ON THE SIGNIFICANCE OF THE PERMUTATION PROBLEM IN NEUROEVOLUTION

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2010

By
Stefan Magnuson
School of Computer Science

Contents

Word Count: 42,000

List of Tables

List of Figures

Abstract

For as long as Evolutionary Algorithms have been applied to the optimisation of Neural Networks, the severity of the Permutation Problem has been debated without consensus. Considered by many to be a serious obstacle to be worked around when designing Neuroevolutionary algorithms, various proposals for solving the problem have been presented over the past two decades. The opposing view that the problem is not severe in practise has also been presented in the earliest and most recent research.

The aim of this thesis is to bring clarity to the issues surrounding the Permutation Problem in the literature. We first survey the presentation of the problem in the literature with the aim of producing a single definition and set of naming conventions by which to discuss the problem. Then, we present the exact probabilities for the likelihood of encountering permutations in the initial population. We then estimate the likely rate of occurrence of the Permutation Problem for the case where not all neurons can be considered unique.

Empirical support for the theoretical results is then given in the form of experiments to test explicitly the rate of occurrence and relative severity of the Permutation Problem in practise. As part of this investigation we also explore the role of crossover in Neuroevolution.

The penultimate chapter uses what we have learned about the nature of permutations in general to present a novel framework for constructing permutation-free search algorithms. The benefits here are not specific to the building of Neuroevolutionary algorithms, rather the presented framework appears to be applicable to any combinatorial optimisation problem.

We conclude by reviewing what we have learned about the Permutation Problem and its severity in practise in Neuroevolution. New directions for this work are discussed, particularly in relation to the application of the presented permutation-free search framework.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://www.campus.manchester.ac.uk/medialibrary/policies/intellectual-property.pdf), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University's policy on presentation of Theses.

Acknowledgements

First and foremost I would like to thank my parents for their love and support, and for providing the best possible working environment I could have hoped for.

In many ways I feel I should now thank my grandfather Árni Sörensson for sending us a computer many years ago, at a time when they were extremely rare. Receiving this mysterious machine of such tangible potential is what sparked my interest in computing. In that vein I am also very grateful to the Talbots for their encouragement in those early years.

Particular thanks go out to Gavin Brown and my good friend Arjun Chandra for their repeated encouragement, advice and for showing an interest in my work. Thanks to you I have a more positive understanding of what research is.

Many thanks to my brothers, to Andri for helping me avoid burning out and to Almar for involving me in his projects. I'm very lucky to have you both looking out for me.

Warm thanks to all the great people who I have met along the way and have helped keep me sane during this process, particularly Martin Irwin and Mariko Hayashida for making Manchester more fun and to Sayaka Nishimura and Chika Miyata for enduring the final months with me. Thanks also to Adam, Richard and Nicolò for making the office more fun!

I am grateful to my supervisor for firstly encouraging me to do this but also for his complete support throughout and for his generosity with his time.

Finally, many thanks to Henry Bottomley, Tom Froese and Nick Radcliffe for taking the time to correspond with me on some of the topics in this thesis.

Chapter 1

Introduction

An idea which has survived largely unchallenged in the field of Neuroevolution is that there exists a fundamental difficulty when attempting to apply Evolutionary Algorithms to the construction and optimisation of Neural Networks. This idea, which has been named variously as the *Permutation* or *Competing Conventions* Problem, has often been demonstrated using intuitive arguments (effectively thought experiments). The aim of this thesis is therefore to explore the implications of this idea from both theoretical and empirical perspectives.

In this chapter we will introduce the problem in more detail and motivate the need for its investigation. We then state the thesis questions and give a chapter-by-chapter synopsis of the thesis.

1.1 Evolved Artificial Neural Networks

Artificial Neural Networks (ANN) are computational devices which are loosely modelled on the animal brain. These networks are capable of performing tasks which might not otherwise be well suited to the more traditional paradigm of programmed software. Examples of such tasks include various kinds of classification [YL97, Zha00, YI08], system control [GM99, GBM01, GM03b, GM03a, GSM06], forecasting/prediction [KAYT90, Zha01, ET05] and game playing [SM04, ATL07, Mat07, Cro08].

Evolutionary Computation (EC) is the field of problem solving using ideas and mechanics taken from the science of natural evolution. The combination of these two fields gives us Neuroevolution, which is the application of Evolutionary Algorithms (EA) to the task of building and optimising ANN.

1.2 The Permutation Problem

The field of Neuroevolution embodies the application of Evolutionary Algorithms to the task of Neural Network construction and optimisation. The Permutation Problem is a difficulty which arises when designing such algorithms. Generally

speaking this difficulty arises in any problem where the search algorithm imposes an ordering on a solution while the solution space is order independent, meaning that each possible ordering of a given solution is equivalent.

This kind of order independence causes a number of problems, particularly for certain classes of Evolutionary Algorithm. This thesis investigates these problems and attempts to quantify their severity in practise.

1.3 Why study Neuroevolution?

The last decade has seen a significant growth in research in the application of Evolutionary Algorithms to the optimisation of neural networks, particularly for problems where traditional reinforcement learning often fails to produce a suitable solution (for a summary of such applications see [GSM06]).

A Neural Network, while biologically inspired, is essentially a hierarchy of functions arranged using weighted connections. When arranged correctly, the network is capable of performing useful computations while still remaining a simple model. The properties of neural networks are presented in more detail in section 2.1; for now it is sufficient to consider neural networks as a compound of functions which accepts and transforms a fixed number of inputs into a fixed number of outputs, which are then interpreted accordingly. For example, if we wish our network to map between some tissue sample readings (the inputs) and labels such as 'malignant' or 'benign' (the outputs) then we might assign one input for each tissue measurement (e.g. average cell radius, cell texture etc.) and one output for each label. The mapping between those two interfaces is the area of interest here, i.e. how to best arrange a collection of neurons such that passing in data for new patients produces a usefully accurate diagnosis.

Designing a neural network by hand is a tedious process which is intractable for all but the simplest of problems (e.g. modelling Boolean logic gates). A neural network consists of a number of *neurons* (essentially functions) and *synapses* (weighted connections). The *architecture* of a network refers to the arrangement and connectivity of these neurons. In the simpler case the architecture is fixed and it is the weights[1] which are adapted in order to solve the given problem.

Neuroevolutionary algorithms are capable of producing networks which can approximate complex, noisy nonlinear functions such as those of stock markets, allowing for limited prediction of future market behaviour (function regression). Another common utilisation of Neural Networks is for classification, where given some data we wish to predict which of a finite number of classes the instance belongs to. For example with our previous medical scan data the possible classes might be 'malignant' or 'benign'. More generally, it has been shown theoretically that neural networks can approximate any function to any degree of accuracy

[1]The terms *weight* and *synapse* will be used interchangeably throughout this thesis, though 'weight' will be favoured. The term *synapse* reflects the biological inspiration for the structure, whereas *weight* reflects its purpose.

required under certain constraints [LLPS93, HN89]. The problem then is how to set or discover the architecture and corresponding optimal weights.

Neuroevolution offers the potential to improve upon traditional uses of neural networks by:

- Allowing the network structure to fit the problem, rather than fitting the problem to a fixed structure [ASP94].
- Overcoming the limitations of traditional learning algorithms in recurrent neural networks [Hoc98].
- Producing solutions to tasks which prove overly difficult for Reinforcement Learning algorithms in general [Sta04, GSM06, GSM08].

As this work is primarily concerned with the nature of and issues surrounding the Permutation Problem, we will focus on the class of feedforward Neural Networks with fixed architecture, composed of a single fully-connected layer of hidden neurons only.

1.4 What is this Thesis about?

In this section we give an overview of the thesis in terms of the questions it aims to answer, the motivation behind the work and finally its contributions to the field.

1.4.1 Thesis Questions

The main question this thesis aims to answer is, *"Is the Permutation Problem Serious in Practise?"*[2]. As 'Permutation Problem' does not have a single, agreed upon definition, we begin by presenting the various definitions in the literature and then clearly state our own interpretation. We note that there are somewhat conflicting or incompatible ideas present in the literature regarding what the Permutation Problem is, and how serious a concern it is. We first examine the various ways in which it has been presented, with the aim of producing a taxonomy of variants of the Permutation Problem such that all work can be related to a single set of definitions and thus be comparable.

With the various interpretations of the Permutation Problem formalised it is then possible to begin answering the question of its severity in practise. This is an important question to ask because of the lack of consensus in the literature regarding to what degree the Permutation Problem affects the performance of Neuroevolutionary algorithms. The severity of the Permutation Problem has been used as justification for the poor performance of recombination (crossover)

[2]By 'serious' we mean, "is the problem a serious consideration when working with Neuroevolutionary Algorithms?".

operators, thus causing their exclusion from Neuroevolutionary algorithms in some cases [ASP94, Yao99]. By quantifying this severity we can determine if the performance of crossover (poor or otherwise) is due to the Permutation Problem or to other factors.

Although not the focus of this work, a closely related issue is that of the role of recombination in Neuroevolutionary algorithms. The Permutation Problem has often been used to justify the exclusion of crossover; we examine the role and utility of crossover in general, in light of what we have learned about the Permutation Problem. In answering the question, *"Is our understanding of how recombination works in the context of Neural Networks sufficient?"*, we aim to demonstrate why and in what cases crossover need not be excluded outright when evolving Neural Networks.

Finally we ask, *"Can our knowledge regarding the Permutation Problem be used to improve Neuroevolutionary search methods?"*. The optimisation of Neural Networks poses a somewhat unique problem to Evolutionary Algorithm practitioners. We investigate whether we can use what we have learned about the nature of the Permutation Problem to improve our algorithms. The result of this is the Multiset Search Framework, a novel representation suitable for evolutionary search that contains only multisets, and so in this case only unique networks.

1.4.2 Motivation

There exist in the literature multiple aliases for what we call the *Permutation Problem*. It would appear that these aliases on the whole emerged independently, around the time when this problem was first discussed and received the most attention.

Whenever we have one concept with many labels it is desirable to reduce the number of naming conventions to one. If we have just one name for a concept or problem it becomes easier to discuss and search for in the literature. With multiple naming conventions we run the risk of spending extra time to affirm with one another that one term is in fact synonymous with another. Conversely if two terms are considered synonymous when they in fact refer to (possibly only slightly) different concepts, then we run the risk of wasting considerably more time discussing two different concepts while believing that we are in fact 'on the same page'.

A final confounding factor when discussing the Permutation Problem comes from intuitive definitions which refer fundamentally to the same idea but with slight differences which may only be alluded to implicitly. This lack of concreteness regarding each definition results in a (possibly unconscious) judgement call being necessary on the part of the reader. Different readers may come to different conclusions regarding the *intention* of a particular author when reading a definition. Thus for some given definitions of the problem we have room for interpretation; this is especially the case with a problem such as the Permutation Problem due to its intuitively simple nature; as it may be described as 'obvious' once known, it

can be tempting to assume that it is a well understood problem. An objective of this work is therefore to reduce confusion regarding the nature of the Permutation Problem by clearly defining each alias present in the literature, presenting a clear definition and suggesting naming conventions for future discussion.

The major motivation for this work is the lack of consensus in the literature regarding the severity of the Permutation Problem. As we will see in section 3.3, early work was divided: its perceived severity ranged from being virtually insurmountable [BMS90] to largely insignificant in practise [Han93]. There are many examples in the literature of *in what form* the problem may manifest. The problem is that these examples are intuitive in nature. Each example can be confirmed to be *possible* but no exact value or estimate is given for *how often* the problem is expected to occur. This work is therefore motivated by the need to make a clear distinction between how serious the problem is *when it occurs* and *how often* it actually does occur.

In this work we therefore aim to provide evidence, both in the form of exact probabilities and empirical estimates, of how often we can expect the problem to occur in a practical setting. With this information we may be able to improve upon current algorithms through a deeper understanding of the conditions under which the problem occurs. Such an attempt from the perspective of redundancy elimination is presented in Chapter 6.

1.4.3 Contributions of the Thesis

The main contribution of this thesis is an analysis of the Permutation Problem from both theoretical and empirical standpoints. We also present a novel Neuroevolutionary search algorithm and problem representation that eliminates the Permutation Problem. In this section we summarise these contributions and note where they appear in the thesis.

Contributions to the Understanding of the Permutation Problem

- The first review of the literature on the Permutation Problem to examine its multiple definitions and aliases in detail, with a focus on creating a taxonomy for the different views and interpretations of the problem (Chapter 3).

- The explicit distinction between exact-weight permutations and those which simply involve similar or effectively-equivalent neurons (Chapter 3).

- The first set of exact probabilities for the likelihood that the Permutation Problem will occur, in its various forms, in the initial population (Chapter 4).

- Common beliefs regarding the nature of the Permutation Problem are explored probabilistically, highlighting areas where intuition alone can fail to accurately estimate the likelihood of an event occurring (Section 4.5).

- A fast method based on the theory of integer partitions for calculating the probability of the Permutation Problem occurring in realistic network spaces (Section 4.3.2).
- First explicit permutation counting experiments, where the existence of permutations in the population is checked for to produce empirical estimates of the rate of occurrence of the problem (Chapter 5).

Contributions in a Wider Context

- A novel framework, Multiset Search, is presented which can be used to reduce a search space down to only the points which represent its unique networks in such a way that existing search algorithms can be applied with only minor modifications. This framework is used in the development of a novel Neuroevolutionary algorithm, Multiset-based Network Search (MBNS) which removes the redundant networks (due to the Permutation Problem) from the search space entirely. This framework can be applied with minor modifications to any problem with a similar type of redundancy (Chapter 6).
- The method for calculating the probability of the Permutation Problem is equivalent to the probability of drawing two strings which are permutations of each other from a space of strings defined by a fixed alphabet and string length. As this is by no means restricted to networks of neurons, the equation and provided implementation can be used to calculate this probability for other problems with these characteristics (Chapter 4).
- A discussion of the use and limitations of Price's Equation [Pri70] in profiling an Evolutionary Algorithm. We highlight the gains in insight gained through the use of so-called 'Price Plots' [BPJ04] but also attend to their limitations. Future work is suggested which could make them a more useful tool for the profiling of Evolutionary Algorithms (Chapter 5).

Overall, this thesis aims to provide practitioners with the background and tools necessary to evaluate the severity of the Permutation Problem for their own network representations and problems, and factor this likelihood into their Evolutionary Algorithm design accordingly. A further contribution is the analysis of the Permutation Problem in practical settings which confirms the main result of past empirical investigations, i.e. that the Permutation Problem is not serious in practise due to its low rate of occurrence. We further contend and provide evidence for the conclusion that the Permutation Problem is also not serious from a theoretical standpoint. While the Permutation Problem does not appear to occur often in practise, the search space nevertheless contains a high level of redundancy. The presented Multiset-based framework then eliminates this redundancy and in doing so eliminates the possibility of the Permutation Problem occurring.

1.4.4 Structure of the Thesis

In Chapter 2 we briefly cover the background of the fields of Neural Networks, Evolutionary Algorithms and their combination, Neuroevolution. This chapter also introduces the test problems that will be used throughout this work.

In Chapter 3 we begin by defining the various kinds of permutation that will be investigated in this work. The origin of the Permutation Problem in the literature is then investigated, with an emphasis on tracing the evolution of the idea of the Permutation Problem itself over time and identifying the wider area of literature to which this thesis contributes.

In Chapter 4 we present the calculation of the exact probability that the (exact weight variant) Permutation Problem occurs in the initial population. An alternative, fast method for calculation of this probability is presented, allowing the calculation of this probability for some real Neuroevolutionary problem settings. Then, in order to estimate the likelihood of the Permutation Problem when we no longer assume the uniqueness of neurons, we estimate the change in probability as the search space becomes increasingly redundant.

In Chapter 5 we analyse empirically the occurrence of (exact weight) permutations and present an analysis of the efficacy of crossover in Neuroevolutionary algorithms.

In Chapter 6 we present a framework for constructing permutation-free search algorithms. Using what we have learned about the nature of permutations we define a search space representation and set of search operators which allow for a one-to-one mapping between points in the search space and unique arrangements of neurons, both into sets and multisets.

Chapter 7 concludes this work with a summary of what we have learned about the Permutation Problem and its significance to Neuroevolutionary algorithms, ending with practical advice for practitioners. Finally, some areas of interest for future work are outlined.

1.4.5 Publications Resulting from the Thesis

[HN08] Stefan Haflidason and Richard Neville
"A Case for Crossover in Neuroevolution", In Proceedings of the Workshop and Summer School on Evolutionary Computing, Lecture Series by Pioneers (WSSEC), 2008.

[HN09a] Stefan Haflidason and Richard Neville
"On the Significance of the Permutation Problem in Neuroevolution", In Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation (GECCO), pages 787-794, 2009.

[HN09b] Stefan Haflidason and Richard Neville
"Quantifying the Severity of the Permutation Problem in Neuroevolution", In

Proceedings of the Fourth International Workshop on Natural Computing (IWNC), pages 149-156, 2009.

Chapter 2

The Evolution of Neural Networks

In this chapter we present the necessary background of Artificial Neural Networks (ANNs) and Evolutionary Algorithms (EAs) followed by their combination, Neuroevolution (NE). We conclude by giving an overview of the type of problems considered in this work.

2.1 Neural Networks

The term *Neural Network* naturally evokes the image of something complex and biological with some level of intelligent ability, such as the animal brain. An Artificial Neural Network, while certainly capable, does not however posses what we commonly understand as being intelligence. The human brain is estimated to contain around 10^{11} or 100 billion neurons, with each having some thousands of connections to others [Gur97]. In contrast, an artificial network may serve some practical purpose with as few as five artificial neurons which might have as few as 10 connections each[1].

Broadly speaking, an Artificial Neural Network (ANN) models a (typically nonlinear) relationship between its inputs and outputs. A problem which we wish to map to a Neural Network therefore typically has a fixed number of inputs and outputs. The inputs will be standardised so as to be well suited for the chosen transfer function of the neurons. This process is described in Section 2.4.2.

Neural Network models belong to the class of *connectionist* models. In such a model, information is stored in a distributed, essentially sub-symbolic manner. For example, if the problem is to form a mapping between handwritten digits and their respective label ('1', '2', etc.) then we might assign one neuron per digit and so have a symbolic mapping, where each neuron is trained to activate highly for its assigned digit. This is not necessary however; we may have more or fewer hidden neurons than there are digits and still find a mapping which solves the

[1]These numbers are by no means the minimum, nor necessarily typical. The number of neurons and connections is nearly always minute in comparison to real biological networks however.

problem satisfactorily.

If the number of hidden neurons is less than that of the inputs then we are performing a kind of *compression* on the input space, aiming to represent only the information we are interested in (for example in this case, what makes a '4' a '4'), but in a lower-dimensional space. For some problems we may instead want to have a hidden layer that is larger than the input layer so that we then have a much higher dimensional space in which to represent the information of the input space, and partition it such that we can accurately distinguish one digit from another.

An Artificial Neural Network is a (possibly cyclic) directed graph of *artifical neurons* (also referred to as 'nodes') connected by *synapses* (weighted connections or 'weights'). A neuron fires by computing a sum of contributions from its inputs (the activation) and passing this sum through an *activation function* σ to produce the output. Thus, for every neuron j, the output y_j is given by:

$$y_j = \sigma\left(\sum_i w_{ij} x_i\right) \tag{2.1}$$

where w_{ij} is the weight for the connection between neurons i and j, and x is the current input vector for that neuron (which may be externally applied, or composed of outputs from other neurons). Starting with the network inputs, each node calculates its activation and fires. The outputs are then propagated along all of the node outputs. This process is repeated until the output nodes are reached and fire, at which point the network output has been produced.

A network can be constructed from many types of artificial neuron. The choice of neuron determines the computational power of the network. Figure 2.1 shows a network composed of Threshold Logic Units (TLUs) which have an activation function of:

$$y(a) = \begin{cases} 1 \text{ if } a \geq \theta \\ 0 \text{ otherwise} \end{cases}, \tag{2.2}$$

where a is the neuron activation and θ is a threshold or bias. Their all-or-nothing nature makes them suitable for modelling Boolean functions or for representing decisions.

Figure 2.2 shows a function regression network which employs a hidden layer of sigmoid units and a linear output. The training algorithm will adapt the parameters of the sigmoids such that they fit the desired function[2]. The sigmoid activation function is calculated as:

$$y(a) = \sigma(a) = \frac{1}{1 + e^{-(a-\theta)/\rho}} \ . \tag{2.3}$$

We can alter the y-axis offset and steepness of the sigmoid by modifying θ and ρ. This makes the sigmoid a versatile activation function as it can be used both in

[2]The connection weights must also be set accordingly.

binary (threshold logic) and non-binary (function mapping) signal communication [Gur97].

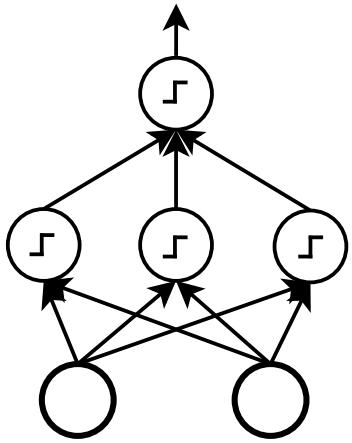


Figure 2.1: A network composed of an input unit layer, hidden TLU layer and output TLU layer. Such a configuration would be suitable for mapping linearly-inseparable tasks such as XOR.

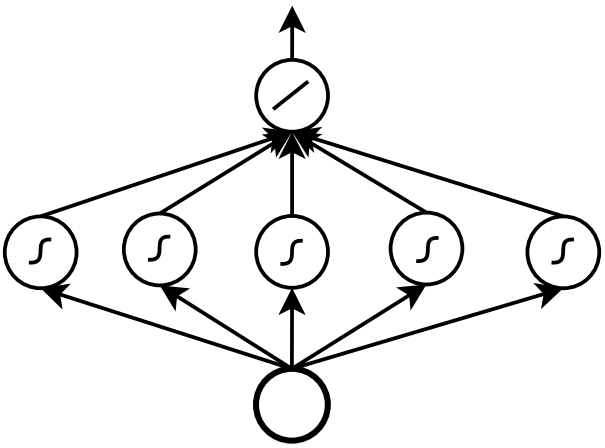


Figure 2.2: A network composed of an input layer, hidden sigmoid layer and linear output layer. This configuration would be suitable for mapping simple 1:1 functions.

2.2 Evolutionary Algorithms

The product of a wish to rigourously explain how natural systems adapt and further how to design artificial systems which exhibit the same properties, Evolutionary Algorithms (EAs) represented at their inception a new paradigm for problem representation and solution searching.

Drawing inspiration from nature, the search process typically starts with a population of possible solutions which have been randomly generated. We can also start the evolution from just a single individual. For any non-trivial problem the likelihood that a solution has been found through random generation is low. A structured random search is then conducted using the initial random population as a starting point.

Through a process of evolution life in the natural world has flourished, producing a wide range of living things that exhibit ingenious and highly optimised behaviour, from gathering energy from the sun to flying half-way across the world without stopping, countless examples could be given. The element that all these living things have in common is survival. They are with us today because they have survived over the millennia where others have failed. Why did the others fail? Broadly, we can put this down to an inability to adapt successfully to their (sometimes drastically) changing surroundings. Those that survived have done so through a process of continuous adaptation, with the tiny changes at each generation cumulatively producing major changes that allow these various organisms to cope with fluctuating temperatures, increased competition, scarcity of resources and so on. In the field of Evolutionary Computation we aim to emulate this adaptive behaviour to produce systems that exhibit the level of optimisation and robustness seen in organic systems. In particular, the ability to improve long after the initial training, based on experience, is a major goal here.

What if we could build such adaptive systems that mimicked nature? As an example consider a system with the purpose of examining scans of patients at a hospital with the aim of determining whether the patient has or does not have cancer. If such a system could be made to learn from its experience and improve over time, then more lives could be saved and the workload on doctors reduced. Even an improvement of a fraction of a percent could translate into a sizeable number of lives saved. Moreover if this process could be more or less *automatic*, as evolution appears to us to be, then our time is suddenly multiplied: it is no longer necessary to build every element of a new system 'by hand', we may instead focus our time on building better adaptive systems which can be used to solve many different kinds of problem.

Interestingly, the simplest of EAs can often show surprising ability in problem solving, simply through repeated application of the basic principles of survival of the fittest and random variation. Our understanding of the mechanics of evolution has improved greatly in the last 60 years, with both biologists and computers scientists mutually benefiting from each other's work. Computer scientists have benefited from the study of natural evolution and organic systems carried out by biologists, and biologists have benefited from the mathematical models and simulation techniques produced by the computer scientists.

Despite this progress, where we are now able to produce highly complex evolved systems, there still exists a huge gulf between artificial and natural evolutionary systems. Not only that, but our understanding of the dynamics of our artificial evolutionary systems, simplified as they are, is still insufficient to the point where

Algorithm 1 A simple Evolutionary Algorithm involving only selection and mutation (taken from [Jon06]).

```
Generate an initial population of M individuals.

Do until a stopping criterion is met:

  Select a member of the current population to be a parent.

  Use the selected parent to produce an offspring which is similar
  to but generally not a precise copy of the parent.

  Select a member of the population to die.

End Do

Return the individuals with the highest global objective fitness.
```

EA practitioners often find themselves using guesswork and intuition in their EA design, as before the simulation is done they have little chance of accurately predicting the outcome.

Over the course of the history of the field of Evolutionary Algorithms a number of specific classes of algorithm have emerged, often due to having been invented by different groups in an age without the ease of Internet-based communication. Each major algorithm type has its own set of assumptions about how artificial evolution should be conducted, and until recently, its own conferences. De Jong [Jon06] has presented his 'Unified Approach' which brings these different major algorithm types together in one framework with the aim of bringing together the knowledge and techniques of each community. We aim to follow the notation and ideas of this unified approach generally throughout this work, but focus in on individual algorithms particularly when discussing the history of the field. We now give an overview of a simple Evolutionary Algorithm using De Jong's framework. A more detailed treatment can be found in [Jon06].

2.2.1 A Simple Evolutionary Algorithm

Evolutionary Algorithms typically work through a process of repeated application of a process of producing successive generations by taking the fittest individuals and copying them to fill up the next generation. If we begin with a random population of M and repeatedly take the best and copy them, we will quickly approach a population which consists only of one of the better individuals in the initial population. The average fitness of the population is going up, but the diversity is crashing, which has the effect of halting progress once the population

has become uniform.

Without some kind of variation in the population, progress will eventually halt as the population converges to a single individual. A simple form of variation mimics copying errors in nature. The process of DNA copying in nature isn't quite perfect, with copy errors occurring infrequently, but often enough so that organisms can change significantly in the long term.

Another form of variation that takes inspiration from the natural world is that of recombination or crossover, where we take two parents of high fitness and form new individuals from the recombination of parts of each parent. The idea is that high fitness parents are composed of high fitness 'building blocks' that we then recombine to potentially produce an individual that is better than either parent in isolation.

This improvement isn't guaranteed, but with a large enough population and the right level of variation it becomes likely, as long as you have the patience to wait for many hundreds or even thousands of generations to pass. While the increase in computer speed has accelerated research in this field by allowing many thousands of generations of evolution to be simulated, simply running an EA for an extended period of time is unlikely to produce the desired results. The dynamics of the EA must be such that progress can be made for long enough until the desired solution is found.

Algorithm 1 describes a very simple EA which contains the most basic elements of most EAs. In this most general form, an EA can be seen to be composed of three stages, which we repeat until we have a suitable solution:

1. Parent Selection: Using some criteria such as high fitness, select a member from the population.

2. Variation: Apply probabilistic operators to the selected member to produce a new individual. This could be mutation which operates on a single individual or a form of recombination (e.g. crossover) which works on two or more.

3. Survival Selection: Select a member of the population (perhaps one of low fitness) to leave the population to make space for the new individual.

Even such a simple model is capable of making progress on simple problems. Neuroevolutionary algorithms are typically much more complex however, examples of which will be presented in Section 2.3.2.

2.2.2 Crossover: Idea vs. Mechanism

We take a short detour now to discuss an issue which will be important in our investigation, namely that of why crossover works, and how we can divide this explanation into the benefit attributable to the *idea* of crossover and that which is due simply to its *mechanism*.

The idea of crossover aiding the search process by recombining short, high fitness sections of the genotype has been shown to be somewhat problem dependent. In some problems there are verifiably no building blocks at the genotype level [BPJ05] and in others, such as that of the evolution of Neural Networks, the building blocks are at the very least not obvious [Yao99].

Despite the lack of building blocks in some problems, crossover may still be an effective search operator. This implies that crossover is able to have a beneficial effect that does not involve the recombination of building blocks. Jones [Jon95] tested this hypothesis, drawing the distinction between the *idea* and the *mechanism* of crossover.

It has been demonstrated that while the application of crossover may provide satisfactory performance, there is a clear distinction between the advantages afforded by the *idea* of crossover and those which are a consequence simply of its *mechanics* [Jon95]. The idea of crossover is to recombine members of a population such that the best characteristics of each are used to form new individuals of higher fitness. The common characteristics of the fit individuals then become *building blocks* which are represented by short, high fitness sections of an individual's genotype. The aim of crossover is to propagate these high fitness building blocks throughout the population, raising average fitness by steering the population towards promising areas of the search space.

In the early days of Genetic Algorithm research there were occasions where an improvement in performance offered by introducing crossover was taken to signify the existence of building blocks in the genotype [Fog06a]. It has however been shown that for many problems where crossover was believed to be recombining building blocks, it was in fact performing a macromutation [JF00, Fog06a]. Additionally, other aspects of how genetic algorithms work have been questioned with the result that long-held principles have been shown to be false or incomplete [WM97, Rud97, FG97, Fog06b].

Jones [Jon95] presents a means by which the *possible* existence of building blocks in a genotype can be ascertained in a less ambiguous manner than with previous methods. The problem is first attempted using a process of proportional selection and crossover, then compared to the same process using *random crossover*. The aim here is to disrupt the building blocks (if any are indeed present) using the random crossover operator and see how performance is affected. Random crossover entails performing crossover on one fit individual and a randomly generated individual. Given that the second parent has been generated randomly, its fitness will on average be very low. Combining a fit individual with a random individual effectively removes the implicit information sharing offered by a population which clearly violates the idea of crossover. If this approach is at least as effective as traditional crossover then it suggests that we do not require the *idea* of crossover, but that its *mechanics* may be effective. This idea is explored further in Section 5.3.

2.3 Neuroevolution

Neuroevolution (NE) is one term often used to refer to the application of Evolutionary Algorithms to the optimisation of Neural Networks. NE algorithms can be divided into those that optimise

- network architecture,
- the weights of a network of pre-determined architecture, or
- both architecture and weights.

We present and discuss some examples of NE algorithms in Section 2.3.2 which exemplify each of these categories. However as the focus of this work is on the Permutation Problem, we only work directly with algorithms which discover and optimise weights for pre-determined network architectures.

2.3.1 Test Problems

Some examples of problems addressed previously using Neuroevolution include:

- Prediction, e.g. stock market performance [KAYT90, SW98, ZPH98, Zha01, ET05, CSW06]
- Medical diagnosis [YL97, WYX04, YI08]
- Classification [Zha00]
- Minimally Invasive Surgery [MGW+06]
- Guidance of a finless rocket [GM03b]
- Allocation of cache in a multi-processor system [GBM01]
- Playing 'Go' [SM04]
- Real-time evolution of game agents [SBM05]
- Simulated car driving [ATL07]
- Robotics, e.g. double pole balancing without velocity information [GM99, SM02, GSM06, HN08, HN09a]

A characteristic common to several of these problems is that while the start and end states are well defined, the exact behaviour required in order to reach the end state is not known. For example, the 'Go' board has a set starting arrangement and a set of arrangements which signify a win for a given player. From the perspective of one of the players, we wish to reach a winning state from

our given starting state. While the goal is clear and well defined, the actions to be taken at each step towards reaching it are not. Likewise we would like our finless rocket to reach as high a height as possible from its starting point on the ground, but what its various thrusters should do at each millisecond in order to achieve this is unclear.

Reinforcement learning emerged as a possible solution to this type of problem. In this type of learning, an agent *reinforces* its perception of an environment through a process of trial and error, not unlike that of an infant. The agent attempts the task and is rewarded/penalised based on its performance. Through these rewards and penalties, the agent is able to adapt its behaviour to perform the task with a greater degree of skill without being told explicitly what it was expected to do at each time step. Neuroevolution has shown promise on a number of problems which have been found to be too difficult for several Reinforcement Learning-based algorithms [GM99], making NE a worthwhile area of research to explore.

2.3.2 Algorithms in the Literature

In this section we give a brief overview of some algorithms found in the literature. There are of course many other algorithms that could be included here; those here have been chosen for the influence they have had on the field as a whole.

The GNARL Algorithm

The GeNeralized Acquisition of Recurrent Links (GNARL) algorithm searches for a solution through mutation only. Each network consists of two fixed interfaces i.e. a fixed number of inputs and fixed number of outputs so the task of the algorithm is to construct the layer(s) in between.

Each initialised network may be sparsely or fully connected. The network connections are subject to the following three constraints [ASP94]:

1. There can be no links *to* an input node.
2. There can be no links *from* an output node.
3. Given two nodes x and y, there is at most one link from x to y.

These constraints serve to restrict the search space. It is worth noting that constraint 1 & 2 together prohibit the construction of Jordan model recurrent networks [Elm90] which is a very simple recurrent model. The network can still form Elman model networks [Elm90] however it is not particularly likely to do so.

At each generation the fitness of each network is evaluated. The top 50% of the population are chosen to be the parents for the next generation. Each network undergoes both parametric and structural mutation. The severity of each mutation to a given network η mutation is determined by that network's temperature $\mathrm{T}(\eta)$:

$$T(\eta) = 1 - \frac{f(\eta)}{f_{max}} \tag{2.4}$$

where $f(\eta)$ is the fitness of the network η and f_{max} is the maximum possible fitness as defined by the problem. The search is therefore initially coarse-grained becoming progressively finer-grained as the population converges on a solution.

The parametric mutation is performed by perturbing each weight in the network with Gaussian noise:

$$w = w + N(0, \alpha\hat{T}(\eta)) \tag{2.5}$$

where $N(\mu, \sigma^2)$ is a Gaussian variable, α is the learning rate and $\hat{T}(\eta)$ is the instantaneous temperature of the network η:

$$\hat{T} = U(0,1)T(\eta) \tag{2.6}$$

where $U(0,1)$ is a uniform random variable over the interval $[0,1]$.

GNARL defines four structural mutations:

1. Add a zero-weighted connection.[3].
2. Remove a connection.
3. Add a hidden node with no incident connections.
4. Remove a hidden node and all incident connections.

The algorithm itself is conceptually rather simple yet Angeline et al. report that it is able to produce solutions to relatively complex problems that should require a significant number of recurrent connections [ASP94].

EPNet (1997)

Another Neuroevolutionary algorithm which closely follows the Evolutionary Programming (EP) paradigm [FFP90] is EPNet, presented by Yao and Liu [YL97] in 1997 and extended over the next decade to evolve ensembles of networks [YI08].

Following the EP paradigm, EPNet discovers new networks by mutating individuals in isolation, i.e. there is no recombination of individuals. Variation is achieved through a collection of additive and deleterious mutation operators, ordered with deletions first to encourage parsimony of networks.

EPNet is an example of a hybrid algorithm which uses a combination of evolution, Backpropagation and Simulated Annealing to produce new networks. In this manner networks evolve essentially independently with no direct sharing of information. Later work makes use of the different solutions to the problem in the

[3]The weight is zero so that the new connection does not affect the behaviour of the network. The network must incorporate the link through incremental evolution.

population by taking the top n networks and combining them to form an *ensemble* which often provides a better solution than any of its individual members [YI08]. Two open issues here are how to discover diverse solutions and then how to select which to include in the ensemble.

Symbiotic, Adaptive Neuro-Evolution (SANE, 1996)

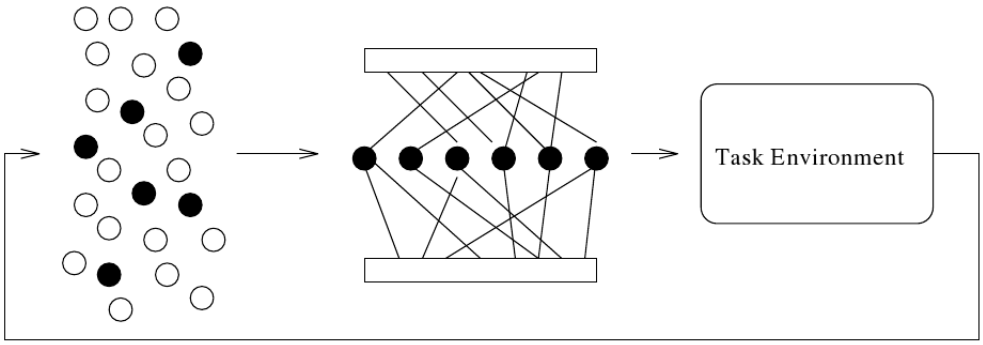


Figure 2.3: The SANE algorithm: Each neuron in the target network is selected from a single population of neurons. Figure taken from [GM99].

In traditional applications of Genetic Algorithms, for example in function optimisation, the convergence of the population towards a solution is desirable. The goal of SANE however was to avoid this convergence so as to foster the ability to adapt to changes in the task environment by always maintaining some diversity [MM96].

It achieves this by departing from the model employed in GNARL or EPNet where each individual in the population is a complete solution to the problem. Instead, each individual represents just a part of the solution which introduces the idea of a symbiotic relationship between the individuals.

Symbiotic Evolution Evolutionary symbiosis can either be cooperative (where members of the population work together to achieve the task) or competitive (where individuals succeed by competing directly with and winning against each other[4]). SANE employs the cooperative variation, where each individual is a single neuron and the aim is to discover a combination of the individuals which solves the given problem.

[4]Typically, when evolving a chess playing Neural Network, the networks would compete against a known chess AI algorithm. In competitive symbiosis the networks would play chess against each other.

Algorithm 2 One generation of the SANE algorithm (adapted from [MM97]).

1. Clear the fitness values of each neuron and blueprint.
2. Select n neurons from the population using a blueprint.
3. Create a Neural Network using the selected neurons.
4. Evaluate the network on the given task.
5. Assign the blueprint the evaluation of the network as its fitness.
6. Repeat steps 2-5 for each individual in the blueprint population.
7. Assign each neuron the evaluation of the best 5 networks in which it participated.
8. Perform crossover and mutation operations on the blueprint and neuron populations.

The SANE Algorithm We begin with a population of randomly initialised neurons. A network is constructed by selecting probabilistically from the population n times where n is the number of hidden units in the architecture (which is predefined). The fitness of the network is calculated and passed to each neuron. The neuron calculates its effective fitness by dividing the sum of its network fitness scores by the number of networks it has participated in.

Over time the neurons with the highest average fitness will be preferred when constructing the network. Viewed from a coevolutionary perspective the high-performing neurons are cooperating effectively with other members of the population. Those with low fitness scores are selected against. One generation of the SANE algorithm is outlined in Algorithm 2.

This average performance metric is used to select which neurons to crossover and mutate in order to sample new points in the problem space. The blueprint representation is an alternative representation of the population where crossover occurs on a higher level i.e. complete neurons and connections are crossed rather than individual bits at the neuron representation level.

