A NEW HYBRID LEARNING ALGORITHM FOR DRIFTING ENVIRONMENTS

THESIS

Presented to the Graduate Council
of Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Enumulapally Anil Kumar, B.E
ANIL K. ENUMULAPALLY

San Marcos, Texas
August 2005

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to Dr. Khosrow Kaikhah. Without his invaluable knowledge and patience I could not have accomplished this. His encouragement and involvement have been extremely helpful in completing this thesis. His mathematical skills and in-depth knowledge in neural networks have been priceless in this research.

My thanks and gratitude also go to Dr. Carol Hazlewood for her patience, advice and encouragement. I appreciate Dr. Hazlewood extending her cooperation in meeting the deadline for the research, even under tough family conditions.
I thank Dr. Moonis Ali for his support and his valuable time.

I would also like to thank my friends Chandu, Kishore, Narsi, Naveen and Nazir for their support, enthusiasm and belief in my abilities.

I express my special gratitude to my mother Surekha and my brother Ramakrishna for their love, care and support. I also appreciate their patience and enduring me and all that comes with me, through this thesis.

Above all I convey my prayers to Sri Saibaba and His Holiness Sri Sri Ravi Shankar for their grace and love.

This manuscript was submitted on July 28, 2005.

TABLE OF CONTENTS

List of Figures

ABSTRACT

A NEW HYBRID LEARNING ALGORITHM FOR DRIFTING ENVIRONMENTS

by

Anil Kumar Enumulapally

Texas State University-San Marcos

August 2005

SUPERVISING PROFESSOR: KHOSROW KAIKHAH

An adaptive algorithm for drifting environments is proposed and tested in simulated environments. Two simple but powerful problem solving technologies – Neural Networks and Genetic Algorithms with Online Learning, help the artificially intelligent agents to adapt to a changing environment. Neural networks and genetic algorithms are combined to evolve weights, architecture, and learning rules for the generation of efficient networks. Online learning helps these networks to capture the dynamics of a changing environment efficiently. Supervised learning is achieved using a variation of regular backpropagation that works on dynamic random networks.

Our algorithm proposes two types of online learning, namely local online learning which requires a pre-defined training set and global online learning which does not It is shown that both types of online learning improve the performance of networks to capture subtleties of the varying environments.

The algorithm's efficiency is demonstrated using a mine sweeper application. Different learning technologies have been compared. The results establish that online learning within the evolutionary process is the most significant factor for adaptation and is far superior to evolutionary algorithms alone. The evolution and learning work in a co-operating fashion to produce excellent results in short time. Offline learning is observed to increase the average fitness of whole population. It is also demonstrated that online learning is self sufficient and can achieve results without any pre-training stage. When mine sweepers are able to learn online, their performance in the drifting environment is significantly improved.

CHAPTER 1 INTRODUCTION

1.1 Introduction of the Problem

The objective of Artificial Intelligence is to support the notion that an intelligent system can demonstrate learning and respond like a human. In other words the program has to pass the "Turing test". Most of the intelligent agents do not adapt to the changes in the environment, as they are designed for a particular scenario and expect few deviations from it. These conditions do not exist in real time dynamic environments. The best solution for intelligent systems in real time drifting environments is to design and apply technologies such as Neural Networks and Genetic Algorithms that mimic the nature. Artificial neural networks and genetic algorithms are two relatively young research areas. Neural networks are massively parallel distributed processors that perform data mapping efficiently. Genetic algorithms attempt to apply evolutionary concepts to the function optimization capabilities of neural networks and are useful in searching large and complex environments. In recent times much research has been undertaken to combine these two important and distinct areas (Yao, 1999).

Evolution and learning are the two most fundamental processes of adaptation and the environment is a vital component of the adaptation process. If the environment were relatively static, there might be little need for learning to evolve. But in real time systems, generally, environments are dynamic and individuals need general adaptive mechanisms to cope with arbitrary environments. In this way, a diverse environment encourages the evolution of learning.

Our algorithm mimics human evolution and development. We have successfully implemented complete evolution and online learning to achieve effective design automation of neural networks with the ability to adapt to the drifting environments. Our experimental results demonstrate the ability of our algorithm to evolve efficient neural networks with simple architectures in few hundreds of generations.

1.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are highly simplified models of the brain. They consist of a combination of neurons and synaptic connections, which are capable of passing data through layers. ANNs posses a generalization property and are tolerant to noise in datasets. Neural networks have been successfully applied to perform regression, classification, control and prediction tasks in a variety of scenarios and architectures.

ANNs can be classified into two categories depending on their connectivity.

- Feedforward ANNs
- Feedback ANNs

Feedforward ANNs allow signals to travel one way only; from input to output. There is no feedback (loops), i.e. the output of any layer does not affect the same layer. Feedforward ANNs end to be straightforward networks that associate inputs to outputs. Feedback ANNs can have signals traveling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic, and the state of a network is continuously changing until it reaches equilibrium.

Neural networks, with their remarkable ability to work with complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques.

1.3 Learning in Artificial Neural Networks

Learning in ANNs is accomplished by adapting the synaptic strengths to the environment. Once a network has been designed for a particular application, it is ready to learn. To start the learning process the initial weights are chosen randomly. Artificial neural network learning algorithms are primarily divided into supervised and unsupervised.

1.3.1 Supervised Learning

In supervised learning, the training data consist of many pairs of input/output training patterns. Therefore, the learning will benefit from the assistance of the teacher (the desired output). The most widely used supervised learning algorithm is Backpropagation. In the training stage of the network, each input pattern is presented to the network, and fed forward through all the layers to the output layer. The actual output is then compared with the desired output corresponding to the input so that an error is computed and propagated backwards through the layers for the adjustments of the weights and thresholds. The process is repeated for all input/output patterns until the mean squared error for all patterns is less than a specified value. After the training phase is complete, it can recall the stored patterns, given an input pattern with a certain level of noise.

1.3.2 Unsupervised Learning

In unsupervised learning, the training set consists of input patterns only. Therefore, the network is trained without benefit of any teacher. Unsupervised neural networks "learn" from correlations of the input. Hebbian learning and adaptive learning are unsupervised.

As an example of adaptive learning, if a new pattern is determined to belong to a previously recognized cluster, then the inclusion of the new pattern into that cluster will affect the representation (e.g., the centroid) of the cluster. This will, in turn, change the weights characterizing the classification network. If the new pattern is determined to belong to none of the previously recognized clusters, then (the structure and the weights of) the neural network will be adjusted to accommodate the new class (cluster).

Training algorithms for ANNs can be broadly classified into two types.

a. Batch or Offline
b. Stochastic or Online

The batch training of ANNs is considered as the classical machine learning approach: a set of examples is used for learning an approximation function, before the network is used in the application. Batch learning can be viewed as the minimization of an error function E, to find a set of weights W such that $W=\min_{w \in R} E(w)$ where the function E is defined as the sum of the squared error over the entire training set.

In online training, the function E is pattern based and is defined as the instantaneous mean squared error function with respect to the currently presented training example. In this case, the ANN weights are updated after the presentation of each training example, which may be sampled with or without repetition. Online learning is appropriate for either problems with large training sets or tasks that slowly vary with respect to time. It helps escaping local minima and provides a more natural approach for learning by continuously adopting in a changing environment.

1.4 Genetic Algorithms

Genetic Algorithms (GAs) are modeled loosely on the principles of evolution via natural selection. These algorithms encode a potential solution to a problem on a simple chromosome-like data structure and apply genetic operators to these chromosomes to preserve critical information. GAs are widely used as function optimizers and can also be applied to a broad range of applications.

The traditional theory of GAs (Holland, 1975) assumes that, at a very general level of description, GAs work by discovering, emphasizing, and recombining good "building blocks" of solutions in a highly parallel fashion. The idea here is that good solutions tend to be made up of good building blocks—combinations of bit values that confer higher fitness on the strings in which they are present.

An implementation of a genetic algorithm begins with a population of random chromosomes and members of current population and gives rise to the next generation population by means of reproduction, mutation, or crossover, which are patterned after processes in biological evolution. At each step the chromosomes in the current population are evaluated relative to a given numerical measure called fitness. The most fit chromosomes are selected probabilistically as seeds for producing the next generation. Chromosomes in GAs are often represented by bit strings, so that they can be manipulated easily by genetic operators.

The popularity of GAs is motivated by the following:

- Evolution is known to be a successful, robust method for adaptation within biological systems.
- GAs can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.
- GAs are easily parallelized.
- GAs are very efficient at optimizing functions.

A genetic algorithm must contain five components

I. ***Representation:***

Chromosomal representation of solutions is problem dependent Representation is a key issue because GAs directly manipulate coded representations of problems

II. ***Evaluation Function:***

Evaluation function is problem dependent. Evaluation functions provide a measure of individual's performance.

III. ***Population:***

Choosing an appropriate size for population of initial solutions is very important and also a difficult task. Very large and very small population sizes have disadvantages. Generally, population is chosen at random.

IV. ***Genetic Operators:***

The three primary operators are Selection, Crossover and Mutation. The effectiveness of a GA depends on the combination and appropriate use of these operators.

a. Selection (Reproduction):

This operator selects the solutions for next generation from the current generation. Sometimes other operators are applied before we form the next generation. In such cases the population reproduced is called the intermediate population.

b. Crossover:

The crossover operator produces two new offspring from two parent strings by copying selected bits from each parent. The bit at position i in each offspring is copied from the bit position 'I' in one of the two parents.

Depending on the crossover mask, it can be divided into the following:

i. Single point crossover
ii. Two-point crossover
iii. Uniform crossover

c. Mutation:

By modifying one or more of an existing individual's gene values, mutation creates new individuals to increase variety in the population. The mutation ensures that the probability of reaching any point in the search space is never zero

V. Parameters:

Executing a genetic algorithm requires setting a number of parameter values. Finding ideal settings for a problem is a difficult task Some of the parameters are crossover rate, mutation rate, and population size and selection strategy.

1.5 Problem Description

Learning and evolution are two fundamental forms of adaptation. Neural networks are inefficient when dealing with large complex problems that generate many local optima. Genetic algorithms deal with complex problems efficiently. However, they are very poor at fine-tuning the solution where the ANN algorithms perform the best. Obviously these two strategies have their own strengths and weaknesses. One possible way of constructing an efficient algorithm is to allow these two strategies to complement each other. These approaches can be combined to achieve a more flexible network that can perform better in varying situations.

There are different approaches one can take in combining the ANNs with GAs. In the supportive approach, ANNs and GAs are applied at different stages. GAs are commonly used to reduce the dimensionality of data space. In the collaborative approach, GAs and ANNs are integrated into a single system in which population of ANNs is evolved. But designing a hybrid system is not sufficient for the drifting environments. In standard hybrid algorithms, a population of networks is evolved to perform a task, and the best fit network is found. This network is fixed and is used for future instances of the problem. Networks evolved this way do not handle dynamic environments very well.

Living organisms not only evolve but also learn in their lifetime and change according to the changes in the surroundings and their needs. So the true adaptation to the surroundings must include life-long (online) learning. Without online learning the process of adaptation to the environment, in drifting environments, is incomplete. Online learning is generally used in applications where there are very large and redundant training sets, or where the environment changes slowly over time. Moreover, online learning helps escaping local minima and provides a more natural approach for learning time varying functions and adapting to a continuously changing environment. Sutton pointed out, "Online learning is essential if we want to obtain learning systems as opposed to merely learned ones". Hence, hybrid algorithms that employ online learning are required to achieve the task of true adaptation. Despite the abundance of methods for learning from examples, there are only few that can be used effectively for online learning. In a majority of approaches evolutionary principles are used in conjunction with ANN training to formulate the problem as finding weights of a fixed architecture.

This approach leads to the following major sources of noise

- Due to the random initialization of weights the same genotype (the ANN without any weight information) may have quite different fitness.
- Different training algorithms may produce different training results even from the same set of initial weights.

To alleviate these problems, we need to evolve the ANN architectures and weights simultaneously. We propose an algorithm which not only uses the best principles of learning and evolution but also employs online learning for the agents.

1.6 Solution Strategy

The evolution of artificial neural networks can be classified according to the goals of evolution. There are three basic approaches by which we can combine learning with evolution.

- Evolution of weights
- Evolution of architectures
- Evolution of learning rules or transfer functions

As evolution of weights, architectures, or learning rules alone do not yield required performance, all three approaches must be combined to design a truly flexible network This also reduces the human intervention in the network design. Combining all three evolutionary approaches with online learning result in the adaptation. By combining evolutionary approaches with online learning, we have developed a hybrid algorithm that can adapt to the changing environments.
In our approach, the artificial intelligent agent is equipped with a neural network brain which learns in two different stages:

- Offline Learning
- Online Learning

In the offline learning stage, we integrate network learning process with evolution. In this stage learning is used as one of the genetic operators. We use the modified backpropagation algorithm with all three operators of GAs on the population. The genetic operators are used only if they are needed.

In the online learning stage, network learning and evolution are applied at different stages. Online gradient descent method is used for learning and GA operators are used to produce a better population for learning process. Learning and evolution are applied to the entire population.

In each stage, GAs are used to evolve the weights and the architecture. Online gradient descent and backpropagation use adaptive step size to evolve the learning rule.

1.5.1 Offline Learning

Step 1: Represent the networks in chromosome form where weight and network evolutions are easily performed.

Step 2: Generate a population of minimal genomes with and without hidden neurons. As our networks are random, we do not have hidden layers.

Step 3. Generate phenotypes or actual networks with all nodes, synapses, and their connections.

Step4: Train the networks using modified backpropagation algorithm by applying the sample data sets for fixed number of iterations.

Step 5: Use genetic operators (mutation and crossover) on the population to create the better networks for population.

Step 6: Evaluate the fitness of each network. Better fit networks are included in the population, which is passed to the next stage. All the other networks are discarded

Step 7: Group the networks into different species. This is required to avoid the "Crowding" effect.

Step 8: Select the best fit 'N' networks for the Online Phase.

This is a onetime process for a network and is applied only when one is generated. Offline learning uses a stepwise approach in which learning, crossover, and mutation may be used if required. Learning involves the modified backpropagation algorithm. A few examples from different environments are applied to the network for a fixed number of loops. Crossover is performed using innovation numbers for the connections and neurons. Innovation number works as an identifier for the synapses and neurons among all networks. Mutation is used to add a link or node, or to delete a link or node

1.5.2 Online Learning

Step1: The ‘N’ networks from the offline stage are trained.

Step 2: The agent is equipped with sensors, a number of vectors that collect information from the environment. Using these sensors the agent gets the inputs from the environment.

Step 3: The agent uses the online gradient descent method to learn the environment and modifies network weights. This helps in adapting to the varying surroundings

Step 4: After a fixed number of time units the networks are modified using genetic operators and a more fit population is generated from the current population of networks for the next generation.

Steps 5: The steps from 2 to 4 are applied repeatedly. The agent gets smarter and the result is achieved.

This stage has two different phases that toggle, i.e. evolution and learning. Genetic operators are applied if the mutation or crossover rate constant is less than a certain threshold generated. The agent uses the online gradient descent method to learn the environment. We employ a history sensitivity function that decreases the amount of learning as the time elapses. We designed an online gradient descent method for evolved networks with hidden nodes.

CHAPTER-2 INTRODUCTION TO NEURAL NETWORKS AND GENETIC ALGORITHMS

Overview

This chapter provides the basis for the underlying concepts of the proposed algorithm. It provides information about Artificial Neural Networks and their learning algorithms, and Genetic Algorithms and their operators in detail.

2.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example. They resemble the brain in two respects:

i. Network acquires knowledge through a learning process.
ii. Inter-neuron connection strengths known as synaptic weights are used to store the knowledge.

According to the DARPA Neural Network Study (AFCEA International Press, 1998), an artificial neural network is a system composed of many simple processing elements operating in parallel whose function is determined by network structure, connection strengths, and the processing performed at computing elements or nodes.

According to Haykin, S. (Neural Networks: A Comprehensive Foundation, 1998) ANNs have been applied to an increasing number of real-world problems of considerable complexity. Their most important advantage is in solving problems that are too complex for conventional technologies -- problems that do not have an algorithmic solution or for which an algorithmic solution is too complex to be found. In general, because of their abstraction from the biological brain, ANNs are well suited to problems that people are good at solving, but for which computers are not. These problems include pattern recognition and forecasting (which requires the recognition of trends in data).

