University of Stirling

Peter J.B. Hancock

Coding strategies for genetic algorithms and neural nets

Centre for Cognitive and Computational Neuroscience
Department of Computing Science and Mathematics

Thesis submitted for the degree of PhD
October 1992

Contents

Acknowledgements

This work has been carried out while employed on two SERC Image Interpretation Initiative grants. My thanks to Leslie Smith, Bill Phillips and latterly especially Roger Watt for allowing me to pursue my own interests to a considerable extent. As my supervisor for this work, Leslie Smith has been ever-willing to give advice, and read and comment on drafts of text at short notice.

The Centre for Cognitive and Computational Neuroscience has provided a wonderful working environment and friendly colleagues. Among these, Roland Baddeley should be singled out as having offered many useful insights, and some of the work in Chapter 2 is the result of a direct collaboration with him.

I believe the work reported in Chapter 5 benefitted from discussions with Nick Radcliffe. He may disagree, since our conclusions differ somewhat!

The departmental computing officer, Sam Nelson, has answered many technical queries, and let me run GAs on most of his computers for several months.

For Laura, who has for too long put up with me spending most evenings in front of a computer.

Declaration

I declare this to be my own work, except for the collaboration with Roland Baddeley that is acknowledged in Chapter 2.

Abstract

The interaction between coding and learning rules in neural nets (NNs), and between coding and genetic operators in genetic algorithms (GAs) is discussed. The underlying principle advocated is that similar things in "the world" should have similar codes. Similarity metrics are suggested for the coding of images and numerical quantities in neural nets, and for the coding of neural network structures in genetic algorithms.

A principal component analysis of natural images yields receptive fields resembling horizontal and vertical edge and bar detectors. The orientation sensitivity of the "bar detector" components is found to match a psychophysical model, suggesting that the brain may make some use of principal components in its visual processing.

Experiments are reported on the effects of different input and output codings on the accuracy of neural nets handling numeric data. It is found that simple analogue and interpolation codes are most successful. Experiments on the coding of image data demonstrate the sensitivity of final performance to the internal structure of the net.

The interaction between the coding of the target problem and reproduction operators of mutation and recombination in GAs are discussed and illustrated. The possibilities for using GAs to adapt aspects of NNs are considered. The permutation problem, which affects attempts to use GAs both to train net weights and adapt net structures, is illustrated and methods to reduce it suggested. Empirical tests using a simulated net design problem to reduce evaluation times indicate that the permutation problem may not be as severe as has been thought, but suggest the utility of a sorting recombination operator, that matches hidden units according to the number of connections they have in common.

A number of experiments using GAs to design network structures are reported, both to specify a net to be trained from random weights, and to prune a pre-trained net. Three different coding methods are tried, and various sorting recombination operators evaluated. The results indicate that appropriate sorting can be beneficial, but the effects are problem-dependent. It is shown that the GA tends to overfit the net to the particular set of test criteria, to the possible detriment of wider generalisation ability. A method of testing the ability of a GA to make progress in the presence of noise, by adding a penalty flag, is described.

Glossary

ABS	Artola, Brőcher and Singer (learning rule)
ES	EvolutionStrategie
GA	genetic algorithm
GANNET	genetic algorithm for neural nets
LoG	Laplacian of Gaussian
MDP	minimal deceptive problem
NN	neural net
RBF	radial basis function
RF	receptive field
SD	standard deviation
SUS	stochastic universal sampling
WYTIWYG	what you test is what you get
α	momentum (Backprop)
η	learning rate (all nets)
p_b	probability of crossover in between unit definition boundaries
p_m	mutation probability
p_t	probability of transmitting matched unit to child
p_u	probability of crossover at a unit definition boundary
p_x	recombination (crossover) probability

Chapter 1

Introduction

1.1 Introduction

This thesis is concerned with two areas of computing that are inspired by natural, biological systems. The first is neural nets (NNs), a versatile form of computation that uses a large number of simple computing elements in parallel. This resembles the style of computation found in biological central nervous systems, though few of the nets to be discussed here have any claim to biological plausibility. The second is genetic algorithms (GAs), which borrow from the ideas of evolution and natural selection to give a general purpose optimisation method. The main application of these algorithms to be explored here is the design of neural nets. However it will be argued that at the heart of any application of either NNs or GAs lies a similar problem: representation of the data. The underlying strategy that will be advocated also applies to both. It may be stated simply: things which are similar in “the world” should have similar internal representations.

In itself this claim is not novel, the problem comes in deciding what measures of similarity to use. The possible list is endless: position, size, speed, temperature, danger, texture, shape, function, colour... Mappings may typically be required between different metrics. For instance, animals with forward-facing eyes are usually dangerous. The ability to generalise from one potential predator to another has obvious survival value. While high-level concepts such as danger are beyond the scope of this work, explorations of similarity metrics will be a recurrent theme.

Chapter 2 considers the interaction between learning rules and representation in nets. An important class of nets are capable of re-coding input data entirely by self-organisation, i.e. according to the architectures and algorithms used. These self-organised codings are potentially useful for further processing. There is also a possible philosophical advantage. Searle (1980) has argued that a computer program can have no semantics, any semblance of such being derived from the programmer. At the heart of all programs is the representation of data: imposed by and having “meaning” only to the programmer. If the system is capable of forming its own representations of its world and then acting upon them, it may start to generate a semantics independent of the programmer. Unfortunately, the semantics thus produced may well be obscure to an outside observer without detailed analysis.

Chapter 3 looks at ways to improve the performance of nets, both by the use of appropriate coding strategies and by fitting the internal structure of the net to the problem. One way to produce appropriate net structures is by evolving them, and the rest of the thesis concerns genetic algorithms and their application to neural nets. GAs offer a naturally appealing approach to the net design problem, biological brains providing an existence proof of the power of evolution to design nets. While both current neural nets and genetic algorithms are extremely simplified versions of their natural analogues, the emergence of quite sophisticated behaviour in evolved networks, such as the ability to learn to find food (or, at least, simulated food suitable for simulated animals) e.g. (Ackley & Littman, 1989; Cecconi & Parisi, 1990; Jefferson *et al.*, 1990), suggests that the analogy is valuable. The discontinuous nature of the parameter space, coupled with a noisy evaluation function argues against the possibility of using traditional hill-climbing techniques. That nature took around three thousand million years to develop mammalian brains, in a highly parallel processor, gives some hint as to the possible drawbacks of the method. However, there are benefits to be had for those with sufficient patience: as a foretaste, figure 1.1 shows a comparison between fully connected nets and a genetically designed one on a face recognition task. There are dramatic improvements in both test performance and training time. The caveats will be explained in chapter 6.

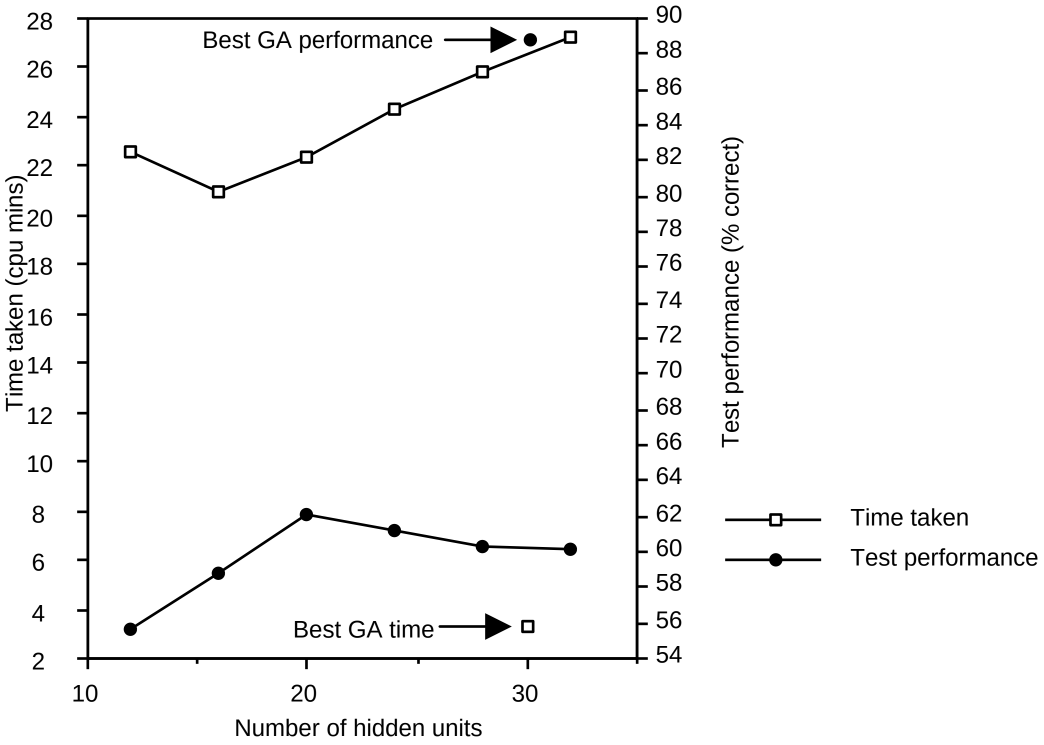


Figure 1.1: Effects of varying the number of hidden units on test performance and training time for a face recognition task. For comparison, the best performance obtained from a GA-designed net, which had 30 hidden units but sparse connectivity, is also shown.

One of the intriguing aspects of working with GAs is the insight they provide into the possible workings of natural evolution. There are still a remarkable number of people, a recent president of the USA among them, who seriously question the theory of evolution and prefer to believe, if not in a single creation event, then in "scientific creationism". Otherwise sound scientists argue that it is simply not conceivable that anything as complex as ourselves arrived "by chance", that it as likely as a tornado in a junkyard producing a jumbo jet. Richard Dawkins attempted to demonstrate the fallacy of this in his book "The Blind Watchmaker" (Dawkins, 1986). Anyone who has worked with GAs on a sufficiently complex problem will need no convincing. A striking example is given by the work of Ray (1992), who has devised a form of replicating computer programs that compete for resources of cpu time and memory. Astonishingly complex programs evolve, with routines that would be useless without every bit in place. It seems such code segments would have no selective advantage until they were complete, indeed they might be lethal, just the kind of thing that creationists claim "disproves" evolution. Manifestly, these did evolve. As with nets, the key to harnessing the power of GAs is the way the problem is coded: the coding for Ray's evolving programs took months of work.

1.2 Structure of the thesis

After an introduction to the terminology used, the next chapter discusses the interaction between learning algorithms and coding in neural nets. It concludes with an experiment that uses a net to produce statistically optimal codes of natural images. Chapter 3 is about the ways that changing input-output coding and internal structure can affect the performance of mapping nets and reports a number of experiments on the coding of numeric data and images. Chapter 4 is an introduction to genetic algorithms, illustrated with a number of demonstrations of the effects of coding on simple problems. Chapter 5 discusses combinations of genetic algorithms and neural nets, with particular reference to the permutation problem. This is a consequence of the fact that functionally identical networks may have many different genetic representations. Evidence is presented that suggests the problem may not in practice be as severe as has been supposed. Chapter 6 reports experiments on the genetic design of neural nets, with further exploration of ways to reduce the permutation problem in practice. Chapter 7 presents a summary and conclusions. The work is in what is hoped is a logical, rather than a chronological order, thus most of the work reported in chapter 6 predates the experimental work of chapter 5.

Chapter 2

Coding in neural nets

With the increasing availability of high speed parallel supercomputers, and new learning procedures, the possibility of solving impossible problems is becoming a reality.
Mitchell Wheat, “Neural Networks”, Computing, 14/2/91

2.1 Introduction

Neural nets need far less introduction now than they would have when this work was begun in 1988: the intervening years have seen a plethora of books on the subject. Unfortunately there remains a diversity of terminology, so it will be necessary to outline that adopted here.

A net consists of a number of interconnected processing elements. Simulated processing elements will be referred to here as units, the term "cell" being reserved for discussions of biology. The connection from unit i to unit j has a strength, or weight, w_{ij}. Algorithms that specify the modification of the weights are referred to as learning rules. The nets described here are mostly feedforward designs, with disjoint sets of input and output units, and possibly a set of "hidden" units in between. A prime ambiguity in the neural net literature is the meaning of layer. It will be taken here to mean a layer of units. Thus a three-layer net has an input, output and one hidden layer.

One possible classification of nets is into three types: associative, auto-associative and self-organising feature extractors. As with many classifications the borders are rather ill-defined, for instance counterpropagation nets (Hecht-Nielsen, 1987) have a purely self-organised hidden layer, which is then used to form associations. Indeed, all nets that have hidden units are to some extent self-organising: the distinction intended here is that self-organising nets simply recode the input data as defined by their internal rules, while the coding in the hidden layer of an associative net is specifically that required for the taught mapping. The bulk of this chapter concerns self-organised codings, while input-output coding for associative nets forms the subject of chapter 3. Auto-associative nets are not considered in any detail.

The most common learning rule for associative nets is the delta rule (Widrow & Hoff, 1960), which performs a gradient descent on the sum squared error. This has been generalised to nets of more than two layers by a number of workers, but most influentially by Rumelhart et al (1986). This generalised delta rule, often called Backpropagation or just Backprop, is probably used in more neural net simulations than any other algorithm. It has the virtue of being simple and reasonably robust, but is often painfully slow. There are two major variants: "online", where the weights are updated after each input, and "batch", where weights are updated only after the complete training set has been presented. Arguments rage on bulletin boards about which method works better, confused by the fact that the online algorithm can be significantly affected by the order of the input data. The author has obtained a factor of two difference in convergence time just by changing the order of presentation, with the batch algorithm in between the two online results. While favourite "toy" problems such as parity and encoder may benefit from batch update, the online algorithm is usually preferred for "real" problems, where a fair degree of correlation between different inputs can be expected. Since many of the inputs should therefore move the weights in similar directions, it makes sense to treat them all separately, many small steps being more able to follow the gradient than one big one. Fogelman Soulie (1991) suggests that the online algorithm can be two orders of magnitude faster than batch on real-world problems.

Each cycle of presenting all the training patterns is traditionally known as an epoch, a name that reflects the time it may take. Many methods of speeding up the convergence of Backprop have been suggested. One used in some of the work reported here is "Quickprop" (Fahlman, 1989), which is, frankly, an educated collection of hacks allied to a crude line search algorithm. Nevertheless, the reduction in training time can be dramatic, as much as an order of magnitude. Unfortunately it is most impressive on toy problems, and may fail completely with difficult real data. It appears quite usual that the faster algorithms are less robust: they may converge rapidly, but not always to the right minimum. This causes difficulty in comparing results: are 8 fast runs and 2 failures better or worse than 10 moderate runs? Most of the work in following chapters therefore uses simple Backprop, with just two parameters: learning rate η and momentum α.

2.2 Learning rules and coding

The earliest work done by the author at Stirling was a study of some of the computational properties of a learning rule proposed by Stent (1973) and later inferred from neurobiological experiments by Rauschecker and Singer (1979). This rule is a simple modification of Hebb's associative rule (1949), to allow weight decrease if the post-synaptic cell is active while the presynaptic cell is silent. The initial aim was to investigate the applicability of such a rule to feed-forward associative nets. The delta rule is commonly used to train such associative net models. Although it appears to give similar results to the Rescorla-Wagner rule inferred for animal conditioning (Rescorla & Wagner, 1972; Sutton & Barto, 1981), there is still no clear biological support for such a synaptic modification rule. It is therefore of interest to consider the capabilities of rules for which there is biological evidence. It was apparent that a unit using the Stent rule cannot learn to be off, since weight changes occur only when it is active. However, if two such units are set up to be mutually inhibitory, then it is possible to train one to be off by training the other to be on (W.A. Phillips, personal communication). Use of the simpler, biological learning rule thus requires a change to a more redundant coding.

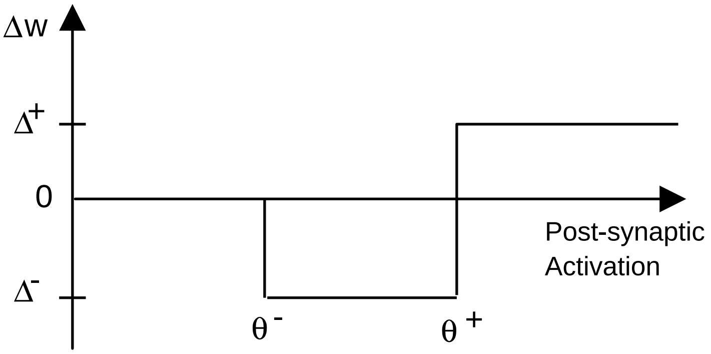
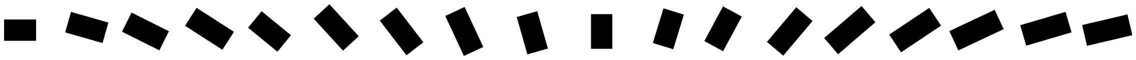


Figure 2.1: Schematic behaviour of ABS learning rule: weight change is negative if activation is above threshold θ^-, positive if above θ^+.

The same technique of opponent units extends the computational capabilities of a more complex learning rule. Artola et al (1990) reported a dual-threshold rule. With an active pre-synaptic cell, there is no change in weight until the post-synaptic cell activation exceeds a threshold, whereupon the weight *decreases*. When a higher threshold is exceeded, the weight change becomes positive, figure 2.1. The biological function of the rule is unclear, but Hancock et al (1991a) suggested a possible role as an error-correction rule. This places the rule in a simple associative net, with binary inputs and outputs. When the first input-output pairs are presented, the rule functions like the Hebb rule, forming associative connections between appropriate units. As the loading increases, some output units may start to respond when, according to the target stimulus, they should not. It is postulated that this unwanted activation takes the unit into the decrement region of the weight change rule, but without the output signal, does not reach the level required for weight increase. This then specifically reduces the weights that are responsible for the false positive output, giving a simple form of error correction. The correction is incomplete, because there is no way to address misses (or false negatives). Introducing opponent output units allows misses to be corrected as well. If the output of a unit is too low, this can be attributed partly to its own weights being low, and partly because of activation in its opponent being too high. The learning rule can reduce the excess activation in the opponent unit, allowing the low output to increase, and thus correcting misses. Such a system was shown to have a performance approaching that of the fully error-correcting delta rule (Hancock *et al.*, 1991b).

Stimuli



Cluster units	90	80	70	60	50	40	30	20	10	0	10	20	30	40	50	60	70	80
1	0	0	0.1	0.4	0.6	0.5	0.6	0.4	0.4	0.3	0.1	0	0	0	0	0	0	0
2	0	0	0	0.2	0.4	0.5	0.7	0.6	0.6	0.7	0.5	0.4	0.3	0.1	0	0	0	0
3	0	0	0	0	0	0.3	0.6	0.6	0.7	0.8	0.7	0.7	0.7	0.5	0.2	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0.1	0.2	0.5	0.5	0.6	0.4	0.1	0
5	0.3	0	0	0	0	0	0	0	0	0	0	0	0	0.1	0.4	0.7	0.7	0.5
6	0.4	0.1	0	0	0	0	0	0	0	0	0	0	0	0	0.1	0.4	0.6	0.6
7	0.4	0.5	0.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.2
8	0	0.5	0.9	0.9	0.6	0.2	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2.2: Responses of a cluster of eight units after 40 presentations of a set of bars at various orientations, adjusting weights with the Stent rule

This pairwise inhibitory arrangement seems too specific to be biological, but mutual inhibition between groups of cells is quite common in real nervous systems. Interesting results can be obtained if the inhibition is non-isotropic, such that nearby units are less inhibited than those slightly further away, again a model supported by biology. Figure 2.2 shows the responses of 8 units after 40 presentations of a training set of bars at various orientations across an 8x8 input matrix. The net resembles a simplified version of von der Malsburg's (1973) model of the formation of orientation selective cell in visual cortex. The 8 output units are arranged in a ring, interconnected such that each mildly excites its nearest neighbour, and inhibits those further away. Adaptive connections from the input array initially have small random weights. Within 5-10 epochs, the net learns a coding such that each unit is most sensitive to a bar of a particular orientation, in between those of its two neighbours. This is a genuinely *distributed representation*: there are more bar orientations than units, but each is coded uniquely, though more than one unit output is needed to recover the orientation. Such coding strategies seem widespread in nature, the most familiar probably being the coding of colour by our own eyes. With only three receptor types, we are able to perceive a huge number of different shades. The merits of this type of coding for input and output of nets will be considered in the following chapter.

A learning rule introduced by Oja (1982) is able to extract the principal component (PC) of its input data. This is the linear descriptor that gives most information about the data, assuming a normal distribution and that there is no knowledge of higher order statistics. Given an input probability distribution in the shape of an ellipse, principal component analysis (PCA) would return the long axis as the first component. Subsequent components are mutually orthogonal, each contributing the most additional information to that conveyed by its predecessors. The rule has been extended by several workers to multiple units. Oja himself (Oja, 1989) and Plumbley and Fallside (1989) have proposed multiple unit models which converge, not to the PCs, but to mutually orthogonal vectors which span the same space. The total information conveyed is therefore the same as the real PCs, but the particular vectors produced, and the output variance of each unit, will differ from run to run. Sanger (1989) developed a model that does converge to the principal components. The difference is that successive components have a variance no greater than, and usually less than their predecessor, and that, since the PCs are unique, the results are consistent between runs.

For a single linear unit, Oja's rule is (Oja, 1982):

$$\begin{aligned} y &= \sum_{i=1}^{N} x_i w_i \\ \Delta w_i &= \eta y (x_i - y w_i) \end{aligned}$$

Here, $\vec{x} = (x_1, .., x_N)$ is a real valued input, y is the output signal, w_i is the strength of the connection (or weight) from input unit i to the single output unit, and η is the learning rate. This produces a weight vector corresponding to the eigenvector of the correlation matrix of all the inputs which has maximal eigenvalue: the principal component of the input data. The weight vector also tends to unit length. For multiple units it is necessary to run the connections backwards, which limits biological plausibility, though Plumbley and Fallside (1989) speculate that a feed-forward variant may be possible. Intuitively, each unit attempts to account for as much as possible of the input signal: by subtracting each unit's weighted output from the input, each unit is led to try and account for that part of the input not covered by other units. Formally, for N units:

$$\Delta w_{ij} = \eta y_j (x_i - \sum_{k=1}^{N} y_k w_{ik})$$

Sanger's variation on this is to impose a hierarchy on the units, such that each only gets that part of the input signal which is unaccounted for by its predecessors. The first unit thus behaves just as if it were the only one, and extracts the first PC. The second then extracts the PC of what is left, which amounts to the second PC of the whole data set, and so on:

$$\Delta w_{ij} = \eta y_j (x_i - \sum_{k=1}^{j} y_k w_{ik})$$

These principal component analysers may be used to reduce the volume of data, for subsequent processing by other systems. An example of their use as pre-processors for a Backprop net classifier is given in section 3.5. Their ability to handle large volumes of data permitted a principal component analysis of natural images, which is described in the following section. These experiments suggest that principal components may play some role in our visual systems.

Some neural net models have binary outputs, others are continuously varying, often in the range 0-1. Relatively few try to model the spiking behaviour of real nerve cells. Early studies of the coding used by nerve cells suggested a simple rate code: the faster the firing, the stronger the signal (Adrian, 1928). This is the model assumed by the use of continuously varying outputs in simulations. However, it is increasingly becoming clear that there is potentially information both within the pattern of spikes from one cell, and in the pattern of firing across cells. In a series of experiments, Optican and Richmond have used principal component analyses to show that there is information in the pattern of firing of cells in the primary visual cortex and inferior temporal cortex (Optican & Richmond, 1987; Richmond *et al.*, 1990). Chung et al (1970) suggested the term "multiple meanings" to describe the ability of frog optic nerve cells to carry different information in their frequency and pattern of firing. Such subtleties are beyond the models considered in this work.

A further subtlety that is not considered here is the possible significance of synchronised firing of cells. Gray et al (1990) first demonstrated synchronised firing in cat primary visual cortex. It appears that the oscillations may have a role in feature binding. Cells that are responding to a coherently moving stimulus (a bar of light) fire in phase. If the bar is split into two independently moving pieces, the relevant cortical cells also separate into two independently phase-locked groups. Such behaviour gives support to a model of feature binding proposed by von der Malsburg (1985), which proposes that feature cells are linked by the action of rapidly switching weights. The question of how features are linked in neural models is important, unresolved, and not discussed further here.

2.3 Principal components of natural images

2.3.1 Introduction

This work, conducted in collaboration with Roland Baddeley, arose from posing the question: suppose primary visual cortex is performing a principal components analysis on its input, what would the resultant receptive fields look like? (Baddeley & Hancock, 1991; Hancock *et al.*, 1992) Ever since Hubel and Wiesel's initial report of the "bar-detector" behaviour of cells in primary visual cortex (Hubel & Wiesel, 1962), there have been attempts to explain both what they are doing and how they come about. Von der Malsburg (1973) used a competitive net with an input of bars of activity at various orientations to produce orientation sensitive units, see also figure 2.2. One problem with this model is that a normal visual diet does not actually consist of helpfully oriented bars (except, perhaps, for cage-raised laboratory animals!). Barrow (1987) pre-processed natural images with centre-surround filters intended to mimic the behaviour of retinal and lateral geniculate cells, and found that competitive learning still produced orientation-sensitive units. However, this still does not explain the observation that some animals are born with orientation sensitive cells. Linsker (1986; 1988) has shown a possible way in which such cells could arise, given only random noise input. In an analysis of this work, Mackay and Miller (1990a; 1990b) found that the oriented receptive fields found by Linsker could be explained in terms of combinations of the principal components of Gaussian filtered noise. Rubner and Schulten (1990) used another net model, which employs anti-Hebbian learning between units to extract PCs, and produced similar oriented receptive fields from Gaussian filtered noise.

Our approach was to use Sanger's net to perform a PCA on samples of natural images. Sanger has done this himself, with a view to image compression (Sanger, 1989). This works differs from his in two respects: we are sampling from many different images at once, and we remove edge effects with a Gaussian mask. This also distinguishes our results from those of Rubner and Schulten (1990), their receptive fields being firmly aligned with the edges of their square input array.

2.3.2 Method

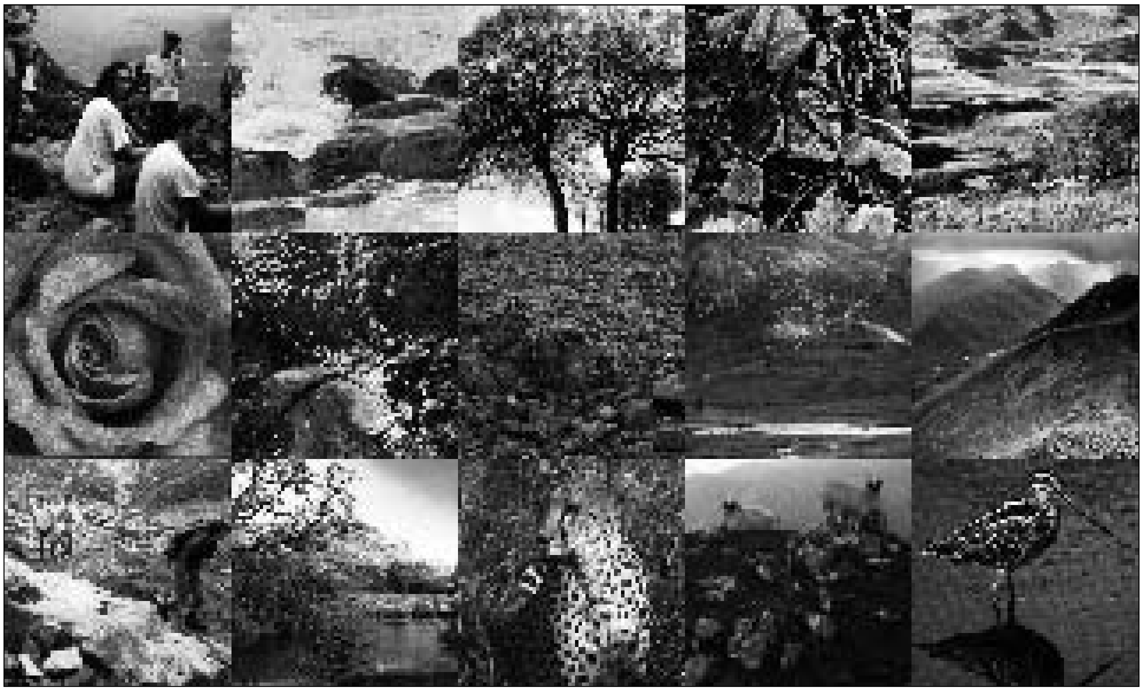


Figure 2.3: The 15 natural images used in the first experiment.

We initially chose 15 images of natural scenes, from a variety of sources. These are shown in figure 2.3. We avoided man-made objects, which tend to be regular, if not actually rectangular in shape, and also tried to avoid obviously flat horizons. Since the pictures came from several different cameras and lenses, no attempt was made to correct any optical irregularities. The photographs were scanned at 300 dots per inch, and reduced to 256 pixels square. 64x64 pixel samples were selected by choosing an image and location within it at random. The mean grey level, estimated from 20000 such samples, was subtracted from each pixel value. The image sample was then masked by (= multiplied, not convolved with) a Gaussian window of standard deviation (SD) 10 pixels. This meant that the edges of the sample were three SDs from the centre, which should be sufficient to avoid edge effects. The sample vector (a concatenation of the rows of the input array) was finally normalised to unit length. This acts as a form of contrast control. In the absence of such normalisation the results would be dominated by the brightest areas of the image.

Each output unit has a connection from all 4096 inputs, the weights being initialised with small random values so as to make the sum of the squares of the weights to each unit approximately 1. We used a rectangular scatter in the range ±0.03. Training proceeds by applying randomly selected inputs. Convergence is assisted by gradually reducing the value of the learning rate η: our standard procedure was to start it at 1.0 and then halve the value every 20000 presentations, for a total of 120000. The exact method of finding PCs is to find the eigenvectors of the correlation matrix of the input data. With 4096 inputs, the matrix would have 2^{24} entries, beyond reasonable computation. Sanger's method allowed us to find reasonable approximations in a few hours on a 1Mflop workstation.

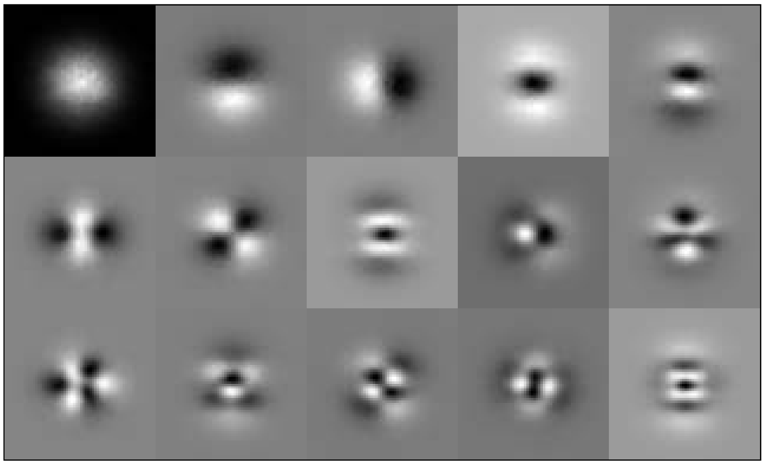


Figure 2.4: The first 15 principal components of our images, numbered from left to right, top to bottom.

2.3.3 Results

We can visualise the learned components by treating each weight vector as a 64x64 image. The first 15 components from our images are shown in Figure 2.4. Note that the sign of each operator shown has no significance: the net may converge such that a given unit has either positive or negative output. The first component is approximately Gaussian. The size is not determined simply by the Gaussian window of our pre-processing, but reflects the correlation scale of the images, as is demonstrated by the use of text images below. The second and third components resemble respectively the horizontal and vertical first derivative operators, modulated by the Gaussian window. We were initially suspicious that this orientation was due to residual edge effects. That it is not is demonstrated by the simple expedient of rotating the photographs in the scanner by 45 degrees. The results of this, shown in Figure 2.5, indicate that the orientation specificity does come from the images.

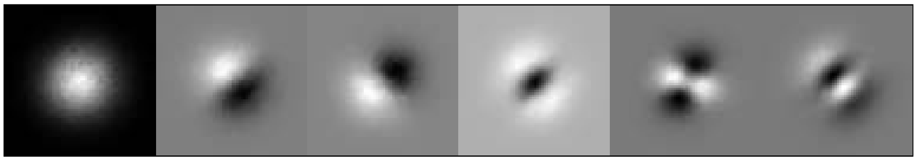


Figure 2.5: The first 6 principal components of 15 images, when rotated by 45 degrees on the scanner.

To test the consistency of the results, we used a number of different image sets. Figure 2.6 shows the results from an extended set of 40 images, including another 10 outdoor scenes and 15 taken in and around buildings. Despite the different input, the resulting components are very similar. Components 5 and 6 have reversed their order, while, for example, 8 and 9 from figure 2.6 appear to be formed by mixing the same components in figure 2.4.

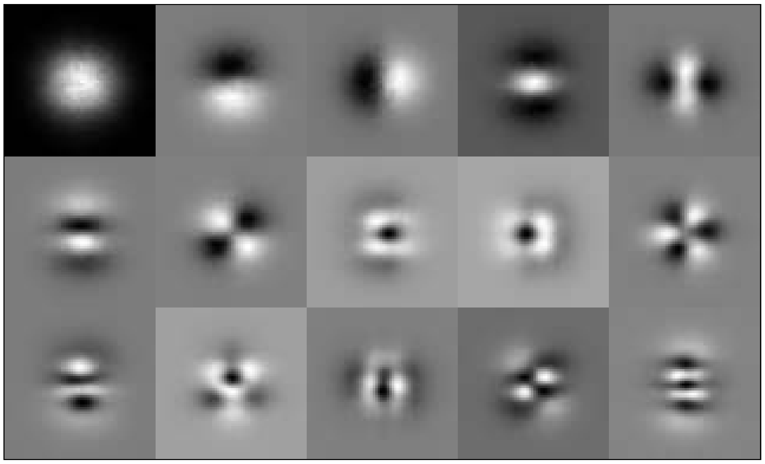


Figure 2.6: The first 15 principal components of an extended set of 40 images, numbered from left to right, top to bottom.

The output variance of each component was measured by applying a test set of randomly chosen inputs. The confidence limits on the variances are quite large, so 10,000 inputs were used to generate the results shown in Figure 2.7. As expected, the variances decrease with increasing component number. The log/log plot of Figure 2.7 shows an approximately straight line. However, there are some deviations caused by groups of two or three components having quite similar variances. The operators that have changed order or become mixed with different input images tend to be those which deviate from the generally straight line and have unusually similar variance.

Because of the variety of our images, there is no reason to suppose that the results would depend on the particular scale that was used for analysis. To confirm this, the experiment was rerun at double and half the original scales. For instance, we used 128x128 samples, with a Gaussian window of size 20. The results shown in Figure 2.8a and 2.8b are indeed very similar. By way of contrast, the same technique was used to extract principal components of text at various scales. Four different samples of the text at each scale were used for input. Rather than change the size of the windows, we simply changed the magnification of the text on the scanner. The results, Figure 2.9, now show a very marked scale dependence (note that the sign differences are again not significant). The first and second components are matched in spatial frequency to the inter-line spacing. Subsequent components match the pitch of the letter strokes. Note that, particularly for the finest scale text, the size of horizontal filters (components three and four) is substantially bigger than the vertical ones (five and six). This reflects the use of a proportional font: the lines of text are evenly spaced while horizontal letter positioning varies. The horizontal correlation distance is therefore smaller than the vertical. The last two components at the finest scale are tuned to word boundaries. This may be demonstrated by treating the component as a filter and convolving it with one of the original text images. The results of this are shown in Figure 2.10: there is a blob in every

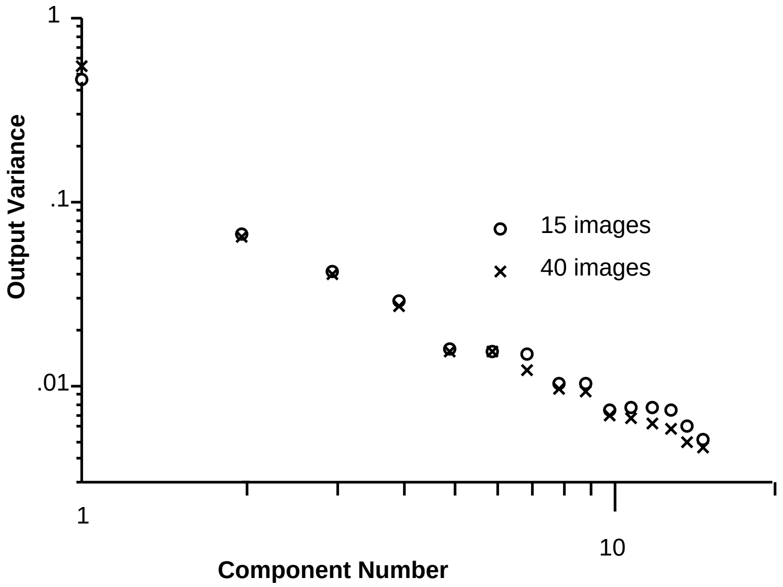


Figure 2.7: Output variances of the first 15 PCs, a) from 15 image set, b) from 40 image set, over 10,000 inputs.

word gap, with the biggest blobs making sentence ends. At the coarsest scale, the components are approximately the same size as individual letters.

Two of the components from the natural image set, 4 and 6 in figure 2.4, 4 and 5 in figure 2.6, resemble "bar-detectors", centred on the horizontal and vertical respectively. Foster and Ward (1991) performed psychophysical experiments on subjects' ability to detect a line segment presented on a background of line segments at a slightly different orientation. They were able to account for their results with a model that had only two orientation-sensitive filters, centred approximately on the vertical and horizontal. The response of the two "bar-detector" units from each of the image sets was measured by presenting a single line segment at 10° intervals. The results are shown in figure 2.11, superimposed on the results of Foster and Ward. The only degree of freedom in fitting the data is the maximum relative output of one of the components. This was matched for the vertical components (centred on 0°): the surprisingly good match for the horizontal filter comes purely from the data.

2.3.4 Discussion

The correlations detected by PCA in the images arise from the probability that adjacent pixels have a similar cause, for instance being part of the same object. The second component is consistently within a few degrees of horizontal, suggesting that horizontal correlations extend further than those in any other direction. There are two reasons why this should be so. The first is the action of gravity: tall, thin objects tend to fall over and become long, thin objects. That alone is not sufficient: horizontal objects lying with their long axis towards the observer still have apparently vertical edges. The second reason is foreshortening: a pencil on a table, end towards the observer, subtends a smaller visual angle than the same pencil rotated by 90°. Two questions naturally arise: a) is the relative sensitivity of the two "bar-detector"

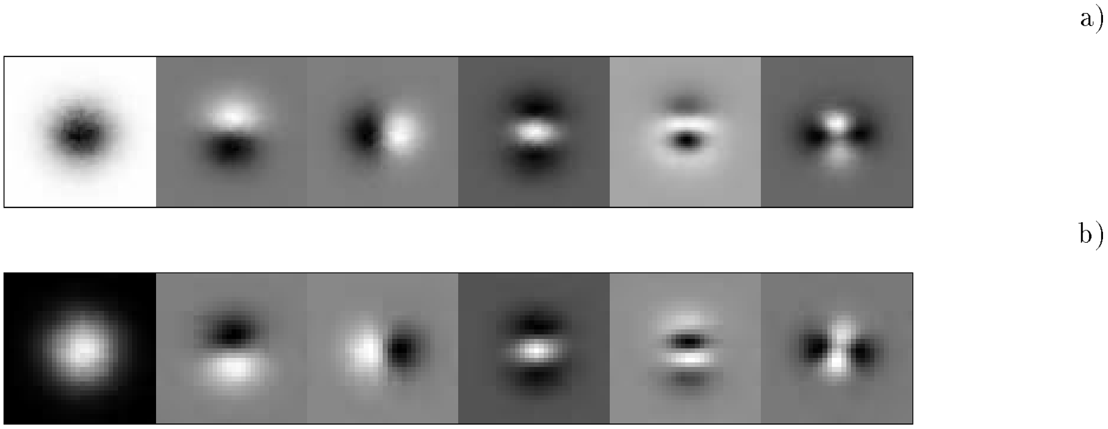


Figure 2.8: The first 6 principal components of the images, at double (a) and half (b) the original scale. Note that sign is not significant.

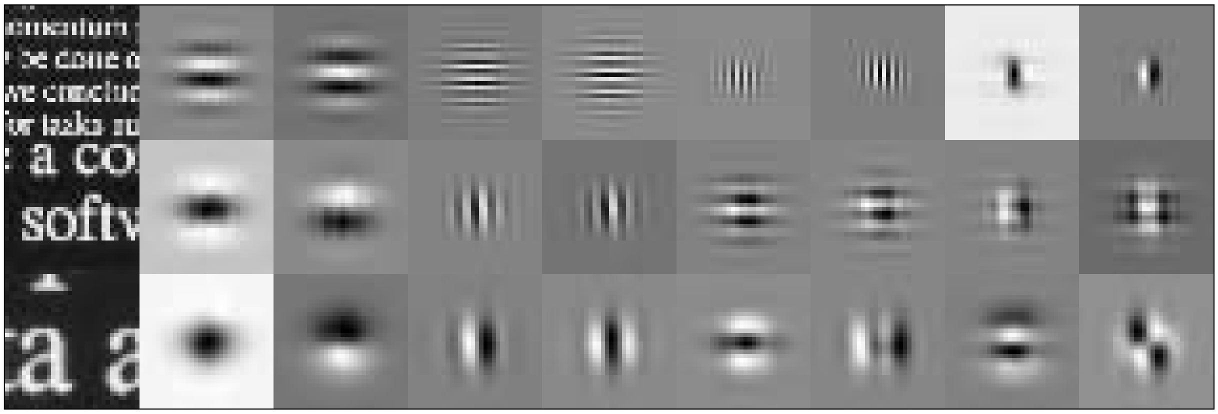


Figure 2.9: Some samples of Times font text, at three different scales, and their first 8 principal components.

components a response to the image statistics, or merely some inevitable and uninteresting side-effect of the method and b) is the match to the Foster and Ward model anything more than coincidence?

It is possible to shed some light on the first question by doing PCA on artificially generated noise images of varying horizontal/vertical anisotropy. Brownian fractals have a similar spatial frequency distribution to that of natural images. They may be generated by placing a large number of dark/light edges with random position and orientation across an image. Anisotropy can be introduced by biasing the selection of angles towards the horizontal, for instance by drawing samples from a rectangular distribution. Figure 2.12 shows three such fractal images of increasing anisotropy, generated from the same random number seed. Figure 2.13 shows the first 15 PCs obtained from a number of Brownian fractal images, generated from a rectangular scatter with height 0.75 times the width. The general form, if not the precise detail, resembles those obtained from natural images. Figure 2.14 shows the first seven components obtained as the vertical/horizontal ration is varied from 0.9 to 0.3. The significant component is the fourth. At the top of figure 2.14, when the input is almost isotropic, the fourth component is close to having a centre-surround structure. As the anisotropy increases, so the component

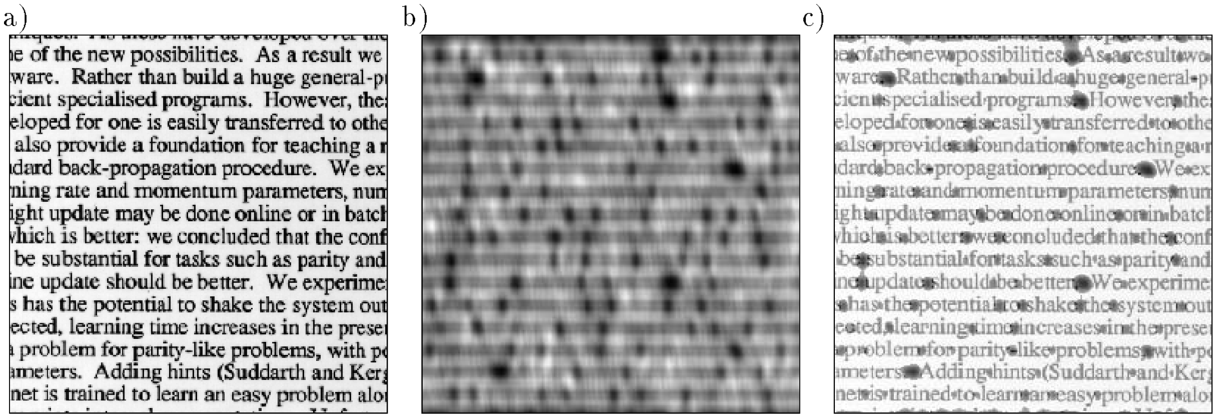


Figure 2.10: a) one of the original text images, at finest scale. b) Same image, convolved with seventh PC from Figure 2.9. c) As b), thresholded and superimposed on a)

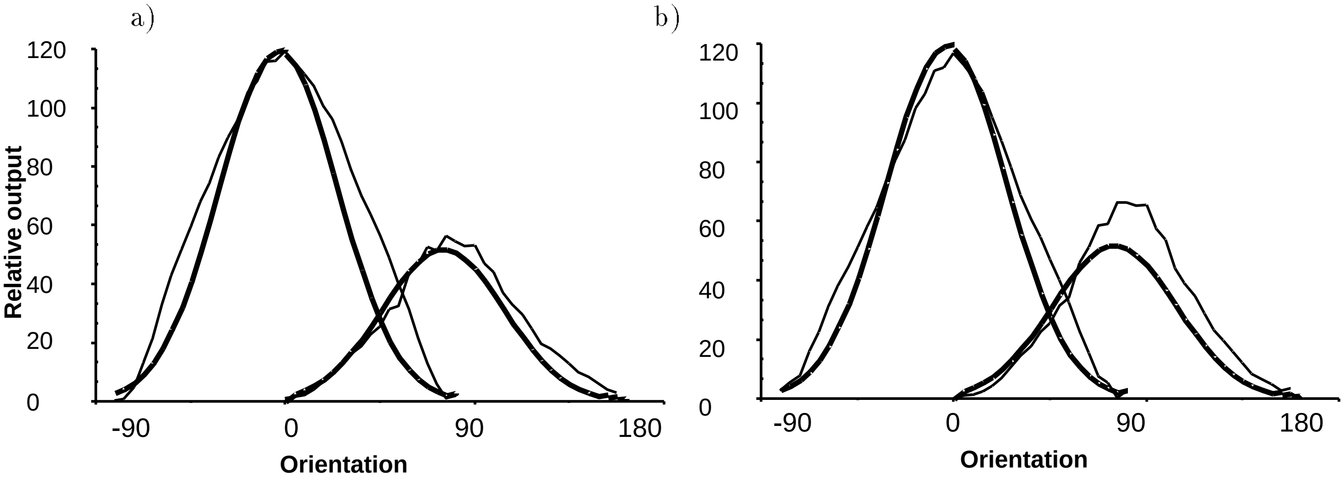


Figure 2.11: The orientation sensitivity model of Foster and Ward (1991) (thick lines) with 'bar-detector' components from (a) subset of 15 images (thin lines) and (b) full image set. Vertical units are arbitrary.

changes to having a horizontal orientation sensitivity. By the last row horizontals are so dominant that the third and fourth components swap places.

The fifth and sixth components in the first row are almost circularly symmetric, like those reported for Gaussian filtered noise by Mackay and Miller (1990a; 1990b). With isotropic input, they return equal variance. As the horizontal dominance increases, the sixth component changes to pick up vertical variance unaccounted for by the increasingly anisotropic fourth component. It also accounts for a decreasing amount of variance. The orientation sensitivities of these two components (fourth and sixth of the first row) were measured as before. The results are shown in figure 2.15, as a ratio of the sensitivity of the two filters. The Foster and Ward model is shown for comparison. It may be seen that the relative output varies smoothly with the horizontal dominance of the images, which suggests that the ratio found for natural images is a result of the statistics and not just the process.

The second question, whether the match obtained is coincidental, is harder to address.

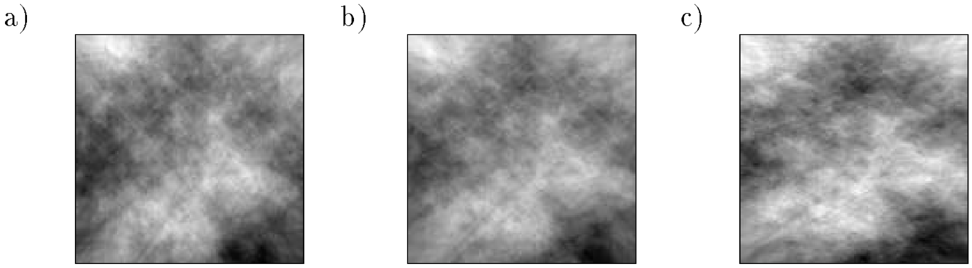


Figure 2.12: Brownian fractal images a) isotropic b) vertical/horizontal ration 0.75 c) ratio 0.5

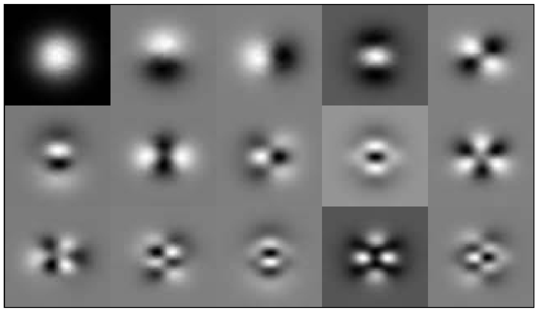


Figure 2.13: 15 principal components from anisotropic Brownian fractals

The paradigms are very different, but both systems are under pressure, time in the case of Foster and Ward (their presentations were of very short duration), information in the case of PCA. It may be that, within their short duration experiments, only the most significant channels are distinguishable from noise. There is evidence of an excess of horizontal and vertical receptors in cats (Vidyasagar & Henry, 1990), especially among the fastest responding cells. Supportive evidence that image statistics affect our visual system comes from a study of the horizontal-vertical illusion. This is the observation that a vertical line looks shorter than a horizontal line of equal length. Ross (1990) has shown that the extent of the illusion in children is correlated with their visual environment. Those raised in open country have a stronger illusion than city children. Baddeley and Craven (unpublished) have measured the correlations within a set of images designed to approximate the visual diet of the two groups. The horizontal/vertical ratio in the two sets is a very good match for the size of the illusion found by Ross. It seems likely that the visual system is tuned to the statistics of the images it receives, though where within the system the tuning occurs remains open.

The original purpose of this piece of work was to see what the response properties of visual cortex cells might be, were they performing PCA. While the general form of some of the components is not dissimilar, there are also marked differences. This is partly because the current work was performed at a single spatial scale, while it is known that visual cortex works at multiple scales simultaneously. The rather complicated higher components are attempting to account for fine structure, that might be done better by smaller, simpler filters. Perhaps more importantly, the PCs gave just two orientation sensitive components (by definition, because of the orthogonality constraint), while visual cortex has “bar-detectors” at all orientations. In a noiseless analogue, or fully digital system, such redundancy would not be needed. It is only necessary to have two orientation selective units to code any orientation.

