REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		FINAL -15 MAR 93 TO 31 AUG 96

4. TITLE AND SUBTITLE
EVOLVING NEURAL NETWORKS FOR NONLINEAR CONTROL

5. FUNDING NUMBERS
F49620-93-1-0238
61102F 2304/HS

6. AUTHOR(S)
DR HINTZ

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
GEORGE MASON UNIVERSITY
OFFICE OF SPONSORED PROGRAMS
4400 UNIVERSITY DRIVE
FAIRFAX VA 22030

AFOSR-TR-96
0518

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
110 DUNCAN AVE, SUITE B115
BOLLING AFB DC 20332-8080

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
F49620-93-1-0238

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
SEE REPORT

14. SUBJECT TERMS	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Final Report

Evolving Neural Networks for Nonlinear Control

AFOSR Grant No. F49620-93-1-0238

Kenneth J. Hintz, khintz@gmu.edu
Z. Zhang
D. Duane
George Mason University
Department of Electrical and Computer Engineering
and Center of Excellence in C^3I
Fairfax, VA 22030

September 30, 1996

prepared for

Dr. Arje Nachman, AFOSR/NM

Table of Contents

List of Figures

Abstract

An approach to creating Amorphous Recurrent Neural Networks (ARNN) using Genetic Algorithms (GA) called 2pGA has been developed and shown to be effective in evolving neural networks for the control and stabilization of both linear and nonlinear plants, the optimal control for a nonlinear regulator problem, the XOR problem, and an amplitude modulation (AM) detector. This new approach consists of a two-phase GA with the first phase using a set of Lindenmayer System (L-System) production rules to evolve the NN architectures, and the second phase using genetic hill-climbing for connectivity weight tuning. The resulting amorphous (non-layered) recurrent NNs are real-valued as opposed to the binary-valued nets generated by the original GANNET program. Integral absolute error was the fitness function used in these experiments.

A striking indirect result of this research is the few number of neurons which are required to effect the compensation and stabilization. Typical networks are from 4-15 neurons.

The inclusion of a neural insertion/deletion operator in both the 2pGA and GANNET2 methods allows for the size of the NN to be evolved. This capability has been used to develop an empirical relationship between problem complexity and the required NN complexity. Problem complexity is measured by the number of symbols required to differentiate among binary patterns in a pattern recognition task. NN complexity is measured by number of neurons. While not yet definitive, empirical data from ARNNs evolved by GANNET2 show what appears to be a logarithmic relationship between the complexity of a regular expression and the size of a recurrent neural network which recognize it. Additional experiments are being performed to extend the region of evolved data to improve our confidence in this conclusion.

1 Introduction to Evolved Amorphous Neural Networks

Amorphous neural networks have no identifiable layered structure which makes them less amenable to mathematical analysis than other structures, yet more similar to their biological counterparts. GMU is unique in its approach to the complete evolution of amorphous NNs including their structure, weights and threshold, size, and selection of input and output neurons.

The inherent recurrence of amorphous recurrent NNs (ARNN) makes them capable of responding to inputs differently depending on the preceding sequence of inputs. Amorphous neural networks are spatially as well as temporally context-sensitive in that they respond both to the current parallel input as well as to the preceding inputs. The result of this is that the ARNNs have both a short-term and long-term memory. Short-term in that they can store recent events in their volatile recurrent connections and long-term in that the structure has been evolved based on evolutionary pressure in the face of a training environment.

Conventional NNs are layered and usually are not recurrent. Without recurrence there is no memory and hence no potential for previous-state-dependent behavior (*e.g.*, finite state machine behavior). This research demonstrates the ability of genetic algorithms to evolve amorphous recurrent neural networks which are capable of mapping input vectors to output vectors as well as behaving like finite state machines (*e.g.*, counters and nonlinear controllers). The genetic algorithm approach allows us to apply a fitness function which includes evolutionary pressure to minimize the size of the NN as well as to insure that the NN operates in the desired manner. Others who have used GAs to evolve NNs have fixed the network architecture and size and evolved the weights, or, have evolved the architecture and used backpropagation to find the weights. GMU's approach completely evolves all aspects of amorphous neural networks.

1.1 Assessment of the current state of the art in GA/NN

An extensive literature search, performed during fall of 1994 and covering the previous five years, yielded only 15 papers of significance to the field of genetically engineered neurocontrollers (GENC). There are three major areas of research in GENC:

1) the NN optimization problem,
2) the adaptive control problem, and,
3) the study of emergent behavior or properties that arise as a result of the optimization process.

In the NN optimization problem, three approaches are taken:

1) evolution of weights/biases while holding the NN structure constant,
2) evolution of weights, biases, and the NN structure, and,

3) a hybrid approach which combines the evolution of the structure with a more common training approach such as back-propagation.

The emphasis in the literature is on the first two approaches with the static feedforward network being the most popular but with recurrent networks being used in many applications. It remains a problem that the structure of the appropriate NN for an application is problem specific. The interest in recurrent NNs rather than feedforward networks seems to stem from the thought that the oscillatory nature of recurrent networks leads to the construction of a control system consisting of a multiplicity of coupled oscillators which allow for "seamless" switching from one mode of operation to another.

Adaptive control with GENC can be subdivided into direct and indirect approaches with the indirect approach (*a priori* specification of a plant model prior to evolution) being the more popular. Its popularity stems from the difficulty in establishing a fitness function which is not dominated by highly nonlinear measures and sensitive to deviations from expected values of states or inputs. This indirect approach to adaptive control can be implemented either on-line or off-line, with off-line being the more popular. The current trend in the literature is towards "immunized neurocontrollers" which have both a long term and short term evolutionary process.

Thorough testing of the evolutionary properties of our first program to evolve artificial NNs using genetic algorithms (GANNET) has revealed several deficiencies which have been resolved. The GANNET program evolves the amorphous (non-layered, recurrent) binary-valued NNs in its entirety including weights, interconnect structure, input neurons, and output neurons. Its first deficiency is in the genome structure itself which limits the number of neurons to 256. The number of neurons has been increased in GANNET2 in anticipation of the larger networks required to solve complex control problems. GANNET also has complete connectivity making the GA crossover operator very disruptive of the NN structure causing a second problem. Since one of the underlying principles of GAs is that of preserving partial solutions from multiple population members and recombining them into more effective offspring, the fact that the neurons could be connected from one end of the genome to the other makes for an unacceptably high probability that the crossover operator will destroy evolved subnetworks which contributed to forming an effective solution of the larger problem. GANNET2 reflects this change in genome structure in that the "spanning" distance of neuron connectivity on the genome is controlled. The effect of this is to generate networks with high local connectivity and sparse long distance connectivity.

A third difficulty with GANNET is that it is binary-valued which means that continuously valued inputs and outputs must be represented by ordered combinations of inputs and outputs. This requirement lays an additional burden on the GA. GANNET2 does not resolve this problem but 2pGA does. GANNET2 is being used to continue the study of NN complexity as a function of problem complexity using serial strings which are 1-bit wide, hence this is not a problem. The 2-phase GA, 2pGA, is real-valued and is the method used in developing the controllers, stabilizers, and solving other demonstration problems.

The result of our experience with the limitations of binary-valued GAs has led us to develop a new genetic algorithm approach based on Lindenmayer systems in which there is a two-phase evolutionary process for evolving NNs. This two-phase approach uses a Lindenmayer system for the parallel rewriting of the genetic code strings. In the first phase of the approach, a GA is used to evolve the production rules (L-system like) which generates the structures of the neural networks. The fitness of each set of production rules is determined by a second GA which is used to fine-tune the weights of the NNs generated by these production rules. The neurons themselves have real-valued inputs rather than binary-valued inputs. This relieves the GA from the requirement to correctly order the parallel binary inputs which represent an analog value as well as also select the outputs in the correct parallel order for generation of an analog control value.

1.2 Genetic Algorithms

Since 1975, when Holland [1] first published his book *Adaptation in Natural and Artificial Systems,* the computational paradigm of genetic algorithms has received favor as method for implementing adaptive systems, and also as a method for searching numerical spaces or optimizing functions. The field of Artificial Neural Networks marked its beginning in 1943 when McCulloch and Pitts [2] published their paper 'A Logical Calculus of Ideas Immanent in Nervous Activity.' In the late 1980s, experiments were performed to examine the feasibility of specifying neural network parameters using Genetic Algorithms.

Researchers have applied Genetic Algorithms (GAs) to problems such as task scheduling, and function optimization, as well as to the task of specifying some or all aspects of an artificial neural network. When applied by researchers, GAs computationally utilize the natural evolutionary process as first described by Darwin in his *The Origin of Species* [3] to evolve a solution to a given problem.

Pseudo-code describing the Canonical Genetic Algorithm (the natural version) [4] appears below:

```
Randomly generate an initial population M(0)
while (1)
        {
        Compute and save the fitness u(m) for each individual m in current population
        M(t)

        Define selection probabilities p(m) for each individual m in M(t) so that p(m) is
        proportional to u(m)

        Generate M(t+1) by probabilistically selecting individuals from M(t) to produce
        offspring via genetic operators
        }
```

The first step is to randomly generate values for the individuals which make up the population of parents. Then, the process enters a loop and continues indefinitely evaluating individuals' fitnesses, probabilistically selecting individuals for reproduction as a function of their fitness, and generating new individuals using genetic operators.

When used as a computational tool, as it is in this research, the process works essentially the same. Evaluation is performed using a user-specified fitness function which yields a numeric value that evaluates the usefulness of the individual with regard to solving the problem. After evaluating the fitness, decisions need to be made about which individuals to keep for the next generation. This is known as the population maintenance step, and can take on a variety of forms. Typically, the fitness of the parents and its offspring are compared and the fittest is retained to be used as a new parent. Furthermore, a GA will generally select certain individuals from the population of new parents to be used more or less often (or not at all) to generate the offspring. After selecting which individuals will be used to generate the offspring, a step unique to the computational version of the GA takes place. The major difference between the computational and natural versions of GAs is that at some point in the loop, termination conditions are checked, and the process is terminated if a termination condition is reached. In this implementation, after each evaluation of fitness, a decision is made about whether to continue the evolutionary process or not. If an individual solution is found which has the appropriate level of fitness or if too many generations have passed without sufficiently good results, then the program terminates. Each pass through the loop is considered a generation. Provided that the evolutionary process doesn't terminate, genetic operators are applied to each population of parents in order to generate a population of offsprings.

Genetic operators typically consist of crossover and mutation. In GANNET1 and GANNET2, a special case of the mutation operator, known as `resize`, is also utilized which probabilistically alters the size of the neural network. After resizing, the process restarts by evaluating the newly generated population of individuals.

1.3 Neural Networks

Artificial neural networks, hereinafter referred to as neural networks (NNs), have been applied to

$$n_i\,(t+1) \;=\; Q\left(\sum_j w_{ij}\; n_i(t) \;-\; m_i\right) \qquad \textbf{(1)}$$

many different problems such as pattern recognition, nonlinear control, and the traveling salesman problem. The building block of a NN is the neuron [5]. A neuron's behavior is based upon the squashed summation of the negated threshold value, and the product of neuron input values and weights as described in Equation **(1)**. m_i is the threshold (DC offset) term, w_{ij} are the weights which determine the behavior, $n_j(t)$ are the values of the inputs to the neuron and $n_j(t+1)$ is the

output of the neuron. $n_j(t)$ and $n_j(t+1)$ take on values of either 1 or 0, as does the squashing function $Q(x)$, described in Equation **(2)**

NNs are formed by connecting specified neuronal outputs to neuronal inputs, designating certain

$$Q(x) = \begin{cases} 1 & \text{if } x \geq 0; \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

neuronal inputs to take input from the environment and designating certain neuronal outputs to act as environmental outputs.

1.4 Combinations of GAs and NNs

A NN consists of three parts: neuronal weight values and squashing function (also known, and referred to later, as neuronal behavior), neuronal interconnections, and network connections to the environment. A search of the literature revealed three distinct methods of applying GAs to the evolution of NNs. The first method involves manually selecting a fixed topology while altering connection weights using genetic algorithms. Montana [6] and Whitley [7] both first decide on a topology and then specify weight values using GAs.

The second application of GAs to evolving NNs specifies a NN architecture or topology while using another method to find the weights of the NN. In Miller *et al.* [8] a procedure is presented that evaluates fitness after generating an interconnection structure and applies the back propogation algorithm for a fixed number of epochs. Harp and Samad [9] take this notion a step further and evolve a NN's structure, size and learning parameters while still using backpropogation to find connection weight values. These approaches are problematic because they don't allow for amorphous (non-layered) recurrent NNs and each NN instantiation's fitness measure is computationally intensive.

A third method involves creating NNs by completely evolving both their structure and weights using the GA approach. This third type of NN is called an AMmorphous neural NETwork (AMNET) because there is no identifiable layered structure, they may be recursive, and they may have multiple neurons serving as input neurons for one source. Research carried out at George Mason University by Spofford [25] in 1990 resulted in the first AMNET. Spofford named his software Genetic Algorithm Neural NETwork (GANNET), a program written in **C** that uses GAs to evolve NNs for a variety of problems and under varying conditions. GANNET was the first attempt at evolving all components of a neural network and is described further below. Since GANNET, there have been other attempts at evolving AMNETs.

Harvey, *et al.*[10] have written software which evolves AMNETs that adaptively control an autonomous mobile robot. They use continuous, real-valued networks with unrestricted connections and time delay between units. Their neurons have two types of inputs: normal and

veto. Normal inputs to neurons are traditional weighted inputs which get summed and have a squashing function applied, as opposed to the veto input to a neuron which activates when a threshold is exceed. The veto inputs connect to the output of another neuron. When the veto input is activated, all normal outputs from that neuron change to zero (turn off). The veto mechanism is included to better model biological neural networks.

The most important component of this research deals with the development of a method for evolving neural networks which can control nonlinear systems having been given nothing more than training data. Sometimes the training data consists of a linear or nonlinear model of the plant and a desired fitness function. An approach which we call the two-phase genetic algorithm (2pGA) has been developed under this grant which evolves modular recurrent neural networks (NNs). It first uses a GA to evolve a near optimal architecture of NNs for specific problems and then fine-tunes the weights and biases of the NN using a second GA. The 1stGA uses the production-rule-based (PRB) techniques to encoding architectures of arbitrary NNs and evolves the production rules instead of evolving NNs directly. The 2ndGA uses genetic hill-climbing to train the NN and determine part of the fitness of the individual in 1stGA.

One goal of this research was to establish an empirical relationship between required network size and problem complexity. Multiple complexity measures were investigated and determined to be equivalent. Pattern recognition problems were researched specifically as they were less difficult to implement than other complexity measures and yet can be shown to be equivalent to them. NNs were evolved to solve selected problems using the `resize` operator to evolve networks with the minimal number of neurons. The details of the experiments plus the results of the comparison of problem complexity with minimal evolved network size appears later.

2 Two-Phase Genetic Algorithm

The original approach of using binary-valued NNs as in our earlier GANNET program has been found to be difficult to use for real-valued control problems since it can be shown that it evolves, in essence, a logic circuit of minimal size, albeit one with memory. This type of network is typically called a Boolean net. While this is of value to digital logic design and finite state machine synthesis, and the approach can be readily adapted to the minimization of multi-input/multi-output functions, it is of limited value to the generation of non-bang-bang controllers. The difficulty arises in that real-valued inputs and outputs must be encoded as multi-bit binary-valued inputs and outputs. This appears to put an inordinate amount of pressure on the evolutionary strategy which requires excessive computer time to evolve effective solutions. We have switched approaches to the evolution of real-valued neural networks using a two-layered GA. The first GA is a meta-GA in that it evolves the production rules for the generation of real-valued NN structures. The second GA essentially trains the weights of the resulting NNs using genetic hill-climbing. The evaluation of the resulting NNs is then used by the first GA to modify the production rules for the generation of the next population.

This approach, called a two-phase genetic algorithm (2pGA), has been developed in order to evolve modular, amorphous recurrent neural networks (ARNNs). It combines two GAs to evolve a near-optimal architecture of NNs. The first GA (1stGA) uses a production-rule based technique, called a Lindenmayer system or L-System, to encode architectures for generating NNs of arbitrary size and form.

2.1 Contemporary Approaches to Evolving NNs

Using the techniques of evolutionary computation to evolve neural networks has received wide attention in recent years [11][12][13][14]. The key to successfully applying genetic algorithms to the evolution of neural networks is in the encoding of a neural network into its genotype. Direct encoding of a neural network, *e.g.* connectivity matrix, normally suffers from the problem of scalability, the production of non-functional offsprings by the crossover operator, and/or by having a large chromosome size, even though neural networks described by connectivity matrices have been shown to work in some cases [15]. Some researchers encode the parameters for specific NN architectures (*e.g.* feedforward NNs) [16]. These approaches have good scalability, but are restricted to specific architectures. The idea of using biological metaphors in designing neural networks has stimulated different approaches [17][18][19].

Lindenmayer system-like (L-system) [20] production rules are sometimes used to model the growth of neural networks. Instead of directly encoding a neural network, a set of production rules can be encoded as the genotype. A genetic algorithm or genetic programming (GP) [21] can then be used to evolve these production rules. Another idea borrowed from nature is modularity. There is much evidence indicating that the modular organization of the brain exists at different anatomical scales [22]. Preliminary results also show that using modularity in designing NNs can improve their performance. From the implementation point-of-view, modular neural networks, as compared to fully interconnected neural networks, can be more easily implemented in VLSI.

Among the production-rule based encoding techniques, Boers and Kuiper's biological metaphors [17] and Gruau's Cellular Encoding (CE) [18] are very promising approaches. In the Boers *et al.* method, two L-systems are used to model the growth of feedforward NNs. These L-systems, which are encoded into fixed length binary strings according to a, so-called, genetic code table are evolved by a conventional GA. The fitness is determined by a traditional learning algorithm (error back-propagation training algorithm). The Boers *et al.* method is very biologically plausible and, although the preliminary results of this approach are promising, the coding mechanism is inefficient in that the coding mechanism does not guarantee that chromosomes always produce usable production rules.