Enforced Sub-Populations (ESP, 1999)

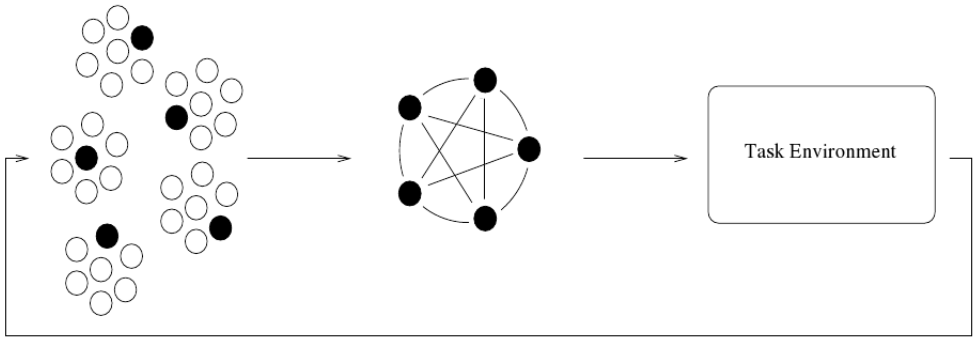


Figure 2.4: ESP: Each neuron in the target network is selected from a sub-population. Figure taken from [GM99].

The ESP algorithm is an extension of SANE. In both algorithms the strategy is such that rather than evolving individuals which are complete solutions to the problem, symbiotic individuals cooperate to form a solution. In SANE each individual is a neuron selected from a single population.

ESP extends this by drawing each neuron from its own sub-population. Each node position in the network is assigned a sub-population. Thus, the task of each sub-population is to evolve a neuron which best fits that position in the network.

The aim of this measure is to only recombine individuals from the same species (sub-population). Each sub-population then converges rapidly towards a solution for the sub-task it is required to perform. We present an extension to this algorithm in Section 5.3.

Neuroevolution of Augmenting Topologies (NEAT, 2002)

The aim of the NEAT algorithm is to evolve minimally complex structures which solve the given problem [SM02]. This is achieved by evolving both the structure and weights from a minimal structure and aiming to only allow contributing structure to survive in the long-term. NEAT speciates the population (as in SANE and ESP (Section 2.3.2)) to ensure that only compatible networks are crossed over. The speciation measure in NEAT extends that which is found in prior algorithms by explicitly numbering each gene as it appears in the population. Thus, at any generation the origins of any particular individual can be tested and used to determine likely compatibility for crossover, providing a significant performance gain over topological analysis methods (at the cost of accuracy) [SM02].

Figure 2.5 demonstrates crossover with explicit gene numbering. The genetic material of the fitter of the two parents will dominate the crossover. This, coupled with the alignment of compatible genes allows for more reliable crossover, addressing some of the concerns presented in [ASP94].

Crossover with explicit gene numbering

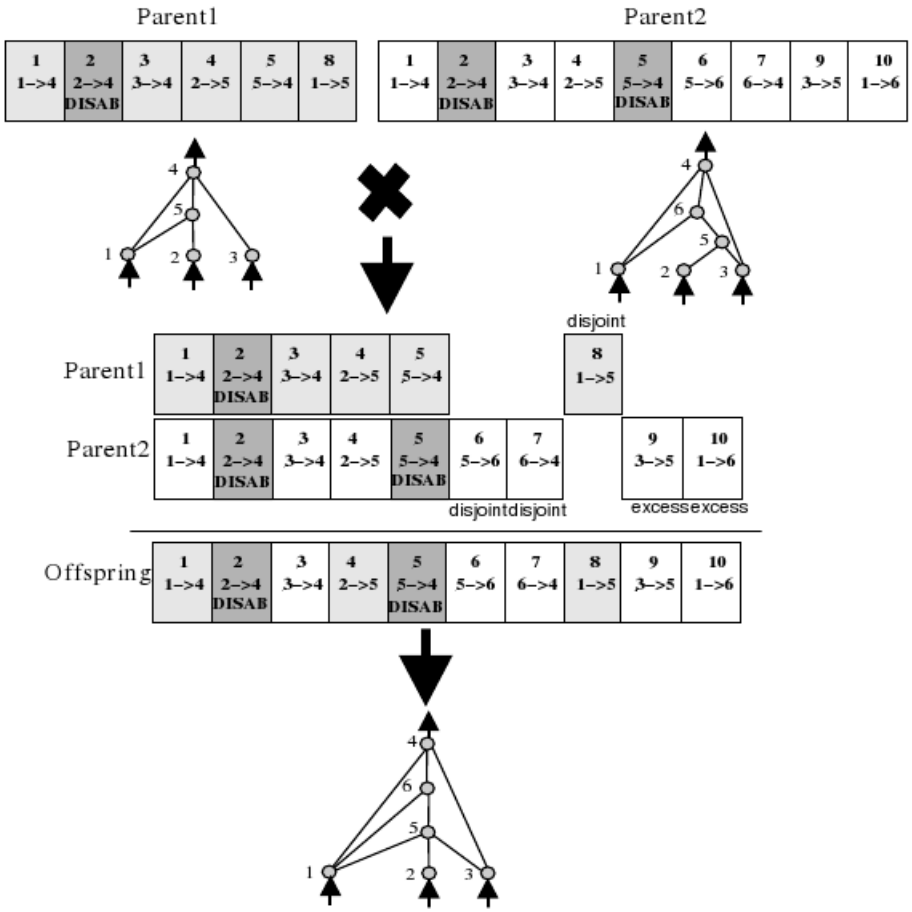


Figure 2.5: Crossover in NEAT: As the genes are numbered, any two individuals can be lined up such that their matching genes will be crossed over directly. The disjoint genes (6 & 7 in parent 2) and the excess genes (9 & 10 in parent 2) are inherited from the more fit individual; thus parent 2 has the higher fitness in this example. Figure taken from [SM02].

Real-time NEAT (rtNEAT, 2005)

An extension of the original NEAT algorithm [SM02] to support online evolution is rtNEAT [SBM05]. In this algorithm, individual members of the population are referred to as *agents* to more accurately reflect their purpose. The population size is approximately constant, with a new generation being formed every n ticks (time steps).

The next generation is formed by removing the poorest performing agent from the population, and replacing it with an agent formed by crossing over two agents from a probabilistically chosen species with a high measure of fitness.

In order to provide the protection of innovation in the original NEAT algorithm it is necessary to adjust the fitness measure. If the fitness alone was used to determine performance, then new structures would likely be removed immediately given that a structural mutation is unlikely to improve the fitness of an individual without further evolution. The solution in rtNEAT is to take into account the size of a species, and the age of agents: agents must be given a minimum of m time steps in which to evolve before being considered for removal. The parameter m is set experimentally and is dependent on the problem type and level of difficulty.

Cooperative Synapse Neuroevolution (CoSyNE, 2006)

In [GSM06] the CoSyNE algorithm is presented and shown to outperform the state of the art in machine learning methods (both single-agent and evolutionary) for the non-Markov dual pole balancing task. The design principles of CoSyNE are very close to that of ESP [GM03b], in that by explicitly forming sub-populations with a particular assigned sub-task the overall population remains diverse. The main distinction between the two algorithms is that CoSyNE co-evolves individual synapses/connections rather than neurons as in ESP.

Compositional Pattern Producing Networks (CPPN-NEAT, 2007)

The CPPN-NEAT algorithm [Sta06] differs considerably to algorithms such as ESP, CoSyNE and even NEAT itself as the genotype is represented by a function graph and the phenotype is created by rendering onto an arbitrarily-sized canvas. Thus the phenotype itself is not a Neural Network but an image.

Figure 2.6 shows a function graph which accepts two inputs x and y and produces one output. The output of the network is a level of intensity which is interpreted as a level of gray on the canvas; figure 2.7 demonstrates how the phenotype is rendered to the canvas. Through the introduction of periodic functions and recurrent connections, symmetric features can appear and propagate throughout the population. The results of [Sta06] show the emergence of developmental traits found in biological development such as repetition and repetition with variation.

As the aim of NEAT is to complexify structures over time, the networks will produce increasingly complex phenotypes over time. The explicit tracking of all genes in the population by the NEAT algorithm also allows the ancestors of each individual to be examined, showing the timeline of how the phenotype was constructed.

While the algorithm at present generates images only, the concepts behind the representation and the emergent symmetry in the phenotypes has significance for future research into developing networks with indirect encodings.

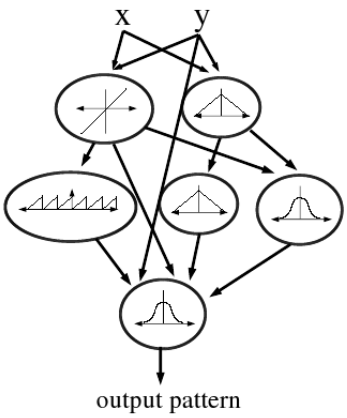


Figure 2.6: CPPN-NEAT: A function graph representing the genotype. The connections between function nodes may be complexified over time to include recurrence. Figure taken from [Sta06].

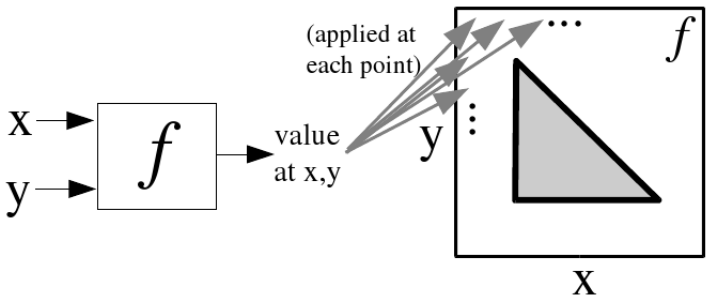


Figure 2.7: CPPN-NEAT: The function network evaluates pairs of (x, y) coordinates and outputs a level of intensity which determines the level of gray for the pixel at that coordinate. This rendering can be performed for any size of canvas. Figure taken from [Sta06].

2.4 Applications

In this section we briefly review the test problems used in this work. More detail for the individual problems is given in the relevant sections where they are used.

2.4.1 Classification Problems

The UCI datasets are from the University of California, Irvine Machine Learning Repositiory [BM98].

UCI Iris

A very simple problem with a long history in academic research, the Iris classification problem is useful as a first problem to test classification algorithms on. It involves classifying types of Iris based on various measurements of the plant itself. One class is linearly separable from the other two; the remaining classes are not linearly separable.

UCI Cancer

The UCI cancer dataset is a set of tissue measurements each taken from a digitised image of a fine needle aspirate of a breast mass. There are 10 features in total, recorded three times for each tissue sample. The two possible classes of benign and malignant are linearly separable if all 30 attributes are used. While a more complicated domain than the Iris problem, similarly high classification rates can be achieved. A model based on this dataset (in other work) has correctly diagnosed at least 176 consecutive new patients [BM98].

UCI Diabetes (Pima)

The UCI Pima Diabetes dataset represents the most difficult classification problem in this suite. The problem consists of 8 input variables about the subject, including number of times pregnant, serum insulin level, age etc. The aim of the problem is to predict whether the subject is diabetic or not. The dataset is somewhat unbalanced, with 500 instances where the subject is not diabetic versus 268 where the diagnosis is positive.

Cart-Pole Balancing

The cart-pole balancing problem involves a controller applying force to a cart on a track which has two hinged poles attached to it. The controller must keep the poles within a certain angle for a fixed number of time steps. Two variants of the problem of increasing difficulty are used in this work. In the more difficult case, the controller is not given the velocity of the poles or the cart and so must either infer the velocity or discover an alternative strategy. This problem is described in more detail in Section 5.2.3.

2.4.2 Standardising Datasets

It is common to standardise datasets such that their inputs and outputs meet certain criteria, such as being centered around zero. Such standardisation is beneficial when the Neural Network is composed of sigmoidal neurons as if the inputs are spread over a wide range they may cause the neurons to saturate. Ideally we would like the inputs to fall within the semi-linear section of the sigmoid curve.

In this work all datasets are standardised such that each input will have mean zero and standard deviation one. Given a variable X of raw training instances X_i we can calculate the corresponding standardised value S_i as

$$S_i = \frac{X_i - \bar{X}}{\text{std}(X)}, \tag{2.7}$$

where $\bar{X}$ is the average of X

$$\bar{X} = \frac{\sum_i X_i}{N} \tag{2.8}$$

where N is the number of training samples for X. The standard deviation $\text{std}(X)$ is then calculated as

$$\text{std}(X) = \sqrt{\frac{\sum_i \left(X_i - \bar{X}\right)^2}{N-1}}. \tag{2.9}$$

Standardising training, testing and validation subsets

In order to avoid bias in the training process it is necessary to avoid standardising the training, testing and validation sets separately. If each subset is standardised individually then each training point may be mapped to a different standardised value in each, confusing the training process. Standardising before dividing the dataset is also not advisable as then aggregate information of the testing and validation set are present in the point mapping of the training dataset. To avoid such a bias, we calculate the mean and standard deviation using the training set and use these values to standardise each subset. In this manner equivalent points will have the same mapping in the standardised space but the training process will truly contain nothing of the validation and testing sets, not even aggregate information.

2.5 Chapter Summary

In this chapter we have introduced the background necessary for the understanding of the main body of work. We began with an overview of Neural Networks, including a presentation of their biological inspiration and the resulting field of Connectionist Computing. We then presented an overview of the different 'families' of Evolutionary Algorithms, and outlined a simple Evolutionary Algorithm which will serve as a rough base for those presented in the main work. The combination of Neural Networks and Evolutionary Computation, Neuroevolution, was then presented. We highlighted some applications where Neuroevolutionary methods tend to be of particular use, and presented overviews of a selection of state-of-the-art Neuroevolutionary Algorithms. Finally, we briefly presented the test problems used and the process employed in standardising them.

Chapter 3

The Permutation Problem

3.1 Introduction

In early research into how Evolutionary Algorithms might be applied to the problem of Neural Network design and weight optimisation (the field of Neuroevolution), a symmetry in the representation was quickly highlighted as a serious problem. The problem, which came to be known variously as the *Permutation*, *Competing Conventions*, *Isomorphism* and *Structural/Functional Mapping Problem*[1] received significant attention in this period of early research, with some researchers highlighting it as a serious problem to be overcome and others stating that the problem did not appear to be as serious as it first appeared.

In its described form, the negative side-effects of the problem only manifest under particular conditions when two neural network genotypes are crossed over. As such, references to the problem in recent research have typically taken the form of citations to the early research, often as a justification for the avoidance of crossover and so the avoidance of the problem. An issue with this is that to date the investigations into this problem have not provided a definitive conclusion regarding the severity of the problem[2]. Those studies that aimed to determine the severity have invariably concluded that the Permutation Problem *appears* not to be serious [MD89, BMS90, Han93, FS08]. These conclusions are based on empirical evidence regarding algorithm performance; what is missing is tests for the presence of the problem itself in order to first determine whether it occurs at all, and then if it does, to estimate its rate of occurrence and effect on performance. In considering the severity of a particular problem it is necessary to consider not only what the effect is when the problem occurs, but also its rate of occurrence. An aim of this work is therefore to provide exact probabilities for the rate of occurrence of the problem as well as an evaluation of its severity when it does occur so that the nature of the problem can be more definitively characterised.

[1] Throughout this work we refer to this problem as the *Permutation Problem*.

[2] This is not a criticism of past work, rather the conclusions are simply stated using relatively non-committal language.

This will be explored in detail in Chapter 4.

A key distinction which must be made would be between an investigation into the efficacy of crossover versus an investigation into the severity of the Permutation Problem. It is a feature of the field that there exists much work where crossover has been avoided, with the Permutation Problem being a commonly-cited justification. However, it is unclear whether the Permutation Problem is necessarily the cause of poor performance when using crossover or whether there is another explanation. Examining performance of an algorithm with and without crossover only tells us about the efficacy of crossover in that particular setup. We may also coincidentally learn about the Permutation Problem in this way, but ultimately we cannot be sure of our conclusions if we do not isolate and remove confounding factors (e.g., crossover may be a good/bad fit for our given test problem(s)). This issue is investigated in more detail in Chapter 5.

In this chapter we begin by identifying the kinds of permutation that we will be discussing in this work. We then present our definition for the Permutation Problem and investigate its origins as a concept, and also how its naming conventions emerged over time. Finally we discuss the distinction between a closely related problem which we term the Incompatible Representations Problem and the Permutation Problem.

3.2 Problem Definition

Before defining what the Permutation Problem itself is, it is necessary to identify the different kinds of permutation which may be encountered either in the literature or in practical settings. If the term 'Permutation Problem' is used without qualifying exactly which kind of permutations are being considered then we have a possible source of ambiguity which must be resolved if the problem is to be discussed effectively and without ambiguity. We therefore first distinguish between the different types of permutation before moving on to the definition of the Permutation Problem itself.

3.2.1 Types of Permutation

In this section we define the four types of permutation that we consider in this work:

- Genotypic Weight Permutations (GWPs)
- Phenotypic Weight Permutations (PWPs)
- Exact Role Permutations (ERPs), and
- Similar Role Permutations (SRPs).

References to 'permutations' in the literature may refer to one or several of these types of permutations implicitly. In this work the two types that we are most concerned with are Phenotypic Weight Permutations (PWPs) and Exact Role Permutations (ERPs).

Genotypic Weight Permutation (GWP)

This is the broadest category of permutations considered. Such permutations occur frequently[3] and have not been identified as being a concern regarding the application of Evolutionary Algorithms. We introduce such permutations as they are the easiest to calculate probabilities for and can be used to illustrate the method which is applied to the other kinds of permutations.

In the context of evolving Neural Networks, Genotypic Weight Permutations (GWP) are not typically phenotypically equivalent or even similar in functionality or fitness. While each genotype will share the same values, they may not necessarily be assigned to the same neurons in the same order. However, every Phenotypic Weight Permutation (PWP) is also a GWP; as such the probability for encountering a GWP pair provides a rough upper bound for that of the phenotypically-equivalent PWP.

We expect GWPs to be common but to have no appreciable impact on performance for most problems. Although GWPs are not a practical concern, their simplicity will aid the explication of the method for probability calculation presented in Chapter 4.

Phenotypic Weight Permutation (PWP)

Given a fully-connected feedforward Neural Network with one hidden layer of N_h hidden neurons, we can form up to $N_h! - 1$ symmetric, phenotypically-equivalent networks by permuting the neurons of the hidden layer. Each network contains exactly the same neurons, arranged in a different order. Two neurons are exactly the same if they are composed of the same input and output weights, in the same order. This definition will be expanded in Section 3.2.2.

Exact Role Permutation (ERP)

Exact Role Permutations (ERPs) are a superset of Phenotypic Weight Permutations where the neurons need not have exactly the same weights; they are required only to map the same function. An intuitive way to see how differing weight-sets could result in identical behaviour is to imagine two neurons with weights which both cause constant saturation of the neuron transfer function; thus while potentially very different in terms of their weights, the neurons are outputing a constant

[3]For example with a binary genotype we only require that the two strings have the same number of ones (and so the same number of zeroes). This will happen quite frequently and is not typically a concern.

value of zero or one (in the case of a sigmoidal neuron). Another example would be when an odd transfer function such as tanh is used: flipping the signs of all weights will result in the same function.

If the space of all possible neurons contains only neurons which can be considered to have unique functionality then the number of ERPs is the same as the number of PWPs, i.e. $|ERP| = |PWP|$.

Similar Role Permutation (SRP)

Similar Role Permutations (SRPs) are a superset of ERPs and contain all pairs of individuals which are composed of neurons encoding exactly identical or only *similar* functions. The level of redundancy in the space is ultimately decided by the definition of similarity used; the less strict the similarity criteria the more redundant the space will be.

3.2.2 Definition

We now present our definition of the Permutation Problem. Given a fully connected feed-forward Neural Network with one hidden layer, we may take a weighted sum of the hidden neuron contributions in any order without affecting the function encoded by the network as a whole. A direct consequence of this is that any given network has *up to*[4] $N_h!$ permutations which while genotypically dissimilar all encode the same function and therefore have the same fitness.

We can see why this is if we examine the equation for a single-layer, feedforward Neural Network. For simplicity the output layer is composed of a single linear output. The network is defined by,

$$g(\mathbf{x}, \mathbf{w}) = \underbrace{\sum_{i=1}^{h}}_{\substack{\text{evaluable} \\ \text{in any} \\ \text{order}}} \left[w_{h+1,i} \tanh \left(\sum_{j=1}^{n} w_{ij} x_j + w_{i0} \right) \right] + w_{h+1,0}, \tag{3.1}$$

Where,

- n is the number of inputs,
- $\mathbf{x}$ is an $(n+1)$-vector of inputs (n inputs plus a fixed bias value),
- $\mathbf{w}$ is an $((n+1)h + (h+1))$-vector of parameters (weights),

[4] It is important to remember to account for the duplicate permutations formed due to repeated allele values; therefore the number of permutations is only $N_h!$ when all alleles are distinct.

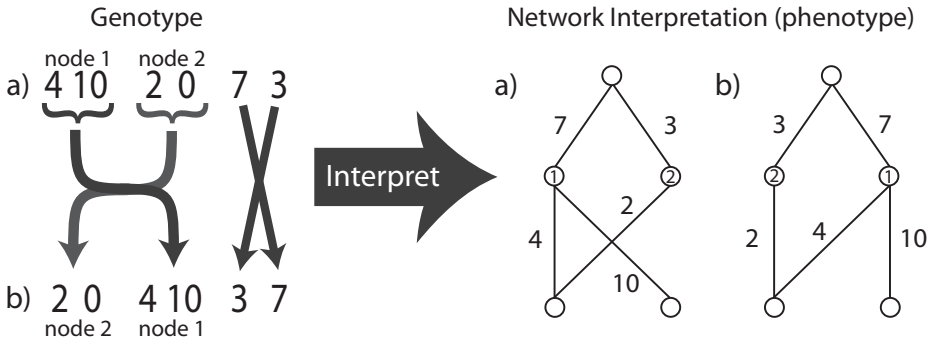


Figure 3.1: An instance of the Permutation Problem: two networks composed of the same hidden neurons but in a different order. While phenotypically equivalent, the genotypes are incompatible for crossover: any recombination will result in information loss. Of note however is that to form (b) from (a) the block of genes for each hidden node must be swapped, and the output weights inverted. Adapted from [YL97].

- w_{ij} is the weight for the contribution from neuron j to neuron i and
- h is the number of hidden neurons.

The part of the equation to note is the outer summation; when calculating this sum the order of the terms is unimportant.

An example of the Permutation Problem can be seen in Figure 3.1. Here, genotypes (a) and (b) are significantly dissimilar but in fact encode phenotypically-equivalent networks. This equivalence is due to the fact that the hidden neurons have been permuted; since neuron position is unimportant in this type of network model the networks then have identical behaviour and so equal chance of being recombined when selected under fitness-proportional selection.

If we apply any form of crossover operator to these genotypes we will inevitably lose information from their respective shared distributed representations due to duplication of one or more genes. If blocks of weights corresponding to whole neurons are swapped, then we have a phenotypic permutation. It is worth noting again that any phenotypic permutation is also a genotypic permutation (but not vice versa). Figure 3.2 explains the mechanism of the Permutation Problem in terms of recombination in more detail.

3.2.3 A General to Specific Ordering of Permutations

Discussion of the Permutation Problem is often hampered due to the fact that we can consider the problem from a number of levels of abstraction, as illustrated in Figure 3.3, though there is no established terminology for these levels. The implications and possible severity of the problem changes at each level. As such, proper discussion of the problem requires first clarifying where in this general-to-specific ordering the Permutation Problem should actually sit.

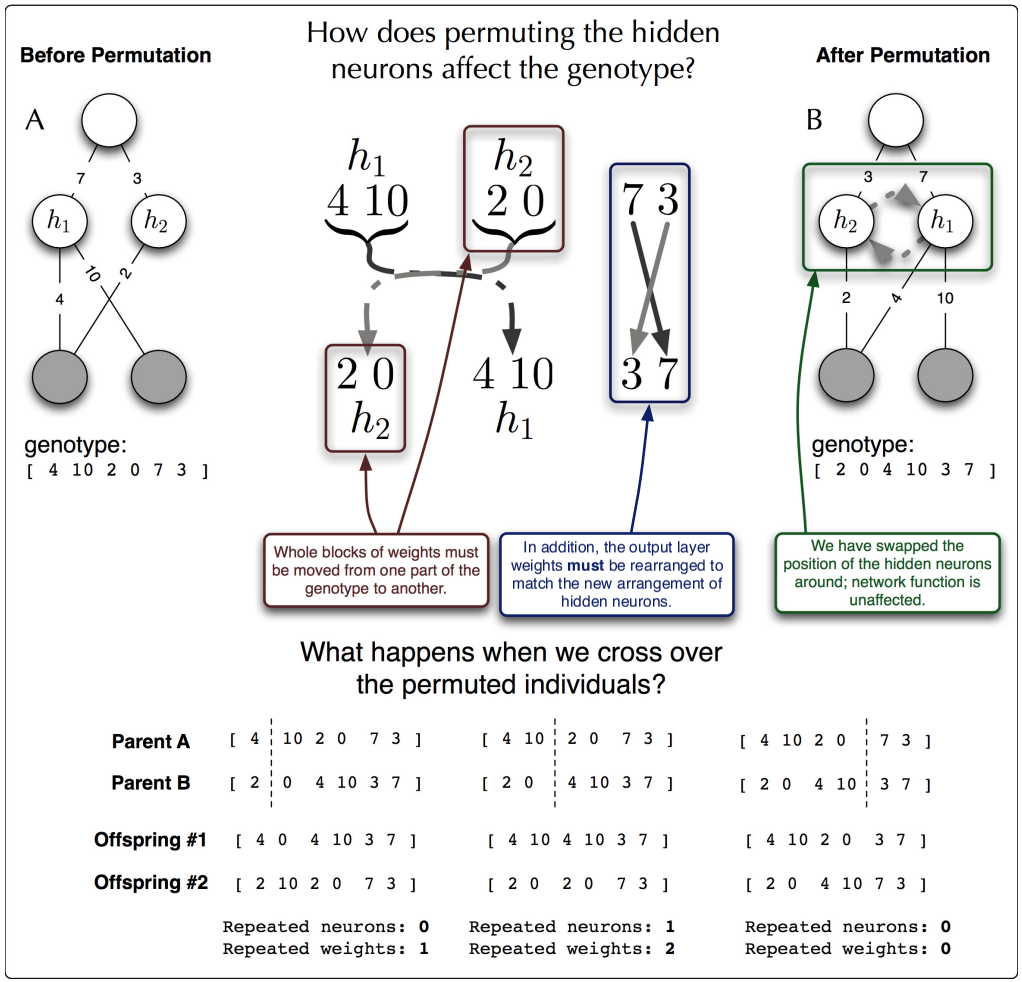


Figure 3.2: A closer look at the operations required in order to transform one network into a permutation of itself. Examples are given for a sample of the possible recombinations that can be produced using 1-point crossover. Adapted from [YL97].

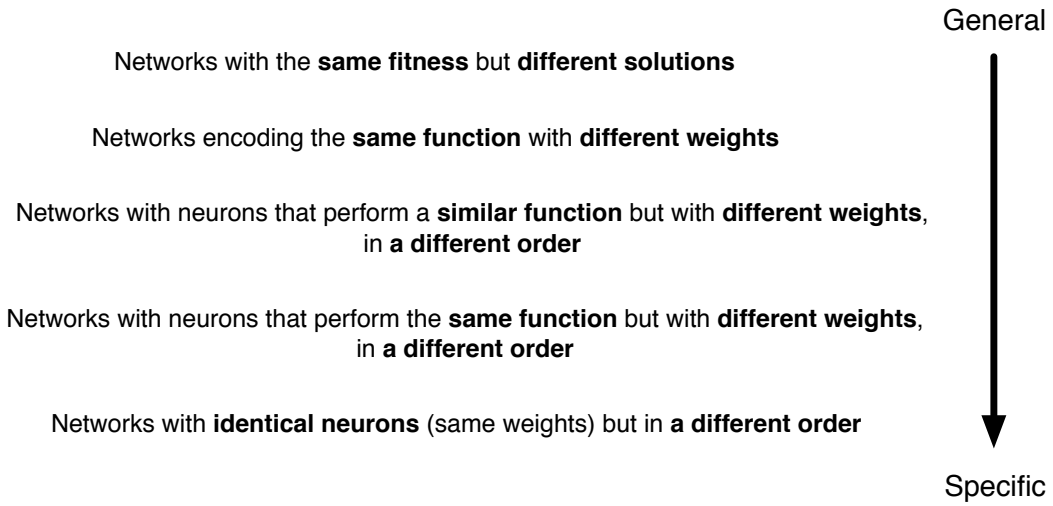


Figure 3.3: A selection of problems related to the recombination of networks, from the most general (multiple solutions with identical fitness but possibly underlying differences) to most specific (two networks with identical neurons in a different order). It is important when discussing the Permutation Problem that we define clearly at which level we place the problem, as this choice of placement is not necessarily the same throughout the literature. Most research tends towards the specific end of the scale.

Arguably there is no right answer to that question. Instead we couch it in terms of where each treatment of the Permutation Problem in the literature fits. This problem is compounded by the number of aliases for the Permutation Problem. Each alias can in some cases indicate from which perspective a researcher views the problem, i.e. at which level of the general-to-specific hierarchy their understanding of the Permutation Problem lies, though this interpretation is naturally subjective and so unreliable. The fact that these aliases are frequently presented as being completely synonymous causes misunderstandings to arise as while researchers may believe that they are discussing the same issue they may in fact be speaking of closely related but distinct problems. This is not often a problem when discussing the general issue of symmetry and equivalence in a genetic representation, but it could lead to drastically different conclusions regarding how ubiquitous the problem will be in practice, and therefore how serious it is. As such, the establishment of naming conventions would aid discussion of these problems and their relative severity.

3.3 On the Origin of the Permutation Problem

We now explore the earliest research into Neuroevolution where the *concept* of the Permutation Problem was first discussed. We then present how its naming conventions emerged, and discuss the two principal interpretations of the problem,

namely whether we view it as involving permutations of identical neurons or whether it is sufficient only that the neurons map the same functionality.

The first published reference to the problem of neurons which play similar roles but appear in different positions appears to be in the work of Montana and Davis in 1989 [MD89]. We note however that this problem is more general than the Exact-Weight Permutation Problem, which requires that the neurons have exactly the same weights. Montana and Davis describe this more general problem but do not explicitly name it. It would appear however that the first to discuss and name the Permutation Problem was Nick Radcliffe, on a prominent Genetic Algorithm Mailing List in 1988[5] (see Appendix A). It would appear that the Permutation Problem received significant discussion under different aliases around this time (1989-1992 approx.) due to the increase in applications of Genetic Algorithms to the optimisation of Neural Networks.

In this early work Montana and Davis provide significant insight into the Permutation Problem. At no point is the problem named, but the issues involved are clearly discussed. At the time of its publication the application of Evolutionary Algorithms (particularly Genetic Algorithms) was still in its infancy; Montana and Davis cite the lack of success in contemporary work such as that of Whitley [Whi89] in applying GAs to Neural Network optimisation. The problem is identified and described without citation to previous work and without use of any specific naming convention for it. Interestingly, the definition even goes further than other, later definitions by noting that the symmetry of the Permutation Problem can be present in networks with more than one layer, asserting that, "For a fully connected, layered network, the role which a given node can play depends only on which layer it is in and not on its position in that layer".

Montana and Davis present a number of variation operators (e.g. variants of crossover and mutation) which aim to address some perceived problems in the application of GAs to NN. The operator of particular interest is the one that they term 'CROSSOVER-FEATURES'. The purpose of this operator is to reduce the, "dependence on internal structure" (introduced by the Permutation Problem symmetry) by rearranging nodes in order to match up those which play the *same* role such that they appear at the same point on their respective genotypes. The method for determining and matching roles of neurons is not discussed beyond stating that it involves a process of presenting each neuron with certain inputs and comparing the response. As the metric for or conditions under which two neurons would be considered to be playing the same role is not given we assume that the method was simplistic in nature, for example each neuron is presented with a range of inputs and its outputs are used for the similarity comparison. Neurons with similar outputs are arranged so that they occupy the same position in each parent network in order to minimise disruption when they are recombined. This similarity measure could for example be a threshold on the mean squared error as calculated between the two vectors of neuron outputs given the same inputs.

[5]Confirmed through personal communication.

The principle here is that "The greatest improvement gained from this operator over the other crossover operators should come at the beginning of a run before all members of a population start looking alike." [MD89]. This same hypothesis, that the problem is only going to cause disruption until the population has largely converged is later echoed in [FS08] although this is not tested in detail. From an intuitive perspective it would seem clear that as the population converges and individuals become more alike, the likelihood of disruption decreases simply because the magnitude (in terms of genes affected) of applying crossover is diminishing with the diversity. In this way, we can consider crossover to be a type of macromutation, the magnitude of which is defined by the current diversity of the population.

We now explore this idea of explicit (exact-weight) permutations vs. those which simply fulfil the same role in the network but may or may not have exactly the same weights.

3.3.1 Explicit Permutations vs. Similar Roles

In reviewing the literature on the Permutation Problem (under its various aliases) a problem which hampers discussion is that it is not always explicitly stated which kind of permutations the authors are referring to. The most 'obvious' kind of symmetry is found when the neurons in a particular layer are rearranged. There is then the more subtle symmetry which is introduced *in addition* to the previous symmetry, where more than one neuron (i.e., more than one set of weights) may fill a particular space in a network without affecting the network function significantly. The qualifier *significantly* is important here; while in some cases authors will be referring to cases where the network function is identical (perfect or exact-weight symmetry), there will be other cases where the requirement is only that the network function remain largely unchanged. As it is possible for the network function to change without network fitness being affected (if the fitness function is not particularly sensitive, e.g. a change in network function brings the network closer to its decision boundaries but does not cross them, resulting in the same classes being predicted for a classification problem), this may be the criteria under which two networks may be considered equivalent. In this case there will be distinct neurons which can fill the same space in the network due to their equivalence of roles (equivalent-role symmetry). In this section we identify the perceived view of the major works in the literature on the Permutation Problem, specifically whether the exact-weight or equivalent-role symmetry is considered. This division of the literature is summarised in Table 3.1.

Montana and Davis identify that the role a neuron plays in a network is dependent only on which layer it is in, not on where it is in that layer [MD89]. While this is identified as being a problem when recombining networks it is not explicitly named. The example given to illustrate this however only covers the symmetry found by exchanging the weights of different neurons in the same layer. As such while the term 'role' is used, it would appear that the authors

Paper	Explicit Permutations	Equiv. Roles	Unspecified
[MD89]		✓?[a]	
[WSB90]		✓?	
[Rad90]		✓	
[BMS90]		✓	
[SWE92]	✓?		
[Han93]		✓[b]	
[ASP94]	✓		
[Whi95]		✓	
[YL97]	✓?		
[HNI04]	✓		
[FS08]		✓[c]	
[DHAI08]			✓

[a] In this work, both kinds of problem are described but not differentiated explicitly. As such it is difficult to determine which interpretation the authors intended. It depends on the interpretation of "identical sets of connections"; this could be identical connection *patterns* or identical *weights* (a question mark indicates that it is not explicitly stated or otherwise unambiguously clear which column the tick should be in, though the ticked column is favoured).

[b] This is defined only for Radial Basis Functions (RBFs) which have local receptive fields which have different characteristics to typical sigmoid-based networks.

[c] Here both the explicit permutation perspective is presented along with Hancock's RBF example. It is therefore not obvious which interpretation of the problem that the authors ascribe to. Confirmed through personal communication.

Table 3.1: A list of investigations into the Permutation Problem ranging from 1989-2008, organised by their perspective on the problem. In this case it is possible to divide the majority of work into the group which appears to view the problem from an exact-weight permutation perspective, and from an equivalent role perspective. A question mark (?) indicates uncertainty for that work.

only considered exact-weight permutations. That said however, the method for determining the role of the neuron is based not on its parameterisation (the weights) but on its outputs given certain patterns, suggesting an equivalent role perspective on the problem on behalf of the authors.

A contemporary investigation into the application of GAs to NN optimisation is that of Whitley et al. [WSB90]. In it they coin the term *"Structural/Functional*

Mapping Problem" to describe what is essentially the exact-weight permutation problem. The problem is discussed in terms of a network which requires the mapping of three 'tasks' A, B and C. We then have two networks which each contain three neurons labelled 1, 2 and 3 which are said to be mapping one task each. One network encodes them in the order $\langle 1, 2, 3 \rangle$ while the other does so in the order $\langle 3, 1, 2 \rangle$.

While this presentation of the problem implies on some level the notion of roles and the fact that more than one neuron may fill this role (thus suggesting a same-role view of the Permutation Problem), Whitley et al. then go on to talk about hidden neurons with "identical" sets of connections: "In general, when two or more hidden units have identical sets of connections (to the input and output layer, for example) then the structural/functional mapping will be arbitrary: these nodes can swap their functionality without altering the functionality of the net as a whole" [WSB90]. This view arguably implies the more restrictive exact-weight perspective on the problem. The notion that the authors had a similar role perspective is somewhat contradicted by their reference to how the neuron replacement transformations of the Structural/Functional Mapping Problem occur "without altering" network functionality, rather than "without altering *significantly*", which would imply that neurons which mapped approximately the same function would also be considered in the symmetry. This is not stated explicitly however, and is therefore open to interpretation.

As the Permutation Problem can be considered from the perspective of wishing to avoid recombining significantly different solutions, any measures which ensure that we are searching around one solution only would effectively prevent the Permutation Problem from occurring. The empirical data of the investigation by Whitley et al. indicates that a small population size (in this case around 50 individuals) will typically contain only one solution [WSB90]. Their analysis "indicates that in these small populations only a single solution is being pursued by the algorithm. Therefore the problem of recombining disparate solutions does not arise.". It is further claimed that the performance of the algorithm comes from stochastic genetic hill-climbing as opposed to intelligent hyperplane sampling as the Schema Theorem might suggest [WSB90]. Rather than recombining short, high fitness schemata, a 'cloud' of points around the one solution is sampled, with the population moving iteratively towards higher fitness regions which have been discovered locally.

Whitley et al. propose training networks where the neurons of the hidden layer have different (rather than full or uniform) connectivity so that swapping of neuron weight positions does not result in an equivalent network. This however adds the problem of discovering a suitable architecture. In training networks for the 2-bit adder problem they find that the minimally-connected version is easier to train than the fully-connected architecture. They take this as evidence to support the hypothesis that such avoidance of uniform connectivity avoids the Permutation Problem, though as other confounding factors are not taken into account it is not possible to say whether this is the deciding factor here.

In the first instance, it would be necessary to demonstrate that in the case of the fully-connected architecture the Permutation Problem is occurring frequently enough to have a significant effect on the progress of the optimisation process.

In a contemporary work, Belew et al. refer to two ways in which networks can be symmetrical [BMS90]:

- Rearrangement of the hidden neurons (assuming full connectivity), as previously discussed.
- Given an odd transfer function, flipping the signs of the weights results in the same function.

Belew et al. then refer generally to the idea of correspondence among hidden neurons, effectively describing the concept of similar roles. It is clear in this case that it is not just the exact-weight case that is being considered here. They do however cite work which demonstrates that minor changes in the weights can have a large effect on fitness. As such it would appear that when referring to similar roles the intended interpretation is different weight sets that produce the same projection in the hidden space.

Belew et al. consider the recombination of two relatively high fitness parents which have discovered different solutions as well as the Permutation Problem. They consider this problem to be "less subtle but just as problematic" as that of the problem of isomorphic solutions which can be formed by permuting the hidden neurons. Regarding how problematic these issues are they consider the symmetry introduced by the Permutation Problem to be, "such a devastating and basic obstacle to the natural mapping of networks onto a GA string that we might consider ways of normalizing network solutions prior to crossover".