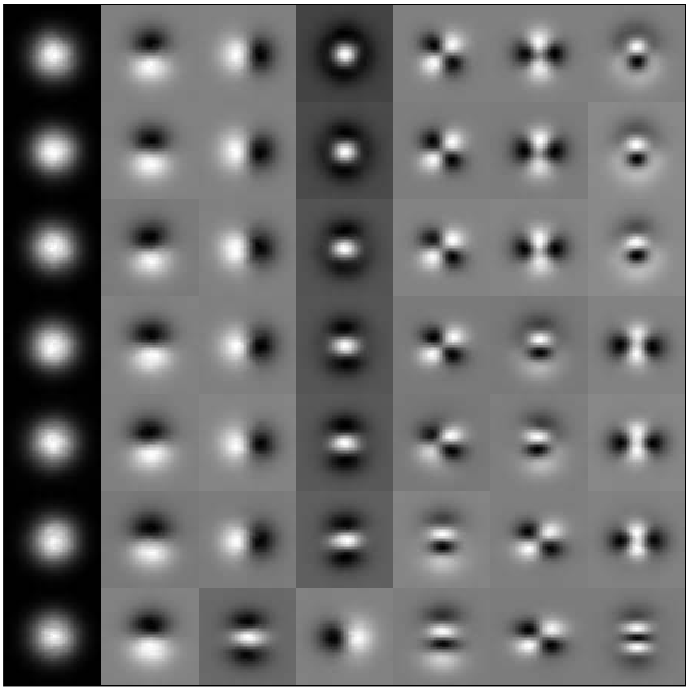


Figure 2.14: First seven PCs from Brownian fractals, varying the vertical/horizontal ratio from 0.9 (top) to 0.3 in steps of 0.1.

However, real neurons are noisy, and communicate by sending discrete spikes of activity. The visual system has to work in “real-time”: evidence of just how fast it can operate comes from observations by Oram and Perrett (1992) that face-sensitive cells in monkeys can identify a presented face within 90mS of stimulus onset. At typical peak firing rates of about 100 per second, this gives time for only about nine action potentials between the retina and the face cell. While it would be possible in principle to work out the angle of an edge, for instance, from only two orientation sensitive cells, it would take far too long because of the low response off-axis. Cells at all orientations may be required simply to ensure that one fires strongly for any input. Such considerations do not concern digital simulation of neural nets. Appropriate coding strategies in this case are the subject of the next chapter.

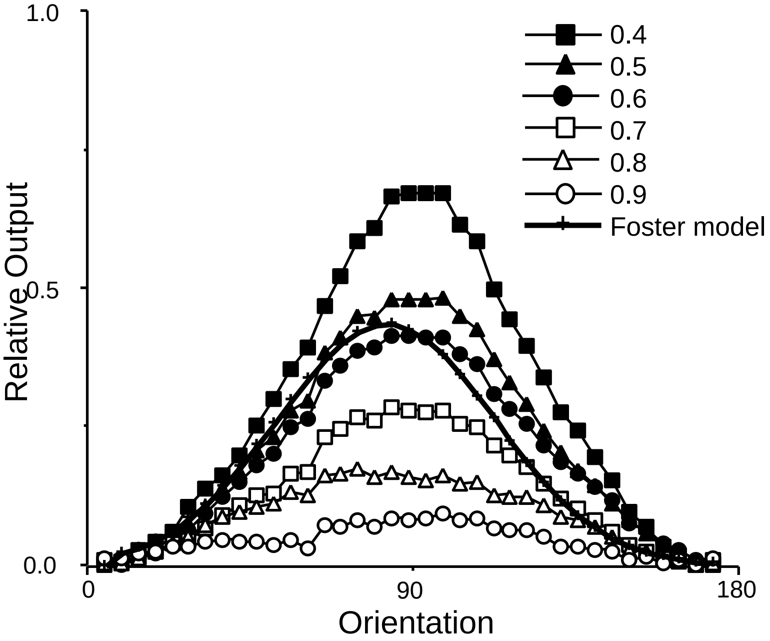


Figure 2.15: Orientation sensitivity of vertical 'bar-detectors', varying the anisotropy of Brownian fractals, adjusted such that the matching horizontal component has a peak output of 1. The model of Foster and Ward is included for comparison (thick line).

Chapter 3

Coding and generalisation

It may seem strange to spend hours of supercomputing time to obtain the wrong answers to simple arithmetic, but that is cognitive science (Anderson *et al.*, 1990).

3.1 Introduction

This chapter considers ways to improve the generalisation ability of nets. After a brief discussion of generalisation and approaches to improving it, four experiments are described. These explore the effects of input and output coding on the performance of the nets. The first two are concerned with the representation of numbers, for both real values and integers. The third and fourth consider coding of images. These are methods for doing data reduction, to reduce a 256×256 pixel image to something manageable for a net. The third explores some methods for handling the image data, the fourth further compresses the data by use of principal component analysis.

3.2 Generalisation

Possibly the most important aspect of neural nets is their generalisation ability, informally described as "training a network to respond reasonably to input data not present in the <training> data set" (Ji *et al.*, 1990). The obvious questions are what is meant by "reasonably", and how it might be measured.

Feedforward nets are typically designed to approximate some function defined over some input space. The function might, for example, be a binary or multiple classification task, or perhaps a real-valued function, such as the Mackey-Glass equation that has become something of a benchmark test (Mackey & Glass, 1977). Where a function has been taught, generalisation may be estimated by presenting a number of untrained inputs and observing how accurate the output of the net is. It might be expected that the net would interpolate smoothly between taught points, but this may well not be the case, especially if overfitting has occurred (see below). For a classification task, ideal generalisation would be given if the net always gives the correct class for any input. An important, unresolved question is what the net should do if the input is not in any of the training classes. For instance, if the net of section 3.4, which is trained to recognise faces, is presented with an image of a telephone, its output will be undefined. Ideally we might wish to have an extra output, to indicate that none of the trained patterns is present. Quite how to train such an output, given the infinite space of possible inputs, is not clear.

Perfect generalisation would give zero errors, but since the input space is often infinite (as many images of a individual's face as you like, and infinitely many points on the Mackey-Glass sequence), it is usually not possible to measure it directly. Only where the input space is finite, for instance with boolean functions such as parity, is it possible to give a precise answer. Paradoxically, generalisation performance is often not the issue in such cases, which are commonly used to test the ability of learning algorithms to learn the function at all. The generalisation ability on real-world tasks is typically estimated by testing the performance on an unseen data set drawn from the same source: this is the method employed here.

Having established a way to estimate generalisation ability, it becomes of interest to try and increase it. There are a variety of approaches.

- Stop training before the net overfits the training set.

- Alter the structure of the net: changing the number of layers or hidden units, using some subset of connections rather than a fully connected net etc.

- Constrain some of the weights to be the same value.

- Add noise during training.
- Alter the coding used at input and/or output of the net.

Overfitting

A common observation when training a net with Backprop is that the test performance increases, reaches a plateau, and then starts to decrease (an example is shown in figure 6.10 on page 132). This phenomenon is known as *overfitting*, where the net is starting to learn the individual training set i-o pairs rather than the underlying function. It is caused by there being too many weights in the net: equivalent to having too many free parameters in the model. In the limit, one might have one hidden unit per training input, in which case the net is able to form a look-up table. Such a method has actually been proposed by Xu and Zheng (1991), in an attempt to speed up learning. Not surprisingly, it achieves this, at the cost of disinventing neural nets!

One method of addressing overfitting is to stop the training just before it starts to occur. There are a variety of ways to do this, one being the use of a *validation set*. This is a third set of data from the same source, and is used to monitor generalisation during training. If all three data sets are similarly representative of the input space, this should correspond to the best performance on the unseen test set. Unfortunately, such a method requires that an often limited amount of training data be reduced further: other things being equal, better performance would be expected by training the net with all the available data.

Internal net structure

A better method might be to address the root problem of having too many free variables, i.e. weights. The baseline approach to this is to vary the number of hidden units in the net. Typical results are shown in figure 1.1 on page 10. Performance gradually improves as the net size is reduced, but then declines quite rapidly when it is too small.

A more sophisticated approach to the task of network design involves careful study of the problem domain. A simple example would be to reflect a two-dimensional input structure in the hidden layer(s). Such an approach is exemplified by the work of leCun et al (1989), who use a highly structured net to do hand-written digit recognition. However, their initial design is still a labour-intensive process of trial and error. There would be a real benefit in being able to do the design work automatically.

The approach adopted by many authors is to develop a learning algorithm that automatically constructs a net to match the needs of the training set. Fahlman's Cascade Correlation Algorithm (Fahlman, 1989), for instance, automatically generates a pool of possible extra hidden units, and adds the one that most reduces the output error. A variety of such constructive algorithms exist, e.g. (Frean, 1990; Marchand *et al.*, 1990; Mézard & Nadal, 1989; Hirose *et al.*, 1991). There is apparently a biological precedent: canaries grow extra neurons as they learn new songs (Bornholdt & Graudenz, 1992). A disadvantage of current algorithms is that they take no account of structure in the data, such as the two-dimensional nature of images.

An alternative approach is to start big, and prune down. The general finding that smaller nets tend to generalize better has lead to a number of attempts to modify the learning algorithm to remove unnecessary connections and nodes. Kruschke (1989) forces bottlenecks to develop by introducing competition between hidden nodes. leCun et al (1990) use calculations of the contribution of each connection to inflict what they term "Optimal brain damage" on the net. Rumelhart (1988) suggested penalising large weights by adding a term to the error gradient, an idea developed with Weigend (1991). As leCun makes clear, such pruning methods are not a panacea, and will work well only if the initial net design is sound. Manual design is a tedious process: some experiments are described in section 3.4, while the possibilities for using a genetic algorithm to specify the structure of the net are considered in chapters 5 and 6.

Weight sharing

In some nets, many of the connections ought to have similar values. An example would be a layer of units that act as feature detectors, each on a different subset of the input space. If the feature detectors are to perform the same role, then equivalent weights should have the same value. The process of training them can be speeded up by constraining them to be equal: although the number of connections remains the same, the number of weight values to be learned is reduced. This was done by leCun et al (1989) in a letter recognition net, it being required to recognise the letter in a number of positions on the input space. A similar method is used in time delay neural nets (Waibel, 1989), with the aim of obtaining time-invariant recognition. The idea was taken a stage further by Nowlan and Hinton (1991), who implement a soft constraint mechanism, by modelling the distribution of weights in the net by Gaussians. Weights which are not well accounted for by one of the Gaussians receive a penalty. The effect is to encourage the weights to move towards the centre of one of the Gaussians, which can themselves move during training. Typically one of them ends up centred on zero, so that weights that it accounts for are effectively pruned. If a number of the other weights need to be approximately the same value, one of the Gaussians will move towards that value and encourage them to be more alike. Nowlan and Hinton's scheme obviates the need for the experimenter to define which weights should be shared.

Addition of noise

Addition of noise during training has a long history in Backprop, being reported in some of the earliest experiments by Plaut et al (1986). Noise is usually added to the input patterns, the idea being to broaden the area of input space around each training exemplar that will give the same output. It presupposes that similar inputs give similar outputs: adding noise to a parity problem wouldn't help! Gardner et al (1987; 1988) proposed its use to broaden the area of attraction around each local minimum in a Hopfield type net. The effect of adding input noise to Backprop has been analysed by Matsuoka (1992), who shows it to be equivalent to adding a term to the error gradient to minimise sensitivity to disturbances of the inputs. Murray (1992) has developed an algorithm designed for hardware implementation that uses noise to improve the convergence rate.

Coding

The importance of input coding may be demonstrated with a simple example. The parity problem is a much-used toy problem for neural nets, precisely because it is so difficult to learn. Maximally similar inputs give rise to opposite outputs. The task is really quite trivial: requiring only that the number of on bits in the input be counted. If, instead of presenting the original string to the net, the binary representation of the number of bits set is presented (a trivial piece of digital hardware), the task for the net becomes equally trivial. All that is required is to pass through the value of the lowest bit. This is an example of the problem becoming easy once it has been solved. However, thoughtless coding can make the job unnecessarily difficult. In particular, traditional binary coding of numbers is a bad idea, because adjacent numbers can have radically different codes. Thus the code for 7 is 0111, that for 8 is 1000, maximally dissimilar in terms of Hamming distance. A net will have difficulty learning that the two inputs are to be treated similarly. It is perhaps not surprising that a code devised for traditional digital computers should not suit neural computation: binary codes are hard for untrained humans as well. However, codes that are meaningful to humans may also be hard for nets. Suppose three bits are coding political leanings thus:

1	extreme	left-wing	liberal
0	moderate	right-wing	conservative

Assuming no overlap in the middle ground, a similarity ordering would be 111, 011, 001, 101, 110, 010, 000, 100 (provided that the curious classification of hard-line communists as right wing in the ex-Soviet Union is excluded!). Such a code would appear arbitrary to the net, although meaningful to a human observer. The self-organising net described in section 2.2 would generate a continuous coding, with the three units responding to left, centre and right. Someone in the political middle ground might then be coded as 0.2, 1.0, 0.2. The practical utility of this kind of coding is explored in the next section.

Nets are often required to give a “one of *n*” output, corresponding to the class of the input. If the classes are completely disparate, this may be satisfactory. However, it may be the case that there is a similarity ordering on the outputs. Pomerleau (1991) designed a net to control an autonomous land vehicle. The output was one of *n* possible directions in which to steer. He found that the net trained more easily if, instead of activating just the target output during training, he activated neighbouring units as well. Fall off from the central unit was Gaussian. Again, this is just the kind of coding that a suitable self-organised net might produce.

3.3 Numerical coding strategies

3.3.1 Coding for arithmetic

As noted in the previous chapter, it is possible to generate distributed representations of an input using a self-organising net. Each individual unit develops an approximately Gaussian response curve, with the receptive fields of neighbouring units overlapping. The work reported in this section sought to ascertain how useful such codings might be for further processing by a net.

It was not clear how this could best be achieved. Ideally, it seemed that the whole system should be self-organising. Thus an association between two data sets might be established by forming self-organised codings of each, and learning a mapping between the codes. The major problem with this is how to measure the results. The usual sum of squared errors will not do, because the pattern of errors is significant. Suppose the target output is 0.3, 1.0, 0.3, corresponding to a signal on the centre of response of the second unit. Outputs such as 0.4, 0.9, 0.4 and 0.2, 0.9, 0.4 are the same distance from the target. However, depending on exactly how the outputs are decoded, the first will probably be interpreted as correct, while the second will be displaced towards the third unit. Since the target outputs are defined by the net rather than the experimenter it becomes even more difficult to interpret the results. One possibility is to train a net to do the reverse mapping, but this introduces a source of uncertainty: are errors being produced by the net being tested, or the decoding net?

It was therefore decided to use idealised codings, which were defined by the experimenter. It was also decided to use Backprop as the training algorithm, at least initially. As specified by Rumelhart et al (1986), Backprop was intended to work with binary coded problems. That the unit output function used is a sigmoid rather than a simple threshold function is a result of the need for it to be differentiable. However, it also allows analogue outputs to be learned, and the widespread use of Backprop with continuous mapping problems made it of some interest to compare a number of different input and output codings. The results reported here use Backprop.

Hinton et al (1986) discuss a binary *coarse coding.* This uses a number of units with overlapping receptive fields, which signal 1 if they see an input and 0 otherwise. Introducing continuous outputs allows the precision of the response to be increased. Such a coding, with a Gaussian response, is referred to here as *Gaussian coarse coding.* It is equivalent to the most common form of *radial basis function* (RBF), introduced as such to the neural net literature, at about the time this work was done in 1988, by Broomhead and Lowe (1988). A similar method has been known in pattern classification literature as as potential functions (Duda & Hart, 1973). Saund (1986) experimented with a similar coding scheme, which he called “scalerized”. He modified Backprop to encourage the hidden layer to develop the same neighbourhood codings, thereby compressing a two dimensional input space into a one dimensional map, rather like Kohonen feature maps. Dawson and Schopflocher (1992) have recently suggested another modification to Backprop to assist training nets which use similar units, which they call value units. Their intention is to improve performance on binary problems, rather than with continuous values considered here. Their results suggest that such units, with their modified training rule, out-perform the traditional Backprop sigmoid units on problems such as Xor and encoder.

Anderson et al (1992) attempt to teach a neural net to do arithmetic in a way that might mimic the way that humans do it. They use a hybrid analogue and symbolic coding. The analogue component is intended to account for observations such as the ability of humans to give the larger of two numbers more rapidly if they are very different than if they are close. It is coded as a bar of active units, whose position on a row of units depends on the number to be coded, further right for larger values. Their system may be significant as a means of linking analogue sensory codings with abstract symbolic codes. Since it is intended specifically for giving “ballpark estimates rather than a precise answer” it is not tried here.

Problem descriptions

Two problems were used for this study. The first is a straightforward, linear task, solving simultaneous equations. Three integers, a, b and c, in the range 0-15, are the target outputs, the inputs are given as $\frac{a+b}{2}$,$\frac{a+c}{2}$ and $\frac{c+b}{2}$. There are 4096 possible input-output pairs. 100, chosen pseudo-randomly without duplicates were used for training, and the whole set used for testing. The task can be solved by a two-layer net, indeed there is a simple, precise solution if the output units are linear. Because of the output non-linearity required by Backprop, performance was found to improve with three layers.

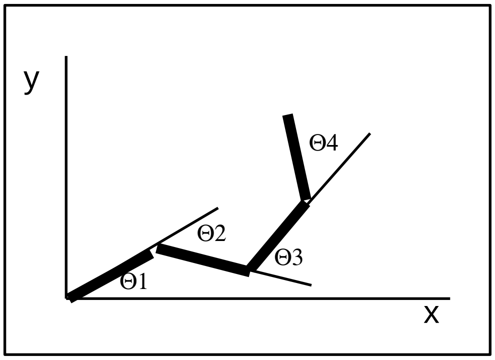


Figure 3.1: Simulated robot arm.

The second problem requires the net to output the x-y position of the tip of a simulated robot arm, given the joint angles. The arm is similar to that used by Jordan (1988), with four joints, see figure 3.1. The problem was restricted by allowing each joint to move through the range $\pm\pi/2$, and allowing only those combinations of randomly generated joint angles that gave positive values for x and y. Each section of the arm is of unit length, giving an output range for x and y of 0-4. 100 samples were used for training, 500 for testing. In initial trials, the inputs were coded as the sine of the angle, as is done by Jordan (1988), giving a range of ± 1. It was found that better results were obtained by simply dividing the angle, in radians, by $\pi/2$, giving the same input range. Use of the sine function should improve coding for angles near zero at the expense of bigger values. In Jordan's experiments the angles usually are near zero, here, larger angles occur frequently.

Coding schemes

Two discrete, or binary, codes were used, and five continuous codes. Takeda and Goodman (1986) considered a number of discrete codings for numbers for use with Hopfield nets. They found that "simple sum", which counts the number of active output units, and a more complex "group and weight" scheme were better than simple binary coding. Since both have many possible representations for one number, which makes creation of a training set difficult, they were not tried here. Those used were as follows, see also figure 3.3.

Value unit encoding. Each unit has a range within which it has an output of 1, otherwise output is zero. For the simultaneous equation problem one unit was used for each integer, which means it was unable to code the input fully, since half-values are used there. It was not tried for the robot arm problem.

Discrete thermometer. As value unit, except that each unit remains on if the coded value is above its range. This means that the number of units on, and therefore the total input signal, is proportional to the value being coded. For output coding, it also has the potential advantage that the net only has to learn to turn a unit on when the stimulus is large enough, and not turn it off again when too large.

Analogue. The simplest continuous coding, using just one unit, with output proportional to the coded value. In biological systems this would have limited resolution, and be prone to noise, problems that do not significantly affect software simulations except by design, but might be of significance in the design of hardware implementations of nets.

Continuous thermometer code. A combination of analogue and the discrete thermometer codes. Four units were used here, each covering a quarter of the total coding range and with an output proportional to the value within its sub-range. If the coded value is above the unit's sub-range, the output remains at 1, below it the output is zero.

Interpolation coding (Ballard, 1987). In its simplest form, this is similar to the simple analogue coding, except that there is a second unit with output inversely proportional to the coded value. Thus the sum of the two units' outputs is always 1. The scheme may be extended to many units, with each adjacent pair coding their section of the total range. Unless the coded value is on the centre of response of a unit, there will always be exactly two units with outputs greater than zero. Tests were carried out with both 2 and 4 such units. With 2 units, the centres of response were at the top and bottom of the coded range (0 and 15 for the simultaneous equation problem), with 4 units, they were evenly spread across the range (0, 5, 10 and 15).

Multi unit interpolation. If the responsive field of a unit extends beyond the centre of response of its immediate neighbours, then more than two units will respond over at least part of the coded range. The spread of the responsive field may be expressed in terms of the inter-centre gap. Thus a spread of 1.0 is identical to the simple interpolation coding, while a spread of 1.5 will give three units responding over all the range, except exactly half-way between unit response centres, see figure 3.2.

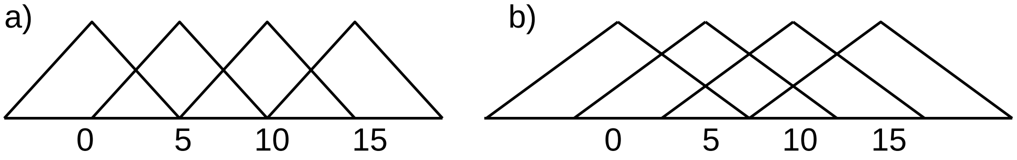


Figure 3.2: Multi unit interpolation coding of 0-15, with unit centres on 0, 5, 10 and 15. a) with spread of 1, b) with spread of 1.5

Gaussian coarse coding. Each number is coded by four units which have a Gaussian response:

$$output = e^{-0.5(\delta/\sigma)^2}$$

where δ is the difference between the unit's centre of response and the given number, and σ is the standard deviation of the curve. The result is similar to multi-unit interpolation coding, but the response fall-off is Gaussian rather than linear. For the the simultaneous equation problem, the centres were set at 1, 5, 9 and 13. Table 3.1 shows some examples, for $\sigma = 2$. For the robot arm, they were at -0.9, -0.3, 0.3 and 0.9 for input and 0.5, 1.5, 2.5 and 3.5 for output. See also figure 3.3. Depending on the value of σ, from one to all four of the units may respond for a given input. If only one responds, it is not possible to recover the input value unambiguously. The value of σ is therefore expected to have a significant effect on the overall accuracy of the net's output.

Coded number	Unit outputs			
2	0.88	0.32	0.0	0.0
6	0.04	0.88	0.32	0.0
7	0.01	0.61	0.61	0.01
12	0.0	0.0	0.32	0.88

Table 3.1: Example of Gaussian coarse coding, with $\sigma = 2$ and centres at 1, 5, 9 and 13, for input domain 0-15.

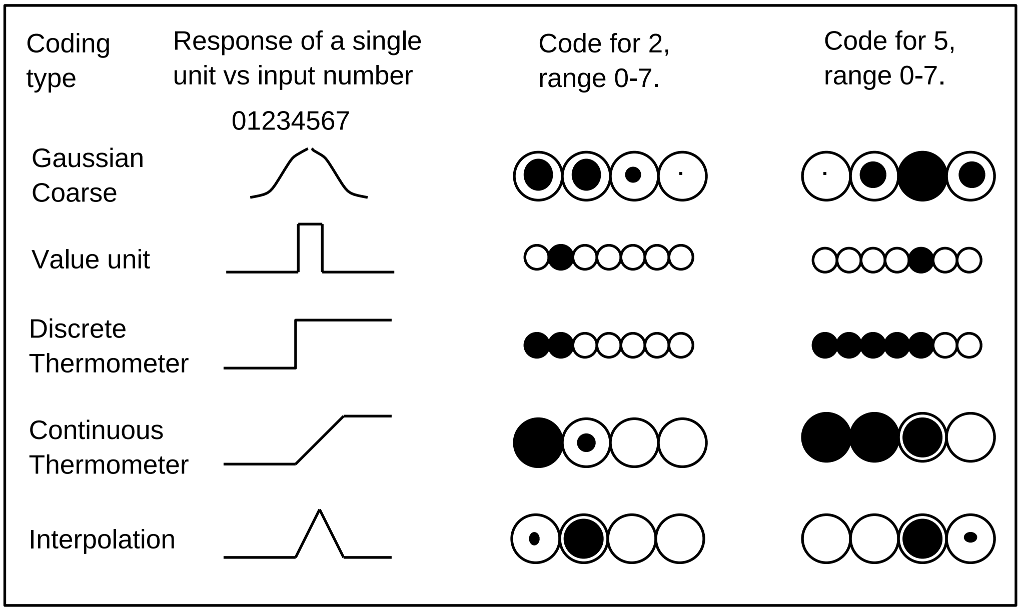


Figure 3.3: The various types of coding used.

Training

The nets were all fully connected between adjacent layers and trained with batch update Backprop. For the simultaneous equation problem, the whole training set was presented 1500 times, for the robot arm problem, 1000 times. At the start of run, momentum $\alpha = 0.3$ and learning rate $\eta = 0.05$: both were increased linearly between 100 and 200 epochs, to give $\alpha = 0.8$ and $\eta = 0.25$. This schedule was the result of a number of experiments, and gave slightly better performance than 3000 epochs at fixed learning rate and momentum. Continuing training to 10,000 epochs gave little additional improvement in performance. Only the simple one and two unit codings went to completion within the epoch limit, where completion was deemed to be no individual unit output error during training greater than 0.05. The same training and test data, in terms of the uncoded values, were used for all the nets.

All the nets used 12 hidden units. Initial experiments in the range of 6 to 24 units showed an improvement in test performance with the number of hidden units, which became less marked above 12. Performance would be expected to deteriorate if the net was too large. That this was not seen may simply be because training was discontinued before over-fitting occurred. It may also be that different coding schemes would have different optimal net sizes, though no evidence for this was observed. It was thought best to leave this particular variable fixed at what appeared to be a reasonable value. Some experiments on varying the structure of the net by genetic algorithm are reported in section 6.6.3. The size chosen is inevitably somewhat arbitrary, but serves to give a constant environment in which to compare the codings. Each net was run 15 times from different sets of pseudo-randomly generated starting weights, in the range ±0.2. The same 15 sets of starting weights were used for all the homologous nets.

Testing

For the simultaneous equation problem, the complete set of 4096 possibilities was used for testing, for the robot arm problem, the same set of 500 random angles was used in each case. The output values were decoded using the methods described in the appendix. The errors in the decoded values were averaged over all 15 trials to give the results shown in tables 3.2 and 3.3. Also shown is the average of the worst individual errors recorded in each run. The tests were rerun with activation noise added to the units, with the aim of making the units a bit more like real neurons. The activation value of each unit was subject to multiplicative Gaussian noise with mean 1 and a standard deviation of 0.1. The unit output calculated from this activation was then reduced in resolution to give 20 discrete steps between 0 and 1.

Results

Results for the simultaneous equation problem are shown in table 3.2. It may be worth reiterating that we are not interested in solving the problem, which in itself is trivial, the aim is to investigate the relative merits of the different coding schemes. The two discrete codings fare badly: value unit coding is not much better than chance. This is probably partly due to the sheer size of the nets: with only 100 training examples they will be poorly constrained even with such a regular problem. It will also be caused by the inadequate input resolution. As would be expected, the thermometer code did better, since the net has more information about the significance of each unit, but still much worse than the continuous codings. Due to their poor performance and long training times, plus the inherent lack of suitability for coding continuous variables, the discrete codings were not attempted for the robot arm problem.

Of the continuous coding methods, without noise, 2 unit and multi-unit interpolation coding stand out as being significantly better than the rest, which are quite similar. The differences are formally significant ($p < 0.001$), with a standard deviation on the 2 unit interpolation result of less than 0.01. When noise is added, the less redundant codings fare worst, as would be expected. The discrete thermometer coding is unaffected, but remains worse than any of the continuous coding methods. Of these, multi-unit interpolation is the clear winner.

For the robot arm problem, shown in table 3.3, the two simplest codings, analogue and two unit interpolation, fare much the best in the absence of noise. Of the others, continuous thermometer is again the worst, while multi-unit interpolation is the best. The absolute level of performance is distinctly worse for this problem: since the output range is only 0-4, the errors should be multiplied by 4 for comparison with those in table 3.2. Adding noise only affects the two less-redundant codings, reducing them to a similar level of performance to multi-unit interpolation. The effects of noise appear to be within the errors already made by

	Weights	No. of units			Without noise		With noise	
Coding type	inc bias	1	2	3	Avg err	Max err	Avg err	Max err
Value unit	1677	90	12	45	4.85	15	not tested	
Discrete therm.	1677	90	12	45	1.44	8.4	1.44	8.4
Analogue	12	3	-	3	0.43	1.2	1.06	7.3
Analogue	87	3	12	3	0.38	1.5	1.12	7.5
Continuous therm.	312	12	12	12	0.39	3.6	0.70	6.0
Gaussian coarse	312	12	12	12	0.41	4.0	0.47	4.2
Interpolation:								
Simple 2 unit	162	6	12	6	0.18	0.9	0.47	2.3
Simple 4 unit	312	12	12	12	0.34	3.1	0.37	4.4
Multi-unit	312	12	12	12	0.20	1.7	0.27	1.9

Table 3.2: Results for the simultaneous equation problem, average of 15 runs

	Weights	No. of units			Without noise		With noise	
Coding type	inc bias	1	2	3	Avg err	Max err	Avg err	Max err
Analogue	76	4	12	2	0.08	0.7	0.17	0.8
Continuous therm.	308	16	12	8	0.29	1.8	0.29	1.9
Gaussian coarse	308	16	12	8	0.20	1.1	0.21	1.3
Interpolation:								
Simple 2 unit	160	8	12	4	0.09	0.7	0.16	0.8
Simple 4 unit	308	16	12	8	0.22	1.7	0.22	1.7
Multi-unit	308	16	12	8	0.16	1.4	0.17	1.5

Table 3.3: Results for the robot arm problem, average of 15 runs

Simultaneous equation problem									
	Sigma	2	2.5	3	3.5	4	5	6	
	Average error	0.64	0.51	0.45	0.41	0.41	0.46	0.46	
Robot arm problem									
Sigma in 0.5	Sigma out:	0.6	0.8	1.0	1.2	1.4	1.6	2.0	2.5
	Average error	0.26	0.24	0.22	0.21	0.21	0.20	0.22	0.45
Sigma out 1.4	Sigma in:	0.3	0.4	0.5	0.6	0.7			
	Average error	0.25	0.23	0.21	0.21	0.25			

Table 3.4: Effects of altering σ in Gaussian coarse coding

the more complex codes.

Effects of response curve spread

As noted above, it is to be expected that the performance of Gaussian coarse coding and multi-unit interpolation coding would depend on the spread of the units' response curves. Lorquet et al (1991) continued the study of the relative merits of these coding methods, introducing an adaptive coding, similar in spirit to that used by Moody and Darken (1989) to tune RBFs. Lorquet et al note that they did not try varying the response curves, "because it is determined according to Shannon's criterion". They do not state the resulting values, except for a diagram which implies a spread of 2, in the terminology used here, for multi-unit interpolation coding. Shannon's criterion suggests that a signalling unit should be on 50% of the time for maximum information transmission. For continuous units, this implies that the average output, over the whole input domain, should be 0.5. This is the case for the analogue and 2-unit interpolation codes used here. For multi-unit interpolation and Gaussian coarse coding, the situation is less obvious, because of overlapping RFs and edge effects. Numerical integration over the four RFs for Gaussian coarse coding indicates an average response of 0.5 if $\sigma = 3.8$ for the simultaneous equation problem, and $\sigma = 0.55$ for the input of the robot arm problem. Trials were run for a range of values: the results reported in tables 3.2 and 3.3 were the best obtained, averaged over 15 runs. The effects of varying the response spread for Gaussian coarse coding are shown in table 3.4 and the best values for input σ are indeed close to those predicted. The optimal value for the output is probably defined more by interactions with Backprop's sigmoidal output function than by information content, and is larger than the predicted value of 0.98.

The results for multi-unit interpolation coding are shown in figure 3.4. The analysis gives a predicted spread of 2.25. This is quite a good value for both problems, but the behaviour at larger values differs, and both show a marked minimum at 1.5. The complex pattern of results in this case suggests competing effects, probably again with the sigmoidal output function. Note that a spread of 1.5 corresponds to an RF that will just cover the input domain, and so would give an average response of 0.5 from a central unit.

These results suggest that the analysis is useful, the predicted values giving good if not optimal performance. A consequent prediction is that units placed near the edges of the input domain should have a larger receptive field than those in the middle, to increase their average response.

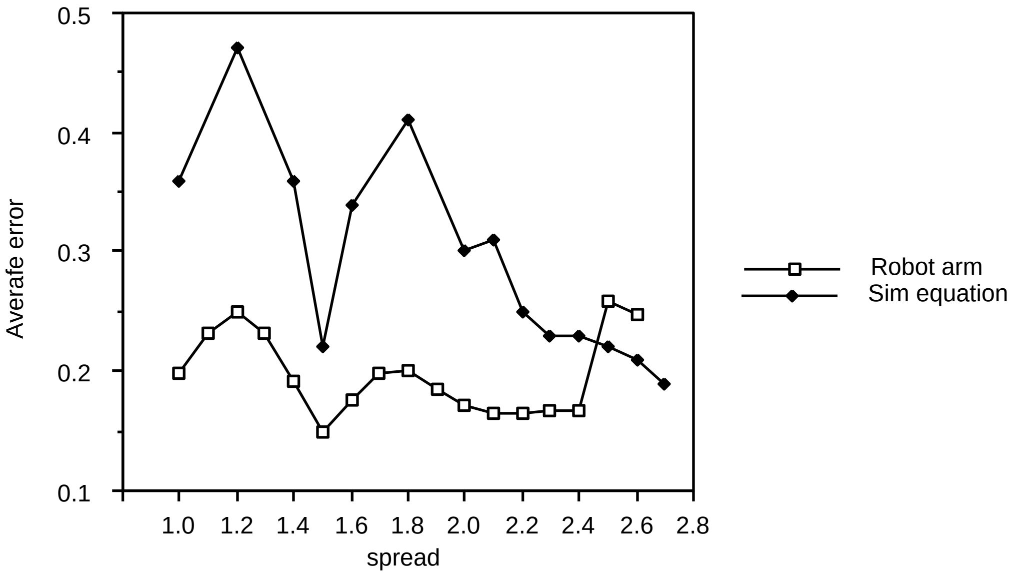


Figure 3.4: Effects of varying i/o spread on average test error for multi-unit interpolation coding on the two problems.

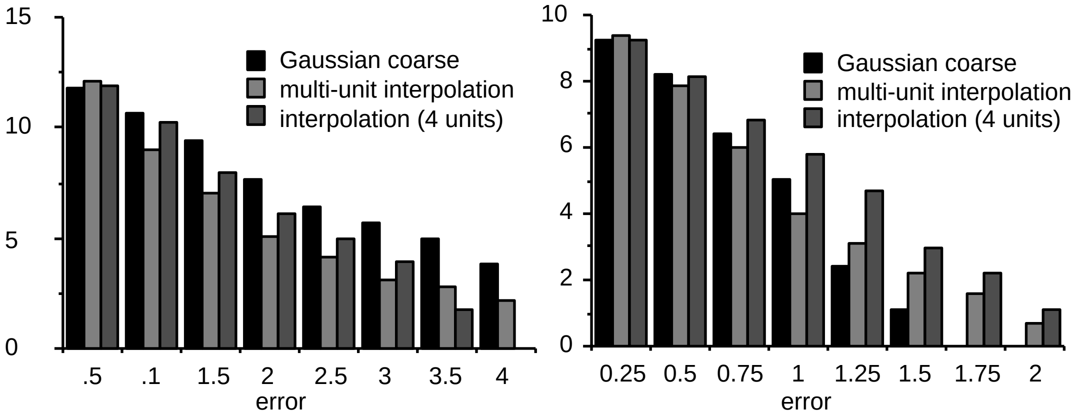


Figure 3.5: Log of frequency of occurrence of errors of given size for simultaneous equation problem (left) and robot arm problem (right). Note that log(0) is regarded as 0.

Consistency of results

In some applications, consistency of results may be more important than minimising the average error. For instance, with a robot arm, it may be better to be always in approximately the right place, rather than usually spot-on, and sometimes wildly out. Tables 3.2 and 3.3 show that the maximum error is not always closely related to average error. Figure 3.5 shows histograms for the (log) frequency of occurrence of errors of various sizes on the two problems. The distribution of errors does indeed differ between the codings but unfortunately not with any consistency. Thus Gaussian coarse coding is best for the robot arm problem, but worst on the simultaneous equations.

The accuracy of the results also varies over the output range, though more so for the simultaneous equation problem than for the robot arm. Average errors for each target output value are shown for the former in figure 3.6. Gaussian coarse coding does badly at 1, 5, 9 and 13, which are the centres of response of the units, and at the ends of the range, especially 15, where only one unit is responding well. Multi-unit interpolation coding also shows clear peaks of error on the four unit response centres. Two-unit interpolation shows a strange double-humped pattern when only two layers of units are used. Addition of a hidden layer removes these two peaks of error, leaving only the extreme ends of the range, where the sigmoid output function is flattest.

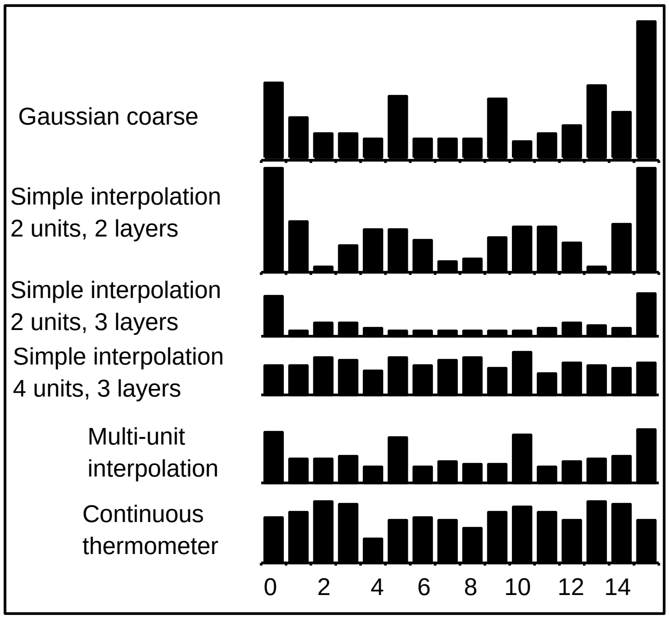


Figure 3.6: Distribution of average size of error with target output value for the simultaneous equation problem. Note that the scale is only constant within each histogram.

Conclusions

Discrete coding methods such as value unit do not work well for these numerical problems. For many problems the simplest, single unit analogue code may be best, since this also minimises the number of units, and, therefore, connections to be trained. However, the additional redundancy offered by two-unit interpolation is certainly worthwhile for the simultaneous equation problem, the error rate being halved. For real problems requiring numerical input, it would probably be worth trying both methods.

The more redundant coding methods do fare better in the presence of artificial noise. Multi-unit interpolation coding does best, though care is needed about the choice of spread. From these results, however, it is only fair to conclude that they are not well-suited to use with Backprop, partly because of interactions between the coding and the sigmoid output function. Gaussian coarse coding is especially vulnerable, because of its own Gaussian response, the unit outputs are more often in the inaccurate tails of the sigmoid.

The multi-unit codings are least accurate around the centres of response of the units. A common biological model appears to be to cover an input space at several different scales, so that cells with small receptive fields cover the same area as others with larger fields. If this

Decimal	Binary	Gray	Decimal	Binary	Gray
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Table 3.5: Comparison of Gray and Binary coding

hierarchical model were used in simulation, it might be best to avoid units at several scales having the same centre of response, since they would then all be least accurate in the same place.

This work appeared in the proceedings of the 1988 Connectionist Models Summer School (Hancock, 1989a) and has since found some diverse applications. Guan and Gertner (1991) found that Gaussian coarse coding gave an improvement over a less constrained analogue coding in a model of Red Pine tree survival, while Dyck et al (1992) used it in a net to help determine the mesh density needed to model magnetic devices.

3.3.2 Coding the alphabet

This piece of work originated from a talk given at Stirling by a visiting speaker (Tony Sanford), who had been trying to teach a net the sequence of the English alphabet. For each letter input, the net was to output the next letter in the sequence. He had been using binary coding, thus A was 00001, B was 00010, etc. The net used was that proposed by Elman (1988), which has a recurrent feedback loop. The input consists of the current letter, plus some units which receive input from the hidden layer units. These units thus provide a context: D is preceded by C which was preceded by B and so on. The units are set to zero when A is presented. Thereafter their activity a at time t is given by mixing in some of the hidden unit activity h:

$$a_t = (1 - f) \times a_{t-1} + f \times h_{t-1}$$

The feedback parameter f is usually set to 0.5. Thus the extra input units carry a decaying history of the hidden units' activity.

Although billed as coding the alphabet, this experiment might as well be numeric, i.e. teaching the net to output the code for the next number, since Sanford simply coded the letters as binary numerals. The discontinuities of binary coding might be expected to make the task rather difficult. A number of alternatives were tried: value unit coding, Ballard's interpolation coding (Ballard, 1987), which was successful in the previous experiments, and Gray coding. Gray codes have the property that the binary codings of adjacent integers differ in only one bit, see table 3.5. For instance, the Gray code for 7 is 0100, for 8 it is 1100, whereas in normal binary coding, 7 and 8 are maximally different. It was expected that the increased adjacency in the codes for neighbouring numbers would assist learning.

	A	B	C	D	E
Binary	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1
Gray	0 0 0 1	0 0 1 1	0 0 1 0	0 1 1 0	0 1 1 1
Value unit	0 0 0 0 1	0 0 0 1 0	0 0 1 0 0	0 1 0 0 0	1 0 0 0 0
Interpolation	0 0 0 1	0 0 .25 .75	0 0 0 .5 .5	0 0 0 .75 .25	0 0 0 1 0

Table 3.6: Four different codings for the alphabet

	Target max error	
Interpolation code	0.15	0.1
0, 0.25, 0.5, 0.75, 1	182	444
0, 0.2, 0.5, 0.8, 1	73	365
0, 0.3, 0.5, 0.7, 1	272	485
0.1, 0.3, 0.5, 0.7, 0.9	75	561

Table 3.7: Average number of epochs required to reach target maximum error for various different interpolation codings

Initial experiments using just four units suggested that while Gray coding was somewhat better than binary coding, interpolation coding required 50 times fewer epochs to achieve learning. However, there are a number of ambiguities concerning the interpolation coding. One concerns the learning rate. Because the coding is easier to learn, the learning rate may be increased without encountering local minimum problems. While doing so is clearly valid, it confuses direct comparison. Another ambiguity is the endpoint for learning. Completion in a Backprop system is typically regarded as getting to within 0.2 of the correct value. While this is satisfactory with binary codings it may not be when values between 0 and 1 have specific meaning as with interpolation coding. Reducing the maximum error from 0.2 to 0.1 may take longer than getting it down to 0.2 in the first place.

A third aspect of interpolation coding in this discrete application is the spacing between adjacent points. Usually the spacing is regular, as in 0, 0.25, 0.5, 0.75, 1. However, the output function in standard back-propagation is sigmoidal: so that the top and bottom of the range are compressed. It might be that an irregular spacing such as 0, 0.3, 0.5, 0.7, 1 would be more appropriate. Alternatively it might be better to avoid the extremes of the sigmoid thus: 0.1, 0.3, 0.5, 0.7, 0.9. Using 6 units, that will code 21 letters, a variety of tests were tried. These used 12 hidden units, $\alpha = 0.8$, $\eta = 0.5$, $f = 0.5$, random weights ±0.5. Figure 3.7 shows results averaged from 6 runs, from the same sets of random weights for each coding.

This is a complex pattern of results which implies more than one competing effect. It is extraordinary that an apparently minor change in the coding, from 0 to 0.1 and from 1.0 to 0.9, should reduce training time by a factor of four. Yet if the training is pursued a little further, the relative performance reverses. It is clear that increasing the required accuracy from 0.15 to 0.1 requires a lot of work. The second coding comes out best on both and was therefore used in tests on the whole alphabet.

	$\alpha = 0.5$ $\eta = 0.3$	$\alpha = 0.7$ $\eta = 0.5$
Value unit	422	159
Interpolation 0, .5, 1	542 (1)	204
Interpolation 0, .2, .5, .8, 1	132	49
Gray	267	536 (3)
Binary	766 (4)	542 (2)

Table 3.8: Average number of epochs required to reach maximum error of 0.2 for various different codings. Figures in brackets are number of times the target was not reached within 1000 epochs.

Binary ($\epsilon = 0.2$)	527(2)	$\alpha = 0.7$ $\eta = 0.5$
Gray ($\epsilon = 0.2$)	192	$\alpha = 0.5$ $\eta = 0.3$
Interpolation ($\epsilon = 0.15$)	553(1)	$\alpha = 0.8$ $\eta = 0.7$
Interpolation ($\epsilon = 0.1$)	696(2)	$\alpha = 0.8$ $\eta = 0.5$

Table 3.9: Number of epochs required to reach target maximum error ϵ.

26 characters requires five units with binary and Gray codings, 26 for value unit encoding and something in between for interpolation depending on the spacing. In addition to the five level interpolation coding above, which requires 8 units, a wider spacing of 0, 0.5, 1 was tried. This requires 14 units. The target maximum error was 0.2, other run details were as above. The results are shown in table 3.8.

The figures in brackets are the number of times when solution was not reached within the epoch limit of 1000. Usually this implies that the system is stuck with a maximum error of 1, a common problem when using binary outputs and too high a learning rate. From these results it is clear that the Gray coding is better than binary, but it appears that the five level interpolation coding is easily the best. However, these figures all relate to a maximum error of 0.2. A fairer comparison would be an error of 0.4 for the binary codings, and 0.15 or even 0.1 for the interpolation coding. 0.15 might be sufficient, because of the inherent redundancy in interpolation coding. The best results obtained at these error levels are shown in table 3.9.

With the necessary accuracy requirements, interpolation coding loses out. Gray coding remains clearly better than binary. The results should not be surprising: the sigmoidal output function is designed for binary representations and is not well suited for accurate analogue coding. Other things being equal, it is much easier to learn mappings where similar inputs give similar outputs. Analogue codings should be better when the net is specifically designed for them, for instance by putting inhibitory connections between the output units.

3.4 Coding images

The interpretation of images must rate with natural language comprehension as one of the Holy Grails of artificial intelligence. A task that we perform apparently so effortlessly turns out to be fiendishly difficult to specify in traditional programming terms. Since neural nets

Area	Mass	Centroid	Length	Width	Ratio	Axis
1.91	2.04	0.84,-0.43	1.98	2.23	-0.06	0.77
1.89	-2.02	1.47,-0.44	1.74	1.37	0.38	-1.66
1.58	-1.84	-1.02,-0.41	1.63	0.88	0.77	0.88

Table 3.10: Example of Mirage "blob" descriptions.

claim some biological inspiration, it seems natural to apply them to images and there have been many and varied attempts.

Images require processing before application to a mapping net: a typical 256 × 256 pixel array contains far too much data. Apart from the training times, it would be difficult to constrain the net sufficiently to prevent it learning the training set on the basis of individual pixel grey levels. One approach is simply to reduce the resolution, perhaps to 32 × 32. This is the method used by Pomerleau (1991) as the input to his vehicle navigation system and by Cottrell and Fleming (1990) in a face recognition system. Cottrell and Fleming then further compress the data using an encoder arrangement of Backprop, that produces a coding resembling the principal components of the input. This method is also used in Sexnet, a system used to identify the likely sex of a face (Golomb *et al.*, 1991). These methods differ from the PCA approach of the previous chapter in that the whole image is coded in one set of components, which would therefore be specific to the subject matter.

There are many more complex input codings in the literature. For input to their "TRAFFIC" system, Zemel et al (1990) use the internal angles of the triangles between triples of stars, to recognise constellations. Kanerva (1990) uses a contour map, i.e. lines of equal intensity in the image, coded as three component vectors sampled on a regular matrix. Weinshall et al (1990) and Frohn et al (1987) use a feature map, though without specifying how the features might be obtained from the image. The principal components procedure of section 2.3 seems a possible candidate for generating suitable feature detectors: such a method has been used as the input to a system to classify textures by Sanger (1989) and by McWalters (1991). There seem to be two general approaches: either assume preprocessing and code data in a way that aids later stages of processing; or make some attempt to follow biology.

The experiments reported here are based on a single-resolution simplification of Mirage (Watt & Morgan, 1985). This model of early visual processing, so named because the authors did not at first believe in it, was developed to account for many results from visual psychophysics. Since Mirage appears to have some biological foundation, it was of interest to see how its output might be applied to a neural net. The experiments also explored the effect of structuring the hidden layers of the net on the test performance.

The pre-processing consists of a Laplacian of a Gaussian (LoG) filter, followed by thresholding. This breaks the image up into positive and negative "blobs": see figure 3.7. Each blob may then be coded by parameters such as its centroid, mass and shape. The processed image can be described by a set of "sentences", table 3.10 showing the first 3 from a set of 10.

There is clearly a problem in scaling such "sentences": it will be the relationship between the different blobs which will be of importance in mapping the images, not their absolute sizes. The values given in table 3.10 are in terms of standard deviations away from the original mean value of each field, so that the new mean of each field is 0. This introduces a degree of invariance with respect to image size and position. Very early trials without this simple form of invariance produced excellent results, which turned out to be based solely on how close the camera was to the subject.