In Gruau's CE, the development of an NN is done by utilizing a grammar tree, which is a program tree that specifies how the NN is to be constructed, based on an axiom and an initial network graph. Each individual (genotype) in the GA is itself a grammar tree, which is used to produce a NN (phenotype) with bipolar weights (± 1) and binary thresholds (*0* or *1*). The resulting NN is used for learning Boolean functions. Gruau uses a rewriting operator, *R*, for recurrent use of total

or part of the grammar tree to make it possible to develop a family of NNs for a class of specific problems. Gruau's CE is efficient and has desirable properties, such as completeness, closure, modularity, and scalability.

Most of the approaches focus on optimizing feedforward NNs. As an alternative, an approach called two-phase genetic algorithm (2pGA), which uses techniques similar to those used in [17,18], is used here to evolve the NNs for control and stabilization of linear and nonlinear systems. Our approach combines two GAs to evolve the architectures of a population of NNs first, then determines the weights and biases of the evolved NNs. It is shown later in the summary that 2pGA has been used to evolve modular recurrent neural networks which can compensate stable linear system, stabilize unstable linear systems, compensate nonlinear systems, and stabilize unstable, nonlinear systems. The detail of this new approach are discussed in the next section.

2.2 Two-phase Genetic Algorithm Implementation

The procedure for evolving neural networks using 2pGA is divided into two stages. The first-stage determines the NN's architecture, with the second-stage fine-tuning its weights and biases. The above idea is based on the conjecture that the initial architecture is not only important for rapid learning, but that it also induces the system to generalize its learned behavior to instances not previously encountered [22]. The advantage of this kind of decomposition is that it greatly reduces the search space compared to that which evolves architectures and weights of neural networks simultaneously [15]. The block diagram of 2pGA is shown in **Figure 1**. The first GA (1stGA) uses a production-rule-based encoding technique to evolve NN architectures. Each individual in the 1stGA is a CE like grammar tree. Its fitness is determined by the second GA (2ndGA), which is a more conventional GA and is used to evolve the weights and biases for each network specified by an individual produced by the 1stGA. The 2ndGA terminates in the *N*th generation, a fixed number chosen by the designer *apriori*. A value which is mapped from the statistical data (*e.g* the average fitness, the best-so-far fitness , *etc.*) of the 2ndGA in the N-th generation will be the major part of the fitness of each individual in the 1stGA. One of the possible mappings which was chosen for this implementation is:

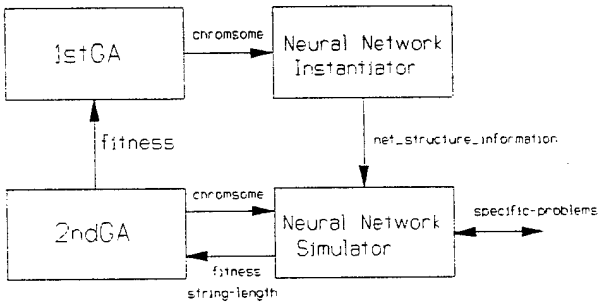


Figure 1 Block diagram of 2pGA.

$$Fitness_{1stGA} = \lambda * BestSoFarFitness_{2ndGA} + (1 - \lambda) * AverageFitness_{2ndGA} \qquad (3)$$

where λ is a value such that $0 \leq \lambda \leq 1$. After 1stGA finds a near-optimal architecture for the specific problem, the weights and biases may be fine-tuned by 2ndGA or any of a number of other traditional training algorithms.

The inputs to the NN Instantiator & Simulator (NNIS) are the chromosomes (individuals) in the 1stGA and 2ndGA. The outputs from the NNIS are the information about how much space is required to represent each individual in 2ndGA and the fitness for each individual in 2ndGA. The tasks of the NNIS are to

1) interpret the chromosomes which are produced by 1stGA into production rules (program trees),
2) use these rules to produce the NN by applying the program trees to a user selected axiom,
3) output the structure information for each resulting NN,
4) determine the required string length in 2ndGA for an individual for the given architecture of the NN,
5) interpret the chromosome produced by 2ndGA into the weights and biases for the associated NN, and,
6) perform the simulation for the NN on the specified test problems.

2.3 The Phase-I Genetic Algorithm (1stGA)

The task of 1stGA is to evolve architectures of modular recurrent NNs. It uses production-rule based encoding techniques. Gruau's CE encodes both the architecture and the Boolean weights of a neural network and Boers' methods are currently only applicable to feedforward neural networks. For these reasons, their approaches cannot be used directly to suit our purposes. An alternative which combines and expands on the ideas of Boers and Gruau is used instead. A relative connect string (RCS) representation of NNs, which is able to represent modular recurrent NNs, is inspired from the relative skip strings (RSS) [17] of Boers *et al.* which is able to represent feedforward NNs only. A Gruau CE-like grammar tree is used to produce the desired string, thereby overcoming the inefficiency inherent in Boers' encoding mechanism.

2.3.1 The Relative Connect String Representation

The alphabet used in the strings is $\Sigma = \{ A\text{-}Z, 0\text{-}9, +, -,), (\}$. The syntax of the terms used in relative connect strings is defined by the following grammar:

```
<network> :=   <module>
<module>  :=   (<nodes>) | (<nodes><module>) | (<module><module>)
<nodes>   :=   <node> | <node><nodes>
<node>    :=   <feedback_links><letter><feedforward_links>
<feedback_links>    :=   -<number> | -<number><feedback_links>
<feedforward_links> :=   +<number> | +<number><feedforward_links>
<number>  :=   <digit> | <digit><number>
```

```
<digit>    :=   0 | 1 | 2 | ... | 9
<letter>   :=   A | B | ... | Z
```

A letter in the string is interpreted as a neuron. A positive number (*e.g.* $+k$) behind the letter (neuron), which is called the feedforward link value, is interpreted as a feedforward link, which means that the neuron connects to the k-th node/module (calculate from the node the number associated) on the right of the letter. A negative number (*e.g.* $-k$) before the letter, which is called feedback link value, is interpreted as a feedback link, which means that the neuron connects to the k-th node/module on the left of the letter. A feedback-to-self connection is indicated by -0. The absence of a number or the number $+0$ means there is no link to other neurons. If $k=1$, it means that there is a link from the associated neuron to its nearest neighbor node/module. If k is so large that the link will go out of the range, it is assumed to indicate a connection to the furthest node/module in the string.

An input node of a module is a node that does not receive feedforward input from within the module. An output node of a module is a node that has no feedforward output to other nodes within the module. A link from module one (*M1*) to module two (*M2*) is interpreted as the output(s) of *M1* are connected to the input(s) of *M2*. A NN itself can be considered as a module, so input(s) and output(s) of a NN are determined by the input(s) and the output(s) of the module which defines the NN.

Careful observation of this formulation shows that the relative connect strings can represent any architecture of NNs. For the NNs in **Figure 2**, their relative connect strings are ($B+2+3B+1+2+3B+2B+1B$) and ($B+2+3B+1+2-2B-2B$) for (a) and (b), respectively. Where link values $+0$ is omitted.

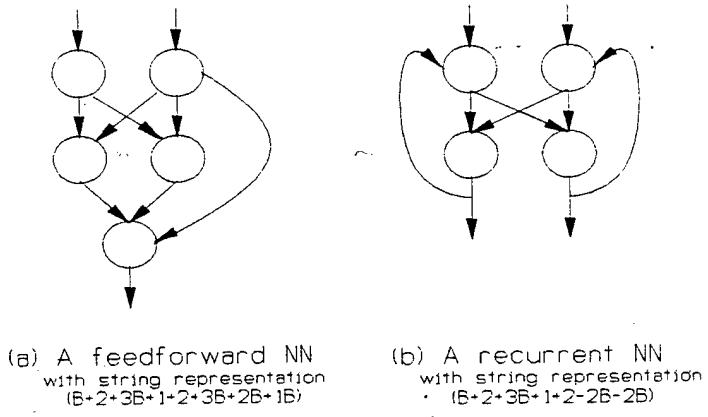


Figure 2 Some examples of neural networks.

2.3.2 The Modified Cellular Encoding (MCE)

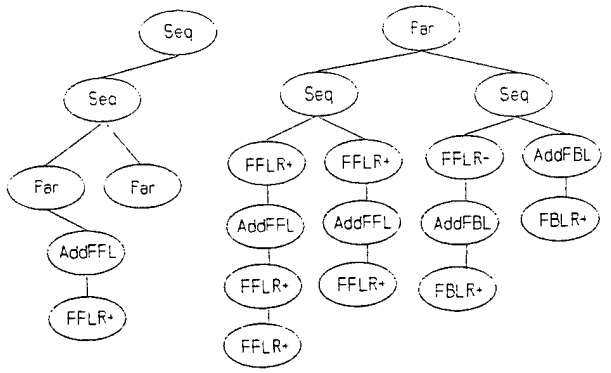
Since the mechanism to produce strings in Boers *et al.* is less efficient compared to the cellular encoding (grammar tree) used by Gruau, a CE-like grammar tree, called modified cellular encoding (MCE), is used to produce the desired strings. For each cell there is a file of feedforward link registers, denoted as FFLRs, and a file of feedback link registers, denoted as FBLRs, associated it. FFLRs and FBLRs are used to store `<feedforward_links>` and `<feedback_links>`, respectively. Each cell has a reading head which points to a node of a grammar tree and rewrites and/or modifies itself according to the

program symbol which labels the node of the grammar tree. After executing the program symbol the reading head will point to its subtree. If there is no subtree of the pointed-to node, then the cell loses its reading head after executing the program symbol and becomes a neurons/module. The axiom of the grammar is (*B*) or (*-0B*) which means that the size of FFLRs and FBLRs for the single cell is *1* and the value in the current FFLR is *0* and the current FBLR is nothing (null or *0*). The axiom corresponds to the initial network string. The reading head pointed to the root of the grammar tree. A set of program symbols used in MCE is Γ = { *Seq, Par, AddFFL, AddFBL, RemFFL, RemFBL, FFLR+, FFLR-, FBLR+, FBLR-, ShiftFF, ShiftFB* }, where *Seq* and *Par*, which are similar to these used in CE, are two division program symbols to expand the size of NNs; *AddFFL*, *AddFBL*, *RemFFL*, and *RemFBL* are four program symbols to add/delete a link; *FFLR+, FFLR-, FBLR+, FBLR-, ShiftFF,* and *ShiftFB* six program symbols to adjust links. The main difference between the MCE and CE is that MCE only requires a maximum arity for each program symbol.

Grammar trees which can produce the NNs of **Figure 2** are shown in **Figure 3**. The procedure of the development of the strings for **Figure 2(a)** is that (*B*) → (*B+1B*) → (*B+1B+1B*) → (*(B+2B+1)(B+2B+1)B*) → (*(B+2B+1+1)(B+2B+1)B*) → (*(B+2B+1+2)(B+2B+1)B*). The procedure for **Figure 2(b)** is that (*B*) → (*(BB)*) → (*(B+1BB+1B)*) → (*(B+2B+1B+0-1B)*) → (*(B+2+1B+1+1-1B-2B)*) → (*(B+2+2B+1+2-2B-2B)*) → (*(B+2+3B+1+2-2B-2B)*). It can be noticed that to develop recurrent NNs from axiom (*B*) is more difficult than to develop feedforward NNs from (*B*).

2.3.3 The Genetic Operators and Fitness in 1stGA

An individual in 1stGA is a grammar tree whose structure is almost the same as that used in GP, so the genetic operators in GP can be used here and all advantages of the operators in GP, such as closure under crossover and mutation, will be inherited. In addition to the crossover and mutation, encapsulation will be used more frequently in the 1stGA than in conventional GP. It is hoped that the modularity will be found by using encapsulation. The dynamically formed module (a subtree in the grammar tree) which is denoted as M_I, $I = 1, 2, ...$, will be added into the set of program symbols Γ and the associated subtree will be stored in a module pool. Since the maximum arity of the program symbols used in 1stGA is less than or equal to two, binary trees are chosen to represent the program trees.



(a) For the NN in Fig.2(a) ((B+2B+1+2)(B+2B+1)B)

(b) For the NN in Fig.2(b) ((B+2+3B+1+2-2B-2B))

Figure 3 Grammar trees for the example NNs.

Since we encode input(s) and output(s) of NNs into grammar trees, the correct number of inputs and outputs of NNs is desired. So it is better to add a extra term in the fitness function to reward the NNs which have the correct number of inputs and outputs.

2.4 The Phase-II Genetic Algorithm (2ndGA)

Since there is no good algorithm to train arbitrary, recurrent NNs, a GA is a good alternative. The 2ndGA used here is similar to the GA proposed by Montana and Davis [12]. It has several differences from the more traditional GAs. One is that it uses the $(\mu+\lambda)$-selection of evolutionary strategy [23] to select the new population. Other major differences are that it uses real-valued encodings, small population size, and relatively high mutation rates. This kind of GA is also called a genetic hill-climber (GHC). The individual in the 2ndGA is a fixed length string consisting of all weights/biases of a NN. Each weight/bias is represented by a floating point (*i.e.*, real-valued encoding). The size of the fixed length string and the fitness are determined by the NN Instantiator & Simulator .

Two kinds of crossover operations are used in 2ndGA. One is similar to the uniform crossover where the weights/biases of the offsprings are randomly selected from their two parents. Another is called average crossover where the weights/biases of the offsprings are the weighted sum of their parents. There are also two kinds of mutation used in 2ndGA. One is similar to Montana's unbiased-mutate-weights where a randomly selected weight/bias is replaced by a randomly selected real value according to a two-side exponential distribution with a mean of *0.0* and a mean absolute value of *1.0*. Another is similar to the mutation used in evolutionary strategy where a randomly selected weight/bias is modified by adding some Gaussian distributed white noise.

There are two major reasons for using a genetic hill-climber as 2ndGA. One of the reasons is that GHC can overcome the competing conventions problems (different genotypes map into the same or equivalent phenotypes even though their genotypes are quite different) in some way [11]. Another reason is to reduce the training time, since the population size is small. The big drawback of using GHC is that it cannot guarantee the GA is producing a global search. One way to overcome this disadvantage is to make several runs instead of one run, although this may become time consuming.

The 2ndGA can also be used to fine-tune the weights and biases for the NNs produced by 2pGA. This can be done by using a larger population size and a larger maximum generation than these used in 2pGA.

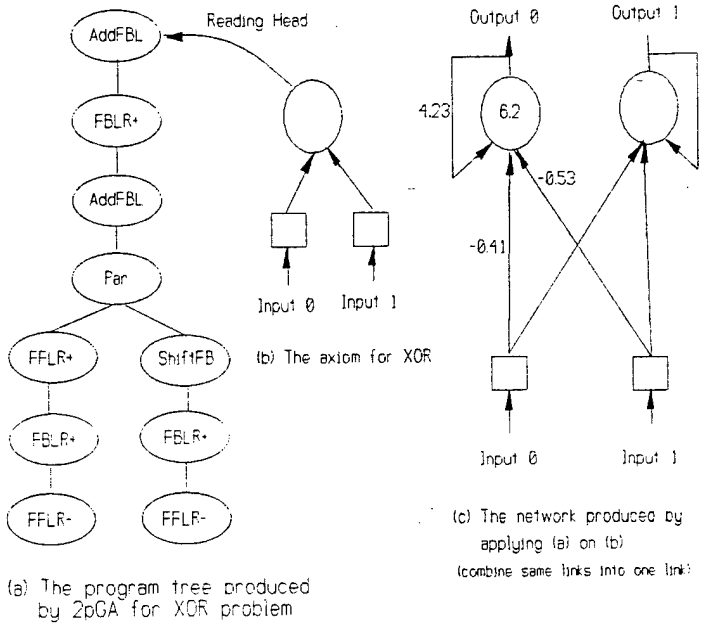


Figure 4 Evolved NN for XOR problem.

2.5 2pGA Experimental Results

In order to demonstrate the capability of 2pGA to evolve a wide variety of solutions to both linear and nonlinear problems, a series of experiments were performed including the traditional XOR problem, an amplitude modulation (AM) detector, various linear controllers, and finally nonlinear controllers. The results of these experiments are reported here.

Figure 4 shows the neural network which is a result of applying 2pGA to the XOR problem. Both the evolved tree describing the NN and the resulting NN itself are shown in the figure. The neurons in this example use the logsig function. The floating point value in the neuron means the bias of the neuron. The values near the links mean the weights of the network and unrelated weights and biases are not shown. The string representation of the produced networks is (*B+2+3B+1+2-0B-0B*). It can be noticed that some program symbols do not affect the functioning of the final neural network, for example the *1* output neuron is useless. The weights in this structure were then fine-tuned with a second GA to correctly solve the XOR problem.

The second experiment involved the evolution of an amplitude modulation (AM) detector, an inherently nonlinear function which is implemented using a diode or a thresholding device. The input signal of the NN is shown in **Figure 5** (the figures referred to here start on page 22) with output of the NN shown in **Figure 6**. The evolved NN is drawn in **Figure 7** and consists of 7 neurons, not all of which are needed. Because of the recurrence of the network, it takes 8 times to go through the NN to produce a result.

The third test case run on 2pGA consists of a hard-disk read/write head controller. This case is adapted from the Matlab Control System ToolBox. The original step response is drawn in **Figure 8**. 2pGA evolved a NN of 29 neurons to stabilize the plant. The controlled system's step response is drawn in **Figure 9**.

The next three test cases are built around compensation of first, second, and third order linear plants and were formulated to test the robustness of the procedure in the face of increasing levels of plant complexity. This was also a test of the sensitivity of the parameters associated with 2pGA. For example, the compensators evolved for these three plants were all developed using

the same parameters such as `convergence_time` of 2, a sample rate of 0.05 seconds, and 80 samples. The fitness function which was used to evaluate the evolved system is the well-known integral absolute error.

