather than use an full feature map, we employed a set of single feature detectors which are correctly positioned to extract enough information from the input image. In this case, the second layer of the network, when trained with a gradient descent algorithm, was able to correctly classify the patterns.

The final set of experiments, presented in section 3.7, were intended to investigate the relative performance of the three versions of the GA at promoting generalization ability of the neural networks produced. The results do not favour any of the algorithms, although the MTA seems to have performed best across all three data sets under consideration. The original criticism of the MTA, and the justification for using the cooperative competitive models, was that on larger problems the chromosome of the MTA would grow too long and the algorithm would not scale well. For the size of network and the task considered here we do not seem to have encountered this difficulty. Experiments with a larger network and a tougher classification problem would be required to notice this effect, but increasing network size and adding more patterns to the data sets increase both the time it takes to construct and to evaluate a single neural network, and this is limited by the computing resources available. One of the most interesting results of the generalization experiments was that using a fitness measure based on the performance of the network on a different pattern set than that used for training can produce networks which are predisposed to perform better on the validation data than on the training data. While this result was not originally expected it is understandable if one considers the vector space interpretation of the function of the hidden units. Each feature detector divides the input space with a hyperplane into two sections, and the goal of learning in the first layer of the network is to find the hyperplanes which best separate the classes of the input patterns. If there is a set of hyperplanes which correctly separate the patterns of the validation data set and not the training set, then the neural network will have a high fitness value, and the GA has no reason to select against such feature detectors, and this fact may detract from the generalization ability of the neural networks. It would be better to find feature detectors which separate both the training and validation data sets.

The combination of genetic algorithm and gradient descent methods in general is a very slow method of optimizing neural networks. The method we have proposed here has some speed advantages over other methods which have been used (Belew et al., 1991; McInerney and Dhawan, 1993), because the gradient descent algorithm is used only to optimize part of the network. Since the gradient descent algorithm executes in the inner loop of the GA, any speed-up is multiplied over the number of generations. In the two layer network, with four units in the hidden layer, and 26 weights in each feature detector, there are 104 weights connecting the input field and the hidden layer. With five weights connected to each output unit, there are a total of 50 weights connecting the hidden layer to the output layer. Thus, the conjugate gradient algorithm is only responsible for optimizing one third of the weights in the network. This impacts the conjugate gradient algorithm in two ways. There are fewer weights for

which to evaluate the error gradient. Also, the number of iterations required for CG to converge to a solution is proportional to the number of parameters being optimized. In other words, reducing the number of weights which CG must optimize improves the convergence rate.

While this method yields some speed improvements over previous methods, it is still quite slow, compared to the process of training a neural network with the back-propagation algorithm alone, especially if the problem being solved involves a large data set. There is a very easy potential speed increase by implementing the algorithm on a parallel computer. As mentioned previously, the genetic algorithm is very easy to parallelize because the evaluation of each neural network is independent of the evaluation of the others, and each evaluation can be distributed to a separate processor with little communication cost. Additionally, if the network is going to be used in an application where 'on-line' learning is not required then, the actual learning time is of less importance than the performance of the final network.

An additional aspect of the use of genetic algorithms to optimize feature detectors is that there is some biological justification for it. The cells in the primary visual cortex have a locally limited, spatial response function, which may develop during the lifetime of the individual, but some neuroscientists believe that the response characteristic of these cells is genetically determined. On the other hand, the use of these low level features by higher levels of the visual system is definitely organized by neural learning during the development of the individual, as shown by experiments conducted on the development of the visual systems in cats (Blakemore, 1990).

4.2 Recommendations for Future Work

There are still some variations on these algorithms which could be explored. In the first set of experiments we looked at using two different selection operators: Goldberg's roulette wheel selection with linear scaling and the local competition selection operator proposed by Dellaert and Vandewalle. While we chose to continue working with the latter operator, because it produced better results in our initial experiments, it might have proved fruitful to look at some different operators, such as the rank-based selection operator (Whitley, 1989). Another potential improvement could be achieved through the incorporation of adaptive creep and mutation operators. These operators work by changing their values according to some measure of the diversity of the population. Thus, they adapt to maintain population diversity to ensure continued genetic search.