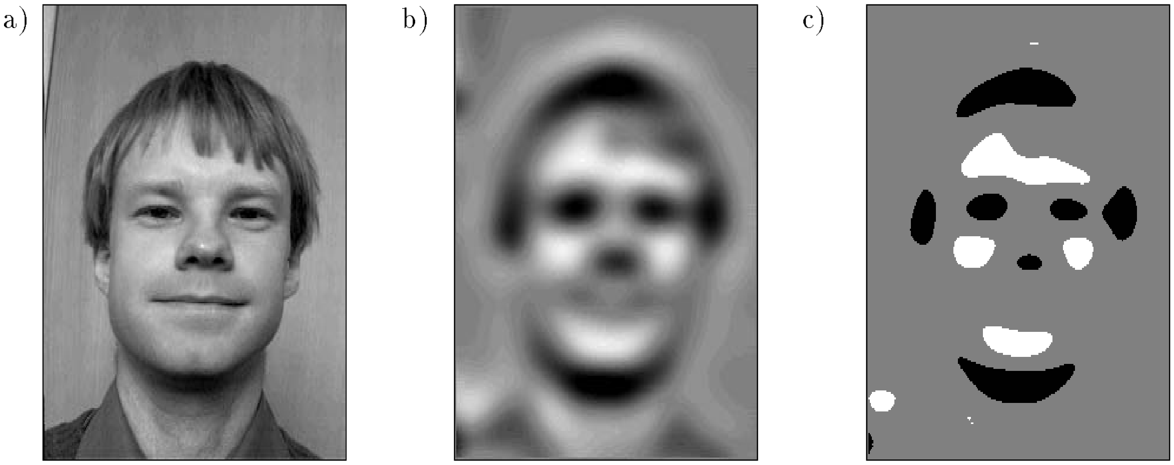


Figure 3.7: a) 256x256 original image. b) After Laplacian of Gaussian filter with standard deviation (SD) = 16 pixels. c) After thresholding (at 1.5 times the SD of the pixel brightness)

This indeterminate number of sentences, in no particular order, must be coded in a way suitable for application to a net, ie as a fixed-length vector. Three types of coding are tried here. A simple method (Type 1) is to order the blob descriptors by mass and use only the n largest. Applying a LoG filter with standard deviation of 16 pixels to the image usually gives about 10 blobs, so $n = 8$ was chosen. This means using 48 input units: the area and ratio fields were not used, being essentially redundant. The other values were passed to the net unaltered, giving an input range of approximately ± 2. The position of each blob is given explicitly, by the value for x and y of the centroid. Thus the input layer is not at all retinotopic. One result of this is that the position of each sentence in the input layer depends on all those before it. Thus, if one blob in a particular image becomes two in another, due to slight differences in lighting or camera angle, the position in the input layer of all the smaller blobs will be affected.

An alternative coding (Type 2) is to give the input to the net a topological structure that reflects that of the image, figure 3.8. Thus the input layer has some degree of retinotopicity. For this, an array of nodes was used, each implicitly corresponding to a particular location in the image. At each node are five units, which carry information about the blob nearest to their implicit location. Four of these units code the mass, length, width and orientation values, the fifth codes how near the centroid of the blob being described is to the node's centre. A direct hit gives a value of 1. If the nearest blob is outside of the node's receptive field, then all the node values are set to 0. The values for mass etc. are rescaled to run from 0 to 1. A value of 0 in one of these fields is therefore ambiguous: if the distance unit is set to zero then no blob is present: if not then zero indicates that that value is 2 standard deviations or more below the mean. Tests were carried out with different input layer sizes: the results reported here are from an 11×11 array of nodes, giving 605 input units.

A simpler alternative is to ignore the sentence level description of the image generated by Mirage completely and code the thresholded LoG filter output directly (Type 3). Input units are arranged in an array, each with a Gaussian receptive field (RF), see figure 3.9. Each

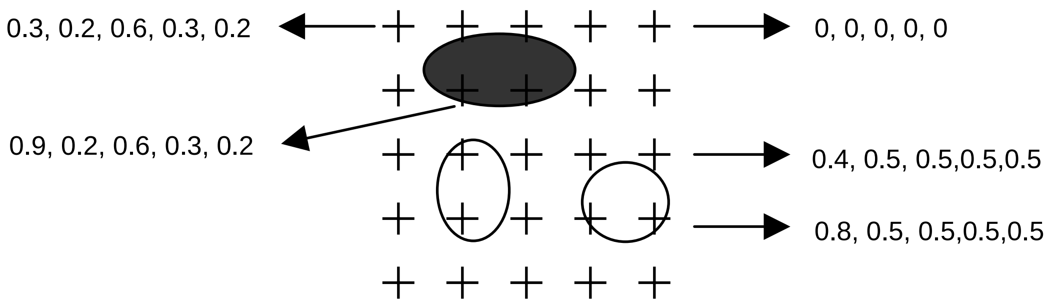
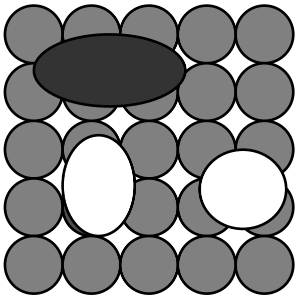


Figure 3.8: Example of Type 2 coding. Each cross represents the centre of response for a node, each containing 5 units. Example responses are shown for 5 such nodes. The first number is a measure of how close the centre of the nearest blob is to the node centre. The receptive field radius is twice the distance between nodes. The next four figures code the mass, width, height and orientation of the blob: these values are the same for all nodes coding the same blob.

unit integrates the image within its RF, and signals the result as a value between -1 and 1. The disadvantage of this is that it loses any kind of position or size invariance. This may be regained by using the statistics of the whole image to centre and scale the RFs. After thresholding, the centroid and sd of all the blobs combined (ie the whole image) are calculated. The centroid then becomes the centre for the input array: an operation equivalent to centering the attention of the net on the face. The area covered by the array is determined by the sd of the combined blobs, with unit RF centres up to 2 sds from the centroid. If this results in units falling outside of the image then the centre and if necessary the area are altered appropriately. The provision of the values for the input layer is then a two-stage process, with the first stage providing some size and location invariance.



-0.1	-0.3	-0.3	0.0	0.0
-0.2	-0.7	-0.5	0.0	0.0
0.0	0.6	0.1	0.2	0.3
0.0	0.9	0.1	0.4	0.7
0.0	0.0	0.0	0.0	0.0

Figure 3.9: Example of Type 3 coding. Each circle represents the receptive field of an input unit. The blobs within that field are integrated, producing a value that is then normalised to the range -1 to 1.

3.4.1 Mapping net structure

The simplest possible structure for a mapping net is to have input units directly connected to the output units, with no hidden layer. If the inputs are linearly separable then this may be sufficient. However, although the net may learn the training set, it may not predict the test set as well as more complex architectures. A two layer net gives a baseline performance to compare others against.

The simplest three layer net will have a feedforward structure with a number of hidden units, fully connected to the input and output units. However, for the first two input representations that code blob information, different units carry different types of information. A fully-connected net will have to learn this from the example data. The process can be assisted by building in suitable connectivity. A given hidden unit might contact for example only the mass input units, or perhaps some combination of types of input unit. Even with only a few hidden units, there are a vast number of possible combinations of connectivity.

For the two codings that use an array input the net would tend to pick up some of the underlying structure because of the repeated correlations between the inputs. However, it may be better to build structure into the net to reflect the topology of the input, to identify which input units are near in the image space. This may be achieved by giving the hidden units receptive fields, so that they receive input from a group of neighbouring input units. This arrangement may be continued for more than one hidden layer, to give a hierarchy with a reducing number of units with bigger RFs e.g. (Linsker, 1986; Fukushima, 1980).

The simulator allows run-time generation of nets with such complicated architectures. The different types of connectivity may be combined: for instance within a layer of hidden units one set might have large, overlapping RFs, contacting all the input units, another set might have small, non-overlapping RFs, and contact only the x and y inputs, and another single hidden unit might contact all the mass input units. There is a potential combinatorial explosion here, and experiments using a genetic algorithms to search the space are reported in chapter 6.

The nets were trained using Quickprop (Fahlman, 1989). The output coding used was a simple binary coding of identity, plus a single bit for gender. The identity was coded in four bits, using the six different ways of having two bits high. This represented a compromise between completely localised coding and a minimisation of the number of output units. A test with a one of six coding for identity actually performed slightly worse, and took 10% more cpu time. The more compact coding also allowed investigation of the role of an auto-associative net in cleaning up the output from Backprop, reported in Hancock and Smith (1989).

Results

Numerous simulations were run, with variations of numbers of hidden units and their connectivities, details of coding, learning rates etc. The details are of little consequence here, the aim being to get a feel for the effects of the various parameters. Summary results only are therefore presented.

The data set used consists of ten examples of each of six different faces, four being male. Each net was trained on nine examples of each face, and had to identify the tenth one. The training set thus contained 54 images. Each image was used for testing in turn, so that the first test set contained the first image of each person, giving 10 different training sets. Each training set was run 50 times from different start weights to give an average performance. In all, each net was thus trained and tested 500 times, a process which took anything from 2 to 55 cpu hours on a MicroVax II (the longer run times resulted from fully connected nets). During testing, the unseen images were applied and the net run forward to produce an output. If this output was closest in terms of squared error to the target then it was counted as correct.

The learning algorithm was Quickprop, using learning rate $\eta = 0.5$ and maximum weight growth rate $\mu = 1.5$. The weight decay factor r was 0.0001. Learning was stopped when all

Coding	Network type	Size	Avg. correct
Type 1	2 layer	48:5	77.5%
	3 layer, fully connected	48:5:5	71.5%
	3 layer, structured	48:14:5	82.7%
Type 2	2 layer	605:5	86.9%
	3 layer, fully connected	605:5:5	76.9%
	3 layer, structured	605:50:5	92.3%
Type 3	2 layer	400:5	96.0%
	3 layer, fully connected	400:6:5	92.9%
	3 layer, structured	400:81:5	97.9%

Table 3.11: Results for the three different codings.

bit errors were less than 0.4. Weights were initialised randomly in the range ±2.

Table 3.11 gives the overall performance for the three different coding schemes, giving the average score on the unseen images. Results are given for a 2 layer net, an unstructured (i.e. fully connected) 3 layer net, and for the best structured net found by trial and error. There is no reason to suppose these particular structures are the best possible even for these data sets, and they might well be worse than other structures tried given different data. The aim was to explore some of the possibilities, and confirm that there are gains to be had from using structured nets. In particular, maintaining topology in the hidden layer is of benefit. It also appeared that the information on orientation of the blob is of little use.

Type 1 coding.

A 2 layer net performs quite well, better than an unstructured 3 layer net. This suggests that our task is not well constrained: there is not sufficient data to train the larger net. A structured net helps to limit the degrees of freedom during training. The net which gave the best result is quite complex: 2 hidden units contacting the mass and x coordinate fields; 2 contacting mass and y coordinate; 3 each contacting only mass, length, and width; and 1 contacting both length and width.

Type 2 coding.

In this case, the structured net consisted of two separate 5x5 arrays of hidden units. Each hidden unit had a RF of 3x3, overlapping by one with neighbouring RFs. One of the arrays contacted only the distance and mass units, the other contacted distance, length and width units: no connection was made to orientation units. Overall scores are higher than for Type 1.

Type 3 coding.

The structured hidden layer here consisted of a 9x9 array of units with an RF of 4x4, overlapping by 2. The scores are the best of the three coding types, with even the 2 layer net making only 4% errors. This may partly reflect the input normalisation procedure, but also the fact that there is more information present in the input, since the blob description step has been omitted. Adding a structured hidden layer almost halved the number of errors. Even these results are not that good: the images are not very dissimilar and any human observer should manage 100% without difficulty. The variance hidden within the scores is quite high: typically 8 or 9 of the images would score close to 100%, with just one or two being much lower. It is encouraging that the particular images found difficult differed between the three coding methods. This opens the possibility of combining different codings in future work.

The relatively good results from systems with no hidden layer reflect the rather small data set for the number of input units. With such an input space many vectors will be linearly separable. The poor showing of fully connected 3 layer nets also reflects the small training set: the nets are not sufficiently constrained to generalise well.

Including topological information on the input layer is helpful. Making the structure of the hidden layer reflect that of the input layer is also beneficial, both when the structure is sentential (Types 1 and 2) and when it is topological (Types 2 and 3). It is somewhat disappointing that the Type 2 coding, which combines both sentential and topological features, does not do better. This may be due to some of the low-level details of the coding, which need looking at further.

3.5 Principal components as a preprocessor

One of the problems of the work reported in the previous section was that the nets were generally under-constrained by the data, with several hundred input units and only a few dozen training examples. As noted in section 2.2, there are a number of neural net methods for extracting the principal components of a data set. These can effect a data compression, capturing much of the variance of hundreds of inputs in perhaps only ten. Their use as pre-processors to classification nets has been suggested by a number of workers e.g. (Hyman *et al.*, 1991; Ghaloum & Azimisadjadi, 1991; Vrckovnik *et al.*, 1990; Golomb *et al.*, 1991). Such use of PCs as a preprocessor has recently been shown to have the same underlying effects as Optimal Brain Damage (Le Cun *et al.*, 1990) and weight decay (Rumelhart, 1988) in reducing overfitting (Guyon *et al.*, 1992). This section reports a short experiment to compare the relative merits of the Sanger and Oja rules described in section 2.2 for preprocessing the Type 2 image data of the previous section.

The data used were from a rather harder set than that of the previous section, produced during some of the work to be described in chapter 6. It consists of 5 examples of each of 12 individuals. This was segmented 5 ways, to produce training and test sets such that each image was used in turn for testing. The training set thus had 48 patterns. These data were processed by two different generalisations of Oja's (1982) learning rule. Sanger's rule produces the genuine principal components (or near approximations), Oja's rule produces orthogonal vectors which span the same space. Intuitively, the latter may be better for subsequent net processing, since it produces a more even distribution of variance across the units. This is illustrated in Table 3.12. The Sanger technique produces a first component with high average value and little variance (because the data have non-zero mean), the following components have averages of around zero and generally decreasing variance levels. The Oja method produces an apparently random set of averages and more similar variances.

In practice Oja's algorithm converges more rapidly than Sanger's. Both benefit from a simple annealing schedule for the learning rate. For this experiment 12 mapping units were used: η was started at 0.005 and divided by 2 every 25 epochs for 250. η was then reset to

	Oja		Sanger	
Component	Avg	Variance	Avg	Variance
1	0.364	2.48	11.479	0.37
2	7.271	1.65	0.042	4.96
3	2.499	2.57	-0.086	3.29
4	0.632	2.56	-0.001	3.32
5	1.556	1.97	0.036	2.80
6	1.422	2.33	0.065	2.13
7	-0.487	2.36	0.037	2.25
8	-4.504	2.14	0.050	1.80
9	-0.762	2.7	0.010	1.64
10	-1.074	2.9	0.057	1.54
11	0.614	2.19	-0.091	1.50
12	6.709	1.4	0.087	1.44

Table 3.12: Average values and variance of twelve components for Oja and Sanger techniques from one of the data sets

Algorithm	Stim1	Stim2	Stim3	Stim4	Stim5
Oja	440.06	477.06	485.75	493.61	475.45
Sanger	474.09	501.2	533.90	521.36	512.95

Table 3.13: Residual sum-squared-errors for training on the five data sets.

0.00001 and the system run for another 100 epochs. The residual sum-squared-errors for the two algorithms are shown in Table 3.13. Note that, given the same training schedule, the Oja algorithm has consistently lower residual error levels, i.e. it has accounted for more of the input variance.

The twelve-component outputs from each run were saved into files, together with the target identities to be used for training a Backprop net to recognise the faces. In each case the twelve images that had not been used for extracting the PCs were also processed and used as a test set. As shown in Table 3.12, the average values for some of the data sets were quite high. A second set of experiments was therefore run, where each of the twelve input fields were normalised by linear adjustment to lie in the range 0-1. Each data set was used to train a 12:12:13 Backprop net (the extra output being the gender of the individual).

Some initial tests were carried out to establish reasonable parameters for online Backprop. Those chosen were $\eta = 0.05$, $\alpha = 0.9$, initial weight range ±0.2, error limit 0.4, with a maximum of 250 epochs. Under these conditions some of the runs did not converge within the epoch limit, however, there was a generally inverse relationship between training ease and test performance, see Table 3.14.

The results shown are not very conclusive. The standard errors on the individual test results are around 1%, so most of the differences shown on individual data sets are formally significant. However, they are not consistent, so that, for instance, for both rules the normalised data performs better than the original data on some sets and worse on others. On

data	Oja		Oja normalised		Sanger		Sanger normalised	
set	test%	epochs	test%	epochs	test%	epochs	test%	epochs
1	63.3	78	71.7	235	62.5	174	64.2	206
2	85.8	208	91.7	250	85.8	189	91.7	250
3	98.3	165	100.0	250	96.7	150	99.2	250
4	79.2	143	82.5	250	82.5	150	74.2	242
5	94.2	135	93.3	250	92.5	143	85.8	230
avg	84.2		87.8		84.0		83.0	

Table 3.14: Backprop test performance and average epochs for training, average of 50 runs.

average the normalised Oja data is the best, with little to choose between the two original, unnormalised sets of data.

Much more conclusive is the improvement over a fully-connected net working on the original 605 input data. The best result obtained there was 62.1% (see figure 1.1 on page 10). All four of the results here are therefore dramatically better, at around half the error rate. Since there must actually be less information present in the reduced data sets, the improvement must come from the reduction in the size of the net. With less opportunities to over-fit the data, generalisation has been improved.

3.6 Conclusions

The coding of data is central to the performance of any computer program, and this chapter has done little more than scratch the surface of codings appropriate for neural nets. The most suitable coding will be problem-dependent, but a couple of claims will be made:

- If in doubt, keep it simple. Analogue coding performed well in the number coding experiment, and with faces the Type 3 coding that ignored the Mirage descriptive output did best. It probably benefited from having more information, but it was the Mirage thresholding process that reduced it for Type 2.
- PCA seems useful as a preprocessor. On the evidence of one experiment, the Oja algorithm that gives more similar variances across units appears better than pure PCA.

The more complex, multi-unit coding types, such as Gaussian coarse coding, show complex interactions. There will be edge effects that were ignored in the experiments, that should be addressed by varying the receptive field width. There may also be interactions with the training process. Although the increased number of inputs appears to make training more difficult by increasing the number of weights, these weights will not be independent. The redundancy of the coding means that the relevant weights must be related, which will reduce the degrees of freedom in the net. This may explain why 2-unit interpolation coding was able to out-perform simple analogue coding in one of the numeric problems.

Exploring all these interactions is beyond the work of one thesis. There is a huge range of codings already in the literature, and the best is problem-dependent: even for the two numerical tasks of section 3.3.1, interpolation and analogue coding each fared best on one.

The focus of this work moved to the problem of specifying the structure of nets by genetic algorithm. This raised the question of appropriate coding in the GAs, which is the subject of the remaining chapters.

Chapter 4

An overview of Genetic Algorithms

God is a hacker.
Francis Crick, 1987.

4.1 Introduction

This chapter is an introduction to genetic algorithms, a class of optimisation algorithms that draw their inspiration from evolution and natural selection. GAs were defined by John Holland in his 1975 book: “Adaptation in Natural and Artificial Systems” (Holland, 1975). Since then the GA community has gradually grown, mostly in the USA, with a series of international conferences starting in 1985. However, Holland’s book is rather theoretical, and a more accessible book, Goldberg’s “Genetic algorithms in search, optimisation and machine learning” (1989b) appeared only relatively recently. With the arrival of this and Davis’s “Handbook of Genetic Algorithms” (1991b), interest in GAs looks set to increase further. What follows is a review of the art of GAs, gleaned both from the literature and experimentation.

GAs are not the only optimisation algorithms to be inspired by evolution. Fogel worked on “simulated evolution” in the 1960’s (Fogel *et al.*, 1966). This work differs from the GA approach in being driven mostly by mutation. In this it is similar to the mainly German “EvolutionStrategie” (ES) model. This originated with Rechenberg (1973) and has been developed by Schwefel (1975; 1981). The two schools, GAs and ESs, developed apparently largely in ignorance of each other. The work described in the following chapters falls mostly in the GA mould. This chapter therefore concentrates on the GA model, though further reference to ESs will be made when discussing the interaction of mutation and coding. The isolation of the two schools was ended with a workshop “Parallel problem solving from nature” in 1990 (Schwefel & Männer, 1991), and the resultant cross-fertilization promises to be fruitful.

4.2 The bare essentials

A GA operates on a problem that is specified in terms of a number of parameters. For a function optimisation, these may be the values of coefficients; for the real time running of an industrial plant, the control settings; for a neural net, the numbers of units or the learning rates. One key feature of GAs is that they hold a population of such parameters, so that many points in the problem space are sampled simultaneously. The population is generated either at random or by some heuristic. The former is usual when the aim is to compare different algorithms, the latter may be more appropriate if the object is to solve a real problem. Each set of parameters may be regarded as a vector, but the traditional name is a string. Another key feature of Holland’s GA is that these are bit strings, with real or integer valued problem parameters being coded by an appropriate number of bits. The nature of this coding is functionally extremely important and is discussed further in section 4.6.1, but is not of concern in this general description. Each string is rated, by running the system that is specified. In the case of a function evaluation, this may be very quick, for an aircraft simulation (Bramlette & Bouchard, 1991) or a neural net the evaluation might take minutes or even hours. A new population is then generated, by choosing the best strings preferentially. A simple way of doing this is to allocate children in proportion to the test performance (or rather, in proportion to the ratio of a string’s test performance to the average of all the strings). With no other operators affecting the population, the result of this is that the best string increases in number exponentially, and hence rapidly takes over the whole population.

Novel structures are generated by a process resembling sexual reproduction. Two members of the new population are chosen at random, and new offspring produced by mixing parameters from the parents. In the earliest work (DeJong, 1975), a single crossover was used, where parameters were copied from one parent up to some randomly chosen point, and then taken from the other. Thus the strings ABCD and EFGH might be crossed to produce AFGH and EBCD. Much subsequent work on GAs has studied the relative merits of different recombination algorithms. The preferred form of recombination is problem and coding-dependent and some other possibilities will be discussed further below.

A second operator that introduces diversity is mutation: the value of a parameter gets changed arbitrarily. This process is not the major source of new structures: that is the role of recombination, but it serves to produce occasional new "ideas", and to replace combinations that might get lost in the stochastic selection processes. The precise role of mutation depends on the coding used in the genes and is also discussed further below.

That, in essence, is it: generate a population of parameter sets, test them against the problem, select for reproduction on the basis of performance, recombine pairs of parameter sets and mutate a few to generate the new population and start the cycle again. We shall now look at each aspect of the algorithm in more detail.

First a note about terminology. GAs are inspired by biological evolution, and exponents often borrow terms from the study of natural genetics. Some workers refer to strings as chromosomes, their natural analogue. Genotype and phenotype may be used to describe the genetic string and the decoded parameter set respectively[1]. We need to distinguish between the parameters of the target problem and the components of the genetic string. The term *gene* is often used for the components. This is strictly inaccurate, since in biology a gene is usually taken to be something that codes for a whole trait, such as blue eyes. The natural analogue of the bits in a bit string are the base pairs of DNA. However, the application of GAs has not advanced to the point where this meaning of gene would be useful. Therefore the term will be adopted here to mean the individual components of a string, while parameter refers to the target problem. A real-valued parameter might thus be coded directly by a real-valued gene, or by a number of binary genes. Possible values of a gene are commonly known as *alleles*: 0 and 1 for a bit string. The set of possible alleles is known as the *alphabet*. Finally, a distinction will be made between *crossover* and the more general *recombination*. Crossover is the traditional form of recombination, simply selecting between the parent strings and not affecting gene values. The simplest form of crossover changes from one parent to the other at a single point. Although unknown in nature, there is no reason why parent genes should not be mixed in more complex ways, such as averaging equivalent genes or parameters. These will be referred to as recombination operators.

4.3 Evaluation

On the face of it there may not seem much to discuss about evaluation of the parameter set. If the task is an artificial one, such as a function evaluation that is being used to test out the GA, then there should indeed be no problem, provided the function is deterministic. Where the function is stochastic, as many real-world processes are, there is the issue of how much to try and reduce the noise. GAs are relatively immune to noisy evaluations, compared with, for instance, gradient ascent methods that may be thrown right off course by an odd result. However, it is still naturally the case that accurate evaluations are to be preferred

[1] In most of the systems to be described here, the string thus is the genotype. Some GAs use more than one string: see diploidy in section 4.9.

to noisy ones. The accuracy can be improved by doing n evaluations and averaging, the noise decreasing with $\sqrt{n}$. However this may not be the best approach, particularly if the evaluation takes a long time. There is evidence (Grefenstette & Fitzpatrick, 1985; Fitzpatrick & Grefenstette, 1988) that it is better to do a fast, noisy evaluation and get on to the next generation, rather than spend time accurately assessing each individual. However, some averaging may be necessary, especially if, as in the work reported in section 6.2, there is a fair chance of scoring 100% on a single test. The approach proposed there is to re-evaluate only those individuals that perform well on the first test. A novel method of testing the efficacy of the evaluation procedure, by observing the elimination of penalty bits, is also discussed in chapter 6.

Another important aspect of the evaluation procedure is that it should reflect the desired target problem. One part of this is simple accuracy. Suppose the aim is to improve the design of a jet engine. The parameters might be values such as the angle and size of fan blades. Clearly the real engines won't be tested as specified, it would be done by computer simulation. However, the end product can only be as good as the simulation, if this is inaccurate, the real engine may be an expensive disappointment.

A rather more subtle aspect of the simulation has to do with constraining it sufficiently. Were GAs human they'd be regarded as devious, forever finding a cheating way to do well at the evaluation without actually solving the problem. This became apparent in some work on tuning neural net parameters (Hancock, 1989b). Bramlette and Bouchard (1991) give a nice example in their work on aircraft design. Their GA discovered that by running the engine at greater than 100% efficiency, it could effectively generate fuel! This behaviour may be summed up in a sentence borrowed from a debate on national performance assessment in primary schools: "What you test is what you get" (the acronym, WYTIWYG, should seem familiar to many users of word-processors). The argument there was that teachers would aim too much for good results on whatever tests were set. The GA has no alternative: the only information it gets is the evaluation result, usually a simple scalar value. When, for reasons of evaluation time, the test is a reduced version of the real task, it must be very carefully constructed.

One complication that may arise is the need to optimise more than one aspect of performance simultaneously, or to optimise one subject to some constraints. For instance a net may be required to do as well as possible, but quickly, or without exceeding some size. It may be possible to build such constraints into the operators that produce new strings. This is usually to be preferred, since it both avoids the problem at evaluation time and concentrates search in fruitful areas. However, such operators may not be feasible, either because it is simply very difficult to satisfy all the constraints, or because the various factors, test score and run time in the net case just mentioned, only become available after evaluation.

The standard GA requires a scalar evaluation value for the parent selection process, so the various test values and constraints need to be combined. The easiest method is some linear combination. It is here that the WYTIWYG principle becomes particularly apparent. If the balance between the components is not good, the GA will surely optimise the easiest one at the expense of the others. It may be that the only way to discover the correct combination is trial and error. A possibility that might merit investigation is to alter the balance dynamically. For instance if, during the GA run, the evaluation time dropped below some limit, the time element in the evaluation function could be reduced.

Richardson et al (1989) have looked at various ways of handling penalty functions for constraint satisfaction. It might be thought that violation of constraints should be harshly penalised. However, Richardson et al argue that this may cause the GA to fail, especially if it is difficult to satisfy the constraints. Their suggested solution is to try and construct a penalty function that is proportional to the distance of the string from feasibility, rather than simply counting the number of constraints that have been violated.

In some cases there is more than one potential measure of the same aspect of a string's performance. In section 5.4 the application of GAs to learning the weights for a net are discussed. The error of a net may be measured in a number of ways, for instance the sum squared error across all the training set, or the worst individual bit error. While the aim of training is usually taken to be minimising the squared error, the real target for a binary training set is to get each individual bit the correct side of 0.5. However, if this was set as an evaluation target when using the traditional sigmoidal output function, the GA always got stuck with all the values just above 0.5. If the squared error alone was used, the GA tended to minimise it quickly by solving the easy bits, and letting the hard ones go to 1.0 error. It was then unable to correct the remaining bits. A combination had to be used: the author has experimented with using the squared error, provided the worst bit error was below 0.9, but applying a steep penalty function for bit errors in excess of 0.9.

GAs are by no means infallible, and sometimes no progress is made on a problem. Perhaps there are too many constraints, or the area of the possible search space that gives scores significantly better than zero is too small. A possible approach used in some work on parameter tuning (Hancock, 1989b) and in section 6.5 is to alter the evaluation function during the GA run. The problem is initially made easier, perhaps by relaxing some of the constraints, so that the GA is able to make some progress. When some performance level is achieved, the task is gradually made harder. This approach makes strong assumptions about the presence of a fairly continuous path in the search space as the task changes, which may be unjustified. While a GA may be expected to do a reasonable job of finding a way past some discontinuities there can be no guarantees, though the results in chapter 6 are encouraging.

In some optimisation procedures, it is natural to talk about the optima being small values. Thus Backprop is a form of gradient descent, used for minimising errors. Others are more naturally described as hill-climbing algorithms. It makes no real difference to a GA whether it is aiming to go up or down. However, descriptions of strings as being high-ranking, or having high fitness, suggest that hill-climbing is the natural target. Except where stated otherwise, this will be the case in this work.

4.4 Selection

Having evaluated the strings, the best need to be selected in some way to form the new population. There are two aspects to this process: how to decide what proportion of the new population should come from each string, and, having decided the ideal number, how to cope with the reality of a finite population size and necessarily whole numbers of copies of any one string.

The simplest means of allocating strings to the new population is in proportion to the ratio of their evaluated fitness to the average of the whole population. Thus if a particular string has twice the average fitness, it would be expected to be chosen twice to act as a parent. This was the method used in the first thorough experimental work on GAs, reported in deJong's thesis (DeJong, 1975). While it works well enough for nicely behaved functions, it can cause problems if the function has large areas of poor performance, with localised good spots.

Once one string finds a good area, its fitness will be far above the average. It will dominate the next generation, with consequent loss of diversity, a phenomenon known as premature convergence. Conversely, towards the end of an optimisation, most of the population should be highly rated. Those that are slightly better than average get little selective advantage, and the search stagnates.

The traditional approach to this, implemented in Grefenstette's public domain GA program Genesis (Grefenstette, 1987), is to use a movable baseline for the evaluation. This is typically set to the evaluation score of the worst string, either in the current generation or within some small (5-10) window of recent generations. The baseline may be set somewhat below the worst value, to ensure that even the worst string gets some chance to reproduce. This can be important, both as a general guard against premature convergence and because poor strings may be poor because they are on the shoulder between different maxima. Galar (1989) showed that allowing poor individuals to reproduce allowed his evolutionary system to escape local maxima. The baseline re-expands the fitness scale such that, for instance, the ratio between 0 and 1 is the same as that between 99 and 100. The problem of exceptionally good strings is handled by using a scaling algorithm that ensures a constant fitness ratio, typically about 2, between the best and the worst.

A more radical approach suggested by Baker (1985) is to use the fitness scores only to give a ranking and then assign a fixed hierarchy of selection probabilities. Montana and Davis (1989) use a geometric scaling, such that the best string is assigned a fitness of say 0.9, the second, 0.9^2, the third, 0.9^3 and so on. The scaling factor can be varied during the run so as gradually to increase the selection pressure, perhaps starting at 0.95 and ending at 0.85. One potential advantage of this method is that the evaluation no longer needs to return a single scalar value. In the net structure work in chapter 6, strings are initially ranked on test score. Where two strings have the same score, quite likely given the modest size of the test set, they are ranked on elapsed time. The search thus concentrates on test score, but automatically switches to reducing time if no progress is being made, without the need to combine the scores into a single value.

A disadvantage of the method is that the selection pressure, in terms of the ratio of selection probability of best to worst, is dependent on the population size. This must be remembered when comparing different GA runs. Whitley has suggested an alternative algorithm for use in Genitor that avoids this effect (Whitley, 1989). However, this implements a linear scaling rather than the geometric scale proposed by Montana and Davis. The latter gives relatively more reproductive opportunities to the better strings.

Having decided the ideal proportions, some finite number of copies of each string must actually be chosen for reproduction. The simplest method of doing this is to add up the total fitness (whether scaled or not). Then, for each string to be selected, pick a random number between 0 and that total and work through the list of strings, summing their fitnesses until a number bigger than the random one is reached. Each string will then be chosen with a probability that reflects its share of the total fitness. The process is known as roulette wheel selection, it being equivalent to spinning a wheel where the sectors are allocated according to each string's fitness. However, Baker (1987) showed that the random nature of the algorithm can result in significant inaccuracies in the selection process. While the proportions would be asymptotically correct with increasing number of generations[2], on any one trial they are

[2] What is correct will normally vary between generations as the population changes. The total number of occasions a string is selected should become increasingly accurate as generations pass.

likely to be significantly wrong. Thus, for instance, the best string may not be picked at all. This source of noise impairs performance, and can even lead to the complete loss of important alleles. Baker suggested a more accurate algorithm, called stochastic universal sampling (SUS) that guarantees the correct whole number of offspring for each string. Fractional numbers of expected offspring are allocated pro-rata, so if 1.7 are expected, 1 will be obtained with probability 0.3 and 2 with probability 0.7. Another way of looking at the algorithm is as a modified roulette wheel, with as many, equally spaced pointers as strings to be selected. One spin only is required. Particularly in small populations this algorithm can make a remarkable difference in performance: the author has seen an order of magnitude improvement in solution time. Figure 4.6 on page 78 shows that it can even make the difference between success and failure because of the accidental loss of low frequency alleles.

4.5 Generations and Crowding

The simplest method of running a GA is to replace the whole population each generation. In this case, therefore, the generation size (the number of strings evaluated in each generation) is equal to the population size. This was the method used for most of deJong's seminal work (DeJong, 1975). A more conservative method is to ensure that the best string from the previous generation survives, by simply adding it to the pool of the new generation if necessary. DeJong calls this the elitist strategy, and he showed that it generally improves performance on unimodal functions. Note that this is not simply a hedge against an inaccurate selection algorithm, since the best string may be selected correctly, but then lost during recombination or mutation. The elitist strategy ensures the best string survives the whole generation procedure. On multimodal functions the strategy may be less beneficial, since it can make escape from a local maximum more difficult. A compromise that has been used by the author was to keep the best for a few, perhaps 5, generations, but then kill it if no further progress has been made.

The generation size may be smaller than the population, in which case some method must be used to decide which of the old population to kill off. This may be done at random, or weighted to make the worst most likely or even certain to go. An interesting alternative, intended to reduce premature convergence, was introduced by deJong (1975) when tackling a function designed to have multiple local maxima. For each member of the new generation, a small number N of the old population are chosen at random. The one with the highest number of bits in common is replaced by the new string. This effectively introduces competition between strings that are close together in the parameter space, discouraging convergence on one good spot. The strategy gave significantly enhanced performance on the multi-modal function. The required value of the crowding factor N is surprisingly small - 2 or 3 for a population of 100. If it is much larger (16, say) then the system will have difficulty converging on any maximum.

A significantly different GA model uses a generation size of just one. This was introduced by Whitley with his Genitor system (Whitley & Knuth, 1988), and termed steady-state reproduction by Syswerda (1989). Genitor is very conservative: the offspring is only added to the population if its performance exceeds the current worst, which is then deleted. An apparent drawback of this method is that the one-at-a-time selection procedure inevitably suffers from the same kind of sampling error as roulette wheel selection. This is unlikely to affect good strings, since they will in any case survive for many evaluations (until they become the worst), but may result in the worse strings getting less chance to breed than they should. This potential loss of diversity is mitigated by ensuring that there are no duplicate strings. However, the potential sampling error on poor strings combined with the very conservative retention of the good ones suggests that the system may have difficulty in escaping from local minima. This is supported by Whitley's own results on deJong's original test set (Whitley, 1989). The only function where Genitor does worse than the standard model is the multi-modal one. The general utility of the model is contentious. Radcliffe (1990b) sees little merit in the idea, and reports no empirical advantage. Davis (1991b) reports that in his experience it usually does offer an improvement in terms of performance achieved for a given number of evaluations when compared with the traditional generational model. He notes a specific exception for noisy functions, because a lucky evaluation will give a string an artificially high ranking that it will keep as long as it survives. The model was tried on some of the net optimisations of chapter 6, with no suggestion of any improvement over systems with a larger generation size.

Goldberg and Deb have recently cut through some of the empirical confusion with an analysis of a variety of selection schemes, including that used in Genitor (Goldberg & Deb, 1991). They conclude that Genitor achieves its results because of the high effective selection pressure and predict difficulties of premature convergence with difficult problems.

4.6 Reproduction and coding

This section discusses the various operators used to create a new generation from the strings selected to be parents. The key to the explorative power of GAs is held to be recombination. This claim puzzles some biologists, who maintain that the effect of sex in real genetics is exactly the opposite: it causes the population to stay genetically fairly uniform. The key to understanding the difference of opinion is to see that most biological systems are already fairly fit in their chosen niche[3], whereas most GAs start with a completely random population. An interesting biological example is given by greenfly, which breed asexually during the easy months of summer, but revert to sexual reproduction to remix the gene pool prior to the rigours of winter. The numerical argument in favour of recombination are easy to see (Davis, 1991b). Suppose two new alleles are required to bring about a big fitness improvement in a population. Such new alleles can only come from mutation, which happens infrequently, say with a probability of 10^{-6} per reproduction. If each new allele confers some advantage, then without recombination strings containing one or other will eventually appear and prosper, but still have to wait for another rare mutation to acquire both. With recombination it requires only that two strings each with one of the alleles interbreed.

Despite this, there have been claims that recombination contributes nothing to the optimisation process (Fogel & Atmar, 1990). Whether or not it does contribute usefully depends very much on how it interacts with the underlying coding of the strings. This interaction is the prime concern of this section: we turn first to the issue of coding.

[3] Though Stork (1992) argues that this should not be taken to mean that biological systems are actually optimal, since most are derived by adaptation of organs previously intended for some other function.

4.6.1 Coding

One of the important differences between the ES approach and Holland's GA is the form of coding of the parameters. ESs hold the parameters as normal computer variables: integer or real as appropriate. While some work on GAs also uses this form of coding, Holland specified a bit-string coding. Some problems contain boolean parameters for which such a coding is ideal. However, real or integer parameters may be coded with arbitrary precision by using sufficient bits. Any digital computer will have such a bit coding in any case, but the details are usually hidden from high-level languages.

Whether or not to use bit coding is a contentious issue. GA-purists tend to regard real-coded algorithms as not being proper GAs. Meanwhile more pragmatic experimenters have produced good results with real codings and suggest that if theory indicates bit coding should be superior, the theory needs revising. We shall first consider the advantages claimed for bit coding.

Access to the bit level gives the crossover operator the ability to explore the whole search space. Given just the strings containing all ones and all zeros, repeated application of simple crossover can in principle produce any desired bit pattern, and therefore any parameter values. Contrast this with parameters held directly as real values, where crossover can only explore new combinations of the values extant in the population[4]. Actually the same is true of bit-coded GAs: crossover cannot affect a bit that has the same value in every member of the population. This is the role of mutation, to replace bit values that may have been lost so that crossover may form new combinations. Real-coded algorithms depend more heavily on mutation to provide new values. Since mutation is random, it will destroy good parameter values as well as improving bad ones.

A possibly more important reason for using bit coding has to do with the way the search space is sampled. It maximises the effect known as intrinsic parallelism, a prediction of schema theory, to which we now turn.

4.6.2 Schemata

A problem with any optimisation problem is credit assignment. Suppose we have a good result: which of the parameters caused it? Most likely several in combination. A *similarity template* or *schema*, in this context, specifies some of the parameters, leaving others as "don't care" (usually shown as "$*$"). Schemata provide a way of describing the underlying similarities between successful strings. There are many such schemata contained within even a short binary string: for instance 1101 contains $11**$, $*10*$, $*1*1$: 16 (2^4) in all. All 16 of these schemata are evaluated when the complete string is. All 16 are also selected: the reproduction operators are processing not only the basic strings but also all the constituent schemata. Each schema is likely to be represented by many strings, so it is possible to work out an average score for each. Such explicit calculation is unnecessary, however, as the process is automatically handled by the selection of whole strings. Good schemata will thus tend to increase in numbers. Using f_S as the fitness of schema S, $C_S(g)$ as the number of copies in the population at generation g and $\bar{f}$ as the average fitness of the whole population, we can write an expression for the expected number of copies in the next generation:

[4] Though it is possible to hold the parameters as real values and still crossover at the bit level (Bos & Weber, 1991)

$$E(C_S(g+1)) = C_S(g)\frac{f_S}{\bar{f}}p(S)$$

The unexplained term p(S) is the probability that the schema survives the reproduction operators. The likelihood that a schema is affected by mutation depends on the number of defined bits in the schema, known as the *order* of the schema. The likelihood that a schema of order o_S survives mutation is $(1-p_m)^{o_S}$, with p_m being the probability of mutation at each bit. For the typically small values of p_m that are used, this may be approximated by $1-o_S.p_m$. Note that some users take p_m to be the probability that a bit is randomly reset, so that the chance of it being changed is actually half p_m. This is implemented in Grefenstette's public domain Genesis package (Grefenstette, 1987), though it has been changed back to the more natural use in Schraudolph's GAucsd development of Genesis (Grefenstette & Schraudolph, 1992). Readers need to be wary of this unfortunate confusion.

The probability that a schema survives crossover is a function of its *defining length* d_S. This is the distance between the first and last defined bits. Thus the schema $1 * 1 * *$ has d_S = 2: there are two possible positions where a crossover could come between the defining bits. Short schemata have a proportionately better chance of surviving crossover than longer ones. For a string of total length l, the chance of a schema surviving a single point crossover is

$$p(S) \geq 1 - p_c\frac{d_S}{(l-1)}$$

because there are $l-1$ possible positions for the cross site. The inequality is because, unlike mutation, crossover does not imply loss of the schema. In the limit, crossing two identical strings will have no effect on any schemata. So the calculated loss is a worst case.

This gives the following expression for the expected number of copies of a schema:

$$E(C_S(g+1)) \geq C_S(g)\frac{f_(S)}{\bar{f}}(1 - p_c\frac{d_S}{(l-1)} - o_S.p_m)$$

This is the *schema theorem*, dubbed *The fundamental algorithm of Genetic Algorithms* by Goldberg (1989b). Provided that the fitness of a schema is sufficiently above average to outweigh the loss terms, its proportion in the population will grow exponentially. This is most likely for short defining length, low order schemata.

One source of the power of GAs is that many schemata are being processed simultaneously. The number may be estimated (Holland, 1975; Goldberg, 1989b): after allowing for the likely disruption of schema it is of the order of n^3, where n is the size of the population. This phenomenon is known as *intrinsic parallelism*, and has been described as the only case where an exponential explosion works to our advantage.

The order of n^3 estimate hides an assumption as to the value of n, which is chosen so as to expect one copy of each schema being processed. The derivation of the estimate is given by Goldberg (1989b). The chance that a schema survives crossover is related to its defining length. Depending on the selection pressure within our GA we may set a required survival probability for schema that will be processed usefully, i.e. that will be able to increase in number if their quality merits it. By using the survival probability equations from above we may calculate a maximum useful schema length l_s. The estimate for the number of usefully processed schemata n_s is then (Goldberg, 1989b):

$$n_s \geq n(l - l_s + 1)2^{l_s - 2}$$

The order of n^3 estimate arises from assuming a population size of $2^{l_s/2}$. This is done to prevent over estimating the total number of schemata by having many copies of each in a large population. For a given small population, it is clear that n_s is highly dependent on l_s. This is the origin of the desire for a *low cardinality alphabet*, i.e. a coding where each gene has few possible alleles, preferably 2, since then the length of the string, and therefore the value of l_s will be maximal.

Unfortunately, the success of this parallel search is not guaranteed. It requires that two good genes in combination will produce a better result than either alone. This is known as the *building block hypothesis*. It's all very well rating lots of schemata in parallel, but how does the performance of a given schema, say 1****, relate to the performance of more defined schemata that incorporate it, such as $1 * * * 1$? For an extremely simple optimisation problem such as maximising the number coded by the binary string, the combination is straightforward. $1 * * * *$ will be highly rated, $* * * * 1$ much less so, but higher on average than $* * * * 0$, so its numbers should increase. If it doesn't already exist, crossover will soon produce $1 * * * 1$, which will be better than either parent.

Although tasks almost this trivial have been used for testing GAs, it is not immediately clear that the hypothesis applies so well in other problems. It is possible to test the behaviour of GAs under such circumstances by designing problems that are deliberately *deceptive* and thus might be expected to mislead the algorithm. Such deliberately deceptive problems are considered further below in section 4.10, but deception may lurk in the simplest of problems because of the binary coding. Suppose that an integer parameter happens to have a maximum at 8. If the function is smooth, then 7 will also get a good score, but its binary coding will be very different. The schema $* * 111$ may be quite highly rated, but is a red herring on the way to 01000. This "Hamming cliff" problem is familiar from the preceding discussion of neural nets (section 3.3.2), as is one possible solution, the use of Gray codes.

4.6.3 Gray codes

Gray codes have the property that the binary codings of adjacent integers differ in only one bit, see table 3.5 on page 41. For instance, the Gray code for 7 is 0100, for 8 it is 1100. This means that such changes can always be made by a single mutation. The use of Gray coding might therefore be expected to improve the hill-climbing ability of a GA. Its use was suggested by Hollstien (1971), who reported tentative benefits, and by Bethke (1981), who also reported empirical success. Caruana and Shaffer (1988) report that it improves performance on deJong's classic 5 problem test suite (DeJong, 1975). Some authors have therefore adopted Gray coding as standard (Eshelman *et al.*, 1989; Schraudolph & Belew, 1990), while it is an option on the Genesis package.

There are also arguments against the use of Gray codes, to do with the schema theorem. Goldberg (1989a) hints at a problem in his analysis of the use of Walsh codes in deception, but it is quite easy to demonstrate. With simple binary coding, a given bit always makes the same contribution to the value of the external parameter. With Gray coding, this is not the case. Thus, coding integers from 0 - 15 in normal binary, the schema $* * *0$ has an average value of 7, while $* * *1$ has an average of 8, reflecting the value of the least significant bit. With Gray coding, both schemata have the same average, 7.5. There is no longer any

Number	Function	Range	Bits per gene
F1	$\sum_{i=1}^{3} x_i^2$	$-5.12 \leq x_i \leq 5.12$	10
F2	$100(x_1^2 - x_2)^2 + (1 - x_1)^2$	$-2.048 \leq x_i \leq 2.048$	12
F3	$\sum_{i=1}^{5} integer(x_i)$	$-5.12 \leq x_i \leq 5.12$	10
F4	$\sum_{i=1}^{30} ix_i^4 + Gauss(0,1)$	$-1.28 \leq x_i \leq 1.28$	8
F5	$0.002 + \sum_{j=1}^{25} \frac{1}{j+\sum_{i=1}^{2}(x_i - a_{ij})^6}$	$-65.536 \leq x_i \leq 65.536$	16

Table 4.1: DeJong's five test functions

information about the merits of setting this bit from the overall averages, which suggests that the degree of implicit parallelism will be reduced. Such interdependence between bits is commonly known as *epistasis*, another term borrowed from biology.

The effects of a single bit mutation clearly differ between the two coding strategies. Mutating the most significant bit in a standard binary coding causes a big change in the number being represented. In Gray coding, adjacent numbers differ in only one bit, so it might appear that a single bit mutation will cause less dramatic effects. This is true if it happens to be the bit that differs that is changed. However, there are still highly significant bits in a Gray code, it's just that it isn't always the same bit. For instance, the Gray code for 0 is 0000, as with ordinary binary, but 15 is 1000. The possible big changes balance out the small ones, so that the expected average change caused by a single mutation is the same for both codings: 3.75 over the range 0-15.

If the use of Gray coding interferes with the parallel search, but makes mutation-driven improvements easier, a Gray coded GA should be more sensitive to the mutation rate. This prediction was tested by looking at deJong's (1975) test set, using a version of the GAucsd simulator (Grefenstette & Schraudolph, 1992) modified to gather more statistics. These experiments differ from those reported by Caruana and Shaffer (1988) in that they did not vary the mutation rate, instead using only the set of parameters (population size, mutation and recombination rates, etc) suggested by Grefenstette (1986).

The test set, although quite carefully constructed to include a variety of problems, is now showing its age. The five functions are given in table 4.1. They have been heavily criticised by Davis (1991a), who shows that a simple bit climbing algorithm out-performs standard GAs on all but one of them. This is because they are rather regular, for instance the optima are conveniently placed at zero in F1 and F4 and at one end of the range in F3. F5 looks horrendous, being a plane with 25 sixth order fox-holes, differing only slightly in depth. However, the holes are laid out on a regular grid, that actually makes solution rather easy since a change in only one parameter can cause the move to the adjacent hole. It seems clear from Davis's results that these functions should no longer be used for comparison of new algorithms. They are used here simply to demonstrate some of the differences caused by changes in coding strategy.

With the exception of F4, all the runs used a population and generation size of 100, the elitist strategy and a two-point crossover probability of 0.6 (if not selected for crossover, a string is passed to the next generation unaltered except for possible mutation). They were run for 4000 evaluations, or, since strings that were simply duplicates of a parent were not re-evaluated, until two generations had passed with no evaluations. F4 has a much longer

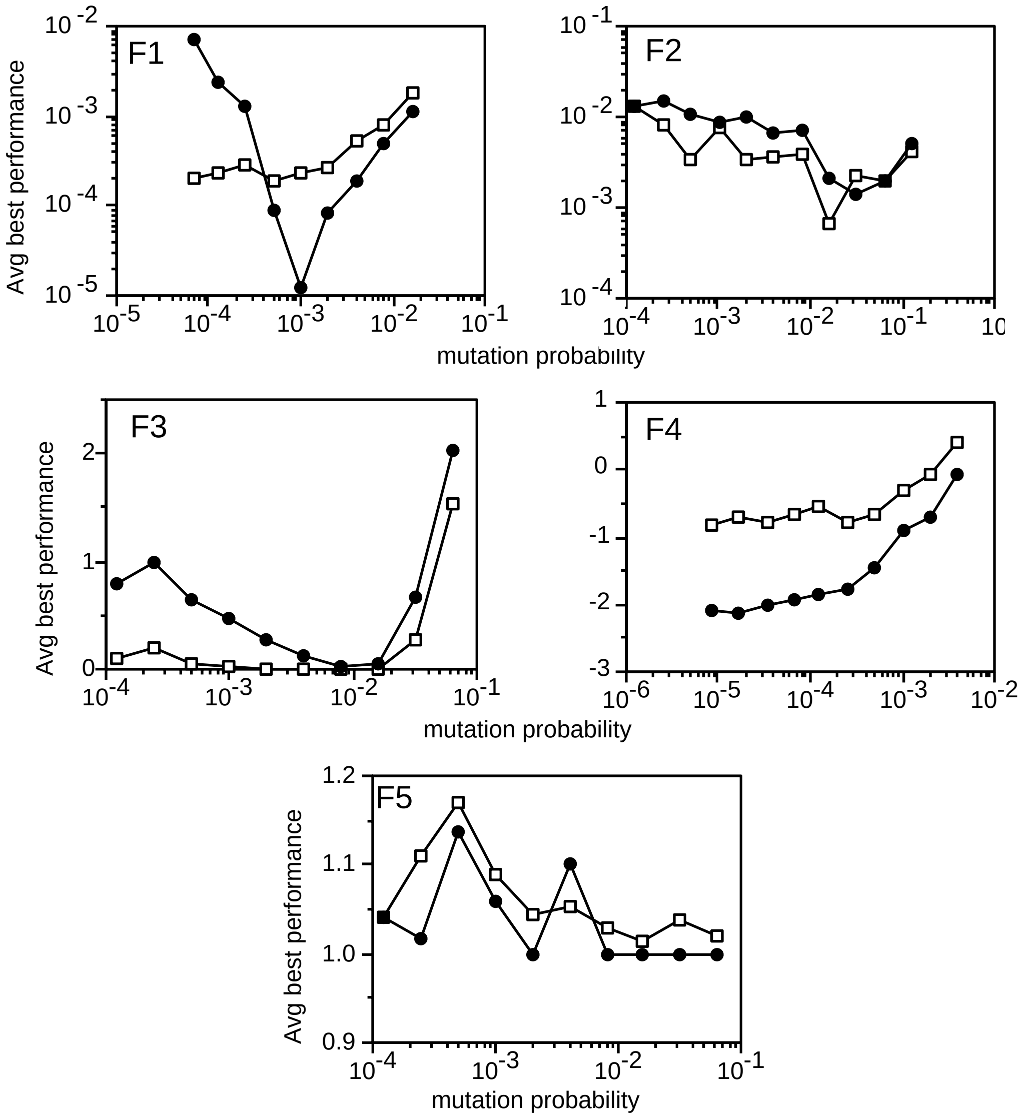


Figure 4.1: Comparison of best performance of binary □ and Gray coding • on deJong's five test functions at a variety of mutation probabilities.

string than the others, and showed a tendency to premature convergence with a population of 100. It was run with a population of 400, for 20000 evaluations, but with duplicates being re-evaluated, since the function has noise added. The results are shown in Figure 4.1. The graphs give the best value obtained, averaged over 50 experiments, at a variety of mutation rates. Note that deJong's functions are defined as minimisation tasks, so low values are good.