2.1.1 Biological Inspiration

The study of ANNs has been inspired by biological learning systems that are built from very complex webs of interconnected neurons. The human brain contains a very large number (approximately 10^{11}) of neurons that are massively interconnected with other neurons. Each neuron is a specialized cell which can propagate an electrochemical signal. The basic computational unit in the nervous system is the nerve cell, or neuron. A neuron has:

- Dendrites (inputs)
- Cell body
- Axon (output)

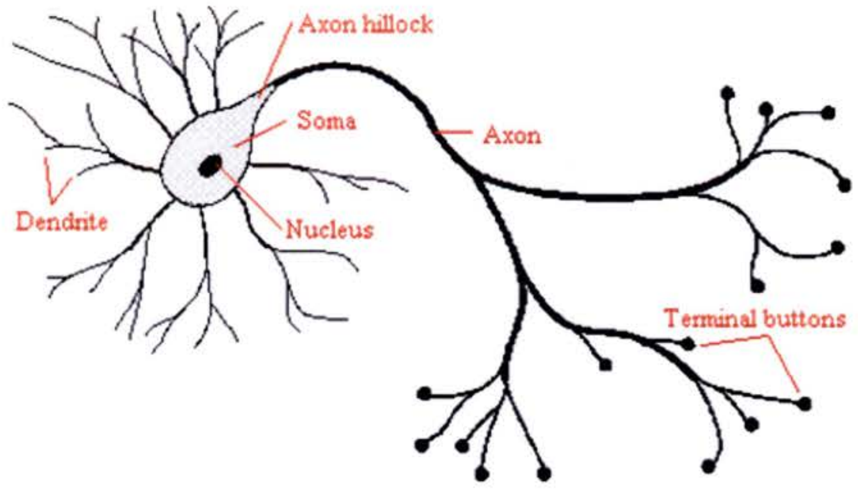


Figure 2.1 Schematic of biological neuron

The neuron has a branching input structure (the dendrites), a cell body, and a branching output structure (the axon). The axons of one cell connect to the dendrites of another via a synapse. A neuron receives input from other neurons (typically many thousands). Once the sum of all inputs exceeds a critical level, the neuron discharges a spike, an electrical pulse that travels from the body, down the axon, to the next neuron(s) (or other receptors). This structure indicates that the information processing capabilities of biological neural systems are the result of highly parallel processes that are distributed over many neurons. Motivation for ANNs is to capture this kind of highly parallel computation based on distributed representations. While ANNs are loosely inspired by biological neural systems, there are many complexities of biological neural systems that are not modeled by ANNs.

2.1.2 Architecture of Artificial Neural Networks

There many different types of artificial neural network structures, each of which has very different computational properties. An artificial neural network is composed of a number of neurons or nodes connected through links or synapses. The structure of the network depends on the way the neurons or nodes are connected to each other. The general structure of a neuron is shown in Figure 3.2

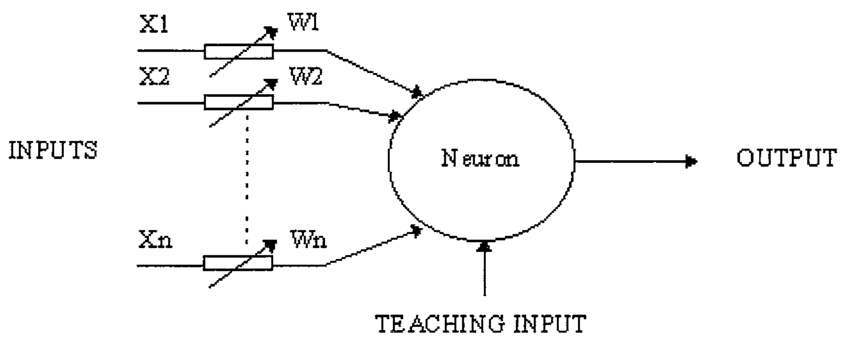


Figure 2.2 A typical neuron with incoming and outgoing connections

Each neuron has one or more incoming synapses and single output value. Each link has a numeric value called weight associated with it. Each neuron performs a simple task of summing the product of input and weights, called weighted inputs, from all input synapses. The neuron fires if the net excitation (summed value) exceeds its inhibition i.e. the threshold of the neuron.

$A^T_j = [a_1, \dots, a_i, \dots, a_n]$	the input vector
$W^T_j = [w_{j1}, \dots, w_{ji}, \dots, w_{jn}]$	the weight vector for the jth node

$$b_j = f(W_j^T . A + w_{j0} \times \theta_j) \quad \text{or} \quad b_j = f((\sum_{i=1}^{n} w_{ji} \times a_j) + w_{j0} \times \theta_j)$$

dot product point-wise

b_j is the output of jth node.

f(x) is the activation function such as $f(x) = 1 / (1+e^{-x})$.

θ_j is the threshold for the jth node

The general structure of the ANNs consists of layers of neurons:

Input Layer:

Each network must have one or more input neurons. Input neurons do not process the input hence produce output equal to their input. Input neurons are connected to hidden layer neurons, if any, or to output layer neurons.

Output Layer:

The network should contain one or more output neurons. Output layer neurons produce an output to the environment based on the activation function. Output neurons receive inputs from either hidden layer neurons, if any, or from the input layer neurons.

Hidden Layer:

A network may contain zero or more hidden layers. Hidden neurons are typically used as feature extractors and sometimes may be present in more than one layer. The hidden layers are bounded by input and output layers and do not interact with the environment directly.

2.2 Learning in Artificial Neural Networks

Learning in ANNs refers to the modification of internal network parameters, so as to bring the mapping from input to output as close as possible to a desired mapping between them. Therefore, any change in the memory or weight space, *W,* is considered as learning for the network.

$$dW / dt \neq 0$$

Learning may also be defined as optimization of the parameter set with respect to a set of training examples. Two important types of learning algorithms are:

- Supervised Learning
- Unsupervised Learning

2.2.1 Supervised Learning

Supervised learning is the most widely used technique. The term *supervised* originates from the fact that the desired signals on individual output nodes are provided by an external *teacher.* We collect many examples to serve as the training set. Each example in this training set comprises of all inputs and the desired outputs for these inputs. A supervised learning algorithm consists of the following steps:

- Present the training input to the input layer, one at a time.
- Calculate the error between the output produced by the network and the desired output.
- Update the network parameters so as to reduce the error.
- Repeat these steps until the error is zero or less than the desired error tolerance.

Back-propagation (BP) algorithm is by far the most popular supervised learning algorithm. The elementary backpropagation network is a three-layer, hetero associative ANN, with feedforward connections.

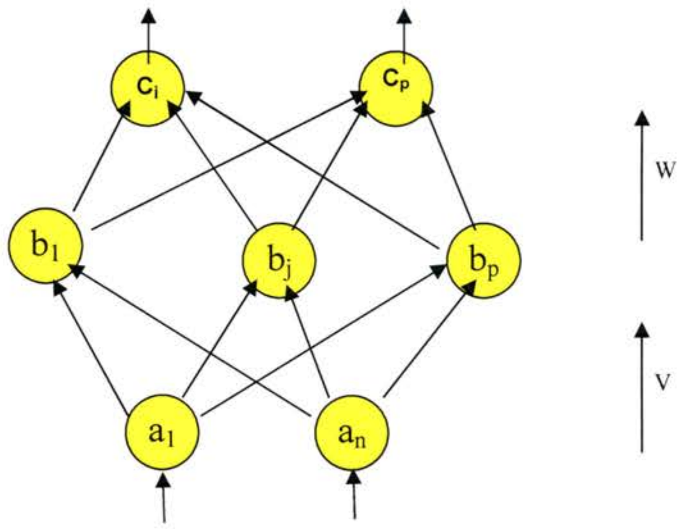


Figure 2.3 A typical feedforward neural network

2.2.1.1 Backpropagation Algorithm:

1. Assign random values in the range [+1, -1] to all the input to hidden layer connections, v_{hi}, all the hidden to output layer connections w_{ij}, to each hidden neuron threshold, θ_i, and to each output neuron threshold, ψ_j

2. For each pattern pair (A_k, C_k), *k = 1,2,...,m*, do the following:
   a) Transfer A_k's values to the input neurons, filter the input neuron activation through *V* and calculate the new hidden neuron values, using the following:

   $$b_i = f((\sum_{h=1}^{n} a_h v_{hi}) - \theta_i) \qquad \text{for all } i = 1,2,...,p$$

   where b_i is the activation value of the i^{th} hidden neuron, θ_i is the i^{th} hidden neuron's threshold value, and *f()* is the sigmoid threshold function:

   $$f(x) = (1 + e^{-x})^{-1}$$

   b) Filter the hidden activation through *W* to output using the equation:

   $$c_j = f((\sum_{i=1}^{p} b_i w_{ij}) - \psi_j) \qquad \text{For all } j = 1,2,...,q$$

   Where c_j is the activation value of the j^{th} output neuron and ψ_j is the j^{th} output neuron's threshold value.

   c) Compute the discrepancy (error) between the computed and desired output neuron values using the equation:

   $$d_j = c_j(1 - c_j)(c_j^k - c_j) \qquad \text{For all } j = 1,2,...,q$$

   Where d_j is the j^{th} output neuron's computed error.

   d) Calculate the error of each hidden neuron relative to each d_j with the equation:

   $$e_i = b_i(1-b_i) \sum_{j=1}^{q} w_{ij} d_j \qquad \text{For all } i = 1,2,...,p$$

   Where e_i is the i^{th} hidden neuron's computed error.

   e) Adjust the hidden to output connections:

   $$\Delta w_{ij} = \alpha\ (b_i\ d_j) \qquad \text{For all } i = 1,2,...,p \text{ and all } j = 1,2,...,q$$

   Where Δw_{ij} is the amount of change made to the connection from the i^{th} hidden neuron to the j^{th} output neuron, and α is a positive constant controlling the learning rate.

f) Adjust the output thresholds:

$$\Delta\psi_j = \alpha\, d_j \qquad \text{For all } j = 1,2,\ldots,q$$

Where $\Delta\psi_j$ is the amount of change to the jth output neuron's threshold value.

g) Adjust the input to hidden connections.

$$\Delta v_{hi} = \beta\,(a_h\, e_i) \qquad \text{for all } h = 1,2,\ldots,n \text{ and all } i = 1,2,\ldots,p$$

Where Δv_{hi} is the amount of change made to the connection from the *hth* input and *ith* hidden neuron, and β is a positive constant controlling the learning rate.

h) Adjust the hidden thresholds:

$$\Delta\theta_i = \beta\, e_i \qquad \text{for all } i = 1,2,\ldots,p$$

Where $\Delta\theta_i$ is the amount of change to the *ith* hidden neuron's threshold value.

3. Repeat step (2) until the error correction value d_j, for each $j = 1,2,\ldots,q$, and each training set $k = 1,2,\ldots,m$, is either sufficiently low or zero.
   - q is the number of neurons in the output layer, and
   - m is the number of input/output pairs.

To summarize backpropagation, the weights leading into an output node are adjusted in proportion to the difference between its actual value and its desired value. Weights leading into hidden nodes are adjusted in proportion to their contributions to error

2.2.2 Online Learning and Offline Learning

An artificial neural network can learn in one of two ways

- Offline or Batch learning
- Online or Stochastic learning

In offline or batch learning, optimization of network parameters is performed with respect to the entire training set and it is an iterative process. In batch learning the network learns using training datasets and with this knowledge network tries to recall near optimal results for the noisy inputs from the field. The swift computation of such an optimization is a difficult task, because generally, the dimension of parameter space is high. The network parameters (weights) are fixed in the operation or testing mode. Although batch learning may be faster for small or medium datasets and networks, it is more prone to problems like overtraining and local minima, and hence is inefficient in case of training large networks and for large training sets. The backpropagation algorithm presented above is an example of offline learning.

In online or stochastic learning, network parameters are updated after the presentation of each training example. Unlike the offline training, the networks can modify its parameters when it is in operation or testing mode. In the online learning scenario, only one example is given at a time and discarded after learning. Hence it consumes less memory and fits well into more natural learning, where user or agent receives new information at every moment and should adapt to it. Online learning is a more natural approach for learning non-stationary tasks, where batch learning needs retraining on the dynamically changing datasets. Apart from easier feasibility and data handling the most important advantage of online learning is its ability to adapt to changing environments. With batch learning these subtle changes go undetected as we average the error over several training examples. Online learning of continuous functions, using gradient based methods on a differentiable error measure is one of the most powerful and commonly used approaches for training non-stationary tasks in particular.

We can obtain the elementary online gradient descent algorithm by dropping the average operation in the batch gradient descent algorithm.

Consider an infinite sequence of independent examples

$(x_1,y_1),(x_2,y_2),\ldots..$

The purpose of learning is to obtain a network with parameter *w* which can represent the rules inherent to this data.

In online learning the ANNs modify their parameter w_t at time t to w_{t+1} using next example (x_{t+1}, y_{t+1}). But this may result in loosing the pre-learned information. To avoid this, we introduce a loss function ($l<x,y;w>$) to evaluate the performance of the network with parameter w. The best network is the network that has minimum $l<x,y;w>$ value.

We use the following parameter updating rule (Amari,1967 and Rumelhart et al., 1986):

$W_{t+1}=w_t-\acute{\eta}_t C(w_t)\partial/\partial w\, l(x_{t+1},y_{t+1};w_t),$

Where $\acute{\eta}_t$ is the learning rate that depends on t and $C(w_t)$ is a positive definitive matrix that depends on w_t .

If $\acute{\eta}_t$ =c/t, the where c is a constant, w converges to w^* (the parameter of best network) locally (Sompolinsky et al, 1995).

On the other hand online training suffers from the following drawbacks

- The main difficulty is the sensitivity of learning methods to the parameters. This dependence may slowdown the learning.
- Most advanced optimization methods like conjugate gradient, rely on a fixed error surface where in online learning task we need to deal with inherently stochastic error surface.

2.3 Genetic Algorithms

Genetic algorithms are best at solving problems for which little information is available. A genetic algorithm is an iterative procedure that consists of a constant-size population of individuals, each represented by a finite string of symbols, known as the *genome*. The genome encodes a possible solution in a given problem space. This problem space, referred to as the *search space*, comprises of all possible solutions to the problem at hand. Genetic algorithms use the principles of selection and evolution to produce several solutions to a given problem. Genetic algorithms tend to thrive in an environment in which there is a very large set of candidate solutions and in which the search space is uneven and has many hills and valleys.

2.3.1 Biological Motivation

The search performed by GAs is based on an analogy to biological evolution. At the turn of the century, it was unclear whether Darwin's or Lamarck's theory better explained evolution. Lamarck believed in direct inheritance of characteristics acquired by individuals during their lifetime. Darwin proposed that natural selection coupled with diversity could largely explain evolution. Darwin himself believed that Lamarckian evolution might play a small role in life, but most Darwinians rejected Lamarckism. One potentially verifiable difference between the two theories was that Darwinians were committed to gradualism (evolution in small, incremental steps), while Lamarckians expected occasional rapid change. One of the most interesting characteristics of natural evolution process is its robustness. The process is not dependent on external support and has a very high degree of fault tolerance. Holland's aim in devising computer models based on natural evolution was primarily to obtain this robustness, badly lacking in the existing systems.

Search techniques postulated in Artificial Intelligence research are largely local. Look-ahead is expensive Without look-ahead, chances of the search getting stuck at local maxima is high, because the optimum values at the distance are not visible to the local search techniques. Pure random search has a higher chance of avoiding local peaks, but is not sufficient for effective exploration of large search spaces.

All living organisms consist of *cells* In each cell there is the same set of *chromosomes.* Chromosomes are strings of DNA that serve as a model for the whole organism. A chromosome's characteristic is determined by the *genes*. Each gene has several forms or alternatives which are called *alleles*. These alleles produce differences in the set of chromosomes called the *genotype*. Each genotype maps to a *phenotype* (the individual) with a certain fitness.

The basic notions of natural evolution are as follows:

- New children are created from existing parents. The children inherit many of the characteristics of the parents.
- Each individual has a set of chromosomes consisting of one or more genes. The chromosomes (called the genotype) form the only genetically significant component for evolution. The genes directly control the external behavior and capabilities (called the phenotype) of the individuals. Changes in the phenotype can be realized by making changes in the genes.
- Natural selection works on the fitness of individuals. By the point above, fitness becomes a direct function of an individual's gene layout, i.e., the chromosome. Therefore, natural selection directly controls the selection of chromosomes for propagation.
- Chromosomes are relevant only at the point of reproduction, where suitably modified chromosomes for the children are created based on the chromosomes of the parents.
- There is no domain knowledge guiding the evolution process. The fitness-based rate of survival is the only guiding factor.

GAs work on the Darwinian principle of natural selection where stronger individuals are likely the winners in a competing environment. Darwinian model of evolution can be visualized as a sophisticated generate and test strategy. The natural selection based on fitness slowly discards potentially bad solutions from the population. The combining of chromosomes in the genetic reproduction process provides an opportunity to exploit already discovered regularities among the different members of the population.

2.3.2 A Prototypical Genetic Algorithm

The basic principles of GAs were first proposed by Holland. GAs presume that potential solution of any problem is a chromosome and can be represented by a set of parameters. These parameters are regarded as genes of a chromosome and can be structured by a string of values. A positive value is used to reflect the degree of fitness of the chromosomes for the problem which would be highly related with its objective value. Although different implementations vary in their details they typically share the same basic structure. The algorithm operates by iteratively updating a population of chromosomes. In every generation members of the population are evaluated according to the degree of fitness. We then select the fittest chromosomes of old population for the next generation without any change. Other solutions or chromosomes, based on their fitness, are used as the source for creating new offspring individuals by applying genetic operators such as crossover and mutation.