Regarding how to circumvent or avoid the permutations, they suggest that the fact that the input and output layers are 'anchored' (i.e., constant and non-arbitrary) might allow the identification of similarly-functioning hidden neurons. They then assert however that, "establishing a correspondence among hidden units of two three-layer networks that have been trained to solve the same problem appears to be computationally intractable, even when we assume that the only difference between the two networks' solutions is a permutation of the hidden units". Normalisation of the networks prior to recombination is therefore deemed infeasible, suggesting that recombination is not an effective operation to perform when optimising neural networks.

Belew et al. agree with Whitley et al. [WSB90] regarding the small population hypothesis, stating that, "If very small populations are used with the GA, there is not 'room' for multiple alternatives to develop.". They offer the hypothesis that, barring stochastic effects, the first discovered solution will come to dominate the population. The reality for any suitably difficult problem is likely to be different however as it is not usually the case that it can be said early on whether a solution has been discovered (rather, several individuals may have very similar fitness), and

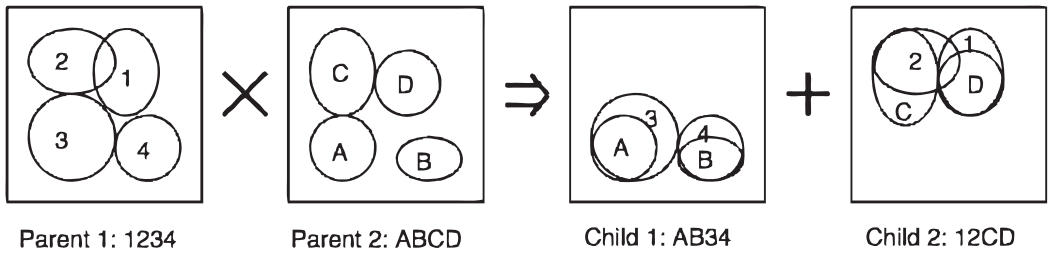


Figure 3.4: Recombination of two networks composed of neurons which define local receptive fields. Each parent could be said to have similar neurons (in that they are active for similar regions). These similar neurons appear at different positions in the network genotypes however, resulting in repetition of similar neurons in the child networks. Figure taken from [Han93].

the genetic drift introduced by the imperfect sampling of the selection mechanism need to be taken into consideration.

A contemporary investigation by Radcliffe [Rad90] expressed surprise that various recent successful results by Whitley et al. [WSB90] had been achieved without addressing the Permutation Problem. Radcliffe reiterates the small population argument of Belew et al., but further notes that in the same recent results by Whitley [WSB90] increasing the population size from 200 to 1000 produced an *improvement* in performance rather than the expected decrease, as increasing the population size should have increased the number of permutations.

Radcliffe, in his thesis, presents one of the more in-depth analyses of the problem during the period of time where discussion reached its peak [Rad90]. In it, he clearly refers to the swapping of *functionally homologous* parts of networks, thus referring quite generally to the symmetry introduced by neurons (or even subnetworks) which perform a similar role.

Hancock presents the problem not in terms of sigmoidal neurons but uses neurons with localised receptive fields (RF) to demonstrate the problem [Han93]. Looking at Figure 3.4 we see a 2D projection of the receptive fields of two networks. In a geometric sense the neurons of each network can be said to be similar, though defined in a different order in the genotype. Immediately we have the notion of neuron similarity, though this similarity may confuse the issue when we switch to a sigmoidal network (arguably the more common type of network used). Once we switch to a distributed representation of a problem, encoded using sigmoidal neurons, we lose the intuitively simple method of quantifying neuron similarity offered by the receptive fields (which might be defined in the 2D sense by a center, and a variance for each dimension, giving the spread and shape of its field). We see in figure 3.4 that the offspring networks over-sample parts of the input retina while other areas are under-sampled. The idea is that such recombination is unlikely to produce offspring with better fitness than their parents. How can this idea be transposed onto networks with distributed representations? In this case the possibility that the Permutation Problem may manifest due to neurons which

are similar is discussed but this discussion is not extended beyond neurons with a local receptive field.

Schaffer, Whitley and Eshelmann provide a survey of the state of the art of the application of Genetic Algorithms to the optimisation of Neural Networks, and in doing so cover the Permutation Problem in some detail, though they refer to is at the *Competing Conventions* problem [SWE92]. The problem is demonstrated through the permutation of the hidden neurons in a network with explicitly labelled connections. The examples therefore all imply the exact-weight perspective of the problem. However, when referring to networks which are permutations of each other they are described as mapping "nearly the same function". If the neurons have simply swapped position in the hidden layer then the function should be identical, not merely similar. It is possible that this is in reference to the result of crossovers that do not occur on a neuron boundary and therefore disrupt the mapping, though again this is somewhat unclear. Such a crossover will not necessarily produce a function that is "nearly the same" however; there is some doubt as to the correct interpretation of the given definition of the problem and therefore the view of the authors regarding which type of permutation is under consideration.

The term 'Competing Conventions' was introduced by Schaffer, Whitley and Eshelman [SWE92]. While this work cites both that of Radcliffe [Rad90] and Whitley et al. [WSB90], the new label is introduced to describe what has previously been labelled the Structural/Functional Mapping Problem and the Permutation Problem. As previously discussed, in this work Schaffer et al. explicitly refer to networks with identical topologies and (selections of) weights, stating that, "the only difference in the phenotypes is the switching of the two hidden nodes". While the definition given does not preclude there being other differences, the single example does not suggest others. The problem is presented as being primarily of the GA being required to overcome the "arbitrariness of the representation (the convention)", of which the GA can have no prior knowledge. The assumption here is that the GA will be required to work with differing/competing conventions frequently enough such that the search process is hindered. Schaffer et al. note that, "the number of competing conventions grows exponentially with respect to the number of hidden neurodes, since each permutation representing a different ordering of hidden neurodes represents a different convention". It is not clear however why the growth is characterised as being exponential as the number of permutations grows at a near-factorial rate which might be better described as being super-exponential. Responding to the work of Hancock [Han93], which found no evidence that the Permutation Problem is severe, it is suggested that the reason the problem did not manifest is due to the use of small populations and high selection pressure.

Hancock investigates the Permutation Problem and finds that 'solving' the problem decreases algorithm performance. This is taken as evidence of permutations actually being "beneficial" on some level [Han92]. While this wasn't stated

explicitly, increasing the population size would increase the number of permutations present simply through having more chances to appear in the population. While the increase in performance can't be attributed to permutations without first eliminating confounding factors, the idea that permutations may be beneficial some how was discussed in the work of the time. The possible benefit extended as far as adding them to the population, though Hancock notes that, "there is little mileage in adding in permutations" [Han92]. Hancock asserts that the Permutation Problem has two important facets. The first is a benefit due to the increased number of possible solutions; the other, a difficulty in bringing the disparate parts of the solution together. The symmetry does indeed reflect all maxima many times, but does this help or hinder the search process? It would appear that Hancock did not consider that every possible network is reflected: all maxima, near-maxima, all the way down to all minima. This is covered in more detail in Section 3.5.

A later investigation by Hancock again equates the Permutation Problem with the Competing Conventions Problem, and concludes by suggesting that neither is serious in practise [Han93]. This conclusion was based on the fact that 'solving' the problem by 'resolving' permutations using special recombination operators which sort "hidden unit definitions by overlap prior to crossover" fared more poorly than regular crossover. This leads to the contention that a regular GA is able to 'resolve' permutations without assistance. Implicit here is the assumption that there are indeed permutations in the population during the running of the GA; something that is not demonstrated or stated explicitly. Without this demonstration it is not possible to confidently draw conclusions based on the presence or lack thereof of permutations.

As it is the recombination of permutations that is theorised to cause the decrease in fitness in general, investigations into the Permutation Problem invariably discuss the efficacy of crossover operators in Neuroevolutionary algorithms. The effectiveness of crossover cannot be said to depend solely on the Permutation Problem alone (it is also necessary to consider the kind of crossover performed, crossover rate, number of crossing points etc.). Hancock states that, "The extent of the permutation problem may be assessed by comparing the performance of GAs with and without crossover enabled". A problem with this statement is that the removal of crossover from an algorithm does more than simply prevent the recombination of permutations. As crossover is a variational operator, its removal alters the balance between selection and variation, exploration and exploitation, which is likely to affect performance irrespective of the presence of permutations. What must be done to test such a hypothesis is to first determine how often permutations are recombined, and then test the algorithm without crossover, modifying the mutation rate to match the previous level of variation as closely as possible (a method which could be used to do this is examined in further detail in Section 5.2).

A frequently cited study of the problems with applying crossover in Neuroevolution is that of Angeline et al. [ASP94]. This work explicitly refers to the

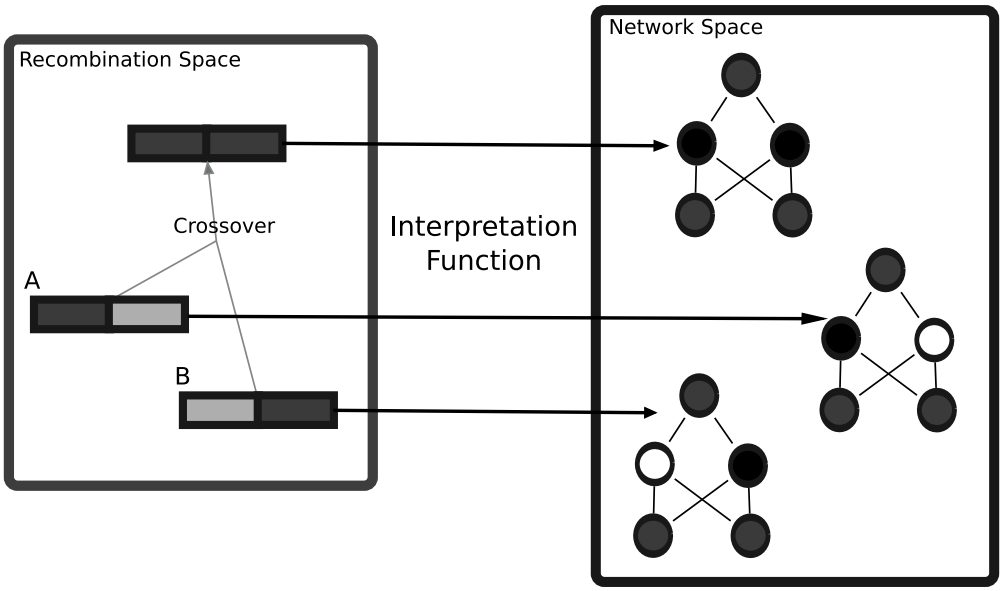


Figure 3.5: The problem of competing conventions. The genotypes A and B contain the same neurons and connections but in a different order. Crossover may create a network which contains duplicated structures, reducing its computational capability. Figure adapted from [ASP94].

Competing Conventions problem (citing Schaffer et al. [SWE92]) as being related to networks that share *both a common topology and weights*. Angeline et al. define the Permutation Problem[6] as referring to networks with common topology and weights, implying that the problem stems from the symmetry introduced by exact-weight permutations. The idea that multiple solutions exist for each problem is discussed but not in terms of the Permutation Problem or similar-role neurons. They refer instead to the kind of alternative solutions found by running the same algorithm multiple times, producing alternative solutions that are incompatible such as those in Figure 3.6. Due to this, the perspective in this work appears to be that of exact-weight permutations. Angeline et al. contend that GAs are simply not well-suited for evolving neural networks due to three forms of *deception* caused by the genotypic representation and its interpretation as a network.

The first form of deception is said to be the Competing Conventions problem, where the concern is over repeated elements. The example given to illustrate this can be seen in Figure 3.5. This marked a change in view of the Permutation Problem, as in this paper the possibility of permutations being caused by neurons which map same/similar projections in the hidden space is not alluded to. Rather, the negative effect of the Permutation Problem is presented as being caused by repeated components in offspring from parents which are *identical* except for the order of their hidden neurons. This view limits the scope of the problem considerably and invites the question, under what circumstances would we expect

[6]Though they use the alias *Competing Conventions*, introduced by Schaffer et al. [SWE92].

the parents selected for crossover to have neurons of identical weights arranged in different orders? This question is examined in Chapter 5.

The second form of deception is said to stem from the strong likelihood that there are multiple solutions in the search space, not caused by the Permutation Problem symmetry but simply by solutions with completely different weights, i.e. different distributed representations for the same solution. The recombination of such solutions is also expected to result in poor quality offspring as the solution in each case is *distributed* over some or all of the hidden neurons such that recombining part of one and another is likely to produce offspring which do not resemble their parents.

The final form of deception involves networks which differ topologically. In this work we consider fully-connected fixed-architecture networks as this is what is typically used to demonstrate the Permutation Problem. The problem of recombining networks with different architectures which may nevertheless be related in some way is investigated in [Sta04], where a custom crossover operator is presented which aims to address this issue (though *not* the Permutation Problem itself) and is to some extent successful, although the algorithm itself gains only slightly from its use [Sta04].

Angeline et al. [ASP94] present an algorithm which evolves both the structure and parameters of the network without using any form of recombination. Networks are evolved in the style of Evolutionary Programming [Fog94], where various custom mutation operators are defined. This algorithm, GNARL (GeNeralized Acquisition of Recurrent Links), is able to solve problems of reasonable complexity, leading the authors to conclude that crossover is simply not necessary and worth avoiding, stating, "the prospect of evolving connectionist networks with crossover appears limited in general". The authors then go on to recommend that operators should respect the distributed nature of the solutions encoded by connectionist networks, and that crossover operators in common use fail this test. While this appears to be a desirable characteristic to aim for when designing a reproductive operator, it does not necessarily mean that the plain crossover operator which has no such 'respect' is not of practical use.

In what is perhaps the most cited paper in the field of Neuroevolution, Yao exemplifies the Permutation Problem using networks which have identical topology and weights (thus aligning the work with that of Angeline et al. [ASP94]), where the hidden neurons have been swapped [Yao99]. The Competing Conventions Problem is identified here as an alias for the Permutation Problem. Regarding the severity of the problem and its relationship to crossover Yao states that, "The permutation problem makes crossover operator very inefficient and ineffective in producing good offspring". Yao then identifies the need for more research in order to further understanding of the impact of the Permutation Problem on the evolution of architectures; an area of investigation that has hitherto received little attention. As this work was a highly cited review paper we can assume that it was influential in the field of Neuroevolution. At this stage the evidence is overwhelmingly in favour of the Permutation Problem as a serious problem which

invalidates crossover as a suitable operator for evolving Neural Networks. This view has essentially become one of the fundamental or at least commonly-held beliefs of the field.

Some later work which reiterated the concerns of some of the earliest work that the Permutation Problem would be unlikely to occur but that nevertheless crossover was a poor operator was that of Stagge and Igel [SI00]. This investigation concerned the more general problem of network isomorphism in networks of any architecture. Stagge and Igel assert that if only canonically-labelled networks are stored in the population, the Competing Conventions / Permutation Problem is effectively solved. Regarding the expected decrease in disruption from solving the problem however, Stagge and Igel note that, "Experiments show that there are not so many isomorphic nets present in one generation as might be expected", suggesting that an enforced canonical representation would not necessarily cause a significant reduction in disruption caused by the recombination of isomorphic / permuted networks, or rather that there are few such networks in the population at any given time. No explicit numbers are given for the rate of occurrence of isomorphic networks though it is stated that, "...the probability of selecting two competing conventions of one net for recombination is rather small". Despite the "rather" small probability that two competing conventions be recombined, the conclusion of this work is that crossover is not a suitable operator for the evolution of Neural Networks.

3.3.2 Challenging the Severity of the Problem

Work which challenges the notion that the Permutation Problem is a serious concern is relatively rare; most such work was published in the early days of research into Neuroevolution [MD89, Han93]. Some major works in the literature are listed in Table 3.2, categorised by their use of particular naming conventions for the Permutation Problem and also their conclusion regarding the severity of the problem. Of the more recent work is that of Froese and Spier, which presents again the notion that a converged population will not be subject to the negative effects of the Permutation Problem [FS08]: as noted by Belew et al. if the population is small and selection pressure high, there will not be "room" for permutations in the population [BMS90].

Froese and Spier extend this idea to all populations, irrespective of size or selection pressure, suggesting that the nature of evolutionary search is for it to continue in a largely converged manner. The hypothesis put forward is that the convergence of the population is inevitable, and that progress will be made by the algorithm traversing nearly-neutral networks in the search space [FS08]. This view is somewhat in line with that of Whitley in a previous work, i.e. that the progress made by Neuroevolutionary algorithms is not due to efficient sampling of high-fitness schemata but is instead a process of genetic hill-climbing [WSB90].

The presented "Convergence Argument" does not preclude the presence of

Paper	Permutation Problem	Competing Conventions	Struc./Func. Mapping	Severe?
[MD89]				No[a]
[WSB90]			✓*	Yes
[Rad90]	✓*			Yes
[BMS90]	✓			No
[SWE92]		✓*		Yes
[Han93]	✓	✓	✓	No
[ASP94]		✓		Yes
[Whi95]	✓[b]	✓		Yes
[YL97]	✓			Yes
[Yao99]	✓			Yes
[HNI04]	✓			Yes
[FS08]	✓	✓	✓	No
[DHAI08]	✓			Yes

[a] The results of this paper show no difference between uniform crossover and a special permutation-avoiding crossover operator. While the paper attempts to solve this redundancy, it is not labelled as being a serious obstacle to the GA.

[b] Again it is unclear which interpretation is being offered in this paper, though the examples all refer to explicit permutations of blocks of weights; there is no explicit consideration of similar solutions.

Table 3.2: A list of investigations into the Permutation Problem ranging from 1989-2008, organised by the naming convention(s) used in each. Multiple ticks indicate the authors suggesting that either term is an acceptable synonym for the Permutation Problem. An asterisk (*) indicates when the term was first coined.

permutations, instead it is presumed that the initial population contains permutations, the elimination of which is a side-effect of the convergence of the population. The population is then expected to move through the search landscape as a largely converged 'cloud' of points. Regarding the severity of the problem Froese and Spier contend that that, "widespread concern with the permutation problem in the literature stems from a disregard of the generally converged nature of practical GA-based search". They then go on to note that past conclusions regarding the problem have been based on "theoretical" arguments (which we would call intuitive arguments or thought experiments) rather than studies into the practical ramifications of the problem.

With the exception of the discussion of Hancock's receptive fields example, this

work appears to be concerned only with permutations of a given network, rather than the additional symmetry introduced by also considering similar neurons. In this work it is suggested that the Permutation Problem is not a serious consideration in practise; this is supported by empirical work, leading to the conclusion that, "...it is unlikely that several distinct genetic permutations of the same phenotypic solution will be present in the population at the same time". As such this work appears to take the exact-weight perspective[7].

3.4 A Problem of Incompatible Representations

While the Permutation Problem appears severe from a theoretical standpoint, the few empirical studies carried out thus far have showed that in practice the problem may not be as severe as previous thought [MD89, Han93, FS08]. A problem which occurs more frequently and perhaps deserves more attention is shown in figure 3.6: we have the decision surface for two networks (a) and (b) which each map the XOR function but do so by partitioning the hidden space in markedly different ways (as illustrated both by the visual interpretation and the weights involved). The two genotypes are not permutations of each other, yet the offspring formed via an application of 1-point crossover are significantly different to their parents both in functionality and in fitness. Both offspring networks have lost a significant portion of the necessary decision surface and (c) has also suffered a loss of range.

The Permutation Problem could be considered to be a special case of the more general problem exemplified here which covers all cases where the recombination of two Neural Networks results in offspring of very low fitness. This problem, which we term the Incompatible Representations Problem (IRP), arises due to fundamental incompatibilities between the distributed representations of two (possibly equivalent) networks.

Given two arbitrary Neural Networks which we wish to use as parents in an evolutionary process we can characterise their relative (in)compatibility in terms of two categories: structural and parametric.

- *Structural* incompatibility arises when the topologies of the networks are different. This will commonly be characterised by the two networks having genotypes of different length. The NEAT algorithm overcomes this problem (to some degree) by globally identifying each gene in the genotypes in such a way that they can later be recombined with reduced disruption [SM04]. Other algorithms have avoided recombination [ASP94, Yao99] or employed fixed architectures.

- *Parametric* incompatibility occurs when given two networks of identical topology which either 1) have fundamentally different internal representations

[7]While this may be implied by the paper, the authors of this work are aware of the similar neuron perspective (confirmed through personal communication).

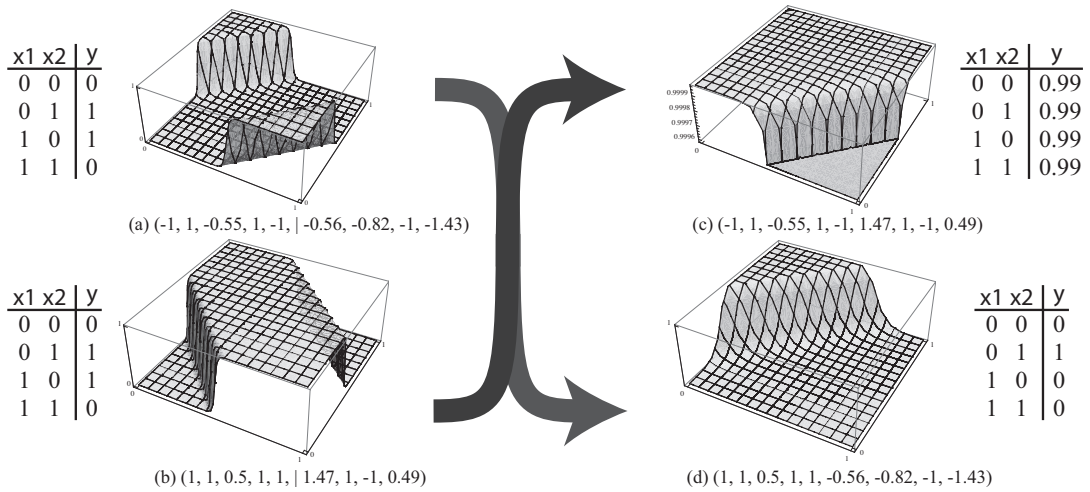


Figure 3.6: In this figure we have two networks (a) and (b) which map the XOR function. The solutions each have different weights (i.e. they were not formed through permutation of the hidden units). Using one-point crossover we can see the deleterious effect recombining these networks has, producing offspring networks (c) and (d). Network genotypes are of the form $(h_1^1, h_1^2, h_1^\theta, h_2^1, h_2^2, h_2^\theta, o^1, o^2, o^\theta)$. The transfer function of the hidden and output neurons is a steep sigmoid with variable bias.

of the hidden space (as in figure 3.6) or 2) are permutations of each other and therefore incompatible as per the Permutation Problem.

This more general problem of incompatibility between network representations is one which will occur more often than the Permutation Problem[8] but will have similar deleterious effects and so arguably warrants consideration when designing a Neuroevolutionary algorithm. This problem is not addressed by major algorithms in the literature such as NEAT [Sta04], ESP [GM03a], CoSyNE [GSM08] or EPNet [YL97] (though the problem is addressed on some level when EPNet is combined with Negative Correlation Learning [YI08].

3.5 Effect on the Search Space

The symmetry introduced by the invariability of a fully-connected feedforward network under permutation of its hidden neurons has often been cited as causing a (potentially) unimodal search landscape to become multimodal. Schaffer et al. consider networks which come from different peaks in the search space to be competing conventions, the recombination of which is to be avoided.

How does this symmetry affect the search space? It certainly causes it to be highly redundant: each point has anything from zero to $N_h! - 1$ equivalent

[8]Due to its relaxed constraints, i.e. the two networks need not be permutations of each other, it requires only that their internal representations be significantly different.

solutions. Given any point that we know the fitness of, we then know the fitness of up to $N_h! - 1$ other points in the search space. There are therefore a multitude of neutral networks in the search space. These networks appear to be of little practical value however, as while the symmetry allows the algorithm to move great distances in the genotype space for 'free' (i.e. fitness is known without using the objective function), each neighbourhood is the same: searching in one cloned neighbourhood versus another will not change the performance of the algorithm, suggesting that such a 'jump' operator has little utility. If by switching the order of the hidden neurons we could jump 'for free' to another area of the search space that had a different topology, we may be able to escape local minima by jumping to another area of the search space which contains a network of the same fitness (the symmetrical solution) but contains points of higher fitness close by. Unfortunately this is not the case; the search space is essentially repeated. One might see it as the same search space being repeated in a kaleidoscopic fashion. Despite this massive redundancy, previous work looking at the removal of this redundancy has suggested that such a removal can result in no change in performance [MD89], an increase in performance [DHAI08] or a decrease performance [Han93]. Hancock noted that the Permutation Problem provided, "a beneficial increase in the number of possible solutions" [Han92]. While the number of optima increases, so do the number of all other points: all near optima, local optima, right down to all minima. While we may think of this as a benefit (we have more solutions spread throughout the space), we simply have the same search landscape repeated on a massive scale. Perhaps the reason why this redundancy does not appear to harm the search overly negatively is because if the search space of unique networks is repeated 10 times, the full search space may have increased in size 10 times but the number of optima increased by the same factor, so we are still just as likely to pick the optima as we were before. Radcliffe has previously shown however that this is not necessarily the case: if we assume without loss of generality that there is a single optimum point characterised by a genotype $\langle a\ a\ a \rangle$ then in the redundant space there will still only be one optimum. Had the optimum been $\langle a\ b\ c \rangle$ then there would be $3! = 6$ optima in the redundant space. This is discussed in more detail in Chapter 6.

3.6 Chapter Summary

In this chapter we have identified and defined four types of permutation, indicating the types that we are interested in and their significance. We then presented a definition for the Permutation Problem which is compatible with the types of interest, demonstrating how in a fully-connected feedforward network the *order* of evaluation of contributions from the hidden neurons is not significant.

We then investigated the origin of the Permutation Problem, particularly where the terminology and its aliases have emerged from, and how these aliases have affected understanding of the problem. The interpretation of the problem in

the literature is then explored, particularly in terms of whether the problem is generally seen to be concerning exact-weight permutations, or exact/similar role permutations. As the choice of interpretation could affect the practical severity of the problem it is essential to understand the context of each work in the literature. The general-to-specific ordering of types of permutations was then discussed. This is another way, closely related to the four types of permutation, in which different works may differ in their implicit interpretation of the problem. We then discussed an alternative problem which may be confused with the Permutation Problem which we term the Incompatible Representations problem, where networks with similar functionality but completely different weight sets are recombined to produce offspring networks of low fitness.

Finally we discussed how the search space is affected by the Permutation Problem. This is explored in greater detail in Chapter 6.

Chapter 4

On the Probability of its Occurrence in the Initial Population

4.1 Introduction

In this chapter we investigate how often the Permutation Problem is likely to occur in a uniformly-initialised initial population from a probabilistic perspective. Previous work has suggested that the Permutation Problem occurs rarely [MD89, Rad90, FS08]. This previous work has however not given any explicit probabilities or empirical estimates on how often it is likely to occur. In the following sections we present exact probabilities and rates of occurrence for the problem and show that the problem does indeed occur rarely. This work forms the basis for our testing of the hypothesis that this rarity is due to there being very few or no permutations in the initial population, and the hypothesis that given a population containing few or no permutations, the typical genetic operators do not readily *produce* permutations. These issues are explored further in Chapter 5.

We begin by distinguishing between two possible types of permutation, *genotypic* permutations, where two genotype strings are permutations of each other, and *phenotypic* permutations, a subset of the genotypic permutations where whole neurons (i.e. blocks of weights) have been permuted as opposed to just single weights (more detail given in Section 3.2.1). The latter definition is the common interpretation of the Permutation Problem in the literature and the interpretation we use in this chapter. We present an equation for the probability that a pair of individuals drawn uniformly are genotypic or phenotypic permutations of each other. This is then generalised to populations of any size, allowing for the calculation of the expected number of permutations in the initial population.

Using this equation to examine even simple network spaces demonstrates the low probability of occurrence of either form of the problem in the initial generation with a range of representations. These results support those of the empirical investigation (Chapter 5), where the number of permutations in the population is determined through explicit counting and is shown to agree approximately with

the predicted value.

4.2 Genotypic Permutations

In this section we outline the method for calculating the probability that two randomly-drawn individuals will be genotypic permutations of each other. We then generalise this to arbitrary population sizes.

Given an allele alphabet α and string length l we can fully enumerate all $|\alpha|^l$ possible strings, where $|\alpha|$ is the cardinality of the alphabet. This space of strings can be decomposed into groups where each group contains all strings with a particular selection of alleles, in any order. There are $\left(\!\binom{|\alpha|}{l}\!\right)$ such groups, where $\left(\!\binom{n}{k}\!\right)$ is the number of ways to select subsets of size k from n items with repetition. This is the multinomial extension to the binomial coefficient where,

$$\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k} . \tag{4.1}$$

We are interested in the probability of picking two distinct strings from the same group. For example, the group which contains strings with one copy of allele one and three copies of allele two, or the group which contains strings with two copies of allele one and two copies of allele two and so on, for all possible selections of allele values. Once we pick one string, the number of possible permutations that can be formed from it determines the size of the group and so the subset of strings which, if chosen, would form a pair which are permutations of each other.

For two allele values and genotype length l we can calculate the probability that a pair of individuals drawn at random are permutations as

$$P(2,l) = \sum_{i=0}^{l} \frac{\left(\frac{l!}{i!(l-i)!}\right)}{2^l} \frac{\left(\frac{l!}{i!(l-i)!}\right) - 1}{2^l}.$$

For three allele values this would be calculated as

$$P(3,l) = \sum_{i=0}^{l} \sum_{j=0}^{l-i} \frac{\left(\frac{l!}{i!j!(l-i-j)!}\right)}{3^l} \frac{\left(\frac{l!}{i!j!(l-i-j)!}\right) - 1}{3^l}.$$

The equation generalises to alphabets of length n as

$$P(n,l) = \sum_{i_1=0}^{l} \sum_{i_2=0}^{l-i_1} \cdots \sum_{i_{n-1}=0}^{l-\left(\sum^{n-2} i\right)} \frac{\left(\frac{l!}{\left(\prod_{j=1}^{n-1} i_j!\right)\left(l-\sum_{k=1}^{n-1} i_k\right)!}\right)}{n^l} \frac{\left(\frac{l!}{\left(\prod_{j=1}^{n-1} i_j!\right)\left(l-\sum_{k=1}^{n-1} i_k\right)!}\right) - 1}{n^l}. \tag{4.2}$$

Figure 4.1 shows this probability calculated for a range of alphabet cardinalities and string lengths.

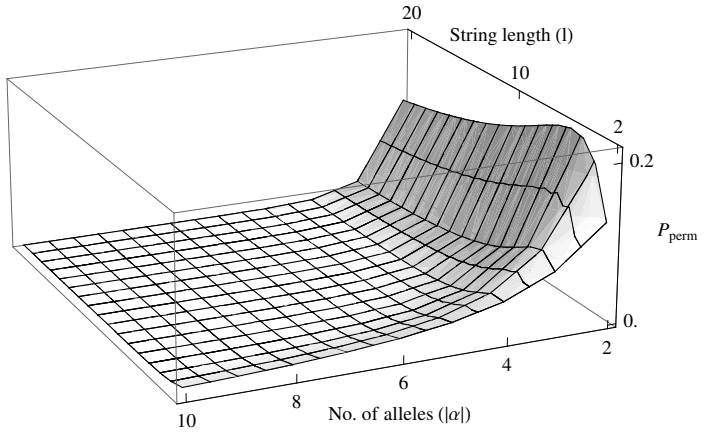


Figure 4.1: The probability P_{perm} that a pair of individuals (drawn uniformly from a solution space defined by the number of alleles $|\alpha|$ and string length l) are genotypic permutations of each other.

To illustrate the strategy behind this arrangement we can expand the equation for the case of a binary alphabet and string length 4:

$$\begin{aligned} P(2,4) &= \sum_{i=0}^{4} \frac{\left(\frac{4!}{i!(4-i)!}\right)}{2^4} \frac{\left(\frac{4!}{i!(4-i)!}\right) - 1}{2^4} \\ &= \left(\frac{\left(\frac{4!}{0!4!}\right)}{2^4} \frac{\left(\frac{4!}{0!4!}\right) - 1}{2^4} \right) + \left(\frac{\left(\frac{4!}{1!3!}\right)}{2^4} \frac{\left(\frac{4!}{1!3!}\right) - 1}{2^4} \right) + \\ &\quad \left(\frac{\left(\frac{4!}{2!2!}\right)}{2^4} \frac{\left(\frac{4!}{2!2!}\right) - 1}{2^4} \right) + \left(\frac{\left(\frac{4!}{3!1!}\right)}{2^4} \frac{\left(\frac{4!}{3!1!}\right) - 1}{2^4} \right) + \\ &\quad \left(\frac{\left(\frac{4!}{4!0!}\right)}{2^4} \frac{\left(\frac{4!}{4!0!}\right) - 1}{2^4} \right). \end{aligned}$$

Here we can see that the probabilities of drawing two distinct strings from each possible group, for example the group with one copy of allele one and three copies of allele two (of which there are $\frac{4!}{1!3!} = \frac{24}{1 \cdot 6} = 4$) are considered, and the union of these disjoint events is taken which gives us the probability that we select two distinct strings from any one of the groups.

If we take X to be a random variable representing the probability of encountering a pair of permutations given a particular permutation space, then the value of interest is its expectation which tells us how many permutations we can expect to see on average when drawing pairs at random. Given then the probability that a pair of individuals drawn at random are permutations of each other, we can calculate the expected number of permutations $E(X)$, which for a pair may be

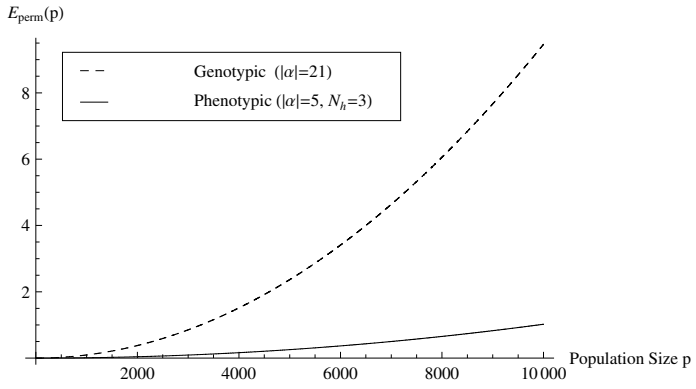


Figure 4.2: Expected number of permutations $E_{\text{perm}}(p)$ for a range of population sizes p, for 1) a genotype space of 21 alleles with string length 9 (genotypic permutations) and 2) for a phenotype space of a 3-input, 1-output, 3-hidden neuron neural network (phenotypic permutations) with a weight space of size 5. These network representations are close to that of the empirical experiments of Section 5.2.

either zero or one:

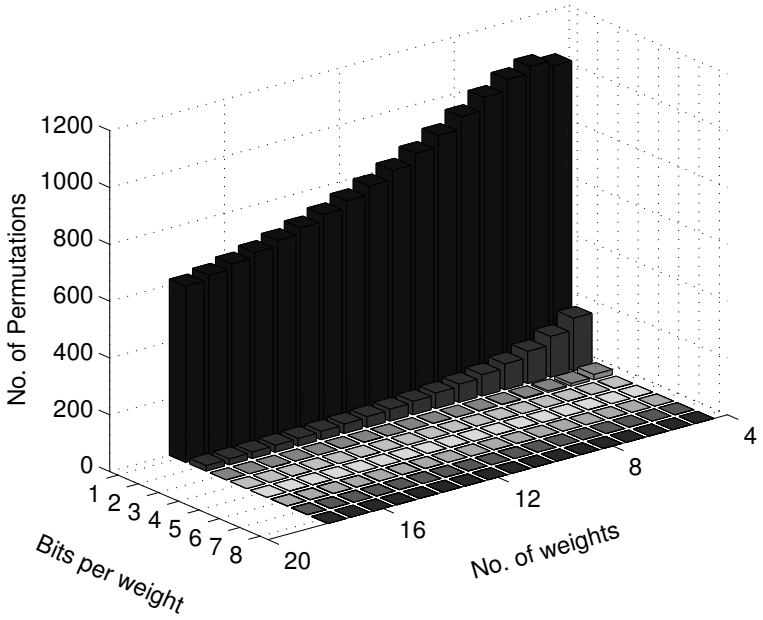
$$E_{\text{perm}}(X) = (0)(1 - P(n,l)) + (1)(P(n,l)) = P(n,l).$$

We then wish to calculate the expected value for a population of size p. Since the probability that one pair are permutations of each other is independent of the probability for any other distinct pair, we can achieve this by taking the probability for a single pair and multiplying by the number of possible pairings of individuals in the population. Thus for a population of size p we form the union of all $\binom{p}{2}$ events where each pair are permutations of each other, giving an expected number of permutation pairs of $\binom{p}{2}P(n,l)$. This is shown in Figure 4.2 where the expected number of genotypic permutations is calculated for all population sizes from 2 to 10,000 (dashed line) for the representation used in the empirical investigation detailed in Section 5.2. As we can see in Figure 4.2, the predicted number of permutations with this representation is very low even for very large populations (e.g. 10,000 individuals).

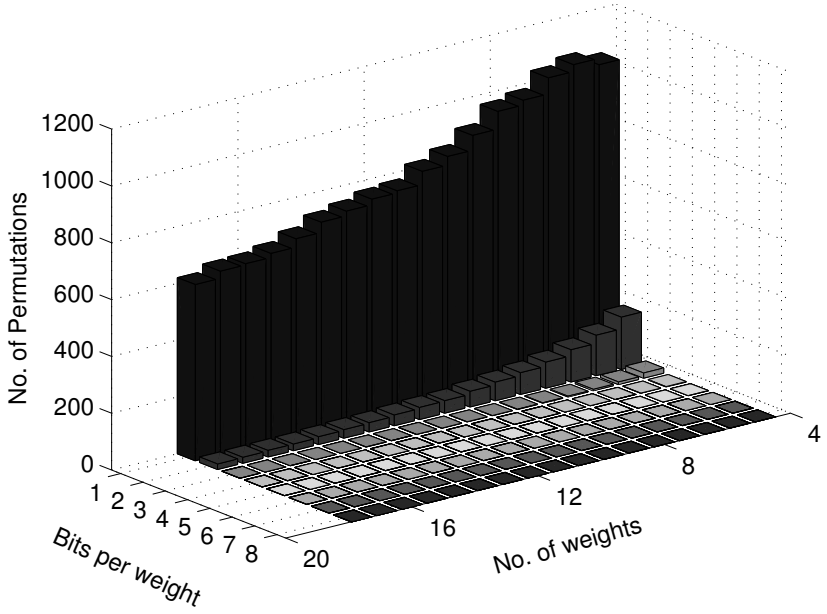
4.2.1 Empirical Validation

We validate the equation empirically by predicting the number of permuted pairs in an initial population of size 100 with a range of alphabet cardinalities and string lengths. The predicted and actual values as averaged over 100 trials are shown in Figure 4.3.

While the actual permutation counts do not match the predicted values we expect that averaging over more than 100 trials will produce values closer to those



(a) Predicted values.



(b) Actual values averaged over 100 trials.

Figure 4.3: The expected number of genotypic permutations in the initial population for a range of weight space granularities and network sizes, with population size 100.

predicted.

4.3 Phenotypic Permutations

We now calculate the probability of drawing two networks which are phenotypic permutations of each other. This is the kind of permutation that we are interested in as unlike the genotypic permutations, phenotypic permutations always result in phenotypically-equivalent networks and therefore are instances of the Permutation Problem.