The most striking result is given by the simplest unimodal task, F1. The Gray coded algorithm showed a very marked dependence on mutation rate, being significantly better than normal binary coding only over a fairly narrow band, and much worse if the mutation rate was too low. This fits with the hypothesis that Gray coding is more dependent on mutation for progress, but is potentially better at hill climbing. The results for F2 and F3 show a similar mutation-dependency for Gray coding, for which the results are somewhat worse than simple binary. Gray coding appears better for F4 and F5. Note that the optimum of F4 is undefined, because of the Gaussian noise term, which accounts for the negative values shown on the graph. F4 in particular bucks the trend, since mutation appears to have no beneficial effect at all. This was confirmed by a run with zero mutation, hard to show on a log plot, which performed as well as any.

These results have been included to show a number of effects:

- Even when averaging over 50 runs, there is a fair amount of noise in the data. GAs are stochastic algorithms, and any paper claiming comparative results based on just one run, of which there are a surprising number, should be treated with suspicion.

- Mutation, on the whole, has the expected effect: too little or too much being deleterious. Too little usually results in premature convergence, too much is disruptive. However, the ideal rate varies between different problems. It is related to string length l, a reasonable first approximation being $1/l$.

- Gray coding is certainly not a panacea. Even in the simple hill-climbing case of F1, precisely the sort of problem it is intended to address, its relative sensitivity to mutation rate means that its advantage over simple binary coding is not consistent.

It is not surprising that Gray codes are not consistently better: it can be shown that all fixed coding schemes of a given length work out equally well when all functions are averaged over (Caruana & Schaffer, 1988). However, this result depends on averaging over literally all functions, including those which are random look-up tables: any given coding then simply reshuffles the entries in the table. Attempting to optimise such functions is rather futile: real-world problems usually have at least some continuity. The relative empirical success of Gray coding suggests that it matches the GA to the continuities of such problems, at least better than the inherently discontinuous binary coding.

Just as coding is related to learning rules in neural nets, so coding is intimately connected to reproduction operators in GAs. The nature of this relationship will be developed further by considering the merits of coding parameters directly as real values.

4.6.4 Real valued genes

Mutation

The arguments against holding parameter values directly as real numbers on the genetic string are the loss of intrinsic parallelism and the inability of crossover to produce new parameter values. The arguments in favour are largely empirical, with Davis and co-workers in particular reporting considerable success with real-coded GAs (Davis & Coombs, 1987; Montana & Davis, 1989). Part of the reason for this success appears to be the mutation operators employed. Consider first the effects of mutation on binary coding. A single bit mutation will produce a random scatter of powers of 2 change in the integer value encoded. Gray coding also gives powers of 2 changes, but the scatter is no longer random: there being by definition always two ways of producing a change of 1. What might be an appropriate operator for a real valued parameter? Two possibilities are immediately obvious: small changes from the given value, and completely random values within the specified range. The former should aid hill-climbing, the latter will introduce variety. Real-coded GAs often use both of these (Davis, 1989; Montana & Davis, 1989), the small change operator sometimes being known as *creep*.

Intuitively, these operators have a better feel about them than random scatters of powers of 2. These intuitions will be formalised later. However, first note that we don't have to hold the parameters directly as real values in order to implement the operators. It is quite possible to define a form of parameter mutation for binary coding strategies. It simply requires that the parameter be decoded, changed, and recoded onto the string. Such a technique has been used by Lucasius and Kateman (1989).

The effects of such a creep operator on deJong's F1 function are shown in Figure 4.2. In this experiment, the underlying coding remains binary, while four different creep operators are compared. The range of the parameter values is ±5.12, covered in 1024 steps. The four operators are: an addition of a Gaussian random variable, with an sd of 1, 10 and 100 steps, and a change by ±1 step. The performance of all four exceeds simple binary coding and mutation, with the smaller changes doing better, though not quite as well as Gray coding at its best. However, the sensitivity to mutation rate is considerably less than for Gray coding. The optimal mutation rate is around an order of magnitude higher for the simple reason that one parameter replaces ten bits.

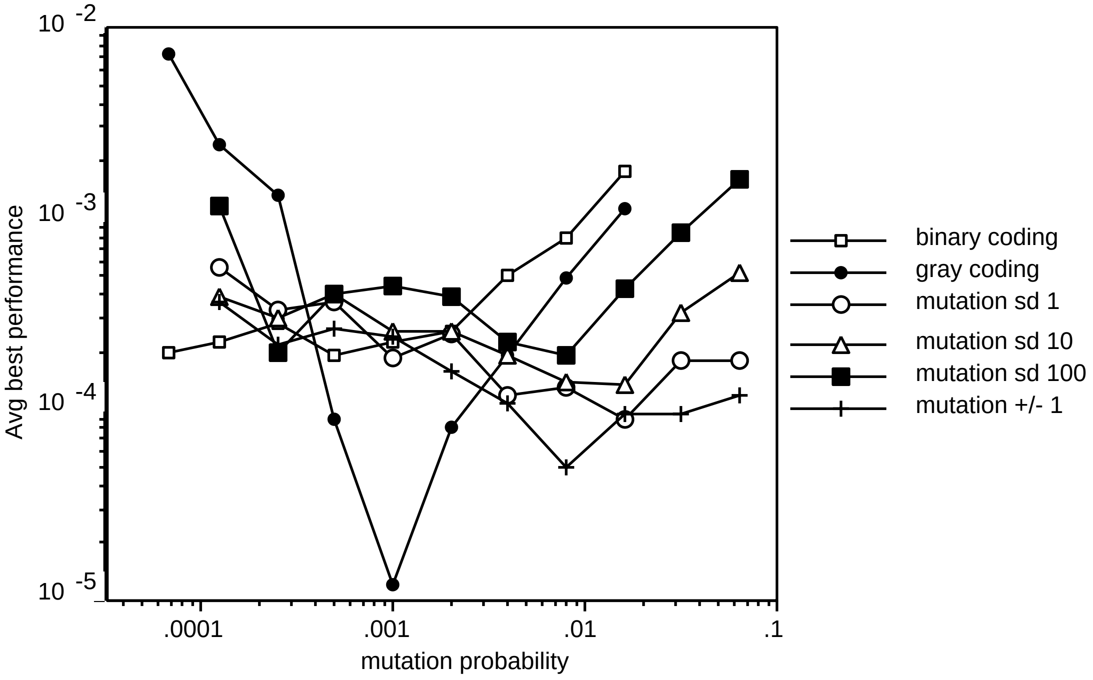


Figure 4.2: Comparison of different mutation methods on F1: binary and Gray coding with simple bit mutation, and binary coding with four different creep mutation operators.

Part of the criticism aimed at deJong's test suite by Davis (1991a) is that the minima are often rather conveniently located. For F1 it is at 0, which is coded by a suspiciously easy-looking 1000000000 in the binary coding. Davis suggests shifting the functions, so that the minimum is displaced by 10% of the range, to 1.2 in the case of F1. This is done by subtracting 1.2 from the decoded parameter value: values that come out below -5.12 are simply wrapped around to fill the gap above 3.92. This has no effect on the function, only on the string coding of each location on its domain. With no change in the function, one might hope there would be no change in performance.

Results for binary and Gray coding with bit mutation, and binary coding with ±1 creep mutation are shown in figure 4.3. Gray coding indeed shows little change, supporting the conclusion that it is operating largely by creep mutation. Straight binary coding shows a marked deterioration at low mutation rates, confirming Davis's results and supporting his opinion that the original F1 is particularly convenient for crossover with binary coding. However, the most spectacular change comes when the creep operator is added. At low mutation rates it is less successful on the shifted function, as in the normal binary case, but at higher rates it finds the global optimum every time. There is no obvious explanation for this effect: it must be some fortuitous interaction between crossover and the creep operator. However, it does clearly show the potentially dramatic effects of a simple coding change. Davis now recommends that any function used for testing GAs should be shifted to several random positions to help eliminate such interactions from comparative results (Davis, GA-digest, 1992).

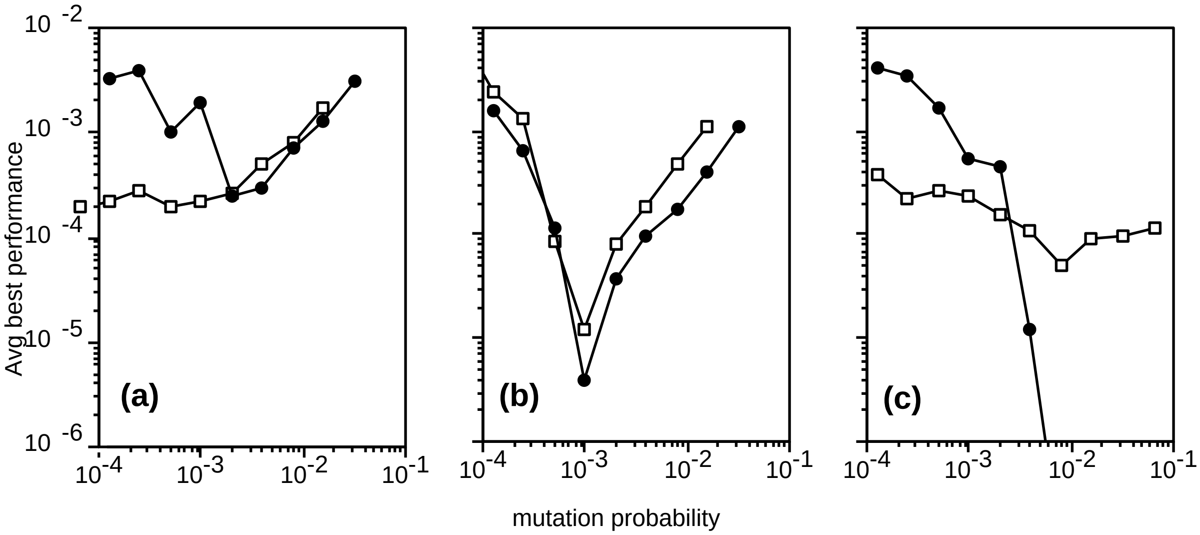


Figure 4.3: Comparison of best performance on F1 (□) and Shifted F1 (•), for (a) binary coding and mutation, (b) Gray coding, (c) binary coding with ±1 creep mutation.

Such a procedure seems profoundly dissatisfying. We should surely rather seek methods to analyse and remove the interaction. Otherwise any application of GAs remains subject to trial and error. In this case much of the problem seems to be due to the underlying binary coding. So we return to the possibility of holding parameters directly as real-valued genes.

Virtual alphabets

Goldberg (1990) has discussed arguments for and against real coding. He argues that selection by the GA rapidly reduces the range of parameter values present in the population, to form a *virtual alphabet*, which is then used for further processing. His reasoning may be summarised as follows. In the early generations of a GA, each parameter can be treated individually, since there has not been time to collect much information about combinations of parameter values. An average fitness can be calculated (in principle) for all values of each parameter, by integrating over all values of all the other parameters. Goldberg calls this a *mean slice*.

Unless the parameter has no effect on the function[5], some parts of its range will be better than others. Goldberg and Deb (1991) show that these above-average regions will come to dominate the population very quickly: in the order of $log\,log\,n$ generations, where n is the population size and log is in base 2. Thus for a typical population of 100, only above-average alleles are left after 3 or 4 generations. Note that an allele in this case may be a sizeable region of the parameter's range, and that there may be several disconnected regions within the range. After the initial selection has taken place, the action of crossover is limited to exploring combinations of these sub-ranges, figure 4.4. Goldberg argues that, presented with a high (infinite) cardinality alphabet, the system effectively produces its own lower-cardinality virtual alphabet, one specifically tailored to the problem in hand. This may explain the empirical success of real-coded GAs.

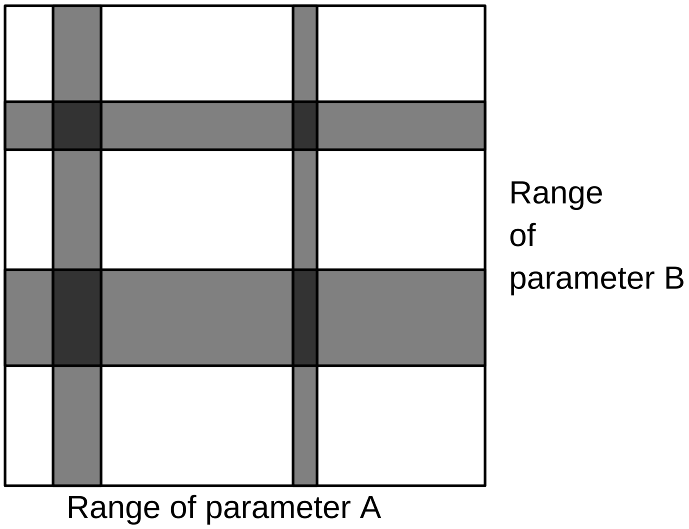


Figure 4.4: Simple crossover with a virtual alphabet. After the first few generations, the parameter values become restricted to the grey areas. Crossover can then only explore the intersection of these areas.

Goldberg then goes on to point out that for some functions, the initial, individual parameter fitness averages may not hold the global solution. Figure 4.5 is an example of a function that is designed to confuse such a real-coded GA. The central maximum is too small to have much effect on the global averages, so in the initial few generations the population settles into the two broad humps. Thereafter, no amount of crossover will reach the central peak. Creep mutation will also have a hard time, because of the ring fence of local maxima. Goldberg describes such functions as being *blocked*, and argues that this is a potentially serious shortcoming of real-coded GAs.

However, this blocking presupposes the traditional form of crossover, that cuts strings between genes. Just as for mutation, we may ask what is an appropriate form of recombination for real-coded genes. Again, an obvious possibility works quite differently: cross the gene values to produce a new value somewhere in between those of the parents. Such a recombination works extremely well on the function shown in figure 4.5, because the global maximum is conveniently situated in the centre of the range, between the two broad local

[5] Or it interacts with the other parameters in such a way that the average fitness comes out level.

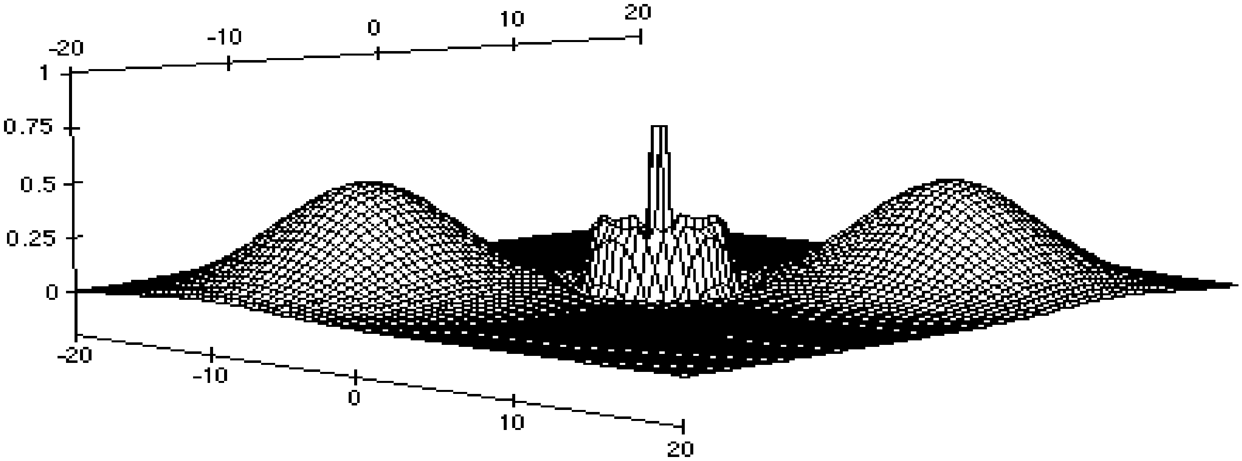


Figure 4.5: Example of a function that might, by Goldberg's analysis, cause problems for a real-coded GA.

maxima. Radcliffe reports that this recombination, that he calls *flat crossover*, along with a mutation operator that introduces new alleles at the end of a gene's range, works better than binary coding on the first four of deJong's functions (Radcliffe, 1990a). Again this is not surprising, given their conveniently situated maxima. However, it should be clear that the form of recombination operator can dramatically affect the behaviour of the GA and has the potential to overcome Goldberg's concern about blocking.

Formae

Radcliffe suggests this operator as a result of an analysis that extends the traditional notion of schemata. Schemata are intended to capture underlying similarities between successful strings, but we have seen that the nature of binary coding results in a failure even to recognise the similarity between, for instance, 7 and 8. What is needed is: a) a method of describing similarities in the target problem, whether it be real valued parameters in a function optimisation, or, as will be discussed in the next chapter, the structure of a neural net; and b) ways for the GA to process these similarities in a way that allows their constructive combination.

Radcliffe (1990b; 1990a; 1991) and Vose and Liepins (1991) have extended the notion of schemata in similar ways, although with very different formalisms. Radcliffe terms this more general notion of similarities in the target problem *formae* and identifies some requirements of operators that will process them. Two formae are termed *compatible* if it is possible for a string to be a member of both, i.e. their intersection is non-zero. Two of the key requirements, together with informal illustrations from (Radcliffe, 1991), are as follows:

- *Respect*: "Crossing two instances of any forma should produce another instance of that forma. [If both parents have blue eyes then all their children produced by recombination must have blue eyes.]" If recombination does not respect formae, then it will be difficult even to identify good building blocks, since they are liable to be destroyed.

- *Proper assortment* "Given instances of two compatible formae, it should be possible to cross them to produce a child which is an instance of both formae. [If one parent has blue eyes and the other has brown hair it must be possible to produce a child with blue eyes and brown hair as a result of the cross.]" This requirement allows building blocks to be brought together constructively.

Respect, in this sense, may seem almost inevitable: if you cross $1*$ with $1*$, surely you must get $1*$. Certainly traditional crossover respects traditional schemata, but in the next chapter we shall see that it is actually very difficult to devise a recombination operator that respects the similarities of neural net structures. Proper assortment is less straightforward. Radcliffe gives an example of strings 1010 and 0101, members of schemata $1*1*$ and $*1*1$ respectively. These schemata are compatible, with intersection 1111, but single point crossover cannot produce this result. An alternative crossover that can is *uniform crossover*, which simply takes each gene from either parent randomly. Traditional analysis has frowned on this crossover, since it is likely to disrupt schemata. However, in an empirical comparison, admittedly using the deJong test set, Eshelman et al (1989) show that uniform crossover performs better than single point. However, a compromise, that crosses in 8 randomly selected positions on the string, came out best.

The term *proper assortment* is perhaps somewhat unfortunate, since it disguises many degrees of propriety. By the definition, for a recombination operator to assort formae properly requires only that it is *able* to produce a child which is an instance of both parent formae. Other things being equal, a recombination operator with the greatest likelihood of achieving this seems preferable. This point will also be important in the consideration of operators for processing neural net structures in the next chapter.

Radcliffe (1990a) suggests plausible similarities for real-valued genes are described by intervals specified by a value and a radius, {c, r}, such that $c - r \leq x \leq c + r$. This does indeed capture an intuitive notion of similarity. Radcliffe goes on to show that standard crossover with real genes will not properly assort formae built from such similarities, but that flat crossover, described above, does.

A severe problem with this recombination operator is that, even with no fitness differences between any strings, the population will rapidly converge, on the centre of the range of each parameter. To help counteract this Radcliffe also suggested *end point mutation* so as to re-introduce alleles at the ends of the parameters' ranges. However, in practice the convergence is so strong that the operator needs to be used rather sparingly.

Radcliffe's flat crossover treats each gene pair individually. It is also possible to produce a recombination operator that works at the level of the whole string. Wright (1991) suggests crossing two points p1 and p2, to yield three children: $\frac{1}{2}p1 + \frac{1}{2}p2$, $\frac{3}{2}p1 - \frac{1}{2}p2$ and $\frac{3}{2}p2 - \frac{1}{2}p1$. The first of these is the mid point, the other two lie on the line defined by the two parents, but outside of either. All three new strings are tested, and the best two incorporated into the new population. On deJong's functions, plus three others, this *linear crossover*, in 50:50 combination with a two point version of traditional crossover, performed significantly better than both Gray coding and real coding with just the traditional crossover (Wright, 1991).

Real valued mutation revisited: the ES approach

Mutation of real-valued parameters lies at the heart of the ES strategy. A gene is usually altered by adding a Gaussian random variable. The critical point is that the spread of the Gaussian is held as a parameter, usually one for each gene, which are themselves subject to mutation. The ideal size of mutation will vary: large if the fitness value changes slowly with a change in the parameter value, small if this derivative is large. Rechenberg identified a guideline for changing the mutation size that appears to work well in practice: if more than 1 in 5 trials leads to an improvement, increase the variance, if less, decrease it. However, allowing the variances to evolve with the string should change them automatically, since a string with the best step size is most likely to improve.

Unfortunately, there is a potential drawback to this approach. If one of the variance parameters gets set to zero, then the associated variable will become fixed. This string is now searching a smaller space and consequently can, at least for a while, make faster progress, eliminating competitors. However, eventually it is bound to get stuck. One answer is simply to put a lower bound on the step size, another is a side effect of introducing *correlated mutation* (Schwefel, 1981). Just as it was possible to treat the whole string as a point for recombination, so it is for mutation. Large step sizes are possible in directions where the derivative of the fitness surface are low, but there is no reason in general to suppose these directions will be along any of the parameter axes. By introducing another set of control parameters, mutations in a number of parameters may be correlated, to give movement in an off-axis direction.

Such sophisticated mutation operators make ESs effective mutation-driven hill-climbers. Rechenberg (1989) gives a very simple program for an ES without correlated mutation in just a few lines of BASIC. Tried on deJong's F1, the initial parameter settings gave a performance slightly better than the best of the GA runs reported above. A bit of fiddling with the step-size change parameter resulted in the system consistently converging to around 10^{-39} within 5000 evaluations, this being the real number precision limit of the computer used. No real problem will be this easy, but it does seem evident that GA practitioners may have something to learn from the ES approach.

4.6.5 Adaptive coding

A problem with coding real variables in a fixed number of bits is the limit on precision that results. When coding a problem for a GA, a sensible designer will restrict each parameter to a reasonable range, but there will still be some compromise between covering this range, and sufficient precision in whatever turns out to be the important part of it. Schraudolph and Belew (1990) have suggested a solution, which they call *dynamic parameter encoding* (DPE).

Genes initially code for the whole of the parameter range. However, when a gene converges sufficiently on one part of the range, the coding automatically "zooms in" on this area. The allocated number of bits is thus brought to bear on a reduced parameter range, increasing the available precision. The process may be iterated as the GA homes in on the best area of each parameter. This method means that each gene can use fewer bits, resulting in faster operation and convergence of the GA. Schraudolph and Belew report encouraging results, again using deJong's test set, but there are attendant risks, since it is possible to narrow the search too quickly and miss something important. Thus their performance on the multi-modal F5 was worse than without DPE, because there was insufficient resolution to find the correct hill to climb.

A more complex adaptive coding strategy has been suggested by Shaefer (1987). His system, known as ARGOT, dynamically adjusts the parameter range coded by each gene. If the population clusters in a small part of the range, the boundaries are drawn in, much as in DPE. However, they may also move out if the population is widely distributed. If the population approaches one end of the range, the boundaries are shifted to re-centre it. The boundaries may also be "dithered": moved randomly by small amounts to effect a general mutation. Finally, the number of bits used may be changed, depending on the degree of convergence. Shaefer's results, for a number of function optimisations, indicate that the adaptive strategy compares well with a simple GA approach. However, it is obvious that there are many parameters associated with decisions about changing the coding, and these are not specified. It seems likely that different adaptive strategies would be necessary for different problems. Referring to the range expansion, Schraudolph and Belew (1990) comment that they "believe it would be impossible to establish a well-founded, general trigger criterion for this operator". Nevertheless, such adaptive methods clearly have potential.

4.6.6 Conclusions

In this section we have looked at some possible codings and reproduction operators for numeric parameters. Many optimisation problems can be expressed purely in numerical terms, but there are also many that cannot, particularly order-based tasks such as the travelling salesrep problem, while the coding for neural net structure used in chapter 6 is largely boolean. The same underlying rule will apply: it is necessary that the reproduction operators can process similarities in the task in such a way as to combine useful building blocks. The real art of applying a GA to a task is therefore:

1. To identify the potential building blocks in a problem.

2. To design operators, principally recombination, that can process these building blocks.

Davidor (1991) has counted 56 different recombination operators in the literature, some more will be introduced in chapter 6. It may seem unfortunate that so much design effort is needed for each new application of a GA; it may also be difficult to identify what the suitable building blocks are. One approach to this problem is to adapt the probabilities of individual operators online. This allows a number of different recombination and mutation operators to compete: the algorithm will use the ones that allow progress to be made. This also addresses the problem of the sensitivity to operator probability displayed in figure 4.1 and one approach will be discussed in the next section.

4.7 Tuning GAs

Optimisation tasks of the sort considered in this chapter consist essentially of two parts: a search of the parameter space and hill climbing. There are numerous techniques for the second job (see Schwefel (1981) for a comparative review): one approach to the first is to restart repeatedly from different positions. The two phases are traditionally called *exploration* and *exploitation*. GAs have the ability to do both. Some researchers seek to advance on the Holy Grail of a universal optimisation algorithm, that will cope with any fitness surface. DeJong's work aimed to provide a set of parameters for a GA that are reasonably robust, but such general algorithms will inevitably be beaten on any one problem by an algorithm that is tuned to the task.

The effect on the balance of exploration and exploitation of a number of GA parameters may be summarised:

Population size A small population will tend to converge more rapidly.

Generation size Changing only a fraction of the population each generation increases inertia, preventing convergence.

Mutation rate Depends on the size of mutation. Big mutations encourage exploration, small mutations can be a means of hill climbing.

Recombination rate Higher recombination rate encourages exploration while the population is diverse, but reduces it when the population has converged.

Selection pressure If selection pressure is increased, either by scaling fitnesses or by a high value for the scaling factor in rank-selection, hill-climbing will be encouraged.

Crowding Maintains diversity, thus promoting exploration.

Elitist strategy If the best individual from the previous population always survives, hill climbing is encouraged.

One possible method of matching these parameters to a given problem is to use a meta-level GA to tune them. The meta-GA specifies a population of GAs that act on the target problem. These are evaluated (see section 4.8) and the information used to improve the match of parameters to the task. CPU demand rather rules out this approach for any real-world task, but Grefenstette (1986) was able to provide an improvement on deJong's parameter set for his 5 functions.

Other workers have attempted introducing controls within the GA, which monitor convergence of the population and adjust control parameters accordingly (Shaefer, 1987; Whitley *et al.*, 1990). Goldberg, who rather seeks the Holy Grail, argues against such "central authority" in his delightfully named "Zen and the art of Genetic Algorithms" (Goldberg, 1989a), on the grounds that it is not easy to establish robust criteria for making any adjustments. However, Ackley (1987) reports empirical success with an ingenious system he calls "Stochastic iterated genetic hillclimbing". This implements a kind of voting system, such that the algorithm climbs up a hill until it effectively gets bored with it, whereupon it goes off to find another hill to climb.

Tanese (1987) has suggested a multiple population GA, intended for running on separate processors, where the different populations have different parameter settings, the hope being that one will be near the ideal for the problem in hand. The author has experimented with a more deliberate approach that has just two populations. One, the "tortoise", is large, with a small generation size, low selection pressure and high crowding factor and mutation rate; this feeds a small population, the "hare", with no crowding, small mutation and high selection pressure. This model introduces yet more parameters, not just those for the two populations, but those for deciding which strings to transfer and when. Results, even on a multimodal problem, suggested that the extra complexity was of doubtful worth.

4.7.1 Adapting operator probabilities

An alternative approach to the tuning problem, mentioned in section 4.6.6 is to adapt operator probabilities while the GA is running. One approach to this is to code, say, the mutation rate on the genetic string, where it will be selected along with the target parameters. This is effectively what is done in an ES. Another, suggested by Davis (1989) takes a rather more interventionist approach of keeping a record of the improvement in fitness caused by each operator, and using this score periodically to adapt the probability of applying each operator. However, there is a credit-assignment problem, since operators often work in harness. For instance, mutation might generate an important new allele, but recombination might combine it with others to produce the good string. If credit is given only to recombination, mutation might die out, whereupon the search stagnates. So credit is handed back, like a bucket-brigade, through parents, to the operators that produced them. Montana and Davis (1989) use this method to good effect in evaluating potential operators for use in training neural net weights. It has the advantage that different operators may be of value at different stages of the search, the adaptive procedure allows those that are contributing most at any point to be selected. It also allows different operators to be compared: those that contribute little die out.

4.8 Evaluating GAs

Traditionally the performance of GAs, and other optimisation techniques, has been reported in terms of *online* and *offline* performance. The latter is the average of the best individuals in each generation, the former refers to the average performance of all the strings since the start of run. This is of particular relevance when the system being optimised is a real-time one, like running a plant, and where getting it wrong costs something. Another measure is *best-yet*, simply the best performance so far seen, which is usually the figure that is of concern, certainly in the work reported in the following chapters. An alternative is the number of evaluations to achieve a given performance. That it is evaluations, not generations, is important, since for any complex problem evaluations are expensive. Sometimes authors fail to notice the significance of this. Herdy (working with an ES, but the principle is the same) reports results for a system with a variety of generation sizes, from 1 to 40 (Herdy, 1991). The size 40 system requires 134 generations to completion, which is claimed to be better than the single string system, which takes 3072. In fact, the single string system requires fewer evaluations than any of the others, the result of a very simple hill-climbing task.

A complication arises because of the inherent noisiness of GAs. As has been noted, it is important to average over a number of runs. Even then, comparison is complicated by the typically non-normal distribution of results, especially with multi-modal functions. Just as with neural net training, some runs converge rapidly on the correct maximum, while others may get firmly stuck on a local peak. As before, what is good will be problem-dependent, but it seems more likely that an algorithm that consistently finds the global maximum will be better than one that does so on average more rapidly, but sometimes fails altogether. As ever, it is important to specify the test conditions fully.

An interesting alternative method of comparing GAs is to let them compete directly. Fogel and Atmar (1990) took this approach in comparing a GA with crossover to one without. Rather than run them separately, they divided the population of strings into those that would use crossover and those that would not. If crossover provided a selective advantage, those strings should prosper at the expense of the others. In fact the opposite occurred, a result of the particular problem chosen. This method of comparison should be highly effective, since small differences in efficiency will be amplified by the selection process. An example is shown in figure 6.7 on page 126.

4.9 Extensions to GAs

One of the entertaining aspects of working with GAs is the degree to which it is possible to incorporate other ideas from natural genetics. While not of immediate relevance to the work discussed in the following chapters, a few of these are worthy of note.

Parallel GAs GAs have been described as embarrassingly parellelisable, by which was meant that it is so obvious that they can be run in parallel that one gets no credit for saying so. The most obvious method is to distribute the function evaluations across processors. With most real-world problems, most of the computation is spent on evaluation, so this is quite efficient. Robertson and Sharman (1990) have suggested an interesting alternative selection model that appears to work quite well and avoids the excessive communication overhead of the usual method for cases where the function evaluation is rapid. However, a number of reports indicate that it is more efficient to have several smaller sub-populations that are relatively isolated, exchanging only occasional strings (Cohoon *et al.*, 1987; Tanese, 1989; Whitley & Starkweather, 1990). This mimics the effect of natural populations separated by oceans or mountains

Inversion was suggested by Holland (1975) as a method of combating the disruptive effects of crossover on long schemata. The idea is randomly to reorder the genes on the string (while maintaining a tag so that it is known what each gene does!), in the hope of bringing useful genes close together, to form small building blocks. Goldberg (1990) reports an advantage with suitably constructed functions. However, Whitley (1991) suggests that merely tagging the genes, and giving the initial population random orderings, may be equally effective, since the usual processes of selection will cause the helpful orderings to dominate.

Diploidy In the terms of natural genetics, the GAs we have considered so far are haploid - they have a single genetic string. Most higher forms of life are diploid: there are two copies of each gene. The original reason is to guard against mutation: it being unlikely that both genes would be damaged. Where they differ, one is usually dominant: thus the human gene for blue eyes is recessive and only expressed if specified by both genes. A side effect of this is to carry a reservoir of genes that may have proved useful in the past, but would otherwise be weeded out by selection. The dominance system allows such genes to be shielded, against the possibility that they will later become useful. The classic natural example of this is the peppered moth, which switched from light to dark form as the industrial revolution killed the lichen on the trees. The principal application in artificial systems is also to track a changing, particularly a cyclically changing, task (Goldberg & Smith, 1987).

Darwin vs Lamarck In biology it is generally accepted that learned characteristics are not inherited. There are occasional odd results, such as Hall's (1991) observation of apparently directed mutations in bacteria, but these can be explained without invoking Lamarckian inheritance of acquired characters (Foster, 1992; Mittler & Lenski, 1992). In computer simulation, it is quite possible to allow phenotypic learning to influence the genotype. One approach, advocated by Mühlenbein (1989) and Davis (1991b), is to incorporate local hill-climbing operators suited to the problem. Davis specifically advocates approaching an optimisation problem by taking an existing algorithm and grafting genetic techniques onto it to form a hybrid, suggesting that the results are likely to outperform either alone. For the optimisation of neural net weights, Montana and Davis (1989), use Backprop as a local hill-climber.

4.10 Deceptive Problems: a look at a GA in action

This short tour of GA theory and practice will be concluded with a detailed look at an example problem, one specifically designed to study the behaviour of a GA in difficult circumstances. The central claim of GA theory is that they work because they identify highly performing building blocks and can then splice them together. It is possible, however, that combination of these building blocks will move away from the global optimum. Goldberg (1987) describes such problems as deceptive. He defines the minimal deceptive problem (MDP), which has only two bits. He identifies two types, I and II, which differ in difficulty, with type II being harder. An example of such a problem, discussed by Goldberg, sets the following fitness values for the four strings: 11 = 1.1, 10 = 0.1, 01 = 0.9 and 00 = 1.0. 11 is the global optimum, but the average value for the schema $1*$ is 0.6, while the average for $0*$ is 0.95. The schema theorem predicts that the latter should therefore increase in number at the expense of the former. The averages for $*1$ and $*0$ are 1.0 and 0.55 respectively (it isn't possible to get deception on both bits while maintaining 11 as the global optimum). The best building blocks appear to be $0*$ and $*1$, but their combination yields a sub-optimal string.

With such a simple system, it is possible to write out the equations for gain and loss of each of the four types of string and then perform a numerical simulation of an ideal GA. This is what would happen in a GA with an infinite population and, therefore, no need for mutation (in an infinite population, no allele would be lost). The results are consequently deterministic, and the results from such a simulation for this type II problem are shown in figure 4.6(a). After an initial decline in the proportion of the optimal string, it climbs back to take over the whole population.

Computational reality demands finite populations, and typically a number in the order of 100 is chosen. As noted in section 4.4, this will introduce stochastic effects. The inadequacy of the simple roulette wheel selection algorithm becomes clear in figure 4.6(b-d) where its behaviour on the same MDP is compared with the Baker SUS algorithm. The graphs are averaged from 100 runs of a real GA using a population of size 60. The Baker selection algorithm produces the results expected from the numerical simulation. The roulette wheel algorithm

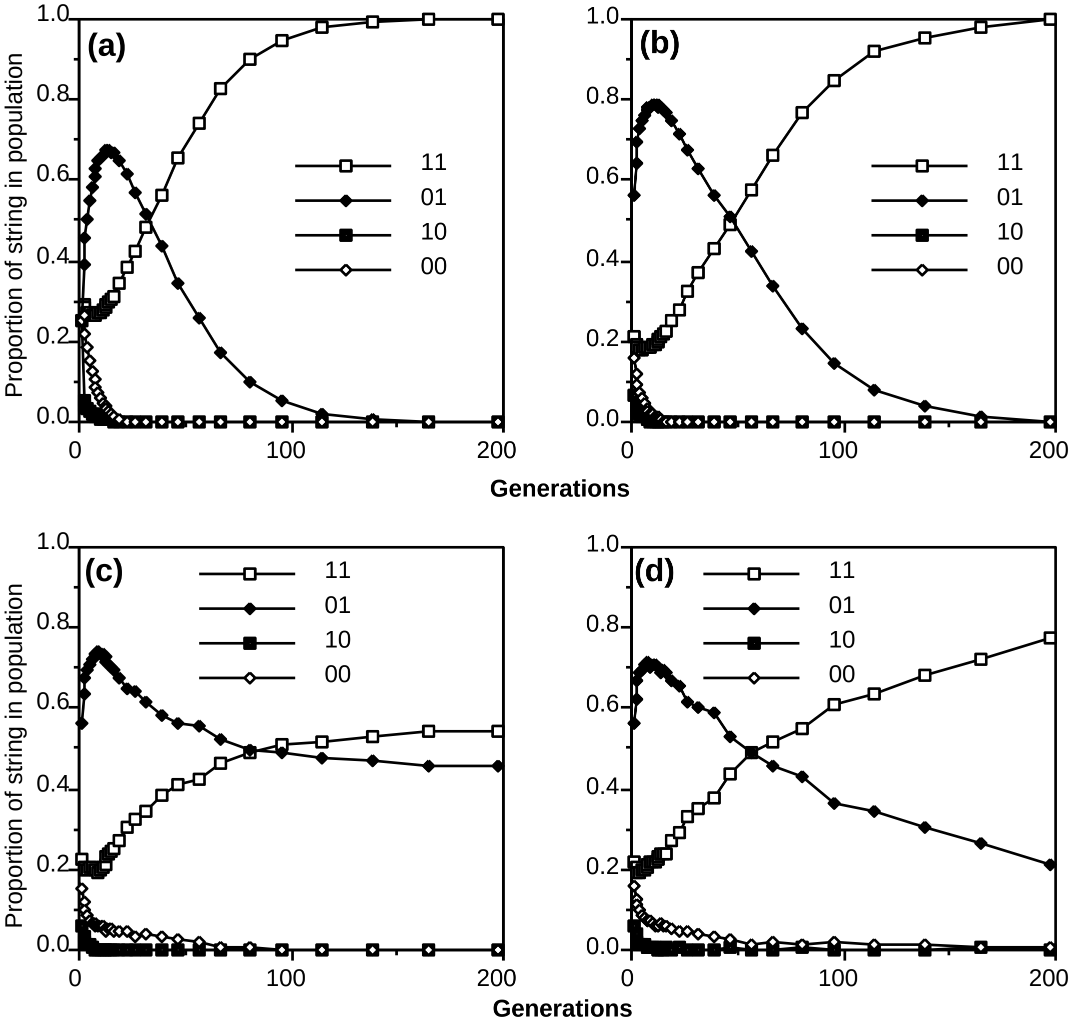


Figure 4.6: (a) Numerical simulation of a minimal deceptive problem, after Goldberg (1987). (b-d) Results of running real GA on same MDP using (b) Baker selection procedure, no mutation, population 60. (c) as (b), but using roulette wheel selection. (d) as (c) but mutation probability 0.05.

does not. The roulette wheel results come from an averaging of two outcomes: the correct solution, and one where the errors in sampling result in the sub-optimal 01 string taking over the population. Introducing mutation allows important lost alleles, in this case 1 in position 1, to be reintroduced, thus mitigating the effects of such sampling errors (figure 4.6(d)).

As noted in the discussion about figure 4.1, too much mutation will slow down convergence by disrupting good strings. However, using a very different analytical method, Vose(1990) claims that a GA will converge on the sub-optimal solution of an MDP when mutation is included. Goldberg, and the results shown above, consider only the effects of crossover. With such a simple problem, it is possible explicitly to compute the effects of mutation in a theoretical, infinite population. The schema theorem gives an estimate for the proportion of a string in a new population. However, it is a lower bound, because it ignores the gains caused by crossover and mutation of other strings. The loss of a string due to crossover may be obtained by noting that only a cross with one of the other three strings will effect a change. Thus crossing 11 with 10 or 01 will result in the same pairs of strings, while crossing 11 with 00 will result in 10 and 01. This leads the following expression for the proportion of the 11 string:

$$P_{11}(g+1) = P_{11}(g)\frac{f_{11}}{\bar{f}}(1 - p_c P_{00}(g)\frac{f_{00}}{\bar{f}}) + p_c P_{01}(g)\frac{f_{01}}{\bar{f}} P_{10}(g)\frac{f_{10}}{\bar{f}}$$

Similar expressions are obtained for each of the other strings, and it is the numerical simulation of these equations which gives the result shown in figure 4.6.

The gains and losses due to mutation can also be calculated explicitly for this simple problem. A string will be changed if there is a mutation in either or both bits: it will therefore survive mutation with probability $(1-p_m)^2$. It will gain due to appropriate mutations in other strings. Denoting the proportion of a string selected for reproduction by S: $S_{11} = P_{11}(g)\frac{f_{11}}{\bar{f}}$, we can then write the full expression for the proportion of the string 11:

$$P_{11}(g+1) = S_{11}(1 - p_c S_{00})(1 - p_m)^2 + p_c S_{01} S_{10} + p_m(1 - p_m)(S_{10} + S_{01}) + p_m^2 S_{00}$$

Similar expressions give the proportions of the other strings. These equations can be simulated numerically as before. Vose (1990) considered an MDP with string fitnesses 11=4.11; 10=0.1; 01=4.10; 00=4.0. Without mutation, this converges successfully to the global optimum, though very slowly due to the small difference in string fitness between 11 and 01. With a mutation probability of 0.01, the 01 string is dominant, as shown in figure 4.7, which confirms Vose's analytical result. The main reason for the change of behaviour is that mutation of the initially dominant 10 string produces a continuing population of 00 strings. This is the string that crosses with 11 destructively, sufficiently to overcome its slight fitness advantage. However, even if the 00 strings are removed artificially, the 11 string cannot take over the population, due to continuing loss from mutation. The same effect occurs with the string fitnesses of Goldberg's example, but the mutation probability required to prevent correct convergence is rather higher, at around 0.08.

Goldberg states: "it is surprising that all Type II problems converge to the best solution for most starting conditions" ((1989b), p51). It is indeed surprising, since they are specifically designed to mislead a GA. The reason for the success is that the GA does not "know" the true schema fitness, only the current average in the population. When the GA is started with equal populations of all strings, the true average and the population estimate of the schema fitnesses coincide, and the overall proportion of $1*$ does decline. However, the very low fitness of 10 causes that to be lost rapidly, so that the representatives of $1*$ become skewed in favour of 11. For string fitnesses of 11 = 1.1, 10 = 0.1, 01 = 0.9 and 00 = 1.0 as in the first example above, the true schema values are 0.6 for $1*$ and 0.95 for $0*$. However, when the ratio in the population of 11 to 10 reaches 6:1, the estimated average for $1*$ will be 0.957, better than $0*$. At this point in the run, the proportions of 01 and 00 have also changed, but in a direction such that the estimated value of $0*$ is actually lower than 0.95. Thus it is that $1*$ wins out in the end. However, if there is not a sufficient proportion of the higher ranking strings, the $1*$ schema will be eliminated.

4.11 Conclusion

This chapter has attempted to give an overview of the current art of GAs, with particular emphasis on the problems of coding. If clear conclusions seem lacking, it is because there is

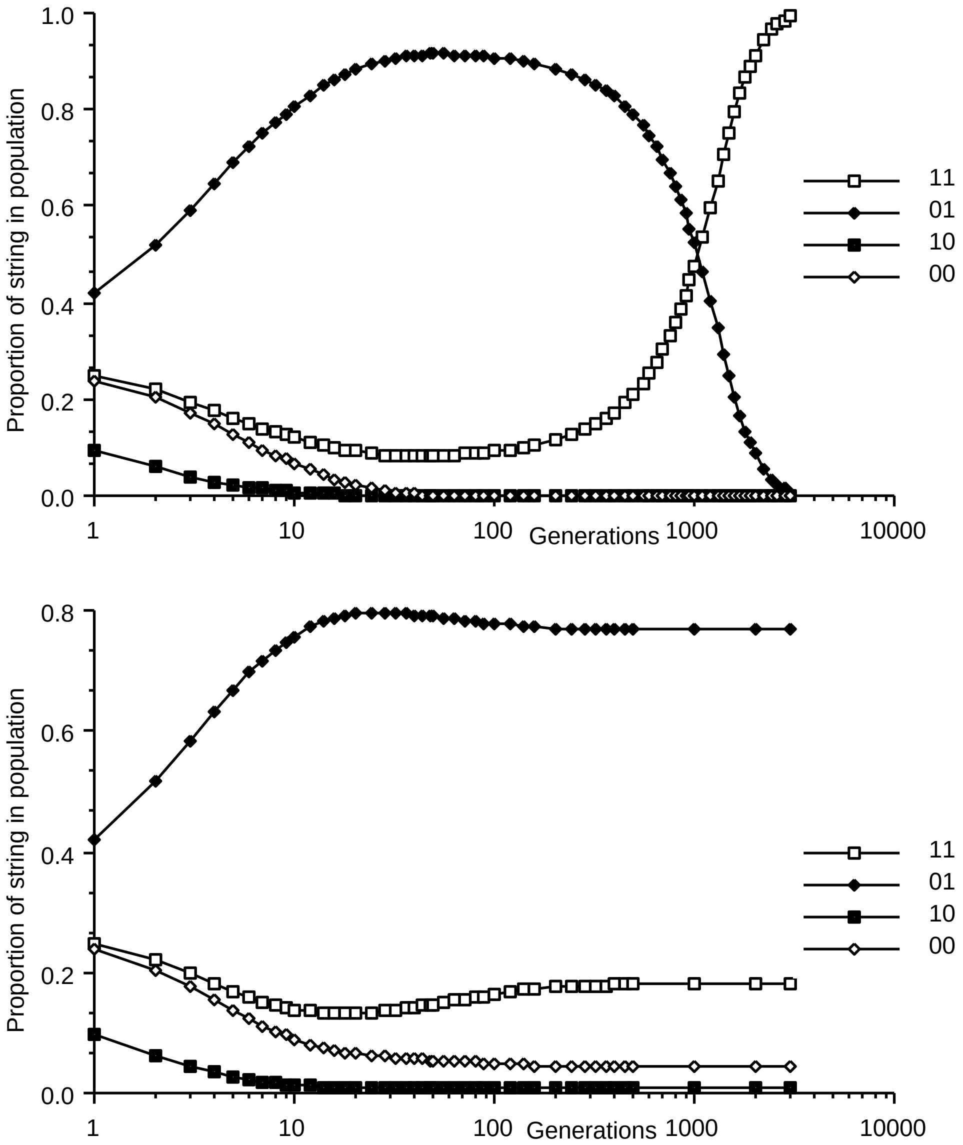


Figure 4.7: Numerical simulation of an MDP with string fitnesses of 11=4.11; 10=0.1; 01=4.10; 00=4.0, a) with no mutation, b) with mutation at a probability of 0.01

still uncertainty about the best means of approach. Application of GAs does remain rather empirical, despite attempts to develop theory (Rawlins, 1991).

For numerical tasks, it should be clear that the actual coding strategy used is not that important, since varying the reproduction operators can change the way the coding relates to the real task. Thus binary coding with binary mutation has severe Hamming cliff problems, which are significantly reduced by the introduction of a creep operator. The real task is to design the operators, especially recombination, that enable useful building blocks to be identified and processed. The design of such operators for the task of optimising neural nets is the subject of the next chapter.

Chapter 5

Optimisation of Neural Nets by Genetic Algorithm

5.1 Introduction

This chapter looks at the possibilities for optimising neural nets by genetic algorithm. Three potential areas are discussed: parameter tuning, direct learning of the weights and defining the structure of the net. They are by no means exclusive: it would be possible, for instance, to try and evolve the connectivity, weights and unit activation functions of a net simultaneously. There is relatively little reported work on parameter tuning alone (Bengio & Bengio, 1990; Chalmers, 1990), though some authors have used the GA to control learning rates for use by Backprop e.g. (Harp *et al.*, 1989a). This parameter tuning is a standard numerical optimisation, like those discussed in the previous chapter, with no need for special genetic operators. The author has used a GA to adjust the parameters of a complex simulation of some of von der Malsburg's dynamic link ideas (von der Malsburg, 1985). This simulation required modelling at the level of cell action potential and had numerous parameters, which the GA was able successfully to adjust (Hancock, 1989b).

There is more reported work on evolving weights and structure, both independently and together e.g. (Harp *et al.*, 1989a; Kitano, 1990b; Koza, 1990; Miller *et al.*, 1989; Mühlenbein & Kinderman, 1989; Nolfi *et al.*, 1990; Whitley *et al.*, 1990). Evolving the weights is a method of training the net to a task and thus serves the same role as Backprop. However, it is more flexible, since there need be no restrictions on recurrent connections or requirement for differentiability of the output function. A slightly different approach is to use the GA to define initial weights, which are then trained by an algorithm such as Backprop. There have been a number of such studies, looking at the interaction between learning and evolution e.g. (Hinton & Nowlan, 1987; Belew, 1989; Cecconi & Parisi, 1990; Keesing & Stork, 1991; Nolfi *et al.*, 1990). Evolving the connectivity is of interest because it is known that the internal structure of a net affects the generalisation performance, see section 3.2. In particular, when the input has a 2 or more dimensional structure, such as the images of section 3.4, there may be benefit in reflecting this structure in the hidden layers(s). Unfortunately, both weights and structure have an apparently severe representational problem, because of the many symmetries and possible permutations inherent in the parallel nature of nets. Although some authors have recognised this, few have attempted to address it (Radcliffe, 1990b; Radcliffe, 1993; Montana & Davis, 1989). The problem is mentioned briefly in two recent reviews of the application of GAs to NNs (Yao, 1993; Jones, 1993).