The following is a pseudo-code for general genetic algorithm approach:

```
0. [Representation] Define a genetic representation of the system.
1. [Start] Generate random population of n chromosomes (suitable solutions for the problem)
2. [Fitness] Evaluate the fitness of each chromosome in the population
3. [New population] Create a new population by repeating following steps until the new
                    population is complete.
   3.1. [Selection]   Select two parent chromosomes from a population according to their fitness
                      (the better fitness, the bigger chance to be selected)
   3.2. [Crossover]   With a crossover probability, cross over the parents to form a new offspring
                      (children). If no crossover was performed, offspring is an exact copy of
                      parents.
   3.3. [Mutation]    With a mutation probability, mutate new offspring at each locus (position in
                      chromosome).
   3.4. [Accepting]   Place new offspring in a new population
4. [Replace] Use new generated population for a further run of algorithm
5. [Test] If the end condition is satisfied, stop, and return the best solution in current population
6. [Loop] Go to step 2
```

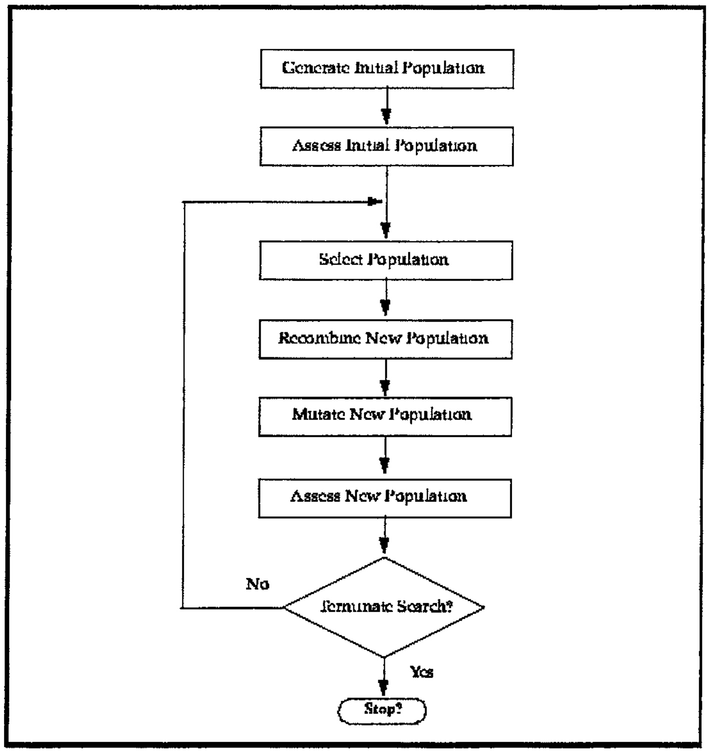


Figure 2.4 Flow chart representation of the algorithm

2.3.3 Chromosome Representation

The coding of chromosome representation may vary with the type of the problem at hand. Generally bit string encoding is used for the benefits of easy manipulation. The chromosomes or solutions represented with bit strings can be very complex. Using Gray code to represent the solutions works better than binary coding (Hollstein,R.B, 1971). Problems with real parameters cannot be solved efficiently with bit strings. Hence we use real value chromosomes for faster computation and high accuracy. The real encoding of solutions require specialized genetic operators. Although real encoding suits the practical problems it does not guarantee good results in all situations. Generally we use fixed length binary strings to represent real values. This approach may result in loss of accuracy but is easier to manipulate.

2.3.4 Genetic Operators

The generation of successors in a genetic algorithm is determined by a set of operators that recombine and mutate selected members of the current population. Typical genetic algorithm operators for manipulating the chromosomes are as follows.

- Selection
- Crossover
- Mutation

i. Selection:

The Selection operator selects the fittest chromosomes in the population for reproduction based on the rule that the fitter the chromosome, the more likely it is to be selected to reproduce. According to Darwin's evolution theory the best chromosomes should survive and create new offspring. There are many methods for selecting the best chromosomes, for example roulette wheel selection, Boltzman selection, tournament selection, rank selection, and steady state selection. Two of these approaches are explained below:

a) **Roulette Wheel Selection:** Parents are selected according to their fitness. The better the chromosomes are, the more chances they have to be selected. Imagine a roulette wheel (pie chart) where all chromosomes in the population are placed in according to their normalized fitness. Then a random number is generated which decides the chromosome to be selected. Chromosomes with better fitness values will be selected more times since they occupy more space on the pie.

b) **Rank Selection:** Roulette wheel selection is not efficient when the fitness of chromosomes is widely spread over a range of values. For example, if the best chromosome fitness is 90% of the entire roulette wheel then the other chromosomes will have very few chances to be selected. Rank selection first ranks the population and then every chromosome is assigned new fitness values from its rankings. The worst chromosome will have fitness 1, second worst 2 etc. and the best will have fitness N (number of chromosomes in population). After the new fitness allocation, all the chromosomes have a chance to be selected But this method can lead to slower convergence, because best chromosomes are generally similar and do not differ much from one other.

ii. Crossover:

A crossover operator manipulates a pair of individuals, called parents, to produce new individuals, called offspring, by interchanging segments from the parents' coding. By interchanging information between two parents, the crossover operator provides a powerful exploration capability of the solution search space. The bit at position *i* in each offspring is copied from the bit at position *i* in one of the two parents. The choice of which parent contributes the bit for position *i* is determined by an additional string called the crossover mask.

In single point crossover, the crossover mask is constructed with contiguous 1's followed by 0's to complete the string This results in offspring in which the first n bits are contributed by one parent and remaining bits by the second parent. Each time single point crossover operator is applied, the crossover point 'n' is randomly chosen, and then crossover mask is created and applied. The mask contains 'n' 1's followed by necessary number of 0's to complete the string.

In two-point crossover, the offspring are created by substituting intermediate segments of one parent into the middle of the second parent string. The cross over mask is a string beginning with n_0 zeros followed by contiguous string of n_1 1's, followed by necessary number of 0's to complete the string. For two point crossover operator, the mask is generated by randomly choosing the integers n_0 and n_1.

Uniform crossover combines bits sampled uniformly from two parents. Here crossover mask is generated as a random bit string with each bit chosen at random and independent of others.

iii. Mutation:

Mutation operator generates offspring from a single parent. Mutation is originally designed for binary represented chromosomes. Mutation operator produces small random changes to the bit string by choosing a single bit at random. As a population evolves, there is a tendency for genes to become predominant until they have spread to all members. Without mutation, these genes will be fixed for ever, since crossover alone cannot introduce new gene values. If the fixed value of the gene is not the value required at the global maximum, the GA will fail to optimize properly. Mutation is, therefore, important to `loosen up' genes which would otherwise become fixed, but if the mutation rate is too high, the selection pressure on genes resulting from breeding with fitter individuals may produce bad results. A common value for the mutation rate is to change one gene in every thousand.

iv. Objective Function:

An objective function is a measuring mechanism that is used to evaluate the status of a chromosome. This function is generally referred to as either evaluation function or fitness function. The notion of evaluation function and fitness function are used interchangeably. However, it is important to distinguish between the evaluation function and fitness function. Evaluation function provides a measure of an individual's performance, where as fitness function provides a measure of individual's reproduction opportunities.

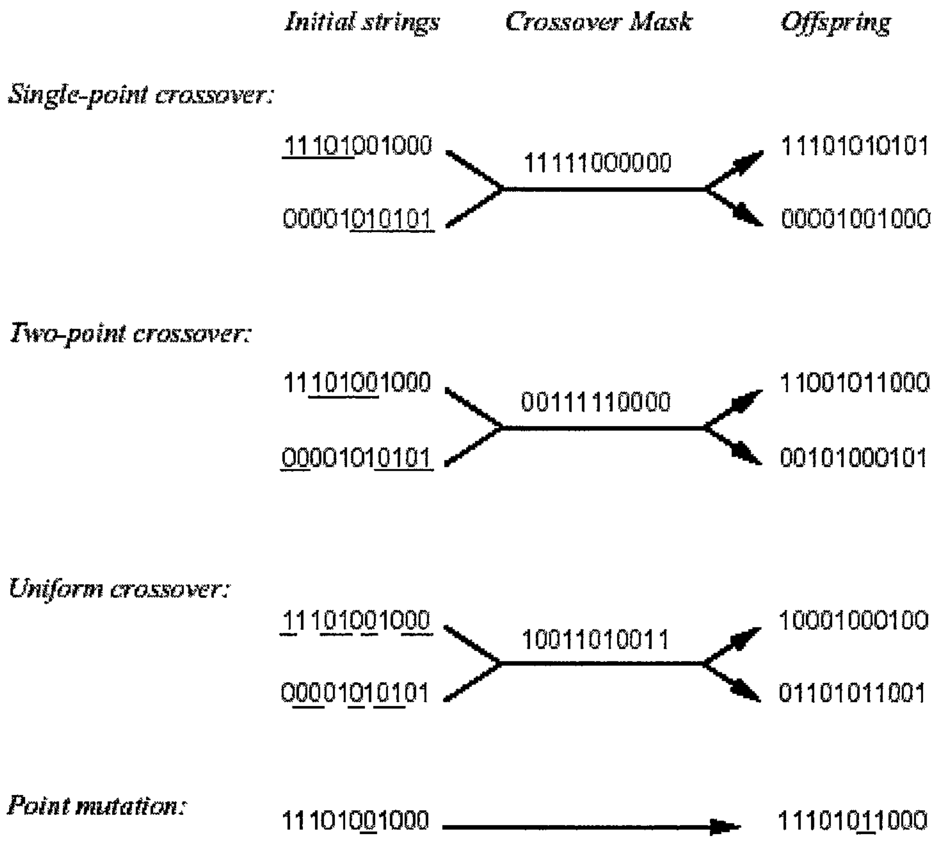


Figure 2.5 Depicting the operators with examples (Machine Learning, Tom Mitchell)

GAs illustrate how learning can be viewed as a special case of optimization. Particularly the learning task is to find the optimal solution according to the pre defined objective function. Like neural networks, genetic algorithms are easy to apply to a wide range of problems The results can be very effective on some problems. As Denker pointed out “Neural networks are the second best way of doing just about anything” and has extended his remark with “and genetic algorithms are the third”.

CHAPTER 3 COMBINING ARTIFICIAL NEURAL NETWORKS AND GENETIC ALGORITHMS

Overview

This chapter discusses the issues concerning with combining the two powerful technologies ANNs and GAs. This chapter gives brief account of the basic techniques of applying evolution to ANNs and assists in understanding our proposed algorithm design concepts.

3.1 Introduction

In the recent years two areas of adaptation, namely ANNs and GAs, captured the imagination of researchers all over the world. Both of these technologies are computational abstractions of biological information processing systems. In general, ANNs are used as learning systems and GAs as optimization systems ANNs are of particular interest because of their robustness, their parallelism, and their learning abilities. GAs are very powerful general learning methods that are based on natural evolution. However, both of these prominent technologies suffer from shortcomings.

ANNs are to a large extent based on

- Trial and Error
- Training examples or past experience
- Lack of sound design principles

Design of ANNs is critically dependent on the choice of primitives such as network architecture and parameters. Generally architectures are manually designed for the desired application and such a task requires lots of expertise and time on the part of the designer.

GAs are inefficient in the fine-tuning local search and may need vast amount of time to converge to a solution. Designing a suitable fitness function for real world applications may be hard. GAs also have weak theoretical basis, require tuning of many parameters for good performance, and sometimes computationally expensive.

The ANNs and GAs are capable of complimenting each other to get beyond their inefficiencies. They provide an extremely rich basis for contrast and hybridization. Hence, the combination results in highly successful adaptive systems (Yao, 1999). Features of these hybrid networks include adaptability to the environment, less human intervention, and more efficiency.

3.2 Combining ANNs and GAs

Researchers have combined ANNs and GAs in a number of different ways. Schaffer et al., have noted that these combinations can be classified into one of two general types - supportive combinations in which the ANNs and GAs are applied sequentially, and collaborative combinations in which they are applied simultaneously.

In a supportive approach, the GAs and the ANNs are applied to two different stages of the problem. The most common combination is to use a GA to pre-process the data set that is used to train an ANN. For instance, the GAs may be used to reduce the dimensionality of the data space by eliminating redundant or unnecessary features.
In supportive combinations the GAs and ANNs are used independent of each other. Some other possible combinations include using an ANN to select the starting population for the GAs; using a GA to analyze the representations of an ANN; and using a GA and ANN to solve the same problem and integrating their responses using a voting scheme (Schaffer et al.). Alternatively, in a collaborative approach, the GAs and ANNs are integrated into a single system in which a population of neural networks is evolved. In other words, the goal of the system is to find the optimal neural network solution. Such collaborative approaches are possible since neural network learning algorithms and genetic algorithms are search algorithms. A neural network learning rule performs a highly constrained search to optimize the network's structure, while a genetic algorithm performs a very general population-based search to find an optimally fit gene. Both are examples of biased search techniques, and "any algorithm that employs a bias to guide its future samples can be mislead in a search space with the right structure. There is always an Achilles heal." (Schaffer et al) The primary reason researchers have looked at integrating ANNs and GAs is the belief that they may compensate for each other's search weaknesses.

3.3 Evolutionary Design of Neural Networks

We can introduce evolution into ANNs primarily in three different levels: connection weights; architectures; and learning rules as noted by Yao. The evolution of connection weights introduces an adaptive and global approach to training, especially in the reinforcement learning and recurrent network learning paradigm where gradient-based training algorithms often experience great difficulties. The evolution of architectures enables ANNs to adapt their topologies to different tasks without human intervention and thus provides an approach to automatic ANN design as both ANN connection weights and structures can be evolved. The evolution of learning rules can be regarded as a process of "learning to learn" in ANNs where the adaptation of learning rules is achieved through evolution. It can also be regarded as an adaptive process of automatic discovery of novel learning rules.

3.3.1 Evolution of Connection Weights:

This is the basic level where we can incorporate genetic operators into neural networks. Generally, the weights of the connections are modified in order to optimize evaluation function such as mean square error To formulate the training process as the evolution of connection weights, we require two phases.

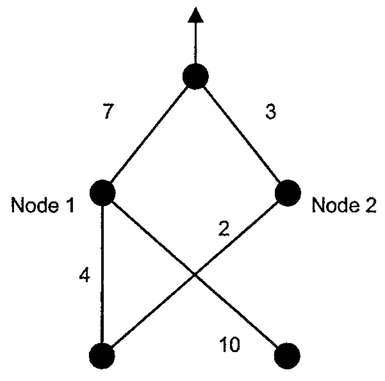
- Representation of connection weights
- Evolutionary process to apply

Genetic operators are efficient and easy to use with binary strings. The most important stage in evolution of weights is to decide on a suitable representation for connection weights, i.e. either we represent them as binary strings or not In the second phase we choose the evolutionary process simulated by a genetic algorithm, in which search operators such as crossover and mutation have to be decided in conjunction with the representation scheme. The training performance depends on the representation scheme we choose.

3.3.1.1 Binary Representation:

Genetic algorithms, in general, use binary strings to encode the population of solutions which are also called chromosomes. In the binary representation scheme, each connection weight is represented by a number of bits with certain length. An ANN is encoded by concatenation of all the connection weights of the network in the chromosome. A heuristic for the order of the concatenation of connection weights in a chromosome is to append all the binary connection weights coming from input nodes to each hidden neuron in the hidden layer from left to right and append all binary connection weights coming from hidden nodes to each output node in the output layer from left to write. Hidden nodes in ANNs are in essence feature extractors and detectors. The above heuristic is based on the fact that separating connection weights from different input nodes to the same hidden node, apart in the chromosome representation This would increase the difficulty of constructing useful feature detectors because these feature detectors, found during the evolutionary process, might be destroyed by crossover.

Figure 3.1.a and Figure 3.1.b provide an example for the binary representation of an ANN whose architecture is predefined. Each connection weight in the ANN is represented by 4 bits, the whole ANN is represented by 24 bits where weight 0000 indicates no connection between two nodes.



0100 1010 0010 0000 0111 0011

a **b**

Figure 3.1.a: An ANN with connection weights Figure 3.1.b. Binary representation of connection weights

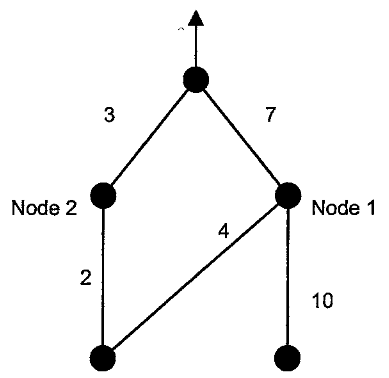
Binary encoding has its advantages and disadvantages

Advantages:

The advantages of binary representation are simplicity of design, generality of representation, and straightforward application of genetic operators such as crossover and mutation. It does not need any complex or tailored operators. Also binary representation facilitates digital hardware implementation of ANNs as weights are represented with 0's and 1's with limited precision in the hardware.

