

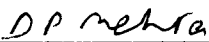
To the Graduate Council:

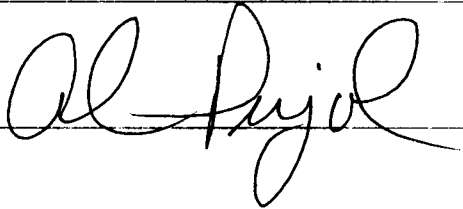
I am submitting herewith a thesis written by Sean Lane Fuller entitled "Using a Cooperative Genetic Algorithm Population to Train Hidden Neural Network Weights." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.



Dr. Bruce Whitehead, Major Professor

We have read this thesis
and recommend its acceptance:





Accepted for the Council:



Associate Vice Chancellor
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master of Science degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature

Sean Tuller

Date

4-22-1994

Using a Cooperative Genetic Algorithm Population to Train Hidden Neural Network Weights

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Sean Lane Fuller

May 1993

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Bruce Whitehead, for his brilliant insight and invaluable guidance during the preparation of this thesis. I would also like to thank my other committee members, Dr. Alfonso Pujol and Dr. Dinesh Mehta. The contributions of the committee members resulted in a much better thesis for which I am grateful. I would also like to thank the United States Air Force and the management of OAO Corporation at Arnold Engineering Development Center for encouraging my graduate studies. I want to thank my friend, Tim Choate, for introducing me to neural networks and genetic algorithms. I would like to thank my coworkers for their encouragement. Lastly, I would like to thank my wife, Terri, for her support and encouragement, and for putting up with my single-minded devotion to this research.

ABSTRACT

In a cooperative genetic neural network the genetic algorithm optimizes the hidden weights in a 3 layer feed-forward neural network. Perceptron training optimizes the output layer of the network and the trained weights of the output layer are used as a measure of fitness for each weight vector in the hidden layer. The hidden weight vectors compete through genetic evolution to produce an optimal transformation for the output layer. Common weight vectors in the output layer provide a sharing function between the hidden weight vectors. This produces a cooperative genetic environment that allows the total population to find the optimal hidden weight matrices. This approach is simulated using different benchmark datasets. This neural network and genetic algorithm hybrid achieves performance during testing comparable to the backpropagation algorithm on these datasets in terms of accuracy and elapsed time.

TABLE OF CONTENTS

CHAPTER	PAGE
I. INTRODUCTION	1
A. Genetic Algorithm and Neural Network Hybrids	1
II. LITERATURE REVIEW	3
A. Genetic Algorithms	3
A.1 Concepts	3
A.2 Mathematical Foundations	7
A.3 Niching and Speciation	11
A.4 Analysis	14
B. Neural Networks	15
B.1 Concepts	15
B.2 Mathematical Foundations	16
B.3 Backpropagation	18
B.4 Analysis	22
C. Genetic Neural Networks	23
C.1 Concepts	23
C.2 Optimizing Network Topology	23
C.3 Optimizing Networks Weights	25
C.4 Analysis	27
III. COOPERATIVE GENETIC NEURAL NETWORK TRAINING	28
A. System Overview	28
B. Genetic Encoding	28
C. Output Layer Training and Fitness Determination	30
D. Internal Genetic Algorithm	30
E. Justification of Technique	34
F. Summary	44
IV. SIMULATION RESULTS AND CONCLUSIONS	46
A. Problem Selection and Benchmarks	46
B. Simulation Platform	47
C. General Parameter Settings	47
D. Automatically Created Data	48
E. Tic-tac-toe Endgame Database	49
F. Pima Indians Diabetes Database	50
G. Breast Cancer Databases	52
H. Predicting Chaotic Time Series	54
I. Conclusions	59
LIST OF REFERENCES	61
VITA	64

LIST OF FIGURES

FIGURE	PAGE
1. Elementary Perceptron Architecture	17
2. Neural Network with Learnability Matrix	24
3. Neural Network for Genetic Encoding	26
4. GANN and Backpropagation Compared.....	26
5. Example NN for Output Layer Fitness Determination.....	31
6. Fully Connected 3-Layer Network	36
7. Middle Layer Activation Cancellation	40
8. Effect of Training Set Size on Accuracy for CGNN on Tic-tac-toe Data.....	51
9. Effect of Middle Layer Size on Accuracy for Backpropagation on Tic-tac-toe Data.....	51
10. Effect of Training Set Size on Accuracy for CGNN on Diabetes Data	53
11. Changes in Total Error During Training of CGNN Diabetes Data.....	53
12. Effect of Training Set Size on Accuracy for CGNN on Breast Cancer Data	55
13. Effect of Middle layer Size on Accuracy for Backpropagation on Cancer Data	55
14. Effect of Training Set Size on Accuracy for CGNN on Time Series.....	58
15. Changing Total Error During Training of CGNN on Time Series.....	58

LIST OF TABLES

TABLE	PAGE
1. Partial Sums for SRA Example.....	33
2. Example Trace of SRA.....	33

LIST OF EQUATIONS

EQUATION	PAGE
1. Percent Fitness.....	4
2. Reproductive Growth Equation	8
3. Adjusted Schema Difference Equation.....	9
4. Survival Probability of a Schema under Simple Crossover	10
5. Survival Probability of a Schema.....	10
6. Combined Effect of Reproduction and Crossover	10
7. Probability of a Schema Surviving Mutation	11
8. Probability Schema Survives Reproduction, Crossover, and Mutation	11
9. Population Distribution for 2-Armed Bandit with Sharing	14
10. Integer Scaling	29
11. Fitness Determination for CGNN	30
12. Fitness Function for CGNN with Multiple Output Layer Neurodes	35
13. CGNN Fitness with 1 Output.....	35
14. Mackey-Glass Equation	56
15. Revised Mackey-Glass Equation	56

LIST OF SYMBOLS

A	input vector into a neural network
a_i	an input into a neural network
a	a constant in the Mackey-Glass equation
B	target output vector for a neural network
b_j	a target output from a neural network
b	a constant in the Mackey-Glass equation
c	a constant greater than 1 for raising an individuals fitness
D	a number of backward time steps in the Mackey-Glass equation
$d(H)$	defining length of schema H
d_k	error term for node k in the output layer of a network
e_j	error term for node j in the middle layer of a network
F	a number of forward time steps in the Mackey-Glass equation
$f(H)$	the average fitness of strings with schema H at time t
fitness().....	fitness function
$f()$	threshold function
$\bar{f}$	the average fitness of a population at time t
g_1	number of individuals in queue 1
g_2	number of individuals in queue 2
H	a specific schema
i	iterates the input nodes of a neural network
j	iterates over the middle layer nodes of a neural network
k	iterates over the output layer nodes of a neural network
l	number of input layer neurodes in a neural network
M	total number of individuals in queue 1 and queue 2

$m(H, t)$	expected number of occurrences of schema H at time t
$m(H, t+1)$	expected number of occurrences of schema H at time $t+1$
m	number of middle layer neurodes in a neural network
n	number of neurodes in the output layer of a network
$o(H)$	order of H , the number of fixed positions in schema H
P	survival probability
p	probability of using the crossover operator
p_1	probability of occurrences of type 1 data in a training dataset
p_u	probability of inconsistent data in a training dataset
P_{select}	probability of selecting an individual
q_1	payoff for individuals in queue 1
q_2	payoff for individuals in queue 2
r	the index of an individual in a genetic population
S	a genetic string
s_1	the value at a position in a genetic string
T	a constant in the Mackey-Glass equation
t	a specific time
u	probability of mutation of a single allele
V	middle layer weights in a neural network
Δv	change in middle layer weight
Δv_{ji}	change in middle layer weight from node i to node j
v_{ji}	middle layer weight from node i to node j
W	output layer weights in a neural network
w_{kj}	output layer weight from node j to node k
w_{ki}	output layer weight from node i to node k

Δw	change in output layer weight
Δw_{kj}	change in output layer weight from node j to node k
Δw_{ki}	change in output layer weight from node i to node k
w_j	weight from node j to single neurode in output layer
X	input layer of a neural network
x_i	neurode in the input layer of a neural network
Y	middle layer of a neural network
y_j	neurode in the middle layer of a neural network
Z	output layer of a neural network
z_k	neurode in the output layer of a neural network
z	the neurode in the output layer of a network with one output
z_{target}	desired output from a neural network with one output
α	training constant for the output layer of a neural network
β	training constant for the middle layer of a neural network
χ_r	individual r in a genetic population
χ_ξ	individual ξ in a genetic population
Γ	a floating point number
η	number of individuals in a population
λ	defining length of a particular schema
ξ	iterator over individuals in a population
ζ	a scaled integer representation of a floating point number

LIST OF ABBREVIATIONS

Artificial Intelligence.....	AI
Artificial Neural Network.....	ANN
Backpropagation.....	BP
Central Processing Unit	CPU
Cooperative Genetic Neural Network.....	CGNN
Genetic Algorithm.....	GA
Genetic Neural Network.....	GNN
International Business Machines.....	IBM
Least Mean Squares.....	LMS
Millions of Floating Point Operations Second.....	MFLOPS
Neural Network.....	NN
Processing Element.....	PE
Simple Genetic Algorithm.....	SGA
Stochastic Remainder Algorithm.....	SRA

CHAPTER I.

INTRODUCTION

A. Genetic Algorithm and Neural Network Hybrids

Alan Turing, often called the "father" of artificial intelligence, launched the field in 1950 when he posed the question, "Can a machine think?" [10]. The goal of artificial intelligence researchers since that time has been to create computer programs that can solve problems in a way that is considered intelligent if done by a human. Neural networks and genetic algorithms are two developing technologies in the field of artificial intelligence.

Neural networks have been used successfully in many pattern recognition tasks including image and speech processing, robotic control, and signal processing [2]. Genetic algorithms have been used in many applications including optimization of gas pipelines, VLSI circuit layout, and job scheduling [1]. Recently, several techniques called genetic neural networks have been developed for using genetic algorithms to optimize neural networks [3, 4, 5, 13, 16, 17, 19, 20]. With each of these techniques the genetic algorithm simultaneously examines many different neural networks. This means that multiple neural networks must be stored simultaneously and trained separately which requires a lot of memory and system resources [32].

This thesis investigates a different way of combining genetic algorithms and neural networks called a cooperative genetic neural network (CGNN). A CGNN stores and trains a single neural network by using the genetic algorithm to optimize its hidden weights. The purpose of this investigation was to find out if the genetic algorithm can be effectively used in this way.

A cooperative genetic neural network is based on a 3-layer feed forward neural network architecture with bimodal thresholding in the middle and output layers. A CGNN

uses perceptron training to optimize the weights in the output layer and the genetic algorithm to optimize the weights in the middle layer. Each of the weight vectors in the middle layer is encoded as a genetic string and the trained weights of the output layer are used as a measure of fitness for each weight vector in the hidden layer. The hidden weight vectors compete within a genetic evolution context to produce an optimal transformation for the output layer. The use of common weight vectors in the output layer serves as a fitness function with sharing between the hidden weight vectors. This produces a cooperative genetic environment that allows the total population to find optimal hidden weight vectors.

Chapter II contains a review of the current literature that is relevant to the cooperative genetic neural network including topics on genetic algorithms, neural networks, and genetic neural networks.

Chapter III describes the cooperative genetic neural network in detail. This chapter also contains a discussion of why the CGNN technique works.

Chapter IV describes an implementation of the cooperative genetic neural network and the results of testing it on five benchmark datasets. The performance of the backpropagation neural network was compared with CGNN on four of these datasets. The performance of the backpropagation algorithm is often used for comparison with other techniques [5, 13, 28]. Averaging over the 4 comparative benchmarks, the backpropagation algorithm was 1.82% more accurate, but CGNN trained 2.81% faster. This indicates that CGNN can be used instead of backpropagation for problems where training time is more important than accuracy.

CHAPTER II.

LITERATURE REVIEW

A. Genetic Algorithms

A.1 Concepts

A genetic algorithm is an algorithm that is loosely based on the Darwinian evolution concept and is used to optimize a population of coded solutions [1]. The genetic algorithm was conceived by John Holland in 1962. It has been developed by him and his colleagues and students at the University of Michigan. The greatest advantage of using genetic algorithms is that they are robust.

The basic version of the genetic algorithm introduced by Holland is called the simple genetic algorithm (SGA). The simple genetic algorithm starts with a randomly generated population of individuals. Each of these individuals is a coded representation of a possible solution to the problem that the GA is being used to solve. The population is replaced repeatedly using the genetic operators of reproduction, crossover, and mutation until an adequate solution is found.

The reproduction operator is used to choose a subset of the current population that will be used to create the next population. This subset of individuals is called the breeding pool. Selection of these individuals is based on their fitness. Fitness is a measure of success that an individual might have on the problem expressed as a single positive number. The breeding pool is selected from the population by giving individuals with higher fitnesses a greater chance of being selected, and then selecting individuals based on this chance. A breeding pool is a subset of the entire population, and can have multiple occurrences of the same individual. A typical breeding pool is half the size of the

population. A small breeding pool size produces a more elitist algorithm. This means that individuals within the population that have lower fitnesses have a smaller chance of having their genetic material passed on to the next generation.