Section 5.3 outlines the nature of this permutation problem, while sections 5.4 and 5.5 survey the literature and discusses possible solutions to the problem for weight learning and structure respectively. Finally, section 5.6 reports an empirical comparison of recombination operators for net structure design, using a simulated evaluation procedure that allows statistically significant results to be obtained rapidly. First, a brief look at net parameter tuning.

5.2 Parameter tuning

As noted in the introduction, parameter tuning for net simulations is a straightforward application of GAs to a numerical optimisation, without the particular problems associated with evolving weights or connections. Most reported combinations of NNs and GAs have used Backprop as a training algorithm. Depending on the precise version of the algorithm, there are a number of parameters that might be tuned. Harp et al (1989a; 1991), primarily in-

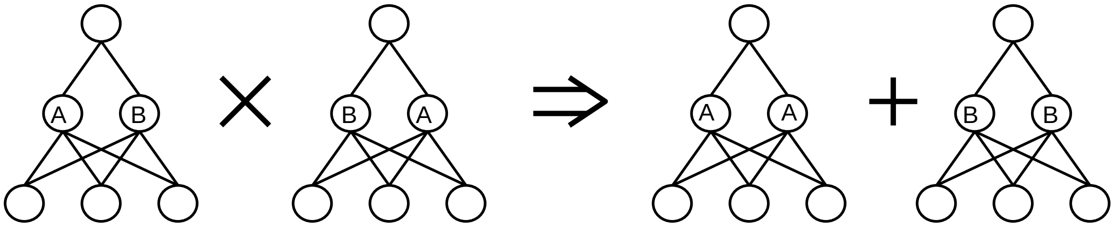


Figure 5.1: Recombination of two nets, which have the same solution but with the hidden units playing opposite roles, produces two nets each containing only part of the solution

terested in evolving the structure of their nets, also use the GA to specify the learning rate. They specify an initial rate, plus an exponential decay rate, for each of several areas within the net.

Modified Backprop algorithms may obviate the need to evolve the learning rate. There have been a number of attempts to vary learning rates automatically as training proceeds, eg (Jacobs, 1988; Tollenare, 1990), while Fahlman's (1989) Quickprop algorithm appears much less sensitive than "vanilla" Backprop to its learning rate parameters. Unfortunately the automatic parameter tuning methods seem to show an increased tendency to get stuck, which would add to the evaluation noise for the GA. In the experiments reported in section 6.2, the author used Quickprop.

It is also possible to evolve more detailed aspects of learning rules. The author has experimented with tuning the weight increment and decrement parameters of biologically motivated learning rules to maximise the capacity of auto-associative nets (unpublished). Bengio and Bengio (1990) discuss the possibility of searching for the parameters of a synaptic modification rule for a feed-forward net by GA, but do not present results. Chalmers (1990) evolves a learning rule by allowing the GA to specify the weight change as a function of four local variables: input and output activations, training signal and the current value of the weight. The rules were evaluated on their ability to train a single layer net on a suite of binary input classification tasks. The GA discovered the delta rule on about 20% of the runs.

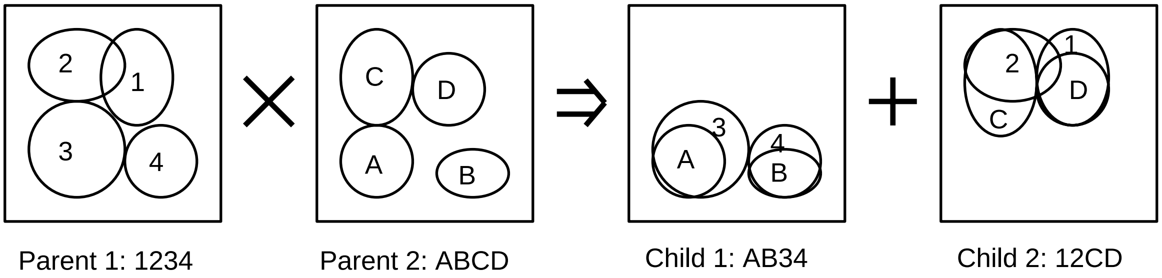
5.3 The permutation problem

In principle the application of a GA to learning the weights or defining the connectivity of a net seems quite straightforward: use one gene per link. This was the approach taken by Miller et al (1989) for structure definition, and by Montana and Davis (1989) for weight learning. That there might be a problem with this may be seen by considering a trained net with just two hidden units, figure 5.1. Unless the task is really trivial, the two units will play different roles, A and B, in achieving the correct solution. Which unit does which job will be immaterial: the order, in the sense defined by the list of units in the computer program, may be AB or BA. Consider now a GA, attempting to optimise the weights of the net. If it attempts to cross a net that has the order AB with one that has order BA, it may produce offspring AA and BB, both of which fail to solve the problem. The number of possible orderings is the factorial of the number of units. Thus for a modestly sized net with 10 hidden units, there are over three million permutations.

The situation is actually worse than this, because there is often more than one way of solving a problem. One net might have roles A and B, another, roles C and D. Crossover will now combine incompatible roles. It is not in general possible to put a number on the scale of this problem: Xor with two hidden units has two major solution types, but it must be expected that the number will rise with additional degrees of freedom in the net.

The preceding discussion rather assumes that crossover treats a unit with all its connections as a single entity, although the problems will still occur if crossover is not so constrained, since mixing two units that are attempting to play different roles is also likely to be disruptive. Confining crossover to node boundaries does tackle a third problem. This is caused by the fact that, with anti-symmetrical output functions such as the sigmoid, changing the sign of all incoming and outgoing weights of a unit will have no effect on its behaviour. As well as the solution AB, we now have -AB, A-B and -A-B. The number of combinations has risen by a further 2^n. If crossover does not disrupt a unit, then A and -A may be interchanged with impunity. However, doing this means that changes in the weights of a unit can only be made by mutation, with crossover limited to producing new combinations of the units. There is also a problem for nets with more than one hidden layer: to which unit does a weight belong? With two hidden layers, only units in one layer may be respected.

A similar problem may occur where the GA is defining the structure of a net, which will then be trained by another algorithm. The aim is to produce a net which is tailored to the task. Thus, roles A and B in the example above might require different sets of inputs. In that case, the situation is the same as before, with crossover liable to produce a net with two units of the same connectivity. An example for a 2-D input is given in figure 5.2. In general, the problem may not be quite so bad, since any units with identical connectivity are in fact interchangeable. The factorial is now an upper bound, with 1 as the lower bound in the rather unlikely situation that all the units have identical connectivity.



Parent 1: 1234

Parent 2: ABCD

Child 1: AB34

Child 2: 12CD

Figure 5.2: Recombination of nets with localised receptive fields. The parent nets each have 4 similar hidden units, but defined in different orders on the genetic string, so unit 2 on parent 1 is most like unit C on parent 2. Here, there is a single crossover point, between the second and third unit definitions. The resulting nets each have two copies of similar hidden units, and do not cover the input space.

We can now see the basis of the comment in section 4.6.4 that it is difficult to design a recombination operator that respects similarities of neural nets. Using the role descriptions A and B, formae for the net might be multisets such as $\{A, *\}$ and $\{B, *\}$: being sets the order is unimportant. Simple crossover may cross two instances of $\{A, *\}$ to produce $\{A, A\}$ and $\{* *\}$.

The problem stems from the fact that functionally equivalent nets may be defined by many different genetic strings. Radcliffe calls this a redundant coding, and one of the principles of coding for GAs in (Radcliffe, 1991), mentioned in section 4.6.4, is that codings should be minimally redundant. In the following sections we shall consider ways to reduce this redundancy.

First, a brief note about terminology is required. There are two possible meanings of the word redundancy when applied to codings. One, used here by Radcliffe, is that there are many possible ways of coding the same thing: "Newspeak" in George Orwell's book "1984" was an extreme attempt to eliminate this kind of redundancy from the English language. The other sense of the word is akin to tautology, coding the same information more than once. This kind of redundancy is often deliberately designed into codes to allow some error correction, simple examples being check digits and parity bits. It has been suggested as a means of addressing permutation problems in GAs (Gerrits & Hogeweg, 1991), unfortunately it is not obvious how to apply this technique to net design.

5.4 Evolving weights

Producing a set of weights for a net is a high dimensional optimisation task. The now standard method is the gradient descent technique of Backprop. However, gradient descent can get trapped in local minima. Genetic algorithms have the potential to escape such minima, one reason perhaps why there are many reports of using GAs to train weights, e.g. (Ackley & Littman, 1989; Belew *et al.*, 1990; Bos & Weber, 1991; Caudell & Dolan, 1989; Cecconi & Parisi, 1990; Fogel *et al.*, 1990; de Garis, 1990; Heisterman, 1990; Jefferson *et al.*, 1990; Keesing & Stork, 1991; Kitano, 1990b; Menczer & Parisi, 1990; Montana & Davis, 1989; Nolfi *et al.*, 1990; Parisi *et al.*, 1990; Radcliffe, 1990b; Torreele, 1991; Whitley, 1989; Whitley & Hanson, 1989). In addition, Backprop requires that output functions be differentiable, while recurrent connections demand a considerably more complex algorithm. Neither of these problems affect a GA.

For a simple, feed-forward net with the typical sigmoidal output function, it seems unlikely that the blind search of a GA would compete with the gradient descent of Backprop for speed of solution, unless the error surface is so full of local minima that Backprop requires many restarts. Another possible advantage comes from the relative ease of implementation of a GA on parallel computers. Nonetheless, Montana and Davis (1989) report that a GA will outperform Backprop. Since they do not give their training data the claim is unverifiable, and it must be regarded with suspicion as there is no suggestion that they attempted to optimise the Backprop parameters, while they certainly worked on the GA. Kitano (1990b) compares Backprop with a GA on the two-spiral and encoder problems, and concludes that, although competitive on very small problems, GAs are much slower on bigger ones. Whitley et al (1990) report GA results that are about as good as Backprop, but suggest that a GA will only really be competitive with more specific methods when no gradient information is available. They give an example of pole-balancing, when the only feedback to the net is when failure has occurred. Given this limited information, their GA out-performs Anderson's Adaptive Heuristic Critic algorithm (Anderson, 1989).

Whitley et al obtain these results by a method they refer to as genetic hill climbing, which is in effect an ES without any sophisticated mutation size control. Mutation rate is high, crossover rate low, and the weights are held as real values. Having a population of strings is equivalent to doing multiple starts in parallel: small populations sometimes do very well, but also fail more often than larger populations. The elimination of crossover from the algorithm is the most common approach to the permutation problem, e.g. (Cecconi & Parisi, 1990; Fogel *et al.*, 1990; de Garis, 1990; Nolfi *et al.*, 1990; Parisi *et al.*, 1990). Given the centrality of recombination in the GA model, this seems rather drastic. There appears to be only one report of simple crossover giving an improvement over a mutation only algorithm: Menczer and Parisi (1990; 1991) show that adding 25% crossover gives a small improvement in performance.

Two authors have tackled the permutation problem head-on: Radcliffe (1990b) and Montana and Davis (1989). The key is to try and identify equivalent hidden units. Montana and Davis's approach is to apply a subset of the training inputs to the net, and monitor the outputs of the hidden units. Prior to crossover between two strings, the units which are most correlated are matched. They report that this method does improve performance particularly early in the run, when the population is most diverse. There is obviously a considerable computational overhead, which perhaps explains why no one else appears to have taken up the idea.

Radcliffe's method involves a labelling algorithm, which also applies to network structure specification. Unfortunately, despite again incurring considerable computational cost, the algorithm will only succeed in making the match if the nets are effectively identical (apart from the permutation). As Radcliffe himself puts it, this is computationally useless (Radcliffe, 1990b). What is needed is either some similarity metric, or some means of imposing an order that will break the symmetries.

A possible approach to breaking the symmetries is to add connections between hidden units. Each hidden unit would have an inhibitory connection to all those after it in the layer. The first unit thus inhibits all the others, but is inhibited by none, the last is inhibited by all the others and inhibits none. The net thus remains feed-forward, but no longer has a simple single hidden layer, figure 5.3. The units are clearly no longer equivalent, but whether the symmetries have been broken will depend on the strength of the inhibition. If the inhibition is very weak, there will be no noticeable effect, but if it is very strong it may be difficult to find a solution to the target problem at all.

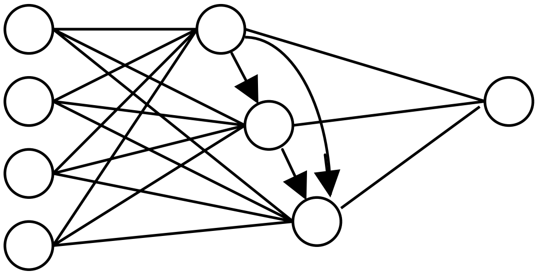


Figure 5.3: A feed-forward net, with inhibition in the hidden layer (arrows).

An alternative approach to the problem is to side-step it, by avoiding a fully-connected net. If the hidden units are no longer interchangeable, there is no permutation problem. In general this claim will be too strong: simply removing, say, one different connection from each hidden unit will not guarantee that they each have consistent roles in the final trained net. It is the interchangeability of hidden unit roles that produces the permutation problem: if the roles are consistent the permutations will be removed. The aim of the work discussed in section 5.5 is to produce a reduced connectivity net that is tailored to the problem in hand. The ideal might be to produce a net that admits only one solution to the problem. As is noted in section 5.5.1, it does appear to be possible to produce sparse nets that train very consistently. It may be that a way to reduce the effect of the permutation problem for evolving weights would be to evolve the structure simultaneously.

Another possibility is suggested by the analysis of the symmetries of network weights by Hecht-Nielsen (1990; 1992) and Chen and Hecht-Nielsen (1991). They note the various symmetries discussed above, and show that gradient descent techniques will always stay within the boundaries of one symmetry sub-space. The permutation problem for GAs is precisely that they do not: recombination does not respect the symmetry sub-space boundaries. However, they show that it is possible to define which symmetry group the system is in solely by the bias weights for each hidden unit. This raises the possibility of specifying that all members of the GA population will be in the same symmetry sub-space. Their method, discussed also by Roberts (1992), is to order the size of the n hidden unit bias weights b_i such that

$$0 \leq b_i < b_{i+1} \quad 1 \leq i \leq n-1$$

On its own, this prescription will not be sufficient, since there is no guarantee that, following crossover, the inequality will still hold. This may be addressed by setting tighter limits for each bias weight in the initial population:

$$(i-1)r \leq b_i < ir \quad 1 \leq i \leq n$$

where r is a constant in the region of 0.1. This will ensure that the bias weights of equivalent units are in the same range, so that the ordering still holds after crossover. A mutation to the bias weight may still cause problems, by disrupting the ordering. This was addressed in some preliminary experiments by preventing mutations to the bias weights altogether. The method was tested on an 8:5:8 encoder problem, five hidden units being used to increase the number of permutations. There was a significant but small decrease in the total error after 100 generations when the bias weights were ordered. Unfortunately, it was discovered that the change remained when crossover was disabled: it appears that the specified bias weights make the problem easier for mutation-based hill climbing alone. Preliminary experiments using inhibition between hidden units also failed to show any significant advantages.

5.4.1 Evolution and learning

Before leaving the subject of evolving net weights, it is worth elaborating a little on the possibility of evolving the start weights, to be trained subsequently by an algorithm such as Backprop. It is found that the action of learning can guide evolution. This is known as the Baldwin effect, having been suggested by Baldwin last century (Baldwin, 1896). It is not Lamarckian: the learned weights do not influence the genetically determined start positions (though such a local "hill climbing" operator is used by Montana and Davis (1989)). Hinton and Nowlan demonstrated the effect in simulation (Hinton & Nowlan, 1987). Their task was to find a sharp minimum located on an otherwise flat plateau. Exploration of the weight space by the phenotype, in this case just by random adjustment of the weights, is able to improve the fitness of genotypes that are close to a solution. The pinhole minimum is effectively smoothed into a much broader valley.

Nolfi et al (1990) showed that the effect is reciprocal: learning guides evolution, but evolution can also produce systems that learn better. This happens even when the evolved task and the learned task are different. In Nolfi's work, the evolved task is to produce a simulated animal that can find food, the learned task is for the creature to predict its next sensory input. The output units for the two tasks are separate, but they share hidden units, and it appears that the representations learned for sensory prediction are also useful for the food-finding task. However, Parisi et al (1991) have found that even learning a task such as Xor can assist evolution of the food-finding ability. In this case it seems implausible that the representations required for Xor are useful for food-finding, and the mechanism is likely to be effectively random weight adjustment like that used by Hinton and Nowlan.

Such combinations of learning and evolution can produce remarkably sophisticated behaviour. The cpu cost is rather high: Collins and Jefferson report using 8 days on an 8K processor Connection Machine (Collins & Jefferson, 1991). Nets tend to be small, but it appears fairly general for workers in this field to use mutation only algorithms in a bid to avoid permutation effects. A notable exception is the work of Menczer and Parisi mentioned above, where it was found that a probability of simple crossover of 0.25 is helpful (Menczer & Parisi, 1990; Menczer, 1991).

5.5 Net structure optimisation

As discussed in section 3.2, the internal structure of a net can have a considerable effect on its performance. That section considered briefly some of the possible approaches to matching the structure to the problem: algorithms that adjust the number of hidden units and weights in response to various measurements of the net's behaviour. An alternative approach is to do a blind search on the space of network structures. The discontinuous nature of the search space and the noisiness of the evaluation makes the use of GAs appealing.

There have been a number of attempts to use GAs in this way. A common approach is to define a net which is then initialised with random weights and trained, usually with Backprop (Dodd *et al.*, 1991; Falcon, 1991; Hancock & Smith, 1991; Harp *et al.*, 1989a; Harp & Samad, 1991; Miller *et al.*, 1989; Robbins *et al.*, 1993; Schiffmann & Mecklenburg, 1990; Schiffmann *et al.*, 1991; Vico & Sandoval, 1991). One exception is a pruning technique suggested by Whitley et al (1990). A number of authors have used GAs to specify the input to a net (Anthony *et al.*, 1990; Badii, 1990; Chang & Lippmann, 1991; Hőffgen *et al.*, 1991). This kind of feature selection does not pose any permutation problem, since the inputs impose an ordering. They will not be considered further here, the other two will now be looked at in more detail.

5.5.1 Genetic pruning

Whitley et al (1990), use a GA to prune down a trained net, thus achieving an effect similar to other pruning algorithms (section 3.2). Whitley et al took this approach because of the problem of excessive cpu requirement when training a net from random weights with Backprop. A net, fully connected in the examples given, is trained on the problem. Pruning is specified by the GA, using one bit per connection. The pruned net is evaluated by instantiating the remaining connections with their trained weights and then retraining with Backprop. By seeding the weights in this way it may be hoped that the net is relatively near to a solution, so that retraining will require fewer epochs than a randomly initialised net. This can only be a hope, since omission of vital connections may make it impossible to reach a solution at all. However, the GA may be expected to discover which are the important links.

Whitley et al acknowledge that there is no guarantee that a net that learns well from this initial position will do so when started from random start weights. They report only Exclusive-Or, and a 2-bit adder problem, with four hidden units. However, in the latter case the design does learn well, and very consistently, from random start positions. The GA appears to have been successful in finding a design that guides Backprop to a specific solution and avoid local minima. This is potentially quite significant, since Backprop can find it difficult to break symmetries in the weights (Hancock, 1990b). Most experimenters will have come across times when it appears stuck on a plateau, and makes no progress at all for a while. Although this problem is probably most apparent in toy problems like Xor, it is still possible to speed up training on real-world problems considerably, as will be shown in the following chapter. This raises the possibility of producing a net tailored to a task, such as face recognition, that will not only generalise well with a new set of data, but also train relatively quickly.

It is not clear from (Whitley *et al.*, 1990) whether the authors realise that they have also found a possible solution to the permutation problem. Because the net starts from a trained position, the units are all effectively labelled. The units have their roles in the solution predefined: the GA simply has to discover which connections are superfluous to that role. This approach is clearly worthy of consideration: some experiments on rather larger problems than Whitley et al used are reported in section 6.5.

5.5.2 Genetic net definition

Coding

In the previous chapter it was argued that the coding of a problem, and its interaction with reproduction operators, is critical to the ability of the GA to make progress. There are many possible string codings for net structure. Miller et al (1989) suggest a possible metric for classifying them. Strong coding schemes define each connection individually, while weak schemes specify more general growth rules. It is clear that biology uses a weak coding method at least for higher animals, though even for animals as complex as insects it appears that their neural circuitry is largely genetically specified (Bentley, 1976). In mammals there simply is not room in DNA to specify every cell in their complex brains individually, still less every connection. The human genome contains about 3×10^9 base pairs, each carrying 2 bits of information, compared with at least 10^{10} cells in the human brain. Miller et al use a strong specification scheme. This has the potential to produce very specific, compact designs to suit a particular problem. It also frees the GA from the preconceptions of the experimenter, which might limit exploration of fruitful areas. This freedom is a two-edged sword, however: the GA may also spend its time blindly exploring useless areas, which the experimenter's knowledge of the problem could have prevented. How much to build in will depend on the aim of the work. Miller et al were principally interested in seeing what designs an unfettered GA might come up with. If the aim is to solve a particular problem then standard engineering practice is to build in available knowledge.

Miller et al's coding scheme, a reduced version of that suggested by Todd (1988), is a simple connectivity matrix, where a 1 represents a connection between two units. They claim that this scheme requires less checks for legality than weaker coding methods. However, it is still quite possible to have units with no input and/or no output, and recurrent connections. The latter cannot be trained by standard Backprop, and have to be removed by the net generation routine. Unconnected units do no particular harm, other than wasting cpu time, and are the system's method of controlling the number of units, which has a fixed maximum. Nets which have no path from input to output are weeded out at birth. There is no attempt to address the permutation problem, other than implicitly by attempting only relatively small nets where the problem is not severe. On problems such as XOR and a real-valued version of XOR called the four quadrant problem, their method shows promising results. Anthony (1990) reports some success using a similar method on a slightly larger task, of predicting the central pixels from their surroundings in a medical image.

Harp et al use a "blueprint" specification, towards the middle of Miller et al's scale. They define a number of areas of units, along with their projective fields. Each area can have a variable number of units, stepped in powers of two. They may also have two dimensional structure, to accommodate image-type data. They can make connections to other areas, using either absolute or relative addressing. The latter possibility is intended to allow the development of cooperating areas that reside physically close on the genetic string. The connections may also have a 2 dimensional extent, allowing localised receptive fields, with an adjustable connection probability within the specified field. The different areas of cells allow the construction either of multiple-layer nets, or of quite complex single hidden layers. The string length, and hence number of areas is not fixed, being altered by the action of crossover. This is constrained to act at one of a number of homologous points defined within each area specification, which ensures that any string generated is legal. The net specified by the string, however, may not be, as there is the same potential for isolated units and recurrent connections. These are, as in Miller's scheme, dealt with at net generation time. Even so there may be multiple connections between units: one example of a single layer net for digit recognition (Harp *et al.*, 1989b) has 32 input units, 16 output units and 2566 connections between them! There is again no explicit attempt to address the permutation problem, which exists because the same net may be generated by many different strings, simply by swapping area labels.

Despite the potential for generating complex net architectures, Harp et al have reported only relatively small test problems: XOR again, a digit recognition task and modelling the sine function. One reason for this is lack of computer power: although a small problem, XOR is quite difficult for Backprop to learn, and one generation of the GA is reported to take a day or two to run on a Symbolics Lisp Machine.

Schiffmann et al (1990; 1991) use a mutation-only evolutionary strategy to produce net structures. This side-steps the permutation problem, which may be the reason for its adoption, though this is not explicitly stated. The precise coding scheme is also not defined, but they start with a fully connected single layer net and gradually add units by random mutations. The mutations may also add and remove connections between existing units, but not, apparently, remove units. The algorithm produces spectacularly "deep" networks, with as many as 25 hidden layers. Although not stated, this cannot be a simple hierarchical design, since Backprop has difficulty with more than 2 or 3 consecutive hidden layers. The nets are selected for training performance, and do train more rapidly than simple fully connected designs, but are no better at generalisation.

Kitano (1990a) uses a net generation method that is, at first sight, towards the weaker end of Miller et al's scale. Rather than coding units and connections directly, the GA specifies a set of rules for a grammar. Capital letters are expanded to a 2 × 2 matrix of other letters, lower case letters expand into a 2 × 2 matrix of ones and zeros which specify the presence of connections. The coding is therefore strong, since each connection can be individually specified. However, changing one connection may require several alterations to the genetic string, because a change to one rule may have effects elsewhere in the connection matrix. Kitano reports that his methods result in nets which train much more rapidly than fully connected nets. The claim needs to be regarded with some doubt, however, since it is not only based solely on the rather unrepresentative encoder problem, but the fully connected versions of the net have as many as 30 hidden units for 4 input and output units. One potential strength of the specification method is that it leads rather naturally to regular patterns of connectivity, which might suit larger problems such as image processing. However, there is still a problem of redundancy: the same net may be coded by more than one genetic string.

Recently, Gruau (1992) has proposed another grammar coding method that he calls Cellular Encoding. He suggests a number of properties desirable in a coding, such as completeness, the ability to code any connectivity; compactness, to keep the search space small; and closure, that any net generated should be valid. He does not mention redundancy, indeed, the paper does not specify how the grammar is to be coded on the GA string at all. This makes the system hard to assess: it seems promising, but would appear still to suffer from redundancy.

Nolfi and Parisi (1992) have experimented with defining growth rules for nets. Each unit grows an output axon in the shape of a fractal tree. The genetic string specifies the x-y location of each unit, the length of each branch and the branching angle. Units with the maximum y value are output units, those with the minimum value, inputs. The x coordinate of a unit has no significance other than its spatial relationship to the other units. Each fractal tree branches four times. Connections are deemed to be made to any units that happen to be crossed by one of the branches of the axon. Any isolated units or dead ends are then removed. The weight of all the outgoing connections from a unit is set to the same value, also held on the string. The resultant net is used to control a food-finding animat. They report successful results, with the final nets being compact.

Nolfi and Parisi's growth rules effectively define the projective field of each unit. This is successful partly because their animats have only two types of input unit. For Backprop nets, especially those used for classification, defining the receptive field is more natural, since there is usually a relatively large number of inputs. The coding could be amended to do this. Defining hidden unit projective fields might be useful if there are many output units. As it stands, their coding scheme would also produce nets that have too many layers of hidden units to train well with Backprop. This too could be amended. However, it is not obvious that a fractal tree is a better way of specifying a receptive field than, say, a circle defined on an array of input units. Their coding is possibly best suited for the purpose for which it was designed: specifying the nervous systems of animats. It still suffers from the permutation problem, there being many ways to code identical nets.

5.5.3 Addressing the permutation problem

In the work described so far, there have been two principal approaches to the permutation problem: ignoring it, or avoiding it by removing crossover from the algorithm. Radcliffe has recently suggested a method of addressing the problem (Radcliffe, 1993). This proposal is quite different from his matching algorithm referred to in section 5.4. He specifies a recombination operator that attempts to respect similarities in net structures. Unfortunately, as we shall see, it does not actually respect similarities, only identities.

Radcliffe identifies a hidden layer structure as a multi-set, the elements of which are the possible patterns of connectivity of one hidden unit with the input and output units[1]. Given n input and m output units there are thus 2^{n+m} possible elements, which may be labelled

[1] For simplicity this analysis is restricted to one hidden layer

a, b, c.... The layer is a multiset because it is quite possible for two or more units to have the same connectivity. A given hidden layer might be described by $\{a, a, d, f\}$, which would be an instance of, in this case, twelve formae of the type $\{a, *\}$, {a,a,$*$}, $\{a, d, f, *\}$, where $*$ here means one or more other elements. According to Radcliffe's principles for GA design referred to in section 4.6.4 the recombination operator should respect and properly assort these formae. Recall that respect means that a child will always inherit characteristics that are common to both parents, while formae are properly assorted if it is possible for the operator to produce a child which is an instance of both formae.

Unfortunately, it is not in general possible simultaneously to respect and properly assort these formae if the sets are of fixed size (Radcliffe, 1993). The sets $\{a, d, f\}$ and $\{a, b, e\}$ are examples of the formae $\{d, f, *\}$ and $\{b, *\}$ (amongst others) respectively. These formae have an intersection, namely $\{b, d, f\}$, which should be obtainable if they are properly assorted, but respect demands that the child be a member of $\{a, *\}$ since both the parents are. There is no solution to this conundrum: in Radcliffe's terms the formae are non-*separable.* Radcliffe therefore suggests a compromise, such that elements common to both parents are more likely to be copied to a child than those present in just one. The process may be tuned: as respect becomes less likely, proper assortment becomes more possible.

If the constraint of fixed set size is lifted, the formae become separable. The solution to the above problem is $\{a, b, d, f\}$. Although there is in general no reason to constrain hidden layers to a fixed size, there will usually be some kind of limit, if only imposed by something mundane such as array sizes. The probabilistic copying of strings common to both parents will therefore be adopted since it is a general solution to the problem.

This method of recombination will only remix the elements already present. As noted, the number of possible elements is large, such that given a typical population size of 100, many elements may not be represented. To allow recombination to explore this space, Radcliffe suggests that one or two of the elements in a child be produced by a traditional, uniform crossover of the bit strings that define two elements in the parents.

Radcliffe's algorithm may be summarised:

1. Decide on the number of units in the child, somewhere in the range bounded by the numbers in each of the parents, except for the possibility of mutation moving one beyond this range[2].

2. Match the hidden units of the two parents to identify any that are identical. There may be more than one copy in each parent. Copy these to the child with a high probability.

3. Fill all but one or two of the remaining slots in the child at random, without replacement, from the parents' remaining unit definitions.

4. Fill the remaining child units by crossover of two units from the parents, picked at random.

The problem with all this is that there is still no notion of similarity, only of identity. If two potential parents have some hidden nodes that are identical, then they will be matched: if they differ by one connection they will be treated as if they differed in every connection. Radcliffe (personal communication) argues that the tendency of GAs to start convergence fairly rapidly

[2] Radcliffe actually suggests that the upper bound should be given by the size of the union of the two parent sets, but agrees that this seems likely to lead to excessively large hidden layers (Personal communication)

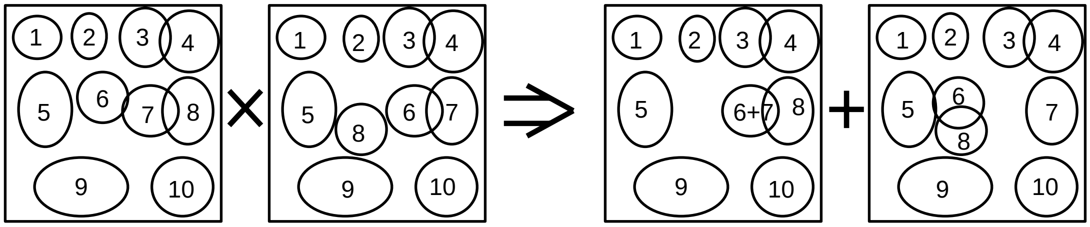


Figure 5.4: X-Y sorting of hidden unit receptive field labels prior to recombination. Although some are transmitted consistently (1-4), the repositioning of the 6th RF of the first parent downwards in the second parent causes a relabelling of that row, leading to duplication of coverage in the offspring.

suggests that there will be a useful number of identical nodes to process. Against this it might be argued that the preferential selection of any nodes that just happen to be the same early in the run may give them an unwarranted advantage and encourage premature convergence.

Similarity metrics

What might be used as a metric of similarity in hidden nodes? In a continuation of the work reported by Hancock and Smith (1991), which is described in section 6.2, the author has proposed two methods. The first is applicable to the 2-D input used in that study, the second is more general.

1. X-Y sort. The input data used by Hancock and Smith (1991) were processed images of faces, that retained a 2-D structure. The hidden units had a localised receptive field (RF), defined by a centre and (approximately) a radius on the input units. A simple means of imposing an order on these hidden units is to sort them into x within y order. Crossover then acts on the sorted strings. Although this might be more successful than an unsorted crossover, it will not be very reliable, since a small change in position of an RF can cause a significant reordering of other units, see figure 5.4.

2. Overlap sort. A much more general method of assessing hidden unit similarity is to count the number of input units they have in common. In the work continuing from Hancock and Smith (1991), the hidden layer was fully connected to the output units for simplicity, but the notion of similarity could clearly be extended to hidden-output unit connections. The hidden unit definitions are sorted by overlap prior to crossover.

How might this notion of overlap similarity be added to Radcliffe's proposal? For simplicity, the following discussion will assume that only input connections are being changed, though for a single hidden layer, output connections can be treated identically. It is clear that the node labels $a, b, c...$ are not helpful, since in common with other high-cardinality alphabets, they give no clue about similarity. The connections of a hidden node may be described by a binary valued vector: we are not interested here in the connection strengths, which will be learned once the net has been created. It therefore appears that this is a case where the traditional binary schemata are ideal for the problem. A metric would then be given by counting common bits:

$$distance = \frac{Hamming\ distance}{number\ of\ bits}$$

$$overlap = 1 - distance$$

Further thought indicates a problem, concerning just how useful 0 is as an indicator of similarity. Consider two hidden unit definitions for an 8-input net, $\{11000000\}$ and $\{00000011\}$. These are both members of the schema $\{**0000**\}$. Would we really wish to say that these units are more alike than the pair $\{11100000\}$ and $\{00000111\}$, which are members of $\{***00***\}$? In terms of the overall functioning of the net, a schema such as $\{***00***\}$ might be meaningful. If it became very frequent, it might suggest that the two middle input units carry little information relevant to solving the given task. For the purpose of identifying units that perform similar tasks, however, such schemata seem uninformative: we are really only interested in where there are connections. This would suggest rather strange-looking schemata composed of just 1 and $*$. There is then no schema that contains either of the pairs above: they have nothing in common. A metric is given by counting just the 1s: for units defined by binary strings k and l:

$$overlap = \frac{inputs\ in\ common}{total\ inputs\ connected} = \frac{|k.and.l|}{|k.or.l|} \tag{5.1}$$

The preceding discussion makes no assumptions about the inputs. If the input has a spatial structure of the kind advocated in chapters 2 and 3 it might be advantageous to extend the notion of similarity. For instance, it might be that $\{11000000\}$ and $\{00110000\}$ actually are more alike than $\{11000000\}$ and $\{00000011\}$. Schemata drawn from $\{1, *\}$ would clearly not capture such similarities. A suitable extension that applies to the simple binary connection lists is not obvious, and would in any case be highly problem-dependent. The approach taken in the face recognition work ((Hancock & Smith, 1991) and section 6.2) is to limit the connections of a hidden unit to a local receptive field. The x-y sorting method described above then does capture something of this notion of similarity for a 2-D input space, albeit rather crudely. It only imposes an ordering along the x axis, and then not consistently. Referring to figure 5.4, it may be seen that while the labelling of units 1-4 correctly reflects their separation, it does not get 4-5 and 1-5 correct.

Building knowledge of the input coding into the similarity metric would be expected to improve performance, at the expense of generality. A general approach to reducing the permutation problem remains of interest. Schemata composed from $\{1, *\}$ capture such a general notion of similarity for connectivity. Having identified the similarities, we need an appropriate recombination operator.

The essence of Radcliffe's approach is to identify identical units, transmit those with high probability, and then fill remaining slots at random. Now suppose there are units on the two parents that differ in only one connection. By analogy, at least the common bits should be transmitted with high probability. This could be achieved either by choosing one or other, or by crossover between them. Consider the following example: two strings, each defining two hidden units with connections from six input units, {111000,000101} and {111000,000011}. The strings agree on one of the units, once matched, any form of crossover would not affect the definition. However, the other units are different, agreeing on only one out of three connections. There is the usual choice between exploration and exploitation. Where the overlap is this small, the conservative choice would be to use Radcliffe's suggestion of choosing one or other of the parent definitions intact. However, it may really be the case that the schema $\{*****1\}$ is highly rated. Uniform crossover will both respect and properly assort schemata drawn from $\{1,*\}$, just as it will those drawn from $\{1,0,*\}$ (Radcliffe, 1991). The problem with its use on unmatched unit definitions is that it will tend to disrupt them, that is, the exploration component will be too large. Here, since the unit definitions have been sorted prior to crossover, it should be possible to increase the probability of such crossover, as any points of agreement will in any case be preserved. A straightforward approach is to regard the unit definitions as having been matched, and therefore apply uniform crossover between them, with a probability equal to their overlap on the similarity metric.

The sorting algorithm needs to be specified rather more precisely. Should unit definitions in the two parents be paired off, or matched with the one they most resemble? And what should be done when the parents have different numbers of units? Some possibilities may be considered. In each case, it is assumed that there are two strings with i and j units respectively, where $j \geq i$.

1. Match each unit in each string with the one it most resembles in the other string. This will give $j+i$ pairs, with up to $j+1$ child slots to fill (allowing for possible mutation in the number). A problem comes in trying to see how to fill these slots. Any matching units should be transmitted with high probability. So pairs might be chosen with a probability that is related to their similarity, with the child unit being produced by uniform crossover of the parents. However, where units match, there will now be two pairs. If both get the same high probability of being chosen, it is likely that there will be two copies in the child, which, being a form of mutation, should happen only rarely. To prevent this, the second pair might be eliminated, either prior to selection or if the first pair is selected. In the latter case the second string would provide another opportunity for the unit to be transmitted if the first pair failed. This could be addressed by lowering the probability of selection to compensate. However, there remains a problem of what to do with partially matched pairs. Returning briefly to the use of labels for unit definitions, suppose we have the strings $\{ab\}$ and $\{cd\}$, where a overlaps most with c, but c overlaps most with b. Assume the ac pair gets picked for copying to a child slot. If the cb pair is left in the pool for selection and picked, c will be over-represented in the child. If it is removed, b will not be represented. Some form of bias is inevitable.

2. For each member of the longer string (or one at random if the same length), find the unit definition in the other string with which it most overlaps. This will give j pairs, plus any unpaired units from the shorter string, from which to fill $j+1$ slots. Pairs might again be selected with a probability related to their overlap, with the child being produced by uniform crossover. This may leave some slots vacant. Since the purpose of not always copying matched pairs is to allow the possibility of producing a child containing only unmatched units (to allow proper assortment), it seems sensible that priority in filling vacant slots should go a) to any unpaired units from the shorter string, b) to the paired units, with probability inversely related to their overlap.

3. Pair off the units, by comparing all with each other, and picking pairs in order of overlap. Where a unit overlaps equally with two or more others, attempt to break ties by picking the one that would do least well if given its second choice. This will give i pairs, plus $j-i$ unmatched units from the longer string. Filling the child slots would proceed as for the previous method.

The first of these seems to raise too many complications to be worth considering. Of the others, the third seems preferable. Consider the two strings {111000,110000} and {111000,000011} again. If the first is chosen as the primary one for matching by the second method, the pairs will be 111000,111000 and 110000,111000. With high probability, this will ignore the 000011 unit completely. The third method will give equal weight to the two non-identical unit definitions. It will also leave unpaired units from a longer string, but these will have a chance of being copied into the vacant slots of the child.

The suggested revised recombination algorithm is thus:

1. Decide on the number of units in the child as before.

2. Sort the unit definitions, by matching pairs using the overlap measure equation 5.1, until all the units of the shorter string have been paired.

3. Transmit identical pairs to the child with a high probability p_t, transmit less well matched pairs with probability of $p_t \times overlap$. Produce the child unit definition by uniform crossover of the parent strings.

4. Fill most remaining places in the child by selecting parent unit definitions at random, weighted towards those that are unpaired or with little overlap.

5. Fill the one or two remaining slots by uniform crossover of two elements from the parents, picked at random.

Despite the extra complexity, this algorithm does not need any parameters in addition to the matched pair transmission probability of Radcliffe's method. The selection process for step 4, filling most remaining places, may conveniently be done using Baker's SUS algorithm (Baker, 1987). The "fitness" of each unit definition is initially set at 1, and reduced by the overlap measure from step 3. Thus units which matched perfectly in step 3 do not get a chance to be selected at stage 4, while those which had an overlap of 0.5 will get only half the chance of unpaired units.

The process of matching and selection means that this recombination algorithm will carry greater computational overhead than a simple crossover. However, any such overhead will be negligible when compared with the cost of evaluating each net design, which might take several minutes.

5.5.4 Summary

This section has considered some approaches to the genetic definition of nets. Most previous work has failed to tackle the permutation problem that results from the many symmetries present within in most nets. The exception is Radcliffe, who treats a hidden layer as a multiset, and advocates a matching process to ensure transmission of units that are common to both parents with high probability. This matching process was extended by the introduction of a similarity metric for hidden units, based on the number of common connections.

5.6 Evaluating the operators

The acid test of the proposed recombination operators will be their ability to produce effective net designs. Some such experiments are reported in the next chapter. However, the full procedure of instantiating, training and testing a net is extremely cpu-intensive for any realistic dataset. Since both GAs and Backprop-trained nets are stochastic, several runs are required for each set of parameters with each operator if significant results are to be obtained. A faster method of evaluating the operators is highly desirable. Some of the results of this section are reported elsewhere (Hancock, 1992c; Hancock, 1992a).

The method used in this section is to specify a net design manually, and test the GA's ability to produce a matching net. This may be done by comparing the connections of each of the target hidden units with each of those specified by the GA. An overlap matrix is built, like that used for the overlap operator. Target and test units are paired off, in order of decreasing overlap. The fitness is given by the sum of the pairs' overlaps, minus an arbitrary penalty of one per unit defined in excess of the number in the target net. No specific penalty is required for having too few units, since such nets will be penalised by being unable to match all the target units. The matching process means that the required units may be defined in any order on the genetic string, so the permutation problem is present.

This match may be evaluated in a fraction of a second, allowing detailed comparisons between the algorithms to be made. The method also allows the operators to be compared on different types of net. It might be, for instance, that one operator fares best if the receptive fields of the hidden units do not overlap, but another is better when they do, or when there are multiple copies of the same hidden unit type.

There are a number of potentially important differences between this method and the evaluation of real nets.

1. The connections of a real net typically differ in importance, whereas the overlap measure treats all equally.

2. It is likely that a real net design will show a significant degree of epistasis, i.e. an interdependence between the various units and connections. The value of one connection may only become realised in the presence of one or more others.

3. Linked to this is the possibility of deception: one configuration for a particular unit may be good on its own, but a different configuration much more so only when combined with another particular unit type. For example, one large receptive field might be improved on by two smaller ones, but be better than either small one on its own.

4. The evaluation of real nets will be noisy.

All of these factors may be added in to the evaluation procedure, albeit not with the subtlety that is likely to be present in a real net.

5.6.1 The recombination operators

Initial trials were performed with a variety of net designs and three different recombination operators: Radcliffe's matching algorithm, the overlap extension to it, and a simple GA using a traditional crossover. A number of things became clear.

- The overlap operator performed better if all matching pairs (i.e. all pairs with an overlap greater than 0) were transmitted to the child with probability p_t, rather than $p_t \times \mathit{overlap}$ as defined in section 5.5.3.

- Radcliffe's prescription of "one or two" unit definitions to be produced by uniform crossover of parent units chosen at random is too disruptive. A better result was obtained by producing one unit this way with a probability p_r of around 0.1.
- The simple GA performed well, rather better on the initial tests than either of the more complex operators.

The last result was particularly unexpected, suggesting that the permutation problem may not in practice be very significant. More extensive trials were therefore conducted using five different operators:

1. **Random.** Like Radcliffe's operator, but without the matching procedure. A child unit was produced by uniform crossover of two parent units with probability p_r, the remainder were picked randomly, without replacement, from the union of the parent unit definitions. This does nothing to address the permutation problem, but is blind to it: the expected result will be the same whatever the order of units on the two parent. It was included to give a baseline performance.
2. **Match.** Radcliffe's operator as described above, with one child unit produced by uniform crossover with probability p_r.
3. **Overlap.** The overlap extension to Match, with amendments just described.
4. **Uniform.** A simple GA, with crossover at unit definition boundaries more likely (probability p_u) than at other bits (probability p_b). Like Random, this does nothing to address the permutation problem, but the results will depend strongly on the ordering of the two parents.
5. **Sort.** An extension to the simple GA, that matches the unit definitions in the two strings, using the same distance measure as Overlap, and sorts the strings into overlap order prior to performing crossover. The major difference between this algorithm and Overlap is that the latter produces child unit definitions by uniform crossover of the matched parent definitions, while Sort is likely to choose one or other, with only a small probability p_b of a cross occurring at any given bit within the definition.

In addition, the evaluation procedure may be amended to remove the permutation problem, so as to investigate what effect it has on Uniform. This is the only operator which does not rearrange the order of the unit definitions during recombination and should therefore be most affected by the permutations. The task for the GA is then simply to match a bit string in the order given, which, at least in the absence of deception, should be trivial. In the graphs that follow, this is labelled **NP** (no permutation).

One other change made to the algorithms described in section 5.5.3 was to alter the way in which the number of units is specified. As noted in section 4.6.4, Radcliffe's suggestion of using flat crossover, which gives a value somewhere in the range bounded by the two parents, causes a strong convergence to the centre of the allowed range. The number of units in the child was therefore specified by picking the number from one of the parents at random. This number might be mutated up or down by one (subject to staying within the range of one to the maximum allowed) with a mutation rate that gave approximately one such change per generation. The genetic string was fixed length, consisting of a number of unit definitions, preceded by a parameter that specifies how many of the units will actually be built. For all the experiments reported below, the maximum number of units allowed was twelve.

5.6.2 Evaluation

The initial evaluation algorithm used exactly the same measure of overlap to compare target and test unit definitions as the Overlap operator uses for its matching. It was felt that this might give undue advantage to Overlap and Sort, since they are specifically attempting to propagate the same similarities that would then be used to evaluate the result. Two other overlap measures were therefore tried. The first simply replaced the measure from equation 5.1 with Hamming distance. The same penalty of 1 per unit defined in excess of the target number was applied. The second matched each pair of target and test units purely on the basis of the fraction of the target connections present in the test unit. A perfect match for any target unit would therefore be obtained by switching on all the connections in the test unit. However, the test string was then penalised for every connection in excess of the total in the target string. If this penalty was too large, the GA tended to respond by reducing the number of units specified in the test string, since this is the quickest way to reduce the total number of connections. A similar effect might be produced with real nets if the cpu time component of the evaluation function was too large. The penalty used in the tests reported here was $0.2 \times (1 - \frac{test\ total}{target\ total})$. This replaces the penalty for excess units used in the other two procedures. The maximum score in this case is 1.0, while in the other two it is equal to the number of target units. The three evaluation operators may be summarised, in each case the summation is over the pairs of test and target unit definitions after they have been matched:

1. **E1** $F = \sum_{i=1}^{i=n_{target}} ov_i - (n_{target} - n_{test})$
2. **E2** $F = \sum_{i=1}^{i=n_{target}} H_i - (n_{target} - n_{test})$
3. **E3** $F = \frac{\sum_{i=1}^{i=n_{target}} |test_i \cap target_i|}{target\ total} - 0.2 \times (1 - \frac{test\ total}{target\ total})$

where F is the performance, and the over-size penalty only applies if the test size is bigger than the target size.

Noise was introduced by addition of a Gaussian random variable to the final score for a string. The results below used a standard deviation of 5% of the maximum evaluation score, which is similar in size to the evaluation noise of Backprop trained nets (see section 6.3.5).

Epistasis and a form of deception were introduced by giving the GA two target strings. The first was scored as usual. For the second, the target units were grouped, with the score for the group being given by the product of the scores of the constituent units. This is epistatic because the score for one unit depends upon the match of others. It can be made deceptive by increasing the value of the group so that it exceeds the combined score of the equivalent individual units in the first string. The building blocks given by the easy task of matching the units of the first string do not combine to give the better score obtainable by the harder job of matching the second string. Although not formally deceptive in the sense defined by Goldberg (1987), such an evaluation function may be expected to cause problems for the GA. In the experiments reported below, the units were gathered in four groups of three. Since each unit scores 1 if correct, the maximum score for the group, being the product of three unit scores, is also 1. The sum of the group scores was multiplied by 9, to give a maximum evaluation of 36, compared with 12 for matching the first string. The rather high bonus for finding the second string was set so as to ensure that it was in fact found (by all but one of the operators, as noted below). There are two points of interest in comparing the recombination operators: are there differences in the frequency of finding the second string, and, having found it, are there differences in the rate of convergence on it? At lower bonus values, all the operators sometimes failed to find the second string. Despite averaging over 1000 runs, no significant differences could be found between the operators in their ability to find the second string. The bonus was therefore increased to test convergence, since stochastic variations in the number of failures might cause differences in the average performance bigger than those produced by the differences in convergence.

Target nets

While a variety of target net designs have been used, results from just three are reported here. These are

1. **N1** A net with 10 hidden units and 30 inputs. Each hidden unit receives input from three adjacent inputs, such that RFs do not overlap. Target string length $l = 300$.

2. **N2** A net with 10 hidden units and 18 inputs, arranged in a 6x3 matrix. The hidden unit RFs tile all the 2x2 squares of the input matrix and are therefore heavily overlapping. Target string length $l = 180$.

3. **N3** A net with 12 hidden units and 10 inputs. The input connections of 10 hidden units were generated at random, with a connection probability of 0.3. Two of these units were then duplicated. This was the design used for the deceptive problem: a second target being produced in the same way, using a different random number seed. Target string length $l = 120$.

Note that some combinations of net and evaluation method ought to be quite straightforward: being soluble in a single pass by the simple algorithm of flipping each bit in turn and keeping the change if it gives an improvement. This requires just l evaluations. Such an algorithm will probably fail in the presence of noise or deception, but in the absence of these can solve problems using E2 or E3. E1, the overlap measure, may require more than one pass, because a target and test unit pair which do not overlap will have a score of zero, which is unaffected by changing any bits not set in the target.

5.6.3 Results

There are five different operators to compare, with three target nets and three evaluation procedures. To each evaluation, noise and or deception may be added. In addition to regular GA parameters such as population and generation size, selection scaling factor and mutation rate, each of the recombination operators has parameters such as transmission probability p_t. It is clearly not possible to report detailed results for all the possible combinations. A number of extensive runs were made, stepping through the key parameter values to get a feel for their interactions. The results were used to guide the parameter settings used in the runs that are reported. The main points may be summarised:

All the experiments used rank-based selection, with a geometric scaling factor. In the absence of noise, the selection pressure could be very high, with a scaling factor as low as 0.85, which gives over half the reproduction opportunities to the top five strings. This underlines the essential simplicity of the task. For all the experiments reported below, the scaling factor was set at 0.96, a value more like those used in earlier work with real nets, reported in chapter 6.

The mutation probability p_m has a marked effect on convergence rates, the optimal value being of the order of the inverse of the string length. For Match, Overlap and Random, the probability of producing a child unit by uniform crossover of two parent units at random, p_r, is related to the mutation rate. Both are playing similar roles of introducing diversity. So if one is set high, the other may need to be reduced to compensate. However, their roles are to some extent complementary, and it proved beneficial to have p_r at around 0.1 even at the best mutation rate.

For Overlap and Match, the probability of transmitting matched pairs to the child should be 1.0. This is encouraging, since it indicates that, at least for these evaluation procedures, the matching process is doing something useful.