Disadvantages:

Real world applications generally need real number representation of weights. But some combinations of real valued connection weights cannot be approximated with sufficient accuracy by binary values. If too many bits are used, chromosomes representing large ANNs will become extremely long and the evolution in turn will become very inefficient. If too few bits are used to represent each connection weight, training might fail because some combinations of real-valued connection weights cannot be approximated with sufficient accuracy by discrete values. So a tradeoff between representation precision and the length of chromosome often has to be made.



a

0010 0000 0100 1010 0011 0111

b

Figure 3.2.a: An ANN with connection weights

Figure 3.2.b: Binary representation of connection weights

An important concern for the evolutionary approach to neural network is the competing conventions problem. It is also called permutation problem. It is caused by the many-to-one mapping from the representation(genotype) to the actual ANN(phenotype) since two ANNs that order their hidden nodes differently in their chromosomes will still be equivalent functionally. For example, ANNs shown in Figure 3.1.a and Figure 3.2.a are functionally equivalent but are represented by different chromosomes as shown in Figure 3.1.b and Figure 3.2.b. The permutation problem makes crossover operator very inefficient and ineffective in producing good offspring.

3.3.1.2 Real Number Representation:

Real numbers represent the reality better than binary numbers. Figure 3.1.a can be represented by real numbers as the following real vector.
{4.0, 10.0, 2.0, 0.0, 7.0, 3.0}.

As connection weights are represented by real numbers, each individual in an evolving population will be a real vector. Traditional genetic operators no longer work with this representation. Real representation needs more complex genetic operators to be designed.

Advantages:

Real values are suitable for most of the problems and can represent values with great accuracy. Evolutionary Algorithms (EAs), which are different from Genetic Algorithms in their primary operator being mutation rather than crossover, work well with real number representation. When used with EAs, this representation tends to reduce the negative impact of permutation problem.

Disadvantages:

As traditional operators are no longer applicable on real valued representation, we need to define special operators. Designing these operators is no easy task. Real valued representation also suffers from permutation problem.

A typical cycle of the evolution of connection weights is shown in the following algorithm.

```
1. Decode each individual genotype in the current generation into a
   set of connection weights and construct corresponding ANNs with
   the weights.
2. Evaluate each ANN by computing its total mean square error
   between actual and desired outputs, or use any general error
   function. The fitness of an individual is determined by the
   error. The optimal mapping from error to the fitness is problem
   dependent.
3. Select the parents for reproduction based on their fitness.
4. Apply genetic operators such as crossover (recombination) and/or
   mutation to parents to generate offspring and then selection on
   these offspring to form the next generation.
```

Repeat the above steps until the fitness is greater than a predefined value or the population has converged (Yao, 1999).

3.3.2 Evolution of Architectures

For a long time the task of designing the architecture of a neural network has been manual and required expertise in the field. Automating design of ANN architectures for applications is always an important issue. The design of neural networks architectures does not have any mathematical basis; hence architecture design requires a tedious trial and error method. There were several attempts, such as constructive and destructive algorithms, to automate the designing process. However, they were only partially successful.

Design of the optimal architecture for an ANN can be formulated as a search problem in the architecture space where each point represents an architecture. Given some performance (optimality) criteria, e.g., lowest training error, lowest network complexity, etc., about architectures, the performance level of all architectures forms a discrete surface in the space. The optimal architecture design is equivalent to finding the highest point on this surface. This kind of vast search space is suitable for applying GAs. Hence evolution of architectures finds near optimal architecture given sufficient time.

As with the evolution of weights, there are two major evolution phases of architectures.

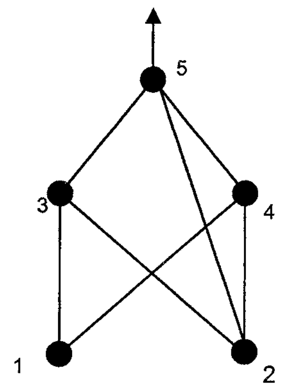
- The representation or encoding of the network
- Genetic operators used to evolve the architecture

There are several encoding schemes based on how much information we want to incorporate into the representation.

3.3.2.1 Direct Encoding Scheme:

In this scheme all the details about the architecture, i.e. every connection and node of an architecture, can be incorporated into the chromosome. In this scheme each connection of architecture is directly specified by its binary representation For example, an matrix can represent an ANN architecture with N nodes, where indicates presence or absence of the connection from node i to node j . We can use to indicate a connection and to indicate no connection.

Each matrix 'C' has a direct one-to-one mapping to the corresponding ANN architecture. The binary string representing an architecture is the concatenation of rows (or columns) of the matrix. Constraints on architectures being explored can easily be incorporated into such a representation scheme by imposing constraints on the matrix, e.g. a feedforward ANN will have nonzero entries only in the upper-right triangle of the matrix. Figure 3.3 and Figure 3 4 are two examples of the direct encoding scheme of ANN architectures. It is obvious that such an encoding scheme can handle both feedforward and recurrent ANNs.

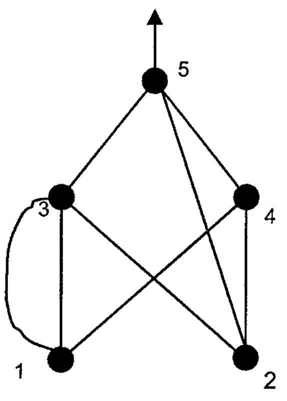


$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$\boxed{\textbf{0110 101 01 1}}$

a b c

Figure 3.3: An example of the direct encoding of a feed forward ANN (a), (b), and (c) show the architecture, its connectivity matrix, and its binary string representation, respectively



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$\boxed{\underline{00110\ 00100\ 10001\ 00001\ 01000}}$

a b c

Figure 3 4. An example of the direct encoding of a recurrent ANN. **(a)**, **(b)**, and **(c)** show the architecture, its connectivity matrix, and its binary string representation, respectively.

The direct encoding is quite straightforward to implement. It is very suitable for the precise and fine tuned search of a compact ANN architecture, since a single connection can be added or removed from the ANN easily. It may facilitate rapid generation and optimization of tightly pruned new designs. The major disadvantage of this encoding scheme is scalability. A large neural network would result in a very large string, hence making the evolutionary process inefficient.

3.3.2.2 Indirect Encoding:

To overcome the scalability problem of direct encoding, indirect encoding scheme is commonly used. In the indirect encoding scheme we encode only important characteristics of architecture, rather than encoding all details, into the chromosome. The details about each connection in an ANN is either predefined according to prior knowledge or specified by a set of deterministic developmental rules. The indirect encoding scheme can produce more compact genotype representation of ANN architectures, but it may not be very good at finding a compact ANN with good generalization ability. The following are few indirect encoding schemes and their details

i. Parametric Indirect Encoding Scheme:

ANN architectures may be specified by a set of parameters such as the number of hidden layers, the number of hidden nodes in each layer, the number of connections between two layers, etc. These parameters can be encoded in various forms in a chromosome. This scheme is proposed and developed by Harp et al.

Although this representation considerably reduces the length of the binary string, the GAs can only search a subset of the whole search space. Hence it is suitable for the problems where we know what kind of architectures we are trying to find.

ii. Developmental Rule Representation Scheme:

In this method, we encode developmental rules which are later used to build architecture into chromosomes. This scheme results in even more compact genotype representation and also increases the efficiency of crossover operator as it saves the details of promising architectures.
A developmental rule is usually described by a recursive equation or a generation rule similar to a production rule in a production system with a left-hand side (LHS) and a right-hand side (RHS). The connectivity pattern of the architecture in the form of a matrix is constructed from a basis, i.e. a single-element matrix, by repetitively applying suitable developmental rules to non-terminal elements in the current matrix until the matrix contains only terminal elements which indicate the presence or absence of a connection, that is, until a connectivity pattern is fully specified.

The following algorithm by Yao, represents evolutionary development of learning rules

1. Decode each individual genotype in the current generation into architecture. If the indirect encoding scheme is used, further detail of architecture is specified by some developmental rules or a training process.
2. Train each ANN with the decoded architecture by a predefined learning rule, starting from different sets of random initial weights and if any, learning parameters.
3. Calculate the fitness of each individual (encoded architecture) according to the above training result and other performance criteria such as complexity of architecture.
4. Select the parents for reproduction based on their fitness.
5. Apply genetic operators such as crossover (recombination) and/or mutation to parents to generate offspring and then selection on these offspring to form the next generation.

Repeat the above steps until the fitness is greater than a predefined value or the population has converged.

3.3.3 Simultaneous Evolution of Architectures and Connection Weights

The above evolutionary methods either keep architecture intact or fine tune the weights after near optimal architecture is found. Both these methods introduce noise and generate less efficient systems. One major problem with the evolution of architectures without evolution of connection weights is noisy fitness evaluation as phenotype's (i.e., an ANN with a full set of weights) fitness was used to approximate its genotype's (i.e., an ANN without any weight information) fitness.

There are two major sources of noise.

- Due to the random initialization of weights, the same genotype (the ANN without any weight information) may have quite different fitness.
- Different training algorithms may produce different training results even from the same set of initial weights.

Hence evolution of architectures without any weight information has difficulties in evaluating fitness accurately. As a result, the evolution would be very inefficient. To alleviate these problems and to build more efficient systems we need to evolve the connection weights and architectures simultaneously.

3.3.4 Evolution of Learning Rules

An ANN training algorithm may yield different performance when applied to different architectures. The design of training algorithms, more fundamentally the learning rules used to adjust connection weights, depends on the type of architectures under investigation. Different variants of the Hebbian learning rule have been proposed to deal with different architectures. However, designing an optimal learning rule becomes very difficult when there is little prior knowledge about the ANNs architecture, which is often the case in practice. It is desirable to develop an automatic and systematic way to adapt the learning rule to an architecture and the task to be performed. Often evolution of learning rules is application specific i.e. it is almost impossible to find a general rule that can be applied to all structures.

what is needed from an ANN is its ability to adjust its learning rule adaptively according to its architecture and the task to be performed. In other words, an ANN should learn its learning rule dynamically rather than have it designed and fixed manually.

Unlike the evolution of connection weights and architectures which only deal with static objects in an ANN, i.e. weights and architectures, the evolution of learning rules has to work on the dynamic behavior of an ANN. The key issue here is how to encode the dynamic behavior of a learning rule into static chromosomes. Trying to develop a universal representation scheme which can specify any kind of dynamic behaviors is impractical, since it requires a very long computation time to search such a large learning rule space. So to keep the representations simple with a short search space, we impose limitations on the type of dynamic behaviors.

Two basic assumptions which have often been made on learning rules are:

1. Weight updating depends only on local information such as the activation of the input node, the activation of the output node, the current connection weight, etc.,
2. The learning rule is the same for all connections in an ANN. A learning rule is assumed to be a linear function of these local variables and their products.

The following illustrate the basic methods of evolving learning rules.

3.3.4.1 Developing Algorithmic Parameters:

The adaptive adjustment of back propagation (BP) parameters (such as the learning rate and momentum) through evolution could be considered as the first attempt to the evolution of learning rules. Harp *et al* encoded BP's parameters in chromosomes together with ANN architecture. This evolutionary approach is different from the non-evolutionary approach. Because the simultaneous evolution of both algorithmic parameters and architectures facilitates exploration of interactions between the learning algorithm and architectures such that a near optimal combination of BP with an architecture can be found.

3.3.4.2 Developing Learning Rules:

The above method serves as the fundamental development of learning rules. There are three major issues involved in the evolution of learning rules:

- Determination of a subset of terms described
- Representation of their real-valued coefficients as chromosomes
- The EA used to evolve these chromosomes.

There is a lot of research going on today to develop this method, since this stands for the true evolution of learning rules. Adapting a learning rule through evolution is expected to enhance ANN adaptivity greatly in a dynamic environment.

The following algorithm by Yao represents evolutionary development of learning rules

```
1. Decode each individual genotype in the current generation into a
   learning rule.
2. Construct a set of ANNs with randomly generated architectures and
   initial connection weights, and train them using the decoded
   learning rule.
3. Calculate the fitness of each individual (encoded learning rule)
   according to the above training result.
4. Select the parents for reproduction based on their fitness.
5. Apply genetic operators such as crossover (recombination) and/or
   mutation to parents to generate offspring and then selection on
   these offspring to form the next generation.
```

Repeat the above steps until the fitness is greater than a predefined value or the population has converged.

As Genetic Algorithms tend to be computationally intensive, we need to use them with prior knowledge or with some heuristic to assist the search. With the increasing power of parallel computers, the evolution of large ANNs becomes feasible. Not only evolution can discover possible new ANN architectures and learning rules, but it also offers a way to model the creative process as a result of ANN adaptation to a dynamic environment.

CHAPTER 4 RELATED RESEARCH

Overview

There has been a lot of research in hybrid algorithms and online learning. This chapter gives details about the research done in the relative fields.

4.1 Evolutionary Design of Neural Networks

4.1.1 EPNet

EPNet (Yao and Liu, 1996) describes an evolutionary system for evolving feedforward ANNs Unlike the other evolutionary algorithms, it tries to evolve the behavior of ANNs. EPNet combines architectural evolution with modification of weights. This simultaneous evolution of weights and architecture reduce the noise in the fitness evaluations.
EPNet is based on evolutionary programming; hence mutation is its only operator. EPNet encourages parsimony of evolved ANNs by attempting different mutations sequentially only if they are needed.

A number of techniques have been adopted in EPNet to maintain a close behavior between parents and their offspring. Partial training is always employed after each architectural mutation in order to reduce the behavioral disruption to an individual. Each individual in a population evolved by EPNet is an ANN with weights. The evolution simulated by EPNet is closer to Lamarckian than Darwinian It relies on five mutation operators to produce better offspring. The five mutations are:

- Hybrid training
- Node deletion
- Connection deletion
- Connection addition
- Node addition

EPNet starts with a population of networks, sorted on the fitness criteria, in the initial partial training. Then the five mutations are applied sequentially. If one mutation leads to a better offspring, it is regarded as successful. No further mutations are applied, otherwise, next mutation is attempted. A hidden node is not added to existing architecture at random, but through splitting an existing node. This process ensures the compact architectures without loosing their ability to generalize.

EPNet uses direct encoding scheme and works only for feedforward networks. Selection mechanism used in EPNet is based on the error generated. Only if other mutations fail to improve the fitness hidden node deletion, connection deletion, and node addition are performed in the proposed order. After each stage a partial training is applied and ANNs are tested for the success. Only on failure of current stage further stages are applied, otherwise training skips the other mutation stages for the next step. The following flowchart explains EPNET training process.

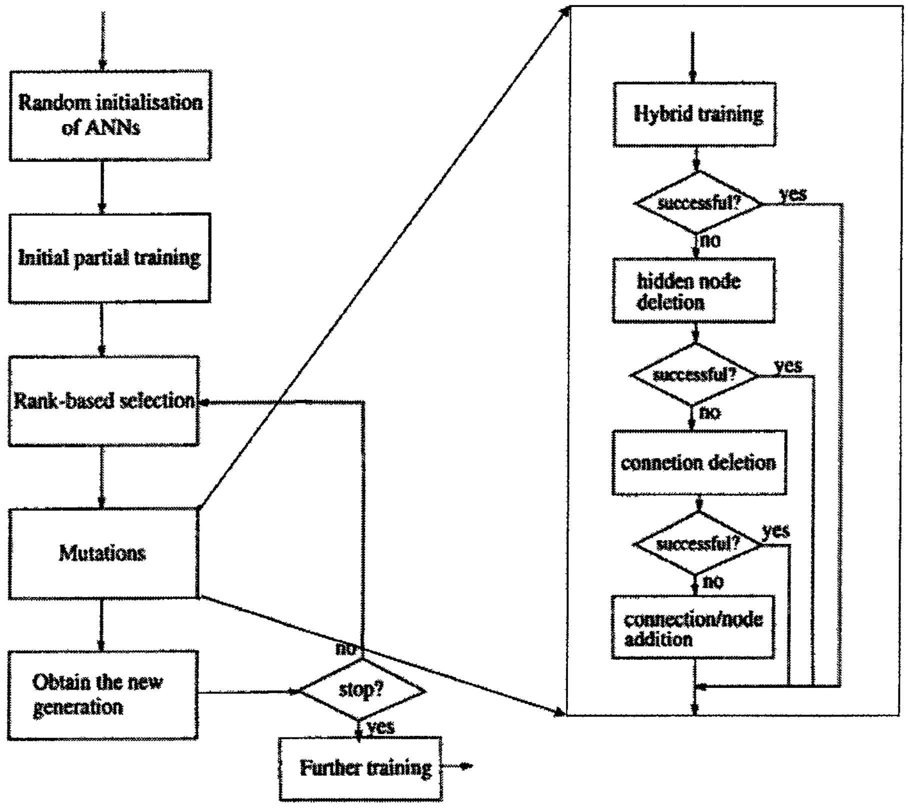


Figure 4.1: The main structure of EPNet

Results:

The data sets used for the experiments were partitioned into three sets for training, validation, and testing.

The EPNet was tested on 4 medical problems

1. Breast cancer: data set contained 349 training, 175 validation, and 175 testing examples
2. Diabetes: data set contained 384 training, 192 validation, and 192 testing examples
3. Heart disease: data set contained 134 training, 68 validation, and 68 testing examples
4. Thyroid: data set contained 2518 training, 1254 validation, and 3428 testing examples

The results showed that evolved ANNs have very small sizes as well as low error rates.

