Evolving Artificial Neural Networks using Cartesian Genetic Programming

Andrew James Turner

PhD

University of York

Electronics

September 2015

Abstract

NeuroEvolution is the application of Evolutionary Algorithms to the training of Artificial Neural Networks. NeuroEvolution is thought to possess many benefits over traditional training methods including: the ability to train recurrent network structures, the capability to adapt network topology, being able to create heterogeneous networks of arbitrary transfer functions, and allowing application to reinforcement as well as supervised learning tasks.

This thesis presents a series of rigorous empirical investigations into many of these perceived advantages of NeuroEvolution. In this work it is demonstrated that the ability to simultaneously adapt network topology along with connection weights represents a significant advantage of many NeuroEvolutionary methods. It is also demonstrated that the ability to create heterogeneous networks comprising a range of transfer functions represents a further significant advantage.

This thesis also investigates many potential benefits and drawbacks of NeuroEvolution which have been largely overlooked in the literature. This includes the presence and role of genetic redundancy in NeuroEvolution's search and whether program bloat is a limitation.

The investigations presented focus on the use of a recently developed NeuroEvolution method based on Cartesian Genetic Programming. This thesis extends Cartesian Genetic Programming such that it can represent recurrent program structures allowing for the creation of recurrent Artificial Neural Networks. Using this newly developed extension, Recurrent Cartesian Genetic Programming, and its application to Artificial Neural Networks, are demonstrated to be extremely competitive in the domain of series forecasting.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to acknowledge the funding provided by the EPSRC, without which this thesis would not have been possible. I am pleased to live in a country which both values and supports academic research.

The experience of undertaking a PhD is strongly determined by the relationship between the student and their supervisor. In this regard I consider myself highly fortunate. Julian has not only been an exceptional supervisor, but also a mentor. He has given academic, and life advice. I am wholly grateful for his time and company, both in a supervisory setting, and in the early hours at conference hotel bars.

There are two particular friends which have been quintessential to my PhD experience. Nils Morozs and Stuart Lacy, we will graduate from York not once, but twice together. I wish you both great success.

I will be forever grateful to my Mum, Dad and Brother, my family. I don't know their initial feelings when I turned down a "*proper job*" to perpetuate my life as a student, but they have shown nothing but loving support. I have the utmost pride in my family, and feel fulfilled in the pride they have for me.

Katherine, we share a life here in York which I hope will never end. We love, support and amuse each other as we ramble through life day to day. I can't wait to see where our ramblings take us next together.

Declaration

I declare that the research described in this thesis is original work, which I undertook at the University of York during 2012 - 2015. This work has not previously been presented for an award at this, or any other, University.

Some parts of this thesis have been published in journals and conference proceedings; where items were published jointly with collaborators, the author of this thesis is responsible for the material presented here. For each published item the primary author is the first listed author.

Journal Articles and Letters

- Turner, A. J. & Miller, J. F. Introducing A Cross Platform Open Source Cartesian Genetic Programming Library. In *Genetic Programming and Evolvable Machines*, vol 16, pages 83-91, 2014.
- Turner, A. J. & Miller, J. F. NeuroEvolution: Evolving Heterogeneous Artificial Neural Networks. In *Evolutionary Intelligence*, vol 7, pages 135-154, 2014.
- Turner, A. J. & Miller, J. F. Neutral Genetic Drift: an Investigation using Cartesian Genetic Programming. In *Genetic Programming and Evolvable Machines*, 2015.

Conference Proceedings

- Turner, A. J. & Miller, J. F. Cartesian Genetic Programming encoded Artificial Neural Networks: A Comparison using Three Benchmarks. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO-13)*, pages 1005-1012, 2013.
- Turner, A. J. & Miller, J. F. The Importance of Topology Evolution in NeuroEvolution: A Case Study Using Cartesian Genetic Programming of Artificial Neural Networks. In *Research and Development in Intelligent Systems XXX*, pages 213-226, 2013.
- Turner, A. J. & Miller, J. F. Cartesian Genetic Programming: Why No Bloat?. In *Genetic Programming: 17th European Conference (EuroGP-2014)*, pages 193-204, 2014.

- Turner, A. J. & Miller, J. F. NeuroEvolution: The Importance of Transfer Function Evolution and Heterogeneous Networks. In *Proceedings of the 50th Anniversary Convention of the AISB*, pages 158-165, 2014.

- Turner, A. J. & Miller, J. F. Recurrent Cartesian Genetic Programming. In *13th International Conference on Parallel Problem Solving from Nature (PPSN 2014)*, pages 476-486, 2014.

- Turner, A. J. & Miller, J. F. Recurrent Cartesian Genetic Programming Applied to Famous Mathematical Sequences. In *Proceedings of the Seventh York Doctoral Symposium on Computer Science & Electronics* (YDS 2014), pages 37-46, 2014.

- Turner, A. J. & Miller, J. F. Recurrent Cartesian Genetic Programming Applied to Series Forecasting. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-15)*, pages 1499-1500, 2015.

Chapter 1

Introduction

This chapter provides an overview of the thesis, presenting the motivation and aims of the work, the main contributions made and a summary of the chapters presented.

1.1 Structure of this Chapter

Section 1.2 provides high level motivation for the work presented throughout this thesis. Section 1.3 gives the high level aims of the thesis. Section 1.4 describes the significant contributions which have been made during the thesis. Finally, Section 1.5 gives an outline of the remaining chapters and appendixes.

1.2 Motivation

Artificial Neural Network (ANN)s and Evolutionary Algorithm (EA)s represent two powerful, widely adopted, machine learning methodologies strongly inspired by biological systems. However, there is one important distinction between ANNs and EAs. ANNs represent an abstraction of biological brains which must be trained (optimised) in order to solve a given task. Whereas EAs are an optimisation method based on Darwinian Evolution. Interestingly, this means that these two, independent, machine learning methods can be combined by using EAs to optimise the configuration of ANNs. This union of ANNs and EAs is termed NeuroEvolution (NE).

The combination of EAs and ANNs is not just of academic interest but is thought to provide a number of significant advantages over many other ANN training methods. For instance, NE can easily be applied to the creation of feed-forward and recurrent networks, it can be used to apply ANNs to supervision and reinforcement learning tasks, the depth of the network has no influence on the learning algorithm, it can be used to evolve networks of heterogeneous neuron transfer functions, and finally, it can be used to adapt both connection weights as well as the network topology.

Despite the increasing popularity of NE methods, there are a number of significant gaps in the literature; notably those concerning the perceived benefits of NE. For example, it is often stated that one of the major advantages of NE is its ability to determine network topology. However, there is surprisingly little empirical research which assess whether, and to what extent, evolving network topology actually provides an advantage. Additionally, it is often stated that the ability for NE to create heterogeneous ANNs represents an advantageous characteristic. However, as with topology configuration, there is currently very little empirical evidence to support this.

There are also highly significant topics within the field of EAs that are mostly absent in the field of NE. For instance, issues concerning program bloat and the effect of genetic redundancy are almost never considered.

Therefore, if the field of NE is to progress, these gaps in the literature must be addressed. This represents one of the key motivations of this work.

The second motivation is concerned with developing further the technique of Cartesian Genetic Programming of Artificial Neural Networks (CGPANN), a recently developed NE method created by Maryam Mahsal Khan et al. in 2010. CGPANN is a NE method based on Cartesian Genetic Programming (CGP). CGP is a graph based form of Genetic Programming (GP) which is uniquely suited to describing ANNs; due to it encoding general directed feed-forward Multiple-Input Multiple-Output (MIMO) graphs of computational elements. The only alterations required to apply CGP to the evolution of ANNs is the addition of connection weights and the use of transfer functions typically associated with ANNs. Additionally, CGP is an established GP method which has undergone much development and theoretical study. A large proportion of this previous work applies directly to CGPANN and therefore provides many possibilities for future ANN developments.

1.3 Thesis Aims

Based on the described motivations, and material presented in the literature review, a number of thesis aims are proposed.

1. To investigate whether the ability to adapt network topology represents an advantageous property of NeuroEvolution.

2. To investigate whether the ability to create heterogeneous Artificial Neural Networks represents an advantageous property of NeuroEvolution.

3. To extend the Cartesian Genetic Programming algorithm to be capable of creating recurrent program structures.

4. To apply the developed recurrent Cartesian Genetic Programming extension to the evolution of recurrent Artificial Neural Networks.

5. To investigate the role of genetic redundancy on Cartesian Genetic Programming's evolutionary search; with a focus on its application to training Artificial Neural Networks.

6. To investigate the suitability of applying Cartesian Genetic Programming as a training method for Artificial Neural Networks in the domains of classification and series forecasting.

1.4 Thesis Contributions

Throughout this thesis a number of substantial contributions are made to CGP, CGPANN and the wider field of NE. The most significant contributions are now summarised.

1. This thesis presents Recurrent Cartesian Genetic Programming (RCGP), a significant CGP extension which enables the creation of both acyclic and *cyclic* program structures. In this thesis RCGP is shown to be capable of solving tasks intractable to standard CGP. Additionally, when applied to series forecasting, RCGP is demonstrated to not only outperform standard CGP, but also a range of popular standard forecasting methods.

2. This thesis presents the application of the RCGP extension to CGPANN in order to facilitate the evolution of Recurrent Artificial Neural Network (RANN)s. This recurrent CGPANN extension is termed Recurrent Cartesian Genetic Programming of Artificial Neural Networks (RCGPANN). As with RCGP, RCGPANN is demonstrated to be a highly effective forecasting method outperforming all standard forecasting methods used for comparison.

3. A significant proportion of this thesis is dedicated to providing rigorous empirical evidence of many of the perceived advantages of NE. This has been achieved for two domains. Firstly, it has been show that the ability for many NE methods to evolve the network topology, as well as connection weights, of ANNs represents a significant advantage over training methods which only optimise connections weights. Results presented indicate that the importance of topology optimisation may even be more significant to training than connection weight optimisation. Secondly, it has been shown that the ability of nearly all NE methods to create heterogeneous ANNs represents a significant, and widely overlooked, advantage over training methods which solely train homogeneous ANNs.

4. This thesis rigorously, and extensively, evaluates the role and benefit of genetic redundancy in both CGP and CGPANN. It is shown that the presence of genetic redundancy in CGP provides a substantial advantage to the evolutionary search; primarily through its ability to aid the escape of local optima in the search space. It is also shown that the benefit of genetic redundancy is substantially lower when evolving ANNs. Theoretical explanations of this interesting discrepancy between CGP and CGPANN are provided based on the influence of connection weights on the search.

5. This thesis rigorously assesses CGPANN in the domains of classification and series forecasting. In both cases the studies go beyond previous applications of CGPANN in terms of the methodology, the range of benchmark problems employed and the number of methods used for comparison. From these applications it is demonstrated that CGPANN performs poorly in the domain of classification and exceptionally well in the domain of series forecasting.

6. Finally, during this thesis an open source cross platform CGP library was developed. This library provides a stable, tested, implementation of CGP along with all of

the extensions and developments made during this thesis: RCGP, CGPANN and RCGPANN. This contribution is significant for two main reasons. Firstly, it allows others to learn, use, research and apply all of the developments made in this thesis. Secondly, it allows the presented experiments to be able to be replicated by others.

1.5 Thesis Outline

This section outlines the remaining chapters of this thesis.

Chapter 2 provides an introduction to NE in general, including descriptions of the more popular NE methods. A substantial review of the NE literature is also undertaken with the insights gained used to guide the research presented.

Chapter 3 provides a detailed description of CGP, the underlying algorithm of the NE method of interest in this thesis; CGPANN. This description includes the basic algorithm, distinguishing features, a summary of common applications and a discussion of the extensions and theoretical work which have been developed.

Chapter 4 describes CGPANN, the NE method of interest in this thesis. This chapter describes the CGPANN implementation, possible advantages over other NE methods and previous applications. Additionally, a number of experiments are presented applying CGPANN to standard benchmark problems, assessing previous design decisions made for CGPANN and demonstrating that CGPANN, like CGP, is immune to program bloat.

Chapter 5 presents a new extension made to the CGP algorithm in order to enable the encoding of cyclic program structures. This new method is termed *Recurrent Cartesian Genetic Programming* and represents a superset of CGP; capable of evolving both recurrent and feed-forward program structures. The chapter also presents a number of initial experiments demonstrating that RCGP is indeed capable of evolving cyclic programs.

Chapter 6 presents a number of experiments demonstrating the advantages of evolving ANN topology. This work fills a current gap in the literature where it is often assumed that evolving ANN topology represents an advantageous property, with little empirical evidence to support the claim.

Chapter 7 presents a number of experiments demonstrating the advantages of using NE to evolve heterogeneous ANNs. Despite many calls in the literature for research in this area, very little work has been previously presented which assesses the benefit of evolving heterogeneous ANNs. This work addresses this gap in the literature.

Chapter 8 presents a number of experiments investigating the role of neutral genetic drift for both CGP and CGPANN. The work demonstrates that CGP greatly benefits from the increased genetic redundancy provided by its encoding. The work also identifies that CGPANN benefits significantly less from neutral genetic drift than CGP, despite being based on the same underlying algorithm.

Chapter 9 presents a rigorous evaluation of CGPANN in the domain of classification. The work demonstrates that CGPANN performs poorly compared to a range of standard classification techniques.

Chapter 10 presents a rigorous evaluation of RCGPANN in the domain of series forecasting where it is shown to outperform all standard forecasting techniques used for comparison. Additionally, CGP, RCGP and CGPANN are applied to the same tasks in order to assess which aspects of RCGPANN contribute to its strong performance; the recurrent extension or the application to ANNs.

Chapter 11 gives the final overall conclusions of the thesis along with proposed future work.

Appendix A describes, in detail, the benchmarks which are used throughout this thesis.

Appendix B describes, and justifies, the statistical significance testing methods used when analysing the empirical investigations undertaken in this thesis.

Appendix C describes an open source cross platforms CGP implementation which was developed during the PhD program.

Chapter 2

NeuroEvolution

NeuroEvolution (NE) [64,112,313] is a sub field of Machine Learning (ML) which combines both Evolutionary Algorithm (EA)s and Artificial Neural Network (ANN)s. This chapter introduces the field of NE with a focus on the more popular NE methods. This is followed by a review of the wider NE literature which is used to guide the research presented throughout this thesis.

2.1 Structure of this Chapter

Section 2.2 introduces the basic principles concerning NE along with a description of a simplistic NE method. Section 2.3 describes a number of advantageous properties of NE over other ANN training methods. Section 2.4 discusses the scope of NE in terms of which aspects of ANNs it can be used to optimise. Section 2.5 describes a range encoding schemes used by NE to describe ANNs. Section 2.6 provides a detailed review of many popular NE methods. Section 2.7 presents a detailed literature review of the general field of NE. Finally, Section 2.8 gives a closing discussion of the chapter.

2.2 Basic Principles

NE is the application of EAs to the training of ANNs. Therefore a brief introduction of both EAs and ANNs is provided. This is followed by a description of using a simple Genetic Algorithm (GA), a type of EA, as an ANN training method. This summary of the basic principles surrounding NE is intended to act as a reminder to those already familiar with the field, and to focus on concepts which will be relevant later in the thesis.

2.2.1 Evolutionary Algorithms

EAs [72, 228] represent a family of stochastic[1], heuristic[2] population based[3] search techniques based on Darwinian evolution [49][4].

EAs are initialised by creating a population of random solutions to a given problem encoded in what are termed chromosomes or genotypes; taking the terminology from the biology from which they are inspired. Each of the chromosomes are then decoded into their corresponding solutions, termed phenotypes, and assigned a fitness value proportional to their effectiveness at solving the given task. The fitness of each chromosome is then used to determine which are selected, mimicking the concept of "survival of the fittest" or "natural selection". The selected chromosomes are then used as parents in the creation of new child chromosomes; the unselected chromosomes are discarded. Child chromosomes can be created from their parents via asexual or sexual reproduction. In the asexual case, the created children are clones of the parents with random alterations applied; these alterations are termed mutations again using biological terminology. In the sexual case, the child chromosomes contain genetic material from two or more of the parent chromosomes; mutation can then also be applied. This next generation of chromosomes is then comprised of the newly created child chromosomes; with or without the selected parents. This process of assigning fitness, selection and reproduction is then repeated until a termination condition is reached. Typical termination conditions specify a maximum number of generations and a target fitness value i.e. when a solution is found which is considered satisfactory. The entire process is shown in Algorithm 1.

Algorithm 1 Basic Evolutionary Algorithm

```
Initialise a population of random chromosomes
while Termination Conditions not met do
    Calculate chromosome fitness
    Select chromosomes for reproduction
    Reproduce via mutation and/or crossover
end while
```

[1]Involves a random element in the search process.
[2]Uses experience and/or learning to guide the search.
[3]Makes used of many agents working together or independently.
[4]Or occasionally Lamarckian Evolution [171]

The creation of EAs is attributed to the contributions of three groups who independently conducted research in the area unaware of each other's work. Each of their works is now considered a separate sub-field of EAs called: Evolutionary Programming (EP) [67], GAs [113] and Evolutionary Strategies (ES) [237]. Interestingly however, the notion of using artificial evolution as a problem solving tool was also proposed by Alan Turing in 1948 (long before the other discoveries), his essay was dismissed by his employer, the grandson of Charles Darwin, as a "schoolboy essay" [277]. For a more detailed and complete history of EAs see [66].

2.2.2 Artificial Neural Networks

Like EAs, ANNs are also heavily inspired by biology and represent, to a greater or lesser extent, a simplified abstraction of real neural networks found within the brains of animals. ANNs comprise a weighted directed acyclic/cyclic graph where each node implements a transfer function which approximates a biological neuron.

Biological neurons[5] take the form of that shown in Figure 2.1. Each neuron gathers at its dendrites (inputs) signals from other neuron's axons (outputs). These input signals propagate along the dendrites down into the soma (cell body). These signals cause the membrane potential of the neuron to increase or decrease depending upon whether the signal is excitatory or inhibitory respectively. If the membrane potential reaches a certain upper threshold, the neuron fires, propagating a signal down its own axon and out to other neurons; otherwise no output signal is produced.

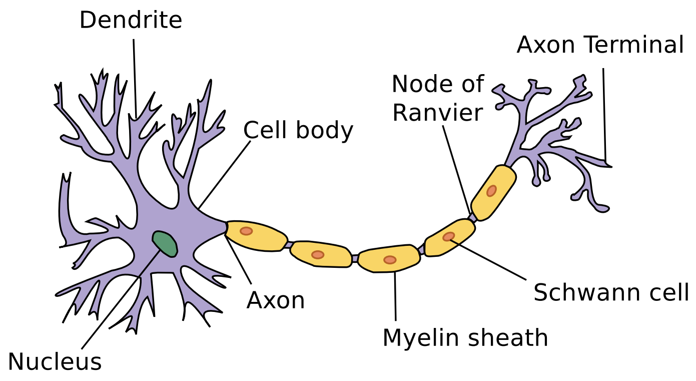


Figure 2.1: Simple depiction of a biological neuron taken from `www.newworldencyclopedia.org`

[5]The existence of neurons was first discovered by Santiago Ramón y Cajal the late 19th century which resulted in him receiving a noble prize for his contributions

Signals are passed between neurons, from axons to dendrites, at synaptic connections. Synapses are the junctions between axon terminals and dendrites. They release a neurotransmitter when excited by a signal from the axon. This neurotransmitter then propagates across the gap between axon terminal and dendrite (synaptic cleft) causing a signal to be induced in the dendrite. The magnitude of this signal is proportional to the connection strength of the synapse. This signal is excitatory or inhibitory depending on the type of synapse between axon and dendrite.

In biological neural networks the signals passed between neurons are spikes of potential difference between the ions internal and external to the neuron. ANNs which model this spiking behaviour are referred to as Spiking Neural Networks (SNN). Neuron models commonly used by SNN include the Hodgkin-Huxley model[6] [111], integrate and fire model[7] [1] and the Izhikevich neuron model [131].

However, the majority of NE methods, and ANN training methods in general, construct ANNs consisting of non-spiking neuron models. This is most likely because simulating non-spiking ANNs is much less computationally expensive whilst being powerful enough to be applied to many applications. Non-spiking neuron models do however contain many of the characteristics of their spiking counterparts. Their output is determined by their inputs and the strength of the connection weights (synapse connections). Additionally, although the signals passed between neurons are not actually spikes, they can be thought of as an abstraction of this concept. For instance, if a neuron outputs 0 or 1, 1 can be considered spiking and a 0 can be considered not spiking. Alternatively, if a neuron outputs a floating point value in the range [0,1], this can be considered as a spiking (firing) rate; with zero being no spikes, 0.5 being alternating spikes and no spikes, and 1.0 being continuous spiking.

Nearly all non-spiking neuron models can be generalised to the model shown in Figure 2.2. The inputs x_i are the inputs from previous neurons, the weights w_i are the connection strengths of the synapses, the function $\varphi()$ describes the transfer function (internal logic) of the neuron, the bias b is used for any internal thresholds, and the output y is the output of the neuron provided by $\varphi()$.

Non-spiking neuron transfer functions are typically a function of the weighted sum of

[6]The implementation of the Hodgkin-Huxley neuron model was so significant that it resulted in Alan Lloyd Hodgkin and Andrew Huxley receiving the Nobel Prise in Physiology or Medicine in 1963.

[7]Interestingly the integrate and fire neuron model was developed by Louis Lapicque in 1907 [1], long before the mechanics of biological neurons were understood.

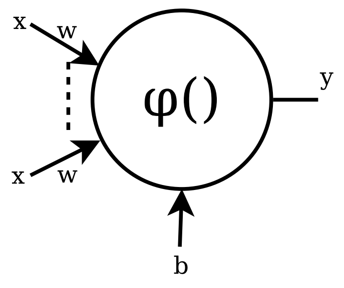


Figure 2.2: Simple generalised model of an artificial neuron.

inputs. The weighted sum of inputs S is equivalent to the dot product of the neuron's inputs as a vector X and the neuron's connection weight as a vector W, as shown in equation 2.1. The transfer function of a non-spiking neural model usually takes the form of Equation 2.2, where $\varphi\left(\right)$ changes from model to model.

$$S = W \cdot X = \sum_{i=0}^{n} w_i x_i \tag{2.1}$$

$$y = \varphi\left(S - b\right) \tag{2.2}$$

There are many neuron transfer functions found in the literature [54] but in this chapter only the most commonly used are introduced. S will be used to denote the weighted sum of inputs and $\varphi\left(\right)$ to denote the particular transfer function of each model.

The McCulloch and Pitts model, developed by W. McCulloch and W. Pitts [196] in 1943, is the earliest artificial neuron model. The model is also commonly referred to as the Heaviside step response model. The McCulloch and Pitts model outputs a '1' if the weighted sum of inputs S is greater than a bias b, otherwise it outputs '0'. The transfer function takes the form of Equation 2.3 and is plotted graphically in Figure 2.3; with the bias set as zero.

$$\varphi\left(x\right) = \begin{cases} 1, & \text{if } x \geqslant 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2.3}$$

The Linear Combination neuron model is simpler than the McCulloch and Pitts model but does not model the threshold nature of real biological neurons. Instead the linear combination neuron model simply returns the weighted sum of inputs as its output. The trivial transfer function is given in Equation 2.4 and plotted graphically in Figure 2.3.

$$\varphi\left(x\right) = x \tag{2.4}$$

The logistic sigmoid function, often simply referred to as the sigmoid function in the ANN literature, is likely the most popular neuron transfer function. It is a continuous non-linear function which produces an output in the range [0,1]. The logistic sigmoid function is given in Equation 2.5 and plotted in Figure 2.3.

$$\varphi\left(x\right) = \frac{1}{1+e^{-x}} \tag{2.5}$$

Finally, another popular family of neuron transfer functions are radial basis functions. A radial basis function is any function which satisfies Equation 2.6; that is to say, symmetrical around an origin. It can also be defined a function which only depends upon the distance from an origin. A commonly radial bias function used by ANNs is the Gaussian function given in Equation 2.7 and shown in Figure 2.3. The value of μ determines the offset from the origin and the value of σ determines the width of the Gaussian curve.

$$\phi\left(x\right) = \phi\left(\|x\|\right) \tag{2.6}$$

$$\varphi(x) = e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.7}$$

Although individual neuron models may be of interest from a biological standpoint, and in some cases capable of simple tasks, they are much more powerful when configured into networks of neurons. In fact, feed-forward networks of logistic sigmoid or radial basis functions have been shown to be capable of universal function approximation using a finite number of neurons; [48] and [221] respectively. Additionally, Recurrent Artificial Neural Network (RANN)s have also been shown to be universal dynamical system approximations [71,249]. This means ANNs and RANNs can be applied to a very wide range of applications *iff* methods are available to configure their topology and connection weights. The vast

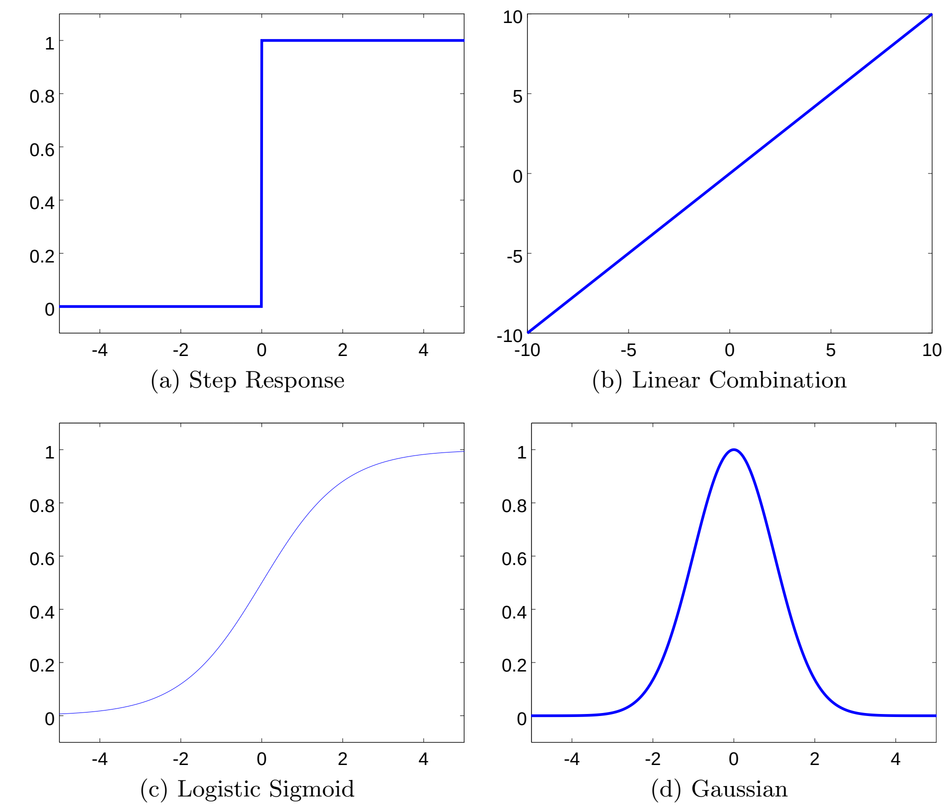


Figure 2.3: Common non-spiking neuron models.

majority of ANN research, including NE, attempts to address this challenge.

The following sub sections describe two very distinctive, non NE, ANN training methods; back prorogation and Reservoir Computing (RC). Back prorogation is introduced as it is the most widely adopted training method for ANNs and is reference throughout this thesis. RC is also introduced as an interesting contrast to back prorogation, and because it is also referenced in this thesis.

2.2.2.1 Back Propagation

The back propagation algorithm is by far the most widely used method for training multi-layered feed-forward ANNs. Its discovery is accredited to David Rumelhart et. al. in 1986 [245], but as described in [304], its history is complicated. Before the discovery of back propagation, ANNs were restricted to one input layer and one output layer. These zero hidden layer networks could only be applied to tasks which were linearly separable, Figure 2.4; such as implementing AND or OR logic gates. Zero hidden layer networks could famously not implement an XOR logic gate; Figure 2.4. The realisation that zero hidden layer ANNs could only solve linearly separable problems, coupled with the fact

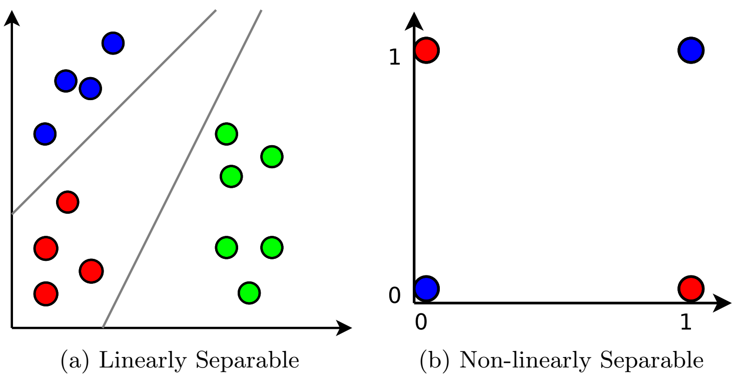


Figure 2.4: Example of data which is linearly separable (a) and the XOR gate which is not linearly separable (b).

there were no training methods for multilayer networks, lead to a decline in ANN interest during the 1970's; until methods such as back propagation and Boltzmann machines [109] were created.

Before the invention of back propagation, networks with zero hidden layers, Figure 2.5, could be trained by updating the weights in order to reduce the network error [8]. Equation 2.8 shows a simple weight update rule where w_i is the weight between the output neuron and the input i and Δw_i is the change in connection strength. Here each weight is increased or decreased depending upon whether it will reduce or increase the network error. For this update rule to operate, the change in network error is required as a function of each connection weight.

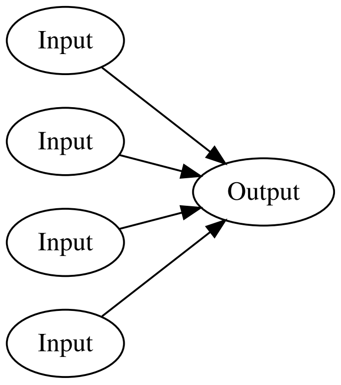


Figure 2.5: Zero hidden layer feed-forward neural network with four inputs and one output.

The behaviour of a simple zero hidden layer network, such as Figure 2.5, is described by Equation 2.9; where y is the network output, w_i is the weight between input neuron i

[8]Where the network error is proportional to how badly the neural network performs on a given task i.e. it is strictly supervised learning.

and the output neuron, x_i is the value of input neuron i, and $\varphi()$ is the transfer function of the output neuron. The error of this simple network is then given by Equation 2.10; where E is the error, d is the desired output and y is the actual output. The error is given in this strange form as it is required that it is easily differentiable; as shall be seen later. The effect of changing each weight w_i on the network error is then given by Equation 2.11. Producing the final weight update rule Equation 2.12; where ε is the leaning rate controlling the level of exploitation and exploration. This weight update rule can then either be applied after the ANN execute one input sample, or each Δw_i can be averaged over the entire training set and then applied once. The process is then repeated until the desired behaviour is observed or a training time budget is reached. It should be noted that it is a requirement of this method that the neurons transfer function $\varphi()$ be differentiable.

$$w_i = w_i + \Delta w_i \tag{2.8}$$

$$y = \varphi\left(\sum_i x_i.w_i\right) \tag{2.9}$$

$$E = \frac{1}{2}\left(d - y\right)^2 \tag{2.10}$$

$$\frac{\delta E}{\delta w_i} = \frac{\delta E}{\delta y}.\frac{\delta y}{\delta w_i} = (y - d)\varphi'(\sum_j x_j.w_j).x_i \tag{2.11}$$

$$\Delta w_i = \varepsilon(y - d)\varphi'(\sum_j x_j.w_j).x_i \tag{2.12}$$

The back propagation algorithms follows a very similar logic to that previously presented and can be applied to networks with multiple layers, see Figure 2.6. The difference being that the output of the network is now a function of the hidden nodes which are in turn a function of other nodes leading back to the inputs. An equation of the form given in Equation 2.9 can then be written for a multiple hidden layer networks and coupled with Equation 2.10 to form an error function in terms of each connection weight. This error function can then be differentiated with respect to each connection weight; as was done in Equation 2.11 for the zero hidden layer case. This $\frac{\delta E}{\delta w_i}$ can then be used to calculate Δw for each connection weight and used to train the ANN.

The algorithm is called back propagation as the error value is effectively back propagated though the network in order to calculate $\frac{\delta E}{\delta w_i}$ for each weight. That is, in order to calculate Δw_i for a weight close to the inputs, all of the other $\frac{\delta E}{\delta w_i}$ between this node and the output need to have been previously calculated. The $\frac{\delta E}{\delta w_i}$ are therefore calculated from

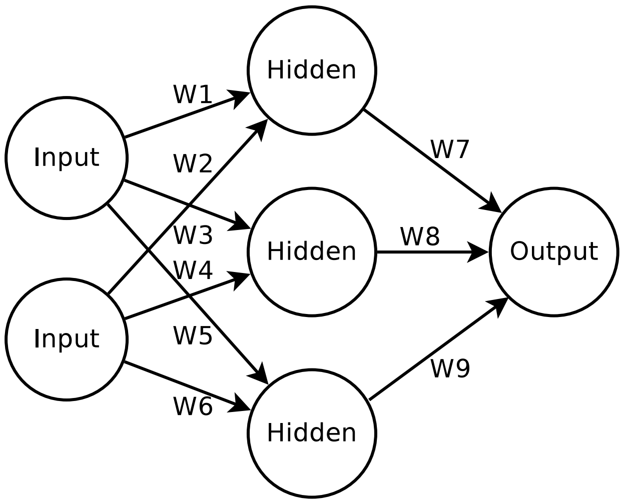


Figure 2.6: Simple ANN comprising two inputs, one hidden layer containing three neurons, and one output neuron.

the outputs moving backwards towards the inputs.

2.2.2.2 Reservoir Computing

Reservoir Computing [182] is a relatively recently proposed training method for RANNs. Within RC there are two distinct sub fields: Liquid State Machines (LSM) [185] and Echo State Networks (ESN) [133]. LSM represents the application of RC to the training of spiking ANNs and ESN represents its application to the training of non-spiking ANNs. As this is thesis is focused on non-spiking ANNs, ESNs are now briefly described.

As ESN is a training method for RANNs, it is mainly applied to series forecasting or transforming one temporal input signals into another. For instance, a popular demonstration of ESNs is the conversion of an input sinusoidal waveform into an output saw tooth waveform.

In their basic form, ESN are initialised by generating a random[9] "reservoir" of sparsely connected neurons connected via feed-forward and recurrent connections. A proportion of these hidden neurons also connect to the program inputs. No output neurons are needed in the initial stages of training. The randomly generated connections are also given randomised connection weights.

Once the reservoir has been created, each of the training set sample inputs are applied to the network in turn, and each neuron in the reservoir updated. The state (output)

[9]In practice random reservoirs are rarely used in favour of reservoirs which exhibit a set of desirable properties.

of each neuron is recorded after each training sample is applied in a $h \times n$ matrix S; where h is the number of hidden neurons in the reservoir and n is the number of training samples. Note that the state of the reservoir is both a function of the current inputs *and* the previous state of the reservoir due to the recurrent connections.

Once this initialisation process is complete, the output neurons are added to the network. Each output neuron is connected to *every* hidden neuron in the reservoir. The connection weights connecting the hidden neurons to the output neurons are specified in an $o \times h$ matrix W_{out}; where o is the number of outputs and h is the number of hidden neurons in the reservoir.

Finally the desired outputs of each training sample are stored in a $o \times n$ matrix D; where o is the number of outputs and n is the number of training samples.

Therefore what is known is the state of the network after each applied training sample S, and the desired outputs of each training sample D. What is needed to be determined are the connection weights W_{out} which transform the corresponding states of the network S into the desirable outputs D. The tasks can be formulated as the matrix equation given in Equation 2.13; where the unknown variable is W_{out}; the output connection weights.

$$D = W_{out}S \tag{2.13}$$

There are many methods available for solving Equation 2.13 in order to determine W_{out}; these include linear regression methods or pseudo inverse techniques. In RC a range of methods are used. Note that it is very unlikely that a value of W_{out} exists which perfectly maps S onto D. Therefore the found value of W_{out} will typically produce an approximate mapping of S onto D.

Once W_{out} has been determined, the output connection weights can be applied to the network. The RANN will then, to a greater or lesser extent, transform the given training data into the desired outputs; thus completing the training.

An interesting property of RC is that the training data only needs to be applied once followed by solving a single, albeit large, matrix equation. This typically makes training ANNs using RC much faster than methods such as back propagation, which requires the training data to be applied multiple times during training.

2.2.3 NeuroEvolution

Now EAs and ANNs have been introduced, a simple NE method can be described. The described NE method is based on Conventional NeuroEvolution (CNE) which is described later in Section 2.6.1.

A simple application of EAs to the training of ANNs is the use of a standard GA to evolve the connection weights of a fixed topology network. Take an ANN of the form shown in Figure 2.6. Each of the connections in Figure 2.6 has an associated connection weight (w_i). Assume also that each neuron uses a logistic sigmoid transfer function.

Here the NE method uses genotypes which comprise a string of floating point values describing the connection weights of each connection in the ANN. The genotypes are decoded into their corresponding phenotypes by assigning the connection weights to the predetermined topology; with the same connection weights always placed on the same connection. A genotype of this NE method, for this given topology, therefore takes the form of that given in Equation 2.14.

$$\{w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9\} \tag{2.14}$$

The fitness assigned to each genotype is then a measure of how well the decoded ANN performed on a given task. This fitness is then used to guide selection. Mutation is implemented by changing a given percentage of the connection weight values to a new random value. Crossover is implemented by creating child chromosomes which contain a proportion of the genes values from one parent, and the remainder from the other. The initial population is created by generating genotypes of random connection weight values. Finally, the search terminates when an ANN is found which performs suitably on the given task; or the maximum number of generations is reached.

As can be seen in this simple example, EAs can very easily be applied to the training of ANNs. However, even such a simple NE method has numerous advantages; as is discussed in Section 2.3. Additionally, many NE methods go beyond the simple application of a GA to the configuration of connection weights; as is described later in Section 2.6.

2.3 Advantages of NeuroEvolution

This section provides a number of possible advantages of using NE to train ANNs. These advantages are summarised as follows:

- Suited to supervised and reinforcement learning applications
- Capable of training feed-forward and recurrent ANNs
- Capable of training shallow and deep topologies
- Capable of manipulating ANN topology
- No transfer function limitations
- Capable of creating homogeneous and heterogeneous ANNs
- Always returns a solution

Of course there are also disadvantages of NE. For instance, like back propagation, NE is an iterative training process. This means that order of the training time is much larger than for non-iterative training methods such as RC [182]. Additionally, NE is a stochastic process which means the training time to produce its best solution varies, as does the quality of the final solutions found.

2.3.1 Application to Reinforcement Learning

In the field of machine learning there are three main types of learning; supervised, unsupervised and reinforcement. These are now briefly described.

The defining feature of supervised learning [163] is that it makes use of labelled data, that is, each training sample takes the form (x_i, y_i) where x_i is an n-dimensional input vector and y_i is a discrete or continuous scalar associated with the input vector. The system must learn to classify, or predict, the value of y_i based on x_i. Supervised learning systems are trained by showing sequences of training sets (x_i, y_i) in order for the system can encode a relationship $f()$ between x_i and y_i; $y_i = f(x_i)$.

Unsupervised learning [77], in contrast with supervised learning, uses unlabelled data x_i and attempts to find patterns and trends within that data. Unsupervised learning systems are so called as they have no predefined measure of effectiveness and hence are unsupervised. Unsupervised learning is often used to cluster similar data sets or to predict future data sets based on previous examples.

Reinforcement learning [138] is where a system has to learn to perform a given task without any examples of desired behaviour, but where its behaviour can be given a performance value. The system has to take actions based on its current, and sometimes previous, inputs which are then given a score proportional to the desirability of the result of those actions. For example, if a human were to balance a broom upon their hand they have to learn in which direction to move their hand in order to keep the broom balanced. Bad actions in a given situation would see the broom fall from its central position and good movements would see the broom remain in, or approach, the central position. By noting which actions in which situations result in positive behaviour a human can learn to balance the broom. Reinforcement learning works much the same, there is no set of training examples to learn from, instead a system is given a score indicating its current performance i.e. the distance the broom is from vertical.

Generally the most popular training algorithms for ANNs allow ANNs to either be applied to supervised or unsupervised learning tasks. Common supervised learning methods include back propagation [245] and radial basis function networks [180]. Common unsupervised learning methods include Hopfield networks [116] and restricted Boltzmann machines [260]. Additionally resent developments in deep belief neural networks, such as the application to playing GO [257], demonstrate a mechanism for training ANNs for reinforcement learning tasks. In such work the ANNs are trained using supervised learning methods to predict the move chosen by expert GO players given a board state. This work effectively trained the ANNs as classifiers; a supervised learning task. The trained networks are then further trained using policy gradient reinforcement learning [270] to further the accuracy of the classifier. The trained networks are then used to estimate values of board states which is in turned used by Monte Carlo tree search [46].

As NE methods are based on EAs, they assess each solution by assigning a fitness value. This fitness value is representative of the ANN's overall performance on a given task. Therefore, NE allows ANNs to be applied to reinforcement learning type tasks; tasks to which many standard ANN training methods cannot be applied. Additionally, as supervised learning tasks can be framed as reinforcement learning tasks, by looking at the overall performance rather than the performance on any particular sample, NE can also be used to train ANNs for supervised learning type tasks. As EAs cannot be applied to unsupervised learning this restriction also applies to NE.

Therefore NE represents a class of ANN training algorithms which can apply ANNs to a wide range of applications; both supervised and reinforcement learning.

2.3.2 Recurrent Artificial Neural Networks

Another important benefit of NE is that it is not influenced by the topology of the ANN being trained. For instance, although back propagation can be extended to be capable of training RANN [19], by effectively "unwinding" the network, there are many unresolved issues [182]. One such issue is that as the error signal is propagated though the ANN it becomes weaker and the effect of training is diminished. This problem is increased if the error signal must traverse the network multiple times following the recurrent paths. Additionally, the computational expense of training increase proportionally to the depth of recurrence considered. Finally, as the error is only fed back a given number of times through the network, the result is always an approximation to the error at any given node, not the actual error.

As NE is not a function of the ANN topology these limitations simply do not apply. This is because NE training methods are not concerned with the internal state of the network at any given time, only the overall behaviour. Therefore, as long as a RANN can be applied to a problem and given some measure of fitness, it is no more complex, and has no more limitations, than training feed-forward ANNs.

2.3.3 Deep Artificial Neural Networks

Since the creation and demonstration of deep belief networks [108] by Geoffrey Hinton and Simon Osindero in 2006, the ability to train deep ANNs has been seen as increasingly significant. This is heightened by the fact that it is thought that deep ANNs can more efficiently implement a given function, in terms of the total number of nodes, than shallow networks [27].

However, back propagation alone has been shown to struggle to train deep ANNs [78, 178]; mainly due to the error gradient becoming smaller in magnitude as more layers are propagated. In resent work this issue has been overcome [79] via the use of rectifier neuron models.

Interestingly, as is described in the previous section, NE training algorithms are not a function of the ANN topology. NE can therefore be used to train deep ANNs without any alteration to the algorithm or use of rectifier neuron models.

2.3.4 Topology

It is known that the choice of topology has a large influence on the effectiveness of back propagation [178]. Additionally, there are no formal rules regarding the optimal/suitable choice of topology, only various "rules-of-thumb" [47]. A solution to this issue is to use training methods which adapt and find suitable topologies during training. To this end there are two common approaches used. 1) Constructive (growing) methods [168] start with a small number of hidden neurons (zero or one) and constructively add neurons during the search. 2) Destructive (pruning) [238] methods train using a large number of hidden neurons and destructively remove those which are unnecessary after the training is complete.

The constructive method of adding neurons is often considered suitable as it promotes the use of a minimal network sizes; aiding the generalisation of the ANN whilst keeping the dimensionality of the search low. When ANNs are trained on a data set for too long they begin to become capable of more accurately classifying samples within the training set at the expense of been able to correctly classify unseen samples. This phenomenon is referred to as over training. The ability to avoid this behaviour is referred to as generalisation. One method for avoiding over training is to limit the number of nodes available, ensuring that the ANN is not capable of very accurately classifying the training set, in order to preserve its ability to classify unseen items. For this reason constructive methods are often used as they are likely to create smaller, rather than larger, networks. It is also common to use regularization techniques with constructive method where the network is penalised for being too large; again favouring smaller networks. This method is often likened to the principle of Occam's razor [30], where simpler solutions should be favoured over complex solutions if they are equivalent in quality.

Destructive methods (pruning) take the opposite approach. They train excessively large networks, which typically learn quickly and over train. Once trained, sections of the network which contribute the least to the output are removed. This method is thought to reduce the complexity of the network, reduce training time and aid generalisation. Destructive methods can be applied to any ANN regardless of how they have been trained and can therefore be used alongside other training methods, including NE [254], to aid generalisation.

However, destructive and constructive methods have their limitations. For instance iteratively adding neurons during the search is akin to topology hill climbing and is subsequently likely to become trapped in topology local optima [15]. Destructive methods also have the issue of the user having to decide a suitably large number of neurons and requires careful removal of sections of the ANN post training.

In most cases however, the choice of topology is left to the user and a certain level of trial and error is often required in order to find suitable topologies. This is where many NE methods can offer a strong advantage. In the simplest case a standard GA [113] could be used to manipulate the number of layers and nodes per layer of ANN which is then trained using back propagation. This effectively removes the trial and error burden from the user and is likely to much more effectively search the space of possible topologies than a simple constructive method or blind trial and error.

Interestingly it has previously been shown that topology optimising/adapting NE produces results which are comparable to those achieved using back propagation and hand-crafted topologies [36]. This demonstrates the benefit of topology optimising NE methods; the topology is self-optimising and does not have to be hand-crafted by the user.

2.3.5 Transfer Functions

The most commonly used ANN transfer function is the logistic sigmoid. This transfer function is likely popular because: 1) it is limited to a [0,1] range, 2) it is non-linear, 3) it has been shown capable of universal function approximation [48] and 4) is differentiable and therefore compatible with gradient descent type algorithms.

However, it is unlikely that the logistic sigmoid is the most suitable transfer function for all applications. For instance, if an ANN were to be used to implement a Boolean circuit, the step function might be more suited. In this case back propagation could not be applied as the step function cannot be differentiated[10].

Therefore an advantage of NE is that it is not dependent on the transfer functions being used. This means that ANNs of any transfer function [55] can be trained.

[10]In fact the step function can be differentiated to produce a infinitely tall, infinitesimally narrow, delta spike, but this cannot be used by the back propagation algorithm.

2.3.6 Heterogeneous Artificial Neural Networks

Although many ANN training methods, such as back propagation, are capable of training heterogeneous ANNs[11], typically they do not[12]. This may be due to the complexity it brings to the implementation, or it may not be considered a beneficial characteristic.

Regardless, since the complexity of many NE algorithms does not depend upon the ANNs being trained, they can be easily used to train heterogeneous ANNs.

2.3.7 Always Returns a Solutions

An advantage NE methods share with some ANN training methods is that they maintain a current best solution during training. For instance, in a given time constraint one can train an ANN using NE for the time available, and then use the solution found after that given time budget.

However, some ANN training methods do not have this property. For instance if RC [182] is not given enough time to complete, one is left without a solutions; partially trained or otherwise.

Therefore in certain applications NE is well suited as it maintains a current best solution during training; a benefit shared with back propagation.

2.4 Scope of NeuroEvolution

In their entirety, ANNs are described by their: connection weights, topology and neuron transfer functions. The scope of a given NE method can therefore train an ANN by manipulating its connections weights and/or topology and/or neuron transfer functions. This has given rise to many NE methods optimising different aspects of ANNs.

This section discussion the various aspects of ANNs which can be manipulated by NE. This also includes types of NE which do not manipulate ANNs directly, but create *rules* which then themselves act as training methods.

[11]Provided the transfer functions are differentiable.

[12]Although many ANNs do use a linear combination neuron at the output of a network comprising logistic sigmoid which could be considered a heterogeneous ANN.

2.4.1 Weight Evolution

Much of the early work in NE evolved the connection weights of fixed topology ANNs as a vector of real values [240]. When only evolving connection weights the topology of the ANN must be chosen in advance by the user. Weight only evolution has many of the drawbacks of other training methods which manipulate connection weights alone, such as back propagation; they only search a small section of the overall ANN search space. It has also been shown that the choice of topology has a large impact on the effectiveness of connection weight only evolution [112]; requiring the user to try many topologies or use task specific information to select a suitable topology in advance; information which is often unavailable.

Weight only NE does however have many advantages over traditional gradient based methods. Weight only NE does not require the neurons transfer functions to be differentiable in order to calculate the error gradient. Gradient based methods struggle to train deep ANN [78], due to the gradient being reduced as it is passed through the network and at the extremes of the logistic sigmoid function; restrictions which do not apply to NE. Additionally, weight only evolving NE can be used for reinforcement learning tasks; something which back propagation cannot natively handle. Weight evolving NE methods can also be applied to recurrent network structures. Finally, gradient based methods are prone to becoming trapped in local optima and are highly dependent upon the initial connection weights; NE is far less dependent upon the initial weights and is capable of escaping local optima.

2.4.2 Topology Evolution

In the NE literature it is thought that topology, as well as connection weights, are highly important in the training of ANNs[13] [64, 313]. Theoretically an ANN can be thought of in terms of a topology and connection weight search space; or as weight spaces associated with any given topology. Therefore, using only fixed topology limits the search to one subset weight space within the wider topology space; a possible disadvantage of methods which only manipulate weights. Clearly, if a suitable topology is known in advance it could be used to decrease the dimensionality of the search. However, often this is not known in advance and since the effectiveness of the search is strongly dependent upon selecting a

[13]Methods which evolve connections weights and topology are sometimes refereed to as Topology and Weight Evolving Artificial Neural Networks (TWEANNs) [267].

suitable topology, this is clearly a major issue of manipulating connection weights alone.

Topology evolving NE methods are therefore thought to be beneficial as they configure the topology for a given application; without the user having to know a suitable topology in advance of training. They may also be capable of utilising relationships between topology and connection weights during the evolutionary search and are likely to evaluate unusual topologies which would otherwise typically not be considered by a human designer.

2.4.3 Transfer Function Evolution

Many ANN training methods make use of only a single type of neuron transfer function; typically logistic sigmoid or Gaussian. Additionally many ANN training methods bring restrictions to the types of neuron transfer functions which can be used; such as back propagation requiring the neurons to be differentiable. However, it has been shown that the neuron transfer function strongly influences the capability of an ANN [55, 313]. Interestingly, NE can be used to manipulate the neurons transfer functions during training and brings no restrictions to the types of transfer functions which can be used. Using this method heterogeneous ANNs of many different transfer functions can be evolved resulting in a search of the node transfer function space.

Therefore NE can be used to select suitable transfer functions for a given task and easily create heterogeneous ANNs.

2.4.4 Learning Rule Evolution

A very different application of EAs to ANNs is not to manipulate the network itself, but to evolve learning rules which are themselves used to train ANNs [313]. This can be achieved using the genotypes to encode dynamic properties of the neurons, enabling them to adapt themselves during their lifetime [25, 37]. Interestingly, in [37] the use of learning rule evolution led to the rediscovery of the delta rule (aka Widrow-Hoff rule); a form of gradient decent used to train ANNs. Learning rule evolution has two main applications: discovering new learning rules and enabling continuous learning.

2.5 NeuroEvolution Encoding and Decoding

This section covers the common encoding schemes used by NE to describe functioning ANNs. The process of morphogenesis, decoding a genotype into a phenotype, is dependent upon which aspects of the ANN each genotype describes and the type of encoding used by the genotype.

Section 2.5.1 first describes which aspects of the final ANN can be described by each genotype and Section 2.5.2 describes the types of encoding methods which are commonly used.

2.5.1 Level of Encoding

Which aspects of a complete ANN are described by each individual genotype varies between NE methods. However they can typically be separated into whether each individual genotype represents a complete ANN or subcomponents of an ANN.

2.5.1.1 Complete Neural Network

When using complete ANN encoding schemes, each genotype in the population describes a complete ANN. This can either take the form of a vector of weights for a fixed topology ANN or can be a complete description of weights and topology (and even node transfer functions).

When evaluating a genotype. the assigned fitness for each genotype is simply a measure of how effective the ANN, described by the given genotype, performs on a given task. In this regard it is similar to traditional EAs.

2.5.1.2 Subcomponents of Neural Network

Subcomponent methods are where each genotype does not describe a complete ANN, but describes a subcomponent of an ANN. When using subcomponent methods each genotype typically describes an individual neuron or an individual connection weight. Complete ANNs are then constructed by combining many individual genotypes.

When describing complete ANNs using one genotype assigning fitness is simple; as each genotype can be evaluated separately. However, when using subcomponent methods each genotype represents only a partial solution, and so assigning fitness is not so straightforward. The exact methods for handling fitness assignment, and other details including how to construct a complete network from many subcomponents, vary between subcomponent NE methods. This topic is discussed further for the subcomponent NE methods described in Section 2.6.

2.5.2 Type of Encoding

Whereas the previous section described which aspect of ANNs each genotype represents, this section describes *how* each genotype is transformed into its section of the final ANN. The types of encoding used by NE mirror those commonly used by EA methods.

There are three main types of encoding used by NE (and EAs in general): direct, indirect and developmental. Direct encoding is where the genotype and phenotype are identical. For instance, if the weights of a neural network were to be evolved using a direct encoding method, the weights could be described by the genotype as a string of values. To create the phenotype these values would simply be directly applied to their corresponding connections; no decoding is necessary.

Indirect encoding is where the genotype has to be decoded into what it describes. For instance, a NE genotype may describe a mathematical function which is then used to assign the weight to a given ANN. In this case the connection weight could not be directly applied from the genotype, but calculated indirectly from the genotype.

Developmental encoding is where the genotype describes rules which are followed to grow and adapt an ANN. Adaptation can influence the weights and topology through the addition and removal of neurons and connections. This adaptation can also be a function of the network's inputs, so development can be dependent upon the environment and change over the ANNs lifetime.

2.6 Review of NeuroEvolutionary Methods

There are many types of NE methods found in the literature. This section reviews a range of the more popular NE methods. This review is then used to draw conclusions and insights concerning the NE literature as a whole. A high level summary of the review is given in Table 2.1.

To date there have been two major reviews of NE, the first in 1999 [313] and the second in 2008 [64]. In these reviews the NE methods are categorised according to a number of criteria. Firstly they are categorised by their representation scheme: direct, indirect or developmental. Secondly they are categorised into whether they evolve connection weights and/or topology and/or transfer functions and/or learning rules. Finally they are categorised into whether they evolve feed forward and/or recurrent network topologies.

This review plans to extend those previously made by building up a taxonomy of NE methods. This will include many of the previously used categories as well as introducing new criteria. The new criteria comprise:

- The presence of genetic redundancy
- The presence of program bloat
- Whether crossover is used
- Incremental topology adaptations
- Whether non-functioning ANNs can be created

Genetic redundancy has been widely studied in the field of EAs [74] and is thought to offer a great advantage to the search through the process of neutral genetic drift [74]. As NE utilises EAs it is also likely to benefit from the presence of genetic redundancy. Therefore, each NE method reviewed will be considered in the light of genetic redundancy. Specifically, whether genetic martial can be active or inactive in the semantics of the decoded phenotype. For all NE methods considered, it is possible for a weight connection to be set to exactly zero. This means that all connections which lead to that connection, if not used elsewhere, would be inactive. However, as this applies to all weight evolving NE methods, and due to the fact that it is highly improbable for a connection gene to be set to exactly zero, this is not considered in each case.

Many EAs, or specifically Genetic Programming (GP) methods, suffer from program bloat [181, 256]; the uncontrolled growth in program size during the evolutionary search. This is a major issue, specifically for tree-based GP, as it results in much longer training times and produces final solutions which are extremely complex and computationally expensive to execute. Additionally, the final solutions are harder to reason about limiting what can be learnt from the discovered solutions. Again, as NE is based on EAs, it is likely that some NE methods also suffer from program bloat. This is therefore an important criterion to categorise NE methods by.

In the NE literature naïvely using crossover is considered a disadvantage: "*One of the main problems for NE is the competing conventions Problem, also known as the Permutations Problem*" [236]. "*Competing conventions means having more than one way to express a solution to a weight optimization problem with a neural network. When genomes representing the same solution do not have the same encoding, crossover is likely to produce damaged offspring.*" [266]. Therefore whether a given NE method uses crossover is another useful method for categorisation, as is whether competing conventions has been considered and what steps have been taken for its prevention.

Another often overlooked possible issue of topology evolving NE methods is the way in which topology is adapted. As has been previously discussed, iteratively adding or removing single nodes/connections via mutation is equivalent to hill climbing and is likely to result in the search becoming trapped in topology local optima [15]. Although it may often be the case that there is a continuous gradient of adding single neurons and adapting connection weights until a optimal/suitable number of nodes is reached, it is unlikely to always be the case. This issue can also be framed in terms of exploration versus exploitation, with the adding of single nodes/connections resulting in a more local search and therefore not being very explorative.

Finally, some of the NE methods discussed allow for ANNs to be described and evaluated which will never correctly function. For instance, if there were no connection to any of the available inputs. Assessing the fitness of such solutions is therefore a waste of computational time. Therefore it is of interest whether this can occur, and if so if any measures are taken to mitigate the performance cost.

2.6.1 Conventional NeuroEvolution

CNE [240, 305] is likely the earliest form of NE and operates using a simple GA which directly encodes the connection weights of a fixed topology network. Each genotype encodes the weights of a complete feed-forward or recurrent ANN which are stored as a fixed dimension vector (genotype); a simple example is shown in Figure 2.6 with the corresponding genotype given in Equation 2.14. This vector can then be subject to mutation and crossover operators similar to a standard GA.

In its most general form, CNE can also encode recurrent [305] and fully connected ANNs, Equation 2.15; where the row and column index the weight value between each neuron in the network. For example the value at row r and column c represents the weight value between neuron r and neuron c in a given ANN. As values can also be specified for recurrent connections, recurrent neural networks can also be described. To limit the types of architectures which are allowed, specific values in the $\mathbf{W}$ matrix are simply ignored.

$$\mathbf{W} = \begin{pmatrix} W_{00} & W_{01} & \cdots & W_{0M} \\ W_{10} & W_{11} & \cdots & W_{1M} \\ \vdots & \vdots & \ddots & \vdots \\ W_{N0} & W_{N1} & \cdots & W_{NM} \end{pmatrix} \tag{2.15}$$

2.6.1.1 Extensions

An extension to CNE, proposed by Christian Igel [130], is the application of Covariance Matrix Adaptation Evolutionary Strategies (CMA-ES) [99]. CMA-ES is a popular extension to ES which adapts a covariance matrix of the mutation distribution in order to make use of any correlations between the parameters under optimisation.

Another extension sometimes employed by CNE is the addition of islands and migration [93]. The inclusion of islands is a popular method for parallelizing EA to run on multiple machines/cores; with each population assigned a separate processor. This method is further taken advantage of by noting that each population is likely to converge on different solutions. Therefore, by allowing a small number of members of each population to migrate between populations "new blood" can be brought to each population preventing early convergence.

These two given extensions to CNE show how techniques and advances in the EA literature are often directly applicable to NE. This is a strong advantage of NE as the study of EA is well established and still advancing.

2.6.2 Symbiotic Adaptive NeuroEvolution

Symbiotic Adaptive NeuroEvolution (SANE) [212] is a directly encoded, feed-forward, limited topology evolving, NE method where each genotype represents an individual neuron; a subcomponent of a complete ANN. The population therefore consists of a number of individual neurons which must be combined in order to produce a complete network. Each genotype represents a hidden node describing its connectivity and connection weights with the input nodes and outputs nodes. The genotypes describe which program inputs are used and the connection weights associated with the connections. The genotypes also describe the output nodes to which the neuron connects along with the associated connection weights. Figure 2.7 shows a single SANE individual comprising two inputs and two outputs. As can be seen in Figure 2.7, evolution is left to determine which inputs and outputs are used by each genotype. Therefore it is possible for ANNs to be described where there is no connection to one or more of the inputs/outputs.

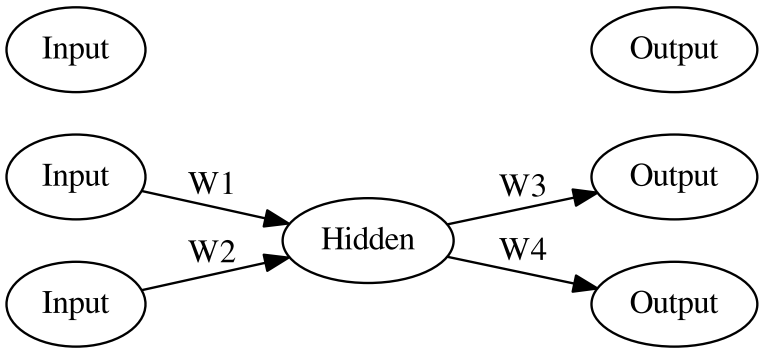


Figure 2.7: Example of a SANE individual showing that each neuron describes the inputs and outputs to which it connects and the corresponding connection weights.

SANE is described here as being capable of limited topology evolution as the networks always contain one hidden layer, a set number of nodes, a set node arity and a set number of connections to outputs. It is only the connection placement (within constraints) and connection weights which are evolved.

The fitness assigned to each genotype is the average fitness of all the complete ANNs evaluated in which it was contained. The complete networks are constructed by selecting a set number of random genotypes from the population. Networks are continued to be constructed and evaluated until each genotype has been selected a minimum number of times. As it is likely that no individual neuron can solve a given problem, they are forced to specialise into separate roles in order to solve the task; hence *symbolic* evolution. This specialisation is also though to improve diversity within the population, making better use of the crossover operator; the mutation operator is also used.

2.6.2.1 Extensions

An extension to SANE is to replace the standard GA with a Eugenic Algorithm [232]. Eugenic Algorithms are similar to GAs except instead of creating the next generation by crossing over genetic material from two parents, the children are constructed using statistics from the entire population. The application of an Eugenic Algorithm to SANE is referred to as Eugenic Symbiotic Adaptive NeuroEvolution (EuSANE) [225].

2.6.3 Enforced SubPopulation

Enforced SubPopulation (ESP) [85] is, broadly speaking, very similar to SANE. The distinguishing difference between SANE and ESP is that ESP maintains a separate sub-population for each neuron within the wider ANN. Crossover operations only take place within these sub-populations and there is no migration. The complete ANNs are then constructed by taking a single neuron from each sub-population and placing them in the *same* location in the complete network. The fact they are always placed in the same location is significant as it enables much more complex ANN structures to be possible. For instance, recurrent neurons can be added or topologies containing more than one hidden layer. This is an advantage over SANE which can only evolve single hidden layered feed forward networks.

Although SANE is thought to result in specialisation over evolutionary time, ESP enforces this from the start. ESP also ensures networks are not created which contain many similar neurons, instead of a range of different specialisations required to solve the given task. For these reasons ESP is considered to be a more efficient evolutionary technique than SANE.

2.6.3.1 Extensions

Once a solutions is approached, each sub-population used by ESP will begin to converge and diversity will be lost. This is an issue if a sub optimal solution is being converged towards or the task to be solved changes during evolution; due to the crossover operator becoming ineffective. In order to tackle this issue Delta-Coding [302] is often used alongside ESP.

Delta-Coding is used to search for small changes which improve upon the best current solution in a population. Delta-Coding is typically only used once population diversity has been lost. Delta-Coding is achieved by selecting the best member of the population and then creating a new sub-population of Δ-chromosome which are of the same form as the selected chromosome, only the gene values are now Δ-values; small differences from the selected chromosome. This new sub-population of Δ-chromosomes is evolved as usual by taking each Δ-chromosome, adding each Δ-value to the originally selected chromosome and evaluating its fitness. If the fitness of the originally selected chromosome is exceeded the process is reiterated using the new fittest chromosome as the starting point.

2.6.4 Cooperative Synapse NeuroEvolution

Cooperative Synapse NeuroEvolution (CoSyNE) [86] is a directly encoded fixed topology NE method which evolves partial networks at a weight level. CoSyNE operates by assigning a sub-population to each weight of a user defined network; this user defines network may be feed-forward or recurrent. The overall population $\mathcal{P}$ is stored as a matrix with each column representing a sub-population p. See Equation 2.16 where there are n weights to evolve and a sub-population size of m. A complete ANN can then be formed by reading off any row from the $\mathcal{P}$ matrix which can be evaluated and assigned a fitness. Once a fitness is assigned to each row the weakest can be removed and replaced by crossing over and mutating the best rows in $\mathcal{P}$. Up until this point CoSyNE has been operating as CNE; albeit framed differently. In order to co-evolve the connection weights, each sup-population is permuted so that the weights now potentially form part of a different network the next generation. This causes each sub-population of weights to specialise to a suitable value which is compatible with the other weights within the wider network.

$$\mathcal{P} = \begin{pmatrix} W0_0 & W1_0 & \cdots & Wn_0 \\ W0_1 & W1_1 & \cdots & Wn_1 \\ \vdots & \vdots & \ddots & \vdots \\ W0_m & W1_m & \cdots & Wn_m \end{pmatrix} \tag{2.16}$$

2.6.4.1 Extensions

Compressed CoSyNE [164] is an indirect encoding version of CoSyNE. Instead of storing an explicit weight for each connection, a series of coefficients are stored which describe the weights across the completed network in the frequency domain; where more coefficients can be used to store higher frequencies more closely resembling the direct encoding form. As this method hopes to take advantage of potentially beneficial regularities in the distribution of connection weights, the topology of the artificial neural networks must be arrange such as to make this possible. For this reason only fully connected recurrent networks are used; with i inputs, one single hidden layer of n neurons and the outputs are taken as the outputs of a predetermined set of the hidden neurons.

2.6.5 Cellular Encoding

Cellular Encoding (CE) [92] is a developmental embryogenic[14] NE method which can evolve both the weights and topology of feed-forward ANNs.

Each CE genotype comprises a set of instructions which are followed in order to "grow" an ANN. CE usually begin with one hidden neuron connected to all the inputs and outputs; Figure 2.8 (a). This original network is then manipulated by the instructions which make up each genotype. Instructions include parallel cell division (Figure 2.8 (b)), sequential cell division (Figure 2.8 (c)) and many other operations which manipulate the topology of the network, connection weights and neuron biases. The genotype is then assigned a fitness indicating how well the constructed network performed on a given task.

As CE can produce large and small final ANNs there may be an issue with bloating. However, no discussion was found in the literature as to whether CE does or does not suffer from program bloat.

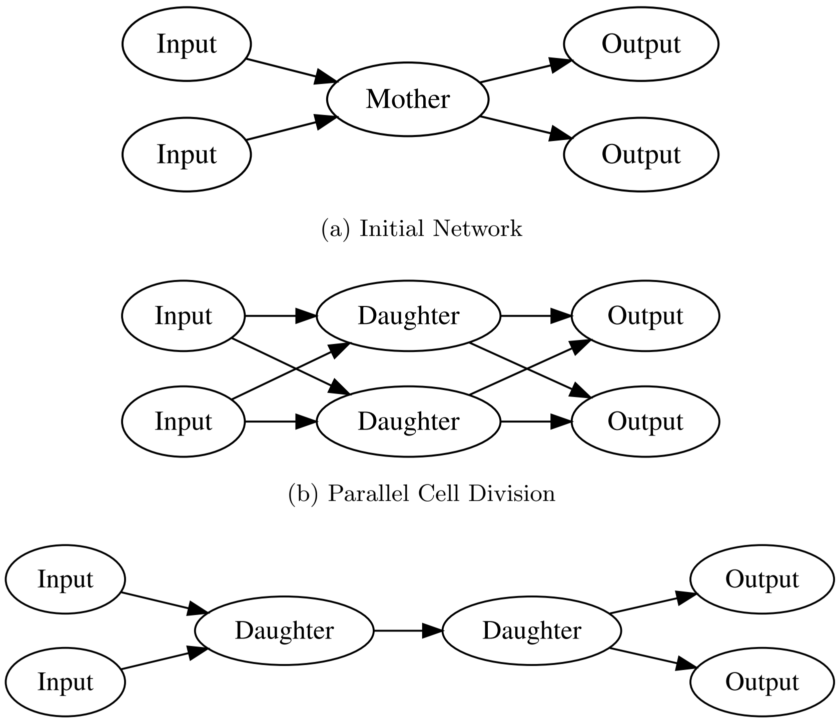
2.6.6 NeuroEvolution of Augmenting Topologies

NeuroEvolution of Augmenting Topologies (NEAT) [266] is a directly encoded NE strategy which can evolve the weights and topology of feed-forward and recurrent ANNs. Each genotype describes a complete network and takes the form of that shown in Figure 2.9.

Each network is described by a list of nodes and a list of connections. Each node is identified by an ID and whether it is an input, hidden or output node. Each connection gene contains an input and output node, a connection weight, an innovation number and whether the connection is enabled or disabled. Using these nodes and connections, complete ANN are described; Figure 2.9.

The mutation operations used by NEAT include add node, add connection (feed forward or recurrent) and mutate connection weights, input or output. Each connection weight is mutated each generation with a given probability. The chromosomes are initialised to contain no hidden nodes with outputs directly connected to inputs. The fact that NEAT starts with this minimal network size and only adds nodes/connections when required is thought to provide two advantages. Firstly, it is thought to aid the search by keeping the dimensionality of the search space as low as possible and only increasing it as

[14]Embryogenesis describes how an embryo develops into its fetus form. Cellular Encoding takes inspiration from embryogenesis as it describes a set of rules which can be used to "grow" an artificial neural network.



(a) Initial Network

(b) Parallel Cell Division

(c) Sequential Cell Division

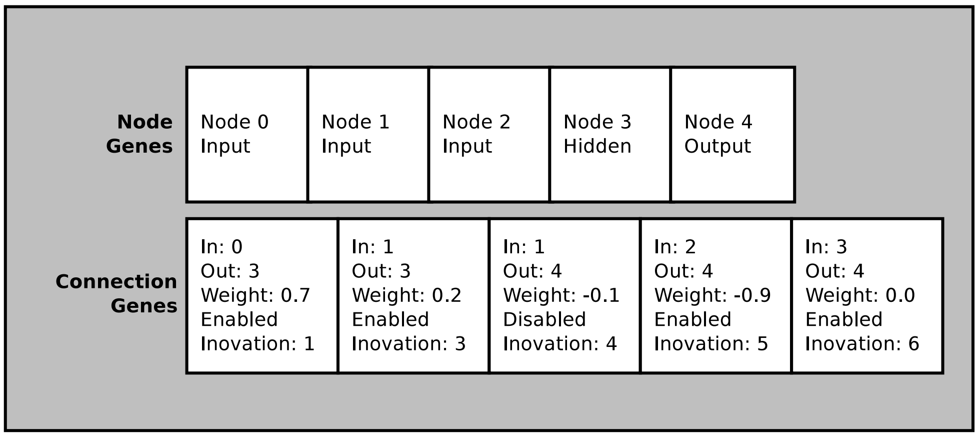
Figure 2.8: Example Cellular Encoding developmental operations. (a) gives an initial network. (b) gives an example of parallel cell division; where the mother cell is replaced (divided) into two identical cells. (b) gives an example of sequential cell division; where the mother cell is replaced (divided) into two cells with the first taking the mother cells inputs, the second taking the mother cells outputs.

required. Secondly, it is thought to aid generalisation as the training of large networks can easily over train on many problems; limiting the number of neurons helps combat this.

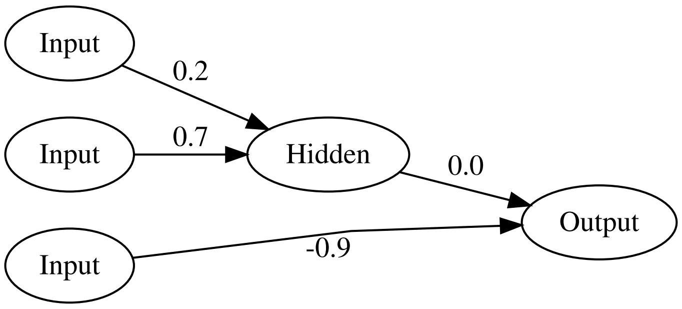
When a mutation adds a new connection it must always connect two previously unconnected nodes. When mutation creates a new node it is always placed on a current connection. For instance if neurons A and C were connected, the current connection would be removed and a new neuron B added such that A connections to B which connects to C.

It could be thought that this bias to small program size would result in NEAT not exhibiting program bloat. However it has been shown that whether NEAT bloats is strongly dictated by the choice of its parameters [274]; with typical parameter values used in early publications causing program bloat.

NEAT also records when specific mutations occur so as to make better use of the crossover operator and to subdivide the population into species. Whenever a new connection gene is introduced it is assigned a unique innovation number. These innovation num-


Node Genes
Node 0 Input
Node 1 Input
Node 2 Input
Node 3 Hidden
Node 4 Output
Connection Genes
In: 0 Out: 3 Weight: 0.7 Enabled Inovation: 1
In: 1 Out: 3 Weight: 0.2 Enabled Inovation: 3
In: 1 Out: 4 Weight: -0.1 Disabled Inovation: 4
In: 2 Out: 4 Weight: -0.9 Enabled Inovation: 5
In: 3 Out: 4 Weight: 0.0 Enabled Inovation: 6

(a) NEAT Genotype.



(b) Corresponding Phenotype

Figure 2.9: An example NEAT genotype and its decoded phenotype.

bers can then be used to identify similarities and differences between two genotypes; even if their phenotypes appear quite different. Knowing which genetic material two genotypes share helps tackle the *competing conventions problem*; where two genomes which describe the same phenotype, but in different ways, undergo crossover to produce children which no longer function like either parent. The competing conventions problem is thought to hinder many NE methods which use crossover. By using innovation numbers NEAT can identify, without analysis, which genetic material two genotypes share and which they do not. Crossover is then implemented by selecting randomly between shared connection genes, with non-shared genes inherited from the fitter parent.

NEAT also splits the population into species based on the amount of shared genetic material they contain. This allows solutions to primarily compete against others of the same species. The motivation for this is that new topology alterations, such as adding nodes, may initially lessen the genotypes fitness, but could lead to higher fitnesses once optimised. When the population is separated into species any new innovations can be given time to optimise before they are discarded.

2.6.6.1 Extensions

As well as borrowing advancements from the EA literature, NE can also utilise advancements from the ANN literature. An example of this is NEAT being used to evolve the reservoir of echo state networks [39]. In this case NEAT was used to create a recurrent reservoir which was then trained as an echo state network. The fitness after being trained as a reservoir was then used to guide the evolution of further reservoirs.

2.6.7 Hypercube-based NeuroEvolution of Augmenting Topologies

Hypercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT) [264] is an indirectly encoded extension to NEAT. HyperNEAT operates by using an ANN[15] constructed by regular NEAT to assign the weights of a second ANN. This is achieved by the first neural network having four inputs which index the position of two neurons in the second network via their Cartesian coordinates. The output of the first network, if above a curtain magnitude, is then used to determine the connection weight value of the connection between the two neurons in the second network; see Figure 2.10. If the output of the first network is below a given magnitude then no connection is created between the two nodes.

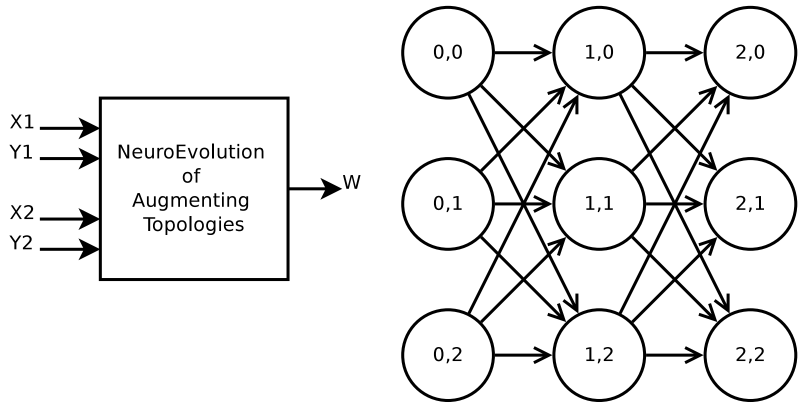


Figure 2.10: Example of HyperNEAT. The first network, constructed using NEAT, is used to assign the weights in the second based on the positions of the two connected nodes. Here only connections between layers are shown but typically connections between all possible nodes would be evaluated.

Each genotype in HyperNEAT is that of a regular NEAT genotype. To evaluate fitness, first the genotype is decoded into its ANN. This first network is then used to assign the weights of a second ANN. This second network is then applied to a given task

[15]Sometimes Compositional Pattern Producing Network (CPPN)s are also used [263].

and evaluated to produce a fitness value. This fitness value is then assigned to the original genotype.

An advantage of HyperNEAT is that it can describe arbitrarily large ANNs with a fixed size genotype. This means networks with large number of inputs, internal nodes and outputs do not impact on the learning algorithm. The nature of the indirect relationship between genotype and the final network also results in symmetry, imperfect symmetry and repetition with variations in the weight/connection distribution of the final network. Geometric patterns of similar forms are found in biological brains and so are thought to be advantageous.

A possible disadvantage of HyperNEAT is that it can construct networks in which there is no complete path between any input and the outputs; resulting in a network which is highly unlikely to be suitable for any task. This can occur if the first ANN, used to assign weights to the second, produces an output less that a given threshold. In this case no connection is made between the nodes under inspection. Therefore it is possible for there to be no complete path between inputs and outputs of the second ANN. Over evolutionary time these networks should be removed from the population, however, computational time is spent on networks which cannot possibly solve any meaningful task.

It is interesting that the concept of HyperNEAT is not limited to the use of NEAT. For instance it has been shown that tree-based GP can be used as a substitute for NEAT, where the evolved program tress were then used to assign the connection weights to ANNs [35]. In this regard almost all NE and GP methods could be used in a 'Hyper' form to create ANNs.

2.6.8 GeNeralized Acquisition of Recurrent Links

GeNeralized Acquisition of Recurrent Links (GNARL) [15] is one of the earliest forms of NE which evolves both the connection weights and topology of recurrent ANNs. GNARL is a directly encoded NE strategy where each chromosome represents a complete ANN.

GNARL operates by first initialising each chromosome to contain a random number of disconnected neurons; within user specified ranges. Then a number of random connections with random weights are added to the network; ensuring no connections leads to an input or from a output. An example of a possible GNARL network is shown in Figure 2.11; where it can be seen that constructing a network with a continuous path from inputs to outputs is not guaranteed.

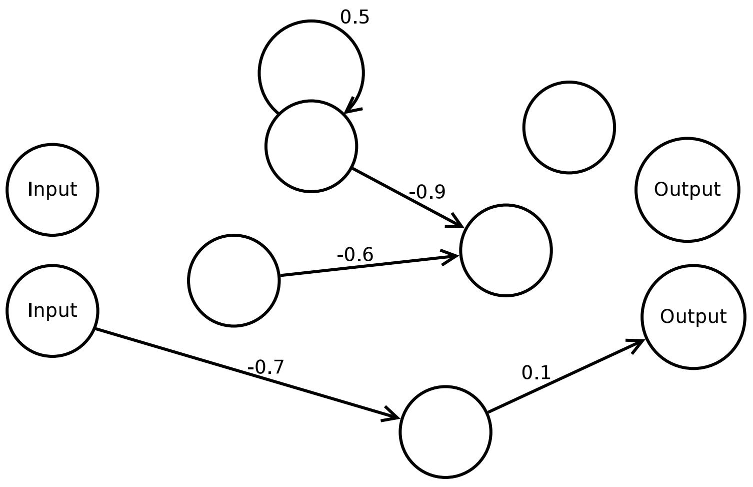


Figure 2.11: An example of a GNARL ANN.

Mutation operators can add/remove neurons, add/remove connections and mutate connection weights. A neuron is added by simply placing an unconnected neuron into the network. A neuron is removed by removing a random neuron from the network, along with all of the connections leading to and from that neuron. A connection is added by placing a connection between two (or the same) neurons and assigning it a connection weight of zero. A connection is removed by simply removing a connection. Connection weights are mutated based on a Gaussian distribution centred on the current weight with a user defined width. The number of topology and weight mutations which take place are a function of a user defined upper and lower bound and an annealing temperature which lessens as the final solution is reached; with the width of the Gaussian curves used when mutating connection weights also being a function of this temperature. No crossover is used.

As GNARL allows multiple topology mutations to occur in the creation of a child, it does not always add/remove a single connection/node. This means it is likely more explorative than NE which only add/remove *single* connections/nodes.

Although never explicitly discussed, it appears that GNARL can contain genetic redundancy in its genotypes. For instance, in Figure 2.11 it can be seen that a number of nodes and connections do not contribute to the semantics (behaviour) of the ANN. The genes associated with these nodes and connections are therefore genetically redundant.

2.6.9 Evolutionary Programming Artificial Neural Networks

Evolutionary Programming Artificial Networks (EPNet) [314] is a directly encoded NE method which can evolve the weights and topology of feed-forward ANNs. EPNet uses a combination of gradient decent and EP.

The encoding scheme used by EPNet is to store every possible weight value of a fully connected ANN in a matrix. A similar matrix is then used to store Boolean values representing whether each connection is active; decoded into the ANN. Finally, there is a vector which stores a Boolean value representing whether each node is active. This direct encoding scheme can describe every possible two dimensional feed-forward network topology; limited by a maximum width and depth. This method can also describe networks where there is no complete connection from inputs to outputs.