Uniform and Sort are not sensitive to variations in p_b, the probability of crossover in between unit definition boundaries. It was set at 0.01. For Uniform, p_u, the probability of crossover at unit boundaries should be 0.5, corresponding to picking each unit from either parent at random. For Sort, p_u was best set to 1.0, which implies picking from the two parents alternately. This odd result was checked for many combinations of target net and evaluation procedure, and was consistent. It appears that maximal mixing of the two parents confers some real advantage, but quite how is unclear.

Generation size has a strong effect, especially for Match, Random and Overlap. In the absence of noise, a slow turnover works best: a generation size of 10 for a population of 100. In the presence of noise, a small generation size is inadvisable, because poor individuals which receive a fortuitously high evaluation can linger in the population. However, increasing the generation size to equal the population size causes problems, with the GA unable in some cases to progress beyond about 0.8 of the maximum evaluation score. It appears that the operators are too disruptive and need a reservoir of proven strings upon which to build. The solution is to apply the recombination operator with a probability lower than 1.0, typically 0.8. The remaining 20% of strings are copied from one of the parents at random, and re-evaluated. The re-evaluation is critical and simply reducing the generation size to 80% of the population does not work. Figure 5.5 shows results for Overlap, with N1 and E1, for generation sizes of 10, 80 and 100, and a generation size of 100 with a recombination probability p_x of 0.8, all with population size 100. Without noise, figure 5.5a, there is relatively little difference in final performance, though the run with p_x of 0.8 is worst because of the unnecessary re-evaluations. With noise, the two runs with generation size smaller than the population get stuck, while that with p_x of 0.8 fares best.

Figures 5.6 to 5.8 show the results for the main sequence of tests of the three target net designs, with the three evaluation procedures. All used a population size of 100, with a generation size of 10 for the runs without noise, and 100 for those with. The results shown are for the best individual in the population, averaged over 20 runs from different starting populations. When noise was added to the evaluation, the results are shown for the true evaluation of the string, without the noise. Note that, in order to maximise distinguishability of the lines, the axes of the graphs are not uniform. Figure 5.6 is for N1 (with non-overlapping RFs). Mutation rate p_m was set at 0.001. Figure 5.7 is for N2 (with overlapping RFs), with p_m 0.002. Figure 5.8 is for N3 (random connectivity), with deception. The third method of evaluation is not shown because it is not compatible with the deceptive problem. The evaluation rates the whole string at once, while the deception works at the level of individual units. Mutation rate for this problem was set at 0.01.

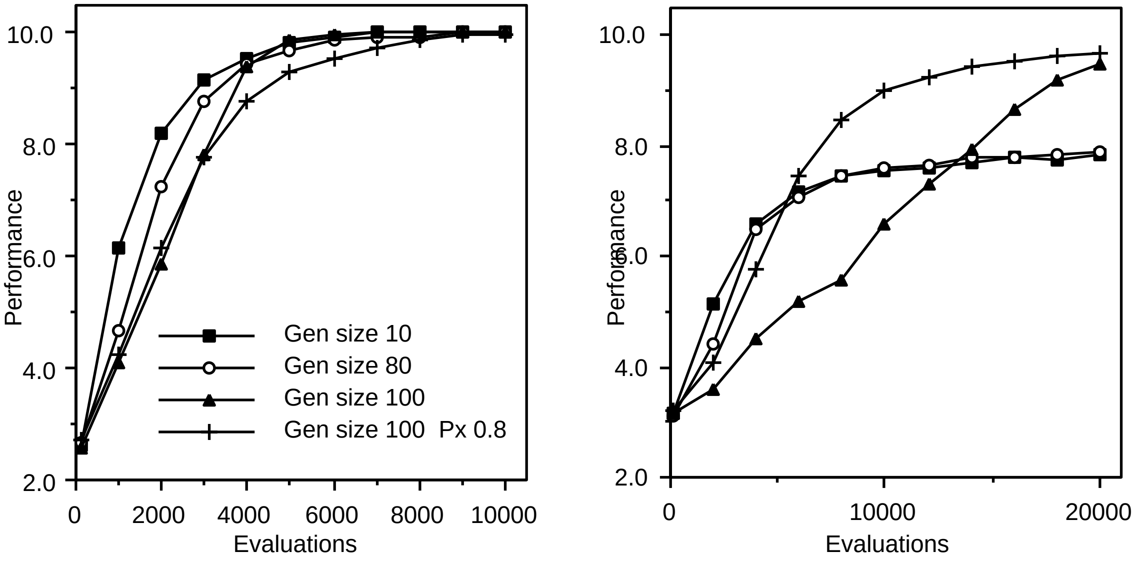


Figure 5.5: Effects of varying generation and population sizes, for Overlap with E1 and N1. a) without noise, b) with noise added

Of the first three recombination operators, there is a clear rank order as expected: Overlap is better than Match which is better than Random. The unexpected result, mentioned above, is that Uniform does so well: in the presence of noise often better than Overlap. It appears that the GA is less troubled by the permutation problem than had been thought. This is confirmed by the results when the permutation "problem" is removed: the results are quite consistently *worse*!

The explanation for this is pointed to by the consistently poorer starting evaluation for NP in all the results. When a string is compared to the target in NP, it has to match each unit with whatever is in that position on the string. When the matching algorithm is added to the evaluation, there are n! ways for each string to be arranged to match the target. With 10 hidden units, there are 3.6×10^6 extra maxima, while the search space remains constant, defined by the string length. On average therefore, the initial population with a permutation scores significantly better than NP. The permutation *problem* is that the strings corresponding to these maxima are all in different orders, so a simple GA should have difficulty combining them. These results indicate that the benefits outweigh the drawbacks and in most cases Uniform is able to keep its advantage over NP. That there is indeed a permutation problem is confirmed by the consistently superior performance of Sort. This reaps the advantages of the permutations, but then seeks to reorder the unit definitions so that they may be recombined advantageously.

Most of the graphs tell the same story: Sort fares outright or equal best in all but one. Sometimes the difference is quite marked, for instance for N1 in the presence of noise, figure 5.6. Note that the gently asymptotic curves can hide quite significant differences in evaluations required to reach a given performance: often a factor of two, which might be several days cpu time with real net evaluations! The performance of Overlap is frequently very close to and occasionally better than Sort, but it seems to deteriorate more in the presence of noise. This is partly an effect of the need to re-evaluate some of the strings and is clearly unfortunate given that the evaluation of real nets is certainly noisy.

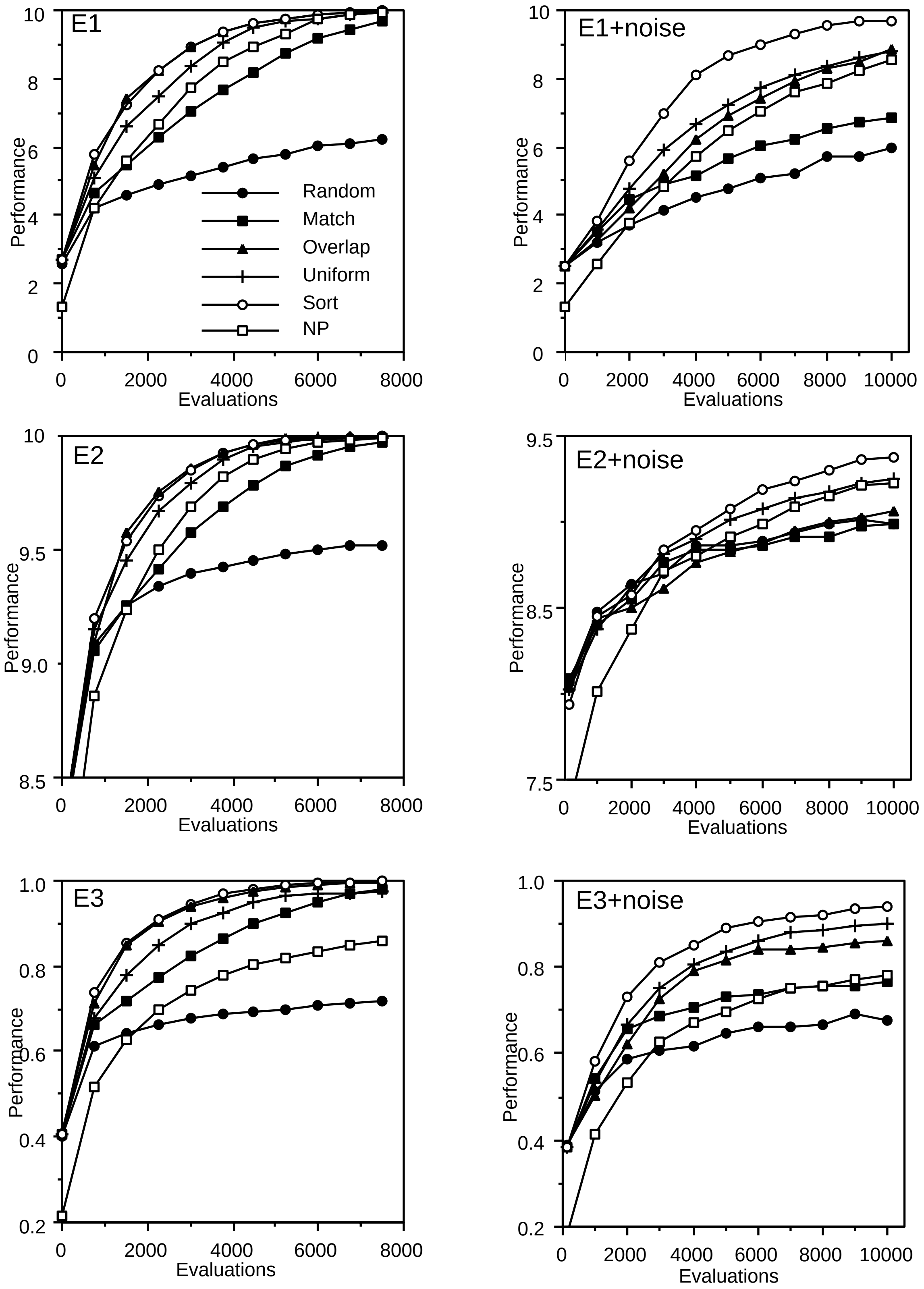


Figure 5.6: Performance of the various recombination algorithms on a 10 hidden unit, non-overlapping receptive field problem N1.

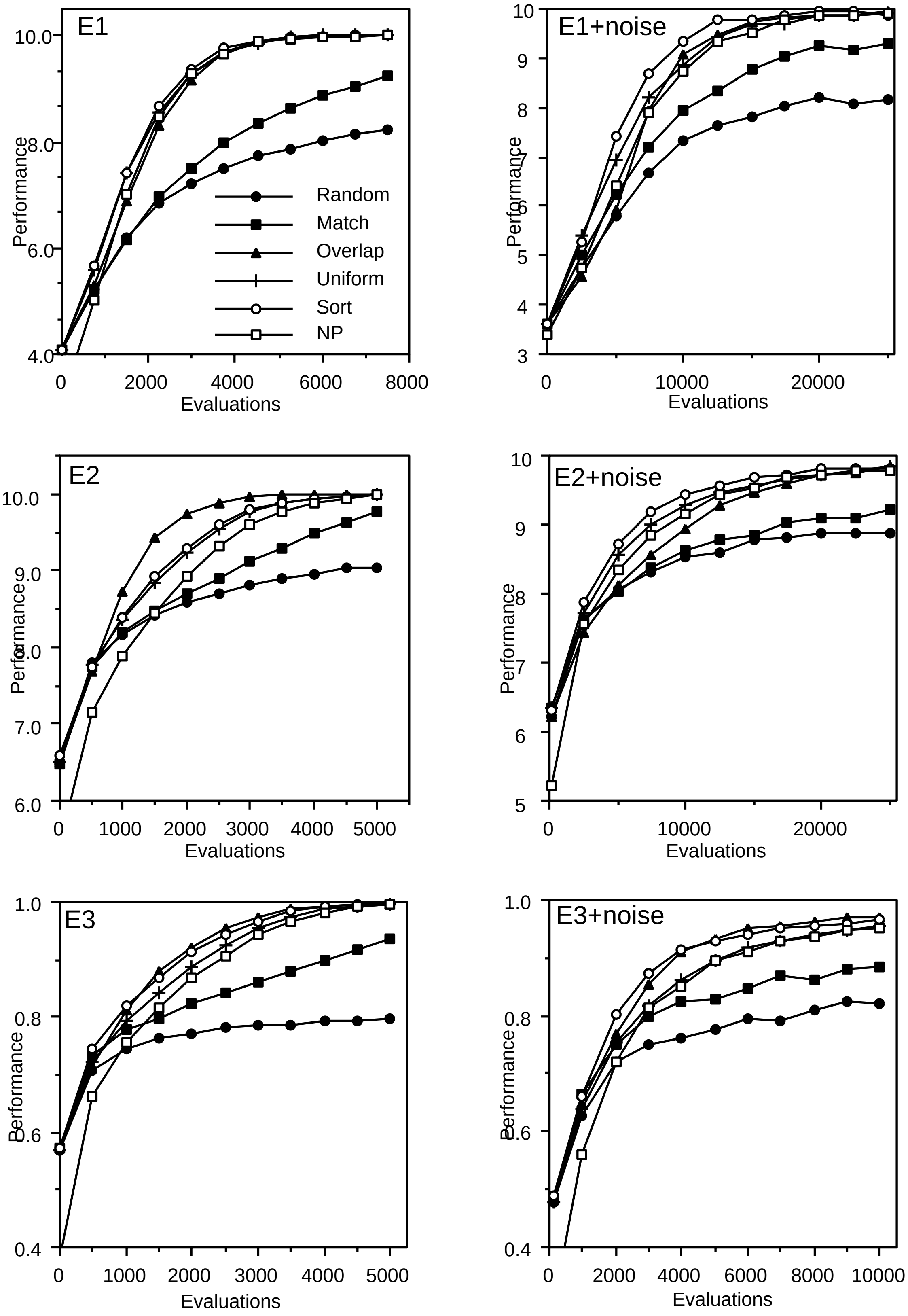


Figure 5.7: N2: Overlapping RF problem

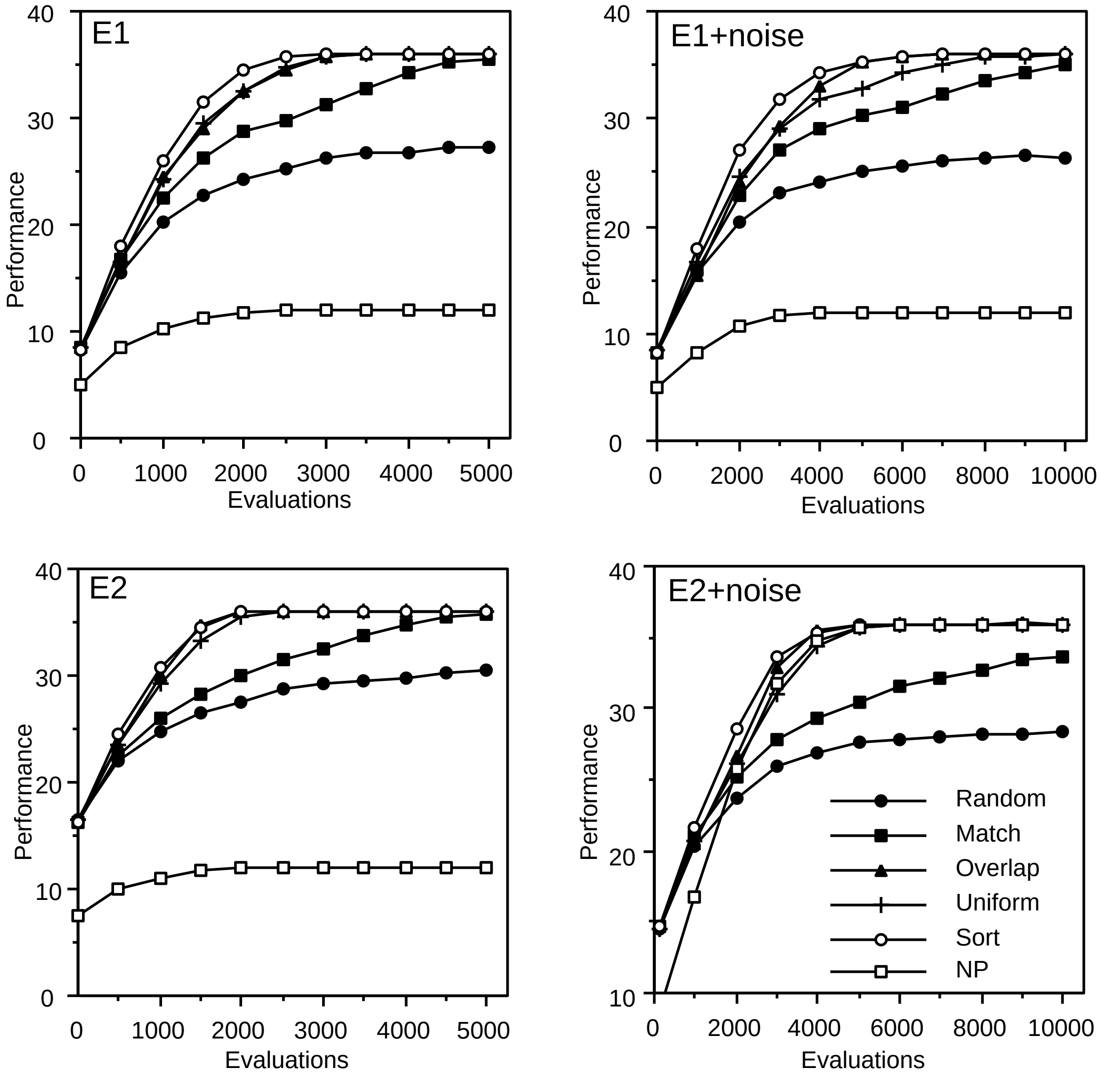


Figure 5.8: N3: Deceptive problem. The failure of NP is discussed in section 5.6.4

It does not appear to be the case that the original evaluation procedure, E1, was unduly favourable to Sort and Overlap: the results from the three evaluation procedures are broadly similar. If anything, Overlap improves relative to Uniform for E2 and E3.

5.6.4 Solving the permutation problem

The most unexpected result here was that permutations are more of a help than a hindrance. Radcliffe devotes much of his thesis to the permutation problem (Radcliffe, 1990b), Belew et al (1990) and Whitley et al (1991) discuss it extensively, while several authors have eschewed crossover to avoid it (Cecconi & Parisi, 1990; Fogel *et al.*, 1990; de Garis, 1990; Nolfi *et al.*, 1990). That permutations should give an initially better evaluation seems clear enough, the surprise is that Uniform is able to maintain its advantage over NP.

It appears that Uniform is able to resolve the permutation problem in practice. An obvious question is how: does it solve the permutation and then get on with the target problem, or do both concurrently? It is possible to observe it in action, by counting how many times a given target string unit appears in each of the possible positions in the population of test strings. If it is in the same position in (nearly) every string, the GA has (effectively) solved that bit of the permutation problem. The test unit definition does not have to be fully correct, or the same in every string, merely closer in each case to the same target unit than to any other. Displaying the data for all the unit definitions gives an indecipherable graph, so figure 5.9 shows the first and last unit to be solved, averaged over five runs, using N1 and E1.

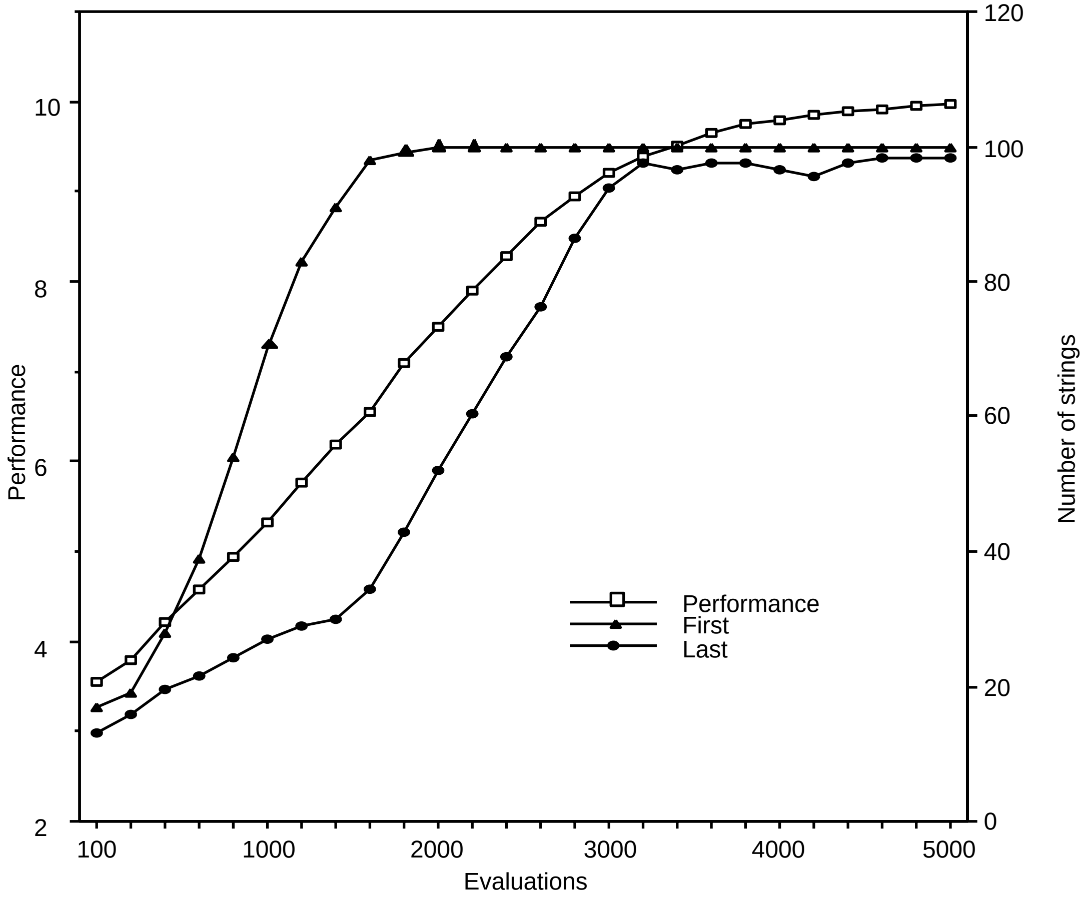


Figure 5.9: Uniform operator solving the permutation problem. In addition to performance of the best string, the number of strings that have the same unit definition in the same location on the string are shown, for the first and last units to be resolved.

This and many other runs (not shown) indicate that solution of the permutations is fairly gradual, and certainly does not precede improvement on the target problem. Resolving the permutations appears to be a semi-sequential process, with each position becoming fixed independently, apart from the last two which must go together. As positions become fixed, the size of the permutation problem is rapidly reduced. However, there is nothing to suggest that the process of solving it accelerates, indeed, figure 5.9 indicates that the final pair of positions becomes fixed more gradually than the first one. This is probably because there is more effective competition between the remaining permuted strings.

What evidently happens is that the whole population gradually gets fixed in the order of the best few individuals. The average initial score is about 3 out of 10. If it is supposed that this results from having got three units correct, and seven with no score (obviously not the case), then it is possible to calculate the probability of combining two random individuals and producing an offspring with four or more units correct. It is in the order of 0.2, quite high enough for the GA to make progress. MonteCarlo simulations would be required to estimate the probabilities for the real problem. However, it is evident that the permutation problem is not as severe as had been thought. It is not necessary to solve it in one go, provided there is a reasonable chance of bringing good alleles together, the GA will do the rest.

A sceptic might suppose that crossover is not contributing to the solution, and that permutation problem is overcome by a mixture of selection and mutation alone, i.e. the system is acting as what Whitley et al (1991) call a "genetic hill-climber". This is easily checked, figure 5.10 shows Uniform, using N1 and E1, with and without crossover enabled, at three mutation rates. Mutation alone evidently can solve the problem, but even with this very simple problem the addition of crossover causes a marked improvement.

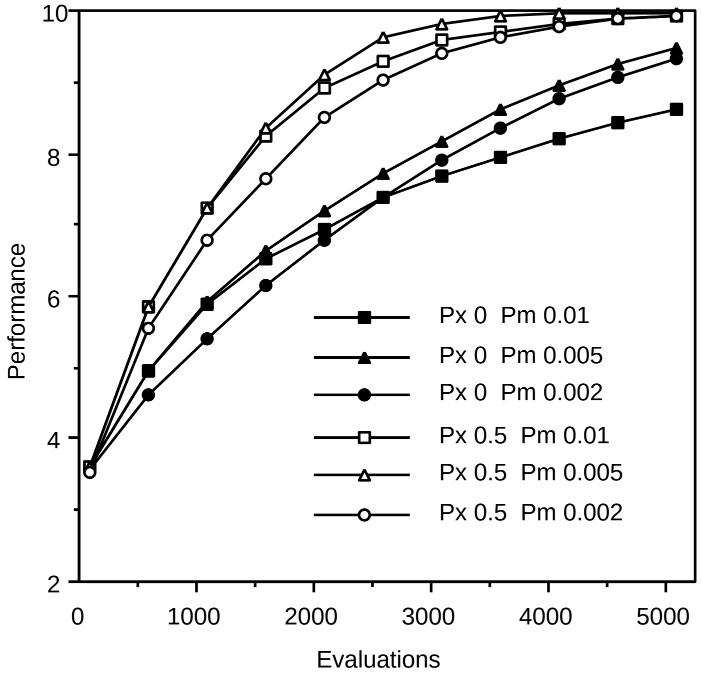


Figure 5.10: Uniform on N1 with E1, with and without crossover enabled

Resolving the permutations is aided by high selection pressure: by increasing the dominance of the top-ranked string, it is better able to enforce its order on the population. It was therefore thought that deceptive problems would pose a challenge, since too high a rate of convergence would appear to reduce the chances of finding the deceptive solution. As figure 5.8 shows, NP was actually worse than Uniform, failing to solve the deceptive problem in three out of four cases. The reason is the same as before: when many permutations can be tried there is a better chance of finding a combination of units that scores well on the harder task. If the value of the second string is increased sufficiently, NP will also find it every time. NP is successful when using Hamming distance, E2, in the presence of evaluation noise. The noise appears to inhibit convergence long enough to allow the better strings to emerge. However, despite extensive testing, no problem could be found for which NP outperformed Uniform, unless the selection pressure was artificially reduced (Hancock, 1992a). The permutation gains still outweigh the drawbacks.

This raises the intriguing possibility that it might be worth deliberately *adding* permutations to a problem where they do not exist. This possibility was tested using Genesis and a modified version of deJong's F1, see section 4.6.3. The modification was to make the three target parameter values different, so the optimal values were at -1, 0 and 1. The three test values from each string were tried in each position, with the best evaluation result being returned. As expected, the initial performance was better than for an algorithm without the permutation search, however, with the relatively weak selection pressure available with Genesis, the advantage was maintained for only around 2000 evaluations. Even this advantage is largely illusory, since each "evaluation" of the permuted problem is in fact six separate evaluations, with the best value being returned. With most problems, every different ordering of the parameters will require such separate evaluation. There is therefore little mileage in adding in permutations: the point about a net design is precisely that it doesn't matter which unit is defined where: all orderings will give the same result.

5.6.5 Summary

This section has presented an empirical comparison of five recombination operators on a simulated net-design task. On this evaluation, the Overlap operator performs better than the simpler Match. However it is rarely better, and sometimes worse, than a simple crossover operator and on this evidence the considerable extra complexity cannot be justified. The permutation "problem" was shown to have two components: a beneficial increase in the number of possible solutions and a difficulty in bringing together different bits of the solution. For these problems, the advantages outweigh the drawbacks, and a GA using the same crossover operator, but without the permuted evaluation consistently does worse. That there is a problem in combining building blocks is demonstrated by the consistently good performance of Sort, which attempts to realign unit definitions prior to crossover. On the basis of these comparisons, Sort is the operator of choice.

Chapter 6

Gannet simulation work

6.1 Introduction

Gannet is a rather obvious acronym for the combination of genetic algorithms and neural nets, and its use seems to have evolved independently on at least three other occasions (Radcliffe (1990b), who is principally concerned with learning weights, Spofford and Hintz (1991) and Robbins et al (1993)). The author has used it in the context of net structure specification (Hancock, 1990a; Hancock & Smith, 1991), which is the subject of this chapter.

The underlying motive for the work in this chapter is the production of better-generalising nets, as was discussed in section 3.2, by matching the structure of the net to the problem. Because of the permutation effects discussed in the previous chapter, the focus shifted to an investigation of the problem of how best to code the net on the genetic string, and how to construct a useful recombination operator. The nets are feed-forward designs for use as classifiers: amongst the simplest types of net, but quite complex enough to demonstrate the permutation effects described in the previous chapter. Most of the work was completed before the results reported in the previous chapter: it was the difficulty of obtaining statistically significant results with real nets that prompted that work.

Three separate experiments are described in this chapter. The first started out as an automated method of searching the space of net designs for the face recognition data of chapter 3 and continued to look at ways of addressing the permutation problem. The overlap measure described in the previous chapter was conceived during this work. The second experiment reports results from the use of Whitley's (1990) pruning method on some of the same data. The final experiment looks at the efficacy of the various recombination operators described in the previous chapter on the real problem of designing nets.

All of these experiments consider feed-forward nets used for classification of input data. The GA specifies the internal connectivity of the net, which will then be trained by Backprop, or some equivalent. The principal aspect of performance to be optimised is therefore the ability to generalise from the training data. An estimate of this ability is obtained by scoring a test set. Subsidiary aspects of performance are training time and net size, which are not necessarily simply related. As stressed in chapter 4, the evaluation procedure used is important, especially since the test performance of a given net is so variable. In section 6.3.2, a method is introduced which gives extra evaluations to the best designs, and its efficacy is demonstrated.

The ability of a GA to design a net may be considered at two levels. The first is the level of feasibility: is it possible for a GA to manipulate net structures in such a way as to produce a better test score? For this purpose, the test data may be within the loop of the GA's evaluation cycle (though not, of course, seen by Backprop during training of the net). If the GA is successful, the net may produce very high scores on that particular test set, which would not be a true estimate of its wider generalisation ability. The second level asks whether a GA can stand comparison with other ways of optimising a net: constructive or pruning algorithms, weight decay etc. In this case, the test data must be kept out of the GA's loop, in order for the comparisons to be valid. Most of the work reported below is at the first level, of feasibility studies.

	Parameter	Possible values
1	Active unit	{0,1}
2	Cluster type	{0,1}
3	RF x-centre	{1,2,..,11}
4	RF y-centre	{1,2,..,11}
5	Receptive field type	{0,1,..,7}
6	Contact blob position units	{0,1}
7	Contact blob mass units	{0,1}
8	Contact blob width units	{0,1}
9	Contact blob height units	{0,1}
10	Contact blob orientation units	{0,1}

Table 6.1: The ten parameters that specify a cluster of hidden units.

6.2 Face recognition

The work described below continues looking at the face recognition data from chapter 3. The experiments reported there indicated that the internal structure of the net had a significant effect on the test performance. Manual search of possible structures was laborious, however, since enough runs to achieve significant differences took several cpu hours. A GA was built to explore the structure space automatically. The dataset chosen was the Type 2 coding, for two reasons: the combination of topographic and symbolic coding gave a large input space to work with, while the best manually-coded figure of 92% on test left more room for possible improvement than the Type 3 coding.

The investigation proceeded in two phases. Initially, the net specification method was quite powerful, in that arrays of hidden units like those found to be useful during the manual search were generated by one set of parameters. This specification method turned out to be so powerful that it left the GA little to optimise. So in the second phase, the task was made harder by weakening the specification method. This raised the permutation problem, which was addressed by using ordering operators prior to crossover.

6.3 Phase 1

6.3.1 Net coding

The size of net developed manually, several hundred input units and perhaps a hundred hidden units, ruled out the possibility of a complete, strong coding like that of Miller et al (1989). A system closer to that used by Harp et al (1989a) was employed, but more specifically tailored to the given task. The number of units and connections is specified precisely, but a whole cluster is defined by one set of genes. These were set up so that the GA could explore a similar space to the earlier manual methods. As the earlier work had indicated no benefit from more than one layer of hidden units, only one layer was allowed. This considerably simplifies coding, and means that relatively few checks are needed for “illegal” designs such as isolated units.

The genetic string has eight blocks, each defining a “cluster” of hidden units. Each block

a) Single unit “clusters”

Type:	0	1	2	3	4	5	6	7
Synapses	1	5	9	13	21	25	37	49

b) Array type clusters

Type	0	1	2	3	4	5	6	7
RF size	6x6	7x7	5x5	3x3	8x8	5x5	2x2	7x7
RF Overlap	1	3	2	1	5	3	1	5
Units	4	4	9	25	4	16	100	9
Synapses	144	196	225	225	256	400	400	441

Table 6.2: Receptive field sizes and patterns. a) for single units where type codes the number of synapses, b) for array type clusters.

has ten parameters, listed in Table 6.1. The first parameter is a switch which specifies whether the cluster actually exists. If it is off, then the rest of the block is ignored when the string is being interpreted. This method allows a variable number of clusters without the added complexity of allowing variable string lengths. It has a side effect in that unexpressed blocks may still undergo mutation without suffering any selection. A mutation on the switch bit will then cause a new, untried cluster to appear.

The second parameter specifies one of two types of cluster. One type is in fact a single hidden unit, which may be connected to a specified subset of the input units. The other type is a square array of units, matching the topology of the input layer, each with its own receptive field (RF). The array format was that found to be useful during the manual trials. The single unit type was added to give room for the algorithm to put extra discriminative power, in the form of more trainable inputs, where it was needed, perhaps in the centre of the face.

The next two parameters are only meaningful for single unit clusters, and specify the x and y centres for the specified receptive field. The same comment about hidden mutation will apply: if the block specifies an array cluster the x and y parameters will not be subject to selection, but may be expressed later if a mutation causes the block to specify a single unit. The fifth parameter specifies the RF of the unit(s) in the cluster. This is implemented via a look-up table which differs for the two cluster types and is shown in table 6.2. The RFs are arranged in order of increasing net size, in terms of synapses and units. In phase one, these three parameters were coded directly as integers, see section 4.6.1.

The remaining five parameters are simple switches that specify which of the five types of input unit members of the cluster will contact. A check is made when each new offspring is generated to see that at least one of these switches is on. If not then one is turned on at random. A similar mechanism ensures that at least one of the blocks specifies an active cluster. No other checks for net “legality” are required.

No attempt was made at this stage to address the permutation problem: precisely the same net may be coded by many different strings. However, the worst effects of the problem are ameliorated by the array cluster specification. Here, one set of parameters defines potentially a whole layer of hidden units, in a specified order. The relationship between all these hidden units is therefore fixed. However, there remains the problem that another string might contain the same array specification in a different location, so that crossover produces two such arrays or none.

The output layer was fully connected to whatever hidden layer was produced by the GA. It is obviously possible that reduced connectivity between these layers would be beneficial, but it was decided that specifying one layer was complex enough. It also maintained continuity with the manual design approach. For the same reason, the five unit output coding of the earlier experiments (section 3.4.1) was retained.

6.3.2 Evaluation

There are a number of possible performance criteria for nets. Usually the most significant is its generalisation ability, since this is the task the final net will have to perform. This is usually estimated by counting errors on a test set, unseen during training. Where the training set is difficult, or the aim is fast learning, the squared training error may be used as a target e.g. (Schiffmann *et al.*, 1991; Anthony *et al.*, 1990). Simply learning the task may be the final aim: for instance in neuro-controllers such as a pole-balancer (Wieland, 1990; Whitley *et al.*, 1991), where the score is the length of time balance is maintained. Harp et al (1989b) use a performance measure constructed from several components, so that they may tune the search. For instance they can give credit for having units with a small fan-out, which is advantageous if the design is ever to be cast in hardware. In the work reported here, we are principally interested in reducing the classification errors on test, and so this is the measure used.

The scoring was done as for the manual test (see section 3.4.1), again using Quickprop (Fahlman, 1989) for training. The procedure involves repeatedly segmenting the data into training and test sets, such that each set of images is used in turn as the test set. This kind of testing, a form of cross-validation, is familiar to statisticians as a means of evaluating a classification algorithm. As noted above, since the test data is within the evaluation loop of the GA, the performance cannot be taken as an indicator of the net's wider generalisation ability.

The effect of noise on the evaluation is a significant issue. Backprop trained nets vary in performance from run to run: especially if the net is under-constrained by the data. One aim of the work is to generate a net that is well-constrained by the data, so it might be hoped that evaluation noise would gradually decrease, but the question remains of how much effort to expend on reducing noise. There is some evidence (Grefenstette & Fitzpatrick, 1985) that it is better to get quick, noisy results and therefore get through more generations than to spend time assessing each individual. One evaluation here took about 20 minutes of MicrovaxII cpu time, but unfortunately also gave a reasonable chance of scoring 100% on test. If the best nets are to be distinguished, some averaging seems necessary. The scheme adopted was to retest any net scoring more than 90% and taking less than 2000 cpu seconds to complete the test procedure. If the average was still better than 94%, a further two tests were run and the results averaged again: relatively few nets achieved this. The efficacy of this evaluation procedure was tested by observing the elimination of a penalty bit: see section 6.3.5.

While the primary desire was to improve test performance, it seemed wise, not least because of the evaluation times, to encourage the production of fast learning and smaller nets. This was done in three ways. Nets that required more than 2000 cpu seconds to evaluate received a fixed penalty of 5%. When the strings are ranked for selection, nets with equal test scores, a fairly frequent occurrence due to the rather coarse quantisation of the test scores, preference was given to those using less cpu time. Slow learning nets were further penalised indirectly by setting a limit of 30 epochs. As the best manual design required an average of 29 epochs, this was, intentionally, a tight limit. Large, slow-learning nets are also most likely to hit the cpu-time limit.

Because of the evaluation noise, there is no guarantee that the highest ranking net found during the run will remain the best when tested again, external to the GA. Occasionally an average net will perform well enough to appear high on the ranking list. Several of the better nets were therefore retested manually, averaging over at first 20 and later 50 restarts. The best scores seen during the GA run are inevitably optimistic, being samples from the top end of the distribution of performance for the net in question. This was reflected in the consistently lower scores observed when nets were retested after the run.

6.3.3 The Genetic Algorithm

Crossover was limited to the cluster boundaries, an arbitrary decision (considered further below in section 6.4.5) that will encourage the system to explore new combinations of existing clusters rather than new cluster designs. Crossover probability was 0.5, meaning that cluster definitions were chosen, in order, randomly from either parent. Mutation acted as usual for binary switches, but was weighted to produce relatively small changes in the values of the integer variables. Mutation rate was set at a probability of 0.02 per gene, a higher than usual rate because the cpu load required the population to be rather small for the size of the parameter space. Goldberg's (1989c) analysis suggests a population of over 150,000 for a string length of 80. As that was out of the question, a traditional (DeJong, 1975) value of 50 was chosen for most of the runs. The effects of a small population were reduced further by using a generation size considerably smaller than the population, typically 10 (in DeJong's (1975) terminology, a generation gap of 0.2).

Selection was done on the basis of rank, using a scaling method (see section 4.4). This helps to prevent premature convergence caused by exceptionally good individuals dominating the selection, a problem likely to be exacerbated by the noisy evaluation. It also obviates the need to combine test score and cpu time into a single figure of merit. The selection algorithm used was initially a simple roulette wheel, later replaced by Baker's optimal stochastic universal selection algorithm (Baker, 1987).

6.3.4 Results: selecting for performance

Two initial experiments, with population sizes of 60 and 20, were run for around 60 generations of size 10, which took about 18 days on a MicroVaxII. Both produced several designs that scored 95% on the four trials of the internal evaluation, which fell to around 93% when averaged for 20. The best was 93.2%, a slightly better figure than the best manual design (92.3%), though the difference is not significant. However, this was encouraging, since the best manual design was around 4% better than any of the others. The GA was thus able to match the best that had been done manually. The GA designs learned rather more rapidly, in around 24 epochs, compared with 29 for the manual design. The nets were slightly bigger, but there was still a net saving in cpu time for training.

A control run was then tried, using the same program, but with the selection scaling factor set at 1. Performance was thus not taken into account when selecting parents. The average performance did not improve, as expected. However, in the 28th generation an individual was produced that scored 93% when tested for 20 runs. The net was rather larger than that produced by the "live" GA, but the result does suggest that the GA is not achieving very much.

A new pair of experiments were tried, from the same, new, random start population of size 50. Generation size was 10, the scaling factor was set to 0.95 in one, and 1.0 in the other. Both ran for 50 generations, with very similar results to the first runs. The best live GA design scored 93.5%, the best control, 92.9%. It was discovered subsequently that the best of the initial random population also managed 92%. It thus seems that the GA was achieving very little. The differences are not significant, despite averaging over 20 runs, since the standard deviation of the scores is around 6%.

More significant differences lay in the size of the nets: the live design had 1317 synapses, the control design, 1675, while the random net had 2486. Given the problem of training times with Backprop, this is a useful improvement, but not really worth spending many cpu days searching for.

6.3.5 Evaluating the evaluation

Given that the GA did not achieve very much, it is worth considering whether it was being asked to do the impossible. GAs are supposed to be resilient to evaluation noise, but perhaps there was simply too much for the system to be able to make useful progress. The standard deviation of the scores on the external runs was around 6%: large compared with the differences between net designs. The evaluation procedure was tested by adding an extra bit to the genetic string, which, if set, caused a fixed penalty to be deducted from the evaluation score. Figure 6.1 shows that the system is able to eliminate even a 0.5% penalty. This is an order of magnitude less than the standard deviation of the evaluation noise and suggests that the GA should indeed be able to make fine distinctions between nets.

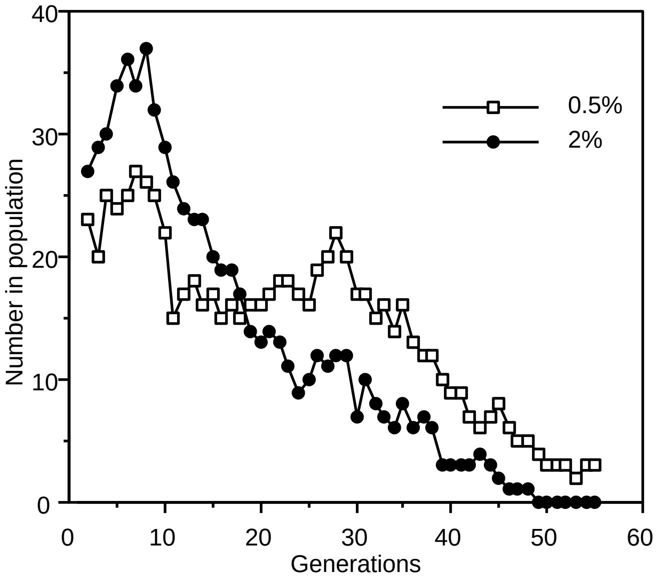


Figure 6.1: Elimination of a penalty flag of value 2% and 0.5%: number of penalty bits set in a population of 50 are gradually reduced.

Selection method	hidden units	connections	test score	training epochs	cpu secs
Test score	101	1333	93.3% (2.9)	24.7 (2.5)	48.8 (5.0)
Speed	26	935	86.8% (4.7)	27.5 (2.7)	29.3 (2.9)

Table 6.3: Comparison of nets selected for test performance and for speed, average of 50 runs. Figures in brackets are standard deviations.

6.3.6 Selecting for speed

Since the improvements seemed to be coming from training times rather than recognition performance, an experiment in selection for speed was tried. The time taken to learn the complete set of 60 stimuli once was measured, and multiplied by the number of epochs to give a figure of merit. Since cpu time is itself dependent on the number of epochs, the evaluation is quadratic in epochs, and linear in net size. It will also be very noisy, but much quicker than the earlier generalisation test. Two runs were tried, both with population size 50, but one with a generation size of 10 and the other of 50. The latter was tried as a hedge against the noisy evaluation: if the whole population is replaced then a freakishly good result will not gain undue advantage.

The results during the run favoured the smaller generation size: which gave a noticeably lower average evaluation time. The scores of both improved during the run, but the end results were similarly disappointing. Not only were the recognition rates consistently below 90%, but the average number of epochs required for training increased to 28 or 29. However, the cpu usage did indeed come down. Despite the emphasis on epochs in the evaluation score, the GA apparently found it easier to reduce the size. Table 6.3 gives a detailed comparison of the results for two of the best nets found by the different selection procedures. The speed-selected net is smaller, requiring slightly more training epochs but much less cpu time. If it were particularly desired to reduce the training epochs, the contribution of this term in the evaluation function would evidently need to be increased.

6.3.7 Phase 1 conclusions

	Test		Training	
Origin	score	Connections	epochs	cpu secs
Manual	92.3%	1175	29	171
Random	92%	2486	23	262
Control	92.9%	1675	27	224
Live GA	93.5%	1317	24	160

Table 6.4: Summary of results for best nets from initial population, control run and the GA, together with best manual design, average of 50 runs.

The aim of this piece of work was to see whether a GA could automate the task of searching for a net architecture that generalised well. The net specification method was deliberately designed so that the GA could readily explore a similar space to that of the manual search.

The results are summarised in table 6.4. Nets were indeed produced that performed at least as well as the best manual design. Since that net had been better by several percent than any other, the aim initially appeared to have been fulfilled. However, the performance of control runs, and indeed of the initial random population, indicate that the power came from the specification method, rather than the genetic search.

The ability to specify individual hidden units was included to give the GA some freedom to add extra discriminatory power where it was needed. The best performing net only had one single unit: when this was deleted, performance averaged over 50 runs fell, but only from 93.5% to 93%. Although the penalty bit experiment indicates that a difference of this size is detectable, there is unlikely to be strong selection pressure on such a single unit.

The manual testing had suggested that the blob orientation information was of little use, and all the later hand-coded designs had no connection to the orientation units. The best GA design included such connections. These were deleted manually, and the net retested by the full procedure of 50 runs. Performance fell from 93.5% to 91.5%: a relatively large difference that suggests that there is indeed useful information in the orientation data. It is precisely this kind of decision that is difficult to make manually: with so many interacting variables it is hard to see which are responsible for improvements. The disappointment is that the GA was not able to improve significantly on the end performance, especially given its extra exposure to the test set during evolution.

The system appeared to hit a limit of around 94% on test. It seemed possible that the limit might either be imposed by the data, or reflect a failure by the optimisation technique. For instance, it might have been that the building blocks offered by the net specification method did not combine successfully, since no attempt was made to address the permutation problem. There might also have been too much of the experimenter's assumed knowledge built-in, such that the net-specification method simply did not allow better nets to be produced. It is certainly the case that the number of connections in the nets produced is still high compared to the number of examples in the training set.

One answer appears from the speed-selected runs, the evaluation performance of which did improve steadily through the generations. It appears that, in this case at least, the GA is able successfully to combine the building blocks to produce better performance. Unfortunately this experiment fell foul of the WYTIWYG syndrome: while evaluation performance improved, the end results on the real task did not.

Near the completion of work reported in this section, a K-nearest neighbours (KNN) classifier was written. This works by measuring the Euclidean distance between a test vector and each of the training vectors. The test vector is classified as the target that appears most frequently among the k nearest training vectors. This algorithm can be expensive in storage space and search time. In this case it is also very effective: values of k from 2 to 6 always scoring 100% on any of the ten different segmentations of the data set. It became evident that the performance limit observed was not imposed by the data. Given the size of the nets, a plausible explanation is over-fitting of the limited training set.

Even before the KNN result, it did not seem worth pursuing this set of experiments further. The average performance was too good for differences to be easily distinguishable. It was therefore decided to create a similar, but harder data set. This could be used as a test-bed for addressing the possible permutation problems more directly.

6.4 Phase 2

In this work the task for the GA was made harder, not only with a new data set, but by removing the ability to define arrays of hidden units in a single definition. The aim here was twofold. By removing array clusters, some of the experimenter's preconceptions of good structures were removed. This would allow the GA freedom to explore new designs, particularly more compact nets that should reduce the problems of over-fitting of the training set. The freedom brings the drawback of a significantly worse permutation problem. In phase one, there had been only 8 cluster definitions. With every "cluster" now reduced to only a single hidden unit, the number of clusters allowed was increased. Since there are $n!$ permutations, the second aim was to find ways to address the problem.

6.4.1 Data

A new data set was produced by doubling the number of identities, and halving the number of examples of each (i.e. 5 examples each of 12 individuals, 6 of each gender). The variety of poses was also increased. Output coding was kept as four bits for identity, one for gender, for compatibility with the earlier work. This meant that 12 of the 16 possible combinations of 4 bits were used. It became clear later that this was a bad choice, and that a simple 1 of n coding would have been preferable, giving better results in less cpu time. However, the main series of results below used this 4 bit coding.

Subsequent to most of the GA work, the data set was classified by the KNN technique. This confirmed that it was more difficult, producing the results shown in table 6.5. However, with k=4 an average score of 92% was returned: better than any of the Backprop results below.

Test set	k:	4	6	8	10
1		83	75	67	67
2		92	83	75	75
3		92	83	83	75
4		100	100	75	67
5		92	100	92	83
Average		92	88	78	73

Table 6.5: Performance of K-nearest neighbours classifier on the second face data set: % correct on the five different segmentations, for varying values of k.

6.4.2 Net coding

The only change to the net coding from the previous runs was the removal of the facility to define array clusters. The second parameter, that defining the cluster type, was removed, leaving nine. Initially the number of cluster definitions (now meaning simply the maximum number of hidden units) was increased to 12. In the light of initial trials, this was further increased, first to 18, then to 36. All these numbers are arbitrary and it is quite possible that a larger number would give better performance. However, the aim is to explore the interaction between GAs and NNs, rather than achieve the ultimate performance on this particular data set.

The three parameters that define the position and size of the receptive field had been coded directly as integers in the phase 1 experiments. For comparison, trials were carried out with a more traditional binary coding, with 4 bits to code the 11 possible RF locations in each axis, and 3 bits to code the 8 possible sizes. One of the disadvantages of this binary coding is the need to do something with the surplus resolution: 4 bits giving 16 possibilities. In this case, each bit contributed 11/16 to the value, which was then rounded to the nearest integer. This meant that some integer values had more than one binary representation. Because of the binary coding, adjacent integers, and possibly even the two different codings of the same value, might have very different bit patterns. To counteract this a creep operator (see section 4.6.4) was introduced, in addition to the usual binary mutation. This allowed mutation to move between adjacent integer values without an unlikely combination of bit flipping.

6.4.3 Initial trials

The purely single unit coding was first tried on the original data set, allowing up to 12 units. A number of GA runs were carried out, varying parameters such as mutation rate and scaling factor, but making no attempt to address the permutation problem. The best performance observed, when averaged over 50 trials on retest, was 90.5%. A number of the better performing nets from these trials were retested against the second data set. The results of this comparison are shown in figure 6.2. There is very little correlation between the scores on the two nets (r=0.13). This may be seen as good or bad: it indicates that the GA was successful in tuning the net to the first data set, but was unsuccessful in producing a net that generalised well to a different example of the same task. The poor generalisation is not surprising given the small number of examples relative to the size of each input vector. The problem is one stage removed from that of training a net with insufficient data. In that case, the net may learn a look-up table on the training set, and fail to generalise to the test set. In this case, the GA may produce a net that performs well on the given test set, by effectively learning the relationship between the training and test sets.