Our algorithm is similar to EPNet in the following areas.

- Supervised training approach is considered to train the ANNs
- A variation of backpropagation is used as training algorithm
- Evolution of connection weights and architectures carried out simultaneously

Our algorithm differs in several ways from EPNet algorithm.

- EPNet does not use crossover operator
- The networks can only be trained offline in EPNet
- EPNet cannot be applied to recurrent or feedback networks
- Evolution of learning rules is not implemented in EPNet

4.1.2 NEAT (Neuro Evolution of Augmenting Topologies)

NEAT (Kenneth Stanley et al, 2000) proposes a new design for simultaneous architecture and weight evolutions. In the NEAT each Genome represents network connectivity and contains connection genes and node genes. A new concept of innovation number is introduced to avoid the competing conventions problem with crossover operator. Each connection gene is given an innovation number which is unique for the whole population. Mutation in NEAT can change both connection weights and network structure.

NEAT works by starting with a minimal structured network and incrementally adding neurons and/or connections. They claim the resulting architecture to be the optimal structure. NEAT uses four genetic operators in topology evolution.

Genetic operators used in NEAT algorithm are:

- Mutation of connection weights.
- Mutation by adding neurons
- Mutation by adding connections
- Crossover

Using a global innovation number, NEAT can track the historical origins with very little computation. This algorithm offers a solution, through historical markings, to the competing conventions problem in a population of diverse topologies. NEAT uses speciation to protect slowly maturing Genomes.

Results:

NEAT was tested with XOR problem and pole balancing task.

For XOR problem on 100 runs, the NEAT system finds a solution structure in an average of 32 generations. NEAT was able to evolve near optimal network for the task.

In the pole balancing task two poles are connected to a moving cart by a hinge and the neural network must apply force to the cart to keep the poles balanced for as long as possible without going beyond the boundaries of the track.

The criterion for success was to balance the poles for 100,000 time units. Results show that NEAT took fewest evaluations to complete the task. The standard deviation for the NEAT evaluations is 2704. The performance was far better than most of the existing evolutionary algorithms.

Our algorithm has some common features with NEAT algorithm.

- The unique numbering scheme, called innovation number in NEAT, is used to avoid competing conventions problem
- Node based direct encoding scheme is used to encode the ANNs

Our algorithm differs from NEAT in several characteristics.

- Our algorithm provides an online learning mechanism which is not present in NEAT
- NEAT algorithm can evolve weights and architectures simultaneously but does not evolve learning rules. Our algorithm evolves the learning rules

While EPNet claims that Lamarckian method works best, the NEAT supports Baldwin effect.

4.2 Online Adaptive Algorithms

Learning may be viewed as an optimization of the internal parameters. This optimization is carried out using a learning rule, which depends on the application. There are two learning paradigms.

1) Offline Learning
2) Online Learning

In Offline Learning, the network parameters are updated after presenting the entire training set. This is repeated several times until all the characteristics of the training set are incorporated in the network.

In Online Learning, the network parameters are updated for each training pattern. The most important advantage of online learning is its ability to adapt to changing environment.
It is also shown that online algorithms are asymptotically as effective as Offline Learning (Robbins and Monro, 1951)

4.2.1 Online Learning for Drifting Environments

An environment that changes over time and is dynamic is called a drifting environment. Klans et al proposed a pure neural network online algorithm that can learn to adopt. They employed supervised approach and used Stochastic Gradient Algorithm with an adaptive learning rate. The idea of adaptively changing the learning rate is called learning of learning rule (Somplinskey et al 1995). Klans et al extended the adaptive learning rate idea to differential loss functions. In their approach when the error is large then learning rate takes large value and if error is small then learning rate also decreases. They use Hessian matrix of the expected loss function in their algorithm. This algorithm applies to feedforward networks and provides a learning strategy where continuous functions are to be learned when no explicit loss function is available.

Results:

Their experiments showed that they could separate original mixed and unmixed artificial signals in less than 500 iterations. Good quality results were observed from 200 iterations only.

Our algorithm is similar to this algorithm (Klans et al) in the following

- Both of the algorithms try to develop neural networks to adapt to dynamic environments
- Both algorithms use a variation of back propagation to train neural networks online

Our algorithm differs from this algorithm (Klans et al) in several ways, they are:

- Their algorithm uses Hessian functions to approximate
- Unlike our algorithm, their algorithm doesn't evolve the architecture
- Their algorithm does not provide offline learning

4.3 Hybrid Online Adaptive Algorithm

4.3.1 Evolution of Learning: An Experiment in Genetic Connectionism

David J Chalmers (1990) proposed a basic framework for evolution of learning in neural networks. Chalmers proposed that a Genome encodes the dynamic properties of weight space dynamics of connectionist system. Here a number of networks are created and placed in different environments for specified amounts of time This helps in determining the fitness of a learning procedure. Each network's final stage is determined by its interaction with the learning procedure and the environment. The fitness of the network is determined by how well it has adapted to the environment in the specified time period. The algorithm claims that from a population of essentially ineffective learning procedures, it can produce learning rules that enable better adoption. This algorithm uses supervised approach because of its simplicity. The evolution of connection weights and architectures is not pursued here. Hence a single layer of fixed and fully connected network is used in the algorithm. The changes to the weight of any connection should only be dependent on the information local to that connection. The algorithm makes use of ten variables and one scale variable to evolve learning rules. The general rule uses these ten variables as coefficients of network and algorithmic parameters and the scale variable to increase or decrease the amount of change. All variables are represented by fixed number of binary digits.

Results:

Chalmers conducted several experiments over 8 tasks. For each task, a network was presented with a number of training examples each consisting of an input pattern and associated output pattern. The results show fitness improvement from 60% to 90% after 1000 generations.

Our algorithm shares some common features with this algorithm (Chalmers, 1990)

- Both of these algorithms use a general linear equation to evolve learning rules
- Both of the algorithms use supervised learning procedure
- Both of the algorithms provide learning for evolved networks

Our algorithm has several features that are different from Chalmers' algorithm

- Unlike this algorithm, our algorithm implements evolution of architectures
- Chalmers algorithm uses single layer fixed and fully connected networks, whereas our algorithm uses dynamic architectures
- Chalmer's algorithm is designed only for feed-forward networks
- The learning rule variables are represented with binary values in Chalmers proposed algorithm. Our algorithm uses real variables in learning rule evolution

4.3.2 Learning to Adapt to Changing Environments in Evolving ANN

From the Institute of Psychology-Rome, Stefano Nolfi et al (1996) proposed their methodology "Learning to adapt to changing environments in evolving ANNs". They used a genetic algorithm to simulate the evolution of a population of neural networks each controlling the behavior of a small mobile robot that must explore an environment surrounded by walls. The environment changes from one generation to another. Their methodology was proposed to overcome the limitations of the simulated aquatic environment set up by Todd-Miller in 1991. Todd and Miller (1991) developed creatures that live in one of the two patches in the environment. Stefano Nolfi et al proposed the evolutionary method to develop a creature, which is able to reach a target area containing food in its environment. The creature should explore the arena as efficiently as possible while avoiding collisions with wall. They have used a feedforward neural network with four input sensors in the input layer, which are connected to four output units in the output layer. The neural network has two distinct sub-networks that share the same inputs but have separate outputs. The first network determines the creatures moving actions while the second network determines updating of connection weights of the standard network. The teaching network's connection weights never change.

Results:

Experiments began with 100 random networks with random weights for standard and teaching sub-networks. Each generation lives for 10 epochs, each epoch containing 500 input/output cycles. The results proved that the networks that learn achieve higher fitness than those that do not learn.

Our algorithm is similar to this algorithm in some aspects. They are:

- Both incorporate learning after evolution
- Both algorithms address architectural evolutions
- Both algorithms are applicable to fast changing environments

Our algorithm differs considerably from this algorithm (Stefano Nolfi et al, 1996) in the following areas.

- It (Stefano Nolfi et al, 1996) addresses only feedforward neural networks
- It does not use hidden layers
- It does not evolve learning rules
- It does not employ simultaneous evolution of structure and weights

4.3.3 Evolutionary Algorithm for Online Learning

Magoulus et al (2001) have proposed a novel hybrid evolutionary approach for online training. As classic batch training algorithms cannot handle non-stationary data, the need for online learning arises. Their Lamarckian inspired hybrid evolutionary algorithm basically consists of two stages. In the first stage, they provide online training using stochastic gradient descent with adaptive step size. In the second stage, differential evolution strategies proposed by R.storn et al (1997) are used as online retraining. The second stage assumes that the SGD in the first stage has produced a good solution. The second stage directly incorporates the solutions produced in the first stage into the genes of off-spring. They have employed a memory based calculation of step size, in the first stage, which considers the previous information to adapt the step size for the next pattern presentation. They claim that the SGD algorithm has low storage requirements and needs less computation. In the second stage, the DE strategy is used for re-training. They perform evolutionary operations on the weight vector. The primary DE operator used is mutation. For each weight vector $w^p{}_i$, a new mutant vector is generator using the following relation:

Mutant Vector = $w^p{}_i + \xi\,(w_{best} - w^p{}_i) + \xi(w^r{}_1 - w^r{}_2)$,

Where W_{best} is the best member of previous generation, $\xi>0$ is a real parameter called mutation constant, $w^r{}_1$ and $w^r{}_2$ are two random weight vectors.

Stage 1 - "Learning"

```
Step 0a: Initialize the weights $w^0$, $\eta^0$ and the meta-stepsize K.
Step 1a: **Repeat** for each pattern p.
Step 2a :Calculate $E(w^p)$ and then $rE(w^p)$.
Step 3a: Update the weights:
        $w^{p+1} = w^p - \eta^p rE(w^p)$.
Step 4a: Calculate the stepsize to be used with the next pattern
        p + 1: $\eta^{p+1} = \eta^p + K\ rE(w^{p-1}),\ rE(w^p)$
Step 5a: **Until** the termination condition is met.
Step 6a: **Return** the final weights $w^{p+1}$ to the Stage 2.
```

Stage 2 - "Evolution"

```
Step 0b: Initialize the DE population in the neighborhood of $w^{p+1}$.

Step 1b: Repeat for each input pattern p.
Step 2b: For i = 1 to NP
Step 3b:     MUTATION($w^{p}_{i}$) → Mutant Vector.
Step 4b:     CROSSOVER(Mutant Vector) → Trial Vector.
Step 5b:     If E(Trial Vector) 6 E($w^{p}_{i}$), accept Trial Vector for
              the next generation.
Step 6b: EndFor
Step 7b: Until the termination condition is met.
```

Algorithm 4.1: Generic Model of the Hybrid On–line Training Algorithm

To further increase the diversity, they used crossover operator. Based on a crossover constant they decide whether to select a bit or not into the target vector.

Results:

They have tested the algorithm with two experiments. The first experiment was to train an ANN online to classify among 12 texture images. The results show that it performed better than batch propagation. The second experiment was to train an ANN online to detect suspicious regions in colonoscopic video sequences. The algorithm provided better results over algorithm without evolution.

Our algorithm is similar to this hybrid algorithm (Magoulus et al, 2001) in some aspects. They are:

- Both have two training stages namely, offline and online
- In both algorithms evolution makes use of mutation and crossover
- Both can be used in slowly varying environments

Our algorithm differs in several ways from this algorithm (Magoulus et al, 2001)

- In this algorithm the architecture is fixed and they only evolve weight vectors
- In this algorithm learning is employed only once and only evolution is repeated until terminating criterion is met
- It does not use speciation and cannot have global online learning

4.4 Online Interactive Learning

Adrian Agogino et al (1999) have built a system based on online neuro evolution. There are only few systems that are capable of online evolution. Agogino et al have proposed to evolve feedforward ANNs to create the agents that improve their performance through real time interaction. Typically the system has two stages:

1. Offline evolution
2. Online evolution

This approach is demonstrated in a game world where ANN controlled agents play against humans. Through offline evolution the agents are trained for various conflicting goals. Then the prepared population is allowed to evolve online.

Each agent has a feedforward ANN as its brain. The outputs from this network guide the agent in the given environment at each time step. The inputs to the network are collected through eight sensors. Four of them provide enemy information and the other four supply mine location information.

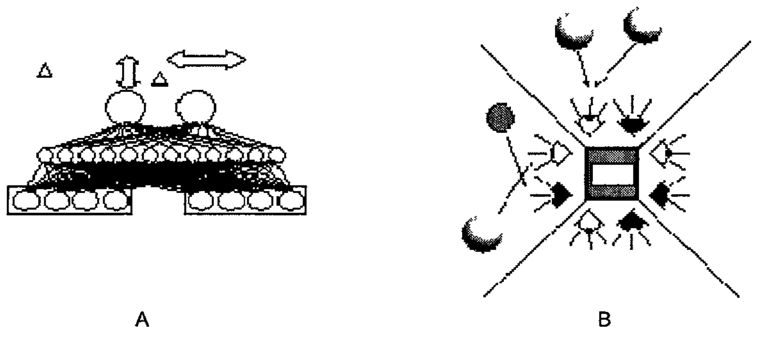


Figure 4.2 (A) Peon's neural net with inputs and outputs. The sensor information is sent to the input layer of the feedforward network. The two output nodes indicate where the peon should go in terms of latitude and longitude distance from the current location. (B) Configuration of a peon's eyes Four of the eyes return the average distances to gold mines in each quadrant and the other four eyes return the average distance of the enemy.

When an agent is killed it is replaced with either a best fit agent or an agent from crossover operation on two better fit agents.

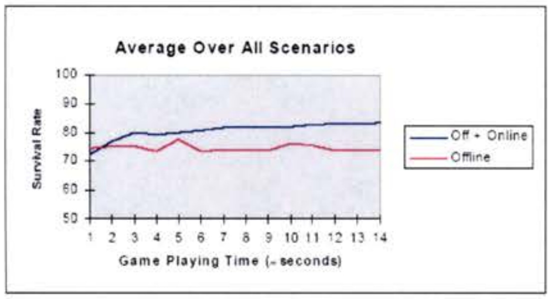
The agents are ranked on their rate of productivity based on the following formula:

Fitness= (Mines found* V-C)/ Age,

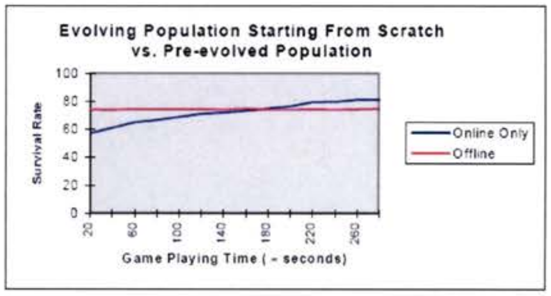
Where V is a constant, which is awarded for finding a mine.

C is a constant that indicates the initial cost of being born.

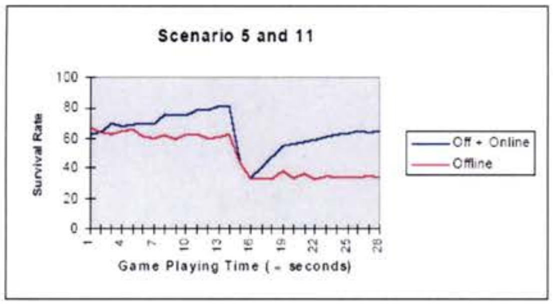
This measure rewards finding mines quickly, but also awards longevity.



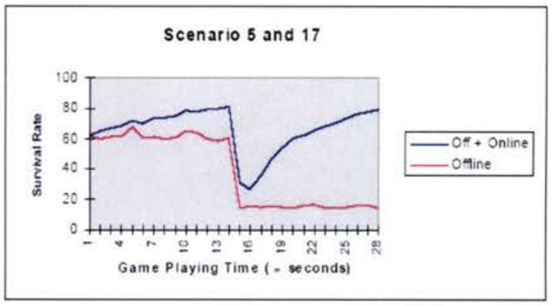
A



B



C



D

Figure 4.3 : (A) Average performance over all scenarios of a population that is allowed to evolve online compared to one that is not (Offline). (B) A population started with random weights that evolves online will outperform the population trained offline when given enough time.

(C) Even after the population has adapted to Scenario 5, it has no trouble adapting to a sudden change to Scenario 11. (D) The improvement is even clearer when the new scenario is the novel Scenario 17.

Results:

They have tested the algorithm with 16 different game scenarios. They evaluated the performance of offline and online evolution combined versus offline evolution. The results show that online evolution significantly improved the performance. When tested with a new scenario, online evolution performed better than offline evolution. They even claim that given sufficient time, online evolution can outperform offline evolution. They have suggested that online evolution can be used in the domains such as search engines, where evolution was not considered before.

There are some similarities between this algorithm and our algorithm:

- Both aim to achieve adaptation to dynamic environments
- Both algorithms try to fine tune the offline evolved networks in the environment

Our algorithm differs from this algorithm in several ways, they are:

- In this algorithm, online evolution is used to adapt to the change in the environment, whereas our algorithm uses ANN learning algorithms for the same purpose.
- This algorithm considers only feedforward networks, whereas our algorithm can handle recurrent networks.
- In this algorithm, architectural evolution is not implemented.
- In this algorithm, the role of ANNs is very limited.