Calculating the probability of phenotypic permutations can be achieved using the same mechanism as for the genotypic permutations, simply by reinterpreting the parameters of Equation 4.2. Now, instead of number of alleles we are considering number of possible neurons, and instead of string length we have number of hidden neurons. The difficulty here is that due to the large number of possible neurons for even modest genotype spaces, calculating the probabilities is computationally infeasible with the method presented. So, for the phenotypic permutations example we initially consider a toy genotype space of only 5 values, $\alpha = \{-1.0, -0.5, 0, 0.5, 1.0\}$ and two hidden neurons. With such a coarse-grained genotype space we might expect that the probability of encountering individuals which are phenotypic permutations of each other would be high. As we will show however, even for this unrealistically coarse space the probability of drawing a population containing an appreciable number of permuted individuals is low.

We now present the method for the calculation of the parameters to use in Equation 4.2 to calculate the expected number of phenotypic permutations in an initial population. Given an alphabet α we can count the number of unique hidden neurons ϕ in a typical feed-forward single-layer fully-connected network space as

$$\phi = |\alpha|^{N_i+N_o}, \tag{4.3}$$

where N_i and N_o are the number of inputs and outputs respectively. We can arrange these neurons into ϕ^{N_h} networks of N_h hidden neurons[1]. Given, for example, a network of three inputs, one output and $|\alpha| = 5$ we would then have $\phi = |\alpha|^{N_i+N_o} = 5^4 = 625$ possible neurons. We can now calculate the probability that two randomly drawn networks are phenotypic permutations of each other using Equation 4.2, giving $P(625, 2) = 2.6 \times 10^{-6}$. The expected number of phenotypic permutations for a range of population sizes using this representation is shown in Figure 4.2 (solid line).

4.3.1 Discussion

Figure 4.1 shows the expected number of genotypic permutations for a uniformly-drawn pair of individuals, for a range of alphabet cardinalities and string lengths. Here we can see that the expectation is high for binary alphabets but drops off

[1]However, due to the invariance of a network's function under permutation of its hidden neurons, there are only $\left(\!\binom{\phi}{N_h}\!\right)$ unique networks.

rapidly for larger alphabets, which tells us that genotypic permutations are only common with binary alphabets. Even with unrealistically small alphabets, such as $\{-1.0, -0.5, 0, 0.5, 1.0\}$ which has cardinality 5, the expectation for genotypic permutations is low for most string lengths. For more realistic alphabets this expectation can be shown to decrease rapidly, suggesting that this kind of permutation is a relatively rare occurrence, at least in the initial population. Extending this to population sizes greater than two, we can see in Figure 4.2 that even for population sizes up to 10,000 the expected number of pairs of individuals which are permutations of each other is very low. For realistic network spaces this expectation will be considerably lower.

Compared to that of phenotypic permutations, the expected number of genotypic permutations in the initial population is higher, though it is debatable as to whether this is of any real concern as genotypic permutations are not frequently cited as being an obstacle in the application of Genetic Algorithms. Given that genotypic permutations may occur in any representation this would potentially have far-reaching implications, particularly for binary representations. The recombination of two individuals which are permutations of each other will result in a higher-than-average likelihood of repeated allele values in the offspring. While this may result in an increase in the production of poor offspring due to repetition of genes [Sta04], population-based search techniques such as Evolutionary Algorithms are largely unaffected by such infrequent events in the general case.

Figure 4.2 shows the expected number of genotypic permutations, and so an upper bound for the expected number of phenotypic permutations for the representation used in the empirical investigation (Section 5.2), where a population size of 100 was used. It can be seen here that the expected number of permutations is low in both cases: $E_{\text{perm}}(100) = 9.4 \times 10^{-4}$ genotypic permutations; this is in accordance with the count of zero permutations in the actual evolutionary runs.

4.3.2 Calculating for high values of n

For high values of n (i.e. number of neurons in the phenotypic case or weights in the genotypic case) the evaluation of Equation 4.2 quickly becomes infeasible. The number of iterations which must be made is proportional to $\left(\!\binom{n}{l}\!\right)$. Figure 4.4 shows the growth of this function as n and l increase; the number of multisets to consider when n exceeds 15 quickly becomes prohibitive. Calculating the probability of the Permutation Problem in terms of phenotypic permutations requires evaluating Equation 4.2 with values of n far in excess of the feasible limit with this method. As n now represents the number of possible neurons and l the number of hidden neurons required, to be limited to 15 neurons would preclude the investigation of network spaces corresponding to real problems.

In order to determine a more efficient method of calculation we re-examine the strategy behind the multiset-based approach. The total number of iterations over all summations in Equation 4.2 will always equal the number of possible multisets $\left(\!\binom{n}{l}\!\right)$; we wish to reduce this number to a feasible level. If we examine the sequence

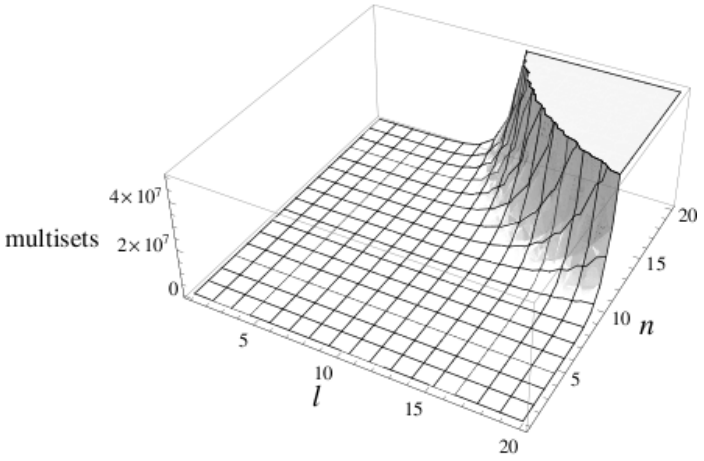


Figure 4.4: The number of ways of picking multisets for a representation space defined by the alphabet α and string length l. Given the rapid growth of this function computing the probability of the Permutation Problem for the case where $|\alpha|$ and l are both greater than or equal to 20 becomes infeasible.

S produced by evaluating the part of Equation 4.2 which counts the number of permutations a particular multiset has we will be able to discern a pattern which will allow us to cut down the number of operations required. The sequence $S(n, l)$ is defined as

$$S(n,l) = \sum_{i_1=0}^{l} \sum_{i_2=0}^{l-i_1} \cdots \sum_{i_{n-1}=0}^{l-\left(\sum^{n-2} i\right)} \left(\frac{l!}{\left(\prod_{j=1}^{n-1} i_j!\right)\left(l - \sum_{k=1}^{n-1} i_k\right)!} \right), \tag{4.4}$$

where n is the number of possible values we can choose from and l is the number of positions to fill, as before. For $n = 2$ this simplifies to

$$S(2,l) = \sum_{i=0}^{l} \frac{l!}{i!(l-i)!} \tag{4.5}$$

Some examples of this sequence where we have two possible neurons ($n = 2$) and from zero to four spaces for hidden neurons ($0 \leq l \leq 4$) are as follows:

$$\begin{aligned} S(2,0) &= 1 \\ S(2,1) &= 1,1 \\ S(2,2) &= 1,2,1 \\ S(2,3) &= 1,3,3,1 \\ S(2,4) &= 1,4,6,4,1. \end{aligned}$$

The sequence $S(2, l)$ corresponds to the $(l+1)^{\text{th}}$ row of Pascal's Triangle. We note

that there exists a symmetry in the sequence such that only half of the values in each row need to be calculated in order to infer the whole. We now examine the sequence $S(3, l)$ which is defined as

$$S(3,l) = \sum_{i=o}^{l} \sum_{j=0}^{l-i} \frac{l!}{l!i!(l-i-j)!}. \tag{4.6}$$

Now the sequence for three possible neurons ($n = 3$) and from zero to four spaces for hidden neurons ($0 \leq l \leq 4$) is as follows:

$$\begin{aligned} S(3,0) &= 1 \\ S(3,1) &= 1,1,1 \\ S(3,2) &= 1,2,1,2,2,1 \\ S(3,3) &= 1,3,3,1,3,6,3,3,3,1 \\ S(3,4) &= 1,4,6,4,1,4,12,12,4,6,12,6,4,4,1. \end{aligned}$$

Evaluating this sequence gives us the $l+1^{\text{th}}$ slice of the n-dimensional generalisation of Pascal's Triangle. For $n = 3$ it calculates the lth layer of Pascal's Pyramid[2], the generalisation of Pascal's Triangle in three dimensions. The first six layers of Pascal's Pyramid are visualised in Figure 4.5.

The length of the sequence $|S(n, l)|$ is the number of multisets which can be composed from n items being placed in l spaces, i.e.

$$|S(n,l)| = \left(\!\!\binom{n}{l}\!\!\right) = \binom{n+k-1}{k}. \tag{4.7}$$

The interpretation of the sequence is that the values represent the number of permutations that each multiset has (taking into account duplicate values). Therefore, the sum of the values of the sequence will be the total number of strings of length l that can be formed from n items, or

$$\sum_{s \in S(n,l)} s = n^l. \tag{4.8}$$

We have already suggested that there is some regularity in Pascal's Triangle and Pyramid. The n-dimensional extension of Pascal's Triangle is called Pascal's Simplex. Using the sequence $S(n, l)$ we are able to construct any of those n-dimensional simplices, and in doing so calculate the necessary values for our calculation of the probability of the Permutation Problem occurring. The number of points in these simplices is prohibitively large beyond the 15-dimensional case however; calculating each individually will not be computationally feasible for realistic spaces. In order to reduce the computation required we exploit a regularity in the sequence based on the theory of *integer partitions*. Briefly, the partitions of

[2]This is a tetrahedral pyramid, as it is composed of layers of triangles.

```
(1)  (2)     (3)          (4)              (5)                  (6)

 1    1       1            1                1                    1

    1   1   2   2        3   3            4   4                5   5

          1   2   1    3   6   3        6  12   6           10  20  10

                     1  3   3   1    4  12   12  4       10  30  30  10

                                  1   4   6   4   1    5  20  30  20  5

                                                      1   5  10  10  5   1
```

```
S(3,0) = {1}
S(3,1) = {1,1,1}
S(3,2) = {1,2,1,2,2,1}
S(3,3) = {1,3,3,1,3,6,3,3,3,1}
S(3,4) = {1,4,6,4,1,4,12,12,4,6,12,6,4,4,1}
S(3,5) = {1,5,10,10,5,1,5,20,30,20,5,10,30,30,10,10,20,10,5,5,1}
```

Figure 4.5: The first six levels of Pascal's Pyramid. The values in each layer correspond to the sequence of numbers of permutations that each multiset has when interpreted as a genotypic string for $n = |\alpha| = 3$ and $1 \leq l \leq 6$. The sequences can be found by reading off the diagonals of each layer.

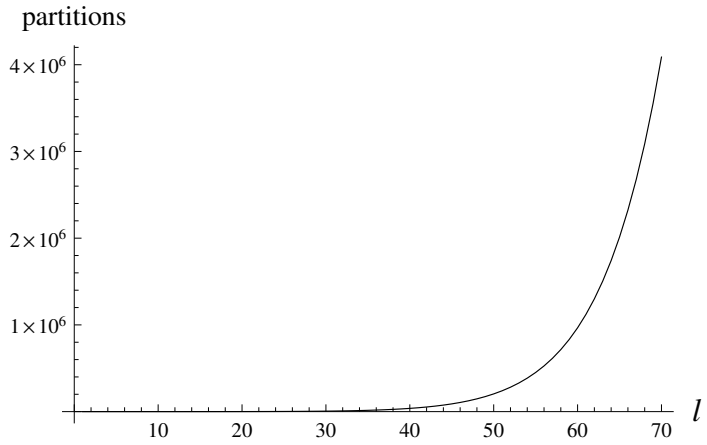


Figure 4.6: The number of partitions of the integer l. Calculating the probability of the Permutation Problem occurring using an approach based on partitions rather than multisets is considerably more efficient as the number of partitions is independent of $|\alpha|$.

a positive integer l are all the ways of writing the number as a sum of positive integers (including l itself). Thus the partitions of 4 are

$$\begin{gathered} 4, \\ 3+1, \\ 2+2, \\ 2+1+1, \\ 1+1+1+1\ . \end{gathered}$$

As the number of partitions depends only on the number of hidden neurons or spaces to fill l and not the number of neurons available n, a method based on partitions can scale to any representation granularity (number of neurons) without increasing the number of operations required significantly. The growth of the number of partitions of the positive integers is shown in Figure 4.6.

The regularity which we will exploit is that only certain numbers will appear in the sequence for a given layer or dimension of the simplex. Looking again at Figure 4.5 we can determine visually what those numbers are and how many times each value occurs. Taking a simpler case as an example, we calculate the sequence $S(2, 4)$:

$$\begin{aligned}
S(2,4) &= \sum_{i=0}^{4} \frac{4!}{i!(4-i)!} \\
&= \frac{4!}{0!(4-0)!} + \frac{4!}{1!(4-1)!} + \frac{4!}{2!(4-2)!} + \\
&\quad \frac{4!}{3!(4-3)!} + \frac{4!}{4!(4-4)!} \\
&= \frac{24}{1(24)} + \frac{24}{1(6)} + \frac{24}{2(2)} + \\
&\quad \frac{24}{6(1)} + \frac{24}{24(1)} \\
&= \frac{24}{24} + \frac{24}{6} + \frac{24}{4} + \frac{24}{6} + \frac{24}{24} \\
&= 1 + 4 + 6 + 4 + 1 \\
&= 2(1) + 2(4) + 1(6).
\end{aligned}$$

Here only the numbers $\{1, 4, 6\}$ appear in the sequence. As we know that we will have $\left(\!\binom{2}{4}\!\right) = 5$ terms we also know that there will necessarily be

$$\left(\!\binom{2}{4}\!\right) - |\{1, 4, 6\}| = 5 - 3 = 2$$

repetitions of terms in the sequence. If we consider the sequence to be a multiset we can say we are interested in determining first the unique members of the multiset and then their respective multiplicities.

If we know the numbers that should appear in this sequence and their multiplicities then we have the information we require in order to reduce the number of terms that we need to explicitly enumerate by grouping individual terms with their respective frequency of occurrence as their coefficient. When we apply this principle to the calculation of the probability of the Permutation Problem for $n = 2$ and $l = 4$ we get

$$
\begin{aligned}
P(2,4) &= \sum_{i=0}^{4} \frac{\left(\frac{4!}{i!(4-i)!}\right)}{2^4} \frac{\left(\frac{4!}{i!(4-i)!}\right)-1}{2^4} \\
&= \left(\frac{\left(\frac{4!}{0!4!}\right)}{2^4} \frac{\left(\frac{4!}{0!4!}\right)-1}{2^4}\right) + \left(\frac{\left(\frac{4!}{1!3!}\right)}{2^4} \frac{\left(\frac{4!}{1!3!}\right)-1}{2^4}\right) + \\
&\quad \left(\frac{\left(\frac{4!}{2!2!}\right)}{2^4} \frac{\left(\frac{4!}{2!2!}\right)-1}{2^4}\right) + \left(\frac{\left(\frac{4!}{3!1!}\right)}{2^4} \frac{\left(\frac{4!}{3!1!}\right)-1}{2^4}\right) + \\
&\quad \left(\frac{\left(\frac{4!}{4!0!}\right)}{2^4} \frac{\left(\frac{4!}{4!0!}\right)-1}{2^4}\right) \\
&= \left(\frac{1}{16}\frac{1-1}{16}\right) + \left(\frac{4}{16}\frac{4-1}{16}\right) + \left(\frac{6}{16}\frac{6-1}{16}\right) + \\
&\quad \left(\frac{4}{16}\frac{4-1}{16}\right) + \left(\frac{1}{16}\frac{1-1}{16}\right) \\
&= \left(\frac{1}{16}\frac{0}{16}\right) + \left(\frac{4}{16}\frac{3}{16}\right) + \left(\frac{6}{16}\frac{5}{16}\right) + \\
&\quad \left(\frac{4}{16}\frac{3}{16}\right) + \left(\frac{1}{16}\frac{0}{16}\right) \\
&= 0 + \frac{12}{256} + \frac{30}{256} + \frac{12}{256} + 0 \\
&= 2(0) + 2\left(\frac{12}{256}\right) + \frac{30}{256}.
\end{aligned}
$$

By determining the probabilities and their frequencies we reduce the number of (probabilistically) identical cases considered. This saving is visualised in Figure 4.7 which shows the percentage of operations required to calculate the probability using a partition-based approach compared to the multiset approach. For a given value of n there will be one or more values of l where the number of partitions will be greater than the number of multisets. For these cases the multiset approach would be more efficient, though as n increases this value of l also increases considerably. Given that we will be dealing with very large values for n the partition-based approach will always be used.

In this particular case we wish to determine first the possible number of permutations that any given string composed of 4 values out of an alphabet of size 2 (in this case $\{1, 4, 6\}$) and their frequencies $\{2(1), 2(4), 1(6)\}$. Given these pairings we then have enough information to calculate the probability of the Permutation Problem occurring, while evaluating only a fraction of the number of operations as compared to the initial approach.

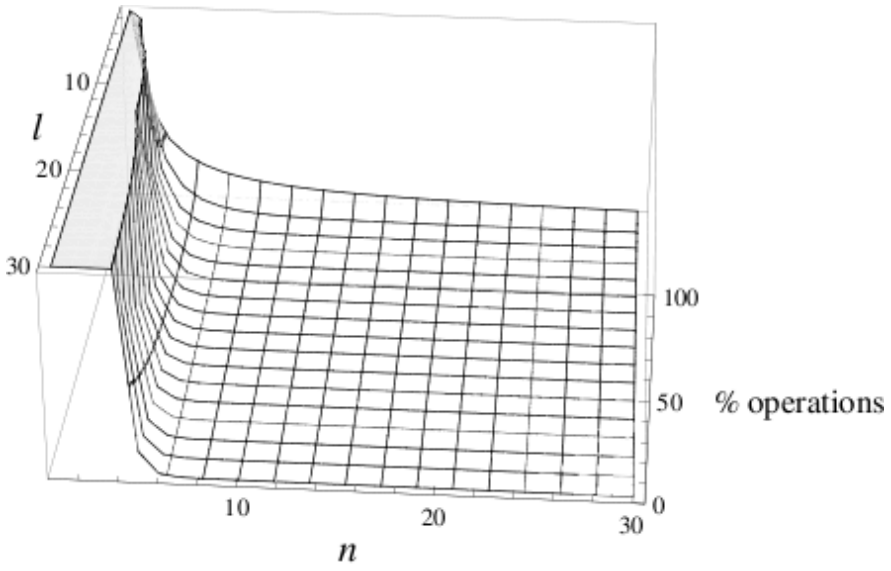


Figure 4.7: Percentage of operations required to calculate Permutation Problem probability using partition-based method versus a multiset-based method. The saving in operation count is considerable for all but the lowest settings of n, which represent unrealistically-small problem spaces.

4.3.3 Partition-based Method

In this section we present the method for calculating the probability of the Permutation Problem occurring using the more efficient partition-based method.

We first need to calculate which numbers will appear in our sequence of multisets ($\{1, 4, 6\}$ in our previous example). In this example we have $N_h = 4$ hidden neurons. We begin by calculating the set of partitions D_4 of the integer 4:

$$D_4 = \{\{1, 1, 1, 1\}, \{2, 1, 1\}, \{2, 2\}, \{3, 1\}, \{4\}\}\,.$$

These are the cardinality patterns of sets that we can form with up to 4 values. For example from right to left we have 'four of a kind', 'three of a kind plus any other', 'two pair' and so on, to borrow from card-playing terminology. No matter how many values we have to draw from, any string drawn will follow one of these patterns (for a draw of four cards). These sets effectively specify different cardinality patterns for multisets. From these patterns we can calculate how many permutations each has; we then know the size of the group of permutations that each multiset type corresponds to. We denote A_l to be the ordered set of permutations that can be formed if we interpret each element of the multiset (for $N_h = l$) as being a distinct value in the string, where its value gives the number of times the values appears in the string. In our example,

$$\begin{aligned} A_4 &= \left\{ \frac{4!}{1!1!1!1!}, \frac{4!}{2!1!1!}, \frac{4!}{2!2!}, \frac{4!}{3!1!}, \frac{4!}{4!}, \right\} \\ &= \left\{ \frac{24}{1}, \frac{24}{2}, \frac{24}{4}, \frac{24}{6}, \frac{24}{24}, \right\} \\ &= \{24, 12, 6, 4, 1\}. \end{aligned}$$

This can be expressed generally as

$$A_l = \left\{ a : a = \frac{l!}{\prod_{d \in D_l} d!} \right\}. \tag{4.9}$$

Given any number of hidden neurons we can calculate the possible number of permutations each possible string may have. In our example, any given string may have one of $\{24, 12, 6, 4, 1\}$ permutations. We now wish to calculate how many of each we have, for a given value of n. In our previous example $n = 4$ and we know that we have zero with 24 permutations, zero with 12, one with 6, two with 4 and two with 1.

We begin with the case of 24 permutations. For the number of strings with 24 permutations the corresponding member of D_4 is $\{4\}$ so we are looking to select four distinct values to occupy one space of the string each. The number of strings in our space with 24 permutations must therefore be $\binom{2}{4} = 0$ as we only have two possible values; this is fewer than the number of spaces, meaning we have no strings with the maximum number of permutations. The same is true for strings with 12 permutations, $(n < l)$ so again we have zero permutations as $\binom{3}{4} = 0$.

For the number of strings with 6 permutations the corresponding member of D_4 is $\{2, 2\}$ so we are looking to select two values to occupy two spaces of our string each. We therefore have $\binom{2}{2} = 1$ permutations.

For the number of strings with 4 permutations the corresponding member of D_4 is $\{3, 1\}$ so we need two values: one will occupy only one position, the other the remaining 3 positions. We therefore have $\binom{2}{1} \cdot \binom{1}{1} = 2$ permutations. We select from only one value in the second term as we have already selected one in the first term.

The process for the number of strings with 1 permutation follows in the same pattern. This pattern is shown more clearly in Table 4.1, and Table 4.2 which shows the same calculation for the case where we have 3 values (or 3 neurons) with which to fill the available 4 positions.

The algorithm for calculating the probability in this manner is given in Algorithm 3. Put more informally, we iterate through all partitions of the integer l (the number of spaces, or hidden neurons). For each partition we count how many times a '1' occurs, how many times '2' occurs and so on, to produce a table as in Table 4.2. We then count the number of ways we can produce strings that match the pattern; this is the number of strings in the space that produce a particular

	1	**2**	**3**	**4**		
`1111`	4	0	0	0	$\binom{2}{4}$	$= 0$
`112`	2	1	0	0	$\binom{2}{2} \cdot \binom{0}{1}$	$= 0$
`22`	0	2	0	0	$\binom{2}{0} \cdot \binom{2}{2}$	$= 1$
`13`	1	0	1	0	$\binom{2}{1} \cdot \binom{1}{0} \cdot \binom{1}{1}$	$= 2$
`4`	0	0	0	1	$\binom{2}{0} \cdot \binom{2}{0} \cdot \binom{2}{0} \cdot \binom{2}{1}$	$= 2$

Table 4.1: The table for calculating a simple example of the partition-based approach with $n = 2$ and $l = 4$.

	1	**2**	**3**	**4**		
`1111`	4	0	0	0	$\binom{3}{4}$	$= 0$
`112`	2	1	0	0	$\binom{3}{2} \cdot \binom{1}{1}$	$= 3$
`22`	0	2	0	0	$\binom{3}{0} \cdot \binom{3}{2}$	$= 3$
`13`	1	0	1	0	$\binom{3}{1} \cdot \binom{2}{0} \cdot \binom{2}{1}$	$= 6$
`4`	0	0	0	1	$\binom{3}{0} \cdot \binom{3}{0} \cdot \binom{3}{0} \cdot \binom{3}{1}$	$= 3$

Table 4.2: The table for calculating a simple example of the partition-based approach with $n = 3$ and $l = 4$.

number of permutations. We therefore have both the number of permutations and their frequencies which together provide all the information necessary to calculate the full probability.

4.3.4 Empirical Validation

Using the faster calculation method we are now able to validate empirically the case for phenotypic permutations. With the initial method, the computational complexity of which depended both on n and l, we were unable to calculate these probabilities.

In order to empirically validate the theoretical results we generate 100 uniformly-initialised populations of individuals using a range of network representations and compare the actual number of permutations with the expected numbers.

The simplest fully-connected feed-forward network we can construct which can exhibit the permutation problem has one input, two hidden neurons and one output. Such a network requires a minimum $1 \cdot 2 + 2 \cdot 1 = 4$ weights. We begin calculations with this network. We can also vary the number of bits per weight to investigate the effect of weight granularity on the appearance of permutations. Figure 4.8 shows the number of permuted pairs for a range of representations in a

Algorithm 3 Calculate the probability of drawing two strings from the same permutation group for a representation with n items and l spaces, using the partition method.

procedure PARTITION_PROBABILITY(n, l)
  $total \leftarrow 0$
  for $partition \in$ partitions(l) **do**
    $counts \leftarrow [\ partition.\text{count}(x)$ **for** $x \in [\ 1, (l+1)\ ]\ ]$ ▷ How many '1's, how many '2's...
    $spaces\_taken \leftarrow 0$ ▷ How many spaces in the string have been filled so far
    $coeff \leftarrow 1$
    for $c \in counts$ **do**
      $coeff \leftarrow coeff \cdot \binom{n-spaces\_taken}{c}$
      $spaces\_taken \leftarrow spaces\_taken + c$
    end for
    $n\_perms \leftarrow \frac{l!}{\prod_{p \in partition} p!}$
    $total \leftarrow total + coeff^{n\_perms^2 - n\_perms}$
  end for
  return $\frac{total}{n^{2l}}$
end procedure

population of size 100. The calculated values are compared to the actual values as averaged over 100 trials and are found to match closely. Figure 4.9 then shows the proportion of permutations we can expect in any initial population with a representation parameterised by a particular choice of weight granularity (bits per weight) and the number of weights in the network.

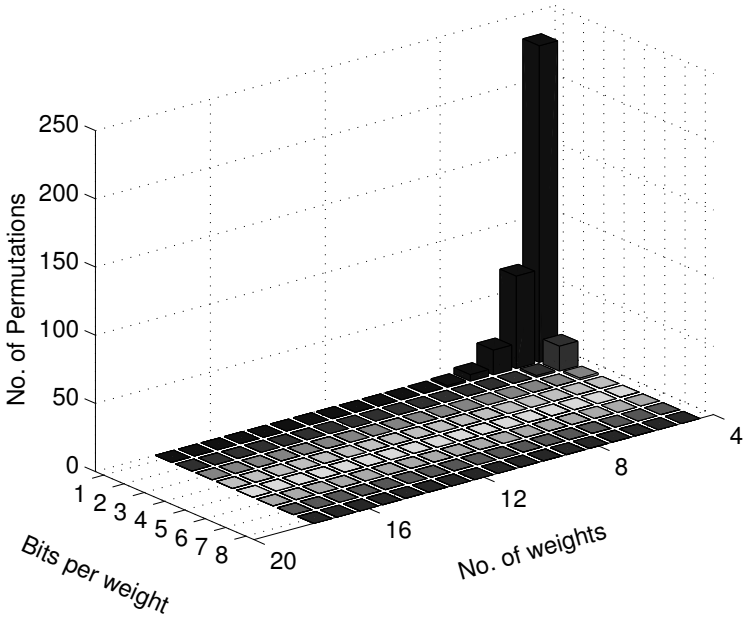
4.4 On the Redundancy of the Representation

In this section we explore how the Permutation Problem causes the search space to contain massive redundancy[3]. We then explore how role redundancy might affect the probability of the Permutation Problem occurring.

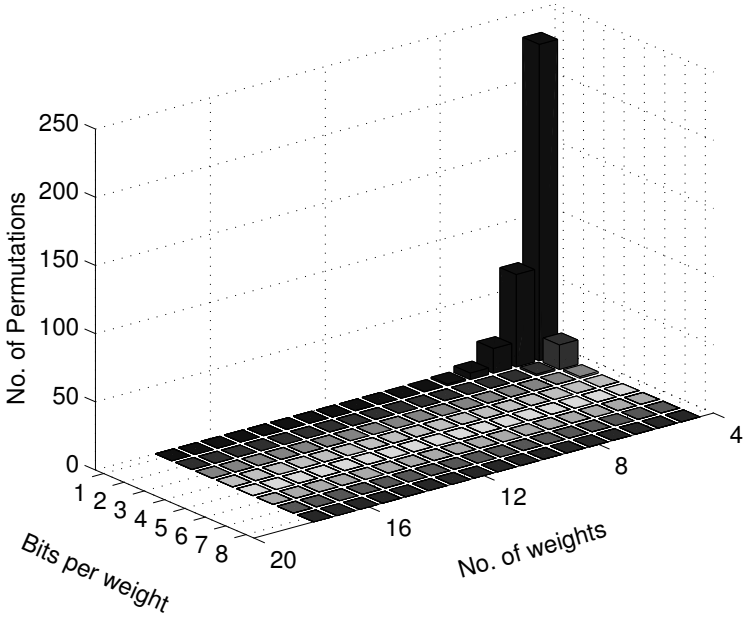
We can count the number of unique networks in the search space by first enumerating all possible neurons, and then counting the number of multisets that can be formed from these neurons. By counting the number of multisets we avoid counting the numerous permutations and count only once for each unique selection of neurons (with repetition).

The number of possible hidden neurons ϕ can be calculated as $\phi = |\alpha|^{N_c}$ where $|\alpha|$ is the number of possible allele values (weights) and $N_c = N_i + N_o$ is

[3]This redundancy is caused by individual solutions being repeated in the search space. By this definition, a representation free of redundancy would contain only one point for each unique solution.



(a) Predicted values.



(b) Actual values averaged over 100 trials.

Figure 4.8: The average number of phenotypic permutations counted in the initial population of an evolutionary run, for various representations (defined by the number of possible weight values and the number of weights in the network).

the number of connections to/from each hidden neuron where N_i and N_o are the number of inputs and outputs respectively. For even a coarse network space with $\alpha = \{-1.0, -0.9, -0.8, ..., 0.8, 0.9, 1.0\}$ ($|\alpha| = 21$), $N_i = 4$ and $N_o = 2$ we have

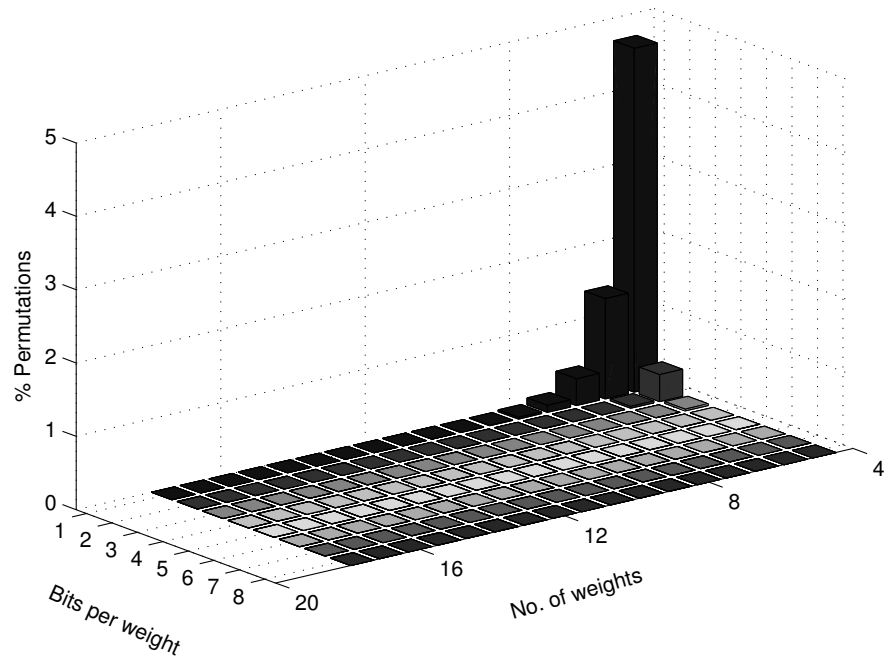


Figure 4.9: Expected proportion of permutations in the initial population.

$\phi = 21^6 = 8.6 \times 10^7$ unique neurons.

The number of unique networks Ψ is then calculated as $\Psi = \left(\!\binom{\phi}{N_h}\!\right)$ where N_h is the number of hidden neurons and $\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k}$. Given $\phi = 8.6 \times 10^7$ and $N_h = 12$ we can form

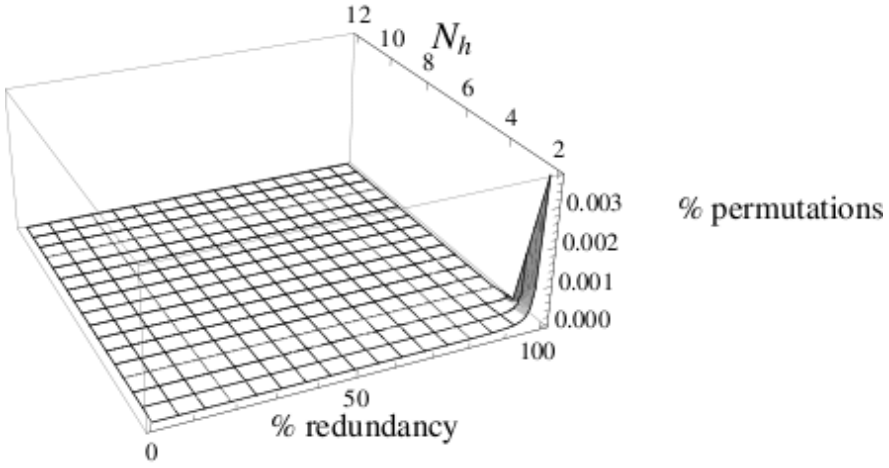
$$\left(\!\binom{8.6 \times 10^7}{12}\!\right) = 3.3 \times 10^{86}$$

unique networks.

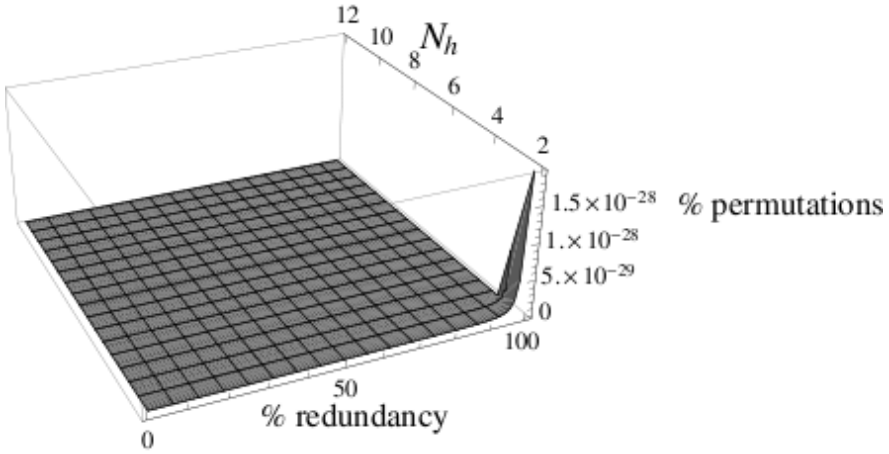
The total number of networks (including permutations) Ω is however $\Omega = \phi^{N_h}$; a considerably larger number. Using the previous example we would then have a total of $(8.6 \times 10^7)^{12} = 1.6 \times 10^{95}$ networks. The percentage of unique networks in the complete search space is therefore only $100 \cdot \frac{\Psi}{\Omega} = 2.1 \times 10^{-7} = 0.00000021\%$. This redundancy is explored further and a solution presented in Chapter 6.

This level of redundancy is essentially a lower bound for the redundancy in a Neural Network search space. If each neuron (arrangement of weights) encodes a unique function then the total redundancy is simply that caused by the Permutation Problem. For a given neuron representation however this is unlikely to be the case. An intuitive example would be the set of neuron weights which, given any problem input, cause the neuron to saturate, outputting either 0 or 1 (for a sigmoidal neuron). Such neurons are indistinguishable by the fitness function and so cause further redundancy.

We can estimate how this redundancy would affect the probability of the Permutation Problem by calculating the probability for a given representation at different levels of assumed redundancy. The change in probability is shown in



(a) (bits per weight) bpw = 2



(b) (bits per weight) bpw = 8

Figure 4.10: The percentage of the population that will be part of one or more pairs of permuted networks as the level of redundancy is increased for the UCI Iris dataset. Rather than thinking of redundancy increasing we can think of the percentage of neurons which are considered unique decreasing.

Figure 4.10 for the UCI Iris problem (a small network) and for the UCI Cancer problem (a much larger network) in Figure 4.11. In the case of the Iris problem, the probability is very low for a representation with 2 bits per weight and becomes extremely low when this is increased to 8 bits. For the UCI Cancer problem the probability is already extremely low for even a representation of just 2 bits per weight. In either case, the only significant increase in probability comes very close to 100% redundancy. Based on these findings we do not expect this kind of redundancy to affect the probability sufficiently to alter our conclusions regarding the severity of the problem.

A limitation of this analysis however is that it assumes that each group of redundant neurons is the same size. If we were to have a few groups of redundant

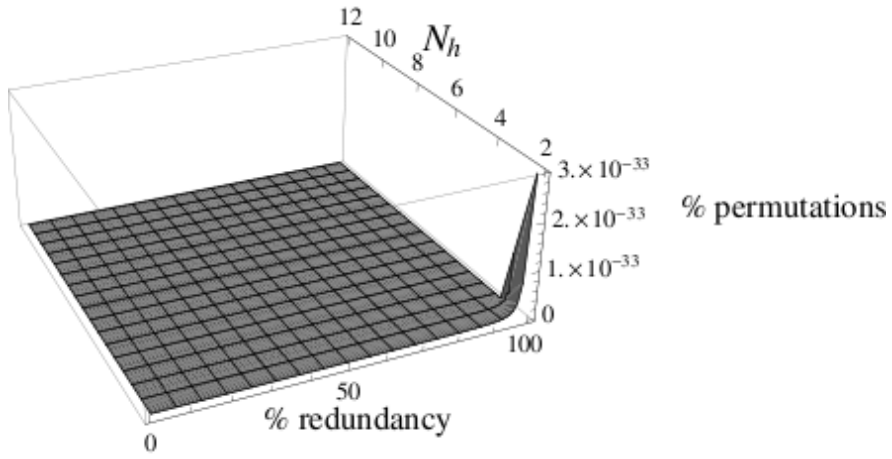


Figure 4.11: The percentage of the population that will be part of one or more pairs of permuted networks as the level of redundancy is increased for the UCI Wisconsin breast cancer dataset. Rather than thinking of redundancy increasing we can think of the percentage of neurons which are considered unique decreasing.

neurons which were especially large (relative to other groups) then this could increase the probability of the Permutation Problem occurring. This issue is not explored in detail in this work but is an interesting avenue for future work.

4.5 Proportion of Networks with N_h! Permutations

A possible explanation for any overestimation of the severity of the Permutation Problem in the literature may have stemmed from an implicit assumption that given a network search space composed of networks with N_h hidden neurons, any given network will have N_h! permutations. While this is true for some or even most networks in a representation space, it depends wholly on the parameterisation of the space. The figure of N_h! assumes that all of the neurons are unique (in terms of their weights for example) but this will not be the case for all networks in a search space; some possible strings will contain repeated neurons. The question of what proportion of networks have the maximal number of permutations (N_h!) has not previously been answered in the literature so we investigate it here for a range of representations and two classification problems.