There are many methods for determining which individuals to select for breeding based on their fitnesses [7]. The most common method, and the one used in the simple genetic algorithm, is called the roulette wheel method. Using the roulette wheel method, each individual is assigned a number that represents what percentage of the total fitness of the population it represents. This number is computed by dividing the individual's fitness by the total fitness of the population as shown in equation 1. In this equation χ_r represents the r^{th} individual, χ_ξ represents the ξ^{th} individual, η is the number of individuals in the population, and $p_{\text{select}}(\chi_r)$ is the r^{th} individual's probability of being selected in a single selection.

$$p_{\text{select}}(\chi_r) = \frac{\text{fitness}(\chi_r)}{\sum_{\xi=1}^{\eta} \text{fitness}(\chi_\xi)}$$

Equation 1. Percent Fitness

For each individual that is needed for the breeding pool, a random number between zero and one is picked. This number is a random cumulative probability that is used to select an individual from the population. Next, the list of individuals is traversed and the selection probability of each individual is added into a cumulative sum. During this traversal, if the cumulative sum is greater than or equal to the cumulative probability, the most recent individual whose selection probability was added into the cumulative sum is put into the breeding pool.

After the breeders have been selected and put into the breeding pool, pairs of individuals called parents are picked at random from the breeding pool and one or two new individuals are created using one or both of the parents. Each newly created individual is inserted into the new population. The number of offspring produced is dependent on the creation technique. Individuals are created until the new population is filled.

The new individuals are created by copying one of the parents, or by applying another genetic operator called crossover. Simple crossover involves cutting each parent into 2 sections with the first section of the each parent being of the same length. Then the two front or back halves are swapped and the result is used to create two new individuals that are put into the new population.

There are 2 other types of crossover that are often used called k-point crossover and uniform crossover. Simple crossover is a special case of k-point crossover where k is 1. In k-point crossover k points are chosen at random along the length of the chromosomes. Each of the parents is broken in these places, and then every other corresponding part of the parents is swapped to produce 2 new individuals.

Uniform crossover is also a special case of k-point crossover where k is equal to the number of characters in a chromosome. This is also known as shuffle crossover. Uniform crossover is considered disruptive by some [3], and it is considered to be better than lower order crossovers for some problems by others.

The mutation operator is used on newly created chromosomes. It is the small random chance that an individual bit in the chromosome will be altered. Mutation can introduce new genetic material into a population, or it can recover genetic material that has been lost in previous generations [1].

The reproduction cycle is usually repeated for a number of generations or until a certain fitness is achieved. For most problems a usable population consists of at least 50 individuals and reproduction continues for 50 to 100 generations.

The genetic algorithm operates on string structures. These structures represent encodings of solutions that are used to search a solution space. The use of string structures that are encodings of solutions differentiates genetic algorithms from other learning algorithms. Other learning algorithms represent learned knowledge about how to achieve the solution. For example, many learning algorithms construct decision trees while learning [22]. GAs do not use domain specific knowledge to search for a solution. This means that the genetic algorithm is a weak technique. This is in contrast to a strong technique which uses specific domain knowledge to solve a problem [31].

Since genetic algorithms search using a population of solution encodings, they do not search the solution space from a single point. Thus, in contrast to highly directed search procedures like hill climbing, genetic algorithms tend not to converge to local optimums because they look at more than one point in the search space at once.

The genetic algorithm combines parts from each of the solutions that it examines to pick new solutions to be examined. This causes it to perform better than an algorithm that performs a directed search of a solution space from many different points. This gives the genetic algorithm an inherent parallelism that makes it desirable for many problems [1].

Many different types of encodings are used by genetic algorithms. An encoding is a representation of a solution. The most common encoding is to use a binary representation of the decision variables for the solution. These encodings parallel chromosomes in biological systems, and the bit positions parallel genes. The encoded solution is called the genotype and the decoded string is referred to as the phenotype.

Using this encoding technique, a string of binary bits is built up. Although the binary alphabet is usually chosen for the strings in a GA, other alphabets can be used. For example, the alphabet used in human DNA uses 4 values, each expressed as a different chemical, adenine, guanine, thymine, or cytosine.

A.2 Mathematical Foundations

The mathematical foundations of genetic algorithms are based on the concept of schema [1]. Genetic encodings can all be simplified to the binary alphabet $A=\{0,1\}$. Each chromosome is composed of a list of characters in this alphabet. A 7 bit string $S=0101101$ can be represented as $S=s_1s_2s_3s_4s_5s_6s_7$, where each s_i represents a single gene. The values of each s_i is called an allele.

Schema are chromosome templates. They are used when comparing strings. For example, $**1*****$ is a schema that represents strings on S that contain a "1" in the second position and "0"s or "1"s in all the other positions. All work with schema in GAs is based on the assumption that the solution to a problem is composed of parts that contribute to the solution and the presence of parts of a solution can be reflected in an organisms fitness.

The order of a schema is the number of fixed positions in the schema. A fixed position in a schema is one that has a value indicated. The schema $H=*1****1$ has an order of 2 which is indicated as $o(H)=2$. The defining length of a schema is the distance between the end most fixed positions in the schema. The end most fixed positions are known as sentinel positions. In the schema $H=*1****1$ the defining length, $d(H)$, is 5.

The concept of schema can be used to show that selection, crossover, and mutation will produce increasingly fit individuals in the population over time. An equation that indicates this can be built up by considering each of these genetic operations in turn.

The primary operator used by the genetic algorithm is selection. If the average fitness of individuals possessing a schema H is used in equation 1 it will produce the average percent fitness for individuals possessing the schema H . The percent fitness of an individual determines the chance that it is picked during breeding. The average percent fitness multiplied by the size of the new population and the number of individuals that contain the schema will give the number of individuals in the new population that contain the schema H . The average fitness of members of a population is determined by dividing the sum of their fitnesses by the number of individuals in the population. This means that the number of individuals possessing a schema in the new generation is determined by multiplying the number of individuals possessing the schema in the old population by the average fitness of individuals possessing that schema and dividing this by the average fitness of the old population. The equation for this is called the reproductive schema growth equation [1]. The reproductive schema growth equation (equation 2) indicates that useful schema are reproduced more often than lesser useful schema. In this equation, H is the schema, t is the time step, $m(H,t)$ is the number of examples of schema H at time t , and $m(H,t+1)$ is the number of examples of schema H at time $t+1$. Also, $f(H)$ is the average fitness of strings that have schema H at time t , and $\bar{f}$ is the average fitness of the population at time t .

$$m(H, t + 1) = \frac{m(H, t)f(H)}{\bar{f}}$$

Equation 2. Reproductive Schema Growth Equation

This indicates that the schema grows as the ratio of the average fitness of the schema to the average fitness of the population. If $f(H)$ is greater than $\bar{f}$ then $m(H,t)$ is multiplied by a number that is greater than 1, so $m(H,t+1)$ will be greater than $m(H,t)$.

Conversely, if $f(H)$ is less than $\bar{f}$, then $m(H,t)$ is multiplied by a number that is smaller than 1, so $m(H,t+1)$ will be less than $m(H,t)$. A schema with a fitness that is above the average fitness of the population has a higher chance of being present in individuals in successive generations.

Now suppose that a particular schema has a new fitness $c \cdot \bar{f}$ where c is a constant that is greater than 1. By substituting this in the Reproductive Schema Growth Equation a new equation (equation 3) is derived.

$$m(H, t + 1) = m(H, t) \frac{c \cdot \bar{f}}{\bar{f}} = c \cdot m(H, t)$$

Equation 3. Adjusted Schema Difference Equation

If a schema has above average fitness then there will be more instances of it in each successive population. The schema difference equation represents exponential growth. This indicates that genetic algorithm reproduction awards exponentially increasing numbers of trials to above average schemata.

Next, the effect of crossover on the populations fitness needs to be considered. A schema survives when the crossover site falls outside the schema. If the crossover site falls inside a schema then the schema is lost in an individual unless the schema appears in both parents. The probability that a given schema will survive a crossover operation can be computed. This is shown in Equation 4. Here P is probability of a schema surviving crossover, and λ is the defining length of the schema. The actual probability may be slightly higher because the schema may not be destroyed, but this will not adversely affect the proof.

$$P = 1 - \frac{d(H)}{\lambda - 1}$$

Equation 4. Survival Probability of a Schema under Simple Crossover

In most genetic algorithms there is a probability that crossover will occur. If this is taken into consideration, the equation for the survivability of a schema (Equation 5) is derived by multiplying equation 4 by this probability and subtracting it from 1.

$$P \geq 1 - \frac{p \cdot d(H)}{\lambda - 1}$$

Equation 5. Survival Probability of a Schema

Here P is the survival probability of the schema, and p is the probability of using the crossover operator.

Multiplying the probability of a schema surviving reproduction by the probability of a schema surviving crossover gives the probability of a schema surviving both. This is shown in equation 6. Equation 6 shows that schema with above average fitness and short defining lengths will be present in exponentially larger amounts in successive generations.

$$m(H, t + 1) \geq m(H, t) \cdot c \cdot \left[1 - \frac{p \cdot d(H)}{\lambda - 1} \right]$$

Equation 6. Combined Effect of Reproduction and Crossover

In addition to reproduction and crossover, mutation has an affect on the survivability of a schema. For a schema to survive, all of its fixed positions must survive. A single allele will survive with probability (1 - u), where u is the probability of a single allele being changed by mutation. A particular instance of a schema survives if all of its alleles

$o(H)$ survive. Given this information the probability of a schema surviving mutation (Equation 7) is derived.

$$P = (1 - u)^{o(H)}$$

Equation 7. Probability of a Schema Surviving Mutation

Ignoring some terms in the final reproduction difference equation and multiplying it by the probability of crossover and mutation survival yields the final equation (Equation 8) which considers the effects of all three operators.

Equation 8 shows that the addition of mutation to the equation changes the outcome very little for very small mutation probabilities. The name of this theorem is the Schema Theorem or the Fundamental Theorem of Genetic Algorithms [1].

$$m(H, t+1) \geq m(H, t) \cdot c \cdot [1 - p \cdot \frac{d(H)}{\lambda - 1} - o(H) \cdot u]$$

Equation 8. Probability Schema Survives Reproduction, Crossover, and Mutation

A.3 Niching and Speciation

Niching and speciation are factors that can be introduced into a genetic algorithm to make it possible for subsets of the population to specialize and leverage subdivisions of a problem [1]. The idea of niching and speciation was developed directly from biological models. It is obvious that different organisms exploit different parts of the environment. For example, there are organisms on earth that thrive in volcanic vents in the ocean in a sulfurous environment as there are animals that survive the harsh Arctic areas of the polar regions.

A niche is an organisms place in the environment, and a species is a group of organisms that have certain characteristics in common. In the biological examples the niches are the areas around volcanic vents, and the Arctic polar regions. The species in these examples are bacteria, and polar bears. The GA counterpart of species are subsets of the population of strings with common schema, and the counterpart of niches are subsets of the domain of a function. The simple genetic algorithm alone does not encourage the development of niche and species.

Niching and speciation are advantageous to life on earth because they allow life to spread as widely as possible. They also make life more able to adapt to fluctuations in the environment. It is advantageous in some problems to introduce niche and species into the genetic algorithm to gain similar benefits.

As an example of how niche and species are advantageous, consider a fitness function that has many global maxima. If the simple genetic algorithm is used to find the individual with the maximum fitness, then eventually most of the solutions will fall on a single peak. If the desired result is to develop stable subpopulations that indicate each of the peaks it is necessary to modify the GA somewhat.

Also, consider a function with unequal peaks. It is desirable, in some instances to find the peaks in a function of this sort also, even though some of the peaks are less than the global maximum. The simple GA clusters most of the individuals around the highest peak. The introduction of niching into the GA enables the new algorithm to allow individuals to exploit a subdomain of the problem and produce individuals that are clustered around each of the peaks. It might also be desirable that the modified algorithm cluster a percentage of the population around each peak that is directly proportional to the height of each peak. The number of individuals clustered around a peak would then

represent the relative desirability of each solution. The proper use of niching achieves all of these goals.

A simple example of how niching can be added to a genetic algorithm is called two-armed bandit with sharing among individuals and is detailed by Goldberg [1]. In this example there is a jackpot machine with two levers. There are also two queues that individuals can line up in corresponding to the two levers. The chromosomes in the GA encode the queue that the individuals will line up in. During the evaluation phase of the genetic algorithm, each organism pulls the lever that corresponds to the queue that it is in. An individual that pulls the lever for queue 1 is awarded \$75 and an individual that pulls the lever for queue 2 is awarded \$25. If the population size is 100, then it is expected that the simple genetic algorithm will cause the population to change so that most of the 100 individuals are lined up in queue 1. To extend the genetic algorithm and introduce niching, the individuals in each queue are made to share in the payoff of its corresponding arm. If all of the 100 individuals end up in queue 1 then each individual will be awarded $\$75 / 100 = \$0.75$. On the other hand, if all of the individuals line up in queue 2, their reward will be $\$25 / 100 = \$0.25$.