CHAPTER 5 HYBRID LEARNING ALGORITHM

Overview

This chapter presents a Hybrid learning system for drifting environments This chapter discusses the details of the core algorithm. The approach presented in this chapter improves the performance of neural networks in drifting environments.

5.1 Introduction

Evolution and learning are the most fundamental processes of adaptation. Evolution itself has an ability to adapt to the internal characteristics or regularities of an environment and this area is well explored with successful results (EPNet by Yao 1999, NEAT by Kenneth Stanley et al) Hence evolution serves as the primary adaptive process. From an evolutionary point of view, learning has at least three adaptive functions (Miller & Todd, 1990)

- It can help and guide evolution
- It allows adaptation to the environmental changes, which are too fast for the evolution to track.
- It helps to overcome the size limitations of genotype by exploiting the regularities of the environment.

Hence learning helps the agent to partially control the input from the environment by developing the agents' behavior. Evolution can only optimize the performance of the agents for the next generation. But when an environment changes from one generation to another generation, the agents may not perform well in the present environment as optimization is made using the performance in the last generation's environment that is different from the present. By being sensitive to environmental conditions that could not be anticipated by evolution, learning can incorporate them in the agents' behavior (Stefano Nolfi et. al, 1995).

When combined with evolution, learning can use the regularities of the environment to build more complex phenotypes. Hence, learning is considered as a secondary adaptation process that provides a continuous active development due to its sensitivity to the dynamics of the environment

5.2 The framework of the algorithm

This algorithm is inspired by the ways living organisms evolve. Over the generations, living organisms employed mutations and crossover to produce better offspring. This process of Darwinian principle is effectively used in the existing algorithms for better results. But in real life the environment surrounding the generations is not static, and if the organisms do not adapt to the current changes in their lifetime they will be extinct in a few generations. The organisms not only change from generation to generation but learn to adapt to the changing surroundings in their lifetime. Lamarckian learning proposes the similar idea that the organisms pass on the learned knowledge, over their lifetime, to the next generations that in turn produce better offspring. For a static environment, we may choose to exclude the lifelong learning since its benefits are limited and can also be achieved without including lifelong learning. But this is not true for drifting environments. To survive in dynamic environments, the artificial intelligence agents need to learn in their lifetime.

Adaptation is defined (Nikola Kasbov, 2002) as:

1. A set of parameters that are subject to change during the interaction with the environment.
2. An incoming continuous flow of information.
3. A goal that is applied to optimize the software performance over time.

For a system to adapt to an environment, it should have the following components.

1. Data acquisition
2. Mechanism to provide general and adaptable frame work.
3. Knowledge acquisition.

As human beings are provided with sensors like eyes and ears to sense the surroundings, in our system, agents are equipped with sensors to acquire required information from the environment. Since evolution of human beings depended on both crossover and mutation of their chromosomes, we use a general framework that includes Genetic Algorithms (Gas) due to the availability of mutation and crossover operators. GAs are used to create a population of networks for every generation. The GA algorithm is influenced by the following issues:

a) Encoding
b) Population size
c) Genetic operators
d) Diversity

The knowledge is acquired through Genetic algorithms and ANN learning methods. GAs can acquire knowledge over generations and produce a collection of better-fit networks. When human beings are born, they are born with some knowledge encoded in their chromosomes. Although this knowledge provides basic abilities, in this ever changing world human beings have to learn in their lifetime to live better in the changing surroundings. For life long learning humans have to collect the signals or inputs from the surroundings and process them in the brain using their accumulated knowledge or experience to gather new experience from the resultant actions. We have used ANNs as agents' brains and by changing the dynamics of these ANNs with the help of learning rule, we make the agents learn to adapt to the environment.

When humans migrate to an unknown place, they use their basic knowledge to understand the surroundings and gather knowledge to adapt. This newly acquired knowledge is passed to their offspring either in written or oral form. This helps the offspring to adapt to the new surroundings quickly and efficiently. This initial knowledge is collected by placing some basic agents in a simulated environment that resembles actual environments. We let the evolution work on the agents over a number of generations recording the inputs from the environment, to the better performing agents, and their corresponding output values. To make use of these facts we divided our algorithm into two stages called offline and online. Both of these stages use the initial knowledge. The offline stage plays the role of first training the offspring and is used to provide a better platform for online stage. Also it attempts to accelerate the online process. Both stages combine evolution with learning but in different approaches. The Offline stage uses the collaborative approach where the learning of GAs and ANNs is integrated into a single system. The online stage uses the supportive approach where GAs and ANNs learning are applied at different stages. In large-scale applications, the offline stage can be removed, as it may prove computationally expensive.

5.3 Requirements for the Proposed Algorithm

5.3.1 Representation of Genotype

Representation or encoding plays an important role in evolution and affects the ease of conversion and details in mapping from genotype to phenotype. Representation influences other factors that affect the GAs and their applicability.

Real value encoding is more natural and closely represents a problem space. Often real world problems have variables that are continuous over a domain rather than discrete. Hence, for our algorithm we require real valued encoding of genotypes for the agents in drifting environments. The proposed algorithm attempts to combine basic types of evolutions on neural networks. Hence, the representation should be able to allow these combinations of evolution. It can be direct or indirect encoding as long as it satisfies the above requirements and allows all genetic operators to be present in the evolutionary process.

5.3.2 Population Size

The size of the population affects the performance of evolution. The diversity of agents depends on the population size. But the requirement of population size is application specific. Hence, we suggest experimenting with different population sizes for the application of concern.

5.3.3 Genetic Operators

The genetic operators are the vital components of the genetic application. If we use binary encoding for chromosomes then the use of traditional GA operators is natural and the application of GA operators such as mutation becomes simple. When we use real valued encoding, we may have to alter the traditional operators to be able to work on the real encoded chromosomes.
The genetic operators are broadly classified as:

1) Crossover
2) Mutation
3) Selection

The genetic operators are affected by two issues:

a) Representation scheme
b) Type of evolutions on neural networks

The representation scheme influences and changes the way the genetic operators work on chromosomes. For example, we can use GAs to evolve the weights, connections, learning rules and/or combination of all of these. So, this scheme also affects the selection of genetic operators. For example, the crossover operator may not be useful due to competing conventions problem associated with the encoding scheme. Our algorithm attempts to mimic the human evolution in which crossover is the main operator. Hence, the design of a crossover operator is also an important factor and representation should allow it.

Most of the present day algorithms tend to exclude the crossover operator due to the competing conventions problem that depends on their representation system.

5.3.4 Diversity

When we use GAs to evolve populations of ANNs, the degree of similarity or differences among the networks is an important criterion. If the population is not diverse, then the problem of crowding arises. Crowding is a problem in GAs where one individual is much more fit than the others, leading the population to concentrate around this individual and variations of it. This makes the population concentrate on a small region of population search space. Therefore, spatially distributing the population into species preserves the diversity, thereby providing an opportunity to increase the fitness. Hence, our algorithm requires the population to be divided into several species based on a numerical measure of the architecture. Thus by preserving the slowly maturing genes through the speciation, our algorithm maintains diversity in the population.

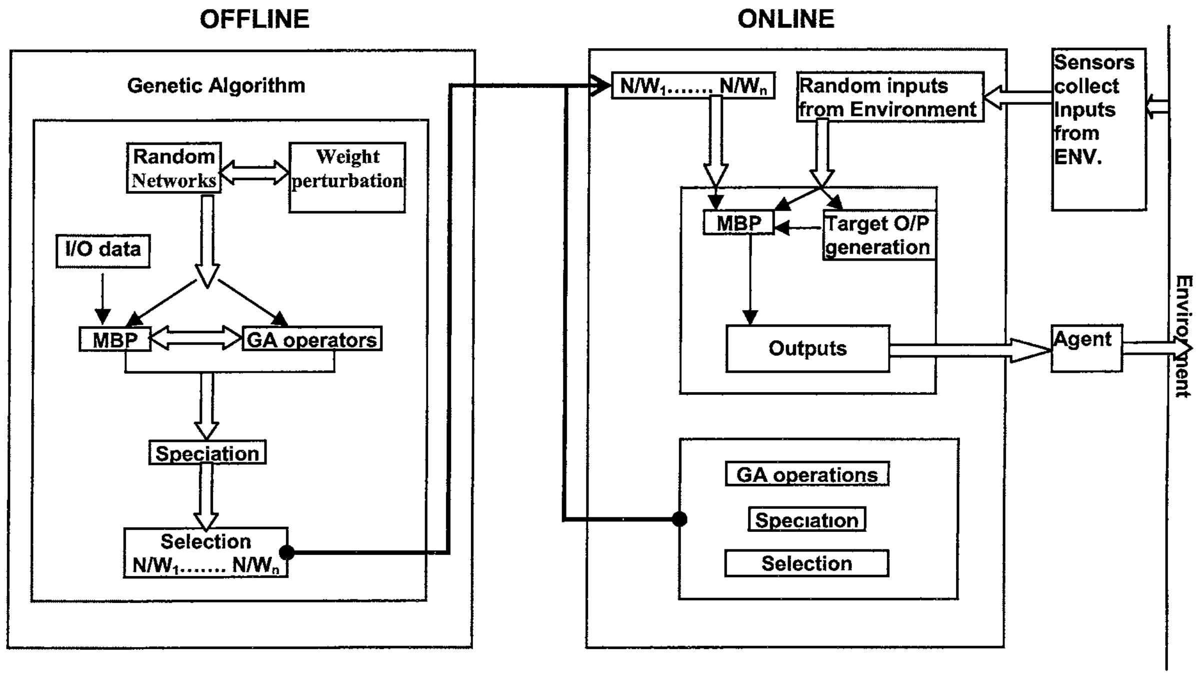


Figure 5.1 Frame work of the algorithm

5.4 Offline Stage

The system architecture for the offline stage is as shown in Figure 5.1

Step1:

Representation

In biological world, crossover occurs more frequently than the mutation, hence the true evolution needs crossover as its primary operator. Due to the competing conventions problem, the hybrid algorithms tend to leave crossover and solely depend on mutation.

We can encode the networks using real valued parameters and make them more applicable to the real world problems. For lifelong learning and better adaptation, the system should be capable of dynamically growing while possessing the ability to prune whenever the need arises. This process is possible by distinguishing the genotype from phenotype. The representation should also allow us to design such a flexible system. Genotype defines the state of characteristics in a collection of chromosomes called genome, and these characteristics are mapped into actual network via the phenotype.

Implementation Details

We have used a type of direct encoding scheme called, node based direct encoding for our chromosome representation. In the node based direct encoding scheme neuron and link genes are provided with all details. For example a link knows the neuron it connects to, the neuron it is coming from, and weight of that link. Our representation is inspired by the NEAT genotype architecture. This flexible representation allows us to map the genotype into the phenotype with ease and uses the "Innovation Number" concept introduced in NEAT.

The chromosome or genotype is divided into two genes:

a) Neuron gene
b) Link gene

The neuron gene contains a unique id called the innovation number for the neuron and information about the type of the neuron i.e. input or output. The Link gene contains a unique id for the link also called the innovation number, the information about the two neurons it connects, the real valued connection weights, whether link is recurrent, and most importantly whether it participates in the firing of neuron's output.

The innovation numbers for neurons and links help in overcoming the competing conventions problem. Our innovative representation allows us to design not only feedforward ANNs but also recurrent ANNs.

Step 2:

We begin by generating the population of networks or genomes for future steps. Conceptually, our algorithm does not suggest starting with a large number of nodes and pruning them when necessary. This method seems less efficient and may result in large architectures. This is even possible when we have a fixed number of output and input neurons. Therefore, we start with a minimum set of nodes and add nodes when it is necessary.

Our algorithm evolves a random and dynamic network of neurons. We do not have layers of hidden neurons rather we have individual hidden neurons. In this step we generate initial random networks with zero or more hidden neurons.

Implementation Details

We keep the number of input and output nodes fixed to enable supervised training. Hence, we generate a population of genomes with only the input and output nodes and random weights. Later, a few of those genomes are perturbed to have hidden nodes. This step allows us to grow near optimal genomes even for a large-scale application.

Step3:

We map the genotypes into phenotypes to create actual neural networks from the minimal genomes created in step2.

Implementation Details

Now, we consider the genotypes of each chromosome. By using the information provided in the neuron gene and link gene, we build a phenotype or actual neural network with all the input, output, hidden nodes, and the synapses connecting them.

Step 4:

Train the networks generated in step3 using a modified backpropagation (MBP) algorithm. The MBP is designed for the random neural networks with hidden nodes rather than networks with hidden layers.

Implementation Details

As our networks are dynamically generated, we do not have any hidden layers. Instead we have randomly introduced hidden nodes between input and output layers. The general backpropagation method for neural networks works only on layers of neurons. With dynamic networks, arranging the randomly generated neurons into layers is difficult. Hence, we modified the backpropagation to work with individual hidden neurons rather than working with hidden layers.

The following steps describe our MBP algorithm:

Prerequisites: each neuron has a means of calculating and storing the number of outgoing links. This is stored in a variable called 'error-status'.

```
For each input/output pair in the training set do the following:
   1) Apply the inputs to the input layer
   2) Propagate the inputs through all hidden and output neurons
      generate the corresponding output.
   3) a) At output neurons set the corresponding 'error-status'
      variables to '0', indicating no further output neurons are
      connected to these neurons.
      b) Calculate the error at output neurons using desired and
      produced output values.
      c) Adjust the weights of all the connections coming into these
      neurons.
   4) a) Now for each hidden neuron set the 'error-status' variable to
         the number of output neurons it is connected to.
      b) Whenever the error from an outgoing neuron is calculated
         reduce the 'error-status' variable by '1'.
      c) When 'error-status' variable value is zero, we have collected
         errors from all outgoing neurons and hence modify all incoming
         synapses weights.
      d) Repeat the process until the incoming neurons are input
         neurons.
```

The MBP training is set for fixed number of iterations and the training error is used as a measurement to rank the networks. Offline learning is intended for accelerating the process. We calculate the fitness as 1/error and rank the networks from most fit to least fit.

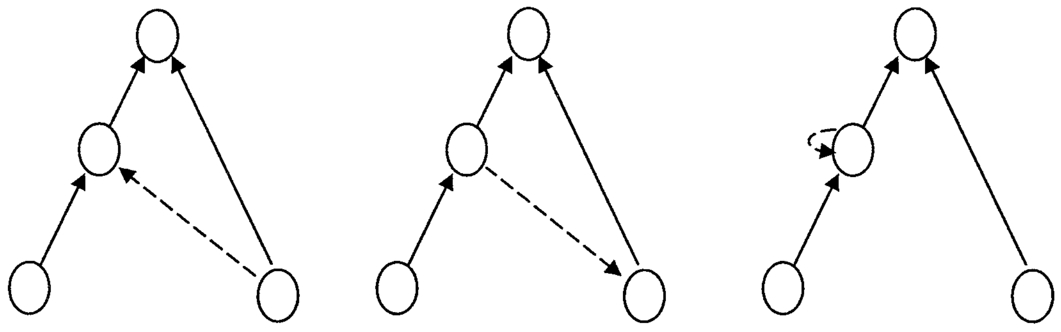
Step5:

We apply the genetic operators on the population. The crossover and/or mutation operators are applied on the sorted networks to produce offspring.

Implementation Details

We have used four types of mutation operators.

1. Add Link: we can add either forward link, feedback, or recurrent link between two nodes.



a. Forward lınk b. Feedback lınk c Recurrent lınk

Fıgure 5.2

2. Add Neuron: we add a neuron between two neurons bisecting the connection. The connection weıght value is divıded approxımately equally between the two new connections formed due to the bisection of old connectıon. Figure 5.3 depicts the process.

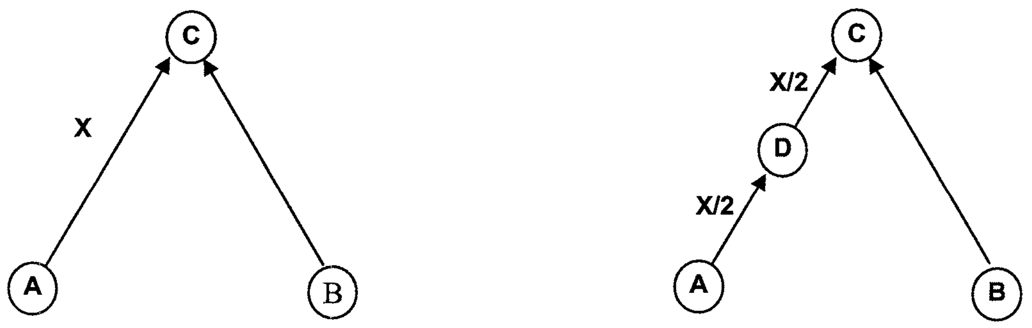


Fıgure 5.3 Before and after addıng a neuron 'D' between neurons 'A' and 'C'

The above two mutations are mainly archıtectural mutations.