The results of the generalization experiment of section 3.7 suggest that another way to improve the algorithm would be to include the network's performance on both the training and validation sets in the genetic algorithm. This will eliminate the effect where the genetic algorithm, in a sense, undoes the work of the conjugate gradient algorithm by selecting fea-

tures which perform well on the validation data set, but do not necessarily aid in the classification of the training data set.

The algorithms presented here used a fixed number of feature detectors, and the fitness function of the algorithm was tailored to encourage the full use of all of the feature detectors. This was initially done out of concern that the GA would not have any information regarding the 'correct' values which the x, and y positions of the feature detectors should take on. However, the best positioning of these feature detectors is evaluated implicitly by the performance of the network. A more sophisticated approach would be to encourage the removal of feature detectors if they are not contributing to the performance of the network. This could be done by adding a reward term to the fitness function. Thus, a network which can classify as well as another with fewer feature detectors would receive a better score. This might solve the problem of determining the number of feature detectors which are needed for a particular problem. If a large number of features are allowed in the representation, and unnecessary features add to the fitness of their respective networks by not participating, then the right number of features will be determined by the algorithm. This leaves the problem of adjusting the fitness function to achieve the proper balance between correct classification and using as few hidden units as possible.

A requirement for a real world neural network classifier is that it be tolerant to shift transformations in the patterns. This is usually implemented in a neural network by incorporating identical feature detectors which are shifted throughout the input field. In the experiments here we have eliminated this problem by our choice of data set. However, it would be easy to incorporate shift invariance into the neural networks by building multiple copies of the feature detectors selected by the genetic algorithm into the constructed networks.

Another issue which we have not addressed is the extension of the algorithms to networks with more than two layers. This is another area where further work could be done. The trend in research in combinations of genetic algorithms appears to be toward more general 'cellular' encoding schemes which describe the architecture of a neural network, and sometimes the initial weights for training as well, using a set of developmental rules or a developmental program. These encodings give the genetic algorithm more leeway to manipulate both the structure and the weights of the neural network. There are indications that, although these representations take longer to evolve useful neural networks, they are in general faster because they remove the trial and error aspect of finding a good architecture (Gruau et al. 1996). Such encodings can be thought of as a superset of the straightforward weight based encoding used here, because, if the representational scheme is flexible enough, the cellular method should be able to represent any particular architecture. However, the flexibility comes at the price of a greatly increased search space. As a general principle, where domain knowledge about a particular problem is available and applicable, then it should be used to constrain the optimization task. In the present case of an image recognition problem, it is known that local feature detectors are used in biology, and they have proven effective in solving other tasks of this nature. Thus, it may be a satisfactory approach to assume a local feature detector structure in the neural network, rather than allowing the genetic algorithm to use any of the possible structures available. Perhaps there is a way of combining the two approaches, by forcing the cellu-

lar neural network description to use local feature detectors in lower layers, while leaving the structure of the higher layers open for the GA to determine.

APPENDIX A

Back-propagation Experimental Results

TABLE A.1. This table shows multiple training runs, using cross-validation, of the single-layer back-propagation network described in section 3.3.3.

Repetition	Training Performance	Testing Performance
0	70.5800	30.5973
1	50.5739	40.5106
2	60.6141	40.5276
3	50.6203	50.5652
4	70.5813	60.5358
5	50.6071	30.5151
6	70.5780	30.5315
7	70.5785	30.3817
8	70.6030	40.4620
9	70.5862	50.5908
10	50.6087	30.6251
11	10.9147	10.8721
12	80.5623	50.6516
13	80.4946	40.4885
14	10.8279	10.8059
15	60.5851	20.5772
16	70.5691	50.5874
17	80.5959	30.4521
18	10.8763	10.8599
19	10.8921	10.8098
20	10.8627	10.9034
21	70.5500	30.6288
22	90.5178	60.4596
23	80.5810	50.5335
24	100.5514	40.5674
25	80.5699	50.5755
26	70.5874	50.4809
27	80.5510	30.3322
28	70.6000	60.6053