It is clear at this point that in each case an individual stands to benefit greatly by changing queues. In the first case an individual switching queues will gain $\$25 - \$0.75 = \$24.25$ and in the second case an individual will gain $\$75 - \$0.25 = \$74.75$. If individuals are allowed to shift until there is no benefit from switching cues, the population will reach a stable point where 75 of the individuals are in the \$25 queue, and 25 of the individuals are in the \$75 queue. If q_1 is the payoff for the lever for queue 1, q_2 is the payoff for the lever for queue 2, g_1 is the number of individuals in queue 1, g_2 is the number of individuals in queue 2, and M is the population size, then equation 9 can be used to determine the solution for the general case.

$$\frac{q_1}{g_1} = \frac{q_1}{M - g_2} = \frac{q_2}{g_2}$$

Equation 9. Population Distribution for 2-Armed Bandit with Sharing

A.4 Analysis

Genetic algorithms are used for optimization. A GA is a weak technique for searching a solution space. The fundamental theorem of genetic algorithms indicates that schema that contribute to a higher fitness appear with greater frequency in successive generations of the genetic algorithm. The genetic algorithm will produce more fit individuals in successive populations if the schema that contribute toward higher individual fitness can be combined to find the optimal solution. Some problems can be deceptive to the genetic algorithm. Deceptive problems are created when short schema that contribute to high fitnesses combine to form longer schema with lower fitnesses. This can be remedied by the correct selection of a fitness function and problem encoding.

A chromosome can represent and optimize the machinery of pattern recognition tasks, but the genetic algorithm is not specifically tailored to pattern recognition tasks. The simple genetic algorithm has been extended in many ways including niching, speciation, and a multitude of breeder selection techniques. It also appears that the algorithm itself is highly configurable and parameters may need to be adjusted for each problem for maximal performance. There also does not seem to be any guarantee as to when the algorithm will converge, but that it will given sufficient time and population size, a good genotype, and a good fitness function.

B. Neural Networks

B.1 Concepts

Neural networks are processing algorithms or devices that are inspired by the workings of the animal brain. A neural network is composed of simple processing units called "units", "processing elements", or "neurodes" [2], and connections between these units called "connections" or "channels". These correspond to neurons and dendrites in a biological brain. A neural network is sometimes called an artificial neural network (ANN) to differentiate between it and the neural networks in the animal brain.

The neurodes in an ANN each have a numerical value associated with them called an activation. A typical neurode performs a weighted sum of its inputs and passes this through an activation function to determine its activation. Each connection has a numerical weight associated with it that affects the flow of activity through the network. A neural network has a training rule that is used to adjust the interconnections between neurodes on the basis of patterns that are presented to it. A neurode may be connected directly to inputs, or it may get its inputs from itself or another neurode. Two typical activation functions are the sigmoidal function and the threshold function. A sigmoidal function is one that is bounded and differentiable. A threshold function is a stair-step function that can output different values based on whether or not one or more thresholds are met.

Most neural networks are constructed with their neurodes arranged in layers. Each of these layers are usually fully connected to other layers and activations flow in one direction through these layers.

B.2 Mathematical Foundations

Training rules are used in NNs to adjust the weights on connections. In addition, some training rules adjust other network parameters. There are many different architectures and training rules used for neural networks. The training rule that is used by CGNN to train its output layer is called the perceptron training rule. The perceptron training rule is normally used to train a two-layer network on linearly separable data. Linearly separable data is data that can be separated into solution categories by a hyperplane in the input data space. To train on non-linearly separable data requires at least three layers. The perceptron training rule is used in a cooperative genetic neural network to adjust the connections between the middle and output layers. The perceptron, one of the oldest neural models, was introduced by Rosenblatt in 1957 [29, 30]. The elementary perceptron is a two-layer feed forward network. The elementary perceptron is a heteroassociative, nearest-neighbor pattern matching algorithm. The architecture of the elementary perceptron is visually represented in figure 1.

The perceptron training algorithm uses data space hyperplanes represented by the network weights to separate the data into 2 classes. During training the hyperplanes are moved until the best hyperplane position is found. The perceptron network adjusts the hyperplanes by attempting to minimize the error between the expected and computed outputs.

The training procedure for perceptron networks follows [2].

- 1) Assign random values in the range [-1, +1] to all weights in **W**.
- 2) For each pattern pair in the training dataset, (**A**, **B**) do the following.
 - a) Transfer **A** to **X**, filter **X** activations through **W** to get the new **Z** activations.

$$\text{for each } k=1..n \text{ do } z_k = f\left(\sum_{i=0}^l w_{ki} \cdot a_i\right) \text{ where } f(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{otherwise} \end{cases}$$

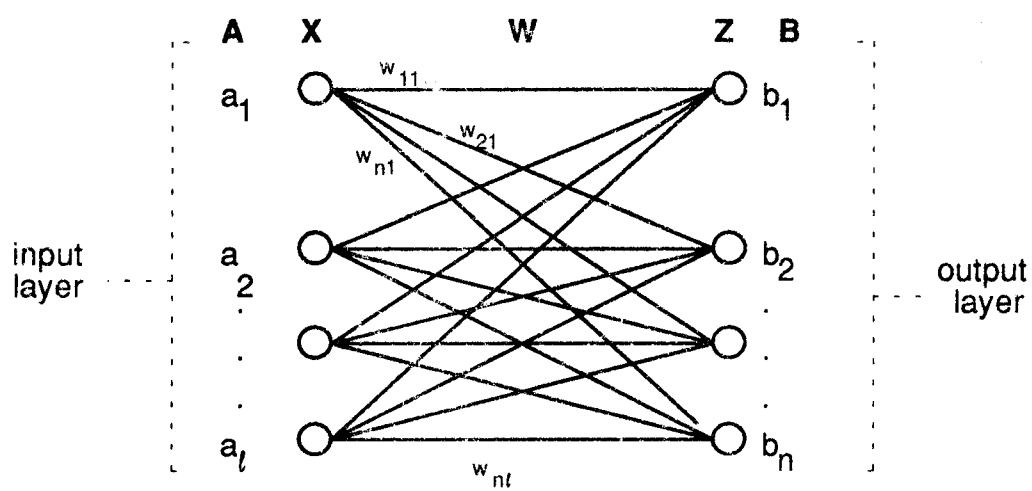


Figure 1. Elementary Perceptron Architecture

b) Compute the error between the computed and desired output.

for each $k=1..n$ do $d_k = z_k - b_k$

c) adjust the **W** connection weights.

for each $k = 1...n$ do

for each $i = 1...l$ do

$$\Delta w_{ki} = \alpha * a_k * d_k$$

Where the α is a positive constant between 0 and 1, which is used to

control the learning rate.

3) Repeat step (2) until the error correction value is sufficiently low or a sufficient number of iterations has been performed.

B.3 Backpropagation

Backpropagation (BP) is a training rule for neural networks. A neural network that uses this training rule is called a backpropagation network. Werbos (1974) discovered the backpropagation algorithm, calling the algorithm dynamic feedback. But it was Rumelhart, Hinton and Williams in 1986 that showed the power of the backpropagation algorithm and excited the scientific community, reviving interest in neural networks [2]. A backpropagation network always has at least 3 layers, the input layer, the middle layer, and the output layer. The classic BP network has 3 layers that are fully connected between layers. Being fully connected between layers means that each neurode in the middle layer receives input from all of the neurodes in the input layer, and each neurode in the output layer receives input from all of the neurodes in the middle layer. Although this is the traditional way that a backpropagation network is presented, the backpropagation algorithm can be used with any multi-layer neural network topology.

Backpropagation is based on the propagation of error corrections in the reverse direction from the normal feed-forward flow in a network. It has been used with good success on a wide variety of practical applications. In addition, backpropagation is a supervised learning technique. This means that training a BP network requires that the training data include the correct output for each set of inputs so that the weights in the network can be adjusted.

Training in a BP network consists of a cycle that continues until the network error has reached a desired minimum or for a given number of iterations. During each cycle a pass is made through the training data in which each input pattern is propagated forward through the network. Next, each output activation is compared with the desired output and an error is determined. This error is then used to adjust the weights in the output layer and the errors for the output layer and the weights between the middle and output layers are used to adjust the weights going into the middle layer.

The full backpropagation training procedure follows [2].

- 1) Assign random values in the range $[-1, +1]$ to all weights in **V** and **W** where **V** is the matrix that contains the weights between the input and middle layers and **W** contains the weights between the middle and output layers.
- 2) For each pattern pair in the training dataset, (**A**, **B**), do the following.
 - a) Transfer **A** to **X**, then filter the **X** activations through **V** and calculate the new **Y** activations.

$$\text{for each } j=1..m \text{ do } y_j = f\left(\sum_{i=0}^l v_{ji} \cdot x_i\right)$$

Where $f(x)$ is the threshold function.

- b) Transfer the **Y** activations through **W** to compute the **Z** activations.

$$\text{for each } k=1..n \text{ do } z_k = f\left(\sum_{j=0}^m w_{kj} \cdot y_j\right)$$

- c) Compute the error between the computed and desired output.

$$\text{for each } k=1..n \text{ do } d_k = z_k (1 - z_k) (b_k - z_k)$$

- d) Compute the error for the middle layer weights.

$$\text{for each } j=1..m \text{ do } e_j = y_j (1 - y_j) \left(\sum_{k=1}^n w_{kj} \cdot d_k \right)$$

- e) Adjust the middle to output layer weights **W** using the following equation.

for each $k=1..n$ do

for each $j=1..m$ do

$$w_{kj} = w_{kj} + \alpha \cdot y_j \cdot d_k$$

Where α is a constant positive constant between 0 and 1, which is used to control the learning rate.

- f) Adjust the connection weights **V** using the delta rule.

for each $j = 1..m$ do

for each $i=1..l$ do

$$v_{ji} = v_{ji} + \beta \cdot x_i \cdot e_j$$

Where β is a constant positive scalar value between 0 and 1, which is used to control the learning rate.

- 3) Repeat step (2) until the error correction value is sufficiently low or a sufficient number of iterations have been performed.

The backpropagation algorithm uses a sigmoidal threshold function. A sigmoidal function is continuous, S-shaped, monotonically increasing, and asymptotically approaches fixed values as the input values approach plus and minus infinity. Typically, the sigmoid function used is $f(x) = 1/(1+e^{-x})$. This function approaches zero at negative infinity and +1 at positive infinity.

The output layer weights in a backpropagation network are adjusted in a way that is similar to the way that the weights of a perceptron network are adjusted, but this cannot be

used to compute an adjustment for the middle layer weights. The problem with using the delta rule here is that there are no known desired activations supplied for the middle layer. The answer to the problem is to notice that bad activations in the output layer can be blamed on both the weights between the middle and output layers and on the activation levels of the middle layer. So, the error correction for the middle layer neurodes must be based on a portion of the error they are associated with in the output layer. This is done in step (2.d) by weighting the error for each PE in the output layer with the connecting weight for the middle layer neurode that the error is being computed for. The activation function for the output layer must also be taken into account, and this is done by multiplying the weighted error by the derivative of the threshold function. The threshold derivative helps stabilize the network by preventing overcorrections of large weights and error terms in the output layer with small activations in the middle layer [11].

In addition to the outlined procedure, most backpropagation networks use a threshold for each processing element that is adjusted during training along with the weights. Also most BP networks use a momentum term that changes the error correction a small amount based on previous corrections. The momentum term was first discovered by Rumelhart, Hinton & Williams in 1986 [2].

In addition to the momentum term, there has been a lot of research with variants of the BP network to make it converge faster. Hush and Salas (1988) extended the original idea of momentum to use several past error corrections. Some research has also been done with using several times the error gradient (Cater 1987). It has also been shown that the performance is enhanced by using a sigmoid function that goes from -1 to +1 (Stornetta and Huberman 1987). Many more complex enhancements have been tried with some success [2].

B.4 Analysis

Perceptron networks have been shown to converge only on linearly separable data [30]. Linearly separable data is data that can be divided into the solution categories by a plane in the input space.

The backpropagation algorithm was responsible for the rebirth of the field of neural networks. This is because a backpropagation network can solve problems that were not solvable using early networks like the perceptron. A backpropagation network is able to do this because the backpropagation network uses more than two layers of processing elements.

The strength of the backpropagation algorithm is in the procedure for updating weights for neurodes that are not in the output layer. The problem of updating the weights for layers that are not directly connected to the output layer is known as the credit assignment problem, because it is difficult to determine which weights in the middle layer cause incorrect output. Because of its strength the backpropagation algorithm and BP networks are the subject of a lot of research.

The primary drawbacks of BP networks are that they train very slowly and can get stuck in local error minima because they follow the error gradient which can have many critical points. Because the BP network has been studied so extensively, it is an ideal candidate for benchmark comparison against other newer network algorithms and topologies.

C. Genetic Neural Networks

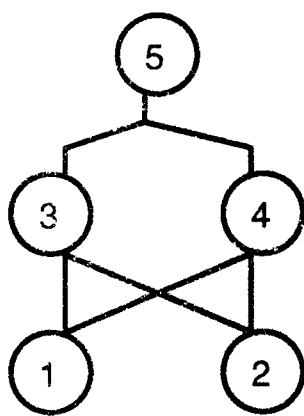
C.1 Concepts

Attempts have been made to integrate neural networks and genetic algorithms to try to combine the benefits of both. These GA and NN hybrids are called genetic neural networks (GNNs). There are two types of GNNs which are classified according to what part of a neural network they optimize. In the first type the GA is used to optimize the NN architecture, and in the second type the GA is used to optimize the NN connection weights. In both approaches the chromosomes represent a whole neural network specification [32].