Starting with the UCI Iris problem which has four inputs and 3 outputs, we can see in Table 4.5 that generally as the number of hidden neurons increases, the proportion of networks with the maximal number of permutations decreases rapidly. However, this is only the case for granular representations with assigned bits per weight less than 16. Once we reach 16 bits per weight and above the difference starts to be negligible.

For the UCI Cancer problem which has 30 inputs and two outputs, the effect is similar but due to the network being larger the effect of increasing the number

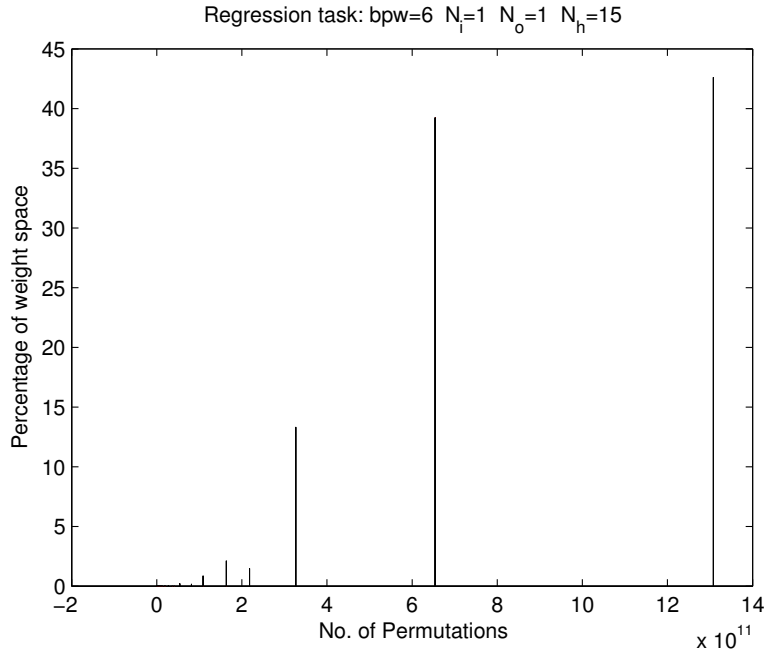


Figure 4.12: Proportions of permutations for a regression problem.

N_h	**bpw=2**	**bpw=4**	**bpw=8**	**bpw=16**
5	68.5%	91.4%	99.4%	∼100%
10	16.1%	66.1%	97.52%	∼100%
15	1%	37.5%	94.3%	∼100%
20	0.01%	16.5%	89.91%	∼100%
25	∼0%	5.5%	84.52%	99.9%

Table 4.3: Proportions of networks with $N_h!$ permutations - UCI Iris: $N_i = 4, N_o = 3$

of hidden neurons is reduced; now as long as the bits per weight is set to at least 8, the assumption that most networks have $N_h!$ permutations holds.

If we have a fairly large network (such as that required for most classification problems) and a relatively fine weight granularity (such as 8 bits per weight or higher) then the assumption that most (but not all) networks have the maximal number of $N_h!$ permutations is a fair one. If we are optimising networks for a regression problem however it is possible that this assumption does not hold. Looking at Figure 4.5 we can see that only $\sim 43\%$ of networks have the maximal number of permutations.

Why is this important? The calculations for the probability of the Permutation Problem occurring already take into account exactly the proportion of networks with each possible number of permutations, so the calculation of the probability

N_h	**bpw=2**	**bpw=4**	**bpw=8**	**bpw=16**
5	92.4%	98.1%	99.9%	∼100%
10	69.7%	91.54%	99.5%	∼100%
15	42.6%	81.3%	98.7%	∼100%
20	20.9%	68.7%	97.7%	∼100%
25	8.1%	55.1%	96.4%	99.9%

Table 4.4: Proportions of networks with $N_h!$ permutations - UCI Cancer: $N_i = 30, N_o = 2$

is unaffected. The aim of this section has been to demonstrate that it is not necessarily fair to assume that all or even most networks have the maximal number of permutations. Given that the probability of the exact-weight Permutation Problem occurring is still low for all but the most granular representations this is not a serious concern.

4.6 Chapter Summary

In this chapter we have presented an equation (Equation 4.2) for calculating the probability of picking two members from the same multiset; effectively the probability of the Permutation Problem occurring when uniformly initialising a population. As a naïve implementation of Equation 4.2 is too inefficient for practical use, a novel method of calculation based on the theory of integer partitions was presented which allows this probability to be calculated for realistic representation spaces. This is achieved by making the computational complexity depend only on the number of hidden neurons in the network rather than the number of neurons available in the representation space. The results from these calculations were then validated empirically.

We examined the redundancy of the search space as introduced by the permutation symmetry and presented a hypothesis for how neuron role redundancy could affect the conclusions of this chapter. Finally we investigated the nature of the search space, particularly what proportion of networks have the maximal number of permutations and how this affects the likelihood of the Permutation Problem occurring.

The results of this work tell us only about the occurrence of the Permutation Problem in the initial population. In later generations when the average fitness is higher we may see higher or lower rates of occurrence of permutations. The motivation of this chapter has therefore been to explore occurrence of the Permutation Problem due to presence of permutations in the initial population.

Chapter 5

Empirical Analysis

5.1 Introduction

In the previous chapter we explored the rate of occurrence of the Permutation Problem in the initial population from a probabilistic perspective. A limitation of this work is that while it characterises accurately the rate of occurrence when initialising the initial population, it does not explore what happens during later generations. In this chapter we therefore investigate the Permutation Problem from an empirical perspective to investigate how this rate of occurrence changes in later generations. We begin by counting explicitly the number of permutations that appear during runs of a typical Genetic Algorithm (GA). As the Permutation Problem and recombination are closely linked, we also investigate the efficacy of crossover in Neuroevolution; we use Price's Equation to analyse the contribution of crossover to the evolutionary process in order to shed light on the role of crossover in Neuroevolutionary algorithms. Following the analysis based on Price's Equation we then investigate whether crossover is acting as a macromutation rather than as a recombination operator as might be expected. Finally we repeat the counting experiment for three classification problems using a binary representation of varying granularity to determine the probability in low-granularity search spaces.

5.2 Counting Permutations in a Genetic Algorithm

In this section we investigate the impact of the Permutation Problem on a typical GA evolving Neural Networks for a difficult control problem, using Price's Equation to analyse the contribution of crossover and its relationship to the Permutation Problem.

5.2.1 Introduction

By explicitly counting the occurrence of permutations in two typical GA configurations we can produce an empirical estimate for the number of permutations

expected in a given run for this problem/algorithm configuration.

It is possible that the lack of even estimates for the rate of occurrence of permutations in the literature has lead to overestimation of the severity of the problem, leading in some cases to crossover having been avoided or customised where it might have been a useful search operator in its canonical form [MD89, BMS90, Bra95, CF96, Thi96, SM04, GPOBHM05]. In addition to the counting of permutations we use Price's equation to examine the interrelationships of the genetic operators to build a more complete picture of how each operator contributes to the search for Neural Networks of increasing fitness. This is necessary as simply comparing performance with and without crossover runs the risk of simply comparing two different selection/variation pressures rather than isolating the effects of crossover[1].

The aim of this investigation is to check empirically the number of permutations which occur in runs of a typical Evolutionary Algorithm, and in doing so more fully understand the individual role each operator plays and their interactions in the evolution of Neural Networks so that informed choices can be made when designing future Neuroevolutionary algorithms.

5.2.2 Price's Equation

We now present Price's equation which is used to provide insight into the contribution and interactions of the genetic operators used in this work.

In 1970 George Price presented an equation which models the covariance relationship between the frequency of a given gene in an offspring and the number of offspring produced by its parent [Pri70]. A high covariance value for a particular gene indicates that it would be a good predictor of selection.

Price's equation requires some measurable attribute by which to compare parents and their offspring. While originally this measure was gene frequency it was later shown that other measures such as mean fitness could be used instead, broadening the applicability of the equation in the Evolutionary Computation domain [Alt95].

Given a parent population P_1 and offspring population P_2 we calculate the average of the measured attribute over each population, producing Q_1 and Q_2 respectively. Price's equation then states that

$$\Delta Q = \frac{Cov(z,q)}{\bar{z}} + \frac{\sum z_i \Delta q_i}{N\bar{z}}, \tag{5.1}$$

where,

- $\Delta Q = Q_2 - Q_1$
- N is the number of individuals in P_1

[1]Removing crossover (a variational operator) will often strengthen the selection pressure (as there is now less variation per generation). For a fair comparison this decrease in variation must be controlled for.

- z_i is the number of offspring to which parent i contributed genetic material
- $\bar{z} = \frac{\sum_i z_i}{N}$
- q_i is the measurement of the chosen attribute of parent i
- q'_i is the average of the attribute as measured in the offspring of parent i
- $\Delta q_i = q'_i - q_i$

Price's equation estimates the change in a measurable attribute from the parent to offspring population. In this work as in [PBJ03] the attribute we are interested in is the average change in fitness between generations:

$$\Delta Q = \overline{f}(t+1) - \overline{f}(t), \tag{5.2}$$

where $\overline{f}(t)$ is the average fitness of the population at generation t.

In terms of the average change in fitness between generations, Price's equation allows us to separate the contribution from selection (first term) and the variation operators (second term). In the design of Evolutionary Algorithms this level of introspection allows the designer to clearly see the trade-off between selection and variation strength and find a balance between the two.

Potter *et al.* have further extended the utility of Price's equation by demonstrating that the second term of the equation is equivalent to the sum of contributions from each individual operator [PBJ03]:

$$\Delta Q = \frac{Cov(z,q)}{\bar{z}} + \sum_{j=1}^{k} \frac{\sum z_i \Delta q_{ij}}{N\bar{z}}, \tag{5.3}$$

where,

- k is the number of genetic operators
- q'_{ij} is the average value of the measured attribute of the offspring of parent i, after the application of operator j
- $\Delta q_{ij} = q'_{ij} - q'_{i(j-1)}$.
- $q'_{i0} = q_i$
- and $q'_{ik} = q'_i$.

This extension of the equation allows the relative contribution of each operator to be compared in a more rigourous fashion than when simply removing each operator one at a time; an approach which fails to capture the *interaction* of the operators.

At this stage we can use the extended equation to, over a single run, determine the average change in fitness caused by each operator at each generation. We can

then repeat this and average the results over several runs. Where this measure falls short is made clear in cases such as that of Figure 5.10 where two operators (in this case crossover and mutation) have a (nearly) identical mean contribution. In order to gain greater insight into the relative merits of each operator we examine the variance of the effect of each operator at each generation, rather than simply the mean [BPJ04]. It should be noted that this is not the variance of the mean effect of an operator averaged over a number of runs, but instead represents the 'reach' of the operator at a particular generation, i.e. the range of fitness values over which it has been capable of producing offspring at.

The variable of interest here is then $Var(X)$ where:

- $X = q_{ijk} - q_{i(j-1)k}$
- and q_{ijk} is the measured attribute of the kth child of parent i after the application of operator j.

In the presented graphs, the variance of each operator is shaded in grey, one standard deviation above and below the mean effect of the operator.

As in [PBJ03], before using the model to analyse operator effects we first check the ability of the extended equation to predict the change in mean fitness on the problem at hand. Figure 5.1 shows the predicted and actual changes in mean fitness over the course of a single evolutionary run of the algorithm. Here we can see the predictions made by Price's equation follow almost exactly the actual average change in fitness between generations, giving us confidence in its ability to accurately characterise the contribution of each operator. The accuracy could be further improved by increasing the population size in order to reduce the sampling error.

5.2.3 Experimental Setup

In this section we present the details of the GA used, the Cart-Pole problem and the method for counting permutations.

Overview

In this work we look at two configurations of a typical GA: one which employs

- fitness-proportional selection,
- 1-point crossover (probability of application 0.6),
- and Gaussian mutation (probability of application 0.6, variance 0.3).

The other adds to this the application of the inversion operator (probability of application 0.6). The two experiments are referred to as Crossover-Mutation (CM) and Crossover-Inversion-Mutation (CIM) throughout this work. All results given have been averaged over 200 runs except for the permutation counting experiments which were averaged over 20 runs.

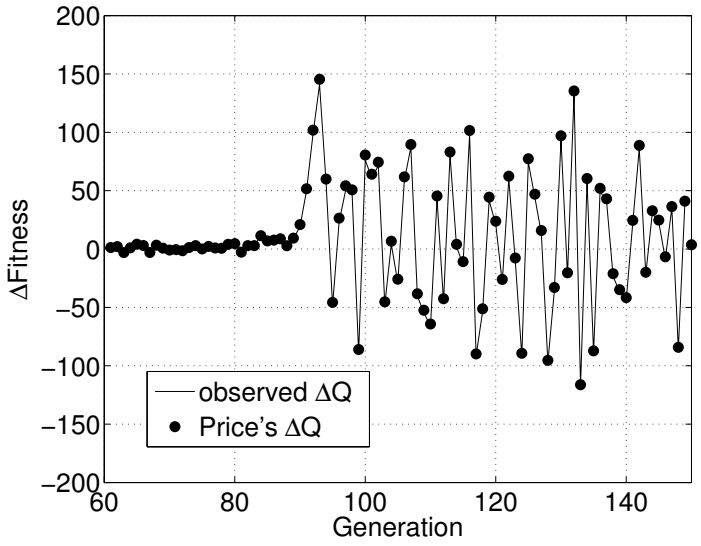


Figure 5.1: The actual average change in fitness each generation in a single run compared with the value as predicted by Price's equation. The graph has been focussed on the area of highest variability; for generations 1 to 60 both the predicted and actual values are around zero.

The Cart-Pole Problem

The Cart-Pole problem requires a controller to balance one or more hinged poles attached to a cart on a finite length of track by applying force in either direction. The controller is considered to have failed if the cart strays outside the boundaries of the track, or the poles exceed a specified angle from vertical.

In this work we employ a variation with two poles of differing lengths, 1.0 and 0.1 metres respectively. The controller must keep the poles within the failure angle of 36° for 1,000 time steps, using continuous control. The longer pole always starts 4° off-centre, the shorter pole vertical.

At each time step, the networks receive a fixed bias value b of 0.5, the position of the cart x and pole angles θ_1 and θ_2. In the first variant of the problem, the velocity information for the cart $\dot{x}$ and pole angular velocities $\left(\dot{\theta}_1, \dot{\theta}_2\right)$ are also passed to the network. For the more difficult second variant however, the network is extended with recurrent connections, which it must use to infer the velocities, providing a significant leap in task difficulty.

The fitness function used here measures the number of time steps that the controller is able to keep the cart and poles within the defined limits. The inputs from the environment consist of the cart position x measured as its deviation from its starting point in the center of the track and the angle from vertical of the two poles, θ_1 and θ_2.

The latter variant of the pole balancing problem is particularly difficult as two poles must be balanced simultaneously, without the provision of cart/pole

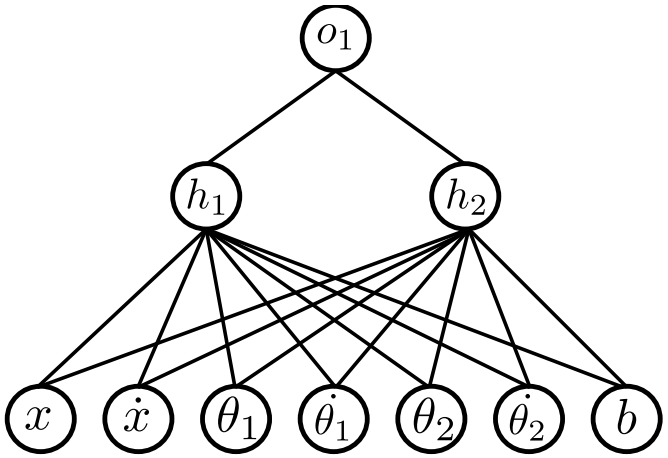


Figure 5.2: Network used for the Dual Pole Balancing with velocities experiment. The six environment inputs are fully connected to two sigmoid nodes, which are in turn connected to an output sigmoid node.

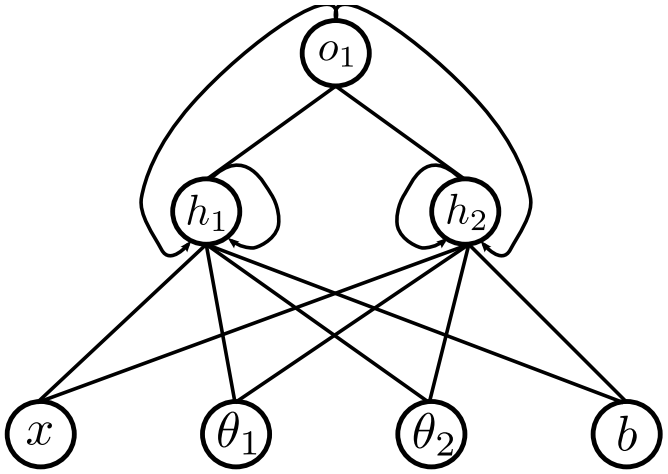


Figure 5.3: Network used for the Dual Pole Balancing without velocities experiment. Each hidden node now receives recurrent inputs from the output unit and their own output from the previous time step.

velocity information. This problem was chosen as the benchmark due to its difficulty. It poses a significant challenge to current Reinforcement Learning and Neuroevolutionary methods in the literature [GSM08].

Following the naming conventions of [OYKM07], we refer to the version of the pole balancing problem where velocity is given as the Dual Pole balancing (DP) problem, and the variant used here with velocity information removed as the Dual Pole balancing No Velocities (DPNV) problem. The network templates used for the DP and DPNV experiments are shown in figures 5.2 and 5.3 respectively. In this section we only make use of the DPNV setup. Both the DP and DPNV variants will be used in the investigation of Section 5.3.

Model

Each individual in the population is a Neural Network with two hidden neurons (h_1, h_2) and one output neuron o. The three environment inputs (x, θ_1, θ_2) are fully connected to the hidden neurons, which are in turn both connected to the network output. The output neuron further receives as input the network output from the previous time step. This recurrent connection allows the network to determine the appropriate direction and magnitude of force to apply based not only on the environment inputs but also on its previous actions[2].

The genotype for an individual network is composed of 9 parameters forming a single string of reals of the form, $(h_1^x, h_1^{\theta_1}, h_1^{\theta_2}, h_2^x, h_2^{\theta_1}, h_2^{\theta_2}, o^{h1}, o^{h2}, o^o)$ where h, o denote hidden and output neurons respectively, subscripts denote a neuron index where applicable and the superscript denotes the neuron input[3].

A limitation which should be clarified at this point is that in order to compare the *performance* of this algorithm with contemporary research it would be necessary to use a more complex fitness function such as that of Gruau [GWP96] which penalises solutions which simply swing the poles back and forth without stabilising the cart in the center of the track. As the purpose of this work is to examine the significance of the Permutation Problem on the evolution of Neural Networks such comparison is, at this stage, outside the scope of investigation.

Counting Permutations

In counting the permutations, each weight is rounded to 1 decimal place so that $(0.13, 0.25)$ and $(0.33, 0.10)$ would be considered to be permutations of each other (as each would resolve to $(0.1, 0.3)$ and $(0.3, 0.1)$ respectively)[4].

It should be noted that the permutations counted here do not necessarily correspond to the example given in Figure 3.1, i.e. Phenotypic Weight Permutations. It is worth noting that in order to transform one genotype into the other (in this example), more than one gene translocation is required, coupled with an inversion. This is not possible given the configuration of operators in this work and would represent a configuration deviating considerably from what is considered a 'typical' GA.

The kind of permutations we are looking for therefore are a superset of the permutations which represent the Permutation Problem as described in the literature. By counting all permutations we obtain an estimate on the upper bound of instances of the Permutation Problem we might expect to see in an evolutionary run.

[2]While it is possible to solve this problem without recurrent connections, the resulting solution is likely to perform and generalise poorly.

[3]So for example $h_1^{\theta_1}$ denotes the first hidden neuron receiving as input the angle of the long pole and o^o denotes the recurrent connection of the output neuron.

[4]There are many possible ways in which the search space can be discretised; this method has been chosen as with such a granular search space we might reasonably expect some permutations to appear.

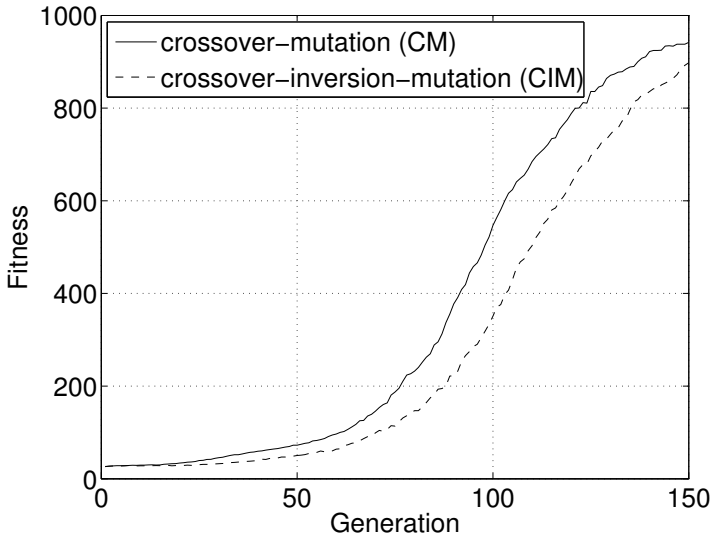


Figure 5.4: Best fitness curves for two configurations (averaged over 200 trials), one which searches using crossover and mutation (CM), the other with crossover, inversion and mutation (CIM).

5.2.4 Discussion

Looking at the best fitness curves in Figure 5.4 we can see that both the CM and CIM configurations are able to make good progress on the problem, with CM progressing more quickly than CIM.

Considering CM first we might assume that, due to the Permutation Problem, crossover is hindering the search process. In order to determine whether this is true or not we could run a test where we examine the performance of selection and mutation alone. This has the possibility of misleading however as the absence of crossover may lead to there being a better/worse match between the chosen selection pressure (which remains the same) and the reduced variation caused by employing only the mutation operator.

A more informative approach would be to use the extension of Price's equation (Equation 5.3) to examine the relative contribution of crossover and mutation. Figure 5.5 shows the average change in fitness at each generation for the operators used in this configuration. Here we can see that both crossover and mutation on average produce offspring with lower fitness than their parents, with crossover appearing to produce better individuals overall. However, in an evolutionary search the aim is not necessarily to employ only the operators which on average produce more fit individuals; large mutations can for example occasionally provide similarly large leaps in fitness even if on average the mutations are deleterious.

Looking at the variance (shaded in grey) then of the crossover (Figure 5.6) and mutation (Figure 5.7) operators in the CM experiment we can confirm that crossover does appear to be contributing more overall to the fitness of the

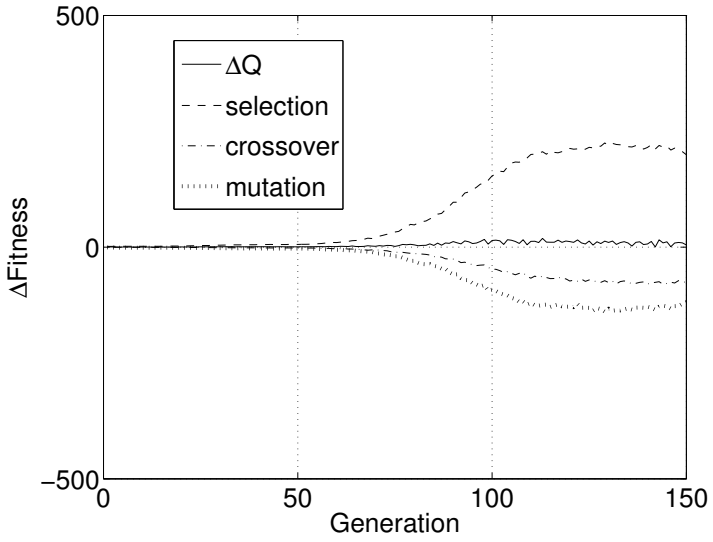


Figure 5.5: Average contribution of each operator at each generation (then averaged over 200 trials) for the Crossover-Mutation (CM) experiment.

population, compared to mutation[5].

Price's equation quantifies the contribution of a particular operator, but can't tell us how the operator achieves this. Based on this analysis so far it appears that crossover is indeed a useful operator for this problem and representation combination. Could we improve the performance of crossover further by eliminating the possibility of permutations occurring? By comparing pair-wise all members of each generation we could determine that *zero* permutations were detected in any of the 20 runs.

Why are there no permutations? The example in Figure 3.1 on page 46 suggests an intuitive argument for why there will be few if any permutations in a run of a typical GA on this problem: in order to transform (a) into (b) we must perform multiple gene translocations and an inversion. With only crossover and mutation it would be necessary for mutation to mutate an individual such that the result was a permutation of another individual in the population. We have demonstrated that in this case such permutations are not being created.

This raises a question which has so far been largely overlooked in the literature: when and under what conditions does the Permutation Problem occur? We have seen from the example in Figure 3.1 that inversion is capable of permuting genes (with the possible effect of creating genotypic permutations). In order for this to happen there must be:

- two or more copies of a selected parent in the population available for selection and,

[5]As the area of the shaded region above zero on the y-axis is greater for crossover than for mutation we conclude that it is on average contributing more to the fitness of the population.

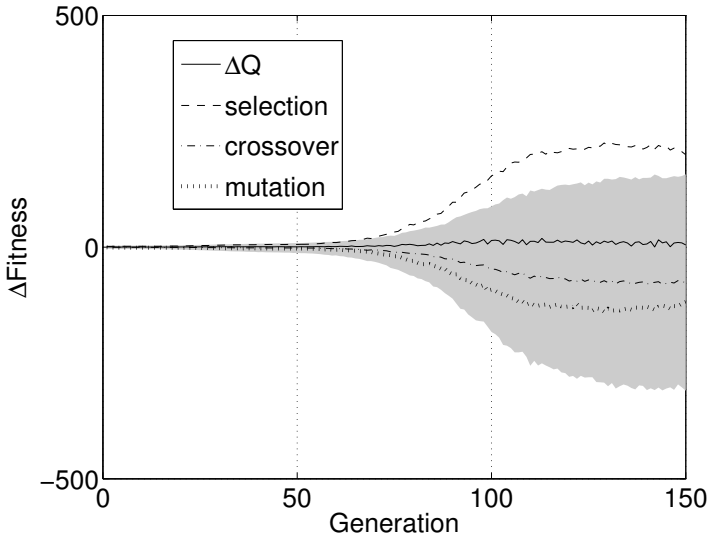


Figure 5.6: Variance of the crossover operator in the CM experiment, averaged over 200 trials.

- one or more of the copies should be (partially or fully) inverted.

The inverted strings are then permutations of each other and the original parent. The recombination of these individuals should on average produce offspring of very low fitness as has been seen in the example of the Incompatible Representations Problem in Figure 3.6[6].

Could this hypothesis—that inversion is introducing permutations—account for the difference in performance between the CM and CIM configurations?

Looking at the relative contributions of each operator in the CIM configuration (which consists of crossover, inversion and mutation) in Figure 5.10, we can see that selection is accounting for more of the fitness gain on average than it is in the CM configuration. This is then matched by the strongly deleterious (on average) effect of inversion. In terms of their mean effect crossover and mutation are now indistinguishable. Of interest here is the shape of the mean effect curve of selection: while under the CM configuration progress appears to be stalling close to the 150^{th} generation, with the addition of inversion the progress is, in comparison, showing a markedly decreased slowdown. A possible reason for this can be found in the diversity profile of each configuration (Figure 5.8): inversion, in conjunction with the other operators, is multiplying the diversity in the population, biasing the algorithm towards exploration rather than exploitation[7]. While this may not

[6]If the offspring post-inversion are of low fitness however then they are unlikely to be selected for mating; while this would decrease the efficiency of the search process it would not lead to occurrences of the Permutation Problem as a necessary requirement is that the permuted individuals both be selected for recombination.

[7]Inversion and crossover have the property of producing diversity from the diversity already present in the population. Repeated applications of both operators therefore results in a

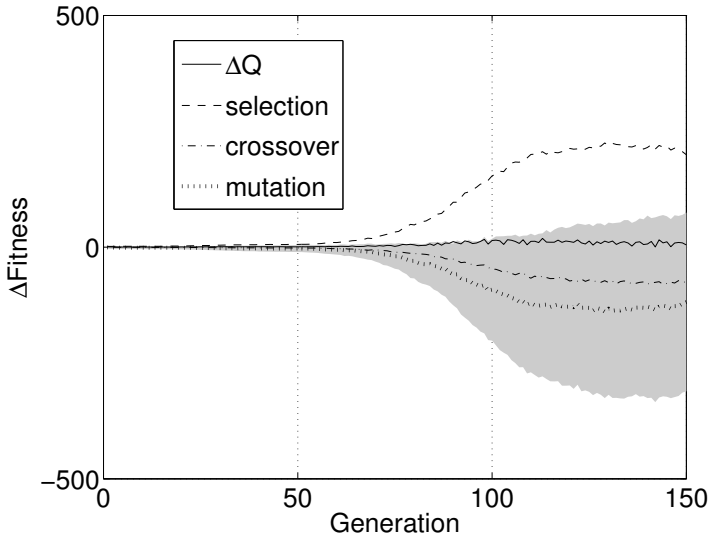


Figure 5.7: Variance of the mutation operator in the CM experiment, averaged over 200 trials.

be a sustainable search strategy (for that, a balance between exploration and exploitation would be necessary), for the given fitness target and generation limit, the introduction of inversion has not had a severely negative impact.

Upon examining the variance of crossover in this configuration (Figure 5.9) and mutation (Figure 5.11) we can see that crossover is still the more productive operator on average. The variance for the inversion operator (Figure 5.12) suggests that it very rarely contributes positively to the fitness of the individuals it is applied to. This finding is in line with expectations based on the literature [Whi95, Yao99, YL97], but is its predominantly negative effect due to its creation of permutations? The same permutation counting experiment is run again with inversion included, but again *zero* permutations are counted. This suggests that while inversion appears to be a poor operator for this problem and/or chosen representation, its lack of utility is not due to the manifestation of the Permutation Problem. Rather, the inverted solutions are rarely selected due to their low fitness and so discarded early or, when selected and recombined with good solutions, are producing offspring of very low fitness as in the example in Figure 3.6 which would be more a problem of incompatible representations rather than of permutations.

5.2.5 Summary

We have used Price's equation and explicit enumeration of permutations in order to investigate empirically the significance of the Permutation Problem in the evolution of a Neural Network for a difficult control problem. We discovered that a typical GA is able to evolve suitable controllers for this non-Markovian variant

considerable shuffling of genes, leading to wider exploration of the search space.

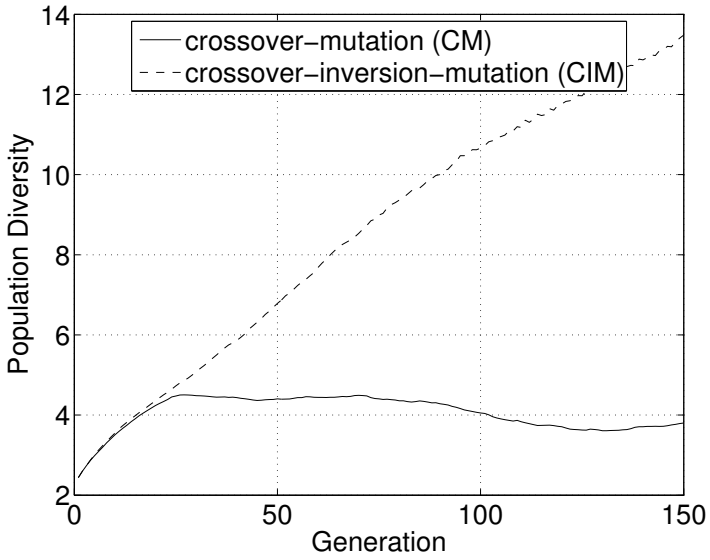


Figure 5.8: A comparison of the diversity of each population over the course of evolution, averaged over 200 trials. Note that the introduction of inversion disrupts the balance between the selection pressure and rate of variation in the population.

of the Cart-Pole problem without the need for specialised operators or genetic representation.

The results of this work suggest that for typical GAs the Permutation Problem is perhaps not quite as serious as originally thought in early research into the evolution of Neural Networks. This work has demonstrated that 1) the performance of a typical GA on a difficult control problem is quite acceptable and 2) the introduction of inversion (which should in theory seriously disrupt the search process) has a relatively minor negative effect. In addition, it has been shown that with a typical GA, crossover and mutation alone are insufficient for generating permutations in the population, and that the addition of inversion likewise results in no permutations, due largely perh aps to the later application of the mutation operator.

Using Potter's extension to Price's equation we have analysed the relative contribution to population fitness of each operator. This analysis, in conjunction with the counting of permutations, has allowed us to determine the relative utility of each operator for this problem. In doing so we were able to avoid wrongly attributing the decrease in performance seen following the introduction of inversion to the Permutation Problem by explicitly checking for and discovering the lack of permutations in the population.

This relationship between a chosen representation and the genetic operators could be investigated further by focusing on the level of disruption caused by an operator on average, with the aim of selecting the most appropriate set of operators. It is not clear exactly what level of disruption is optimal: too little and

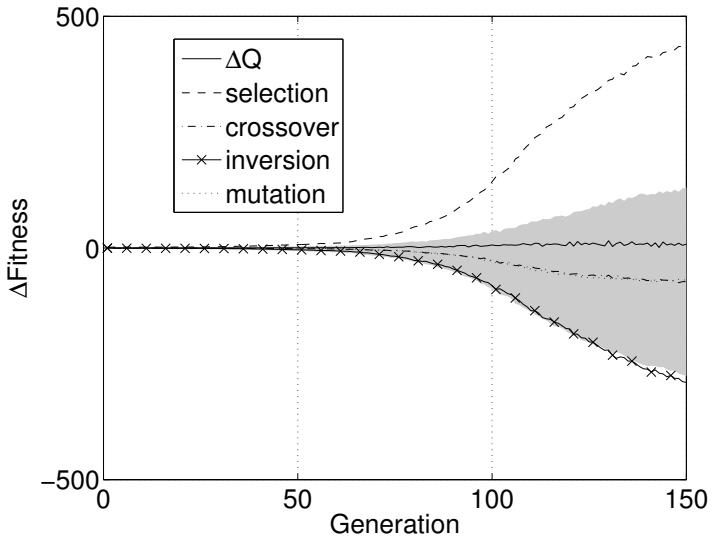


Figure 5.9: Variance of the crossover operator in the CIM experiment, averaged over 200 trials.

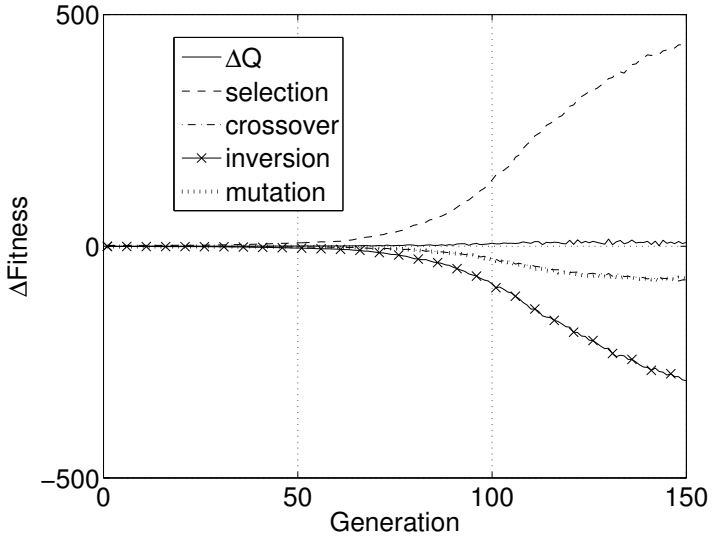


Figure 5.10: Average contribution of each operator for the Crossover-Inversion-Mutation (CIM) experiment, averaged over 200 trials.

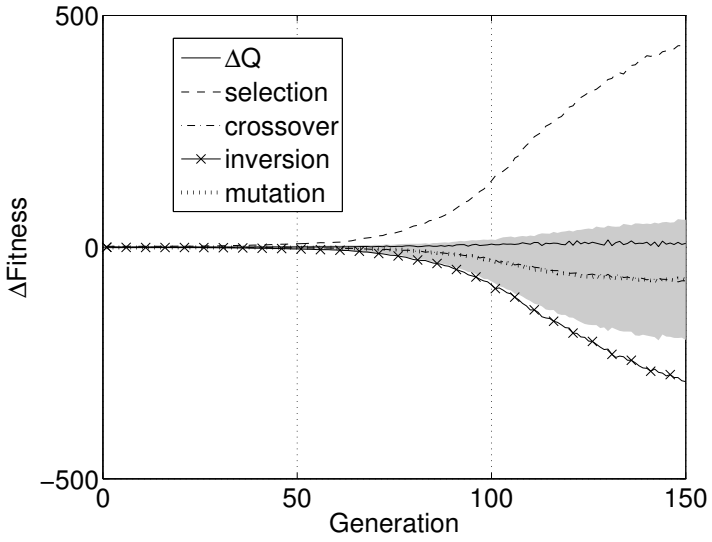


Figure 5.11: Variance of the mutation operator in the CIM experiment, averaged over 200 trials.

we make little progress; too much and the search becomes too noisy. Ideally we should match the selection pressure with an appropriate level of disruption that allows for continued, controlled (rather than random) exploration of the search space.

5.2.6 Future Work

The problem of incompatible network representations suggests that the recombination of significantly dissimilar networks should be avoided; this view is shared more generally with respect to most evolutionary algorithms. This then suggests that speciation may be an important consideration in the design of evolutionary algorithms for Neural Network design and optimisation. The role of speciation in the evolution of Neural Networks will therefore be an area of interest for future investigations.

Additionally, the 'Price plots', (e.g. Figure 5.12, where operator effect variance is shaded in gray) give greater insight into the effects of the variational operators, beyond simply examining their average effect on individuals. This is useful in identifying operators which, while apparently poor in the average case, produce occasional but significant jumps in fitness. The current method does not however take the *skew* of the operator effects into account. While shading one standard deviation to either side of the average gives us some idea of the effect of an operator, it assumes perhaps too strongly the normality of the operator effects. The operator may in fact be significantly better or worse than the picture given by this method. For future work we would recommend examining not only the variance of operator effects but also their skew. In the cases where the operator

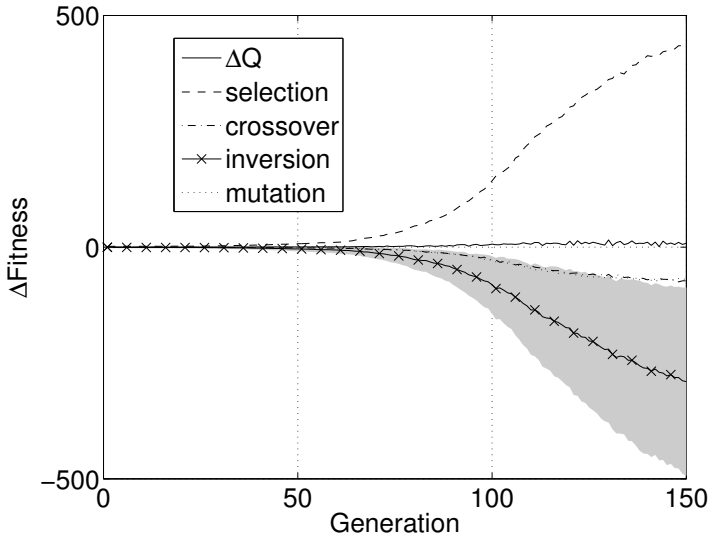


Figure 5.12: Variance of the inversion operator in the CIM experiment, averaged over 200 trials.

effects are significantly non-normal the Price plots would then lose their symmetry and more accurately reflect the contribution of each operator the evolutionary process.