Case four is a first order plant with a single pole at $s = -1$. A neural network was evolved to put in the feedback path. **Figure 10** shows the step response with the dashed line representing the highly underdamped original response, and the solid line the result with the evolved 4-neuron, NN compensator. In case five, 2pGA evolved a 5-neuron NN to stabilize a second order plant with two poles at $s = -1$ and -2. **Figure 11** is the result for this 2-order, stable linear system, the dashed line being the uncompensated response and the solid line with the evolved NN in the feedback path. The final test in this sequence, case six, applied 2pGA to a stable, linear, third-order system. This system had poles at $s = -1, -2$, and -3. The results of this experiment were similar to the earlier two and are shown in **Figure 12**. Compensation of the third-order, linear plant required an evolved NN size of 10. Cases 4 through 6 demonstrate the ability of 2pGA to evolve compensator neural networks which can improve the time-domain response of stable linear systems, at least up to 3rd order. Two more experiments were run in order to test the ability of 2pGA to stabilize linear unstable plants.

Case 7 developed a stabilizing feedback NN for a first-order unstable plant with a pole at $s = +1$. The solid line of **Figure 13** shows the open-loop response, and the dashed line shows the step response when a 15-neuron feedback NN is used. Cases 8 and 9 developed 5-neuron, stabilizing feedback NNs for second and third order unstable systems. These systems were comprised of poles at $s = +1, -2$, and also -3 for the third order system. The open loop (solid line) and stabilized (dashed line) step responses are shown in **Figure 14** and **Figure 15**. A typical NN for stabilizing a 2nd-order, unstable, linear plant is shown in **Figure 16** along with its associated tree in **Figure 17**.

A final test to establish the robustness of the evolved NN stabilization technique is to use one of the evolved NNs as the stabilizer and evaluate its ability to stabilize the unstable linear system while the exact position of the pole in the RHP is varied. This was done with a second-order system consisting of a LH-pole at *2* and the-RH pole varied about its nominal, evolved NN value of *1*. Remember that the same fitness criterion is utilized for the evolution of the stabilizers for the unstable systems as was used for the compensation of the stable systems, namely integral absolute error. That is a constant fitness function throughout all of these experiments.

Figure 18 shows the value of the integral absolute error and demonstrates that for small variations about the nominal pole position, a single evolved NN is able to stabilize the system.

Encouraged by the results of the linear stable and unstable plant compensator experiments, the investigation was expanded to include nonlinear plants Case 10 is a 2nd-order linear stable plant followed by a nonlinear component, the function *exp (1.5)*. The results of this case are plotted in **Figure 19**. Since this is a nonlinear plant whose behavior is sensitive to input level, three different values of step input were used for evolution of the NN. These correspond to a step input of *1*

with a desired output of *0.5*, a step input of *2* with a desired output of *1*, and a step input of *4* with a desired output of *2*. It can clearly be seen that the 2pGA was able to evolve a single NN controller to improve the performance of a plant containing a nonlinearity in the forward path, again only using integral absolute error as a fitness function. The lines in **Figure 19** correspond to the response to the three different step inputs: *1* (dashed), *2* (dot-dash), and *4* (solid). This compensator only required 3 neurons and it is shown in **Figure 20**.

As a final set of experiments to demonstrate the ability of an evolved NN to control a nonlinear plant, the second example used by Goh [24] was implemented so as to have a basis for comparison. Goh's second example is a two state system which is nonlinear in the control:

$$\begin{aligned} \dot{x}_1 &= -x_1 + 0.5x_1^2 + 0.2x_2u \\ \dot{x}_2 &= 0.1x_2 + u + u^3 \end{aligned} \tag{4}$$

which is unstable. The control objective is to construct a state feedback controller such that the performance index

$$J = \frac{1}{2}\int_0^\infty \left(\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} + \boldsymbol{u}^T\boldsymbol{R}\boldsymbol{u}\right)dt \tag{5}$$

with the penalty that

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{6}$$

and $r = 1$.

For this final Case 11, the following parameters were used by the 2pGA program and all used a sample interval of (T_s) = 0.05 for the plant simulation.

	NN Structure GA	NN Coarse Weight Tuning GA	NN Fine Weight Tuning GA
Population	50	40	200
Generation	20	20	300
Crossover	0.6	0.6	0.6
Mutation	0.2	0.2	0.2

Since this is a regulator problem, the training data sets were generated by randomly creating 20 pairs of initial states (state vectors) in the range [-1, 1]. Each pair of initial states uses 100 sample points.

Figure 21, **Figure 22**, and **Figure 23** are the simulation results corresponding to initial states of (1,1), (0.5, 0.5), and (2,2), for the evolved NN of **Figure 24**.

While a little more complex than some of the earlier examples, the evolved NN is still quite small and therefore easy to implement as well as having fast execution. The evolved NN tree which generates the NN of **Figure 24** is shown in **Figure 25**.

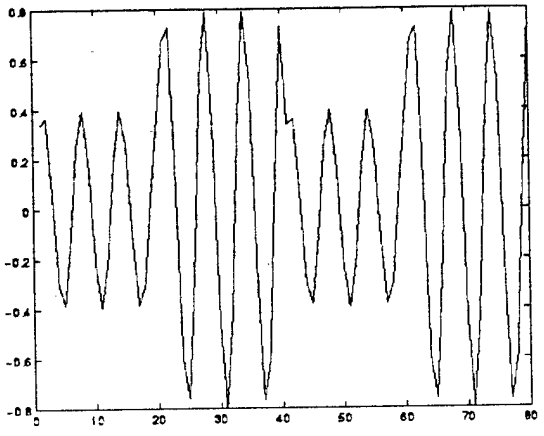


Figure 5 Amplitude modulated input signal.

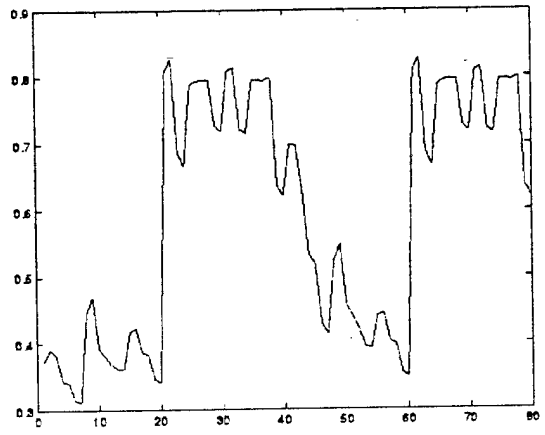


Figure 6 Detected AM signal.

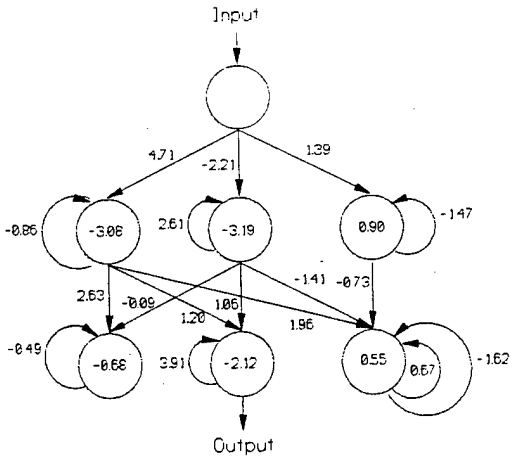


Figure 7 Evolved NN implementing AM detector.

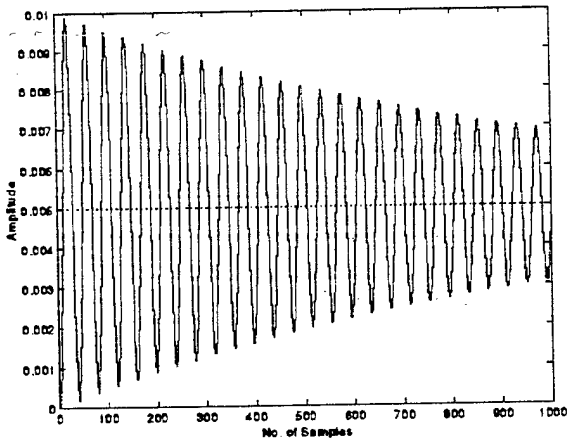


Figure 8 Hard disk head plant step response.

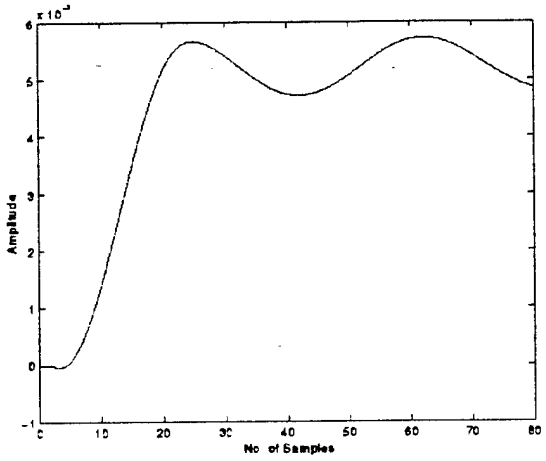


Figure 9 Step response of hard disk when stabilized using evolved NN.

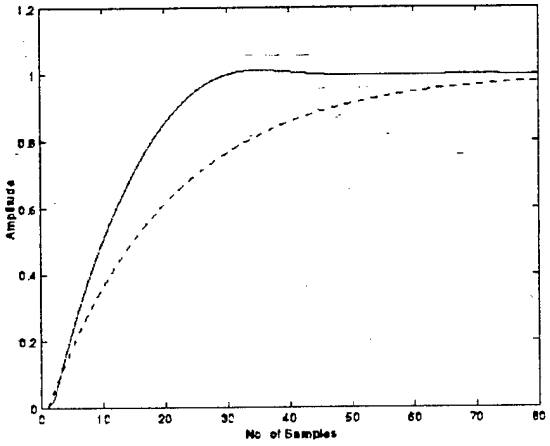


Figure 10 Step response of first order system with evolved NN compensator.

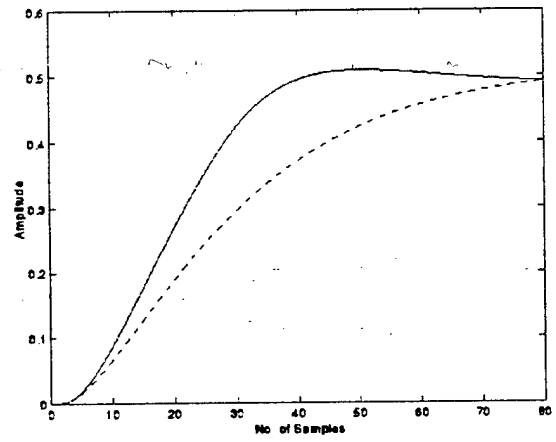


Figure 11 Step response of second order system with evolved NN compensator.

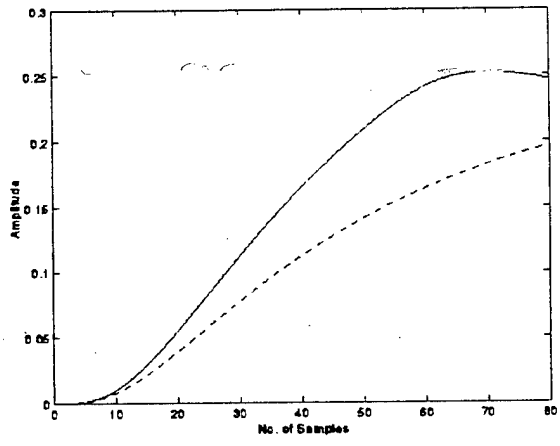


Figure 12 Step response of third order linear system with evolved NN compensator.

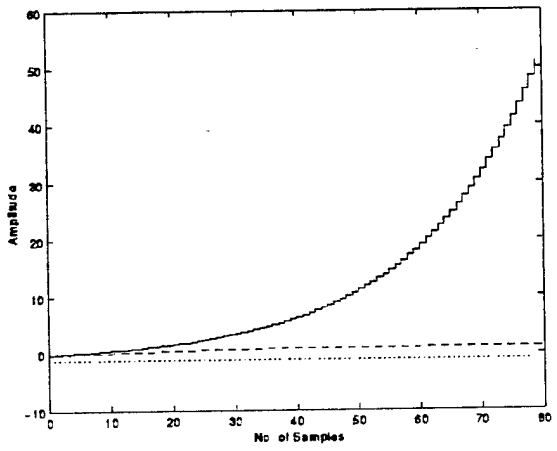


Figure 13 Open-loop and NN stabilized response of first order unstable linear system.

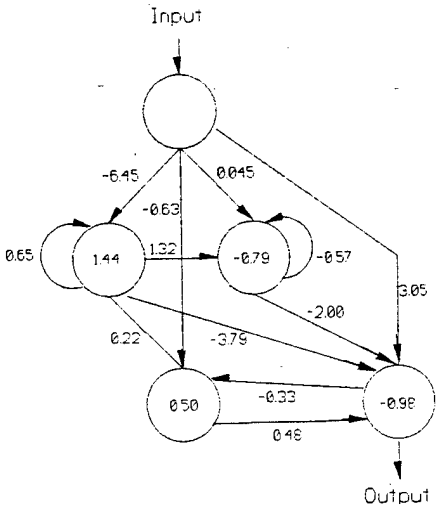


Figure 16 NN evolved for stabilization of unstable 2nd order system.

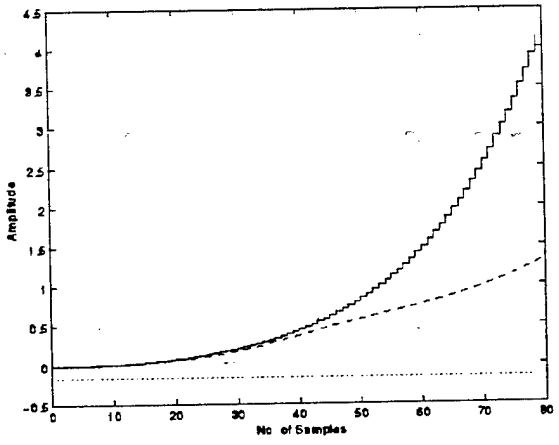


Figure 15 Open-loop and NN stabilized response of 3-rd order system with one pole in RH of S-plane.

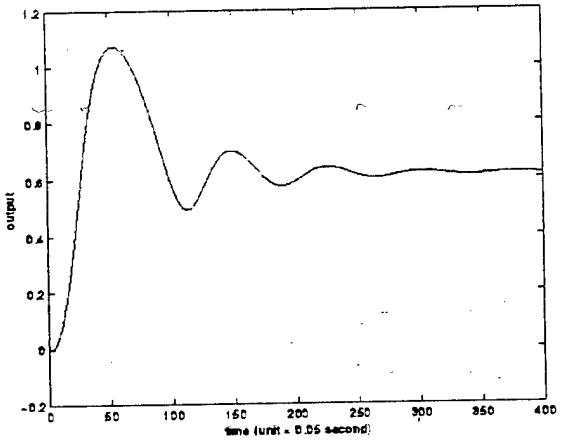


Figure 14 Step-response of NN compensated, 2nd order, unstable linear system.

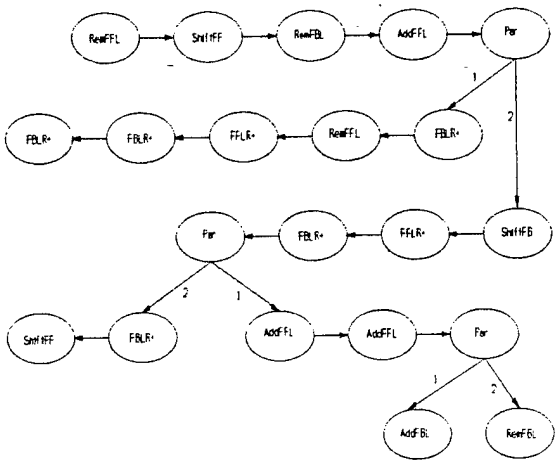


Figure 17 Tree structure of evolved NN for stabilization of 2nd order unstable plant.

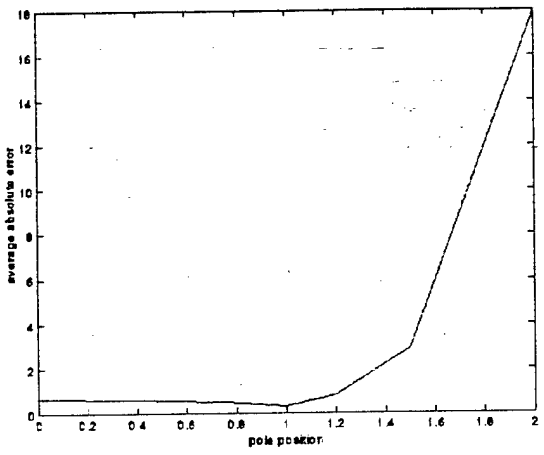


Figure 18 Value of integral absolute error fitness function of stabilized system as unstable pole is varied with stabilizing NN remaining fixed.

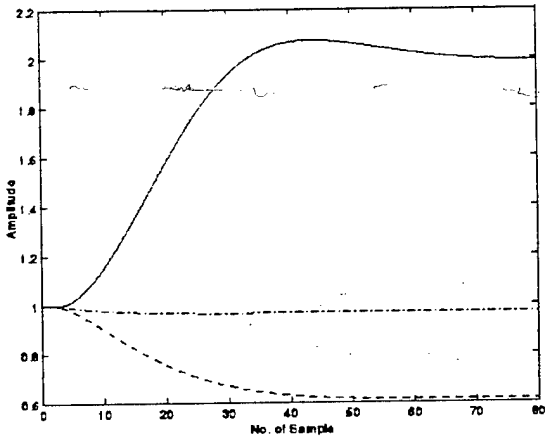


Figure 19 Compensated response of 2nd order plant with exponential function to 3 different step input values.