EPNet starts by initialising a population of random networks and then partly training them using a form of gradient decent and simulated annealing. Then rank-based selection is used followed by mutation. The mutation operation takes a novel form. First further partial training using gradient descent and simulated annealing is applied to the network. If partial training improves the fitness, then the changes are kept and this child replaces its parent; similar to Lamarckian evolution. If this partial training fails to improve on the fitness, then a randomly selected number of hidden nodes are removed and the network retrained using gradient descent and simulated annealing. This mutated and retrained child replaces the least fit member of the population only if it is fitter. If removing nodes fails to improve the fitness, then a random number of connections are removed and the network is partly retrained. This mutated and re-trained child replaces the least fit member of the population only if it is fitter. Finally if removing connections fails to improve on the fitness then a random number of connections and nodes are added to the network and the network partly retrained. If adding nodes and connections improves the fitness it always replaces the least fit member of the population. The mutation stage is then complete. No crossover is used. The process is then repeated until a suitable solution reached.

EPNet therefore allows multiple connections/nodes to be added or removed. This means it is more explorative than NE methods which do only add/remove single connections/nodes.

Although never explicitly discussed, it appears that EPNet contains genetic redundancy in its genotypes. For instance, if a connection is marked as inactive in the matrix of connections, the corresponding connection weight is still present in the connection weight matrix. Although this connection weight is not therefore decoded into the phenotype, it is still able to undergo mutation and become active at a later stage during evolution. Therefore EPNet genotypes can contain genetic redundancy which is also subject to neutral genetic drift.

The use of three matrices to describe network topology and connection weights is interesting as it allows for all possible two dimensional feed-forward topologies to be described; limited by the number of rows and columns. Interestingly this could easily be extended into three dimensions, or more, to allow for more complex network structures. Additionally, if the use of gradient descent was removed from the training process, the same method could also be used to describe RANNs. The method also represents an effective, simple mechanism of allowing for genetic redundancy

2.6.10 Cooperative Co-evolution model for Evolving Artificial Neural Networks

Cooperative Co-evolution Model for evolving Artificial Neural Networks (COVNET) [75] evolves partial ANNs with each genotype representing a sub network. Each of these sub networks is evolved in an isolated population with no migration. Although the arrangement of the sub networks within the wider network is fixed, the topology of each sub network is evolved along with the connection weights; therefore COVNET is, partially, a topology and weight evolving NE strategy. Or an alternative view would be a topology and weight evolving method with enforced modules. The encoding of each sub network is direct and can describe feed-forward and recurrent networks.

Each sub network is described by a number of nodes each with a number of connections and weights. Each COVNET sub population uses a $(\mu + \lambda)$-ES, and uses the mutation evolutionary operator; no crossover. The weights are updated using mutation and simulated annealing as a function of current fitness; the weight updates take the following form $w = w + \Delta w$ where Δw is a random value which is a function of the current fitness. Structural mutations take the form of: add node, remove node, add connection and remove connection with the number of these operations randomly chosen from user defined ranges. The actual structure of each sub-network, and the mutation operators applied, are similar to GNARL. Additionally, like GNARL, COVNET allows multiple connection/nodes to be added or removed. This means it is likely to be more explorative than NE methods which do only add/remove single connections/nodes.

As the evolved sub-networks of COVNET are very similar in form to GNARL genotypes it is also likely it contains genetic redundancy for the same reasoning as described previously for GNARL.

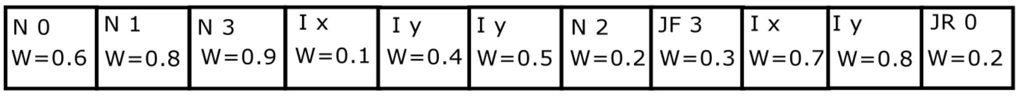
2.6.11 Evolutionary Acquisition of Neural Topologies

Evolutionary Acquisition of Neural Topologies (EANT) [143] is a directly encoded NE method which evolves the connection weights and topology of feed forward and recurrent ANNs.

EANT encodes full ANNs as linear genomes describing a weighted tree structure with additional weighted *jumper* connections between nodes; see Figure 2.12. These jumper connections can be feed-forward or recurrent enabling EANT to evolve RANNs. Topology mutation involves adding or removing jumper connections or adding new sub-networks to the tree; note that sub-network tree structures are never removed. This is done following the same reasoning as for NEAT. The search starts with a minimally complex network and then, if required, becomes more complex during the search. EANT also maintains several sub-populations based on topology similarity. A newly created topology is then given its own sub-population and only competes at a global scale after a given number of generations. This gives new topologies the time to be trained before being considered globally. Topology mutations only take place if mutations to connection weights are seen to no longer improve the solutions. Connection weight mutations were originally carried out using a simple self-adapting ES, but later used CMA-ES with each sub-population maintaining its own covariance matrix. This later version was called Evolutionary Acquisition of Neural Topologies 2 (EANT2) [255].

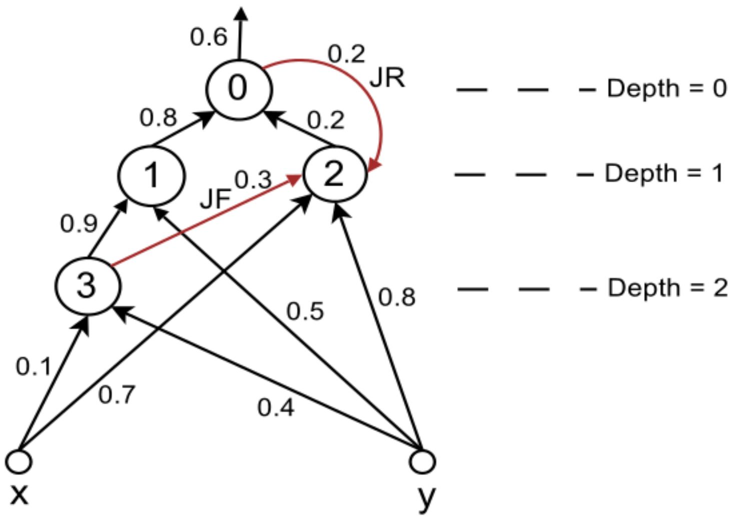
Like NEAT, EANTs maintains sub-populations of newly created topologies so as they can be evaluated before being accepted long term. Although, as has been shown for NEAT, this method, coupled with minimal initial solutions, is not sufficient to remove bloat without careful selection of parameters. Additionally, EANT allows the addition of new random sub trees (not individual nodes) as well as individual connections which is likely to amplify any presence of bloat. For these reasons EANT appears likely to suffer from bloat.

Finally, the addition of sub trees instead of individual nodes is likely to result in both exploitive and explorative mutations. This may represent an advantage over methods such as NEAT which conduct a much more exploitive topology search.



N 0	N 1	N 3	I x	I y	I y	N 2	JF 3	I x	I y	JR 0
W=0.6	W=0.8	W=0.9	W=0.1	W=0.4	W=0.5	W=0.2	W=0.3	W=0.7	W=0.8	W=0.2

(a) EANT Genotype.



(b) Corresponding Phenotype

Figure 2.12: EANT Genotype Phenotype mapping, taken from [143]. The Genotype (a) is decoded as a tree from root to leaves. N gives the node to connect to and I the input. JF and JR stand for feed forward and recurrent jumper connections respectively and are added to the tree by assuming they increase the arity of the current node. The corresponding phenotype is given in (b).

2.6.12 NeuroEvolutionary Algorithm

NeuroEvolutionary Algorithm (NevA) [276] is a directly encoded NE method which evolves the connection weights and topology of feed forward and recurrent ANNs. NevA Genotypes describe the connection weight value between every two nodes which are connected; see Figure 2.13. The genotypes are initialised with no hidden nodes, only outputs connected to inputs with connection weights in the range [-0.5,0.5]. Mutation can add or remove both connections and nodes and set connection weight to random values. Adding or removing connections is achieved by simply removing or adding a gene from the genotype. Adding a neuron is achieved by adding two new genes to the genotype, one which connects the new neurons input to a random neuron or input and one which connects the new neurons output to a random neuron or output. Removing a neuron involves removing all genes in the genotype which reference the removed neuron. Crossover is also utilised by selecting two parents with greater than average fitness. Child chromosomes are created which share the similar genes of the parents and randomly selected between the

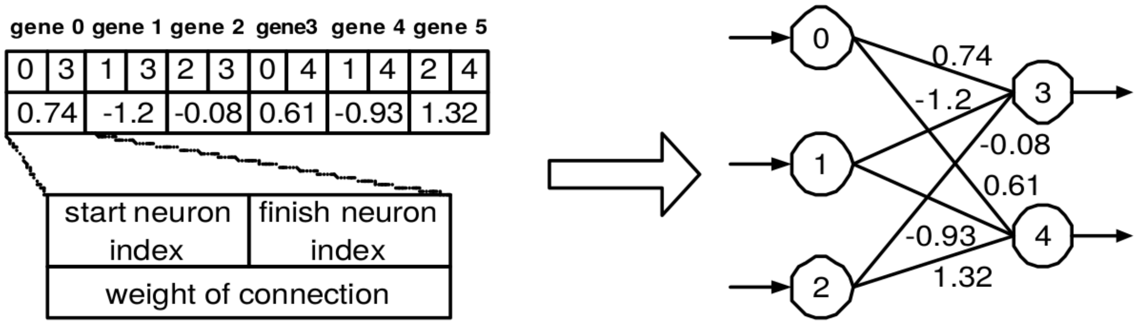


Figure 2.13: NevA Genotype Phenotype mapping, taken from [276]

differences. Elitism is also used to ensure solutions are not lost.

Similarly to NEAT, the solutions are initialised to be small and only increase in complexity if required. This is further enforced by biasing the probability of removing neurons and connections to be much more likely than adding them. This bias to smaller topologies may compensate for any bloat if present. Additionally the addition/removal of one neuron/connection per mutation operation is likely to be restrictive to a local search of topologies.

Note: it is unclear from the description presented in [276] whether it is possible for NevA to describe ANNs where there are no connections between inputs and outputs. It appears this should be possible if all of the connection genes are removed. However there may be unstated preventions for such an occurrence.

Table 2.1: Taxonomy of NeuroEvolutionary methods

Method	Encoding	Genotype Level	Evolves Topology	Incremental Topology Adaptations	Encodes Recurrent ANNs	Always Functioning ANNs	Utilises Crossover	Contains Genetic Redundancy	Exhibits Program Bloat
CNE	Direct	Network	No	-	Yes	-	Yes	No	-
SANE	Direct	Neuron	limited	-	No	-	Yes	No	-
ESP	Direct	Neuron	No	-	Yes	-	Yes	No	-
CoSyNE	Direct	Weight	No	-	Yes	-	Yes	No	-
Comp. CoSyNE	Indirect	Weight	No	-	Yes	-	No	No	-
CE	Developmental	Network	Yes	No	No	Yes	No	No	?
NEAT	Direct	Network	Yes	Yes	Yes	Yes	Yes	No	Typically
HyperNEAT	Indirect	Network	Yes	No	Yes	No	Yes	No	?
GNARL	Direct	Network	Yes	No	Yes	No	No	Yes	?
EPNet	Direct	Network	Yes	No	No	No	No	Yes	?
COVNET	Direct	Sub Network	Mostly	Yes	Yes	No	No	Yes	?
EANT/EANT2	Direct	Network	Yes	No	Yes	Yes	No	No	Likely
NevA	Direct	Network	Yes	Yes	Yes	-	Yes	No	Unlikely
CGPANN	Direct	Network	Yes	No	No	Yes	No	Yes	No
RCGPANN	Direct	Network	Yes	No	Yes	No	No	Yes	No

2.7 Review of the NeuroEvolutionary Literature

The previous section, Section 2.6, presented a review of many popular NE methods; this is summarised in Table 2.1. This section reviews the NE literature generally drawing from observations made in Section 2.6 as well as the wider literature.

2.7.1 Encoding

The vast majority of NE methods evolve directly encoded ANNs which are described by a single genotype. That is to say most genotypes do not use indirect or developmental methods and most genotypes do not describe individual neurons or individual connection weights. Which of the methods is most effective is not known due to a lack of good comparative empirical study. What is clear, however, is that none of those which evolve at a connection weight or neuron level fully adapt topology, and as topology is thought to be highly significant in the training of ANNs this could be considered a disadvantage. Therefore it appears that the perceived benefit of topology evolution has driven NE research towards methods which encode whole networks within each genotype; in order to facilitate topology evolution.

2.7.2 Topology Evolution

Of the NE methods which evolve topology, many use mutation operators which incrementally add/remove individual connections/nodes. However, as has been discussed, this may result in a local search of topology; possibly resulting in the search becoming trapped in topology local optima [15]. A possible advantage of using such a method is that the topology size is likely to grow much more slowly than the case where many connections / nodes can be added via a single mutation instance. As smaller networks are considered to generalise more effectively this may be an advantageous property. However, methods such as NEAT, which do add single connections/nodes, still exhibit program bloat unless the parameters are carefully crafted for its prevention [274]. Therefore, it is not clear if only allowing a small number of nodes to be added is an effective method for promoting small program sizes. This, coupled with the fact it may add restrictions to the search of topologies, leaves a question over whether incrementally adding/removing individual connections/nodes is an effective method for adapting topology.

With regard to evolving recurrent ANNs, all fixed topology methods are capable of utilising recurrence simply by enforcing a recurrent topology[16]. This is because the topology to be evolved can simply be chosen to be recurrent by the user. For instance, although the implementation of SANE does not allow for recurrent connections, this is a restriction which can be easily lifted; by allowing each node to connect its inputs to: program inputs, the node itself, or program outputs. With this addition SANE would be capable of evolving RANNs; but still with the limited topology evolution.

Of the NE methods which fully evolved topology, almost all were capable of evolving RANNs[17]. This is intuitive as if the connection placement is being evolved, why limit it to only feed-forward connections unless required. For instance, there will be tasks which do not require recurrence; in these cases it is reasonable to limit the search to feed-forward topologies. However, in general, restrictions should only be applied if they are known to be beneficial.

There are three consequences of unrestrained topology evolution which are not typically discussed in the literature. Firstly, whether the same two nodes are allowed to be connected by two or more connections. Secondly, whether a node is allowed to connect to itself via a recurrent connection. Thirdly, whether networks can be created without a continuous path from inputs to outputs. One reference found to these issues was in the description of NEAT: "*add connection mutation, a single new connection gene is added connecting two previously unconnected nodes*" [267]. Therefore NEAT does not allow multiple connections between the same two nodes, nor does it allow for self-connections. However, it seems reasonable that others may have also added such restrictions. The interesting aspect of this, is that any restriction to the search space should be justified, and no justification was found in the literature. For instance, it may be beneficial to allow for self-connections (as with GNARL), or even allow two nodes to be connected by multiple connections.

Of the NE methods which evolved topology, around half were capable of describing non-functioning ANNs i.e. those without complete paths connecting inputs to outputs. Such ANNs are unlikely to be of use and so the fact that they can occur is likely a disadvantage. This is an example of where the search space can be justifiably restrained. However in a completely unrestrained system, such topologies are possible. However it may be possible to computationally cheaply identify such topologies and not suffer the

[16]That is, unless they are combined with additional training methods which are not compatible with RANNs. Such as gradient based methods.

[17]All except CE which could also likely be extended to include the development of recurrent connections.

computational expense of evaluating such solutions using the fitness function.

It is often argued that the ability of NE methods to adapt network topology represents a significant advantage over many other ANNs training methods. Arguments for this include the fact that the user does not have to know a suitable topology in advance of training, relationships between topology and connection weights can be exploited and topologies which would be unlikely to be chosen/designed by a human can be considered.

However, little empirical investigation was found in the literature which assessed whether evolving topology actually offers an advantage over evolving only connection weights. The only example found investigated applying CMA-ES, a weight only evolving method, to a number of control benchmark tasks using a range of fixed topologies. Although the work was limited, in terms of assessing the importance of topology evolution, the results did show "*that the topology of the networks considerably influences the time to find a suitable control strategy*", [130]. It was also reported that "*results with fixed network topologies are significantly better than those reported for the best evolutionary method so far, which adapts both the weights and the structure of the networks*" [130]. However, this is to be expected, if the topology of a weight only evolving method is swept by hand and the best topology found compared to a topology and weight evolving method, it is likely to produce superior results. This is because the topology search has effectively already been completed in advance by the user and the time required to do so was not considered in the comparison. Therefore it is unfair to compare weight only and topology and weight evolving methods when the topology of the weight only method has been 'pre-optimised' by hand.

Therefore there are still many open questions surrounding the benefits of topology and weight evolving NE methods over just weight evolving methods. For instance, it may be the case that evolving topology results in final solutions equivalent to those found using "standard" topologies chosen following rules of thumb [47]; and then only evolving connection weights. Although it is assumed there is a complex relationship between topology and connections weights which can be exploited by evolution, this has never been shown. For instance it may be the case that provided the topology is sufficiently complex through the adaptation of connection weights alone solutions may be found easily; although factors such as over training may also be influential in determining suitable topologies. Finally, it is not currently known whether topology adaptation or weight adaptation is more significant to the training of ANNs. For instance, topology manipulation may provide a benefit, but it could be so minor that it is not worth the additional complexity it brings to NE methods.

2.7.3 Transfer Function Evolution

Although not widely utilised, all NE methods are capable of adapting the transfer function used within the ANNs. This can be achieved by the inclusion of additional genes describing the transfer function used by each node. Methods which already optimise the transfer functions used include General Neural Networks (GNN) [179], Parallel Distributed Genetic Programming (PDGP) [227] and Hierarchical Coevolutionary Genetic Algorithm (HCGA) [299]. Although in each case it was simply demonstrated that evolving heterogeneous ANNs was possible, not that doing so provided a benefit to the search.

Interestingly, the amount of research concerning the use of NE to create heterogeneous ANNs of evolved transfer functions is very limited considering repeated discussions that it should be researched further. "*Relatively little has been done on the evolution of node transfer functions, let alone the simultaneous evolution of both topological structure and node transfer functions*" [313], "*The current emphasis in neural network research is on learning algorithms and architectures, neglecting the importance of transfer functions*" [54] and "*Selection and/or optimisation of transfer functions performed by artificial neurons have been so far little explored ways to improve performance of neural networks in complex problems*" [55]. Therefore, a thorough investigation into the use of NE to create heterogeneous ANNs is a current omission from the literature.

2.7.4 Competing Conventions

As has been discussed, it is widely believed in the literature that NE suffers from competing conventions when employing the crossover operator. This may explain why many NE methods do not make use of crossover. Of those which do use crossover, only NEAT, and by extension HyperNEAT, has addressed the issue[18].

However, there is very little research which actually supports the claim that competing conventions poses an issue for NE. The often cited source for competing conventions [236, 301] provides only a theoretic discussion of its presence; no empirical evidence. Additionally, in a comparison of many NE recombination operators it was shown that

[18]Of the NE methods which have been investigated in this thesis.

"*simple crossover also fared well, suggesting that the permutation problem is not serious in practice*" [98].

Additionally, there is little evidence that the use of crossover poses a benefit for NEAT. In Kenneth Stanley's thesis [265] it was demonstrated that the use of crossover produced superior results for NEAT than without. However, the improvement brought about by the use of crossover was substantially less than any other aspect of NEATs algorithm investigated. Additionally, the only benchmark used for the investigation was the trivial XOR task and no statistical analysis was under taken. A thesis dedicated to the permutation problem (competing conventions) also failed to demonstrate any empirical benefit from using crossover [95]. Although in [95] the aim of the thesis was the study of the permutation problem, not whether it resulted in a worse search, the fact that this was not demonstrated is a questionable omission.

Therefore, although in the NE literature it is believed that competing conventions represents a major issue when making use of crossover unchecked, there is surprisingly little empirical evidence to support this.

2.7.5 Genetic Redundancy

None of the NE methods reviewed discussed the possibility of genetic redundancy being present or whether it aided the evolutionary search. As genetic redundancy, and neutral genetic drift, are important concepts in EA, and are thought to aid the search by facilitating the escape of local optima, this appears to be an omission in the literature. Interestingly, many NE methods appear to contain genetic material which can be active or inactive in determining the phenotypes semantics; see Table 2.1. Therefore many NE may actually be benefiting from neutral genetic drift without it currently being considered.

Whether the presence of genetic redundancy is beneficial is still an open question. However, if it were shown to be of benefit it may help in the selection of NE methods and guide future developments.

2.7.6 Program Bloat

A significant topic in the field of GP is program bloat [181, 256]. The only instance of bloat being referenced in relation to the reviewed NE methods was for NEAT. Although many NE methods do not adapt topology, therefore making bloat of no concern, of those which do, bloat could be occurring. In the case of NEAT, it was found that it does not exhibit bloat provided the parameters were correctly chosen [274], or conversely, that it does exhibit bloat if steps are not taken to prevent it. Additionally, methods such as EANT appear extremely likely to exhibit program bloat due it being closely modelled on tree-based GP; which is known to suffer significantly from bloat.

If it were found that many NE methods were suffering from program bloat, it could represent a disadvantage to the field in general. Interestingly however, there could also be parallels to non-evolutionary ANN training methods. For instance, constructive training methods which iteratively add neurons during training [169], are known to produce non-optimal network sizes [169, 170]; where the target is the smallest possible topology which solves the given task. The issue of constructive methods producing larger than necessary program structures could be likened to program bloat in the GP community. An alternative solution to the issue of choosing a suitable topology in the ANN literature is to use excessively large networks which are easier, if slower, to train. Then, if the issue of over training is encountered, which is often the case with large network sizes, pruning methods [238] are then employed to reduce the network size and improve generalisation. Again this has parallels with program bloat; even if the networks are not gaining in size during training. Interesting the wealth of pruning algorithms available in the ANN literature may also be applied, post training, to solutions found using NE. Therefore if bloating is found to be a concern, there are already a range of algorithms which can be used to lessen its detrimental effects. However, there are issues with program bloat besides final solution size, such as increased training times, which pruning does not address.

It may be the case that as soon as topology is allowed to be freely adapted program bloat is inevitable [175], unless prevention methods are taken, and so it is a “price to pay” for adapting topology. This may be true for both evolutionary and non-evolutionary training methods. Irrespective, program bloat is a topic which has been largely overlooked by the NE community and one which should be assessed and addressed if present.

2.7.7 Empirical Study

A serious issue within the field of NE is the lack of empirical comparative study. The field appears to be mainly application driven, which although important research in its own right, does not give insight into the underlying algorithms. For instance, if a given NE method appears to perform well on a new application which has never been investigated previously, it is not known how other NE would have performed; or other standard ML

Table 2.2: Differences in the single pole benchmark implementations used in the NeuroEvolutionary literature.

Method	Control System	Bias Input	Pole Starting position
NEAT [88]	Continuous	No	$4°$
CNE [87]	Continuous	No	$4°$
NEvA [89]	Bang-Bang	No	$0°$
SANE [211]	Bang-Bang (two outputs)	Yes	$random°$
ESP [89]	Bang-Bang	No	$0°$
CoSyNE [87]	Continuous	No	$4°$
CGPANN [154]	Bang-Bang	No	not stated

methods generally. Therefore, in order to be confident that a particular algorithm is worth considering, it must be compared to other methods; both NE methods and more general ML methods.

Unfortunately in the NE literature there is only one benchmark which is widely used for comparison; single/double pole balancing[19]; see Appendix A for a description of the pole balancing benchmark. This makes rigorous comparisons of NE methods challenging if not impossible. It is desirable for the NE community to create a suite of benchmarks for comparing new and existing methods; like the steps being taken in the GP community [197]. Additionally, statistical significance testing is rarely used nor is any comparison of the spread of results other than using standard deviations[20].

The situation is worsened by the fact that one of the two pole balancing tasks, single pole, is far too simplistic and completely unstandardised. For instance, Kenneth Stanley, inventor of NEAT, dismissed the single pole balancing as being too simplistic for modern methods [265]. He also stated, though email correspondence, that the task is so simple that the distribution of the random numbers used during the search could affect the results as much as the learning algorithm used. Additionally, Table 2.2 shows the single pole benchmark set-up used by a range of popular NE methods. As can be seen, the benchmark is not standardised meaning the comparisons are of little use.

[19]The breast cancer classification benchmark is also occasionally used within the NE literature. However the version of the benchmark used is inconsistent.

[20]Which as the spread of results are often non-normal, or at least not shown to be normal, is of limited use.

2.8 Discussion

This chapter has introduced NE along with a brief discussion of the underlying technologies which are combined; namely EAs and ANNs. This introduction is given along with the advantageous properties of NE and related concepts.

This chapter has also provided a taxonomy of the more popular NE methods comparing and contrasting different aspects of each method. This taxonomy is also combined with a wider literature survey to provide a high level assessment of the NE literature. Form this literature review a number of insights were made; these are now summarised.

The ability to adapt ANN topology is considered a major advantage of many NE methods over more traditional training methods such as gradient decent. These benefits are thought to include: removing the requirement for users to know a suitable topology in advance of training, exploiting relationships between connection weights and topology and considering topologies which would otherwise not be considered by a human designer. However, as has been discussed, there has been little research attesting to these perceived benefits. Much of the reasoning for why it is beneficial is based on intuition or more theoretical possibilities, rather than empirical evidence. Therefore, more research is needed to assess the relative benefits of using NE to adapt network topology.

Despite many calls for more research, there have been relatively few NE methods which utilise the ability to adapt and create heterogeneous ANNs. To what extent the ability to evolve heterogeneous ANNs represents an advantage is unknown. Therefore, as it is a feature implementable in nearly all NE methods, more research is needed to assess its usefulness.

It is widely cited in the NE literature that competing conventions means that the use of crossover is detrimental unless specific measures are taken for its prevention. This has led to many NE not employing crossover. However, the most cited reasoning that competing conventions represents a issue are is a theoretical paper [236], with empirical investigations [98] giving evidence to the contrary. Therefore, it has not been convincingly demonstrated that competing conventions does represent an issue for NE when employing crossover.

Genetic redundancy is a major topic in the field of EAs and one which has been overlooked in the field of NE. As many NE methods appear to contain genetic redundancy, it is likely the case that they are benefiting from its presence. However, research is required to demonstrate that this is the case.

Program bloat is a major drawback of many GP methods. As many NE methods share features with GP, it is likely that some/many of them also suffer from program bloat. However, there has been relatively little study of program bloat in the NE literature. The only reference found to bloat in the NE literature was for NEAT, in which it was demonstrated that NEAT did indeed bloat unless careful parameter choice were made. Therefore more research is needed into the presence of bloat in NE methods.

Finally, it has been discussed that the lack of good empirical comparison, and the lack of standardised benchmarks, is a major issue in the field of NE. This is an issue which needs to be addressed by the community as currently it is very challenging, if not impossible, to compare the numerous NE methods. Issues like these can be highly detrimental to a research field and need to be addressed if NE is to be taken seriously in the wider ML community.

Chapter 3

Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) [202,208] is a form of Genetic Programming (GP) [165,228] created by Julian F Miller and Peter Thomson in 2000. This Chapter introduces CGP along with its extensions and related theory which will be referenced throughout the thesis.

3.1 Structure of this Chapter

This chapter describes the general operation of CGP by first describing the encoding and decoding scheme used in Sections 3.2 and 3.3 respectively. Then the evolutionary strategy employed along with typical parameter choices are discussed in Sections 3.4 and 3.5 respectively.

Once CGP has been introduced, a number of advantageous properties over other GP are discussed in Section 3.6. This is followed by a number of common applications of CGP in Section 3.7.

The Chapter then introduces more advanced CGP topics comprising a number of extensions which have been made to the basic algorithm in Section 3.8 and a discussion of the theoretical research related to CGP in Section 3.9.

Finally a closing summary is given in Section 3.10.

3.2 Encoding

Each CGP genotype describes a directed acyclic graph of computational nodes; see Figure 3.1. The genotypes comprise function genes (F_i), connection genes ($C_{i,j}$) and output

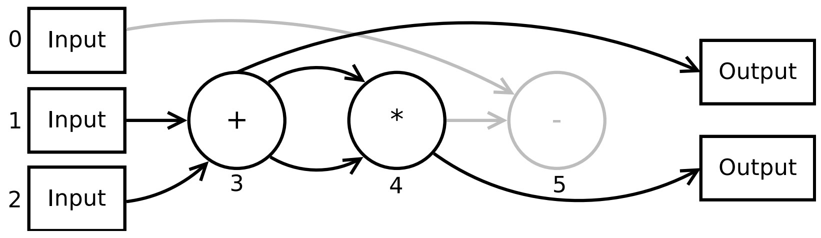


Figure 3.1: Example CGP program with three inputs, three available nodes and two outputs. The active genes are shown in bold with the inactive in grey. The corresponding chromosome is as follows: $\underline{0}12$ $\underline{2}33$ $\underline{1}04$ 3 4

genes (O_i); where i indexes each node in the graph and j indexes each node's inputs. The function genes represent indexes in a function look-up-table and describe the functionality (transfer function) of each node. The connection genes determine where each node acquires its inputs. Connection genes may connect a given node to any previous node in the program, or any of the program inputs; thus obeying the acyclic constraint. The output genes address any program input or internal node and define which nodes are used as the program outputs.

Originally, CGP programs were organised with nodes arranged in rows and columns, with each node indexed by its *Cartesian* coordinates; see Figure 3.2. However, this is an unnecessary constraint[1] as any configuration possible using a given number of rows and columns is also possible using one row with many columns; provided the total number of nodes remains constant. This is because CGP can evolve where each node obtains its inputs. Consequently, it is now common to use CGP with one row and n columns; with each node only indexed by its column as shown in Figure 3.1. A generic (one row) CGP chromosome is given in Equation 3.1; where α is the arity of each node, n is the number of nodes and m is the number of program outputs.

$$F_0C_{0,0}...C_{0,\alpha-1}...F_{n-1}C_{n-1,0}...C_{n-1,\alpha-1}\text{: } O_0...O_{m-1} \tag{3.1}$$

An example CGP program is given in Figure 3.1 along with its corresponding chromosome; for clarity, and following previous conventions, the function genes are underlined. As can be seen, all nodes are connected to previous nodes or program inputs. Not all program inputs have to be used, enabling evolution to decide which inputs are significant.

[1]However adding topology constraints to CGP can often be desirable. For instance when creating Boolean logic circuits the depth of the circuit is related to the time it takes to execute. Therefore by limiting the depth of the circuits created, by allowing many gates in each 'layer', the search can be constrained to produce faster circuits.

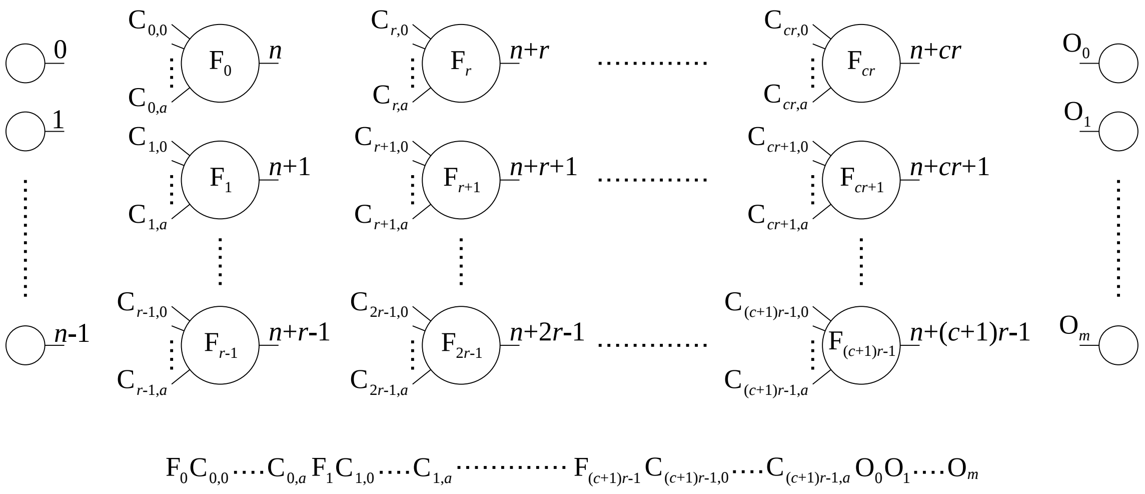

0
1
n-1
F0
n
F1
n+1
Fr-1
n+r-1
Fr
n+r
Fr+1
n+r+1
F2r-1
n+2r-1
Fcr
n+cr
Fcr+1
n+cr+1
F(c+1)r-1
n+(c+1)r-1
O0
O1
Om
F0C0,0....C0,a F1C1,0....C1,a .... F(c+1)r-1 C(c+1)r-1,0....C(c+1)r-1,a O0O1....Om

Figure 3.2: Generalised depiction of the original rows and columns form of a CGP chromosome; shown graphically above and as a string below. Image taken from [202]

An advantage of CGP over tree-based GP, again seen in Figure 3.1, is that node outputs can be reused multiple times, rather than requiring the same value to be recalculated if it is needed more than once. A further advantage is that CGP can easily be applied to Multiple-Input Multiple-Output (MIMO) problems. Finally, not all nodes contribute to the final program output; these represent the inactive nodes (described by explicitly redundant genes) which enable variable length phenotypes and neutral genetic drift.

3.3 Decoding and Executing

CGP is often thought of as an indirect encoding scheme as there is a process of removing the inactive genes from the genotype; the genes associated with the inactive nodes as seen in Figure 3.1. However, as the removal of the inactive nodes is not strictly necessary for CGP to operate[2], it is strictly a direct encoding scheme.

CGP phenotypes are decoded from their genotypes by first determining the active nodes. This is often achieved recursively. First add the nodes indexed by the output genes to an active node list. Then, for every node added to the active node list, add all nodes to which they also connect. If an input node is reached, or a node is encountered which has already been added to the active node list, do not add anything to the active node list. The active node list then contains all of the active nodes. The list is then often sorted from low node indexes (closest to the inputs) to high node indexes (closest to the outputs).

[2]Removing the inactive nodes serves to speed up the execution time of the phenotypes. Their removal does not alter the phenotypes semantics.

As previously mentioned, although determining which nodes are active is not strictly necessary for CGP to operate, calculating the outputs of nodes which are never used is a waste of computation time. Additionally, as the number of inactive nodes is typically a large proportion of the chromosome [82, 84, 207], not calculating their output results in a large computational saving.

CGP phenotypes are then executed by first applying a set of inputs to the input nodes. Each node present in the active node list is then updated in index order (i). Whenever a node calculates an output value it is made available to all other nodes which connect to that node. Once all of the nodes in the active node list have calculated their output value, the node(s) indexed by the output gene(s) are then read and their value(s) used as the output(s) of the phenotype for the give set of inputs.

Although anecdotal, it appears that many consider the computational expense of decoding CGP chromosomes to represent a disadvantage. It may be the case that some believe that the decoding stage is necessary each time a chromosome is executed. This is not the case, CGP chromosomes have their active nodes determined once and then only these nodes are ever evaluated regardless of the number of times the phenotype is executed. It may also be the case that some believe, that even though the chromosomes only have to be decoded once, that this is still a disadvantage over directly encoded methods which do not require this step. However, the computational time required to determine the active nodes in a CGP phenotype is drastically smaller that the time to run any meaningful fitness evaluation, and is therefore not significant in the total running time of the algorithm.

3.4 Evolutionary Strategy

CGP typically uses an elitist $(1+\lambda)$-ES with $\lambda = 4$ [202] coupled with point or probabilistic mutation. It is typically thought that using such a greedy algorithm would result in a population with poor genetic diversity, which would often, and easily, became trapped in local optima. However, the reason such a simple algorithm is thought to be so effective for CGP is twofold. Firstly, CGP does not typically utilise crossover, see Section 3.8.1, which reduces the requirement to maintain a genetically diverse population. Secondly, the reason this does not lead to CGP easily becoming trapped in local optima is due to the inactive genes creating many plateaus in the search space. These plateaus can then be navigated across when the search becomes trapped in local optima via neutral genetic drift [202,287,318]. This provides a powerful mechanism for both increasing genetic diversity in the population, and escaping local optima; see Section 3.9.1.

The initial population of (1+λ) chromosomes is typically generated randomly by setting each gene in each chromosome to a random valid allele. Other methods are also sometimes employed such as seeding the populations with previously known partial solution.

CGP often makes use of point mutation [208]. Point mutation is where the number of gene to be mutated is chosen in advance and then this many randomly chosen genes are selected and changed to a random allele each mutation instance. The number of genes to be mutated is often expressed as a percentage of the total number of genes. An alternative mutation method is probabilistic mutation; where each gene in the genotype is changed to a random allele with a given probability.

Discussions with Dr Miller, a co-inventor of CGP, uncovered that point mutation was favoured due to it representing a faster method for selecting the genes to be mutated. For point mutation, n random genes are chosen. For probabilistic mutation, each of the genes must be considered in turn and a randomly generated value compared to a mutation probability in each case. However, as the main computation expense of Evolutionary Algorithm (EA)s is the fitness evaluation, such computational saving in the mutation method is likely to be insignificant. Additionally, the smallest increment in mutation rate which can be used by point mutation is the reciprocal of the total number of genes in the chromosome. However, probabilistic mutation can use any mutation rate. Probabilistic mutation also causes a varying number of mutations to take place each time it is applied. For instance, it would be possible, albeit unlikely, for all of the genes to be mutated, or none. Conceptually this could provide an advantage. Take a situation where the population is trapped in a local optimum which requires five of the one hundred genes to change value in order to escape. Unfortunately the point mutation rate was set as 4%, resulting in four gene alterations each instance, and so the local optima can *never* be escaped. Now, if probabilistic mutation were used, although on average a 4% mutation rate would result in four genes being mutated, it is also reasonably likely that three or five genes would be mutated. This means that it is at least now *possible* to escape the local optima. Whether this happens in practice is unknown, but considering the small additional computation expense, the fact it could happen may be enough to favour it as a mutation method. A final, lesser advantage is that probabilistic mutation rates means it is trivial to set separate mutation rates for the separate type of genes; if this were at any point desired. For these reasons, probabilistic mutation is the mutation method used throughout this thesis.

3.5 Parameters

The CGP search is controlled using a number of parameters. These parameters control the evolutionary process and dictate the maximum size of the evolved solutions along with other topology constraints. Common CGP parameters are given in Table 3.1. Typical values for these parameters given where relevant. The remaining parameters are dictated by the problem under investigation i.e. the node arity (α) depends on the functions in the look-up-table and the number of nodes (n) on the complexity of the problem; although overestimating the number of nodes has been shown previously to be advantageous [207].

Additional parameters occasionally used by CGP [202] include: levels back and short-cut connections. The levels back parameters limits the maximum 'distance' allowed between two connected nodes. For instance, if the levels back parameter was set as one, then a given node could only connect to a node in the previous column; when using the rows and columns form of CGP. This can be used to limit the range of topologies searched. The short-cut connections parameter is a Boolean flag which dictates whether output genes can directly index input nodes; if short-cut connections are allowed then outputs may map directly to inputs. This parameter is also used to restrict the range of topologies searched.

Table 3.1: Standard parameters used by CGP.

Parameter	Description	Values	Typical Value
Generations	The maximum number of generations allowed before terminating the search.	$\mathbb{Z}_{>0}$	-
Mu (μ)	The number of elite parents promoted from each generation unchanged and used to generate the children.	$\mathbb{Z}_{>0}$	1
Lambda (λ)	The number of children generated from the μ parents.	$\mathbb{Z}_{>0}$	4
Mutation Rate	The proportion of the parent which is mutated when creating a child.	$\mathbb{R}_{[0,1]}$	~ 0.03
Nodes (n)	The maximum number of nodes.	$\mathbb{Z}_{>0}$	-
Arity (α)	The arity of each node.	$\mathbb{Z}_{>0}$	-
Function Set	The functions made available to the nodes.	-	-

3.6 Advantages of Cartesian Genetic Programming

Although CGP has not been adopted to the same extent as the more popular tree-based GP [300], it has a number of advantageous properties over tree-based GP which often make it a suitable alternative:

- CGP does not suffer from program bloat.
- CGP greatly benefits neutral genetic drift.
- CGP is naturally suited to MIMO tasks.
- CGP allows internally calculated values to be reused.

Each of these advantages is now discussed.

3.6.1 No Bloat

Bloat can be defined as “program growth without (significant) return in terms of fitness” [228], that is, if program size is increasing disproportionately to fitness improvements, then bloat is said to be occurring. Bloat is a serious issue for tree-based GP [256] which often results in excessively large program sizes unless actively prevented through fitness penalisation or imposed upper limits on program size. Conversely, CGP has a natural resilience to bloat [201] and so the whole topic of bloat is not an issue when using CGP.

Although anecdotal, the author has often encountered the perception that CGP does not (or indeed cannot) suffer from bloat due to it using a fixed sized genotype. This is incorrect. As CGP nodes can be active or inactive in the phenotype, the size of the phenotype can rise and fall. Although this is not typically considered for tree-based GP, as the genotype and phenotype are effectively equivalent, it is the bloat of the phenotype which is of concern. Therefore, it would be possible for CGP to bloat, which would manifest itself as an increase in the number of active nodes; or equivalently the size of the phenotype.

3.6.2 Heightened Neutral Genetic Drift

The encoding used by CGP results in a high proportion of the genes typically being inactive [207]. This enables a large amount of neutral generic drift to take place. As neutral generic drift is thought to be greatly beneficial to the evolutionary search [287,318], allowing easier escape from local optima, this is thought to be major advantage of CGP.

3.6.3 Multiple-Input Multiple Output

Typically tree-based GP is only capable of creating programs with multiple inputs and a single output[3]. Conversely CGP is directly compatible with MIMO problems.

Additionally as CGP evolves directed graphs it is not limited by the constraints of a tree structure whilst still being able to evolve tree structures given an evolutionary pressure to do so.

3.6.4 Reuse of Internally Calculated Values

When using tree-based GP, a given node's output can never be read by more than one other node. This is limiting if the same functionality is needed multiple times. For instance, take a trigonometric task where no trigonometric functions are provided, nor the value of π. Discovering an approximation to π is likely to be beneficial to the evolutionary search and used throughout the evolved program. In tree-based GP, if a value of π were to be used multiple times, then it must be *rediscovered* multiple times; as an internally calculate value can only be used once.

Conversely, CGP allows the outputs of any node to be used by any other node in the program; provided it obeys the rules of acyclic connectivity. This means that if π were discovered the *same* value could be used multiple times.

[3]Although there are forms of tree-based GP which are compatible with multiple outputs [228, 324], these are typically not used. Additionally these forms of GP break the tree based representation removing many of it's beneficial properties including simple crossover operations.

3.7 Applications

Although CGP was originally developed for the implementation and optimisation of digital circuits [204, 205], it has subsequently been applied to many other domains. This section describes a brief sample of the applications to which CGP has been applied.

When applying CGP to digital circuit implementation [204,205], the function look-up table is populated with Boolean logic functions, {AND,OR,NAND} etc, and the arity of each node is set to two for two input logic gates or to three for three input logic gates etc. Using this configuration CGP has been used to evolve implementations of given truth tables. Further to this, the size of the evolved digital circuit has been incorporated into the fitness function [139]; where smaller circuits were favoured once the truth table is successfully implemented. This allowed CGP to evolved smaller circuits which typically use less power, compute faster and are cheaper to manufacture. Recently this application of CGP to Boolean circuit synthesis was extended to make use of SAT solvers to speed up fitness evaluations [286]. This work was shown capable of generating circuits with hundreds of inputs and thousands of gates [286][4].

CGP has also been applied to image processing [101] where it has been shown to be a very powerful technique. Here the function set was comprised of a large number of previously known image processing functions; incorporating domain knowledge into the search. The technique was termed CGP-Image Processing and has been successfully applied to image filtering [250], edge detection [202], medical imaging and real time object detection with varying lighting, scale and rotation.

There have also been many medical applications including assessment of Alzheimer's disease [105] and classification of mammograms as containing benign or malignant tumours [115, 289, 294].

Additionally CGP has also found application in function optimisation [206], Artificial life [242], Bent function synthesis[5] [119] and even visual art [18].

[4]Zdenek Vasicek's and Lukas Sekanina's work on using CGP to synthesise digital circuits received the GECCO 2015 "Humies" Gold award.

[5]Which received the GECCO 2014 "Humies" Bronze award.

3.8 Extensions

Since its first development in 2000 CGP has undergone a number of extensions. This section describes a range of notable extensions which have been applied to CGP.

An additional significant extension to CGP is its application to training Artificial Neural Network (ANN)s; the wider topic of this thesis. However, a discussion of this extension is left until Chapter 4.

3.8.1 Floating Point Encoding

CGP has been previously extended such than all of the genes were encoded as floating point, rather than integer, numbers in order to facilitate crossover [42]. Here the connection, function and output genes were represented using floating point numbers in the range [0,1]. These floating point numbers were then mapped to their corresponding typical integer values using Equation 3.2 for the function gene values and Equation 3.3 for the genes which correspond to node indexes (connection genes and outputs); where $gene_i$ is the floating point value of gene i, $func_{total}$ is the total number of functions in the function look-up table (LUT), $node_{total}$ is the total number of nodes (inputs and functioning nodes) and $floor\left(x\right)$ returns the largest integer which is $\leq x$.

$$floor\left(gene_i * func_{total}\right) \tag{3.2}$$

$$floor(gene_i * node_{total}) \tag{3.3}$$

Using this floating point representation crossover was implemented by averaging the gene values of two parent chromosomes, or selecting a random value within their ranges. This is only possible if the number of rows and columns used for each chromosome is equal.

The work presented in [42], and subsequent work in [198], demonstrated that using a floating point representation to facilitate crossover showed an advantage on a range of symbolic regression tasks. This was also confirmed by the author via a unpublished masters dissertation [278]. However, the authors masters dissertation [278] also demonstrated that crossover implemented via a floating point representation showed no advantage generally when a wider number of task domains were studied. No explanation for the discontinuity was found.

3.8.2 Self Modifying CGP

Self Modifying Cartesian Genetic Programming (SMCGP) [103, 104] is a developmental extension to CGP. Each SMCGP chromosome begins by describing a valid CGP phenotype which includes nodes which adapt the phenotype configuration upon execution. As an example, one SMCGP node type, upon execution, copies a section of the phenotype and inserts it elsewhere in the phenotype.

SMCGP has been applied to scalable tasks such as generating the parity bit for arbitrary sized bit strings. On the first iteration the phenotype would generate the parity bit for a one bit string and on the second iteration a two bit string and so on. SMCGP has also been applied to predicting the digits of π and e to very high precision [102].

3.8.3 Multi-Chromosome CGP

A multi-chromosome version of CGP [293] was developed where each member of the population is described by multiple chromosomes. Here an individual chromosome is responsible for each output of a given task. For instance, if multi-chromosome CGP were applied to implementing a two output circuit, there would be two chromosomes, one for each output. Each of these chromosomes represents a standard CGP program with no interconnections between them.

Although multi-chromosome CGP can be used to lower the dimensionality of the search, it also removes code reuse. CGP enables code reuse by allowing sections of the program to contribute to multiple outputs. For instance, if two outputs were both reliant on two inputs being active and then on separate secondary criteria, multi-chromosome CGP would have to separately evolve logic to check if these two inputs were high in both chromosomes. Whereas in regular CGP this would only have to be evolved once and then the result reused. Therefore lowering the dimensionality of the search by using multiple chromosomes comes at the cost of limiting code reuse. Which of these two behaviours is of greater benefit, is likely task dependent.

3.8.4 Modular CGP

Modular Cartesian Genetic Programming (MCGP), also termed Embedded Cartesian Genetic Programming (ECGP), [292] is an extension to CGP which allows Automatically Defined Functions (ADF) [166] through module acquisition. Here sections of the CGP chromosomes are selected and added to the function look-up-table to be used as a function by other nodes. These captured sections of CGP chromosome are termed modules. The power of module acquisition is that it allows the reuse of potentially useful pieces of code as functions. For instance, if MCGP were applied to evolving digital circuits without the XOR logic gate being available, MCGP could, potentially, create an XOR gate and use it throughout the network as a function. The captured modules are also subject to mutations such as those used by regular CGP; with additional mutation operations more specific to MCGP.

3.8.5 Balanced CGP

Balanced Cartesian Genetic Programming (BCGP) [316] is the application of Biogeography-Based Optimisation (BBO) [258] mechanisms to CGP. This is undertaken by allowing migration of genetic material between solutions; similar to crossover in the EA literature. The work also employed a mutation operator which considers the genetic material of the entire population; not just the individual being mutated. The motivation is to create a more balanced distribution of exploration and exploitation for CGP.

Although the results presented in [316] appear very promising, the only class of problem investigated was symbolic regression. Interestingly, this is also the only type of task which regular crossover applied to CGP has been shown to be beneficial [42, 198]. Therefore important future work for BCGP is to identify if the previously observed benefits also extend to tasks other than symbolic regression.

3.9 Related Theory

As well as empirical investigation there has also been substantial theoretical research surrounding CGP. This research has not only resulted in a better understanding of the operations taking place but has also influenced subsequent development. This section describes some of the main theoretical work surrounding CGP.

3.9.1 Neutrality and Genetic Drift

The fact that CGP naturally contains inactive genes enables it to make use of neutral genetic drift [202, 287, 318, 320]. Neutral genetic drift is where genetic material in the chromosome has no effect on the phenotype's semantics but is subject to mutation which is passed on to subsequent generations. This process can be useful for the evolutionary search. Take, for example, a population which is stable in a local optimum. Almost all change in the active genes cause a decrease in fitness and so are penalised and not passed on to the next generation. Any mutation in the inactive genes, however, has no effect on the phenotype and are passes on from generation to generation. This process results in a population with very similar active genetic material but possibly very diverse inactive genetic material. Now, in this situation, any mutation which activates previously inactive genetic material can have a very different effect depending upon the individual mutated; as the inactive genetic differs between individuals. This means that a very wide range of possibilities are available to the next generation. If there were no inactive genes, or all of the inactive genes were very similar, the number of possibilities would be vastly reduced. The ability of a population to accesses a much wider range of solution in one generation is advantageous in two regards. Firstly, if there are environmental changes, the next generation is sampling a much wider set of new solutions and so is more likely to find new optima quickly. Secondly, the population is much more likely to escape local optima again because it is sampling a much wider area of the solution space; whilst remaining "trapped" in the local optima.

3.9.2 Length Bias

It has been shown by Brian Goldman [82, 84] that CGP has a bias to networks of a certain size; typically a low percentage of the available nodes. This is because every node connects to previous nodes in the graph. This means that the number of nodes which can connect to the nodes 'closest' to the inputs, low node index (i), is far higher than the number of nodes which can connect to nodes 'closest' to the outputs, high node index (i). This means that the probability of a give node being active is directly proportional to its node index; position in the genotype. This results in the distribution of randomly created CGP chromosomes, or mutations to chromosomes, not being even across the possible number of active nodes. This finding is also in keeping with previous research which noted that CGP typically uses a very low percentage of active nodes [207]. The exact number of active nodes to which there is a bias is a function of the number of inputs, number of available nodes, the arity of each node and the number of outputs.

Although methods which lessened length bias, by rearranging the active nodes during the search [82, 84], were presented, it may be the case that length bias actually offers an advantage. For instance, biases in a search are usually considered a disadvantage when the applied task is a black box; as one does not know whether a bias to a given topology will be appropriate. However, for classification tasks, smaller solutions are often favoured over larger as they typically perform better on unseen data; mirroring the concept of Occams razor [30]. Additionally, smaller solutions are often favoured generally because (a) they are quicker to execute and (b) they are easier to understand and reason about. Finally, a bias towards certain topologies does not limit the topologies which can be found given sufficient evolutionary pressure. In this regard if a task requires a number of nodes larger or smaller than the number to which there is a bias, this is still possible. Therefore, although results were presented which showed removing length bias produced better results on problems specifically designed to require a very large percentage of the possible nodes to be active [82, 84], on many real world applications, length bias may actually be of benefit.

3.9.3 Resilience to Bloat

As has been previously discussed, in Section 3.6, one of the advantages of CGP over other GP methods is that it does not exhibit program bloat [201]. Although the cause of bloat in GP is still debated [261], there are three main hypotheses: the protective hypothesis, the drift hypothesis and the removal bias hypothesis.

The drift hypothesis is as follows. When a population is trapped in a local optimum, many of the parent's children will have the same or very similar fitness. A method often used by EAs is to replace parents with their children if their fitnesss are equal or very similar; with the aim to improve genetic diversity and escape the local optima with future mutations. If adding or removing a small number of nodes does not lessen the fitness, then the children can be larger or smaller. Additionally, it has been shown that for a given chromosome size, there exists more solutions with the same fitness which are larger than smaller [261]. Therefore, there exists an evolutionary pressure to increase the size of the network when trapped in local optima.

In the CGP literature there are two rival hypotheses as to why CGP does not suffer from bloat. One based on the role of neutral genetic drift [201] and the other on length bias [82, 84].

The first hypothesis is that CGP does not suffer from bloat due to the presence of inactive genes meaning that the drift hypothesis no longer applies. It is argued in [201] that CGP does not suffer from bloat due to the high levels of genetic redundancy. Their argument is that when a population is trapped in a local optimum, the majority of the mutations which do not course a reduction in the fitness, will be mutations affecting inactive genes; as opposed to active genes. As mutations to active genes are likely to be disruptive to the operation of the phenotype, this results in a lower fitness. Therefore, when CGP is trapped in a local optima, it is more likely to mutate inactive genes than increase the number of active genes. Mutating inactive genes does not cause the phenotype to become larger, using more active genes does. Therefore CGP does not suffer from bloat.

The second hypothesis is that CGP does not suffer from bloat due to length bias causing an evolutionary pressure to topologies of a certain size. It is though that this bias overcomes the pressure to bloat to greater sizes; regardless of what is actually causing the pressure to bloat.

3.9.4 Better Mutation

Typically the most computationally expensive aspect of an EA is evaluating the fitness function. As CGP chromosomes contain many inactive genes two chromosomes are often genotypically different but phenotypically the same. This can easily occur when creating a child through mutation. If the mutation only changes inactive genes, then the parent and child will be phenotypically identical. Evaluating the fitness of this phenotypically identical child is therefore a waste of computational time [83]. This result has been analysed to create more efficient mutation methods which do not result in wasted fitness evaluations [83].

In order to combat these wasted fitness function evaluations three new mutation methods have been proposed [83]. The first method, *skip*, is to simply assign the parents fitness to the child if they are phenotypically identical i.e. skip the fitness evaluation. This can easily be achieved by keeping track of whether any active genes have been mutated; if only inactive genes are mutated, then skip.

The second method, *accumulate*, attempts to increase the rate of neutral genetic drift. This is achieved by first creating a child from the parent using regular mutation. If this child is phenotypically identical to the parent, do not evaluate it but treat it as a parent and use it to create a new child. This process is repeated until a child is created which is phenotypically different. The final child is then evaluated using the fitness function. If it is fitter than the parent than it replaces it, else the parent which was used to create this final child replaces the original parent. This causes a large amount of neutral genetic drift while ensuring only phenotypically different chromosomes are evaluated.

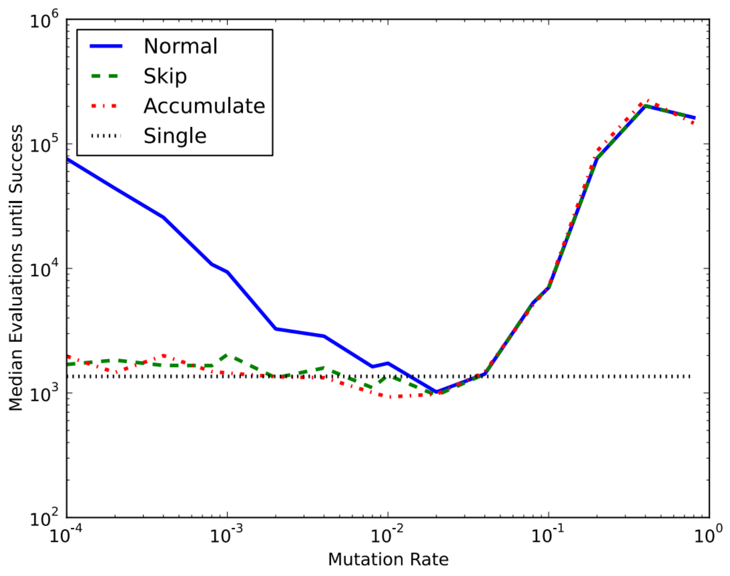


Figure 3.3: Comparison of various mutation methods on the 3 Bit Even Parity benchmark. Image taken from [83].

The third method, *single*, attempts to always produce children which are phenotypically different. This is achieved by continuing to mutate random genes until a single active gene is mutated; at this point the mutation stops. This ensures each child is phenotypically different from its parents. Single also has the advantage that it represents a mutation method which does not require the user to specify a mutation rate.

The three new mutation methods were compared in [83] and a typical example result on the three bit parity task is given in Figure 3.3. As can be seen, skip never performs worse than normal mutation and significantly reduces evaluation time. It is therefore an important development for CGP.

It can also be seen that the additional neutral genetic drift brought about by accumulate does not improve over skip; which is also utilised by accumulate. It can therefore be seen that increasing the levels of neutral genetic drift does not improve the search. It was later shown that CGP rarely re-activates an inactive node without it first being mutated . Using accumulate to increase the level of neutral genetic drift is therefore likely having little effect; as the genes made active have already typically been mutated at least once anyway.

Finally, it can be seen in Figure 3.3 that single mutation method performs very well and does not require the user to specify a mutation rate; a significant advantage. However, in the experiments presented in [84] the following parameters were used: 3000 nodes each with an arity of two. From [207] it is known that these parameters, on this benchmark, result in around 5% of the nodes being active on average. This means on average 20 nodes will be mutated before an active node is reached and the mutation terminates. Mutating 20 nodes in 3000 represents an effective mutation rate of 0.67%; with the additional constraint that at least one active gene will always be mutated.

This means using the parameters chosen in [83], *single* is effectively equivalent to using a mutation rate of 0.67% coupled with the advantage of skip; chromosomes are only evaluated which have changes to active nodes. However, if more or less nodes were used, or a higher or lower arity, single would result in a different effective mutation rate. Therefore, although single appears to have the advantage of removing a parameter choice from the user, it effectively chooses a mutation rate based on the configuration of the chromosomes; which is still chosen by the user. Additionally, there is also no guarantee that the effective mutation rate determined by single will be effective; for instance when using a smaller/larger number of available nodes or a smaller/ larger node arity. Therefore more research is needed into the single mutation method before it can be recommended generally.

3.10 Summary

This chapter has provided a background to CGP. The basic technique has been introduced along with typical parameter choices and common applications. Additionally a number of extensions to the original algorithm have been described as well as theoretical research relating to CGP.

Chapter 4

Cartesian Genetic Programming of Artificial Neural Networks

Cartesian Genetic Programming of Artificial Neural Networks (CGPANN), originally developed by, M Khan in 2010 [154] is a NeuroEvolution (NE) method based on the application of Cartesian Genetic Programming (CGP) to the training of Artificial Neural Network (ANN)s.

This chapter introduces and presents CGPANN as used throughout this thesis along with a collection of initial experiments.

4.1 Structure of this Chapter

Section 4.2 introduces the NE method CGPANN. Section 4.3 gives a discussion of the possible advantageous properties of CGPANN. Section 4.4 gives a summary of previous applications of CGPANN.

The Chapter then presents a number of initial experiments concerning the use of CGPANN. Section 4.5 presents the application of CGPANN to a number of standard benchmarks tasks. Section 4.6 investigates the suitability of connection switch genes; an additional gene type added to previous implementations of CGPANN. Section 4.7 then presents an investigation into whether CGPANN, like CGP, has a natural resistance to program bloat. Finally Section 4.8 gives a closing summary of the Chapter.

4.2 Implementation

CGP is adapted to evolving ANNs simply by the inclusion of connection weight genes ($W_{i,j}$) for every connection gene ($C_{i,j}$) in the chromosome. This alteration, coupled with the use of typical ANN transfer functions (such as logistic sigmoid), and a higher node arity than that typically used by CGP, is all that is needed to encode ANNs using CGP. This alteration results in a genotype of the form given in Equation 4.1; using the less constrained one row form of CGP. Where the symbol definitions are as follows: F_i are the node function genes, $C_{i,j}$ are the connection genes, $W_{i,j}$ are the connection weights genes, O_i are the output genes, i indexes each node in the graph, j indexes each node's inputs, n is the number of available nodes, α is the node arity and m is the number of outputs. A simple example CGPANN phenotype is shown in Figure 4.1. Note that no connection weight genes are associated with the output gene(s). Although this would be possible, in CGP the output gene(s) only index the nodes which are themselves used as outputs, and do not represent computational entities in their own right.

$$F_0C_{0,0}W_{0,0}...C_{0,\alpha-1}W_{0,\alpha-1}.....F_{n-1}C_{n-1,0}W_{n-1,0}...C_{n-1,\alpha-1}W_{n-1,\alpha-1}\colon O_0...O_{m-1} \quad (4.1)$$

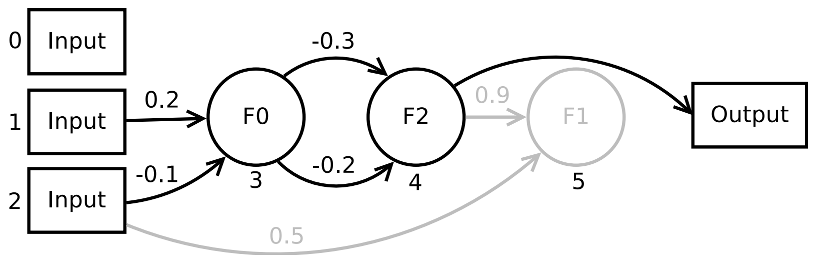


Figure 4.1: Example CGPANN program with three inputs, three available nodes and two outputs.

The initialisation of CGPANN genotypes is almost identical to that of standard CGP. The only difference is the initialisation of the newly added connection weight genes. The connection weights are initialised randomly from an evenly distributed[1] user defined range e.g. ±1 or ±5.

Mutations also follow the same procedure as for standard CGP. With mutation to connection weight genes following the same procedure as for initialisation; changed to

[1]Although other distributions could also be used.

a random value from an evenly distributed user defined range. The selection of which connection weight genes are to be mutated depends upon the mutation method used; typically point or probabilistic mutation. In the work presented throughout this thesis, probabilistic mutation is used. Therefore each connection weight gene is mutated with a give user defined probability.

CGPANN genotypes are decoded into their phenotypes via the same mechanism as described for CGP; Section 3.3. The additional connection weight genes are decoded and assigned to their corresponding connections; at least for those associated with active nodes. Similarly, CGPANN phenotypes are executed via the same mechanism as described for CGP, Section 3.3. Except that the inputs to each functional node are first multiplied by their corresponding connection weight before being passed into, and processed by, the node transfer function.

In the work presented here CGPANN uses the same Evolutionary Strategies (ES) as for CGP, namely a $(1 + \lambda)$-ES.

Interestingly, CGP has also been applied to creating ANNs using a very different method to that described. In work presented by G. Khan [147], multiple regular CGP chromosomes were used to describe different aspect of a biologically plausible neuron. This application of CGP to ANNs was termed Cartesian Genetic Programming Computational Network (CGPCN). In CGPCN groups of neurons, described by multiple CGP chromosomes, are placed together and depending upon their chromosomes, the neuron's dendrites and axons grow, live or die. These dynamic ANNs were inspired by biological brains and when applied to wumpus world [246], a popular artificial intelligence problem, were shown to exhibit continuous learning throughout the task. CGPCN has also been applied to co-evolving two agents which played checkers [148] with promising results.

4.3 Possible Advantages

CGPANN potentially has a number of advantages over many other NE methods. This section discusses a range of interesting properties which come from applying CGP to the training of ANNs. Some of these advantages carry over from advantages which apply to the underlying CGP algorithm.

- CGP is naturally suited to describing ANNs
- CGPANN allows a non-local search of topology

- CGPANN allows custom topology constraints
- CGPANN can evolve recurrent ANNs
- CGPANN is likely not to suffer from program bloat
- CGPANN is likely to benefit from neutral genetic drift
- CGPANN does not utilise crossover
- CGPANN can benefit from previous CGP extensions
- CGPANN can benefit from theory relating to CGP

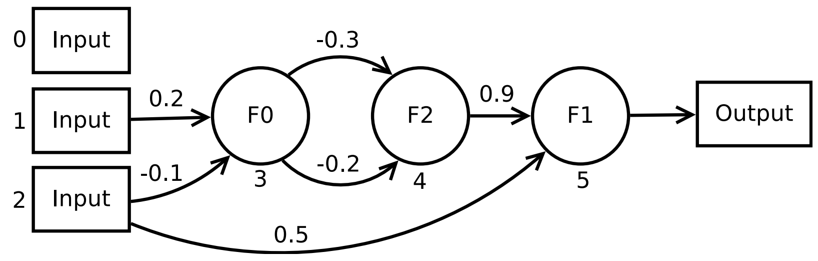
4.3.1 Naturally Suited

As CGP evolves directed acyclic computation graphs, and ANNs are an instance of a directed acyclic computation graph, it is naturally suited to evolving ANNs. The only distinguishing feature of ANNs is that the connections have associated connection weights i.e. ANNs are an instance of a *weighted* directed computational graph. However, as has been described, this is a trivial addition to CGP.

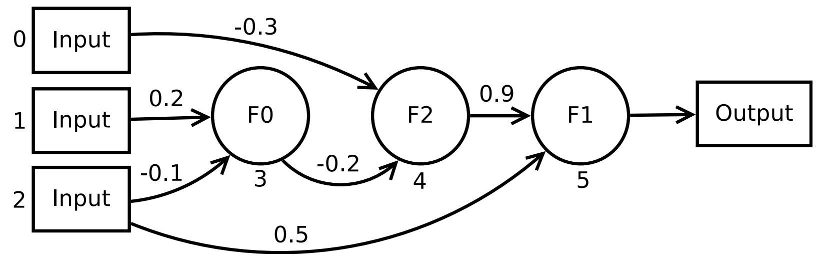
Although a weighted form of tree-based Genetic Programming (GP), such as [275], could be applied as a NE method, it would only be capable of evolving rather limited tree-based ANNs. This means node output values would not be able to be used multiple times; as with CGPANN and ANNs generally. Additionally, the ANNs would only support a single output; unless more atypical forms of tree-based GP were used [324]. This makes CGP much more suited to evolving ANNs than many other GP methods including standard tree-based GP.

4.3.2 Non-Local Search of Topologies

A common technique for manipulating topology used by many NE methods is to start with a minimal network and iteratively add/remove individual nodes/connections during the evolutionary search. This has the possible advantage of first searching over smaller topology instances and only becoming larger if there is evolutionary pressure to do so; promoting simpler solutions potentially aiding generalisation. However, adding individual neurons/connections via mutation is akin to a local search of topology, which is likely to be prone to becoming trapped in topology local optima [15].

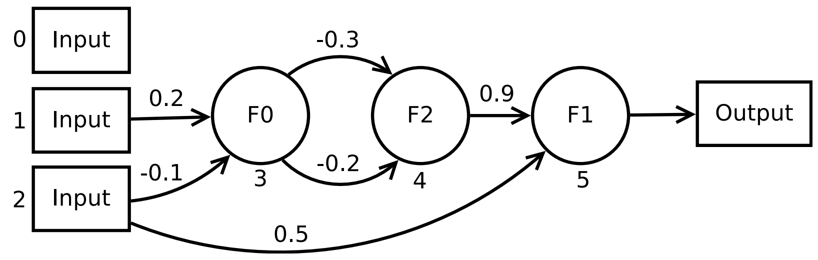


(a) Parent

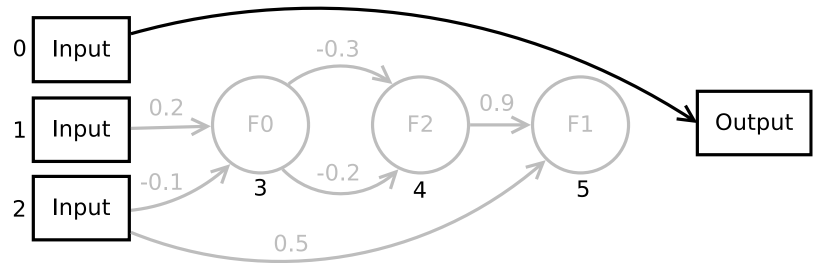


(b) Child

Figure 4.2: Small topology mutation. The parent chromosome (a) has had a single topology mutation resulting in a child (b) where node 4 is now connected to input 0. All other nodes are left unchanged.



(a) Parent



(b) Child

Figure 4.3: Large topology mutation. The parent chromosome (a) has had a single topology mutation resulting in a child (b) where the output is now connected to input 0. All other nodes are now unused.

The mechanism by which CGPANN evolves topology means that mutation can cause small or large structural changes; see Figures 4.2 and 4.3 respectively. This means that, at least in theory, CGPANN could more easily escape topology local optima.

Additionally, although CGPANN does not start with minimal solutions and grow larger, there is a length bias [82] to a given, often small, number of active nodes. This could therefore aid with generalisation.

4.3.3 Topology Constraints

Another topology benefit of CGPANN is that it can also be used to easily evolve topology under given user constraints. If the original rows and columns forms of CGP is used, CGPANN can restrain the search to topologies of a given width or depth. This can have a number of advantages. For instance, if it were known that the final application had n parallel processors to run the ANNs created by CGPANN. CGPANN could be configured to never contain more than n nodes in a given layer; by setting the number of rows and columns and the levels back parameter accordingly. Additionally, if there were a constraint that the final ANNs must run in a time budget of m node evaluates, CGPANN could evolve solutions with n rows, m columns and levels back equal to one; thus meeting the time restraint by utilising the parallel hardware. Under these conditions CGPANN is free to evolve topology within the computational constrains of the final application.

Additionally, if the evolved ANNs were to be implemented in hardware, there may be limited connectivity between nodes. For instance a node in a given layer may only have connections to nodes in the previous layer, or the previous two layers. In this scenario CGPANN could evolve ANNs with CGPs levels-back parameter set as one or two (see Section 3.5); again allowing topology evolution within real world constrains; this time connectivity limitations.

It may also be possible to limit the topologies which can be reached using other topology and weight evolving NE methods. For instance, Evolutionary Programming Artificial Networks (EPNet) could be used where the disallowed connections could be set in the matrix of connections and never allowed to be altered during mutation. Additionally, Hypercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT) could also be used by not allowed certain connections to be present in the final ANNs regardless of the output of the NeuroEvolution of Augmenting Topologies (NEAT) program constructing the network. However, for other NE methods, such as NEAT, Cellular Encoding (CE) or GeNeralized Acquisition of Recurrent Links (GNARL), restricting the created topologies would require analysis of the constructed phenotypes and then retrospective prevention, rather than certain topologies never being possible.

4.3.4 Recurrence

As is shown later in Chapter 5, CGP can be easily extended to be capable of describing cyclic program structures. As this same extension can also be applied to CGPANN it too is capable of evolving Recurrent Artificial Neural Network (RANN)s; as is demonstrated in Chapter 10.

Although many other NE methods can also create RANNs, such as Conventional NeuroEvolution (CNE), GNARL or NEAT, this means the ability of CGPANN to create RANNs is not an exceptional feature. However, if it was not present it could be considered a limitation.

4.3.5 Program Bloat

Due to an absence of discussion in the literature, it is unknown whether many topology optimising NE methods suffer from program bloat; See Table 2.1. It does however appear likely that many methods, such as Evolutionary Acquisition of Neural Topologies (EANT)'s tree-based representations which utilises sub branch crossover, would be likely to bloat.

One of the main theories concerning why GP in general suffers from bloat is the drift hypothesis [261]; described in Section 3.6.1. As the drift hypothesis applies to many of the topology evolving NE method it appears likely that they would also / do suffer from program bloat. However many NE method bias the search towards smaller topologies, for instance by making the likelihood of removing a node much higher than adding a node. Therefore without empirical investigation it is unknown whether these methods do or do not suffer from program bloat.

Whether CGPANN, like CGP, is resilient to bloat is investigated in Section 4.7.

4.3.6 Neutral Genetic Drift

One of the distinguishing features of CGP is the high proportion of inactive genetic material aiding the search though the process of neutral genetic drift; see Section 3.6.2. As CGPANN contains all of the properties of CGP which enable neutral genetic drift, it too is likely to benefit from its presence.

The role of neutral genetic drift in CGP and CGPANN is investigated in detail in Chapter 7.

4.3.7 No Crossover

Previous research has investigated using CGPANN with the crossover evolutionary operator [191]. This work utilised the same floating point representation as previously used for CGP [42, 198, 278]. However, in the results presented [191], it was shown that the use of crossover provided no benefit to CGPANN.

In the NE literature crossover is also often considered a disadvantage: "*One of the main problems for NE is the Competing Conventions Problem, also known as the Permutations Problem* [236]. "*Competing conventions means having more than one way to express a solution to a weight optimization problem with a neural network. When genomes representing the same solution do not have the same encoding, crossover is likely to produce damaged offspring.*" [266]. Although, as is discussed in Section 2.7.4, the evidence for believing that competing conventions is a problem for NE is not as strong as is often suggested in the literature. Additionally, methods which have been shown to prevent the issue of competing conventions, such as NEAT, are typically very complex.

Therefore, in this thesis, CGPANN is implemented without crossover due to it empirically being shown not to be beneficial and because of the possible issue of competing conventions; even if the evidence for its detriment is not strong. However, this does not represent a disadvantage of CGPANN. It has been shown that CGP makes use of neutral genetic drift to aid the escape of local optima without requiring large populations or the use of crossover; see Section 3.9.1. Additionally, this results in a much simpler algorithm as not only does crossover not have to be implemented, but no complexities have to be added to combat the possible issue of competing conventions.

4.3.8 Extensions

As is described in Section 3.8, CGP has had a number of extensions applied the base algorithm since its original formulation in 2000. Many, if not all, of the extensions described would also be applicable to CGPANN. An example of this has already been described where CGP's crossover extension [42, 198, 278] was applied to CGPANN [191].

Another interesting possibility would be to applying the Modular Cartesian Genetic Programming (MCGP) extension to CGPANN. The application of MCGP would enable the evolution of Modular Neural Network (MNN) where whole sections of the ANN can be captured and reused throughout the wider ANN. Interestingly, the modules created from applying MCGP to ANNs would be very different from what are currently termed MNN.

In the ANN literature, MNNs are typically enforced by selecting n modules and training them independently; or alternatively by identifying modules if they happen to be present in trained ANN. For instance in an extension to EPNet [173], the beginning of modules are identified and their shared nodes actively removed to enforce pure MNNs. The modules created by applying MCGP to ANNs would always be pure (no interconnections) and could also be repeated multiple times throughout the MNN.

It would also be possible to apply the Self Modifying Cartesian Genetic Programming (SMCGP) extension to CGPANN thus evolving ANNs which grew and self-adapted during their lifetime; possibly allowing similar application to that of CGPCN.

Additionally it would be possible to apply the multi-chromosome CGP extension in order to describe ANNs using many chromosomes. This would results in a multi-chromosome NE method such as those described in [195].

What these extensions demonstrate is that by basing a NE method on a mature Evolutionary Algorithm (EA), many of the previous developments can be directly applied and potentially benefited from. Additionally, any future developments to CGP are also likely applicable, such as better use of neutral genetic drift or the discovery of more suitable mutation operators; such as those discussed in [83]. Similarly any developments made to CGPANN can likely be “back-ported” to the underlying CGP algorithm.

Finally, although CGPANN is based on CGP, it does not mean that it cannot also benefit from extensions made to other GP methods. For example, tree-based GP has been previously combined with reinforcement Q-learning [295] and applied to the domain of maze searching [53]. In such work it was possible to facilitate both Lamarckian [171] and/or Baldwinian evolution[2] [21] through continuous learning and adaptation during fitness evaluation. Interestingly, a similar combination of CGPANN and Q-learning may also be viable. This could be achieved by using CGPANN to transform input data into the states of a Q-table to be adapted by Q-learning; possibly lowering the dimensionality or providing more tractable state information. Alternatively, CGPANN could be used to selected between differing Q-tables depending upon the current inputs; more similarly matching the work in [53]. Therefore, not only can CGPANN benefit from CGP developments and extensions, but it can also benefit from developments in the wider GP community.

[2]Where Baldwinian evolution considers the ability of an individual to learn during its lifetime with regard to the likelihood of it reproducing and passing on this ability to the next generation.

4.3.9 Previous Theory

One of the advantages of basing a NE method on an established EA is that existing theory can often be directly applied in order reason about the behaviour of the new method.

An example of this is the previously developed theory concerning the presence of length bias in CGP; see Section 3.9.2. The rationale behind why CGP exhibits a length bias directly applies to CGPANN. Although the larger arity typically used by ANNs means that the length to which there is a bias will be different to when only one or two node inputs are used. Additionally, the presence of a bias may provide a benefit to CGPANN, as the type of application to which ANNs are often applied include classification, symbolic regression and forecasting; tasks which typically suffer from over training. Having a length bias restraining the growth/size of the network may therefore be an advantage by aiding generalisation.

Additionally, the mutation operators described in Section 3.9.4 will also directly apply to CGPANN. However as many of the methods indirectly make assumptions about the node arity used they may not be as beneficial. For instance, as discussed in Section 3.9.4, the reason the *single* mutation method may have been so effective is because it was equivalent to a 0.67% mutation rate; based on the average number of active nodes for the particular parameters used. However, using a larger arity will shift the length bias so more nodes on average are active; loosely because more connections means more things can be connected to. This means using the higher arity associated with ANNs would result in *single* having a much *lower* effective mutation rate; as there are more active genes the chances of randomly selecting one is higher. Therefore the influence of a higher arity may diminish the effectiveness of the single mutation method. However, mutation methods such as skip will still be likely to offer a significant advantage.

4.4 Previous Applications

In addition to the work presented in this thesis, CGPANN has also been previously applied to a range of application; both real world and “toy”.

CGPANN has previously been applied to both the single and double pole balancing control problems; [154] and [156] respectively. A form of recurrence has also been previously added to CGPANN, in the form of an enforced Jordan network[3], which has been

[3]Where one or more program outputs are fed back as inputs.

applied to a form of the double pole balancing where velocity information is withheld; see Appendix A.

CGPANN has been applied to a range of classification tasks including the breast cancer data set [5, 191] provided by the University of Wisconsin Hospital [189]. Additional classification applications include classification of arrhythmia [6] and detecting breast cancer in mammograms [7].

Additionally, CGPANN has been applied to forecasting using both purely feed forward topologies and those which enforced a Jordan architecture. Forecasting applications include predicting client requests in Cloud Data centres [12], estimating the fame size for multimedia streaming applications [146, 151, 280], predicting foreign currency exchange rates [214], forecasting short-term daily peak electric power supply load [149, 152] and assessing the sustainability of new food products [11].

4.5 Initial Experiments

This section describes a number of initial experiments applying CGPANN to a range of standard benchmark tasks. These experiments serve to demonstrate the basic application of CGPANN to standard benchmarks. It also allows the performance of CGPANN to be compared against other NE methods.

Although CGPANN has been previously applied to two of the benchmarks used here (double pole balancing and breast cancer classification), the experiments are repeated for a number of reasons. Firstly, the previous CGPANN results included the use of connections switch genes. This is an unnecessary addition to CGP which is not required to evolve ANNs; as shown in Section 4.6. Secondly, the implementation of the benchmarks used in previous application of CGPANN does not match that typically used; this makes the previous comparisons less valid. Thirdly, these initial experiments serve to demonstrate the functionality of the CGPANN implementation used throughout this thesis before it is extended and applied to more challenging applications.

4.5.1 Double Pole Balancing

The double pole balancing benchmark is a popular benchmark commonly used within the NE community; described in Appendix A. It is also more standardised than its single pole counterpart, making it more useful as a benchmarking task; see Section 2.7.7.

As the pole balancing benchmark requires the production of both positive and negative outputs the bipolar logistic sigmoid function is typically used; as it is here. The bipolar logistic sigmoid function is the logistic sigmoid 'stretched' to the range ±1; Equation 4.2. Additionally, the majority of the NE methods applied to the double pole balancing used a connection weight of ±1. Although this may not be an optimal choice, as the majority of the other NE methods this restriction of weights, it should be considered part of the benchmark. This is because if two methods produce differing results but both use differing connection weight ranges, it would not be known if this were because of the algorithmic differences or the connection weight range used.

$$f(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{4.2}$$

CGPANN has been previously applied to the double, and single, pole balancing tasks in [154] and [156] respectively. In such work CGPANN showed remarkable results for both tasks, solving them faster (in terms of number of evaluations) than any other NE method. However, due to implementational differences in the benchmark, the usefulness of the result is diminished. For instance when M. Khan et al. conducted their experiments they used both logistic and radial basis function transfer functions rather than just logistic transfer functions. They also implemented a "bang-bang" control system rather than the typical continuous force control system; see Appendix A for details of the two control systems. As it is unknown what effect these differences have on the difficulty of the benchmark, the results cannot be used in a fair comparison with other NE methods.