C.2 Optimizing Network Topology

Several researchers have used genetic algorithms to find optimum network architectures [14,15]. The teams of Harp, Samad, and Gutha and Miller, Todd, and Hegde both published papers on this in 1989 [3].

Miller, Todd, and Hegde published in 1989 on using genetic algorithms to find network topologies. They used an $n \times n+1$ learnability matrix converted to a bit string to determine necessary connections for a neural network. The learnability matrix's rows are used for connections to neurons and the columns are used for connections from neurons or bias nodes. If a cell contains a 1 then there is a connection and if there is a 0 in the cell there is no connection. The rows in the matrix are then laid end-to-end to form a bit string representing a chromosome. This process is illustrated in figure 2. In this example a network with 2 input neurodes, 2 middle layer neurodes, and 1 output layer neurode is encoded. The encoded string for this neural network is "000000000000110001110001001101".



network

		from				
		1	2	3	4	5
to	1	0	0	0	0	0
	2	0	0	0	0	0
	3	1	0	0	0	1
	4	1	0	0	0	1
	5	0	0	1	0	1

learnability matrix

Figure 2. Neural Network with Learnability Matrix

C.3 Optimizing Network Weights

Several attempts have been made at using genetic algorithms to find optimum weights in a neural network [5,3,16,17]. Caudell and Dolan, Montana and Davis, and Whitley and Hanson all published work in this area in 1989 [3].

Montana and Davis took the approach of encoding the weights of a neural network as a string in the genetic algorithm. The GA worked with a population of neural networks. For example, using the network shown in figure 3, the encoding would be: (.3, -.4, 1.2, .8, -.3, -.1, .7, -6.3). They used a fixed number of layers and nodes. In addition, each layer of the network was always fully connected.

Montana and Davis compared the genetic neural network with backpropagation and showed that the GNN did not get stuck in false minimums like the BP algorithm. The GNN showed better performance as shown by the graph in figure 4.

Hiroaki Kitano [20] has done work showing the speed of convergence of using a genetic algorithm with a population of neural networks. This work also pointed out the major two problems encountered in using this approach: 1) failures in fine-tuned local search, and 2) a trade off between population size and speed of convergence. When genetic algorithms are used for finding network weights they will converge very quickly at the beginning of training (outperforming backpropagation), but they eventually fail to reduce the error as much as backpropagation at the end of training.

Maureen Caudill [32] has pointed out the Lamarckian flavor of this approach. Lamarck was a biologist who believed that knowledge acquired during an organisms lifetime could be transmitted genetically. These ideas do not fit our current knowledge of Darwinian genetics.

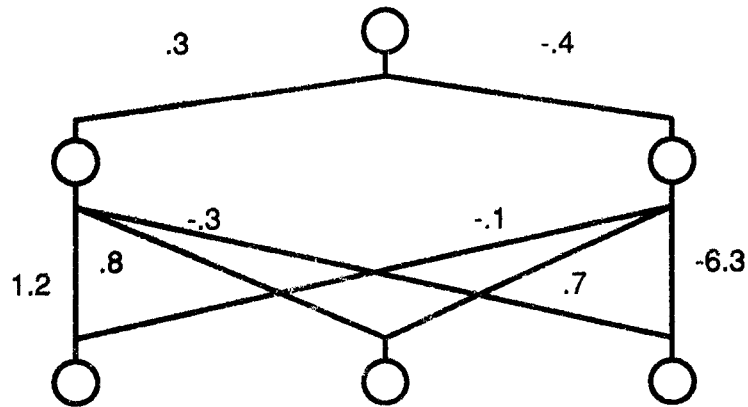


Figure 3. Neural Network for Genetic Encoding

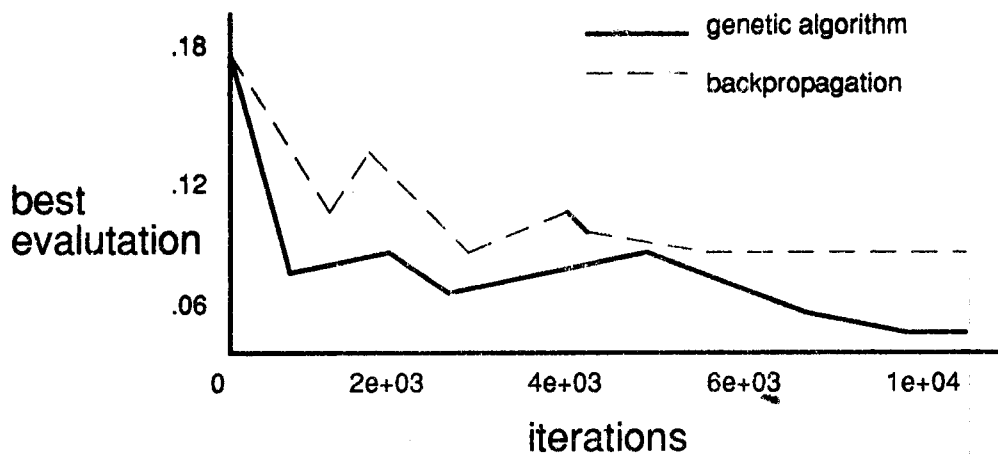


Figure 4. GANN and Backpropagation Compared

C.4 Analysis

Genetic Algorithms have been used successfully to optimize neural networks. Different approaches have been successful, but both of these approaches to genetic algorithm and neural network hybrids are plagued by the same problem, they require huge populations, because each individual in the population represents an entire network. For real problems the string encoding is very large [32].

Genetic algorithms converge very quickly during the beginning of training, but fail to fine tune at the end of training. This problem may be overcome by further research in this area.

CHAPTER III.

COOPERATIVE GENETIC NEURAL NETWORK TRAINING

A. System Overview

The system presented here is called CGNN for Cooperative Genetic Neural Network. The system simulates an interacting genetic algorithm and neural network. The neural network has a feed-forward architecture with 3 layers. Both the genetic algorithm and perceptron training are used to adjust the weights of the neural network. The basic procedure underlying CGNN is as follows.

- 1) Initialize the middle layer weights randomly
- 2) Initialize the output layer weights to 0
- 3) Make a small number of passes through the training dataset performing neural network feed-forward computations and adjusting the output layer weight vectors using Perceptron training rules.
- 4) Use the resulting weights in the output layer as fitnesses for a genetic population composed of individuals representing weight vectors in the middle layer and perform genetic selection and recombination to form new middle layer weights.
- 5) Repeat steps 2 through 4 for a predetermined number of iterations.

B. Genetic Encoding

Each weight vector in the hidden layer is coded into one individual in the population. The individual is a string of binary bits. The string is formed by concatenating the binary representation of a scaled integer representing each component of the weight

vector. Several other ways of encoding the weights were considered for use in CGNN, including fixed point representation and floating point representation. Scaled integer representation was chosen because it seemed to be the most predominant in the literature on genetic algorithms. The number of bits allocated for each number in the weight vector is variable and may be assigned in the configuration file. An example will make the representation technique more clear. If a weight vector in the middle layer is [0.2 -0.4 0.33] and 8 bits have been allocated for each weight vector component, the string representing the weight will have 24 bits, and each group of 8 bits will be a scaled integer between 0 and 255 representing each weight vector component. In order to scale floating point numbers a minimum and maximum value must be known. CGNN normally uses -1.0 and +1.0 as the minimum and maximum weight vector components. This value can be changed in the configuration file. So, using these minimum and maximum values the resulting scaled integer representation for the weight vector in question is [153 76 169]. Each of the scaled components was derived by using Equation 10. In the equation Γ is the floating point number and ζ is the scaled integer representation.

$$\zeta = \left\lceil \frac{\Gamma + 1}{2} \right\rceil 255.0$$

Equation 10. Integer Scaling

The binary string resulting from concatenation of the binary representation of these scaled values is "100110010100110010101001". If only a few bits are allocated for the encoding the search will be coarse, and conversely, if many bits are allocated, the search will be more fine. Of course, longer bit strings mean that the search space will be bigger and the algorithm will take longer to converge.

C. Output Layer Training and Fitness Determination

CGNN training is an iterative procedure. During a training iteration several passes are made through the training dataset and perceptron training is used to adjust the connections between the middle and output layers, and then the genetic algorithm is used to generate new middle layer weights. The outputs from a cooperative genetic neural network are -1 or +1, because of the perceptron architecture of the output layer.

In the internal genetic algorithm the fitnesses of individuals are derived from the weights that go from the individuals corresponding neuron to output layer neurons. The fitness is found by summing the absolute values of these weights. Consider the fully connected 3 layer CGNN network of figure 5. Bias nodes have been used in CGNN, but they have been left out of the diagram because they are not used in fitness determination.

If the individual representing neurode j in the middle layer is being evaluated and the number of output layer neurodes is n , then the fitness of the weight vector associated with the middle layer neuron is computed using Equation 11, because it contributes to the output of each neurode that it is connected to in the output layer, and the weights on the connections to the output layer indicate the extent to which the output from the middle layer node affects the output of the output layer neurons.

$$\text{fitness}(j) = \sum_{k=1}^n |w_{kj}|$$

Equation 11. Fitness Determination for CGNN

D. Internal Genetic Algorithm

CGNN uses the genetic algorithm to optimize the middle layer weights. The version of the genetic algorithm used differs from the simple genetic algorithm in one

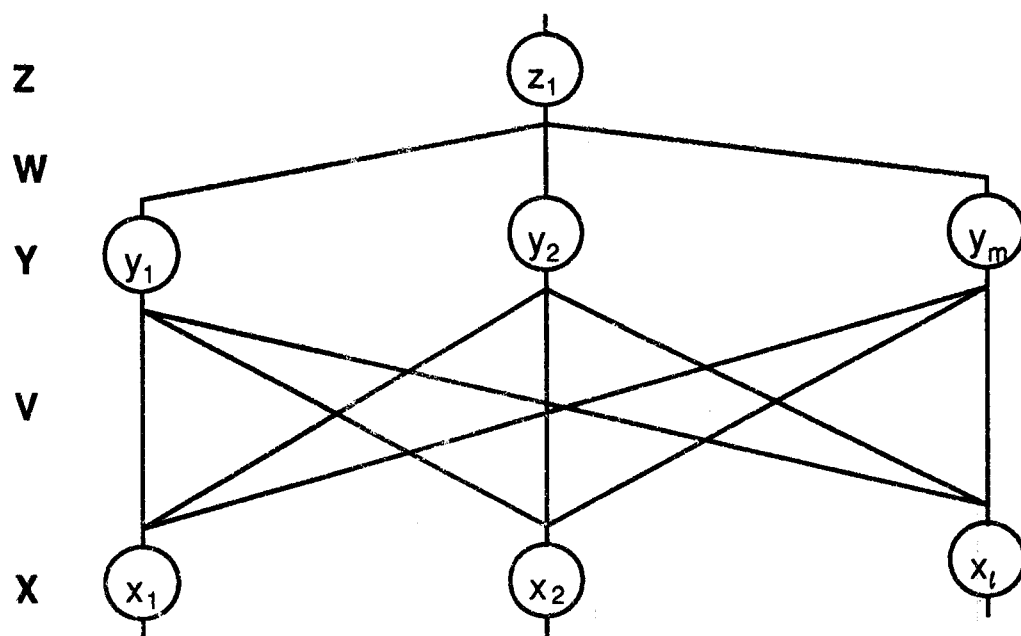


Figure 5. Example NN for Output Layer Fitness Determination

respect, it uses a selection technique called the stochastic remainder algorithm (SRA).

This variation was chosen because it has been shown to give a better distribution of breeders than the roulette wheel method.

Using SRA, fitnesses are laid out end to end. Then, the total fitness is determined. The fitnesses as they are laid out must be traversed by finding an initial starting point and a delta. The delta fitness for traversal is determined by dividing the total fitness by the number of breeders wanted. The initial starting point is a random number between 0 and the delta. A moving sum is set to 0. As a final preparation a target sum fitness is initialized to the starting point.

The list of individual fitnesses is traversed adding in each fitness to the moving sum. When the sum exceeds or meets the target sum fitness the current individual is put into the list of breeders, and the target moving sum is incremented by the walk delta. As an example of stochastic remainder algorithm consider the population and fitnesses shown in table 1. If the breeding pool size is 4, the population size is 8, and the sum of all fitnesses is 38, then the initial delta is 9.5. The initially generated search sum, which has been determined randomly is 4.

The individuals in the population are traversed until the moving partial sum of fitnesses is greater than or equal to 4. This is found at individual 1. A copy of individual 1 is placed into the breeding pool. Next, the delta, 9.5, is added to the search sum giving 13.5. The individuals are then traversed until a partial sum that meets or exceeds the search sum is found. This is found at individual number 5. This individual is also placed into the breeding pool and the search for breeders continues until the breeding pool is filled. This is shown in table 2.