Repetition	Training Performance	Testing Performance
29	10.9117	10.9117
30	90.5641	40.5246
31	60.5990	30.4016
32	80.5485	40.6460
33	50.6315	40.5666
34	80.5918	30.5321
35	20.7407	30.6989
36	90.5569	60.5390
37	60.6107	50.7680
38	90.5593	60.6316
39	60.6072	40.5440
40	50.5648	50.4942
41	80.5874	40.5086
42	70.6070	40.7004
43	20.6955	20.6977
44	90.5405	50.5963
45	50.6217	50.6189
46	60.5286	50.3262
47	10.9235	10.8115
48	60.5888	10.5838
49	70.5782	30.5661
50	70.5854	20.6850
51	90.5865	80.5465
52	70.5647	30.7176
53	10.8234	10.8921
54	80.5805	40.4111

Repetition	Training Performance	Testing Performance
55	60.5981	10.6244
56	70.5728	50.4709
57	90.5910	50.4185
58	80.5851	30.5173
59	100.5601	60.5515
60	80.5669	60.4230
61	10.9123	10.9559
62	60.5721	50.6039
63	80.5952	70.4937
64	80.6068	50.6990
65	70.5530	20.5682
66	10.8967	10.9685
67	70.5737	20.5900
68	90.5881	40.4850
69	10.9339	10.9037
70	80.5530	20.6071
71	50.5002	30.6901
72	90.5086	50.4858
73	40.5480	20.4693
74	70.5876	50.5786
75	90.5508	60.5811
76	80.5571	30.5522
77	80.5201	30.3979
78	80.5482	40.5051
79	70.5724	20.6277
80	50.6083	30.6709

Repetition	Training Performance	Testing Performance
81	60.6105	20.8231
82	10.9394	10.9228
83	40.5652	30.5617
84	90.5535	50.5455
85	60.6345	40.5885
86	70.5718	30.6219
87	80.5655	40.5629
88	90.5528	60.4898
89	80.5756	40.6074
90	70.5281	30.3294
91	10.8914	10.8068
92	90.4989	40.6183
93	60.5512	30.4758
94	70.6133	30.5318
95	90.5812	40.5212
96	70.5846	40.6103
97	70.5376	30.5422
98	70.5835	30.6958
99	80.5620	40.5031

TABLE A.2. This table shows multiple training runs, using cross-validation, of the two-layer back-propagation network described in section 3.3.3.

Repetition	Training Performance	Testing Performance
0	100.3858	60.3481
1	100.4641	90.3010
2	100.4641	80.3295
3	100.4601	70.3578
4	100.4786	50.4010
5	100.4434	70.3293
6	100.4635	90.2969
7	100.4360	70.3246
8	100.4603	70.3635
9	100.4680	80.2897
10	100.4522	70.3341
11	100.4599	60.2884
12	100.4772	80.3312
13	100.4582	90.2895
14	100.4717	60.3609
15	100.4719	60.3207
16	100.4900	80.2724
17	100.4445	60.2893
18	100.4767	70.3843
19	100.4498	80.2622
20	100.4765	70.4034
21	100.4787	50.2735
22	100.4664	90.3522
23	100.4689	70.3584
24	100.4832	90.3307
25	100.4859	50.3634
26	100.3975	60.3353
27	100.4502	70.3757
28	100.4571	70.3602
29	100.4728	60.4049
30	100.4722	60.4780
31	100.4545	80.3680
32	100.4560	70.3043
33	100.4777	80.3812
34	100.4403	80.3217
35	100.4201	70.4323
36	100.4910	60.4663
37	100.4673	70.1247
38	100.4571	90.3742
39	100.4768	70.3886
40	100.4529	70.2875
41	100.4667	60.4730
42	100.4679	70.3323
43	100.4878	80.2995
44	100.4776	80.3275
45	100.4593	70.3356
46	100.4482	80.3105
47	100.4674	70.3867
48	100.4917	70.3688