Like many of the other NE methods applied to this task, the parameters used here for CGPANN were optimised before presenting results. The parameters found which produced the best results were: a mutation rate of five percent, one hundred available nodes each with an arity of thirty. The evolutionary strategy was left unchanged as (1 + 4)-ES. Following the most commonly used conventions, a continuous control system was used with the magnitude of the applied force to always made to be greater than $\frac{1}{256} \times 10$N; see Appendix A.

The results of applying CGPANN to the double pole balancing benchmark, averaged over 100 runs, are given in Table 4.1 where it is compared to many other NE results.

As can be seen in Table 4.1, CGPANN compares very well, outperforming many popular NE methods including: Symbiotic Adaptive NeuroEvolution (SANE), Enforced Sub-Population (ESP) and NEAT. CGPANN also produces comparable results to Coopera-

Table 4.1: Comparison of results for the Double Pole Balancing benchmark

Method	Evaluations	Standard Deviation	Averaged Over
EP [305]	307200	-	-
CE [93]	34000	-	>30
CNE [90]	22100	-	50
EuSAIN [225]	≈19000	-	100
SAIN [90]	12600	-	50
Q-MPL [90]	10583	-	50
ESP [90]	3800	-	50
NEAT [266]	3578	2704	120
NEvA [276]	2177	-	50
CGPANN	**1111**	**1476**	**100**
CoSyNE [87]	954	-	50
CMA-ES [130]	895	-	50
DXNN [252]	725	-	100
DirE [93]	410	-	>30

tive Synapse NeuroEvolution (CoSyNE) and Covariance Matrix Adaptation Evolutionary Strategies (CMA-ES) but requires significantly more evaluations than Deus Ex Neural Network (DXNN) and Directly Encoded NeuroEvolution (DirE). As the complete results of the other methods are not available, no statistical significance testing can be undertaken; the standard deviations are often given, but as the distributions are typically non-normal, their usefulness is diminished[4].

Unfortunately the benchmark implementation used by some of the other NE methods differed from the typical implementation meaning a true comparison is not possible. These differences include: NEAT used their own modified sigmoid function, DirE included an extra unit bias input, ESP used a weight range of $[-6, 6]$ and CoSyNE $[-10, 10]$; the others used a range of $[-1, 1]$ as did CGPANN in the results presented here. It is likely that the transfer function, extra bias input and connection weight ranges will have an effect on the difficulty of this benchmark, but to what extent is not known. Therefore this diminishes the comparison. As previously discussed in Section 2.7.7, the lack of standardised benchmarks in the NE literature is a major issue and one which needs to be addressed if fair comparative NE studies can be undertaken.

[4]In fact as can be seen from the CGPANN results in Table 4.1, the standard deviation is larger than the mean. This demonstrates that the data is non-normal in distribution.

Table 4.2: Comparison of results for the Ball Throwing benchmark

Method	Evaluations	Standard Deviation	Averaged Over
CoSyNE [164]	10224	-	100
Compressed CoSyNE [164]	8220	-	100
CGPANN	**6069**	**5990**	100

4.5.2 Ball Throwing

Although not as widely adopted as the double pole balancing benchmark, the ball throwing benchmark has been used by one of the more popular NE methods, CoSyNE, and a derived method, Compressed CoSyNE. Although this makes for a small comparison, it does help evaluate CGPANN and provides further results for future NE to compare against. It is also an example of the type of task to which ANNs typically cannot be applied, a reinforcement learning type task, and so, like the double pole balancing, serves to showcase NE. The ball throwing benchmark is described in Appendix A.

For this benchmark, the CGPANN parameters were also varied in order to find suitable values. The parameters which produced the best results were as follows: a mutation rate of ten percent, a maximum of forty nodes each with a maximum arity of ten. The evolutionary strategy was left unchanged as $(1+4)$-ES as was the connection weight range as ± 1. Using these parameters ninety-eight of the one hundred runs found a solution in the allowed 20001 evaluations.

The results of applying CGPANN to the ball throwing benchmark, averaged over 100 runs, are given in Table 4.2. It can be seen from Table 4.2 that CGPANN outperformed both CoSyNE and Compressed CoSyNE in terms of the number of evaluations, with two of the one hundred runs failing to find solutions.

4.5.3 Proben1 - Cancer

The Proben 1 - cancer benchmark is a popular classification task often used in the field of ANNs; see Appendix A. Unfortunately there are many version of the breast cancer data set and many papers present, *and compare*, results using very different experimental set-ups. This phenomena of popular classification benchmarks splitting up into many separate versions has previously been documented for the Iris dataset [28] and also serves to diminish the worth of the benchmark, limiting the number of fair comparisons which can be made. Here the results presented use the Proben 1 version of the cancer dataset and follow the methodology laid out in the Proben 1 document [229].

Table 4.3: Comparison of results for the Proben 1: Cancer 1 benchmark

Method	Proben Compliant	% Train Err	% Test Err
LM [94]	No	5.5	12.2
MLP [235]	No	-	6
SCG [94]	No	0.2	5.4
MLP [213]	No	-	5.18
RAIC [97]	No	-	5.01
NEFCLASS [213]	No	-	4.94
SFC [2]	No	-	4.43
C4.5 [97]	No	-	4
LLS [94]	No	4.0	4.0
Fuzzy-GA [223]	No	3.00	3.98
CMAC ANN [312]	Yes	0.59	3.94
RBFNN-Kalman [251]	No	-	3.6
BP [224]	Yes	-	3.506
GDX [94]	No	2.3	3.3
RBFNN-RLS [251]	No	-	3.2
LLWNN-RLS [251]	No	-	2.8
AR + ANN [141]	No	-	2.6
SBS-BP-PSO [120]	No	-	2.49
ACS [224]	Yes	-	2.184
CGPANN	**Yes**	**2.34**	**2.18**
MFNNCA [140]	Yes	24.86	2.00
GA-MOO-ANN [9]	Yes	-	1.9
M-RAN [317]	Yes	-	1.72
LP MSM [188]	No	0.0	1.7
LS-SVM [226]	No	-	1.47
MFN [317]	Yes	-	1.38
EPNet [314]	No	3.773	1.376
LSA machine [10]	No	-	1.2
SBS-BP-LM [120]	No	-	1.17
HS [144]	No	-	0.71

In this study CGPANN is naively applied to the classification task with no steps taken to prevent over training. CGPANN was trained using the training set and then evaluated on the testing set. A more rigorous application of CGPANN as a classification method is presented in Chapter 9.

The parameters which produced the best training performance used a mutation rate of one percent, a maximum of one hundred nodes each with a maximum arity of forty. The evolutionary strategy was left unchanged as (1 + 4)-ES. The search was given 5000 generations; 20001 evaluations. The average result from using these parameters, average over fifty runs, are given in Table 4.3 along with the results of many other methods.

Most of the results published in the literature, given in Table 4.3, do not conform to the Proben1 documentation [229]. Although there is no expectation for other researchers to have used a specific form of the benchmarks over another, the fact that there is a large range of implementations weakens any comparisons which can be made and diminishes the benefit of using benchmark problems. Of the results which do not follow the methodology of the Proben1 document, the methods used vary and are not standardised. The differences include: size of the data set, pre-processing of the data, sizes of training, validation and test sets, different permutations of the data set and differing fitness functions used. Of the results which do follow the Proben1 document, some split the testing dataset into testing and validation; this is allowed and documented by the Proben1 standards. This validation dataset can then be used to test how well each solution has generalised and is used as an early stopping criterion. An early stopping criteria was not used by the CGPANN presented here; however CMAC ANN, MFNNCA and GA-MOO-NN did use an early stopping criterion. Additionally MFNNCA investigated using a range of training epochs and presented the results which performed best on the testing data. This is of course invalid, and removes the point of using a testing set to assess generalisation.

4.5.4 Discussion

As can be seen in the presented results, overall CGPANN performs reasonably well on the double pole balancing compared to many other NE methods, performs very well on the ball throwing and performs well on the Proben1: Cancer1 benchmark compared to a wide range of Machine Learning (ML) methods.

This section has also repeated previous experiments applying CGPANN to a number of benchmarks. Although the results are not as impressive as previously presented, the implementation of the benchmarks was much more standardised and CGPANN was still shown to be a competitive technique. The results therefore demonstrate that the basic application of CGP as a training method for ANNs produces good initial results and is worthy of further investigation.

4.6 Connection Switch Genes

In the original implementation of CGPANN [154] by M. Khan, binary valued connections switch genes were added to CGPANN in order to enable variable node arity. These connection switch genes were added to every node input and could be set as open or

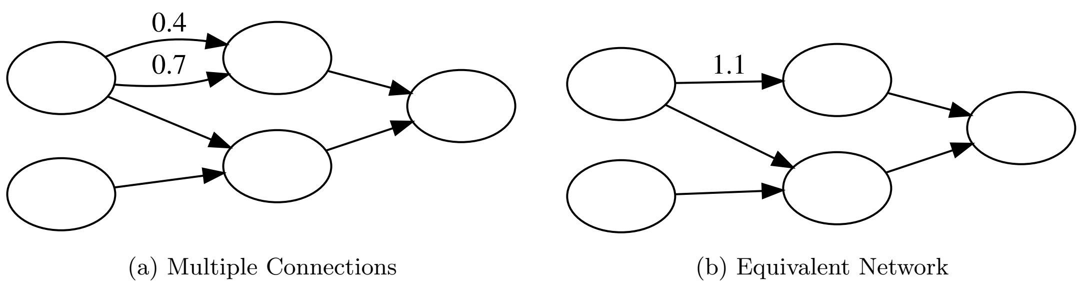


Figure 4.4: Depiction that multiple inter-node connections (a) is equivalent to one connection with the sum of the individual connection weights (b).

closed; effectively adding and removing the connection. With the presence of connection switch genes the user specifies a maximum node arity, allowing evolution to determine the number of connections actually used within the [0,α] range; where α is the maximum node arity.

The rationale behind variable arity is that neurons can have a wide range of arities; unlike GP nodes which typically have fixed arity. Connection switch genes therefore represent a simple method of implementing variable node arity during evolution.

However, even without the inclusion of connection switch genes CGPANN could be argued to be capable of evolving node arity. This is because CGPANN places no limitation on which previous nodes a give node connects to. For instance, it is possible for two nodes to be connected by multiple connections. Figure 4.4 (a) gives a simplified example. In this case multiple connections between the same two nodes is equivalent to one connection with the sum of the individual connection weights; as shown in Figure 4.4 (b). It could also be argued that all NE methods can evolve node arity by thinking of connection weights as representing 'fuzzy' connections. This however is not considered in this section as for weights values other than zero there is still technically a connection. Additionally, the arity of each node could also be adapted by mutation setting the connection weight value of a given connection to zero. However, as the probability of mutation setting a connection weight value to exactly zero is very low, this is not considered here.

This means that CGPANN can *effectively* vary each nodes arity by exhibiting multiple connections between the same two nodes. However, this behaviour means that the maximum connection weight range set by the user can be exceeded. For instance, if the connection weight range was set as ±1, the presence of multiple connections between the same two nodes would effectively increase the range to $\pm(1\times\alpha)$; where α is the node arity. If allowing the connection weight range to exceed that set by the user is considered undesirable, alternative decoding strategies could be used for its prevention. For instance, only the first of multiple connections between two nodes could be decoded from the genotype into the phenotype; thus, allowing variable arity whilst disallowing the connection weight range to be exceeded. Alternatively, the average connection weight of multiple connections between two nodes could be used.

Interestingly however, even when connection switch genes are used it is still possible for the node arity to effectively vary using the same mechanism described. Therefore the effective connection weight range can also still be exceeded. Interestingly, no previous discussion of this artefact has been presented for any of the NE methods investigated. However, unless actively prevented, the behaviour would be possible in many topology adapting NE methods.

Finally, the presence of connection switch genes means that it is possible for CGPANN genotypes to describe phenotypes where there is no continuous connection from inputs to outputs. For instance, if the node index by an output gene had all of its connection switches set to open i.e. not connected. Although such solutions would likely score low fitness and be quickly dropped from the population, it does add knowingly poor solutions to the solution space.

4.6.1 Empirical Investigation

Although theoretically it appears that connection switch genes may not be required in order to allow CGPANN to evolve the arity of each node, they still may be of benefit to the search. Additionally, previous use of connection switch genes in [154] did not provide empirical evidence as to whether or not they aided CGPANN's evolutionary search. Therefore this chapter provides an empirical investigation into the use of connection switch genes on CGPANNs evolutionary search.

In order to assess whether connection switch genes are beneficial for CGPANN, a simple experiment is now presented. The experiment applies CGPANN to a range of benchmark tasks with and without the use of connection switch genes. The results are then presented along with statistical significance testing. The statistical significance testing methods utilised are described in appendix B.

In all cases CGPANN uses following parameters: $(1 + 4)$-ES, 1000 generations, 50 runs, 4% probabilistic mutation, fifty nodes, ± 5 connection weight range, a node arity of ten and a function set consisting of only the logistic sigmoid. The number of generations is kept low so many runs failed to find a solution. This means that the average fitness achieved can be compared. An alternative strategy would be to allow a very large number of generations and compare the number of evaluations required to reach a given fitness value. As comparing fitness requires less computational time, and means benchmarks can be used with no known best fitness, the former method of evaluation is used. The number of nodes is also kept relatively low to increase the difficulty of the task.

The benchmarks used for this investigation are: ball throwing, double pole balancing, full adder and monks problem 1. These benchmarks are described in Appendix A. For clarity, for the first three benchmarks, ball throwing, double pole balancing and full adder, a higher fitness represent a better solution. In the remaining case of monks problem 1, a lower fitness represents a better solution.

The results of the investigation are presented in Table 4.4. As can be seen in Table 4.4, for half of the benchmarks investigated the use of connection switch genes produced better results and for the other half they produced worse results. In the cases where connection switch genes were shown to improve the results, no statistical significance was found. In the cases where connection switch genes were show to worsen the results, the difference was statistically significant. Finally in all cases the effect size was small.

Table 4.4: Applying CGPANN to a range of benchmarks with and without the use of connection switch (CS) genes. The best performance is given in bold. Statistical significance is also give with $p < 0.05$ given in bold. The effect size is also given.

Benchmark	Without CS	With CS	U-test	KS-test	Effect Size
Ball Throwing	**7.04**	6.09	**7.00E-3**	**3.17E-2**	0.6226
Double Pole balancing	96015	**98007**	5.68E-1	1.00E-0	0.51
Full Adder	**15.94**	15.64	**1.30E-3**	9.51E-2	0.6188
Monks Problem 1	3.89	**2.94**	2.68E-1	6.78E-1	0.5594

4.6.2 Discussion

The results presented in Table 4.4 demonstrate that the use of connection switch genes have no real influence on the search for CGPANN. The only statistically significant difference seen is that, on average, they worsen the results slightly. It has also been noted that CGPANN can effectively evolve node arity without the presence of connections switch genes. These reasons, coupled with the fact that using connections switch genes allows CGPANN to create networks where there is no connection between inputs and outputs, led to the decision not to include connection switch genes in the work presented throughout this thesis.

4.7 Program Bloat

An important property of CGP is that it does not suffer from program bloat, as is discussed in Section 3.6.1. As bloat is an undesirable property of EAs, or more specifically GP, it is important that this characteristic of CGP is also present in CGPANN. This section presents an empirical investigation which assess if this is the case.

Bloat can be defined as "*program growth without (significant) return in terms of fitness*" [228], that is, if program length is increasing disproportionately to fitness improvements, then bloat is said to be occurring. This definition has been formally stated as a metric which measures the amount of bloat on any given generation [284]. Here a variation on this bloat equation is used; given in Equations 4.3-4.5:

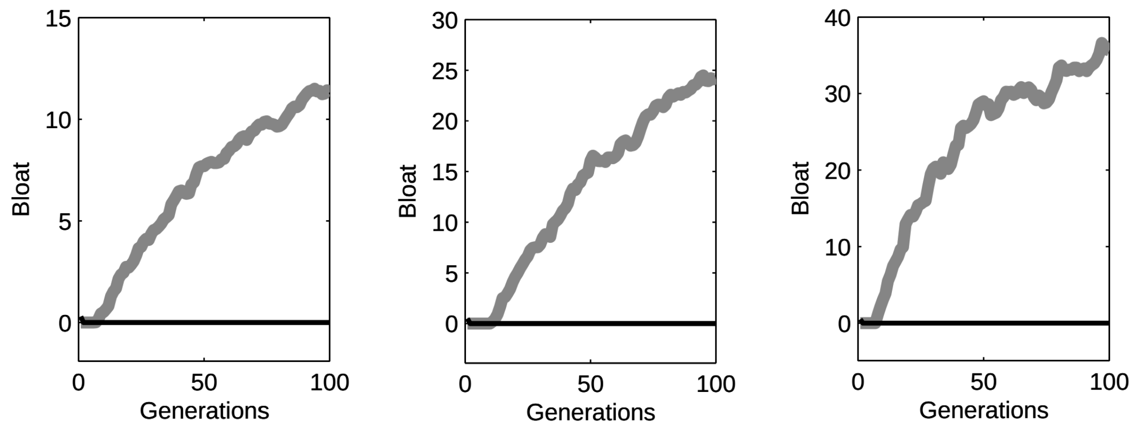
$$N(g) = \frac{\hat{A}(g) - \bar{A}(0)}{\bar{A}(0)} \tag{4.3}$$

$$D(g) = \frac{\bar{F}(0) - \hat{F}(g)}{\bar{F}(0)} \tag{4.4}$$

$$B(g) = \frac{N(g)}{D(g)} \tag{4.5}$$

Where $B(g)$ is the bloat at generation g, $\hat{A}(g)$ is the number of active nodes used by the fittest member of the population at generation g, $\bar{A}(0)$ is the average number of active nodes used by the population at generation 0, $\bar{F}(0)$ is the average fitness of the population at generation 0 and $\hat{F}(g)$ is the fitness of the fittest member of the population at generation g. Equation 4.5 holds when the target is to minimise the fitness to zero. When the fitness is to be maximised the fitness values can be amended by subtracting the current fitness from the target fitness; thus transforming the problem into a minimisation task. The equation gives the ratio of increase in program size to improvement in fitness since the initial population. If the program size is increasing disproportionately to fitness, then the bloat value will also increase; thus indicating bloat.

The bloat equation given in [284] was adapted here to show the amount of bloat exhibited by the fittest member of the population; as opposed to the average bloat of the population. There are two reasons for this alteration: 1) CGP uses a $(1 + \lambda)$-ES without crossover, and so the only solution of interest is the current fittest. 2) The small population



(a) $f(x) = log(x)$ (b) %F (c) LD50

Figure 4.5: The bloat metric comparing standard tree-based GP (light gray) and DynOpEq GP (black) on (a) symbolic regression and (b)(c) two real world classification tasks. Images taken from [284].

sizes typically used by CGP leads to very noisy level of average active nodes and fitness values which are hard to analyse graphically; at least without applying further averaging.

It is worth noting that the value of bloat gives the ratio of program size and fitness compared to the random initial population. This means that a consistent bloat value of x for n generations indicates no bloat over those n generations. If the bloat value x increases over n generations this indicates program bloat. Therefore it is increasing x which indicates bloat; not necessarily high values of bloat. A high but stable value of x indicates that the solutions are disproportionally larger than randomly created initial solutions, compared to the fitnesses of the initial solutions, but that their size is not increasing disproportionally to fitness.

Figure 4.5 gives three examples of the unaltered bloat metric when used by the original authors; see [284] for further details of their experimental implementation. As can be seen in Figure 4.5, bloat is easily detected by a high continuous increase in the bloat metric.

Although it has been previously identified that CGP does not suffer from program bloat [201], previous research did not use a formal measure of bloat, such as those given in Equations 4.3-4.5; only raw program size. Although this is unlikely to influence the conclusions, it is more rigorous to consider program size in relation to fitness. Additionally, as stated in [201] with regard to CGP: "*Experiments performed indicate that implicit intron growth is not a problem and no measures need to be taken to suppress it (at least for some Boolean problems).*". That is to say, problems other than Boolean functions have not been investigated. For these reasons the presence of bloat is evaluated first for CGP, which is then also used for comparison when assessing CGPANN.

4.7.1 CGP

When investigating program bloat in CGP, the Boolean benchmark of six bit even parity and the symbolic regression task Pagie 1 are used; see Appendix A. Although a wider selection of benchmarks would be more rigorous, as this work is to supplement previous results [201], and to serve as a comparison to CGPANN, it is sufficient.

The six bit even parity benchmark is a Boolean benchmark task of creating a digital circuit which generates the even parity bit for a given six bit string; See Appendix A.4. The task is to maximise the number of correct parity bits generated over all possible inputs representing a maximum fitness of $2^6 = 64$. The function set contains: {AND, NAND, OR, NOR}; the XOR and NXOR gates are omitted to increase the difficulty of the tasks.

The Pagie 1 is a symbolic regression tasks described in Appendix A.5.2. Here the function set contains: $+ \ - \ \times \ \% \ e^n$ and $\ln(|n|)$ and the goal is to minimise the sum of absolute error.

For both benchmarks the parameters used are as follows: (1 + 4)-ES, three percent probabilistic mutation, one hundred nodes, ten thousand generations and fifty runs.

The results of applying CGP to the two benchmarks are given in Figures 4.6 and 4.7 for the six bit parity and Pagie benchmarks respectively. Each figure gives the fitness achieved, the number of active nodes and the level of bloat at each generation all averaged over fifty runs.

As can be seen in Figures 4.6 and 4.7, CGP is not exhibiting program bloat. The level of bloat is low and, most importantly, stable throughout the evolutionary search. In the case of the six bit parity it can be seen that the level of bloat is initially high but falls and becomes stable as the generations elapse. A possible explanation of this could be due to the average random program size containing an insufficiently small number of nodes to solve the task. In which case there would be an evolutionary pressure to increase the program size. Once the program size is sufficiently large the problem is then tractable. Regardless, it can be seen than in the long term CGP is not bloating when applied to the parity task. In the case of the Pagie symbolic regression task it can be that there is a slight increase in the level of bloat during the beginning of the search. However, as the actual increase is so minor, it is unlikely to cause any meaningful consequences. Additionally, the level of bloat appears to stabilise as the number of generations elapse.

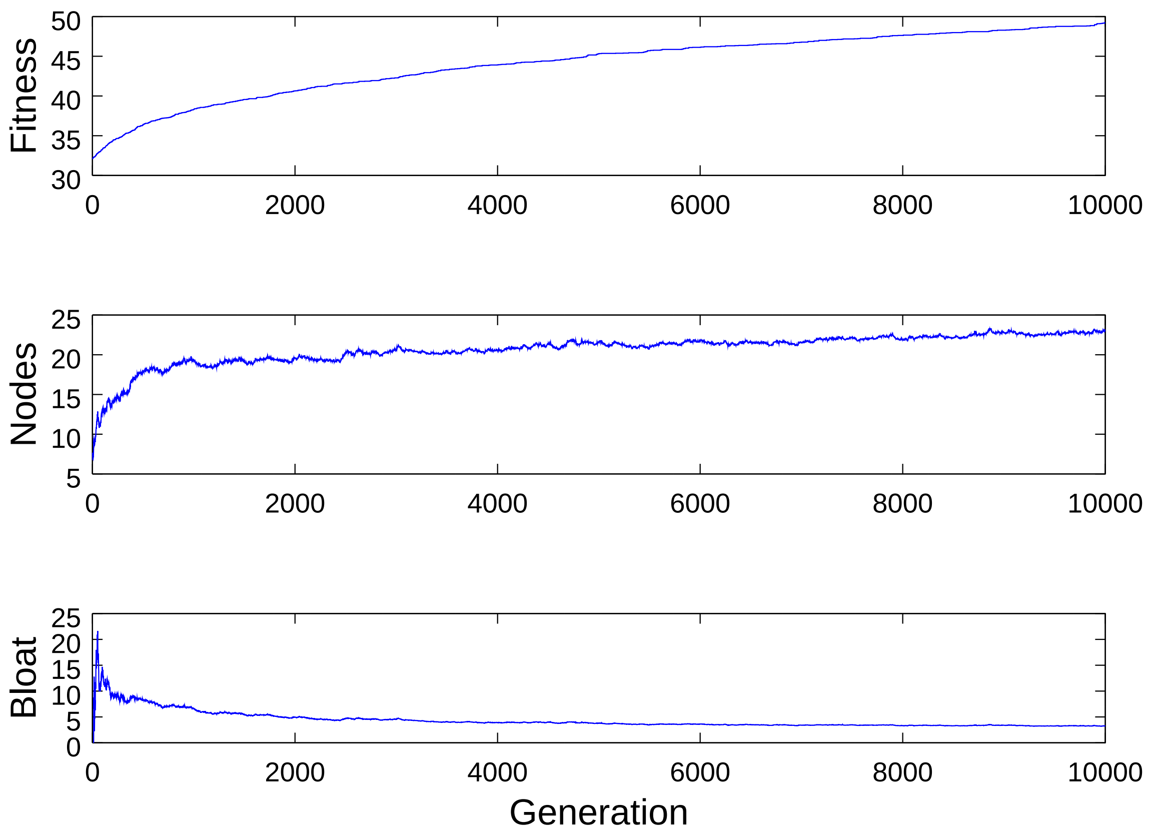


Figure 4.6: Average fitness, number of active nodes and program bloat Vs. generation for CGP applied to the six bit even parity benchmark.

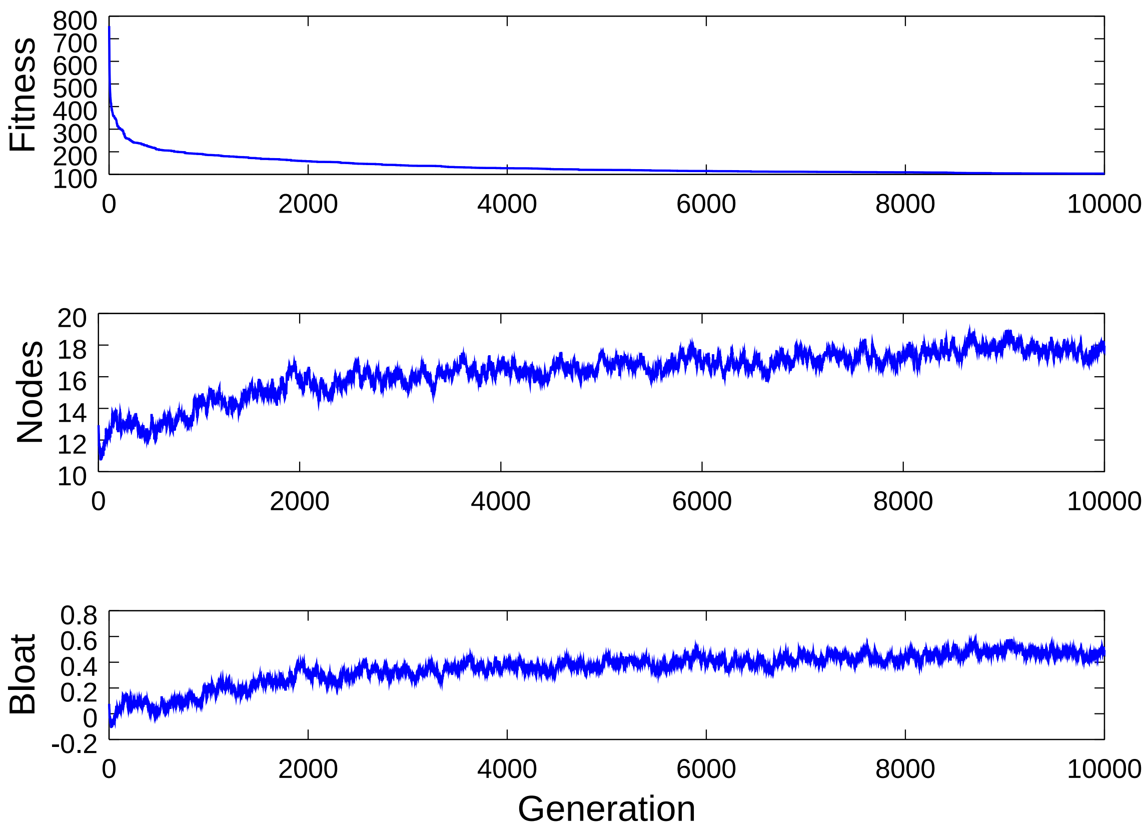


Figure 4.7: Average fitness, number of active nodes and program bloat Vs. generation for CGP applied to the Pagie 1 benchmark.

4.7.2 CGPANN

The experiments undertaken for CGP are now presented for CGPANN. In this case, CGPANN is applied to the double pole, ball throwing and Monks Problem 1 benchmarks. The double pole balancing and ball throwing are control reinforcement learning type challenges and the Monks Problem 1 is a classification task; See Appendix A for further details.

The parameters used for the CGPANN experiments are: (1 + 4)-ES, three percent probabilistic mutation, one hundred nodes, ten thousand generations and fifty runs. The arity of each node was set as five and the connection weight range as ± 5. In the case of the double pole balancing the transfer function used was the bipolar logistic sigmoid, Equation 4.2, for the other two benchmarks the regular (unipolar) logistic sigmoid is used.

The results of applying CGPANN to the three benchmarks are given in Figures 4.8 - 4.10. In these figures it can be seen that CGPANN is also not exhibiting program bloat. In each case the level of bloat is low and stable. Again, as for CGP, the level of bloat can be seen to vary slightly, such as in Figure 4.9, but the variation is minor compared to that seen for tree-based GP in Figure 4.5.

4.7.3 Discussion

This section has confirmed the result that CGP does not suffer from program bloat. The analysis of bloat in relation to both fitness and program size used here provides a more rigorous study than previously presented [201]. Additionally, the experiments considered domains other than implementing Boolean circuits. Although the number of benchmarks used was small, the analysis was intended to supplement previous results and provide a comparison for CGPANN.

It was also shown that CGPANN, like CGP, does not suffer from program bloat. Although expected, it is an important result as bloat can be a major drawback of GP methods; resulting in slower run times and excessively large final solutions. Additionally it is thought that larger (bloated) programs generalise less easily.

The fact that CGPANN has been shown here not to suffer from program bloat represents an advantageous property. Other popular NE methods such as NEAT have been previously shown to suffer from program bloat [274]; unless careful consideration is taken with regard to the parameter choices. Additionally, it appears likely that NE methods such as CE, GNARL and EANT would also suffer from program bloat unless additional

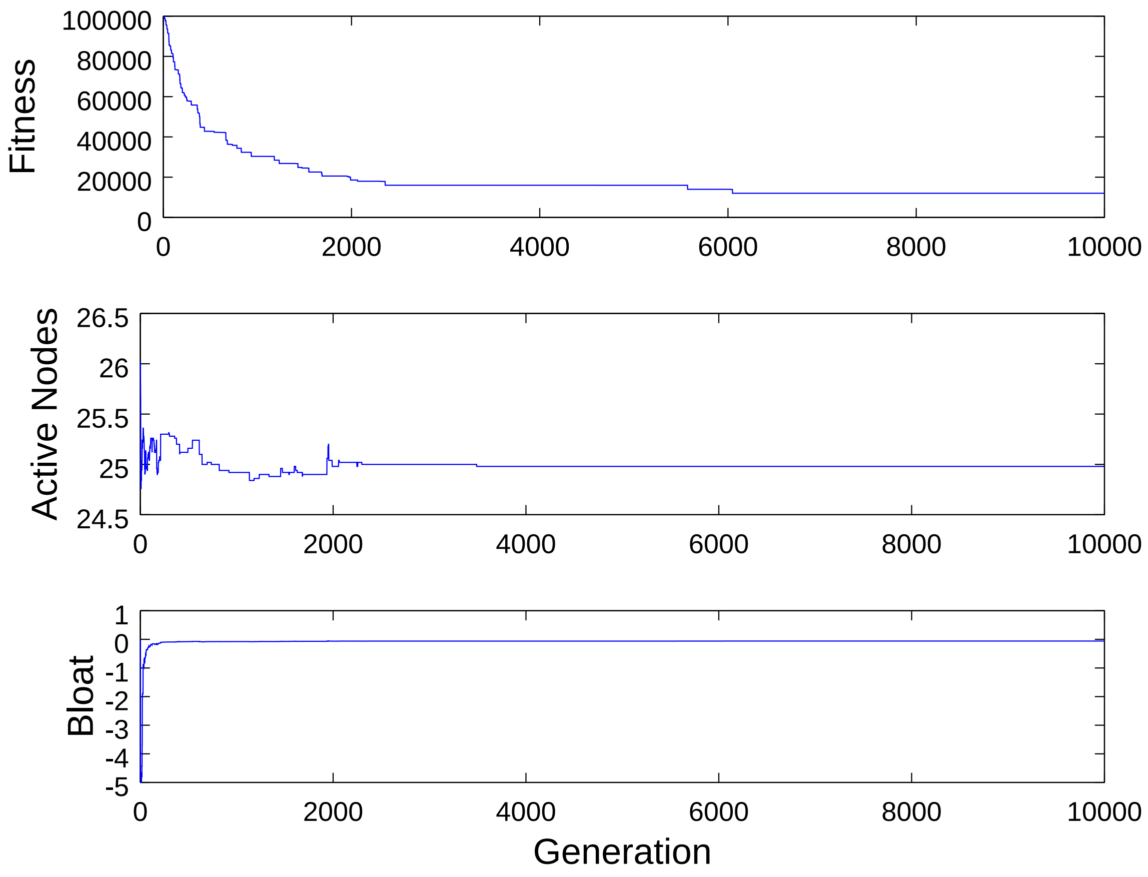


Figure 4.8: Average fitness, number of active nodes and program bloat Vs. generation for CGPANN applied to the double pole benchmark.

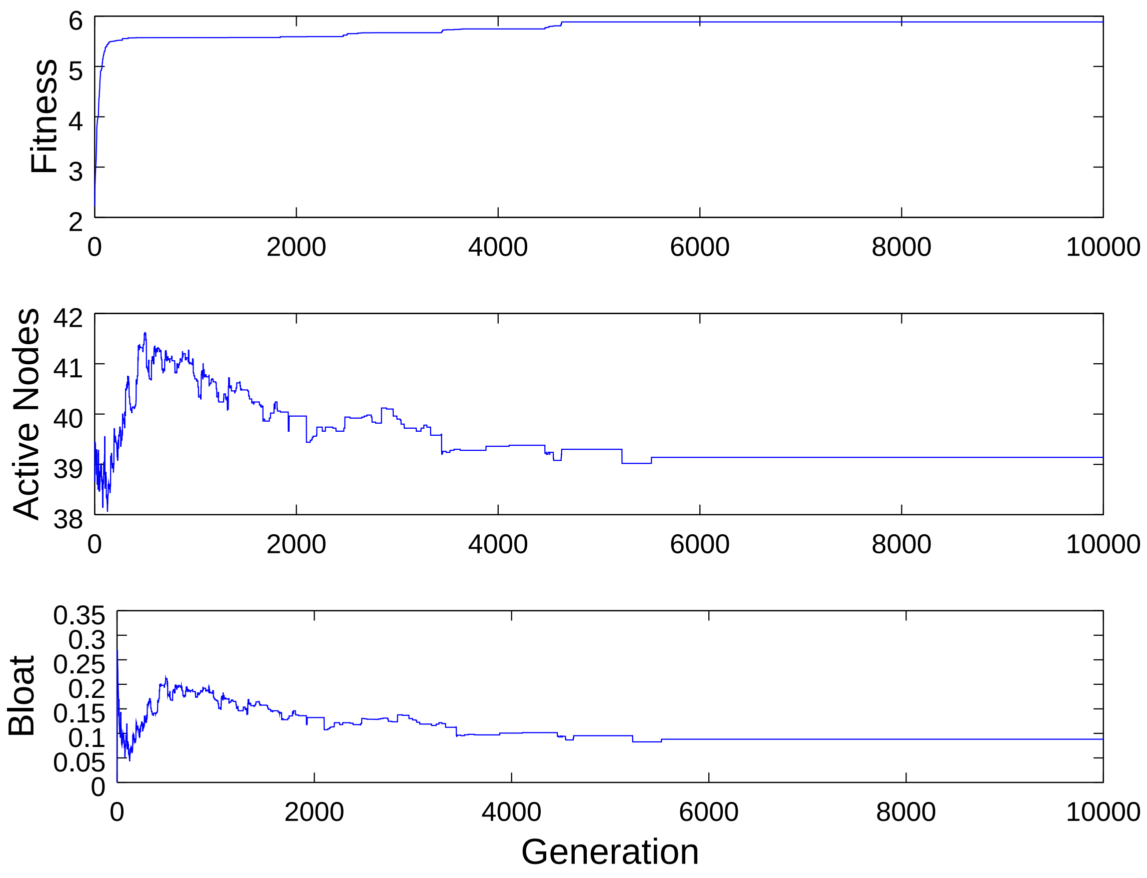


Figure 4.9: Average fitness, number of active nodes and program bloat Vs. generation for CGPANN applied to the ball throwing benchmark.

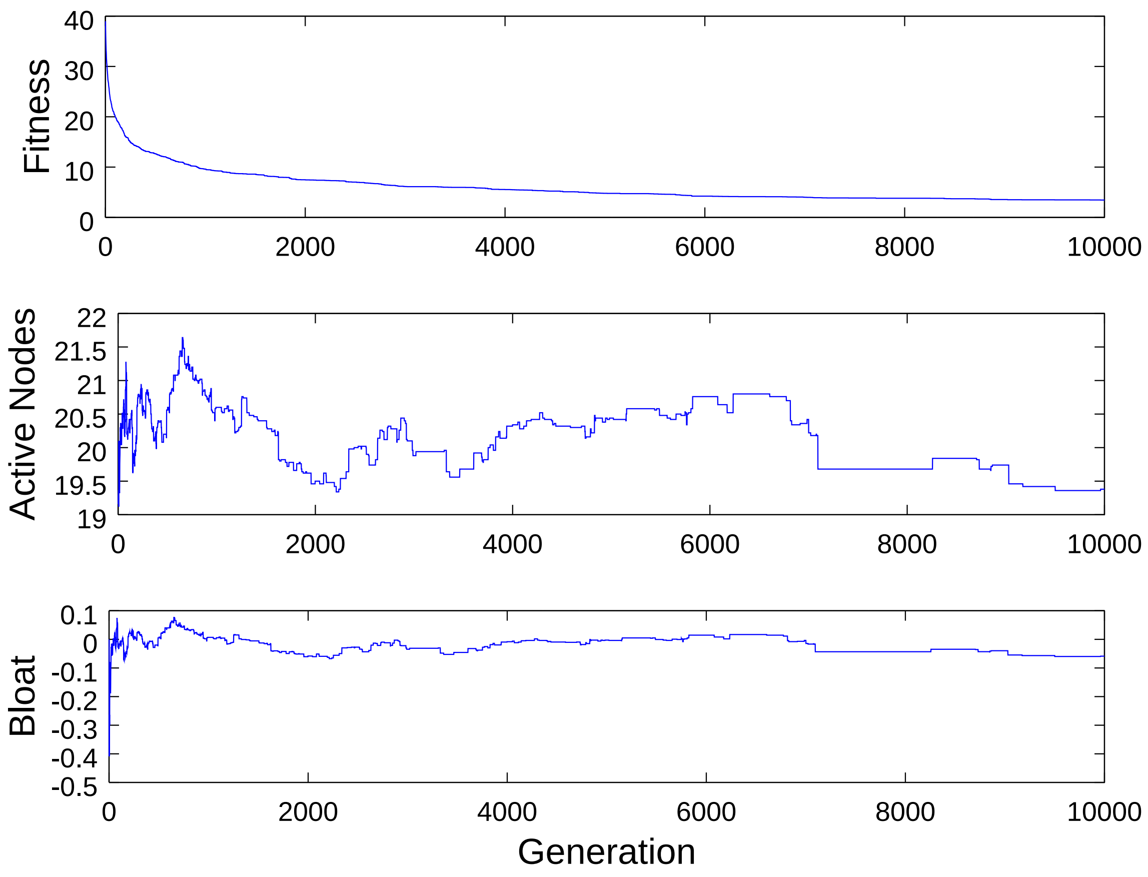


Figure 4.10: Average fitness, number of active nodes and program bloat Vs. generation for CGPANN applied to the Monks Problem 1 benchmark.

prevention methods were taken; although additional research is required to confirm this.

4.8 Summary

This chapter has described CGPANN as used throughout the remainder of this thesis. This description has been contrasted with previous implementations which introduced a new connection switch gene type. Results presented in this Chapter have demonstrated that the previously proposed connection switch gene are not necessary for CGPANN to evolve node arity. It has also been empirically demonstrated that the use of connection switch genes has a slight negative influence on CGPANN's effectiveness. For these reasons they are considered unnecessary and are not used in the work presented in this thesis.

Additionally, CGPANN has been applied to a number of benchmark NE tasks in order to evaluate its performance. As was discussed, previous applications of CGPANN to these tasks provided invalid comparisons due to non-standardised implication of the benchmarks. The results presented here, although less impressive than those presented previously, fairly demonstrate CGPANN as a highly competitive NE method.

A number of possible benefits of using CGP to evolve ANNs have also been discussed. Among these was the resilience to program bloat. This possible benefit was confirmed in Section 4.7. As other popular NE methods such as NEAT have been previously demonstrated to suffer from bloat, and other methods appears very likely to also suffer, this could represent a strong advantage for the use of CGPANN.

Chapter 5

Recurrent Cartesian Genetic Programming

One of the goals of this thesis is to apply Cartesian Genetic Programming of Artificial Neural Networks (CGPANN) to the evolution of Recurrent Artificial Neural Network (RANN)s. The first step to achieving this is to adapt Cartesian Genetic Programming (CGP) to be capable of creating recurrent program structures. This chapter describes a new extension of CGP termed Recurrent Cartesian Genetic Programming (RCGP) which implements this desired extension. The chapter also evaluates RCGP on a number of benchmark tasks demonstrating functionality and investigating the newly introduced parameter *recurrent connection probability.*

5.1 Structure of this Chapter

Section 5.2 provides a background discussion of recurrence in relation to CGP. Section 5.3 describes the newly proposed RCGP extension. Section 5.4 discusses a number of important implications of the new recurrent extension. Section 5.5 presents a number of experiments contrasting CGP and RCGP along with an investigation into the necessity of the new recurrent connection probability parameter. Finally, Section 5.6 gives a closing summary.

5.2 Background

Although RCGP has never been formally presented, it has been previously discussed as a possible extension to CGP: "*The representation of graphs used in CGP is easily adapted to encode cyclic graphs. One merely needs to remove the restriction that alleles for a particular node have to take values less than the position (address) of the node.*" [202].

A form of CGP has also been used in combination with a Jordan type architecture[1] [136] for allowing feedback [155]. In [155] the application was CGPANN [153, 279]. Although using Jordan type architectures represents a simple method for allowing recurrent connections, it does so in a very restricted form. For instance, the user must decide in advance how many and what type of recurrent connections will be used[2].

Additionally, an alternative approach to enabling recurrence in CGP was used in combination with multi-chromosome CGP [294]. Recurrence was created by allowing connection *between* the multiple chromosomes so as to allow simple, restrained, cycles. The application was to utilise CGP to create transistor circuits [294].

Here the main motivation of creating RCGP is so it can later be applied to CGPANN in order to evolve RANNs. However, there are advantages to RCGP in its own right. For instance, as the inclusion of recurrent connections brings a form of memory to CGP, it can be applied to a much wider range of tasks. Such applications include partially observable tasks; those where the inputs do not contain enough information to directly produce the correct outputs. Partially observable tasks require the program to infer additional state information from current *and* previous inputs. Currently standard CGP could not be successfully applied to partially observable tasks whereas RCGP may be capable of inferring the necessary hidden state information by utilising recurrence to implement a form of memory.

5.3 Implementation

In standard CGP, the connection genes ($C_{i,j}$) are restricted so as to only allow acyclic connections i.e. nodes can only connect their inputs to nodes with a lower index i than themselves; where i indexes each node and j each nodes input. In RCGP, this restriction

[1]A Jordan architecture is where one or more program outputs are made available as program inputs.

[2]In some cases choosing the number and type of recurrent connections could be an advantage, for instance if the user knew in advance how many recurrent connections a given task requires. However for more general black box learning, having a fixed number of recurrent connections is likely a disadvantage.

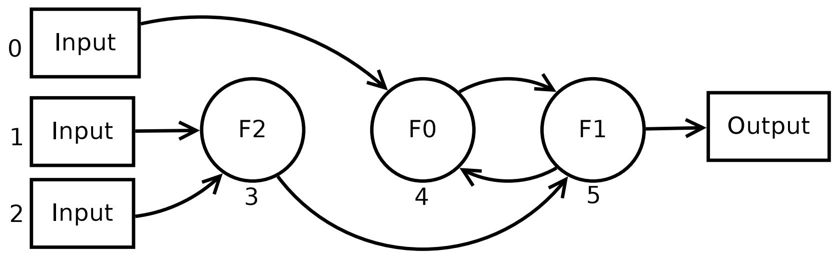


Figure 5.1: Example RCGP program corresponding to the chromosome: $\underline{2}12$ $\underline{0}05$ $\underline{1}34$ 5

is lifted so as to allow connections between a given node and *any* node in the program, including itself, or program inputs. Therefore, connection genes can take any value between zero and the number of inputs plus the number of available nodes. An example program which could be generated using RCGP is given in Figure 5.1 along with the corresponding chromosome. As can be seen in Figure 5.1, the RCGP phenotype contains both feed-forward and recurrent connections. Although not shown in Figure 5.1, RCGP chromosomes, like CGP chromosomes, can also contain inactive genes and do not have to connect to all of the available inputs.

RCGP phenotypes are executed following a similar method to CGP phenotypes. First a set of program inputs are applied. Then, starting at the active node closest to the inputs (low node index i), each node calculates its output value, in turn, based on its inputs. Once all active nodes have been updated, the program outputs are read. However, with the presence of recurrent connections, the output value of a node can be read before it has been calculated. To accommodate this, all nodes are initialised to output a default value until they calculate their own output. The initial output value used throughout this thesis is zero. Therefore when executing a RCGP phenotype the following process is used:

1. set all active nodes to output zero
2. apply the next set of program inputs
3. update all active nodes **once** in index order
4. read the program outputs
5. repeat from 2 until all sets of program inputs have been applied

It is important to note that each node in the phenotype is updated *once* for each set of applied program inputs followed by the output being read. It would also be possible to execute the program multiple times for each set of program inputs. In such a case, the average of the program outputs, or the settled program outputs[3], could be used. The method described here was chosen for its simplicity, speed, and because there is no guarantee that the program outputs would ever settle/converge.

An artefact of placing no constraints on connection gene values is, on average, connection gene mutations will result in as many feed-forward connections as recurrent. As it is unlikely that many tasks will require fifty percent of connections to be recurrent, this places a bias to possibly unsuitable areas of the solution space. For this reason, a new parameter is introduced which controls the likelihood of connection gene mutations creating recurrent connections. This parameter is called *recurrent connection probability.* A recurrent connection probability of zero percent results in only feed-forward connections i.e. regular CGP; of which RCGP is a superset. A recurrent connection probability of fifty percent results in mutations causing as many feed-forward connections as recurrent i.e. RCGP without the new parameter. A recurrent connection probability of ten percent still allows recurrence, but with a bias to a much lower level of recurrence than without using the new parameter. A recurrent connection probability of one hundred percent results in only recurrent connections; which is unlikely to be of use. It should be noted that this parameter does not directly control the number of recurrent connections, only the probability of mutation creating recurrent connections. That is to say, although the recurrent connection probability may be set at a given value, the final solutions can contain more or less that this proportion of recurrent connections; or theoretically none at all.

When creating the initial population for RCGP, the random connection genes values are also chosen considering the recurrent connection probability. That is to say, the randomly generated chromosomes also contain recurrence, with the level of recurrence being determined by the recurrent connection probability.

An important property of CGP is that the active nodes can be determined before executing the program. This is significant as a high proportion of nodes are often inactive [207] and calculating their outputs wastes computation time. To determine which nodes are active the following algorithm is used [202]: 1) add each program output node to a list of active nodes 2) for each node added to the active node list, add the nodes to which they also connect 3) if the inputs are reached, do not add anything to the active node list. Determining the active nodes for RCGP follows a similar algorithm except only nodes

[3]Where settled output refers to the converged program output value(s) after many updates of the active nodes whilst applying the same program inputs.

which are not currently present in the active node list are added. This extra criterion breaks cycles enabling active nodes to be easily determined for RCGP.

5.4 Implications of Recurrent Connections

The introduction of recurrent connections to CGP has a number of interesting implications for the algorithm. These implications are now discussed.

An interesting implication of RCGP, is that it is now possible for chromosomes to describe phenotypes where none of the active nodes connect to the program inputs; See Figure 5.2. These programs are therefore unsuited to any realistic task. However such programs are likely to score a low fitness and be quickly dropped from the population.

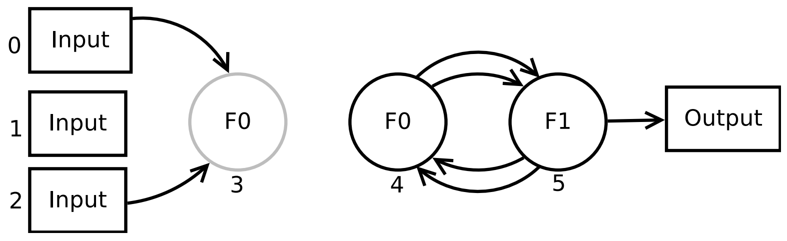


Figure 5.2: Example RCGP program corresponding to the chromosome: $\underline{0}02$ $\underline{0}55$ $\underline{1}44$ 5

Another implication of allowing recurrent connections occurs when applying RCGP to tasks where each set of inputs are unrelated. For example, suppose we are trying to evolve a program that can implement a parity circuit. Typically we think of each line of the truth table as being independent of one another (i.e. the order in which the inputs are applied is unimportant). However, if the fitness function always tests each line of the truth table in the same order, RCGP could, in principle, learn to produce the correct output bits without implementing a parity circuit. This could be achieved by implementing a state machine which just happens to produce the correct outputs when the correct sequence of inputs is applied. Additionally, a quick experiment demonstrated that RCGP could "*solve*" the six bit parity benchmark with a single fixed input of value 1; by implementing such a state machine.

This second implication has further consequences for fitness function design. For instance, it may appear that a simple method for preventing RCGP learning to "predict" the correct outputs sequence would be to randomise the order in which the inputs are applied. However, take again the parity task, if in a given population one recurrent solution *happens* to produce a state machine which scores well given its *random* input ordering, this solution is not in fact a good parity circuit (which is strictly non-recurrent) but does score well. As CGP typically uses an elitist $(1+\lambda)$-ES, this high scoring but poor solution will be retained. Now all of the produced children, even if they are identical, are likely to perform worse because the order of applied inputs will be different (random). This means that not only is the poor solution retained, but it pollutes subsequent child solutions.

One possible solution to this issue is to assess the fitness of solutions using multiple random input orderings. This method sacrifices convergence time (longer fitness evaluations) for diminishing risk of this issue occurring. An alternative solution is to use a non-elitist (μ, λ)-ES. However it is known that CGP is much more efficient using a $(1+\lambda)$-ES than a (μ, λ)-ES [202]. Additionally, this means that genuinely good solutions can be easily lost.

It is therefore highly important that RCGP should only be applied to tasks where the series of inputs are *related*, such as in time series prediction or control tasks, otherwise additional precautions are required to prevent this undesirable behaviour. However, in most cases it is known in advance whether a recurrent solution may be of benefit i.e. classification does not require recurrence whereas a control system may.

5.5 Experiments

A number of experiments are now presented which investigate the newly proposed recurrent CGP extension. The experiments have three aims. Firstly, to investigate whether or not RCGP outperforms standard acyclic CGP on tasks specifically designed to benefit from recurrence. This is to demonstrate that RCGP is capable of utilising the ability to create recurrent solutions. Secondly, to investigate the effect of varying the recurrent connection parameter. This is investigated with the aim of determining whether the new recurrent connection parameter is necessary, and to what extent it affects the search. As with many Evolutionary Algorithm (EA) methods, the presence of many parameters is both beneficial and detrimental; it allows fine control, but often suitable values are not known. Therefore the addition of new parameters should be justified. Finally, the solutions created by RCGP are inspected to gain insight into the evolved structures and to confirm that recurrence is being utilised.

5.5.1 Experiment 1

This first experiment has two roles. Firstly, is to demonstrate that RCGP can successfully utilise recurrent connections on tasks which benefit from their inclusive. This is achieved by comparing the performance of CGP and RCGP on tasks which require recurrent connections. Secondly, it is to investigate the effect of varying the recurrent connection probability in order to determine if the new parameter is beneficial, or indeed necessary.

5.5.1.1 Set Up

In this experiment CGP and RCGP both use typical CGP parameters: $(1+4)$-ES, 10,000 generations, 50 runs, 3% probabilistic mutation, one hundred available nodes each with a node arity of two. The recurrent connection probability will be swept over the following percentages [0,5,10,20,30,40,50]; with 0% representing standard acyclic CGP. The benchmarks used for the experiments and the associated function sets are described in Section 5.5.1.2.

If RCGP achieves statistically significantly better fitness than CGP on the given tasks, then it will demonstrate that RCGP is a suitable extension to CGP when applied to tasks which require recurrent connections. Additionally, if any value of recurrent connection probability produces a statistically significantly better fitness than fifty percent, it will demonstrate that the new parameter is a suitable method of controlling the bias of recurrence. This is because RCGP without the recurrent connection probability parameter is equivalent to using RCGP with a recurrent connection probability of fifty percent. Therefore, if any value of recurrent connection probability produces a better search than fifty percent it shows that this new parameter can positively influence the search.

5.5.1.2 Benchmarks

For this experiment two benchmarks are chosen which are specifically designed to benefit from recurrent connections, namely Artificial Ant and a modified version of Sunspots forecasting. This is to force any ability to create effective recurrent solutions to be highlighted.

The Artificial Ant problem [134] is a classic, challenging [177], benchmark commonly used by Genetic Programming (GP) [165]. The task is to design a controller which navigates an ant around a toroidal Cartesian map maximising food intake. The ant can only perceive whether the location ahead of is current position contains food. Each time step the ant undertakes one of four actions: move forward, turn left 90°, turn right 90° or do

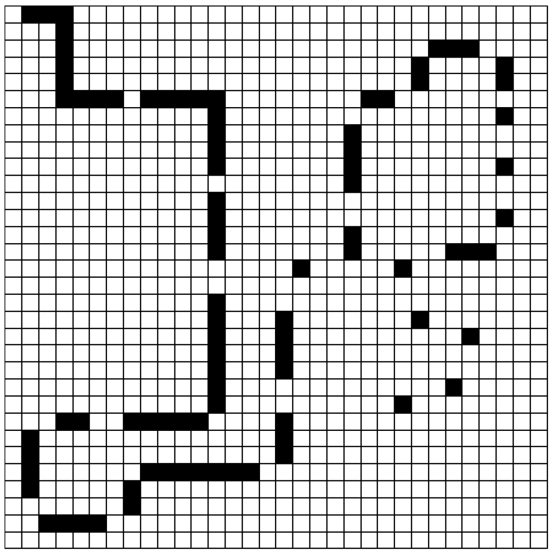


Figure 5.3: Depiction of the "Santa Fe Ant Trail". Black and white represents food and no food respectively.

nothing. If the ant occupies the same position as a piece of food that food is "eaten" and removed from the map. The map used here is the "Santa Fe Ant Trail" [165] given in Figure 5.3.

Here the form of the controller differs from that commonly used by GP [165]. The evolved program's inputs describe if the location ahead contains food and the program's outputs are decoded into one of the possible four actions; this is interestingly not dissimilar to its original implementation [134]. Other GP implementations [165] create programs where the program inputs are the possible actions and the program outputs are unused. The function set used by the nodes causes the inputs (actions) to either be implemented outright or to be conditional on whether food is ahead. Once the program outputs are reached the program starts over[4]. CGP has also previously been applied to the benchmark in its more commonly used form [208].

Here, the evolved controllers have two mutually exclusive inputs, whereby the first input is set as '1' if the location ahead of the ant contains food, else it is set as '0'. The controller has two outputs, where: [1 1] is decoded as move forward, [0 1] as turn right, [1 0] as turn left and [0 0] as do nothing. The ant starts in the top left $(0, 0)$ of the toroidal map facing east and is allowed 400 time steps to consume as much food as possible. The amount of food eaten is then used as the fitness measure; out of a maximum 89. The

[4]This way of approaching the artificial ant problem appears to be a rather inelegant way of accommodating the fact that standard tree-based GP cannot create state machine or have any form of memory.

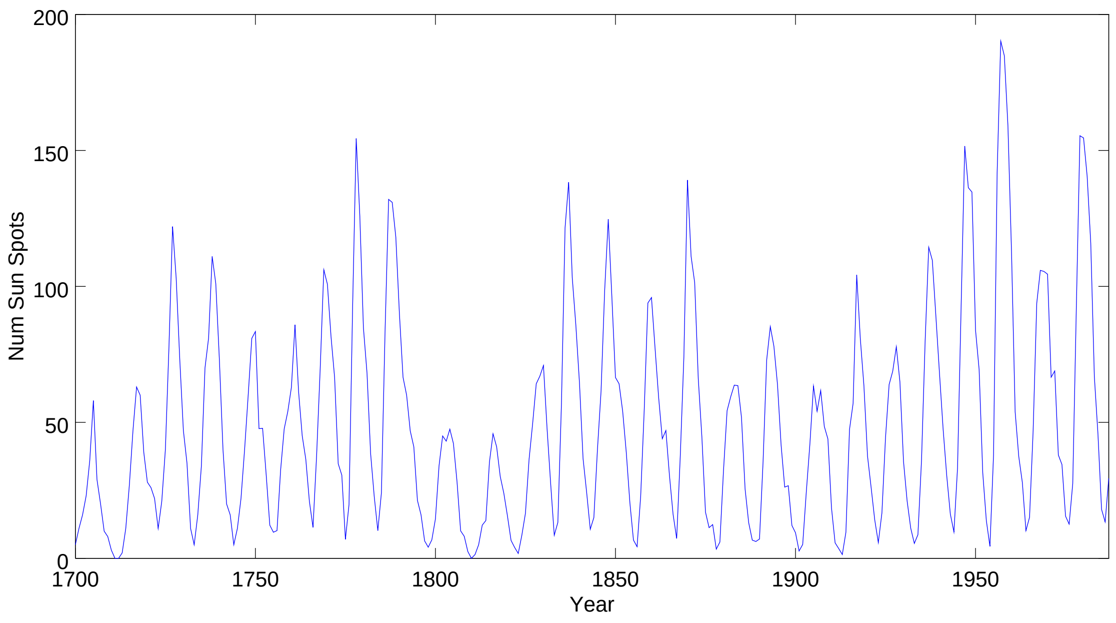


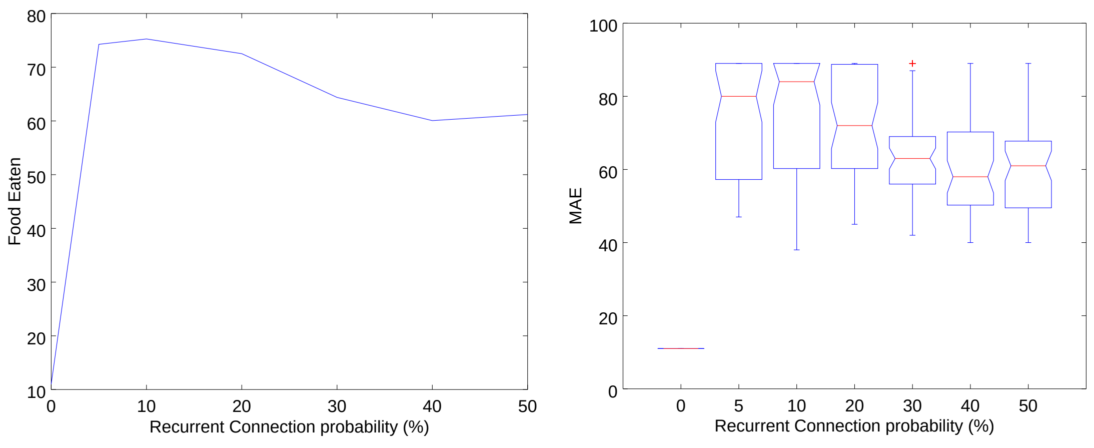
Figure 5.4: Number of yearly recorded sunspots between 1700 and 1987.

function set used comprises of the four Boolean logic gates: AND, OR, NOT, and XOR.

The Sunspots benchmark [253] is a commonly used [157] time series prediction benchmark which describes the number of observed sunspots dating back to 1700. The data was recorded by the SIDC-team, at the World Data Center for the Sunspot Index, Royal Observatory of Belgium [253]. The dataset contains the yearly number of recorded sunspots between 1700 and 1987; given in Figure 5.4. The first 221 years (1700-1920) are used as the training set with the remaining 67 years (1921-1987) used as the testing set.

Most series forecasters which are applied to the Sunspots benchmark use multiple inputs consisting of the current and previous years number of sunspots. However, here only one input is used which gives the current number of sunspots. This restriction to one input is imposed to force the task to become less tractable without internal recurrence. This restriction also makes the task much more challenging since any trends in the data must be calculated internally as the data is passed in year by year. The single output is the predicted number of sunspots 35 years ahead of the current input. The single input to the series forecaster is normalised into a $[0, 1]$ range by dividing by two hundred; a value greater than the highest number of sunspots in any observed year. The single output is also multiplied by two hundred before being used as the predicted number of sunspots.

The fitness measure is the Mean Absolute Error (MAE) given by: $\frac{1}{N}\sum_{i=1}^{N}|e_i|$ where N is the number of samples and e is the difference between the actual and predicted number of sunspots. The function set used for this task comprises of ten symbolic expressions:



(a) Average Fitness (b) Box and Whisker Plots

Figure 5.5: Results of varying RCGP's recurrent connection probability on the Artificial Ant benchmark.

$x_1 + x_2$, $x_1 - x_2$, $x_1 \times x_2$, $x_i \div x_j$, $|x_1|$, x_1^2, x_1^3, e^{x_1}, $\sin(x_1)$ and $\cos(x_1)$. Where x_1 and x_2 are the two inputs to each node and the division operator is protected so as to return one when dividing by zero.

5.5.1.3 Results

The results of the described experiment on the two benchmarks are now presented. In each case the average fitness verses the recurrent connection probability is plotted. This gives a high level view of the results. Additionally, the spread of the data is also plotted as box and whisker plots for a more detailed view of the data; with outliers marked as follows: '+' represents fitnesses between 1.5 and 3 times the interquartile range and '∘' represents fitnesses greater than 3 times the interquartile range. Finally, the pairwise differences between using each set of recurrent connection probability is analysis for statistical significance using the non-parametric Mann-Whitney U-test with $\rho \leq 0.05$ representing statistical significance.

The average fitness achieved, and the spread of fitnesses, are given in Figure 5.5 for the Artificial Ant benchmark; where a higher fitness represent a better solution. As can be clearly seen in Figure 5.5, allowing recurrent connections consistently produces a much better fitness than not; where zero percent recurrent connection probability represents no recurrence i.e. standard acyclic CGP. Additionally, it can be seen that the fitness achieved varies substantially with varying levels of recurrent connection probability.

The statistical analysis of the results given in Figure 5.5 are given in Table 5.1. As

Table 5.1: Artificial Ant: p values comparing pairs of recurrent connection probabilities.

	0%	5%	10%	20%	30%	40%	50%
0%	1	∼0	∼0	∼0	∼0	∼0	∼0
5%	-	1	8.85E-1	3.79E-1	4.78E-3	4.97E-6	8.83E-5
10%	-	-	1	2.60E-1	1.47E-3	1.38E-6	2.42E-5
20%	-	-	-	1	9.22E-3	3.53E-5	2.63E-4
30%	-	-	-	-	1	6.86E-2	1.76E-1
40%	-	-	-	-	-	1	6.29E-1
50%	-	-	-	-	-	-	1

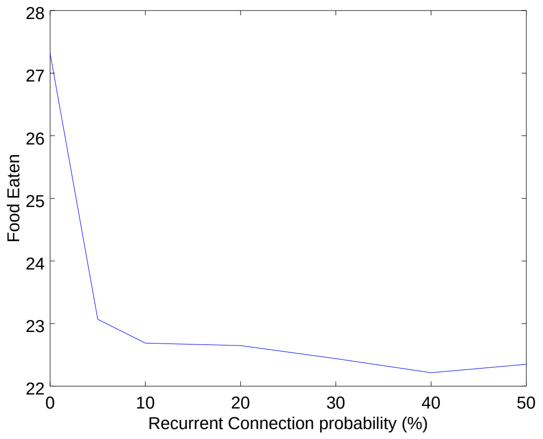
can be seen in Table 5.1, the difference between CGP and RCGP is statically significant, diminishing small p values, for all levels of recurrence investigated. Additionally, it can be seen that many values of recurrent connection probability produced a statistically significantly better fitness score than using a fifty percent recurrent connection probability.

The results of the Sunspots benchmark follow the same format as for the Artificial Ant benchmark, except for the presence of both training and testing performance. Although here we are only concerned with differences caused by the presence of recurrence, and not generalisation performance, both training and testing are presented for completeness. Later, in Chapter 10, RCGP is rigorously assessed as a forecasting method where generalisation is the overall measure of merit.

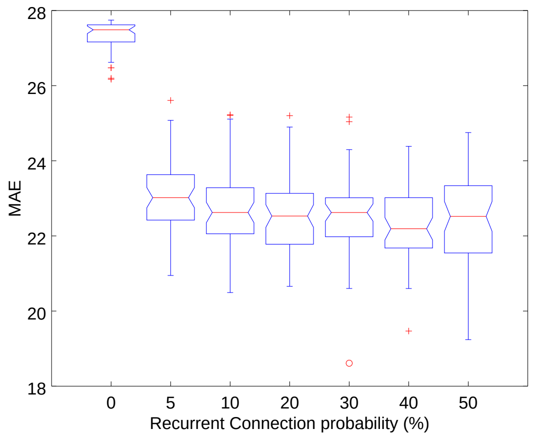
The average training fitness achieved, and the spread of training fitnesses, are given in Figure 5.6 for the Sunspots benchmark; where a lower MAE represents a better solution. As can be clearly seen in Figure 5.6, allowing recurrent connections consistently leads to a much superior fitness than not allowing recurrent connections. Additionally, it can be seen that the fitness achieved varies with the level of recurrent connection probability; although not to the same extent as for the Artificial Ant benchmark.

The statistical analysis of the results given in Figure 5.6 are given in Table 5.2. It can be seen in Table 5.2 that the difference between CGP and RCGP is statistically significant for all levels of recurrence investigated. Additionally it can be seen that one value of recurrent connection probability, five percent, produced a statistically significantly better fitness score that using a fifty percent recurrent connection probability. There were also many instances of statistically significant differences between other levels of recurrent connection probability.

The average testing performance on the Sunspots benchmark is given in Figure 5.7 with the statistical analysis given in Table 5.3. It can be seen that there is still a large, statistically significant, difference between CGP and RCGP when assessed on the testing



(a) Average Fitness

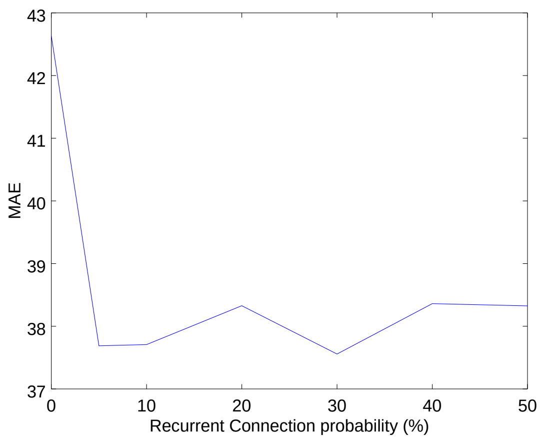


(b) Box and Whisker Plots

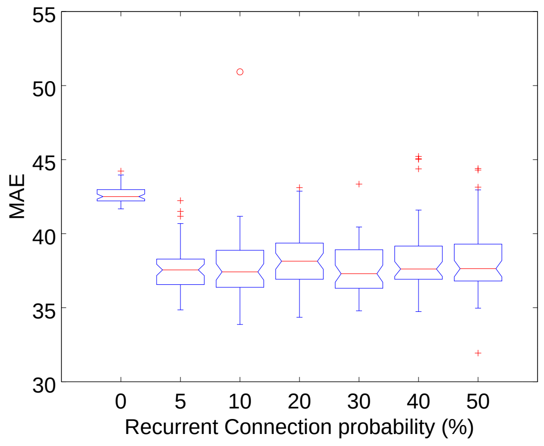
Figure 5.6: Results of varying RCGP's recurrent connection probability on the Sunspots benchmark: training data.

Table 5.2: Sunspots training fitness: p values comparing pairs of recurrent connection probabilities.

	0%	5%	10%	20%	30%	40%	50%
0%	1	∼0	∼0	∼0	∼0	∼0	∼0
5%	-	1	4.99E-2	1.89E-2	2.68E-3	4.04E-5	5.76E-3
10%	-	-	1	6.72E-1	3.76E-1	3.23E-2	3.13E-1
20%	-	-	-	1	6.62E-1	1.39E-1	4.59E-1
30%	-	-	-	-	1	2.37E-1	8.71E-1
40%	-	-	-	-	-	1	3.91E-1
50%	-	-	-	-	-	-	1



(a) Average Fitness



(b) Box Plots

Figure 5.7: Results of varying RCGP's recurrent connection probability on the Sunspots benchmark: testing data.

Table 5.3: Sunspots testing fitness: p values comparing pairs of recurrent connection probabilities.

	0%	5%	10%	20%	30%	40%	50%
0%	1	∼0	∼0	∼0	∼0	∼0	∼0
5%	-	1	7.12E-1	8.42E-2	5.74E-1	3.16E-1	2.81E-1
10%	-	-	1	5.40E-2	8.82E-1	1.37E-1	1.69E-1
20%	-	-	-	1	3.34E-3	6.08E-1	6.47E-1
30%	-	-	-	-	1	7.70E-2	1.39E-1
40%	-	-	-	-	-	1	9.53E-1
50%	-	-	-	-	-	-	1

data. However, there is no statistically significant difference between fifty percent recurrent connection probability and any other levels of recurrence. There are still however two instance of statistically significant difference between other levels of recurrence.

5.5.1.4 Discussion

One of the goals of the experiment presented is to determine if RCGP could successfully utilise the available recurrent connections. This was investigated by comparing standard acyclic CGP and RCGP on a number of tasks which required, or would greatly benefit from, recurrence in the solutions.

For both of the benchmarks investigated it can be seen that RCGP drastically, and statistically significantly, outperformed CGP. This demonstrates that RCGP is capable of effectively using the available recurrent connections.

An additional goal was to investigate whether the proposed recurrent connection probability is a suitable, or indeed necessary, new parameter for biasing the level of recurrence. Again as can be seen in the results, for both the Artificial Ant and Sunspots, there were levels of recurrent connection probability which outperformed a fifty percent recurrent connection probability; with statistical significance. This demonstrates that the level of recurrence present without the use of a recurrent connection probability is not necessarily suitable. This in turn demonstrates that the use of the new recurrent connection probability is a useful addition to RCGP for biasing the level of recurrence.

5.5.2 Experiment 2

The second part of the experimental investigation is to inspect solutions found using RCGP to see if any insights can be gained. The experiments also continue the comparison between CGP and RCGP, but now to applications which can be solved both with and without recurrence. Finally, a small comparison is also presented between RCGP and other GP methods capable of creating recurrent program structures.

5.5.2.1 Set Up

In this experiment CGP and RCGP will both use the following parameters: (1 + 4)-ES, 1,000,000 generations, 50 runs, 5% probabilistic mutation, 20 nodes each with a node arity of two. In the case of RCGP the recurrent connection probability is set as 10%.

The number of nodes is kept low, despite larger numbers likely producing better results [207], so the final solutions can be more easily inspected. For instance reasoning about solutions comprising up to 100 nodes (with recurrent connections) is far more challenging than for 20 nodes.

5.5.2.2 Benchmarks

The benchmarks chosen for this investigation are all concerned with producing explicit or recurrent symbolic equations which predict famous mathematical sequences. This is an interesting comparison for CGP and RCGP as all the sequences chosen have both explicit and recurrent forms[5].

When using CGP, the single input to the phenotypes is n and the expected output is the n^{th} value in the sequence. When using RCGP the single input is fixed at the value of one and the phenotype is updated multiple times to produce a sequence of numbers. When the phenotype is updated n times it should produce, in order, the first n values in the sequence. It would also be possible to use RCGP and input the value n instead of the constant one. In this case RCGP could produce explicit as well as recurrent solutions. Here however, RCGP is forced to produce recurrent solutions to a) exaggerate any differences between CGP and RCGP and b) so that the inspected solutions are more likely to contain recurrence.

The function set used by both CGP and RCGP contains: addition, subtraction, multiplication and protected division.

The mathematical sequences used for comparison comprise: hexagonal numbers, the lazy caterers sequence, the magic constants and the Fibonacci sequence. Each of these is

[5]Mathematical sequences can be defined in two forms, either explicitly or recursively. An explicit equation returns the n^{th} value in a sequence when passed the value of n. A recursive equation returns the n^{th} value in a sequence upon its n^{th} iteration.

now introduced.

The Hexagonal number sequence, A000384 from [259], is the number of evenly distanced dots which make up a sequence of hexagons and all the hexagons it contains; see Figure 5.8. It is defined explicitly by Equation 5.1 where $n \geq 1$. This produces the following sequence: 1,6,15,28,45,66,91,120,153,190,...

$$y(n) = \frac{2n(2n-1)}{2} \tag{5.1}$$

The Lazy Caterers Sequence (or more formally the central polygonal numbers), A000124 from [259], is the number of pieces a cake can be divided into with n cuts. The sequence is shown graphically in Figure 5.8 and described explicitly by Equations 5.2; where $n \geq 0$. This produces the following sequence: 1,2,4,7,11,16,22,29,37,46,...

$$y(n) = \frac{n^2+n+2}{2} \tag{5.2}$$

The sequence of Magic Constants, A006003 from [259], are the minimum values each row, column and diagonal of a $n \times n$ magic square[6] can sum to. The magic squares corresponding to n=3, 4 and 5 are given in Figure 5.8. The sequence is described explicitly by Equations 5.3; where $n \geq 1$. This produces the following sequence: 1,5,15,34,65,111,175,260,369,505,...

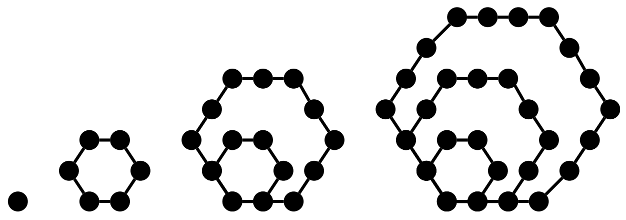
$$y(n) = \frac{n(n^2+1)}{2} \tag{5.3}$$

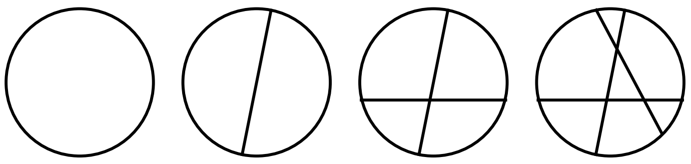
The Fibonacci sequence, A000045 from [259], is such that each value is the sum of the previous two values; with the first two values set as one. The sequence is described explicitly in Equation 5.4, but is more commonly given recursively such as in Equation 5.5. This produces the following sequence: 1,1,2,3,5,8,13,21,34,55,...

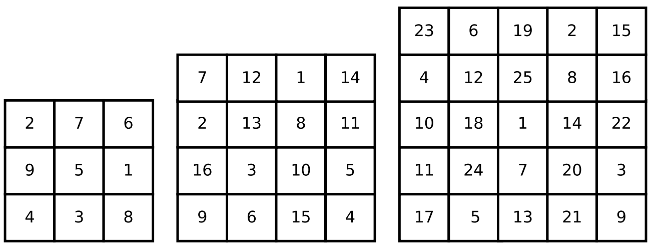
$$y(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}} \tag{5.4}$$

$$y(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ y(n-1) + y(n-2), & \text{otherwise} \end{cases} \tag{5.5}$$

[6]A magic square is an $n \times n$ grid of numbers where the sum of each row, column and diagonal are equal.


(a) Hexagonal Numbers (n=1,2,3,4)


(b) Lazy Caterers Numbers (n=0,1,2,3)



(c) Magic Constants (n=3,4,5)

Figure 5.8: Number sequences shown graphically.

5.5.2.3 Results

The results of applying CGP and RCGP to the mathematical sequences benchmarks are given in Table 5.4. The statistical significance between CGP and RCGP is also given using the non-parametric Mann-Whitney U-test with $\rho \leq 0.05$ representing statistical significance.

Table 5.4: Performance of CGP and RCGP finding explicit and recurrent equations respectively which produce famous mathematical sequences. In each case the average number of evaluations is given followed by the number of runs which successfully solved the task in brackets.

Sequence	CGP	RCGP	U-Test (p)
Hexagonal	2,481 (50/50)	39,279 (50/50)	6.15E-13
Lazy Caterers	- (0/50)	7,626 (48/50)	-
Magic Constants	557,592 (50/50)	686,929 (43/50)	7.25E-1
Fibonacci	- (0/50)	27,075 (50/50)	-

A number of solution found using CGP and RCGP are now presented. In all cases the functioning nodes are labelled with their node index and their function. When executing the programs, the nodes are updated in index order. The inputs to each node are also labelled with their input ordering. This is significant in the case of division and subtraction. For the divide function, the input labelled (0) is the numerator and the input labelled (1) is the denominator. For the subtract function, the input labelled (1) is subtracted from the input labelled (0).

Two example solutions found for the hexagonal benchmark are given in Figure 5.9 for CGP and RCGP respectively. As no lazy caterers solutions were found using CGP only a solution for RCGP is presented in Figure 5.10. Two example solutions found on the magic constants benchmark are given in Figure 5.11 for CGP and RCGP respectively. Finally as no Fibonacci solutions were found using CGP therefore only a solution for RCGP is presented in Figure 5.12.

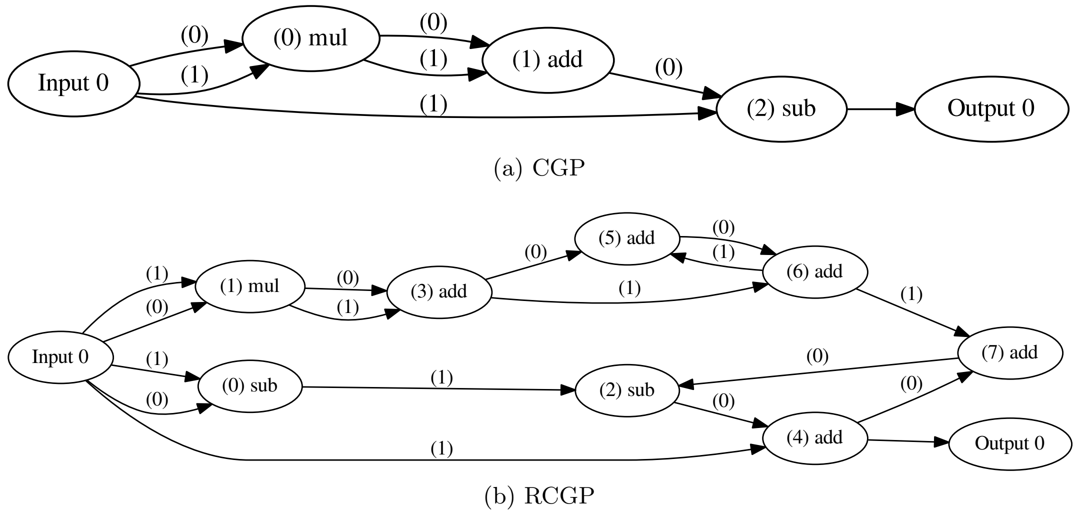


Figure 5.9: Example CGP (a) and RCGP (b) Hexagonal solutions.