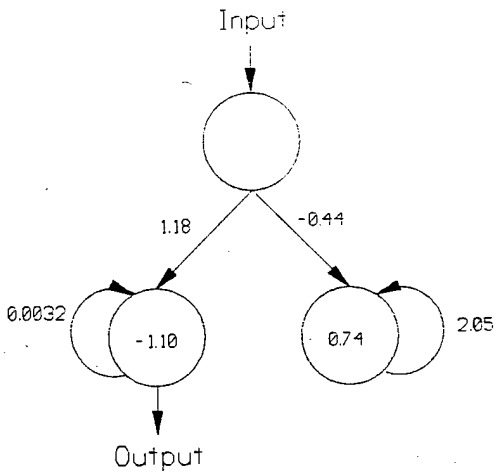


Figure 20 NN compensator for 2nd order plant with exponential function in forward path.

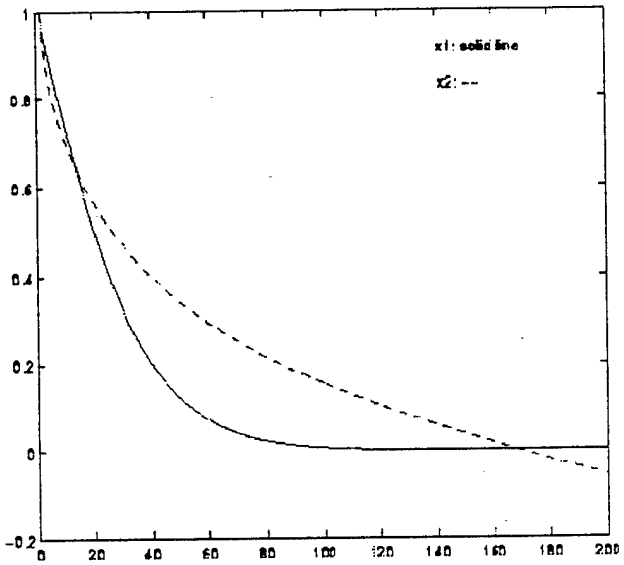


Figure 21 Goh's 2nd example controlled by evolved NN, $\mathbf{X}_0 = [1,1]$.

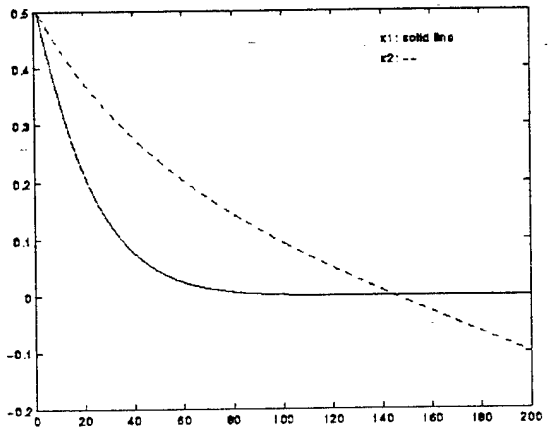


Figure 22 Goh's 2nd example controlled by evolved NN, $\mathbf{X}_0 = [0.5, 0.5]$.

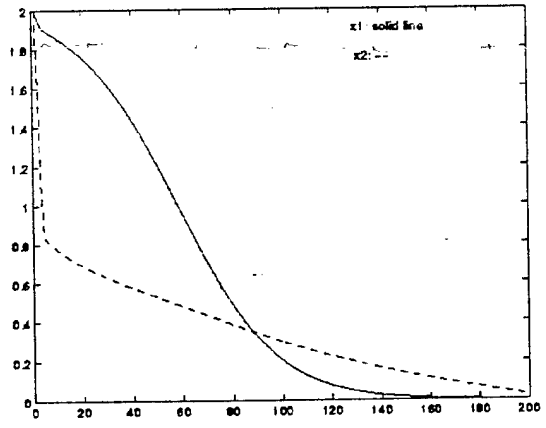


Figure 23 Goh's 2nd example controlled by evolved NN, $\mathbf{X}_0 = [2, 2]$.

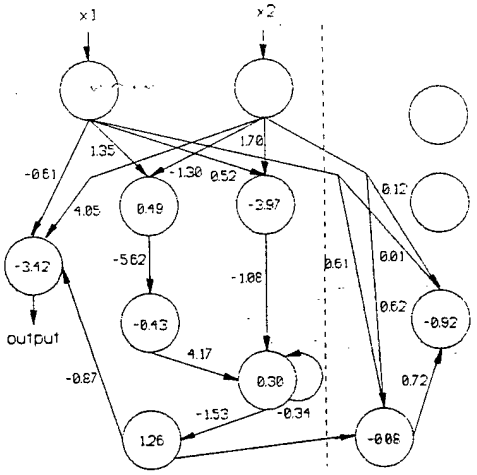


Figure 24 Evolved NN for control of Goh's 2nd example.

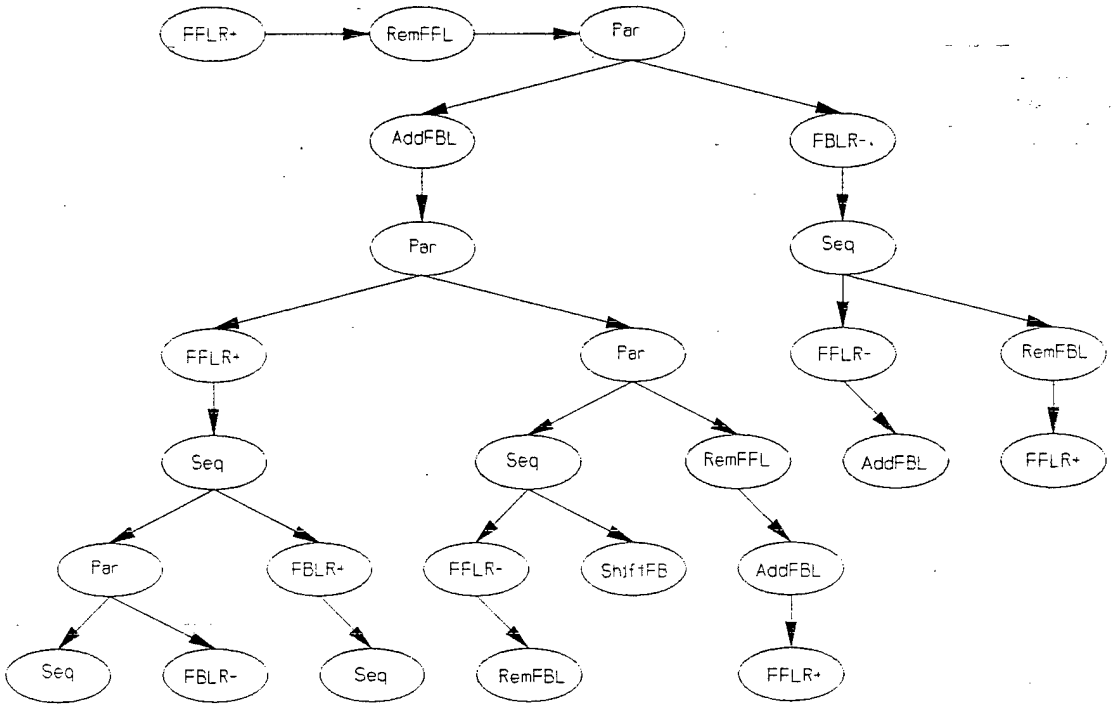


Figure 25 Evolved tree which generates the NN structure for controlling the plant of example 2 in Goh's paper.

3 Evaluating Neural Network Complexity Using GANNET

The second goal of this research is to empirically derive an upper bound on the computational resources required by a neural network to solve a problem of a given complexity. Specifically, it is desired to find the minimum number of neurons required to solve various problems of increasing complexity such that a relationship could be established between the two. Knowing the relationship between NN size and problem complexity allows one to estimate the cost of constructing a NN for a particular task, the time required to train the NN, and the minimum throughput time which is related to the bandwidth of the control system.

The discussion which follows includes two topics which are intertwined followed by the results of the experiment. The first is the binary-valued GA program developed at GMU to evolve amorphous recurrent networks (ARNN) and the modifications made to it to perform this study. The second is the various methods of measuring the complexity of a problem and the criteria used for selecting a suitable complexity measure. Complexity measures involving pattern recognition

problems were researched and detailed and a method of measuring problem complexity was selected, as were test problems. NNs were then evolved to solve selected problems with genetic pressure applied to evolve networks with the minimal number of neurons.

3.1 Genetic Algorithm, Neural NETwork (GANNET)

The GANNET (Genetic Algorithm, Neural NETwork) software, written in 1990 by Spofford [25] consists of approximately 4500 lines of ANSI **C** code. GANNET was developed to answer the question: 'Can a genetic algorithm be used to evolve all aspects of a neural network?' Indeed, GANNET was used by Spofford to evolve a full specification for NNs which solved four different problems:

1. Exclusive-OR(two-bit parity) problem
2. 26 letter recognition problem
3. 37 character recognition problem
4. 3-bit counter problem

GANNET evolves recurrent neural networks which use hard-limiting, bi-polar neurons to solve these problems. Each neuron accepts two or four inputs from either the training data sets or from another neuron. Problems are presented to GANNET in the form of input and output data sets, which GANNET is expected to correlate.

GANNET begins the genetic process by randomly generating a population of NNs based upon the values of the initialization parameters. Each NN in this population is evaluated using the fitness function, which evaluates how well a network solves the problem. The fitness function used by GANNET allows the experimenter to optimize the networks based on one or more of three criteria:

a) correct output for a given input,
b) minimal time to result convergence, and,
c) minimal network size.

A neural network is considered to converge if its output values remain consistent for two iterations. If a network is found which solves the problem, then GANNET terminates. If it does not find a solution which meets the termination criteria, it continues by mating the population and applying genetic operators to the mated pairs.

GANNET utilizes the genetic operators of mutation, crossover, neuron insertion/deletion (`resize`). After the genetic operators are applied, a new population is formed, and the evolutionary process begins again. This new population of NNs is evaluated in the same manner as the initial, randomly generated population was evaluated. The process continues until the desired fitness level is reached, or the maximum permitted number of epochs is reached.

The GANNET code was expanded from its original approximately 4500 lines of code to 6000 lines of code. The code was thoroughly commented and several operational and ancillary options were added. The genetic representation of the NN and its implementation was modified such that the crossover operator can be configured to be less destructive.

3.2 Operation of, and Upgrades to, GANNET

In an effort to make the GANNET program easier to understand and use for the NN complexity study, it was rewritten in a form that makes it self-similar to the process itself. The order of the items in the configuration file are now in the order that they are used, and variables are declared in the order that they are used in each function. A thorough review of the GANNET software was performed, which entailed adding numerous comments, replacing generic variable names with self-descriptive names, eliminating several minor programming errors, and incorporating improvements to GANNET's operation which became apparent after extended use. This analysis and rethinking of GANNET has lead to the addition of 14 configuration parameters, new procedures, and modifications to the current modes of operation. Previous modes of operation were maintained in most cases as legacy code. The format of the improved GANNET (version 2.0) is described in the text that follows, with remarks made about changes that have occurred.

This section begins with an overview of the configuration parameters that control the operation of an experiment within GANNET followed by a review of the genetic representation of the neural networks, a description of the operation of neural networks by GANNET, and finally, an in-depth examination of the genetic process itself.

3.3 GANNET Configuration Parameters

GANNET 2.0 has 47 configuration parameters, 41 of which affect the genetic process. The remaining six parameters are considered ancillary and are used to specify file names and how often statistics should be recorded. Of the 41 parameters which impact the genetic process, four are used to initialize the genetic code and GA process, five define the structure of the networks, four control how neuronal behavior occurs, and three are used to specify termination conditions.

The naming convention for the parameters has several patterns. Binary valued parameter names which name both behavior possibilities act in the first named format when the parameter is set to one, and in the second named format when set to zero. Binary parameters with only one format named behave in that format when the parameter is set to one. Alternatively, the one parameter which has three values, `conf.co_bit_word_or_neu`, behaves in the first named manner when set to zero, and the last named manner when set to two. Many of the parameters which accept a range of values act differently when set to zero. For instance, if `conf.stats_interval` is set to zero, no statistics will be recorded. Or, if `conf.co_mean_dist` is set to zero, then the crossover operator will be disabled.

Of the 39 parameters in GANNET 1.0, all but 6 have survived to version 2.0. `conf.save_memory`, which was set by appending a *-s* to the command line when starting a GANNET session, is not a part of version 2.0. This parameter allowed genetic code to be saved to the hard disk if the machine didn't have enough RAM available. Due to the addition of new population maintenance routines, all genetic code is maintained in RAM. Many of the parameters names have changed from version 1.0 to 2.0 in order to support the naming convention described above. Further, the configuration parameters of GANNET 1.0 did not follow a format which was easy to remember. Consequently, they were reordered in a format that follows program execution and hence makes them easier to use. `conf.biased` was deleted will be described in the neuronal behavior section. `conf.norm_cross`, and `conf.mutate_s`, which allowed for genetic crossover to take place, was deleted due to the fact that no useful results were obtained from its use. All crossover is now normal; the no longer existent genetic crossover mode is described in the crossover section. `conf.stable_per` and `conf.conv_per` were also deleted. These parameters held values used to control how many cycles the network was sent through before a valid output was accepted. These parameters were replaced by the new parameter, `conf.dead_states`. The parameter `conf.fit_wt_convrg_time` was also used as a part of this old method, and still exists, however it is just a place holder and should always be set to *0.0*. These changes are further described in the following sections.

`conf.inp_noise` was deleted as was its related code since little use could be found for adding input noise to the training patterns, and the maintenance of the code which performed this task was time consuming. The GANNET 2.0 configuration parameters are shown in the following table.

`exp_name`[1]	Suffix for configuration and other files
`data_name`[1,n]	Suffix for input and output training data files
`cur_gen`[1]	Current generation
`seed_create_gen_code`[2]	Initialization seed
`seed_evolve_loop`[2]	Seed before entering loop
`mean_init_net_size`[2]	Mean of initial number of neurons
`dev_init_net_size`[2]	Deviation in initial number of neurons
`max_net_size`	Maximum number of neurons
`pop_size`	Population size (must be even)
`dead_states`[n]	Number of dead states sent in to inputs after sequence
`net_ins`[3].	Number of inputs to network

net_outs[3]	Number of outputs from network
io_sets	Number of training data sets
neu_ins[3]	Number of inputs per neuron (2 or 4)
max_span_dist[3,n]	Number of neus (in either direction)across genome conns are allowed to span
sum_or_full_behav[4,n]	Neuronal behavior based on summation or full functionality
input_states[4,n]	Number of states (2 or 3) accepted by neuron input
behav_lookup[4,n]	Whether behavior should be looked up or generated from weights
fixed_outs[3]	Network outputs fixed or determined by bid
clear_neu_out[4]	Clear neurons' outputs at each cycle
uni_or_abs_out_pats	Network output patterns merely unique or required to be same as output file
sim_dif	Similarity vs. difference count weight for I/O fitness
bit_or_entire_out_pat	Count bits or pat
fit_wt_io	Fitness weight of input / output performance
fit_wt_convrg_time	Fitness weight of minimal convergence time (not operational)
fit_wt_net_size	Fitness weight of minimal net size
increment_fitness[n]	Fitness required to advance to next set of training patterns
increment_size[n]	Number of training patterns to add to training set at each increment
indiv_or_pop_sel[n]	Save nets by comparing against parents or population
max_age[0]	Maximum age of networks
top_heavy	Top heavy
quota_scale	Probability that high fitness nets will mate twice
best_interval[1,0]	Best net reporting interval
stats_interval[1,0]	Statistic reporting interval

stats_detail_or_sim[1,n]	Detailed or simple reports
term_gen[5,0]	Termination generation
term_fit[5,0]	Termination fitness
term_fit_mean_or_any[5,n]	Use mean or any net's fitness to check for termination
cloning[n]	Permit identical genotypes to mate with each other
co_norm_or_gen	Normal or genetic crossover
co_bit_word_or_neu[n]	Crossover Format: bit or network
co_mean_dist[0]	Mean distance to next c/o point per neuron
mu_bit_or_word[0]	Mutate at bit or word level
mu_mean_dist_in[0]	Mean distance to neuronal input connection mutation per net
mu_mean_dist_behav[0]	Mean distance to neuronal behavior mutation per net
mu_mean_dist_obid[0]	Mean distance to network output bid mutation per net
resize_prob[0]	Probability of resize

Key for Parameter Table:

0 = value of zero has special effect on behavior
1 = Ancillary parameter-- Doesn't affect genetic process
2 = Initialization parameter
3 = Structural parameter
4 = Neuronal Behavior Configuration
5 = Termination condition parameter
n = new parameter to version 2.0 of GANNET

3.4 Genetic Representation of Neural Networks

In order to understand GANNET's operation, one must first have an understanding of the genotypic representation of the neural networks which is distinctly different from that of 2pGA. GANNET uses twelve or more bytes per neuron in the genotype. The first twelve bytes are used to describe the neuron itself as shown in the following table.

Neuron Input Connections				Neuron Behavior	Network Output Bid
BYTES 1 & 2	BYTES 3 & 4	BYTES 5 & 6	BYTES 7 & 8	BYTES 9 & 10 (may use more depending on configuration)	BYTES 11 & 12

3.4.1 Neuronal Connectivity

The description of the connectivity from the environmental inputs and between neurons is maintained by the first eight bytes of the destination neurons. GANNET allows for either two-input or four-input neurons as configured by the `conf.neu_ins` parameter. If GANNET is configured for two-input neurons, then the last four of these bytes are unused. There were concerns about the crossover operator's effect on network topology.

De Jong [26] hypothesized that by limiting the distance across the genotype that neurons can connect to other neurons, the crossover operator will have less of an opportunity to be destructive to a network. Without this limitation, connections would be permitted to span the distance of the chromosome and connections would be broken more often when crossover occurs. Hence, the new version of GANNET allows for a maximum span distance to be specified. The value in `conf.max_span_dist` is the number of neurons away in either direction from the present neuron from which it may receive input. All neurons are also allowed to connect to any of the network inputs. For instance, if `conf.max_span_dist` is set to *0*, then the neuron may only take input from itself, or from any of the network inputs to the network. However, if this value is 4, then the neuron can take input from a total of nine neurons: from itself, and the eight neurons (four on each side) which are closest to it in genotypic space. If the present neuron is at the end or beginning of the genotype and is permitted to connect to neurons beyond the beginning or end, then it may connect to the neurons at the end or beginning of the genotype, respectively. Hence, the genotype is best thought of a seamless loop, rather than a string with a beginning and an end.