Using this method only one random number has to be determined, and the fitnesses are traversed only once. Using roulette wheel selection a random number must

Table 1. Partial Sums for SRA Example

INDIVIDUAL	0	1	2	3	4	5	6	7
FITNESS	5	3	2	1	6	7	4	10
PARTIAL SUM	5	8	10	11	17	24	28	38

Table 2. Example Trace of SRA

INDIVIDUAL	SUM	SEARCH SUM	COMMENTS
0	0	4	
1	5	4	0 is a breeder
1	5	13.5	
2	8	13.5	
3	10	13.5	
4	11	13.5	
5	17	13.5	4 is a breeder
5	17	23	
6	24	23	5 is a breeder
...	...	...	

be computed, and the traversal of the fitnesses must be performed for each individual in the breeding pool. This means that the stochastic remainder algorithm is faster than the roulette wheel method. It has also has been shown to produce a better distribution than the roulette wheel selection method.

E. Justification of Technique

The convergence of CGNN relies on fitness determination. The mathematics of the genetic algorithm has already been shown in previous sections. Given a fitness function that awards better than average fitnesses to more desirable strings and a good solution encoding, the genetic algorithm will converge on a near optimum solution. The output layer's weights will converge on a solution that minimizes the error between the desired and real outputs, because it uses the perceptron training rule. This also has been shown in previous sections. It is the job of the genetic algorithm in CGNN to determine the weights for the middle layer neurons. This is analogous to the backward propagation of error in the BP algorithm.

To adjust the middle layer weight vectors the algorithm must determine what must be adjusted, or in this case which weights must be changed and how much they must be changed to minimize the error in training. This problem is known as the credit assignment problem. Because of the way the genetic algorithm works, the more fit strings or important parts of strings survive reproduction and appear with more frequency in successive generations, and less fit strings or parts of strings appear with less frequency in successive generations. If the fitness function for CGNN awards higher fitnesses to weight vectors that produce useful output and helps to minimize the training error, and the fitness function awards lower fitnesses to weight vectors that do not prove useful and do not help to reduce

```

101 UT KNOXVILLE LIBRARY      - GEAC LIBRARY SYSTEM - ALL *BOOLEAN SEARCH
                                Citation 7 of 7

AUTHOR      Fuller, Sean Lane.
TITLE       Using a cooperative genetic algorithm population to train hidden n>
PUBLISHER   1994.
CALL NUMBER Thesis94.F86

Location      Loan      Call
              Type      Number

MNTH         THESIS    Thesis94.F86
ARCHV        ARCH      Thesis94.F86

Status        In Library
              In Library

CIT - return to your citation list
CAT - begin a new search
CMD - see additional commands

FUL - see complete citation
BAC - see previous citation
REV - revise current search

Enter code.      Then press SEND

```

the training error in the output layer, then the error will be reduced on the average in successive generations. Thus, in order for CGNN to work, it must be shown that the fitness function used reflects the utility of the weight vectors of the middle layer. The fitness function used is mathematically very simple. the fitness is equal to the sum of the absolute value of the weights on the connections leading out from the middle layer neuron that uses the weight vector.

Consider the fully connected 3 layer network in figure 6. The network has l input neurons, m middle layer neurons, and n output layer neurons. Bias nodes are used in CGNN, but they have been left out of the diagram because they are not used in fitness determination.

If the individual representing neuron j in the middle layer is being evaluated, then its fitness is given by equation 12.

$$\text{fitness}(j) = \sum_{k=1}^n |w_{kj}|$$

Equation 12. Fitness Function for CGNN with Multiple Output Layer Neurodes

In the simple case, where there is only one output layer neuron, the fitness of a middle layer neuron is equal to the absolute value of the weight on the connection from its neuron to the output layer neuron. The fitness equation for the general network with one output neuron in figure 6 is shown in equation 13.

$$\text{fitness}(j) = |w_j|$$

Equation 13. CGNN Fitness with 1 Output

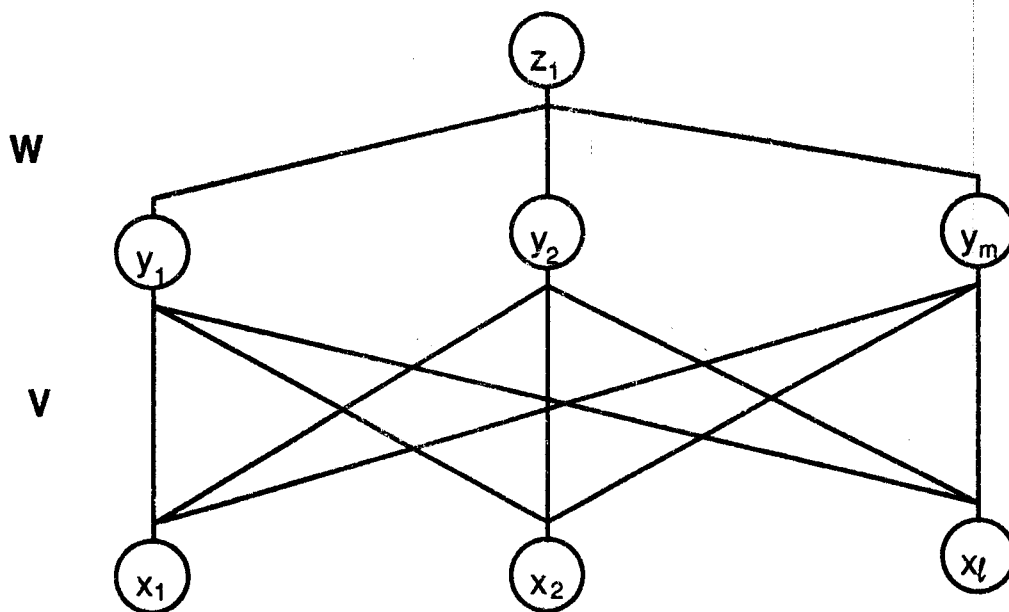


Figure 6. Fully Connected 3 Layer Network

There is currently no proof that the method that CGNN uses for adjusting the weights in the middle layer will converge. The performance of GAs and NNs using perceptron training are known. A justification for the CGNN technique is formed by ensuring that these two techniques are sufficiently linked so that the requirements of each are met. The primary requirement that must be examined in detail is that the fitness for chromosomes in the middle layer rewards chromosomes that are useful for producing middle layer activations that prove useful during Perceptron training of the output layer.

It can easily be deduced that a weight of 0.0 on an output layer weight means that the output from the neuron that it is connected to does not influence the output of the output layer neuron. No matter what the inputs are and no matter what the middle layer weight vector is the resulting term in the single layer feed forward equations will always be zero. This means that the associated weight vector does not contribute to the output of the network, and does not affect the error of the trained network. This indicates that the associated middle layer weight vector has not been utilized. This would indicate that the weight vector is not useful and should have a low fitness if the genetic algorithm is to correctly optimize the middle layer. This is in fact the case for the fitness function used in CGNN. If $w_j = 0$ then $f(j) = w_j = 0$, which is the lowest fitness that can be used by the simple genetic algorithm.

Another case in which it can easily be shown that CGNN's fitness operator is correct is where the dataset is linearly separable and one of the weight vectors in the hidden layer causes its neurode to output values which are nearly equal to the required output of the output layer. If the outgoing connection from this neurode has a weight that is very close to one and all other connections in the output layer have weights that are very close to zero, then the middle layer neuron in question has a great effect on how well the network performs, and so it has been utilized to a high degree by the output layer. Since

the CGNN fitness function uses the absolute value of the weights on the outgoing connections, the fitness of the node that is used will have a fitness that is very close to one and the other nodes will have a fitness that is very close to zero.

It is important to recognize here that the outputs from the middle layer neurons are thresholded. This means that a middle layer neuron can only output a +1 or a -1. This causes the output vector from the hidden layer to be the coordinates of vertices on a hypercube. It also introduces a valuable regularity to the middle layer outputs. If not for this it would be more difficult to correlate the output layer weights with the middle layer neuron utilization. Consider what would happen if the threshold function was not used. If a middle layer weight vector is very large and another middle layer weight vector is very small, but they are both equally useful in the output layer, the second weight vector will require a much stronger weight going out from it than the first in order to equally use its outputs in the final result. Using the threshold function, the dot product is normalized and weight vectors with equal utility will have similar associated output layer weights.

Similarly, if there are two or more neuron weight vectors that correctly partition the input dataset then perceptron training will either divide weights between their associated outgoing connections or give a larger weight to a subset of the neurons. The CGNN fitness function works in this case also because high fitnesses are awarded to more useful weight vectors. When it takes a combination of middle layer neurons to be used in the output layer, the Perceptron training in the output layer will cause the associated weights in the output layer to use the neurode's activation and assign noticeable weights to the useful middle layer neurons. Any middle layer neurons that have weights that generate middle layer activations that contribute to the training error will be diminished toward zero, or they will be canceled out by opposing vectors.

Finally, it is important to examine the case where weight vectors in the middle layer have associated weights in the output layer that cause their effects to be canceled out. The network in figure 7 is an example of a network with this property. In this network the weights along the connection from y_1 to z_1 and the weights along the connection from y_3 to z_1 cancel each other out if $w_1 = w_3$.

Whatever the inputs on y_1 and y_3 are, their outputs will be canceled when computing the output z_1 , because the middle layer weight vectors associated with y_1 and y_3 are exact opposites. It is important to inspect the general case where there are two or more weight vectors connected to the middle layer that cancel each other out during computation of the output layer activation. CGNN requires that the weights going to the output layer indicate utility so that the absolute values of the weights can be used as a measure of fitness. Although individually each of the canceling weight vectors are needed to cancel other weight vectors, the canceling weight vectors do not contribute to the output or performance of the network. Very large weights can be assigned to connections to the output layer from the canceling nodes with no effect on the output of the network as long as the weights cause the input to middle layer weight vector activations to cancel. It can be shown that canceling weight vector fitness is never greater than utilized weight vector fitness for CGNN using a network with 3 middle layer nodes where two of their outputs are always equal in magnitude but opposite in sign. For cases with larger numbers of middle layer neurodes and canceling weight vectors it has been experimentally verified that this requirement is met also.

Consider the case of a cooperative genetic neural network with l input neurodes $x_1, \dots, x_l$, 3 middle layer neurodes y_1, y_2 , and y_3 , and 1 output layer neurode z_1 . The desired output for z_1 during training is z_1^{target} . The network uses the threshold function

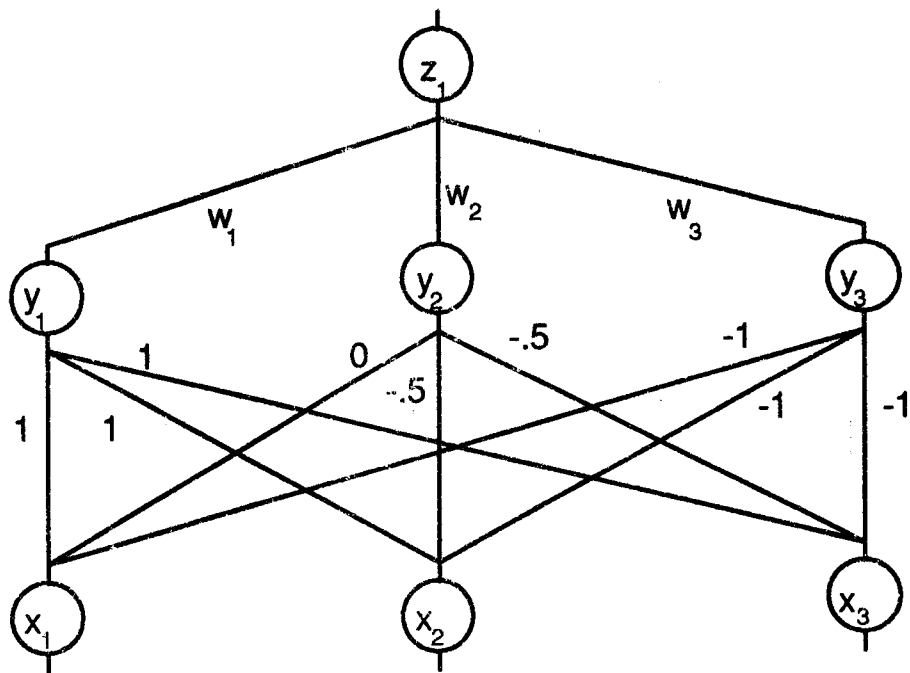


Figure 7. Middle Layer Activation Canceling

$$f(x) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases} \text{ for both the middle and output layers. Vectors } v_1, v_2, \text{ and } v_3 \text{ are}$$

the weight vectors between the input and middle layers connected into y_1, y_2 , and y_3 respectively. The scalars w_1, w_2 , and w_3 are the weights along the connections to the output neurode from neurodes y_1, y_2 , and y_3 respectively. Also, v_1 is equal to $-v_3$ because they are the canceling weight vectors. Because $v_1 = -v_3$ and $\Delta w_j = \alpha \cdot y_j \cdot (z^{\text{target}} - z_1)$ for $j \in \{1, 2, 3\}$, it can be shown that $y_1 = -y_3$. During output layer training for CGNN the output layer weights are initialized to 0 and then perceptron training is used to modify the output layer weights while the middle layer weights are held constant. CGNN uses the absolute value of the weights w_1, w_2 , and w_3 to determine the fitness of weight vectors v_1, v_2 , and v_3 . It will be shown that the relation $|w_1| \leq |w_2| \geq |w_3|$ is maintained during output layer training.