5.3 Is Crossover Acting as a Macromutation?

Having investigated the role and contribution of crossover in a Neuroevolutionary algorithm using Price's Equation, we are now left with the question of *how* crossover contributes to the search process, particularly whether recombination is combining high fitness building blocks or whether it is instead acting as a large (macro) mutation. In this section we explore whether the function of crossover is simply as a large mutation or whether it has some other beneficial properties.

5.3.1 Introduction

The problem of how to effectively combine Neural Networks is still an open issue in the field of Neuroevolution. While results in this work and others suggest that simple 1-point crossover can be a useful operator when optimising Neural Networks [Han93, GM03a], it is not clear why this is so. In this section we investigate whether crossover is simply performing a large mutation (a macromutation) on networks or whether there is some possibility that some kind of useful recombination is occurring.

We present an analysis of the effect of crossover and a randomised crossover operator on a simple co-evolutionary Neuroevolution algorithm, which is based on an established algorithm in the literature (Enforced Sub-Populations). This

algorithm is applied to two variants of the dual pole balancing task. The results show that there is some benefit in recombining individuals with other individuals from the same population, rather than randomly-generated individuals. This itself does not show that there are building blocks in the genotype (though this may be the case) but does show that it is not simply the mechanism of the crossing of strings that causes the increase in fitness, i.e. that the choice of string will be important.

While recent algorithms have employed crossover with encouraging results [SM02, GSM06, MGW$^+$06, GSM08], the recombination of networks has been criticised in the literature due to the problem of parental incompatibility: two networks which appear to be outwardly similar or identical (e.g. in terms of their respective outputs or architecture) may internally have different structures and/or parameters sets, causing them to be incompatible for crossover [ASP94, OYKM07, YI08].

This incompatibility can be classified as being either parametric, architectural or a combination of the two:

- Parametric incompatibility refers to the case where two networks of identical architecture are crossed over genetically after a period of evolution in isolation. Despite producing similar outputs given the same input, there is no guarantee that the internal network weights are similarly close. In this case the crossover mechanism is said to disrupt the *internal* or *distributed* representation of the problem solution encoded in each network.

- Architectural incompatibility is commonly exemplified with reference to the Permutation Problem: crossing over two genotypes which individually produce functionally-equivalent networks (a different ordering of the same neurons and connection weights) can result in a child network which possesses less computational ability than either parent due to the duplication of resources [ASP94]. Another form of architectural incompatibility could involve networks of different toplogy; in this case it is not clear how the networks should be recombined although a sensible method is presented in [Sta04].

In this work we are mostly interested in the effect of crossover as part of the Permutation Problem and so limit the investigation to that of architectural incompatibility with fixed architectures.

The problem of incompatibility has led to Evolutionary Programming (EP) being recommended as an alternative as it does not rely on information sharing between members of the population and so does not carry the requirement of compatibility [YL96, Yao99, YI08].

Our analysis of the effect of crossover and the random crossover of [Jon95] is based on a simple but novel co-evolutionary NE algorithm, Co-evolution of Sigmoidal Decision Surfaces (CSDS), which itself is based on an established algorithm in the literature (Enforced Sub-Populations (ESP) [GM99]).

The two pole balancing tasks are used as benchmarks for the CSDS algorithm as together they provide control problems of significant and increasing difficulty. The problems were chosen as they are of sufficient difficulty such that a process of random weight guessing is insufficient for finding a solution in the allowed experiment time [GSM08].

In this work we demonstrate that the evolutionary progress facilitated by the crossover operator is not simply due to the mechanical recombination of strings but that the choice of parents is important. While this may at first seem to be an obvious result, there exist problems which contain no building blocks yet benefit considerably from crossover, even when the second parent is randomly generated and so should be of low fitness [Jon95]. The benefit comes from the large mutations that crossover performs; as such while crossover is not functioning in its intended capacity (i.e. recombining building blocks) its mechanism is still beneficial to the evolutionary process.

5.3.2 The CSDS Algorithm

The presented algorithm, Co-evolution of Sigmoidal Decision Surfaces (CSDS), is based on the Enforced Sub-Populations (ESP) algorithm [GM99]. ESP was chosen as the base algorithm as it is crossover based and has been successfully employed on difficult applications such as cache allocation on a multi-core CPU [GBM01] and rocket guidance [GM03b].

Like ESP, the CSDS algorithm works by evolving populations of neurons which compete to form part of the whole network as in the ESP algorithm. A population (referred to as a sub-population in ESP) is maintained for each hidden neuron in the network template. At the sub-population level the aim is to find the best neuron for that particular position in the network. At the network level the aim is then to cooperatively co-evolve a set of neurons which together solve the given task.

5.3.3 Randomised Crossover

In [Jon95], Jones highlights the distinction between the *idea* and the *mechanism* of crossover. The idea of crossover is what we expect it to contribute to an evolutionary search, i.e. that it will recombine short, low order, high fitness building blocks from the best individuals to form better solutions [Gol89]. In problems where the assumptions of the idea do not hold, for example in cases where there are verifiably no building blocks [BPJ05], then how can we explain its contribution? Jones suggests that it may simply be the *mechanism*, the mechanical process of recombining two strings that provides the benefit. In this case, selecting an individual and crossing it with a randomly-generated individual can in some cases be a useful operator [Jon95].

5.3.4 Method

Algorithm 4 The CSDS algorithm, used to evaluate the different evolutionary operators on the pole balancing tasks.

```
while task not complete do
    if generations_since_last_improvement is 2 then
        for all sp ∈ subpopulations do
            sp.weights.perturb_cauchy(γ = 0.3)
            sp.sigmoids.perturb_gauss(μ = 0.0, σ² = 0.3)
        end for
    end if
    perform_uniform_trials(n_trials = 100)
    for all sp ∈ subpopulations do
        new_population ← ⟨⟩
        sp.sort()
        for all n ∈ sp.top_quartile() do
            parent_1 ← n
            parent_2 ← random(sp.top_quartile())
            if b_random_crossover then
                randomise_weights(parent_2)
            end if
            child1, child2 ← crossover(parent_1, parent_2)
            new_population.append(parent_1, parent_2,
                                  child_1, child_2)
        end for
        sp.population ← new_population
    end for
end while
```

The problem of pole balancing requires a controller to apply force at regular intervals to a cart on a track of finite length, to which a configurable number of hinged poles are attached (the problem is defined in more detail in Section 5.2.3).

The CSDS algorithm was applied to both the DP and DPNV problems using each of the following evolutionary operators:

- 1-point crossover; and
- Randomised crossover [Jon95].

In order to test for parameter sensitivity the experiments were repeated for:

- 2, 4 and 8 hidden units; and
- weight ranges of [-1,1], [-10,10] and [-100,100].

The full set of parameters is given in Table 5.1.

Parameter	Value
Sub-population size	100
No. of hidden nodes	$\{2, 4, 8\}$
Weight range	$\{[-1, 1], [-10, 10], [-100, 100]\}$
Max. generations	1000
Target fitness	1000
Initial sigmoid node slope ρ	1.0
Initial sigmoid node threshold θ	0.0
Pole failure angle	36°

Table 5.1: Common parameters for the Dual Pole Balancing experiments

5.3.5 Results

Table 5.2 shows the worst, average and best cases for each operator solving the simpler pole balancing task (DP) for 1,000 time steps, averaged over 50 runs. Table 5.3 shows the same comparison for the harder, non-Markovian pole balancing task (DPNV). So as not to bias the averages the results are based only on the successful runs and as such must be interpreted along with the related success rate.

The results are not listed in full but are instead summarised here. Overall the best performance for each operator was seen with the widest weight range of [-100,100]. For both the DP and DPNV problems, crossover achieves a success rate of 100% whereas random crossover (RC) achieves only 24% and 8% for DP and DPNV respectively.

Figure 5.13 shows the average fitness at each generation, taken over 50 runs, including both negative and positive results. It can be seen here that for the simpler problem, crossover produces solutions with increased accuracy and in less time than with the random crossover operator which fails to make significant progress within the generation limit of the experiment. All results are statistically significant with $p < 0.05$.

Figure 5.14 shows a measure of the average diversity of each subpopulation over time, calculated by taking the average pairwise Euclidean distance between subpopulation members, which has then been averaged over all trials. Pairwise Euclidean distance d of N population members with genotype length L is defined as:

$$d = \sum_{j=1}^{j=N-1} \sum_{j'=j+1}^{j'=N} \left(\sqrt{\sum_{i=1}^{i=L} |s_{ij} - s_{ij'}|^2} \right)$$

Here we can see that crossover causes the uniformly initialised subpopulations

Operator	low	high	avg.	s.d.	% success
Crossover	400	8797	3255	1738.75	100%
Random crossover	167	62493	27674	20238.58	24%

Table 5.2: Lowest, highest and average no. of network evaluations taken to solve the Dual Pole With Velocities (DP) task over 50 trials.

Operator	low	high	avg.	s.d.	% success
Crossover	1168	13913	4779	3051.64	100%
Random crossover	8630	62631	40788	24545.98	8%

Table 5.3: Lowest, highest and average no. of network evaluations taken to solve the harder Dual Pole No Velocities (DPNV) task over 50 trials.

to converge rapidly, with mutation maintaining a minimum level of diversity. The random crossover operator maintains a constantly high level of diversity by effectively performing large mutations at each generation. A similar trend appears in the [-10,10] weight range. In the more limited weight space of [-1,1] both operators perform very poorly with the only solutions found by crossover for the simpler DP problem.

5.3.6 Discussion

By comparing the results of applying 1-point crossover to that of the randomised crossover, we can see that in the former case the parent choice (possibly indicating useful sharing of information between individuals) improves the performance of the algorithm, suggesting either

- the presence of building blocks in the genotype, or
- that crossover is macromutation of smaller magnitude than the randomised crossover operator.

If we assume that crossover is acting as a mutation operator with its average magnitude determined by the diversity of the sub-populations then the rapid decrease in diversity (as shown in Figure 5.14) would mean that crossover is behaving like a variable-magnitude mutation operator. The pattern is similar to that of a rapid Simulated Annealing cooling schedule [Kir84].

While testing performance with Jones' macromutational crossover suggests that the benefits of the *idea* of crossover are in effect as well as simply the effects of its *mechanism*, it does not prove that this is so. The truth of this statement also depends heavily of course on what we consider the idea of crossover to be. If we view it from the traditional interpretation of it being successful due to its

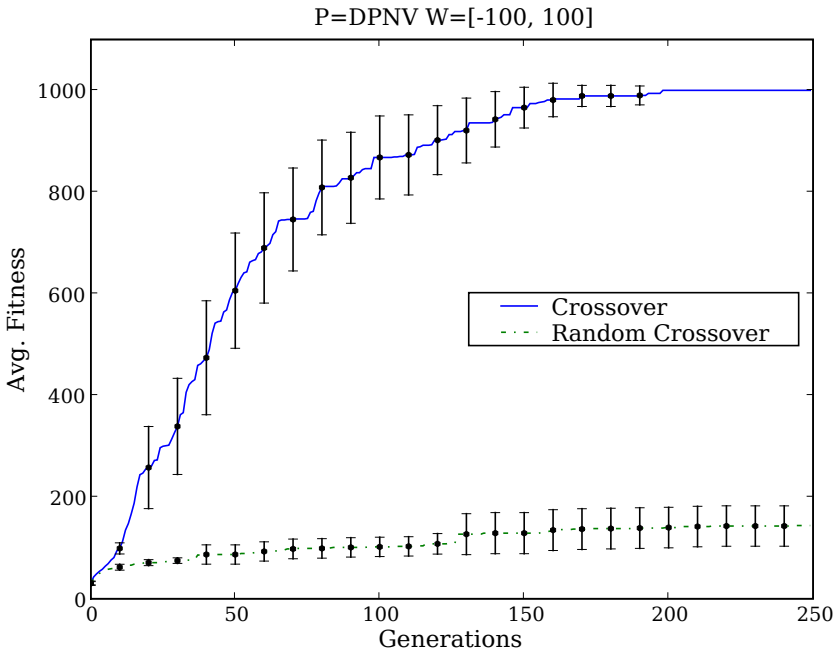


Figure 5.13: Average fitness at each generation for each evolutionary operator on the harder non-Markovian version of the pole balancing task (DPNV) where the controller must keep the two poles balanced without the pole and cart velocity information (synapse weight range [-100, 100]). Here the error bars represent a confidence interval of 95%.

recombination of short, low order, high fitness building blocks then we may be tempted to conclude that this is occurring. This is of course only one of the possible explanations for any increase in fitness however. In this case the test only allows us to say what crossover is *not* doing.

With this kind of feed-forward Neural Network we are already aware that the genotype does not contain obvious building blocks [Yao99]. If we take this as evidence for the idea of crossover not being in effect (building blocks are not being recombined), we may then look at the effect of the mechanism of combining strings alone. Perhaps unsurprisingly the performance is poor, leading us to conclude that the mechanism alone is not particularly useful *in this problem setting*. This leaves us with the conclusion that the benefit cannot be ascribed purely to the mechanism, but that there must be at least one other factor at work.

We propose a view of crossover as a kind of mutation operator, where the average magnitude of its application is determined by the diversity in the population. If we have no diversity in the population, then applications of crossover always result in the same string. In recombining strings from a population with some diversity the resulting strings can be considered to be mutated with respect to each other, with the restriction that the mutations can only be composed of

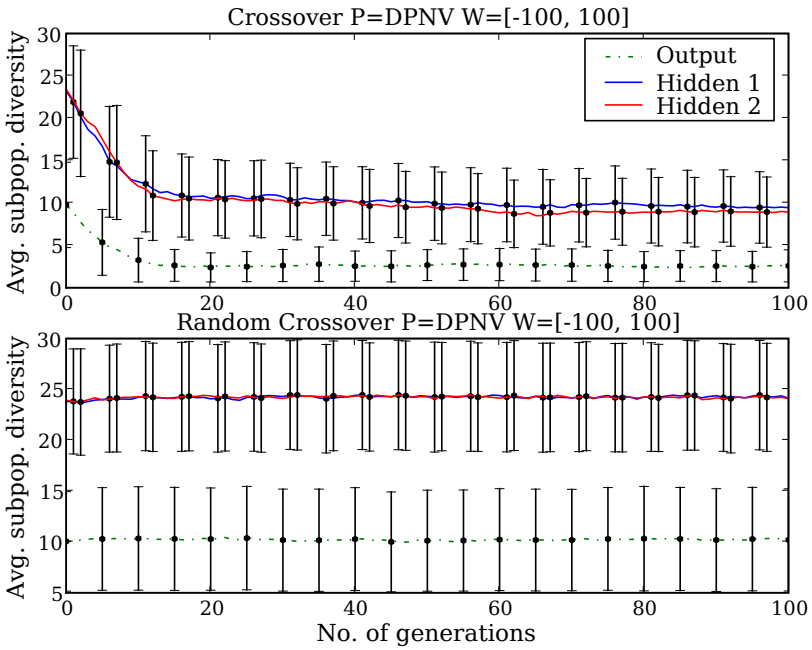


Figure 5.14: Average diversity over all runs for each sub-population over time. The crossover operator causes the sub-population diversity to decrease rapidly in the early stages of evolution in contrast to the randomised crossover which maintains a high level of diversity throughout. Here the error bars represent the standard deviation.

alleles already present in the population.

Given a search space with no obvious building blocks we can therefore view crossover as a form of mutation where its magnitude is determined by the population diversity, and its functions as being one of spreading around the new alleles brought in by mutation. Assuming diversity decreases over time as we converge around a solution, the magnitude of the mutation offered by crossover decreases similarly, offering an effect akin to simulated annealing. As such, crossover may be helping the search process not by recombining high-fitness building blocks but instead broadening the search early on (multiplying the diversity already present in the population) and then later providing smaller and smaller perturbations as the algorithm homes in on a solution. It is possible that this effect could help the algorithm escape local minima early on in the search process.

5.3.7 Future Work

A key limitation of this work is that establishing that crossover is not performing a macromutation tells us only that; it does not mean that crossover is therefore performing some kind of recombination (though this may be the case), it may simply mean that crossover is performing macromutations but on a smaller scale

than those performed here. By recombining fit individuals with random individuals, the average magnitude of mutation will be higher than if two similar networks were recombined. As such we maintain our hypothesis that the utility of crossover may be due to its behaviour as a mutation operator, the magnitude of which is defined indirectly by the diversity of the population. If the diversity of the population decreases steadily (a common pattern for an evolutionary algorithm) then the effect will be similar to that of simulated annealing. For future work we would suggest examining the performance of crossover versus a randomised crossover where the average difference between members of the population and the randomly-generated individuals is controlled for to investigate whether there is a smooth relationship between the usefulness of the operator and the average magnitude of the difference post-application.

An issue worth investigating is then perhaps whether crossover can be replaced by a mutation operator which simply matches the average magnitude of the effect of the crossover operator. As mutation operators can be tuned and do not depend on the diversity of the population as such for their magnitude of effect, this could result in more effective variational operators.

It may however be that the benefits of crossover come from its geometric properties. In crossing over two genotypes a high-dimensional lattice of points of points which can be reached is formed. If points in this lattice are on average of higher fitness than the average of the rest of the search space then crossover will produce more improvements than a more random operator such as the previously-presented random crossover. It may therefore be worthwhile to investigate the average fitness of points reachable by crossover compared to the average fitness of points in the search space to investigate the efficacy of crossover as an evolutionary operator.

5.4 Counting Binary Representation Permutations

In section 5.2 we counted the number of permutations which appeared in runs of an Evolutionary Algorithm which used a discretised real-weighted representation. This representation had been discretised by rounding the weights to one decimal place, which facilitated the counting of permutations.

In this section we repeat this experiment with a binary representation where we vary the number of bits per weight (and so the granularity of the space). Also instead of the pole-balancing problem we now use three classification problems from the UCI dataset repository.

5.4.1 Permutations with a Binary Encoding

Due to their simplicity, binary representations were often used in early work into the evolution of Neural Networks [Yao99]. In this section we count the number of

permuted individuals which appear in simulated evolutionary runs using binary-encoded network genotypes. The encoding scheme that we use is a typical method, outlined in [MF04] and is as follows:

Each weight in the network is encoded as a binary string $\langle b_{k-1}...b_0 \rangle$ where k is the number of bits in the string. We begin by converting this string from base 2 into base 10:

$$(\langle b_{k-1}...b_0 \rangle)_2 = \left(\sum_{i=0}^{k-1} b_i \cdot 2^i \right)_{10} = x'$$

Then given x' we find the corresponding value x within the required range $[l, u]$:

$$x = l + x' \cdot \frac{u - l}{2^k - 1}$$

For example, given the binary string 1101011001, $x' = 512 + 256 + 64 + 16 + 8 + 1 = 857$. Then for a weight range of [-1, 1], $x = -1 + 857 \cdot \frac{1-(-1)}{2^10-1} = 0.67546432$.

The weights for the network are arranged in the same order as with the floating point representation (Section 5.2.3); the only difference is their method of encoding. The recombination operator remains as the standard 1-point crossover operator, though now operating on strings of bits rather than reals. The mutation operator is a bit flip operator with a configurable probability that each bit may be flipped at each application.

5.4.2 Method

We apply an Evolutionary Algorithm based on a typical GA to the problem of weight optimisation for three classification problems. The parameters for the experiment are given in Table 5.4. As we are interested in counting permutations, the principal parameter of interest is the number of bits assigned to each weight. The number of bits per weight in part defines the granularity of the search space, and so greatly affects the probability of the Permutation Problem occurring. We evolve networks using representations with as few as 1 bit per weight, up to 8 bits per weight. The calculated probabilities of Chapter 4 predict that we will only see a significant number of permutations for the case where the number of bits per weight is 1, and only for smaller networks.

5.4.3 Results

Looking at Table 5.5 we can see that the only problem configuration with which permutations were observed was with the UCI Iris problem with 1 bit per weight. This is in line with the predictions made using the probabilities calculated in Chapter 4. It becomes clear why this is so if we look at Figure 4.9 which shows

Parameter	Value
No. of runs	25
Proportion of data for validation	20%
Proportion of data for testing	20%
Problem	{ Cancer, Pima Diabetes }
Population Size	100
No. of Generations	200
Selection type	Tournament($k = 2$)
No. of hidden neurons (N_h)	5
Weight range	(-1, 1)
Bits per weight (bpw)	$\{1, 2, 4, 8\}$
P_{cross}	0.6
P_{mutate}	$\frac{1}{L}$ [a]

[a] L = the genotype string length, which is dependent on the problem.

Table 5.4: Parameters for the binary permutation counting experiment Genetic Algorithm.

that the proportion of permutations in the initial population is very close to zero for all but the smallest networks and coarsest weight granularity.

The number of observed genotypic permutations is quite high for the cases with 1 or 2 bits assigned per weight, though as expected the performance in each case appears to be largely unaffected.

5.5 Discussion

Although this investigation is not immediately concerned with the performance of the resulting networks, we would still like to see reasonable performance so that we can confirm that the Evolutionary Algorithm is functioning as expected. Table 5.6 shows that while the test performance of the networks is not particularly competitive (this is perhaps to be expected given that the EA has not been tuned for this purpose) the training performance is satisfactory, certainly showing that the EA is behaving as expected and producing networks of greater fitness than those of the initial population.

Given the empirical results, combined with those of the theoretical investigation, we can have some confidence that the counts of zero exact-weight phenotypic permutations are representative of other Neuroevolutionary configurations. The

Problem	Phenotypic Avg.	Std.	Genotypic Avg.	Std.
Iris(bpw=1)	0.04	0.20	1273.76	65.64
Iris(bpw=2)	0	0	27.16	6.12
Iris(bpw=4)	0	0	0	0
Iris(bpw=6)	0	0	0	0
Iris(bpw=8)	0	0	0	0
Cancer(bpw=1)	0	0	794.52	57.38
Cancer(bpw=2)	0	0	9.92	3.59
Cancer(bpw=4)	0	0	0	0
Cancer(bpw=6)	0	0	0	0
Cancer(bpw=8)	0	0	0	0
Pima(bpw=1)	0	0	1125.96	69.34
Pima(bpw=2)	0	0	21.60	4.11
Pima(bpw=4)	0	0	0	0
Pima(bpw=6)	0	0	0	0
Pima(bpw=8)	0	0	0	0

Table 5.5: Average no. of permutations counted during an evolutionary run $(N_i = 4, N_o = 3, N_h = 5)(Iris)$

results strongly suggest that exact-weight phenotypic permutations will be extremely rare for realistic network representations and that, given no permutations in the initial population, a typical Evolutionary Algorithm will not actively *create* them.

5.6 Chapter Summary

In this chapter we have presented an empirical investigation which has provided support for the conclusions drawn in Chapter 4. We began by counting the number of permutations in the population of a typical GA used to optimise weights for a Neural Network on a selection of classification problems. The results here again showed that,

- Exact-weight phenotypic permutations will only occur with the simplest of representations, and essentially never with the kinds of representation used in most practical work.
- Genotypic weight permutations are common for extremely simple representations, but do not appear to affect performance significantly as hypothesised.

Problem	Avg. Best Train Error	Std.	Avg. Best Test Error	Std. Test
Iris(bpw=1)	0.06	0.03	0.02	0.06
Iris(bpw=2)	0.04	0.01	0.05	0.04
Iris(bpw=4)	0.02	0.01	0.09	0.06
Iris(bpw=6)	0.03	0.01	0.04	0.05
Iris(bpw=8)	0.02	0.01	0.05	0.03
Cancer(bpw=1)	0.00	0.00	0.11	0.02
Cancer(bpw=2)	0.01	0.00	0.08	0.02
Cancer(bpw=4)	0.01	0.00	0.09	0.02
Cancer(bpw=6)	0.01	0.00	0.10	0.02
Cancer(bpw=8)	0.01	0.00	0.08	0.02
Pima(bpw=1)	0.21	0.01	0.38	0.02
Pima(bpw=2)	0.20	0.00	0.38	0.04
Pima(bpw=4)	0.19	0.00	0.37	0.05
Pima(bpw=6)	0.18	0.00	0.37	0.03
Pima(bpw=8)	0.21	0.00	0.31	0.02

Table 5.6: Average error of the EA used to estimate the no. of permutations during an evolutionary run

The remainder of the investigation dealt with two questions related to the nature and efficacy of crossover in Neuroevolutionary algorithms. In each case the answering of these questions is closely related to the issue of the Permutation Problem as it is only when permutations are recombined that they are though to be a serious problem[8]. To this end we first investigated how Price's Equation could be used to profile an Evolutionary Algorithm, with a focus on characterising the contribution of crossover to the overall algorithm performance. We then asked whether Neuroevolutionary crossover could be better thought of as a macromutation rather than a recombination operator, concluding that this appears to be a more plausible interpretation, though the appropriate magnitude of mutation is yet to be fully ascertained.

Finally we repeated the permutation-counting experiment for three classification problems this time with a variable-granularity binary representation. In all cases very few or no permutations were counted, providing strong evidence for the

[8]However, as we will see in Chapter 6, the redundancy introduced by the position invariance of the representation poses a potential problem with or without recombination, i.e. that of efficiency.

case that exact-weight phenotypic permutations are not a serious practical concern when evolving Neural Networks. The experiments were however conducted using one specific crossover operator; similar experiments using alternative crossover operators should be conducted in order to determine how general the results of this chapter are.

We have so far investigated the problem of recombining networks which are exact permutations of each other. While the event that two such networks are recombined appears low in practise, there still exists the problem of the redundancy caused by the symmetry in the search space. This aspect of the problem is explored in the next chapter.

Chapter 6

Multiset Search Framework

6.1 Introduction

In Section 4.4 it was noted that the symmetry in a particular class of Neural Network representations causes the search space to be massively redundant, to be composed of virtually the same search neighbourhood multiplied on a massive scale. We can take any network, permute its hidden neurons and 'jump' to another point in the search space with equal fitness. We have then moved a potentially large distance in the genotype space but have not moved at all in the phenotype space. Furthermore, we are effectively in the *same* search space as before. Any operation we might perform on a particular individual (for example a mutation of its weights) has an exactly equivalent operation in all of its 'clone' search spaces found by permuting the neurons of the network. This results in an extreme form of redundancy whereby each point in the phenotypic space is a member of anything from 1 to $N_h!$ neighbourhoods in the genotypic space. In the previous chapter we demonstrated that the Permutation Problem, specifically the recombination of two networks which are permutations of each other, does not occur often and is therefore not a serious concern. The symmetry in the search space does however result in a representational redundancy, the removal of which may improve the search process.

The search space consists of a fixed number of *neutral networks*[1]. Given any point in the search space we can traverse the network of points of equal fitness formed by its permutations and always obtain the same phenotype. While neutral networks and nearly neutral networks can be beneficial when searching complex spaces [NE98], in this case the neutral network appears to offer no benefits owing to the symmetrical nature of the neighbourhood at each point on the network.

What we would like to do is to evaluate as many *unique* Neural Networks as

[1]At this point we must be careful not to confuse the concept of networks of (near) equal fitness points in the search space and the Neural Networks that we are optimising. In this chapter we will always capitalise and write out in full 'Neural Network' so as to avoid the unfortunate fact that not only do 'neural' and 'neutral' look similar, the two types of network have the same acronym (NN).

	N_h=**1**	N_h=**2**	N_h=**3**	N_h=**4**	N_h=**5**	N_h=**6**	N_h=**7**
UCI Iris	0%	50%	83.33%	95.83%	99.17%	99.86%	99.98%
UCI Cancer	0%	50%	83.33%	95.83%	99.17%	99.86%	99.98%

Table 6.1: Redundancy in the search space for different problem configurations and sizes of network. We note that the proportion of redundancy in the search space is dependent on the number of hidden neurons rather than the problem space (number of inputs and outputs).

possible. Given that each neutral network represents one Neural Network we wish to move freely from one neutral network to another, only evaluating one member of each. As we have seen in Section 4.5, each network has a variable number of permutations depending on how many unique neurons[2] it contains. As such, the neutral networks are also of different sizes. The network size can be calculated however, just as the number of permutations can be, allowing the search space to be 'mapped' without running a search algorithm. It is this ability to map the search space that will allow us to remove the redundancy.

Table 6.1 shows how the redundancy of the space increases as we add hidden neurons to a network. We might assume that, due to this redundancy, the search process is grossly inefficient and that it could be improved by searching the canonical search space of unique networks. The question is then how to modify the representation or search process such that time is not wasted evaluating the same solutions multiple times. Ideally, rather than modifying the search operators to simply correct for the redundancy, the representation would simply not contain the redundancy. Were this to be achieved, existing search algorithms could be applied to this search space without major modification.

One method which has been tested previously is that of working with canonical representations of networks [Han93, Thi96]. A potential pitfall of this approach is that the process of converting a network to its canonical form may add noise or deception to the problem.

In this chapter we propose a novel way to search the space of unique networks directly. The method was inspired by the underlying structure of the equation for the probability of the Permutation Problem occurring in the initial population (Chapter 4) which is also based on multisets. Working with a discrete space for simplicity, we begin by labelling every possible neuron with a number. Now, rather than select weights to form a network, we select whole neurons. The problem is now to find the right combination of neurons to solve the problem. This recasting of the problem as one of combinatorial optimisation is similar in nature to that of García-Pedrajas et al. [GPOBHM05] though the approaches differ in how this recasting is achieved. We begin by presenting a genotypic representation for the

[2]By unique here we mean unique in terms of its weights, not necessarily its functionality or role in the network.

space of multisets. Each individual in this representation is a unique multiselection (selection with repetition) of neurons. As we are dealing with multisets and the aim is to eliminate permutations and therefore redundancy, the representation is order independent and so permutation free. We then define and implement a set of mappings from multisets to actual network definitions (weights), and back again. Using this framework we construct a Neuroevolutionary algorithm that searches the space of unique networks.

6.2 Related Work

An early work which aimed to remove the redundancy present in the Neural Network search space was that of Thierens [Thi96]. Thierens divided past work on the problem into principally three camps, namely those that:

- are heuristic in nature [Rad93, TSVM93],
- reduce or eliminate the role of crossover [WSB90], or
- espouse the Evolutionary Programming Paradigm [FFP90, ASP94].

Thierens proposed two simple transformations that, when applied to any permutation of a Neural Network, would map it to a single member of that group of permutations. The first transformation was a rule for when to flip the sign of weights so that all symmetric solutions found by flipping the signs of weights were reduced to one representation. The second rule sorted the neurons by their bias weight, enforcing a canonical order on the neurons. Used together, these transformations could be used to take any network and reduce all of its symmetries to one single point in the genotype space.

Ultimately this approach involves an *online transform* on the individuals and not the representation / search space itself. Standard evolutionary operators can therefore reach individuals which are not of canonical form; these individuals would be rejected and 'snapped back' into the enforced ordering. As this kind of transform can cause individuals to make large jumps in the genotype space it could potentially be a source of noise or deception in the search process.

More recent work on solving, or at least reducing the (supposed) negative effects of the Permutation Problem is that of García-Pedrajas et al. where the problem of Neural Network optimisation is recast as a problem of combinatorial optimisation [GPOBHM05]. The motivation behind this work is that while the problem can be avoided by relying on mutation alone, a new kind of crossover is necessary as, "crossover is the most innovative operator within the field of evolutionary computation" [GPOBHM05]. The authors suggest that while approaches based on Evolutionary Programming have demonstrated success in the past, the search process could be improved by additionally making use of the crossover operator. In order to do so they propose a modification to the crossover process to reduce (but not completely remove) the potential negative effect of the Permutation Problem.

The first modification is to recombine whole neurons and never 'mix' their internal representations. The positive effects of only performing crossover at neuron boundaries, or rather the low correlation between recombined neuron definitions has been demonstrated previously [GPOBHM05]. Now, considering neurons as indivisible elements of the network we can consider networks as selections of neurons. An example demonstrating the Permutation Problem is given using two networks which contain the same neurons in a different order, $\{a, b, c, d\}$ and $\{c, d, a, b\}$. The crossover of these two networks could then produce offspring such as $\{a, b, a, b\}$ and $\{c, d, c, d\}$. The authors repeat the contention of [ASP94] that the loss of some particular neurons in each network results in a loss of computational ability and is therefore highly likely to result in a reduction in fitness [GPOBHM05]. This example is somewhat contrived however as no hypothesis is given for how we have come to recombine two networks which contain the same neurons in a different order.

Their solution to this particular problem is to form offspring from the union of the neurons of each parent. Offspring networks are constructed by selecting neurons from this set, with the aim of selecting the most useful neurons. This is presented in contrast to the way that standard crossover can 'lose' useful neurons by repeating some and omitting others; here there is always the opportunity for each neuron to progress to the next generation.

Networks constructed in this way are then trained on the problem using backpropagation to optimise the weights. Here the the purpose of the Evolutionary Algorithm is primarily to select promising combinations of neurons that are then optimised by the backpropagation routine. This is necessary as while the neurons may have previously been elements in successful networks, if taken out of that context and placed in another network the output weights will require retraining in order to balance the contribution of each in light of the effect of the others.

This approach of finding the best combination of neurons shows competitive results on various problems from the UCI Machine Learning database. While this is evidence in support of the efficacy of the presented algorithm it does not say much about the occurrence or lack thereof of the Permutation Problem. It is argued from an intuitive standpoint that the proposed approach, "greatly alleviate[s] most of [the Permutation Problem's] harmful effects", however this is not checked for explicitly.

A useful principle that we can take away from this work however is that avoiding enforcing an order on the hidden neurons of a network can help to circumvent the redundancy that is introduced by such order. However, this approach requires significant changes to a typical Evolutionary Algorithm in order to work with networks which are represented as sets of neurons. Additionally, while it may not necessarily be a disadvantage, the approach as it stands is limited to sets of neurons, disallowing the repetition of neurons. The consequences of this limitation are not obvious; whether this is a disadvantage or an advantage would need to be determined through further investigation.

A more formal treatment of the problem is given by Radcliffe [Rad93] where

both set and multiset approaches are defined on a theoretical level but a method of implementation is not provided. In this work Radcliffe provides an insight which does not appear to have become part of the mainstream body of knowledge in the field of Neuroevolution. Particularly, that the number of optima in the search space may not increase by the same factor as the rest of the search space. For example, if the optimum in the unique network space is composed by n copies of the same neuron then there will only be one optimum in the redundant space. If the optimum instead has n *distinct* neurons, then there will be $n!$ optima in the redundant space. The issue of how similar the redundant and non-redundant spaces are is explored in Section 6.4.4.

6.3 A Look at the Search Space

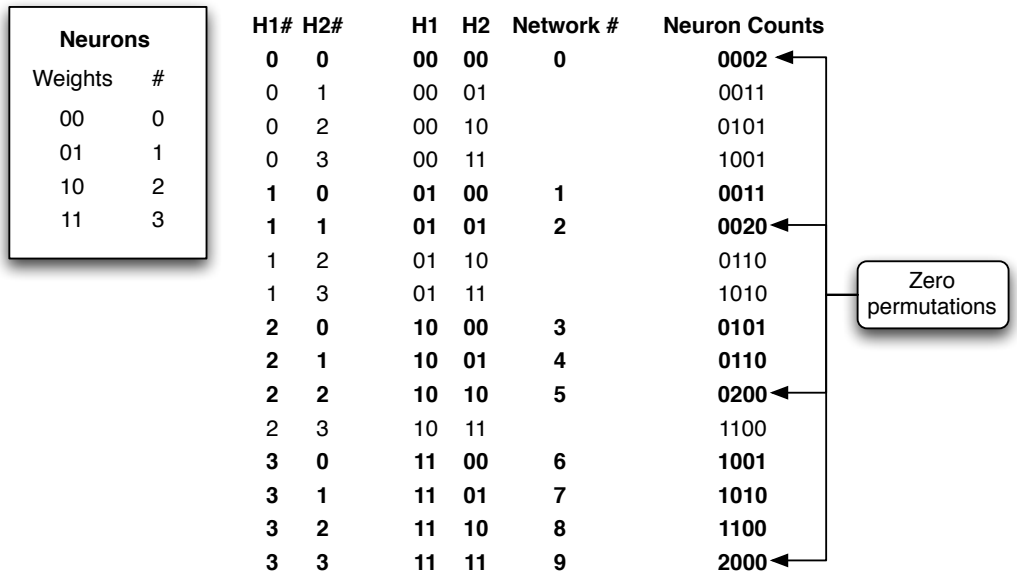


Figure 6.1: The complete search space formed by arranging 4 neurons into networks of 2 neurons ($N_h = 2$). Multisets are highlighted in bold; multisets with no possible permutations are also indicated.

In this section we examine a simple search space in order to highlight the redundancy that we wish to remove, and in doing so identify the unique points that we are interested in. Looking at Figure 6.1 we have a collection of four neurons which have been arranged into all possible networks[3] with $N_h = 2$.

In this example each neuron is a single weight, encoded by two bits. We therefore have four possible neurons which have been numbered from 0 to 3. The

[3]While the number of neurons and weights is of course unrealistically small, a typical search space will often be of the same form as that presented here, simply with a greater number of weights and bits per weight.

next two columns show the $4^2 = 16$ ways of arranging the four neurons in terms of their identifying number and weights respectively. Each line in bold represents a unique multiselection of neurons, each of which has now been given a network number from 0 to 9 (as there are $\left(\!\binom{4}{2}\!\right) = 10$ unique multiselections). The final column counts the number of occurrences of each neuron in each network; it is this representation that highlights the redundancy. The neuron count is indexed from 0 from the right, so the interpretation of 0002 is, "two of neuron 0, zero of neuron 1, zero of neuron 2 and zero of neuron 3". Neuron counts which contain a '2' only have one representative in the search space[4]. Those with two '1' counts have one other equivalent network in the search space. We can see this if we pick any of the numbered lines in bold which does not contain a '2' we can find another network with the same neuron count; this is the redundancy of the space.

Looking at the pattern of bold vs. plain lines in Figure 6.1 we can begin to discern a pattern which may help us identify these multisets once and only once: if we group the bold lines together we first have one line, then two, then three, then four. Similarly, the 'gaps' between them, formed by non-bold lines contain three lines, then two lines, then one. Understanding the nature of this pattern, and crucially discovering the pattern for numbers of hidden neurons greater than two will be crucial when devising a method for designing a search space around these unique multisets.

Given that there are two copies of every multiset with distinct neurons, how have we chosen which multiset to number as a unique network? Looking at the networks in terms of their numbered neurons, we note that the bold lines contain networks which, when read left-to-right, are composed of neuron identifiers in strictly *non-increasing* order. This is illustrated in Figure 6.2. Under this constraint, when selecting between multiple equivalent multisets we choose the only one that satisfies this constraint. For example when choosing the representative network for multiset '0011' we reject the network defined by neurons '0 1' (as the digits are increasing) and instead select the network '1 0' (as the digits are non-increasing). This ordering will be crucial when forming a $1:1$ mapping in both directions between the search space and representation space as it ensures that we always select the same network for a given multiselection of neurons.

6.4 Multiset Representation

In this section we define the Neural Network representation, and so the search space of interest. We begin by reviewing the redundancy of the space. We then define the neuron representation and extend this to a representation for multisets of neurons (networks). We conclude this section by presenting an intuitive method for freely navigating the space of multisets.