GANNET translates from 16-bit genotypic representation of input connections to phenotypic values of network inputs and based on the `conf.max_span_dist` and `conf.net_ins` parameters. First, it is determined whether there are more neurons in the network than the region of permitted connections, *i.e.*, it is determined whether the maximum permitted span distance is affecting the representation. The genotypic value is converted from its distributed format which ranges from 0 to 65535 down to a compact form which has a range that is equal to the number of network inputs and neurons to which it is permitted to connect. Next, it is determined whether the neuron input is connected to a network input or another neuron. This is determined by whether the compact value is less than the number of network inputs, in which case it is

connected to a network input. Otherwise, it is considered to be connected to a neuron. Because the input connection field is kept in distributed format ranging from 0 to 65535 during the genetic operations of crossover and mutation, it will still describe a valid phenotypic structure no matter what effect the operators have.

Previously, the maximum genotype length was 250 neurons long, and each connection was described by an absolute reference to a neuron in the genotype using eight bits. Furthermore, GANNET could not evolve recurrent neural networks which had more than 250 neurons or inputs.

3.4.2 Neuronal Behavior

GANNET's neurons accept binary- or ternary-valued inputs and produce binary-valued outputs. Savage [27] notes that for the relation f: $\{-1, 1\}^n \rightarrow \{-1,1\}^m$, which corresponds to a binary input, binary output neuron with n inputs and m outputs, there are $2^{(m(2^n))}$ distinct functions, or behaviors, that a neuron can have. For a neuron with n binary valued inputs, there are 2^n possible input combinations; because the output is also binary valued, there are $2^{(2^n)}$ possible behaviors resulting from each set of input combinations. This means that there are:

$2^{(2^2)} = 16$ distinct behaviors for two-input neurons with binary valued inputs
$2^{(2^4)} = 65{,}536$ distinct behaviors for four-input neurons with binary valued inputs

GANNET provides for three different methods of implementing a neuron's behavior:

a. Simple Neuron Model (SNM) using weights
b. Simple Neuron Model (SNM) using a lookup table
c. Full Neuronal Functionality

The `conf.sum_or_full_behav` option provides for two methods of interpreting the behavior description, either using summation of thresholds and products of input values and weights as specified by the Simple Neuron Model, or, providing full functionality and treating each neuron as if it was a fully configurable logic gate.

3.5 Simple Neuron Model (Summation) Neuronal Behavior

The Simple Neuron Model (SNM), as termed by Spofford, is a minor variation of the model proposed by McCulloch and Pitts [2]. Its difference lies in the squashing function wherein GANNET uses the *sgn (*)* function in place of the $\Theta(*)$ unit step function.

$$n_i\,(t+1) \;=\; sgn\left(\sum_j w_{ij} n_i(t) - \mu_i\right) \qquad (7)$$

It calculates neuronal behavior based upon the squashed summation of the negated threshold value, and the product of neuron input values and weight as described in Equation **(7)**. μ_I is the threshold (DC offset) term, w_{ij} are the weights, both of which determine the neuron's behavior. $n_j(t)$ are the values of the inputs to the neuron and $n_j(t+1)$ is the output of the neuron. The $sgn(*)$ function is hard-limiting, and results in values of -1 and +1 as described in Equation **(8)**.

$$sgn(*) = \begin{cases} 1 & \text{if } * \geq 0; \\ -1 & \text{otherwise} \end{cases} \qquad \textbf{(8)}$$

In GANNET, when using the SNM, the weights and threshold can take on one of eight integer values between -4 and +3, hence they each require three bits of storage. For both neuronal behavior modes, the permissible input values match those found at the output and are -1 or +1. For four input neurons, Spofford found that although the weights and bias consisted of fifteen bits which allowed for $2^{15} = 32{,}768$ different combinations, many of them generated the same behaviors. Spofford found through exhaustive numerical analysis that there are only 1882 unique behaviors that could be described by combinations of weight and bias values. These 1882 behaviors were found by indexing through all of the unique behaviors that could be generated by a hard-limiting neuron with integer weights and a threshold varying between -4 and +3. These unique behaviors are looked-up from the `behav4.data` file by GANNET. Similarly, for two input neurons, there are nine bits which would allow for $2^9 = 512$ different behaviors, yet only fourteen of them are unique and are listed in the `behav2.data` file. These behaviors are the linearly separable, or threshold, functions which are described at length by Hurst *et al.* [28] Linearly separable functions are those functions which have a hyperplane in the input space separating all of the $f(X) = 0$ outputs from the $f(X) = 1$ outputs. Hurst *et al.* also found that there are 104 linearly separable functions out of a total $2^{(2^3)} = 256$ possible boolean functions for gates with three inputs, and there are 325,262 linearly separable functions out of a total of $2^{(2^5)} = 4{,}294{,}967{,}296$ Boolean functions for gates with five inputs.

The `conf.behav_lookup` configuration option controls whether behavior is obtained from a lookup table where only one example of the unique behaviors appears (`behav_lookup` = *1*) or if the behavior is obtained using three bit encoded weights (`behav_lookup` = *0*). Typically, the lookup table provides the best performance. The use of this option in the weight mode will be described later. Note that this option is only applicable when using the SNM, that is, when `conf.sum_or_full_behav` = *1*.

With the SNM behavior restored for both two and four input neurons, GANNET maps from the sixteen-bit (genotypic) behavior description to the SNM (phenotypic) description by dividing the genotypic value by the number that provides each of the 1882 or fourteen behaviors an equal opportunity to be selected. It should be noted that the behavior files previously used the combo. prefix. The prefix was changed to `behav` to enhance clarity.

GANNET 1.0, when utilizing the SNM, allows for a second method of mapping from the genotypic behavior description to the phenotypic description for the four input neuron case. If

`conf.biased_sel_of_behav` is set, rather than giving all 1882 behaviors an equal opportunity to be chosen, GANNET used a biased search which favored the selection of some behaviors. This parameter was provided by Spofford, and it was not obvious why this method would be useful, hence it was deleted for version 2.0.

3.5.1 Full Neuronal Behavior

The SNM behavior mode only allows for fourteen out of a possible sixteen behaviors for a two-input neuron, and 1882 out of a possible 65536 behaviors for a four-input neuron. Because so many behaviors were left out by the SNM, GANNET was upgraded to support a second form of neuronal behavior. When `conf.sum_or_full_behav` is set to zero, the neurons operate with full functionality. If configured for full behavior, GANNET uses either the least significant four bits or all sixteen bits of the behavior genotype for two and four-input neurons, respectively, when calculating the output of a neuron.

$$\text{if } t_0 > \sum_{i=1}^{n} s_i w_i > t_1 \text{ then } x=1; \text{ otherwise } x=-1 \qquad (9)$$

Version 1.0 of GANNET did support full functionality for two-input neurons. The two-input neurons utilized two threshold values as described in Equation **(9)**. The `behav2.data.full` file was used, which has sixteen sets of weight and threshold values and allow for all sixteen behaviors listed in the following table to be generated. The linearly separable version of the two-input SNM was disabled, but the behavior file with the rest of the GANNET files was provided. Hence, in GANNET 2.0, the previously unused file `behav2.data` was reactivated for the SNM mode for two input neurons. `behav2.data.full` is no longer used to provide full functionality; instead, the sixteen possible behaviors are selected based upon the value of the four least significant bits in the behavior field of the genotype as described above.

Of the sixteen possible behaviors for binary valued, two-input neurons, fourteen can be generated by the SNM binary-valued neuron having two inputs. The Exclusive-OR, and Exclusive-NOR, which correspond to behaviors $0110_2 = 6_{10}$ and $1001_2 = 9_{10}$ cannot be generated by the SNM, hence there are only fourteen behaviors for the SNM mode. These two behaviors cannot be generated by the SNM because they are not linearly separable. An excellent review of linear separability and its relation to neurons is provided by Abu-Mostafa in [29]. The following table shows behaviors for a two-input neuron with binary valued input states

Behavior Number	Inputs	Output	Name
0	00 01 10 11	0 0 0 0	Ground

Behavior Number	Inputs	Output	Name
1	00 01 10 11	1 0 0 0	NAND
2	00 01 10 11	0 1 0 0	
3	00 01 10 11	1 1 0 0	Invert first input
4	00 01 10 11	0 0 1 0	
5	00 01 10 11	1 0 1 0	Invert second input
6	00 01 10 11	0 1 1 0	XOR
7	00 01 10 11	1 1 1 0	NOR
8	00 01 10 11	0 0 0 1	AND

Behavior Number	Inputs	Output	Name
9	00 01 10 11	1 0 0 1	XNOR
10	00 01 10 11	0 1 0 1	second input
11	00 01 10 11	1 1 0 1	
12	00 01 10 11	0 0 1 1	first input
13	00 01 10 11	1 0 1 1	
14	00 01 10 11	0 1 1 1	OR
15	00 01 10 11	1 1 1 1	V+

3.5.2 Ternary Valued Inputs to Neurons

When GANNET was first used to evolve recurrent networks which recognized regular languages, it became apparent that it would have to be upgraded to support three different input values to the neurons. The problem of sending in a variable length string serially to a network and having it

compute whether the string was a valid word for a given language cannot be easily done with neurons having binary-valued inputs. This is because there is no easy way to indicate when the end of the string has been reached. Ternary-valued inputs allow a third "dead state" can be sent into the environmental input of the network.

A third value for an input state was first implemented using the SNM with weights, and using the value *0* as the third input state. The lookup table couldn't be used for the SNM since it was set up to work with only the *-1* and *1* input states. Difficulty was encountered with evolving solutions to many of the language recognizers. Recalling the boost in problem solving ability that the fully functional neurons provided for binary-valued inputs, fully functional neurons with ternary-valued inputs were implemented. A new configuration parameter, `conf.input_states` was established along with modifications to the GANNET code to support this operation.

Savage's [27] equation can be further generalized for the relation f: $\{a_1, a_2...a_y\}^n \rightarrow \{b_1, b_2...b_x\}^m$ and shown through simple enumeration that there are $x^{(m(y^n))}$ distinct behaviors that a neuron can have. Hence, a neuron with four ternary-valued inputs and one binary-valued output could take on any one of $2^{(1 \cdot 3^4)} = 2^{81} = 2.41 \times 10^{24}$ behaviors, and would require $\log_2(2^{81}) = 81$ bits to represent the behavior. These 81 bits are stored in three 32 bit long integers in the **C** code for GANNET.

3.5.3 Determining Network Outputs

GANNET has two modes of operation for selecting which neurons will provide output from the network as selected by `conf.fixed_outs`. When using fixed outputs, GANNET takes its environmental outputs from neurons *1* through *n*, where *n* is the number of outputs. If GANNET is not configured for fixed outputs, the seventh and eighth bits are used by each neuron to 'bid' on the opportunity to become a network output. Outputs are selected by finding the neurons with bid values that are closest to the division lines. Division lines are pre-selected by dividing the search space into as many sections as there are outputs. An example is shown in **Figure 26**.

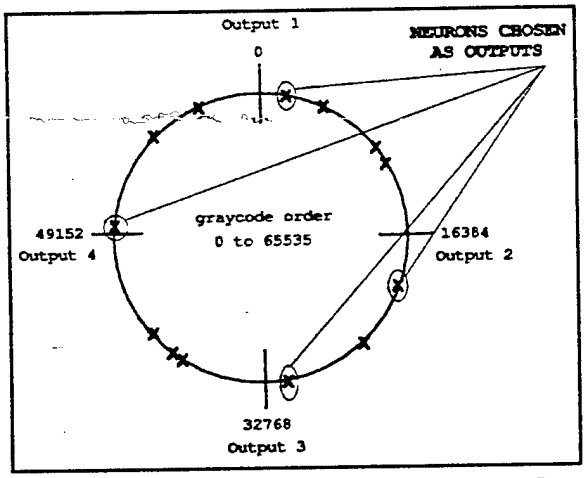


Figure 26 Determining output neurons. In this example, four neurons (out of a total of thirteen in the network) are selected as network outputs. Taken from [25].

3.6 Network Operation

In operating on a dataset, GANNET begins by setting the present state of every neuronal output to zero if indicated by `conf.clear_neu_out`. Then, it sets the value of each neuronal input to either the first bit from the environmental input or to the value of the neuron's previous output as specified by the connection structure for a given network. It then

calculates the output of each neuron based on these values. These two activities, setting inputs for each neuron and calculating the next output value of each neuron in a network, is a cycle. GANNET continues with cycles for each remaining bit in the input pattern. Then, the process continues with cycles for the number of dead states configured for the experiment as specified in `conf.dead_states`. The value *0* is sent in as input to the network if configured for ternary valued neuronal inputs. If the experiment is configured for binary valued inputs, then the value of the last bit sent in on the given environmental input is fed in for each dead state. After all of the dead states have been sent into the network, the value of the neurons selected to be environmental output neurons are taken as the outputs.

Previously, GANNET didn't allow for data to be fed serially into a network. It only accepted data in parallel. That is, an input pattern of some fixed number of bits would be repetitively applied to the environmental inputs of a network until an output was obtained. GANNET had an elaborate mechanism for evaluating the quality of a network's output in order to decide for how many cycles to apply the input pattern to the network. This mechanism evaluated whether the output value remained the same for a certain number of cycles (after an initial set of cycles had past) or varied. If it varied, then it was considered to be an invalid response and hence received a low fitness evaluation. This mechanism was eliminated and replaced with the code that sent in the configured number of dead states after each training pattern.

Several advantages were found in having the ability to send a pattern to a network in serial (which didn't preclude the use of two or more inputs, hence it could still be parallel at the same time it was serial). These advantages are outlined in following table.

Input Mode:	Parallel	Serial (and Parallel)
Can emulate a flip-flop	No	Yes
Inputs required to solve parity problem of n bits	n	1
Time steps required to solve parity problem of n bits	equation a function of n and the number of inputs per neuron; typically < n.	n
Can accept variable length input data	No	Yes
Minimal size network architecture	Feedforward	Recurrent

The notion that a feedforward network architecture is the only useful architecture for networks that can accept data only in parallel is expressed in the conjecture:

> Conjecture: If data required for a given pattern recognition problem is fed into a network in parallel, the minimal size network will always be feed-forward.

This conjecture is made based upon a Gedanken experiment in an effort to solve a parity problem of some number of bits. The parity problem is quite easily solved with two neurons constructed to form a flip-flop when the input is fed in serially. Hence, one might believe that it would be easiest to solve this problem in parallel by using neurons to act as "glue logic" to multiplex the data into the flip-flop. However, when one works to implement this multiplexer, it is obvious that the gates required to build the multiplexer exceed the requirements of a solution to the parity problem in a feedforward manner.

Because networks that accept data in parallel can't accept variable length data, and because it is desired to evaluate the number of neurons required by networks to solve various regular language recognition problems which require variable length data, it was decided to modify GANNET to allow it to accept its input in serial. GANNET's ability to accept more than one input at a time and hence accept data in parallel was retained. Each dataset is terminated by a carriage return in the training data file. The data is fed into the network starting with the first bit listed through the last bit before the carriage return.

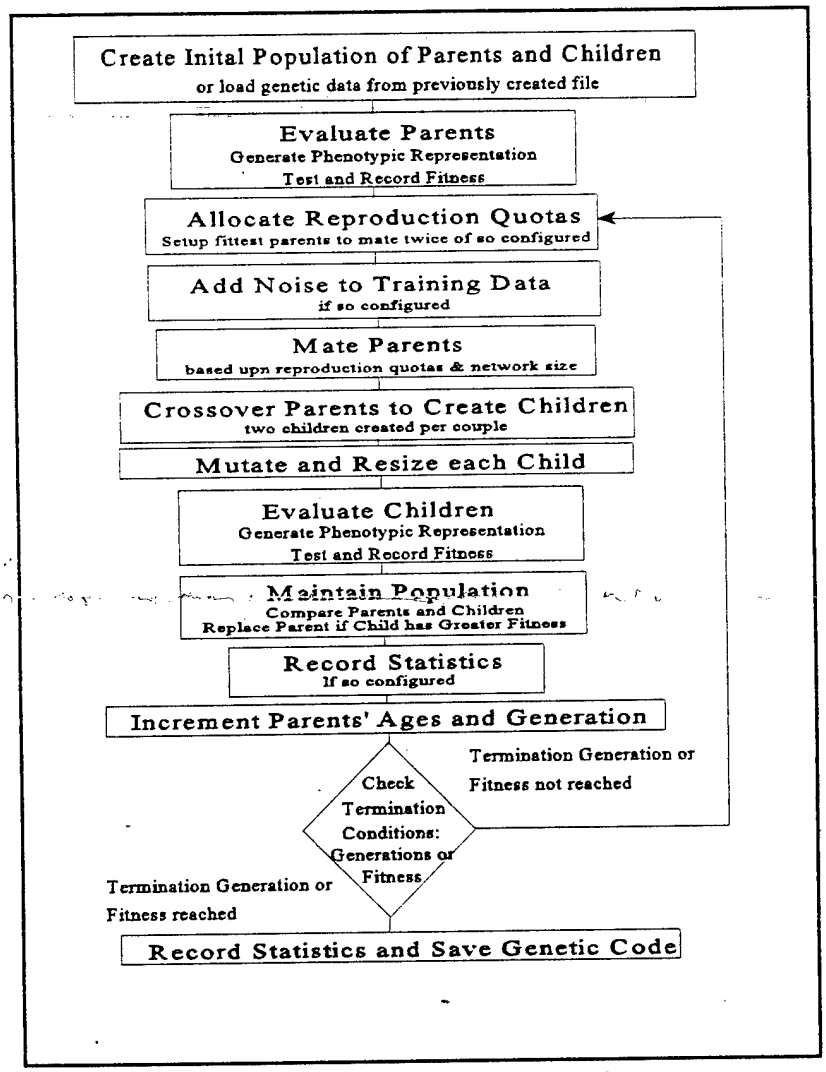


Figure 27. Flow chart of GANNET1.0 operation.