Without loss of generality $y_1 = 1$, and $y_3 = -1$. Because of the threshold function $z^{\text{target}} = -1$ or $+1$. If $z = z^{\text{target}}$ then $\Delta w_i = 0$ for all $i \in \{1, 2, 3\}$. If $z \neq z^{\text{target}}$ then $\Delta w_i = \begin{cases} -2\alpha y_i & \text{if } z^{\text{target}} = -1 \\ 2\alpha y_i & \text{if } z^{\text{target}} = +1 \end{cases}$. Also, if $z \neq z^{\text{target}}$ then $z = -z^{\text{target}}$, because z and z^{target} can only be -1 or $+1$. Since initially, $w_i = 0$ for $i \in \{1, 2, 3\}$, and $\Delta w_i = 0$ when $z = z^{\text{target}}$, there will be no weight adjustments when the network is performing correctly. This means that if the network is performing correctly the relationship $|w_1| \leq |w_2| \geq |w_3|$ will hold.

The remaining cases to be considered deal with weight adjustments when the network produces an incorrect output. First, consider cases where the training data is consistent. Consistent training data will always have the same z^{target} whenever a given set of inputs appears in the training dataset. If $z^{\text{target}} = -1$, then $\Delta w_1 = -2\alpha$, $\Delta w_2 = -2\alpha y_2$, and $\Delta w_3 = +2\alpha$. If $z^{\text{target}} = +1$, then $\Delta w_1 = +2\alpha$, $\Delta w_2 = +2\alpha y_2$, and $\Delta w_3 = -2\alpha$. y_2 is -1 or $+1$. Because only 1 middle layer node y_2 is used to compute z_1 , there must be a linear relationship between y_2 and z_1 .

Thus, training adjustments can be divided into two types, each with two subtypes. In a type 1 adjustment $z = -y_2$. In type 1 subtype 1, $z = +1$ and $y_2 = -1$. In type 1 subtype 2, $z = -1$ and $y_2 = +1$. In type 2 adjustments $z = y_2$. In type 2 subtype 1 $z = +1$ and $y_2 = +1$, and in type 2 subtype 2 $z = -1$ and $y_2 = -1$. Type 2 adjustments and type 1 adjustments will not both occur in a consistent training dataset, because the data would then be inconsistent. A training dataset that causes only type 1 adjustments is called a type 1 dataset and a training dataset that causes only type 2 adjustments is called a type 2 dataset.

For type 1 subtype 1 adjustments $\Delta w_1 = -2\alpha$, $\Delta w_2 = +2\alpha$, and $\Delta w_3 = +2\alpha$. For type 1 subtype 2 adjustments $\Delta w_1 = +2\alpha$, $\Delta w_2 = +2\alpha$, and $\Delta w_3 = -2\alpha$. For type 2 subtype 1 adjustments $\Delta w_1 = -2\alpha$, $\Delta w_2 = -2\alpha$, and $\Delta w_3 = +2\alpha$. Lastly, for type 2 subtype 2 adjustments $\Delta w_1 = +2\alpha$, $\Delta w_2 = -2\alpha$, and $\Delta w_3 = -2\alpha$.

Now, consider inconsistent data. Inconsistent data in a type 1 dataset will cause an adjustment of type 2, and inconsistent data in a type 2 dataset will cause an adjustment of type 1. Let p_u be the probability of inconsistent data, and let p_1 be the probability of type 1 data. This means that $1 - p_u$ is the probability of consistent data and $1 - p_1$ is the probability of type 2 data.

Using this information the values of the w_1 , w_2 , and w_3 at a finite time t can be derived. In this derivation all sums are considered to be over the range from 1 to t . For

type 1 data

$$w_1 = \left[\left(\sum 2\alpha \right) p_1 + \left(\sum -2\alpha \right) (1 - p_1) \right] (1 - p_u) + \left[\left(\sum -2\alpha \right) p_1 + \left(\sum 2\alpha \right) (1 - p_1) \right] p_u,$$

$$w_1 = 2\alpha t (1 - 2p_1) (1 - 2p_u),$$

$$w_2 = \left(\sum 2\alpha \right) (1 - p_u) + \left(\sum -2\alpha \right) p_u,$$

$$w_2 = 2\alpha t (1 - 2p_u),$$

$$w_3 = \left[\left(\sum -2\alpha \right) p_u + \left(\sum 2\alpha \right) (1 - p_1) \right] (1 - p_u) + \left[\left(\sum 2\alpha \right) p_1 + \left(\sum -2\alpha \right) (1 - p_1) \right] p_u,$$

$$w_3 = 2\alpha t(2p_1 - 1)(2p_u - 1),$$

and for type 2 data

$$w_1 = [(\sum -2\alpha)p_1 + (\sum 2\alpha)(1 - p_1)](1 - p_u) + [(\sum 2\alpha)p_1 + (\sum -2\alpha)(1 - p_1)]p_u,$$

$$w_1 = 2\alpha t(1 - 2p_1)(1 - 2p_u),$$

$$w_2 = (\sum -2\alpha)(1 - p_u) + (\sum 2\alpha)p_u,$$

$$w_2 = 2\alpha t(1 - 2p_u),$$

$$w_3 = [(\sum 2\alpha)p_1 + (\sum -2\alpha)(1 - p_1)](1 - p_u) + [(\sum -2\alpha)p_1 + (\sum 2\alpha)(1 - p_1)]p_u,$$

$$w_3 = 2\alpha t(2p_1 - 1)(2p_u - 1).$$

It is assumed that $p_1 \approx .5$ which means that data of subtype 1 and subtype 2 are of approximately equal distribution in the training data. It is also clear that $p_u \ll .5$ because of its definition. Substituting the derived values of the weights into $|w_1| \leq |w_2| \geq |w_3|$ gives $|1 - 2p_1| \leq |1| \geq |2p_1 - 1|$ for type 1 data, and $|1 - 2p_1| \leq |-1| \geq |2p_1 - 1|$ for type 2 data.

Since p_1 has a range of 0 to 1, this gives a maximum of 1 for all weights for both types of data. It should also be noted that if p_1 is close to .5, then the approximate fitness assigned to the canceling vectors will be 0 and the vector that is utilized will have a fitness of 1.

These relations hold for the constraints given for p_1 , so this proves that for a cooperative genetic neural network with 3 middle layer nodes and no bias nodes, canceling weight vectors in the middle layer will not be awarded a fitness that is greater than the fitness assigned to the weight vector that is utilized.

It should also be noted that CGNN works with larger amounts of middle layer neurons than are necessary so that the genetic algorithm has a large enough population to work with. There is a high probability that these canceling weight vectors will be lost during crossover, because they are opposites.

Another way of examining CGNN's fitness function and its correctness is to look at the utility of a middle layer weight vector from another viewpoint. Another way of gauging

the utility of a middle layer neuron is by looking at how much effect severing the connections out from that neuron has on the resulting output of the whole network. By considering the output layer feed forward equation and an input dataset with average distribution it is evident that severing stronger connections (those with higher weights) has a total effect that is much greater than severing connections with lower weights. This is because severing a connection is, in effect, dropping a term from the output layer equation. The larger the constant associated with the term, the bigger the overall effect during a training pass. By looking at the fitness function in this way, it also seems to hold up to the conditions for a good fitness function. This is also another answer to the problem of two neurons that have outputs that cancel one another in the final equation. They are each very necessary for the overall performance of the network, because if one of them is lost, the overall results produced by the network will be greatly affected, although a network using them would not be minimal.

These two different ways of looking into the fitness function used in CGNN have shown that the fitness function of CGNN indicates utilization and that utilization directly indicates contribution towards performance, which in turn serves to produce more weight vectors during successive generations that contribute toward better network performance. This means that the performance of the network will get better with training.

F. Summary

The CGNN algorithm starts with random weights in the middle layer. It then makes a small number of passes through the training dataset and adjusts the output layer weights using the perceptron training rule. A variation of the simple genetic algorithm that uses the stochastic remainder algorithm for selection is used to create a new set of middle layer

weights. The process repeats, making passes through the training data for a number of generations.

The fitness for a middle layer weight vector is equal to the sum of the absolute values of the weights on the connections leading from the associated middle layer neuron to the output layer. CGNN relies on the fact that the its fitness function indicates the utilization of the middle layer weight vectors. It has been shown that because of this, CGNN will try to minimize the training error.

A complete proof of CGNN's convergence has not been constructed. This suggests that experimentation with this platform must be used to get a feeling for the convergence of the algorithm.

CHAPTER IV.

SIMULATION RESULTS AND CONCLUSIONS

A. Problem Selection and Benchmarks

The selection of benchmark problems for examining a new connectionist algorithm is complicated. The only standard is the XOR or parity problem, but this is considered a "toy" problem. The XOR problem is one of the simplest classification problems that is not linearly separable. There are no real standards, although some problems are common between two or more papers in this field. One method of selecting a benchmark is to look for a problem that cannot be solved using a common architecture like backpropagation. This was the approach taken by Karunanithi when the "two-spirals" problem was selected [13]. Also, some work has been done with encoder/decoder problems. CGNN and many other algorithms easily solve the XOR problem. CGNN does it in one generation. Contrived problems seem to be somewhat artificial, and so they may not show real-world performance.

The approach taken in testing CGNN was to try several different types of problems. First, CGNN was tested with a wide range of data using a Nx1 decoding problem generator. Next, CGNN was tested with 3 sets of data taken from the CMU-based collection of learning benchmarks for neural networks [9]. The last set of data selected falls into the contrived database category, and is based on a chaotic function which is difficult to learn because of its seeming randomness. The wide range of different types of training data shows CGNN's overall performance.

For each of the benchmark datasets, a raw dataset containing all data was obtained. The raw datasets were then row scaled so that the minimum and maximum dimension for each parameter was -1 and +1 respectively. Each benchmark dataset was

also modified so that it had 1 discrete output that is either +1 or -1. During testing the output from the backpropagation algorithm was thresholded so that an output below 0.5 was considered to be -1 and an output equal to or above 0.5 was considered to be +1. The training and testing datasets were then taken from this raw dataset by random sampling. All reported results are the average results from 5 trials using the same parameters and a different random sample of the raw dataset for both the training and testing datasets.

B. Simulation Platform

CGNN is computationally intensive on large problems. This is also true for BP and other NNs. Initial tests with both algorithms were performed on an IBM 386/SX/PC with a numeric coprocessor. The code, which is written in C, was then ported to a Silicon Graphics IRIS 4D/240 GTX which had become available for experimentation. This machine has performance in the 10 MFLOP range. Although the machine has 4 processors, the code runs on only 1 processor because it has not been optimized for four, and a vectorizing C compiler was not available. Some code development was also done on a Silicon Graphics Indigo workstation. The Indigo is binary compatible with the IRIS 4D, but it has only one CPU and achieves around 4 MIPS. Data reduction was done on an IBM compatible with a Pentium CPU and 8 megabytes of RAM.

C. General Parameter Settings

The CGNN algorithm and the backpropagation algorithm both contain many parameters. To aid in comparison of these two algorithms, and to show the overall performance of CGNN only one parameter was varied for each of these algorithms. A good general set of values for the other parameters was determined through a lot of initial

experimentation and tweaking. The parameter that was chosen to be varied for CGNN was the number of data points that the network was trained on, and the parameter that was varied for the backpropagation algorithm was the size of the middle layer. By gathering data with various training set sizes, the generalization capabilities of CGNN are shown, and by changing the size of the middle layer of the BP algorithm, its best performance is shown.

The perceptron training for the output layer of CGNN was run for a maximum of 20 epochs for each generation of the genetic algorithm and CGNN was run for 100 generations. This gives CGNN a maximum of 2000 output layer training epochs. The learning rate adjustment constant for the output layer was set to 0.01. Ten bits were used for the scaled integer representation of each weight in the middle layer. The activation threshold was set to 0.0. The mutation probability was set to 0.005 and the crossover probability was set to 0.9. The population size, which is also the number of weight vectors in the middle layer, was set to 100 and the breeding pool size was set to 50. CGNN was trained on 10, 50, 100, 150, and 200 data points for each benchmark.

The backpropagation algorithm was run for 2000 epochs. This number was chosen because it yielded similar execution times for backpropagation and the CGNN. It is also the maximum number of output layer training epochs that the CGNN algorithm was allowed. The alpha and beta learning rate adjustments were both set to 0.1. In addition the momentum term was set to 0.9. The backpropagation algorithm was tested with 10, 20, 30, and 40 middle layer nodes.

D. Automatically Created Artificial Data

The first data tested on CGNN was automatically created by a program called MAKEDATA which was written for this research. MAKEDATA creates a random feed-

forward network of the desired complexity and pushes data through it to produce training datasets. This approach to dataset creation allows for experimentation with datasets with a wide variety of complexity. An added advantage is that there is also a known solution for each dataset created. There are 11 input parameters, 1 output parameter, and 200 data points in the dataset that was used for this experiment. The random network contained 3 layers and used bimodal thresholding in the middle and output layers.

The best accuracy obtained for CGNN on this dataset was 77.69% and the worst accuracy on this dataset was 59.40%. The execution time increased with increasing training set size, from a minimum of 32.6 seconds to a maximum training time of 968.2 seconds.