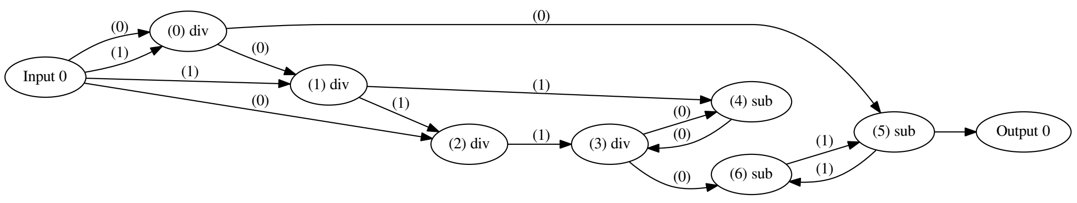
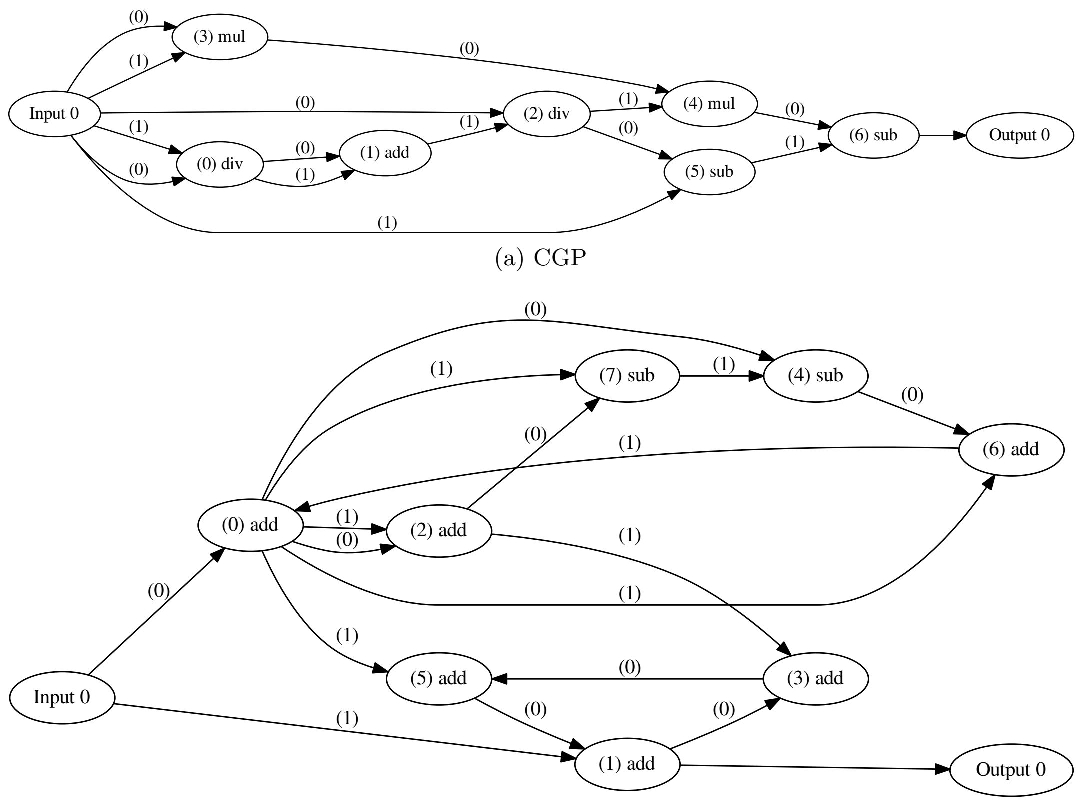


Figure 5.10: Example RCGP Lazy Caterer solutions.



(a) CGP

(b) RCGP

Figure 5.11: Example CGP (a) and RCGP (b) Magic Constants solutions.

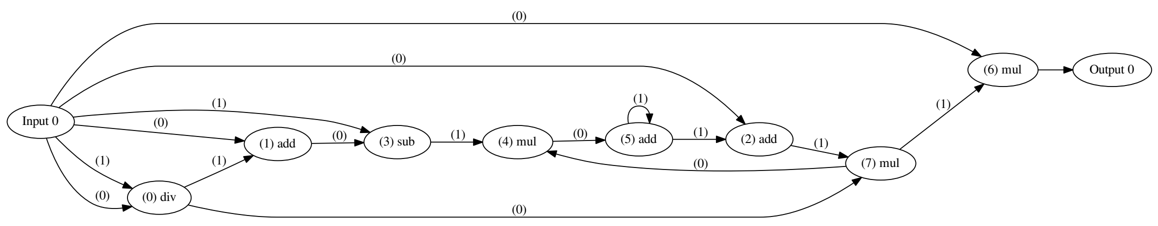


Figure 5.12: Example RCGP Fibonacci solution.

Although the aim of this chapter is not to compare RCGP to other methods capable of creating recurrent solutions, as other GP methods have also been previously applied to the Fibonacci sequence a comparison can easily be made. However, it should be noted that the implementations used vary between methods e.g. the length of the sequences used, the use of training and testing sets, and the percentage of runs which found a solution. Therefore only a superficial comparison can be made. The results of this comparison are given in Table 5.5 where RCGP is shown to be very competitive. Interestingly the comparative methods include an alternative extension to CGP, Self Modifying Cartesian

Genetic Programming (SMCGP)[7].

Table 5.5: GP methods applied to the Fibonacci Sequence benchmark

Method	Evaluations
RCGP	27,075
Multi-niche Genetic Programming [216]	∼200,000
Probabilistic Adaptive Mapping Developmental GP [307]	212,000
SMCGP [100]	∼1,000,000
Machine Language Programs [122]	∼1,000,000
Object-Oriented GP [3]	∼20,000,000

5.5.3 Discussion

As can be seen from the results in Table 5.4, using CGP to find explicit solutions and RCGP to find recurrent solutions has a marked effected on the tractability of the tasks. In all cases RCGP found recurrent solutions in the majority of runs, whereas CGP only found solutions for two of the four sequences. This again demonstrates that there are tasks to which RCGP is more suited than CGP; even when CGP is capable of solving the task. It is likely however that the performance of CGP would have been improved if given more available nodes and/or number of generations; although this is likely also be the case for RCGP.

In the case of the hexagonal sequence, CGP strongly outperformed RCGP with statistical significance. This demonstrates that although RCGP could solve this task, CGP was more suited to the challenge. That is to say, it appears that the task is more easily solved explicitly than recursively. As the set up for these experiments forced RCGP to produce recurrent solutions, by fixing the single input as one, it had so be solved recursively rather using the easier explicit form. In general RCGP can evolve solutions with or without recurrent connections and so this is not a limitation of RCGP. In fact if it were not known whether recurrence was required, or whether the task was more tractable with/without recurrence, RCGP could be applied and allowed to follow the easiest evolutionary gradient; resulting in an explicit or recurrent solution. Although whether this would actuality occur is speculative.

From inspection of the RCGP solutions, it was noticed that many solutions contained addition nodes with their outputs fed back as an input; such as node (5) in Figure 5.12.

[7]Although RCGP and SMCGP are described here as alternative extensions, they may actually be compatible with one another. This would create a recurrent form of SMCGP.

As all nodes are initialised to output zero before they calculate their own output value, this has the effect of implementing a summation; where the output of node (5) is the running sum of all the previous outputs of node (4). This interesting behaviour also holds for subtraction. However it does not hold for multiplication as the node is also initialised to output zero; if a multiplication node's output were fed back as an input it would forever output zero. It is therefore possible that simpler recurrent equations could be formed if multiplication nodes could be used to store the product of previous inputs; akin to how addition nodes store the summation. This can be achieved by initialising multiplication nodes to output *one*, not zero, until they have calculated their own output value. Then multiplication nodes could be arranged such that they produce the product of previous inputs. It is therefore recommended that future developments of RCGP consider what initial output values different node functions should use to make best use of their presence. Alternatively, the value to which each node is initialised could also be determined by the evolutionary process.

Being capable of implementing summation operations indicates additional possible applications of RCGP. For instance, a summation operation is the discrete equivalent of an integral in the continuous domain. Additionally, RCGP is likely also capable of implementing a moving averaging function; for instance by keeping a sum of the previous two inputs and dividing the sum by two. Again calculating a moving average in the discrete domain has symmetries with calculating the gradient in the continuous domain. It therefore appears that RCGP could create equations which can calculate gradients and integrals; or at least approximations. This may mean that RCGP can be applied to tasks where the solution required is a differential equation.

Finally, it was shown that RCGP represents a highly competitive method of creating equations which produce the Fibonacci sequence; outperforming all other methods used for comparison. Although in this chapter only a single comparison was made to other methods, this at least indicates that RCGP is a powerful method for creating recurrent program structures.

5.6 Summary

This chapter has introduced a new extension to CGP which enables the creation of recurrent program structures. In both sets of experiments presented it has been shown that RCGP is capable of utilising recurrent connections and that for certain tasks this provides a strong advantage over CGP. It was also shown that RCGP represents a powerful method compared to other GP techniques capable of creating recurrent solutions; although the comparison was limited.

From the inspection of evolved solutions in can be seen that RCGP does indeed create recurrent solutions. It was also noted that many solutions contained addition nodes with the output fed back as an input, thus implementing a summation. As this was only possible due to the initial value used, the choice of initial values should be a consideration in future works.

As RCGP was introduced here solely so it can later be applied to CGPANN, there are many open questions left uninvestigated. For instance, it is not known how strongly the bias produced by the recurrent connection probability influences the search and final solutions. For example, it may be the case that using a recurrent probability of x actually produces final solutions with x percent recurrent connections regardless of application; in which case the bias is too strong overcoming other evolutionary pressures. Additionally, it may be the case that many tasks can be solved more efficiently, in terms of the number of nodes, when recurrence is allowed. If this were the case then tasks could be solved with fewer nodes reducing the dimensionality of the search, possibly leading to increased convergence time. However, it may also be the case the presence of recurrence creates more challenging fitness landscapes increasing the difficulty of the search. Finally, it is not yet known what value of recurrent connection probability is likely to be suitable. Although certainly task dependent, it may be the case that for the majority of tasks a value around x is reasonable. For instance, when selecting a mutation value for CGP a value between 1% and 10% would be considered responsible; whereas a value <0.1% or >20% would be considered strange. This intuition is not yet available for the recurrent connection probability.

Chapter 6

Topology Evolution

One of the often stated benefits of NeuroEvolution (NE) is the ability to evolve the topology of Artificial Neural Network (ANN). However, in the literature there is no rigorous study assessing whether evolving topology actually provides any benefit. This chapter aims to fill this void by investigating whether the ability to adapt network topology does indeed represent an advantageous property for NE. This investigation also includes an assessment of the relative importance/benefit of evolving network topology compared to connection weights.

6.1 Structure of this Chapter

Section 6.2 provides a background to topology evolution in the field of NE with a focus on the perceived benefits of topology adaptation. Section 6.3 presents an experiment investigating whether the ability to adapt topology does indeed provide a benefit for topology evolving NE methods. Section 6.4 continues by investigating the relative importance of topology and connection weight evolution. Finally Section 6.5 presents a closing summary of the investigations.

6.2 Background

NE methods are often categorised into two groups, those which do, and those which do not, adapt network topology [64, 313]; to the authors knowledge all NE methods adapt connection weights. Theoretically, training an ANN can be thought of in terms of searching a topology and weight space; or a weight spaces associated with each given topology. Using only fixed topologies therefore limits the search to one subset weight space within the wider search space.

It is often assumed in the NE literature that the ability to evolve topology offers an advantage over only weight adapting methods [64,313]; such as back propagation and NE method which only adapt connection weights. These advantages include 1) not requiring a suitable topology to be known in advance of training[1], 2) utilising topologies which would be unlikely to be considered by a human designer, and 3) exploiting relationships between topology and connections weights during evolution.

Although it is often thought that topology and connection weight evolution offers a significant advantage over weight evolution alone, to the authors knowledge there are no publications which truly compare the two approaches. Initial work in [130] present a limited set of results on varying the topology for Covariance Matrix Adaptation Evolutionary Strategies (CMA-ES); a weight only evolving NE method. In such work it was shown that the choice of topology did strongly influence the results founds. Additionally, it is also known that the choice of topology has a large influence on the performance of ANNs trained using back propagation type algorithms [178]. However, this prior work alone is insufficient to truly assess the benefit of topology evolving NE methods.

It may be possible to assess the benefit of evolving topology by comparing results of NE methods which do and do not evolve network topology found in the literature. However, it is difficult to draw empirical conclusions about the benefit of evolving topology as techniques use different Evolutionary Strategies (ES) and different encodings to describe the ANNs during evolution. When, for example, a weight and topology evolving NE technique outperforms a weight only technique, we may assume that the increase in performance was due to its ability to evolve topologies, but it could equally be due to other implementational differences. For instance, the weight only evolving method Symbiotic Adaptive NeuroEvolution (SANE) [212] evolves ANNs at a neuron level, with the complete networks assembled using neurons selected from the population. In contrast, the weight and topology evolving method NeuroEvolution of Augmenting Topologies (NEAT) [267] evolves ANNs at a network level and employs strategies to track when ancestral changes take place in order to more effectively make use of the crossover operator. So when it is

[1] "*What is usually done in practice is that the developer trains a number of networks with different sizes, and then the smallest network that can fulfill all or most of the required performance requirements is selected. This amounts to a tedious process of trial and errors that seems to be unfortunately unavoidable.*" [183]

shown that NEAT outperforms SANE on a given benchmark [267], it is not clear if this is due to the ability to evolve topology or due to other differences between the two methods, or both.

Additionally, when results are presented using a weight only evolving method, the topology used has typically been 'optimised' by hand before presenting the results. Therefore comparing these results to a topology and weight evolving method is unfair as the tasks are different. In one case, a suitable topology is provided and the task is to find suitable connection weights, and in the other, neither a suitable topology nor connection weights are provided and both must be found.

Finally, to the author's knowledge there are no NE techniques which solely rely on the evolution of topology with no alterations to connection weights. It is therefore difficult to assess the relative benefits/importance of connection weight evolution and topology evolution.

Therefore, there are important open questions regarding topology optimising NE methods. Firstly, whether or not it is beneficial to evolve network topology. Secondly, whether topology or connection weight evolution is more significant to the training of ANNs; when using NE.

6.3 Is it Beneficial to Evolve Network Topology?

This section presents a set of experiments investigating the benefit of using NE to adapt ANN topology. This is undertaken via a comparison between Conventional NeuroEvolution (CNE), a weight only evolving NE method, and Cartesian Genetic Programming of Artificial Neural Networks (CGPANN), a weight and topology evolving NE method. As has been previously discussed, this investigation will focus on three perceived benefits of topology evolving methods: 1) that it removes the requirement to know a suitable topology in advance of training, 2) that it allows evolution utilises topologies unlikely to be considered by a human designer, and 3) that it allows evolution to exploit relationships between topology and connection weights resulting in a more effective search.

Firstly, the influence of topology on CNE, a weight only evolving method, is investigated. This is undertaken by applying CNE to a number of benchmark tasks using a range of topologies. This is to identify if the onus on the user to select a suitable topology represents a disadvantage. For instance, it may be the case that the choice of topology has little effect on the solutions found; demonstrating that there is no need to use evolution to find suitable topologies. However, it may be the case that the choice of topology has a large influence on the solutions found, demonstrating that there is scope for using evolution to find suitable topologies.

The results of investigating CNE using a range of topologies are also compared to the use of CGPANN; a topology and weight evolving NE method. This is to assess if the ability to adapt network topology can be used to alleviate the requirement on the user to select a suitable topology. Recall that although CGPANN is free to adapt topology, it does so within user defined topology limits. For instance, the user defines a maximum number of nodes or specifies a maximum number of rows and columns; see Chapter 3. Therefore, with CNE the user must specify a fixed topology, but with CGPANN, the user must specify maximum topology *limits*. If the choice of topology limits for CGPANN influences the suitability of the solutions found more than the choice of the fixed topology for CNE, then it would have failed to lessen the onus on the user to specify a suitable topology. If the influence on the solutions found is equal, then there is little reason for the added complexity of adapting network topology. However, if the influence on the effectiveness of the search is reduced when specifying CGPANN topology limits, then it demonstrates that adapting topology lessens the onus on the user to know a suitable topology in advance of training.

Note that the comparison is not between the best solutions found using CNE and the best solutions found CGPANN. This is because the tasks assigned to CNE and CGPANN are different. In the case of CNE, the task is to optimise the connection weights of a given topology. Whereas in the case of CGPANN, the task is to optimise both connection weights and topology. What is being compared is the effect of varying the fixed topology for CNE, and the effect of varying the topology limits for CGPANN, on the solutions which are found.

Secondly, whether or not topology adapting NE methods result in the use of topologies unlikely to be considered by a human designer is investigated. However, this is challenging due to it not being known what a human designer would consider. Therefore the definition used here for topologies which would be considered by a human designer is that they can be described using the standard layers and nodes per layer formation often used by ANNs. That is to say, if they can be described in the format of “three fully connected hidden layers containing ten, five and five neurons respectively”. If effective topologies are found which cannot be described by this standard topology description, then it will be concluded that they would have been unlikely to be considered by a human designer.

Finally, whether topology adapting NE methods benefit from the ability to exploit relationships between topology and connection weights during the search is investigated. An indication of this behaviour would be demonstrated if any of the CGPANN experiments outperformed the best CNE result. This is because when CNE is provided with a suitable topology, it only has to optimise the connections weights. However, CGPANN always has to optimise the connection weights and topology; a harder task. If simultaneously optimising topology and connection weight results in a fitness landscape which is easier to navigate, CGPANN could still outperform CNE even if the task was "harder". Therefore, if CGPANN is shown to outperform CNE, even when the best found topology is used, it would indicate that relationships between connection weights and topology are being exploited by the search. The reason this would only be an indication of this behaviour is because CGPANN has access to topologies not investigated for CNE. For instance, it may be the case that a highly effective topology was never tested for CNE, but could be reached using CGPANN. If this were the case, CGPANN could be seen to outperform the best CNE topology without exploiting any relationships between topology and connections weights.

6.3.1 Experimental Setup

In the case of CNE, the number of layers and nodes per layer are both swept over the range [1,2,3,..,19,20] resulting in 400 separate topologies investigated. Typically, when using CGPANN the user must specify a maximum number of nodes, of which only a proportion are used. This is based on the "one row" form of Cartesian Genetic Programming (CGP); See Chapter 3. However, to make for simple comparisons with CNE, here the "rows and columns" form of CGPANN is used. Therefore, in the case of CGPANN the number of rows and columns are also both swept over the same range [1,2,3,..,19,20]. In both cases, CNE and CGPANN, each experiment is run for 5000 generations and repeated fifty times in order to produce a reliable arithmetic mean fitness which are used for comparison.

It was previously stated that assessing the benefit of evolving topology by comparing the results of different NE methods is challenging due to implementation differences other than the ability to evolve topology. Here both CNE and CGPANN use a $(1 + 4)$-ES, a five percent weight gene probabilistic mutation rate, no crossover and a connection weight range of ± 10. In the case of CNE, the topologies are fixed and fully connected between layers. In the case of CGPANN, the topology is free to evolve within the constraints of the given number of rows and columns and using a maximum node arity of ten.

As discussed in Chapter 4, when CGPANN evolves topology it is possible for two nodes to be connected by multiple connections. Previously this was left unchanged as CGPANN was assessed in it 'raw' form. However, an implication of this is that the maximum connection weight range can exceed that set by the user; as two connections between two nodes is equivalent to one connection with the sum of the individual connection weights. However, the maximum connection weight range used may make a task easier or harder to solve. As the experiments here are trying to isolate, as much as possible, the effect of topology evolution, this effect should be removed. Therefore in the experiments presented, only the first of multiple connections between two nodes are decoded into the phenotype. This means that the arity of each node is under the control of evolution, up to the given maximum (ten). It also means that the maximum connection weight range set by the user cannot be exceeded.

Although there are still differences between CNE and CGPANN, the differences have been minimised. In fact, in this work CNE was implemented by constraining the CGPANN implementation; by setting the initial chromosome to be of a given configuration and then disallowing topology mutations. Other than the specified initial topology, and the lack of topology mutations, the implementations of CNE and CGPANN are identical.

In this investigation three benchmark tasks are employed: Ball Throwing, Double Pole Balancing and the Monks Problem 1. Each of these benchmarks is described in Appendix A. In the case of the Ball Throwing and the Double Pole balancing, a higher fitness represents a better solution. In the case of the Monks Problem 1, a lower fitness represents a better solution. Additionally, in the case of the Monks Problem 1 benchmark, the fitnesses presented are those achieved on the training set. The generalisation ability is not considered here to simplify the experiments; otherwise early stopping methods would need to be employed. A rigorous evaluation of CGPANN as a classification method, where the figure of merit is the ability to generalise to unseen data, is given in Chapter 9. The logistic sigmoid is used for both CNE and CGPANN in the case of the Ball Throwing and the Monks Problem 1. In the case of the Double Pole Balancing, the bipolar logistic sigmoid is used; the regular logistic sigmoid scaled to a ± 1 range.

6.3.2 Results

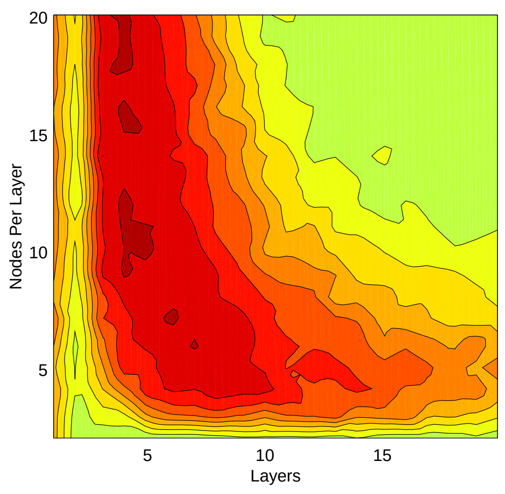
The results of the described experiment are given in Figure 6.1. The Figure shows heat maps of the fitness achieved using CNE and CGPANN over a range of topologies and topology limits respectively. In the cases of Ball Throwing and the Double Pole Balancing higher fitnesses represent better solution, in the case of the Monks Problem 1 lower fitnesses represent better solutions.

First, the effect of varying CNEs topology is evaluated; sub figures (a), (c) and (e) of Figure 6.1. As can be seen in Figure 6.1, the choice of topology has a large influence on the effectiveness of CNE's search. It is also interesting to note the types of topologies which performed well. Typically, in the ANN literature one or two hidden layers are used; with the number of nodes per layer 'optimised' by hand. Whereas from the CNE results presented in Figure 6.1, the best topologies for the Ball Throwing benchmark used around five hidden layers. On the Double Pole Balancing task, much deeper networks were found to be of benefit, all the way up to the maximum of 20 layers. A similar trend is also been for the Monks Problem 1 where the range of suitable layers extends into deeper topologies.

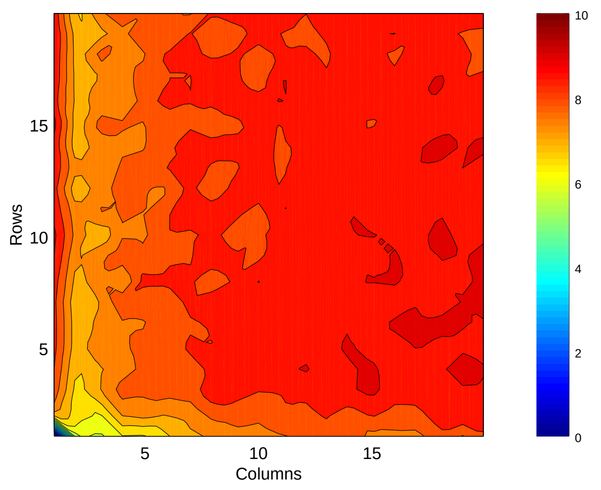
The fact that the Double Pole Balancing produced better results using topologies which would not typically be considered demonstrates the danger of relying on user defined topologies. Additionally, even if a suitable number of layers can be 'guessed' it still does not mean a suitable number of nodes per layer is known. In the case of the ball throwing benchmark, it can be seen that a larger number of nodes per layer ($\gtrsim$5) produced better results, whereas in the case of the Monks problem 1, larger numbers of nodes per layer ($\gtrsim$10) produced worse results.

It has been previously demonstrated, in work such as [178], that the performance of ANNs trained using back propagation are also strongly influenced by the topology used. Here, for completeness, the same experiment is also undertaken using back propagation. The work used the Fast Artificial Neural Network Library (FANN) library [217] and trained Multilayer Perceptrons (MLP) of logistic sigmoid functions using resilient back propagation [241] for 1000 epochs. As back propagation is typically not compatible with reinforcement tasks, only results on the Monks Problem 1 can be presented; which are given in Figure 6.2.

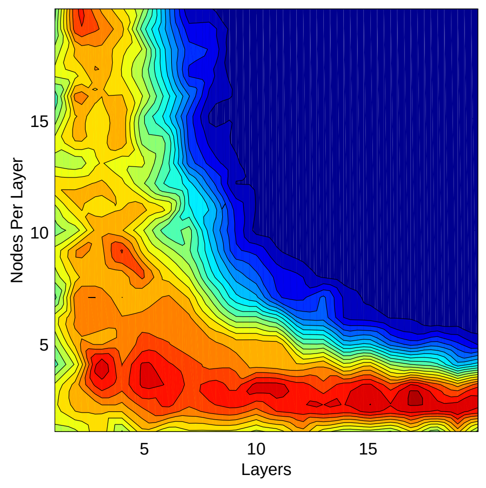
As can be seen in Figure 6.2, the effectiveness of the training ANNs using back propagation is also strongly dependent on topology. Interestingly however, the topologies which produced the best results differ to those found suitable using CNE. This is interesting



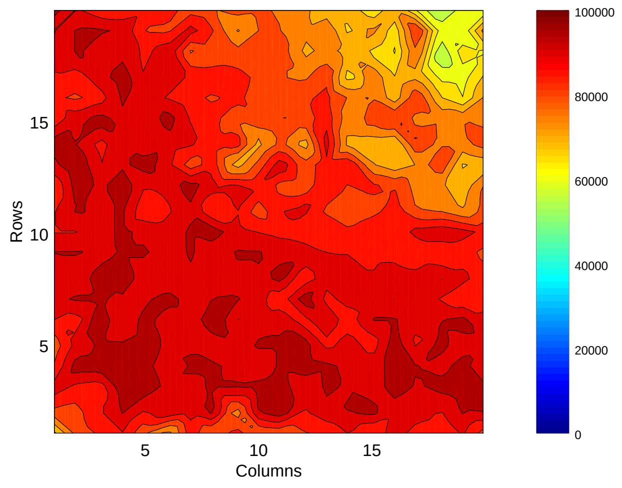
(a) CNE - Ball Throwing



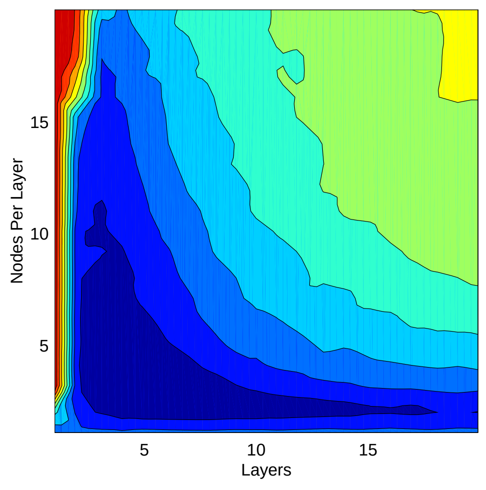
(b) CGPANN - Ball Throwing



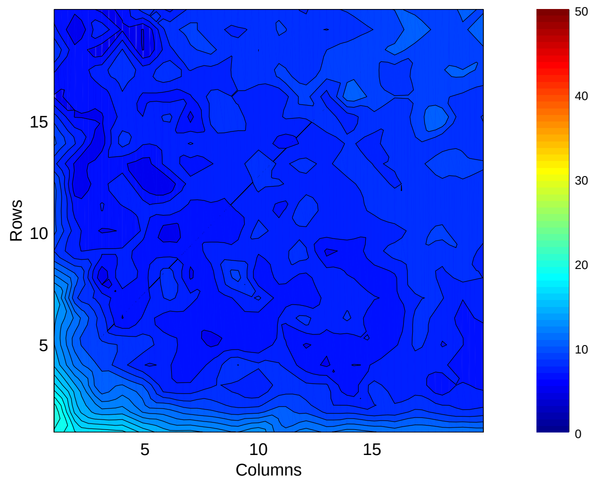
(c) CNE - Double Pole Balancing



(d) CGPANN - Double Pole Balancing



(e) CNE - Monks Problem 1



(f) CGPANN - Monks Problem 1

Figure 6.1: Effect of sweeping topology and topology limits for CNE and CGPANN respectively.

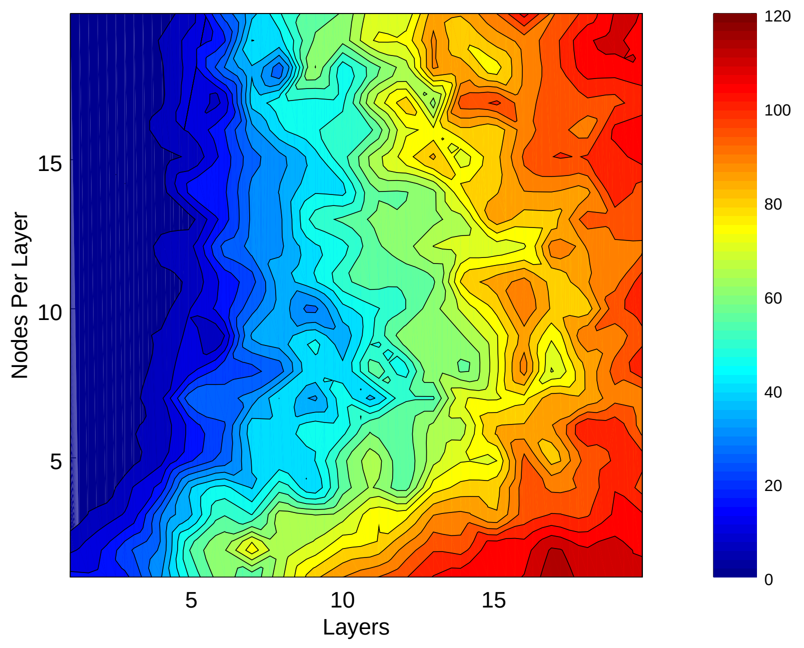


Figure 6.2: MLP trained using resilient back propagation on the Monks Problem. Note the larger range of finesses than displayed in Figure 6.1.

because it indicates that a suitable choice of topology is not only a function of a given task, but also the training method used. This has a significant consequence, current rules of thumb for network design [47] may not be applicable to NE. This is because they assume the use of back propagation as the training method. However, as only one benchmark is used here for comparison, further work would be needed to confirm this.

The results from applying CGPANN with varying topology limits are now evaluated; given in sub figures (b), (d) and (f) of Figure 6.1. When using CGPANN it can be seen that the effectiveness of the search is much more uniform across the range of topology limits investigated. This has an important implication. Even if a suitable topology is not known in advance, topology evolving methods can be used to find suitable solutions. That is to say, poor results can be avoided even if a suitable topology is not known. This is an often quoted advantage of topology and weight evolving methods which is explicitly demonstrated here.

Recall that in the case of CGPANN, topology *limits* must be provided. If the limits are set too small then certain tasks will be less tractable; as can be seen in the lower left of the sub figures in Figure 6.1. Additionally, as the limits are increased the search space widens. It is likely the case that the optimal limits are task dependent. However, as can be seen in Figure 6.1, the gradient between good and bad CNE topologies is much higher than for good and bad CGPANN topology limits. This is the advantageous property. The requirements of knowing a suitable topology is drastically *relaxed*, meaning CGPANN is likely to perform better than CNE when a suitable topology must be “guessed”.

It can also be seen in Figure 6.1, on the Ball Throwing and Monks Problem 1 benchmarks, that ANNs trained using CNE with a suitable topology produces better solutions than using CGPANN and evolving topology. This is not a surprising result. The task of searching for a suitable topology and connection weights is more substantial than optimising the connection weights of an already suitable topology. Comparing one with the other is unfair as the best result produced using CNE does not contain the computation cost of searching all the topologies in order to find the most suited. The benefit of topology optimisation is *not* that it produces a better solution even when a suitable topology is known. The advantage is that it can find suitable topologies when a suitable topology is not known. To this end, CGPANN appears to be capable of finding suiting topologies; seen in the reasonably uniform fitness across topology limits.

However, a very interesting result is seen for the Double Pole Balancing benchmark. CGPANN produced better results than the best CNE topology across a wide range of topology limits. This indicates that CGPANN’s evolutionary search may have been improved by the ability to simultaneously adapt topology and connection weights; a perceived advantage of topology optimising NE. However, as CGPANN has access to topologies not investigated for CNE, it may also be the case that CGPANN was utilising topologies unavailable to CNE. For this reason the result only indicates that CGPANN is benefit from the simultaneous evolution of connection weights and topology. Although if it is the case the CGPANN outperformed CNE due to accessing topologies not investigated by CNE, then this demonstrates an alternative advantage of evolving topology. It utilises topologies not typically considered.

Finally, one of the quoted benefits of topology evolving NE methods is that it can create topologies which are unlikely to be considered by a human designer. Figure 6.3 gives a sample solution which was found for the Double Pole benchmark from the CGPANN results. As can be seen, the topology is highly unusual, in that it does not consist of layers and nodes per layer and the node arity is varied. Although it is difficult to argue whether a given topology could have been considered by a human designer, it is clear that the topologies found can be atypical compared to those usually used by ANNs.

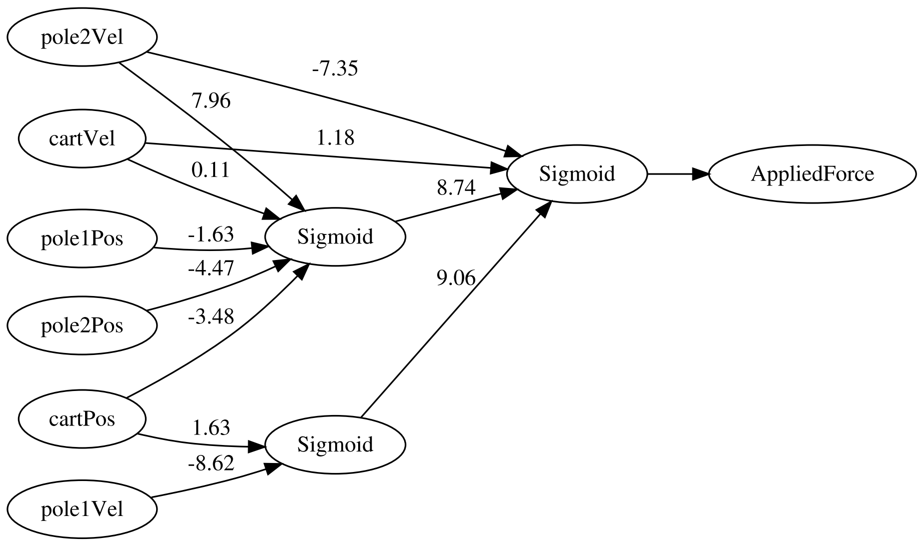


Figure 6.3: Solution found by CGPANN for the Double Pole Balancing benchmark.

6.3.3 Discussion

The results presented for CNE in Figure 6.1 demonstrates that the choice of topology has a large influence on the effectiveness of the trained ANNs. This confirms previously seen results for CMA-ES [130]. This results has also been seen previously using back propagation [178] which is in-line with the results seen in Figure 6.2. Although, from comparisons between Figures 6.1 and Figure 6.2, it appears that suitable topologies are both task and training method dependent. This potentially limits the usefulness of previous ANN topology rules of thumb [47] for NE training methods.

It was also shown in Figure 6.1, that when using CGPANN to evolve ANNs, the requirement on the user to know a suitable topology in advance of training is drastically relaxed. This demonstrates that topology and weight evolving NE methods can indeed be used to find suitable topologies when a suitable topology is not known. Additionally, this result is also likely to extend to other similar NE methods such as GeNeralized Acquisition of Recurrent Links (GNARL); where topology limits must be specified for the initial population.

It was also shown in Figure 6.3, that when allowing evolution to adapt network topology, topologies can be found which are unlikely to be considered by a human designer. That is to say, the range of topologies typically investigated is must less constrained.

There was also an indication that CGPANN may have been exploiting evolutionary paths created through the combined evolution of connection weights and topology. This was demonstrated by many CGPANN experiments outperforming all of the CNE topologies investigated on the Double Pole Balancing benchmark. However, it may also have been due to CGPANN having access to topologies not consider for CNE. Therefore, interesting future work would be to take the CGPANN solutions which outperformed CNE, randomise the connections weight, and re-evolve them using CNE. If CNE failed to equal or better the solution found using CGPANN, it would provide evidence that CGPANN was exploiting relationships between connection weights and topology. However, if CNE performed equally or better, it would be evidence that CGPANN outperformed CNE by utilising topologies not considered for CNE.

Interestingly, if it were found that CGPANN outperformed CNE due to having access to topologies not considered by CNE, then this itself would represent an advantage of optimising topology. Alternatively, if CGPANN outperformed CNE due it exploiting relationships between connections weights and topology, then this is also an advantage of optimising topology. Regardless of the cause, the fact that CGPANN was seen to outperform CNE, even when a large number of topologies were considered, clearly demonstrates that there is an advantage to evolving network topology.

However, there are a class of topology evolving NE methods which have not been considered in the work presented, those which start with minimally sized solutions and continuously add and remove nodes during training. Such NE methods include NEAT, Cooperative Co-evolution Model for evolving Artificial Neural Networks (COVNET) and NeuroEvolutionary Algorithm (NevA). Interesting future work would be to investigate such methods to identify how they compare to fixed topology methods and topology restrained methods. It may be the case that topology limited methods represent a compromise between fully constrained and unconstrained topology evolution. For instance, if a suitable topology were known then fixed topology methods could be used. If it were known that a suitable topology was likely to lie within certain limits, then evolving topology within these limits is likely to be beneficial. Finally, if it were completely unknown what topologies were suitable, allowing evolution to search the entire space of topologies is likely to be beneficial; like methods such as GNARL. This is an example of how user defined knowledge can be used to restrict the search space to known suitable areas.

One caveat, however, is methods such as NEAT, COVNET and NevA are unlikely to freely evolve topologies without any bias. This means that they are unlikely to be searching over all possible topologies. This is due to them iteratively adding/removing nodes/connections starting from a minimal form. This is undertaken to bias the search to smaller topologies which are considered to generalise more effectively to unseen data. This can also be thought of as limiting the search with user defined knowledge. Additionally, such methods are likely to become trapped in topology local optima [15], and so may not be as free to evolve topology as would be desired.

A possible criticism of this work could be that CGPANN does not truly evolve topology because the possible topologies are always constrained by upper limits. However, to the author's knowledge there are no NE methods which are truly unconstrained. For instance, even methods such as GNARL or NEAT which can iteratively add nodes are also effectively limited. This is because the length of the runs is finite. If GNARL started with twenty nodes and added a new node on every generation up to the maximum number of generations, then the possible maximum number of nodes would be twenty plus the maximum number of generations. Therefore the reachable search space does not contain every possible network. It is just more explicit for CGPANN.

6.4 Relative Importance of Topology Evolution

In the NE literature it is often assumed that topology evolution is highly important to the evolutionary search. However, its relative importance to connection weight evolution is currently unknown. This section aims to fill this gap in literature by investigating the relative importance of weight evolution and topology evolution.

Here the importance of topology evolution is investigated by using CGPANN under the following constraints:

1. Only evolving connection weights.
2. Only evolving topology.
3. Evolving both connection weights and topology.

In each case the connection weights and topologies are randomly initialised; within the limits described in the following section. In the first constraint, the randomly generated topologies remained fixed and only the connection weights are evolved. In the second constraint, the randomly generated weights remained fixed and only the topology is evolved. In the third constraint, both the randomly generated weights and topologies are evolved. By looking at the final fitnesses produced under each constraint, over a range of benchmarks, the relative importance of connection weight and topology evolution can be assessed.

6.4.1 Experimental Setup

CGPANN is limited to only evolving topology or connection weights by using probabilistic mutation and setting the mutation rate of connection genes or weight genes to zero percent respectively. For instance, to stop topology evolution, the connection genes and output genes are never mutated. Similarly, to prevent connection weight evolution, connection weight genes are never mutated.

The CGPANN parameters used are as follows: a maximum of 1000 generations, 50 runs, a $(1+4)$-ES, 30 nodes (in the single row format), a connection weight range of ± 10 and an arity of 10; where only the first of multiple connections between nodes are decoded into the phenotype. The benchmarks used for this investigation are the same as were used in the previous section; Section 6.3.1. In all cases the logistic sigmoid or the bipolar logistic sigmoid was used; as described in Section 6.3.1.

6.4.2 Results

The results of restricting CGPANN to evolving only connection weights, only topology, and evolving both connection weights and topology, are given in Figures 6.4, 6.5 and 6.6 for the Ball Throwing, Double Pole Balancing and Monks Problem 1 benchmarks respectively. In the case of the Ball Throwing and the Double Pole Balancing a higher fitness represents a better solutions, in the case of the Monks Problem 1 a lower fitness represents a better solution.

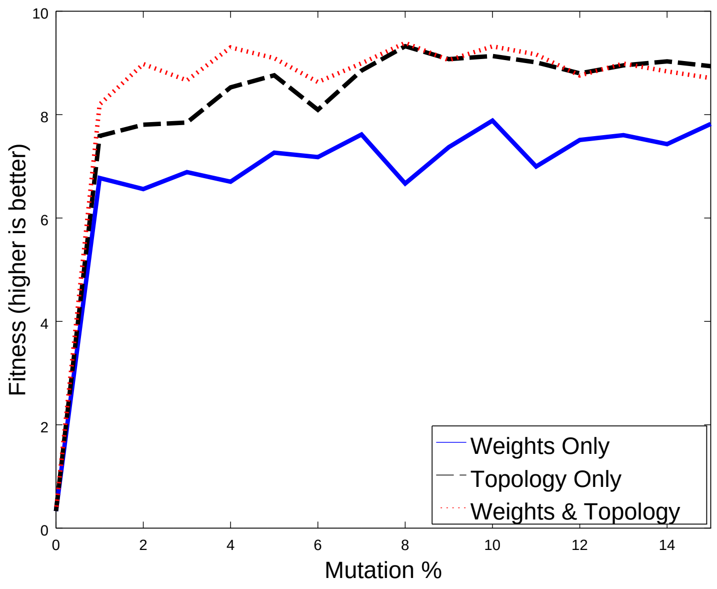


Figure 6.4: Comparing the relative importance of connection weight evolution and topology evolution using CGPANN on the Ball Throwing benchmark.

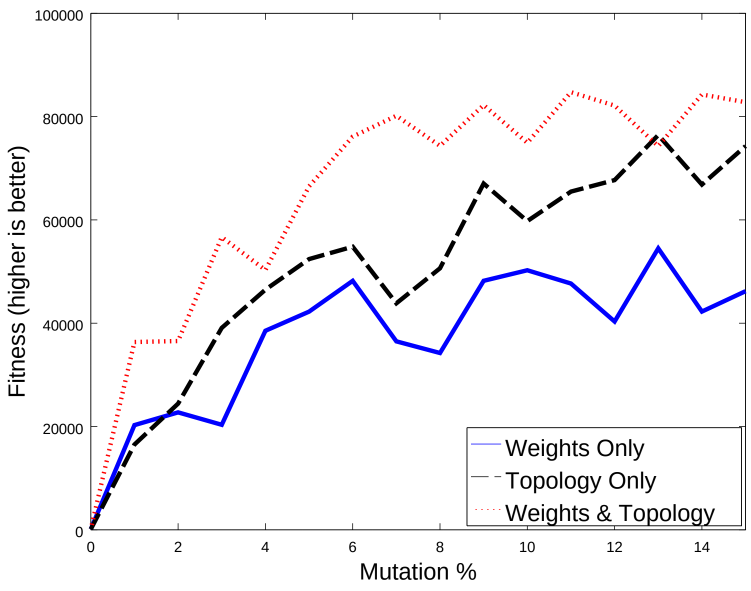


Figure 6.5: Comparing the relative importance of connection weight evolution and topology evolution using CGPANN on the Double Pole Balancing benchmark.

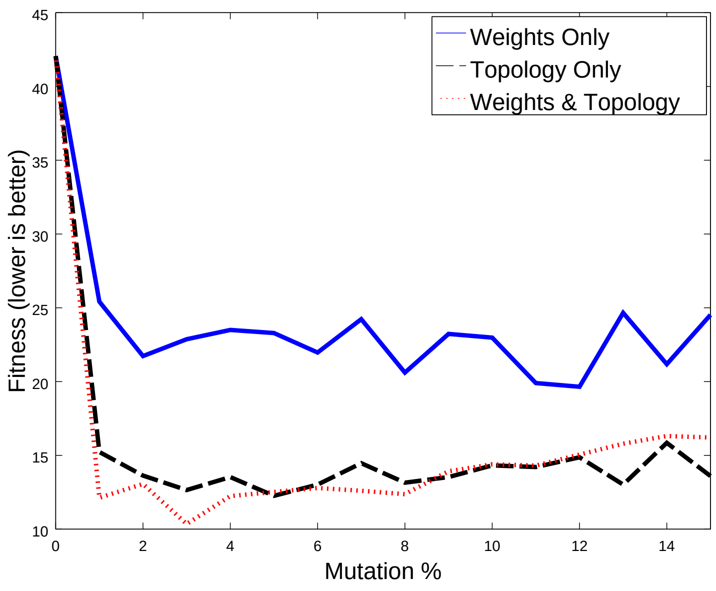


Figure 6.6: Comparing the relative importance of connection weight evolution and topology evolution using CGPANN on the Monks Problem 1 benchmark.

The results are also analysed using non-parametric statistical tests; See Appendix B. The results of these statistical tests are given In Tables 6.1, 6.2 and 6.3 for the Ball Throwing, Double Pole Balancing and Monks Problem 1 benchmarks respectively. To simplify the analysis, the statistical significance testing is only undertaken for mutation rates of 3 and 5 percent.

A number of interesting and surprising features can be seen in the results presented.

Firstly, evolving the connection weights of random fixed topologies produced a significantly worse search than evolving the topology of random fixed connection weights; with most cases being statically significant with a medium or greater effect size. This is a surprising result as it indicates that topology is more important to the training of ANNs than connection weights. Secondly, there is a large difference between evolving connection weights and topology and just evolving connection weights; with evolving connection weights and topology producing a much better search with statistical significance and often large effect sizes. This indicates that the ability to evolve topology is having a large influence on the search. Finally there is little difference between evolving connection weights and topology with just evolving topology. This indicates that the ability to evolve connection weights has little effect over and above the ability to evolve topology.

Table 6.1: Statistical analysis of the relative importance of connection weight evolution and topology evolution on the Ball Throwing benchmark.

Comparison	Mutation %	U-test	KS	A
Weights and Topology with Topology	3	5.74E-2	9.51E-2	0.609
Weights and Topology with Topology	5	4.33E-1	5.08E-1	0.546
Weights and Topology with Weights	3	2.30E-2	3.17E-2	0.631
Weights and Topology with Weights	5	1.06E-3	2.11E-3	0.689
Weights with Topology	3	5.19E-1	3.58E-1	0.537
Weights with Topology	5	9.12E-3	5.60E-2	0.650

Table 6.2: Statistical analysis of the relative importance of connection weight evolution and topology evolution on the Double Pole benchmark.

Comparison	Mutation %	U-test	KS	A
Weights and Topology with Topology	3	1.76E-1	3.58E-1	0.574
Weights and Topology with Topology	5	1.41E-1	5.08E-1	0.576
Weights and Topology with Weights	3	2.31E-6	2.76E-6	0.766
Weights and Topology with Weights	5	2.59E-3	1.71E-2	0.661
Weights with Topology	3	4.43E-5	1.78E-4	0.734
Weights with Topology	5	8.11E-2	3.12E-2	0.596

Table 6.3: Statistical analysis of the relative importance of connection weight evolution and topology evolution on the Monks Problem 1 benchmark.

Comparison	Mutation %	U-test	KS	A
Weights and Topology with Topology	3	1.24E-1	2.41E-1	0.589
Weights and Topology with Topology	5	5.93E-1	3.58E-1	0.531
Weights and Topology with Weights	3	9.46E-6	1.78E-4	0.757
Weights and Topology with Weights	5	2.67E-5	2.76E-5	0.743
Weights with Topology	3	8.35E-4	4.43E-3	0.694
Weights with Topology	5	2.48E-5	4.23E-4	0.744

6.4.3 Discussion

This section has demonstrated that the ability to evolve topology is very significant to the evolutionary search; at least for CGPANN. The surprising result was seen that the ability to evolve topology may in fact be more important to the search than the ability to optimise connection weights.

Although in real applications random topologies would never be used[2], the aim of this experiment was to identify the relative importance of topology and connection weights generally; not necessarily in the presence of user knowledge.

Additionally, the random topologies generated using CGPANN are unlikely to be evenly distributed across all possible topologies due to length bias [84]. Although the topologies are random, some configurations will be more probable than others. Therefore this experiment could be improved by ensuring that the random topologies came from an even distribution; rather than using the typical CGPANN method for initialising new chromosomes. Additionally, the range of random topologies was limited to a maximum of 30 nodes with a maximum node arity of 10. However, in the same regard so were the range of connection weights limited to ± 10.

However, despite these limitations, it certainly appears that topology has a large influence on the effectiveness of the search for CGPANN and this is likely to extend to other topology optimising NE methods. Additionally, it appears that topology optimisation is more important than weight optimisation in the general case i.e. without additional user knowledge. Therefore, it can be said, for CGPANN at least, that topology optimisation is more significant to training than weight evolution; when considering random topologies and random connection weights.

6.5 Summary

This chapter has investigated previous claims that adapting topology represents a number of advantageous properties for NE [64, 313]. The chapter has provided evidence for these claims, filling a previous gap in the literature.

Firstly, it has previously been assumed that the adaptation of topology is advantageous due to it removing the requirement for the user to know suitable topologies in advance of training. This chapter has demonstrated that when using CNE, a weight only evolving

[2]Many rules of thumb [47] can be followed to indicate suitable topologies

method, that not only was the achieved fitness highly related to topology, but that the gradient of fitnesses between suited and ill-suited topologies was very high. That is to say, there was little room for error when choosing a suitable topology.

However, in the case of CGPANN, although the user must select topology limits, the gradient of fitnesses was very low between suited and ill-suited limits. That is to say, the requirement to select a good topology was far more relaxed. Therefore, the ability to evolve topology does indeed represent an advantage in that it lowers the requirement of the user to know a suitable topology in advance of training.

This ability could be a further advantage given that it was also shown that suitable topologies were both a function of the application and the training method used. That is to say, which topologies were suitable varied between the use of CNE and back propagation. Therefore, previous methods for predicting suitable topologies may not be applicable to NE; or indeed outside of the context of back propagation.

Another previously proposed benefit of topology and weight evolving methods is that relationships between topology and connection weights can be exploited by evolution leading to better overall searches/solutions. Although not conclusive, results presented provide an indication of this effect. This was seen in the results from applying CNE and CGPANN to the Double Pole Balancing benchmark. On this benchmark CGPANN was seen to produce better results than CNE, even when using the best found topology for CNE. This indicates that CGPANN was able to utilise the ability of adjusting both connection weights and topology. The reason that this only indicates this effect, and does not prove it, is because CGPANN has access to topologies not tested for CNE. Therefore it may also be the case the topologies accessible to CGPANN, and not tested for CNE, were highly suited to the task resulting in the better results seen.

Additionally, it is thought that evolving topology can create/utilise configurations which would not typically be considered by a human designer. Although this is a difficult possible benefit to assess, an example solution was demonstrated which does not conform to the typical layers and nodes per layer format. Additionally, as has just been discussed, the cause of CGPANN outperforming CNE on the Double Pole Balancing benchmark may also be evidence of evolving topology having access to beneficial topologies unlikely to be considered by a human designer.

Finally, the relative importance of connection weight evolution and topology evolution was not previously known. Here the relative benefits have been assessed by comparing evolving the connection weights of random topologies with evolving the topology of random connection weights. It was shown that, at least for CGPANN, that topology is more significant to training than connection weights. Although random topologies are not typically used, it may help guide future developments and understanding of ANNs. For instance, currently most ANN training methods are initialised with random connection weights and what is thought to be a suitable topology. Whereas it may be equally feasible to initialise connection weights to what are thought to be suitable values and then adapt topology. Although this concept appears counter intuitive, if topology has more influence on the training than connection weights (at least in the random case), why focus on the connection weights? It is also interesting that, to the author's knowledge, all ANN training methods adjust connection weights with a smaller subset also adjusting topology. However, if it were the case that topology was more significant to training, the focus may be wrong. It may be the case that by default all ANNs training methods should adjust topology with some methods also adjusting connection weights.

It may also be the case that NE represents a highly effective method for determining network topology, but is ill suited to configuring connection weights. If this is the case, combinations of NE with back propagation, such as previously used by NEAT [41] and Evolutionary Programming Artificial Networks (EPNet) [314], may represent a suitable direction for NE research. By utilising NE methods ability to configure topology with back propagations ability to configure connection weights. However it is important to note that combining NE with back propagation removes many of the advantage properties of NE; such as being applicable to Recurrent Artificial Neural Network (RANN)s, being able to use arbitrary transfer functions and being applicable to reinforcement type learning tasks. Therefore the combination of NE with back propagation does not come without disadvantages.

This chapter has helped answer many open questions concerning topology evolution and has given evidence to previous claims. It can now be more confidently argued that the ability of topology adapting NE methods to optimise topology is a significant advantage over only evolving connection weights. Additionally it has been shown, at least in the random case, that evolving topology may be more important to NE training than evolving connection weights.

Chapter 7

Evolving Heterogeneous Artificial Neural Networks

One of the often overlooked features of NeuroEvolution (NE) methods is their ability to create heterogeneous Artificial Neural Network (ANN)s. Heterogeneous ANNs are those which comprise of two or more different types of neuron Transfer Function (TF); for instance a combination of logistic sigmoid and Gaussian. This chapter assesses whether the ability to create heterogeneous ANNs is advantageous for NE.

7.1 Structure of this Chapter

Section 7.2 provides a background of using NE to evolving heterogeneous ANNs. Section 7.3 describes a number of experiments investigating the use of NE method for evolving heterogeneous ANNs. Section 7.4 presents the results of experiments which investigate the influence the choice of TF has when training homogeneous networks using NE. Sections 7.5 and 7.6 present the results of investigations into two separate methods of training heterogeneous ANNs using NE. A combination of these two methods is also presented in Section 7.7. Finally, a discussion of the presented results is given in Section 7.9 with a closing summary in Section 7.10.

7.2 Background

One of the interesting, but less commonly utilised, features of NE is that it can be used to optimise the TF of each neuron within heterogeneous ANNs[1]. However, this capability has been widely overlooked in recent research. Indeed, at the turn of the 21st century many ANN publications stated that more research was required concerning the optimisation of TFs: "*Relatively little has been done on the evolution of node transfer functions, let alone the simultaneous evolution of both topological structure and node transfer functions"* [313], "*The current emphasis in neural network research is on learning algorithms and architectures, neglecting the importance of transfer functions*" [54] and "*Selection and/or optimisation of transfer functions performed by artificial neurons have been so far little explored ways to improve performance of neural networks in complex problems*" [55]. However, a search of the literature reveals that there has been little active research in this area. This chapter intends to fill this gap in the literature by showing how NE can easily optimise neuron TFs during evolution and that doing so is beneficial.

There are many ANN TFs found in the literature [55]. However, the majority of NE implementations only evolve homogeneous ANNs of logistic sigmoid or Gaussian functions, which have both been shown capable of universal approximation; [118] and [221] respectively. Of those which do evolve heterogeneous ANNs, there are two main methods.

The first method selects the TF of each neuron from a predetermined list of TFs. NE training methods which use this method include General Neural Networks (GNN) [179]; which randomly adds or removes logistic or Gaussian TFs using an evolutionary programming method. GNN is also a hybrid approach which makes use of back propagation during training. Other NE methods which select specific TFs for each neuron include Parallel Distributed Genetic Programming (PDGP) [227], modified Hierarchical Coevolutionary Genetic Algorithm (HCGA) [299] and Cartesian Genetic Programming of Artificial Neural Networks (CGPANN) [153]. These methods use genes to encode which TF is used by each neuron. These genes are then subject to mutation and/or crossover during evolution.

The second method optimises ANNs of TFs which are each described by a number of parameters [55]. The training methods then optimise these parameters for each individual neuron. A simple version of this technique has been used by CGPANN [191]; where the

[1]Homogeneous ANNs are ANNs where each neuron's TF is identical. Heterogeneous ANNs are ANNs is where one or more of the neuron's TF is different.

widths of Gaussian functions were optimised for each neuron. Again, the parameter(s) associated with each neuron's TF were encoded in the chromosome by the inclusion of additional gene(s). A more complex version of this method was used in [20], where each neuron's TF was itself an evolved tree-based genetic program. This method allowed for an almost limitless variation of TFs. Another example where each neuron is described by a number of genes, is state-enhanced neural networks [209], where the dynamics of each neuron are evolved.

Until now however, there has been little research which empirically and rigorously investigates whether the ability for NE to evolve heterogeneous ANNs actually provides any benefit. This is important research, as if it is shown to be beneficial it could easily be adopted by other NE methods; as the described methods just require additional genes for each neuron. As discussed, there are two ways in which NE can evolve TFs: 1) by choosing the TF of each neuron from a predetermined list or 2) by optimising parameters associated with each individual neuron. Additionally these two methods can be combined by allowing evolution to both select the TF for each neuron and optimises parameters associated with that TF. Here both of these methods are investigated along with their combination. The investigation uses two NE strategies and compares the results to evolving regular homogeneous ANNs.

7.3 Investigations

The investigation presented on evolving heterogeneous ANNs using NE takes four parts. The first is to identify whether, and to what extent, the choice of TF impacts on the effectiveness of NE when evolving homogeneous ANNs. The second investigates if evolving heterogeneous ANNs, by allowing evolution to select each neuron's TF from a predetermined list, outperforms evolving homogeneous ANNs. The third investigates if using NE to optimise parameters associated with each neuron's TF outperforms evolving homogeneous ANNs. Finally, the fourth investigates using NE to both select each neuron's TF from a predetermined list and optimise parameters associated with that TF.

The remainder of this section introduces the NE methods employed by the investigation, the TFs made available and the benchmarks used.

7.3.1 NeuroEvolutionary Methods

In order to ensure that any conclusions reached are not specific to a particular type of NE method, the investigations are undertaken using two separate NE methods. The chosen NE methods include Conventional NeuroEvolution (CNE) and CGPANN. CNE is a simple NE method which evolves only the connection weights of fixed topology ANNs; see Section 2.6.1. As described in Chapter 4, CGPANN is a more complex NE method which evolves both the connection weights and topology of ANNs. These two NE methods represent the two main types of NE; those which evolve only connection weights and those which evolve connection weights and topology.

For both CNE and CGPANN, the following parameters are used: 50 runs, 100,000 generations, a $(1+4)$-ES, a connection weight range of ± 5, 3% probabilistic mutation and no crossover. When using CNE, three hidden layers are used, each containing ten neurons; plus one input layer and one output layer. The arity of each neuron is such that the ANNs are fully connected between layers. When using CGPANN, the maximum number of nodes is set as thirty each with a maximum arity of ten; the one row from of CGPANN is used with multiple connections between pairs of nodes allowed.

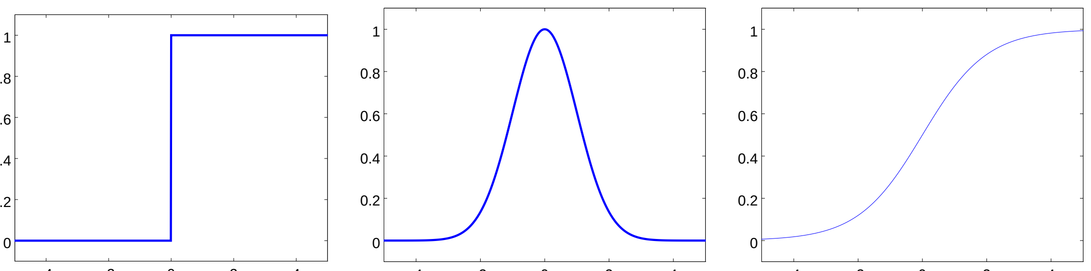
7.3.2 Transfer Functions

The TFs used in this investigation are the Heaviside step function, Equation 7.1, the Gaussian function, Equation 7.2, and the logistic sigmoid function, Equation 7.3. Each of these TFs is also shown graphically in Figure 7.1. These particular TFs were selected as they are, or have been, commonly used by ANNs.

$$f(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{7.1}$$

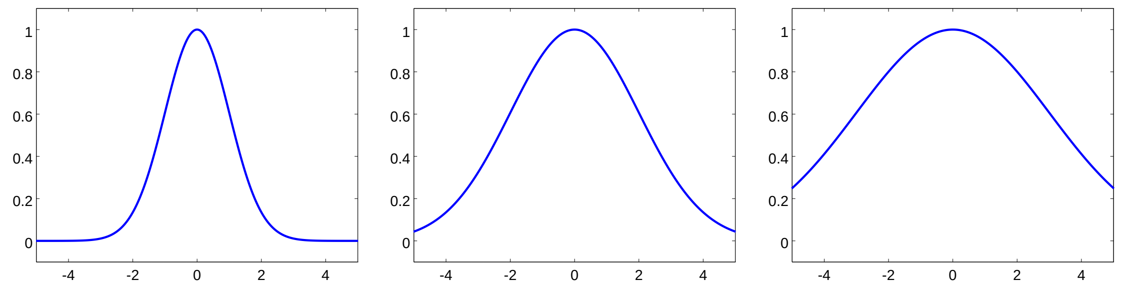
$$f(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{7.2}$$

$$f(x) = \frac{1}{1+e^{-\sigma x}} \tag{7.3}$$



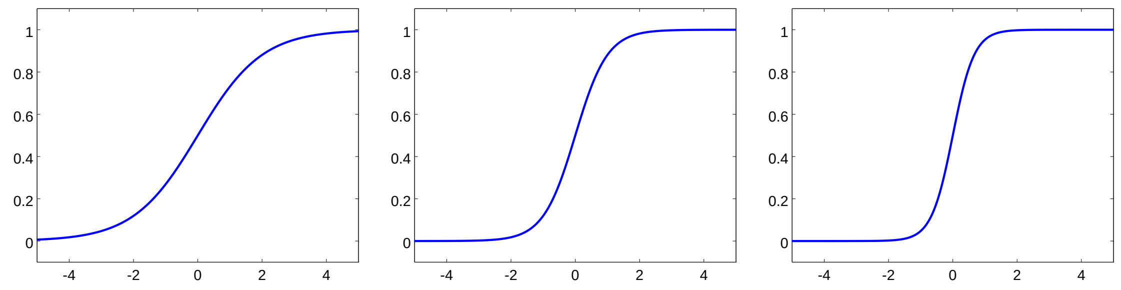
(a) Heaviside step function (b) Gaussian function (c) Logistic sigmoid function

Figure 7.1: Heaviside step function (a), Gaussian function (b) and the logistic sigmoid function (c). With $\sigma = 1$ for the Gaussian and logistic functions.



(a) $\sigma = 1$ (b) $\sigma = 2$ (c) $\sigma = 3$

Figure 7.2: Variable Gaussian function.



(a) $\sigma = 1$ (b) $\sigma = 2$ (c) $\sigma = 3$

Figure 7.3: Variable logistic sigmoid function.

As can be seen in Equations 7.2 and 7.3, the Gaussian and logistic functions have been given in a form which contains a σ variable. Where $\sigma = 1$ gives the typical form of these TFs. When using NE to evolve parameters associated with each neuron's TF, it is this σ value which is evolved/adapted. Figures 7.2 and 7.3 show the Gaussian and logistic function respectively for a range of σ values.

7.3.3 Benchmarks

In order to draw strong conclusions regarding whether it is beneficial to evolve TFs, it is necessary to examine its effectiveness on a wide range of benchmarks. For this purpose five benchmarks are employed. The chosen benchmarks mainly include supervised learning classification tasks, a common application of ANNs, but also include a reinforcement learning control task.

Despite many of the described benchmarks being classification tasks, they each use their own type of fitness function. Although this adds complexity, the fitness functions used here are those typically used with these benchmarks. This is done to continue the standard use of these benchmarks; which is important when comparing machine learning methods generally.

The following benchmarks are employed: Ball Throwing, Full Adder, Monks Problem 1, Two Spirals and Proben: Cancer 1. These benchmarks are all described in Appendix A. For reference, the fitness for the ball throwing task the distance the controller throws the ball, with higher values representing a fitter solution. In the case of the full adder, the fitness score is the number of correct output bits generated over all possible input conditions, with higher values representing a fitter solution. In the case of the Monks Problem the fitness is the percentage of misclassified robots, with lower values representing a better solution. In the case of the Two Spirals the fitness is the number of points misclassified, with lower values representing a fitter solution. Finally, in the case of the Proben:1 Cancer, the fitness is the squared error percentage, with lower values representing a fitter solution.

7.4 Evolving Homogeneous Networks

The first experiment identifies whether, and to what extent, the choice of TF impacts on the effectiveness of training homogeneous ANNs using NE. As previously discussed, the three TFs used for this investigation are the Heaviside step, Gaussian and logistic sigmoid functions; see Section 7.3.2.

The average fitness achieved when using each TF is given for the five benchmarks in Tables 7.1 and 7.3; when using CNE and CGPANN respectively. The average fitness value is given in bold if it represents the best fitness for the given benchmark; indicating the most suited TF. When appropriate, the fitness is given for the training and testing sets. Where the testing fitness is the average fitness achieved by each of the fifty runs on the testing set after training on the training set is complete. The statistical significance between the fitnesses achieved using each TF are given in Tables 7.5 and 7.7; when using CNE and CGPANN respectively. When the difference is statistically significant, $p \leq 0.05$, the value is given in bold. The effect sizes of the differences between the fitnesses are also given in Tables 7.9 and 7.11; when using CNE and CGPANN respectively. When the effect size is of medium or greater importance the value is given in bold. For a justification and description of the statistical significance measures used, see Appendix B.

In all cases a perfect solutions was found for the Full Adder benchmark and similarly in many cases for the Ball Throwing benchmark. When perfect solutions are found no comparisons can be made in terms of the fitnesses achieved. For this reason, the number of generations required to find perfect solutions are also presented for these cases. Tables 7.2 and 7.4 give the average number of generations required by CNE and CGPANN respectively for the cases where perfect solutions were found. These generation results are analysed using the same statistical significance testing as before, given in Tables 7.6 and 7.10 for CNE and Tables 7.8 and 7.12 for CGPANN.

From the results given in Tables 7.1 - 7.4, it can be seen, for both CNE and CGPANN, that the choice of TF has a large impact on the effectiveness of NE. Additionally, in the majority of cases these differences are shown to be statistically significant with a medium or large effect size; Tables 7.5 - 7.12. This confirms that the choice of TF has a large impact on the effectiveness of evolving homogeneous ANNs when trained using NE.

A further interesting result, also seen in Tables 7.1 - 7.4, is that despite being the least commonly used of the three TFs, the Heaviside step function produced the best results in many cases. It can also be seen that the best TF was often also dependent on both the task *and* the NE method used.

In the case of CGPANN, all TFs found a solution to the Ball Throwing benchmark; throwing the ball a distance of ≥ 9.5 m. Interestingly however, some TFs managed, on average, to throw the ball much further than the 9.5 m target. This could be framed as greater generalisation. However this aspect of the ball throwing results are not analysed further in this chapter.

Table 7.1: Fitness achieved using homogeneous ANNs of different TFs trained using CNE.

Benchmark	Step	Gaussian	Logistic	Average
Ball Throwing	**9.71**	9.30	5.89	8.30
Full Adder	**16.00**	**16.00**	**16.00**	16.00
Monks Problem 1 Train	**0.065**	14.016	0.258	4.80
Monks Problem 1 Test	18.991	39.116	**14.981**	24.363
Two Spirals	54.64	**34.04**	74.58	54.42
Proben1: Cancer Train	5.185	2.389	**1.798**	3.124
Proben1: Cancer Test	12.816	6.736	**3.034**	7.529

Table 7.2: Number of generations required to find optimal solutions using homogeneous ANNs of different TFs trained using CNE.

Benchmark	Step	Gaussian	Logistic	Average
Ball Throwing	20365.44	-	-	-
Full Adder	**132.94**	317.04	476.54	308.84

Table 7.3: fitness achieved using homogeneous ANNs of different TFs trained using CGPANN.

Benchmark	Step	Gaussian	Logistic	Average
Ball Throwing	**9.72**	**9.58**	**9.65**	9.65
Full Adder	**16.00**	**16.00**	**16.00**	16.00
Monks Problem 1 Train	**0.210**	15.161	0.952	5.44
Monks Problem 1 Test	4.398	19.532	**4.352**	9.43
Two Spirals	**39.56**	49.28	71.28	53.37
Proben1: Cancer Train	**0.457**	1.364	1.429	1.083
Proben1: Cancer Test	3.678	2.989	**2.218**	2.962

Table 7.4: Number of generations required to find optimal solutions using homogeneous ANNs of different TFs trained using CGPANN.

Benchmark	Step	Gaussian	Logistic	Average
Ball Throwing	**487.76**	9850.46	20401.14	10246.45
Full Adder	**386.60**	729.20	1092.50	736.10

Table 7.5: Statistical significance between the homogeneous CNE fitness results given in Table 7.1.

Benchmark	Step Vs. Gaussian	Step Vs. Logistic	Gaussian Vs. Logistic
Ball Throwing	**2.65E-1**	**8.19E-19**	**1.13E-14**
Full Adder	-	-	-
Monks Problem 1 Train	**1.16E-19**	1.81E-1	**3.30E-19**
Monks Problem 1 Test	**6.97E-18**	**1.49E-04**	**6.96E-18**
Two Spirals	**4.42E-18**	**4.30E-18**	**6.55E-18**
Proben1: Cancer Train	**1.15E-14**	**5.59E-18**	**6.76E-3**
Proben1: Cancer Test	**4.90E-14**	**6.29E-18**	**5.98E-14**

Table 7.6: Statistical significance between the homogeneous CNE generational results given in Table 7.2.

Benchmark	Step Vs. Gaussian	Step Vs. Logistic	Gaussian Vs. Logistic
Full Adder	**1.19E-07**	**1.26E-10**	**4.63E-2**

Table 7.7: Statistical significance between the homogeneous CGPANN fitness results given in Table 7.3.

Benchmark	Step Vs. Gaussian	Step Vs. Logistic	Gaussian Vs. Logistic
Ball Throwing	**7.25E-5**	7.36E-2	**8.20E-3**
Full Adder	-	-	-
Monks Problem 1 Train	**1.42E-13**	9.90E-2	**1.35E-11**
Monks Problem 1 Test	**1.96E-9**	9.17E-1	**2.94E-9**
Two Spirals	**6.36E-10**	**6.61E-18**	**4.39E-17**
Proben1: Cancer Train	**1.36E-16**	**1.11E-17**	7.16E-2
Proben1: Cancer Test	**9.10E-3**	**2.48E-9**	**1.78E-3**

Table 7.8: Statistical significance between the homogeneous CGPANN generational results given in Table 7.4.

Benchmark	Step Vs. Gaussian	Step Vs. Logistic	Gaussian Vs. Logistic
Ball Throwing	**5.69E-15**	**4.06E-14**	**2.04E-3**
Full Adder	8.25E-1	**3.03E-4**	**8.89E-3**

Table 7.9: Effect Size between the homogeneous CNE fitness results given in Table 7.1.

Benchmark	Step Vs. Gaussian	Step Vs. Logistic	Gaussian Vs. Logistic
Ball Throwing	**0.90600**	**0.98840**	**0.92600**
Full Adder	0.50000	0.50000	0.50000
Monks Problem 1 Train	**1.00000**	0.54400	**1.00000**
Monks Problem 1 Test	**1.00000**	**0.72020**	**1.00000**
Two Spirals	**1.00000**	**1.00000**	**1.00000**
Proben1: Cancer Train	**0.94760**	**1.00000**	**0.65560**
Proben1: Cancer Test	**0.93680**	**0.99980**	**0.93420**

Table 7.10: Effect Size between the homogeneous CNE generational results given in Table 7.2.

Benchmark	Step Vs. Gaussian	Step Vs. Logistic	Gaussian Vs. Logistic
Full Adder	**0.80740**	**0.87340**	**0.61580**

Table 7.11: Effect Size between the homogeneous CGPANN fitness results given in Table 7.3.

Benchmark	Step Vs. Gaussian	Step Vs. Logistic	Gaussian Vs. Logistic
Ball Throwing	**0.73040**	**0.60400**	**0.65360**
Full Adder	0.50000	0.50000	0.50000
Monks Problem 1 Train	**0.87360**	0.53960	**0.85060**
Monks Problem 1 Test	**0.84540**	0.50620	**0.84120**
Two Spirals	**0.85840**	**1**	**0.98740**
Proben1: Cancer Train	**0.97580**	**0.99260**	**0.60320**
Proben1: Cancer Test	**0.65000**	**0.84240**	**0.67920**

Table 7.12: Effect Size between the homogeneous CGPANN generational results given in Table 7.4.

Benchmark	Step Vs. Gaussian	Step Vs. Logistic	Gaussian Vs. Logistic
Ball Throwing	**0.95340**	**0.93880**	**0.67920**
Full Adder	0.51300	**0.70980**	**0.65200**

7.5 Evolving Heterogeneous Networks

The second experiment identifies if allowing NE to evolve heterogeneous ANNs, by selecting each neuron's TF from a predetermined list, produces better results than evolving homogeneous ANNs. Evolving the TF used by each neuron is considered beneficial if the result is better than the average of using each TF individually. This measure is chosen because when approaching a new task, it not generally known which TF would be most suited, therefore a TF would have to be selected arbitrarily (randomly). However, when evolving heterogeneous ANNs, the need to make this choice is removed, and hence it should be considered beneficial if it outperforms the average random choice of TF.

The average fitnesses from evolving homogeneous ANNs, using each TF individually for the five benchmarks, are given in Tables 7.1 and 7.3 for CNE and CGPANN receptively. In the cases where a perfect solution is always found, the required number of generations is also given and used for the comparison; Tables 7.2 and 7.4.

The results of allowing CNE and CGPANN to evolve heterogeneous ANNs are given in Tables 7.13 and 7.14 respectively. Where appropriate, both the average fitness achieved and the average number of generations required to reach that fitness is given. In Tables 7.13 and 7.14 the results are given in bold if the fitness is better, or equal, to the average of using each TF individually; or the average number of generations required to find a perfect solution is equal or lower. The percentage of neurons which use each TF is also given in Tables 7.13 and 7.14; this is only for the active nodes in the case of CGPANN. No statistical analysis is undertaken for this experiment as the comparison is against the average result of using each TF individually.

As can be seen in Tables 7.13 and 7.14, in the majority of cases evolving heterogeneous ANNs outperformed the average result of evolving homogeneous ANNs. This indicates that evolving heterogeneous ANNs is typically a better strategy than evolving homogeneous ANNs. This holds unless the user knows in advance which TF is most suited to a given task; in which case that TF should be used.

Table 7.13: Average results achieved using heterogeneous ANNs trained using CNE. The result is given in bold if evolving heterogeneous ANNs outperformed the average fitness of using each transfer function individually.

Benchmark	Train	Test	Generations	Step	Gaussian	Logistic
Ball Throwing	**9.71**	-	2931.04	34.3%	34.1%	31.6%
Full Adder	**16.00**	-	**201.90**	32.3%	36.0%	31.7%
Monks Problem 1	**3.597**	27.30	-	33.2%	34.5%	32.4%
Two Spirals	**38.52**	-	-	37.2%	32.7%	30.1%
Proben1: Cancer	**1.505**	**4.379**	-	33.6%	30.1%	36.3%

Table 7.14: Average results achieved using heterogeneous ANNs trained using CGPANN. The result is given in bold if evolving heterogeneous ANNs outperformed the average fitness of using each transfer function individually.

Benchmark	Train	Test	Generations	Step	Gaussian	Logistic
Ball Throwing	**9.68**	-	**1000.76**	31.4%	34.0%	34.5%
Full Adder	**16.00**	-	**698.10**	32.5%	35.2%	32.3%
Monks Problem 1	**1.226**	**6.139**	-	38.4%	27.3%	34.3%
Two Spirals	**50.20**	-	-	34.0%	34.3%	31.8%
Proben1: Cancer	1.086	3.126	-	34.9%	33.5%	31.6%

7.6 Evolving Transfer Function Parameters

The third experiment is to identify if creating heterogeneous ANNs by optimising parameters associated with each neuron's TF is beneficial for NE. As previously discussed, the parameters to be optimised vary the shape of the Gaussian and logistic functions; see Section 7.3.2. In each case, the ANNs are comprised of the same TF, Gaussian or logistic, but a parameter controlling the shape of each neuron's TF is evolved. Here the parameter values for each TF are limited to the set $\{1, 2, 3\}$, see Equations 7.2 and 7.3; but this is not a requirement of the method.

Evolving parameters associated with each neuron's TF will be considered beneficial if it produces stronger results than the use of the non-parameterised counterpart e.g. if variable Gaussian produces stronger results than the standard Gaussian TF. This comparison is used as it isolates the aspect of interest; whether the ability to vary the shape of each neuron's TF provides any benefit.

The average fitnesses achieved using variable Gaussian and variable logistic functions are given in Tables 7.15 and 7.17 respectively when using CNE. In cases where the target fitness is always reached, the average number of generations required are given in Tables 7.16 and 7.18. Similarly, Tables 7.19 and 7.21 give the fitnesses achieved when using CGPANN. Again, in cases where the target fitness is always reached, the average number of generations required are given in Tables 7.20 and 7.22. In all cases the results are compared against those obtained using the non-variable form of the TFs. In the given tables, a bold fitness/number of generations indicates the best performance; standard or variable TFs. Additionally, a bold value for the U-test indicates statistical significance and a bold value for the effect size indicates a medium or large effect size.

As can be seen in Tables 7.15-7.22, in the majority of cases, the variable version of the TF outperformed the non-variable form. Additionally, many instances where the variable form is superior are also shown to be statistically significance with a medium to large effect size. Only four of the twenty cases show the non-variable form outperforming the variable form with statistical significance and medium to large effect size. Eleven of the twenty cases showed the variable form outperforms the non-variable form with statistical significance with a medium to large effect size. Of the remaining five cases, one showed variable TFs offered a weak disadvantage and the remainder showed no significant difference[2]. Therefore, using variable TFs is shown to offer a significant improvement in the majority of cases.

The differences can also be given in terms of the NE method used. In seven of the ten cases which used CNE, the variable TFs offered an advantage compared with three cases offering a disadvantage. For CGPANN, five out of the ten cases showed variable TFs offered an advantage compared to one of the ten cases showing a disadvantage. The results can also be given in terms of the TF employed. In six of the ten cases the variable logistic TF offered an advantage and in two cases a disadvantage. In five of the ten cases the variable Gaussian TF offered an advantage and in three cases a disadvantage.

Table 7.15: Average fitness of ANNs of variable Gaussian TFs trained using CNE.

Benchmark	Gaussian	Variable Gaussian	U-test	Effect Size
Ball Throwing	9.30	**9.58**	**1.20E-04**	**0.72340**
Full Adder	16.00	16.00	-	0.50000
Monks Problem 1 Train	14.016	**13.500**	0.54481	0.53520
Monks Problem 1 Test	39.116	**37.634**	**3.25E-2**	**0.62420**
Two Spirals	**34.04**	49.36	**5.42E-15**	**0.95340**
Proben1: Cancer Train	2.389	**1.512**	**2.41E-07**	**0.79700**
Proben1: Cancer Test	6.736	**3.598**	**1.10E-10**	**0.87320**

[2]In the case where training and testing results are given the training result is used for comparison. This is because no steps to combat over training were made. Chapter 9 presents a more rigorous evaluation of CGPANN as a classification method.

Table 7.16: Average number of generations required to find optimal solutions using ANNs of variable Gaussian TFs trained using CNE.

Benchmark	Gaussian	Variable Gaussian	U-test	Effect Size
Full Adder	317.04	**221.08**	**6.54E-3**	**0.65800**

Table 7.17: Average fitness of ANNs of variable logistic TFs trained using CNE.

Benchmark	Logistic	Variable Logistic	U-test	Effect Size
Ball Throwing	5.89	**9.62**	**3.22E-18**	**0.98000**
Full Adder	16.00	16.00	-	0.50000
Monks Problem 1 Train	**0.258**	0.710	**1.67E-2**	**0.60720**
Monks Problem 1 Test	**14.981**	19.218	**8.04E-04**	**0.69460**
Two Spirals	74.58	**58.70**	**6.51E-18**	**1.00000**
Proben1: Cancer Train	1.798	**1.684**	0.124	**0.58800**
Proben1: Cancer Test	**3.034**	3.598	**1.74E-2**	**0.63660**

Table 7.18: Average number of generations required to find optimal solutions using ANNs of variable logistic TFs trained using CNE.

Benchmark	Logistic	Variable Logistic	U-test	Effect Size
Full Adder	132.94	**270.18**	**1.61E-3**	**0.68320**

Table 7.19: Average fitness of ANNs of variable Gaussian TFs trained using CGPANN.

Benchmark	Gaussian	Variable Gaussian	U-test	Effect Size
Ball Throwing	9.58	9.60	3.72E-1	0.55200
Full Adder	16.00	16.00	-	0.50000
Monks Problem 1 Train	15.161	**6.725**	**2.72E-4**	**0.69980**
Monks Problem 1 Test	19.532	**11.903**	**6.41E-4**	**0.69380**
Two Spirals	**49.28**	54.78	8.02E-5	**0.72880**
Proben1: Cancer Train	**1.364**	1.375	4.69E-1	0.54160
Proben1: Cancer Test	2.989	**2.690**	2.71E-1	**0.56340**

Table 7.20: Average number of generations required to find optimal solutions using ANNs of variable Gaussian TFs trained using CGPANN.

Benchmark	Gaussian	Variable Gaussian	U-test	Effect Size
Ball Throwing	9850.46	**4730.86**	1.12E-1	**0.59240**
Full Adder	729.20	**507.98**	7.75E-1	0.51680

Table 7.21: Average fitness of ANNs of variable logistic TFs trained using CGPANN.