3.6.1 Genetic Process

The process which GANNET uses to generate neural networks was significantly modified. The original format of this process appears in **Figure 27**, and its modification appears in **Figure 28**. The order of two sub-processes were moved; previously, the allocation of reproduction quotas took place at the entry point to the loop. Now, reproduction quotas are allocated after the population is maintained. This change allows for three important sub-processes to occur consecutively. That is, the population is now evaluated, maintained, and prepared for mating without any interference. The other change

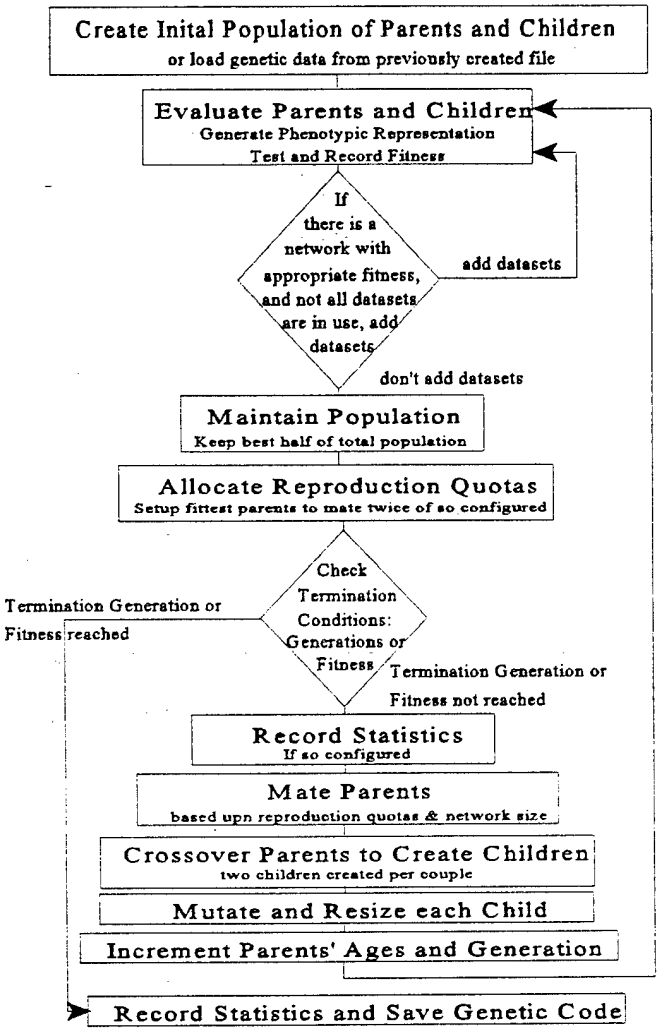


Figure 28 Flowchart of GANNET 2.0.

in sequence occurred relative to incrementing the parents' ages. Now, parents' ages are incremented immediately after children are generated so that when population maintenance takes place, it can use a more accurate count of the number of generations that a network has been around for deciding whether to keep it or not. However, the most significant change to the genetic process is that the point of entry to the loop was changed. Previously, after randomly generating the first population of NNs, GANNET would have to evaluate them (outside of the loop), and then enter the loop by mating the members of the population and reproducing. Now, GANNET randomly generates the NNs and goes to the point of the loop which evaluates the population. Even though the same function was called to perform both population evaluations, the new version is more concise since only one section of code is dedicated to evaluating the population. Furthermore, this modification allows for termination conditions to be checked before performing any reproduction. It is important to check the termination conditions before reproducing because it is important to see if the reproduction process has had anything to do with the success in solving the problem.

3.6.2 Initializing GANNET

GANNET begins the initialization process by setting the configuration parameters to default values, followed by an attempt to read the configuration file. This file has its suffix specified by the user in the command line when calling GANNET. If this file is not found, GANNET prompts the user for the configuration parameters and stores them to disk. GANNET then allocates memory for an entire population's parent and child genetic code and genetic dimensions based on `conf.pop_size`.

GANNET continues by reading in the training data from the input and output files which are held in files with prefixes of `input.`, and `output.`, and suffix as specified in the configuration file. If there is more or less training data than would be expected, an error message is written to the history file and GANNET terminates. Previously, GANNET would only terminate if there wasn't enough data. GANNET takes training data in as files containing *0*s and *1*s which represent the patterns to be learned. GANNET translates each value of *0* in these files as a *-1* for the network.

Next, the neuronal behavior files, `behav2.data` and `behav4.data`, which are used to translate behavior descriptions from genotype to phenotype, are loaded. The statistics and best network files are opened and prepared. GANNET then attempts to read in the genetic dimensions file. If the file exists, then it reads the file in, followed by the genetic code file, otherwise, GANNET randomly creates the genetic code.

If GANNET has to create an initial set of parents, it first seeds the random number generator with the value found in `conf.seed_create_gen_code`. Next, the first location where crossover should be toggled is randomly selected using a Poisson distribution at a rate prescribed by `conf.co_mean_dist`. The values for the entire population's dimensions are set to default values, except for the number of neurons. The number of neurons for each network is randomly selected using the value in `conf.mean_init_net_size` and then sampling the random variable specified by the uniform distribution across two times the value in `conf.dev_init_net_size`. This is done to select network sizes which vary in both directions around the specified mean size. After randomly selecting the size of each network in the initial population, values for the neural input connections, neural behavior, and network output bids are randomly selected across a uniform distribution in genetic space.

Previously, the section of code which calculates the location of next crossover point using a Poisson random variable didn't allow for the direct specification of an average rate. Rather, the value supplied by the user in `conf.co_mean_dist` was a floating point value that varied between *0.0* and *1.0* and had indeterminable units. This method of generating a Poisson random variable was also used by GANNET for selecting mutation points in the genetic code (`conf.mu_mean_dist_in`, `conf.mu_mean_dist_behav`, `conf.mu_mean_dist_obid`, `conf.mu_mean_dist_co`). In all of these places, it was replaced with a Poisson random variable generator which takes as input a direct specification of the average distance to the next incidence per network. This code was copied from *Numerical Recipes in C* [30] for my personal use in this experiment. The publicly available version of GANNET does not include this code, but simply a call to it.

The value for these four parameters indicates the average distance to the next bit of activity per network. For instance, a value of *0.5* for behavior mutation would provide, on average, two mutations of the behavior information per network; a value of *4* for crossover would apply the crossover operator on every fourth child.

After the genetic code is accessed, either by opening the file it is in and reading it into memory or by randomly generating it, the current generation is set from the value in `conf.cur_gen`. Next, the random number generator is seeded for the evolutionary process using the value in `conf.seed_evolve_loop`. From here, GANNET enters the evolutionary loop.

Previously, when GANNET allocated memory for the genetic code, it *a priori* assumed that each network would have up to 250 neurons in it. Now, GANNET allows for up to 65535 neurons, but it only takes up the amount of memory as specified by `conf.max_net_size`. Further, memory would be allocated for output bidding and for genetic crossover even if parameters which use these fields weren't selected. Now, GANNET only allocates memory for the output bidding part of the genotype if the memory will get used.

3.6.3 Evolutionary Loop

The evolutionary process involves many steps which continue until either of the termination conditions are met as described in **Figure 28**. GANNET's process begins by measuring the fitness of each child, promoting the children to replace their parents if the children have a higher fitness, allocating reproduction quotas, recording statistics, checking the termination conditions, mating the parents, applying the genetic operators to the mated couples to generate a new set of children. After the children are generated, the ages of each parent are incremented, as is the generation counter, and the loop returns to its beginning and starts again by evaluating the new children.

3.6.4 Fitness Evaluation

GANNET was designed with three fitness functions which can be scaled as desired to make up the total fitness function. The I/O fitness function measures how well a network produces outputs when presented with input patterns which are a part of its dataset. The reduce neuron fitness function provides a result based upon how many neurons there are in a network, with increasing fitness being allocated to those networks with the fewest number of neurons. The role that each of these measures plays in determining the fitness of a given network is prescribed by `conf.fit_wt_io`, and `conf.fit_wt_net_size`. The values of these parameters must sum to *1.0*.

GANNET begins evaluating a network by translating it from genotypic to phenotypic space as described in the representation section, above. Then, it enters a loop which tests each dataset on the network as described in the network operations section, above. Results from the test of each dataset in use are used by the fitness functions to come up with a final fitness function for each network.

The I/O fitness function in GANNET is configured by three binary valued parameters: `conf.uni_or_abs_out_pats`, `conf.sim_dif`, and `conf.bit_or_entire_out_pat`. GANNET can award I/O fitness points based either upon the absolute representation or based on free representation. If configured for absolute

representation (`conf.uni_or_abs_out_pats` = 0), GANNET compares each bit of the actual output pattern with each bit of the desired output pattern in the `output.suffix` file. Depending on the setting of the `conf.bit_or_entire_out_pat parameter`, GANNET either returns a score that represents the percentage of correct bits out of the total number of bits in the output training set (`conf.bit_or_entire_out_pat` = 1), or the percentage of correct patterns out of the total number of training patterns (`conf.bit_or_entire_out_pat` = 0).

When configured for free representation (`conf.uni_or_abs_out_pats` = 1), GANNET compares every possible pair of output patterns once. If the two desired patterns under comparison are the same, GANNET awards one point per actual bit (`conf.bit_or_entire_out_pat` = 1) or actual pattern (`conf.bit_or_entire_out_pat` = 0) which are the same. Alternatively, if the pair of desired patterns is different, then a point is awarded only if the actual patterns are different, regardless of the setting of `conf.bit_or_entire_out_pat`. After each pair of patterns is compared, the I/O fitness is calculated as a function of `conf.sim_dif`. The greater the value of this parameter, the more influence the presence or absence of desired similarities in pairs of output patterns have on the I/O fitness. If this parameter is equal to zero, then the fitness is based solely on the percentage of actual unique bits/patterns that exist out of the total number of desired unique bits/patterns.

3.6.5 Dataset Presentation

In order to regulate how much of the training data is tested against the networks, functionally was added to GANNET so that the datasets used to test each network could be incrementally added as networks were evolved which did a better job of solving the problem. When GANNET starts, it randomly selects enough datasets such that there are datasets that have two different outputs. Then, after each time the entire population has been evaluated, it checks to see if there is a network that has a high enough fitness that more datasets should be added to the problem. If so, and if there are datasets left to be enabled, the configured number of datasets are enabled and the population is re-evaluated. If the current population doesn't have a high enough fitness, then GANNET continues with the evolutionary process, entering the maintain population function.

3.6.6 Population Maintenance

After evaluating the child which just was created, a decision must be made about whether to keep the child or replace it with one of the parents. GANNET has three parameters which direct the method which GANNET uses to maintain the population of networks: `conf.indiv_or_pop_sel`, `conf.max_age`, and `conf.top_heavy`. `conf.indiv_or_pop_sel`, when set, uses the management scheme which was provided with GANNET 1.0. Each child is compared with the parent that first contributed genetic information to the child, and the child is replaced if that parent has a greater fitness, as long as the parent hasn't reached the maximum permitted age. If the program is configured to operate in the

top heavy mode as specified by a set bit in the `conf.top_heavy` parameter, and if either of the two parents has a fitness which is greater than the highest fitness found in the previous generation (and also greater than that of the child), then the child will get replaced by that parent. This replacement will happen regardless of the age of that parent. If both parents meet these conditions, then GANNET defaults to saving the first parent picked to be a part of the couple. The `top_heavy` parameter is good for maintaining the fittest members of the population if a maximum age is set. Alternatively, `conf.max_age` can be set to zero, and thus no network will be deleted just because it is too old, and `conf.top_heavy` will have no effect.

The `conf.indiv_or_pop_sel` is a new parameter which was added to allow for a new method of population maintenance for GANNET 2.0. When this parameter is cleared, decisions about saving parents are made after every child has had its fitness evaluated. The entire population of parents and children are sorted based upon fitness, and the fittest are kept as parents. If a network has an age equal to or older than `conf.max_age`, then its fitness is set to zero. The `conf.top_heavy` parameter has no effect when the entire population is sorted and maintained as one.

Nissen [31] defines two methods of population maintenance from the field of evolutionary strategies: The first consists of generating λ offspring from μ parents and putting both λ and μ phenotypes into competition with each other. This is named as $(\lambda+\mu)$-ES, and corresponds to the method of population maintenance just mentioned. In the second form, denoted by (λ,μ)-ES, all parents are eliminated at each generation and only children survive. This would occur if `conf.max_age` were set to *1*. A variant of the $(\lambda+\mu)$-ES form is GANNET's original method of population maintenance. The difference is that the child is compared with the parents that created it. This variant form might be denoted by $(\lambda_i+\mu_1)$-ES.

In the $(\lambda_i+\mu_1)$-ES method, parents are only compared with their children to make the decision about keeping or eliminating the parents. In itself, it was hypothesized that this would allow for some very unfit parents or children to remain in the population since there is no way to compare parents with other parents or children with other children at this stage of the game. It was further hypothesized that it would be better if the parents and the children could be compared with each other as a group when deciding which network descriptions should stay in the gene pool. Granted the new set of parents are sorted and given reproduction allocations before being mated, but there is no way to sort both populations of parents and children at the same time. It is believed that it would be better to use a $(\lambda+\mu)$-ES form.

Previously, there were two separate arrays for the parents and child code, and another two arrays for their dimensions. Also, GANNET operated by copying parents to be saved into child array, then moving a pointer from the old parent array to the child array. Now, one array is used to store both parents and children's code, and a second array holds all of their dimensions. This change was necessitated by the addition of the $(\lambda+\mu)$-ES population maintenance scheme.

3.6.7 Allocation of Reproduction Quotas

After deciding which networks should be saved as parents, the program allocates authorizations to mate and reproduce for each parent (0, 1, or 2 times), and returns the number of the parent which has highest fitness. Using Stochastic Universal Sampling as proposed by Baker [32] and described in Spofford's [25] thesis, authorization to reproduce is calculated as a function of the fitness of each network and the value of the `conf.quota_scale` configuration parameter. In essence, the `conf.quota_scale` parameter gives the fittest parents two allowances to reproduce and prohibits the least fit parents from having any chance to reproduce. A value of 0.0 gives all members of the population one chance of reproducing; whereas, a value of 1.0 gives the upper half of the population a second reproduction quota an average of 25% of the time, while reducing the lower half of the population as ranked in fitness to zero quota 25% of the time. The higher or lower the fitness, the greater the chance of having a deviation from one in the reproduction quota.

3.6.8 Termination Criteria

After checking to see if statistics should be recorded, two conditions are checked to see if the evolutionary process should be terminated. First, if the number of generations for the current execution of GANNET exceeds `conf.max_gen`, then GANNET terminates. For instance, GANNET can be restarted by calling it again from the operating system prompt, and it will reset its generation counter and run for another period of generations as specified by this parameter as long as the fitness termination condition doesn't take effect.

GANNET will also terminate if based upon the values of pre-existing parameter `conf.term_fit` and new parameter `conf.term_fit_mean_or_any`. If `conf.term_fit_mean_or_any` is set, GANNET operates in its original mode, that is, it terminates if the average fitness of the parents population reaches the fitness specified in `conf.term_fit`. However, if this new parameter is cleared, GANNET terminates if any network reaches the fitness specified in `conf.term_fit`. When GANNET terminates, it stops at this point in the evolutionary loop, and performs the actions described below in close up shop. GANNET continues the evolutionary cycle if neither of these conditions are met.

3.6.9 Statistics Recording

After allocating reproduction quotas, GANNET determines whether best network data and statistics should be recorded as specified by the `conf.best_interval` and `conf.stats_interval` parameters. These parameters specify how often, in generations, data is to be stored to the files that hold it. If these parameters are set to *0*, then the data will not be recorded. If it is found that either of these pieces of data is to be recorded at the present generation, then it is. Best network data consists of a phenotypic (structural) description of the network in the parents population with the highest fitness. Statistics data can take on one of two formats as specified by the new configuration parameter `conf.stats_detail_or_sim`.

Detailed statistics are recorded when this parameter is set, and are explained in Spofford's thesis. Simple statistics are recorded when this parameter is cleared, and consist of the generation number and the fitness of the parent with the highest fitness.

3.6.10 Mating

After generating noise in the input training patterns if so configured, GANNET begins the mating and reproduction process. This process is performed on two parents at a time. First, two networks are selected by the `mate()` function. Networks are paired as a function of the previously allocated reproduction quotas, their size, and the `conf.cloning` parameter.

The mating process begins by randomly selecting a network out of the population of parents and testing it to see if it has one or more reproduction allocations. If it does, then it becomes the first partner of the couple. If not, GANNET searches across the population until it finds a parent that does meet this requirement. Next, GANNET begins its search for a second partner by scanning the population for a parent with reproduction authorization which is the same size as the first partner. If one is found, then it becomes the second partner and the `mate()` function concludes so that the two selected partners can reproduce. Otherwise, GANNET continues searching for a second parent by expanding the desired network size conditions by one neuron in each direction after the entire population is searched without positive results at the present size restrictions.

In GANNET 1.0, it was found that if a parent was selected as a first partner when it had more than one authorization quota, the program favored selecting the same parent as its second partner. This result, named cloning, intuitively seems worthless, since crossing over two of the same genome will result in the same genome, and no new networks will be generated. A network can be selected for reproduction 0, 1, or 2 times per generation. After randomly selecting the first eligible parent, the original code looks for a second parent that has exactly the same number of neurons as the first parent. The code does not reject the second parent chosen if it is the same as the first parent. GANNET 2.0 was modified so that a parent can't reproduce with another instantiation of itself if `conf.cloning` is cleared. If `conf.cloning` is cleared, the program rejects the second parent chosen if it is the same as the first parent. When GANNET was first modified in this manner, it would get stuck in endless loops anytime two or four of the same parent were all that were remaining in the population. Subsequently, GANNET was modified to accept cloning if there are two or four parents left to be mated in the population.