The problem with a 'traditional' mapping of the network weights to the

[4]This is the case because we select two hidden neurons and if they are both the same, there are naturally no permutations.

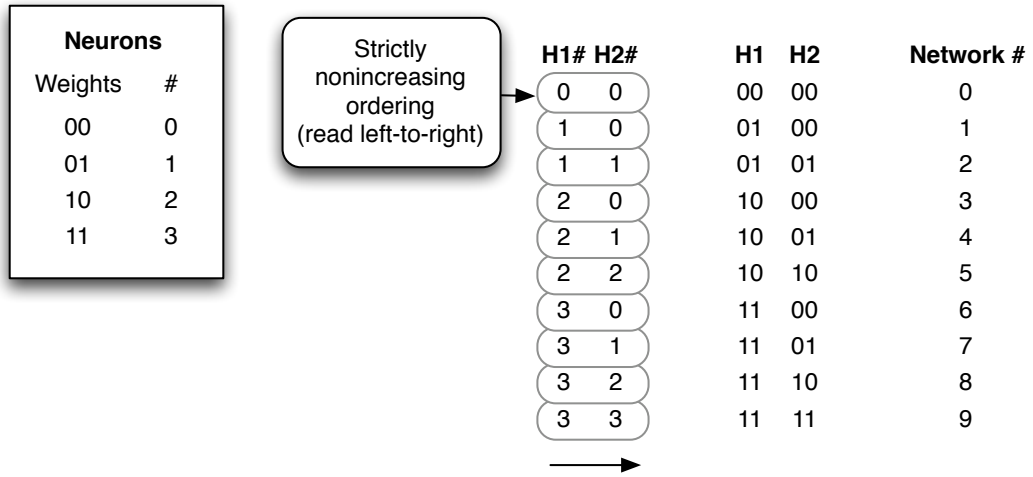


Figure 6.2: All unique multisets formed by arranging 4 neurons into multiselections of 2 neurons ($N_h = 2$). We note that the neuron identifiers are always in non-increasing order when read left-to-right.

genotype is that it there is a many-to-one mapping from genotype to phenotype. If at every generation we map each network to its canonical representation we avoid recombining networks which are permutations of each other but do so at the cost of introducing 'jumps' in the search, where networks are shifted in the genotype space suddenly.

Instead of searching the original search space we propose searching a space which simply does not contain this redundancy: the space of neuron multisets. We have seen that the function of a fully-connected feedforward network depends only on which neurons are present in the hidden layer, but not on their order. Finding a canonical representation by enforcing certain ordering constraints so that we may map equivalent networks to the same genotypic string addresses but does not *remove* the redundancy from the search space.

Our aim is to define a search space which contains each multiselection (selection with replacement) of the possible neurons which can be defined by the search space, but does not contain permutations of these multiselections. Taking the UCI Iris problem as an example, we have 4 inputs and 3 outputs, one for each class. Each hidden neuron will therefore have 7 weights[5]. Without loss of generality we define the weights to be binary-encoded reals encoded using 8 bits. We therefore have $\phi = (2^8)^7 = 7.2 \times 10^{16}$ unique strings of weights from which we can form neurons[6]. We can model this generally as,

[5]Without loss of generality we we do not include a bias input at this stage. A bias input can be included as an extra input of constant value without modification to the framework.

[6]Depending on the neuron transfer function, and to some extent on the fitness function not all of these neurons may be considered unique; some may have functionality that is so similar as to be effectively indistinguishable. At this stage we assume that each encodes a distinct mapping though this may not hold in practice.

$$\phi = \left(2^{bpw}\right)^{(N_i+N_o)}, \tag{6.1}$$

where bpw is the number of bits per weight, N_i the number of network inputs and N_o the number of network outputs. With the number of possible neurons known we can calculate the total number of possible networks that can be formed and compare it to the total number of unique networks (that is, unique multisets of neurons). The total number of networks is then,

$$\Omega = (\phi)^{N_h}. \tag{6.2}$$

Without loss of generally we set $N_h = 5$. For our UCI Iris example this then gives us

$$\Omega = \left(7.2 \times 10^{16}\right)^5 = 1.9 \times 10^{84} \text{ networks.}$$

Considering only the number of unique ways of selecting neurons irrespective of their order, we then have

$$\Psi = \left(\!\!\binom{\phi}{N_h}\!\!\right) = 1.6 \times 10^{82} \text{ unique networks,}$$

where

$$\left(\!\!\binom{\phi}{N_h}\!\!\right) = \binom{\phi + N_h - 1}{N_h}.$$

The space of unique networks forms

$$100 \cdot \frac{\Psi}{\Omega} = 0.83\%$$

of the full search space. As Table 6.1 demonstrates, this redundancy increases rapidly as the number of hidden neurons increases.

6.4.1 Neuron Representation

We begin by first identifying each neuron in our solution space. Each neuron should have a unique identifier, which for simplicity we choose to be a non-negative integer $n \in \mathbb{N}_0$. To illustrate the strategy behind the mapping we take a simple example with a network with 2 inputs and 1 output. Each hidden neuron is therefore defined by 3 weights. Without loss of generality we limit the weights to be one of two values. We denote these values by '0' and '1' but these are simply labels; the actual weight values may be different[7]. We therefore have $\left(2^1\right)^{2+1} = 8$ possible neurons, as shown in Table 6.2.

The interpretation in constructing a neuron with definition $\langle 100 \rangle$ is "one copy of weight 1 and two copies of weight 0, in this order". If we wish to find out the

[7]For example, if we set our weight range to be $\{-1, 1\}$ then '0' would map to -1 and '1' would map to 1

Neuron Number	Representation (base-2)
0	$\langle 000 \rangle$
1	$\langle 001 \rangle$
2	$\langle 010 \rangle$
3	$\langle 011 \rangle$
4	$\langle 100 \rangle$
5	$\langle 101 \rangle$
6	$\langle 110 \rangle$
7	$\langle 111 \rangle$

Table 6.2: The eight possible neuron definitions for three-weight neurons in a binary weight space.

actual weights for a particular neuron, e.g. the third neuron in the sequence, we can simply evaluate the sequence three times to reach it. For a realistic neuron space however this would be prohibitively inefficient; a way to jump directly to a given neuron directly is required. As networks will be formed by selecting neurons rather than weights, we need a way to map directly from the integer identifier of the neurons to their respective weight strings. Given an identifier, e.g. '5' (the sixth neuron), we can calculate the weight string directly using modulo arithmetic. The sequence of ones and zeros in this case can be calculated as follows:

$$\left\langle \left\lfloor \frac{5}{4} \mod 2 \right\rfloor, \left\lfloor \frac{5}{2} \mod 2 \right\rfloor, \left\lfloor \frac{5}{1} \mod 2 \right\rfloor \right\rangle = \langle 1, 0, 1 \rangle.$$

This is calculated in the general case (in reverse order, from right to left) as

$$\sum_{p=0}^{l-1} \frac{x}{b^{(l-p-1)}} \mod b \tag{6.3}$$

where

- x is the $n + 1^{\text{th}}$ neuron in the sequence (from right to left),
- b is the 'base' (the number of possible weights),
- p represents the position in the sequence,
- and l the total number of hidden neurons.

We are simply mapping the integer to its (in this case) binary representation. We can map the decimal numbers to any base required; the base is determined by the number of possible weights. We therefore interpret the sequence $\langle 1, 0, 1 \rangle$

as a template telling us which weights to select to satisfy it. If our weights were $\{-1, 1\}$ then this neuron would be defined by the weights $\langle 1, -1, 1\rangle$.

6.4.2 Network Representation

Now that we have a means of identifying neurons by a unique identifier, we can look at compositions of neurons into networks. We cannot use the same scheme, as this would include all networks and therefore all permutations. While we are interested in all neurons, we wish to encode only for unique multisets of the neurons to eliminate the redundancy.

We begin by defining the method of encoding the multisets. The expanded form of the encoding counts the number of instances of each neuron in the network. As there will typically be many thousands of neurons and we will only be selecting a small fraction, this representation will be very sparse, i.e. mostly zero. We illustrate this approach with an example where there are four neurons in the representation space and the network is composed of $N_h = 2$ neurons. The list of all $\left(\!\binom{4}{2}\!\right) = 10$ possible multisets is as follows:

$$\langle\langle 0002\rangle, \langle 0011\rangle, \langle 0020\rangle, \langle 0101\rangle, \langle 0110\rangle, \langle 0200\rangle, \langle 1001\rangle, \langle 1010\rangle, \langle 1100\rangle, \langle 2000\rangle\rangle$$

The individual digits will always sum to N_h as that will mean that the right number of neurons has been selected. This form of the representation tells us which neurons to use in constructing a network (order is unimportant). When constructing the network we read the digits from left to right and read off how many of each neuron is required to build the network. If we take the neurons to be indexed from $l - 1$ to 0 from left to right then we could say the the first network is composed of 2 copies of neuron 0; the next network is composed of one copy of neuron 1 and one copy of neuron 0, and so on.

This representation will be extremely sparse in general as the number of neurons will typically be many orders of magnitude larger than N_h. We can produce a compact form of this representation that can be used to uniquely identify each multiset by interpreting the network specification as a ternary (base-3) number; the resulting base-10 integer is our network identifier which not only identifies the multiset but contains its definition in a manner related to that of *combinadic numbers*. This interpretation gives us the mapping shown in Table 6.3. The base is decided by the number of hidden neurons so the base will always be $N_h + 1$ as N_h is the maximum number that can appear in any of the positions.

Suppose now that we wish to search this space of multisets and we further wish to use an existing algorithm such as Evolutionary Programming or an Evolutionary Strategy. If we use the expanded (base-3) representation then discretised Gaussian mutation (for example) is likely to produce infeasible solutions often as the total of all digits must sum to N_h[8]. The same problem will also occur frequently

[8]If we increase the count of one neuron then we *must* decrease the count of another neuron and the nearest non-zero neuron count may be thousands of positions away on the string.

Network Number	Representation (base-3)	Representation (base-10)	Step Size
0	⟨0002⟩	2	-
1	⟨0011⟩	4	2
2	⟨0020⟩	6	2
3	⟨0101⟩	10	4
4	⟨0110⟩	12	2
5	⟨0200⟩	18	6
6	⟨1001⟩	28	10
7	⟨1010⟩	30	2
8	⟨1100⟩	36	6
9	⟨2000⟩	54	8

Table 6.3: The mapping from network number to its sparse and compact representations. The step size between the base-10 representation is shown to illustrate that there is seemingly no obvious pattern to its progression.

with crossover. To mitigate this we may instead search the compressed (base-10) representation. A population of networks would simply be a tuple of multiset identifiers. At this stage we do not consider how to perform crossover (this is explored in detail in Section 6.6).

A problem with the base-10 representation is that the step size between multisets is not constant. How do we traverse from the first to the second multiset, and then to the third, up the last? We can convert the base-10 representation into the base-3 representation that tells us which neurons to select, but in order to do so we need to determine how to generate the sequence so that we may enumerate each multiset only once and in order. A further difficulty is that we must be able to jump to any point in the sequence; for realistic problem spaces enumerating all networks in order to find a particular network in the sequence will be prohibitively expensive.

For our example with four neurons and $N_h = 2$ it turns out that inverting the sequence of triangular numbers[9] allows us to to map from a natural number

[9]Weisstein, Eric W. "Triangular Number." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/TriangularNumber.html

Network #	**Powers**	**Multiset #**
0	$\langle 0\ 0 \rangle$	$3^0 + 3^0 = 2$
1	$\langle 1\ 0 \rangle$	$3^1 + 3^0 = 4$
2	$\langle 1\ 1 \rangle$	$3^1 + 3^1 = 6$
3	$\langle 2\ 0 \rangle$	$3^2 + 3^0 = 10$
4	$\langle 2\ 1 \rangle$	$3^2 + 3^1 = 12$
5	$\langle 2\ 2 \rangle$	$3^2 + 3^2 = 18$
6	$\langle 3\ 0 \rangle$	$3^3 + 3^0 = 28$
7	$\langle 3\ 1 \rangle$	$3^3 + 3^1 = 30$
8	$\langle 3\ 2 \rangle$	$3^3 + 3^2 = 36$
9	$\langle 3\ 3 \rangle$	$3^3 + 3^3 = 54$

Table 6.4: List of the 10 networks with their corresponding powers which make up each multiset number.

$n \in \mathbb{N}_0$ to its unique multiset number m:

$$ntrinv(n) = \left\lfloor \frac{\sqrt{1+8n}-1}{2} \right\rfloor \tag{6.4}$$

$$trinv(n) = n - \binom{\left\lfloor \frac{1}{2} + \sqrt{2+2n} \right\rfloor}{2} \tag{6.5}$$

$$m(n) = 3^{ntrinv(n)} + 3^{trinv(n)} \tag{6.6}$$

At this level of dimensionality the sequence is basically defined by a self-counting sequence (n appears n times[10]). The dimensionality is determined by the number of hidden neurons in the network. In this case we have $N_h + 1 = 3$ so we are dealing with triangular geometry. If we add another neuron then we must extend the above calculation to tetrahedral geometry. Every identifier is always composed of l powers of $(l + 1)$. In this case, $l = 2$ so we have two powers of three.

It is however unclear how this calculation can be extended to higher dimensions. A conceptually simpler and general method for calculation of this sequence is to

[10]Weisstein, Eric W. "Self-Counting Sequence." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Self-CountingSequence.html

use nested iteration:

$$A_2(n) = \sum_{i_0=0}^{n-1} \sum_{i_1=0}^{i_0} 3^{i_0} + 3^{i_1} \tag{6.7}$$

$$A_3(n) = \sum_{i_0=0}^{n} \sum_{i_1=0}^{i_0} \sum_{i_2=0}^{i_1} 4^{i_0} + 4^{i_1} + 4^{i_2} \tag{6.8}$$

$$A_l(n) = \sum_{i_0=0}^{n} \sum_{i_1=0}^{i_0} \cdots \sum_{i_{l-3}=0}^{i_{l-4}} \sum_{j=0}^{l-2} (l-1)^{i_j} \tag{6.9}$$

For example the sequence M_2 for our current example (listed in Table 6.4) can be calculated (with number of neurons available for selection $n = 4$) as

$$\begin{aligned} M_2(n) &= \sum_{i_0=0}^{n-1} \sum_{i_1=0}^{i_0} 3^{i_0} + 3^{i_1} \\ &= (3^0 + 3^0) + (3^1 + 3^0) + (3^1 + 3^1) + (3^2 + 3^0) + (3^2 + 3^1) + \\ &\quad (3^2 + 3^2) + (3^3 + 3^0) + (3^3 + 3^1) + (3^3 + 3^2) + (3^3 + 3^3) \\ &= 2 + 4 + 6 + 10 + 12 + 18 + 28 + 30 + 36 + 54. \end{aligned}$$

When extended to the case where we have $N_h = 3$ we now calculate the sequence M_3 (still with $n = 4$) as

$$\begin{aligned} M_3(n) &= \sum_{i_0=0}^{n-1} \sum_{i_1=0}^{i_0} \sum_{i_2=0}^{i_1} 4^{i_0} + 4^{i_1} + 4^{i_2} \\ &= (4^0 + 4^0 + 4^0) + (4^1 + 4^0 + 4^0) + (4^1 + 4^1 + 4^0) + \\ &\quad (4^1 + 4^1 + 4^1) + (4^2 + 4^0 + 4^0) + (4^2 + 4^1 + 4^0) + \\ &\quad (4^2 + 4^1 + 4^1) + (4^2 + 4^2 + 4^0) + (4^2 + 4^2 + 4^1) + \\ &\quad (4^2 + 4^2 + 4^2) + (4^3 + 4^0 + 4^0) + (4^3 + 4^1 + 4^0) + \\ &\quad (4^3 + 4^1 + 4^1) + (4^3 + 4^2 + 4^0) + (4^3 + 4^2 + 4^1) + \\ &\quad (4^3 + 4^2 + 4^2) + (4^3 + 4^3 + 4^0) + (4^3 + 4^3 + 4^1) + \\ &\quad (4^3 + 4^3 + 4^2) + (4^3 + 4^3 + 4^3) \\ &= 3 + 6 + 9 + 12 + 18 + 21 + 24 + 33 + 36 + 48 + 66 + \\ &\quad 69 + 72 + 81 + 84 + 96 + 129 + 132 + 144 + 192. \end{aligned}$$

For any realistically-sized search space however, enumerating the full sequence of network identifiers is intractable. Similarly, evaluating the sequence in order until we read the right identifier will be too slow in practise. In order for this to be practically viable there must be a method in place to look up network identifiers in

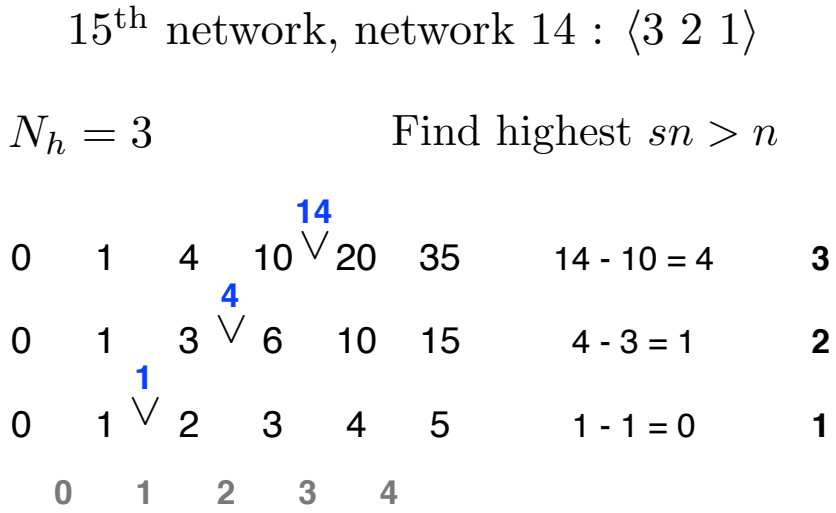


Figure 6.3: Finding the neuron identifiers for the 15th network in a network space where $N_h = 3$.

a more direct manner. In the following section we present computational methods which efficiently map from multiset identifiers to neuron definitions and back.

6.4.3 Jumping to the n^{th} Multiset

In a realistic search space we will often have many millions of possible multisets. A method such as that presented so far will be computationally intractable for such a space; for the method to be practical we need to be able to jump directly to any multiset in the space without enumerating all multisets before it in the sequence.

The question is then, which neurons make up the n^{th} multiset? We can use the relationship between multisets and certain geometric sequences (e.g. triangular, tetrahedral) to jump more or less directly to the multiset definition that we require. We explain the method intuitively in this section and then algorithmically in Section 6.5.

Table 6.5 lists the mapping from consecutive network numbers to the powers of 4 that make up its network definitions. We note that the powers are simply the numbers we assigned as the neuron identifiers. As previously highlighted we cannot expect to be able to enumerate all networks in this way. While we do not yet have a method for calculating the neuron identifiers for a given multiset directly, we now present a method which requires significantly fewer operations, thus allowing for a computationally feasible implementation.

We begin the demonstration of the method by finding the neuron identifiers for network #15. This corresponds to #14 in Table 6.5, which has neuron identifiers ⟨3 2 1⟩. Figure 6.3 demonstrates the method, which we now explain informally. We begin by taking the network identifier (14) and finding the interval or gap in

Network #	**Powers**	**Multiset #**
0	$\langle 0\ 0\ 0 \rangle$	$4^0 + 4^0 + 4^0 = 3$
1	$\langle 1\ 0\ 0 \rangle$	$4^1 + 4^0 + 4^0 = 6$
2	$\langle 1\ 1\ 0 \rangle$	$4^1 + 4^1 + 4^0 = 9$
3	$\langle 1\ 1\ 1 \rangle$	$4^1 + 4^1 + 4^1 = 12$
4	$\langle 2\ 0\ 0 \rangle$	$4^2 + 4^0 + 4^0 = 18$
5	$\langle 2\ 1\ 0 \rangle$	$4^2 + 4^1 + 4^0 = 21$
6	$\langle 2\ 1\ 1 \rangle$	$4^2 + 4^1 + 4^1 = 24$
7	$\langle 2\ 2\ 0 \rangle$	$4^2 + 4^2 + 4^0 = 33$
8	$\langle 2\ 2\ 1 \rangle$	$4^2 + 4^2 + 4^1 = 36$
9	$\langle 2\ 2\ 2 \rangle$	$4^2 + 4^2 + 4^2 = 48$
10	$\langle 3\ 0\ 0 \rangle$	$4^3 + 4^0 + 4^0 = 66$
11	$\langle 3\ 1\ 0 \rangle$	$4^3 + 4^1 + 4^0 = 69$
12	$\langle 3\ 1\ 1 \rangle$	$4^3 + 4^1 + 4^1 = 72$
13	$\langle 3\ 2\ 0 \rangle$	$4^3 + 4^2 + 4^0 = 81$
14	$\langle 3\ 2\ 1 \rangle$	$4^3 + 4^2 + 4^1 = 84$
15	$\langle 3\ 2\ 2 \rangle$	$4^3 + 4^2 + 4^2 = 96$
16	$\langle 3\ 3\ 0 \rangle$	$4^3 + 4^3 + 4^0 = 129$
17	$\langle 3\ 3\ 1 \rangle$	$4^3 + 4^3 + 4^1 = 132$
18	$\langle 3\ 3\ 2 \rangle$	$4^3 + 4^3 + 4^2 = 144$
19	$\langle 3\ 3\ 3 \rangle$	$4^3 + 4^3 + 4^3 = 192$

Table 6.5: The 20 networks with three hidden neurons, each power represents the identifier for a particular neuron.

the first sequence that 14 fits into. We are looking for the number which is just less than 14. In this case it is 10; we note that we are in gap 3, subtract the 10 from 14 to get 4, and repeat with the next sequence. As we move through the sequences we obtain another power/identifier by counting the number of the gap that our network identifier fits into.

We then repeat this process to find the neuron identifiers (powers) for the 19^{th} network (Figure 6.4).

Why does this work?

We have demonstrated *how* this method allows us to jump more or less directly to the powers which make up the multiset number of a given network, but have not explained why we use the sequences of Figures 6.3 and 6.4 to achieve this

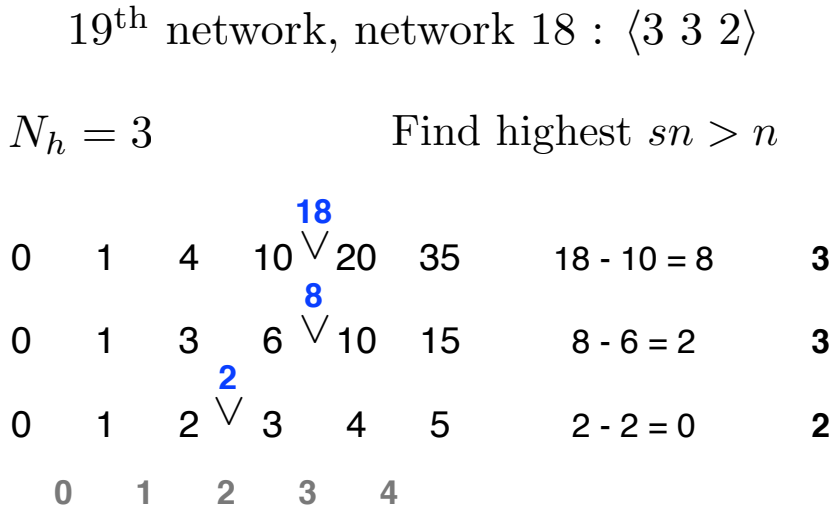


Figure 6.4: Finding the neuron identifiers for the 19th network in a network space where $N_h = 3$.

or why this method works. In mapping the search space in this manner, so that consecutive network numbers map to a sequence of multisets, we are taking advantage of a relationship between multisets and certain geometric sequences, starting with the triangular numbers. The triangular numbers are calculated as

$$\sum_n \binom{n+1}{2} = 1 + 3 + 6 + 10 + 15 + 21 + 28 + 36 + 45 + 55 + \ldots$$

The n^{th} triangular number is equivalent to the number of ways to choose with repetition 2 items from a collection of n items. If we go up one level of dimensionality we have the sequence of tetrahedral numbers, the n^{th} member of which is the same as the number of ways to choosing with repetition 3 items from a collection of n items. In higher dimensions, we refer to these sequences as sequences of *simplex* numbers. Generally, the n^{th} simplex number for dimension d, $sn(n, k)$ is defined as being

$$sn(n, d) = \binom{n+d-1}{d}. \tag{6.10}$$

We can can replace the dimension with the number of neurons in our network N_h, which gives us the starting sequence within which to find our first neuron identifier (power), giving us

$$sn(n) = \binom{n+N_h-1}{N_h}. \tag{6.11}$$

This relationship is possible because these numbers are present in Pascal's

$$\binom{n+2}{3} = 1, 4, 10, 20, 35, 56, 84, 120, 165, 220, \ldots$$

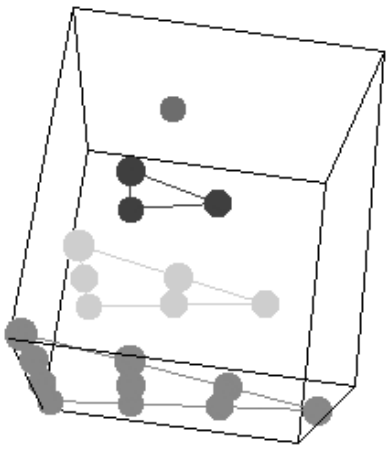


Figure 6.5: The tetrahedral numbers are the number of points required to make triangular-based pyramids (tetrahedrons) of strictly increasing size. The sequence can be calculated by adding up the preceding triangular numbers. Figure taken from http://mathworld.wolfram.com/TetrahedralNumber.html

Triangle and are therefore expressible as binomial coefficients. This process of dropping down through the sequences of simplex numbers is akin to, in the case where $N_h = 3$, the process of finding to which level of a tetrahedron our multiset belongs to (Figure 6.5) and then which row of an equilateral triangle it belongs to (Figure 6.6). The cumulative nature of simplex numbers is illustrated in Figure 6.7 which gives us another clue as to how this process works.

6.4.4 Desirable Properties of the Search Space

In this section we briefly review some desirable properties of a search space for Neural Networks and highlight how each is achieved by the search space we define in this chapter.

Ideally our search space should be free from redundancy, i.e. each point in the space should be a unique (in terms of its weights) network. For fully-connected feedforward networks this means that our search space should be order-independent at the neuron level. This is achieved by mapping each point in the space to a unique multiselection of neurons.

In traversing the search space, moving from a given network x to either consecutive network ($\langle x-1, x+1 \rangle$) should entail a difference of one neuron only. Further, the difference between the replaced neuron n and its replacement $(n+1)$

$$\binom{n+1}{2} = 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, \ldots$$

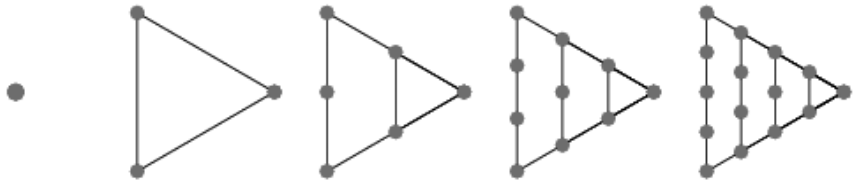


Figure 6.6: The triangular numbers are the number of points required to make equilateral triangles of increasing size. Figure taken from http://mathworld.wolfram.com/TriangularNumber.html

$$\begin{aligned}
1, 4, 10, 20, 35, 56, ... &= \sum_{x=0}^{\infty} \binom{x+3-1}{3} \textit{tetrahedral numbers} \\
1, 3, 6, 10, 15, 21, ... &= \sum_{x=0}^{\infty} \binom{x+2-1}{2} \textit{triangular numbers} \\
1, 2, 3, 4, 5, 6, ... &= \sum_{x=0}^{\infty} \binom{x+1-1}{1}
\end{aligned}$$

Figure 6.7: Figure showing the cumulative nature of simplex numbers. The value at any point is the sum of the values in the dimension (sequence) below.

Problem	Max. Fitness	Average Fitness	Std. Dev.
Pima	77.3%	50.0%	13.1%
Pima (multiset space)	77.4%	49.0%	13.0%
Iris	94.2%	33.3%	17.1%
Iris (multiset space)	95.0%	29.8%	17.2%

Table 6.6: Average fitness observed in 100,000 uniform network generations for the redundant vs. multiset space for two classification problems.

should be one weight step only. This is achieved by the numbering scheme of both the neurons and networks, with the result that network numbers which are close are similarly close in Euclidean distance in the weight space, and so more likely to be similar in terms of functionality, when compared to the same measure in the redundant space. This is because two networks in the redundant space may be far apart in terms of Euclidean distance but in fact encode the same network (due to the permutation symmetry). This suggests that the average difference between network numbers (grouped in a population) may be a simple but useful measure of the diversity of the networks in a given population.

Finally, there should ideally be a bi-directional $1:1$ mapping between network numbers and network definitions, meaning that translating from a consecutive network number to the network itself is reversible. This allows for the definition of search operators which work at the network number level, at the neuron number level and at the network weight level. In other words both genotypic and phenotypic variational operators may be used to search their respective spaces.

How similar are the spaces?

We briefly compare here the redundant and multiset spaces by sampling uniformly to inspect their aggregate statistics. We hypothesise that the spaces should have nearly identical average fitness as they contain the same networks (phenotypes). The redundant space repeats some networks more than others (depending on how many distinct alleles/neurons they are composed of) so we would expect this to cause the spaces to not be completely identical but nearly so.

Table 6.6 lists the average fitness for the redundant and multiset spaces for two classification problems, sampled by generating 100,000 networks uniformly for each space and measuring their fitness. The differences in the averages can perhaps be attributed partly to sampling error and partly to differences in the space; a more comprehensive study would be required to determine the level of similarity more conclusively. For the purposes of this investigation it is sufficient to know that they are approximately similar in terms of the networks present, i.e. that in reducing the size of the space so dramatically we do not appear to have 'lost' any networks.

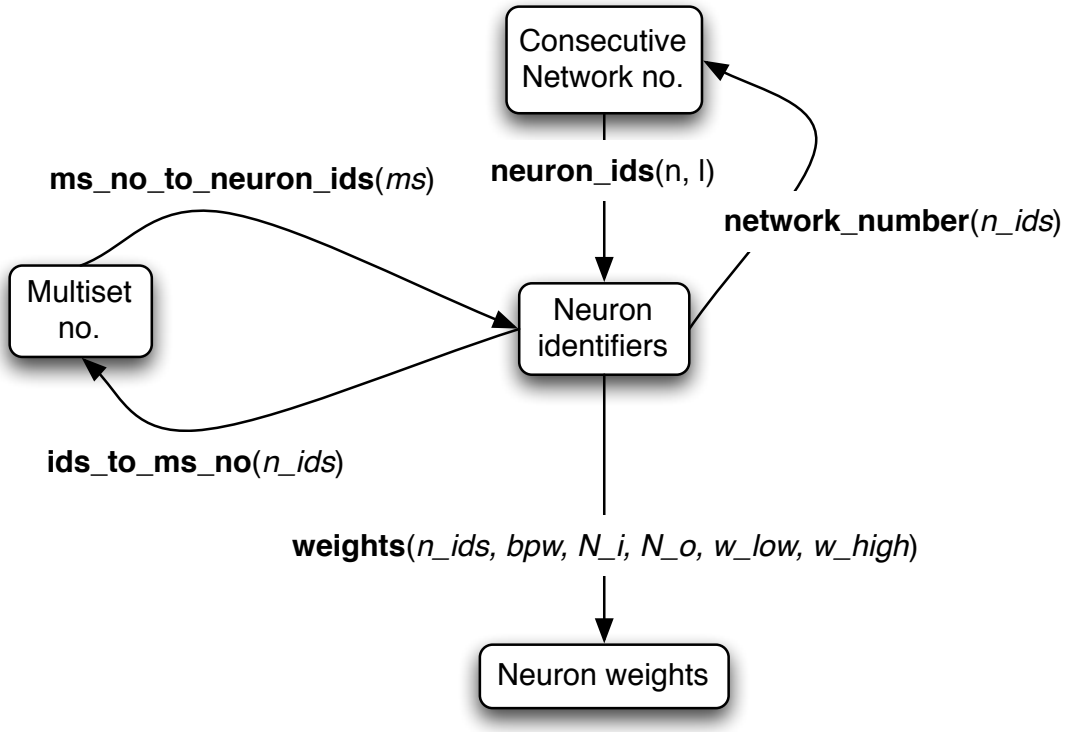


Figure 6.8: Each element of the Multiset Search framework is shown with the functions which allow traversal from one element to the other.

6.5 Search Framework

In this section we define the building blocks of the Multiset Search Framework which will form the foundation for our multiset-based search algorithms.

The Multiset Search Framework defines a search space which consists only of multisets. For example if we have n items we can form n^l strings of length l. This will include strings which are permutations of each other. The search space defined by this framework provides a mapping which allows for the consecutive numbering of the unique multisets within this space. As the mapped search space consists of consecutive numbers with no gaps, we can use any search algorithm capable of searching in a space of integers, with only minor modification. The framework defined in this section aims to give the practitioner the tools to represent and manipulate points in this search space such that a search algorithm can be applied to it. Each element of the Multiset Search Framework is shown in Figure 6.8, including the functions to use to convert elements into their alternative representations. In this section we define each of these functions and demonstrate their role in the framework.

If we have Ψ possible multisets (that is, ways to choose N_h neurons, with replacement but without respect to order), then we have a search space of networks which we number from 0 to $\Psi - 1$. We can map each of these networks to their

Algorithm 5 Calculate the position of the simplex number that is just less than n in the l^{th} dimension

```
procedure NEXT_SMALLEST(n, l)
    i ← 0
    while true do
        if n ≤ simplex_number(i + l, l + 1) then
            return ⟨i − 1, simplex_number(i − 1 + l, l + 1))
        end if
    end while
end procedure
```

Algorithm 6 Calculate the neuron identifiers for the n^{th} network with l hidden neurons.

```
procedure NEURON_IDS(n, l)
    ids ← ⟨⟩
    n′ = n + 1
    while l ≠ 0 do
        id, n′ = next_smallest(n′, l − 1)
        ids ← ids + ⟨id⟩
        l ← l − 1
    end while
    return ids
end procedure
```

respective multiset identifier (ID) which acts not only to identify the network uniquely (i.e. there is a 1 : 1 mapping between the consecutively numbered networks and the multiset identifiers) but also to *encode its neuronal composition*[11].

The multiset ID of network number n can be calculated from its neuron identifiers as:

$$\text{multiset\_id}(n, l) = \sum_{n\_id \in \text{neuron\_ids}(n,l)} (l+1)^{n\_id}.$$

This calculation is shown in full in Algorithm 7 and is dependent on Algorithm 5. We first discover the identifiers for the neurons of that particular network using Algorithm 6, and then form the multiset identifier by summing the l powers of $(l + 1)$, raised to each of the neuron identifiers (interpreted as base-10 integers).

The mapping from a multiset identifier to a vector of neuron identifiers is given in Algorithm 8. The reverse operation, to recover the neuron identifiers from a multiset identifier is given in Algorithm 9. This process demonstrates how a multiset number not only identifies but fully defines the multiset that it

[11] Due to the nature of the mapping the number itself contains the neuron selection information, making it a very compact representation for a network.

Algorithm 7 Calculate the network/multiset identifier for a particular network in the sequence of possible multisets. The value returned is the n^{th} term in the sequence for multisets containing l neurons (and so networks containing l hidden neurons).

procedure MULTISET_ID(n, l)
  $m\_id \leftarrow 0$
  $n' = n + 1$
  while $l \neq 0$ **do**
    $id, n' = \text{next\_smallest}(n', l - 1)$
    $m\_id \leftarrow m\_id + (l + 1)^{id}$
    $l \leftarrow l - 1$
  end while
  return m_id
end procedure

Algorithm 8 Calculate multiset identifier from neuron identifiers.

procedure IDS_TO_MS_NO(n_ids)
  $l \leftarrow \text{len}(n\_ids)$
  $base \leftarrow l + 1$
  return $\sum_{n \in n\_ids} base^n$
end procedure

represents.

Given our neuron identifiers we then wish to map each to its actual weights. To do this we need to specify the number of inputs and outputs of each neuron (which must be the same for each neuron), the number of bits per weight assigned (bpw) and the weight range $[lower, upper]$ used for the network weights.

This process requires us to convert bases using a particular number of digits. An algorithm for this purpose is given in Algorithm 11. We also require the function defined in Algorithm 12 to take the base-10 form of the binary weight and map it to a point within the given weight range.

Using this collection of functions we therefore have the means to translate our consecutive network numbers into multisets of neurons, and then into vectors of weights so that the actual networks can be evaluated. Then, we can reverse the mapping so that changes made to the neurons can then be mapped to appropriate network number. The process of mapping from a vector of neuron identifiers back to a network number is detailed in Algorithm 13. The strict non-increasing ordering of the neuron identifiers allows us to map from the network number to the identifiers and back again to the same network number in a deterministic manner.

Algorithm 9 Calculate neuron identifiers from multiset identifier.

```
procedure MS_NO_TO_NEURON_IDS(ms_no, l)
    base ← l + 1
    i ← 0
    while base^i <= ms_no do                    ▷ Iterate just past the id
        i ← i + i
    end while
    i ← i − 1                                   ▷ Step back one to actual id
    if i ≥ 0 then
        return ⟨i⟩ + ms_no_to_neuron_ids(ms_no − base^i, l)
    else
        return ⟨⟩
    end if
end procedure
```

Algorithm 10 Calculate network weights from neuron identifiers.

```
procedure WEIGHTS(neuron_ids, N_i, N_o, bpw, lower, upper)
    weights ← ⟨⟩
    for id ∈ neuron_ids do
        neuron ← convert_base(id, 2^bpw, N_i + N_o)
        neuron ← neuron_id_to_dec(x, lower, upper, 2^bpw) for x ∈ neuron
        weights ← weights + ⟨neuron⟩
    end for
    return weights
end procedure
```

6.5.1 Efficient 'next_smallest' Implementation

As defined before, we calculate the n^{th} l-dimensional simplex number as

$$\text{simplex\_number}(n, l) = \binom{n+l-1}{l} = \binom{n+l}{l+1} \ .$$

In the next_smallest procedure (Algorithm 5), starting from 0 each time and iterating until we find the simplex number that is just smaller than n will take a prohibitively long time for any realistic network space; this is the na ive method used in Algorithm 5. If we could estimate a start point for the iteration that is at least relatively close to the number we are looking for we can save a considerable number of iterations. In this case we can estimate the simplex number as follows[12]:

$$\text{estimate\_simplex\_position}(n, k) = \left\lfloor (n \cdot k!)^{\frac{1}{k}} - \left(\frac{k-1}{2}\right) \right\rfloor ,$$

[12]The simplex position estimate is due to Henry Bottomley, conveyed through personal communication.