Benchmark	Logistic	Variable Logistic	U-test	Effect Size
Ball Throwing	9.65	9.66	7.17E-1	0.52120
Full Adder	16.00	16.00	-	0.50000
Monks Problem 1 Train	0.952	**0.290**	2.25E-1	0.53000
Monks Problem 1 Test	4.352	**4.287**	9.42E-1	0.50440
Two Spirals	71.28	**63.00**	**2.81E-9**	**0.84460**
Proben1: Cancer Train	1.429	**1.082**	**5.03E-7**	**0.78760**
Proben1: Cancer Test	**2.218**	2.966	**7.39E-4**	**0.69220**

Table 7.22: Average number of generations required to find optimal solutions using ANNs of variable logistic TFs trained using CGPANN.

Benchmark	Logistic	Variable Logistic	U-test	Effect Size
Ball Throwing	20401.14	**1959.20**	**2.53E-9**	**0.84600**
Full Adder	1092.50	**630.16**	9.32E-2	**0.59760**

7.7 Evolving Heterogeneous Networks and Transfer Function Parameters

The fourth experiment investigates the evolution of heterogeneous ANNs while also optimising parameters associated with each neuron's TF. The function set contains the step, Gaussian and logistic functions where the Gaussian and logistic functions can have their σ values in the range {1,2,3}; the step function is not parameterised.

Evolving heterogeneous ANNs which also optimise TF parameters will be considered beneficial if it produces stronger results than the use of the non-parameterised heterogeneous counterpart; given in Section 7.5. This comparison is made to identify if including variable TFs can improve again upon the use of heterogeneous ANNs.

The results of evolving heterogeneous ANNs with variable TF parameters are given in Tables 7.23 and 7.25 when using CNE and CGPANN respectively. The tables give the average fitness for applying heterogeneous ANN with variable TFs to each of the five benchmarks. The average fitness is given in bold if it represents a better average fitness than that found when evolving non-parameterised heterogeneous ANNs. The U-test and effect size are also in the same format as previously in Section 7.6. Additionally, as undertaken previously, if perfect solutions are always found, the number of generations required to find perfect solutions is also give so as to allow comparison. These results are given in Tables 7.24 and 7.26 for CNE and CGPANN respectively.

As can be seen in Tables 7.23-7.26, in the majority of cases the addition of varying TF parameters to evolving heterogeneous ANNs does not improve the performance. When using CNE it produced statistically significant superior results for the Ball Throwing benchmark and produced statistically significant worse results for the Two Spirals benchmark. When using CGPANN it produced no statistically significant superior results and produced statistically significantly worse results for the Two Spirals benchmark. In all other cases there was no statistical significance between the two techniques. It can therefore be concluded that the addition of variable TFs to evolving heterogeneous ANNs does not improve the performance over the use of non-variable TFs.

Table 7.23: Average fitness of variable heterogeneous ANNs trained using CNE.

Benchmark	Heterogeneous	Variable Heterogeneous	U-test	Effect Size
Ball Throwing	9.71	9.67	0.230	**0.56980**
Full Adder	16.00	16.00	-	0.50000
Monks Problem 1 Train	3.597	**3.387**	7.00E-1	0.52240
Monks Problem 1 Test	27.30	**25.065**	5.93E-2	**0.60960**
Two Spirals	**38.52**	49.86	**5.58E-12**	**0.89960**
Proben1: Cancer Train	1.505	**1.383**	1.31E-1	**0.58660**
Proben1: Cancer Test	4.379	**3.287**	**4.71E-04**	**0.70160**

Table 7.24: Average number of generations required to find optimal solutions using variable heterogeneous ANNs trained using CNE.

Benchmark	Heterogeneous	Variable Heterogeneous	U-test	Effect Size
Ball Throwing	2931.04	**2288.98**	**2.57E-2**	**0.62960**
Full Adder	**201.90**	282.22	5.36E-2	**0.61220**

Table 7.25: Average fitness of variable heterogeneous ANNs trained using CGPANN.

Benchmark	Heterogeneous	Variable Heterogeneous	U-test	Effect Size
Ball Throwing	9.68	9.67	3.78E-1	0.55140
Full Adder	16.00	16.00	-	0.50000
Monks Problem 1 Train	1.226	**0.8226**	7.82E-1	0.50960
Monks Problem 1 Test	**6.139**	6.907	2.44E-1	0.56780
Two Spirals	**50.20**	58.50	**4.52E-9**	**0.84000**
Proben1: Cancer Train	1.086	1.086	4.42E-1	0.54420
Proben1: Cancer Test	3.126	3.126	9.75E-1	0.50200

Table 7.26: Average number of generations required to find optimal solutions using variable heterogeneous ANNs trained using CGPANN.

Benchmark	Heterogeneous	Variable Heterogeneous	U-test	Effect Size
Ball Throwing	**1000.76**	1025.80	5.06E-1	0.53880
Full Adder	698.10	**480.40**	2.52E-1	**0.56660**

7.8 Box and Whisker Plots

For high-level inspection, all of the previously given results are presented as box and whisker plots. Where the boxes represent the lower and upper quartile ranges and the whiskers are set as 1.5 times the interquartile range. Points greater than 1.5 times the interquartile range are labelled with a red '+' and points greater than 3 times the interquartile range labelled with a red '∘'. The median of each plot is given as a solid red line and the arithmetic mean is given as a dashed black line.

The average homogeneous fitness, given in Section 7.4, is also given as a dashed green line spanning the three homogeneous functions; step, Gaussian and logistic. The average homogeneous fitness is calculated as the arithmetic mean of the arithmetic means of the fitnesses achieved for the step, Gaussian and logistic function.

As a perfect fitness was often achieved for the Ball Throwing and the Full Adder benchmarks, the average number of generations to find the perfect solution are also given as box and whisker plots.

The CNE results are given in Figures 7.4-7.12 and the CGPANN results are given in Figures 7.13-7.21.

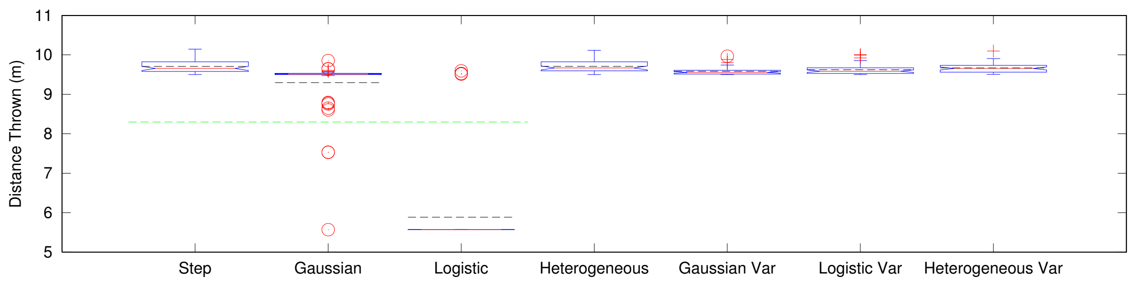


Figure 7.4: Fitnesses achieved from applying CNE to the Ball Throwing benchmark.

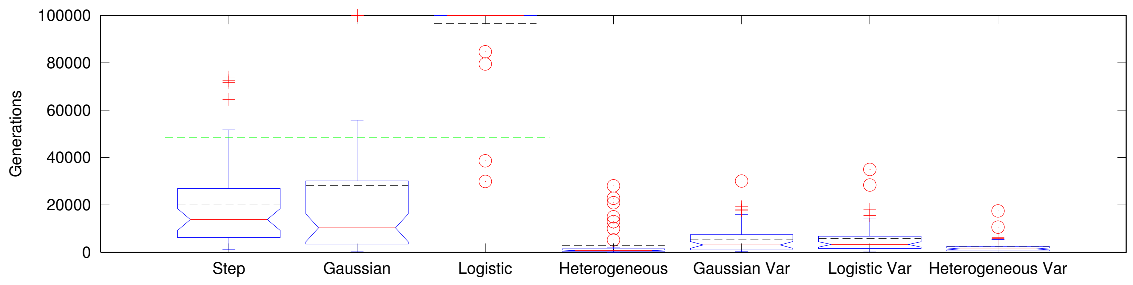


Figure 7.5: Generations required from applying CNE to the Ball Throwing benchmark.

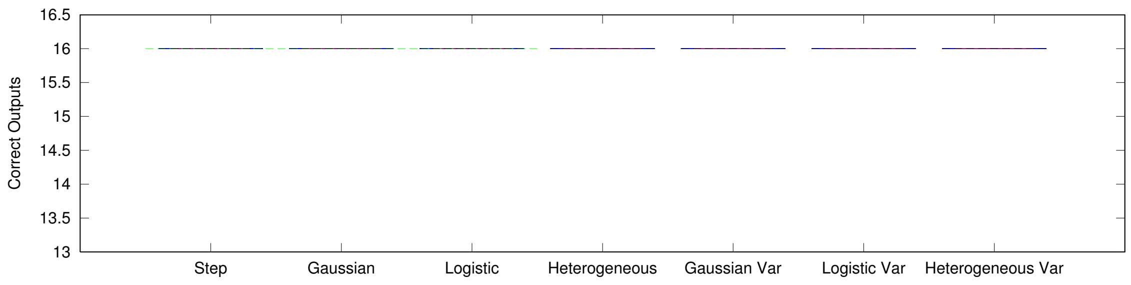


Figure 7.6: Fitnesses achieved in applying CNE to the Full Adder benchmark.

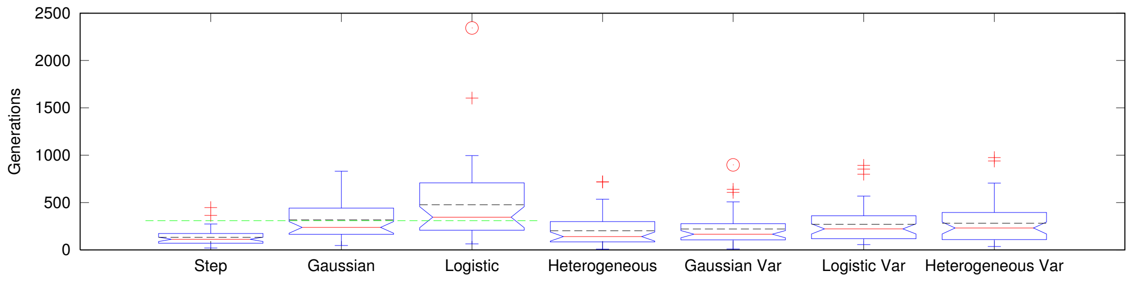


Figure 7.7: Generations required from applying CNE to the Full Adder benchmark.

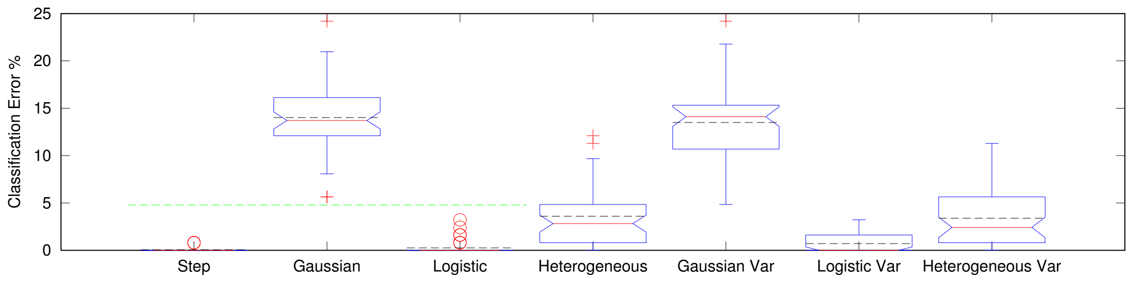


Figure 7.8: Fitnesses achieved in applying CNE to the Monks Problem 1 benchmark - Training.

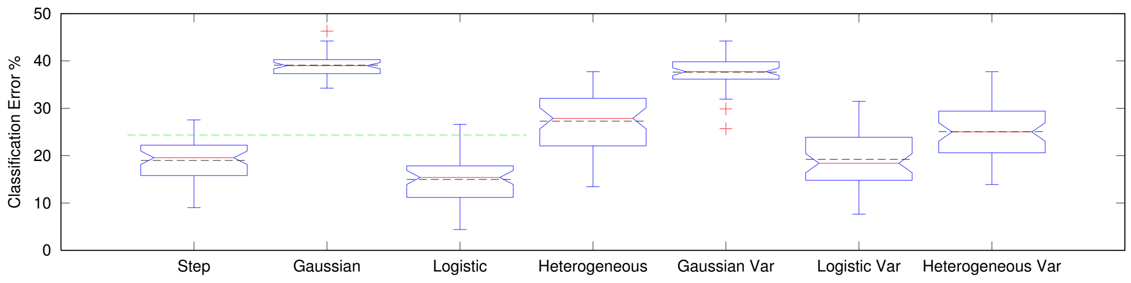


Figure 7.9: Fitnesses achieved in applying CNE to the Monks Problem 1 benchmark - Testing.

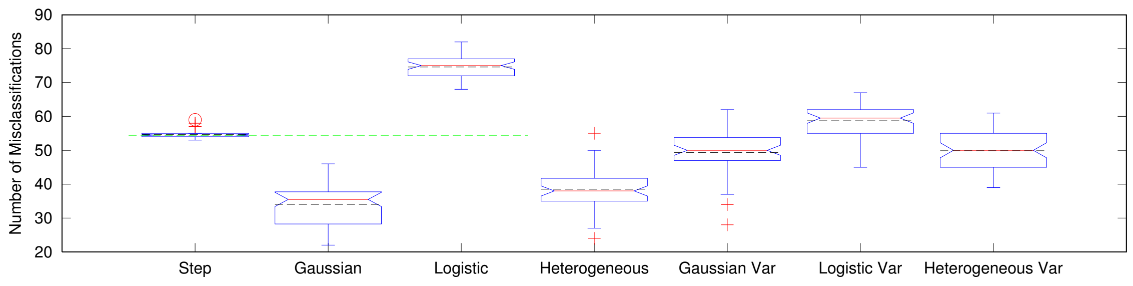


Figure 7.10: Fitnesses achieved in applying CNE to the Two Spirals benchmark.

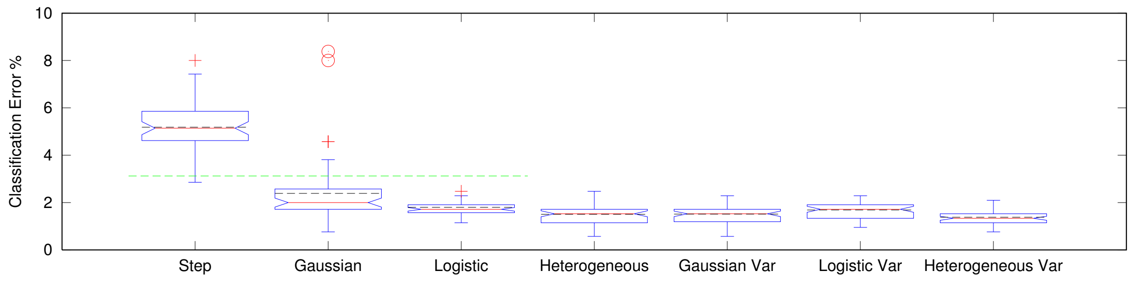


Figure 7.11: Fitnesses achieved in applying CNE to the Proben Cancer1 benchmark - Training.

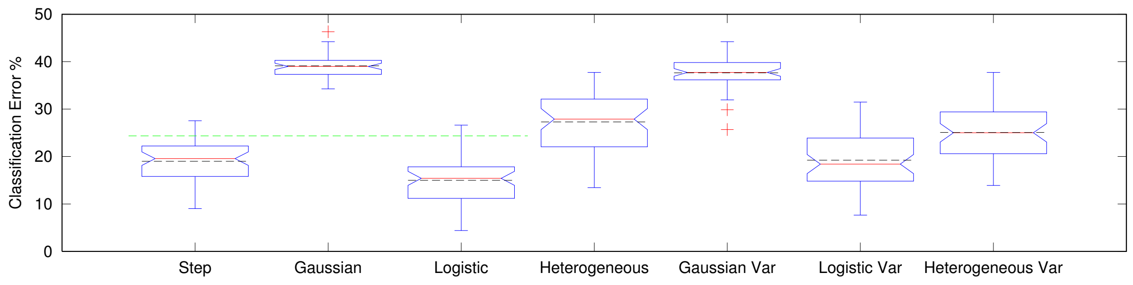


Figure 7.12: Fitnesses achieved in applying CNE to the Proben Cancer1 benchmark - Testing.

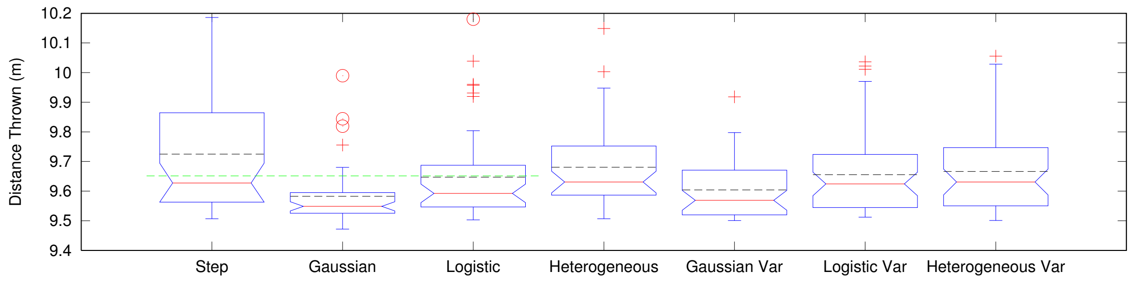


Figure 7.13: Fitnesses achieved in applying CGPANN to the Ball Throwing benchmark.

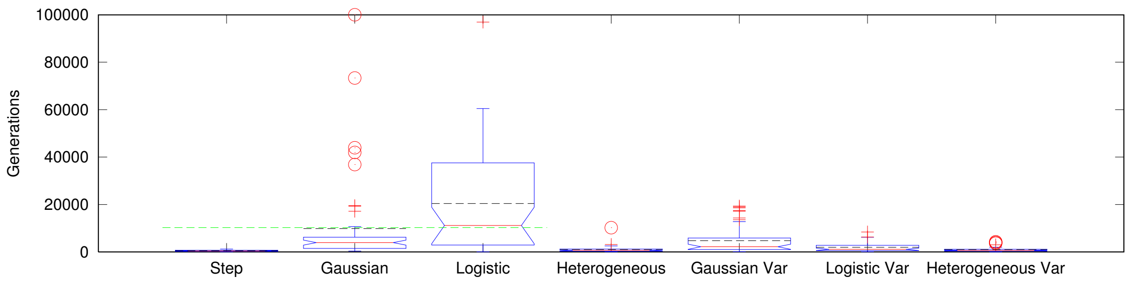


Figure 7.14: Generations required from applying CGPANN to the Ball Throwing benchmark.

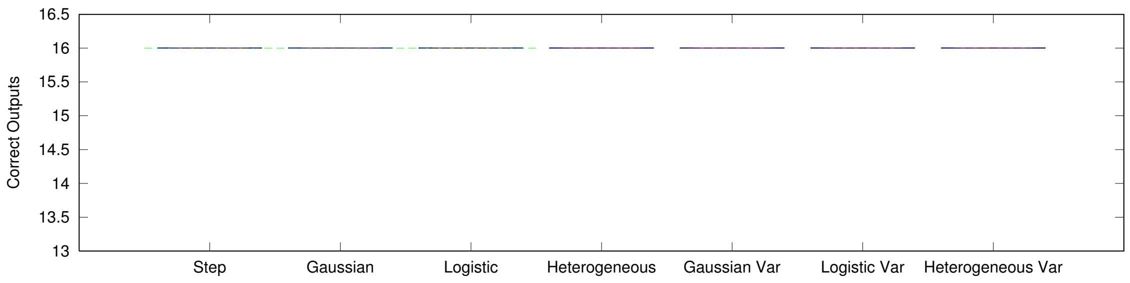


Figure 7.15: Fitnesses achieved in applying CGPANN to the Full Adder benchmark.

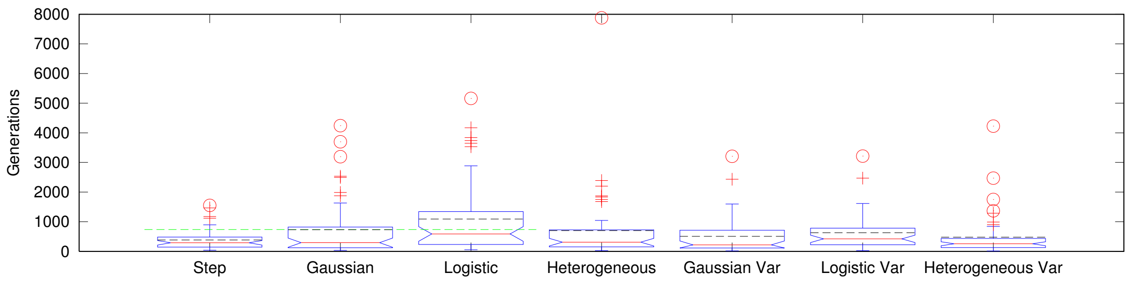


Figure 7.16: Generations required from applying CGPANN to the Full Adder benchmark.

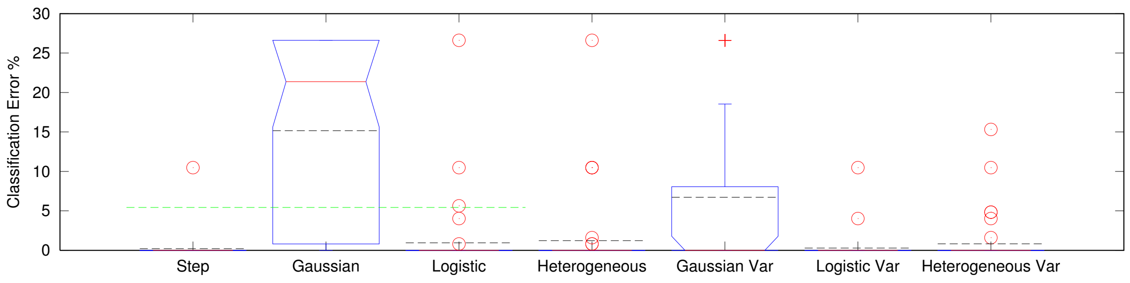


Figure 7.17: Fitnesses achieved in applying CGPANN to the Monks Problem 1 benchmark - Training.

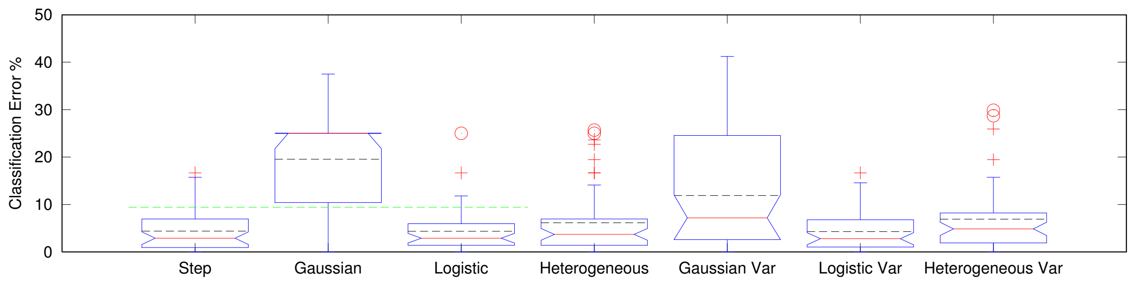


Figure 7.18: Fitnesses achieved in applying CGPANN to the Monks Problem 1 benchmark - Testing.

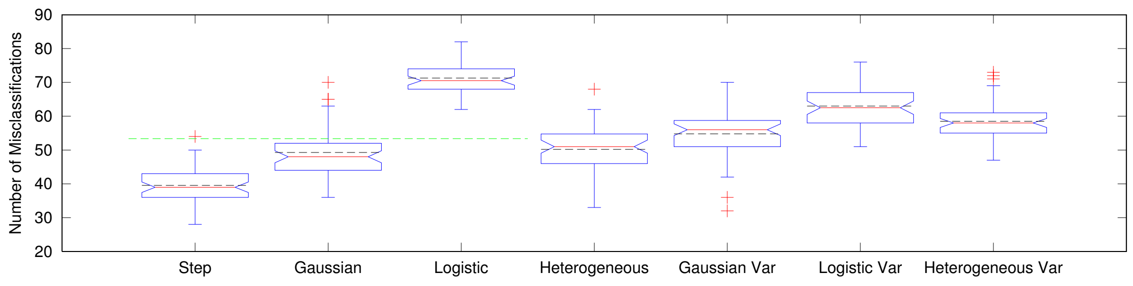


Figure 7.19: Fitnesses achieved in applying CGPANN to the Two Spirals benchmark.

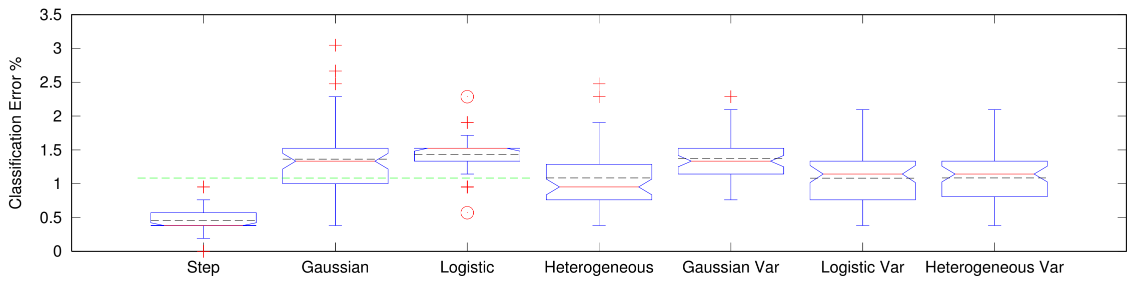


Figure 7.20: Fitnesses achieved in applying CGPANN to the Proben Cancer1 benchmark - Training.

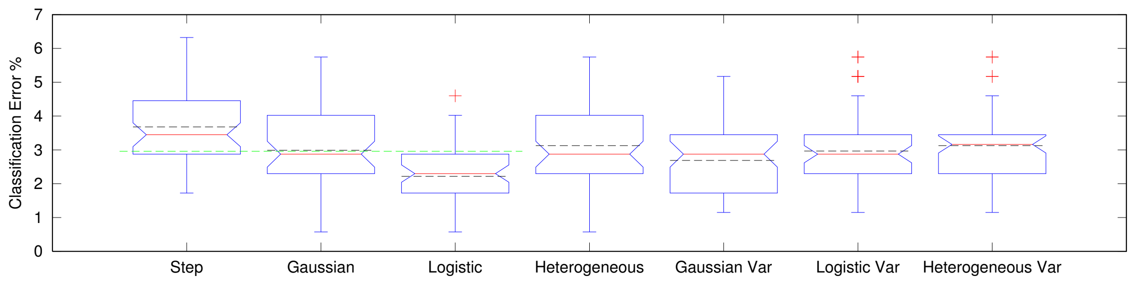


Figure 7.21: Fitnesses achieved in applying CGPANN to the Proben Cancer1 benchmark - Training.

7.9 Discussion

This chapter has empirically demonstrated that when using NE to create homogeneous ANNs, the choice of TF has a large impact on the fitness of the solutions found. It was also shown that no single TF produced the best results in all cases. Therefore when using homogeneous ANNs, one must accept possibly inferior results, or repeat training using a range of TFs. This chapter has also empirically demonstrated that, on average, evolving heterogeneous ANNs produces better results than the average of using each TF individually. Additionally, it has been shown that optimising parameters associated with each neuron's TF produces better results, on average, than using the typical fixed TFs. Interestingly however, a combination of evolving heterogeneous ANNs where each neuron's TF can also be adapted was shown not to produce better results than simply evolving heterogeneous ANNs of static TFs.

The results presented in Section 7.4 demonstrate that the choice of TF has a large impact on the effectiveness of NE. This is an intuitive result as it is likely that particular TFs are more or less suited to given tasks; it appears to mirror the 'No Free Lunch' theorem [310] but concerning TFs. However, although intuitive, it is a significant result as a user is unlikely to know, in advance of training, which TF is most suited to a given task. Additionally, it was seen that the most suited TF was also dependent upon the NE training method used; increasing the difficulty of knowing a suited TF ahead of training.

A further interesting, and unexpected result, is that in many cases the Heaviside step function was found to be the most effective TF; particularly for CGPANN. The step function was the original TF used by the 1943 McCulloch and Pitts neuron models [196]. The fact that the step function is incompatible with back propagation algorithms and is only suited to tasks with binary outputs is probably the reason other TFs have been favoured. Here, however, it has been shown that when using NE the Heaviside step function is still a suitable TF for contemporary ANNs; provided the task is compatible with binary outputs.

It is also worth noting that, at least when using NE, simply using the logistic sigmoid TF does not result in the best results in the majority of cases. This is significant as it appears from the literature that the logistics sigmoid is the most favoured TF.

The second experiment demonstrated that allowing NE to evolve heterogeneous ANNs produced better results, on average, than the average result obtained by evolving homogeneous ANNs of each TF. This is significant because, as the first experiment demonstrates, the choice of TF has a large impact on the effectiveness of the final ANNs. This, coupled with the fact there is no way of knowing which TF will be most suited to a given task in advance of training, puts homogeneous ANNs at a disadvantage. The importance of this result is heightened by the fact that the majority of NE methods are probably capable of evolving heterogeneous ANNs. The evolution of heterogeneous ANNs may even be further improved by the inclusion of additional TFs not considered here; and as NE places no restrictions on the types of TFs used, the range of possible TF is vast. It may even be the case that certain TFs complement each other while others may not.

A further result from the second experiment concerns the percentage of neurons which used each type of TF in the evolved heterogeneous ANNs. Interestingly, it was never the case that one type of TF strongly dominated the networks. If this had occurred, it would have indicated that evolution has found a particular TF to be the most suited toward the given task. There is, however, reasonable variation in the percentages of each type of TF used for CNE applied to the Full Adder and Two Spirals benchmarks and CGPANN applied to the Monks Problem1 benchmark. This demonstrates that in certain conditions, there is an evolutionary pressure to use a particular type of TF i.e. it is not simply random. The fact that a particular TF was not favoured in many cases may also indicate that evolution is combining the functionality of all the TFs, or there is no strong evolutionary pressure pushing away from a random (typically even) mixture.

The third experiment demonstrated that, in the majority of cases, using NE to optimise parameters associated with each neuron provided superior results compared to using non-parameterised TFs. This is also an important result as the inclusion of an additional gene (or genes) which alter the characteristics of each neuron's TF is again likely compatible with all NE methods.

It was also seen in the third experiment that the logistic sigmoid TF benefited from being parameterised significantly more than the Gaussian TF. This was despite the non-parameterised Gaussian TF producing worse results than the logistic TF overall i.e. there was more room for improvement. It therefore appears that the logistic TF strongly benefits from being parametrised. It could be the case however that this result is dependent on the range of values the TF parameters can take; which was not investigated here. For instance the Gaussian TF may work best over a much smaller or much larger range of values.

Combining the results that the Heaviside step function was found to be highly effective, and the fact the logistic sigmoid was shown to benefit the most from parameterisation, leads to an interesting insight. As the value of σ is increased for the logistic sigmoid function, it more closely approximates the Heaviside step function. This may explain the significant benefits seen from using the parameterised logistic sigmoid. It also highlights the benefit of allowing NE to evolve the "shape" of the TF.

In Sections 7.5 and 7.6 evolving heterogeneous ANNs and evolving parameters associated with each neuron's TF were individually demonstrated to be beneficial for NE. However, in Section 7.7 it was shown that when combined they produced no additional benefit, on average, over evolving heterogeneous ANNs with fixed TF parameters. It may be the case that using parameterised TFs and heterogeneous ANN produce similar benefits, and therefore no additional benefits are introduced by their combination. It could also be possible that performance depends subtly on evolutionary parameter settings so that when these methods are combined new parameter settings are required for optimum performance. It could also be the case that their combination does provide a benefit, but it was not observed due to the limited number of evaluations the experiments were ran over, or limited difficulty of the tasks. That is to say, there may well be a benefit on more challenging tasks, or towards the end of evolutionary searches where improving upon the current fitness becomes increasingly difficult. Further research would be required to determine this.

As mentioned in Section 7.2, homogeneous ANNs comprised of logistic or Gaussian TFs can be universal approximators. This means that with the correct topology and connection weight values, standard homogeneous ANNs are capable of implementing everything which heterogeneous ANNs can also implement. However, this fact says nothing about how efficiently standard homogeneous ANNs can implement certain configurations. Where here the term efficiently refers to the computational effort needed to configure the ANNs. Therefore the fact that ANNs are universal approximators is not enough to be considered useful. It is also necessary that the ANNs can easily be configured to a given task. To this end it appears that heterogeneous ANNs are, on average, more efficiently configurable using NE methods.

Although this chapter has demonstrated using NE to evolve heterogeneous ANNs it only used a limited set of TFs and optimised only a single parameter associated with each neuron over a small range of values. Further research should therefore investigate additional TFs and allow for more complex TFs described by multiple parameters; such as those described in [20]. Although certain TFs have been shown to be universal approximators this tells us nothing about how “trainable” / “evolvable” they are. For instance other TFs, such as the step function, were demonstrated to produce better results than other universal approximator TFs.

Finally, as results have been presented using two NE methods, CNE and CGPANN, it may appear appropriate to draw conclusions concerning their relative performance. However, drawing comparisons between topology and non-topology optimising methods is challenging. This is because in the non-topology case the task is to only find suitable connection weights, whereas in the topology optimising case the challenge is to find a suitable topology *and* suitable connections weights. Therefore the task assigned to the topology optimising case is more challenging; or at least different. Additionally, the performance of CNE is a strong function of the topology used; as is shown in Chapter 6. Whereas the performance of CGPANN is less of a function of the topology limits used; again shown in Chapter 6. Therefore, whether or not CNE outperforms CGPANN is also a strong function of the topology used by CNE. However, in this work the topology used by CNE was not optimised for each benchmark; and remained fixed throughout. Therefore, although the experimental methodology is sufficient to assess the relative benefits of evolving heterogeneous ANNs, it is not sufficient to assess the relative performances of CNE and CGPANN.

7.10 Summary

The use of NE to optimise the weights and topology of ANNs is well established and offers a number of advantages over traditional training methods; such as back propagation. However, the use of NE to create heterogeneous ANNs has so far been under researched and underutilised. This chapter has demonstrated the use of two methods for allowing NE to create heterogeneous ANNs. That is, selecting each neuron’s TF from a predetermined list of TFs or by optimising parameters associated with each neuron’s TF. The chapter has also shown that the effectiveness of using NE to train homogeneous ANNs is highly dependent on the selected TF. Using NE to optimise each neuron’s TF has been empirically demonstrated to alleviate this issue.

The results presented in this chapter are significant as the methods described for creating heterogeneous ANNs are likely to be compatible with all NE methods. That is, almost all NE method could benefit from evolving heterogeneous ANNs.

Chapter 8

Neutral Genetic Drift

This chapter investigates the role of explicit genetic redundancy in Cartesian Genetic Programming (CGP) and Cartesian Genetic Programming of Artificial Neural Networks (CGPANN). This is undertaken to allow a more thorough evaluation of the benefits of genetically inactive material in CGP than has previously been presented, and to identify if CGPANN also benefits from its presence. The investigation is also used to make an assessment as to whether explicit genetic redundancy is beneficial to NeuroEvolution (NE) generally.

8.1 Structure of this Chapter

Section 8.2 provides a background to Neutral Genetic Drift (NGD) with a focus on its influence on Genetic Programming (GP); specifically CGP. Section 8.3 presents an in-depth discussion of the literature surrounding the influence of genetic redundancy on CGP's evolutionary search.

Section 8.4 presents an investigation into the influence of NGD on CGP's search process. Section 8.5 investigates the effectiveness of an often used method for increasing the level of genetic redundancy in CGP. Section 8.6 then repeats the experiments presented in Section 8.4, but in the context of CGPANN.

A closing summary of the chapter is then given in Section 8.7.

8.2 Background

Sewall Wright first introduced the notion that natural evolution is searching over a space of possible solutions [311]. This space of solutions, commonly referred to as a fitness landscape, gives the fitness of each solution in relation to one another. This concept was later adopted by the machine learning literature, most prominently in the field of Evolutionary Algorithm (EA)s [174].

One of the insights gained from visualising the fitness landscape is the idea of local optima. These are sub-optimal areas of the search space where most, if not all, local solutions are worse than the current solution. As a result, escaping local optima is challenging. Typically mutation is thought of as the mechanism for escaping local optima, but mutation alone is often insufficient depending upon the local fitness landscape. Another mechanism for escaping local optima is provided by the presence of NGD. NGD was also first proposed in the domain of evolutionary biology by Motoo Kimura [158]. NGD is where mutations are made to a genotype which do not influence its semantics, but are preserved during selection and passed onto the next generation. This causes the position within the fitness landscape to change without requiring fitness improving mutations. Once in a new area of the fitness landscape, the possible solutions accessible through mutation are different to what was possible before the neutral mutation(s) had taken place. Therefore, when in local optima, the presence of NGD causes the number of possible solutions one mutation away to *vary*, thus aiding the escape from local optima.

An additional consequence of NGD is that it can also occur when the search is not trapped in local optima. This occurs when neutral mutations take place at the same time as fitness improving mutations (i.e. the inactive mutations ‘hitch-hike’). Although in this scenario NGD cannot be aiding the escape from local optima, it may still be influencing the evolutionary search.

It is important to understand and utilise the role of NGD in GP [165, 228], and EAs in general, as it represents a powerful mechanism for searching the fitness landscape. Additionally, as more challenging applications are investigated the benefits of NGD become more significant. This is because more challenging applications are likely to represent fitness landscapes with more numerous local optima. As NGD is thought to aid the escape from local optima, it becomes increasingly significant as the difficulty of the landscape increases.

In tree-based GP it is common to refer to inactive/redundant genes[1] as being those which are present in the genotype but which do not influence the semantics. An example of this form of genetic redundancy is where a section of the phenotype is multiplied by zero and hence does not contribute to the programs output; see Figure 8.1. In this chapter, this type of redundancy is referred to as *implicit genetic redundancy* as the genes are decoded into the phenotype, so in that sense are active, however they do not contribute to the semantics of the phenotype. Here NGD of implicitly redundant genetic material is referred to Implicit Neutral Genetic Drift (INGD) to distinguish it from NGD in general and other forms of NGD. Other GP methods which contain implicit genetic redundancy include Push Genetic Programming (Push-GP), which uses a stack based representation in which evolved programs manipulate program and data via stacks [262], since instructions can be left on the CODE or EXEC stacks which are ignored.

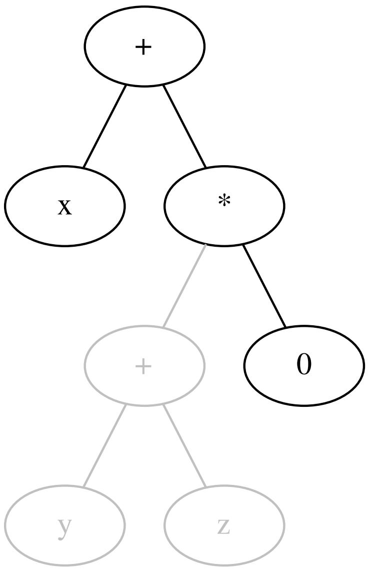


Figure 8.1: Implicit genetic redundancy in tree-based GP. Active nodes are shown in black, implicitly redundant nodes in grey.

Interestingly, other forms of GP contain a more explicit form of genetic redundancy. Here explicit genetic redundancy refers to genes which are removed during the decoding of genotypes into phenotypes. For instance, CGP, previously discussed in Chapter 3, contains inactive genes. Linear Genetic Programming (LGP) [33], a register based form of GP, also contains inactive genes which are not decoded into the phenotype. Both CGP and LGP employ a genotype-phenotype mapping which removes explicitly redundant

[1]The terms inactive genes and redundant genes are used interchangeably and refer to genes which do not influence the semantics and/or the phenotype.

genes [320] or structurally non-effective operations [33] respectively. This form of genetic redundancy is referred to as *explicit genetic redundancy.* Again as before, NGD acting on explicitly redundant genetic material is referred to as Explicit Neutral Genetic Drift (ENGD). Additionally, forms of GP like CGP and LGP also contain implicit genetic redundancy via the same mechanism described for tree-based GP, and so also exhibit INGD.

One of the difficulties in studying NGD is identifying which genes are genetically redundant. For instance, identifying implicitly redundant genes in tree-based GP is challenging [29] as it involves analysing the semantics of the program. However, identifying explicitly inactive genes in CGP is trivial as they are the genes not associated with the nodes connecting inputs to outputs. This makes CGP a useful tool for investigating the role of NGD. As is shown in this chapter, by using a range of restrictions to CGP, various aspects of INGD *and* ENGD can be isolated and studied independently.

Previous research investigating NGD in CGP [202,287] examined the effect of removing NGD by preventing the selection of child chromosomes over their parents if they have equal fitness. The rational being, that if neutral mutations cannot be passed on without fitness improvements, they cannot be used to escape local optima. However, this methodology prevents all forms of NGD and any other forms of non-genetic redundancy aiding the escape from local optima. The experiments undertaken did not isolate the benefit of ENGD; the type of NGD present in CGP but not in tree-based GP. Additionally, the methodology did not consider potential benefits of NGD other than escaping local optima; such as NGD occurring at the same time as fitness improving mutations to active genes.

Other studies into the role of genetic redundancy for CGP investigated the effect of increasing the levels of explicit genetic redundancy [207]. This was achieved by increasing the number of available nodes which increased the proportion of inactive genes. The work reported that increasing the level of explicit genetic redundancy, by increasing the number of available nodes, consistently improved the evolutionary search. However, it is clear that this tread cannot continue indefinitely since it implies that using an infinite number of nodes would solve a given task in zero time. An alternative explanation, other than the increase in explicit genetic redundancy, has also since been presented. It speculates that the reason using high numbers of available nodes is beneficial, is because it compensates for the length bias present in CGP [82, 84]; see Section 3.9.2.

The contribution of this chapter is a substantial evaluation of the role of NGD in CGP.

The benefits of both INGD and ENGD are rigorously evaluated including aspects other than aiding the escape from local optima. Additionally, past experiments are repeated and extended in order to confirm previous results whilst offering further insights. Although the focus of this chapter is NGD in CGP, the results presented are relevant to other GP and EA techniques which also contain genetic redundancy.

The chapter also investigates the role of NGD in CGPANN. As was discussed in Chapter 2, the presence and effect of NGD in NE methods have so far not been considered in the literature. As NE algorithms are based on EAs, which are thought to benefit from NGD, it appears likely that NE would also benefit. Additionally, it appears that many NE methods also contain explicit genetic redundancy; such as GeNeralized Acquisition of Recurrent Links (GNARL) [15], Evolutionary Programming Artificial Networks (EPNet) [314] and Cooperative Co-evolution Model for evolving Artificial Neural Networks (COVNET) [75]; see Section 2.6.

8.3 Redundancy in CGP

The role of redundancy in EAss, and its impact on neutrality, has been widely studied: [22, 23, 57–59, 68, 69, 73, 123, 124, 243, 306] and recently reviewed [74]. There are many forms of redundancy in GP. For instance, as has been previously discussed, genes can be explicitly or implicitly redundant. It is possible for many genotypes to produce the same phenotypes; this creates redundancy in the genotype-phenotype mapping. It is also possible for many genotypes to produce phenotypes with the same semantics. This creates redundancy in the genotype-semantic mapping. Additionally, the solution spaces themselves can contain redundancy with many possible solutions being awarded equal fitness; even if they represent very different solutions. Finally, redundancy can occur in the imprecision of implementing abstract computer programs on real world hardware. For instance, the imprecision of floating point operations causing two differing solutions to be awarded the same fitness.

CGP chromosomes contain genes which are ignored in the construction of the phenotype. These inactive genes are therefore genetically redundant. However, such redundant genes can later become active via mutation. For instance a mutation could cause an active node to connect to a previously inactive node, thus making the inactive node active. This type of genetic redundancy is referred to as explicit genetic redundancy because the genes are explicitly removed during the genotype-phenotype decoding stage. As discussed, implicit genetic redundancy is also possible in CGP where a section of the *phenotype* has no influence on its semantics.

It is worth noting that it has been previously claimed that the presence of explicit genetic redundancy in CGP helps the evolutionary search on "needle in the haystack" type problems [319]. However, this was refuted and an alternative explanation given [44] where it was concluded that explicit genetic redundancy does not aid the evolutionary search on needle-in-haystack type problems. However, this conclusion said nothing about the merits or demerits of explicit genetic redundancy in CGP generally i.e. when applied to tasks which are not needle-in-haystack problems. That is to say, explicit genetic redundancy could be, and is indeed shown here to be, generally beneficial.

Additionally, in recent work by Z. Vassicek [286] the role of neutral mutations were investigated in relation to the application of CGP to the minimisation of Boolean circuits. In their work three criteria for the prevention of neutral mutations were proposed: "*(1) inactive gates are never modified; (2) it is not possible to connect an active gate (or primary output) to an inactive gate; (3) the gene which encodes the connection of the second input of a single-input gate is never mutated.*". However, their second criterion means their methodology is not solely investigating the benefit of neutral mutations. This is because connecting an active node to an inactive node does not constitute a neutral mutation; as no inactive genes have been mutated (changed allele). Vassicek found their proposed method for preventing neutral mutations, using the three described constraints, showed no impact on the results of CGP applied to minimising Boolean circuits. They therefore concluded that neutral mutations were of no benefit to CGP. However, in Vassicek's work, the growth of the phenotype size resulted in a worse fitness; as their application was circuit minimisation. Note that in order for neutral mutations to be able to influence the search, inactive genes must be allowed to become active; thus often initially increasing the size of the solution. As in Vassalages work the growth in phenotype size was penalised by the fitness function used, it is not surprising that the prevention of neutral mutation was seen to have little influence on the search. Therefore, Vassicek's work does not demonstrate that explicit genetic redundancy is of no use generally; for instance when not actively penalising any instance of growth in solution size.

The remainder of this section discusses NGD with a focus on CGP followed by a discussion of previous work which has attempted to increase the levels of genetic redundancy in CGP. For a further discussion of genetic redundancy and NGD in CGP see [202, 287, 318, 320].

8.3.1 Neutral Genetic Drift

The idea that NGD might be beneficial was first proposed in the field of biological evolution in a highly influential paper *Neutral Theory of Molecular Evolution* [158]. The arguments given apply to NGD caused by both implicit and explicit genetic redundancy.

NGD describes the change in inactive genetic material during evolution. The *neutral* refers to it having no effect on the semantics of the phenotype, the *genetic* refers to it acting upon the genetic material and the *drift* refers to the genetic material drifting i.e. it is changing in an unguided manner. One of the reasons NGD is thought to be important is because it can lead to genetically (not phenotypically) diverse populations even when trapped in local optima.

Consider an evolutionary run which has reached a local optimum. Over time the population will converge on the best found solution. This is because, when trapped in a local optimum, most mutations will create worse children than their parents and therefore only children which are genetically similar to their parents are selected and survive. This causes the entire population to become genetically similar. If all members of the population are genetically similar, then the number of possible solutions which can be reached via mutation (or crossover) in one generation is vastly reduced. Thus escaping the local optimum becomes increasingly unlikely.

When redundant genes are present, the active genes will still converge but there is no pressure for the inactive genes to converge; as they do not affect fitness and are not guided by selection. This causes the active genetic material to converge, but the inactive genetic material to randomly *drift*. This results in a population with similar active genes but differing inactive genes. As mutation (or crossover) can result in inactive genes becoming active, the number of solutions accessible in one generation is therefore much larger, and more dynamic, than without the inactive genes; thus making the escape of the local optima more probable. For this reason, inactive genes are thought to be beneficial due to the process of NGD aiding the escape of local optima.

An alternative view on the influence of NGD is through the effect redundant genes have on the search space. As redundant genes, by definition, do not affect the fitness, they create plateaus of equal fitness in the search space. These plateaus can be randomly *drifted* across when mutations of inactive/redundant genes occur. Depending upon the position on the plateau, different solutions may be possible via mutation. For instance, an inactive node may produce behaviour (a) if made active in one area of the search space or behaviour (b) if made active in another area of the search space. Again this can be seen to cause the possible solutions one mutation away to vary when trapped in local optima.

Since CGP typically uses a (1+λ)-ES, all the children are created by mutating the single selected parent. Usually using such a greedy strategy would result in a search becoming very easily trapped in local optima. However, as has been discussed, CGP contains inactive genes which are subject to NGD. When in a local optima, any children created from the parent via mutations to inactive genes alone will have the same fitness as the parent; as they are semantically identical. An important aspect of the CGP algorithm is that children are selected over parents if they have equal fitness. Therefore the solution selected may have the same fitness, but different inactive genetic material to the parent. This causes the possible solutions accessible through mutation to change from generation to generation and thus help escape the local optima.

A further behaviour of NGD is its ability to alter inactive genes even when not trapped in local optima. For instance, when redundant genes are mutated at the same time as beneficial fitness improving mutations are made to active genes. In this case the changes to the inactive genes are also passed on to the selected child. This results in the possible solutions one mutation away being different to what they would have been had the neutral mutation not taken place. This effectively means that the plateaus in the search space can be drifted across even when not trapped in a local optimum.

As the genotypes of other forms of GP also contain implicit genetic redundancy, they too benefit from INGD. Likewise, CGP genotypes also contain implicit genetic redundancy and so also benefit from INGD.

In the wider field of EAs, it is often argued that the benefit of genetic redundancy is related to its ability to protect from damaging mutations [218, 282, 291]. For instance, when using a (μ, λ)-ES each new generation is entirely comprised of children; no parents. If the mutation rate were sufficiently high, then the majority of the children would be drastically different from their parents. In this case good solutions may be easily lost from one generation to the next. Mutations to inactive genes do not change the phenotypes semantics. Therefore inactive genes help regulate the amount of semantic altering mutations. This is thought to help prevent good solutions from being so easily lost. However, using redundant genes to absorb excess mutations appears to be equivalent to using a lower mutation rate [160]. It may also be the case that a varying level of genetic redundancy results in a varying number of semantic altering mutations. However, this could be simply achieved by using different mutation methods; for instance a variable mutation rate. Regardless of whether this aspect of genetic redundancy is beneficial, CGP typically uses a $(1 + \lambda)$-ES so there is no danger of losing the best solution from one generation to the next. Additionally, in the experiments reported in this chapter, probabilistic mutation is used; which already allows a variable number of mutations to occur; see Section 3.4.

8.3.2 Increasing Genetic Redundancy

It has been previously reported [207] that high levels of explicit genetic redundancy (95%) produce the best evolutionary search for CGP. In their work the percentage of inactive nodes was controlled by varying the number of available nodes. As CGP typically uses a fraction of the available nodes, increasing the number of available nodes has the effect of increasing the percentage of inactive nodes (genetic redundancy). The work showed that by increasing the number of available nodes the percentage of active nodes decreased and the effectiveness of the evolutionary search increased. The paper concluded that the increased genetic redundancy was responsible for the improved evolutionary search. Unfortunately, the paper did not expose the full relationship between the effectiveness of the evolutionary search and the percentage of active nodes. For instance, from their presented results one could be left with the impression that increasing the number of available nodes always improved the convergence time. However, intuitively this cannot be true since it implies that using an unbounded number of nodes would solve a given task in the first generation; a clear falsehood since the initial population is randomly initialized. Therefore, there should be a point at which increasing the number of nodes no longer improves the evolutionary search.

It has also been shown that CGP is naturally biased to phenotypes of a given size [82, 84]; typically a small percentage of the available nodes. Although this does result in high levels of explicit genetic redundancy, it was shown that the active nodes are concentrated towards the inputs (low nodes indexes) and the inactive nodes towards the outputs (high nodes indexes). This results in few redundant genes *between* the active genes limiting the possible benefits of the redundant genes; as the vast majority of inactive genes are extremely unlikely to ever be made active. It was speculated in [82,84] that using very high numbers of available nodes, such as in [207], may be effective because it compensates for the lack of inactive nodes between the active. Due to using high numbers of nodes increasing the likelihood of there being some inactive nodes between the active. Therefore, an increased fitness is seen as the number of available nodes is increased.

8.4 Investigating Neutral Genetic Drift in CGP

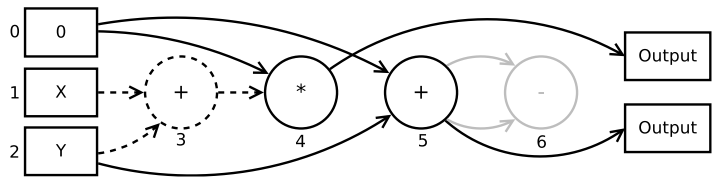
Investigating INGD is challenging in tree-based GP due to the difficulty in identifying implicitly redundant genes. For instance, in Figure 8.1, the addition of y and z is only redundant because it is then later multiplied by zero. Although likely possible, detecting all instance of this type of redundancy would be very difficult and computationally expensive. Conversely, investigating the role of explicit genetic redundancy present in CGP is much simpler. This is because the explicitly inactive genes can easily be identified; they are the genes not associated with the active nodes connecting inputs to outputs. It is this difference which is central to how CGP can be used to investigate NGD in general.

As previously discussed, NGD can be separated into two forms, namely INGD and ENGD. Additionally NGD poses two possible advantages/behaviours, the ability to help escape local optima by following plateaus in the search space and the ability to follow the same plateaus simultaneously with positive mutations to active genes. Interestingly, these two behaviours are related to the transfer of neutral mutation from parents to children. For instance, in order for NGD to help escape local optima, neutral mutations must be able to be passed on when the child's fitness is equal to that of its parents. Additionally, for NGD to be able to change the inactive genes alongside positive mutations, neutral mutations must be passed on when the child's fitness is greater than its parents. Therefore the behaviour of NGD, implicit or explicit, is determined by *when* neutral mutations are passed on.

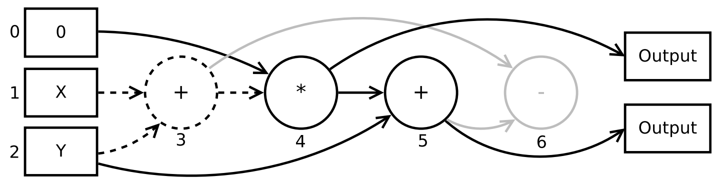
This means that there are four possible behaviours of NGD in total: ENGD occurring with fitness improvements, ENGD occurring without fitness improvements, INGD occurring with fitness improvements and finally INGD occurring without fitness improvements. These four behaviours are discussed in more detail for clarity.

In all the following figures presented, Figures 8.2 - 8.5, active nodes are given as solid black, explicitly inactive nodes as a solid grey and implicitly inactive nodes as a dashed black. Additionally (a) represents a given parent and (b) represents a possible child which would have been selected over the given parent. The chromosomes of parent and child are also given with the function genes underlined and the differences highlighted in bold. Also note that in all cases the first input to the chromosomes is set as zero, thus facilitating implicit genetic redundancy.

Figure 8.2 depicts an example of ENGD taking place at the same time as a fitness improving mutation. As can be seen in Figure 8.2, the child differs from the parent in two regards; both an explicitly inactive gene and an active gene have been mutated. This has resulted in a fitter solution which is why it is selected and therefore why the neutral mutation was passed on to the next generation.



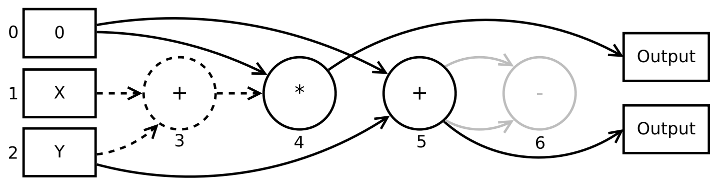
(a) Parent: $\underline{0}$12 $\underline{2}$03 $\underline{0}$02 $\underline{1}$55 4 5, fitness $= x$



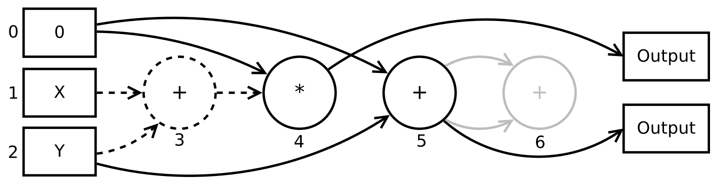
(b) Selected Child: $\underline{0}$12 $\underline{2}$03 $\underline{0}$**4**2 $\underline{1}$**3**5 4 5, fitness $> x$

Figure 8.2: Example of ENGD occurring with a fitness improvement. Active genes are given in bold, explicitly inactive in grey and implicitly inactive in dashed.

Figure 8.3 depicts ENGD taking place without a fitness improvement. In this case the selected child differs from the parent in one mutation to an explicitly inactive gene; again marked in bold in the chromosome. This mutation resulted in a child of equal fitness to the parent which is then selected because it is not worse than the parent. This behaviour is only possible when children are selected over their parents when they have equal fitness. This aspect of NGD is thought to help the escape of local optima.



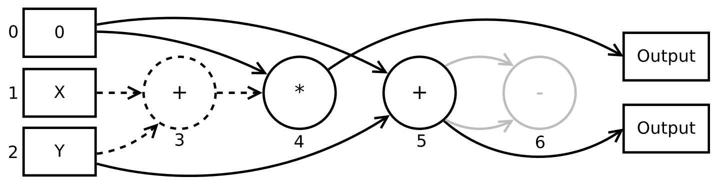
(a) Parent: $\underline{0}$12 $\underline{2}$03 $\underline{0}$02 $\underline{1}$55 4 5, fitness $= x$



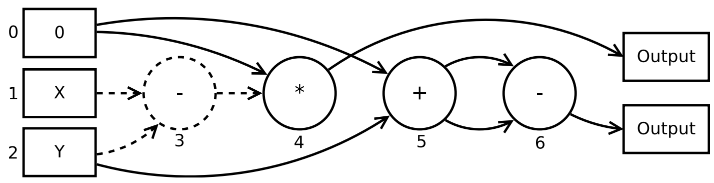
(b) Selected Child: $\underline{0}$12 $\underline{2}$03 $\underline{0}$02 $\underline{\mathbf{0}}$55 4 5, fitness $= x$

Figure 8.3: Example of ENGD occurring without a fitness improvement. Active genes are given in bold, explicitly inactive in grey and implicitly inactive in dashed.

Figure 8.4 depicts INGD taking place at the same time as a fitness improving mutation. In this case the child differs from the parent due to one mutation to an implicitly inactive gene and one mutation to an active gene. These mutations resulted in a child of greater fitness than the parent which is why it is selected. In this case it can also be seen that the active mutation resulted in inactive genes becoming active. However this does *not* constitute NGD as no explicitly neutral genes have changed. Making inactive genes active is not an instance of NGD.



(a) Parent: $\underline{0}$12 $\underline{2}$03 $\underline{0}$02 $\underline{1}$55 4 5, fitness $= x$

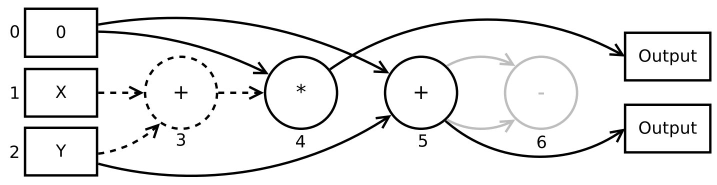


(b) Selected Child: $\underline{\mathbf{0}}$11 $\underline{2}$03 $\underline{0}$02 $\underline{1}$55 4 **6**, fitness $> x$

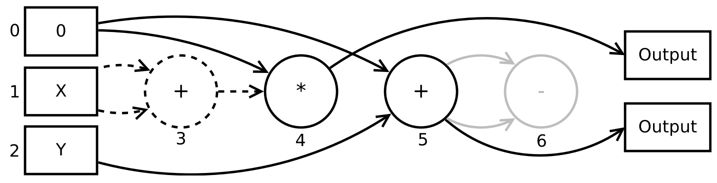
Figure 8.4: Example of INGD occurring with a fitness improvement. Active genes are given in bold, explicitly inactive in grey and implicitly inactive in dashed.

Finally, Figure 8.5 depicts INGD taking place without a fitness improving mutation. In this case the child differs from the parent due to one mutation to an implicitly inactive gene. Again, in this case the child is selected over the parent because it has equal fitness.

This behaviour of INGD is thought to help escape local optima.



(a) Parent: $\underline{0}$12 $\underline{2}$03 $\underline{0}$02 $\underline{1}$55 4 5, fitness = x



(b) Selected Child: $\underline{0}$**11** $\underline{2}$03 $\underline{0}$02 $\underline{1}$55 4 5, fitness = x

Figure 8.5: Example of INGD occurring without a fitness improvement. Active genes are given in bold, explicitly inactive in grey and implicitly inactive in dashed.

Now the four behaviours of NGD have been defined, methods for isolation are introduced. For reference, these methods are summarised in Table 8.1 along with the aspects of NGD they isolate. The presence of types of non-genetic redundancy is also included in Table 8.1. Where non-genetic redundancy comprises redundancy in the phenotype-semantic mapping (i.e. two separate solutions which exhibit the same behaviours) and redundancy in the semantics-fitness mapping (i.e. two distinct behaviours being awarded the same fitness). These types of redundancy are separated as they are not genetic, but do influence the search.

The first method used for comparison is regular CGP; CGP with no alterations. Regular CGP is listed in Table 8.1 where it can be seen that it exhibits all of the behaviours of NGD discussed as well as all other forms of redundancy.

The second method, also given in Table 8.1, is to only mutate active genes, never inactive. This method is the same as standard CGP with one exception; explicitly inactive genes are never mutated. This is easily achieved for CGP as explicitly inactive genes are those genes not associated with the path of nodes connecting inputs to outputs. Once mutations obey this constraint, both behaviours of ENGD are prevented but both behaviours of INGD and other forms of redundancy are retained. This is because it completely prevents any change to explicitly inactive genes.

The third method, again given in Table 8.1, is to only select fitness improvements. This method is the same as standard CGP except that children are only selected over their parents *iff* they have an *improved* fitness; not equal or worse. This alteration there-

Table 8.1: Restrictions made to CGP used in order to investigate NGD.

Behaviour	Regular CGP	Only Mutate Active Genes	Only Fitness Improvements	Only Mutate Active Genes and Only Fitness Improvements
ENGD with fitness improvement	Yes	No	Yes	No
ENGD without fitness improvement	Yes	No	No	No
INGD with fitness improvement	Yes	Yes	Yes	Yes
INGD without fitness improvement	Yes	Yes	No	No
Non-genetic redundancy	Yes	Yes	No	No

fore prevents both ENGD and INGD occurring unless it is accompanied by a fitness improving mutation. Only selecting child chromosomes with fitness greater than their parents prevents NGD, implicit and explicit, and other forms of redundancy from aiding the escape from local optima. It does however leave the possible benefit of NGD taking place alongside fitness improving mutations.

The final method combines the previous two described methods; only mutating active genes and only selecting fitness improvements. This removes all benefits of NGD, implicit and explicit, and non-genetic redundancy except specifically INGD taking place simultaneously with fitness improvements.

Using these described restrictions to CGP, a series of experiments are proposed and presented which isolate various aspects of NGD.

During the investigations, the following CGP parameters are used: 50 runs, 100,000 generations, a (1 + 4)-ES, 3 percent probabilistic mutation, no crossover, a node arity of two, 100 nodes and a function set dependent upon the benchmark. The benchmarks used comprise: Double Pole (+,−,×,÷), Full Adder 4 bit (AND, NAND, OR, NOR), Multiplier 4 bit (AND, NAND, OR, NOR), Nguyen 10 (+,−,×, ÷, sin, cos, exp, log), Pagie 1 (+,−,×,÷), Parity 8 Bit (AND, NAND, OR, NOR) and the Tower Problem (+,−,×, ÷, sin, cos, exp, log). All of these benchmarks are described in Appendix A.

8.4.1 Experiments

The first experiment is to isolate the overall benefit of ENGD. This is achieved by comparing the difference in performance between regular CGP and CGP when only active genes are mutated. These two methods differ only in the presence of ENGD and therefore isolate the benefit of ENGD. This comparison is given in Table 8.2 for the benchmarks described in Section A.

Table 8.2: Comparing regular CGP to only mutating active genes in order to isolate the benefit of ENGD. In all cases a lower fitness represents a better search.

Benchmark	Regular CGP	Only Mutate Active Genes	U-Test	Effect Size
Double Pole	32517	58273	3.97E-3	0.65840
Full Adder 4 bit	352.92	400.06	1.16E-2	0.64660
Multiplier 4 bit	301.84	314.86	1.29E-2	0.64440
Nguyen 10	0.78	1.60	3.88E-1	0.54780
Pagie 1	73.13	105.97	1.32E-2	0.64400
Parity 8 Bit	45.08	68.22	1.01E-6	0.78380
Tower Problem	174404	241640	1.66E-11	0.89080

As can be seen in Table 8.2, in all cases the presence of ENGD resulted in a more effective evolutionary search. Additionally, in all but one case the difference resulted in a medium or large effect size with statistical significance. This clearly demonstrates that the presence of ENGD is greatly beneficial to the evolutionary search; at least in the case of CGP. It can therefore be concluded that the ability of ENGD to escape local optima, or the ability to mutate explicitly inactive genes at the same time as positive mutations, or both, is beneficial to the evolutionary search. Additionally, as INGD is present in both of the methods used for comparison, it can be seen that the benefit of ENGD is additional to that already provided by INGD.

The second experiment investigates whether redundancy, both implicit and explicit genetic redundancy and other non-genetic redundancy, provides an ability to aid the escape from local optima. This is achieved by comparing the difference in performance between regular CGP and CGP in which only fitness improvements are allowed to be selected i.e. children are only selected over parents *iff* they are fitter. These two methods differ only in that the former (regular CGP) has the ability to pass on neutral mutations without the presence of beneficial mutations; a requirement for use in escaping local optima.

This experiment is similar to that undertaken in [202,287]; however there are important differences. For instance, the range of benchmarks is much larger and statistical analysis is used to confirm any differences. Additionally, in [202,287] the experiment was proposed to remove all benefits of NGD. However, as discussed, it actually only removes the benefit of NGD, and other forms of redundancy, *aiding the escape from local optima.* For instance NGD can still occur when neutral genes are mutated along with beneficial mutations to active genes. Here the results of the experiment are analysed with regard to only what is isolated.

As can be seen in Table 8.3, in all cases allowing NGD and other forms of redundancy to escape local optima produces a superior evolutionary search with statistical significance. Additionally, in all but two cases this difference resulted in a large effect size; with the other two resulting in one medium effect size and one small. It can therefore be concluded that the ability of NGD and other forms of redundancy to assist in escape from local optima is a major advantage; at least in the case of CGP.

Table 8.3: Comparing regular CGP to only selecting fitness improvements in order to isolate the benefit of NGD and other forms of redundancy aiding the escape from local optima. In all cases a lower fitness represents a better search.

Benchmark	Regular CGP	Only Fitness Improvements	U-Test	Effect Size
Double Pole	32517	52225	2.62E-2	0.61920
Full Adder 4 bit	352.92	698.66	1.83E-17	0.99360
Multiplier 4 bit	301.84	402.68	4.05E-17	0.98820
Nguyen 10	0.78	1.77	6.42E-3	0.65280
Pagie 1	73.13	152.56	1.21E-8	0.83080
Parity 8 Bit	45.08	117.98	6.89E-18	1
Tower Problem	174404	218169	4.58E-12	0.90160

The third experiment investigates the benefit of ENGD when not being used to escape local optima. For instance, the benefit of ENGD occurring at the same time as other beneficial mutations. This is achieved by comparing the difference between only allowing fitness improvements to be passed on with only allowing mutations to active genes whilst also only allowing fitness improvements to be passed on. For clarity this is a comparison between the methods in the final two columns of Table 8.1. These two methods differ only in the ability of ENGD to occur in combination with fitness improving mutations to active genes.

As can be seen in Table 8.4, removing the benefit of ENGD occurring alongside beneficial mutations consistently produced inferior results with statistical significance in all but one case. However, in the majority of cases, all but two, the effect size was small. Therefore it appears that although ENGD does have benefit other than assisting in the escape from local optima, this benefit is comparatively small.

Table 8.4: Selecting only fitness improvements compared with only allowing mutations to active genes while also only selecting fitness improvements. This isolates the benefits of ENGD other than aiding the escape of local optima. In all cases a lower fitness represents a better search.

Benchmark	Only Fitness Improvements	Only Mutate Active Genes, Only Fitness Improvements	U-Test	Effect Size
Double Pole	52225	69017	3.30E-2	0.62160
Full Adder 4 bit	698.66	745.36	3.33E-2	0.62380
Multiplier 4 bit	402.68	410.04	4.46E-1	0.54440
Nguyen 10	1.77	4.70	7.11E-3	0.65480
Pagie 1	152.56	175.88	1.72E-1	0.57940
Parity 8 Bit	117.98	120.80	2.74E-2	0.62780
Tower Problem	218169	260060	1.90E-6	0.77640

8.4.2 Discussion

From the experiments presented, three properties of NGD and non-genetic redundancy in CGP have been identified. It has been shown that ENGD is greatly beneficial to CGP. It has been shown that the ability of NGD, and non-genetic redundancy, to escape local optima represent the most beneficial aspect of redundancy. Finally, it has been shown that ENGD occurring when not trapped in local optima also provides a benefit to the search.

For many years it has been thought that ENGD is highly beneficial to CGP. However, previous work investigating the effect of ENGD did so by preventing both INGD, ENGD, along with non-genetic redundancy, from escaping local optima [202, 287]. Therefore previous research did not isolate the effect of ENGD, nor did it consider benefits other than escaping local optima. In the work presented here the role of specifically ENGD has been isolated and demonstrated to be of great benefit. Although this means that the hypothesis of ENGD being beneficial to the evolutionary search remains the same, the evidence for believing this has changed.

It has also been identified that ENGD provides a small benefit to the evolutionary search when taking place alongside positive mutations to active genes. This is an aspect of ENGD not previously considered. A possible explanation for this benefit could lie in the effect it has on the positioning within the search space. Mutating inactive genes along with active genes could cause the semantics of the solution to alter only slightly but place the solution in a very different area of the search space; in terms of what solutions are reachable in the next generation. Although highly speculative, this could be causing a wider area of the solution space to be sampled during the evolutionary search; as it is not limited to the solutions which can only be reached (or easily reached) via successive fitness improving mutations to active genes.

An important insight from the experiments presented is that it is now known that the benefits of ENGD are *additive* to INGD and non-genetic redundancy; at least in the case of CGP. This was shown to be the case as preventing ENGD, whilst preserving INGD and non-genetic redundancy, produced a worse evolutionary search. This is important as it demonstrates that the presence of specifically explicit genetic redundancy is advantageous to the evolutionary search. Therefore other GP methods which also contain explicit genetic redundancy, such as LGP [33], could also be benefiting from ENGD. Interesting future work would be to repeat the experiments presented here using other GP methods which contain explicit genetic redundancy. This would identify if the results are general or specific to CGP. In addition, the work suggests that if explicitly neutral genes could be added to other GP methods, it could lead to a more effective search. For instance, if "connection switch genes"[2] were introduced on links between nodes in tree-based GP that could connect or disconnect sub-trees, then all genes in switched off sub-trees would become explicitly inactive. This could enable tree-based GP to benefit from ENGD.

The fact that the presence of ENGD in CGP was shown to provide a benefit indicates that the amount of genetic redundancy provided implicitly is not sufficient to fully utilise NGD. Therefore it may be the case that other GP methods, such as tree-based GP, do not fully utilise NGD. Although the amount of useful explicit genetic redundancy present in CGP is unlikely to be optimal, the presence of explicit genetic redundancy is providing a further benefit.

Interestingly, it could be the case that increasing or decreasing the amount of mutation to explicitly inactive genetic material would improve the evolutionary search of CGP. For instance, it has been shown in this chapter that never mutating explicitly inactive genes significantly worsens the evolutionary search. This method is equivalent to an explicitly inactive gene mutation rate of 0%. When mutations were allowed to alter explicitly inactive genetic material, the same mutation rate was used as for the active genes; 3%. However, there is no reason to assume that this is a suitable mutation rate to make best use of explicit genetic redundancy. For instance, it could be the case that a much larger or smaller mutation rate for inactive genes typically produces better results. Additionally, it may be the case that different explicitly inactive genetic mutation rates produce the best search depending upon whether the search is trapped in a local optimum. Interestingly,

[2]Such as those used previously for CGPANN [154].

previous results presented in [84] demonstrated that heightening the level of mutation to inactive genes failed to improve the performance of CGP. However, as the method used the newly proposed accumulate mutation method, it is not clear what percentage of genes were actually mutated.

A related topic is whether partial solutions are maintained within the inactive genes ready to be reactivated at a later stage. For instance, on dynamic fitness landscapes a reasonable solution could be found to be no longer suitable. This may lead to sections of the chromosome becoming inactive whilst looking for a new suitable solution. Now if the fitness landscape continues to change then the old solution, still present in the inactive genetic material, may once again be beneficial. However, this time, instead of completely rediscovering the solution, the old solution can be reactivated ('remembered'). Whether this would actually occur is speculative, and it has been shown that CGP rarely reactivates a previously active node without it first being mutated [84]. However, it does demonstrate additional possible behaviours of explicitly inactive genes, other than NGD, which could also be manipulated.

The fact that NGD was shown to aid the escape from local optima is an important result as it indicates that it may be increasingly beneficial on more challenging tasks. This is because one of the reasons a given task is more challenging than another is due to the number local optima in the search space and how difficult they are to escape. Therefore, methods which aid the escape of local optima are likely to be of greater benefit on more challenging fitness landscapes. Future work should therefore investigate applying CGP to increasingly difficult problem instances whilst repeating the experiments presented here. This would identify if NGD does become increasingly beneficial on harder tasks.

One of the reasons the research presented here is possible is due to the ease of which explicitly inactive genes in CGP can be identified. This makes the study of NGD much simpler. Additionally, if it were found that certain characteristics were desirable in inactive genetic material, such as specific mutation methods, this could be easily implemented for explicit genetic redundancy. Therefore, although many methods, such as tree-based GP, do contain implicit genetic redundancy, empirically studying its role is harder. It remains for future investigation whether there is an effective method for identifying implicitly inactive genetic redundancy in tree-based GP. Notwithstanding this, the speed at which explicitly inactive genetic redundancy can be identified may still be significant in further work.

It has been discussed that the reason CGP can be so effective using a simple $(1+\lambda)$-ES is that high levels of genetic redundancy facilitate the escape from local optima. More typically, GP methods use large populations of diverse genetic material and utilise the crossover operator. This method is thought to prevent the population becoming trapped in local optima due to it sampling a wider area of the search space. For instance, if an individual (or group of individuals) does become trapped, due to the diversity of the other members of the population, the search can still progress. Additionally, due to crossover, those stuck in the local optima may later be recombined with solutions outside of the local optima, resulting in their children escaping. Therefore, a possible criticism of the work presented here could be that the only reason CGP benefits so strongly from explicit genetic redundancy is that is does not utilise crossover to help escape local optima. However, publications investigating the use of crossover for CGP are mixed as to whether it offers an advantage [42, 191, 198, 202, 278], indicating that the benefits of crossover are not that substantial compared to other GP methods. Whereas it has been shown that the benefits of explicit genetic redundancy are substantial for CGP. However, to confirm that explicit genetic redundancy is not simply compensating for the fact that crossover is not utilised by CGP, important future work would be to apply the same experiments as presented here using CGP with crossover. If it were shown that explicit genetic redundancy still provided a benefit when crossover was employed, it would demonstrate that it is not simply compensating for the fact crossover is not typically used. However, if it were shown that explicit genetic redundancy no longer provides a benefit when crossover is used, it would demonstrate that explicit genetic redundancy is indeed compensating for the fact crossover is not employed.

8.5 Investigating Increasing Explicit Genetic Redundancy

As has been discussed in Section 8.3, it has been previously presented that CGP produces a more effective search as increased numbers of available nodes are used [207]. The proposed explanation was that using more available nodes increased the level of explicit genetic redundancy which led to a more effective evolutionary search. However, an alternative explanation was later presented arguing that the additional genetic redundancy would be unlikely to be ever used, due to its position within the genotype, and that the observed benefit was due to the high levels of available nodes compensating for length bias [82, 84].

Regardless of the explanation, it still stands that using an increased number of available nodes was seen to improve the evolutionary search for CGP. Additionally, as previous work did not show the full trend of this relationship (i.e. it did not allow large enough genotypes) it is difficult to draw further conclusions. Once the full trend is identified it may be possible to distinguish between the two explanations as to why allowing larger genotypes is more effective.

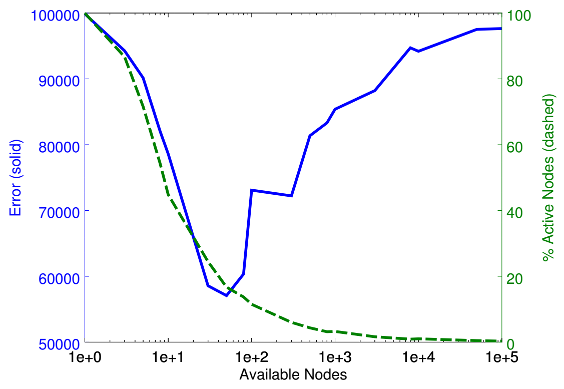
Here, when investigating the effect of increasing the number of available nodes, the same benchmarks and function sets are used as described in Section 8.4. During the investigation the following CGP parameters are used: 50 runs, 10,000 generations, a (1+4)-ES, 3 percent probabilistic mutation, no crossover, a node arity of two. The number of available nodes investigated are in the range one to one hundred thousand; note previous research studied the relationship up to four thousand available nodes [207].

8.5.1 Experiments

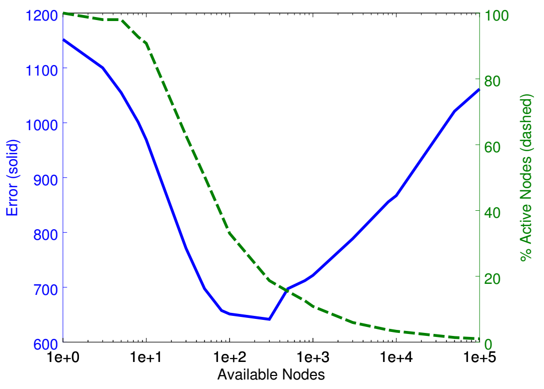
The experiment presented is an extension of previously presented works [207] that investigated the effect of increasing the number of available nodes on CGPs evolutionary search. The experiment extends previous work in a number of regards. The number of benchmarks is much larger, the range of problem types is larger, the benchmarks are much more challenging and, most importantly, the range of the number of available nodes is much larger.

The results of varying the number of available nodes are given in Figure 8.6 for all of the benchmarks described. Each of the individual subplots show the fitness achieved and the percentage of active nodes versus the number of available nodes; note the logarithmic scale on the x-axis. Both fitness and percentage of active nodes are presented so the relationship between the levels of explicit genetic redundancy and the effectiveness of the evolutionary search can be compared.

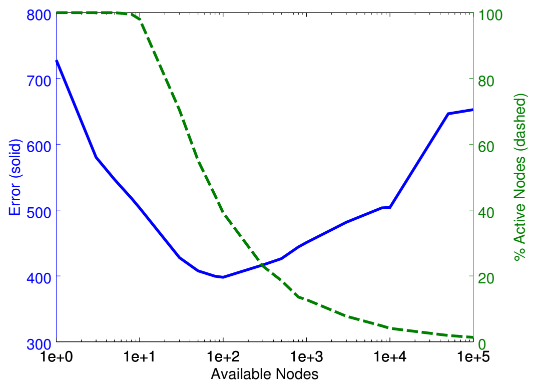
For reference, the number of available nodes which resulted in the lowest errors achieved in Figure 8.6 are given in Table 8.5. However, it should be noted that these values are likely to be a function of the evolutionary parameters used, and so should not be taken as the definitive best number of available nodes when applying CGP to these tasks.