3.6.11 Crossover

After two parents are selected to be mates, GANNET enters the `reproduce()` function and generates two new children networks by applying the genetic operators of crossover to the mates, and then applying mutation and resize to the children. GANNET performs crossover in one of two methods as configured by the new parameter `conf.co_bit_word_or_neu`. If this parameter is set to *0*, crossover is performed at the bit level, meaning that any bit in the genome can be a crossover point. This is the only format in which GANNET 1.0 would operate.

It was hypothesized that the crossover operator is generally destructive when working to crossover data that is internal to a neuron. Hence, a new crossover function was written for GANNET 2.0 which allows crossover to occur only at the divisions between each neuron in a network. Selected when `conf.co_bit_word_or_neu` is set to *2*, this mode preserves the genetic descriptions of the input connections, behaviors, and output bids of each neuron. Setting this parameter to *1* is an error, as this setting is reserved for future use.

3.6.12 Crossover Overview

GANNET utilizes *N*-point crossover, which means that there can be any number of crossover sites on the genotype. When generating the two new children in bit oriented crossover, the software incrementally examines the two mates' genotypes, and if either mate has a bit set in the extra array, crossover will be toggled on or off at the location of the set bit. Alternatively, if performing neuron oriented crossover, if any of the bits are set in the extra array which corresponds to a given neuron, then crossover will occur at the break between the given neuron and the next neuron. The crossover operator then copies each parent's array into one or the other children's array, depending upon whether crossover is on or off. The bits or neurons at which crossover is toggled are randomly generated using a Poisson random variable at each generation at a rate specified by `conf.co_mean_dist`.

For bit oriented crossover, GANNET crosses over genetic data in two separate segments which can be thought of as the biological analog of chromosomes. One chromosome, known as the input/behavior chromosome, holds the genetic data describing the four input connections to the neuron and behavior for each neuron in the network. The second chromosome, known as the output bidding chromosome, holds the genetic data for each neuron's bid to become an output neuron.

3.6.13 Crossover Process

GANNET begins the bit-oriented crossover process by comparing the number of neurons in each parent recording these values for future reference. This is important because when a child is made, there will be no genetic information available during the last part of the crossover process if the parents have differing sizes. Next, the crossover operator turns crossover off for both chromosomes. Hence, when the process starts, genetic data from the first parent is copied to the first child, and genetic data from the second parent is copied to the second child for both chromosomes. When crossover first gets toggled, genetic data will go from each parent to the opposite child.

The crossover operator continues by incrementing through both parents, one byte at a time, copying data from the parents to the children as indicated by the bitmask. The bitmask consists of one byte and is computed as a function of the toggle bits. The toggle bits are taken, one byte at a time, from either the crossover field of the genetic code or by testing to see if the next toggle bit (as randomly selected using a Poisson random variable) is in the range of the current byte. The

source of the crossover location data is prescribed by the setting of `conf.co_norm_or_gen`. For each bit in a byte of toggle bits, crossover is turned on if it was previously turned off, or turned off if it was previously on. There can be more than one bit set in the toggle bits byte. The actual change as to which parent is contributing genetic information to a child doesn't occur until the bit after the toggle bit which is set.

In order to compute the bitmask for the current byte, the program first determines whether the current byte being operated on is with the input/behavior chromosome or the output bidding chromosome, and whether crossover is on or off for said chromosome. Next, the program increments through the toggle bits byte and either sets the bits in the bitmask high or low depending upon whether crossover is on or off, respectively. While incrementing bit-by-bit through the toggle bits byte, when a high bit is found, crossover is toggled, and the remaining bits in the bitmask are set appropriately. This section of code completes by updating the semaphore which indicates the crossover status for either the input/behavior chromosome or the output bidding chromosome.

Both children have genetic data contributed to them in the manner listed above until the end of the shorter of the two parents is reached. At this point, the child which was receiving neuron input and behavior genetic code from the second parent is provided with an exact copy of the remaining data from the larger parent, while the other child receives no more data. If both parents are of equal length, then the procedure listed above is moot. With the actual crossover operation completed, the size of each child is loaded into the child info array, and the bred and age statistics are reset to zero.

The neuron oriented crossover process occurs in the same manner, except that the entire string of genetic code is treated as one chromosome. The value of the Poisson random variable generated is divided by the number of bits in each neuron such that a distance can be specified to the next neuron (as opposed to the next bit) where crossover should occur. Hence, the value for `conf.co_mean_dist` indicates the mean distance, in networks, to the next crossover position no matter what the setting of `conf.co_bit_word_or_neu`.

3.6.14 Crossover Details

The crossover function, which generates two new children by applying the crossover operator to those selected parents, had many flaws in the previous version of GANNET. A bug was found in the crossover operator. Crossover performed on a neuron's behavior field is performed two bytes at a time; however, the crossover routine works on each byte, one at a time. Spofford's original version was written to work on the lower byte first, then the upper byte. Included with the segment of code which is called to work on the lower byte is the code to perform the actual crossover. The result was that information which was kept in the upper byte wouldn't get crossed over if the bitmask indicated it should be. The crossover bug was rectified such that work on the upper byte gets performed before work on the lower byte does.

GANNET had a strange way of selecting which child is to be the longer child. Previously, when the end of the shorter parent was reached, GANNET would dump any surplus genetic code from the longer parent to the child which was receiving code from the second parent. The second parent is the parent that was selected second during the mating process. GANNET 2.0 was reprogrammed such that the longer child is the child who is currently receiving neuronal input and behavior code from the longer parent when genotypic information runs out from the shorter parent.

3.6.15 Mutation

GANNET's `mutate()` function allows for mutations to occur to the genetic code at the bit level or at the word level as prescribed by `conf.mutate_bit_or_word`. The genotype is broken up into four sections for the purpose of the mutation operator. Sections consist of the four two-byte neuronal input connections field, the two-byte neuronal behavior field, the two-byte output bid field, and the twelve-byte toggle bit field for genetic crossover. Each section has its own independent configuration parameter in which to specify the rate of mutation applied to it. However, if GANNET is configured to mutate at the word level, only the first three sections will perform in this manner; the code which holds toggle bits for genetic crossover will continue to operate at the bit level.

After performing crossover and generating two children, the `mutate()` function is called and operates on each child one at a time. It begins by testing to see if the next incidence of mutation for each section was randomly selected or not; if not, it randomly selects the locations of the next mutations. GANNET allows for all four sections to have unique mutation rates as specified by the `conf.mu_mean_dist_in`, `conf.mu_mean_dist_behav`, `conf.mu_mean_dist_obid`, and `conf.mu_mean_dist_co` parameters, which respectively correspond to the input connections, behavior, output bid and genetic crossover toggle point mutation rates. Each section is tested in turn to see if it should be mutated. For each section of the genotype, the randomly generated value of the variable which indicates where the next mutation should occur is tested to see if its pointing to a bit in the current child. If so, the number of the neuron which has the bit in it to be mutated is calculated, and the value of `conf.mutate_bit` is tested. If GANNET is configured for bit level mutation, then a two byte bitmask is created which has a high bit where the mutation is supposed to take place. This bitmask is bitwise XORed with the selected neuron's field, and the result is written back into that field. If GANNET is configured for word level mutation, then the two-byte field holding the selected bit is reselected by writing to it with a randomly chosen, uniformly distributed, two-byte value. After writing the mutated value to this field, the distance to the next bit or field to be mutated is randomly selected.

If word level mutation is selected, it doesn't apply to the genetic crossover section. The genetic crossover section will continue to operate in a bit oriented manner. Also, it should be noted that the input connection section only mutates one two-byte value, and not all four two-byte values, when word level mutation is selected.

3.6.16 Resize Operator

The `resize()` function randomly inserts or deletes a neuron out of a network with a uniformly distributed probability specified by `conf.resize_prob`. If the test to see if resizing should take place passes, then resize samples a Bernoulli random variable to select whether it should insert or delete a neuron. If the network has only one neuron, or has only as many neurons as network outputs, then a neuron deletion is prohibited. If the network has the maximum permitted number of neurons as specified by `conf.max_net_size`, then neuron insertion is prohibited.

As long as one of the conditions above is satisfied, the deletion or insertion process begins. The position for the action to occur in the genotype is randomly selected using a uniform distribution across the entire genotype, and each input is checked to see if it references a neuron that appears beyond the action point. If it does, then the reference is incremented or decremented as appropriate. If neuron deletion is chosen, this function continues by copying genotypic information over the neuron to be deleted from the neuron above it, and continues for each remaining neuron in the genotype. Alternatively, if insertion is chosen, it copies information into a new space above the top of the genotype from the last neuron, and continues such that a duplicate neuron is inserted at the point of action. Also, the information regarding the number of neurons in the network is updated.

The `resize()` operator can be disruptive to network performance in many ways. When using fixed outputs, the network outputs will be disrupted if the resize occurs in the first part of the genome where the neurons are that are providing output to the network. However, if `conf.fixed_outs` is cleared, the networks will be more robust under the resize operator, since only one network output has the same chance of being affected.

3.6.17 Close Up Shop and Other Files

After GANNET reaches one of its termination conditions, the `close_up_shop()` function is called. This function records the best network and statistics information for the current population of parents, and then stores all of the genetic code and genetic dimensions so that it can be used later.

A great amount of time is spent working with GANNET's configuration files when working with it. A configuration file setup utility was developed. This utility establishes a set of configuration files for testing the upgraded options and also provides a shell script for initiating these experiments. This utility has the prefix `cg_`, which is short for configuration [file] generation. Its suffix is the name of the set of training data to be used for the experiment such as xor, alp or par5bit. The name of the script file which executes each of these experiments is the name of the set of training data.

In order to generate accurate configuration files for GANNET 2.0 based upon configuration files used with GANNET 1.0, a utility was written which measured the mean value generated by

Spofford's Poisson random number generator. This file is called `poidevmean.c`, and prompts the user for a value between 0.0 and 1.0. It returns the mean, taken over 1000 trials, of Spofford's Poisson random number generator. This is how the rates for crossover and mutation were calculated for the configuration files which have a pound sign (#) in them. This symbol indicates that the configuration file is based upon an original experiment evaluated by Spofford.

4 Complexity Measures

The literature contains many instances of complexity measures being applied in the field of neural networks in the literature. Three distinct applications in the field were found:

1) measuring the complexity of the problem to be learned by the neural network [33]
2) measuring the complexity of the neural network structure itself [34]
3) measuring the complexity of the meta-problem of training the neural network [35]

Typically, a comparison has been made in the literature between two of the three applications. Hence, there are three comparisons which can be made:

1) a comparison between the complexity of training a neural network and the complexity of problems to be learned
2) a comparison between the complexity of the training algorithm and the structure of the network.
3) a comparison between the required structure of the network and the complexity of problems to be learned.

It is this last comparison which is addressed here. Applications of complexity to three different aspects of the neural network paradigm, and corresponding complexity measures.

Lindgren *et al.* [36] exemplify the first comparison, that of comparing the complexity of the meta-problem of training a neural network with a measure of the complexity of the problem to be learned. The complexity of the learning process was quantified by required learning time; Effective Measure Complexity (EMC) was used to quantify the complexity of selected finite automata. They utilized GAs to evolve discrete, recurrent neural networks for recognizing words in the regular languages associated with the finite automata. The evolutionary process used in Lindgren's paper was only the mutation operator. Limited results suggest that the learning time increases exponentially as a function of EMC.

Only one reference was found concerning the second comparison, that of the complexity of the meta-problem of the training algorithm versus the complexity of the network. Research performed in this area typically compared the training complexity with the problem complexity as well. Judd [35] was the only example of research found in this area.

	Complexity Aspects		
	Complexity of problem to be learned	Complexity of network structure	Complexity of network training algorithm
Complexity Measures	Effective Measure Complexity Kolmogorov Complexity Circuit Complexity Number of points to be distinguished (Hamming Distance)	Number of Neurons Interconnections Weights	Time/ Computational steps required: Computational Complexity (P vs. NP, etc.)

Many papers were found in the area of comparing neural network structures with the complexity of the problem solved by it. Abu-Mostafa and St. Jacques [33] and McEliece *et al.* [37] evaluate the information capacity of the Hopfield networks by comparing the maximum number of unique patterns that can be stored by a network with the network size. In another approach, Parberry [34] theoretically establishes upper and lower bounds for the number of neurons required by feedforward networks to solve problems of various complexities. The complexity of problems is measured using circuit complexity.

The research documented in this thesis is also concerned with establishing a relationship between problem complexity and network structural complexity. However, the concern here is with much more than just evaluating the information capacity of a network. The ability for neural networks to store information is only one aspect of what makes them useful; the other aspect is their ability to compute the answer to a problem. In this case, the problem is the same as chosen by Lindgren *et al.* [36], that of recognizing whether a given binary string is a part of a regular language. The problem complexity is also measured using EMC, and the number of neurons required to solve a problem is taken as a measure of network structural complexity.

4.1 Measuring Problem Complexity

Establishing an empirical relationship between problem complexity and the minimum network size required to recognize it will enable one to estimate, or at least upper bound, the amount of computational resources needed to solve problems which are equally complex. A thorough review of problem complexity measures was performed. Two distinct types of analytical problems solved by neural networks were found:

1) Mapping from input to output with numeric sequences: $\{0,1\}^N \rightarrow \{0,1\}^M$
2) Recognizing regular languages or performing some other "computation" on the data

The first type of problem, when solved by GANNET, takes patterns into the network in parallel, and sends the results out in parallel. It doesn't require state memory, and uses an environmental input for each bit in the input pattern. The second type of problem, that of recognizing regular languages, is best solved using a network which has state memory and which also has the word to be recognized serially clocked into one input. A solution will typically utilize a network with a recurrent architecture.

A concerted effort was first made towards finding an empirical relationship between the complexity of mapping problems and the required number of neurons. Problem complexity was to be measured utilizing circuit complexity. It was found, however, that evaluating the number of neurons required to solve a problem of specified circuit complexity is best performed theoretically, rather than empirically. Parberry [34] has found many such relations, theoretically establishing upper and lower bounds for the number of neurons required for various problems.

Because theoretical work had already answered the questions that were sought to be answered empirically, this research has been redirected toward measuring the empirical relationship between the complexity of recognizing regular languages and the required number of neurons. Effective Measure Complexity (EMC), which is defined in the next section, is utilized to evaluate the complexity of the regular languages used in this research.

4.2 Effective Measure Complexity

Effective Measure Complexity (EMC), as defined by Grassberger [38] and Lindgren *et al.* [36], measure the complexity of a finite automata based upon the Shannon information, or entropy, added as each successive word length is extended by one bit. For words in a language, the entropy of the words of a given length m is:

$$H_m = - \sum p_{\sigma_m}(\sigma_m) \log_2 p(\sigma_m) \text{ bits.} \qquad \textbf{(10)}$$

where $p(\sigma_m)$ is the probability that the finite automata will generate the word σ_m. For $m=0$, $H_0 = 0$ since there aren't any words of length 0.

Entropy is a good complexity measure in itself for some applications. Entropy measures the uncertainty of a random variable, or the number of bits needed on average to describe the random variable [39]. In the case of words of a given length, the entropy measures the average number of bits that would be required to describe the value of the word in a transmission based upon the probabilities of having to transmit each word. Because words of various lengths are recognized, and because they are recognized by evaluating each letter in the word one at a time, the

differential entropy must be considered. The information needed to predict s_m (the last letter of the word σ_m) if given the first m-1 bits (σ_{m-1}) is represented by:

$$\Delta h_m = H_m - H_{m-1} \tag{11}$$

for $m > 0$

Where m is the length of the word. It should be noted that $\Delta H \geq 0$, since H_m is non-decreasing with m, that is, $H_m \geq H_{m-1}$. Grassberger points out that "it is intuitively obvious that the uncertainty about s_{m+1} cannot increase if more and more of its predecessors are known." Hence, ΔH_1 is the uncertainty of the value of one bit for any word in the language since no information is provided by H_0. Next, the average amount by which the uncertainty of σ_m changes is considered:

$$\Delta^2 H_m = \Delta H_{m+1} - \Delta H_m \tag{12}$$

for $m > 0$

Where $\Delta^2 H_m$ is the average amount by which the uncertainty of s_{m+1} decreases due to knowledge of s_m, which when negated ($k_m = -\Delta^2 H_m$) can also be interpreted as the information in correlations of length m. EMC is then defined to be the sum of these average uncertainties for each correlation of length m:

$$\eta = -\sum_{m=0}^{\infty} m\Delta^2 H_m = k_{corr} \cdot m_{ave} = \lim_{m \to \infty} (H_m - m \cdot s_\mu) \tag{13}$$

EMC can also be written as the product of the total redundant information due to correlations for a given word $k_{corr} = \sum^{\infty} k_m$, and the average correlation length m_{ave}, or as the limit of the difference between the total entropy and the measure entropy of a given word as the length of that word goes to infinity. The measure entropy of a word is the product of its length and which is the Shannon entropy per letter. The sum $k_{corr} + s_\mu = 1$ represents the decomposition of the *a priori* information per symbol.

A C program was written to calculate the EMC for any stationary regular language. EMC was calculated for the languages listed in the following table. The table contains regular languages, their EMC, and the minimum number of neurons required to evolve a language recognizer ascending by EMC.