Algorithm 11 Convert x to x_{base} using l digits.

procedure CONVERT_BASE(x, base, l)
  $digits = \langle\rangle$
  $p \leftarrow 0$
  while $p < l$ **do**
    $digits \leftarrow digits + \left\langle \frac{x}{base^p} \mod base \right\rangle$
  end while
  return reverse($digits$)
end procedure

Algorithm 12 Map integer $value \leq value\_max$ to point in range $[lower, upper]$.

procedure NEURON_ID_TO_DEC(value, lower, upper, value_max)
  return $lower + \left(value \cdot \frac{upper - lower}{value\_max} \right)$
end procedure

Algorithm 13 Calculate the identifier in the space of consecutive network numbers for the network defined by the given vector of neurons

procedure NETWORK_NUMBER(neuron_ids)
  $l \leftarrow$ len($neuron\_ids$) ▷ Infer dimensionality from number of neuron identifiers
  $reversed\_ids \leftarrow$ reverse($neuron\_ids$)
  $net\_number \leftarrow 0$
  $i \leftarrow 0$
  while $i < l$ **do**
    $net\_number \leftarrow net\_number$ + simplex_number($neuron\_ids[l - 1 - i], i$)
    $i \leftarrow i + 1$
  end while
  return net_number
end procedure

where n is the number we wish to find the smaller neighbour of in the simplex and $k + 1$ is simplex dimension we are searching. The more efficient implementation which makes use of this estimate is given in Algorithm 14, allowing for the efficient calculation of positions in the simplex sequences for realistic network spaces. Without this optimisation the calculation of the neuron identifiers would not be feasible.

6.6 Evolutionary Application

In this section we use the Multiset Search Framework to construct a simple Evolutionary Algorithm to search the space of unique networks.

Algorithm 14 Calculate the term in the sequence of l-dimensional simplex numbers which is just less than n. For $l = 2$ we are looking for the largest triangular number less than n. For $l = 3$ we iterate through the tetrahedral numbers, and so on.

```
procedure NEXT_SMALLEST(n, l)
    estimate = estimate_simplex_position(n, l + 1)
    sn_for_estimate = simplex_number(estimate, l)
    i ← estimate
    if sn_for_estimate = n then
        return ⟨max(0, estimate − 1), n − simplex_number(estimate − 1, l)⟩
    else if sn_for_estimate > n then          ▷ Count down from overestimate
        while true do
            if simplex_number(i, l) < n then
                return ⟨max(0, i − 1), n − simplex_number(i − 1, l)⟩
                i ← i − 1
            end if
        end while
    else                                      ▷ Count up from underestimate
        while true do
            if n ≤ simplex_number(i, l) then
                return ⟨max(0, i − 1), n − simplex_number(i − 1, l)⟩
                i ← i + 1
            end if
        end while
    end if
end procedure
```

The presented algorithm is much like any simple crossover/mutation-based Evolutionary Algorithm; the principal difference between this and other Neuroevolutionary algorithms is the genotype space and the way it is mapped onto the phenotype space of networks. We are directly searching the space of multisets in a way that is amenable to search using an Evolutionary Algorithm. An assumption here is that our mapping from consecutive integers to multisets is a useful one. For this mapping to be useful, networks 3 and 4 should be more similar to each other than to, say, networks 89 or 125.

The algorithm is outlined in Algorithm 15. Figure 6.9 demonstrates how neuron crossover works in the multiset case. We first map from the network numbers to their respective multiset identifiers, then to vectors of neuron identifiers in non-increasing order. Given these vectors we are then free to recombine them as we wish. In this case we opt for 1-point crossover though any other form of crossover can be used. The only constraint is that after crossover the produced vectors of neuron identifiers must be re-ordered such that they are again in non-increasing order. Given this need for re-ordering it may be beneficial to replace this step with

Algorithm 15 Simple Evolutionary Algorithm to directly search the space of multisets.

```
pop ← random_network_identifiers(pop_size)
while gen < generation_limit do
    Evaluate fitness of each network
    Select parents for next generation
    Map networks to neuron identifiers
    Randomly cross over neuron identifier vectors
    Randomly perturb neuron identifiers
    Map perturbed neuron identifiers back to networks
end while
```

Network 1	Network 2	
42	89	*Network numbers*
480	1980	*Convert to multiset identifiers*
[3, 3, 2, 1, 1]	[4, 3, 3, 3, 2]	*Convert to neuron identifiers*
[3, 3, 2 | 1, 1]	[4, 3, 3 | 3, 2]	*Select crossover point*
[3, 3, 2, 3, 2]	[4, 3, 3, 1, 1]	*Perform crossover*
[3, 3, 3, 2, 2]	[4, 3, 3, 1, 1]	*Re-map so as to be in non-increasing order*
720	1740	*Convert back to multiset identifiers*
51	83	*Convert back to network numbers*

Figure 6.9: An example of how the crossover process is performed for two example networks in a search space with $N_h = 5$.

a crossover operator such as that defined in [Rad93] which is designed specifically with multisets in mind. Given that we are investigating the application of existing Evolutionary Algorithms with minimal modifications to the space of multisets we do not investigate the development of such an operator at this stage.

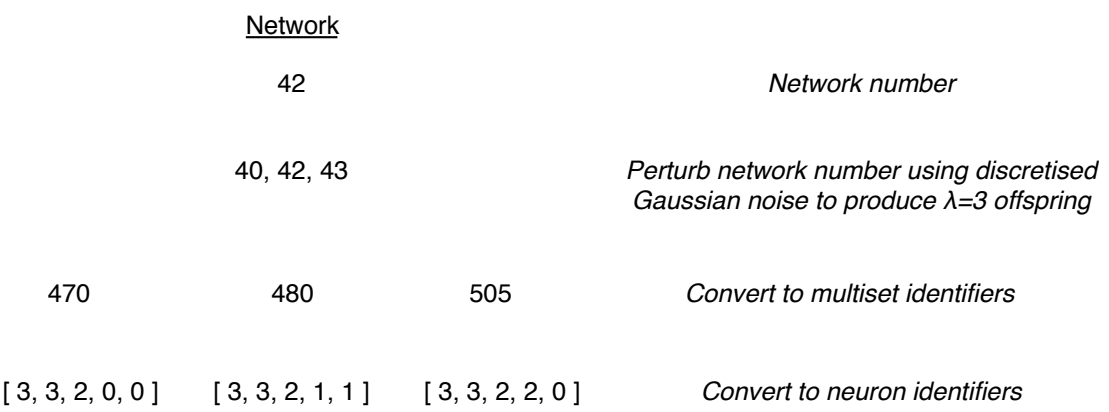


Figure 6.10: An example of how the mutation process is performed at the network number level for an example network in a search space with $N_h = 5$.

Mutation is essentially unchanged from a typical discretised mutation operator as might be used for an integer-based genotypic representation. Figure 6.10 shows the creation of offspring through applying discretised Gaussian noise to a network number. In this case there will be no need for re-ordering of the neuron identifiers as we are working strictly at the network number level of abstraction. We hypothesise it is fully possible to search the space using only mutation operators and so avoid the need for re-ordering.

It is however possible to perform mutation at the neuron level. This is illustrated in Figure 6.11. The process is similar to that of crossover in that we convert from the network number to the neuron identifiers, apply the perturbation and then re-order the neuron identifiers such that they are again in non-increasing order.

6.6.1 Multiset-based Populations

Recent work by Manso And Correia demonstrates that evaluations can be saved in Evolutionary Algorithms by considering the population as a multiset [MC09]. Not to be confused with neuron multisets, multiset-based populations are populations where individuals are stored once along with their respective count, i.e. how many times they appear in the population. If the population is to have 3 copies of an individual, then we simply record 3 as its count and use this value when calculating selection probabilities.

Given that the fitness is the same for all copies of a particular individual, it makes sense to only evaluate the fitness of one of each copy and share the fitness between the copies. The multiset representation provides a convenient way of tracking individuals and calculating selection probabilities based on their multiplicity (number of times they appear in the population) by condensing network definitions into single numeric identifiers.

An advantage of the network encoding method presented in this chapter is that

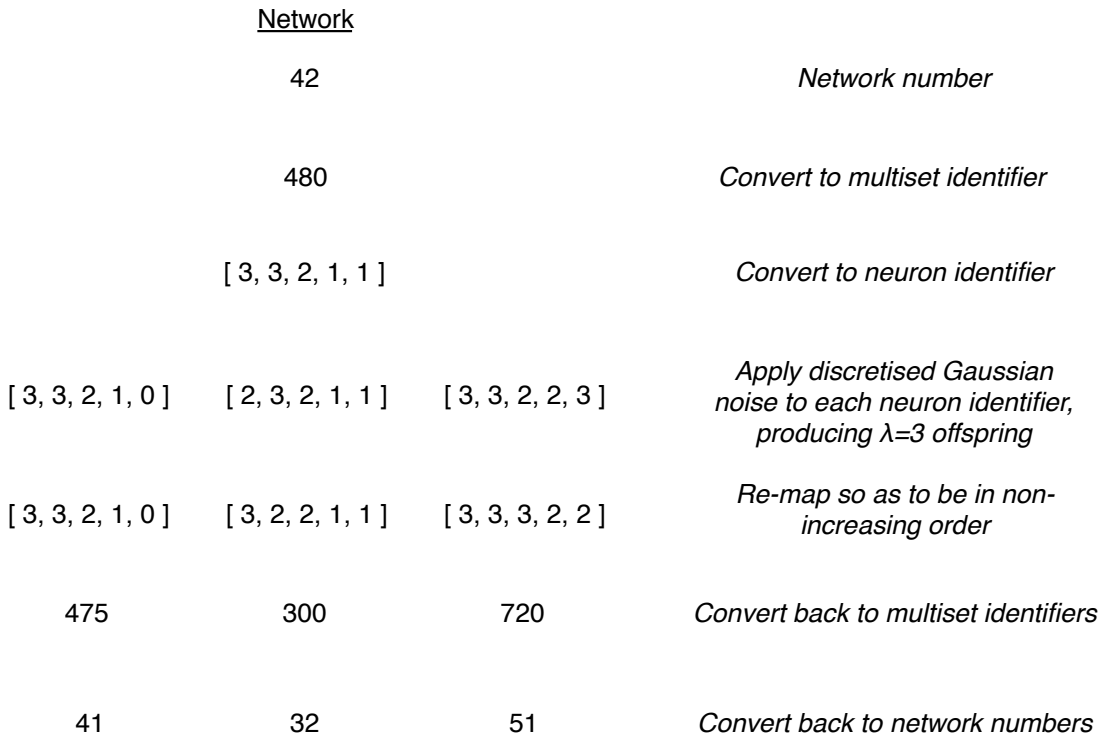


Figure 6.11: An example of how the mutation process is performed at the neuron level for an example network in a search space with $N_h = 5$.

the population is guaranteed to consist only of unique networks. As each network has a unique identifier which is not simply attached to it online, on a per-run basis as in the NEAT algorithm [Sta04] but is inherent to it, we can very easily identify copies of the same network and ensure that we calculate their fitness only once. The algorithm using this kind of population has the property that the time spent evaluating networks per generation decreases as the population converges (there are fewer unique individuals), causing the algorithm to speed up.

Without loss of generality if we set our population size to 1000 and number of generations to 100 then we should expect to perform $100,000$ evaluations over the course of a run. With a multiset-based population the number of evaluations will invariably be less than this. In the next section we investigate a strategy for using the 'left over' evaluations to broaden the search without requiring any evaluations above the number expected were we to be using a standard individual-based population.

6.6.2 Zero-cost Population Inflation

In this section we define the concepts of Zero-cost Population Inflation and Virtual Populations. The former refers to a process of inflating the population size to consider more solutions without increasing the number of evaluations required

when compared to a standard population model. A virtual population is essentially a population with the dynamics of a large population while containing relatively few individuals.

If we have multiple copies of individuals in a given population then we will require fewer evaluations to determine their fitness if we use a multiset-based population. At each generation we therefore have a percentage of evaluations that can be considered to be a 'saving' that can either be kept or 'spent' elsewhere.

We propose taking the saving of evaluations at each generation and spending it on new individuals which are introduced to the population either through a process of heavily mutating (macromutating) existing individuals or randomly generating new individuals. The aim here is to broaden the search and provide more genetic material for the existing elite members of the population to be recombined with. The end result of this is that if we set our population size to be 100 then we are stating that at each generation we will evaluate 100 *unique* individuals rather than simply 100 individuals, some of which are likely to be copies of each other.

We then have the concept of the *virtual* population which refers to a population that is larger than its number of unique individuals. Using multiset populations there is effectively no cost in increasing the count for a given network; we will still only evaluate it once. This increase should however modify its chance of selection, for example under fitness-proportional selection. This virtual population that we select individuals from will tend to be much larger than the initial population size, though this broader search will cost no more in evaluations than the equivalent algorithm using a standard individual-based population. It also means that we don't 'spend' any of the spaces on the population on simply storing copies so that fitness-proportional selection functions. We can instead use the virtual population to provide the desired selection dynamics and focus actual evaluation time on new networks.

For the empirical tests in this chapter we will use multiset populations to reduce the number of evaluations required for the initial individuals, and use the remaining evaluations to explore the validity of performing macromutations and adding new networks to the population. At this stage an implementation based on virtual populations is left for future work.

6.6.3 Empirical Results

In 15 runs of evolving multiset-based networks for the UCI Pima Diabetes classification problem the average best (training) accuracy after 250 generations of the presented Evolutionary Algorithm was 75.9% with a standard deviation of 1.14%. The best network found in all runs had accuracy 78.14%.

While these results are not particularly competitive with other results on the same problem the main use of these results is to demonstrate that the presented algorithm makes good progress on the problem while searching the space of unique multisets. The results also lend early insight into which operators may work

Operator	Average Successful Applications	Std. Dev.
Mutation	7.8%	1.3%
Macromutation	25.1%	2.8%
Crossover	13.0%	1.4%
New Network	0.5%	0.2%

Table 6.7: Percentage of applications of each operator which resulted in an improvement in fitness.

Network Number	Set Number	Representation (base-3)	Representation (base-10)
0		$\langle 0002 \rangle$	2
1	**0**	$\langle 0011 \rangle$	**4**
2		$\langle 0020 \rangle$	6
3	**1**	$\langle 0101 \rangle$	**10**
4	**2**	$\langle 0110 \rangle$	**12**
5		$\langle 0200 \rangle$	18
6	**3**	$\langle 1001 \rangle$	**28**
7	**4**	$\langle 1010 \rangle$	**30**
8	**5**	$\langle 1100 \rangle$	**36**
9		$\langle 2000 \rangle$	54

Table 6.8: The $\binom{4}{2} = 6$ sets embedded in the space of $\binom{4+2-1}{2} = 10$ multisets.

best for this new search space. Table 6.7 lists the average success rates for each operator where success is deemed to be cases where the operator has improved the fitness of at least one (in the case of crossover) of the networks it is operating on. From these results we can see that perhaps surprisingly macromutation is the most useful operator. Given the high correlation between individuals that are close in the search space and their weights (a relationship that is not necessarily the case in the redundant space), it may be that a larger magnitude of mutation is often needed to escape local minima. As such, what we term as macromutation here may in fact be mutation, meaning that a macromutation would have a larger magnitude still.

$$
\begin{aligned}
0,0,0,1,4,10,20,35,... &= \sum_{x=0}^{\infty} \binom{x}{3} \\
0,0,1,3,6,10,15,21,... &= \sum_{x=0}^{\infty} \binom{x}{2} \\
0,1,2,3,4,5,6,7,... &= \sum_{x=0}^{\infty} \binom{x}{1}
\end{aligned}
$$

Figure 6.12: Figure showing the cumulative nature of Pascal simplex numbers. The value at any point is the sum of the values in the dimension (sequence) below.

6.7 Set-based Representation

The search space can be reduced further by eliminating repetition of neurons. If we reduce the representation to selections of neurons rather than multisets, then we make a modest reduction in the search space. As the difference between the two spaces will depend largely on N_h, this saving will increase as the network size increases. This kind of representation would also allow problems where repetition is not allowed to be mapped in this way, broadening the scope of the representation to the vast majority of combinatorial optimisation problems. For larger search spaces the motivation would not be the smaller search space but rather the fact that a set-based representation would automatically satisfy the constraints of combinatorial optimisation problems where we wish to select any one object only once, i.e. operators would not be able to reach infeasible points.

A set-based representation can in fact be constructed using the algorithmic machinery presented in this chapter. To see the strategy behind this approach we examine Table 6.8 which shows the network space used to exemplify the multiset-based representation (Table 6.3). We have numbered and highlighted in bold the $\binom{4}{2} = 6$ sets embedded in the space of $\binom{4+2-1}{2} = 10$ multisets. As such we can see that the sequence of combinadic numbers that describes our sets can be found by enumerating the sequence for multisets and skipping any where the base-3 representation is not composed entirely of zeroes and ones.

Thus for the case where $N_h = 2$ we would be interested in mapping the sequence A038464[13], "Sums of 2 distinct powers of 3." to consecutive network identifiers. Put another way, we are looking for members of the same sequence as with multiset search but we are skipping those which are not made exclusively of distinct identifiers (the powers used to produce the multiset number).

The same cumulative relationship between dimensions of the sequences of what we call Pascal simplex numbers is illustrated in Figure 6.12. The method for finding the neuron identifiers of a given set is identical to that presented for

[13]http://www.research.att.com/ njas/sequences/A038464

the multiset case, except now we are using combinadic numbers composed of strictly distinct powers. We also use slightly different but related sequences for the identifier calculation and require a new approximation for the position of the next smallest member of these sequences.

6.8 Chapter Summary

In this chapter we have presented a framework for building Evolutionary Algorithms based on a permutation-free representation. This approach differs from previous work by re-mapping the representation itself rather than simply removing redundancy algorithmically at each generation. We began with the motivation for this work, which is the massive redundancy introduced by the Permutation Problem symmetry. We then surveyed previous approaches at removing the redundancy in the literature and contrasted the previous methods to our method.

The relationship between a typical genotypic representation often chosen for Neuroevolutionary algorithms and the space of unique networks (or, multisets) was presented, and a proposal for a mapping presented. A computationally-inefficient but intuitive method for translating between the smaller space of unique networks and the 'full' space was then presented. This method was replaced with a more efficient method that allows the representation to be used in practical settings.

The relationship between multisets (unique multiselections of neurons in this case), combinadic numbers and triangular/tetrahedral numbers was then investigated; a concept which forms the basis for the representation.

A search framework based on this representation was then presented, providing the building blocks necessary to build search algorithms on top of the new non-redundant representation. This framework was then used to build a simple Evolutionary Algorithm which we applied to a difficult classification problem, providing preliminary results.

Finally we introduced the concepts of multiset and virtual populations, and detailed how the presented representation can be reduced to the space of sets, broadening its applicability to the majority of combinatorial optimisation problems. The work of this chapter is however preliminary in nature; future work will be to investigate the efficacy of the approach more fully.

Chapter 7

Conclusions and Further Work

The aim of this thesis has been to characterise the severity of the Permutation Problem when evolving Neural Networks. In attempting to determine this we have encountered two possible interpretations of the problem. For the exact-weight permutation case we have shown that the probability of this kind of permutation appearing for realistic network search spaces is very low. The empirical results then suggest that given few or no permutations in the initial population, an Evolutionary Algorithm does not readily produce new permutations. In the case of permutations due to role equivalence we have estimated how the probability of the Permutation Problem will increase as the redundancy of the search space increases. This increase has been hypothesised to only be significant at very high levels of redundancy, though this issue requires further investigation.

As the Permutation Problem is closely linked with the efficacy of crossover operators we investigated whether crossover can be a useful operator in Neuroevolutionary search. A key difference between this and past work is that we check for permutations explicitly as well as measuring the effect of crossover. In this way we avoid explaining the performance of crossover in terms of permutations as we have started by demonstrating that no permutations are being recombined. As our results show that crossover has been effective in this particular experimental setup we recommend further work into determining *when* and *why* crossover is able to contribute to the evolutionary process. It should be noted that the conclusions regarding crossover presented here are based on a particular experimental setup and so are not general conclusions regarding the efficacy of crossover in Neuroevolutionary search. For this to be determined the experiments should be repeated with alternative crossover operators and a wider range of test problems.

Finally we asked how what we have learned about the Permutation Problem can be used to improve the search process. We exploit the structure and symmetry in the search space introduced by the Permutation Problem to remove the redundancy such that we search only the space of unique multiselections of neurons. This search space is then reduced further to only those networks with distinct neurons. Each of these spaces is typically a tiny fraction of the full redundant space and so could lead to more efficient evolutionary search algorithms for combinatorial

optimisation problems such as neural network construction (neuron selection). We presented preliminary empirical results using a simple Evolutionary Algorithm. As this algorithm was largely unmodified we expect that future algorithms which are designed specifically for this space may offer better performance.

7.1 What have we learned...

7.1.1 About the Permutation Problem?

Thesis Question 1: Is the Permutation Problem Serious in Practise? In Chapter 3 we began by identifying the multiple levels at which we can interpret the Permutation Problem. In discussing the severity of the problem it is essential that we are clear on the definition in use as what is true for one interpretation may not be true for others.

We have focussed primarily on exact-weight permutations of neurons. In Chapter 4 we presented the exact probability for drawing permutations in an initial population. Using a method based on partitions we were able to calculate this for realistic network spaces, thus demonstrating the extremely low probability of exact-weight permutations. The same method was then used to estimate the change in probability as the level of redundancy increased, resulting in the conclusion that the level of redundancy would need to be very high in order to increase the probability significantly.

Thesis Question 2: Is our understanding of how recombination works in the context of Neural Networks sufficient? In Chapter 5 we conducted experiments which aimed to provide empirical data on the Permutation Problem for recombination-based Evolutionary Algorithms. We also investigated the efficacy of crossover from the perspective that its contribution or lack thereof must be explained by something other than exact-weight permutations as we found that such permutations were not being recombined.

We began with an analysis based on Price's Equation which showed that crossover did indeed appear to be contributing significantly to the evolutionary process. Then in Section 5.3 we attempted to determine whether the contribution of crossover can be explained from the perspective of it being a macromutational operator rather than a recombination operator. Our conclusion here was that crossover may be a macromutation of lesser magnitude than checked for in this investigation. Further investigation is required if we wish to determine with greater certainty when and why crossover can be a useful operator when evolving Neural Networks. At this stage we can see that both at the neuron and network level crossover appears to contribute to the search process. As such a recommendation of this work would be that it may be worthwhile to investigate the recombination of Neural Networks in more depth.

7.1.2 About Neuroevolutionary Search?

Thesis Question 3: Can our knowledge regarding the Permutation Problem be used to improve Neuroevolutionary search? In Chapter 6 we presented our finding that it is possible to build a permutation-free search space using a mapping based on combinadic numbers. In its presented form the representation is amenable to search using existing algorithms in the literature; future work would be to construct an algorithm specifically designed for searching the space of unique sets or multisets. An efficient implementation was then presented, making the framework viable in practise. The initial results show performance that is similar to that of the empirical investigations of Chapter 5. This is perhaps unsurprising given that the representations contain the same networks and the algorithms are very similar. In later work we would hope to exploit the removal of the redundancy further by developing search operators which are tailored to this space. We have demonstrated however that it is possible to reduce massively redundant spaces to their unique points in a computationally efficient manner.

7.2 Future Work

The conclusions of this thesis regarding the Permutation Problem and the effective optimisation of Artificial Neural Networks have been investigated for fixed-architecture single-layer feed-forward networks. It is expected that the theory presented will apply equally well to multi-layered networks, though it will need to be extended in order to precisely determine how multiple layers will affect the probability of the problem occurring.

Additionally, an investigation into the *role redundancy* of typical neuron spaces may shed more light on how to make Neuroevolutionary search more effective. Particularly, if many possible neurons in the search space perform essentially the same function (given the chosen fitness function) then the likelihood of encountering same-role permutations increases. While the Multiset Search Framework provides a means to search the space of unique networks, if the neurons themselves are not functionally unique then some choices of neurons will result in effectively the same network. We consider work towards reducing the neuron search space to those of significantly unique functionality a worthy research goal for future work.

7.2.1 Extensions to the Multiset Search Framework

Gray Coding

The representation scheme for neurons is designed such that moving from a neuron to either of its neighbouring neurons will result in moving one weight step away from the original. With a binary representation we have the problem of 'Hamming Cliffs' where a one-bit change can result in a large jump in the weight space.

The encoding for the neurons could be converted to Gray Coding to reduce the distance from one neuron to another to 'smooth' out the neuron search landscape.

Bias Weights

The current representation does not explicitly support bias weights but as they can be considered as extra input weights it is possible to add them in as such without modifying the representation. The presented algorithm does not provide this however. The negative effects of not having a bias can be mitigated to a certain extent by normalising the inputs such that most pass through the origin; this is the approach we have taken in this work. A future version of the algorithm will also optimise bias weights for each hidden neuron.

Set-based Representation

Future work is to implement the set-based representation to facilitate application to problems where repetition of elements is undesirable, for example in ensemble construction or in the majority of combinatorial optimisation problems. The only practical consideration in implementing this kind of representation is the derivation of a new approximation to use in speeding up the discovery of item (e.g. neuron) identifiers. A slight modification to the simplex number position approximation[1] that may prove to be a good starting point is

$$\text{estimate\_pascal\_simplex\_position}(n,k) = \left\lfloor (n \cdot k!)^{\frac{1}{k}} + \left(\frac{k-1}{2}\right) \right\rfloor . \qquad (7.1)$$

Ensemble Construction

In the current model, individuals in the population are single numbers which identify a network (each individual is therefore a single network). If we extend the population genotype to be composed of several network numbers rather than one, then each individual effectively encodes a particular network ensemble.

We now have combinatorial optimisation at two levels: on one level we are looking for the best combination of neurons to build networks with and on the other we are looking for the best combination of networks for our ensemble. Related to the future work on employing set rather than multiset-based representations, we believe this could be a fruitful avenue for research in the field of ensemble learning. The new search space would be that of unique selections of distinct classifiers.

As the search space now contains no permutations, distant networks (in terms of their network) now always differ considerably in terms of their weights. An interesting question is whether this now provides a very simple estimator of ensemble (or population) diversity, by looking at the difference between the network numbers. If this is the case then this could lead to better population

[1] Due to Henry Bottomley, conveyed in informal communication.

initialisation routines and a simple similarity metric that might be useful when choosing parents for recombination.

References

[Alt95] L Altenberg. The schema theorem and price's theorem. *Foundations of Genetic Algorithms*, 3:23–49, 1995.

[ASP94] P Angeline, G Saunders, and J Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks. Vol 5, no. 1*, pages 54–65, 1994.

[ATL07] A Agapitos, J Togelius, and S Lucas. Evolving controllers for simulated car racing using object oriented genetic programming. *In Proceedings of the 2007 Genetic and Evolutionary Computation Conference (GECCO)*, pages 1543–1550, 2007.

[BM98] C Blake and C Merz. Uci repository of machine learning databases. *ci.nii.ac.jp*, 1998.

[BMS90] R Belew, J McInerney, and N Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning. *Tech. rept. CSE TR90-174 UCSD*, 1990.

[BPJ04] JK Bassett, MA Potter, and K De Jong. Looking under the EA hood with price's equation. *In Proceedings of the 2004 Genetic and Evolutionary Computation Conference (GECCO)*, pages 914–922, 2004.

[BPJ05] JK Bassett, MA Potter, and K De Jong. Applying price's equation to survival selection. *In Proceedings of the 2005 Genetic and Evolutionary Computation Conference (GECCO)*, pages 1371–1378, 2005.

[Bra95] J Branke. Evolutionary algorithms for neural network design and training. *Proceedings of the 1st Nordic Workshop on Genetic Algorithms*, 1995.

[CF96] B Carse and TC Fogarty. Fast evolutionary learning of minimal radial basis function neural networks using a genetic algorithm. *Selected Papers from AISB Workshop on Evolutionary Computing*, pages 1–22, 1996.

[Cro08] M Crosby. Evolving a roving eye for go. *Master's Dissertation, Edinburgh University School of Informatics*, 2008.

[CSW06] W Chen, J Shih, and S Wu. Comparison of support-vector machines and back propagation neural networks in forecasting the six major asian stock markets. *International Journal of Electronic Finance*, 1(1):49–67, 2006.

[DHAI08] A Das, M Hossain, S Abdullah, and R Islam. Permutation free encoding technique for evolving neural networks. *In proceedings of the 5th International Symposium on Neural Networks (ISNN)*, 2008.

[Elm90] J Elman. Finding structure in time. *Connectionist Psychology: A Text with Readings*, 14(2):179–211, 1990.

[ET05] D Enke and S Thawornwong. The use of data mining and neural networks for forecasting stock market returns. *Expert systems with applications*, 29(4):927–940, 2005.

[FFP90] D Fogel, L Fogel, and V Porto. Evolving neural networks. *Biological Cybernetics*, 63(6):487–493, 1990.

[FG97] D Fogel and A Ghozeil. A note on representations and variation operators. *IEEE Transactions on Evolutionary Computation*, 1(2):159–161, 1997.

[Fog94] D Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1994.

[Fog06a] D Fogel. Evolutionary computation: toward a new philosophy of machine intelligence. *John Wiley and Sons*, 2006.

[Fog06b] D Fogel. Foundations of evolutionary computation. *Proceedings of the Society of Photographic Instrumentation Engineers (SPIE)*, 6228:1–13, 2006.

[FS08] T Froese and E Spier. Convergence and crossover: The permutation problem revisited. *University of Sussex Cognitive Science Research Paper CSRP 596*, 2008.

[GBM01] F Gomez, D Burger, and R Miikkulainen. A neuro-evolution method for dynamic resource allocation on a chip multiprocessor. *In Proceedings of the INNS-IEEE International Joint Conference on Neural Networks*, pages 2355–2361, 2001.

[GM99] F Gomez and R Miikkulainen. Solving non-markovian control tasks with neuroevolution. *International Joint Conference on Artificial Intelligence*, 16(2):1356–1361, 1999.

[GM03a] F Gomez and R Miikkulainen. Active guidance for a finless rocket using neuroevolution. *Lecture Notes in Computer Science*, pages 2084–2095, 2003.

[GM03b] F Gomez and R Miikkulainen. Robust non-linear control through neuroevolution. *Unpublished doctoral dissertation*, 2003.

[Gol89] D Goldberg. Genetic algorithms in search and optimization. *Addison-Wesley Professional*, 1989.

[GPOBHM05] N García-Pedrajas, D Ortiz-Boyer, and C Hervás-Martínez. An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization. *Neural Networks*, 19(4):514–528, 2005.

[GSM06] F Gomez, J Schmidhuber, and R Miikkulainen. Efficient non-linear control through neuroevolution. *Lecture Notes in Computer Science*, 4212/2006:654–662, 2006.

[GSM08] F Gomez, J Schmidhuber, and R Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *The Journal of Machine Learning Research*, 9:937–965, 2008.

[Gur97] K Gurney. An introduction to neural networks. *CRC Press*, 1997.

[GWP96] F Gruau, D Whitley, and L Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. *Proceedings of the First Annual Conference on Genetic Programming*, pages 81–89, 1996.

[Han92] P Hancock. Coding strategies for genetic algorithms and neural nets. *Unpublished doctoral dissertation*, 1992.

[Han93] P Hancock. Genetic algorithms and permutation problems: a comparison of recombination operators for neural net structure specification. *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks*, 1993.

[HN89] R Hecht-Nielsen. Theory of the backpropagation neural network. *Neural networks for perception*, 2:65–93, 1989.

[HN08] S Haflidason and R Neville. A case for crossover in neuroevolution. *In Proceedings of the Workshop and Summer School on Evolutionary Computing (WSSEC), Lecture Series by Pioneers*, 2008.

[HN09a] S Haflidason and R Neville. On the significance of the permutation problem in neuroevolution. *In Proceedings of the 2009 Genetic and Evolutionary Computation Conference (GECCO)*, pages 787–794, 2009.

[HN09b] S Haflidason and R Neville. Quantifying the severity of the permutation problem in neuroevolution. *In Proceedings of the 4th International Workshop on Natural Computing (IWNC)*, pages 149–156, 2009.

[HNI04] MF Habib, SMS Nejhum, and MM Islam. Partial node crossover (pnx)–a permutation problem free crossover operator to evolve artificial neural networks. *In Proceedings of the 7th International Conference on Computer and Information Technology*, 2004.

[Hoc98] S Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*, 6(2):107–116, 1998.

[JF00] A Jain and D Fogel. Case studies in applying fitness distributions in evolutionary algorithms: I. simple neural networks and gaussion mutation. *Proceedings of The International Society for Optical Engineering (SPIE)*, pages 168–175, 2000.

[Jon95] T Jones. Crossover, macromutation, and population-based search. *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 73–80, 1995.

[Jon06] K De Jong. Evolutionary computation: A unified approach. *MIT Press*, 2006.

[KAYT90] T Kimoto, K Asakawa, M Yoda, and M Takeoka. Stock market prediction system with modular neural networks. *Neural Networks*, 2:11–16, 1990.

[Kir84] S Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5-6):975–986, 1984.

[LLPS93] M Leshno, V Lin, A Pinkus, and S Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[Mat07] B Mathisen. Evolving a roving-eye for go revisited. *Master's Dissertation, Norwegian University of Science and Technology, Department of Computer and Information Science*, 2007.

[MC09] A Manso and L Correia. Genetic algorithms using populations based on multisets. *The Fourteenth Portuguese Conference on Artificial Intelligence*, 2009.

[MD89] D Montana and L Davis. Training feedforward neural networks using genetic algorithms. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767, 1989.

[MF04] Z Michalewicz and D B Fogel. How to solve it: modern heuristics. *Springer*, 2004.

[MGW+06] H Mayer, F Gomez, D Wierstra, I Nagy, and A Knoll. A system for robotic heart surgery that learns to tie knots using recurrent neural networks. *Proceedings of the International Conference on Intelligent Robotics and Systems (IROS)*, 22(13-14):1521–1537, 2006.

[MM96] D Moriarty and R Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1-3):11–32, 1996.

[MM97] D Moriarty and R Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5(4):373–399, 1997.

[NE98] M Newman and R Engelhardt. Effects of neutral selection on the evolution of molecular species. *Proceedings of the Royal Society of London*, 1998.

[OYKM07] K Ohkura, T Yasuda, Y Kawamatsu, and Y Matsumura. Mbeann: Mutation-based evolving artificial neural networks. *Lecture Notes in Computer Science*, 2007.

[PBJ03] M Potter, J Bassett, and K De Jong. Visualizing evolvability with price's equation. *In Proceedings of the 2003 Congress on Evolutionary Computation*, 3102/2004:914–922, 2003.

[Pri70] G Price. Selection and covariance. *Nature*, 227(5257):520–521, 1970.

[Rad90] NJ Radcliffe. Genetic neural networks on mimd computers. *Unpublished doctoral dissertation*, 1990.

[Rad93] NJ Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, 1(1):67–90, 1993.

[Rud97] G Rudolph. Reflections on bandit problems and selection methods in uncertain environments. *In Proceedings of the 7th International Conference on Genetic Algorithms*, pages 166–173, 1997.

[SBM05] K Stanley, B Bryant, and R Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668, 2005.

[SI00] P Stagge and C Igel. Neural network structures and isomorphisms: Random walk characteristics of the search space. *IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks (ECNN)*, pages 82–90, 2000.

[SM02] K Stanley and R Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[SM04] K Stanley and R Miikkulainen. Evolving a roving eye for go. *Lecture Notes in Computer Science*, 3103/2004:1226–1238, 2004.

[Sta04] K Stanley. Efficient evolution of neural networks through complexification. *Unpublished doctoral dissertation*, 2004.

[Sta06] K Stanley. Exploiting regularity without development. *In Proceedings of the AAAI Fall Symposium on Developmental Systems*, 2006.

[SW98] E Saad and D Wunsch. Comparative study of stock trend prediction using time delay, recurrent and probabilistic neural networks. *IEEE Transactions on Neural Networks*, 9(6):1456–1470, 1998.

[SWE92] J Schaffer, D Whitley, and L Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. *Combinations of Genetic Algorithms and Neural Networks*, 1992.

[Thi96] D Thierens. Non-redundant genetic coding of neural networks. *In Proceedings of IEEE International Conference on Evolutionary Computation*, pages 571–575, 1996.

[TSVM93] D Thierens, J Suykens, J Vandewalle, and B De Moor. Genetic weight optimization of a feedforward neural network controller. *In Proceedings of the Conference on Artificial Neural Nets and Genetic Algorithms*, pages 658–663, 1993.

[Whi89] D Whitley. Applying genetic algorithms to neural network learning. *Proceedings of the Seventh Conference on Artificial Intelligence and Simulation of Behaviour (AISB)*, pages 137–144, 1989.

[Whi95] D Whitley. Genetic algorithms and neural networks. *Genetic Algorithms in Engineering and Computer Science*, 1995.

[WM97] D Wolpert and W Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[WSB90] D Whitley, T Starkweather, and C Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14(3):347–361, 1990.

[WYX04] Z Wang, X Yao, and Y Xu. An improved constructive neural network ensemble approach to medical diagnoses. *Lecture Notes in Computer Science*, pages 572–577, 2004.

[Yao99] X Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 8(3):694–713, 1999.

[YI08] X Yao and M Islam. Evolving artificial neural network ensembles. *Studies in Computational Intelligence*, 115/2008:851–880, 2008.

[YL96] X Yao and Y Liu. Evolving artificial neural networks through evolutionary programming. *Proceedings of the 5th Annual Conference on Evolutionary Programming*, 1996.

[YL97] X Yao and Y Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, 8(3):694–713, 1997.

[Zha00] G Zhang. Neural networks for classification: a survey. *IEEE Transactions on Systems*, 30(4):451–462, 2000.

[Zha01] G Zhang. An investigation of neural networks for linear time-series forecasting. *Computers and Operations Research*, 28(12):1183–1202, 2001.

[ZPH98] G Zhang, BE Patuwo, and MY Hu. Forecasting with artificial neural networks: The state of the art. *International journal of forecasting*, 14(1):35–62, 1998.

Appendix A

GA-ML Digest Archive Dec. 1988

```
Genetic Algorithms Digest
Friday, 9 December 1988  Volume 2 : Issue 24

 - Send submissions to GA-List@AIC.NRL.NAVY.MIL
 - Send administrative requests to
    GA-List-Request@AIC.NRL.NAVY.MIL

Today's Topics:

- Administrivia & Reminder about Conference Deadline
- Re: GAs for control systems
- Alternative knowledge representations for GA learning (2)
- GAs and neural nets (2)

Date: Wed, 7 Dec 88 11:41:49 GMT
From: Nick Radcliffe
       <njr\%itspna.edinburgh.ac.uk@NSS.Cs.Ucl.AC.UK>
Subject: GAs and neural nets

I am about to begin an investigation of Genetic Learning
Algorithms for layered, feed-forward Neural Networks and would
appreciate any information/comments/references anyone has
about similar work. Specifically, the process of training a
Neural Network amounts to the selection of an optimal set
of weights (or connection strengths) between the neurons it
comprises.

A major subtlety that I see in this problem is that in fully
connected nets (and to a lesser degree in partially connected
nets) with n hidden nodes there are n! equivalent optimal
solutions which may be generated by permuting the labels
```

of the hidden units. It seems to me very likely that if no action is taken to try and overcome this the crossover operation will be very destructive unless cleverly implemented.

This is because even crossing over two equivalent networks which use different labellings will not, in general, generate another equivalent network. I have various ideas about how to overcome this, but it would clearly be silly to reinvent the wheel.

The only papers I am aware of in this area are one by Darrell Whitley (Applying Genetic Algorithms to Neural Network Problems: A Preliminary Report), and one that takes a hierarchical approach by Eric Mjolsness, David Sharp and Bradley Alpert (Scaling, Machine Learning and Genetic Neural Nets).
Neither of these addresses the permutation problem directly.

All help will be appreciated.

Nick

Nick Radcliffe
Theoretical Physics
The King's Buildings
Edinburgh University
Edinburgh
Scotland
(031) 667 1081 x 2850
JANET: NickRadcliffe@uk.ac.ed