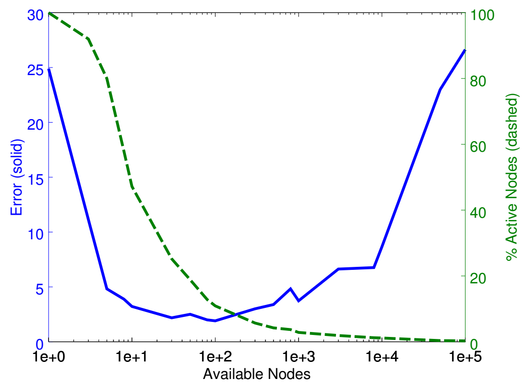
(a) Double Pole



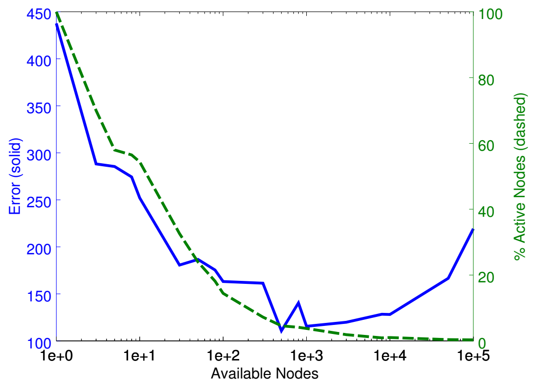
(b) Full Adder 4 bit



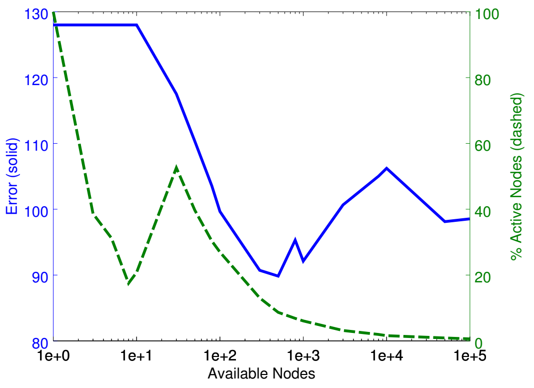
(c) Multiplier 4 bit



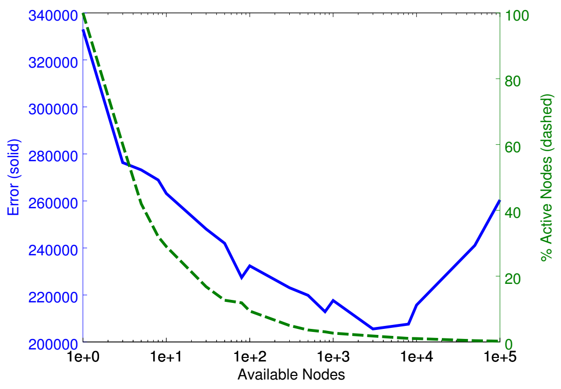
(d) Nguyen 10



(e) Pagie 1



(f) Parity 8 Bit



(g) Tower Problem

Figure 8.6: Number of available nodes versus fitness and percentage of active nodes.

Table 8.5: Number of available nodes which resulted in the lowest errors presented in Figure 8.6.

Benchmark	Nodes
Double Pole	50
Full Adder 4 bit	300
Multiplier 4 bit	100
Nguyen 10	100
Pagie 1	500
Parity 8 Bit	500
Tower Problem	3000

A number of interesting features can be identified in Figure 8.6. Firstly, consider the fitness achieved under variation in the number of available nodes. It can be seen that in each case the fitness first improves as the number of available nodes is increased, until an optimum is reached, and then the fitness worsens as the number of available nodes continues to increase.

In the case of the percentage of available nodes, the general trend is to start at approximately 100% and then continuously fall, following an approximate sigmoid curve, as the number of available nodes is increased; asymptotically approaching zero percent.

It can also be seen that initially the fitness achieved is closely correlated with the percentage of active nodes. This accounts for the conclusions reached in [207]. However, beyond the optimal number of available nodes, the percentage of active nodes continues to fall but the achieved fitness *worsens*. That is to say, it is not the case that increasing the percentage of inactive nodes improves the evolutionary search for CGP. Nor is it that case that increasing the number of available nodes indefinitely improves the evolutionary search for CGP. It appears that the effectiveness of the evolutionary search is determined by the number of available nodes but this is not correlated with the percentage of inactive nodes.

8.5.2 Discussion

From Figure 8.6, it can be seen that increasing the number of available nodes does cause the percentage of active nodes to decrease. Additionally, increasing the number of available nodes also *initially* improves the evolutionary search for CGP. This agrees with the findings in [207] and what lead to their conclusions. However, as the number of available nodes continues to increase, the percentage of activate nodes continues to fall but the effectiveness of the evolutionary search begins to *worsen*. This disproves the theory that the effectiveness of the evolutionary search is correlated with the level of genetic redundancy.

It has been previously shown that CGP exhibits a length bias towards using a particular number of active nodes [82, 84]. This length bias is a function of the number of inputs, the number of nodes, the arity of each node and the number of outputs. Therefore, by varying the number of available nodes one is effectively varying the number of nodes to which there is a bias. As there is likely an optimal number of nodes for a given task, it is logical that as the bias to a number of nodes is increases, the performance first improves to an optimum and then worsens. In this regard, the number of available nodes is like many other evolutionary parameters whose optimal value is task dependent. Note that this result does not contradict the evidence for the benefit of explicit genetic redundancy. It is only that explicit genetic redundancy introduced by increasing the number of available nodes is not beneficial. As was described in [82, 84], the majority of these inactive genes will describe nodes with a high index (positioned towards the outputs), and will be very unlikely to ever be made active. That is to say, the level of explicit genetic redundancy is increased, but in such a way that it is rarely used. Therefore, increasing the number of available nodes is not an effective method of increasing the levels of genetic redundancy.

Additionally, from the plots given in Figure 8.6 it does not appear that 95% genetic redundancy provides the best evolutionary search for CGP; as was proposed in [207].

This result corrects a previous misconception that continuously increasing the number of available nodes is beneficial to CGP's evolutionary search [207]. Additionally, this result, coupled with previous research [82, 84], demonstrates that increasing the number of available nodes is not an effective method of increasing the levels of *useful* genetic redundancy in CGP.

8.6 Investigating Neutral Genetic Drift in CGPANN

So far this chapter has focused on the influence of NGD on CGP's evolutionary search. This section now investigates the role of NGD for CGPANN.

As has been previously discussed in Chapter 2, the role of NGD in NE has so far been overlooked. These experiments are therefore significant as they are the first, to the author's knowledge, to investigate the possible benefits of NGD for NE.

Like CGP, CGPANN contains both implicit and explicit genetic redundancy. Therefore it seems likely that CGPANN will also benefit from its presence. However, there are important distinctions between the two techniques which may influence the effect of NGD in CGPANN. Firstly, Artificial Neural Network (ANN) typically uses a higher node arity than GP methods. This higher arity is likely to result in CGPANN chromosomes containing a higher proportion of active genes; as length bias is a function of node arity [82, 84]. Secondly, ANNs use connection weights which are likely to allow evolution finer control in the search space.

It has been previously discussed that unconstrained, CGPANN allows multiple connections between the same two nodes. As this can sometimes produce undesirable behaviour, such as the maximum weight range set by the user being exceeded, methods for it prevention have been presented. One such method is achieved by only decoding the first of multiple connections between the same two nodes into the phenotype. This is interesting in the context of genetic redundancy, as the additional connection genes and associated connection weight genes are now genetically redundant. Additionally, as this decoding stage is *required* in order for the genotype to describe its corresponding phenotype, it means that in this case, CGPANN could be described as an indirect encoding scheme. However, for simplicity disallowing multiple connections between the same two nodes is not investigated in this chapter.

When using CGPANN the following default parameters are used: 50 runs, 100,000 generations, a (1 + 4)-ES, 5 percent probabilistic mutation, no crossover, a connection weight range of ± 5, logistic sigmoid transfer function, 100 nodes each with an arity of 5.

The benchmarks used comprise: Ball Throwing, Double Pole, Full Adder 4 Bit, Multiplier 4 bit, Monks Problem 1, Nguyen 10, Pagie 1, Parity 6 Bit, Parity 7 Bit, Parity 8 Bit and Two Spirals. All the benchmarks used are described in Appendix A. In cases where the benchmarks require both positive and negative outputs the logistic sigmoid was 'stretched' to a ± 1 range. In cases where the benchmarks required inputs/outputs greater than ± 1, the inputs/outputs are linearly mapped into a ± 1 range using the maximum and minimum un-normalised values. Additionally, in the case of the Ball Throwing benchmark the fitness achieved were subtracted from 10 in order to make it a minimisation task, similarly for the Double Pole Balancing benchmark. This ensures that in all cases lower fitness values represents a better solution consistently across all the benchmark used.

8.6.1 Experiments

The experiments presented investigate the role of NGD in CGPANN following the same methodology as those presented for CGP in Section 8.4. For this reason they will not be reintroduced.

The first experiment is to isolate the overall benefit of ENGD. This is achieved by comparing the difference in performance between regular CGPANN and CGPANN when only active genes are mutated. These two methods differ only in the presence of ENGD and therefore isolate the benefit of ENGD. This comparison is given in Table 8.6. In the case of the Double Pole Balancing benchmark all of the fifty runs found a perfect solution in the allowed 100,000 generations. Therefore, in order to assess the relative performances the average number of generations to find perfect solutions must be compared. These results are given in Table 8.7.

Table 8.6: Comparing regular CGPANN to only mutating active genes in order to isolate the benefit of ENGD. In all cases a lower fitness represents a better search.

Benchmark	Regular CGPANN	Only Mutate Active Genes	U-Test	Effect Size
Ball Throwing	2.02	2.50	5.02E-1	0.53720
Double Pole	0	0	-	-
Full Adder 4 Bit	776.88	783.58	7.20E-1	0.52100
Multiplier 4 bit	510.84	502.20	2.40E-1	0.56840
Monks Problem 1	0.82	0.77	7.70E-1	0.50940
Nguyen 10	3.14	3.22	7.91E-1	0.51560
Pagie 1	198.03	198.16	3.72E-1	0.55200
Parity 6 Bit	14.94	15.78	2.23E-1	0.57060
Parity 7 Bit	39.72	40.56	1.19E-1	0.58780
Parity 8 Bit	92.74	94.62	4.57E-1	0.54120
Two Spirals	73.62	73.44	8.68E-1	0.50980

Table 8.7: Comparing regular CGPANN to only mutating active genes in order to isolate the benefit of ENGD. In all cases a lower number of generations represents a better search.

Benchmark	Regular CGPANN	Only Mutate Active Genes	U-Test	Effect Size
Double Pole	1081	1369	2.60E-1	0.56560

As can be seen in Tables 8.6 and 8.7, in all cases the presence of ENGD has no notable impact on the effectiveness of the search. In some cases the search is marginally better and in others worse. In no cases are the differences statistically significant and in no cases is the effect size greater than small. It therefore appears that ENGD provides no benefit for CGPANN.

The second experiment investigates whether NGD, both implicit and explicit, and other forms of redundancy, aid the escape from local optima. This is achieved by comparing the difference in performance between regular CGPANN and CGPANN in which only fitness improvements are allowed to be selected. These two methods differ only in that the former (regular CGPANN) has the ability to pass on neutral mutations without the presence of beneficial mutations; a requirement for use in escaping local optima. This comparison is given in Table 8.8 with the generational results given in Table 8.9 for the Double Pole Balancing.

Table 8.8: Comparing regular CGPANN to only selecting fitness improvements in order to isolate the benefit of NGD aiding the escape from local optima. In all cases a lower fitness represents a better search.

Benchmark	Regular CGPANN	Only Fitness Improvements	U-Test	Effect Size
Ball Throwing	2.02	2.83	5.22E-2	0.60560
Double Pole	0	0	-	-
Full Adder 4 bit	776.88	774.70	6.27E-1	0.52840
Multiplier 4 bit	510.84	513.38	3.79E-1	0.55120
Monks Problem	0.82	7.94	1.39E-13	0.90200
Nguyen 10	3.14	3.22	7.51E-1	0.51860
Pagie 1	198.03	197.20	6.97E-1	0.52280
Parity 6 Bit	14.94	21.20	2.40E-15	0.95500
Parity 7 Bit	39.72	46.14	2.23E-11	0.88100
Parity 8 Bit	92.74	97.76	1.30E-5	0.74960
Tower Problem				
Two Spirals	73.62	76.58	2.74E-4	0.71060

Table 8.9: Comparing regular CGPANN to only selecting fitness improvements in order to isolate the benefit of NGD aiding the escape from local optima. In all cases a lower number of generations represents a better search.

Benchmark	Regular CGPANN	Only Fitness Improvements	U-Test	Effect Size
Double Pole	1081	597	1.83E-1	0.57740

As can be seen in Tables 8.8 and 8.9 results are mixed. For around half of the benchmarks investigated the ability for NGD and other type of redundancy to escape local optima provided no notable benefit. For the remaining half of the benchmarks however, the ability to escape local optima provided a substantial benefit with statistical significance and a large effect size. It therefore appears that the ability to escape local optima is beneficial to CGPANN; but to a much lesser extent than for CGP.

The third experiment investigates the benefit of ENGD when not being used to escape local optima. For instance, the benefit of ENGD occurring at the same time as other beneficial mutations. This is achieved by comparing the difference between only allowing fitness improvements to be passed on with only allowing mutations to active genes whilst also only allowing fitness improvements to be passed on. These two methods differ only in the ability of ENGD to occur in combination with fitness improving mutations to active genes. This comparison is given in Table 8.10 with the generational results given in Table 8.11 for the Double Pole Balancing.

Table 8.10: Selecting only fitness improvements compared with only allowing mutations to active genes while only selecting fitness improvements. This isolates the benefits of ENGD other than aiding the escape of local optima. In all cases a lower fitness represents a better search.

Benchmark	Only Fitness Improvements	Only Mutate Active Genes, Only Fitness Improvements	U-Test	Effect Size
Ball Throwing	2.83	3.29	2.95E-1	0.55200
Double Pole	0	1228	3.27E-1	0.51000
Full Adder 4 bit	774.70	784.12	4.59E-1	0.54320
Multiplier 4 bit	513.38	513.84	8.17E-1	0.51360
Monks Problem 1	7.94	8.13	9.72E-1	0.50220
Nguyen 10	3.22	3.05	4.57E-1	0.54340
Pagie 1	197.20	197.90	9.59E-2	0.59680
Parity 6 Bit	21.20	21.24	3.81E-1	0.54880
Parity 7 Bit	46.14	45.88	3.60E-1	0.55100
Parity 8 Bit	97.76	96.32	3.96E-1	0.54900
Two Spirals	76.58	77.08	5.20E-1	0.53740

Table 8.11: Selecting only fitness improvements compared with only allowing mutations to active genes while only selecting fitness improvements. This isolates the benefits of ENGD other than aiding the escape of local optima. In all cases a lower number of generations represents a better search.

Benchmark	Only Fitness Improvements	Only Mutate Active Genes, Only Fitness Improvements	U-Test	Effect Size
Double Pole	596.56	4268.66	1.01E-2	0.64940

As can be seen in Tables 8.10 and 8.11, removing the benefit of ENGD occurring together with other beneficial mutations never impacted the search with any statistical significance or meaningful effect size. Therefore it appears that there is no benefit of ENGD occurring alongside positive mutations.

8.6.2 Further Experiments

As can be seen from the previous set of experiments, it appears that the benefit of ENGD is greatly diminished when CGP is applied to optimising ANNs; CGPANN. To confirm this result an extension to Section 8.6.1 is now presented.

In order for NGD to be of benefit there must be inactive genes which can drift. It may therefore be the case that CGPANN contain insufficient levels of genetic redundancy (compared to CGP) such that it no longer beneficial. This is due to the higher arity increasing the percentage of nodes to which there is a length bias. To confirm this, the percentage of active nodes used by CGP and CGPANN in Sections 8.4 and 8.6.1 respectively are given in Table 8.12.

Table 8.12: Percentage of active nodes used by CGP and CGPANN.

Benchmark	CGP 100 Nodes	CGPANN 100 Nodes	CGPANN 200 Nodes
Ball Throwing	-	36.30 %	28.93 %
Double Pole	13.68 %	25.80 %	18.42 %
Full Adder 4 bit	39.44 %	52.74 %	42.49 %
Multiplier 4 bit	45.42 %	59.06 %	51.15 %
Monks Problem 1	-	19.14 %	20.93 %
Nguyen 10	15.74 %	22.88 %	25.19 %
Pagie 1	24.20 %	31.74 %	28.07 %
Parity 6 Bit	-	29.16 %	25.02 %
Parity 7 Bit	-	30.22 %	24.45 %
Parity 8 Bit	31.62 %	31.92 %	24.63 %
Tower Problem	23.70 %	-	-
Two Spirals	-	31.48 %	22.88 %

As can be seen in Table 8.12, there is a notable difference between the percentage of active nodes used by CGP and CGPANN; with CGP chromosomes containing a lower percentage of active genes.

To compensate for this difference in the levels of genetic redundancy, the same CGPANN experiments are repeated using 200 nodes. This is undertaken to increase the level of genetic redundancy. The average percentage of active nodes used when the number of available nodes is increased to 200 is also given in Table 8.12. As can be seen, although in most cases increasing the number of available nodes did cause the percentage of active nodes to fall, in others it had little effect; or even a marginal increase.

Only the first of the previous experiments is now repeated, whether or not ENGD offers an advantage to the evolutionary search with the increased level of explicit genetic redundancy. This experiment is to isolate the overall benefit of ENGD. This is achieved by comparing the difference in performance between regular CGPANN and CGPANN when only active genes are mutated. Again in the case of the Double Pole Balancing the average number of generations to find a solution is used for comparison. The results are given in Tables 8.13 and 8.14.

As can be seen in Tables 8.13 and 8.14, even with the increased number of nodes there

Table 8.13: Comparing regular CGPANN using an increased number of available nodes to only mutating active genes in order to isolate the benefit of ENGD. In all cases a lower fitness represents a better search.

Benchmark	Regular CGPANN	Only Mutate Active Genes	U-Test	Effect Size
Ball Throwing	2.49	2.65	6.71E-1	0.52280
Double Pole	0	0	-	-
Full Adder 4 Bit	825.20	827.22	7.93E-1	0.51540
Multiplier 4 bit	517.44	526.30	5.80E-2	0.61020
Monks Problem 1	0.301	0.21	9.80E-1	0.50080
Nguyen 10	19.89	19.92	8.55E-1	0.51080
Pagie 1	198.59	199.44	1.10E-1	0.59320
Parity 6 Bit	16.60	16.16	5.63E-1	0.53360
Parity 7 Bit	41.16	42.24	2.19E-1	0.56880
Parity 8 Bit	93.76	94.30	3.72E-1	0.54980
Two Spirals	74.48	74.58	9.37E-1	0.50480

Table 8.14: Comparing regular CGPANN using an increased number of available nodes to only mutating active genes in order to isolate the benefit of ENGD. In all cases a lower number of generations represents a better search.

Benchmark	Regular CGPANN	Only Mutate Active Genes	U-Test	Effect Size
Double Pole	960	2506	3.24E-1	0.55740

is no notable benefit of ENGD for CGPANN. In no case was the difference statistically significant and in no case was the effect size greater than small. It therefore appears that ENGD still has no benefit for CGPANN; even with the increased genetic redundancy. Therefore it can be seen that the reduced level of inactive genes was not the reason CGPANN did not benefit from ENGD to the same extent as CGP.

8.6.3 Discussion

An interesting result was seen when investigating the effect of NGD in CGPANN compared to CGP. The overall benefit of NGD and other forms of redundancy was much lower for CGPANN than for CGP. With the benefit of ENGD being completely absent. This is a surprising result as the underlying algorithm for both methods is the same. The only differences are the use of connection weight genes, higher node arity and the transfer functions employed. Therefore the reason for this distinction is likely due to one of these differences.

The effect of the higher arity causing a lower average number of explicitly inactive genes was investigated, where it was shown not to be the cause of ENGD being ineffective.

However, as increasing the number of available nodes does not proportionately increase the number of inactive nodes between the active nodes, this effort may never have been likely to increase the benefit of ENGD. Additionally, as explicitly inactive genes only make up a proportion of the types of redundancy which can be used to help escape local optima, the fact that CGPANN exhibits a lower percentage of explicit genetic redundancy does not explain the highly reduced effect from NGD and other forms of redundancy generally. This leaves the presence of connection weights and the differing transfer functions as the possible causes of the asymmetry.

Although the transfer function used may be having an influence, the author believes the difference is mainly due to the use of connection weight genes compensating for a possible limitation in CGP.

When mutations are applied in standard CGP, they either change the structure of the program or the computation undertaken in the nodes; or both. Both of these changes are likely to have a large influence on the semantics, resulting in a very different fitness score. However, CGPANN contains connection weights, and so another possible mutation is available; mutating connections weight values. Now it appears intuitive than mutations to connection weights can have both a small and a large influence on the semantics; depending upon how far the new connection weight values lies from the old. This provides the possibility of mutations which are more likely to have a small influence on semantics. Now, if there is a bias to larger mutations then it becomes easier to become trapped in an area of the search space; because the search is inherently more random. That is to say, it is less likely to exploit a local search to improve on the current solution and more likely to conduct a more global search. Therefore, CGP may be more likely to become trapped in areas of the search space, even if they are not local optima, than CGPANN. This is because CGPANN has the capability to conduct a more local search. As NGD provides a mechanism for aiding the escape when trapped in a region of the search space, it may not be as beneficial to CGPANN simply because it is less likely to become trapped. For this reason, CGPANN and other forms of NE may not benefit as greatly from NGD as GP methods.

If it is the case that the presence of connection weight genes results in a search which is less likely to become trapped in areas of the search space, then any benefit of NGD aiding the escape of these areas would be diminished; as is seen in the results. Additionally, as explicit genetic redundancy represents only a proportion of the redundancy available, the benefit of its effect specifically could be diminished to the point it could not be detected. This was supported by the results.

In the case of NE in general, it appears that as other NE methods also utilise connection weights, they would also not benefit as strongly from NGD and non-genetic redundancy. Therefore, the presence of explicit genetic redundancy in methods such as GNARL, EPNet and COVNET may not provide a significant advantage.

However, as the benefit of NGD is thought to increase as the difficulty of the application increase, NGD may still hold a benefit for NE. It may be the case that the benchmarks investigated here were not challenging enough, or the search was not allowed to run for long enough, for its benefit to be identified. For instance, if an experiment were left for a sufficiently long time, CGPANN may reach an actual local optimum by conducting a local search. At this point NGD may once again be beneficial. Such investigations are left as further work.

8.7 Summary

The work presented has undertaken a detailed analysis of the role of NGD in CGP and CGPANN. Previous misconceptions have been corrected and further insights made.

It is now known that the level of genetic redundancy provided implicitly is not sufficient to best utilise NGD in CGP. It has been shown that the presence of explicit genetic redundancy offers a significant further advantage. It is also likely that other forms of EA which contain explicit genetic redundancy are also benefiting from its presence. Additionally, other forms of EA which do not contain explicit genetic redundancy may benefit from its inclusion.

It has also been shown that isolating and manipulating explicit genetic redundancy is far easier than controlling implicit genetic redundancy. This makes the study of explicit genetic redundancy much simpler and enables easy adaptation to make best use of its presence. For instance, by varying its mutation rate to increase or decrease the rate of drift. This may lead to explicit genetic redundancy being more desirable than implicit redundancy, if it can be fully utilised and controlled.

Additional benefits of NGD for CGP, other than escaping local optima, have also been empirically isolated and demonstrated to aid the evolutionary search. This benefit is the ability to “drift” along plateaus in the search space even when not trapped in a local optima.

It has also been shown for CGP that increasing the number of available nodes is not an effective method for increasing the benefit of genetic redundancy. It appears that for CGP, there is a task dependent optimum number of available node, above and below which, the search worsens. Rather than the previous belief that increasing the number of available nodes always benefits the search.

From the results from investigating ENGD in CGPANN, it was seen that the benefits are completely absent. Additionally, it was shown that the benefit of all types of redundancy aiding the escape of local optima, not just ENGD, is greatly diminished; although still present. A possible explanation has been proposed which suggests that the connection weight genes present in CGPANN allow for more local, finer mutations to take place. These mutations may aid the navigation of the search space meaning the search is less likely to become trapped. In this regard it may be the case that NGD is compensating for a limitation of CGP; in that it can only make relativity large scale mutations. Although further research would be required to truly assess if this is the case.

If this hypothesis is correct, then it may help inform future developments in GP. For instance, the inclusion of mechanisms to help conduct a more local search may be greatly beneficial to GP methods; such as the inclusion of connection weights.

This work has opened the discussion concerning NGD in NE. It has also shown that currently it appears that NE does not benefit from genetic redundancy to the same extent as GP generally; which may be due to a limitation in GP.

Chapter 9

CGPANN Applied to Classification

This chapter presents a rigorous evaluation of the application of Cartesian Genetic Programming of Artificial Neural Networks (CGPANN) to the domain of classification. This evaluation involves the comparison of CGPANN with a wide range of standard classification methods over a wide range of benchmark tasks.

9.1 Structure of this Chapter

Section 9.2 provides a summary of previous applications of CGPANN to classification and identifies where further research is needed. Section 9.3 describes the methodology used to ensure a fair evaluation of CGPANN as a classification method. Section 9.4 describes how CGPANN is applied to the task of classification. Sections 9.5 and 9.6 describe the standard classification methods used for comparison and the benchmarks which are employed. Section 9.7 presents the results of the described experiments with Sections 9.8 and 9.9 giving a discussion of the findings and a closing summary.

9.2 Background

This section provides a background to previous applications of CGPANN to the domain of classification. Previous studies of CGPANN as a classification method have mainly focused on real world applications, rather than comparing CGPANN with other standard classification methods. Form these results it is not possible to assess the suitability of

CGPANN as a classification technique; as it is not known how other methods would have performed. For instance A. Ahmad has previously used CGPANN to classify arrhythmia [6] as well as the detection of cancerous cells from mammogram scans [7].

Works which have compared CGPANN to other standard classification methods [5, 8, 191], have so far only utilised a single benchmark task; breast cancer classification. Again this means it is not possible to assess CGPANN as a classification method; as a single benchmark is insufficient to draw meaningful conclusions.

Additionally, in previous applications of CGPANN to the breast cancer benchmark, the results presented for CGPANN were those found after optimising CGPANN parameters on the *testing* set performance [5, 8]; specifically the number of nodes used. This is very bad practice which removes the purpose of a testing set. Selection of any form, including parameter settings, should never be based on the *testing* set performance. This drastically weakens the rigour of previous studies.

Therefore, although CGPANN has been previously applied to classification, so far the comparisons to other methods have been fairly weak. This chapter presents a much more rigorous evaluation of CGPANN as a classification method, including comparisons to a wide range of standard classification methods using a wide range of benchmark tasks.

A major issue in comparative classification studies is the difficulty in ensuring the implementation of the benchmarks is the same in each case. For instance, it has been previously noted for the Iris benchmark, that so many version of the dataset exist, that it is difficult to know which results are, and which are not, comparable [28]. Additionally, even if the same version of the datasets are used, differences in which samples are used for training and which for testing can have a huge influence on performance. For instance, an extreme, albeit unrealistic, example would be a situation where all instances of a particular class were absent in the training set. Therefore it is important that when comparing classification methods, the training and testing split is the same for every comparative method; both in terms of the proportion of the data in the training and testing sets, and in terms of the actual samples which comprise each split. Finally, it is important that equal levels of effort is spent optimising the parameters for each comparative method. Contrasting two methods when for one substantial time has been spent optimising the parameters, and for the other it has not, is not a fair comparison.

This chapter aims to present a much fairer evaluation of CGPANN as a classification method than has been previously presented. This is undertaken by ensuring that all of the previously described issues are considered. Overall this is achieved by personally conducting experiments using CGPANN and a range of comparative methods. That is to say, all of the results present are the results of experimentation done by the author i.e. not taken from the literature. This ensures that the same methodology for finding suitable parameters can be used (leave group out cross validation), the same version of the dataset can be used with the same split of training and testing samples. Additionally, a much wider range of benchmark tasks are considered than has been previously for CGPANN. This makes for a much more rigorous, and fairer, evaluation of CGPANN as a classification technique.

9.3 Methodology

When benchmarking new classification methods it is paramount that the comparisons made to other methods are rigorous and fair. To ensure fairness of comparisons between different classification methods a number of steps need to be taken. Firstly, the training and testing set split of each benchmark should be the same for all classification techniques compared. Secondly, in cases where classification methods require parameter tuning, the same parameter tuning method should be used in each case. It would be unfair to compare method A which used un-optimised parameters with method B where great care had been taken to find near optimal parameters. Thirdly, in the case of stochastic methods the arithmetic mean of many runs should be presented so the typical performance is used for comparison. Finally, a reasonable number of benchmarks should be considered so the typical performance of each method can be evaluated.

In the work presented in this chapter, each method's parameters are found using leave group out cross validation (also referred to as holdout) [161]. When using leave group out cross validation, the training set is separated into a training set and a validation set. The validation set is the 'group left out'. The classification methods are trained on the training set using a range of parameter configurations. The performance of the produced classifiers is then validated on the validation set which is used as a figure of merit for the given parameters. Using this method a range of parameters can be assessed with the final chosen parameters being those with the best performance on the validation set.

Once the parameters have been determined, each classification method is then trained on all of the training data, including the validation set, using the best found parameters. The trained classifiers are then assessed on the testing data with the classification accuracy on the testing data used as the performance indicator. This final testing classification accuracy can then be used to compare separate classification methods.

Here, no early stopping [230] or other over training preventions are used by any of the classification methods. This is for two reasons. Firstly, the techniques used for preventing over fitting differ between classification techniques. Additionally, there are often many alternative over fitting prevention techniques for each classification method. This makes comparisons more challenging and less general to the underlying classification method. Secondly, optimising the parameters can be considered an indirect method for preventing over training. For instance, the parameters which produce classifiers which generalise the best on the validation set are those chosen for the final assessment. Therefore over training can be prevented through the correct choice of parameters.

9.4 Applying CGPANN to Classification

CGPANN is easily applied to classification and requires no alteration to the general algorithm. The number of chromosome inputs are set as the number of attributes used by each sample in the data set. These attributes are always floating point values in the range $[0, 1]$. This is ensured to be the case by linearly mapping the attribute values into the $[0, 1]$ range using the maximum and minimum values in the training set; no information from the testing set should be used. In cases where the attributes are categorical, each category is given an integer representation starting at zero which is incremented for each new category in the training data. For instance if an attribute could have a value of 'A', 'B' or 'C', these would be mapped to values of 0, 1 and 2. Then these integer values are normalised into a $[0, 1]$ range as previously described.

The number of chromosome outputs is set as the number of classification categories; with each output representing a separate class. When a samples normalised attributes are applied to the inputs, and the phenotype executed, the class corresponding to the output with the largest value is interpreted as the predicted class.

It would also have been possible to use a single output with the produced single value determining the class. For instance, in the case of binary classification, output values less than 0.5 it could be interpreted as class A, and greater or equal as class B. However, using this output decoding is likely to influence the difficulty of the task. For instance, if the task were to classify a plant species, does species 'A' more smoothly 'flow' into species 'B' or 'C'? Do any of the species 'flow' into one another? For this reason a separate output was used to indicate each classification category.

The fitness assigned to each chromosome is simply the classification accuracy i.e. the proportion of the samples which were classified correctly. Although other classification metrics, such as Cohen's kappa [43] measure or Matthews correlation coefficient [194], could also be used, here classification accuracy is used for its simplicity and wide adoption[1].

It would also have been possible to use a fitness function which assigns a fitness measure of the Mean Square Error (MSE) between the outputs and the target classes. For instance if there were three classes, an example output could be $\{0.1, 0.5, 0.9\}$ and an example target output could be $\{0, 0, 1\}$ i.e. the third class is the correct class. This would result in a mean squared error of 0.09; $((0.1 - 0.0)^2 + (0.5 - 0.0)^2 + (0.9 - 1.0)^2)/3$. Although this measure does not use the classification accuracy, it would produce an evolutionary pressure to create outputs which, when decoded as before, by taking the highest output as the predicted class, performed as a classifier. Additionally, in this case the output values would represent a confidence in the classification. This is because there is an evolutionary pressure to set the outputs corresponding to the incorrect classes as low as possible, and the output corresponding to the correct class as high was possible[2]. For example, if the outputs produced were $\{0.4, 0.4, 0.5\}$ this could be interpreted as the third class with little confidence. Whereas if the outputs produced were $\{1.0, 0.2, 0.1\}$ this could be interpreted as the first class with high confidence. However, as the comparisons presented here are not concerned with confidence, only accuracy, this form of fitness function is not used. It does however demonstrate the flexibility of using Evolutionary Algorithm (EA) based methods; the fitness function can be adapted to differing requirements.

As described in Section 9.3, the parameters used when comparing the classification methods are determined using leave group out cross validation. When using CGPANN the parameters investigated are in the following ranges: generation $\{50, 100, 1000, 10000, 100000, 1000000\}$, mutation rates $\{0.01, 0.03, 0.05, 0.1\}$, number of available nodes $\{10, 50, 100, 200, 500\}$, node arities of $\{5, 10, 20\}$ and connection weight ranges of $\{1, 5, 10\}$. In all cases a (1+4)-ES was used and the neuron transfer function chosen was the logistic sigmoid. Additionally, all connections between active nodes were used in the phenotype i.e. multiple connection between pairs of nodes were allowed.

[1]For instance, a range of libraries are used to compare various classification methods; all of which provide the option to use standard classification accuracy as the measure of fitness.

[2]There may even be an evolutionary pressure to set all the outputs close to 0.5 in cases of uncertainty

9.5 Comparative Methods

In order to assess the suitability of CGPANN as a classification method it must be compared to a wide range of standard classification techniques. In the work presented, the implementation of the comparative classification methods were accessed through the Caret Package [70] for the R programming language [234]. The Caret Package provides a common Application Programming Interface (API) for interfacing with a wide range of R classification packages. The Caret Package also provides functionally for using leave group out cross validation to optimise the parameters used by each classification method. This is achieved by the Caret Package internally storing sensible upper and lower limits for each parameter of each classification method. The user then specifies how coarsely this range of parameter values is swept over. For all the comparative classification methods presented here, each parameter is swept over an evenly distributed set of 10 values

The classification methods used for comparison are now described.

9.5.1 Recursive Partitioning Decision Trees

Many Recursive Partitioning Decision Trees (RPDT) methods are used for comparison as they represent one of the more popular classification methods. The method referred to here as RPDT is a standard implementation provided in the rpart package [271, 272]. The rpart package chooses the splitting method based on features of the training data. The possible splitting methods comprise anova, Poisson, class or exp. When selecting suitable parameters for RPDT, Caret varies the value of the complexity parameter. The complexity parameter specified a cost of using additional leaf nodes in the tree. Therefore small complexity parameter values favours larger trees (possibly over fitting), and large complexity parameter values favour smaller trees (possibly under fitting).

9.5.2 Multilayer Perceptrons

Another popular classification method is the use of Artificial Neural Network (ANN)s trained using back propagation. This method is commonly referred to as Multilayer Perceptrons (MLP)s. Since CGPANN is an ANN training method it is important to include the more common training method for ANNs. Here the nnet package [288] is used to train ANNs using back propagation. The nnet package trains single hidden layer ANNs with or without skip-layer connections (connections from outputs straight to inputs). When selecting suitable parameters for the MLPs, Caret varies the size of the network (number of hidden nodes) and the decay parameter (learning rate).

9.5.3 Supervised Self Organizing Maps

Although Kohonen's Self Organizing Maps (SOM) are typically an unsupervised learning method, a supervised learning version has recently been developed [199]. Supervised SOM are also referred to as Bi-Directional Kohonen maps. The Supervised SOM used here are provided by the kohonen package [296]. When selecting suitable parameters for the SOM, Caret varies the x and y dimension, the initial weights and the topology.

9.5.4 C4.5

The second from of RPDT used is C4.5 [233]. Here the C4.5 implementation is from the RWeka [117] package, a R wrapper for the Weka library [308], and is termed J48. J48 uses C4.5 to create pruned or unpruned decision trees. When selecting suitable parameters for the J48 package, Caret varies the pruning confidence.

9.5.5 C5.0

The third form of RPDT employed is an extension to C4.5 termed C5.0 [167]. The implementation of C5.0 used here is from the C50 package [167]. When selecting suitable parameters for the C50 package, Caret varies the number of boosting iterations (if any), the model uses (full decision tree or a collection of rules) and whether winnowing is used (true or false).

9.5.6 Support Vector Machines

Support Vector Machines (SVM) [45] are another popular standard classification method. Here the SVM implementation used is that provided by the kernlab Package [142]. The kernlab package provides many implementations which use differing kernels. Here the results of both linear (svmLinear) and polynomial (svmPoly) are presented. When selecting suitable parameters for svmLinear and svmPoly, Caret varies the cost of constraints violation in both cases and also the degree and scale of the polynomials in the svmPoly case.

9.5.7 K Nearest Neighbours

The K Nearest Neighbours (KNN) algorithm is another popular classification (and regression) method [63]. Here a weighted version of KNN is used which is provided in the kknn package [248]. When selecting suitable parameters for kknn, Caret varies the value of k, the Minkowski distance and the kernel used (rectangular, triangular, epanechnikov, biweight, triweight, cos, inv, Gaussian, rank and optimal).

9.5.8 Partial Least Squares

The final method used for comparison is Partial Least Squares (PLS) [309]. The implementation of PLS used here is widekernelpls from the pls Package [200]. When selecting suitable parameters for pls, Caret varies ncomp (the number of components to be used in the modelling).

9.6 Benchmarks

In order to draw strong conclusion concerning the effectiveness of CGPANN as a classification method, a reasonable number of benchmarks must be employed. To this end, seven standard benchmarks are used: Breast Cancer, DNA, Glass, Ionosphere, Iris, Olives and Segmentation. Each of these benchmarks is described in Appendix A.

For all of the classification benchmarks used, the samples are first separated into training and testing sets; 75% training and 25% testing. The data is separated such that the training and testing sets contain approximately the same proportion of each class as in the original set. For instance, if the dataset contains 30% class A and 70% class B, then both the training and testing sets will also comprise of approximately 30% class A and 70% class B. Additionally, the same data split is always used for each method.

When using leave group out to determine suitable parameters for each method, the training data is further separated into training and validation sets. The split is always 75% training and 25% validation. For clarity, this means that the validation set actually represents 18.75% of the entire initial dataset. The proportion of each class in the training and validation sets is also made to be approximately equal to the original dataset. Again the same split is used for all classification methods and for each parameter evaluation.

9.7 Results

The results of comparing CGPANN to a number of standard classification methods are now presented. In each case CGPANN is repeated 50 times so reasonable averages can be used. In the case of the other stochastic methods the experiments are repeated 25 times.

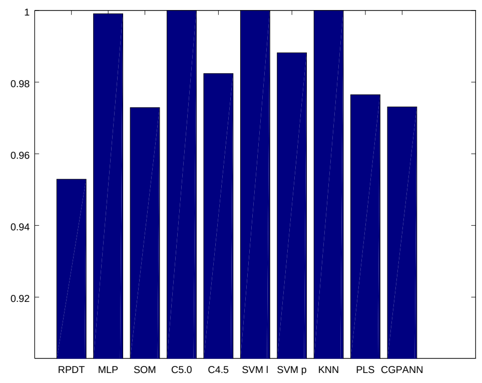
The results of applying the ten classification methods to the seven benchmarks are presented in Table 9.1. The method(s) which produced the best results on each benchmark are given in bold. Additionally, for easy high-level inspection the same results are provided as bar charts in Figure 9.1.

As can be seen in Table 9.1, and Figure 9.1, CGPANN does not perform well compared to standard classification techniques. In two cases CGPANN produced the worse result of all methods investigated. Additionally, in only one case is CGPANN shown to perform better than the average of all the classification methods investigated.

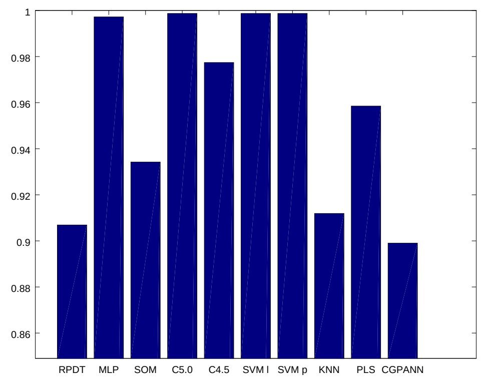
Finally, a closer evaluation of the relative performance of CGPANN and MLP is presented. This is undertaken because both CGPANN and MLP represent training methods for ANNs. This analysis is given in Table 9.2. In Table 9.2 it can be seen that in all cases the difference between CGPANN and MLP is statistically significant with a large effect size. Additionally, in five of the seven cases MLP outperformed CGPANN; with CGPANN outperforming MLP in the remaining two cases. It therefore appears that MLP represent a superior training method than CGPANN in the domain of classification.

Table 9.1: Classification accuracy of a range of classification methods.

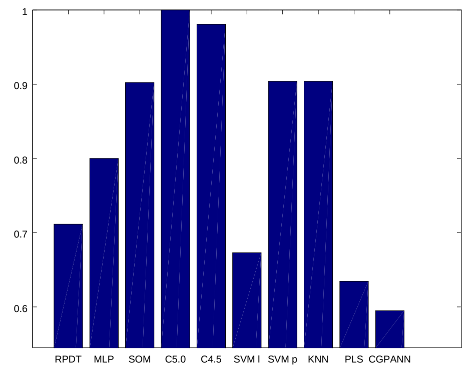
Method	Cancer	DNA	Glass	Ionosphere	Iris	Olives	Segmentation
RPDT	0.9529	0.9069	0.7115	0.8851	0.6667	**1.0000**	0.8790
MLP	0.9991	0.9978	0.8000	0.9991	0.9722	0.6137	0.7490
SOM	0.9729	0.9382	0.9023	0.8460	0.9711	0.9120	0.7852
C5.0	**1.0000**	**0.9987**	**1.0000**	0.9885	0.9444	**1.0000**	**0.9802**
C4.5	0.9824	0.9774	0.9808	**1.0000**	0.9722	0.9786	0.9405
SVM (lin)	**1.0000**	**0.9987**	0.6731	0.9080	0.9444	0.9786	0.8492
SVM (poly)	0.9882	**0.9987**	0.9038	0.9770	**1.0000**	0.9929	0.8770
KNN	**1.0000**	0.9119	0.9038	**1.0000**	0.9722	0.9929	0.9187
PLS	0.9765	0.9585	0.6346	0.8506	0.8333	0.8643	0.7917
CGPANN	0.9731	0.8990	0.5950	0.8901	0.9378	0.9234	0.8214
Average	0.9845	0.9586	0.8105	0.9344	0.9214	0.9256	0.8592



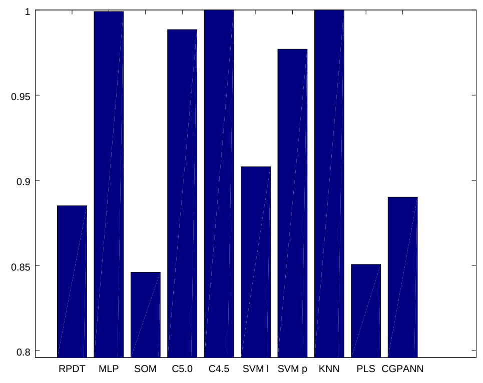
(a) Breast Cancer



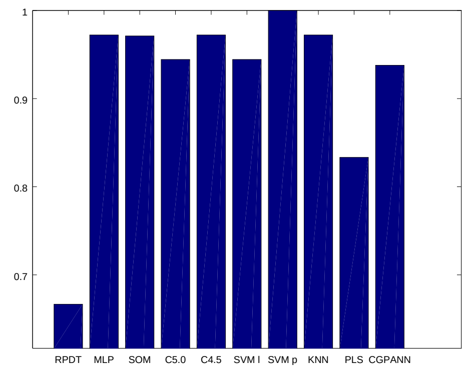
(b) DNA



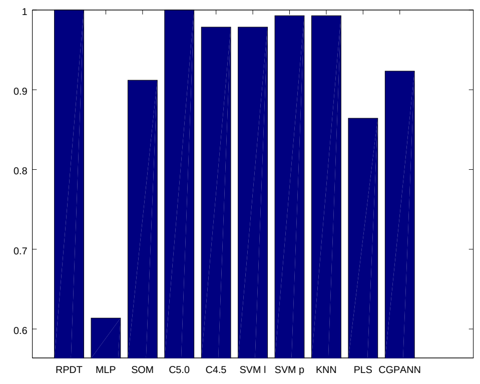
(c) Glass



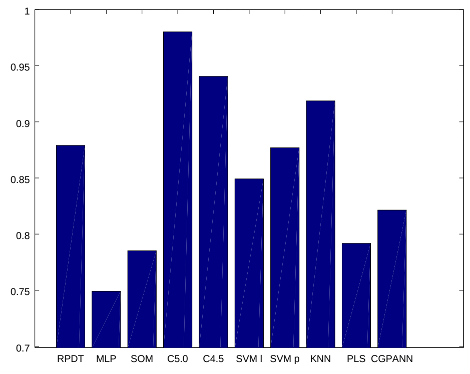
(d) Ionosphere



(e) Iris



(f) Olives



(g) Segmentation

Figure 9.1: Results given in Table 9.1, presented as bar charts.

Table 9.2: Closer Comparison of MLPs and CGPANN

Benchmark	MLP	CGPANN	U-test	Effect Size
Cancer	**0.9991**	0.9731	2.57E-12	0.98880
DNA	**0.9978**	0.8990	1.99E-12	1.00000
Glass	**0.8000**	0.5950	5.08E-12	0.99040
Ionosphere	**0.9991**	0.8901	7.27E-13	1.00000
Iris	**0.9722**	0.9378	8.83E-14	1.00000
Olives	0.6137	**0.9234**	2.00E-12	1.00000
Segmentation	0.7490	**0.8214**	2.16E-12	1.00000

9.8 Discussion

As can be seen from Section 9.7, CGPANN does not represent an effective method in the domain of classification. Although in most cases CGPANN did produce classifiers which performed reasonably, they were never competitive with the majority of the other methods used for comparison. Additionally, when specifically compared to MLP, an alternative popular method for training ANNs, it is shown that in the majority of cases MLP outperformed CGPANN.

Therefore, it appears conclusive that CGPANN performs poorly in the domain of classification. This result applies generally in the field of classification and specifically compared to other methods for training ANNs.

Interestingly, a search of the literature reveals that there is surprisingly little comparative study of the use of NeuroEvolution (NE) with standard methods in the domain of classification. However, it has been shown previously that NeuroEvolution of Augmenting Topologies (NEAT) also performed very poorly as a classification method when compared to standard back propagation [41]. Interestingly, a combination of NEAT and back propagation has previously been shown to outperform standard back propagation alone [41]. Additionally, an alternative NE method was also seen to outperform standard back propagation only when combined with back propagation [210].

It therefore appears, due to the lack of publications, the fact that other NE have been shown to perform poorly, the common combination with back propagation, and the poor results presented in this chapter, that standard NE does not perform well in the domain of classification.

However, this result does not demonstrate that NE is not useful in other domains. For instance, NE has been shown to be extremely effective in the domain of reinforcement type control tasks [64, 112, 313]. Additionally, CGPANN is shown in Chapter 10 to be extremely effective in the domain of series forecasting. Therefore, although it may be the case that NE is not effective in the domain of classification, it does not mean it is not effective in other domains.

Interestingly, the results obtained here using CGPANN on the breast cancer dataset are very similar to those presented previously for CGPANN [5,8,191]; here 97.3%, 98% in [5,8] and 97.2% in [191]. However, in previous research it was concluded that CGPANN was a highly competitive classification method [5, 8, 191]; whereas here it has been concluded that CGPANN performs very poorly. It appears this discrepancy lies not in the presented performance of CGPANN, but in results presented for the comparative methods; see Table 9.3.

Table 9.3: Previously presented classification performance of standard classification algorithms and CGPANN on the breast cancer benchmark.

Benchmark	Presented here	Presented in [5, 8]
MLP	0.9991	0.96
SVM (linear)	1.0000	0.94
KNN	1.0000	0.97
CGPANN	0.9731	0.98

As can be seen in Table 9.3, the performance of the comparative methods presented in [5, 8] are substantially and consistently worse than has been presented here. This led to CGPANN previously being reported as a highly effective classification method.

It is important to note that the performance of the comparative classification methods presented in [5, 8] were all taken from other sources. However, it is clear that the performance of these other methods has been misrepresented, leading to CGPANN previously being show to be more effective than it truly is. Although the cause of this discrepancy is unknown, it is likely due to the experimental set up used. For instance, it may be the case that the versions of the datasets, or the data split, or the methods used to determine the parameters used were different; resulting in a drastically different performance being presented. Regardless of the cause, it serves to confirm the importance of conducting fair and rigorous comparative studies.

Finally, it may be the case that the presented methodology for comparing classification methods is unfair; despite being the standard used in the Machine Learning (ML) literature. For instance, in the EA community it is common to compare the number of evaluations required to reach a given level of fitness, or the fitness which is reach after a given number of evaluations. The justification of this type of comparison is that algorithms are compared given the same level of computational expense. It is assumed that computation expense is proportional to the number of fitness evaluations. However, in the wider ML literature, classification methods are compared solely on their testing performance; regardless of the computational overhead in reaching that performance.

However, if algorithm A outperformed algorithm B, in terms of classification error, but required *significantly* more computational resources to produce this better accuracy, is it a superior algorithm? It appears that some measure of classification accuracy in relation to the computational cost should be considered.

The challenge is how to measure computational cost. Simply using "wall clock time" or the number of arithmetic and logical operations performed is not appropriate as they are a function of the algorithm, the specific hardware used, how well optimised the implementation is, and the programming language. This is why the number evaluations is used in the field of EA; as it is implementation agnostic. However, the notion of evaluations does not apply outside of the field of EAs. Another measure of computational expense is epochs; as used by back prorogation. However, again the notion of epochs has little meaning to other ML methods. Finding a fair measure of computational expense suited to all classification methods is therefore an open challenge for the machine learning community. Until this is found, comparisons can only rely on final testing performance.

Note that this discussion on comparing classification methods is not implying that CGPANN would perform better under differing figures of merit. Methods based on EAs are typically computationally expensive due to their generational and stochastic nature. In fact, it may be the case that CGPANN compares more unfavourably given computational expense considerations. This discussion is presented as the author believes there is currently a general issue concerning how classification methods are compared in the ML literature.

9.9 Summary

This chapter has presented an investigation into the suitability of CGPANN as a classification method. Here it has been shown that CGPANN does not represent an effective classification method; consistently producing one of the worse classifiers of the ten methods investigated.

Previous applications of CGPANN to classification have been discussed along with reasons why previous publications falsely reported CGPANN as a highly competitive classification method. The main cause of this discrepancy is likely due to the performance of other classification methods being understated. The reason for this understating is likely due to differences in benchmark implementation. To this end, the results presented in this chapter are much more rigorous than those presented previously for CGPANN.

Additionally, it has been suggested that NE may be ineffective as a classification method generally. For instance, there is very little published work applying NE to classification. Of the work which has been published, the popular NEAT method was shown to be ineffective at classification unless combined with back propagation; as were other NE methods.

Therefore, important future work is the combination of CGPANN and back propagation. As this union has been shown to be highly effective for other NE methods [41, 210, 314], it is likely to result in a more effective classification performance for CGPANN. Additionally, the application of a wider number of NE methods to classification would confirm the hypothesis that it is generally ill-suited as a classification method.

Chapter 10

Recurrent CGPANN Applied to Series Forecasting

This chapter describes and presents the application of Recurrent Cartesian Genetic Programming of Artificial Neural Networks (RCGPANN) to the domain of series forecasting. As well as comparisons to a range of standard forecasting techniques, RCGPANN is also compared to Cartesian Genetic Programming (CGP), Recurrent Cartesian Genetic Programming (RCGP) and Cartesian Genetic Programming of Artificial Neural Networks (CGPANN). These comparisons identify which of the utilised CGP extensions are beneficial in the domain of series forecasting.

10.1 Structure of this Chapter

Section 10.2 provides a background to NeuroEvolution (NE) applied to series forecasting with a specific focus on previous applications of CGPANN. Section 10.3 describes how CGPANN is extended to be capable of evolving Recurrent Artificial Neural Network (RANN). Section 10.4 describes how CGP, RCGP, CGPANN and RCGPANN are applied as a series forecasting method. Section 10.5 described the standard forecasting methods which are used for comparison in assessing the suitability of RCGPANN as a series forecasting methods. Section 10.6 presents the results of applying the forecasting methods on a number of standard benchmark tasks. Finally, Section 10.7 gives a discussion of the presented results and Section 10.8 gives a closing summary of the chapter.

10.2 Background

The aim of this chapter is to introduce and investigate the suitability of RCGPANN. The application used for this evaluation of RCGPANN is series forecasting [50, 127]. Series forecasting is an important application of machine learning and statistical modelling techniques, including Genetic Programming (GP) [76, 137, 176, 247] and Artificial Neural Network (ANN)s [110, 322, 323], finding application in many disciplines including: economics, politics and planning.

Series forecasting is also a common application of NE [52, 149, 267] including CGPANN [149, 239]. Additionally, an alternative recurrent form of CGPANN, using an imposed Jordan type architecture[1] [136], has also previously been used to create a form of recurrent CGPANN; again with application to series forecasting [150, 321]. Although this method is not based on RCGP and is much more limited in terms of the topologies which can be produced. For instance, the user must decide, in advance of training, how many recurrent connections will be used. Previous applications of CGPANN to series forecasting also makes use of connection switch genes. However, as is discussed in Chapter 4.6, these are an unnecessary addition to CGPANN. The RCGPANN used in this chapter is the application of RCGP, described in Chapter 5, to the training of RANN; not simply the use of an imposed Jordan structure.

Additionally, a number of issues can be identified in previous comparisons of a recurrent form of CGPANN with other series forecasting methods [321]. For instance, the number of nodes used in previously presented results [150, 321] were chosen as they produced the best testing performance. However, this practice breaks the very reason for using training and testing sets; to assess generalisation to unseen data. No level of selection should be based on testing performance, including the number of available nodes. This means that the comparisons made to other methods, such as presented in [150, 321], become invalid.

In this chapter the performance of RCGPANN is evaluated as a series forecasting method by comparing its effectiveness with two naïvely and three more complex standard forecasting techniques: random walk forecasting (RWF), Mean Forecast (MEAN), Exponential Smoothing (ETS), Autoregressive Integrated Moving Average (ARIMA) and Multilayer Perceptrons (MLP) respectively. The comparison with at least two naive and two complex standard forecasting methods, including ARIMA, follows the methodology

[1]A topology where certain outputs are made available as inputs.

recommended by Hyndman [127], an acknowledged expert in the field of forecasting, on benchmarking new forecasting methods[2]. Comparisons to MLPs are also made as they represent the current standard approach for training ANNs.

This chapter also examines the performance of standard CGP, RCGP and CGPANN as series forecasting methods. This enables an evaluation of the various extensions which have been applied to CGP in order to create RCGPANN. Firstly, the benefit of the recurrent extension is evaluated by comparing CGP and RCGP as well as comparing CGPANN with RCGPANN. Secondly the benefit of optimising ANNs, rather than using standard mathematical functions commonly used by GP, is evaluated by comparing CGP and CGPANN as well as RCGP and RCGPANN. These comparisons allow insight into which aspects of the RCGPANN approach are beneficial.

10.3 Recurrent CGPANN

RCGPANN is implemented by applying the same recurrent extension used by RCGP, see Chapter 5, to CGPANN, see Chapter 4. This is undertaken to allow CGPANN to evolve RANNs. The modifications required to extend CGPANN to RCGPANN are the same as those required to extend CGP to RCGP. The requirement of all connection genes to be acyclic is lifted and the probability of mutation creating recurrent connections is controlled via a recurrent connection probability. As with RCGP, the chromosomes are executed by applying each set of inputs, updating each active node/neuron once in index order (i), and then reading the outputs. Again, as with RCGP, this can result in node/neuron outputs being read before they have been calculated. Therefore, again as with RCGP, each node/neuron is set to output zero until they have calculated their own output value.

Once these changes have been incorporated, RCGPANN can be used to evolve recurrent ANNs. It is important to note that RCGPANN can create feed-forward *and* recurrent ANNs; as allowing recurrent connections does not force evolution to use them. Additionally, RCGPANN can be easily restricted to creating only feed-forward ANNs by setting the recurrent connection probability to zero. RCGPANN is therefore a superset of CGPANN.

[2]This particular advice is given on his personal blog `http://robjhyndman.com/hyndsight/benchmarks/`.

10.4 Applying CGP, RCGP, CGPANN and RCGPANN to Series Forecasting

In this chapter CGP, RCGP, CGPANN and RCGPANN are applied to series forecasting using a recursive forecasting method [106, 135]. This method involves the feedback of previously made forecasts as inputs to be used in the prediction of subsequent forecasts. Using this method it is possible to make forecasts to any given horizon.

A common technique used by forecasting techniques is to calculate the embedding dimension (D) and time delay (T) of the training data. This provides a suitable number of past data points, and a suitable number of time steps between these data points, in order to accurately predict the next data point. For instance if $D = 4$ and $T = 2$ then the inputs would be $[x(t), x(t-2), x(t-4), x(t-6)]$; where t indexes each sample in the series $x()$ and the task is to predict $x(t+1)$. Here, suitable embedding dimensions and time delays are found for each benchmark and these determine the number of past values to be used as inputs. The embedding dimensions and time delays are found using the pdc package [34] for the R programming language [234][3].

As an example, Figure 10.1 shows how recursive forecasting using multiple previous values is used. Here $D = 3$ and T is left unspecified. The buffer, containing $x(t-4)$ through to $x(t)$, is initially populated with known observed values which are replaced with predicted values during the recursive forecasting process. The inputs to the network are initially taken from the known observed values, but as the forecast progresses into the future, the inputs are taken from the buffer of previously made forecasts.

A minor disadvantage of using multiple inputs determined by D and T is that it reduces the amount of training data which can be used. For instance if $D = 2$ and $T = 2$ at time $t = 0$, $x(t-2)$ is before the start of the training data and so $x(t+1)$ cannot be predicted.

The fitness function used in this chapter represents how well the solutions recursively predict sections of the training data. This is achieved by recursively predicting the next fifty samples[4] from $t = 50$, $t = 100$, ..., $t = 950$. The predictions start from $t = 50$ and not $t = 0$ to compensate for the use of embedding dimensions and time delays removing the first few samples from the training data. The fitness awarded is the Mean Square

[3]For reference the exact function used was *entropy.heuristic*.

[4]The number of predictions could take any value. Fifty was used here as a compromise between forecasting to a similar horizon required by the testing data and allowing for a reasonable number of separate forecasts.

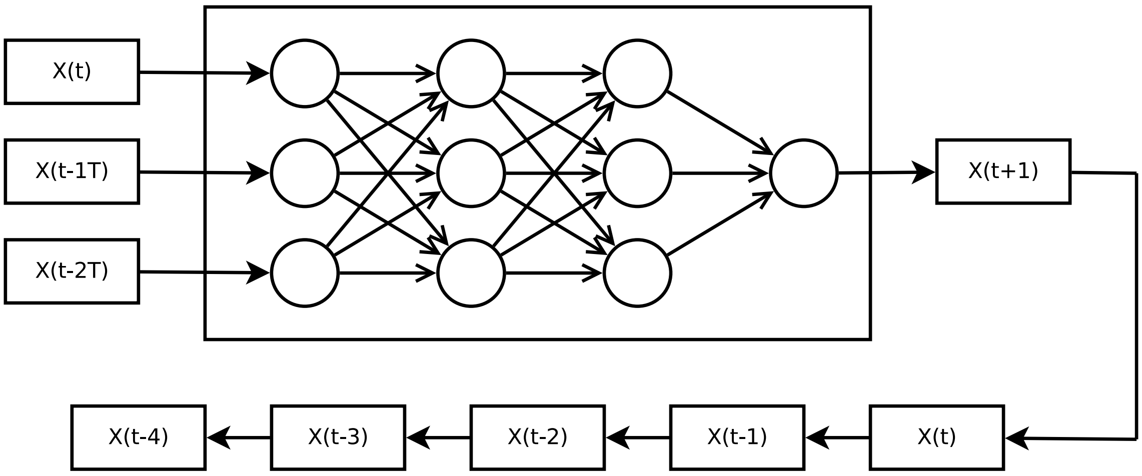


Figure 10.1: Depiction of recurrent forecasting and the use of embedding dimension and time delay to determine the number of inputs; $D = 3$.

Error (MSE) between the predicted and observed values.

Unlike feed-forward programs, when using RCGP and RCGPANN the outputs are a function of the current inputs *and* the current program state (internal node outputs). This means the program must be 'primed' before it can be used to make forecasts. The priming process is to apply previous observed values to the program, in sequence, and execute the program in each case. The outputs are not used. This causes the internal nodes to calculate suitable values before the forecasting begins. Here, when using RCGP and RCGPANN, the previous 50 samples from each starting point are applied to the network before making future predictions. For instance, if the predictions were to be from $t = 150$ then all the values from $t = 100$ to $t = 150$ are applied in turn and the program executed in each case. Once this is undertaken, the forecasting process can begin.

A disadvantage of many machine learning techniques is that they can easily over train on the training data and consequently lose their ability to generalise. CGP and its derivatives are no exception and are also likely to suffer from over-training when applied to series forecasting. For this reason, a validation scheme is used to prevent over-training. Here, generalisation is assessed by recording how well the solutions perform beyond the forecast horizon used during training. Starting at times $t_1 = 100$, $t_2 = 200$, ..., $t_9 = 900$ the programs are used to make forecasts up to a horizon of 100 samples. The MSE of the forecasts occurring between a time horizon of $t_i + 50$ samples and $t_i + 100$ samples are then used as a validation fitness score.

The validation score could be used by any of a range of early stopping techniques [230] in order to prevent over-training. However, the choice of early stopping technique is likely

Table 10.1: Parameters by CGP and its derivatives.

Parameter	CGP	RCGP	CGPANN	RCGPANN
Evolutionary Strategy	(1+4)-ES	(1+4)-ES	(1+4)-ES	(1+4)-ES
Max Generations	10,000	10,000	10,000	10,000
Mutation Scheme	probabilistic	probabilistic	probabilistic	probabilistic
Mutation Rate	3%	3%	1%	1%
Recurrence	0%	10%	0%	10%
Number of Nodes	100	100	100	100
Node Arity	2	2	5	5
Weight Range	-	-	± 5	± 5
Transfer Function(s)	$+,-,*,/$,sin, cos,exp,log	$+,-,*,/$,sin, cos,exp,log	logistic	logistic

to influence results. For this reason, here, the chromosome which is awarded the best validation score is retained throughout evolution and is used as the final chromosome to be assessed using the testing data. For instance, if the chromosome with the best validation score was found on generation 100, after the maximum number of generations have elapsed, this chromosome is used as the final chromosome to be evaluated on the testing data. Although this means the training does not stop early, in terms of the overall training time, it does help prevent over training.

In the work presented, the parameters specified in Table 10.1 are used. These parameters are relatively 'off-the-shelf' choices and have not been optimised for each benchmark. CGPANN uses a lower mutation rate to accommodate the fact it is more suited to gradual hill climbing though the adjustment of connection weights; standard CGP relies on slightly larger beneficial mutations.

10.5 Comparative Methods

A number of comparative methods are used to evaluate the performance of RCGPANN; as well as CGP, RCGP and CGPANN. These comparative methods comprise RWF, MEAN, ETS, ARIMA and MLPs. These methods are used to compare RCGPANN to standard forecasting techniques and to the more common method of training ANNs.

10.5.1 Random Walk Forecasting

The random walk forecasting method is a very simple naïve forecasting technique which is useful to compare new forecasting methods against; as any newly proposed forecasting method should at least be able to outperform it. RWF predicts that all future unknown values are equal to the last observed value.

10.5.2 Mean

The mean forecasting method is again a very simple naïve forecasting technique which is also useful to compare new forecasting methods against. The mean forecasting method predicts that all future values are equal to the arithmetic mean of the observed values i.e. the training set.

10.5.3 Exponential Smoothing

Exponential smoothing [114] is a popular forecasting technique which, in its simplest form, bases its prediction on a weighted average of previous observations. Commonly the further ahead the prediction is from the last observation the more previous values are used in the weighted average.

The exponential smoothing used in this chapter is from the Forecast package [128] for the R programming language [234]. When creating exponential smoothing models the *ets* function is used to find suitable parameters using the methods described in [126].

10.5.4 Autoregressive Integrated Moving Average

Autoregressive integrated moving average [5] [32] is a popular generalised forecasting technique. ARIMA models use a collection of three forecasting techniques: autoregressive (AR), integrated (I) and moving average (MA); hence ARIMA. ARIMA models are often written in the form ARIMA(p,d,q), with the p, d and q values referring to the AR, I and MA aspects of the ARIMA model respectively. By using different p,d and q parameters ARIMA models can implement a wide range of forecasting techniques including random walk, random trend, autoregressive and exponential smoothing models.

The ARIMA implementation used in this chapter is from the Forecast package [128] for the R programming language [234]. When creating ARIMA models the *auto.arima* function [125] is used to find suitable p, d and q parameters as well as further sub parameters associated with the specific model. The *auto.arima* function uses a variation of the Hyndman and Khandakar algorithm [125] to obtain a suitable ARIMA model.

[5] Also referred to as Box-Jenkins after the original authors.

10.5.5 Multilayer Perceptron

Multilayer Perceptrons are a standard ANN training method which makes use of the back propagation algorithm. When applied to series forecasting it is common practice to use multiple inputs determined by the embedding dimension and time delay of the series; so this is undertaken here.

The MLP implementation used in this chapter is the Fast Artificial Neural Network Library (FANN) [217]. The FANN library is configured to use standard fully connected ANNs of unipolar logistic sigmoid transfer functions trained using a variant on back propagation called resilient back propagation (Rprop) [241] for 1000 epochs. As back propagation does not optimise topology, a range topologies are investigated comprising one and two hidden layers of five, ten, twenty and fifty nodes per hidden layers (eight separate topologies in total).

As MLPs use a strictly supervised learning method they must be trained using input-output pairs. However, this style of learning is not directly compatible with recursive forecasting. This is because future forecasts are made using previously made forecasts. When using previous forecasts as inputs, the input-output pairs do not represent a correct learning example; as the inputs are based on a previous prediction. In this case the ANN would be trained using incorrect data.

Therefore, here, the MLPs are trained for one-step-ahead prediction; always using valid input-output pairs. The recursive forecasting performance of the ANN is then recorded after each epoch by using the ANN to recursively predict the next 100 samples starting at $t = 100$, $t = 200$, ..., $t = 900$. After the maximum number of epochs have elapsed the configuration which resulted in the best recursive forecasting performance is then returned as the final trained ANN. This method effectively trains for one-step-ahead prediction and uses the recursive forecasting performance to prevent over-training.

10.6 Results

The results presented investigate the suitability of RCGPANN as a series forecasting method. This is achieved by comparing RCGPANN with a range of standard series forecasting methods; described in Section 10.5. Additionally, the two extensions to CGP utilised by RCGPANN, recurrence and application to ANNs, are also investigated by applying CGP, RCGP and CGPANN to series forecasting.

Three standard benchmark tasks are used in this comparative study: Laser, Mackey-Glass and Sunspots. These benchmarks are all described in Appendix A.

There are many measurements found in the literature which are used to assess the performance of forecasting methods [17,129]. However, in the machine learning literature the most commonly used methods are the MSE, Root Mean Square Error (RMSE) and the Normalised Mean Square Error (NMSE). For this reason MSE and NMSE are used[6]; despite other measurements possibly being more representative of forecasting accuracy [17,129].

The MSE and NMSE are given in Equations 10.1 and 10.2 respectively where: N is the number of predicted samples, p_i is the i^{th} predicted value, o_i is the i^{th} observed value and $\bar{o}$ is the average of all the observed values. Note that the NMSE measurement gives the MSE normalised by the MSE which would be achieved if all predictions were equal to the arithmetic mean of the observed values.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(p_i - o_i)^2 \tag{10.1}$$

$$\text{NMSE} = \left(\frac{\sum\limits_{i=1}^{N}(p_i - o_i)^2}{\sum\limits_{i=1}^{N}(o_i - \bar{o})^2} \right) \tag{10.2}$$

The forecasts produced by the various forecasting methods are evaluated on the testing data using the two measures described. For the stochastic methods (CGP, RCGP, CGPANN, RCGPANN and MLP) the average[7] performance of 50 runs is used for comparison; as this represents typical performance. Additionally, the testing performance of the run which scored the best training fitness is also presented. In a real world scenario, this is likely to be the forecaster which would be used. Note this is not the solution which produced the best testing fitness, as selection should never be (and typically cannot be) based on testing performance.

In the case of MLPs, many topologies were investigated. Here, the results of using the best topology are presented. The best topology is determined by the average training performance; not the testing performance which would typically not be known in advance. Specifically the recursive prediction performance on the training set is used as this more

[6]As RMSE is simply the root of the MSE value it is not also explicitly presented.

[7]Arithmetic mean.

closely matches the final application.

Again, in the case of the stochastic methods, statistical significance testing is used to assess any differences. The non-parametric Mann-Whitney U-test and the non-parametric Kolmogorov-Smirnoff test (KS) are used to test for statistical significance; with $\rho \leq 0.05$ representing statistical significance. Additionally the effect-size, as defined in [285], is also used to indicate the importance of any statistical difference; with values > 0.56 indicating a small effect size, > 0.64 a medium and > 0.71 a large. For a more in-depth description of these statistical significance measures, and a justification of their use, see Appendix B. The spread of results are also given graphically as box and whisker plots for visual inspection; with outliers marked as follows: '+' represents forecasts between 1.5 and 3 times the interquartile range and '$\circ$' represents forecasts greater than 3 times the interquartile range.

The forecasts produced by each method are also given in Figures 10.5, 10.6 and 10.7; for each benchmark respectively. In the case of the stochastic methods, the best forecast, as previously defined, is presented.

10.6.1 Laser

The results of applying the investigated series forecasting methods to the Laser benchmark are given in Table 10.2. In the case of the stochastic methods, Table 10.3 gives the statistical analysis and Figure 10.2 gives the box and whisker plots. The forecasts produced are plotted in Figure 10.5.

The MLP topology which produced the best recursive forecast on the training set used two hidden layers each containing five nodes. The ARIMA model produced was ARIMA(5,0,4) with non-zero mean.

Overall it can be seen in Table 10.2 that RCGPANN produce the best average forecast of all the other methods used for comparison.

When comparing CGPANN and MLPs as training methods for feed-forward ANNs, it can be seen that, on average, CGPANN strongly outperformed MLP with statistical significance and a medium effect size. This indicates that CGPANN is providing a superior training method to MLPs.

When evaluating the recurrent extension of RCGP it can be seen that, on average, RCGP outperformed CGP with statistical significance and a close to medium effect size. Additionally, it can be seen that, on average, RCGPANN outperformed CGPANN with statistical significance and a large effect size. This indicates than the recurrent extension is advantageous to both CGP and CGPANN.

Finally, when comparing evolving ANNs to the use of standard GP mathematical functions, it can be seen that, on average, CGPANN and CGP produced very similar average results with no statistical significance or meaningful effect size. It can also be seen that RCGPANN outperforms RCGP with statistical significance and a medium effect size. This indicates that evolving ANNs does not produce worse results than using standard GP mathematical functions and can produce superior results.

Table 10.2: Results from applying various forecasting methods to the Laser benchmark.

Method	MSE		NMSE	
	Avg	Best	Avg	Best
RWF	0.034227		1.260675	
MEAN	0.027151		1.000030	
ETS	0.034223		1.260508	
ARIMA	0.034148		1.257749	
MLP	0.043985	0.035237	1.620058	1.000184
CGP	0.027946	0.027091	1.0292	0.997707
RCGP	0.025823	0.004424	0.95100	0.162913
CGPANN	0.027655	0.02938	1.0185	1.081971
RCGPANN	0.021467	0.016424	0.79058	0.604839

Table 10.3: Statistical significance testing between the stochastic methods applied to the Laser benchmark.

Comparison	U-Test	KS-Test	Effect Size
MLP - CGP	3.75E-2	3.76E-8	0.66820
MLP - RCGP	5.92E-6	2.97E-9	0.76280
MLP - CGPANN	1.49E-2	7.84E-10	0.68440
MLP - RCGPANN	1.21E-10	7.84E-10	0.87340
CGP - RCGP	3.64E-2	8.90E-3	0.62160
CGPANN - RCGPANN	1.43E-5	1.08E-8	0.75200
CGP - CGPANN	4.04E-1	5.08E-1	0.54860
RCGP - RCGPANN	6.81E-3	4.43E-3	0.65720

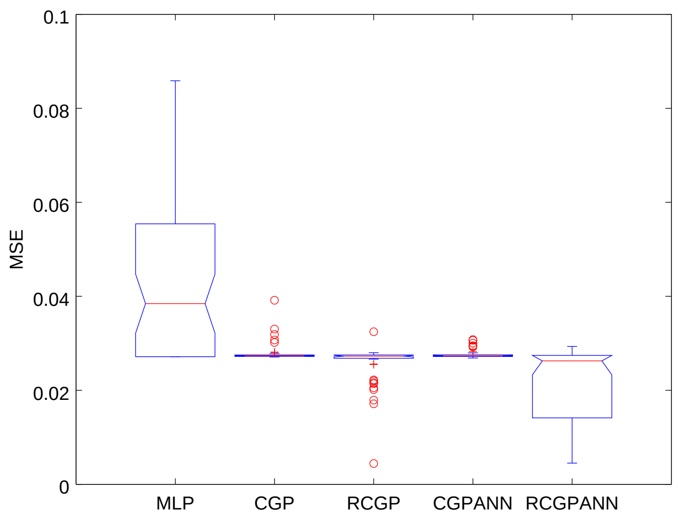


Figure 10.2: Spread of the forecasts produced using stochastic methods on the Laser benchmark.

10.6.2 Mackey-Glass

The results of applying the investigated series forecasting methods to the Mackey-Glass benchmark are given in Table 10.4. In the case of the stochastic methods, Table 10.5 gives the statistical analysis and Figure 10.3 gives the box and whisker plots. The forecasts produced are plotted in Figure 10.6.

The MLP topology which produced the best recursive forecast on the training set had one hidden layer containing twenty nodes. The ARIMA model produced was ARIMA(3,0,5) with non-zero mean.

Overall it can be seen in Table 10.4 that RCGPANN produces the best average forecast compared with all the other methods.

When comparing CGPANN and MLPs as training methods for feed-forward ANNs it can be seen that on average CGPANN outperformed MLPs but the difference was not statistically significant and the effect size was very small. This indicates that MLPs and CGPANN represent similarly suitable training methods for feed-forward ANNs.

When evaluating the recurrent extension of RCGP it can be seen that on average RCGP outperformed CGP with statistical significance and a nearly medium effect size. Additionally, on average RCGPANN outperformed CGPANN with statistical significance and a large effect size. This indicates that the recurrent extension is advantageous to both CGP and CGPANN.

Finally, when comparing evolving ANNs to the use of standard GP mathematical functions it can be seen that CGPANN and CGP produced very similar average results with no statistical significance or meaningful effect size. Additionally RCGPANN strongly outperformed RCGP on average with statistical significance and a large effect size. This indicates that evolving ANN can produce better results than standard mathematical functions or at least does not produce worse results.

Table 10.4: Results from applying various forecasting methods to the Mackey-Glass benchmark.

Method	MSE		NMSE	
	Avg	Best	Avg	Best
RWF	0.109334		1.624736	
MEAN	0.067324		1.000447	
ETS	0.357603		5.314079	
ARIMA	0.071481		1.062226	
MLP	0.075798	0.048297	1.126385	0.717701
CGP	0.069947	0.058746	1.0394	0.872979
RCGP	0.064501	0.025706	0.95850	0.381999
CGPANN	0.065563	0.049188	0.97428	0.730944
RCGPANN	0.047575	0.033219	0.70698	0.49364

Table 10.5: Statistical significance testing between stochastic methods on the Mackey-Glass benchmark.

Comparison	U-Test	KS-Test	Effect Size
MLP - CGP	2.34E-1	3.63E-6	0.56920
MLP - RCGP	9.42E-1	2.11E-2	0.50440
MLP - CGPANN	7.59E-1	1.78E-4	0.51800
MLP - RCGPANN	1.70E-4	4.23E-4	0.71840
CGP - RCGP	2.29E-2	1.71E-2	0.63220
CGPANN - RCGPANN	5.19E-6	1.02E-5	0.76460
CGP - CGPANN	2.87E-1	6.78E-1	0.56200
RCGP - RCGPANN	8.15E-5	1.78E-4	0.72880

10.6.3 Sunspots

The results of applying the investigated series forecasting methods to the Sunspots benchmark are given in Table 10.6. In the case of the stochastic methods, Table 10.7 gives the statistical analysis and Figure 10.4 gives the box and whisker plots. The forecasts produced are given in Figure 10.7.

One oddity seen in the results is the extremely poor average forecast achieved using RCGP. This was due to one of the fifty runs producing multiple large spikes in the middle of the forecast, resulting in a *very* large error which strongly influenced the average. If this one run is removed, the average results in a MSE of 0.026701 and a NMSE of 0.910010;

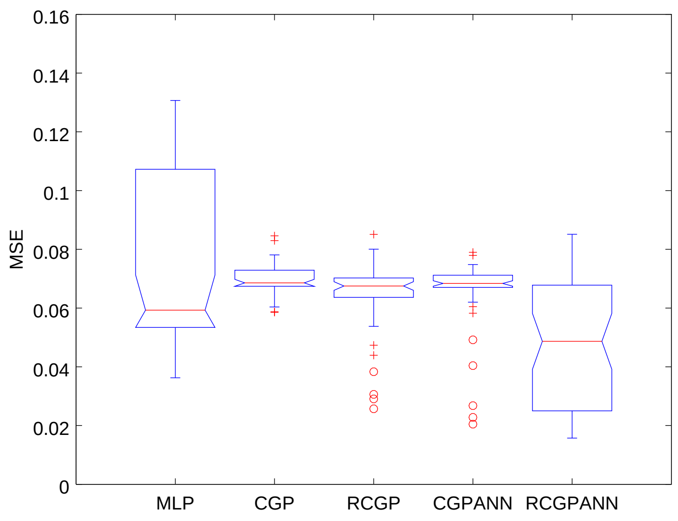


Figure 10.3: Spread of the forecasts produced using stochastic methods on the Mackey Glass benchmark.

which does outperform CGP. This outlier is also removed from the box-plots in Figure 10.4 in order for the other details to be visible.

The MLP topology which produced the best recursive forecast on the training set used two hidden layers each containing five nodes. The ARIMA model produced was ARIMA(5,1,4).

Overall it can be seen in Table 10.6 that the best average result was achieved using CGPANN and RCGPANN with almost no difference between the two.

When comparing CGPANN and MLPs as training methods for feed-forward ANNs it can be seen that on average CGPANN strongly outperformed MLPs with statistically significant and a large effect size. This indicates that CGPANN is much more effective at training ANN than MLPs.

When evaluating the recurrent extension of RCGP it can be seen that on average RCGP was strongly outperformed by CGP but with no statistical significance and a very small effect size. If the one very poor run is removed from the RCGP results, then RCGP does outperform CGP but still without statistical significance. Additionally, on average RCGPANN produces very similar results to CGPANN with no statistical significance and a very small effect size. This indicates that the recurrent extension is not providing an advantage or disadvantage.

Finally, when comparing evolving ANNs to the use of standard GP mathematical functions it can be seen that on average CGPANN strongly outperformed CGP with statistical significance and a high medium effect size. Although RCGPANN did outperform

RCGP (with or without the outlier) there was no statistical significance and a very small effect size. This indicates that evolving ANNs may be producing better results than standard mathematical functions, or at least does not produce worse results.

Table 10.6: Results from applying various forecasting methods to the Sunspots benchmark.

Method	MSE		NMSE	
	Avg	Best	Avg	Best
RWF	0.176262		6.008159	
MEAN	0.034399		1.172533	
ETS	0.546006		18.61142	
ARIMA	0.034972		1.192063	
MLP	0.043773	0.035228	1.492071	1.200790
CGP	0.031940	0.026894	1.0886	0.916592
RCGP	1.20e+30	0.011922	4.10E+31	0.406331
CGPANN	0.024991	0.018114	0.85175	0.61736
RCGPANN	0.024992	0.004925	0.85177	0.167851

Table 10.7: Statistical significance testing between stochastic methods on the Sunspots benchmark.