E. Tic-tac-toe Endgame Database

A database that encodes the complete set of 958 possible board configurations that can occur at the end of a Tic-tac-toe game was used as a training set. Each board configuration is represented by a record that is a sequential listing of the cell contents of the board followed by a "positive" if the board represents a game in which x won or "negative" if x did not win. There are a total of 9 attributes for input and 1 for output. There are about 65.3% winning boards for x in the dataset. A program was written to convert this data into a format suitable for CGNN. Each input X was represented by a -1.0. Each O was represented by a +1.0, and each blank cell was represented by a 0.0. A win for "X" was converted to +1.0 and a loss for "X" was converted to a -1.0. The dataset contains 65.3% wins for "X".

This dataset has been used by several researchers. CITRE was applied to a 100 instance training set and 200 instance test set, in a study using different amounts of domain-specific knowledge [25]. Its highest degree of accuracy was 76.7% using the final

decision tree created for testing. Similar experiments, also using CITRE [26], included learning curves for up to 500-instance training sets and all of the database for testing. Accuracies in this instance reached above 90%.

D. W. Aha. (1991) used instance-based constructive induction with 70% of the data used for training and 30% used for testing evaluated over 10 trials. Results ranged from 82.0% for IB3 to 99.1% accuracy for IB3-CI [27].

The results for CGNN are shown graphically in figure 8. The worst accuracy is 60.8% in 94.8 seconds using 10 data points. The best accuracy is 82.02% which was obtained in 1391.59 seconds using 200 training data points. CGNN's accuracy went up when given more training data points.

Backpropagation was also trained on the dataset. The worst accuracy obtained using backpropagation was 85.00% in an elapsed time of 603 seconds using 10 middle layer neurodes. The best accuracy obtained was 85.93% using 40 middle layer nodes during a run of 1550 seconds.

The graph in figure 9 indicates that accuracy increases with middle layer size for backpropagation. Although better performance was obtained using more middle layer nodes, The difference between the lowest and highest accuracies was 0.94%.

F. Pima Indians Diabetes Database

The Pima Indians Diabetes Database was the second standard database chosen for testing the performance of the Cooperative Genetic Neural Network. This is a database of 768 data points. Each data point has 8 inputs and 1 output and represents a subset of medical data taken from a Pima Indian. The expected output that the network is trained to predict indicates whether or not the Indian has diabetes, and the inputs are various medical readings that were thought to be related to diabetes [9].

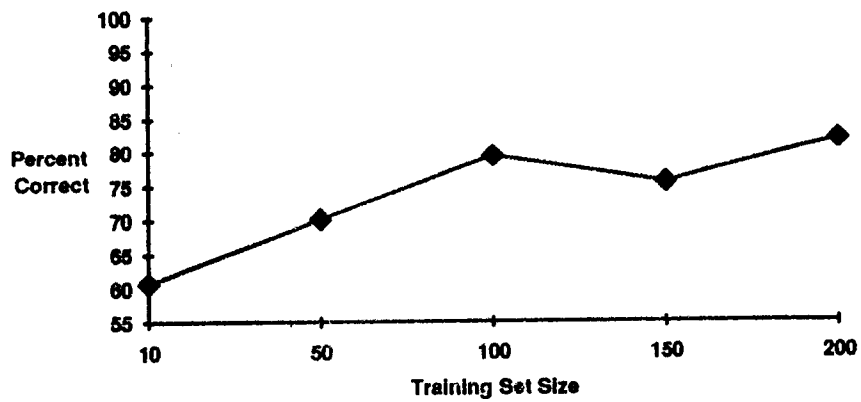


Figure 8. Effect of Training Set Size on Accuracy for CGNN on Tic-tac-toe Data

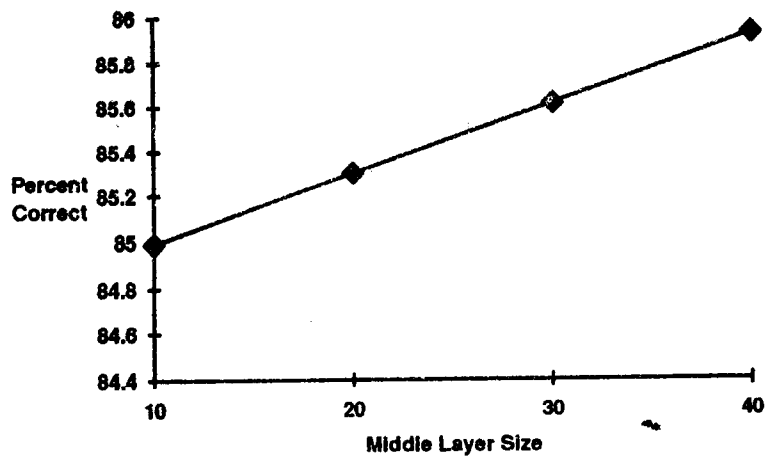


Figure 9. Effect of Middle Layer Size on Accuracy for Backpropagation on Tic-tac-toe Data

The worst accuracy obtained for CGNN was 68.18% in 387.6 seconds using 50 training data points. The best accuracy obtained was 70.99% during a run time of 1489.00 seconds.

The relationship between accuracy and training set size for CGNN using the Pima Indian diabetes data is shown in figure 10. It shows that the best accuracy was obtained using a population size of 150. There is a noticeable dip in accuracy between a population size of 10 and a population size of 150.

Figure 11 shows a trace of the total error on the last pass of each generation while CGNN is training on the pima diabetes database. This graph shows that the total error decreases initially, but that the convergence stops at generation 5. The pattern of error convergence represented by this graph occurred when training on all of the benchmark datasets.

The worst accuracy obtained for backpropagation was 72.27% in 498 seconds using 10 middle layer nodes. The best accuracy was 74.11% during a run time of 1048.00 seconds using 20 middle layer nodes.

G. Breast Cancer Databases

The third database selected from the CMU-based collection of learning benchmarks for neural networks contains breast cancer data. This breast cancer database was obtained from the University of Wisconsin Hospitals, Madison from Dr. William H. Wolberg [23,24]. There are 699 data points in the database. Each data point has 10 inputs and 1 output. The 10 inputs are taken from various medical tests thought to be related to breast cancer, and the output indicates whether the cancer was benign or malignant. The inputs represent the sample code number, clump thickness, uniformity of cell size,

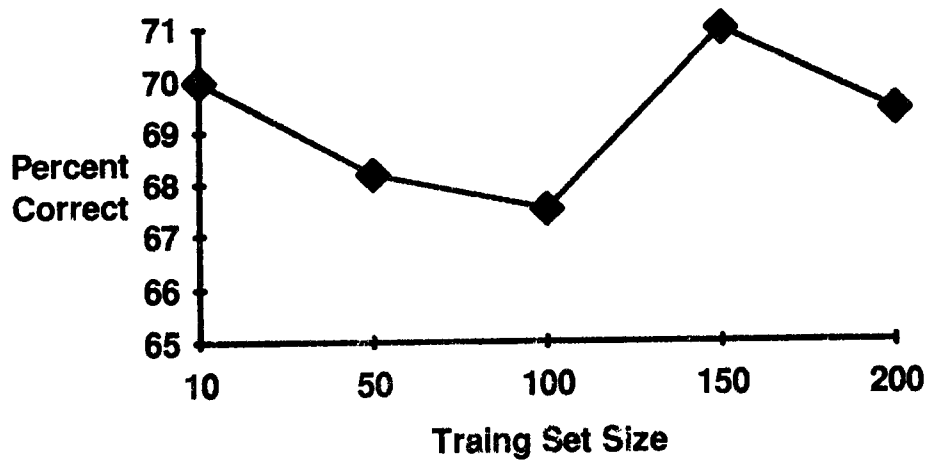


Figure 10. Effect of Training Set Size on Accuracy for CGNN on Diabetes Data

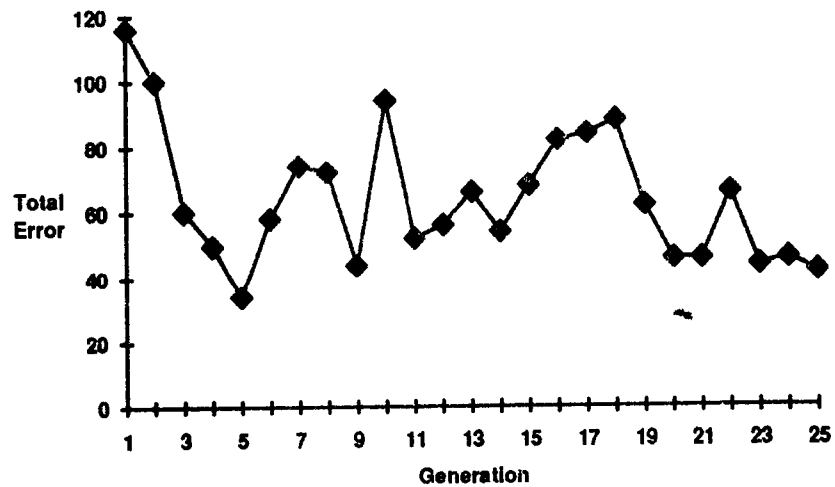


Figure 11. Changes In Total Error During Training of CGNN on Diabetes Data

uniformity of cell shape, marginal adhesion, single epithelial cell size, bare nuclei, bland chromatin, normal nucleoli, and mitosis.

This dataset has been growing continually as Dr. Wolberg gathers new data. Dr. Wolberg has used hyperplanes to try to separate the data. At the time it consisted of 369 patterns. Only one trial was attempted. Two pairs of parallel hyperplanes were found to be consistent with 50% of the data, and to achieve an accuracy of 93.5% on the remaining data. In addition, three parallel hyperplanes were found to be consistent with 67% of the data, with an accuracy of 95.9% on the remaining data [24].

The worst accuracy for CGNN on this dataset was 89.87% in 426 seconds using 50 training data points. The best accuracy for CGNN was 95.79%, used 200 training data points, and took 1642 seconds. The results are graphically depicted in figure 12. This graph shows that the network performs better when it has been trained on more data points.

The backpropagation algorithm had the worst accuracy (94.96%) when it was given 10 middle layer neurodes. This result was achieved in 897 seconds. The best accuracy was 95.85% in 1774 seconds using a middle layer of 40 neurodes.

The results obtained using backpropagation on this dataset are shown in the graph in figure 13. CGNN achieved 95.79% correct, while BP achieved 95.85% correct. For this performance CGNN took 1642 seconds and BP took 1774 seconds.

H. Predicting Chaotic Time Series

The final database chosen for testing CGNN was created with the purpose of testing a network's ability to predict future values of a chaotic time series, as described by Moody and Darken [4]. This is a difficult task to ask a network to perform, but many real-

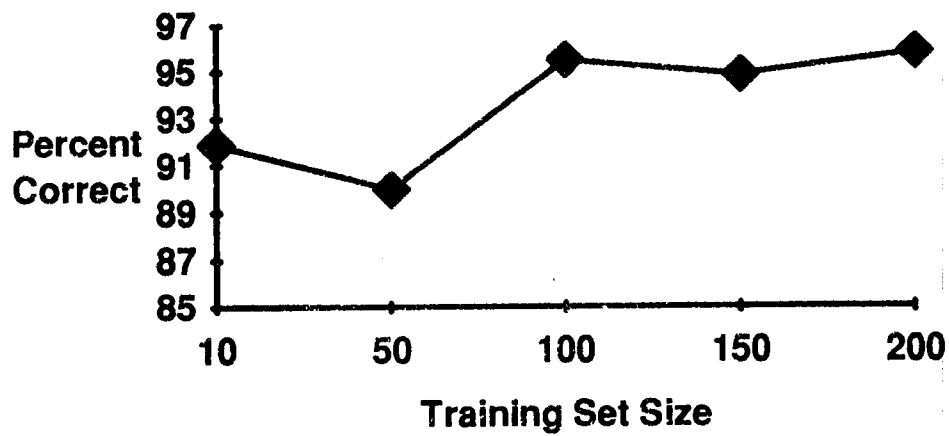


Figure 12. Effect of Training Set Size on Accuracy for CGNN on Breast Cancer Data

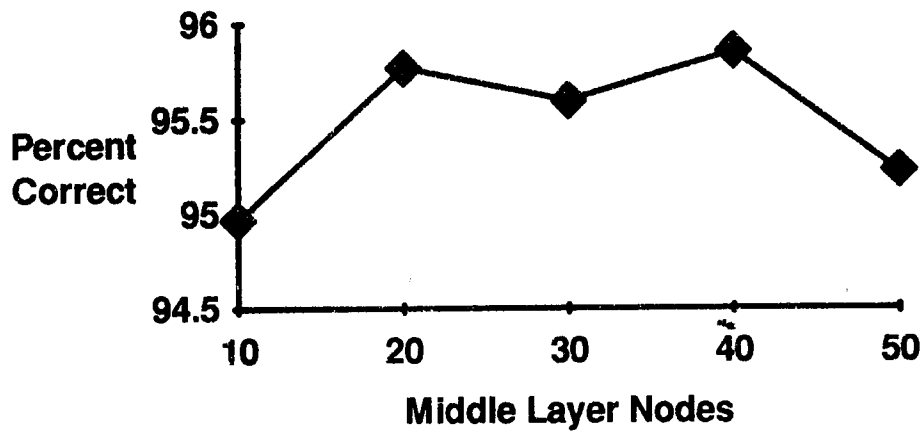


Figure 13. Effect of Middle Layer Size on Accuracy for Backpropagation on Breast Cancer Data

world problems are based on chaotic patterns. The chaotic time series described by Moody and Darken is based on the Mackey-Glass differential Equation (equation 14).

$$\frac{dx[t]}{dt} = -bx[t] + a \left(\frac{x[t-T]}{1 + x[t-T]^{10}} \right)$$

Equation 14. Mackey-Glass Equation

The Mackey-Glass equation has also been used by Matrinetz, et. al. for testing the "neural gas" Network [28]. They trained the network for 200,000 training steps and 100,000 training pairs. Each training point in their data represented $x[t]$, $x[t-6]$, $x[t-12]$, and $x[t-18]$.