A question of immediate concern, given the lack of much obvious contribution from the GA in phase 1, is whether live runs fared any better than control experiments. This was checked, as before, by running the same algorithm with a selection scaling factor of 1.0. The result was reassuring: the best found by the control run being only 48.9%, despite searching for twice as many generations as a live run that gave 60%. The control nets were also slower to train, taking 4-5 hours for 50 trials, compared with 3-4 hours for the live run nets.

6.4.4 Evaluation

There are two aspects to evaluation here: evaluation of the net within the GA, and evaluation of the GAs themselves. The internal evaluation was modified in the light of early trials to take account of the harder data set: an additional evaluation being carried out if the score exceeded 55%, and a further two if it still exceeded 62%. Evaluating the GA is more of a problem. Being stochastic, performance will differ from run to run. Ideally, therefore, results should be averaged from many runs with different random number seeds. Unfortunately, even with a move to faster workstations, a run of 150 generations of size 10 took about a week. So “many” became four different initial populations, which were used to compare different

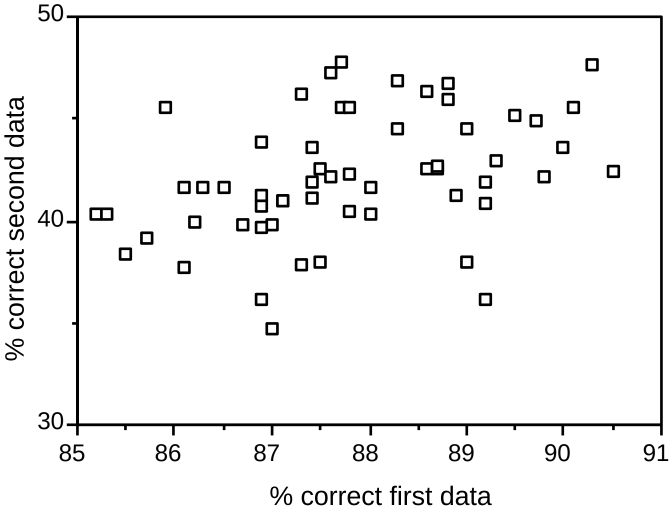


Figure 6.2: Comparative performance on first and second datasets of nets evolved using the first data.

algorithms. At the end of the run, which was arbitrarily fixed at 150 generations, the best five nets from each population were retested, averaging over 50 trials. This produced twenty results, which are the runs shown on several performance graphs below. It will be seen that there is a wide variation in the results from nets produced by a single run of the GA. We are primarily interested in the best performance produced, so one means of evaluation is to compare the best scores produced by a given GA from each of the four start positions. These scores, along with the error bars resulting from averaging over 50 Quickprop runs, are also shown below.

A more formal means of comparison is given by a two-way analysis of variance (anova), the first factor being the two GAs to be compared, the second being the four different start populations. This averages over all the nets tested from each GA. 50 trials of each of five nets from four different runs gives a total of 1000 data points for each GA, which allows quite small differences to be resolved. Usually, $p < 0.05$ is accepted as being a significant difference for one-off comparisons. Since a number of GAs are being compared the acceptable p value needs to be rather smaller, to counter the increased chance that a null result will show as significantly different. The statistic needs to be regarded with some caution in any case, because the range of performances produced even by the five nets from one run can be bigger than the difference between the means for the two GAs. In all cases, therefore, the calculated p value will be given, usually along with the two mean values, μ_1 and μ_2. A value of $p < 0.001$ will be regarded as significant.

There are a number of parameters associated with a GA, which may have a significant effect of performance: population size, selection pressure, mutation and crossover rates etc. A number of early trials indicated no preference for anything other than a population of 50 and a generation size of 10, so these values were adopted for all the other work reported here.

Given the amount of evaluation noise, selection pressure was felt to be especially important: too high and the system might converge prematurely, too low and the noise might swamp useful differences. Eight runs were carried out, varying initial seeds and other attributes of the algorithm, to compare selection scaling values of 0.95 and 0.98. An anova run from a total of 40 extended trials was significant ($p < 0.001$) in favour of 0.95. Some trials with a scaling value of 0.92 indicated that selection pressure was then too high, leading to very early convergence. A value of 0.95 was therefore chosen. This decision was supported later by a repeat of the penalty elimination test, using the proportionate overlap crossover algorithm to be described below. This indicated that a scaling value of 0.95 was able to eliminate a 0.5% penalty, while a scaling value of 0.98 was not. We are not particularly concerned with the optimum value of such a parameter: merely one that works sufficiently well to allow other changes in the algorithm to be compared. Unfortunately, there is always the possibility of interactions between the parameters, for instance different recombination operators might react differently to variations in selection pressure. This might be addressed either by varying the parameters with a meta-level GA (Grefenstette, 1986), or by exhaustive manual variation of each parameter. The evaluation time rules out either possibility: the best that can be done is to keep to fairly conservative values for parameters such as selection pressure and mutation rate, and accept that the various algorithms may not be displaying their full potential.

6.4.5 Other aspects

An obvious question associated with the move from specified array structures to individual hidden units is how many to allow. A couple of runs with the original net specification method on the second data set achieved a best performance of 59.6%. Allowing up to 12 hidden units scored only 53.6%, allowing 18 scored 56.9%, while allowing 36 managed 60.5%, though this net only actually used 19 hidden units. As allowing 36 units achieved a level of performance comparable with the original scheme, this number was adopted for further work. Although the recognition performances are comparable, the array-generated net are much bigger, with over 100 hidden units and typically three times as many connections. Despite requiring fewer training epochs (30-40 cf 70-80), the overall training time is noticeably longer for the array-generated nets (5-6 hours for 50 runs cf 3-4 hours).

Another detail of the GA is whether to confine crossover to the boundaries of hidden unit specifications. In the phase 1 work, this restriction had been imposed, because it was felt that allowing unrestricted crossover between cluster definitions which, because of the permutation problem, might be entirely different would be too disruptive. GA tradition, formalised by Radcliffe as proper assortment (see section 4.6.4), would argue against such limits, since it would reduce exploration of untried hidden unit definitions to the effects of mutation. In this case the mutation probability was already set fairly high to counteract the small population. There was the additional possibility of mutation occurring on the switch bit that controls expression of a unit definition, that would bring into play untried combinations. It seemed that it might be more important to let crossover concentrate on exploring new combinations of existing unit definitions.

This decision was tested, by the full procedure of four runs from different initial populations. The results are in favour of the restricted crossover (anova, $\mu_1 = 57.5$, $\mu_2 = 56.0$, $p < 0.001$). It was therefore used for future experiments.

Coding of integers

Initially, the algorithms that used binary coding of the RF centre and size parameters did not have a creep operator, and direct coding gave better results. Introduction of the creep operator brought about an improvement, so that a comparison between two algorithms that both used the X-Y sort (see below) showed a probable advantage for binary coding (anova, $\mu_1 = 57.5$, $\mu_2 = 56.7$, $p = 0.0013$). This difference may partly be caused by different effective mutation rates, it being difficult to equate the effects of a Gaussian random variable used for the direct coding with the combination of a creep operator and bit flips used for the integer coding. Ideally, a series of runs at various mutation rates would be compared, but since the difference observed was not very big, and the cpu cost of such an exercise would be considerable, it was not pursued further, and all the results below are from binary coding, with a creep operator.

6.4.6 Addressing the permutation problem

Three methods, the first two of which were outlined in section 5.5.3, were developed to try and address the permutation problem in this work. All seek to identify units in the two parents that may be playing a similar role, so that they may be paired prior to crossover. The child will then inherit one or the other.

1. X-Y sort. The hidden unit definitions were sorted according to the $x - y$ coordinates of the centre of their receptive fields on the 11×11 input array. As explained in section 5.5.3 and figure 5.4 on page 93, this will not be very reliable, since the position of units later in the list will depend on those earlier. It will also regard a unit at the right-hand end of one row as being next to one at the left hand end of the row below.

2. Overlap sort. Another problem with the X-Y sort is that there is no reason why two units with the same RF centre should actually have any inputs in common, since there are five different input types. One might contact, say, blob mass units, the other the width and height units. Presumably such hidden units would serve different roles. On the basis that units with similar inputs will have similar roles, they are matched on the basis of what proportion of their inputs are in common. An overlap matrix is built, before pairing units in order of decreasing match.

3. RF centre distance. This was a late addition, matching units simply on the distance on the input array between their RF centres, otherwise building a matrix as for overlap sort. It was intended to address the problem with row ends associated with the X-Y sort, but still has the potential failing that matched units may actually have no inputs in common.

The first experiment compares X-Y sort with unsorted crossover. Anova indicates a clear advantage for the sort procedure ($\mu_1 = 57.5$, $\mu_2 = 55.1$, $p < 0.001$). A graphical presentation of the results is shown in figure 6.3. The left hand graph shows the test performance of all 20 nets, arranged in ascending order for each algorithm. There is no other connection between points in the same x position for the two algorithms: each point may have come from any of the four GA runs. The intention is to give an idea of the range of results from each algorithm. There is an overlap: although the unsorted algorithm is on average around 2% worse, its best results are much better than the sorted algorithm's worst. Such a range of results from nets which were all in the top five at the end of the GA runs gives an indication of the problems of evaluation noise. Some of the worse results come from nets which were generated quite early in the GA run, but got a fortuitously high evaluation, despite averaging over four trials, and survived through to the end. Other, later nets lower in the final ranking might prove to be better on extended testing.

The right hand graph compares the best result from each algorithm for each start population: here, x-position is significant. The error bars are standard errors resulting from the fifty evaluations of each net. In this case, three of the four runs show a significant advantage for the sorted algorithm. That the fourth does not is another indication of the difficulties of evaluating the GAs.

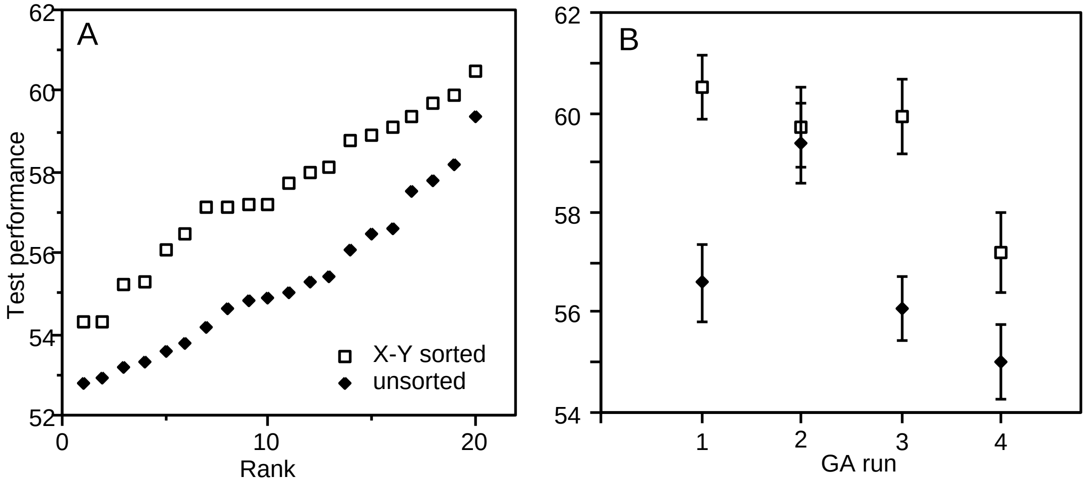


Figure 6.3: Performance of nets produced by GAs using X-Y sort and no sort prior to crossover. A) Best 5 nets from each of four GA runs, in ascending order for each GA. B) Best individual net from each of the four starting populations.

A set of runs was also performed with crossover disabled completely. This was worse than even the unsorted crossover ($\mu_1 = 55.1$, $\mu_2 = 52.7$, $p < 0.001$): recombination is beneficial, even in the presence of the permutation problem.

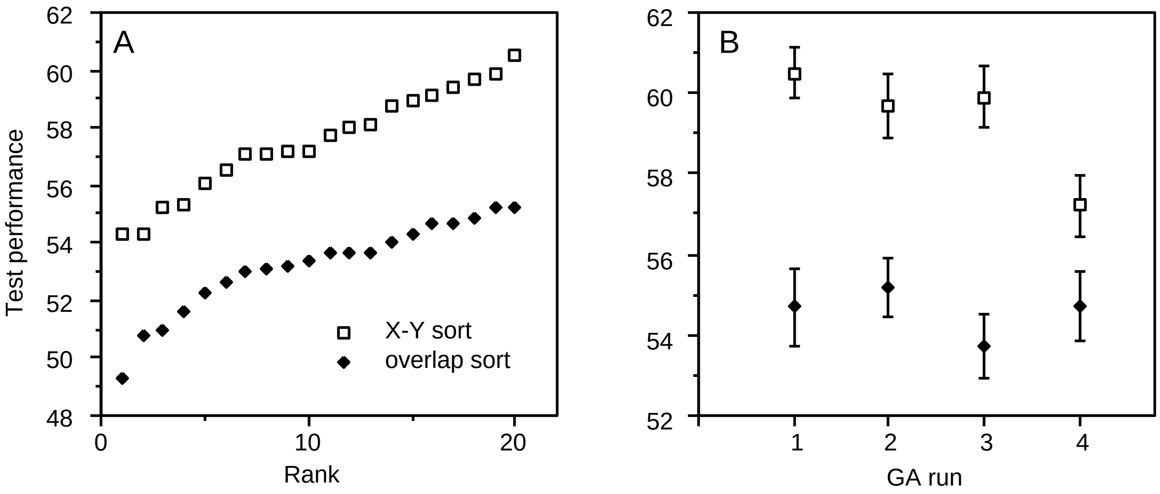


Figure 6.4: Performance of nets produced by GAs using X-Y sort and overlap sort prior to crossover.

Overlap sort

The next comparison is between X-Y sort and overlap sort, figure 6.4. Unexpectedly, the overlap sort is quite clearly worse, by around 4% (μ_1 = 57.5, μ_2 = 53.2, $p < 0.001$). It is even worse than the unsorted runs (μ_1 = 55.1, μ_2 = 53.2, $p < 0.001$). The search for an explanation led to figure 6.5, which plots the test scores against the average number of connections per hidden unit. Although there is not a complete separation, the best results all come from nets with relatively few connections per unit, which are produced by X-Y sort.

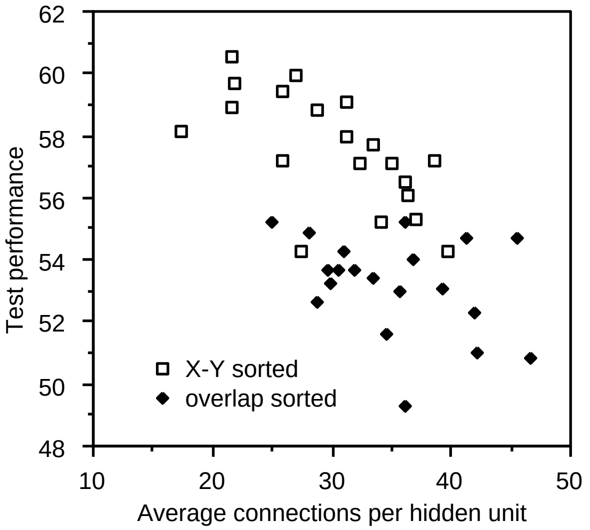


Figure 6.5: Test performance against average number of connections per hidden unit for X-Y sort and overlap sort.

There is no obvious explanation for why the overlap sort should lead to larger RFs. That it was not caused by some systematic bias (a program bug, perhaps) was checked by switching off the selection pressure for both algorithms: there was no obvious drift in size. Failing an explanation, the receptive field size parameter was fixed at 2 in the algorithm with overlap sort, corresponding to a 3 × 3 input array (see table 6.2 on page 112). The performance of the algorithm improved markedly, to become similar to that of the X-Y sort (μ_{xy} = 57.5, μ_{ov} = 57.9, p = 0.07). However, restricting the RF size on the algorithm with X-Y sort improved that as well, though the difference is still not significant (μ_{xy} = 58.5, μ_{ov} = 57.9, p = 0.013). Figure 6.6 shows the results as before, and indicates that averages can be deceptive. The X-Y sort algorithm gives remarkably consistent results, while those from the overlap sort vary by almost 10%, so it returns the 4 or 5 best and worst, the final GA run producing most of the latter. Although promising, this behaviour is somewhat disconcerting: given the cpu costs of a run it might seem advisable to use the more consistent X-Y sort.

To try and separate the two algorithms more clearly, two methods were used. One was to continue the GA runs for another 250 generations, to give 400 in total. As would be hoped, both sets of results improved, but there was even less difference between them (μ_{xy} = 62.5, μ_{ov} = 62.6, $p > 0.1$). The other method was a direct competition, similar to the method used by Fogel et al (1990). An extra bit was added to the string. If set in the first parent chosen for reproduction, X-Y sort is used prior to crossover, if not set, overlap sort was used. The child acquired the bit corresponding to the method used to produce it. The bit was set in half

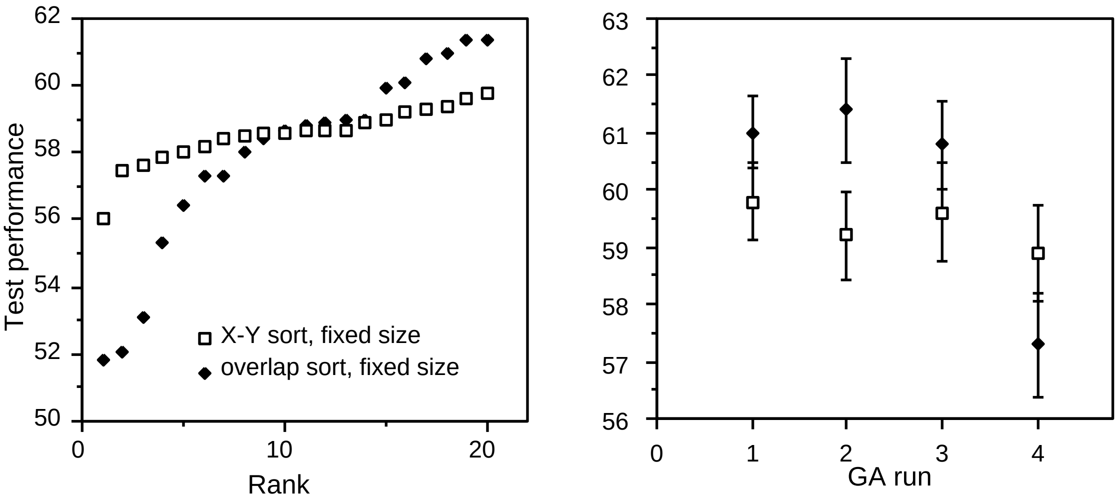


Figure 6.6: Performance of nets produced by GAs using X-Y sort and overlap sort prior to crossover, with fixed size receptive fields

the start population and the GA left to run. If one sort method persistently produces better offspring, its bit setting should take over the population. To guard against the possibility that the half of the population with the bit set happened to be significantly better or worse than the other half, the run was then repeated from the same start position, but with the meaning of the bit reversed. If the effect is genuine, the number of bits set will move accordingly. Since any small differences in efficacy will be amplified by the selection of the GA, this should be a sensitive test.

The first run produced an unambiguous result in favour of the overlap sort, figure 6.7a. Unfortunately the other runs were less convincing, a result reflected in the average of the four runs, figure 6.7b. This suggests that overlap sort is better than the X-Y sort, but the difference must be small.

Sorting on RF distance

The final method of sorting the hidden unit definitions is on the basis of the distance between their RF centres. As noted above, this was a late addition, introduced to try and improve on the already unexpectedly successful X-Y sort by removing the problem that a unit at the end of one row is seen as being close to one at the beginning of the next. However, a comparison without any restriction on the size of RFs indicated no significant difference (μ_{xy} = 57.5, μ_{dist} = 57.0, $p = 0.033$).

The test was repeated with the RF size fixed at 2, and, as with the overlap sort, there was a marked improvement, shown in figure 6.8. With the exception of one stray result, again a fortuitous survivor from an early generation, there is a significant advantage for the distance-based sort (μ_{xy} = 58.5, μ_{dist} = 61.3, $p < 0.001$).

6.4.7 Conclusions

The experiments reported in this section required rather more than a cpu year, running in the background on a number of workstations. Each GA run lasted about a week. One reasonable

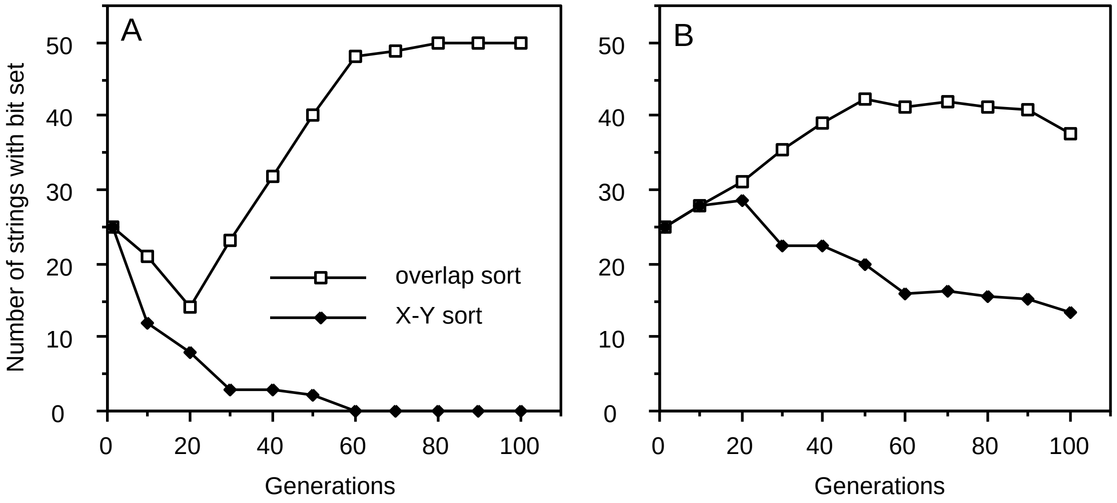


Figure 6.7: Change in frequency of a bit specifying whether to sort by overlap or by X-Y position,: A) First run, B) average of four runs each.

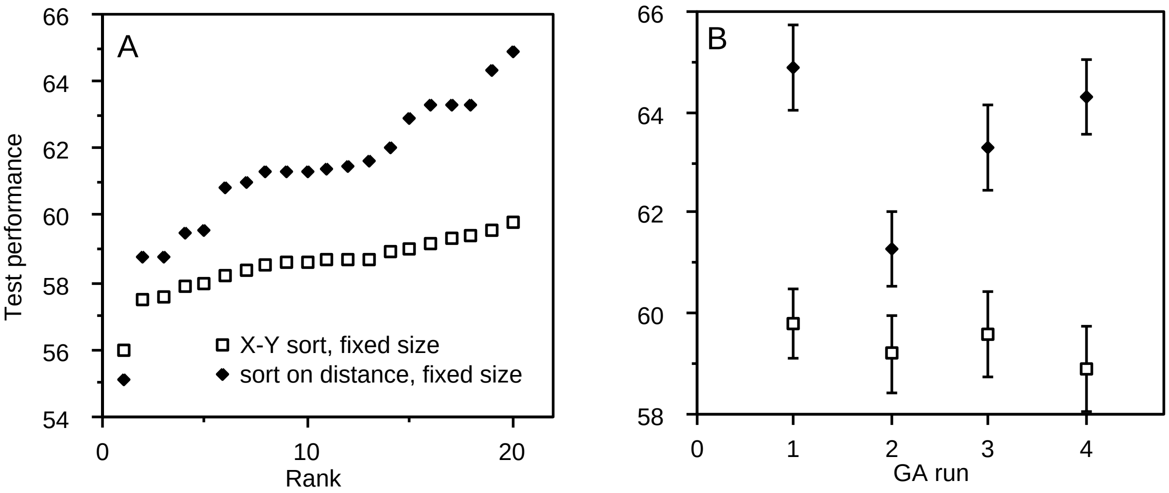


Figure 6.8: Performance of nets produced by GAs using X-Y sort and sort on RF centre distance prior to crossover, with fixed size receptive fields.

conclusion, therefore, is that the project was somewhat ill-judged: especially since the use of only one, rather strange, data set makes the generality of the results questionable. With that caveat in mind, there are a number of points to be made.

Recombination is beneficial, even in the presence of the permutation problem. Sorting unit definitions prior to crossover can give a further significant improvement. However, it seems that matching units by counting common input connections is not especially helpful. Though perhaps marginally better than the X-Y sort, a sort based simply on the distance of RF centres was clearly superior.

The overall results produced are really rather poor, at best around 65%. As mentioned above, this was partly caused by the output coding, that used just four units for identity. A run was tried using one of n output coding and an amended evaluation routine. Any net that scored above the current population average was tested again, if it was still good enough for the top ten, it was tested twice more, if then in the top five, another four evaluations were given. The average scores were recalculated at the end of each generation. Using distance sort for 500 generations produced a net which averaged 89% on test. This is only slightly better than the best of the results from the PCA pre-processed data reported in section 3.5 and worse than K nearest neighbours (table 6.5), despite having seen the test data within the GA loop. The problem is presumably still over-fitted, since this net had 30 hidden units and 611 connections.

The result suggests a failure by the GA: if a smaller net would generalise better, why weren't they produced? Some of the problem may be caused by the coding of the net: which specified that connections be made to neighbouring input units. That this is at least not the entire story is indicated by the improvement observed when the size of the input RFs was fixed. If this size produced better results, the GA should have been able to find it. Perhaps it would, given more time (another cpu month or two...). There was no indication from the runs that were pursued for 400 generations that they had by then stopped making progress, though the population diversity was understandably reduced.

A further indication that the GAs had not converged was the absence of any obvious consistency in the different net designs. A display program was written to show the input connectivity of the nets, unfortunately, they cannot readily be reproduced without colour. In any case, there is little of significance to show, there being nothing obvious to indicate what caused successful nets so to be. The only result that did appear consistent was that the overlap sort was more successful than the X-Y sort in producing nets that covered the whole of the input array. The X-Y sort-produced designs had RFs that tended to cluster along one edge of the input space, an apparent defect which was not reflected in the test performance. The variety of designs produced is also likely to be another reflection of the lack of constraints imposed by the training set: there are probably many similarly effective designs.

Figure 6.9 shows the correlation, or lack of it, of test performance with a number of other measures. 130 of the results were picked at random (or, at least, without any design) from the notebook. Figure 6.9a shows that there is a reasonable correlation between training time and test performance, and in the right direction, better nets training more quickly. Given that training of nets is often inconveniently slow, this is an encouraging trend. There is actually rather little correlation between performance and the total number of connections or the number of hidden units, but a moderate one with the average number of connections per unit. This will be affected by the inclusion of some results from runs where the RF size was fixed, but was evident before those experiments (which is why they were tried). It does seem that there is pressure towards simpler net designs, but since most had several hundred connections there was still a fair way to go.

The evaluation procedure, that retested the better designs, was moderately successful. Its efficacy was demonstrated by showing that it could remove a small value penalty flag from the population. However, designs produced early in the run occasionally remained in the population through to the end, resulting in a scatter of bad results from the final top five nets. The amended evaluation procedure used for the one of n coding, which gave eight evaluations to any net getting into the top five, did seem to produce more consistent end results. An alternative that might be worth considering would be to increase the probability of removal of strings as they get older, a technique used by Robertson and Sharman (1990).

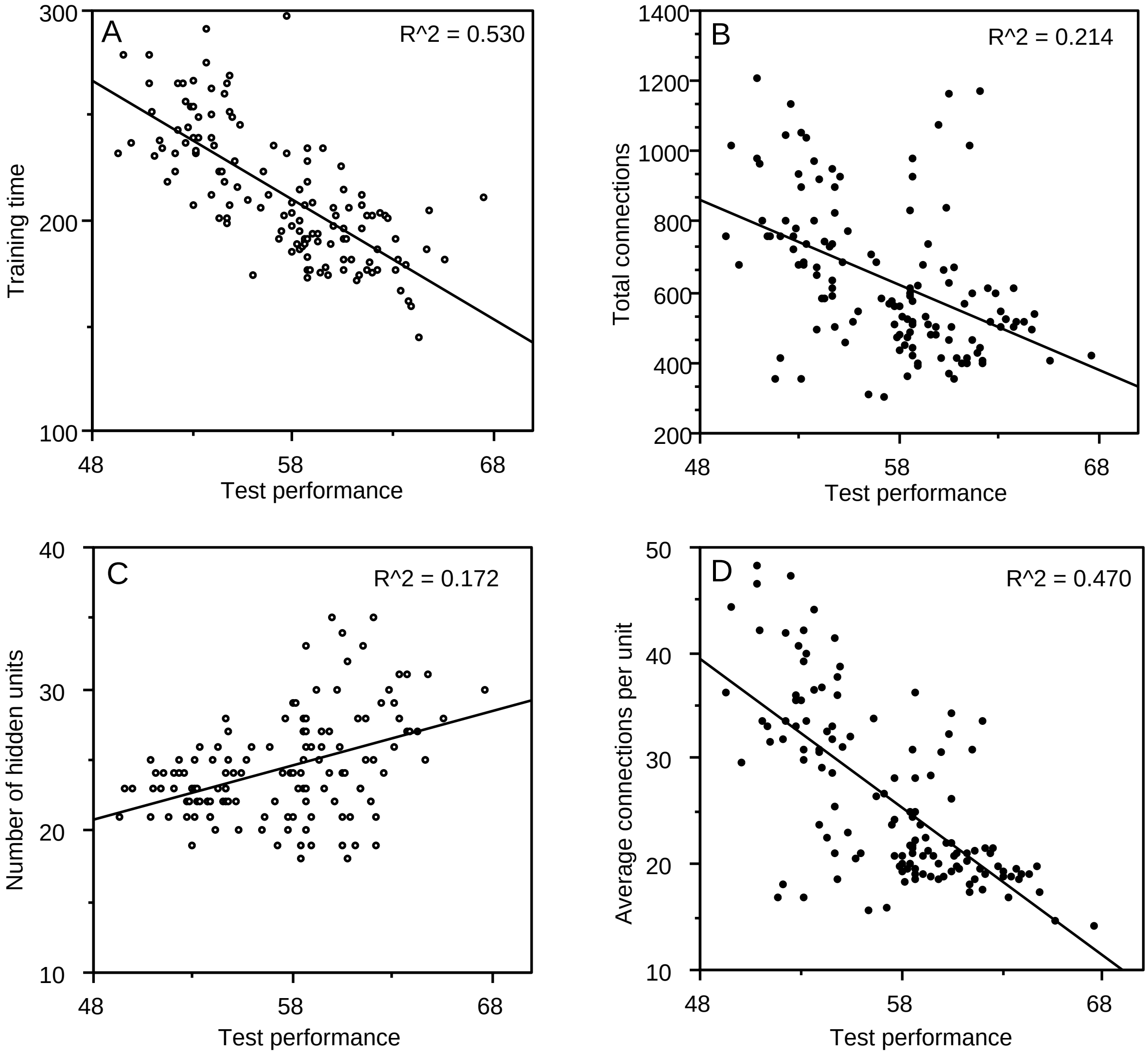


Figure 6.9: Correlating test performance with: A) training time; B) Total number of connections in net; C) Number of hidden units and D) average number of connections per hidden unit.

6.5 Pruning a net by Genetic Algorithm

This section reports some experiments using Whitley et al's genetic pruning method (Whitley *et al.*, 1990), described in section 5.5.1. Weights from a trained, fully-connected net are used to initialise the connections of a pruned version of the net. Whitley reported results only from toy problems, and has not tried using real-world data (personal communication). The main motive behind the method is to reduce the cpu cost of evaluation, but, as noted above, it also addresses the permutation problem almost as a side-effect.

The procedure is to pick a plausible fully-connected architecture, and train it as usual using Backprop (or potentially, some other training algorithm). The genetic string contains one bit per weight. When a net is instantiated for evaluation by the GA, the specified connections are set to the value of the equivalent weight in the trained net. The pruned net is retrained, using the same data. The evaluation is given by the performance on a test set. In these simulations, nets with the same test score were ranked according to which had used least cpu during training.

The evaluation in this case is not noisy, since the initial weights are not random. However, the sensitivity of Backprop to its initial conditions (Kolen & Pollack, 1990) suggests that it might be sensible to hold several different sets of start weights, and use an average score. This would be counter-productive, as it would reintroduce permutation effects. The different sets of weights from a fully connected net would be most likely to represent at least different permutations of the solution, if not completely different methods of solution. It seems reasonable that the permissible pruning will be strongly dependent on the nature of the solution. If the net is required to accommodate a range of solutions, little pruning may be possible. By sticking to just one set of initial weights, the GA will produce a net tailored for that solution. Eliminating repeat tests further speeds up the evaluation cycle.

Two data sets are used in the experiments reported here (Hancock, 1992b). These were one partition of the face data from section 6.4, and the sonar data of Gorman and Sejnowski (1988). The experiments are mostly at the level of feasibility, with the test data within the evaluation loop.

6.5.1 Face data

In the previous section, the face data was segmented five different ways, with the test score being the average of all five. It was not possible to do this with the pruning technique, since each of the five training sets would need a different set of start weights. Just the first partition, which seems to be the hardest, was used. It scored 83% (10/12) by K-nearest neighbours, table 6.5, and 71.7% at best using principal component pre-processing, table 3.14. A fully connected 605:16:5 net was used, which scored 8/12 (67%) on test. The GA was run with a population of 50 and a generation size of 10, using rank selection with a geometric scaling factor of 0.95, a mutation probability of 0.01 and a crossover probability of 0.9. Each net was retrained for up to 20 epochs using online Backprop, $\eta = 0.2$, $\alpha = 0.3$. After 100 generations the best was 11 (92%) correct, with a population average of 10.4 (86.7%). It is evident that the GA is indeed able to tailor the net to give a good performance on the evaluation test set. Whitley et al report that their pruned nets also train well from random initial weights. In this case, however, randomly initialised nets did no better than the original, fully-connected net.

This may not be a problem, since the train, prune and retrain sequence may be an acceptable learning algorithm. However, it remains of at least academic interest whether it is possible to produce a net that does train well from random weights. This was addressed by gradually changing the start weights, using linear interpolation, from their pre-trained positions to random ones, during the GA run. The hope was that the GA would find a path through the space of structures from one that trained well from the initial weights, to one that trained well from a random start. The generation size was increased to match the population size of 100, so that good scores obtained from early generations, seeded by the trained weights, did not remain in the population as the evaluation criteria changed. The selection pressure was reduced by using a scaling factor of 0.99. After 300 generations a net was obtained that consistently scores 11 on the test set, from random start positions. The run required several days on a 2 Mflop workstation.

This result gives some idea of the extent to which it is possible to fit a net to a given data set. The GA has managed to produce an architecture which, when trained on one particular set of data, consistently finds a solution that does well on another particular set. The net is still vastly under-constrained by the 48 training examples, there being 6094 connections!

6.5.2 Sonar data

This data was that used by Gorman and Sejnowski (1988), obtained from the archive at Carnegie-Mellon University. It consists of sonar returns from objects on the sea floor, pre-processed by a Fourier transform to give 60 inputs. The objects are either rocks or mines. There are 104 examples in both training and test set, Sejnowski (1988) is of the opinion that 2 or 3 are misclassified (by the US Navy!). A fully connected net with 10 hidden units scores an average of around 88.5% correct, with a top score of 91%. K nearest neighbours gives 91% correct. The GA was seeded with a net that used 6 hidden units, and scored 89.5% on test. After 100 generations, using the technique of gradually changing to random start weights, a net was produced that scored 97% on test.

Having established that the GA is able to prune the net to improve performance on a given test set, the second question is whether the GA can produce a net that generalises well to unseen data. A validation set is required, to use within the GA's evaluation loop. One possibility is to halve the training set and use each half in turn as training and test sets, summing the total score for the evaluation to be passed back to the GA. Using this method for the sonar data, a net was produced which averaged 94% on the unseen test data, thus almost halving the error of the unpruned net. It has to be said, however, that this result, although itself consistent, was distinctly better than any other obtained, the next best being only 91%, despite a number of GA runs with variations on parameters.

The pruning technique appears to be quite powerful as a method of fitting a net to a given set of data. Unfortunately, on the evidence presented, it is not really possible to claim that it is useful for designing nets with wider generalisation abilities.

6.6 Testing recombination operators, Sort, Overlap and Uniform

The aim of the work in this section was to test the recombination operators described in the previous chapter on some real net design problems. Since the process is extremely cpu-demanding, only the three more promising operators, Sort, Overlap and Uniform, were tested. Three different data sets were used: the face and sonar data sets of the previous section, and a version of the robot arm data from section 3.3.1.

The algorithms and parameters used were exactly as in chapter 5, see table 6.6, the only difference was to change the evaluation procedure to instantiate a net with random weights in the range ±0.1 and train it with online Backprop prior to testing. Population and generation size were 100. The bit mutation rate was 0.001, the probability of mutating the number of hidden units up or down by one was 0.1.

6.6.1 Face data

The same single partition of the phase 2 face data was used as in the previous section. The nets were allowed up to 10 hidden units, and trained for up to 100 epochs, with $\alpha = 0.5$ and $\eta = 0.2$. After 100 generations, the best five nets were retested, averaging over 20 runs.

Parameter	Sort	Overlap	Uniform
Transmission probability p_t	-	1.0	-
Recombination probability p_x	1.0	0.8	1.0
Per bit crossover probability p_b	0.01	-	0.01
Per unit crossover probability p_u	1.0		0.5
Probability of uniform crossover p_r	-	0.1	-

Table 6.6: Parameter values for the three recombination operators

All three algorithms produced nets which scored 11/12 during the run, Sort actually produced two that scored 12. However, the average results of the top five at end of run were unambiguous. The nets produced by Uniform averaged 10.9, while those produced by Sort and Overlap both averaged 9.7. The standard error on these results is around 0.1, so the difference is significant.

It appears that sorting based on input overlap, by either method, does not suit this problem and that its application hinders the GA. Uniform produced two nets that consistently scored 11, a result equalling that from the net pruning, but without the need to change the starting conditions during the run. The nets produced, with 10 hidden units and over 3000 connections, are again hugely under-constrained by the data. The GA again appears able to fit the net to the particular test data.

6.6.2 Sonar data

This was the same data that was used in section 6.5. The nets were again allowed up to 10 hidden units, and this time trained for up to 200 epochs, with $\alpha = 0.3$ and $\eta = 0.2$.

This data set turned out to be too easy to show up any differences between the algorithms. Most runs converged to solutions with just two hidden units and around 65 connections: there isn't much of a permutation problem to solve with only two units. The best nets consistently scored 97% on test, again mirroring the best produced by the net pruning method.

During testing, it was noticed that the test performance tended to peak at around 200 epochs of training. Figure 6.10 shows a typical example, in this case the best appears to be around 180 epochs. The fall-off above 200 epochs is typical of over-fitting. Recall that during the GA run, the nets were trained for 200 epochs. The GA has contrived to produce a net, which, *when trained for 200 epochs*, performs near optimally on this test set, a striking case of WYTIWYG. By way of confirmation, the GA run was repeated, with all parameters the same except that training was extended to 250 epochs. The test results of this are also shown in figure 6.10: this net reaches is peak performance, which as might be hoped is slightly better, at 250 epochs. This run produced the best results observed on this data set: 102 out of 104 correct on test.

6.6.3 Robot arms revisited

The search for another problem that was computationally tractable and yet might require sufficient hidden units to demonstrate the permutation problem led back to the robot arm problem of chapter 3. This requires the net to output the xy coordinate of the end of the

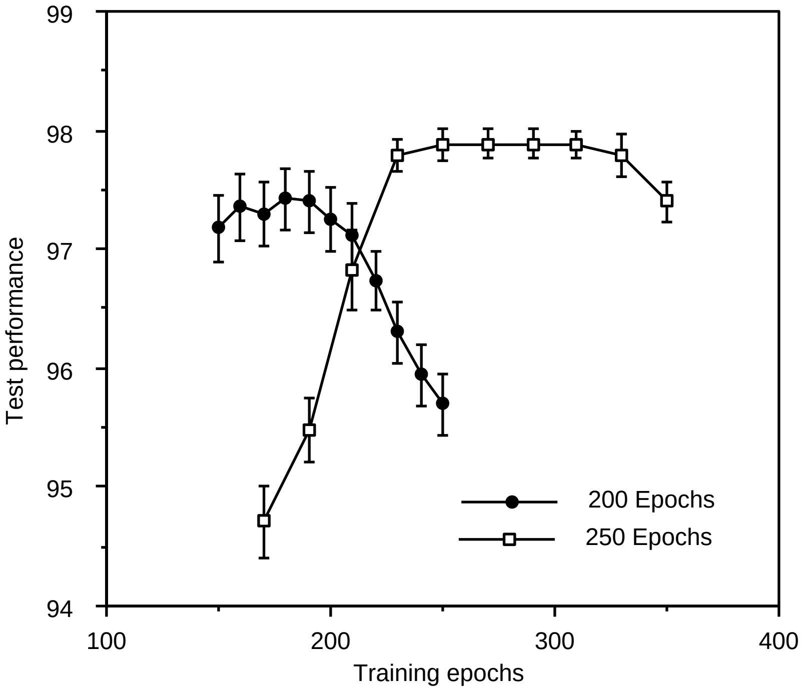


Figure 6.10: Test performance (% correct) of GA designed nets after various lengths of training on sonar data. Results are shown for two different runs of the GA, with different training periods during the evaluation. They are an average of 20 test runs. The fall off with longer training is characteristic of overfitting.

arm, given the four joint angles. Gaussian coarse coding was chosen, since this has a definite input structure, with four units coding the value, at least two of which must be consulted to decode it. The other advantage of this problem is that, because the data comes from a function, the supply is not limited. This allowed the use of a validation set for use during the GA run. 250 examples were used for training, the evaluation for the GA being given by the test performance on a validation set of 100. A separate test set of 1000 was then used to evaluate the nets at the end of the GA runs. The nets were again allowed up to 10 hidden units, and trained for up to 200 epochs, with $\alpha = 0.5$ and $\eta = 0.3$.

The relative performance of the three GAs may be compared in two ways: by recording the best net on the internal evaluation after each generation, and by the usual process of testing the best five nets at the end of run. Results for the former, averaged from three separate runs for each GA, are shown in figure 6.11. The rank order follows that suggested by the experiments of section 5.6, Sort > Uniform > Overlap. This is confirmed by the results from end of run, which show significant differences in the same order.

While the first two runs were in progress, it was noticed that a mistake had been made when generating the data. A parameter in the program sets the standard deviation of the Gaussian response used in the coding, and this had been set to 1.5 at input and output. The best value for the input from section 3.3.1 was 0.5. Since this should give better results, subsequent tests used a new set of data: the same values, but differently coded. For reasons which are unclear, the performance of Overlap improved relative to Uniform, to become indistinguishable from Sort, figure 6.12. Since Sort did well in both runs, it appears preferable.

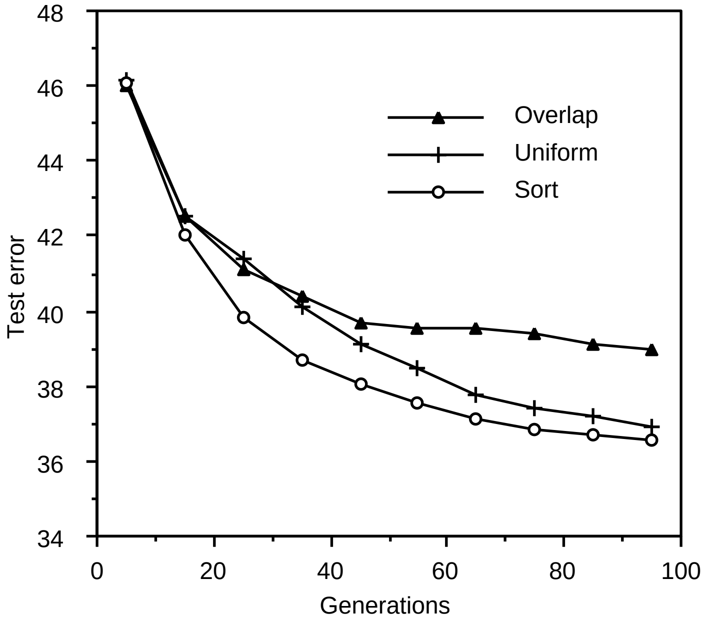


Figure 6.11: Performance of best net tested (on the validation set) during GA runs, first robot arm data set

An indication of what the GAs are achieving is given by figure 6.13, which shows performance on the first data set of the best GA designed net (from Sort) and a fully connected net with 12 hidden units, using the same learning rate parameters. The GA designed net learns much more rapidly than the fully connected version. However, in another clear example of WYTIWYG, the GA design abruptly asymptotes, making little further progress after 400 epochs, while the full net continues its steady downward trend. The GA was asked to produce a net that performed well after 200 epochs of training, and that it has done. In this case it wasn't quite able to peak at 200 epochs, but it is clear that, if the aim is to produce a net which outperforms the fully connected one in ultimate performance, much longer training will be necessary. As it was, these runs took a week each, on rather faster machines than were available for the work of section 6.4.

6.7 Conclusions

There were two principal aims of this work: to investigate the utility of GAs for the specification on net structure, and to address the permutation problem.

The results from phase 2 of the first set of experiments suggest that it is possible to reduce the impact of the permutation problem, by appropriate sorting of the hidden unit definitions. However, sorting based on common input connections does not appear to be generally helpful: although better than no sort in that case, a sort based on RF centre distance was better. In other experiments, sort on input connections is ambiguous, with one success and one failure. A general method has still to be found.

That GAs can manipulate net structures has been amply demonstrated. Whether or not they can do so usefully is another question: the WYTIWYG principle is very powerful. The GA sees only the particular evaluation conditions and will readily overfit the net to those

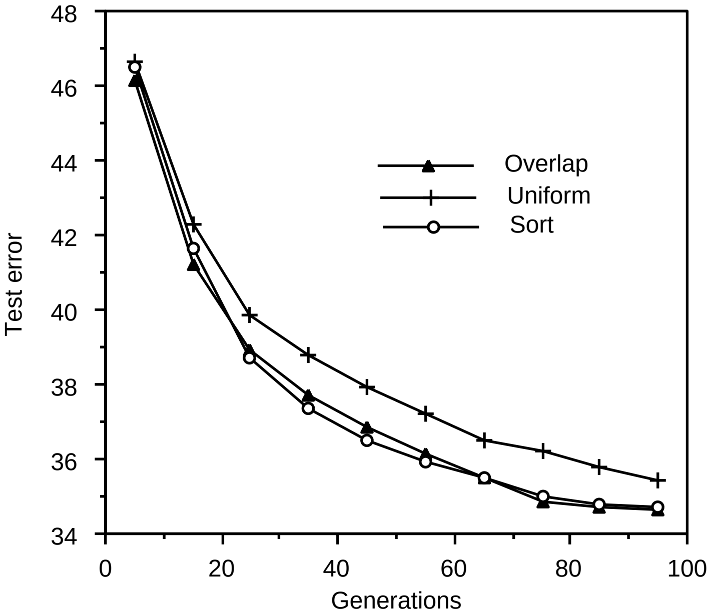


Figure 6.12: Performance of best net tested during GA runs, second robot arm data set

conditions. The result is a net that performs very well as specified, but whose wider utility may be limited.

The final, serious question mark concerns the computing time required. One extended run on a Microvax failed after about 24 days. On investigation, this was found to be because of an integer overflow on the variable used to read in the cpu time! Most of the runs reported here required in the order of a cpu week, and the GAs had rarely converged then. Ever-faster machines will help and the problem is particularly suited to coarse-grained parallel machines such as those based on transputers. Each node can get on with evaluating a net, which will take perhaps a minute or two, so communication overhead is minimal. However, these are still relatively small problems and evolving a better structure for, say, the Nettalk task, which takes hours to train, would still not be practical.

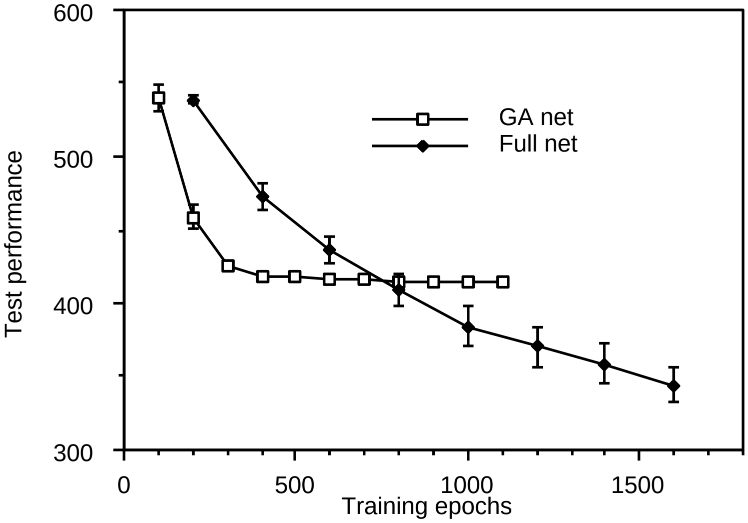


Figure 6.13: Test error of GA designed and fully connected nets after various lengths of training on robot arm data. Average of 5 runs.

Chapter 7

Conclusions

Central to this thesis has been the theme of appropriate similarity metrics. It is no news to computer scientists to say that input-output codings are important: the aim of this work has been to explore the interactions between neural nets, genetic algorithms and coding strategies.

The work reported is largely empirical, which requires the usual hesitations about general applicability of results. One way of strengthening the claim for an empirical finding is to match it to another, especially if the results come from different disciplines. The fit between our PCA data reported in Chapter 2 and the psychophysical model of Foster and Ward is striking. It is perhaps unlikely that there are any single cells with the response properties that our simulation gives, but the match suggests that in some way the two systems are performing similar functions. Roland Baddeley and Ben Craven are pursuing the relationship between image statistics and psychophysics. A promising area for future computational modelling work is to consider the formation of stereo receptive fields, using appropriate pairs of images. The target would be to match the measured RFs reported by deAngelis et al (1991) .

The work in Chapter 3 indicates some of the ways that coding of inputs and outputs can have an effect on the performance of a net. It is also apparent that there are complex interactions between the coding, the learning rule (including aspects such as the necessary output function) and the architecture of the net. That binary coding is not helpful for nets is better known now than when the work in section 3.3.1 was performed, yet appropriate coding still remains relatively little discussed in the recent plethora of NN books. Consequently, those new to the field may continue to use inappropriate methods. The recommendation from the work of chapter 3 is to use a simple analogue or 2-unit interpolation code for continuous variables. Where the data is pre-categorised, for instance into age bands, the coding used should reflect the information that age 1-10 is adjacent to 11-20, for instance by Gaussian smearing of adjacent units' responses (Pomerleau, 1991). If resolution or accuracy is limited, as it may be in hardware implementations of nets, the added redundancy of multi-unit interpolation coding might be valuable.