3. Weıght Mutation: We perturb each connectıon weıght with a predefined mutatıon rate.

4. Mutation of Activation for Response Curve: Mutating the activatıon of the response curve helps in evolving the learning rules. This is achieved by perturbing the controller variable in

the sigmoid activation function and thus changing the range of threshold for the firing neurons.

Sigmoid function $f(a) = 1/(1+e^{-(a/c)})$

Where 'a' is the activation value

'c' is the controller variable.

The controller variable 'c' affects the shape of the curve. This mutation helps in evolving the learning rules.

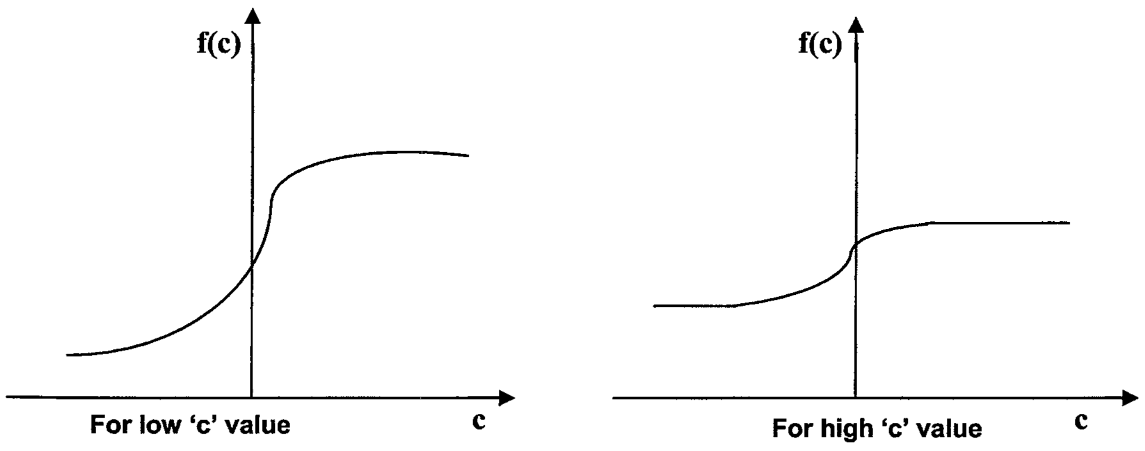


Figure 5.4 The sigmoid function for different values of controller variable (c)

Crossover:

While being an important evolutionary parameter in the biological world, crossover was omitted in most of the present evolutionary algorithms due to the competing conventions problem it creates. This problem makes the crossover operator inefficient in producing better offspring. Using a unique global numbering scheme for links and neurons, we can avoid the production of invalid networks. Based on these unique identification numbers, the genes are tracked and aligned chronologically. Matching genes are inherited randomly. Suppose two genomes are selected for the crossover. Their genes are ordered according to their unique global numbers. One genome may contain some genes that are not present in the other genome at a particular position. The genes that are not present in both genomes and are not present in either the beginning or the end of their respective sorted genomes, are called 'disjoint genes'. But the genes that are not matched and are either at the beginning or at the end of their respective sorted genomes, are called 'excess genes'. Disjoint and excess genes are inherited only from the fittest parent.

Step 6:

In order to preserve diversity, we speciate the networks into different species using a mathematical criterion based on architectural parameters. This speciation into groups not only mimics natural evolution but also helps in avoiding the crowding effect.

Implementation Details

We calculate the 'compatibility distance' using architecture specific measures like disjoint and excess genes. If the compatibility distance is within certain boundaries, then the individual is added to that species. If the individual is incompatible with all the current species then a new species is created and the individual is added to this newly created species.

Step 7:

Perform steps 4 to 6 until required fitness is achieved.

Implementation Details

Each time we execute steps 5 and 6, we perform step 4 to calculate the MSE. If the error is less than or equal to the minimum value, the loop is terminated. The other approach is to run steps 4 to 6 for a fixed number of iterations regardless of the MSE. If we follow the second approach, we can avoid applying MBP each time we perform steps 5 and 6.

Step 8:

Select 'N' fit networks for the next phase.

Implementation Details

We have used Roulette wheel selection due to its simplicity and effectiveness.

5.5 Online Stage

The architecture for online stage is diagrammed in Figure 5.1. The online stage is similar to the humans applying and updating their acquired knowledge in new surroundings and passing it to the next generations for better adaptation to the new surroundings. This is the stage where the advantage of our algorithm is observed and tested with agents in a drifting environment. We create intelligent agents with neural networks brains. The 'N' networks selected from the offline `stage are used as the brains of the agents. If the offline stage is not available then we create random minimal networks and use them as the brains of intelligent agents. The online stage follows the collaborative approach rather than the supportive approach for hybridization of evolution and learning. The online stage is the combination of two phases that toggle

1. Evolution
2. Learning online or on field

The offline stage is sometimes optional and used only to accelerate the online learning process. When the offline stage is not present, we first apply 'evolution phase' and then 'learning online phase'. Otherwise we can begin with the online learning stage.

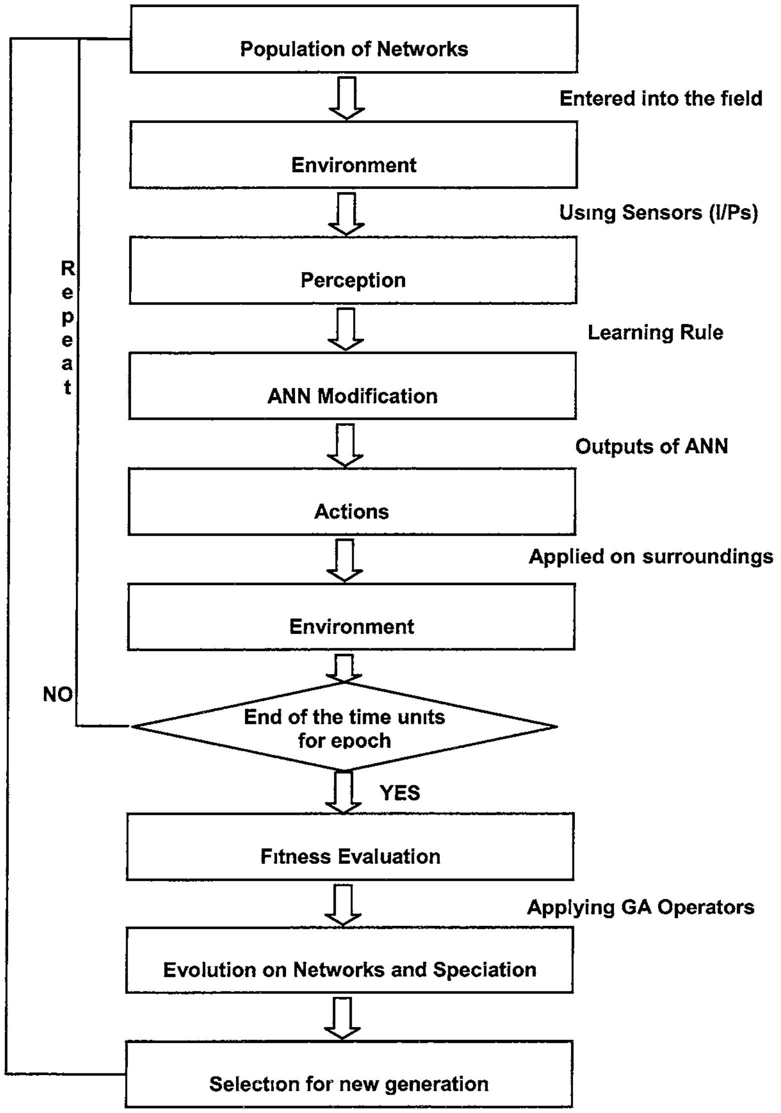


Figure 5.5 Frame work of ONLINE stage.

5.5.1 Learning Phase

This phase generates networks that can learn continuously, rather than using pre-learned networks. The networks that can learn can adapt more efficiently to the subtleties of the environment. The performance difference between learned and learning networks can be observed clearly in dynamic environments. This notion is supported by J.M Baldwin's (1896) views:

a) If the environment is continually changing, those individuals capable of learning and adapting quickly to the environment will have greater advantage compared to other individuals.
b) Those individuals who can learn and adapt quickly will have less dependence on the genetic code and will help to achieve more rapid evolutionary adaptation.

In this stage, the learning method used is called online learning where the network parameters are updated after the presentation of each example.

The steps performed in the learning stage are described below:

Step1:

The basic knowledge is incorporated into networks using an evolutionary phase where genetic algorithms are applied to them. By inserting these networks as their brains, we make the agents intelligent.

Implementation Details:

The agents equipped with the networks from the offline stage or evolution phase enter into an environment that changes from one generation to the next.

As humans have a lifespan of certain number of years, the agents are given a fixed number of time units to explore their environment and learn to adapt. This can be considered as a lifespan for the agents in a generation.

Step2:

The agents' world consists of many obstacles and they must achieve certain goals in their lifetime. To avoid the obstacles, agents should have a means of sensing the environment so as to avoid the obstacles while reaching their goals.

Implementation Details

The agents perceive the environment with their sensors and the sensor readings serve as inputs to the ANNs.

Step 3:

Similar to the human tendency of using oral or written knowledge to gain experience about their surroundings in their lifetime for better living, our agents are provided with a learning mechanism to make them more adaptable to the changes and dynamics of the environment.

Implementation Details

The agent's brains (ANNs) are updated using a learning method from input collected from the sensors. Our algorithm is based on supervised learning, hence we only modify online backpropagation algorithm where the input is collected randomly from the environment. The algorithmic parameters are modified for each input, hence, to reduce the loss of previously learned knowledge, we employ a history sensitivity function. The online learning can be of two types:

i. Global
ii. Local

Global Online Learning:

In this type of learning, the exact desired values are not required.

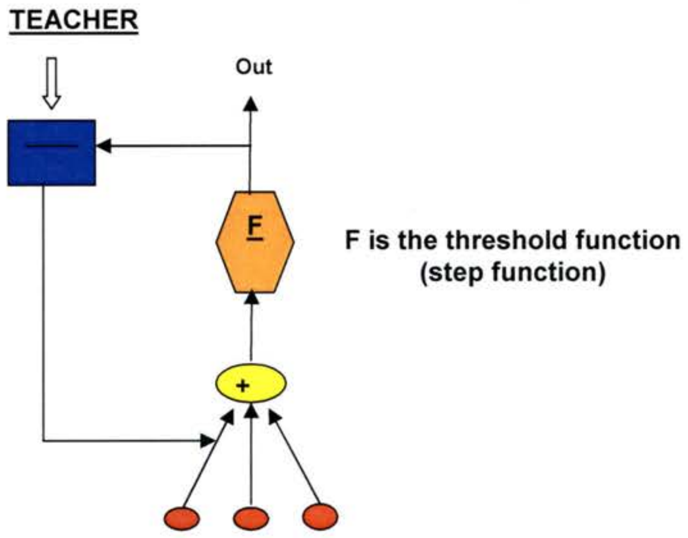


Figure 5.6 Illustration of Global Learning

In the global online learning, the inputs are random and the network does not have the exact desired outputs, making it difficult to apply supervised training. Hence, when using global online learning, we use one or more fitness parameters to produce the desired outputs for each set of random inputs from the environment. In the global online learning, we do not modify network parameters such as connection weights using a training set, but can optimize the networks using one or more fitness parameters.

Advantages:

1. It does not require any training input-output set.
2. Optimization depends solely on the parameters that affect the fitness.

Disadvantages:

1. Cannot optimize the networks for the environment in the current generation.
2. Needs to produce target output for each random input from the environment.
3. It is difficult to include all fitness parameters to produce a good target output set for random input set from the environment.

Local Online Learning:

In local online learning, the objective of neural network training is to find optimal network parameters (e.g. Connection weights) to minimize the error between the desired value and the actual response. The local online learning uses a set of input-output pairs to guide the network learning in a relatively new environment. We need to use a filter that compares the random input collected from the environment and selects an output of a closely matching input from the training pair. These outputs are used as target outputs. Local online learning optimizes the network fitness by changing parameters like connection weights in the current generation. The effectiveness is affected by the learning method and the training set.

Advantages:

1. Simple to use.
2. Optimizes the fitness function to adapt to the dynamics of the environment in every generation.
3. Dependency on evolution is less when a stable architecture is found.
4. Accelerates the evolution towards adaptation.

Disadvantages:

1. The design of the training set requires a lot of expertise and time.
2. The optimization is greatly affected by the efficiency of the training set.
3. The dependency on environment fitness parameters is less.

The online Modified Back Propagation (MBP) that is used to train the networks is similar to its offline counterpart with the following differences:

- It is designed on online learning principle hence the network parameters are modified on application of every input and parameter modification does not guarantee the desired output on re-application of the same input.
- History sensitivity function is used to reduce the amount of learning over time. This function is designed in such a way that learning is faster in the beginning and decreases over time to preserve past learning.

Step 4:

The steps 2 and 3 are repeated for a fixed number of time units. These time units indicate the life span of agents per generation.

5.5.2 Evolutionary Phase

This phase is applied between generations GAs are used to identify the superior architecture, weight and learning rule to determine a set of best fit networks for the next generation. This phase is also similar to the one in the offline stage but differs in the method of usage and order of application of its operators and also in fitness evaluation. The important operators used are:

1) Crossover
2) Mutation
3) Selection

This stage also performs speciation and evaluation. The goal of evolution is to build 'N' fit networks for the learning stage.

Step1: Representation

This is similar to offline stage representation scheme. The representation should allow all three kinds of evolutions (weight, architecture, learning rule) as well as their combinations.

Step 2: New Population

In this step, we generate a new population from the current population. The genetic operators are applied on the current population and thus new population is generated.

Implementation

If the current generation is empty, we generate random initial networks with and without hidden nodes. The networks have fixed input and output neurons. In all the other cases we apply genetic operators.

1. *Crossover:*

We generate a random variable for each parameter and compare it with user defined crossover constant, and if it is greater, we then perform crossover. The crossover constant value can be set at the beginning and is constant throughout the process. The crossover operation is similar to the offline stage.

2. *Mutation:*

Mutation is performed in 5 ways.

i. Add link
ii. Add neuron
iii. Weight mutation
iv. Mutation of activation response for responsive curve
v. Mutation of learning rule parameters

The first four mutations are similar to offline learning. Mutation of learning parameters can be implemented in a similar way to that of weight mutation.

We have implemented the learning rule as a linear equation with 6 parameters to enable the evolution of learning rule for connection weight modification.

```
ΔWeight = p0 * (p1 * weight - p2 * error * learning rate - p3 * weight
          * learning rate + p4 * error + p5 * output * learning rate)
```

Where, p0 is a real valued variable used to scale the result, and p1, p2, p3, p4 & p5 are real variables.

This general linear equation tries to reduce the amount of modification applied to the weights with respect to the error.

We generate a random variable for each parameter and compare it with standard mutation rate and if it is greater, we perform mutation. We use Gaussian mutation method.

The design of the learning rule is based on the following important criteria.

- Outputs generated
- Error from target outputs
- Learning rate

Step 3: Selection

Here we perform two tasks

a) Maintaining diversity
b) Selecting best individuals

To preserve the diversity, we divide the population into different species This process of speciation is similar to the speciation in offline stage. We calculate the average fitness for each species using the age and the performance of networks in the environment. Networks are sorted based on their fitness in each species, and most fit network from each species is added to the new population intact. The rest of the new population is selected from the networks generated using genetic operators and their fitness. We select the required 'N' networks from all the species depending on their average fitness.

Implementation Details

Speciation is similar to the offline method which uses "compatibility distance" measure to speciate the generated networks. Fitness is designed on agent's efficiency to avoid the obstacles while fulfilling its goals.

Step 4:

'N' networks are selected to perform online.