Repetition	Training Performance	Testing Performance
49	100.4565	60.3931
50	100.4633	80.3713
51	100.4421	70.3265
52	100.4726	50.4020
53	100.4606	90.2821
54	100.4799	80.3986
55	100.4730	90.3371
56	100.4435	70.3185
57	100.4816	60.3426
58	100.4754	80.4450
59	100.4633	90.3978
60	100.4781	80.3488
61	100.4475	70.2568
62	100.4340	70.3042
63	100.4857	70.3756
64	100.4809	80.3472
65	100.4710	60.3554
66	100.4612	80.3699
67	100.4892	70.3323
68	100.4523	80.2900
69	100.4779	90.3921
70	100.4543	50.3563
71	100.4738	60.4012
72	100.4845	80.3750
73	100.4256	80.2581
74	100.4423	70.3590

Repetition	Training Performance	Testing Performance
75	100.4585	80.3889
76	100.4776	60.4265
77	100.4237	70.3905
78	100.4606	80.3349
79	100.4337	70.3264
80	100.3949	80.2854
81	100.4593	80.3257
82	100.4672	70.3961
83	100.4603	80.3457
84	100.4809	90.3487
85	100.4597	60.1699
86	100.4684	80.3550
87	100.4706	70.3986
88	100.4512	80.2723
89	100.4722	90.4022
90	100.4384	70.3333
91	100.4364	50.3438
92	100.4444	60.3997
93	100.4362	80.1938
94	100.4675	90.3712
95	100.4613	60.3887
96	100.4662	60.2833
97	100.4245	80.3279
98	100.4544	70.4007
99	100.4772	50.3347

References

Atmar, W. (1994). Notes on the simulation of evolution. *IEEE Transactions on Neural Networks* **5** (1), pp. 130-147.

Balakrishnan, K. and V. Honovar (1995). Evolutionary designs of neural architectures – a preliminary taxonomy and guide to literature. *Technical Report CS-TR-95-01*. Iowa State University. http://www.cs.iastate.edu/~gannadm/Bib/TR95-01.ps

Bishop, C.M. (1995). *Neural Networks for Pattern Recognition*. Oxford: Oxford University Press.

Belew, R.K. (1993). Interposing an ontogenetic model betweeen genetic algorithms and neural networks. In S.J. Hanson, J.D. Cown, C.L. Giles (Eds.) *Advances in Neural Information Processing 5*, pp. 99-106. San Mateo, CA: Morgan Kaufmann.

Belew, R.K., J. McInerney and N. Schraudolph (1991). Evolving networks: using the genetic algorithm with connectionist learning. In C.G. Langton, C. Taylor, J.D. Farmer, S. Rasmussen (Eds.) *Artificial Life II, SFI Studies in the Science of Complexity,* Volume 10, Redwood City, CA: Addison-Wesley.

Blakemore, C. (1990). *Vision: Coding and Efficiency*. Cambridge: Cambridge University Press.

Boers, E.J.W., H. Kuiper, B.L.M. Happel, I.G. Sprinkhuizen-Kuyper (1993). Designing modular artificial neural networks. *Technical Report TR 93-24*. Leiden, The Netherlands: Dept. of Computer Science, Leiden University. http://www.wi.leidenuniv.nl/TechRep/tr93-24.html

Bornholdt, S., and D. Graudenz (1992). General asymmetric neural networks and structure de-

sign by genetic algorithms. *Neural networks* **5**, pp. 327-334.

Brill, F.Z., D.E. Brown and W.N. Martin (1992). Fast genetic selection of features for neural network classifiers. *IEEE Transactions on Neural Networks* **3** (2), 324-328.

Chua, L.O. and T. Roska (1993). The CNN paradigm. *IEEE Transactions on Circuits and Systems–I: Fundamental Theory and Applications* **40** (3), pp. 147-156.

Chua, L.O. and L. Yang (1988a). Cellular neural networks: theory. *IEEE Transactions on Circuits and Systems* **35** (10), pp. 1257-1272.

Chua, L.O. and L. Yang (1988b). Cellular neural networks: applications. *IEEE Transactions on Circuits and Systems* **35** (10), pp. 1273-1290.

Davis, L. (1991). *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.