The encoding of images described in section 3.4 is at best preliminary, though it served to illustrate some of the interactions between coding and architecture, and led to the investigation of genetic design of nets. The method of pre-processing images into a set of blob descriptions is not readily amenable to further processing by a net and requires further thought. A key aspect of future work would be to combine data from more than one resolution, as specified by the full Mirage process (Watt & Morgan, 1985). An observation from section 3.3.1 to be borne in mind is that RF centres at the different resolutions ought not to be coincident, because accuracy is low at the centre.

In nets there is an interaction between coding and learning rules, in GAs there is a similar interaction between coding and genetic operators. At the heart of any application of GAs is the need to identify the building blocks that may be used, and attempt to ensure that the reproduction operators process them effectively. Unfortunately it is therefore the case that GAs, like any other optimisation algorithm, cannot be treated as a black box, and their successful use will always be application-specific. Current attempts to provide a theory of GAs tend to require that the system be simplified beyond recognition, for example to try and estimate the optimal mutation rate, recombination had to be excluded (Bặck, 1992).

Chapters 5 and 6 concentrated on the problem of designing NNs by GA, and particularly on addressing the permutation problem. The novel use of a simulated net building task allowed preliminary comparison of a number of recombination operators in a fraction of the time normally required. An unexpected side result was the discovery that the GA seems able to overcome the permutation problem in practice, so that a simple crossover operator is quite successful, contrary to the analysis of Radcliffe (1993).

The simulated net building results suggested that a method of sorting the hidden unit definitions by overlap of connections assisted convergence. Unfortunately, the empirical tests on real nets gave somewhat mixed results, so it is not possible to claim that a general method has been identified.

A problem common to all attempts to fit a net to data is the possibility of overfitting. The WYTIWYG principle causes GAs to fit the net to the particular test conditions, rather than giving the wider generalisation that is sought. This might be addressed by altering the evaluation procedure. For instance, the fitting of the net to the number of training epochs reported in section 6.6.2 might be reduced by randomly varying the number of epochs, as suggested by Keesing and Stork (1991). A more general approach might come from work done on the problem of overfitting during the training of nets. This has been addressed by the application of Bayesian methods (MacKay, 1991; MacKay, 1992), and on information theory grounds, for instance generalisations of Akaike's information criterion (Akaike, 1974; Fogel, 1991; Moody, 1992). The underlying aim of both approaches is to penalise complex nets. In principle, such methods might be applied to GANNET work. Thus, instead of simply using test performance as the evaluation of a net, its complexity measure might be computed. However, there must be a strong suspicion that the WYTIWYG syndrome would strike again. Fogel's measure, for instance, requires that the distribution of output activations be approximately normal. With so many degrees of freedom to play with, it seems likely that a GA would find some way to satisfy the evaluation procedure, with no guarantee as to its performance on the real problem.

GAs are most relevant to problems for which there is no gradient information available, for instance because of discontinuities in the search space. It thus appears unlikely that they will out-perform gradient descent techniques for training the weights of nets. In addition, it seems possible that recent advances in network pruning algorithms, e.g. Hassibi and Stork (1993), will out-perform GA design of nets at least in cpu time taken, though direct comparisons need to be made. It is a sad conclusion after much work, but the enterprise of using GAs to design nets for tasks such as classification may have little practical value.

A more fruitful area of interaction between GAs and NNs may be the emerging field of "artificial life". Even a simple simulated animal looking for food cannot readily be trained by gradient descent techniques, since the target outputs are not specified. More complex systems, with multiple organisms and richer environments show emergent behaviour, such as a tendency for organisms to avoid each other if competing for food (Floreano, 1992). Such systems will provide a rich testbed for exploring coding strategies within nets, and GAs remain the obvious way of adapting them. As the size of the nets increases, it may be more appropriate to use growth rules, rather than the strong coding scheme used here. With the coding schemes will come a requirement for appropriate genetic operators. Since most currently proposed coding schemes contain some element of the permutation problem, the result that the GA is able to overcome it in practice remains relevant.

Finally, a problem common to many optimisations is noise, against which GAs are said to be relatively robust. Recent theoretical work (Goldberg & Rudnick, 1991) has suggested a basis for estimating the reliability of a GA, if the form of the noise is known. In practical applications, such as evaluation of a net, this information may not be available. In such cases, evaluation of the evaluation procedure, by adding a penalty bit as suggested in section 6.3.5, may offer reassurance that the GA is not being asked to do the impossible.

Appendix: decoding routines

The accuracy of the calculated results in section 3.3.1 depends partly on the algorithm used to decode the unit outputs. Those used were as follows:

Gaussian coarse coding. Each of the units $A..D$ has a value at which it gives a peak response: $P_A..P_D$.

1. Find the unit, x, with highest output, O_x. The decoded value must be somewhere near P_x, but to find out which side of it, it is necessary to consult the unit with the second highest output.
2. Find the unit, y, with the second highest output, O_y.
3. Calculate the offset δ_y from P_y using $\delta_y = \sigma\sqrt{-2ln(O_y)}$
4. Calculate an initial estimate for the decoded value E_y:

   if $P_x > P_y$ then $E_y = P_y + \delta_y$ else $E_y = P_y - \delta_y$
5. Form an estimate E_i for each of the other units i:

   if $E_y > P_i$ then $E_i = P_i + \delta_i$ else $E_i = P_i - \delta_i$
6. Calculate the decoded output value by weighting each of the individual estimates by the actual unit outputs.

   $Output = \frac{\sum E_i O_i}{\sum O_i}$

Value unit coding. If no unit has output > 0.35, return zero. Otherwise, the unit with the biggest response is declared the winner. The value of 0.35, rather than 0.5, was used to reduce the number of false zeros: as each unit is only switched on infrequently the active values tend to be rather low.

Discrete thermometer coding. If the first unit has an output below 0.5, return zero. Otherwise, the units are examined in ascending order, adding one to the decoded value for each unit with output of 0.5 or greater. Counting is stopped at the first unit with output below 0.5. Thus a pattern of outputs such as 1, 1, 1, 0.9, 0.3, 0.6, 0.2... would be returned as 4, with the active unit 6 being ignored. In practice, the nets were not observed to give such spurious outputs.

Continuous thermometer code. Each of the units covers one quarter of the total coded range, R. The units are examined in ascending order. If the output $O_i > 0.9$, it is counted as 1, and $0.25R$ is added to the output value. For the first unit with output below 0.9, $0.25O_iR$ is added to the decoded value. Any remaining units are ignored. Thus, for $R = 8$, the outputs 1.0, 0.95, 0.5, 0.2 would be decoded as 5.

Interpolation coding. Find the unit, x, with maximum output O_x. If it has two neighbours, pick the one, y, with the bigger output O_y. The decoded value lies between the two units' centres of response, P_x and P_y: $Output = \frac{P_x O_x + P_y O_y}{O_x + O_y}$

Multi-unit interpolation coding. The process used is similar to that for Gaussian coarse coding. The offset is easier to calculate: $\delta_i = (1 - O_i) \times s \times g$, where s is the spread, and g is the inter-centre gap. The estimated values from all units with output greater than 0.05 were then averaged. When spread was 1, the average error returned was within 0.01 of that given by the simpler process for simple interpolation coding above.

References

Ackley, D.H. 1987. *Stochastic iterated genetic hillclimbing.* Ph.D. thesis, Computer Science Department, Carnegie Mellon University.

Ackley, D.H., & Littman, M.S. 1989. *Learning from natural selection in an artificial environment.* Internal short preprint of proposed paper.

Adrian, E.D. 1928. *The basis of sensation.* Christopher, London.

Akaike, H. 1974. A new look at the statistical model identification. *IEEE Transactions Automat. Contr.*, **19**, 716–723.

Anderson, C.W. 1989. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, **9**, 31–37.

Anderson, J.A., Rossen, M.I., Vicuso, S.R., & Sereno, M.E. 1990. Experiments with representation in neural networks: object motion, speech and arithmetic. *In:* Haken, H., & Stadler, M. (eds), *Synergetics of Cognition.* Berlin, Springer.

Anderson, J.A., Spoehr, K.T., & Bennet, D.J. 1992. A study in numerical peversity, teaching arithmetic to a neural network. *In:* Levine, D.S., & Aparicio, M. (eds), *Neural networks for knowledge representation and inference.* Lawrence Earlbaum Associates, Hillsdale, New Jersey.

Anthony, D., Hines, E., Barham, J., & Taylor, D. 1990. The use of genetic algorithms to learn the most appropriate inputs to a neural network. *In: IASTED Conference Artificial Intelligence Applications and Neural Networks.*

Artola, A., Bröcher, S., & Singer, W. 1990. Different voltage-dependent thresholds for the induction of long-term depression and long-term potentiation in slices of the rat visual cortex. *Nature*, **347**, 69–72.

Baddeley, R.J., & Hancock, P.J.B. 1991. A statistical analysis of natural images matches psychophysically derived orientation tuning curves. *Proceedings of the Royal Society B*, **246**, 219–223.

Badii, A. 1990. *Genetic Algorithm efficiency through responsive self-bias of search strategy at evolution epochs in optimisation of n-tuple vowel recognisers.* Unpublished report, Imperial College.

Baker, J.E. 1985. Adaptive selection methods for Genetic Algorithms. *Pages 101–111 of:* Grefenstette, J.J. (ed), *Proceedings of an international conference on Genetic Algorithms.* Lawrence Earlbaum.

Baker, J.E. 1987. Reducing bias and inefficiency in the selection algorithm. *Pages 14–21 of:* Grefenstette, J.J. (ed), *Proceedings of the second international conference on Genetic Algorithms.* Lawrence Earlbaum.

Baldwin, J.M. 1896. A new factor in evolution. *The American Naturalist*, **30**, 441–451.

Ballard, D.H. 1987. Interpolation coding: a representation for numbers in neural models. *Biological Cybernetics*, **57**, 389–402.

Barrow, H.G. 1987. Learning receptive fields. *Pages 115–121 of: IEEE International Conference on Neural Networks*, vol. 4. New York: IEEE.

Belew, R.K. 1989. When both individuals and populations search: adding simple learning to the Genetic Algorithm. *Pages 34–41 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Belew, R.K., McInerney, J., & Schraudolph, N.N. 1990. *Evolving networks: using the genetic algorithm with connectionist learning.* Tech. rept. CSE TR90-174. UCSD.

Bengio, Y., & Bengio, S. 1990. *Learning a synaptic learning rule.* Tech. rept. TR571. Montreal.

Bentley, D. 1976. Development of insect nervous systems. *Pages 461–481 of:* Hoyle, G. (ed), *Identified neuons and behaviour of arthropods.* New York: Plenum Press.

Bethke, A.D. 1981. *Genetic algorithms as function optimisers.* Ph.D. thesis, Dept of Computer and Communication Sciences, University of Michigan.

Băck, T. 1992. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. *Pages 85–94 of:* Mănner, R., & Manderick, B. (eds), *Parallel Problem Solving from Nature 2.* Elsevier, North Holland.

Bornholdt, S., & Graudenz, D. 1992. General asymmetric neural networks and structure design by genetic algorithms. *Neural Networks*, **5**, 327–334.

Bos, M., & Weber, H.T. 1991. Comparison of the training of neural networks for quantitative x-ray fluorescence spectrometry by a genetic algorithm and backward error propagation. *Analytica Chimica Acta*, **247**, 97–105.

Bramlette, M.B., & Bouchard, E.E. 1991. Genetic algorithms in parametric design of aircraft. *Chap. 10, pages 109–123 of:* Davis, L. (ed), *Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

Broomhead, D.S., & Lowe, D. 1988. Multivariable functional interpolation and adaptive networks. *Complex Systems*, **2**, 321–355.

Caruana, R.A., & Schaffer, J.D. 1988. Representation and hidden bias: Gray vs Binary coding for genetic algorithms. *Pages 153–161 of: Proceedings of the 5th International conference on machine learning.* Morgan Kaufmann, Los Altos, CA.

Caudell, T.P., & Dolan, C.P. 1989. Parametric connectivity: training of constrained networks using Genetic Algorithms. *Pages 370–374 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Cecconi, F., & Parisi, D. 1990. *Evolving organisms that can reach for objects.* Tech. rept. Institute of Psychology, CNR, Rome.

Chalmers, D.J. 1990. The evolution of learning: an experiment in genetic connectionism. *In:* Touretzky, D.S., Elman, J.L., Sejnowski, T.J., & Hinton, G.E. (eds), *Proceedings of the 1990 Connectionist Models Summer School.* Morgan Kaufman.

Chang, E.I., & Lippmann, R.P. 1991. Using genetic algorithms to improve pattern classification performance. *Pages 797–803 of:* Lippmann, R.P., Moody, J.E., & Touretzky, D.S (eds), *Advances in Neural Information Processing Systems 3.* Morgan Kaufmann.

Chen, S., Cowan, C.F.N., & Grant, P.M. 1991. Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks*, **2**, 302–309.

Chung, S-H., Raymond, S.A., & Lettvin, J.Y. 1970. Multiple meaning in single visual units. *Brain and Behaviour*, **3**, 72–101.

Cohoon, J.P., Hegde, S.U., Martin, W.N., & Richards, D. 1987. Punctuated equilibria: a parallel Genetic Algorithm. *In:* Grefenstette, J.J. (ed), *Proceedings of the second international conference on Genetic Algorithms.* Lawrence Earlbaum.

Collins, R.J., & Jefferson, D.R. 1991. An artificial neural network representation for artificial organisms. *Pages 259–263 of:* Schwefel, H-P., & Männer, R. (eds), *Parallel problem solving from nature.* Lecture notes in Computer Science 496, Springer Verlag.

Cottrell, G.W., & Fleming, M. 1990. Face recognition using unsupervised feature-extraction. *Pages 322–325 of:* Thellier, N. (ed), *International Neural Network Conference, Paris*, vol. 2. Kluwer Academic Publishers, Dordrecht.

Davidor, Y. 1991. A genetic algorithm applied to robot trajectory generation. *Chap. 12, pages 144–165 of:* Davis, L. (ed), *Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

Davis, L. 1989. Adapting operator probabilities in Genetic Algorithms. *Pages 61–69 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Davis, L. 1991a. Bit climbing, representational bias and test suite design. *Pages 18–23 of:* Belew, R.K., & Booker, L.B. (eds), *Proceedings of the Fourth International Conference on Genetic Algorithms.* Morgan Kaufmann.

Davis, L. (ed). 1991b. *Handbook of genetic algorithms.* Van Nostrand Reinhold, New York.

Davis, L., & Coombs, S. 1987. Genetic Algorithms and communication link speed design: theoretical considerations. *In:* Grefenstette, J.J. (ed), *Proceedings of the second international conference on Genetic Algorithms.* Lawrence Earlbaum.

Dawkins, R. 1986. *The blind watchmaker.* Longman Scientific and Technical, Harlow.

Dawson, M.R.W., & Schopflocher, D.P. 1992. Modifying the generalized delta rule to train networks of non-monotonic processors for pattern classification. *Connection Science*, **4**, 18–32.

de Garis, H. 1990. Genetic Programming: Modular neural evolution for Darwin machines. *In: Proceedings of IJCNN Washington Jan 1990.*

deAngelis, G.C., Ohzawa, I., & Freeman, R.D. 1991. Depth is encoded in the visual cortex by a specialized receptive-field structure. *Nature*, **352**, 156–159.

DeJong, K.A. 1975. *An analysis of the behavior of a class of genetic adaptive systems.* Ph.D. thesis, University of Michigan, Dissertation Abstracts International 36(10), 5140B.

Dodd, N., Macfarlane, D., & Marland, C. 1991. Optimisation of artificial neural network structure using genetic technique implemented on multiple transputers. *In: Transputing 91.* Geneva: ISO Press.

Duda, R.O., & Hart, P.E. 1973. *Pattern Classification and Scene Analysis.* New York: Wiley.

Dyck, D.N., Lowther, D.A., & McFee, S. 1992. Determining an approximate finite element mesh density using neural network techniques. *IEEE Transactions on Magnetics*, **28**, 1767–1770.

Elman, J.L. 1988. *Finding structure in time.* Tech. rept. 8801. Center for Research in Language, UCSD.

Eshelman, L.J., Caruana, R.A., & Schaffer, J.D. 1989. Biases in the crossover landscape. *Pages 10–19 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Fahlman, S.E. 1989. Fast-Learning Variations on Back-Propagation: An Empirical Study. *Pages 38–51 of:* Touretzky, D., Hinton, G., & Sejnowski, T. (eds), *Proceedings of the 1988 Connectionist Models Summer School.* Pittsburg 1988: Morgan Kaufmann, San Mateo.

Falcon, J.F. 1991. Simulated evolution of modular networks. *Pages 204–211 of:* Prieto, A. (ed), *Artificial Neural Networks, IWANN91, Granada.* Lecture notes in Computer Science 540, Springer Verlag.

Fitzpatrick, J.M., & Grefenstette, J.J. 1988. Genetic algorithms in noisy environments. *Machine Learning*, **3**, 101–120.

Floreano, D. 1992. *Patterns of interactions in ecosystems of neural networks.* M.Phil. thesis, Neural Computation, Dept of Comp Sci., Univ of Stirling.

Fogel, D.B. 1991. An information criterion for optimal neural network selection. *IEEE Transactions on Neural Networks*, **2**, 490–497.

Fogel, D.B., & Atmar, J.W. 1990. Comparing genetic operators with Gaussian mutations in simulated evolutionary processes using linear systems. *Biological Cybernetics*, **63**, 111–114.

Fogel, D.B., Fogel, L.J., & Porto, V.W. 1990. Evolving Neural Networks. *Biological Cybernetics*, **63**, 487–493.

Fogel, L.J.., Owens, A.J., & Walsh, M.J. 1966. *Artificial intelligence through simulated evolution.* Wiley, New York.

Fogelman-Soulie, F. 1991. Neural network architectures and algorithms: a perspective. *Pages 605–615 of:* Kohonen, T., Mäkisara, K., Simula, O., & Kangas, J. (eds), *Artificial Neural Networks*, vol. 1. Amsterdam: North-Holland.

Foster, D.H., & Ward, P.A. 1991. Asymmetries in oriented-line detection indicate two orthogonal filters in early vision. *Proceedings of the Royal Society of London B*, **243**, 75–81.

Foster, P.L. 1992. Directed mutation - between Unicorns and Goats. *Journal of Bacteriology*, **174**, 1711–1716.

Frean, M. 1990. The Upstart Algorithm: A Method for Constructing and Training Feedforward Neural Networks. *Neural Computation*, **2**, 198–209.

Frohn, H., Geiger, H., & Singer, W. 1987. A self-organizing neural network sharing features of the mammalian visual-system. *Biological Cybernetics*, **55**, 333–343.

Fukushima, K. 1980. Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. *Biological Cybernetics*, **36**, 193–202.

Galar, R. 1989. Evolutionary search with soft selection. *Biological Cybernetics*, **60**, 357–364.

Gardner, E., Stroud, N., & Wallace, D.J. 1987. *Training with noise, and the storage of correlated patterns in a neural network model.* Tech. rept. 87/394. Edinburgh, Theoretical Physics.

Gardner, E., Stroud, N., & Wallace, D.J. 1988. Training with noise, application to word and text storage. *Pages 251–260 of:* Eckmiller, R., & von der Malsburg, C. (eds), *Neural Computers.* Berlin: Springer Verlag, for Nato ASI Series F41.

Gerrits, M., & Hogeweg, P. 1991. Redundant coding of an NP-complete problem allows effective genetic algorithm search. *Pages 70–74 of:* Schwefel, H-P., & Männer, R. (eds), *Parallel problem solving from nature.* Lecture notes in Computer Science 496, Springer Verlag.

Ghaloum, S., & Azimisadjadi, M.R. 1991. Terrain classification in sar images using principal components-analysis and neural networks. *Pages 2390–2395 of: Proceedings of IJCNN, Singapore*, vol. 3. IEE, Stevenage, Herts.

Goldberg, D.E. 1987. Simple genetic algorithms and the minimal deceptive problem. *Pages 74–88 of:* Davis, L. (ed), *Genetic Algorithms and Simulated Annealing.* Pitman, London.

Goldberg, D.E. 1989a. Genetic Algorithms and Walsh functions: Part 2, Deception and its analysis. *Complex Systems*, **3**, 129–152.

Goldberg, D.E. 1989b. *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley, Reading, Mass.

Goldberg, D.E. 1989c. Sizing populations for serial and parallel Genetic Algorithms. *Pages 70–79 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Goldberg, D.E. 1990. *Real-coded genetic algorithms, virtual alphabets and blocking.* Tech. rept. IlliGAL 90001. The Illinois Genetic Algorithms Laboratory.

Goldberg, D.E., & Bridges, C.L. 1990. An analysis of a reordering operator on a GA-hard problem. *Biological Cybernetics*, **62**, 397–405.

Goldberg, D.E, & Deb, K. 1991. A comparative analysis of selection schemes used in Genetic Algorithms. *Pages 69–93 of:* Rawlins, G.J.E. (ed), *Foundations of Genetic Algorithms.* Morgan Kaufmann.

Goldberg, D.E., & Rudnick, M. 1991. *Genetic algorithms and the variance of fitness.* Tech. rept. IlliGAL 91001. The Illinois Genetic Algorithms Laboratory.

Goldberg, D.E., & Smith, R.E. 1987. Nonstationary function optimization using Genetic Algorithms with dominance and diploidy. *In:* Grefenstette, J.J. (ed), *Proceedings of the second international conference on Genetic Algorithms.* Lawrence Earlbaum.

Golomb, B.A., Lawrence, D.T., & Sejnowski, T.J. 1991. Sexnet: a neural network identifies sex from human faces. *Pages 572–577 of:* Lippmann, R.P., Moody, J., & Touretzky, D.S. (eds), *Advances in Neural Information Processing Systems 3.* San Mateo: Morgan Kaufmann.

Gorman, R.P., & Sejnowski, T.J. 1988. Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets. *Neural Networks*, **1**, 75–89.

Gray, C.M., Kőnig, P, Engel, A.K., & Singer, W. 1990. Oscillatory responses in cat visual cortex exhibit inter-columnar synchronization which reflects global stimulus properties. *Nature*, **338**, 334–337.

Grefenstette, J.J. 1986. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybenetics*, **SMC16(1)**, 122–128.

Grefenstette, J.J. 1987. *Genesis 4.5.* GA software, from gref@aic.nrl.navy.mil.

Grefenstette, J.J., & Fitzpatrick, J.M. 1985. Genetic search with approximate function evaluations. *Pages 112–120 of:* Grefenstette, J.J. (ed), *Proceedings of an international conference on Genetic Algorithms.* Lawrence Earlbaum.

Grefenstette, J.J., & Schraudolph, N. 1992. *GENESIS 1.4ucsd.* GA software, available by ftp from cs.ucsd.edu (132.239.51.3).

Gruau, F.C. 1992. *Cellular encoding of Genetic Neural Networks.* Tech. rept. 92-21. Laboratoire de l'Informatique du Parallelisme, Ecole Normal Superieure de Lyon.

Guan, B.T., & Gertner, G. 1991. Modelling red pine tree survival with an artificial neural network. *Forest Science*, **5**, 1429–1440.

Guyon, I., Vapnik, V., Boser, B., Bottou, L., & Solla, S.A. 1992. Structural risk minimization for character recognition. *Pages 471–479 of:* Moody, J., Hanson, S.J., & Lippmann, R.P. (eds), *Advances in Neural Information Processing Systems 4.* San Mateo: Morgan Kaufmann.

Hall, B.G. 1991. Adaptive evolution that requires multiple spontaneous mutations - mutations involving base substitutions. *Proceedings of the National Academy of Sciences of the USA*, **88**, 5882–5886.

Hancock, P.J.B. 1989a. Data representation in neural nets: an empirical study. *Pages 11–20 of:* Touretzky, D., Hinton, G., & Sejnowski, T. (eds), *Proceedings of the 1988 Connectionist models summer school.* Morgan Kaufmann.

Hancock, P.J.B. 1989b. Optimising parameters in neural net simulations by genetic algorithm. *In: Mini-symposium on neural network computation.* Rank Prize Funds, Broadway: unpublished.

Hancock, P.J.B. 1990a. GANNET: Design of a neural net for face recognition by Genetic Algorithm. *In: Proceedings of IEEE Workshop on Genetic Algorithms, Neural Networks and Simulated Annealing applied to problems in signal and image processing, Glasgow.*

Hancock, P.J.B. 1990b. *Improving Backprop: hints, parameters and algorithms.* Unpublished technical note, CCCN, Stirling.

Hancock, P.J.B. 1992a. Genetic algorithms and permutation problems: a comparison of recombination operators for neural net structure specification. *In:* Whitley, D. (ed), *Proceedings of COGANN workshop, IJCNN, Baltimore.* IEEE.

Hancock, P.J.B. 1992b. Pruning neural nets by genetic algorithm. *Pages 991–994 of:* Aleksander, I., & Taylor, J.G. (eds), *Proceedings of the International Conference on Artificial Neural Networks, Brighton.* Elsevier.

Hancock, P.J.B. 1992c. Recombination operators for the design of neural nets by genetic algorithm. *Pages 441–450 of:* Männer, R., & Manderick, B. (eds), *Parallel Problem Solving from Nature 2.* Elsevier, North Holland.

Hancock, P.J.B., & Smith, L.S. 1989. Data representation and net structure for object perception. *Pages 395–405 of:* Herault, J. (ed), *Proceedings of Neuro-Nimes.* Nimes: EC2.

Hancock, P.J.B., & Smith, L.S. 1991. GANNET: Genetic design of a neural net for face recognition. *Pages 292–296 of:* Schwefel, H-P., & Männer, R. (eds), *Parallel problem solving from nature.* Lecture notes in Computer Science 496, Springer Verlag.

Hancock, P.J.B., Smith, L.S., & Phillips, W.A. 1991a. A biologically supported error-correcting learning rule. *Pages 531–536 of:* Kohonen, T., Mäkisara, K., Simula, O., & Kangas, J. (eds), *Artificial Neural Networks*, vol. 1. Amsterdam: North-Holland.

Hancock, P.J.B., Smith, L.S., & Phillips, W.A. 1991b. A biologically supported error-correcting rule. *Neural Computation*, **3**, 201–212.

Hancock, P.J.B., Baddeley, R.J., & Smith, L.S. 1992. Principal components of natural images. *Network*, **3**, 61–70.

Harp, S.A., & Samad, T. 1991. Genetic Synthesis of Neural Network Architecture. *Chap. 15, pages 202–221 of:* Davis, L. (ed), *Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

Harp, S.A., Samad, T., & A., Guha. 1989a. *The Genetic synthesis of neural networks*. Tech. rept. TR CSDD-89-I4852-2. Honeywell CSDD.

Harp, S.A., Samad, T., & Guha, A. 1989b. Towards the genetic synthesis of neural networks. *Pages 360–369 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms*. Morgan Kaufmann.

Hassibi, B., & Stork, D.G. 1993. Second order derivatives for network pruning: optimal brain surgeon. *In:* Giles, C.L., Hanson, S.J., & Cowan, J.D. (eds), *Advances in Neural Information Processing Systems 5*. San Mateo, CA: Morgan Kaufmann.

Hebb, D.O. 1949. *The Organization of Behavior*. New York: Wiley. Partially reprinted in (**?**).

Hecht-Nielsen, R. 1987. Counterpropagation Networks. *Applied Optics*, **26**, 4979–4984.

Hecht-Nielsen, R. 1990. On the algebraic structure of feedforward network weight spaces. *Pages 129–135 of:* Eckmiller, R. (ed), *Advanced neural computers*. Elsevier, North Holland.

Hecht-Nielsen, R. 1992. The munificence of high dimensionality. *Pages 1017–1030 of:* Aleksander, I., & Taylor, J.G. (eds), *Artificial Neural Networks 2*, vol. 2. Elsevier.

Heisterman, J. 1990. Learning in neural nets by genetic algorithms. *In:* Eckmiller, R., Hartmann, G., & Hauske, G. (eds), *Parallel Processing in Neural Systems and Computers*. Elsevier (North-Holland).

Herdy, M. 1991. Application of Evolutionstrategie to discrete optimization problems. *Pages 188–192 of:* Schwefel, H-P., & Männer, R. (eds), *Parallel problem solving from nature*. Lecture notes in Computer Science 496, Springer Verlag.

Hőffgen, K-U., Siemon, H.P., & Ultsch, A. 1991. Genetic improvement of feedforward nets for approximating functions. *Pages 302–306 of:* Schwefel, H-P., & Männer, R. (eds), *Parallel problem solving from nature*. Lecture notes in Computer Science 496, Springer Verlag.

Hinton, G.E., & Nowlan, S.J. 1987. How learning can guide evolution. *Complex Systems*, **1**, 495–502.

Hinton, G.E., McClelland, J.L., & Rumelhart, D.E. 1986. Distributed representations. *Pages 77–109 of:* Rumelhart, D.E., & McClelland, J.L. (eds), *Parallel Distributed Processing*, vol. 1, Foundations. Cambridge, Mass: MIT press.

Hirose, Y., Yamashita, K., & Hijiya, S. 1991. Back-propagation algorithm which varies the number of hidden units. *Neural Networks*, **4**, 61–66.

Holland, J.H. 1975. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor.

Hollstien, R.B. 1971. *Artificial genetic adaptation in computer control systems*. Ph.D. thesis, Dept of Computer and Communication Sciences, University of Michigan.

Hubel, D.H., & Wiesel, T.N. 1962. Receptive Fields, Binocular Interaction, and Functional Architecture in the Cat's Visual Cortex. *Journal of Physiology (London)*, **160**, 106–154.

Hyman, S.D., Vogl, T.P., Blackwell, K.T., Barbour, G.S., Irvine, J.M., & Alkon, D.L. 1991. Classification of Japanese kanji using principal component analysis as a preprocessor to an artificial neural network. *Pages 233–238 of: Proceedings of IJCNN-91-Seattle*, vol. 2. IEEE, New York.

Jacobs, R.A. 1988. Increased rate of convergence through learning rate adaptation. *Neural Networks*, **1**, 295–307.

Jefferson, D., Collins, R., Cooper, C., Dyer, M., Flowers, M., Korf, R., Taylor, C., & Wang, A. 1990. *Evolution as a theme in artificial life: the Genesys/Tracker system.* Tech. rept. UCLA-AI-90-09. Computer Science, UCLA.

Ji, C., Snapp, R.R., & Psaltis, D. 1990. Generalizing smoothness constraints from discrete samples. *Neural Computation*, **2**, 188–197.

Jones, A.J. 1993. Genetic algorithms and their applications to the design of neural networks. *Neural computing and applications*, **1**, 32–45.

Jordan, M.I. 1988. *Supervised learning and systems with excess degrees of freedom.* Tech. rept. COINS 88-27. M.I.T.

Kanerva, P. 1990. Contour-map encoding of shape for early vision. *Pages 282–289 of:* Touretzky, D. (ed), *Advances in Neural Information Processing Systems 2.* San Mateo: Morgan Kaufmann.

Keesing, R., & Stork, D.G. 1991. Evolution and learning in neural networks, the number and distribution of learning trials affect the rate of evolution. *Pages 804–810 of:* Lippmann, R.P., Moody, J.E., & Touretzky, D.S (eds), *Advances in Neural Information Processing Systems 3.* Morgan Kaufmann.

Kitano, H. 1990a. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, **4**, 461–476.

Kitano, H. 1990b. Empirical studies on the speed of convergence of neural network training using genetic algorithms. *Pages 789–795 of: Proceedings of the 8th National Conference on Artificial Intelligence (AAAI-90).* MIT Press, Cambridge.

Kolen, J.F., & Pollack, J.B. 1990. Scenes from exclusive-or - back propagation is sensitive to initial conditions. *Pages 868–875 of: Program of the twelfth annual conference of the cognitive science society.*

Koza, J.R. 1990. *Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems.* Tech. rept. STAN-CS-90-1314. Stanford.

Kruschke, J.K. 1989. Creating local and distributed bottlenecks in hidden layers of backpropagation networks. *Pages 120–126 of:* Touretzky, D., Hinton, G., & Sejnowski, T. (eds), *Proceedings of the 1988 Connectionist models summer school.* Morgan Kaufmann.

Le Cun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., & Jackel, L.D. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, **1**, 541–551.

Le Cun, Y., Denker, J.S., & Solla, S.A. 1990. Optimal Brain Damage. *Pages 598–605 of:* Touretzky, D.S. (ed), *Advances in Neural Information Processing Systems*, vol. 2. Denver 1989: Morgan Kaufmann, San Mateo.

Linsker, R. 1986. From Basic Network Principles to Neural Architecture. *Proceedings of the National Academy of Sciences, USA*, **83**, 7508–7512, 8390–8394, 8779–8783.

Linsker, R. 1988. Self-Organization in a Perceptual Network. *Computer*, March, 105–117.

Lorquet, V., Puget, P., Guillemaud, R., & Niez, J-J. 1991. Improving half distributed coding methods: a filtering process approach applied to data representation in neural nets. *Pages 931–936 of:* Kohonen, T., Măkisara, K., Simula, O., & Kangas, J. (eds), *Artificial Neural Networks*, vol. 1. Amsterdam: North-Holland.

Lucasius, C.B., & Kateman, G. 1989. Application of genetic algorithm in chemometrics. *Pages 170–176 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms*. Morgan Kaufmann.

MacKay, D.J.C. 1991. *Bayesian methods for adaptive models*. Ph.D. thesis, California Institute of Technology.

MacKay, D.J.C. 1992. A practical Bayesian framework for backpropagation networks. *Neural Computation*, **4**, 448–472.

MacKay, D.J.C., & Miller, K.D. 1990a. Analysis of Linsker's Simulation of Hebbian Rules. *Neural Computation*, **2**, 173–187.

MacKay, D.J.C., & Miller, K.D. 1990b. Analysis of Linsker's simulations of Hebbian rules to linear networks. *Network*, **1**, 257–297.

Mackey, M.C., & Glass, L. 1977. Oscillation and Chaos in Physiological Control Systems. *Science*, **197**, 287.

Marchand, M., Golea, M., & Rubján, P. 1990. A Convergence Theorem for Sequential Learning in Two-Layer Perceptrons. *Europhysics Letters*, **11**, 487–492.

Matsuoka, K. 1992. Noise injection into inputs in back-propagation learning. *IEEE Transactions on systems, man and cybernetics*, **22**, 436–440.

McWalters, G. 1991. *Two techniques of image preprocessing by principal component analysis in neural networks*. M.Phil. thesis, Information Technology, Dept of Comp Sci., Univ of Stirling.

Menczer, F. 1991. *Sexual reproduction in the genetic learning of a neural network simulating artificial life*. Submitted to Biological Cybernetics.

Menczer, F., & Parisi, D. 1990. *'Sexual' reproduction in neural networks*. Tech. rept. PCIA-90-06. C.N.R.Rome.

Mézard, M., & Nadal, J.-P. 1989. Learning in Feedforward Layered Networks: The Tiling Algorithm. *Journal of Physics A*, **22**, 2191–2204.

Mühlenbein, H. 1989. Parallel Genetic Algorithms, population genetics and combinatorial optimization. *Pages 416–421 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms*. Morgan Kaufmann.

Mühlenbein, H., & Kinderman, J. 1989. The dynamics of evolution and learning - towards genetic neural networks. *Pages 173–197 of:* Pfeifer, R., Schreter, Z., Fogelman-Soulie, F., & Steels, L. (eds), *Connectionism in perspective*. Elsevier Sceince (North Holland).

Miller, G.F., Todd, P.M., & Hegde, S.U. 1989. Designing neural networks using Genetic Algorithms. *Pages 379–384 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms*. Morgan Kaufmann.

Mittler, J.E., & Lenski, R.E. 1992. Experimental-evidence for an alternative to directed mutation in the bgl operon. *Nature*, **356**, 446–448.

Montana, D.J., & Davis, L. 1989. Training feedforward neural networks using Genetic Algorithms. *Pages 762–767 of: Proceedings of the Eleventh IJCAI*.

Moody, J. 1992. The effective number of parameters: an analysis of generalization and regularization in nonlinear learning systems. *Pages 847–854 of:* Moody, J., Hanson, S.J., & Lippmann, R.P. (eds), *Advances in Neural Information Processing Systems 4*. San Mateo: Morgan Kaufmann.

Moody, J., & Darken, C. 1989. Fast Learning in Networks of Locally-Tuned Processing Units. *Neural Computation*, **1**, 281–294.

Murray, A.F. 1992. Multilayer perceptron learning optimized for on-chip implementation: a noise-robust system. *Neural Computation*, **4**, 366–381.

Nolfi, S., & Parisi, D. 1992. *Growing neural networks*. Unpublished report, Institute of Psychology, CNR, Rome.

Nolfi, S., Parisi, D., Vallar, G., & Burani, C. 1990. Recall of sequences of items by a neural network. *In:* Touretzky, D., Hinton, G., & Sejnowski, T. (eds), *Proceedings of the 1990 Connectionist models summer school*. Morgan Kaufmann.

Nowlan, S.J., & Hinton, G.E. 1991. *Simplifying neural networks by soft weight-sharing*. Tech. rept. Toronto, Computer Science.

Oja, E. 1982. A Simplified Neuron Model As a Principal Component Analyzer. *Journal of Mathematical Biology*, **15**, 267–273.

Oja, E. 1989. Neural Networks, Principal Components, and Subspaces. *International Journal of Neural Systems*, **1**, 61–68.

Optican, L.M., & Richmond, B.J. 1987. Temporal encoding of two dimensional patterns by single units in primate inferior temporal cortex III: Information theoretic analysis. *Journal of Neurophsiology*, **57**, 162–178.

Oram, M.W., & Perrett, D.I. 1992. Time course of neural responses discriminating different views of the face and head. *Journal of neurophysiology*, **68**, 70–84.

Parisi, D., Cecconi, F., & Nolfi, S. 1990. Econets: neural networks that learn in an environment. *Network*, **1**, 149–168.

Parisi, D., Nolfi, S., & Cecconi, F. 1991. *Learning, behaviour and evolution.* Tech. rept. PCIA-91-14. C.N.R. Rome. Also in Proceedings of the first european conference on artificial life ECAL 91, pp.207-216, Varela,F.J. and Bourgine,P. (Eds).

Plaut, D.C., Nowlan, S.J., & Hinton, G.E. 1986. *Experiments on learning by back propagation.* Tech. rept. CMU-CS-86-126. Carnegie Mellon University.

Plumbley, M.D., & Fallside, F. 1989. An information-theoretic approach to unsupervised connectionist models. *Pages 239–245 of:* Touretzky, D., Hinton, G., & Sejnowski, T. (eds), *Proceedings of the 1988 Connectionist models summer school.* Morgan Kaufmann.

Pomerleau, D.A. 1991. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, **3**, 88–97.

Radcliffe, N. 1990a. *Equivalence class analysis of genetic algorithms.* Submitted to Complex Systems.

Radcliffe, N. 1990b. *Genetic neural networks on MIMD computers.* Ph.D. thesis, Edinburgh.

Radcliffe, N. 1991. Forma Analysis and random respectful recombination. *Pages 222–229 of:* Belew, R.K., & Booker, LB. (eds), *Proceedings of the fourth international conference on Genetic Algorithms.* Morgan Kaufmann.

Radcliffe, N. 1993. Genetic set recombination and its application to neural network topology optimisation. *Neural computing and applications*, **1**, 67–90.

Rauschecker, J.P., & Singer, W. 1979. Changes in the circuitry of the kitten's visual cortex are gated by post-synaptic activity. *Nature*, **280**, 58–60.

Rawlins, G.J.E. (ed). 1991. *Foundations of Genetic Algorithms.* Morgan Kaufmann.

Ray, T.S. 1992. An approach to the synthesis of life. *Pages 371–408 of:* Langton, C.G., Taylor, C., Farmer, J.D., & Rasmussen, S. (eds), *Artificial life II.* Addison-Wesley.

Rechenberg, I. 1973. *Evolutionstrategie: Optimierung technische Systeme nach Prinzipien der biologischen Evolution.* Friedrich Frommann Verlag, Stuttgart.

Rechenberg, I. 1989. Evolution strategy, nature's way of optimization. *In:* Bergman, H.W. (ed), *Optimization: methods and applications, possibilities and limitations.* Lecture notes in Engineering 47, Springer Verlag.

Rescorla, R.A., & Wagner, A.R. 1972. A Theory of Pavlovian Conditioning: The Effectiveness of Reinforcement and Nonreinforcement. *Pages 64–69 of:* Black, A.H., & Prokasy, W.F. (eds), *Classical Conditioning II: Current Research and Theory.* New York: Appleton-Century-Crofts.

Richardson, J.T., Palmer, M.R., Liepens, G., & Hilliard, M. 1989. Some guidelines for Genetic Algorithms with penalty functions. *Pages 191–197 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Richmond, B.J., Optican, L.M., & Spitzer, H. 1990. Temporal encoding of two dimensional patterns by single units in primate visual cortex: I quantification of response waveform. *Journal of Neurophsiology*, **64**, 351–369.

Robbins, G.E., Plumbley, M.D., Hughes, J.C., Fallside, F., & Prager, R. 1993. Generation and adaptation of neural networks by evolutionary techniques (GANNET). *Neural computing and applications*, **1**, 23–31.

Roberts, M.J. 1992. *Symmetry properties of feed-forward neural networks.* Tech. rept. CCCN-9. Dept of Computing Science and Mathemetics, University of Stirling.

Robertson, G.J.S, & Sharman, K.C. 1990. POLGA - Persistence of life Genetic Algorithm on Transputer surfaces. *In:* Sharman, K. (ed), *GAs, NNs and SA applied to problems in signal and image processing.* IEEE.

Ross, H.E. 1990. Evironmental influences on geometrical illusions. *Pages 216–221 of:* Muller, F. (ed), *Frechner Day 90: Proceeding of the 6th annual meeting of the international society of psychophysicists.*

Rubner, J., & Schulten, K. 1990. Development of Feature Detectors by Self-Organization. *Biological Cybernetics*, **62**, 193–199.

Rumelhart, D.E. 1988. Talk given at Connectionist Models Summer School.

Rumelhart, D.E., Hinton, G.E., & Williams, R.J. 1986. Learning Representations by Back-Propagating Errors. *Nature*, **323**, 533–536. Reprinted in (**?**).

Sanger, T.D. 1989. Optimal Unsupervised Learning in a Single-Layer Linear Feedforward Neural Network. *Neural Networks*, **2**, 459–473.

Saund, E. 1986. Abstraction and representation of continuous variables in connectionist networks. *Pages 638–644 of: Proc AAAI-86*, vol. 1. American Association for Artificial Intelligence.

Schiffmann, W., & Mecklenburg, K. 1990. Genetic generation of backpropagation trained neural nets. *Pages 205–208 of:* Eckmiller, R., Hartmann, G., & Hauske, G. (eds), *Parallel Processing in Neural Systems and Computers.* Elsevier (North-Holland).

Schiffmann, W., Joost, M., & Werner, R. 1991. Performance evaluation of evolutionarily created neural network topologies. *In:* Schwefel, H-P., & Männer, R. (eds), *Parallel problem solving from nature.* Lecture notes in Computer Science 496, Springer Verlag.

Schraudolph, N.N., & Belew, R.K. 1990. *Dynamic parameter encoding for genetic algorithms.* Tech. rept. CSE TR 90-175. UCSD. Accepted for publication in Machine Learning.

Schwefel, H-P. 1975. *Evolutionstrategie und numerische Optimierung.* Ph.D. thesis, Technische Universität Berlin. Translated into English: Schwefel (1981).

Schwefel, H-P. 1981. *Numerical optimization of computer models.* Wiley, Chichester. Extended translation of Schwefel (1975).

Schwefel, H-P., & Männer, R. (eds). 1991. *Parallel Problem Solving from Nature.* Lecture notes in Computer Science 496, Springer Verlag.

Searle, J. 1980. Minds, Brains and Programs. *Behavioural and Brain Sciences*, **3**, 417–457.

Sejnowski, T.J. 1988. Talk given at Connectionist Models Summer School.

Shaefer, C.G. 1987. The ARGOT strategy: adaptive representation genetic optimizer technique. *In:* Grefenstette, J.J. (ed), *Proceedings of the second international conference on Genetic Algorithms.* Lawrence Earlbaum.

Spofford, J.J., & Hintz, K.J. 1991. Evolving sequential machines in amorphous neural networks. *Pages 973–978 of:* Kohonen, T., Mäkisara, K., Simula, O., & Kangas, J. (eds), *Artificial Neural Networks*, vol. 1. Amsterdam: North-Holland.

Stent, G.S. 1973. A physiological mechanism for Hebb's postulate of learning. *Proceedings of the National Academy of Sciences, USA*, **70**, 997–1001.

Stork, D. 1992. Non-optimality in neurobiological systems. *In: Optimality conference, University of Texas, Dallas.*

Sutton, R.S., & Barto, A.G. 1981. Toward a modern theory of adaptive networks: expectation and prediction. *Psychological Review*, **88**, 135–170.

Syswerda, G. 1989. Uniform crossover in Genetic Algorithms. *Pages 2–9 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Takeda, M., & Goodman, J.W. 1986. Neural networks for computation: number representations and programming complexity. *Applied Optics*, **25**, 3033–3046.

Tanese, R. 1987. Parallel Genetic Algorithm for a hypercube. *In:* Grefenstette, J.J. (ed), *Proceedings of the second international conference on Genetic Algorithms.* Lawrence Earlbaum.

Tanese, R. 1989. Distributed Genetic Algorithms. *Pages 434–439 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Todd, P. 1988. *Evolutionary methods for connectionist architectures.* unpublished internal report, Stanford.

Tollenare, T. 1990. SuperSAB: fast adaptive back propagation with good scaling properties. *Neural Networks*, **3**, 561–573.

Torreele, J. 1991. Temporal processing with recurrent networks: an evolutionary approach. *Pages 555–561 of:* Belew, R.K., & Booker, LB. (eds), *Proceedings of the fourth international conference on Genetic Algorithms.* Morgan Kaufmann.

Vico, F.J., & Sandoval, F. 1991. Use of genetic algorithms in neural networks definition. *Pages 196–203 of:* Prieto, A. (ed), *Artificial Neural Networks, IWANN91, Granada.* Lecture notes in Computer Science 540, Springer Verlag.

Vidyasagar, T.R., & Henry, G.H. 1990. Relationship between preferred orientation and ordinal position in neurones of cat striate cortex. *Visual Neuroscience*, **5**, 565–569.

von der Malsburg, C. 1985. Nervous structures with dynamical links. *Ber. Bunsenges Phys. Chem.*, **89**, 703–710.

von der Malsburg, Ch. 1973. Self-Organization of Orientation Sensitive Cells in the Striate Cortex. *Kybernetik*, **14**. Reprinted in (**?**).

Vose, M. 1990. Formalizing Genetic Algorithms. *In: Proceedings of IEEE Workshop on Genetic Algorithms, Neural Networks and Simulated Annealing applied to problems in signal and image processing, Glasgow.*

Vose, M., & Liepins, G.E. 1991. Schema disruption. *Pages 237–242 of:* Belew, R.K., & Booker, L.B. (eds), *Proceedings of the Fourth International Conference on Genetic Algorithms.* Morgan Kaufmann.

Vrckovnik, G., Chung, T., & Carter, C.R. 1990. Classifying impulse radar wave-forms using principal components-analysis and neural networks. *Pages 69–74 of: Proceedings of IJCNN, San Diego*, vol. 3. IEEE, New York.

Waibel, A. 1989. Modular Construction of Time-Delay Neural Networks for Speech Recognition. *Neural Computation*, **1**, 39–46.

Watt, R.J., & Morgan, M.J. 1985. A theory of the primitive spatial code in human vision. *Vision Research*, **25**, 1661–1674.

Weigend, A.S., Rumelhart, D.E., & Huberman, B.A. 1991. Generalization by weight-elimination applied to currency exchange-rate prediction. *Pages 837–841 of: IJCNN-91-SEATTLE*, vol. 2. IEEE, New York.

Weinshall. 1990. A self-organising multiple-view representation of 3D objects. *Pages 274–281 of:* Touretzky, D. (ed), *Advances in Neural Information Processing Systems 2.* San Mateo: Morgan Kaufmann.

Whitley, D. 1989. The Genitor algorithm and selection pressure: why rank-based allocation of trials is best. *Pages 116–121 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Whitley, D. 1991. Fundamental principles of deception in genetic search. *Pages 221–241 of:* Rawlins, G.J.E. (ed), *Foundations of Genetic Algorithms.* Morgan Kaufmann.

Whitley, D., & Hanson, T. 1989. Optimizing neural networks using faster, more accurate genetic search. *Pages 391–396 of:* Schaffer, J.D. (ed), *Proceedings of the third international conference on Genetic Algorithms.* Morgan Kaufmann.

Whitley, D., & Knuth, J. 1988. GENITOR: a different genetic algorithm. *Pages 118–130 of: Proceedings of the Rocky Mountain Conference on Artificial Intelligence.* Denver Colorado.

Whitley, D., & Starkweather, T. 1990. Genitor II: A distributed genetic algorithm. *Journal of Experimental and Theoretical Artificial Intelligence (preprint).*

Whitley, D., Starkweather, T, & Bogart, C. 1990. Genetic Algorithms and Neural Networks: optimizing connections and connectivity. *Parallel Computing*, **14-3**, 347–361.

Whitley, D., Dominic, S., & Das, R. 1991. Genetic reinforcement learning with multilayer neural networks. *Pages 562–569 of:* Belew, R.K., & Booker, LB. (eds), *Proceedings of the fourth international conference on Genetic Algorithms.* Morgan Kaufmann.

Widrow, B., & Hoff, M.E. 1960. Adaptive Switching Circuits. *Pages 96–104 of: 1960 IRE WESCON Convention Record*, vol. 4. New York: IRE.

Wieland, A.P. 1990. Evolving controls for unstable systems. *Pages 91–102 of:* Touretzky, D.S., Elman, J.L., Sejnowski, T.J., & Hinton, G.E. (eds), *Proceedings of the 1990 Connectionist Models Summer School.* Morgan Kaufman.

Wright, A.H. 1991. Genetic algorithms for real parameter optimization. *Pages 205–218 of:* Rawlins, G.J.E. (ed), *Foundations of Genetic Algorithms.* Morgan Kaufmann.

Xu, B., & Zheng, L.Q. 1991. PPNN: A faster learning and better generalizing neural net. *Pages 893–898 of: Proceedings of IJCNN, Singapore.* IEE, Stevenage, Herts.

Yao, X. 1993. A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems.*

Zemel, R.S., Mozer, M.C., & Hinton, G.E. 1990. TRAFFIC: recognizing objects using hierachical reference frame transformations. *Pages 266–273 of:* Touretzky, D. (ed), *Advances in Neural Information Processing Systems 2.* San Mateo: Morgan Kaufmann.