Comparison	U-Test	KS-Test	Effect Size
MLP - CGP	6.97E-13	2.13E-14	0.91680
MLP - RCGP	1.09E-11	1.09E-13	0.89440
MLP - CGPANN	6.31E-16	3.28E-18	0.96920
MLP - RCGPANN	8.10E-15	2.07E-17	0.95080
CGP - RCGP	1.31E-1	3.17E-2	0.58780
CGPANN - RCGPANN	8.12E-1	8.41E-1	0.51400
CGP - CGPANN	3.51E-3	4.43E-3	0.66960
RCGP - RCGPANN	6.52E-1	2.41E-1	0.52640

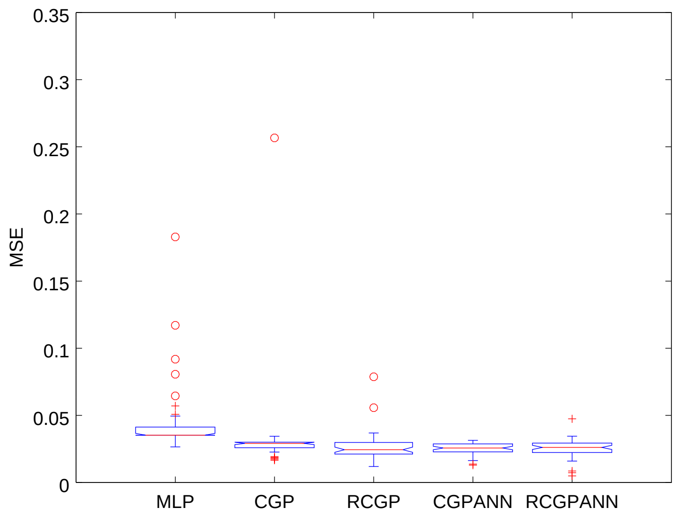
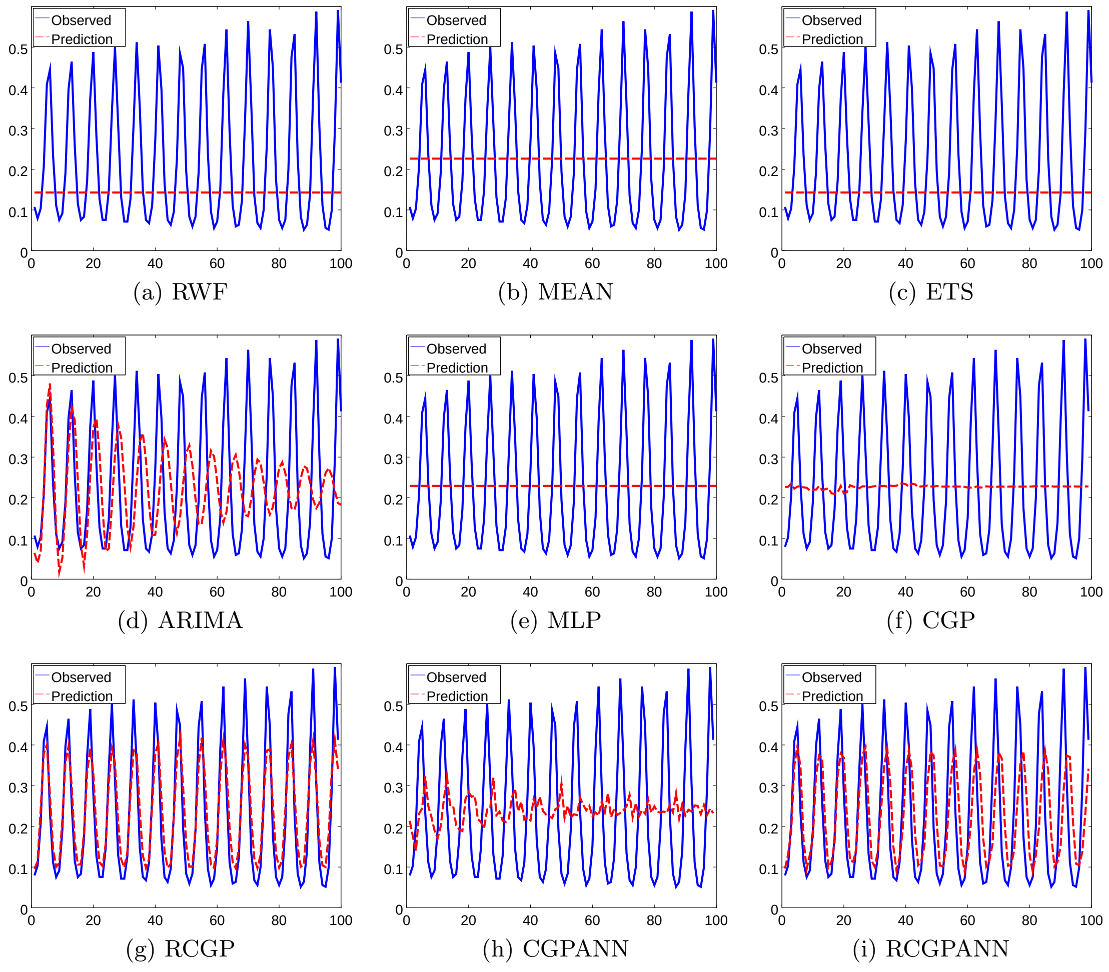


Figure 10.4: Spread of the forecasts produced using stochastic methods on the Sunspots benchmark.

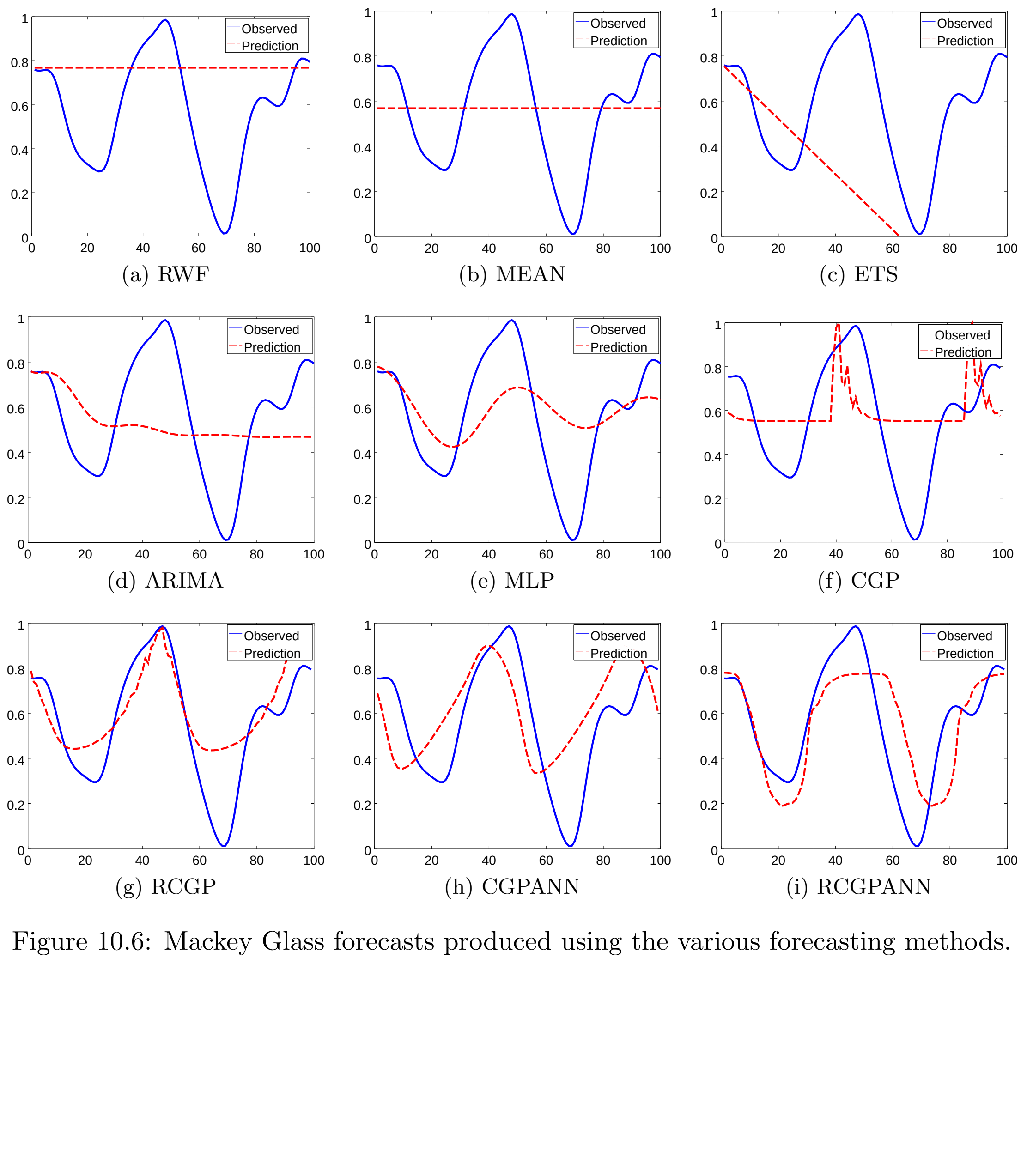


(a) RWF (b) MEAN (c) ETS

(d) ARIMA (e) MLP (f) CGP

(g) RCGP (h) CGPANN (i) RCGPANN

Figure 10.5: Laser forecasts produced using the various forecasting methods.



(a) RWF (b) MEAN (c) ETS

(d) ARIMA (e) MLP (f) CGP

(g) RCGP (h) CGPANN (i) RCGPANN

Figure 10.6: Mackey Glass forecasts produced using the various forecasting methods.

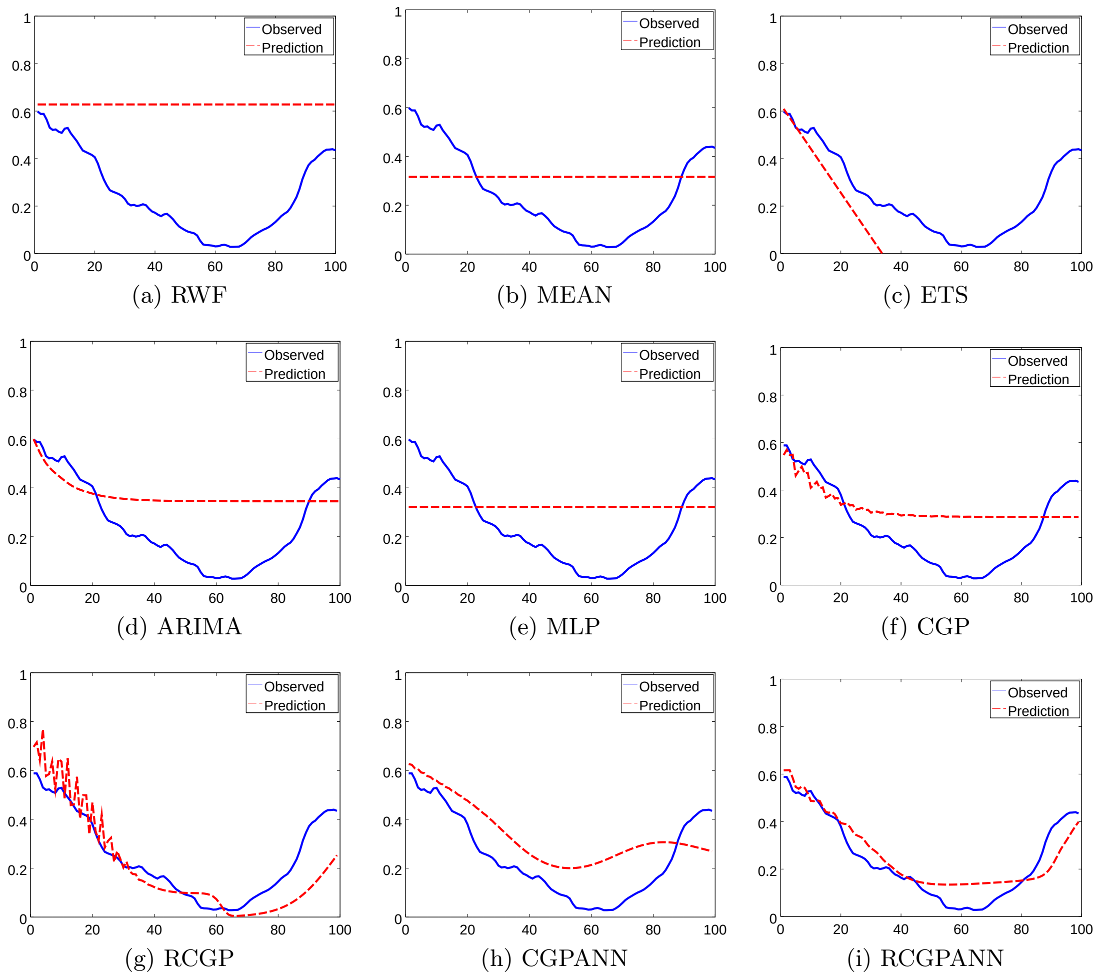


Figure 10.7: Sunspots forecasts produced using the various forecasting methods.

10.7 Discussion

As can be seen from the results, in all cases RCGPANN produced the best (or joint best) average forecasts compared with all the other methods used for comparison. This demonstrates that RCGPANN is a highly competitive series forecasting technique compared to standard methods. RCGPANN also outperformed all the other methods based on CGP: CGP, RCGP and CGPANN. This clearly demonstrates the suitability of the newly proposed RCGPANN method.

When comparing CGPANN with MLPs, CGPANN strongly outperformed MLPs on two of the three benchmarks and produced a better average forecast on the remaining benchmark; but without convincing statistical significance. Therefore it was demonstrated that CGPANN often outperformed MLPs and never produced a worse result. This indicates that CGPANN is a superior training method than MLPs in the domain of recursive series forecasting.

The results show that the addition of recurrent connections to CGP and CGPANN offer a significant advantage to series forecasting. On two of the three benchmarks both RCGP and RCGPANN outperformed CGP and CGPANN respectively with statistical significance. On the remaining benchmark the results were approximately equal between CGP and RCGP and between CGPANN and RCGPANN with no statistical significance. Therefore recurrent connections were often seen to be beneficial to CGP and CGPANN and never worse. This further demonstrates the suitability of using RCGP to create recurrent program structures, complementing previous research; Chapter 5.

Here, when applying CGP and its variants to series forecasting, multiple previous values from the sequence were made available to the evolved programs. These previous values were determined by the embedding dimension and time delay of the sequence so as to provide suitable values for making subsequent forecasts. Therefore, it would not have been surprising if the recurrent extension present in RCGP and RCGPANN failed to outperform the non-recurrent counterparts; as a form of recurrence has effectively already been added that has been specifically designed for predicting future values. The fact that RCGP and RCGPANN were shown to outperform their non-recurrent counterparts demonstrates that evolution managed to find additional recurrence which improved again upon the level of recurrence already provided. This also serves to demonstrate the benefit of the type of recurrence which can be created using RCGPANN, over simply using an enforced Jordan architecture [150,321]; where the user must effectively choose the level of recurrence ahead of training.

The results also demonstrated that the use of neuron transfer functions and connection weights produce better forecasts on average than the use of standard GP mathematical functions without connections weights; at least in the case of CGP. On two of the three benchmarks, CGP produced similar forecasts to CGPANN with no statistical significance but RCGPANN strongly outperformed RCGP with statistical significance. On the remaining benchmark, CGPANN strongly outperformed CGP with statistical significance but RCGP and RCGPANN produced similar forecasts with no statistical significance. Therefore the use of neuron transfer functions with connection weights never performed worse than the use of standard mathematical functions, and often produced superior forecasts.

The fact that the use of neuron transfer functions and connection weights produced superior results, in comparison with using standard mathematical functions, may have wider implications for GP in general. It could be the case that many GP methods could be improved by using neuron transfer functions, or the addition of connection weights, or the use of both connection weights and neuron transfer functions (thus implementing NE). It is true that both ANNs [48] and GP [315] are (or can be depending on the function set used) universal approximators. But this does not say anything about how trainable or evolvable they are. For instance, the addition of connection weights to GP, previously termed weighted GP [275], may make for a more evolvable fitness landscape. Future research should investigate the use of weighted connections and neuron transfer functions, independently and in union, for other GP methods. This may help indicate whether the power of ANNs is in their transfer functions, connection weights or training methods. This could even lead to interesting mixtures of GP and ANNs such as back propagation applied to a weighted form of GP.

A seemingly odd result is how well the MEAN method preformed compared to the other standard forecasting methods. MEAN was seen to outperform RWF, ETS, ARIMA and MLPs for all of the benchmarks investigated. However as described in [127] "*some forecasting methods are very simple and surprisingly effective*". Additionally it has previously been noted in the real-time forecasting M2-competition [187] that ARIMA (referred to as Box-Jenkins) "*proved to be one of the least-accurate methods and its overall median error was 17% greater than that for a naïvely forecast*" [16]. Therefore it can be seen that it is not uncommon for naïvely methods to perform very well.

Interestingly, many of the forecasts provided using CGP and it variants either produced an output very close to MEAN or exhibited behaviours which eventually settled on an output close to the mean value of the training set. Examples of this can be seen in: Figure 10.5 where CGP is applied to the Laser benchmark, Figure 10.6 where CGP is applied to the Mackey-Glass benchmark and Figure 10.6 where CGP is applied to the Sunspots benchmark. As MEAN was shown to produce a reasonable forecast, it may be the case that this method represents a local optimum in the search space. It could also be an example of evolutionary methods rediscovering a previously known technique.

Although not explored in this chapter, an additional advantage of using Evolutionary Algorithm (EA)s for forecasting is the ability to alter the fitness function to favour certain characteristics. For instance, the forecast horizon can easily be altered. The maximum error during the forecast could be considered. Frequency information from the training data could be used to award or penalise frequencies present or not present in the produced forecasts. The fitness awarded could represent the number of time steps predicted with an error lower than a given threshold, rather than the error up to a given forecast horizon. The solutions could be optimised for speed, complexity or size. As the ability to set custom fitness functions also applies to NE this is another possible benefit of its use in series forecasting.

Finally, the best result of the stochastic methods often represented a much better forecast than the average. Note that this is the best result as determined by the *training* data; not the *testing*. This is not surprising as early stopping methods were utilised to prevent over training. Although the average performance was used here for comparison, as it represents typical performance of the algorithms, in real applications the best of many runs would be used. In this case, RCGPANN could be argued to outperform the standard forecasting methods to an even greater extent than has been presented; although the comparison would be less rigorous.

10.8 Summary

This Chapter has introduced RCGPANN, a new NE method based on CGPANN which utilises the recurrent extension of RCGP. The application used here to assess the performance of RCGPANN is series forecasting. The results demonstrate that RCGPANN produces highly competitive forecasts, outperforming all of the other standard forecasting methods used for comparison. RCGPANN is therefore shown to be a powerful NE method;

at least in the domain of series forecasting.

RCGPANN differs from standard CGP in two regards; it allows recurrent connections and uses neuron transfer function with connections weights. Both of these aspects were individually investigated, revealing that they both provide significant benefit to standard CGP; at least in the domain of series forecasting. This demonstrates the importance of the two presented CGP extensions and helps explain why the RCGPANN approach was shown here to be so effective. This result may also be significant for other GP methods which could also benefit from similar extensions.

Finally, it is important to note that RCGPANN is a superset of CGPANN. By setting the RCGPANN recurrent connection probability to zero it implements standard feed-forward CGPANN. Additionally, just because RCGPANN is capable of utilising recurrent connections does not force evolution to utilise them. For instance, it is possible for RCGPANN to create purely feed-forward ANNs if there were an evolutionary advantage in doing so. This, coupled with the advantageous results presented, makes RCGPANN an important, significant extension to CGPANN.

Chapter 11

Conclusions and Further Work

This chapter presents the overall conclusions reached during this thesis in the fields of Cartesian Genetic Programming (CGP), Cartesian Genetic Programming of Artificial Neural Networks (CGPANN) and NeuroEvolution (NE) in general. A number of possible further works are also presented.

11.1 Structure of this Chapter

Section 11.2 provides the overall high-level conclusions which have been reached throughout the work presented in this thesis. Section 11.3 suggests a number of further investigations which would complement and extend the presented research. Finally, Section 11.4 gives a collection of final remarks.

11.2 Overall Conclusions

This thesis has reached many significant conclusions in the domains of CGP, CGPANN and the wider field of NE. These are now summarised.

11.2.1 Recurrent Cartesian Genetic Programming

A new extension to CGP has been developed which allows for the creation of recurrent program structures; namely Recurrent Cartesian Genetic Programming (RCGP). The proposed implementation is much simpler than the use of multiple chromosomes to facilitate recurrence [294], whilst being much more flexible than simply enforcing a Jordan type architecture [155]. The presented implementation requires minimal alteration to the underlying algorithm and has been demonstrated to be highly effective for tasks intractable to standard CGP. Additionally, the development of RCGP allows CGP to be applied to many new domains, such as those which require internal state information to be inferred and maintained, or those which require memory.

11.2.2 Bloat in Cartesian Genetic Programming

It has been previously demonstrated that CGP does not suffer from program bloat [201]. However, previous work only investigated its presence on Boolean circuit synthesis tasks and only considered raw program size as a measure of bloat. Work presented in this thesis complements previous work by extending the investigation to non-Boolean circuit tasks and by using a measure of bloat which considers program size in relation to fitness. The work confirms the previously observed result that CGP does not suffer from program bloat. This strengthens the previous evidence of a significant advantage over CGP over other Genetic Programming (GP) methods such as tree-based GP.

11.2.3 Genetic Redundancy in Cartesian Genetic Programming

It has previously been reported that CGP greatly benefits from the presence of inactive genes allowing for increased Neutral Genetic Drift (NGD) aiding the escape of local optima [202, 287, 318, 320]. However, previous research failed to isolate the benefit of explicitly inactive genes; the type of redundancy more unique to CGP. In work presented in this thesis, it has been demonstrated that the presence of explicitly inactive genes greatly aids CGPs evolutionary search and that its benefit is additive to other forms of genetic redundancy found more commonly in GP methods. It was also demonstrated that the identification and manipulation of explicitly inactive genes is far simpler and computationally cheaper than for other types of redundancy. This means that it is now possible to study and manipulate genetic redundancy in ways that were not previously possible. This is significant because as tasks become increasingly challenging, their search spaces becomes harder to navigate. Therefore mechanisms which aid the navigation of search landscapes become increasingly important as more challenging tasks are approached.

That is to say, the befit of NGD has the potential to scale favourably with application difficulty. Therefore the ability to easily identify, study and manipulate it behaviour could facilitate substantial improvements for CGP and other GP methods.

11.2.4 Connection Switch Genes

In the original implementation of CGPANN [154] connection switch genes were added in order to allow the evolution of variable node arity. However, no empirical results were presented demonstrating whether their addition provided any benefit to CGPANN. This thesis has demonstrated that CGPANN is capable of adapting node arity without the inclusion of connection switch genes; due to multiple connections between pairs of nodes being equivalent to a single connection. It has been empirically demonstrated that the inclusion of connection switch genes produces a slightly worse evolutionary search. There removal also makes the implementation of CGPANN simpler and removes the possibility of there being no complete path from inputs to outputs.

For these reasons, it is now recommended that CGPANN be implemented without the use of connection switch genes.

11.2.5 Bloat in Cartesian Genetic Programming of Artificial Neural Networks

This thesis has also demonstrated that, like CGP, CGPANN does not suffer from program bloat. Although this is an intuitive result, it is highly significant as if CGPANN had been shown to suffer from program bloat, it would have represented a major disadvantage. Additionally, the issue of bloat in the wider NE literature is largely overlooked. In the little work which does addresses the issue, it has been shown that NeuroEvolution of Augmenting Topologies (NEAT) suffers from program bloat unless careful parameter choices are made [274]. Additionally, other NE method including GeNeralized Acquisition of Recurrent Links (GNARL) and Evolutionary Acquisition of Neural Topologies (EANT), appear very likely to exhibit program bloat. Therefore, the fact that CGPANN has been shown not to suffer from program bloat may represent a significant advantage over many other NE methods. Although more researching concerning bloat and NE is generally needed.

11.2.6 Recurrent Cartesian Genetic Programming of Artificial Neural Networks

A substantial development made to the CGPANN algorithm is an extension which allows the evolution of Recurrent Artificial Neural Network (RANN)s. This extension follows the same principles as for RCGP. The described extension goes beyond previous attempts to use CGPANN to evolve RANN by simply feeding back a user defined number of outputs to be made available as inputs [155]. Using the newly proposed recurrent extension, it has been demonstrated that Recurrent Cartesian Genetic Programming of Artificial Neural Networks (RCGPANN)s outperforms all standard methods used for comparison in the domain of series forecasting. This indicates that RCGPANN represents an extremely powerful method for evolving RANNs worthy of further study and analysis.

11.2.7 Genetic Redundancy in Cartesian Genetic Programming of Artificial Neural Networks

The surprising result was presented that CGPANN, unlike CGP, does not benefit from explicit genetic redundancy. It was also shown that the benefit of all types of redundancy were greatly diminished. This is interesting as it appeared likely that NE methods, like many GP methods, would strongly benefit from the effect of redundancy aiding the escape of local optima. A possible explanation for this discrepancy has been proposed based on the influence of connection weights on the navigation of the fitness landscape. These connection weights may create a search space which is much more easily navigated; thus diminishing the need of NGD and other types of redundancy aiding the escape of local optima.

If it is the case that connection weights result in a search which benefits less from mechanisms which aid the escape from local optima, then it implies that their presence results in a search less prone to becoming trapped in local optima. Therefore, the inclusion of connection weights may be providing a more effective evolutionary search. This result may also be significant to future GP developments, which typically do not use weighted connections. An alternative view is that currently GP utilises evolutionary operators which cause large changed to the phenotypes semantics; changing only program structure or the transfer function of computational nodes. Therefore, it may be beneficial to include genes, such as connection weights genes, or evolutionary operators which also allow for smaller semantic changing mutations to be possible.

11.2.8 Topology Adaptation

The ability of many NE methods to adapt both the network topology and connection weights during evolution has long been considered a significant advantage of NE. These benefits include: removing the requirement that the user knows a suitable topology in advance of training, exploiting relationships between connection weights and topology during the search, and finally, utilising topologies which would be unlikely to be considered by a human designer.

However, upon inspection of the literature there is surprisingly little empirical evidence to support these perceived advantages. One of the contributions of this thesis has been a rigorous empirical investigation demonstrating that the ability for many NE methods to adapt network topology does indeed represent a significant advantage for NE. Now this is confirmed, researchers can more confidently cite this ability of NE. Results presented also demonstrate that topology adaptation may even be more significant to training than connection weight manipulation. This gives further insight into NE, demonstrating that it potential lies in its ability to manipulate network topology.

11.2.9 Heterogeneous Artificial Neural Networks

In the NE literature many researchers have called for further investigations into the benefit of using NE to evolve heterogeneous Artificial Neural Network (ANN)s. However, there is very little active research in this area. A further contribution of this thesis has been a rigorous empirical investigation demonstrating that the ability for many NE methods to create heterogeneous ANNs represents a significant advantage. This can now also be cited as an advantage of NE methods.

One of the benefits NE has over gradient based method is that it does not require that the transfer functions used are differentiable. In fact NE is completely agnostic to the transfer functions used. This means a much wider range of transfer function can be employed. Interestingly, the earliest used transfer function, the Heaviside step function, is one such function which cannot be differentiated; or at least into a form which can be used by gradient descent methods. However, results presented in this thesis demonstrate that the Heaviside step function is a highly effective transfer function for many tasks; provided the training method is *compatible*. Therefore, the fact that NE methods are not conditional on the transfer functions used represents a significant advantage. This advantage goes beyond simply being able to use a wider range of transfer functions. For instance, many NE algorithms could be used to train networks of transfer functions which model the spiking behaviour of real biological neurons, with little or no change to the underlying algorithm [65, 96, 219, 222].

11.2.10 NeuroEvolution and Connection Weights

A high level view of the thesis indicates that NE may be extremely effective for adapting network topology but ineffective for adapting connection weights. Evidence of this is seen in a number of areas. For instance, it was demonstrated that the adaptation of connection weights is far less significant to CGPANN training than the adaptation of topology. Although it has been argued that this provides evidence that topology is more significant to ANN training than connection weights, it may also be the case that NE is simply not *effective* at adapting connection weights. If NE were more effective at adapting topology than connection weight, than this would be an alternative explanation as to why topology was seen to be more significant than connection weights. Additionally, it has been discussed that NE is often combined with gradient based methods; where Evolutionary Algorithm (EA)s are used to adjust topology and back propagation to adjust the connections weights. For instance, NEAT's performance was seen to improve when combined with back prorogation in the domain of classification. Finally, there was a large discrepancy between CGPANN's performance when applied to classification and its performance when applied series forecasting. As the forecasting methods are likely to greatly benefit from recurrence, in order to maintain internal state information and to match the frequency of the changing output, it appears likely that topology is highly significant. However, topology may be less significant in the domain of classification, where no recurrence is needed or specific topology is needed. Therefore, if NE was highly suited to topology adaptation, and topology is significant in the domain of series forecasting, it would explains RCGPANNs strong performance. Additionally, if classification tasks are more dependent on connection weights, and NE is ineffective at configuring connection weights, than this would explain the poor CGPANN results seen.

Therefore, although much further work would be needed to confirm this, it appears that NE may be ineffective at connection weight adjustment. Or rather, its benefit lies in its ability to adapt topology. Therefore, more research is required into how to best apply NE to the adjustment of connection weights. Although the combination with back propagation appears to be the logical choice, this comes with a number of disadvantages. For instance, if NE and back propagation are combined then it means NE can no longer be applied to RANNs, or be applied to reinforcement tasks, or use arbitrary transfer functions. Therefore the simple combination of NE and back propagation may not be suitable as it removes many of the advantageous properties of NE.

11.2.11 Benchmarking

This thesis has raised a number of concerns surrounding benchmarking in the field of NE and the wider field of Machine Learning (ML).

In the field of NE there is very little rigorous comparative study of algorithms. This means it is extremely challenging, if not impossible, to assess which aspects of differing NE method are advantageous and which are not. For instance, the only widely adopted benchmarks are the single and double pole balancing. One of which, the single pole variant, is so trivial and unstandardised that the comparative results are of almost no use. This leaves one benchmark, double pole balancing, which has been widely used to assess the relative performance of NE methods. This is clearly insufficient. Additionally, statistical significance testing is rarely used; only arithmetic means and mediums. Therefore, the field of NE is in great need of good empirical comparative analysis, and, like GP, "needs better benchmarks" [197].

The issue of comparing classification methods has also been raised. Currently in the field of ML, classification methods are compared solely on their testing set performance. Although testing set performance is an important measure of comparison, as it is representative of how methods will be used in real world applications, it fails to assess the computational cost in reaching a given performance. For instance, if method A achieved an accuracy of 75.998 percent after training for a few seconds on a personal computer, and method B achieved an accuracy of 75.999 percent after training for a few days on a cluster of computers, is it the case that method B is better than method A? Clearly there are differing measures of "better" and this is not currently considered in most classification comparisons.

It is also interesting to note that in the field of EAs, the notion of comparing final fitness given a certain computational budget has long been considered; via the use of evaluations. The issue comes however in determining a suitable measure of computation expense which can be applied to arbitrary classification methods. This is an open question, but one which must be answered if ML methods of differing sub fields can be fairly compared.

Note that this is *not* implying that CGPANN would have performed more favourably in the domain of classification if computational expense was also considered, this is unknown, it is just that there may be important factors involved in comparisons other than classification accuracy.

11.3 Further Work

The work presented in this thesis has created a strong foundation demonstrating and evaluating the use of CGP as a NE method. It has also investigated many properties of NE generally. However, due to time constraints, there were a number of research opportunities which could not be completed. This section describes future work which would extend and complement the research presented in this thesis.

11.3.1 Controlling Length Bias

As has been discussed, CGP, and by extension CGPANN, exhibits a length bias to networks of a certain number of active nodes [82, 84]. With the number of nodes to which there is a bias being a function of the number of inputs, the number of available nodes, the arity of each node and the number of outputs.

Previous methods have been proposed to limit the effect of length bias [82, 84]. These methods include *reorder* which rearranges the nodes in a given chromosomes so as to more evenly distribute the active nodes whilst preserving semantics. The second method, *DAG*, allows nodes to connect to other nodes even if they have a higher index, provided that when the chromosomes are decoded, the phenotype does not contain cycles. Using such methods it was shown that CGP more evenly sampled the range of possible solutions when random chromosomes are created and when mutation operations take place, with DAG producing the most even distribution (although far from evenly distributed); see Figure 11.1.

As a side note, as RCGP is effectively equivalent to DAG, only without the requirement that no cycles can be created, it may also more evenly sample the space of possible solutions than regulator CGP. Although, as described in [82, 84], it is still far from an even distribution.

However, as has been previously discussed, that fact that CGP contains a length bias is not necessarily a disadvantage. This is because many task domains favour smaller, rather than larger, solutions. Additionally, length bias may be the mechanism opposing program bloat in CGP. Therefore, it is not certain whether the removal of length bias would be beneficial.

Additionally, the methods provided in [82, 84] for preventing length bias did not result in an even distribution of topologies, as can be seen in Figure 11.1, only a more even distribution. Therefore, it is not known if removing length bias is beneficial *and* it is clear

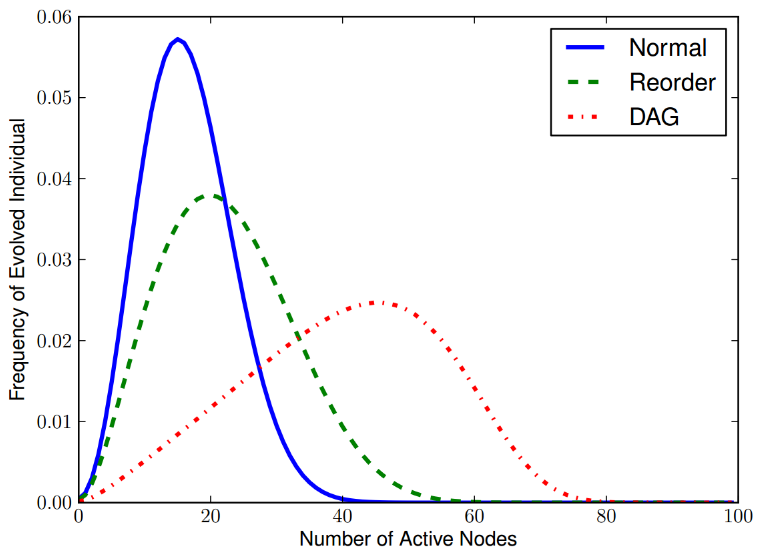


Figure 11.1: Depiction of length bias in CGP. Generated by applying CGP to a flat fitness landscape. Image taken from [82]

that the methods proposed for its removed only lessen and shift the length bias.

However, there may be an alternative method for preventing length bias in CGP. As previously discussed, the probability of a given node being active is a function of the number of nodes which can connect to that node. However, determining the exact probability is complex due to it being possible for nodes to be connected to by two or more other nodes. This effectively means that in order to calculate the probability of a given node been active, one must consider all of the possible topologies which can be created. Which although possible, is highly non-trivial[1].

Interestingly however, it would also be possible to determine an approximation of the probability that a given node is active by generating many random chromosomes; for a given number of inputs, available nodes, outputs and node arity. Using these approximate probabilities of each node being active, it would be possible to bias connection gene mutation to connecting to nodes which typically have a low probability of being active. Thus compensating for the bias of certain nodes being more likely to be active.

If this were shown to remove the length bias three interesting additional steps could be taken. Firstly, it could be assessed whether the total removal of length bias represents an advantage. Secondly, the challenging task of discovering equations which model the probability of each node being active could be embarked upon, knowing that it would be a beneficial if such a model were found. Thirdly, if it were possible to change the distribution

[1]The author tried, and failed, to solve the problem in the general case.

of active nodes to become flat, then it would also be possible to bias the search to program lengths of our choosing. For instance, a parameter could be added which controls the point to which there is a bias. Therefore the bias could be placed towards minimal solution if this was thought to represent an advantage, or completely removed if an unbiased search was desired.

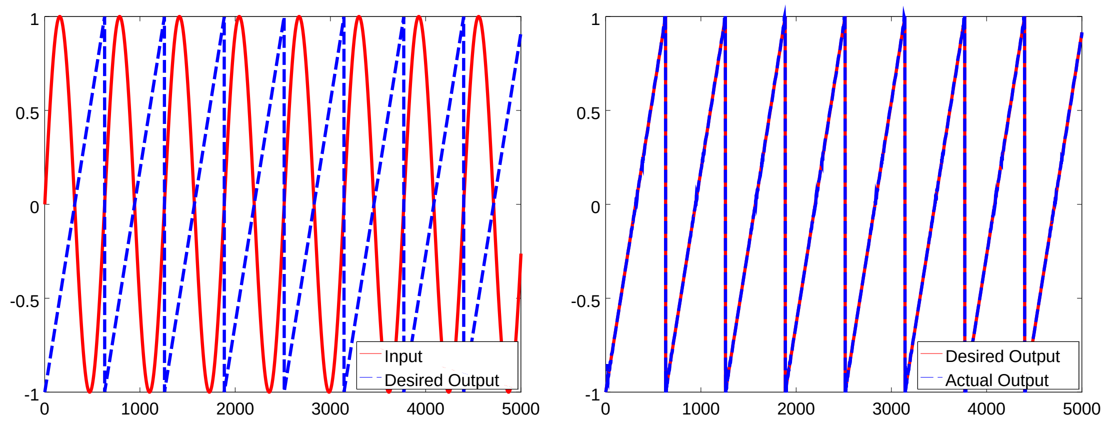
Finally, if it were not possible to mathematically model the probability of each node being active, then the empirical method would still be of benefit. For instance, before CGP begins an evolutionary search, it could always generate n random chromosomes to assess probability of each node being active and then use the results to combat the length bias during the search. Although this would represent a computation overhead, no fitness evaluations would be required; which is typically the most computational expensive component of an EA. Additionally, it may be the case that relatively low values of n are required to approximate the probability of each node being active sufficiently to have a marked advantage.

11.3.2 Reservoir Computing

A more recent development for RANNs has been the field of Reservoir Computing (RC) [182]; see Section 2.2.2.2. One of the main challenges in RC is determining a suitable initial reservoir. As many NE methods adapt the connection weights and topology of RANNs they represent a good candidate for solving the issue of determining suitable reservoirs.

As a preliminary study, RCGPANN is applied to the optimisation of the reservoir to be used by RC. RCGPANN is applied to evolving the reservoir by using a genotype with one output. From this single output, the active nodes in the RCGPANN chromosome are determined. The single output is then ignored. All of the active nodes, determined by the now ignored output, are then each connected to the actual output(s) of the RANN. Linear regression using least squares is then used to determine the connection weight of these new connections between the active nodes and the outputs(s). This, initially strange, method of determining which nodes to connect to the outputs is undertaken as it is typical in RC for each hidden node to contribute to each output. However, in RCGPANN, only one node contributes to each output. It would also have been possible to connect all of the available nodes to each output, but this would not have not enabled to the number of nodes in the reservoir to vary during evolution.

The parameters used by RCGPANN are as follows: 100 nodes, a node arity of 5,



(a) Depiction of the benchmark. (b) Output of resulting RANN.

Figure 11.2: Results from applying RCGPANN to the configuration of the reservoir to be used by reservoir computing.

a connection weight range of ±10, a mutation rate of 5% and a recurrent connection probability of 50%. The node function set contained only the logistic sigmoid function.

The standard RC benchmark of transforming a sine wave into a saw tooth was used to assess if RCGPANN was capable of optimising the reservoir used by RC. A depiction of this benchmark is given in Figure 11.2. The reservoir was then evolved for 10 generations using a (1+4)-ES. The fitness awarded to each chromosome was the sum of absolute errors between the target and generated output waveforms after the output weights had been determined using least squares. Using this method the target waveform is very closely matched, see Figure 11.2. Additionally, there was a 44% increase in fitness after only 10 generations (41 evaluations) demonstrating that RCGPANN can clearly improve upon a random choice of reservoir.

This very preliminary study demonstrates that RCGPANN has the potential to be a suitable method for optimising reservoirs to be used by RC. Similar to how other NE methods such as NEAT have also been shown to be useful in the context of determining good reservoirs [39].

A significant advantage of RC over many other methods is the speed at which the RANNs are trained. This is because each input in the training set only has to be applied *once*. This is then followed by solving a single matrix equation using standard methods. However, NE is a significantly slower training method for ANNs. Therefore, it could be argued that the described combination of NE and RC removes one of the advantageous properties of RC. However, as was shown here, RCGPANN improved the reservoir by 44% in only 41 evaluations, so the computational overhead may not be that high.

A further possible use of NE for RC would be to evolve RANNs which exhibited desirable reservoir properties; for instance being near the edge of chaos. It may be the case that evaluating the RANN with regard to a number of metrics may be quicker than applying all the input data and training the network using RC. This is increasingly likely to be the case as the number of samples in the training data scales. Although this would still be a generational approach, and therefore reasonably computationally expensive, it may serve to decrease the expense of optimising reservoirs using NE.

11.3.3 Spiking Neural Networks

This thesis has focused on the application of EAs to the training of non-spiking ANNs. However, for many years it has been considered that the computational power of spiking ANNs could surpass that of non-spiking given suitable training methods [184,290]. Despite efforts to apply traditional non-spiking training method to the training of spiking ANNs, such as back propagation [31], the promised ability of spiking ANNs has not yet been realised.

To date there have been a number of applications of EAs as training methods for spiking ANNs [65,96,222] including those which make use of the NEAT NE algorithm [219]. However, so far CGPANN has not been applied to the domain of spiking ANN.

The application of CGPANN to spiking ANN could have a number of advantageous properties. For instance, as for non-spiking ANNs, the topology and connection weights could both be simultaneously evolved. Additionally, heterogeneous networks of *differing* transfer functions could be combined; for instance a mixture of Hodgkin-Huxley [132], Izhikevich [131] and (leaky) integrate and fire neuron models.

There are also a number of choices which have to be made when applying spiking ANNs to a given task which are likely to influence the results. Such choices include how to encode and decode the inputs and outputs of the application into spike trains for the spiking ANN. Additionally, how often each neuron in the network should be updated i.e. the δt. When using NE these aspects of the spiking ANNs could also be encoded into the genotypes. Not only could this help in the application of spiking ANNs, but it could also give insight into how spiking ANNs should be configured generally.

11.3.4 Wider Study of NeuroEvolutionary Methods

This thesis has carried out a substantial evaluation of many benefits of NE including the ability to adapt program structure, create heterogeneous ANN and the effect of genetic redundancy. However, much of the work is limited to one or two NE methods; specifically CGPANN and Conventional NeuroEvolution (CNE).

Therefore, important future work would be to repeat the experiment presented here using a wider range of NE methods. To further the investigation into the benefits of evolving topology, a wider range of topology and non-topology evolving methods should be compared. To investigate further the benefits of creating heterogeneous ANNs, simply using a larger number of NE would be sufficient. Finally, to investigate further whether there is a benefit associated with genetic redundancy in NE, a wider range of NE methods which were identified as containing genetic redundancy, see Table 2.1, should be studied.

The extension of the work presented in this thesis to other NE methods would both help to confirm the results seen, and identify if they are general to the wider scope of NE methods.

11.3.5 Hyper-CGPANN

As has been previously discussed, a widely used extension to the NEAT NE method is Hypercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT) [264]. An advantageous property of HyperNEAT is that it can encode arbitrary large ANNs using a fixed sized genome. Additionally, the hyper extension can be a applied to a wide range of evolutionary methods; including standard tree-based GP [35].

It therefore appears that the same hyper extension would be directly compatible with CGP, RCGP, CGPANN and RCGPANN. An interesting study would therefore be to use these four methods to assess whether using neuron transfer functions or standard GP function produced a better hyper network. Additionally, whether the presence of recurrence in the program used to produce the final network is of any benefit.

11.3.6 Modular CGPANN

A significant extension developed by James Walker for the CGP algorithm was the ability to capture and re-use sub modules; Modular Cartesian Genetic Programming (MCGP) [292]. As has been discussed in Section 4.3.8, it would also be possible to apply this same modular extension to CGPANN in order to evolve ANNs comprised of interconnected, possibly repeated, sub networks.

This could result in truly modular ANNs of repeating structure throughout the network.

11.3.7 CGPANN and Back Propagation

As is discussed in Chapter 9, CGPANN performed poorly in the domain of classification. It was also noted that NEAT has also previously been shown to perform poorly in the domain of classification unless combined with back propagation in order to configure the connections weights [41]. Additionally, other NE methods, such as Evolutionary Programming Artificial Networks (EPNet), also rely on back propagation during training. It was also shown in Chapter 6 that NE benefits much more from evolving the topology of ANNs than evolving the connection weights. These separate insights could indicate that NE may be an effective method for manipulating ANN topology, but not connection weights.

Therefore, the combination of NE with back propagation appears to be a suitable method for training ANNs. Exploiting NE's ability to adapt topology and back propagations ability to configure connection weights. Future work would therefore be to combine CGPANN with a form of gradient descent and repeat the experiments presented in Chapter 9. This would identify if the use of gradient descent can be used to improve the performance of CGPANN for the task domain of classification.

Interestingly, NE may provide a benefit to back propagation other than optimising topology. As back propagation is a gradient based method, the solutions found are a strong function of the initial connection weights; the starting position in the search landscape. Therefore, by utilising NE to optimise both topology and connection weight, followed by the application of back propagation, it would allow evolution to find suitable topologies *and* starting connection weights for back propagation.

However, the application of gradient descent methods is not always suitable for NE. For instance, a common application of NE is reinforcement learning. Back propagation is not applicable to such tasks as it is only compatible with supervised learning. Therefore the use of back propagation with NE restricts the types of tasks to which NE can be applied; thus removing one of the advantages of NE.

Additionally, in its standard form, back propagation is not compatible with recurrent ANNs. Again one of the advantages of NE is that it can be used to train RANNs. Therefore the combination with back propagation places further restrictions on NE.

Back propagation also places restriction on the transfer functions which can be used by the ANNs. As has been shown in this theses, the ability to use arbitrary transfer function represents a benefit of NE. Again this benefit is removed if NE is combined with back propagation.

Finally, back propagation and EAs are both iterative stochastic processes which require the entire learning set to be observed many times during training. This means that back propagation and EAs are both individually slow algorithms. Therefore, a further disadvantage of combining the two approaches is that it would significantly increase the training time, especially if back propagation were applied to each member of the population, every generation. A possible solution to this could be to simply apply back propagation *once* to the final solutions found using NE; resulting in a possible compromise in computing speed and benefits of finding the nearest optima to the final solution after running NE.

11.3.8 Neutral Genetic Drift in NeuroEvolution

Despite providing a strong benefit for CGP, Chapter 8 presented the surprising result that the CGPANN benefits much less from the presence of NGD and redundancy in general. It was proposed that this discrepancy may be due to the presence of connection weights and their effect on the evolutionary search. The presence of connection weights allows mutation to have a much smaller effect on semantics than mutations to topology and transfer functions. This means that the presence of connection weights allows CGPANN to conduct local searches much more easily than un-weighted CGP.

It may at first appear counter intuitive that the ability to conduct a local search would decrease the likelihood of becoming stuck in an area of the search space. However, consider a truly random evolutionary search; the most global search. After n generations the best found solution is compared to the x random solutions of this generation. As the current best solution is the best of $n \times x$ random solutions it is more likely to be better than any of the x random solutions. This trend increased in likelihood as n is increased. Therefore, despite being a very global search, the random search can very easily become stuck in an area of the search space. However, if the ability to also conduct a local search around the current best solution is added, the chances of improving upon the current best solution increases. Therefore, the presence of connection weights in CGPANN may be increasing the ability to conduct more local searches. This in term could be decreasing the likelihood that CGPANN becomes trapped in a point in the search space. If CGPANN is less likely to become trapped in an area of the search space, then it is less likely to require the aid of NGD to escape. Resulting in NGD providing less of a benefit to the search, as was seen in the results.

Note the distinction between being suck at a point in the search space and being stuck in a local optimum. The search can become suck in an area of a search space even if it is not a local optimum; as was the case in the random search example.

Therefore, future work should investigate if it is the case that CGPANN does not benefit from NGD due to the ability to conduct a more local search. The remainder of this section describes such possible future work.

Firstly, connection weights could be added to regular CGP to create a weighted form of CGP. This weighted CGP would be similar to CGPANN except it would not use neuron transfer functions or the higher arities associated with ANNs. If it were found that the benefit of NGD was diminished when weighted CGP was used, than it would demonstrate that the reduced benefit of NGD seen in CGPANN is due to the presence of connection weights. However, if it were found that weighted CGP still strongly benefited from NGD, then it would demonstrate that it is not the influence of the connection weights which cause CGPANN not to benefit from NGD.

Additionally, if a weighted form of CGP were shown not to benefit from NGD, but also outperform standard CGP, than this would demonstrate that the benefit seen from NGD is compensating for poor fitness landscape navigation. However, if a weighted form of CGP was shown not to benefit from NGD, and standard CGP outperformed weighted CGP, then it would show that the strong benefit of NGD is not merely compensating for a weakness in CGP's search.

Secondly, the same experiments presented investigating NGD in CGPANN could be repeated without ever mutating connection weight genes. That is to say, the connection weights would be present, but not mutated; left as random initial values. If it were seen that CGPANN, without mutating connection weights, still did not benefit from NGD, then it would demonstrate that the ability to manipulate connection weights was not the cause of the diminished benefit of NGD for CGPANN. However, if it were seen that CGPANN, without mutating connection weights, did benefit from NGD, then this would identify that the connection weights were likely to be the cause of CGPANN not benefiting from NGD.

It was also speculated that the ability to mutate connection weights may be suited to a more local search, whereas mutations to topology may be more suited to a global search. This is because it appears intuitive that alterations to topology would influence the semantics more than alterations to a connection weight; at least for small alterations to connection weights. As it is important for a search to be both explorative on a global scale and exploitive on a local scale, both behaviours are desirable. It has been stated that CGP may benefit more from NGD because it lacks this local exploitive ability.

Therefore, it may be the case that by using separate mutation rates for topology genes and connection weight genes, it would be possible to vary the levels of exploration and exploitation. For instance, increasing the connection weight mutation rate relative to the topology mutation rate may result in a more local search. However, this behaviour may be reliant upon the connection weight mutation rate never being too high, otherwise it would be too random to conduct a local search. Additionally, it may be beneficial for the connections to be mutated within a given Gaussian distribution of their current value, again biasing to a more local search.

It may also be the case that the search would greatly benefit from the levels of topology and connection weight mutation rates varying as the search progresses. For instance, to initially conduct a more global search which becomes more local as time progresses; similar to simulated annealing [159]. Alternatively, varying the relative mutation rates of topology and connection genes based on fitnesses improvements. For instance to conduct a more global search when trapped in local optima.

11.4 Final Remarks

EAs are famous for being “the second best solver for any problem” [60]. This is due to their highly general nature. EAs only require that solutions are describable in a form which can be manipulated, and that each solution can be given a fitness score.

ANNs are an extremely powerful and flexible method for arbitrary data transformations. The challenge for ANNs is finding effective training methods for different problem domains. Although there are many highly optimised training methods for applying feed-forward ANNs to supervised learning tasks, effective training methods for recurrent topologies and other domains need further research.

In the authors opinion, this is the great benefit of NE. By harnessing the generality of EAs, NE is an effective training method for configuring recurrent topologies and applying ANNs to domains other than supervised learning. In fact NE is so general that the same methods can be used to create spiking as well as non-spiking networks with little to no change to the underlying algorithms. This generality is the power of NE.

However, significantly more research is required for NE, including rigorous empirical comparisons between methods and rigorous empirical study of their abilities. Without this it is not known which aspect of which NE methods are beneficial, and which are not. This thesis has made significant progress into this area, but much more further work is needed.

Finally, there needs to be a change in the committee's direction, from mainly applications driven research to more analysis and comparison of the underlying algorithms. Although applications are important, and can be used to showcase the ability of NE, it is difficult to use their results to further the field. Related to this topic is the need for the committee to adopt better empirical research methodologies, including the use of a wider range of standardised benchmarks, and rigorous significance testing. Without this NE many never progress into the wider ML literature.

Appendix A

Benchmarks

This appendix describes a number of benchmark problems which are used throughout the thesis.

A.1 Structure of this Appendix

This appendix separates each domain of benchmark into individual sections. Section A.2 describes reinforcement learning control tasks. Section A.3 describes supervised learning classification tasks. Section A.4 describes supervised learning Boolean logic circuit synthesis tasks. Section A.5 describes supervised learning symbolic regression tasks. Finally, Section A.6 describes series forecasting tasks.

A.2 Control

One of the advantages of NeuroEvolution (NE) is that it can be applied to reinforcement learning type tasks. These are tasks where there is no list of desirable input-output pairs, only a single figure of merit of the overall performance of a solution.

As standard training methods for Artificial Neural Network (ANN)s, such as back propagation, cannot be applied to reinforcement learning tasks, only supervised learning, control benchmarks serve to showcase one of the advantages of NE methods. That is to say, they can apply ANNs to a whole new domain of tasks.

A.2.1 Pole Balancing

The pole balancing benchmarks have been used for many years in the Artificial Intelligence (AI) literature and represent a range of difficult multiple-input single-output control tasks. The task is to balance one or more poles on a moveable cart by applying a horizontal force.

The most widely used pole balancing benchmarks are single pole balancing and double pole balancing; Figures A.1 and A.2 respectively. The equations which describe the dynamics of the pole-cart system are given in Equations A.2.1-A.2.4 with the symbol definitions and commonly used constants given in Table A.1.[1]

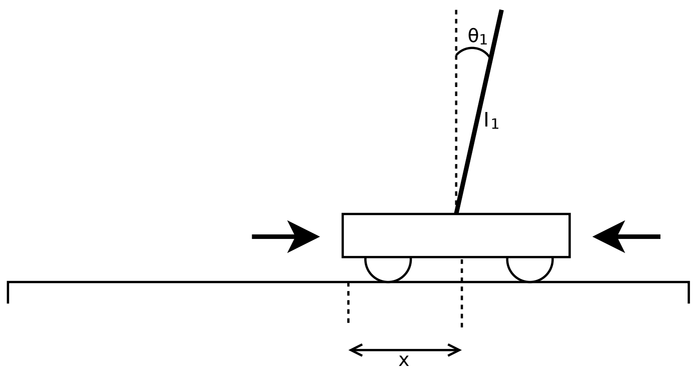


Figure A.1: Depiction of the single pole balancing benchmark.

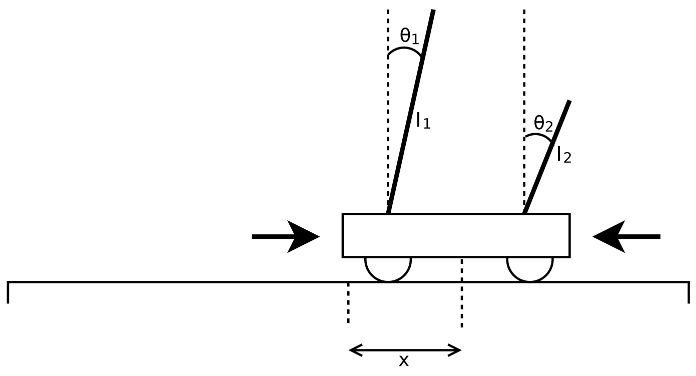


Figure A.2: Depiction of the double pole balancing benchmark.

[1]As an aside, these equations are not particularly accurate at describing the dynamics of the cart-pole system; due to the friction between the cart and the track being applied as a binary force opposing the motion of travel, rather than proportionally to the speed of travel. However, as these equations are the benchmark they must be used as stated. Interestingly, more accurate equations of motion were used during the early years of the benchmark such as in [13, 24], but for some reason later researchers changed them.

$$\ddot{x} = \frac{F - \mu_c sgn(\dot{x}) + \sum_{i=1}^{N} \tilde{F}_i}{M + \sum_{i=1}^{N} \tilde{m}_i} \tag{A.2.1}$$

$$\ddot{\theta}_i = -\frac{3}{4l_i}\left(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_{pi}\dot{\theta}_i}{m_i l_i}\right) \tag{A.2.2}$$

$$\tilde{F}_i = m_i l_i \dot{\theta}_i^2 \sin\theta_i + \frac{3}{4} m_i \cos\theta_i \left(\frac{\mu_{pi}\dot{\theta}_i}{m_i l_i} + g\sin\theta_i\right) \tag{A.2.3}$$

$$\tilde{m}_i = m_i\left(1 - \frac{3}{4}\cos^2\theta_i\right) \tag{A.2.4}$$

Table A.1: Pole balancing symbol definitions and commonly used values.

Symbol	Description	Value
x	Cart position	$[-2.4, 2.4]m$
θ_i	ith pole angle	$[-n, n]deg$
F	Force applied to cart	$[-10, 10]N$
$\tilde{F}_i$	Force on cart due to ith pole	
l_i	Half-length of ith pole	$l_1 = 0.5m$
		$l_2 = 0.05m$
M	Cart mass	$1.0kg$
m_i	Mass of ith pole	$m_1 = 0.1kg$
		$m_2 = 0.01kg$
μ_c	Friction coefficient between cart and track	0.0005
μ_{pi}	Friction coefficient between ith pole and cart	0.000002

The initial position of the cart is typically in the centre of the track; for both single and double pole cases. In the single pole case, the initial angle of the pole changes between implementations; this is bad practice and weakens any comparisons. Common values are $0°$ and $4°$. For the double pole case, the initial position is more standardised with the longer and shorter poles starting at $1°$ and $0°$ from vertical respectively.

The pole-cart system models are then simulated using Euler integration with a time step of $0.01sec$ and the controller outputs are updated every $0.02sec$. The simulations are then run for 100000 time steps. The inputs to the created controller are the position and velocity of the cart and the angle and angular velocity of the pole(s); making four inputs for the single pole case and six for the double. These inputs are scaled linearly into the control systems input range by assuming the following maximum ranges. Single pole: cart position $[-2.4, 2.4]m$, cart velocity $[-1.5, 1.5]m/s$, pole angle $[-12, 12]deg$ and pole angular velocity $[-60, 60]deg/s$. Double pole: cart position $[-2.4, 2.4]m$, cart velocity $[-1.5, 1.5]m/s$, pole angles $[-36, 36]deg$ and pole angular velocities $[-115, 115]deg/s$. Although these values can change between implementations found in the literature. The output of the controller is also scaled to the range of the allowed force on the cart. This is typically achieved using one of two methods. The Bang-Bang control system, sometimes used on the single pole benchmark, applies a force of -10N or $+10$N if the controller output is $< n$ or $\geq n$ respectively; where n is a threshold value. The continuous control system simply linearly maps the control systems output range to the required $[-10, 10]$N range; allowing for a continuous range of possible applied forces. Additionally the continuous control system restricts the magnitude of the applied force to always be greater than $\frac{1}{256} \times 10$N; thus increasing the difficulty of the task.

The simulation is terminated if the cart leaves the bounds of the track or the angle of the pole(s) exceeds the maximum angle from vertical. For the single pole benchmark the most commonly used angle the pole can deviate from vertical before the experiment is terminated is $12°$. For the double pole case it is $36°$.

The fitness of each proposed solution is set as the number of time steps the pole(s) remain within the set range and the cart remains within the track bounds. Some implementations also penalise rapid swinging [89].

A popular extension to the pole balancing benchmark is the removal of cart and pole(s) velocity information, making the task non-Markovian [89]. Without the velocity information being explicitly available to the controller, it must be estimated internally.

A.2.2 Ball Throwing

The recently presented ball throwing benchmark [164] represents a Multiple-Input Multiple-Output (MIMO) control task. The task is to design a controller which throws a ball as far as possible; see Figure A.3.

The dynamics of the arm are described by Equations A.2.6-A.2.9; with the symbol definitions given in Table A.2. The control system has two inputs, θ, the arm angle from vertical and ω, the angular velocity of the arm. It has two outputs, T, the applied torque to the arm and an output which dictates when to release the ball. The inputs to the controller are linearly scaled from $\pm\pi/2$ and ± 5 rad/s for θ and ω respectively; the maximum input range of the controller. The maximum range for ω, ± 5 rad/s , was found from simulation; actual maximum value: ± 4.4915 rad/s . The first output of the controller sets the torque applied to the arm and is linearly mapped to $[-5, 5]$N. The ball

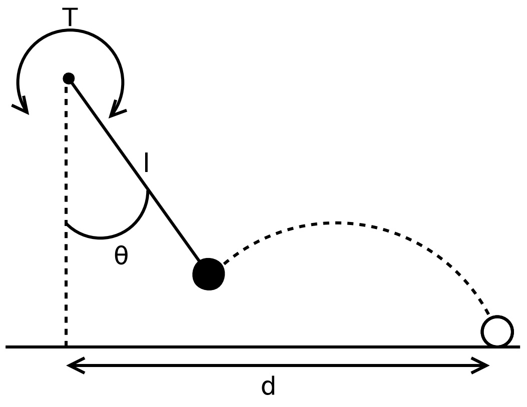


Figure A.3: Depiction of the ball throwing benchmark.

Table A.2: Ball throwing symbol definitions and commonly used constants.

Symbol	Description	Value
θ	The arm angle	$[-\frac{\pi}{2}, \frac{\pi}{2}]rad$
ω	The arms angular Velocity	
c	Friction constant	$2.5s^{-1}$
l	Arm length	$2m$
g	Gravity	$9.81ms^{-2}$
m	Ball mass	$0.1kg$
T	Torque applied to arm	$[-5, 5]Nm$

is thrown if the second output exceeds a threshold; for example, zero if the output range is ±1, or 0.5 if the output range is $[0, 1]$. The system is initialised with the ball held, the arm hanging vertically downwards and the angular velocity set to 0 rad/s.

$$\dot{\theta} = \omega \tag{A.2.5}$$

$$\dot{\omega} = -c \cdot w + \frac{g \cdot \sin(\theta)}{l} + \frac{T}{m \cdot l^2} \tag{A.2.6}$$

$$\omega = 0 \text{ if } |\theta| \geq \pi/2 \tag{A.2.7}$$

$$\theta = \pi/2 \text{ if } |\theta| \geq \pi/2 \tag{A.2.8}$$

$$\theta = -\pi/2 \text{ if } |\theta| \leq -\pi/2 \tag{A.2.9}$$

The ball throwing model is simulated using Euler integration with a time step of $0.01s$ with the outputs of the controller also updated every $0.01s$. When the arm is holding the ball the dynamics are described using Equations A.2.6 - A.2.9. Additionally if θ exceeds $\pi/2$ or $-\pi/2$ it is set as $\pi/2$ or $-\pi/2$ respectively and ω is set to zero; as is specified in Equations A.2.7 - A.2.9. When the ball is released it obeys regular Newtonian mechanics, assuming no air resistance, until the ground is reached. The distance the ball is thrown is then used to assign a fitness value; see Figure A.3. The maximum possible distance the ball can be thrown is $10.202m$ but the task is considered complete when the ball is thrown $9.50m$ or further.

A.3 Classification

Classification is one of the main applications of ANNs and is typically an instance of supervised learning.

A.3.1 Breast Cancer

The Breast Cancer dataset was originally constructed at the University of Wisconsin Hospital [189]. Each sample in the dataset described nine values recorded by a surgeon using fine needle aspiration of a tumour located in the breast of patients. Each sample has attributes describing the levels of: clump thickness, uniformity of cell size, uniformity of cell shape, marginal adhesion, single epithelial cell size, bare nuclei, bland chromatin, normal nucleoli and mitoses. Each sample is then classified as describing a benign or malignant cancer cell. The dataset contains 683 samples (after removing sixteen samples with missing data) with 65.0% representing benign tumours and 35.0% representing malignant.

The Breast Cancer dataset is available from the UCI repository [215] and is also provided in the mlbench Package [51] for the R programming language [234]. On the UCI repository two datasets are listed for the Breast Cancer Wisconsin Data Set. Here the ‘original’ version of the data set is used with samples containing missing data removed. The ‘diagnostic’ version was *not* used.

A.3.2 DNA

The DNA dataset was originally donated by G. Towell, M. Noordewier, and J. Shavlik. The dataset describes primate splice-junction gene sequences and the task is to classify each sample as containing a boundary between exons, a boundary between introns or neither. Boundaries between exons are retained after splicing, however the boundary between introns are not.

The dataset contains 3186 samples (splice junctions) with each sample containing 180 binary attributes (genes) and three classes; whether the sample contains a boundary between exons (24.1%), a boundary between introns (24.0%) or neither (51.9%).

The DNA dataset is available from the mlbench Package [51] for the R programming language [234].

A.3.3 Glass

The Glass dataset was originally created by B. German at the Central Research Establishment, Home Office Forensic Science Service, Reading. The dataset describes the chemical analysis of seven different types of glass. The motivation was the classification of glass for criminological investigations.

The Glass dataset contains 214 samples each with 9 attributes and the type of glass it describes. Each sample describes the following properties of the glass: refractive index, Sodium level, Magnesium level, Aluminum level, Silicon level, Potassium level, Calcium level, Barium level and Iron level. There are six classes of glass: building windows float processed (32.7%), building windows non float processed (35.5%), vehicle windows float processed (7.9%), containers (6.1%), tableware (4.2%) and headlamps (13.6%).

The Glass dataset is available from the mlbench Package [51] for the R programming language [234].

A.3.4 Ionosphere

The Ionosphere was originally created by the Space Physics Group, Applied Physics Laboratory, Johns Hopkins University. The dataset describes the high frequency, high power radar recordings of the ionosphere. The subject of interest was free electrons in the ionosphere with each recording specifying whether some type of structure was detected.

The Ionosphere dataset contains 351 samples each with 34 attributes describing the class and whether or not the sample detected structure in the ionosphere. The dataset describes 35.9% of samples detecting structure and 64.1% of samples not.

The Ionosphere dataset is available from the UCI repository [215] and is also provided in the mlbench Package [51] for the R programming language [234].

A.3.5 Iris

The Iris dataset was originally collected by E. Anderson [14] and made available by A. Fisher [62]. The dataset describes various measurements recorded from a number of iris species. The task is to classify the species of iris based on the recorded measurements.

The Iris dataset contains 150 samples each describing the sepal length, sepal width, petal length, petal width and the species to which the iris belongs: setosa (33.3%), versicolor (33.3%), and virginica (33.3%).

The Iris dataset is available from the UCI repository [215] and is also provided in the core datasets Package for the R programming language [234]. Although as has been previously noted in the literature, many version of the Iris dataset exists which has led to confusion and unfair comparative experiments [28].

A.3.6 Olives

The Olives dataset was originally collected and presented by E. Stefanoudaki [268]. The dataset describes the percentages of fatty acids found in olives from different regions of Italy. The task is to classify to which region a given sample belongs.

The Olives dataset contains 572 samples each decried by 8 percentage levels of fatty acids: palmitic, palmitoleic, stearic, oleic, linoleic, linolenic, arachidic and eicosenoic. Each sample then belongs to one of 9 classes describing from which region of Italy the olive came: Calabria (9.8%), Coast-Sardinia (5.8%), East-Liguria (8.7%), Inland-Sardinia (11.4%), North-Apulia (4.4%), Sicily (6.3%), South-Apulia (36.0%), Umbria (8.9%) and West-Liguria (8.7%).

The Olives dataset is available from the classifly Package [303] for the R programming language [234].

A.3.7 Segmentation

The Segmentation dataset was originally collected and made available by Andrew Hill [107]. The dataset describes imaging measurements of cell bodies and the task is to determine how well the cell body is segmented.

The Segmentation dataset contains 2019 samples each containing 119 imaging measurements and whether the described cell body is poorly segmented (64.4%) or well segmented (35.6%).

The Segmentation dataset is available from the Caret Package [70] for the R programming language [234].

A.3.8 PROBEN1 - A Set of Neural Network Benchmark Problems and Benchmarking Rules

The Proben1 report [229] outlines a number of, mainly real world, classification (discrete) and approximation (continuous) benchmarks for ANNs. The report also describes best practices and how to present results. The report contains ten classification benchmarks (Cancer, Card, Diabetes, Gene, Glass, Heart, Horse, Mushroom, Soy Bean and Thyroid) and three approximation benchmarks (Building, Flair, Hearta).

Each of the datasets is provided in three permutations so as to allow cross validation. The three permutations are indexed with a value $[1, 2, 3]$; for instance Cancer1, Cancer2 and Cancer3. The data sets are also separated into two groups, training and testing. The ANN is to be trained using the training set and tested on the testing set; the testing set should never be used during training. The user may also subdivide the training set into a training set and a validation set. In order to ensure that the same samples are used for training, validation and testing, three percentages are stated with each classification task. The first percentage represent the size of the training set, the second the size of the validation set and the last the testing. The training set then comprised of the first $x\%$ of the dataset, the validation the following $y\%$ and the testing the final $z\%$.

It is also recommended that the squared error percentage be used for the fitness function, Equation A.3.10. Where o_{min} and o_{max} are the minimum and maximum output values form the ANN, N is the number of outputs from the ANN, P is the number of training examples, o_{pi} are the actual output values from the ANN and t_{pi} are the target outputs.

$$E = 100 \cdot \frac{o_{max} - o_{min}}{N \cdot P} \sum_{p=1}^{P} \sum_{i=1}^{N} (o_{pi} - t_{pi})^2 \tag{A.3.10}$$

By defining the datasets, the fitness function, the permutations and the sizes of the training and testing sets the results of many training algorithms can be fairly compared. This was the goal of the Proben1 report. Specific data sets used in the proben1 document can be accessed via the FTP address `ftp.ira.uka.dein/pub/neuron/proben1.tar.gz`.

A.3.9 The Monks Problems

The Monks Problems [273] are a set of classification benchmarks intended for comparing learning algorithms. The classification tasks are based on the appearance of robots which are described by six attributes each with a range of values; see Table A.3. There are three classification tasks described in [273]:

1. head_shape = body_shape OR jacket_color = red
2. exactly two of the six attributes have their first value
3. (jacket_color = green AND holding = sword) OR (jacket_color != blue AND body_shape != octagon)

Where if the condition is met the robot belongs to the given class else it does not. The classification tasks are called Monks Problem 1, 2 and 3 in the order given above. Problem 1 uses 124 randomly selected robots, from the possible 432, to be used as the training set. Problem 2 uses 169 randomly selected robots to be used as the training set. Problem 3 uses 122 randomly selected robots to be used as the training set with additional noise causing 5% to be misclassified.

Table A.3: Monks Problem Robot Appearances.

Description	Attributes
head_shape	round, square, octagon
body_shape	round, square, octagon
is_smiling	yes, no
holding	sword, balloon, flag
jacket_color	red, yellow green, blue
has_tie	yes, no

The implementation commonly used by ANNs is to assign each attribute value its own input to the network; totalling seventeen inputs. Each of these inputs is set as one if the particular attributes value is present and as zero otherwise. The ANN then classifies each sample as belonging to the class if its single output is greater than a threshold. This threshold is commonly set as 0.5 for transfer functions with outputs in the range $[0, 1]$ and 0 for transfer functions with outputs in the range $[-1, 1]$.

The fitness commonly used is the percentage of misclassified robots; with zero representing the best solution.

A.3.10 Two Spirals

The two spirals classification benchmark was created in the 1980s and was originally posted on a connectionist mailing list by Alexis Wieland [38]. The task is considered highly challenging for ANNs [40].

The benchmark consists of 194 data points describing samples taken from two spirals in Cartesian space; see Figure A.4. The task is to classify to which spiral each sample belongs using only the (x, y) Cartesian coordinates. The fitness awarded is the number of misclassification made; meaning a target fitness of zero.

The ANNs applied to the task comprises of two inputs, for the (x, y) Cartesian coordinates of each sample, and one output. The two inputs are linearly scaled into a $[0, 1]$ range by assuming the minimum and maximum values are ± 6.5 and ± 6 for the x and y axis respectively. When the output value is < 0.5 it is interpreted as one spiral and ≥ 0.5 as the other.

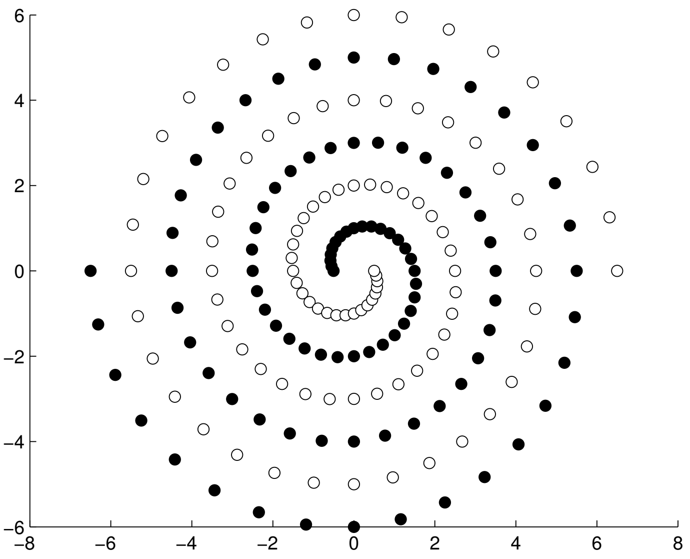


Figure A.4: Depiction of the Two Spiral Classification benchmark.

A.4 Boolean Circuits

Boolean circuit synthesis is the task of implementing a given truth table as a digital circuit. The benchmarks are often coupled with additional constrains such as using only certain logic gates, using a minimal number of gates or executing in a given time constraint.

Common Boolean benchmarks include n bit even parity, n bit full adder and n bit multiplier. These tasks all have the property that the difficulty can be scaled by varying

n; where higher n represents a more challenging task. Additionally, the parity tasks represent tasks with a single output, whereas full adder and multiplier represent MIMO benchmarks.

The benchmarks can be framed as fitness minimising or fitness maximising tasks depending upon whether the number of incorrect or correct outputs are recorded. In the maximising case each correct output for each row in the truth table increments the fitness. Therefore, for a given task the maximum fitness is $o2^i$; where o is the number of outputs and i is the number of inputs. Therefore if a truth table was described by eight inputs and four outputs, the maximum fitness would be $4 \times 2^8 = 1048$.

A.5 Symbolic Regression

Symbolic regression is the task of fitting a symbolic equation to a given set of data points. These data points may be recordings of a real world event, such as experimental data, or samples from an already known equation. The task is to minimise an error between the given points and the values predicted by the symbolic equation at those points. Common fitness measures used are: the Sum of Absolute Errors (SAE), Mean Absolute Error (MAE) or the Mean Square Error (MSE). These fitness measures are given respectively in Equations A.5.11-A.5.13 where; N is the number of samples, a_i is the actual value at point i and p_i is the predicted value at point i.

$$\sum_{i=1}^{N} |a_i - p_i| \tag{A.5.11}$$

$$\frac{1}{N}\sum_{i=1}^{N} |a_i - p_i| \tag{A.5.12}$$

$$\frac{1}{N}\sum_{i=1}^{N} (a_i - p_i)^2 \tag{A.5.13}$$

A number of symbolic regression benchmark tasks are now described.

A.5.1 Nguyen 10

The Nguyen 10 benchmark is a symbolic regression task defined and used in [281]; given in Equation A.5.14 and plotted in Figure A.5. The task uses 100 random samples taken from the range x_1 in [-1,1] and x_2 in [-1,1]. The function set used contains: $+, -, \times, \div, \sin, \cos, \exp, \log$.

The fitness function aims to minimise the SAE between the correct value at a given x_1 and x_2 and the value produced by the symbolic equation.

$$f(x_1, x_2) = 2\sin(x_1)\cos(x_2) \tag{A.5.14}$$

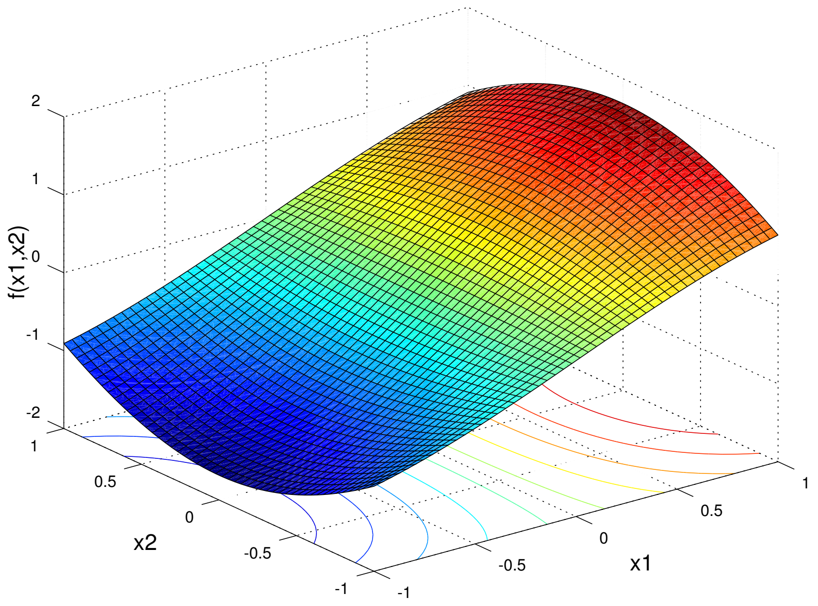


Figure A.5: Nguyen 10

A.5.2 Pagie

The Pagie benchmark is a challenging [197] symbolic regression task defined and used in [220]; given in Equation A.5.15 and plotted in Figure A.6. The task uses 676 samples evenly taken from the range x_1 in [-5,5] and x_2 in [-5,5]. The function set contains: $+, -, \times, \div$. The fitness function is defined to be the SAE between the correct value at a given x_1 and x_2 and the value produced by the symbolic equation. Thus the objective is to minimise the fitness (zero being the perfect score).

$$f(x_1, x_2) = \frac{1}{1 + x_1^{-4}} + \frac{1}{1 + x_2^{-4}} \tag{A.5.15}$$

A.5.3 Tower Problem

The Tower Problem [162] is a *very* challenging real world symbolic regression task; plotted in Figure A.7. The data was provided by Arthur Kordon and each sample describes 25 measurements comprising of temperatures, flows, and pressures over time in a distillation

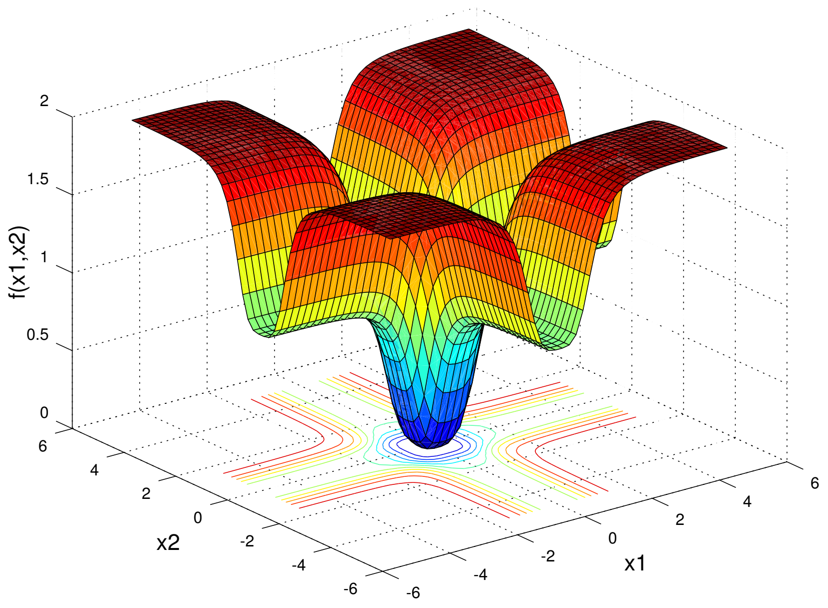


Figure A.6: Pagie

tower. The task is to create a symbolic equation which models the propylene concentration in the distillation tower based on these inputs.

The dataset contains 4999 samples (each containing 25 separate measurements and the propylene concentration) taken 15 minutes apart from within the distillation tower. The fitness awarded is the SAE of the output after all inputs have been applied. The function set used contains: $+, -, \times, \div, \sin, \cos, \exp, \log$.

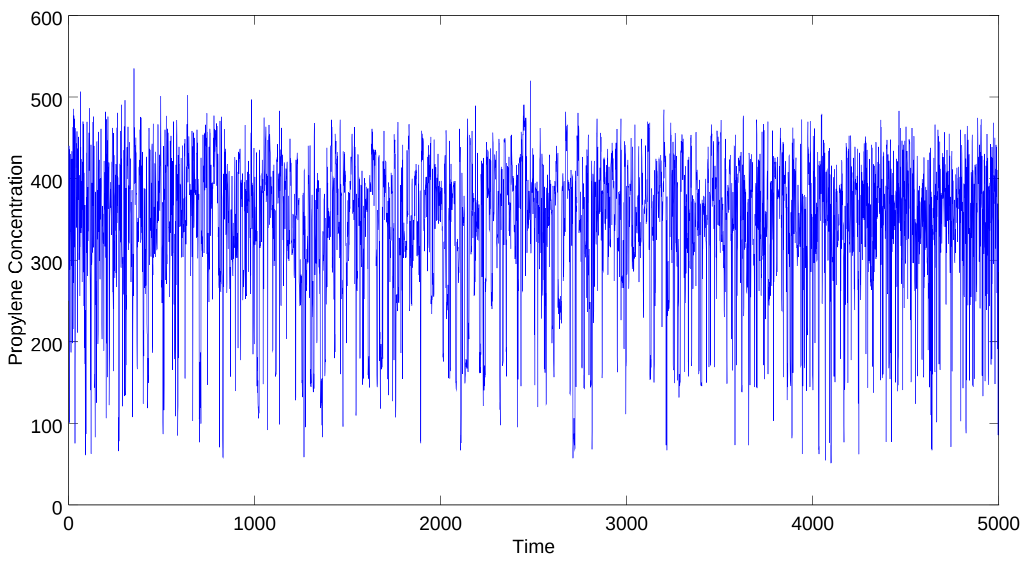


Figure A.7: Tower Problem

A.6 Forecasting

Forecasting is another common application of ANNs. The task is to predict future values in a series from a set of previous recording. A common application is the prediction of stock market values or currency exchange rates.

All of the forecasting benchmarks presented here consist of a training set of 1000 data points and a testing set of 100 data points. In each case the embedding dimension and time delay are given; found using the pdc package [34] for the R programming language [234].

A.6.1 Laser

The Laser benchmark is the recording of a "*81.5-micron 14NH3 cw (FIR) laser, pumped optically by the P(13) line of an N2O laser via the vibrational aQ(8,7) NH3 transition*" [121]. The benchmark was used in the Santa Fe Competition [297] and the dataset is available at [298].