Dellaert F. and J. Vandewalle (1994). Automatic Design of cellular neural networks by means of genetic algorithms: finding a feature detector. In *Third IEEE International Workshop on Cellular Neural Networks and Their Applications*, pp. 189-194. New Jersey: IEEE.

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

Flotzinger, D., M. Pregenzer and G. Pfurtscheller (1994). Feature selection with distinction sensitive learning vector quantisation and genetic algorithms. In *Proceedings of the 1994 IEEE International Conference on Neural Networks*, Volume 6, pp. 3448-3451. New Jersey: IEEE.

Fogel, D.B. (1993). Using evolutionary programming to create neural networks that are capable of playing tic-tac-toe. In *Proceedings of the IEEE International Conference on Neural Networks*, pp. 875-880. New Jersey: IEEE.

Fogel, D.B. (1994). An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks* **5** (1), pp. 3-14.

Fogel, D.B. (1995). *Evolutionary Computation*. New York: IEEE Press.

Fogel, D.B., L.J. Fogel and V.W. Porto (1990). Evolutionary programming for training neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, Volume 1, pp. 601-605. New Jersey: IEEE.

Geman, S. and E. Geman (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Learning* **6**, pp. 721-741.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading, MA: Addison-Wesley.

Goldberg, D.E. (1991). Real coded genetic algorithms, virtual alphabets, and blocking. *Complex Systems* **5** (2), 139-168.

Grefenstette, J.J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics* **16** (1), 122-128.

Gruau, F. (1992). Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In L.D. Whitley and J.D. Schaffer (Eds.) *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 55-72. Los Alamitos, CA: IEEE Computer Society Press.

Gruau, F., D. Whitley and L. Pyeatt (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. *NeuroCOLT Technical Report Series NC-TR-96-048.* Surrey, England: Royal Holloway University of London. ftp://platon.dcs.rhbnc.ac.uk/pub/neurocolt/tech_reports/nc-tr-96-048.ps.Z.

Harp, S.A and T. Samad (1991). Genetic synthesis of neural network architecture. In L. Davis (Ed.), *Handbook of Genetic Algorithms*, pp. 202-221. New York: Van Nostrand Reinhold.

Haykin, S. (1994) *Neural Networks: A Comprehensive Foundation.* Englewood Cliffs, NJ: Macmillan.

Hertz, J., A. Krogh and R.G. Palmer (1991). *Introduction to the Theory of Neural Computation.* Redwood City, CA: Addison-Wesley.

Hinton, G.E. (1987). Connectionist learning procedures. *Technical Report CMU-CS-87-115.* Pittsburgh, PA: Carnegie-Mellon University.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press.

Kirkpatrick, S., C.D. Gelatt, Jr. and M.P. Vecchi (1983). Optimization by simulated annealing. *Science* **220**, pp. 671-680.

Kitano, H. (1990). Empirical studies on the utility of genetic algorithms for training and designing of neural networks. *Technical Report CMU-CMT-90-120.* Pittsburgh, PA: Center for Machine Translation, Carnegie-Mellon University.

Kitano, H. (1992). Neurogenetic learning: an integrated method of designing and training neural networks using genetic algorithms. *Technical Report CMU-CMT-92-134.* Pittsburgh, PA: Center for Machine Translation, Carnegie-Mellon University.

Koehn, P. (1994). Combining genetic algorithms and neural networks: the encoding problem. *M.Sc. Thesis*. Knoxville: University of Tennessee. ftp://archive.cis.ohio-state.edu/pub/neuroprose/koehn.encoding.ps.Z

Koza, J.R. (1992). *Genetic Programming*. Cambridge, MA:MIT Press.

Koza, J.R. (1993). Simultaneous discovery of detectors and a way of using the detectors via genetic programming. In *Proceedings of the IEEE International Conference on Neural Networks*, Volume 3, pp. 1794-1801. New Jersey:IEEE.

Koza, J.R., and J.P. Rice (1991). In *Proceedings of the International Joint Conference on Neural Networks*, Volume 2, pp. 397-404. New Jersey: IEEE.

Kozek, T., T. Roska and L.O. Chua (1993). Genetic algorithm for CNN template learning. *IEEE Transactions on Circuits and Systems–I: Fundamental Theory and Applications* **40** (6), pp. 392-402.