ID	Regular Language	EMC	Minimum Net Size Found	Notes
M	0*	0	1	Accepts words with zero or more 0s
N	(01)*	0	2	Accepts words with zero or more 01s
C	(0+1)*	0	1	Accepts any word
D	(0+01)*	0.251629	2	no 11s
E	(0+01+011)*	0.333333	2	no 111s
F	(0+01+011+0111)*	0.370951	4	no 1111s
B	(00+010+100)*	0.47	4	
a	(00+1)*	0.9183	2	no $10^{2n-1}1$
J	(000+1)*	1.46	2	no $10^{3n-1}1$ nor $10^{3n-2}1$
H	(00+11)*	1.5	2	
K	(0000+1)*	1.750	3	no $10^{4n-1}1$ nor $10^{4n-2}1$ nor $10^{4n-3}1$
L	(000+111)*	2.20	7	

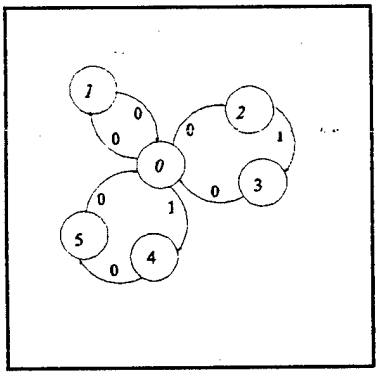


Figure 29: Finite Automata for Regular Language B: (00+010+100)*

Details of calculating EMC for languages A and D appear in [36], and an example of calculating language H appears in [38]. As an example, the EMC for language B will be calculated. This language can also be represented by the finite automata in **Figure 29**. The brunt of the task is to calculate the information in correlations of length m, which means calculating the probabilities of the sequences. For length zero, the calculation is straightforward:

$k_0 = 1\text{-}H_1$

$p(0) = 6/8$

$p(1) = 2/8$

$H_1 = -(6/8 \log_2 6/8 + 2/8 \log_2 2/8) = 0.8112$

$k_0 = 1\text{-}0.8113 = 0.1887$

For length $m = 1$, $k_1 = 2H_1 - H_2$; it becomes challenging since H_2 must be calculated. One must consider every possible route of length $m+1 = 2$ throughout the automata in generating $2^{m+1} = 4$ possible sequences. Further, those edges which lead to states that have less than the maximum number of edges of all states x must be considered x times. Since the maximum number of edges emanating from any one state is $x = 3$ (state 0), each second edge must be considered 3 times. The calculation is best done by considering each edge at a time. Edges are numbered by their start state and destination state separated by a comma. From state 0, there are three edges, all other states provide only one edge.

States, Edges, and Patterns of Depth 2 generated by the given edge for Regular Language B.

State	Edge	Possibilities
0	0,1	00, 00, 00
0	0,2	01, 01, 01
0	0,4	10, 10, 10
1	1,0	00, 01, 00
2	2,3	10, 10, 10
3	3,0	00, 00, 01
4	4,5	00, 00, 00
5	5,0	00, 00, 01

This gives total probabilities of observing the sub-sequences as listed below.

p(00)= 12/24 = 0.5000

p(01)= 4/24 = 0.2500

p(10)= 4/24 = 0.2500

p(11)= 0/24 = 0.0

The entropy calculation gives H_2= 1.5, and k_1 = 0.1226. This process of calculating information in correlations of length m continues until a desired level of accuracy for the value of EMC is achieved (or until one's computational resources are exhausted). EMC for this language was calculated on an Intel Paragon. Calculations were only possible up to m= 17, which took 23 hours, and resulted in the EMC of 0.47 for this automata, with an accuracy of +/-0.018. The results for each value appear in the table below of parameters required for calculating EMC of Regular Language B: (00+010+100)*.

m	H_m	ΔH_m	$\Delta^2 H_m$	H_m/m
-1	-1.00000000			
0	0.00000000	1.00000000	-0.18872188	
1	0.81127812	0.81127812	-0.12255625	0.81127812
2	1.5000000	0.68872188	-0.03628189	0.75000000
3	2.15243998	0.65243998	-0.00029480	0.71747999
4	2.80458517	0.65214519	-0.01226616	0.70114629
5	3.44446420	0.63987903	-0.00151344	0.68889284
6	4.08282978	0.63836559	-0.00079367	0.68047163
7	4.72040170	0.63757192	-0.00464728	0.67434310
8	5.35332634	0.63292464	-0.00127684	0.66916579
9	5.98497414	0.63164780	-0.00204545	0.66499713
10	6.61457649	0.62960235	-0.00216089	0.66145765
11	7.24201795	0.62744146	-0.00136488	0.65836527
12	7.86809453	0.62607658	-0.00199575	0.65567454
13	8.49217537	0.62408083	-0.00137919	0.65324426
14	9.11487701	0.62270164	-0.00143330	0.65106264
15	9.73614536	0.62126835	-0.00156295	0.64907636
16	10.35585075	0.61970539	-0.00115713	0.64724067
17	10.97439901	0.61854827	-0.00132486	0.64555288
18	11.59162242	0.61722340		0.64397902

EMC can be used to characterize other problems besides regular languages, including data of greater than one dimension. The interested reader is referred to Grassberger [38] for further details.

Note that EMC can only be calculated on stationary regular languages [38]. That is, the valid strings in a language must be translation independent, and cannot be described as having a special sequence occur only once in a string. Hence, the finite automaton used to generate test data allowed for all edges to be reached at some point in the sequence, repetitively; none of the finite automata can have edges which when traversed made them never again traversable. Examples of non-stationary finite automata appear in **Figure 30**.

4.3 Complexity Measure vs. Required Network Size Comparison Results

Neural Networks of minimal size were evolved to recognize each of the languages previously listed. Difficulty was encountered utilizing the resize functionality of GANNET, so network sizes were fixed for each trial. After a solution to a problem was found at a given size, a trial was made for the next descending size for 10,000 generations to validate that the minimum size had been found. Each of the networks utilized neurons with full functionality and three input states. The third input state was used to accept dead states, which were fed into the network after the word so that the entire word could be "analyzed" by the network.

While not yet definitive, there does appear to be a logarithmic relationship between the complexity of a regular expression which is recognized and the number of nurons required by a recurrent amorphous network as evolved by GANNET2. A graph of EMC *vs.* the smallest network size evolved appears in **Figure 31** for values of complexity greater than 1.4. This is a semilog plot with the log of the number of neurons being plotted against the regular language complexity. It can be seen that there is a linear relationship between these variables. The dashed line extrapolates the data to estimate that a regular language with a complexity of 2.4 would require approximately 10 neurons of this type to recognize the language. Values of complexity less than 1.4 are not included in this graph since they result in NNs of inconsistent and seemingly unrealistic sizes for their complexity. Additional experiments will be undertaken to evolve NNs for more complex languages to further support the conclusion of a logarithmic relationship between NN size and langauge complexity.

4.4 Further Work

As GANNET has been upgraded and utilized to recognize regular languages, several ideas have manifested themselves which may warrant further investigation.

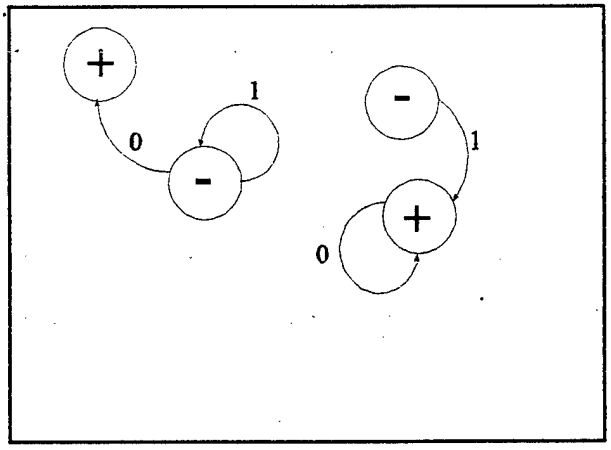


Figure 30 Two Examples of Non-Stationary Finite Automata: 1*0 and 10*. Minus(-) indicates start state; plus(+) indicates stop.

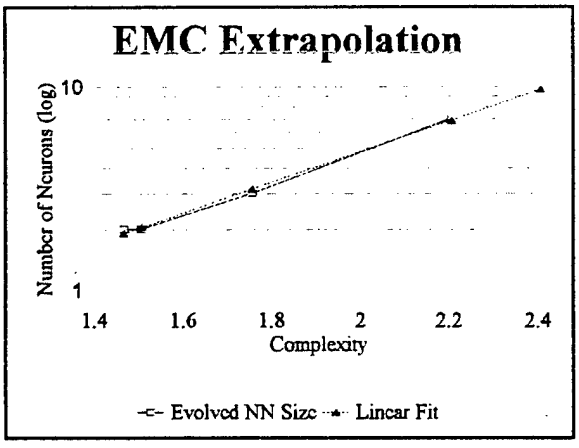


Figure 31 Effective Measure Complexity vs. Minimum Number of Neurons required to recognize test regular languages.

4.4.1 Crossover

GANNET should have a parameter that allows for crossover to occur at the byte/word level, as mutation does. It is hypothesized that for a neural network to be evolved using crossover, it would be best to separate the genotype into three separate chromosomes: neural input connections, neural behavior, and bid to become output, rather than one as it is now implemented. Furthermore, independent rates of crossover should be able to be selected for these three chromosomes, in the same way that the mutation operator allows for separate rates.

4.4.2 Resize

The `resize()` function works well, and has an easy to understand way to specify how often it should act. The only upgrade which might be necessary would be one that treated the connections at the point of action with more care. Presently, neurons receiving input from the neuron that is deleted now receive input from the neuron below it. It might make more sense to have neurons receiving input from the deleted neuron take input from one of the four (or two if GANNET is using two input neurons) inputs that were provided to the deleted neuron. When a neuron is inserted, that neuron's output doesn't connect to anything without another mutation taking place, so there is no sense in trying to make that process any cleaner.

4.4.3 Automatic Dataset Reduction

It was found that GANNET's ability to find solutions to problems was enhanced when starting a new attempt to solve a problem for which it previously had been unable to find one. There were many cases where GANNET wouldn't be able to find a solution to a problem in the first 5000 generations, but when it was restarted using the same set of evolved neurons on a new collection of randomly selected training datasets, it would find a solution within a few hundred generations. It is presumed that this enhancement is due to the fact that the set of active datasets used to train the network is reset. This observation can be thought of as GANNET being caught in a local minima of the solution space, and it being reset and climbing to the global minima by taking another path. Hence, a possible upgrade would be to subtract all or some datasets from the current collection of datasets if GANNET hasn't found a network with a higher fitness after some configurable number of generations.

5 Summary

An approach called two-phase genetic algorithm (2pGA) has been developed which has been used to evolve modular, recurrent NNs. It combines two GAs, the first is used to evolve a near optimal architecture of NNs for specific problems, then a second GA is used to fine-tune the weights and biases of the NN structure produced by the first GA. The first GA uses production-rule based techniques to encode architectures of arbitrary NNs. The second GA uses a genetic hill-climber to evolve the weights and biases of the evolved NN structure. The fitness of the second GA is used to determine the fitness of the individuals in the first GA. It is felt that not only is the 2pGA

biologically plausible, but has been demonstrated in this research to enable one to achieve excellent results in evolving NNs for controlling linear and nonlinear plants up to 3rd order, stabilizing unstable plants up to 3rd order, as well as finding the optimal control for a nonlinear regulator problem. In order to demonstrate the capability of 2pGA to evolve a wide variety of solutions in addition to linear and nonlinear control problems, a series of experiments were performed including the traditional XOR problem and an amplitude modulation (AM) detector.

While not yet definitive, experiments performed under this grant show that there appears to be a logarithmic relationship between the complexity of a language represented by a regular expression and the size of a recurrent neural network which recognize it. The size of the recurrent NN is measured by the minimum number of neurons required by a recurrent amorphous network as evolved by GANNET2. Additional experiments are being performed to extend the region of evolved data to improve our confidence in this conclusion.

References

[1] J. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge: MIT Press, 1992.

[2] W. S. McCulloch, and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics* Vol. 5, pp. 115-133, 1943.

[3] C. Darwin, *The Origin of Species*. London: John Murray, 1859.

[4] K. A. De Jong, "Genetic Algorithms Are NOT Function Optimizers," *Proceedings of Foundations of Genetic Algorithms Workshop.*, Darrell Whitley, ed., pp. 5-17, Morgan Kaufmann, 1993.

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

[6] D. J. Montana, "Automated Parameter Tuning for Interpretation of Synthetic Images," *Handbook of Genetic Algorithms.* Lawrence Davis, Ed., New York:Van Nostrand Reinhold, 1991.

[7] D. Whitley "Applying Genetic Algorithms to Neural Network Learning," *Proceedings of the Seventh Conference of the Society of Artificial Intelligence and Simulation of Behavior.* pp. 137-144, Sussex, England:Pitman Publishing, 1989.

[8] G. Miller, P. Todd, and S. Hegde, "Designing Neural Networks using Genetic Algorithms," *Proceedings of the Third International Conference on Genetic Algorithms.* pp. 379-384, San Mateo, CA:Morgan Kaufmann Publishers, 1989.

[9] S. Harp and T. Samad, "Genetic Synthesis of Neural Network Architecture," *Handbook of Genetic Algorithms.* New York:Van Nostrand Reinhold, 1991.

[10] I. P. Harvey, P. Husbands, and D. Cliff, "Issues in Evolutionary Robotics," *The Second International Conference on Simulation of Adaptive Behaviour.* J. A. Meyer, H. Roiblat, and S. Wilson, Eds., Cambridge, MA:MIT Press Bradford Books, 1993.

[11.] J. D. Schaffer etc., "Combinations of genetic algorithms and neural networks: A survey of the state of the art," *Proceedings of the Workshop on CGANN'92*, The IEEE Computer Society Press.

[12.] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," *Proceedings of Eleventh International Joint Conference on Artificial Intelligence*, pp.762-767, San Mateo, CA:Morgan Kaufmann, 1989.

[13.] K. J. Hintz and J. J. Spofford, "Evolving a neural network,", *Proceedings 5th IEEE International symposium on Intelligent Control,* pp.479-484, Los Alamitos, Ca, U.S.A. IEEE Computer Society Press. Vol-I.

[14.] V. Maniezzo, "Genetic evolution of the topology and weight distribution of neural networks," *IEEE Trans. on Neural Networks*, Vol.5, No.1, Jan. 1994.

[15.] W. M. Rudnick, *Genetic algorithms and fitness variance with an application to the automated design of artificial neural networks.* PhD dissertation, Oregon Graduate Institute of Science & Technology, 1992.

[16.] S. A. Harp, T. Samad, and A. Guha, "Towards the genetic synthesis of neural networks," In J. D. Shaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms,* pp.360-369, San Mateo, CA: Morgan Kaufmann, 1989.

[17.] E. J. W. Boers and H. Kuiper, "Biological metaphors and the design of modular artificial neural networks," Master's thesis, Leiden University, Netherlands, 1992..

[18.] F. Gruau, *Neural network synthesis using cellular encoding and the genetic algorithm.* PhD thesis, 1994. Laboratoire de l'Informatique du Parallélisme.

[19.] C. Jacob and J. Rehder, "Evolution of neural net architectures by a hierarchical grammar-based genetic system," *ICNNGA'93.*

[20.] P. Prusinkiewicz and J. Hanan, *Lindenmayer systems, fractals, and plants*. Springer-Verlag, 1989.

[21.] J. R. Koza, *Genetic programming -- on the programming of computers by means of natural selection*. MIT Press, 1992.

[22.] B. L. M. Happel and J. M. J. Murre, "Design and evolution of modular neural network architectures," *Neural Networks*, Vol.7, Nos.6/7, pp.985-1004, 1994.

[23.] T. Bäck and H.-P. Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evolutionary Computation*, Volume 1, Number 1, pp.1-23, 1993.

[24.] Goh, C. J., "On the Nonlinear Optimal Regulator Problem," Auto 29:3-L.

[25] J. J. Spofford, *Evolving Neural Networks with a Genetic Algorithm, Master's Thesis.* Fairfax, VA:George Mason University, 1990.

[26] K. A. De Jong, Personal Communication, February 1993.

[27.] J. Savage, *The Complexity of Computing*, Malabar, Florida: Robert E. Krieger Publishing Company, 1976.

[28.] S. L. Hurst, D. M. Miller, and J. C. Muzio, *Spectral Techniques in Digital Logic*. London: Academic Press, 1985.

[29] Y. S. Abu-Mostafa, "Information Theory, Complexity, and Neural Networks," in *IEEE Communications Magazine*, pp. 25-28, 81, November 1989.

[30] W. H. Press, S. a. Teukolsky, W. T. Vetterling, and B.P. Flannery, *Numerical Recipes in C, Second Edition* Cambridge: Cambridge Univeristy Press, 1992.

[31] V. Nissen, "Solving the Quadratic Assignment Problem with Clues from Nature," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 66-72, 1994.

[32] J. E. Baker, "Reducing Bias and Inefficiency in the Selection Algorithm," *Proceedings of Second International Conference on Genetic Algorithms*, MIT, Cambridge, MA, pp. 14-21, July 28-31, 1987.

[33] Y. S. Abu-Mostafa and J. St. Jacques, "Information Capacity of the Hopfield Model," in *IEEE Transactions on Information Theory*, vol. IT-31, No. 4, July 1985, pp. 461-464.

[34] I. Parberry, *Circuit Complexity and Neural Networks*, Cambridge, MA: MIT Press, 1994.

[35.] S. Judd, "On the Complexity of Loading Shallow Neural Networks," in *Journal of Complexity*, Vol. 4, 1988, pp. 177-192.

[36] K. Lindgren, a. Nilsson, M. G.. Nordahl, I. Rade.,"Regular Language Inference Using Evolving Neural Networks," in COGANN-92: *Proceedings of Combinations of Genetic Algorithms and Neural Networks*, IEEE, *1992*.

[37.] R. J. McEliece, E. C. Posner, E. R. Rodemich, and S. S. Venkatesh, "The Capacity of the Hopfield Associative Memory," in *IEEE Transactions on Information Theory*, vol. IT-33, July 1987 pp. 461-482.

[38] P. Grassberger, "Toward a Quantitative Theory of Self-Generated Complexity," in *International Journal of Theoretical Physics*, Vol. 25, No. 9, 1986 pp. 907-938.

[39] T. Cover, J. Thomas, *Elements of Information Theory*, New York: John Wiley & Sons, Inc., 1991.