Two versions of the dataset exist, one containing one thousand samples and another extended version containing ten thousand. The one thousand sample version was used by the Santa Fe Competition and the extended version is made available for further testing of methods. Here the first 1000 samples of the extended version are used as a training set and the following 100 samples are used as the testing set. This series is also normalised into a [0,1] range using Equation A.6.16 where: x_i is the sample to be normalised, x'_i is the normalised sample, X is the entire series and the *min* and *max* functions return the minimum and maximum sample value in the series X respectively. The Laser benchmark is plotted in Figure A.8.

The embedding dimension and time delay used for the Laser series are $D = 4$ and $T = 7$ respectively.

$$x'_i = \frac{x_i - min(X)}{max(X) - min(X)} \tag{A.6.16}$$

A.6.2 Mackey-Glass

The Mackey-Glass equation was originally used to model blood cell regulation [186]. However, the Mackey-Glass equation has also been used as a forecasting benchmark due to its interesting chaotic properties. The Mackey-Glass equation is given in Equation A.6.17. By adjusting the value of the delay parameter τ the equation produces chaotic and non-chaotic

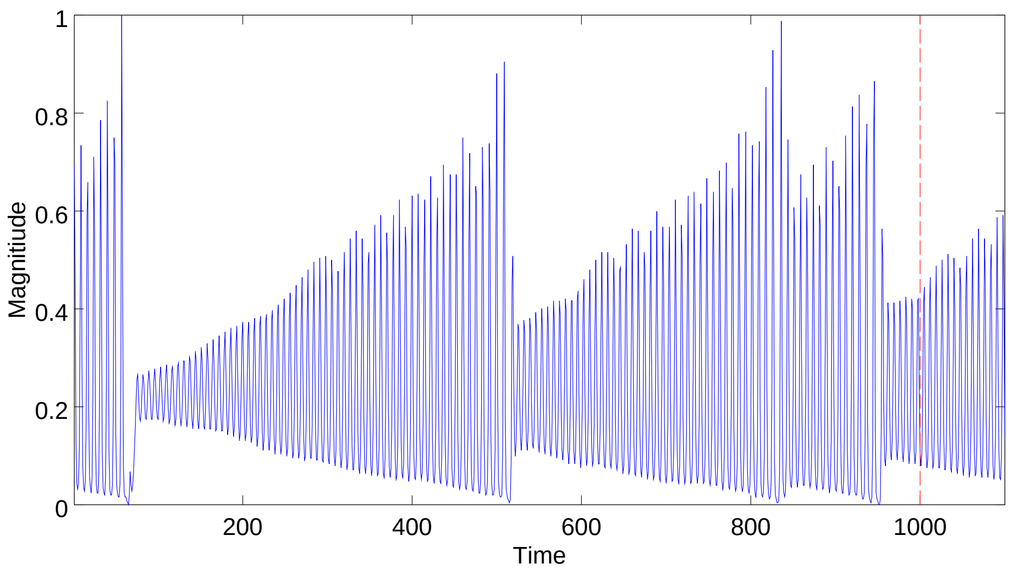


Figure A.8: Laser series forecasting benchmarks.

series; $\tau > 16.8$ produces chaotic behaviour.

$$\frac{dx(t)}{dt} = \frac{a \cdot x(t-\tau)}{1 + x^c(t-\tau)} - b \cdot x(t) \tag{A.6.17}$$

Here the Mackey-Glass equation parameters are set as $a = 0.2$, $b = 0.1$ and $c = 10$. The delay parameter τ is set as 17 and $x(t) = 0$ when $t \leq 0$. A series is produced using the 4th order Runge-Kutta integration method with a time step of $dt = 0.01$ seconds. This series is then sampled once a second to produce the series used as the benchmark. This series is also normalised using Equation A.6.16. The first 117 seconds (samples) are removed to avoid the transient response time. Then the following 1100 seconds (samples) are used for the training and testing sets; plotted in Figure A.9. The first 1000 seconds are used for training and the following 100 are used for testing.

The embedding dimension and time delay used for the Mackey-Glass series are $D = 4$ and $T = 1$ respectively.

A.6.3 Sunspots

Predicting the number of yearly/monthly Sunspots [253] is a commonly used [157], challenging [4], series prediction benchmark. The data is recorded by the SIDC-team, at the World Data Center for the Sunspot Index, Royal Observatory of Belgium [253] and is available from [244]. Here the smoothed number of monthly sunspots is used as the series. 1100 months (samples) of data are taken from November 1834 to June 1926. The first

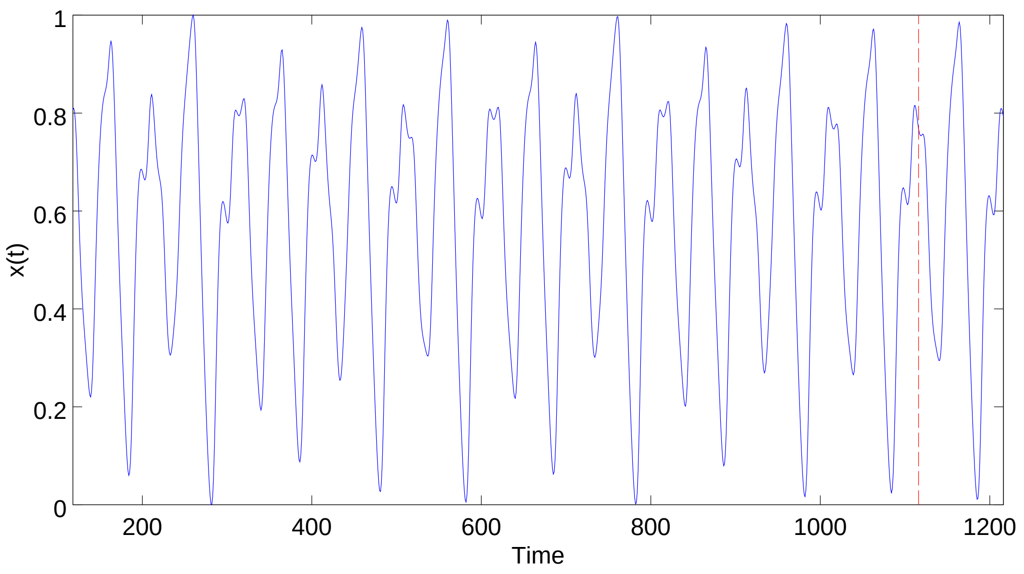


Figure A.9: Mackey-Glass series forecasting benchmarks.

1000 samples are used for training with the remaining 100 used for testing. The series is once again normalised using Equation A.6.16. The series is plotted in Figure A.10.

The embedding dimension and time delay used for the smoothed monthly sunspots series are $D = 5$ and $T = 1$ respectively.

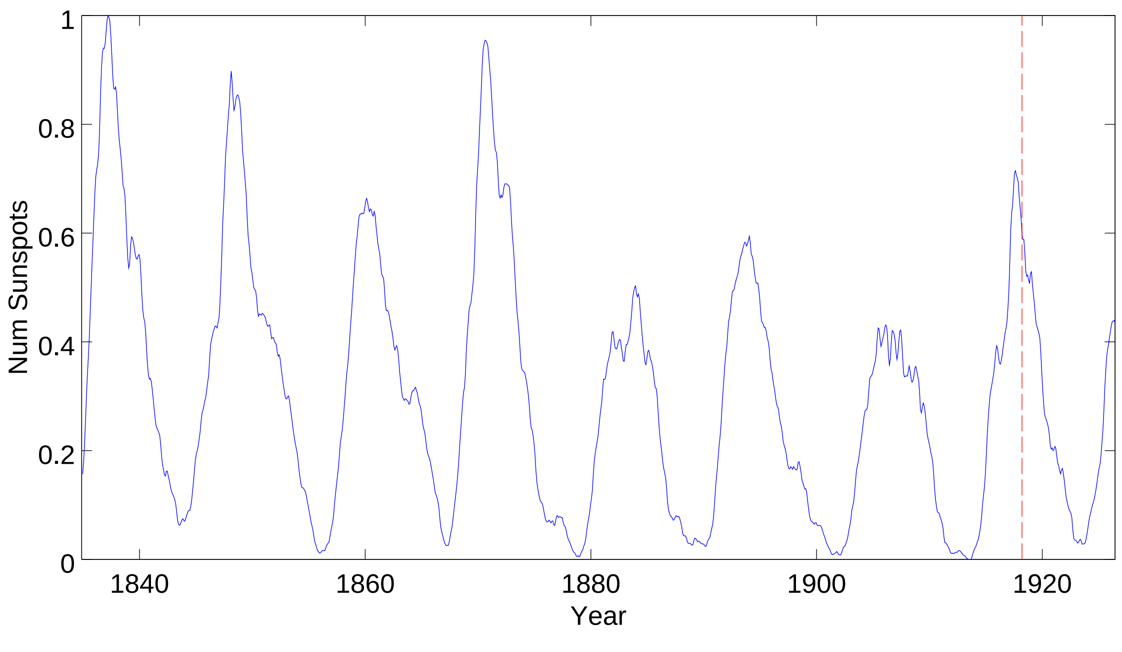


Figure A.10: Sunspot series forecasting benchmarks.

Appendix B

Statistical Significance Testing

When comparing stochastic algorithms, it is important to use statistical significance testing to ensure that any differences are not due to under sampling. This appendix justifies and discusses the statistical significance testing used throughout the thesis.

B.1 Structure of this Appendix

Section B.2 describes and justifies why statistical significance testing should be used when comparing stochastic algorithms. Section B.3 examines the distribution of results generated using an Evolutionary Algorithm (EA) in order to select appropriate statistical significance testing methods. From this analysis, Section B.4 described the statistical significance tests used throughout this thesis.

B.2 Background

When comparing the results of stochastic machine learning methods, such as EAs or Multilayer Perceptrons (MLP)s, it is often insufficient to only compare arithmetic means or medians. This is because one can never know that the sample of results taken from repeating an experiment are representative of the results which would be produced if the experiment were repeated infinite times. That is, one cannot know how accurately the average of the undertaken experiment represents the true average behaviour of the algorithm. For this reason the distributions of the experimental results are often used for comparison. For this statistical significance testing is required.

Statistical significance testing usually operates by giving the probability that the results of repeated runs of two experiments could have been taken from the same underlying distribution i.e. they are not actually different. This is usually used as a null hypothesis, where the aim is to identify that the results of two experiments could have been taken from the same distributions. If there is $< 5\%$ change that, based on the given data, the results of experiment A and experiment B could have been taken from the same distribution, then we can be reasonably confident that they were not. We can therefore reject the null hypothesis and conclude that results of experiment A and experiment B are statistically different.

For instance, take an example where Experiment A produces results with an average greater than Experiment B, but this is not shown to be statistically significant. This means that we cannot be confident that the average result of Experiment A would be greater than Experiment B if a larger average were used. However, suppose that Experiment A produces results with an average greater than Experiment B, and it was shown to be statistically significant. In this case we can be confident that the result of Experiment A producing an average greater than Experiment B was not due to under sampling.

There are two main categories of statistical test, those which assume the distribution of data is normal and those which make no assumption about the distribution; called parametric and non-parametric tests respectively. Typically parametric tests are considered more rigorous and non-parametric tests more general. The next section will assess the distribution of results produced from multiple EA experiments.

B.3 Distribution of Evolutionary Algorithm Results

This section presents an experiment identifying the distribution of the number of generations required to solve a given task using an EA. In this work Cartesian Genetic Programming (CGP) is applied to the task of implementing a full adder circuit using two input AND, NAND, OR and NOR logic gates. A mutation rate of five percent is used with fifty nodes. The experiment is repeated one thousand times in order to generate the histograms shown in Figures B.1. The histogram plots the proportion of runs which required a given number of generations to find a solution. As the number of generations required to find a solution is a commonly used metric to compare EAs, this is a suitable view of the data.

As can be seen in Figure B.1, the data generated from using EAs can clearly be non-normal in distribution. Therefore non-parametric statistical tests are required for fair

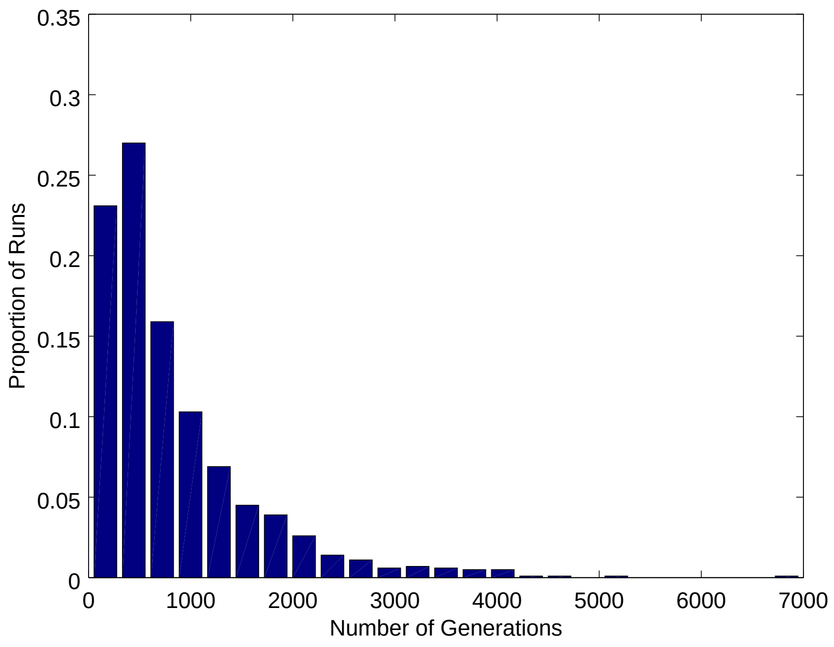


Figure B.1: Histogram of 1000 runs showing the number of generations required for CGP to find a solution to the full adder task.

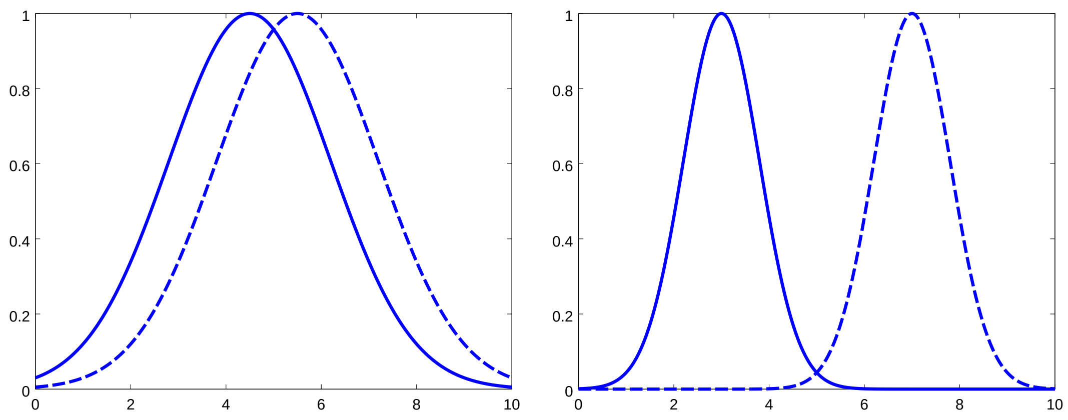
comparisons.

Finally, whether or not the results shown in Figure B.1 are typical for all EAs does not affect the conclusions reached. The fact that an EA *can* produce a non-normal spread of data is all that is needed to justify the use of non-parametric testing. This is because non-parametric test can be applied to both normal and non-normal distributions, whereas parametric tests can only be applied to normal distributions.

B.4 Non-Parametric Statistical Significance Testing

Two commonly used non-parametric statistical significance tests for determining if the difference between two datasets is statically significant are the Mann-Whitney U-test [190] and the Kolmogorov-Smirnoff test [192].

The Mann-Whitney U-test is used to calculate the probability p that two datasets could have the same medians based on their distributions. Typically a value of $p < 0.05$ is used to reject the null hypothesis that the two data sets are taken from the same distribution, and therefore conclude, with reasonable confidence, that any difference is statically significant. The Kolmogorov-Smirnoff operates in a similar way to the Mann-Whitney U-test only it gives the probability that the two data sets could have had the same distributions; not medians.



(a) Smaller Effect Size (b) Larger Effect Size

Figure B.2: Depiction of the effect size measure. (a) shows two distribution which would be awarded a smaller effect size and (b) a larger.

Finally, just because there is a statistical difference between two datasets does not mean that the difference is meaningful. In fact, a statistical difference can usually be ‘forced’ by using very large sample sizes. This is because there is typically likely to be a small difference between two algorithms, however in order to detect this, very large number of runs are required for the comparison. However, in cases where this is necessary, the difference between the two algorithms is likely so small to be of little interest. This is why an additional metric termed effect size [285] is often used. The effect size test produces a value in the range $0.5 \leq A \leq 1$. Where 0.5 is no effect size, ≥ 0.56 is a small effect size, ≥ 0.64 is a medium effect size and ≥ 0.71 is a big effect size. A small effect means that the *spread* of data is much larger than the difference between the two medians of the samples; resulting in any improvement not effecting most individual results. A big effect size means that the difference in medians is proportionally large compared to the distribution of data; see Figure B.2. A big effect size coupled with statistical significance gives confidence that two datasets are statistically significantly different in a meaningful way.

Appendix C

Cartesian Genetic Programming Library

The Cartesian Genetic Programming (CGP) Library is an open source cross platform CGP implementation developed by the author during the PhD. The code base is hosted with github [231] at `https://github.com/AndrewJamesTurner/CGP-Library` and is documented and distributed at `www.cgplibrary.co.uk`.

C.1 Structure of this Appendix

Section C.2 provides a discussion of the need for the developed CGP library including a high level description of the developed library. Section C.3 discusses the overall functionality of the CGP library. Section C.4 discusses a range of visualisation methods provided by the library for inspecting individual chromosomes. Section C.5 describes how the library can be used in the application of CGP to the training of Artificial Neural Network (ANN)s. Section C.6 describes how the library can be used to create recurrent program structures. Section C.7 gives the licenses under which the CGP library is distributed. Section C.8 gives a simple example of how to use the CGP library including sample source code. Finally, Section C.9 gives a closing discussion of the newly released CGP library.

C.2 Background

Despite many advantages of CGP, it has not been adopted to the same extent as standard tree-based Genetic Programming (GP) [300]. Key necessities for a method or technique becoming widely adopted include: benefits over other methods, ease of use and availability. Although a number of CGP implementations are available via the CartesianGP homepage [203], these implementations are typically under documented, unfriendly to new users and adapting them to new situations requires editing and understanding of the entire code base.

It is therefore apparent that in order for CGP to gain in popularity, the issue of there being no standard "go-to" implementation must be addressed. For this reason, an open source cross platform implementation of CGP was developed called CGP library.

The CGP library is intended to be of use in teaching, academic research and real world applications. What distinguishes this implementation from others is that it defines a well-documented CGP Application Programming Interface (API); as opposed to a graphical or command line tool. This API provides functionality for high level applications of CGP, lower level customisation of the CGP algorithm, embedding CGP in wider systems and deploying evolved solutions in their intended final application. The library should be thought of as a set of tools for working with CGP, not as a standalone program in its own right. An advantage of using a well-defined API is that the user does not need to understand or edit the underlying implementation in order to use the CGP library. Additionally, users can benefit from backwards compatible updates to the library. A further advantage of creating a compiled library is that it can be used natively by C [145] and C++ [269] programming languages, but also called from other languages such as Python [283], JAVA [91], Matlab [193] / GNU Octave [56] or R [234] using tools such as SWIG [26] to create wrappers for the compiled library.

The CGP library can also be compiled for a wide range of operating systems as it only depends upon standard C libraries. The CGP library also comes with complete documentation and numerous tutorials intended to ease the learning curve.

C.3 Overall Functionality

The CGP library is intended to be used for teaching, academic research and real world applications. This very broad scope is achieved by hiding detail from the user unless required, maintaining a well-documented API and providing extra tools for specific scenarios.

For instance, in the case of teaching, the CGP library can be applied to a given task with very little "boilerplate" code; see Section C.8. Then, if required, all of the typical parameters (mutation rate, μ, λ, evolutionary strategy, etc) can be controlled via simple setter functions. Additionally, each evolutionary stage (selection scheme, fitness function etc) can be inspected and edited in isolation. All of this is achieved through the API, meaning details can be hidden or exposed as required.

In the case of academic research, the ability to control evolutionary parameters and implement custom evolutionary stages becomes important. Nearly all of the evolutionary stages used by the CGP library can be re-defined to custom versions through the API[1]. Additional functionality is also provided to conduct multiple runs to assess average behaviour. The ability to set random number seeds in order to repeat experiments is also provided. Additionally, results can be saved to easily parsed comma separated value (.csv) files for storage and further analysis.

In the case of real world applications, extra functions are provided to save, load and execute individual chromosomes. This enables found solutions to be stored, distributed and deployed in their intended application. The ability to remove inactive nodes is also provided to reduce the size of saved and loaded chromosomes.

In all cases accessibility and ease of use are important. To this end the CGP library is an open source project available at `www.cgplibrary.co.uk` with the development code hosted with github [231] at `https://github.com/AndrewJamesTurner/CGP-Library`. The ease of use comes from providing a simple API, full documentation and numerous tutorials introducing various aspects of the library with example code.

“Out of the box” the CGP library can be used to create symbolic equations, Boolean logic circuits and ANNs; Cartesian Genetic Programming of Artificial Neural Networks (CGPANN). However, the CGP library also allows users to define their own custom node functions; using the API. Therefore the CGP library can be applied to many additional domains. By default the CGP library fitness function is configured for supervised learning tasks, but by implementing custom fitness functions CGP can also be applied to reinforcement learning tasks. The CGP library can also be used to create feed-forward as well as recurrent solutions; Recurrent Cartesian Genetic Programming (RCGP).

[1]Currently mutation methods can only be implemented by editing the libraries source, not via an API function, but future updates will improve this.

C.4 Visualisation

Visualisation tools are often useful in order to gain an understanding of the solutions found or attempted during evolution. The CGP library currently provides three methods for inspecting chromosomes.

The first function, *printChromosome*, displays chromosomes as text in the terminal / command prompt. A typical CGP chromosome displayed using *printChromosome* is given in Table C.1. Each input and functioning node is labelled with its index in the chromosome. There is a textual description of the node e.g. *input* for input nodes or the operation for the function nodes. Function node operations are followed by space separated values describing each node's inputs. Active nodes are also labelled with an '*'. Finally the last line gives the nodes used as chromosome outputs.

Table C.1: Example CGP chromosome displayed using *printChromosome*.

(0):	input			
(1):	sin	0		*
(2):	mul	0	0	
(3):	mul	0	1	*
(4):	add	1	3	*
(5):	div	1	0	
(6):	div	2	5	
(7):	sin	1		
(8):	mul	4	1	*
outputs:	8			

The second method for visualising CGP chromosomes makes use of the open source cross platform Graphviz utility [61]. The CGP library function *saveChromosomeDot* creates a Graphviz ".dot" file which can be used by Graphviz to create a image similar to that in Figure C.1. The chromosomes are displayed with the inputs on the left, outputs on the right and the position of the internal nodes optimised by Graphviz. Function nodes are labelled with their functionality and given in bold if active.

The third method for visualising CGP chromosomes makes use of the open source cross platform LaTeX typesetting program [172]. The CGP library function *saveChromosomeLatex* creates a LaTeX ".tex" file which can be used by LaTeX (or pdfLaTeX) to create an equation similar to that in Equation C.4.1. Equation C.4.1 gives a mathematical description of the same chromosome given in Table C.1 and Figure C.1. Where x_0 is the single program input and $f_0(x_0)$ is the mapping between this input and

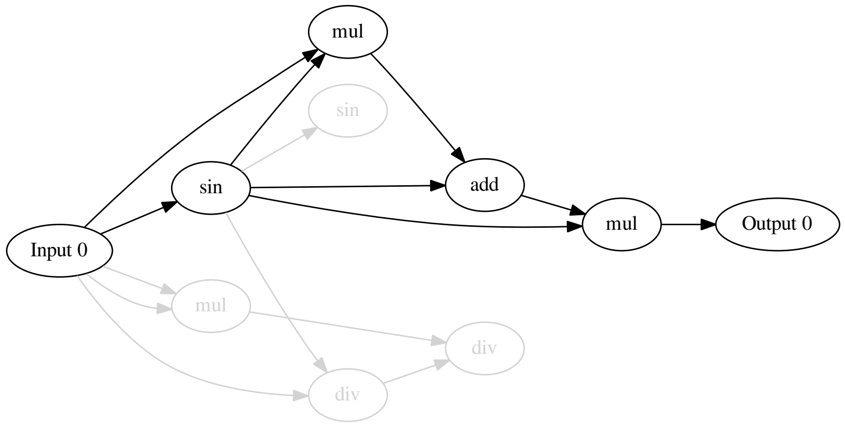


Figure C.1: Example CGP chromosome displayed using *saveChromosomeDot*.

the single chromosome output[2]. This equation represents the phenotype behaviour of the chromosomes and so only represents the functionality of the active nodes in the chromosome. The previous two methods show the active and inactive nodes.

$$f_0(x_0) = ((\sin(x_0) + (x_0 \times \sin(x_0))) \times \sin(x_0)) \tag{C.4.1}$$

These three very distinct methods for visualising CGP chromosomes should enable users to gain an understanding of the evolved solutions. Additionally CGP chromosomes can be saved, using *saveChromosome*, to an easily parsed ".csv" file. This file can then be read by other user created tools for more bespoke visualisation.

C.5 NeuroEvolution

CGPANN is a NeuroEvolution (NE) extension of CGP used throughout this thesis. The CGP library is capable of using CGP to evolve ANNs by simply using suitable node functions i.e. logistic sigmoid. The necessary connection weights are always present in the CGP library but ignored unless required by the transfer function. The range of the connection weights, and other parameters associated with evolving ANNs, can also be set through the API.

[2]For simplicity, here a single input single output CGP chromosome has been displayed. However, in general the CGP library is capable of creating multiple input multiple output programs.

C.6 Recurrent Networks

Another development made to CGP during this thesis is RCGP. RCGP allows recurrent program structure to be evolved including recurrent equations and Recurrent Artificial Neural Network (RANN). This functionality of the CGP library is controlled by a *recurrentConnectionProbability* parameter which is initially set as zero percent; standard CGP. The value of the *recurrentConnectionProbability* can then be set to a non-zero value using the API to enable the creation of recurrent program structures.

C.7 Licenses

The CGP library is released under the open source GNU lesser general public license version 3 [81]. The library is released under the *lesser* general public license so it can be used in open source, closed source and commercial applications under the conditions described in the license. The documentation associated with the CGP library is released under the open source GNU Free Documentation License version 1.3 [80].

C.8 Using the CGP Library

Complete documentation for the CGP library, including installation, tutorials and the API can be found at `www.cgplibrary.co.uk`. In this section only a basic use case is described.

The CGP library uses a number of structures to store data associated with the library; such as the CGP library parameters, training sets and chromosomes. Functions are provided to initialise and free these structures.

A *parameters* structure is used to store the general parameters which control the evolutionary strategy used by CGP; for instance it describes the selection scheme and mutation method to be used. The *parameters* structures are initialised using *initialiseParameters* which takes as arguments the dimensions of the chromosomes to be evolved. Many of the default values stored in *parameters* structures, such as the mutation rate, can be altered but this is not necessary for basic use. Newly initialised *parameters* structures contain an empty function set and it is the responsibility of the user to populate this function set with functions. This is achieved using *addNodeFunction* which takes as arguments an initialised *parameters* structure and a comma separated string of function names. There are currently 30 possible node functions separated into four types: symbolic functions, Boolean logic gates, neuron transfer functions and other[3]. For inspection of the status of the *parameters* structure, *printParameters* displays the stored parameters in the terminal / command prompt.

The *dataSet* structures store training or testing data which can be used by the fitness function when assigning a fitnesses to chromosomes. The *dataSet* structures can be initialised using *initialiseDataSetFromFile* which takes as arguments the location of the file containing the training/testing data; given in a specific format[4].

Once a *parameters* structure (with a populated function set) and a *dataSet* structure have been initialised, CGP can be applied to a given task. This can be achieved using *runCGP*, which takes as arguments a *parameters* and *dataSet* structure as well as the number of generations allowed before terminating the search. After *runCGP* has terminated, it returns an initialised *chromosome* structure containing the best chromosome (solution) found. This *chromosome* structure can be visualised using *printChromosome*.

As can be seen, very little "boilerplate" code is required to use the CGP library. Additionally, repeatedly applying CGP to a given task to determine average behaviour requires only slight modification and the use of *repeatCGP* instead of *runCGP*.

Example C code which follows the previous description is given in Listing C.1. As can be seen very few lines of C are required to apply CGP to a given task.

C.9 Discussion

This Appendix has described a new cross platform open source CGP library intended for teaching, academic research and real world applications. What distinguishes this CGP library from previous implementations is that it defines a well-documented API which can be used to apply CGP to many areas. For instance, it can be used to apply CGP to a given task, used as the "back-end" to implement other CGP software (such as a CGP command line tool) or used in real applications to utilise evolved solutions. The CGP library also includes many helper functions to allow easy application to teaching, academic research and real applications.

The CGP library also includes full documentation and tutorials. This is included to ease the learning curve for new users and to introduce more advanced features of the

[3]Interestingly using a mixture of these three types is also possible.

[4]The first line contains the number of inputs, outputs and the number of data samples. Subsequent lines contain the inputs followed by the outputs for each sample. All values are comma separated.

library. It is hoped that this new CGP library will encourage others to try CGP as an alternative to tree-based GP.

Listing C.1: Example use of CGP library.

```c
#include <stdio.h>
#include <cgp.h>

int main(void){

    struct parameters *params = NULL;
    struct dataSet *trainingData = NULL;
    struct chromosome *chromo = NULL;

    int numInputs = 1;
    int numNodes = 50;
    int numOutputs = 1;
    int nodeArity = 2;

    int numGens = 1000;

    params = initialiseParameters(numInputs, numNodes,
       numOutputs, nodeArity);

    addNodeFunction(params, "add,sub,mul,div");

    printParameters(params);

    trainingData = initialiseDataSetFromFile("temp.data");

    chromo = runCGP(params, trainingData, numGens);

    printChromosome(chromo);

    freeDataSet(trainingData);
    freeChromosome(chromo);
    freeParameters(params);

    return 0;
}
```

Abbreviations

ADF	Automatically Defined Functions
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
ARIMA	Autoregressive Integrated Moving Average
BBO	Biogeography-Based Optimisation
BCGP	Balanced Cartesian Genetic Programming
CE	Cellular Encoding
CGPANN	Cartesian Genetic Programming of Artificial Neural Networks
CGP	Cartesian Genetic Programming
CGPCN	Cartesian Genetic Programming Computational Network
CMA-ES	Covariance Matrix Adaptation Evolutionary Strategies
CNE	Conventional NeuroEvolution
CoSyNE	Cooperative Synapse NeuroEvolution
COVNET	Cooperative Co-evolution Model for evolving Artificial Neural Networks
CPPN	Compositional Pattern Producing Network
CPU	Central Processing Unit
DE	Differential Evolution
DirE	Directly Encoded NeuroEvolution
DXNN	Deus Ex Neural Network
EA	Evolutionary Algorithm
EANT	Evolutionary Acquisition of Neural Topologies
EANT2	Evolutionary Acquisition of Neural Topologies 2

EC Evolutionary Computation

ECGP Embedded Cartesian Genetic Programming

ENGD Explicit Neutral Genetic Drift

EP Evolutionary Programming

EPNet Evolutionary Programming Artificial Networks

ES Evolutionary Strategies

ESN Echo State Networks

ESP Enforced SubPopulation

ETS Exponential Smoothing

EuSANE Eugenic Symbiotic Adaptive NeuroEvolution

FANN Fast Artificial Neural Network Library

GA Genetic Algorithm

GE Grammatical Evolution

GEP Gene Expression Programming

GNARL GeNeralized Acquisition of Recurrent Links

GNN General Neural Networks

GP Genetic Programming

GSGP Geometric Semantic Genetic Programming

HCGA Hierarchical Coevolutionary Genetic Algorithm

HyperNEAT Hypercube-based NeuroEvolution of Augmenting Topologies

INGD Implicit Neutral Genetic Drift

KNN K Nearest Neighbours

LGP Linear Genetic Programming

LSM Liquid State Machines

LUT look-up table

MAE Mean Absolute Error

MA Memetic Algorithms

MCGPANN Modular Cartesian Genetic Programming Artificial Neural Networks

MCGP Modular Cartesian Genetic Programming

MEAN Mean Forecast

MIMO	Multiple-Input Multiple-Output
ML	Machine Learning
MLP	Multilayer Perceptrons
MNN	Modular Neural Network
MSE	Mean Square Error
NEAT	NeuroEvolution of Augmenting Topologies
NE	NeuroEvolution
NevA	NeuroEvolutionary Algorithm
NGD	Neutral Genetic Drift
NMSE	Normalised Mean Square Error
PDGP	Parallel Distributed Genetic Programming
PLS	Partial Least Squares
Push-GP	Push Genetic Programming
RANN	Recurrent Artificial Neural Network
RCGPANN	Recurrent Cartesian Genetic Programming of Artificial Neural Networks
RCGP	Recurrent Cartesian Genetic Programming
RC	Reservoir Computing
RMSE	Root Mean Square Error
RPDT	Recursive Partitioning Decision Trees
RWF	random walk forecasting
SAE	Sum of Absolute Errors
SANE	Symbiotic Adaptive NeuroEvolution
SI	Swarm Intelligence
SMCGP	Self Modifying Cartesian Genetic Programming
SNN	Spiking Neural Networks
SOM	Self Organizing Maps
SRM	Spike Response Model
SVM	Support Vector Machines
TF	Transfer Function
TWEANNs	Topology and Weight Evolving Artificial Neural Networks

References

[1] L F Abbott. Lapicques introduction of the integrate-and-fire model neuron (1907). *Brain Research Bulletin*, 50(5):303–304, 1999.

[2] J. Abonyi and F. Szeifert. Supervised fuzzy clustering for the identification of fuzzy classifiers. *Pattern Recognition Letters*, 24(14):2195–2207, 2003.

[3] A. Agapitos and S. M. Lucas. Learning recursive functions with object oriented genetic programming. In *European Conferance on Genetic Prpgramming (EuroGP)*, pages 166–177, 2006.

[4] LA Aguirre, C Letellier, and J Maquet. Forecasting the time series of sunspot numbers. *Solar Physics*, 249(1):103–120, 2008.

[5] Arbab Masood Ahmad and Gul Muhammad Khan. Bio-signal Processing Using Cartesian Genetic Programming Evolved Artificial Neural Network (CGPANN). In *Frontiers of Information Technology (FIT), 2012 10th International Conference on*, pages 261–268. IEEE, 2012.

[6] Arbab Masood Ahmad, Gul Muhammad Khan, and Sahibzada Ali Mahmud. Classification of Arrhythmia Types using Cartesian Genetic Programming Evolved Artificial Neural Networks. In *Engineering Applications of Neural Networks*, pages 282–291. Springer, 2013.

[7] Arbab Masood Ahmad, Gul Muhammad Khan, and Sahibzada Ali Mahmud. Classification of Mammograms ising Cartesian Genetic Programming Evolved Artificial Neural Networks. In *Artificial Intelligence Applications and Innovations*, pages 203–213. Springer, 2014.

[8] Arbab Masood Ahmad, Gul Muhammad Khan, Sahibzada Ali Mahmud, and Julian Francis Miller. Breast cancer detection using Cartesian Genetic Programming evolved Artificial Neural Networks. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 1031–1038. ACM, 2012.

[9] Fadzil Ahmad, Nor Ashidi Mat Isa, Zakaria Hussain, and Siti Noraini Sulaiman. A genetic algorithm-based multi-objective optimization of an artificial neural network classifier for breast cancer diagnosis. *Neural Computing and Applications*, pages 1–9, 2012.

[10] A.A. Albrecht, G. Lappas, S.A. Vinterbo, C. Wong, and L. Ohno-Machado. Two applications of the LSA machine. In *Proceedings of the 9th International Conference on Neural Information Processing*, volume 1, pages 184–189. IEEE, 2002.

[11] Jawad Ali, Gul Muhammad Khan, and Sahibzada Ali Mahmud. Enhancing Growth Curve Approach Using CGPANN for Predicting the Sustainability of New Food Products. In *Artificial Intelligence Applications and Innovations*, pages 286–297. Springer, 2014.

[12] Jawad Ali, Faheem Zafari, Gul Muhammad Khan, and S Ali Mahmud. Future Clients' Requests Estimation for Dynamic Resource Allocation in Cloud Data Center Using CGPANN. In *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, volume 2, pages 331–334. IEEE, 2013.

[13] C.W. Anderson. Learning to control an inverted pendulum using neural networks. *Control Systems Magazine, IEEE*, 9(3):31–37, 1989.

[14] Edgar Anderson. The irises of the gaspe peninsula. *Bulletin of the American Iris society*, 59:2–5, 1935.

[15] P.J. Angeline, G.M. Saunders, and J.B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *Neural Networks, IEEE Transactions on*, 5(1):54–65, 1994.

[16] J Scott Armstrong. *Principles of forecasting: A Handbook for Researchers and Practitioners*, chapter Extrapolation for time-series and cross-sectional data, pages 217–243. Springer, 2001.

[17] J Scott Armstrong and Fred Collopy. Error measures for generalizing about forecasting methods: Empirical comparisons. *International journal of forecasting*, 8(1):69–80, 1992.

[18] Laurence Ashmore and Julian Francis Miller. Evolutionary art with cartesian genetic programming. https://sites.google.com/site/julianfrancismiller/publications, 2013.

[19] Amir F Atiya and Alexander G Parlos. New results on recurrent network training: unifying the algorithms and accelerating convergence. *Neural Networks, IEEE Transactions on*, 11(3):697–709, 2000.

[20] Marijke F Augusteijn and Thomas P Harrington. Evolving transfer functions for artificial neural networks. *Neural Computing & Applications*, 13(1):38–46, 2004.

[21] J Mark Baldwin. A new factor in evolution (continued). *American naturalist*, pages 536–553, 1896.

[22] Wolfgang Banzhaf. Genotype-phenotype-mapping and neutral variation-a case study in genetic programming. In *Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: Parallel Problem Solving from Nature*, pages 322–332. Springer-Verlag, 1994.

[23] Lionel Barnett. Ruggedness and neutrality: The nkp family of fitness landscapes. In *Artificial Life VI: Proceedings of the sixth international conference on Artificial life*, pages 18–27, 1998.

[24] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on systems, man, and cybernetics*, 13(5):834–846, 1983.

[25] Jonathan Baxter. The evolution of learning algorithms for artificial neural networks. *Complex systems*, pages 313–326, 1993.

[26] David M. Beazley. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[27] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

[28] James C Bezdek, James M Keller, Raghu Krishnapuram, Ludmila I Kuncheva, and Nikhil R Pal. Will the real iris data please stand up? *IEEE Transactions on Fuzzy Systems*, 7(3):368–369, 1999.

[29] Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. *choice*, 1000:2, 1994.

[30] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Occam's razor. *Information processing letters*, 24(6):377–380, 1987.

[31] Sander M Bohte, Joost N Kok, and Han La Poutre. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1):17–37, 2002.

[32] George EP Box, Gwilym M Jenkins, and Gregory C Reinsel. *Time series analysis: forecasting and control*. John Wiley & Sons, 2013.

[33] M.F. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 2007.

[34] Andreas M. Brandmaier. *pdc: Permutation Distribution Clustering*, 2014. R package version 0.5.

[35] Zdeněk Buk, Jan Koutník, and Miroslav Šnorek. NEAT in HyperNEAT substituted with genetic programming. In *Adaptive and Natural Computing Algorithms*, pages 243–252. Springer, 2009.

[36] Erick Cantú-Paz and Chandrika Kamath. An empirical comparison of combinations of evolutionary algorithms and neural networks for classification problems. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 35(5):915–927, 2005.

[37] David J Chalmers. The evolution of learning: An experiment in genetic connectionism. In *Proceedings of the 1990 connectionist models summer school*, pages 81–90. San Mateo, CA, 1990.

[38] Stephan K Chalup and Lukasz Wiklendt. Variations of the two-spiral task. *Connection Science*, 19(2):183–199, 2007.

[39] Kyriakos C Chatzidimitriou and Pericles A Mitkas. A neat way for evolving echo state networks. In *ECAI*, pages 909–914, 2010.

[40] Abdennasser Chebira and Kourosh Madani. *Advances in Soft Computing*, volume 19, chapter A Neural Network Based Approach For Sensors Issued Data Fusion, pages 155–160. Physica-Verlag HD, 2003.

[41] Lin Chen and Damminda Alahakoon. Neuroevolution of augmenting topologies with learning for data classification. In *Information and Automation, 2006. ICIA 2006. International Conference on*, pages 367–371. IEEE, 2006.

[42] Janet Clegg, James Alfred Walker, and Julian Frances Miller. A new crossover technique for Cartesian Genetic Programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1580–1587. ACM, 2007.

[43] Jacob Cohen et al. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[44] Mark Collins. Finding needles in haystacks is harder with neutrality. *Genetic Programming and Evolvable Machines*, 7(2):131–144, 2006.

[45] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[46] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*, pages 72–83. Springer, 2006.

[47] CrossValidated. How to choose the number of hidden layers and nodes in a feedforward neural network?, April 2015.

[48] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[49] Charles Darwin. *On the Origin of Species*. 1859.

[50] Jan G De Gooijer and Rob J Hyndman. 25 years of time series forecasting. *International journal of forecasting*, 22(3):443–473, 2006.

[51] Friedrich Leisch & Evgenia Dimitriadou. *mlbench: Machine Learning Benchmark Problems. R package version 2.1-1*, 2010.

[52] Juan Peralta Donate, German Gutierrez Sanchez, and Araceli Sanchis de Miguel. Time series forecasting. a comparative study between an evolving artificial neural networks system and statistical methods. *International Journal on Artificial Intelligence Tools*, 21(01), 2012.

[53] Keith L. Downing. Adaptive genetic programs via reinforcement learning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 19–26. Morgan Kaufmann, 2001.

[54] Włodzisław Duch and Norbert Jankowski. Survey of neural transfer functions. *Neural Computing Surveys*, 2(1):163–212, 1999.

[55] Wlodzislaw Duch and Norbert Jankowski. Transfer functions: hidden possibilities for better neural networks. In *ESANN*, pages 81–94, 2001.

[56] John W. Eaton, David Bateman, and Soren Hauberg. *GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations*. CreateSpace Independent Publishing Platform, 2009.

[57] Marc Ebner. On the search space of genetic programming and its relation to nature's search space. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2. IEEE, 1999.

[58] Marc Ebner, Patrick Langguth, Juergen Albert, Mark Shackleton, and Rob Shipman. On neutral networks and evolvability. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 1–8. IEEE, 2001.

[59] Marc Ebner, Mark Shackleton, and Rob Shipman. How neutral networks influence evolvability. *Complexity*, 7(2):19–33, 2001.

[60] Agoston E Eiben and Jim Smith. From evolutionary computation to the evolution of things. *Nature*, 521(7553):476–482, 2015.

[61] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.

[62] R. A Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[63] Evelyn Fix and Joseph L Hodges Jr. Discriminatory analysis-nonparametric discrimination: consistency properties. Technical report, DTIC Document, 1951.

[64] D. Floreano, P. Dürr, and C. Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[65] Dario Floreano and Claudio Mattiussi. Evolution of spiking neural controllers for autonomous vision-based robots. *LNCS*, page 3861, 2001.

[66] David B. Fogel. *Evolutionary Computation: The Fossil Record.* Wiley-IEEE Press, 1st edition, 1998.

[67] LJ Fogel, AJ Owens, and MJ Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley, 1966.

[68] Carlos M Fonseca and Marisol B Correia. Developing redundant binary representations for genetic search. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 1675–1682. IEEE, 2005.

[69] Stephanie Forrest and Melanie Mitchell. Relative building-block fitness and the building-block hypothesis, 1993.

[70] Max Kuhn. Contributions from Jed Wing, Steve Weston, Andre Williams, Chris Keefer, Allan Engelhardt, Tony Cooper, Zachary Mayer, Brenton Kenkel, the R Core Team, and Michael Benesty. *caret: Classification and Regression Training*, 2014. R package version 6.0-37.

[71] Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806, 1993.

[72] John Galletly. Evolutionary algorithms in theory and practice: Evolution strategies, evolutionary programming, genetic algorithms. *Kybernetes*, 27(8):979–980, 1998.

[73] Edgar Galván-López, Stephen Dignum, and Riccardo Poli. The effects of constant neutrality on performance and problem hardness in GP. In *EuroGP 2008*, pages 312–324. Springer, 2008.

[74] Edgar Galván-López, Riccardo Poli, Ahmed Kattan, Michael ONeill, and Anthony Brabazon. Neutrality in evolutionary algorithms what do we know? *Evolving Systems*, 2(3):145–163, 2011.

[75] Nicolás García-Pedrajas, César Hervás-Martínez, and José Muñoz-Pérez. Covnet: a cooperative coevolutionary model for evolving artificial neural networks. *Neural Networks, IEEE Transactions on*, 14(3):575–596, 2003.

[76] Surabhi Gaur and MC Deo. Real-time wave forecasting using genetic programming. *Ocean Engineering*, 35(11):1166–1172, 2008.

[77] Zoubin Ghahramani. *Advanced Lectures on Machine Learning*, chapter Unsupervised Learning, pages 72–112. Springer, 2004.

[78] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[79] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[80] GNU. GNU Free Documentation License, April 2014.

[81] GNU. GNU Lesser General Public License, April 2014.

[82] Brian W Goldman and William F Punch. Length bias and search limitations in Cartesian Genetic Programming. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 933–940. ACM, 2013.

[83] Brian W Goldman and William F Punch. Reducing Wasted Evaluations in Cartesian Genetic Programming. In *Proceedings of the 16th European Conference on Genetic Programming (EuroGP)*, volume 7831, pages 61–72. Springer Verlag, 2013.

[84] B.W. Goldman and W.F. Punch. Analysis of Cartesian Genetic Programmings Evolutionary Mechanisms. *Evolutionary Computation, IEEE Transactions on*, PP(99):1–1, 2014. In press.

[85] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, 1997.

[86] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Efficient non-linear control through neuroevolution. *Machine Learning: ECML 2006*, pages 654–662, 2006.

[87] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *The Journal of Machine Learning Research*, 9:937–965, 2008.

[88] Faustino John Gomez and Risto Miikkulainen. *Robust non-linear control through neuroevolution.* PhD thesis, The University of Texas at Austin, 2003.

[89] F.J. Gomez and R. Miikkulainen. Solving non-markovian control tasks with neuroevolution. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 1356–1361, 1999.

[90] F.J. Gomez and R. Miikkulainen. *Robust non-linear control through neuroevolution.* Computer Science Department, University of Texas at Austin, 2003.

[91] James Gosling. *The Java language specification.* Addison-Wesley Professional, 2000.

[92] F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, LUniversiteClaudeBernard-Lyon I, 1994.

[93] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the First Annual Conference on Genetic Programming*, pages 81–89. MIT Press, 1996.

[94] B. Guijarro-Berdiñas, O. Fontenla-Romero, B. Pérez-Sánchez, and P. Fraguela. A linear learning method for multilayer perceptrons using least-squares. *Intelligent Data Engineering and Automated Learning*, pages 365–374, 2007.

[95] Stefan Haflidason. *On the significance of the permutation problem in neuroevolution.* PhD thesis, The University of Manchester, 2010.

[96] Hani Hagras, Anthony Pounds-Cornish, Martin Colley, Victor Callaghan, and Graham Clarke. Evolving spiking neural network controllers for autonomous robots. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 5, pages 4620–4626. IEEE, 2004.

[97] H.J. Hamilton, N. Shan, and N. Cercone. RIAC: A rule induction algorithm based on approximate classification. Technical report, University of Regina, 1996.

[98] Peter JB Hancock. Genetic algorithms and permutation problems: A comparison of recombination operators for neural net structure specification. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 108–122. IEEE, 1992.

[99] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[100] S. Harding, J. F. Miller, and W. Banzhaf. Self Modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing. In *12th European Conference, EuroGP*, pages 133–144. Springer-Verlang, 2009.

[101] Simon Harding, Jrgen Leitner, and Jrgen Schmidhube. *Genetic Programming Theory and Practice X*, chapter Cartesian Genetic Programming for Image Processing, pages 31–44. Springer, 2013.

[102] Simon Harding, Julian F Miller, and Wolfgang Banzhaf. Self modifying Cartesian Genetic Programming: finding algorithms that calculate pi and e to arbitrary precision. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 579–586. ACM, 2010.

[103] Simon Harding, Julian F Miller, and Wolfgang Banzhaf. SMCGP2: self modifying Cartesian Genetic Programming in two dimensions. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1491–1498. ACM, 2011.

[104] Simon L Harding, Julian F Miller, and Wolfgang Banzhaf. Self-modifying Cartesian Genetic Programming. pages 101–124, 2011.

[105] Alex Hazell and Stephen L Smith. Towards an objective assessment of alzheimer's disease: the application of a novel evolutionary algorithm in the analysis of figure copying tasks. In *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, pages 2073–2080. ACM, 2008.

[106] Luis Javier Herrera, Héctor Pomares, Ignacio Rojas, Alberto Guillén, Alberto Prieto, and Olga Valenzuela. Recursive prediction for long term time series forecasting using advanced models. *Neurocomputing*, 70(16):2870–2880, 2007.

[107] Andrew A Hill, Peter LaPan, Yizheng Li, and Steve Haney. Impact of image segmentation on high-content screening data quality for SK-BR-3 cells. *BMC bioinformatics*, 8(1):340, 2007.

[108] G.E. Hinton, S. Osindero, and Y.W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[109] Geoffrey E Hinton and Terrence J Sejnowski. Optimal perceptual inference. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 448–453. IEEE Piscataway, NJ, 1983.

[110] Henrique Steinherz Hippert, Carlos Eduardo Pedreira, and Reinaldo Castro Souza. Neural networks for short-term load forecasting: A review and evaluation. *Power Systems, IEEE Transactions on*, 16(1):44–55, 2001.

[111] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*, 117(4):500544, 1952.

[112] V. Hoekstra. An overview of neuroevolution techniques. 2011.

[113] John H Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[114] Charles C Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting*, 20(1):5–10, 2004.

[115] DC Hope, E Munday, and SL Smith. Evolutionary algorithms in the classification of mammograms. In *Computational Intelligence in Image and Signal Processing, 2007. CIISP 2007. IEEE Symposium on*, pages 258–265. IEEE, 2007.

[116] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[117] Kurt Hornik, Christian Buchta, and Achim Zeileis. Open-source machine learning: R meets weka. *Computational Statistics*, 24(2):225–232, 2009.

[118] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[119] Radek Hrbacek and Vaclav Dvorak. Bent Function Synthesis by Means of Cartesian Genetic Programming. In *Parallel Problem Solving from Nature–PPSN XIII*, pages 414–423. Springer, 2014.

[120] M.L. Huang, Y.H. Hung, and W.Y. Chen. Neural network classifier with entropy based feature selection on breast cancer diagnosis. *Journal of medical systems*, 34(5):865–873, 2010.

[121] U Huebner, NB Abraham, and CO Weiss. Dimensions and entropies of chaotic intensity pulsations in a single-mode far-infrared NH 3 laser. *Physical Review A*, 40(11):6354, 1989.

[122] L. Huelsbergen. Learning recursive sequences via evolution of machine-language programs. In *Proceedings of the Second Annual Conference on Genetic Programming*, page 186194, 1997.

[123] Martijn A Huynen. Exploring phenotype space through neutral evolution. *Journal of molecular evolution*, 43(3):165–169, 1996.

[124] Martijn A Huynen, Peter F Stadler, and Walter Fontana. Smoothness within ruggedness: the role of neutrality in adaptation. *Proceedings of the National Academy of Sciences*, 93(1):397–401, 1996.

[125] R.J. Hyndman and Y. Khandakar. Automatic time series forecasting: The forecast package for R. *Statistical Software*, 26(3), 2008.

[126] Rob J Hyndman, Muhammad Akram, and Blyth C Archibald. The admissible parameter space for exponential smoothing models. *Annals of the Institute of Statistical Mathematics*, 60(2):407–426, 2008.

[127] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice.* OTexts, 2014.

[128] Rob J Hyndman, George Athanasopoulos, Slava Razbash, Drew Schmidt, Zhenyu Zhou, Yousaf Khan, Christoph Bergmeir, and Earo Wang. *forecast: Forecasting functions for time series and linear models*, 2014. R package version 5.4.

[129] Rob J Hyndman and Anne B Koehler. Another look at measures of forecast accuracy. *International journal of forecasting*, 22(4):679–688, 2006.

[130] C. Igel. Neuroevolution for reinforcement learning using evolution strategies. In *The Congress on Evolutionary Computation, CEC'03*, volume 4, pages 2588–2595. CEC, 2003.

[131] Eugene M Izhikevich. Simple model of spiking neurons. *Neural Networks, IEEE Transactions on*, 14(6):1569–1572, 2003.

[132] Eugene M Izhikevich. Which model to use for cortical spiking neurons? *Neural Networks, IEEE Transactions on*, 15(5):1063–1070, 2004.

[133] Herbert Jaeger. The echo state approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148:34, 2001.

[134] David Jefferson, Rob Collins, Claus Cooper, Michael Dyer, Margot Flowers, Richard Korf, Charles Taylor, and Alan Wang. The genesys system: Evolution as a theme in artificial life. In *Proceedings of Second Conference on Artificial Life. Redwood City, CA: Addison-Wesley*, 1990.

[135] Yongnan Ji, Jin Hao, Nima Reyhani, and Amaury Lendasse. Direct and recursive prediction of time series using mutual information selection. In *Proceedings of the 8th international conference on Artificial Neural Networks: computational Intelligence and Bioinspired Systems*, pages 1010–1017. Springer-Verlag, 2005.

[136] Michael I Jordan. Serial order: A parallel distributed processing approach. Technical report, Institute for Cognitive Science, 1986.

[137] Mark A Kaboudan. Genetic programming prediction of stock prices. *Computational Economics*, 16(3):207–236, 2000.

[138] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Artificial Intelligence Research*, 4:237–285, 1996.

[139] Tatiana Kalganova and Julian Miller. Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In *Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on*, pages 54–63. IEEE, 1999.

[140] SM Kamruzzaman, A.R. Hasan, A.B. Siddiquee, M. Mazumder, and E. Hoque. Medical diagnosis using neural network. In *Proceedings of the International Conference on Electrical and Computer Engineering (ICECE-2004)*, pages 537–540, 2004.

[141] M. Karabatak and M.C. Ince. An expert system for detection of breast cancer based on association rules and neural network. *Expert Systems with Applications*, 36(2):3465–3469, 2009.

[142] Alexandros Karatzoglou, Alex Smola, Kurt Hornik, and Achim Zeileis. kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9):1–20, 2004.

[143] Yohannes Kassahun. *Towards a unified approach to learning and adaptation.* PhD thesis, Inst. für Informatik und Praktische Mathematik, 2006.

[144] A. Kattan, R. Abdullah, and R.A. Salam. Harmony search based supervised training of artificial neural networks. In *International Conference on Intelligent Systems, Modelling and Simulation (ISMS)*, pages 105–110. IEEE, 2010.

[145] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[146] G Khan, Rabia Arshad, S Mahmud, and Fahad Ullah. Intelligent bandwidth estimation for variable bit rate traffic. *Evolutionary Computation, IEEE Transactions on*, 19(1):151–155, 2015.

[147] G.M. Khan, J.F. Miller, and D.M. Halliday. A developmental model of neural computation using Cartesian Genetic Programming. In *Genetic And Evolutionary Computation Conference: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2535–2542, 2007.

[148] G.M. Khan, J.F. Miller, and D.M. Halliday. Developing neural structure of two agents that play checkers using Cartesian Genetic Programming. In *Proceedings of the 2008 GECCO conference on Genetic and evolutionary computation*, pages 2169–2174. ACM, 2008.

[149] Gul Muhammad Khan, Shahid Khan, and Fahad Ullah. Short-term daily peak load forecasting using fast learning neural network. In *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*, pages 843–848. IEEE, 2011.

[150] Gul Muhammad Khan, Atif Rashid Khattak, Faheem Zafari, and Sahibzada Ali Mahmud. Electrical load forecasting using fast learning recurrent neural networks. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–6. IEEE, 2013.

[151] Gul Muhammad Khan, Fahad Ullah, and Sahibzada Ali Mahmud. MPEG-4 Internet Traffic Estimation Using Recurrent CGPANN. In *Engineering Applications of Neural Networks*, pages 22–31. Springer, 2013.

[152] Gul Muhammad Khan, Faheem Zafari, and S Ali Mahmud. Very short term foad forecasting using Cartesian Genetic Programming evolved recurrent neural networks (CGPRNN). In *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, volume 2, pages 152–155. IEEE, 2013.

[153] Maryam Mahsal Khan, Masood Arbab Ahmad, Muhammad Gul Khan, and Julian F Miller. Fast learning neural networks using Cartesian Genetic Programming. *Neurocomputing*, 121:274–289, 2013.

[154] Maryam Mahsal Khan, Gul Muhammad Khan, and Julian F Miller. Evolution of neural networks using cartesian genetic programming. In *Proceedings of IEEE World Congress on Computational Intelligence CEC 2010*, 2010.

[155] M.M. Khan, G.M. Khan, and J.F. Miller. Efficient representation of recurrent neural networks for markovian/non-markovian non-linear control problems. In *Intelligent Systems Design and Applications (ISDA), 2010 10th International Conference on*, pages 615–620. IEEE, 2010.

[156] M.M. Khan, G.M. Khan, and J.F. Miller. Evolution of optimal ANNs for non-linear control problems using cartesian genetic programming. In *Proceedings of International Conference on Artificial Intelligence (ICAI 2010)*, 2010.

[157] Mehdi Khashei and Mehdi Bijari. An artificial neural network (p,d,q) model for timeseries forecasting. *Expert Systems with Applications*, 37(1):479–489, 2010.

[158] Motoo Kimura et al. Evolutionary rate at the molecular level. *Nature*, 217(5129):624–626, 1968.

[159] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[160] Joshua D Knowles and Richard A Watson. On the utility of redundant encodings in mutation-based evolutionary search. In *Parallel Problem Solving from NaturePPSN VII*, pages 88–98. Springer, 2002.

[161] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145, 1995.

[162] Arthur Kordon. Tower problem, 2015.

[163] SB Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica*, 31:249–268, 2007.

[164] J. Koutník, F. Gomez, and J. Schmidhuber. Evolving neural networks in compressed weight space. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO-10)*, pages 619–626, 2010.

[165] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[166] John R Koza and James P Rice. *Genetic programming II: automatic discovery of reusable programs*, volume 40. MIT press Cambridge, 1994.

[167] Max Kuhn, Steve Weston, and Nathan Coulter. C code for C5.0 by R. Quinlan. *C50: C5.0 Decision Trees and Rule-Based Models*, 2014. R package version 0.1.0-21.

[168] Tin-Yau Kwok and Dit-Yan Yeung. Constructive feedforward neural networks for regression problems: A survey. Technical report, Hong Kong: Department of Computer Science, 1995.

[169] Tin-Yau Kwok and Dit-Yan Yeung. Constructive algorithms for structure learning in feedforward neural networks for regression problems. *Neural Networks, IEEE Transactions on*, 8(3):630–645, 1997.

[170] Tin-Yau Kwok and Dit-Yan Yeung. Objective functions for training new hidden units in constructive neural networks. *Neural Networks, IEEE Transactions on*, 8(5):1131–1148, 1997.

[171] Jean-Baptiste Lamarck. *Philosophie Zoologique.* Museum d'Histoire Naturelle, 1809.

[172] Leslie Lamport. *LaTeX: User's Guide & Reference Manual.* Addison-Wesley Publishing Company, Inc., 1986.

[173] V Landassuri-Moreno and John A Bullinaria. Biasing the evolution of modular neural networks. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 1958–1965. IEEE, 2011.

[174] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[175] W.B. Langdon and R. Poli. Fitness causes bloat. In P.K. Chawdhry, R. Roy, and R.K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer London, 1998.

[176] William B Langdon and Wolfgang Banzhaf. Repeated sequences in linear genetic programming genomes. *Complex Systems*, 15(4 (c)):285–306, 2005.

[177] William B Langdon and Riccardo Poli. Why ants are hard. Technical report, School of Computer Science, The University of Birmingham, Birmingham, UK, 1998.

[178] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. Exploring strategies for training deep neural networks. *The Journal of Machine Learning Research*, 10:1–40, 2009.

[179] Yong Liu and Xin Yao. Evolutionary design of artificial neural networks with different nodes. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 670–675. IEEE, 1996.

[180] David Lowe and D Broomhead. Multivariable functional interpolation and adaptive networks. *Complex systems*, 2:321–355, 1988.

[181] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.

[182] Mantas LukošEvičIus and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[183] Liying Ma and Khashayar Khorasani. New training strategies for constructive neural networks with application to regression problems. *Neural networks*, 17(4):589–609, 2004.

[184] W. Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[185] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

[186] Michael C Mackey and Leon Glass. Oscillation and chaos in physiological control systems. *Science*, 197(4300):287–289, 1977.

[187] Spyros Makridakis, Chris Chatfield, Michele Hibon, Michael Lawrence, Terence Mills, Keith Ord, and LeRoy F Simmons. The M2-competition: A real-time judgmentally based forecasting study. *International Journal of Forecasting*, 9(1):5–22, 1993.

[188] K.P.B.O.L. Mangasarian. Neural network training via linear programming. *Advances in Optimisation and Parallel Computing*, pages 56–67, 1992.

[189] OL Mangasarian, R. Setiono, and WH Wolberg. *Large-Scale Numerical Optimization*, chapter Pattern recognition via linear programming: Theory and application to medical diagnosis, pages 22–31. Philadelphia, PA: SIAM, 1990.

[190] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[191] Timmy Manning and Paul Walsh. Improving the performance of CGPANN for breast cancer diagnosis using crossover and radial basis functions. In *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*, pages 165–176. Springer, 2013.

[192] Frank J Massey Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[193] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[194] Brian W Matthews. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[195] August Mayer and Helmut A Mayer. Multi-chromosomal representations in neuroevolution. In *Computational Intelligence*, pages 245–250, 2006.

[196] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of mathematical biology*, 5(4):115–133, 1943.

[197] James McDermott, David R White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, et al. Genetic programming needs better benchmarks. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798. ACM, 2012.

[198] Andreas Meier, Mark Gonter, and Rudolf Kruse. Accelerating convergence in cartesian genetic programming by using a new genetic operator. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 981–988. ACM, 2013.

[199] Willem Melssen, Ron Wehrens, and Lutgarde Buydens. Supervised kohonen networks for classification problems. *Chemometrics and Intelligent Laboratory Systems*, 83(2):99–113, 2006.

[200] Bjrn-Helge Mevik, Ron Wehrens, and Kristian Hovde Liland. *pls: Partial Least Squares and Principal Component regression*, 2013. R package version 2.4-3.

[201] Julian Francis Miller. What bloat? Cartesian genetic programming on Boolean problems. In *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, 2001.

[202] Julian Francis Miller, editor. *Cartesian Genetic Programming.* Springer, 2011.

[203] Julian Francis Miller. http://www.cartesiangp.co.uk/, April 2014.

[204] Julian Francis Miller, Dominic Job, and Vesselin K Vassilev. Principles in the evolutionary design of digital circuitspart i. *Genetic programming and evolvable machines*, 1(1-2):7–35, 2000.

[205] Julian Francis Miller, Dominic Job, and Vesselin K Vassilev. Principles in the evolutionary design of digital circuitspart ii. *Genetic programming and evolvable machines*, 1(3):259–288, 2000.

[206] Julian Francis Miller and Maktuba Mohid. Function optimization using Cartesian genetic programming. In *Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 147–148. ACM, 2013.

[207] Julian Francis Miller and S.L. Smith. Redundancy and computational efficiency in Cartesian Genetic Programming. *Evolutionary Computation, IEEE Transactions on*, 10(2):167–174, 2006.

[208] Julian Francis Miller and P Thomson. Cartesian genetic programming. In *Proceedings of the Third European Conference on Genetic Programming (EuroGP)*, volume 1820, pages 121–132. Springer-Verlag, 2000.

[209] David Montana, Eric VanWyk, Marshall Brinn, Joshua Montana, and Stephen Milligan. Evolution of internal dynamics for neural network nodes. *Evolutionary Intelligence*, 1(4):233–251, 2009.

[210] David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

[211] D.E. Moriarty. *Symbiotic evolution of neural networks in sequential decision tasks.* PhD thesis, University of Texas at Austin, 1997.

[212] D.E. Moriarty and R. Mikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine learning*, 22(1):11–32, 1996.

[213] D. Nauck, R. Kruse, et al. Obtaining interpretable fuzzy classification rules from medical data. *Artificial intelligence in medicine*, 16(2):149, 1999.

[214] Durre Nayab, Gul Muhammad Khan, and Sahibzada Ali Mahmud. Prediction of foreign currency exchange rates using CGPANN. In *Engineering Applications of Neural Networks*, pages 91–101. Springer, 2013.

[215] D J Newman, S Hettich, C L Blake, and Merz C J. UCI Repository of machine learning databases. Technical report, Irvine, CA: University of California, Department of Information and Computer Science, 1998.

[216] M. Nishiguchi and Y. Fujimoto. Evolution of recursive programs with multi-niche Genetic Programming (mnGP). In *Evolutionary Computation IEEE World Congress on Computational Intelligence*, page 247252, 1998.

[217] Steffen Nissen. Implementation of a fast Artificial Neural Network library (FANN). *Report, Department of Computer Science University of Copenhagen (DIKU)*, 31, 2003.

[218] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Peter J. Angeline and Kenneth E. Kinnear, Jr., editors, *Advances in Genetic Programming*, pages 111–134. MIT Press, Cambridge, MA, USA, 1996.

[219] Martin O'Halloran, Seamus Cawley, Brian McGinley, Raquel Cruz Conceicao, Fearghal Morgan, Edward Jones, and Martin Glavin. Evolving spiking neural network topologies for breast cancer classification in a dielectrically heterogeneous breast. *Progress In Electromagnetics Research Letters*, 25:153–162, 2011.

[220] Ludo Pagie and Paulien Hogeweg. Evolutionary consequences of coevolving targets. *Evolutionary computation*, 5(4):401–418, 1997.

[221] Jooyoung Park and Irwin W Sandberg. Universal approximation using radial-basis-function networks. *Neural computation*, 3(2):246–257, 1991.

[222] NG Pavlidis, OK Tasoulis, Vassilis P Plagianakos, G Nikiforidis, and MN Vrahatis. Spiking neural network training using evolutionary algorithms. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 4, pages 2190–2194. IEEE, 2005.

[223] C.A. Pena-Reyes and M. Sipper. A fuzzy-genetic approach to breast cancer diagnosis. *Artificial intelligence in medicine*, 17(2):131–155, 1999.

[224] N. Pokudom. Determine of appropriate neural networks structure using ant colony system. In *ICCAS-SICE, 2009*, pages 4522–4525. IEEE, 2009.

[225] D. Polani and R. Miikkulainen. Fast reinforcement learning through eugenic neuro-evolution. Technical report, University of Texas at Austin, Austin, TX, 1999.

[226] K. Polat and S. Güneş. Breast cancer diagnosis using least square support vector machine. *Digital Signal Processing*, 17(4):694–701, 2007.

[227] Riccardo Poli. Parallel distributed genetic programming. *New Ideas in Optimization, Advanced Topics in Computer Science*, pages 403–431, 1999.

[228] Riccardo Poli, W William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to Genetic Programming.* Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008.

[229] L. Prechelt. Proben1: A set of neural network benchmark problems and benchmarking rules. *Fakultät für Informatik, Univ. Karlsruhe, Karlsruhe, Germany, Tech. Rep*, 21:94, 1994.

[230] Lutz Prechelt. *Neural Networks: Tricks of the Trade*, chapter Early Stopping - But When?, pages 53–67. Springer Berlin Heidelberg, 2012.

[231] Tom Preston-Werner, Chris Wanstrath, and P.J. Hyett. github, April 2014.

[232] J. W Prior. Eugenic evolution for combinatorial optimization. Master's thesis, The University of Texas at Austin, 1998.

[233] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, 1993.

[234] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2014.

[235] A. Raad, A. Kalakech, and M. Ayache. Breast cancer classification using neural network approach: MLP and RBF. *Networks*, 7(8):9, 2012.

[236] Nicholas J Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, 1(1):67–90, 1993.

[237] Ingo Rechenberg. *Evolutionsstrategien.* Springer, 1978.

[238] Russell Reed. Pruning algorithms-a survey. *Neural Networks, IEEE Transactions on*, 4(5):740–747, 1993.

[239] Mehreen Rehman, Jawad Ali, Gul Muhammad Khan, and Sahibzada Ali Mahmud. Extracting trends ensembles in solar irradiance for green energy generation using neuro-evolution. In *Artificial Intelligence Applications and Innovations*, pages 456–465. Springer, 2014.

[240] Belew Richard K, John Mcinerney, and Schraudolph Nico l, N. Evolving networks: Using the genetic algorithm with connectionist learning. Technical report, Cognitive Computer Science Research group, Computer Science and Engr. Dept (C-014), Univ. California at San Diego, 1990.

[241] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[242] Joseph A Rothermich and Julian F Miller. Studying the Emergence of Multicellularity with Cartesian Genetic Programming in Artificial Life. In *GECCO Late Breaking Papers*, pages 397–403, 2002.

[243] Franz Rothlauf and David E Goldberg. Redundant representations in evolutionary computation. *Evolutionary Computation*, 11(4):381–415, 2003.

[244] Royal Observatory of Belgium. World data center for the production, preservation and dissemination of the international sunspot number, July 2014.

[245] David E Rumelhart, Geoffrey E Hintont, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[246] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, 1995.

[247] Massimo Santini, Andrea Tettamanzi, Julian F Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea GB Tettamanzi, and William B Langdon. Genetic programming for financial time series. In *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038, pages 361–370. Springer-Verlag, 2001.

[248] Klaus Schliep and Klaus Hechenbichler. *kknn: Weighted k-Nearest Neighbors*, 2014. R package version 1.2-5.

[249] AntonMaximilian Schfer and HansGeorg Zimmermann. Recurrent neural networks are universal approximators. In StefanosD. Kollias, Andreas Stafylopatis, Wodzisaw Duch, and Erkki Oja, editors, *Artificial Neural Networks ICANN 2006*, volume 4131 of *Lecture Notes in Computer Science*, pages 632–640. Springer Berlin Heidelberg, 2006.

[250] Lukáš Sekanina. Image filter design with evolvable hardware. In *Applications of Evolutionary Computing*, pages 255–266. Springer, 2002.

[251] MR Senapati, AK Mohanty, S. Dash, and PK Dash. Local linear wavelet neural network for breast cancer recognition. *Neural Computing & Applications*, pages 1–7, 2011.

[252] G.I. Sher. Dxnn platform: The shedding of biological inefficiencies. *arXiv preprint arXiv:1011.6022*, 2010.

[253] SIDC-team. The International Sunspot Number. *Monthly Report on the International Sunspot Number, online catalogue*, 1700-1987.

[254] Nils T Siebel, Jonas Botel, and Gerald Sommer. Efficient neural network pruning during neuro-evolution. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 2920–2927. IEEE, 2009.

[255] Nils T Siebel and Gerald Sommer. Evolutionary reinforcement learning of artificial neural networks. *International Journal of Hybrid Intelligent Systems*, 4(3):171–183, 2007.

[256] Sara Silva and Ernesto Costa. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179, 2009.

[257] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[258] Dan Simon. Biogeography-based optimization. *Evolutionary Computation, IEEE Transactions on*, 12(6):702–713, 2008.

[259] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences, Aug 2014.

[260] P. Smolensky. *Parallel distributed processing: explorations in the microstructure of cognition*, chapter Information processing in dynamical systems: foundations of harmony theory, pages 194–281. MIT Press, 1986.

[261] Terence Soule and Robert B Heckendorn. An analysis of the causes of code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 3(3):283–309, 2002.

[262] L Spector and A. Robinson. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines*, 3(1):7 – 40, 2002.

[263] Kenneth O Stanley. Exploiting regularity without development. In *Proceedings of the AAAI Fall Symposium on Developmental Systems*, page 37. AAAI Press Menlo Park, CA, 2006.

[264] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009.

[265] Kenneth Owen Stanley. *Efficient evolution of neural networks through complexification.* PhD thesis, The University of Texas at Austin, 2004.

[266] K.O. Stanley and R. Miikkulainen. Efficient evolution of neural network topologies. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 2, pages 1757–1762. IEEE, 2002.

[267] K.O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[268] E Stefanoudaki, F Kotsifaki, and A Koutsaftakis. Classification of virgin olive oils of the two major cretan cultivars based on their fatty acid composition. *Journal of the American Oil Chemists' Society*, 76(5):623–626, 1999.

[269] Bjarne Stroustrup. *The C++ programming language.* Pearson Education India, 1986.

[270] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[271] Terry Therneau, Beth Atkinson, and Brian Ripley. *rpart: Recursive Partitioning and Regression Trees*, 2014. R package version 4.1-8.

[272] Terry M Therneau, Elizabeth J Atkinson, et al. An introduction to recursive partitioning using the rpart routines. Technical report, Technical Report 61. URL http://www. mayo. edu/hsr/techrpt/61. pdf, 1997.

[273] S.B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski, S.E. Fahlman, D. Fisher, et al. The monk's problems a performance comparison of different learning algorithms. Technical report, Carnegie Mellon University, 1991.

[274] Leonardo Trujillo, Luis Muñoz, Enrique Naredo, and Yuliana Martínez. NEAT, There's No Bloat. In *Genetic Programming*, pages 174–185. Springer, 2014.

[275] Hsing-Chih Tsai. Using weighted genetic programming to program squat wall strengths and tune associated formulas. *Engineering Applications of Artificial Intelligence*, 24(3):526–533, 2011.

[276] YR Tsoy and VG Spitsyn. Using genetic algorithm with adaptive mutation mechanism for neural networks design and training. In *Science and Technology. Proceedings. The 9th Russian-Korean International Symposium on*, pages 709–714. IEEE, 2005.

[277] Alan Mathison Turing and Darrel Ince. *Mechanical intelligence*, volume 3. North Holland, 1992.

[278] Andrew James Turner. Improving crossover techniques in a genetic program. Masters Dissertation, 2012.

[279] Andrew James Turner and Julian Francis Miller. Cartesian Genetic Programming encoded Artificial Neural Networks: A Comparison using Three Benchmarks. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO-13)*, pages 1005–1012, 2013.

[280] Fahad Ullah, Gul M Khan, and Sahibzada Ali Mahmud. Intelligent bandwidth management using fast learning neural networks. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 867–872. IEEE, 2012.

[281] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael ONeill, Robert I McKay, and Edgar Galván-López. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, 2011.

[282] Erik Van Nimwegen, James P Crutchfield, and Martijn Huynen. Neutral evolution of mutational robustness. *Proceedings of the National Academy of Sciences*, 96(17):9716–9720, 1999.

[283] Guido Van Rossum and Fred L Drake Jr. *Python tutorial.* Centrum voor Wiskunde en Informatica, 1995.

[284] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. Measuring bloat, overfitting and functional complexity in genetic programming. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 877–884. ACM, 2010.

[285] András Vargha and Harold D Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[286] Zdenek Vasicek. Cartesian gp in optimization of combinational circuits with hundreds of inputs and thousands of gates. In *Genetic Programming*, pages 139–150. Springer, 2015.

[287] V. K. Vassilev and J. F. Miller. The Advantages of Landscape Neutrality in Digital Circuit Evolution. In *Proc. International Conference on Evolvable Systems*, volume 1801 of *LNCS*, pages 252–263. Springer Verlag, 2000.

[288] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S.* Springer, New York, fourth edition, 2002. ISBN 0-387-95457-0.

[289] Katharina Völk, Julian F Miller, and Stephen L Smith. Multiple network CGP for the classification of mammograms. In *Applications of Evolutionary Computing*, pages 405–413. Springer, 2009.

[290] Jilles Vreeken. Spiking neural networks, an introduction. *Technical Report UU-CS*, (2003-008):1–5, 2003.

[291] Andreas Wagner. Robustness, evolvability, and neutrality. *FEBS letters*, 579(8):1772–1778, 2005.

[292] James Alfred Walker and Julian Francis Miller. The automatic acquisition, evolution and reuse of modules in Cartesian Genetic Programming. *Evolutionary Computation, IEEE Transactions on*, 12(4):397–417, 2008.

[293] James Alfred Walker, Julian Francis Miller, and Rachel Cavill. A multi-chromosome approach to standard and embedded Cartesian Genetic Programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 903–910. ACM, 2006.

[294] James Alfred Walker, Katharina Völk, Stephen L Smith, and Julian Francis Miller. Parallel evolution using multi-chromosome Cartesian Genetic Programming. *Genetic Programming and Evolvable Machines*, 10(4):417–445, 2009.

[295] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[296] R. Wehrens and L.M.C. Buydens. Self- and Super-organising Maps in R: the kohonen package. *J. Stat. Softw.*, 21(5), 2007.

[297] A. S. Weigend and N. A. Gershenfeld. *Time Series Prediction: Forecasting the Future and Understanding the Past.* Addison-Wesley, 1994.

[298] Andreas Weigend. Santa fe competition data sets, July 2014.

[299] Daniel Weingaertner, Victor K Tatai, Ricardo R Gudwin, and Fernando J Von Zuben. Hierarchical evolution of heterogeneous neural networks. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 2, pages 1775–1780. IEEE, 2002.

[300] David R White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May OReilly, and Sean Luke. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, 2013.

[301] Darrell Whitley. Genetic algorithms and neural networks. *Genetic algorithms in engineering and computer science*, 3:203–216, 1995.

[302] Darrell Whitley, Keith Mathias, and Patrick Fitzhorn. Delta coding: An iterative search strategy for genetic algorithms. In *ICGA*, volume 91, pages 77–84, 1991.

[303] Hadley Wickham. *classifly: Explore classification models in high dimensions*, 2014. R package version 0.4.

[304] Bernard Widrow and Michael A Lehr. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

[305] A.P. Wieland. Evolving neural network controllers for unstable systems. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 667–673. IEEE, 1991.

[306] D. Wilson and D. Kaur. Search, neutral evolution, and mapping in evolutionary computing: A case study of grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 13(3):566–590, 2009.

[307] Garnett Wilson and Malcolm Heywood. Learning recursive programs with cooperative coevolution of genetic code mapping and genotype. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1053–1061. ACM, 2007.

[308] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, 2005.

[309] Herman Wold et al. Estimation of principal components and related models by iterative least squares. *Multivariate analysis*, 1:391–420, 1966.

[310] David H Wolpert and William G Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

[311] Sewall Wright. The Roles of Mutation, Inbreeding, Crossbreeding, and Selection in Evolution. In *Sixth International Congress of Genetics*, pages 356–366. Brooklyn Botanic Garden, 1932.

[312] J.Y. Wu. MIMO CMAC neural network classifier for solving classification problems. *Applied Soft Computing*, 11(2):2326–2333, 2011.

[313] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[314] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *Neural Networks, IEEE Transactions on*, 8(3):694–713, 1997.

[315] Xin Yao. Universal approximation by genetic programming. In *In Foundations of Genetic Programming*, 1999.

[316] Samaneh Yazdani and Jamshid Shanbehzadeh. Balanced Cartesian Genetic Programming via migration and opposition-based learning: application to symbolic regression. *Genetic Programming and Evolvable Machines*, 16(2):133–150, 2015.

[317] L. Yingwei, N. Sundararajan, and P. Saratchandran. Performance evaluation of a sequential minimal radial basis function (RBF) neural network learning algorithm. *Neural Networks, IEEE Transactions on*, 9(2):308–318, 1998.

[318] Tina Yu and Julian Miller. Neutrality and the evolvability of Boolean function landscape. In Julian Miller, Marco Tomassini, PierLuca Lanzi, Conor Ryan, AndreaG.B. Tettamanzi, and WilliamB. Langdon, editors, *Genetic Programming*, volume 2038 of *Lecture Notes in Computer Science*, pages 204–217. Springer Berlin Heidelberg, 2001.

[319] Tina Yu and Julian Miller. Finding needles in haystacks is not hard with neutrality. In James A Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea Tettamanzi, editors, *Genetic Programming*, volume 2278 of *Lecture Notes in Computer Science*, pages 13–25. Springer Berlin Heidelberg, 2002.

[320] Tina Yu and Julian Francis Miller. Through the interaction of neutral and adaptive mutations, evolutionary search finds a way. *Artificial Life*, 12(4):525–551, 2006.

[321] Faheem Zafari, Gul Muhammad Khan, Mehreen Rehman, and Sahibzada Ali Mahmud. Evolving recurrent neural network using cartesian genetic programming to predict the trend in foreign currency exchange rates. *Applied Artificial Intelligence*, 28(6):597–628, 2014.

[322] G Peter Zhang, B Eddy Patuwo, and Michael Y Hu. A simulation study of artificial neural networks for nonlinear time-series forecasting. *Computers & Operations Research*, 28(4):381–396, 2001.

[323] Guoqiang Zhang, B Eddy Patuwo, and Michael Y Hu. Forecasting with artificial neural networks:: The state of the art. *International journal of forecasting*, 14(1):35–62, 1998.

[324] Yun Zhang and Mengjie Zhang. A multiple-output program tree structure in genetic programming. In *Proceedings of the 7th Asia-Pacific Conference on Complex Systems*, 2004.