To produce the training data, values chosen for a , b , and T were 0.2, 0.1, and 17 respectively. The substitution is shown in equation 15.

$$\frac{dx[t]}{dt} = -0.1x[t] + 0.2 \left(\frac{x[t-17]}{1 + x[t-17]^{10}} \right)$$

Equation 15. Revised Mackey-Glass Equation

Equation 14 is taken directly from Moody and Darken [4]. The specific parameters for equation 15 are taken from Choate [5]. The characteristic time (also called wavelength) of the series is approximately 50. The chaotic time series database was generated for 1000 data points each with 4 inputs and 1 output using numerical integration. Numerical integration is necessary to produce the time series, because using the difference equation with each output depending on the last introduces a lot of error after a few iterations. The 4 inputs for each point are taken around a time t in the chaotic series. The 4 inputs are $x[t]$, $x[t-D]$, $x[t-2D]$, and $x[t-3D]$ where D is 6. The output for the data point is $x[t+F]$, where F is 85. F was chosen to be greater than the characteristic time for the series. This is

important because traditional techniques are no better than chance at predicting the series beyond its characteristic period.

Since CGNN uses the perceptron training rule to produce its output, the output can only be +1 or -1. This means that it cannot be used to predict the actual value of the time series at $x[t+85]$. Because of this, the desired output for this dataset was modified so that it was either -1 or +1. When the desired output was below the average of the maximum output and the minimum output it was replaced with a -1, and when the desired output was equal to or above the average of the minimum and maximum desired outputs it was replaced with a +1. There is no proof that the transformed outputs are chaotic, but initial experimentation with both CGNN showed that these outputs were not easy to predict.

The results achieved using CGNN on the chaotic time series are graphically depicted in figure 14. The total error for the last training data pass for each of the training set sizes was 0.0, but that the accuracy increases as the training set size increases. The best accuracy was 92.82% in 167.60 seconds when the network was given 200 training data points. The worst accuracy was 83.90% in 7.8 seconds using 10 training data points.

Figure 15 shows the total error during the last pass made through the training data during each generation. This graph shows that CGNN converges to a minimum error at generation 12. It also shows that the error is higher for several generations after 12, but that the error again reaches this level at generation 28. There are big peaks in the data before generation 12. This kind of noise was noticeable in all training runs for the cooperative genetic neural network.

The best training accuracy obtained for backpropagation was 93.08% during a run time of 214 seconds using 10 middle layer nodes. The worst accuracy was 76.68% using 50 middle layer neurodes in an elapsed time of 4897 seconds.

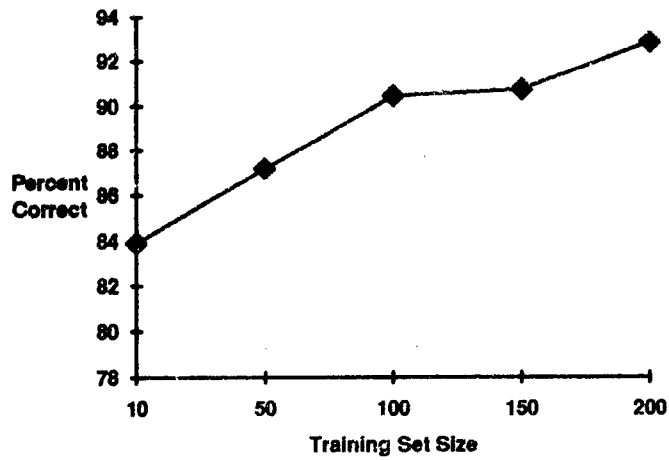


Figure 14. Effect of Training Set Size on Accuracy for CGNN on Time Series

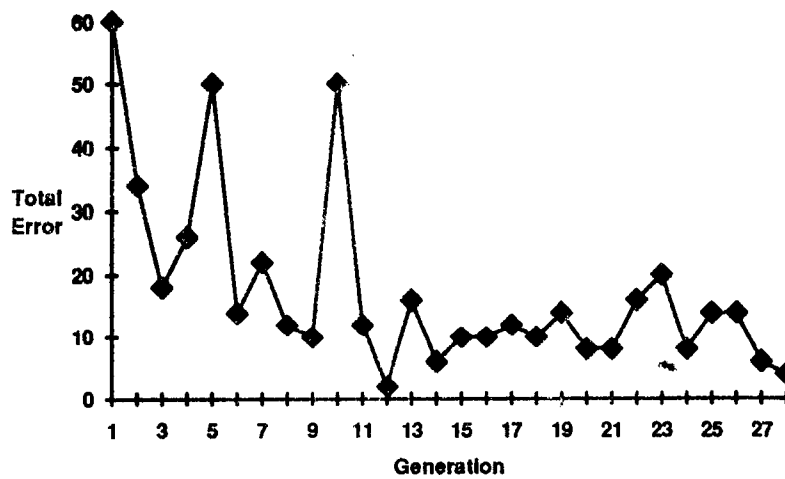


Figure 15. Changing Total Error During Training of CGNN on Time Series

I. Conclusions

On the average CGNN trained 2.81% faster than backpropagation, but backpropagation was 1.81% more accurate. On the tic-tac-toe dataset, CGNN trained 10.26% faster, but backpropagation was 3.91% more accurate. On the Pima Indian diabetes dataset backpropagation trained 42.08% faster and it was 3.12% more accurate. On the breast cancer dataset CGNN trained 21.36% faster and it was 0.03% more accurate. Lastly, on the chaotic time series dataset, CGNN trained 21.68% faster, but backpropagation was 0.26% more accurate.

CGNN trained faster on all the benchmark datasets except the Pima Indian diabetes dataset. The longer training time on the Pima Indian diabetes dataset may have been caused by a heavy load on the computer during CGNN training.

Backpropagation was more accurate on three of the benchmarks. This may be attributed to the sigmoidal threshold function it uses. The use of a similar threshold function in CGNN may improve its accuracy.

There are several changes to the CGNN algorithm that should make it faster and more accurate. Other areas of future research include more powerful output layer training rules, improved threshold functions, other middle layer feed-forward techniques, and parallelization.

As a contribution to creating standard benchmarks against which other architectures can be compared, the results of these experiments will be submitted to the CMU-based collection of learning benchmarks for neural networks.

LIST OF REFERENCES

LIST OF REFERENCES

- [1] Goldberg, David E. Genetic Algorithms in Search, Optimization, & Machine Learning. New York: Addison-Wesley, 1989.
- [2] Simpson, Patrick K. Artificial Neural Systems: Foundations, Paradigms, Applications, and Implementations. New York: Pergamon Press, 1989.
- [3] Goldberg, David E. and Whitehead, Bruce. A Short Course on: Genetic Algorithms and Neural Networks. Tullahoma, Tennessee: The University of Tennessee Space Institute, 1992.
- [4] Belew, Rickard K., McInerney, John, and Schraudolph, Nicol N. "Evolving Networks: Using the Genetic Algorithm with Connectionist Learning." La Jolla, California: Cognitive Computer Science Research Group, University of California at San Diego, June 1990.
- [5] Choate, Timothy D. Evolutionary Neural Network Design. Knoxville, Tennessee: The University of Tennessee, August 1991.
- [6] Vose, Michael D. "Models of Genetic Algorithms." Knoxville, Tennessee: Computer Science Department, January 1992.
- [7] Battle, David L. and Vose, Michael D. "Genetic: A Modular Genetic Algorithm." Knoxville, Tennessee: University of Tennessee, June 1992.
- [9] Murphy, P. M. and Aha, D. W. UCI Repository of machine learning databases [Machine-readable data repository]. Irvine, California: University of California, Department of Information and Computer Science, 1992.
- [10] Mishkoff, Henry C. Understanding Artificial Intelligence, Dallas, Texas: Texas Instruments Learning Center, 1986.
- [11] Caudill, Maureen, "Neural Networks Primer Part III." AI Expert, February 1988, pp. 17-23.
- [12] Lippmann, Richard P. "An Introduction to Computing with Neural Nets." IEEE ASSP Magazine, April 1987.
- [13] Whitley, D., Starkweather, T., and Bogart, C. "Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity." Fort Collins, Colorado: Colorado State University, 1993.
- [14] Harp, Steven A., Samad, Tariq, and Guha, Alok. "Designing Application-Specific Neural Networks Using the Genetic Algorithm." Advances in Neural Information Processing Systems 2, San Mateo, California: Morgan Kaufmann, 1990, pp. 447-454.

- [15] Bornholdt, Stefan, and Graudenz, Dirk. "General Asymmetric Neural Networks and Structure Design by Genetic Algorithms." Neural Information Processing Systems 2, San Mateo, California: Morgan Kaufmann Publishers, pp. 327-334, 1991.
- [16] Dominic, S., Das, R., Whitley, D., and Anderson, C. "Genetic Reinforcement Learning for Neural Networks." IJCNN-91-Seattle: International Joint Conference on Neural Networks, IEEE, July 1991, pp II-71-II76.
- [17] Whitehead, Bruce A., and Choate, Timothy D. "Evolving Space-Filling Curves to Distribute Radial Basis Functions Over an Input Space." IEEE Transactions on Neural Networks, Vol. 5, No. 1, January 1994. pp. 1-9.
- [18] Fogel, David B. "Using Evolutionary Programming to Create Neural Networks that are Capable of Playing Tic-tac-toe." International Conference on Neural Networks, pp. 875-880, 1993.
- [19] Koza, John R., and Rice, James P. "Genetic Generation of Both the Weights and Architecture for a Neural Network." IEEE International Joint Conference on Neural Networks, July 8-12, 1991, Volume II., pp. 397-404.
- [20] Kitano, Hiroaki. "Empirical Studies on the Speed of Convergence of Neural Network Training using Genetic Algorithms." Proceedings of the Eighth National Conference on AI, MIT Press, Cambridge, Massachusetts, Volume Two, pp.134-145, 1990.
- [21] Karunanithi, Nachimuthu, Das. Rajarshi. and Whitley, Darrell. "Genetic Cascade Learning for Neural Networks." International Workshop on Combinations of Genetic Algorithms and Neural Networks, IEEE Computer Society Press, Los Alamitos, California, 1992.
- [22] Tanimoto, Steven L. The Elements of Artificial Intelligence: An Introduction Using Using LISP. Rockville, Maryland: Computer Science Press, 1987.
- [23] Mangasarian, O. L., and Wolberg, W. H. "Cancer diagnosis via linear programming.", SIAM News, Volume 23, Number 5, September 1990, pp 1, 18.
- [24] Wolberg, William H. and Mangasarian, O. L. "Multisurface method of pattern separation for medical diagnosis applied to breast cytology." Proceedings of the National Academy of Sciences, Volume 87, December 1990, pp 9193-9196.
- [25] Matheus, C. J. and Rendell, L. A. "Constructive induction on decision trees." Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Morgan Kaufmann, Detroit, Michigan. 1989, pp 645-650.
- [26] Matheus, C. J. "Adding domain knowledge to SBL through feature construction." Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI Press, pp. 803-808.

- [27] Aha, D. W. "Incremental constructive induction: An instance-based approach." Proceedings of the Eighth International Workshop on Machine Learning, Morgan Kaufmann, 1991, pp. 117-121.
- [28] Martinetz, Thomas M., Berkovich, Stanislav G., and Schulten, Klaus J. "'Neural-Gas' Network for Vector Quantization and its Application to Time Series Prediction." IEEE Transactions on Neural Networks, Volume 4 Number 4, 1993, pp. 558-569.
- [29] Widrow, B. "An adaptive 'adaline' neuron using chemical 'memistors'." Stanford Electronics Laboratory Technical Report, 1553-2, 1960.
- [30] Rosenblatt, F. "Perceptron Simulation Experiments," Proceedings of the IRE, vol. 48, 1960, pp. 301-309.
- [31] Waterman, Donald A. A Guide to Expert Systems. Massachusetts: Addison-Wesley Publishing Company, 1986.
- [32] Caudill, Maureen. "Evolutionary Neural Networks." AI Expert, March 1991, pp. 28-33.

VITA

Sean Lane Fuller was born in Hopkinsville, Kentucky on December 25, 1965. He attended elementary school at Sinking Fork Elementary and graduated from Christian County High School in May 1983. He entered Murray State University in August of 1983 and graduated on May 1989 with a Bachelor of Science degree in Computer Science. In May of 1988 he accepted employment with OAO Corporation at the Arnold Engineering and Development Center as a database administrator. He began graduate studies at the University of Tennessee Space Institute in August of 1988.

Mr. Fuller is currently employed with OAO Corporation as a UNIX systems programmer and resides with his wife Terri in Manchester, Tennessee.