LeCun, Y., B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard and L.D. Jackel (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation* **1**, pp. 541-551.

Maniezzo, V. (1994). Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on Neural Networks* **5** (1), pp. 39-53.

McDonnell, J.R., and D. Waagen (1993). Evolving neural connectivity. In *IEEE International Conference on Neural Networks*, pp. 863-868. New Jersey: IEEE.

McInerney, M. and A.P. Dhawan (1993). Use of genetic algorithms with back propagation in training of feed-forward neural networks. In *IEEE International Conference on Neural Networks*, Volume 1, pp. 203-208. New Jersey: IEEE.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Third Edition. New York: Springer-Verlag.

Minsky, M.L. and S.A. Papert (1988). *Perceptrons*. Expanded Edition. Cambridge, MA:MIT Press.

Moriarty, D.E. and R. Miikkulainen (1994). Efficient learning through symbiotic evolution. *Technical Report AI94-224*. Austin, TX: University of Texas.

Porto, V.W., D.B. Fogel and L.J. Fogel (1995). Alternative neural network training methods. *IEEE Expert* **10** (3), pp. 16-22.

Rudolph, G. (1994). Convergence analysis of canonical genetic algorithms. *IEEE Transactions*

on Neural Networks **5** (1), 96-101.

Rumelhart, D.E., G.E. Hinton and R.J. Williams (1986a). Learning internal representations by error propagation. In D.E. Rumelhart, J.L. McLelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1, pp. 318-362. Cambridge, MA: MIT Press.

Rumelhart, D.E., J.L. McLelland and the PDP Research Group (1986b). *Parallel Distributed Processing: Explorations in the Microstructure of Cogntion, Volume 1: Foundations.* Cambridge: MIT Press.

Schaffer, J.D., R.A. Caruana and L.J. Eshelman (1990). Using genetic search to exploit the emergent behavior of neural networks. *Physica D* **42** 244-248.

Schaffer, J.D., L.D. Whitley and L.J. Eshelman (1992). Combinations of genetic algorithms and neural networks: a survey of the state of the art. In L.D. Whitley and J.D. Schaffer (Eds.) *International Workshop on Combinations of Genetic Algorithms and Neural Networks,* pp. 1-37. Los Alamitos, CA: IEEE Computer Society Press.

Smith, R.E., D.E. Goldberg and J.A. Earickson (1994). SGA-C: A C-language implementation of a simple genetic algorithm. *Technical Report TCGA Report No. 91002.* Tuscaloosa, AL: The University of Alabama. ftp://Alife.santafe.edu/pub/USER-AREA/EC/GA/src/sga-c.tar.gz

Tamburino, L.A., M.A. Zmuda, and M. M. Riziki (1995) Generating pattern recognition systems using evolutionary learning. *IEEE Expert* **10** (4), 63-68.

van Camp, D., T. Plate and G.E. Hinton (1992). *The Xerion Neural Network Simulator and Documentation.* Toronto: Department of Computer Science, University of Toronto.

Weigend, A.S., D.E. Rumelhart and B.A. Huberman (1991). Generalization by weight elimination with application to forecasting. In R.P. Lippmann, J.E. Moody and D.S. Touretzky (Eds.) *Advances in Neural Information Processing 3*, pp. 875-882. San Mateo, CA: Morgan Kaufmann.

Weiss, S. M. and K.A. Kulikowski (1991). *Computer Systems that Learn: Classification and Predicition Methods from Statistics, Neural Nets, Machine Learning and Expert Systems.* San Mateo, CA: Morgan Kauffman.

Whitley, D. (1989). The GENITOR algorithm and selective pressure: Why rank based allocation of reproductive trials is best. In J.D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, pp. 116-121. San Mateo, CA: Morgan Kauffman.

Whitley, D. and T. Hanson (1989). Optimizing neural networks using faster, more accurate ge-

netic search. In J.D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, pp. 391-396. San Mateo, CA: Morgan Kauffman.

Yao, X. (1993). A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems* **8**, pp. 539-567.