The learning and evolution phases are repeated until some terminating criterion is met

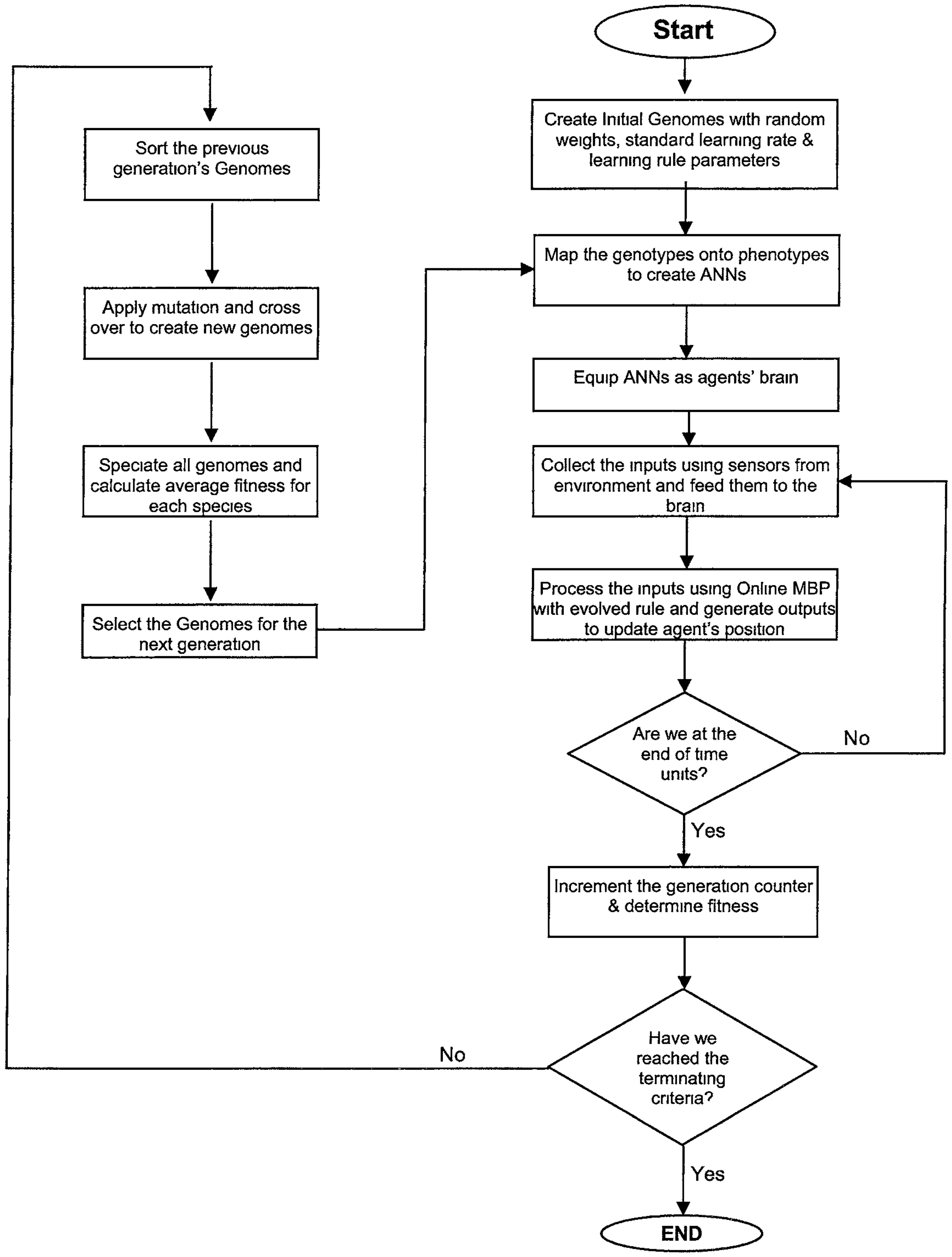

Start
Create Initial Genomes with random weights, standard learning rate & learning rule parameters
Map the genotypes onto phenotypes to create ANNs
Equip ANNs as agents' brain
Collect the inputs using sensors from environment and feed them to the brain
Process the inputs using Online MBP with evolved rule and generate outputs to update agent's position
Are we at the end of time units?
No
Yes
Increment the generation counter & determine fitness
Have we reached the terminating criteria?
No
Yes
END
Sort the previous generation's Genomes
Apply mutation and cross over to create new genomes
Speciate all genomes and calculate average fitness for each species
Select the Genomes for the next generation

Figure 5.7 Flowchart of ONLINE stage

CHAPTER 6. APPLICATION ANALYSIS

Overview

This chapter introduces and analyses the mine sweeper application implemented using our algorithm.

6.1 Introduction

Our algorithm attempts to develop intelligent agents which can adapt to a changing environment effectively and more quickly than existing implementations. Since the algorithm is inspired by human behavior and evolution, we need an application that allows us to test and observe all aspects of the proposed algorithm. A mine sweeper application is used to demonstrate the capabilities of our method in adapting to drifting environments. The mine sweeper's initial framework is implemented by Mat Buckland (AI techniques for game programming, 2002). We modified the classes and the visualization graphics in this framework to implement our algorithm. In our application, we use neural networks to control the behavior of the mine sweepers and to make them intelligent. The mine sweepers live in a drifting environment with a few different obstacles and several mines The positions and shapes of these obstacles change from one generation to the next. The goal of the application is to evolve intelligent mine sweepers to explore as much area as possible, while avoiding the obstacles within certain time limit.

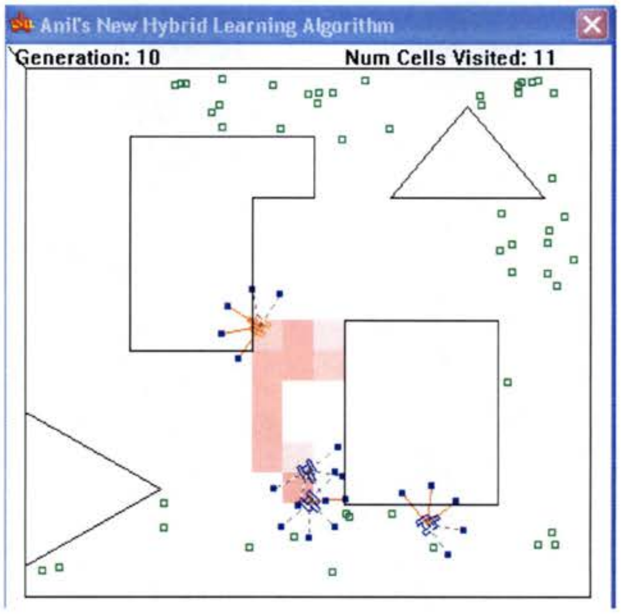


Figure 6.1 The demo program in action.

The mine sweepers that collide with obstacles or walls appear in red. They remain in red until they move away from the obstacles or walls. The others are shown in blue. When F key is pressed the graphics are hidden from view and the statistics are displayed instead. The application starts in two windows, one showing the mine sweepers exploring the environment and the other displaying the best networks from the previous environment.

6.2 Architecture of ANNs

To design the architecture of ANNs, we need to determine the required number of inputs and outputs and a mechanism to obtain the inputs from the environment. To determine the number of inputs for the ANN, we need to recognize the type of information a mine sweeper needs to navigate through the environment and the issues related to acquiring that information. This application involves solving two game related problems.

- Obstacle avoidance
- Environment exploration

6.2.1 Obstacle Avoidance

Obstacle avoidance is a very common task in game theory. It is the responsibility of the game agent to perceive its environment and to navigate without colliding with the obstacles in the game world.

To perform successful obstacle avoidance, the agent must be able to perform the following:

- Observe its environment
- Take action to avoid potential collisions

To observe the environment, the agents (mine sweepers) must have a way to see the world. Mine sweepers are equipped with a number of sensors, which enable them to perceive the obstacles in the world around them. The sensors are the line segments that radiate outward from the center of the mine sweepers' bodies. Sensors, which are represented as vectors, have a direction and length associated with them.

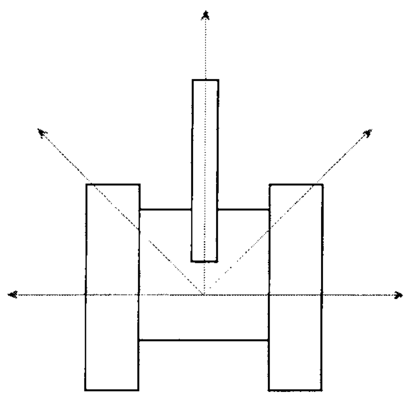


Figure 6.2 A mine sweeper with sensors

In our experiments, mine sweepers can have any number of sensors with various lengths. However by default, a mine sweeper has five sensors that radiate outward for 25 pixels. Every time unit of a generation is divided into certain number of frames. The mine sweeper's sensors explore each frame for possible obstacles in the game world. Every mine sweeper is equipped with a mechanism to determine the distance to any obstacle it may encounter. The distances between the mine sweeper and the obstacle are measured using sensors. The closer the object is to the mine sweeper, the closer to zero is the reading provided by the sensors. When there are no obstacles intercepted by the sensors, then the sensors provide a value of -1.

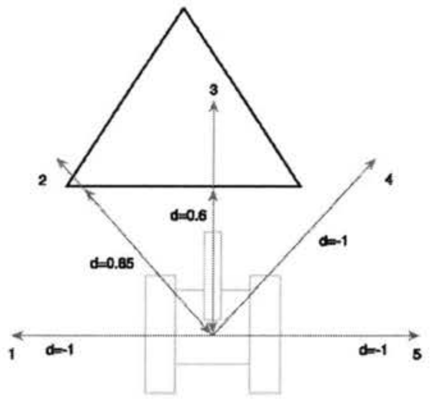


Figure 6.3 A mine sweeper seeing the obstacle through its sensor readings.

To check whether a mine sweeper has actually collided with an object, we check the readings provided by its sensors. These readings are compared to a collision distance value that is calculated from the scale of the mine sweeper and the length of the sensor line segment.

6.2.2 Environment Exploration

Equipped with only sensors, the mine sweepers can see the obstacles and learn to avoid them in a few generations, but they do not explore the environment efficiently since they do not have any guidance. To develop a useful behavior for exploring the environment, in addition to learning to avoid the obstacles, mine sweepers need additional guidance for exploration. This guidance is provided in the form of memory. The environment is divided into a number of equal sized cells. These cells are represented by a simple data structure. This data structure is used as a memory map to store information about the number of time units a mine sweeper has spent in that cell. This information helps the mine sweepers to evolve the weights, architecture and learning rules of the ANNs to favor the unvisited cells.

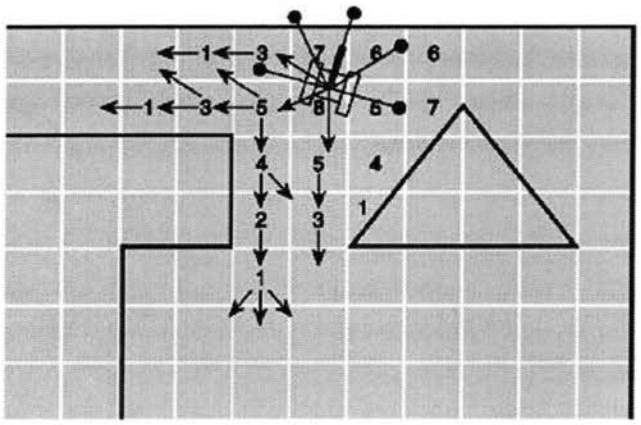


Figure 6.4 The memory readings help the mine sweeper to explore unvisited cells in the environment.

The end points of the sensors act as antennas for the mine sweeper and retrieve the information stored in the cell. These end points are referred to as feelers and the readings from these feelers enable the mine sweepers to navigate the environment. The number of time units a mine sweeper spent in the surrounding cells is retrieved by these feelers. Using this information, feelers provide the corresponding readings which are between -1 and 1. For example, if a mine sweeper previously spent 0 time units in a surrounding cell then the corresponding feeler provides a reading of -1. If it spent 20 time units in a surrounding cell then the reading would be 0.2, and if it spent 80 time units the reading would be 0.8. if it spent 100 or more time units in a cell then the reading would be 1.

With these feeler and sensor values the mine sweeper can navigate through the environment. The readings from feelers along with sensors are used as inputs to the neural network. An additional input is supplied to indicate whether the current mine sweeper has collided with some obstacle in the environment. Therefore, the default number of inputs for the neural network would be 11, namely, five feelers, five sensors, and an additional input indicating collisions.

6.2.3 Outputs

The number of outputs for the ANN depends on how we control the movements of the mine sweepers. We assume that mine sweepers run on two tracks. Tracks are the endless metal belts on which vehicles such as battle tanks travel. The rotation and velocity of the mine sweepers are adjusted by altering the relative speed of the tracks. Hence, we need two outputs, one for each track. To make the movements more realistic, we need to produce real valued outputs for each track. This can be achieved by using a sigmoid function as the activation function for the output neurons. The rotation and speed of a mine sweeper are determined using the outputs generated for the left and right tracks. The mine sweeper's rotational force is calculated by subtracting the force applied by the right track from the left track. The mine sweeper's speed is the sum of the values of left and right tracks. With this information about the inputs and outputs of the agents, we can proceed to discuss the details of the network's architecture and encoding.

We start with a minimal architecture that includes few networks with hidden nodes for effective exploration of architectural search space. We have used a direct encoding method called node-based encoding. Node-based encoding encodes all the required information about each neuron in a single gene. For each neuron (or node), its gene will contain information about the other connected neurons and/or the weights associated with those connections.

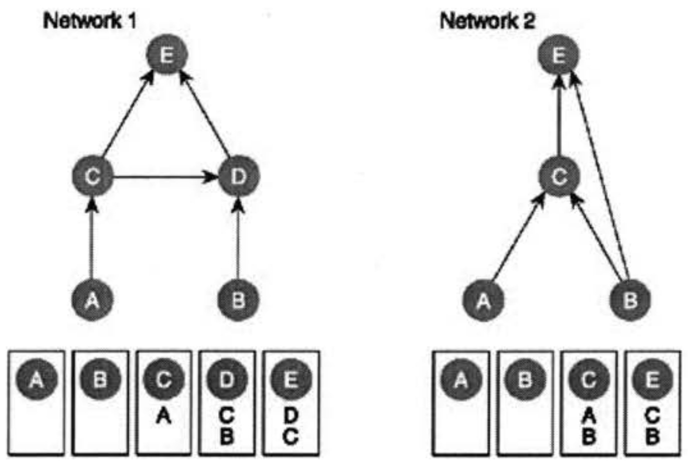


Figure 6.5 Two networks with their chromosomes using node-based encoding.

Our application uses a genome structure containing two kinds of genes namely neuron genes and link genes. Both of these genes contain information about their connectivity and respective parameters. Both of these genes make use of a concept of a unique number called the innovation number (Kenneth Stanley at al., 2000) to avoid the competing conventions problem. These innovation numbers are provided for both neurons and links and hence are present in both the neuron and link genes. The links can be forward or recurrent, whereas neurons can be of input, output, hidden or bias types.

Genotypes:

NeuronGene
Begin

Innovation number: *It is the unique id for the neuron*
Type of neuron : *This indicates whether the neuron is input, output, hidden or bias*

End;

LinkGene
Begin

Innovation number : *Unique id for the link*
Link from neuron : *Id of the Neuron from which link comes from*
Link to neuron : *Id of the Neuron to which the link goes to*
Weight : *A real value attached to the link*
Recurrent : *Indicates whether the link is recurrent or not*
Enabled : *Indicates whether the link is active or not*

End;

Figure 6.6 The neuron and link genes' parameters and their description

After creating the genotypes, we need to create actual neural networks with all the neurons and the links among them. This mapping from genotype to phenotype is implemented in a container class called Genome. The Genome class contains both genotype objects and phenotype objects. The phenotype object has information about learning rate and learning rule parameters. The learning rate is common for all the neurons in the network. In addition, the learning rule parameters are used in evolving learning rules for the whole network.

We start with genomes containing zero or few hidden neurons and evolve them into larger architectures with improved fitness. This approach helps in maintaining small architectures and is inspired by two facts:

1. Nature has evolved from small (less complex) organisms to the larger (more complex) life forms.
2. By including genomes with hidden neurons, in addition to minimal genomes (genomes with zero hidden neurons) in the initial population, genetic algorithms can have a larger architectural search space.

6.3 Our Framework

The mine sweeper application is controlled by a class called "*CController*". The *CController* class controls the relevant invocation of methods from various classes.

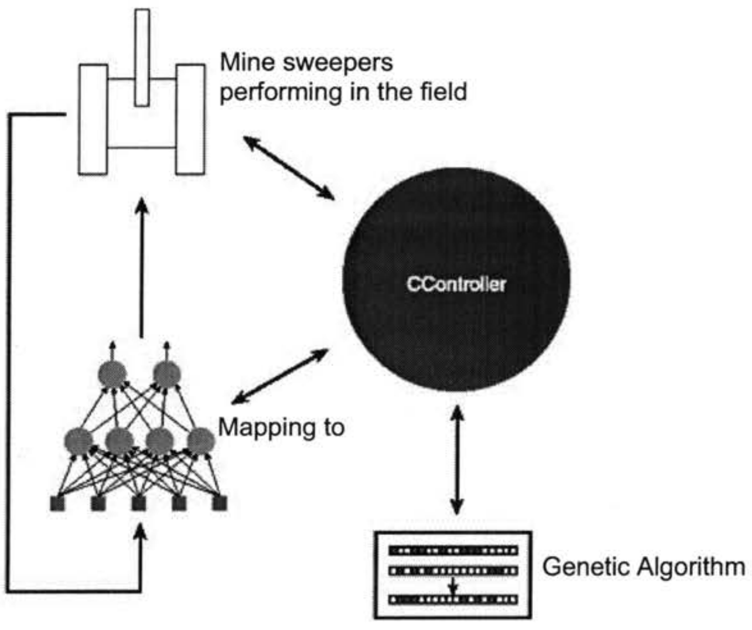


Figure 6.7 Program flow for the mine sweeper application

When an instance of the CController class is created, the following steps take place:

- Our framework provides an option to use offline training. If offline training is used then initial networks are obtained from the offline learning stage. Otherwise, the constructor generates the random initial networks for online stage.
- The generated networks are inserted into the mine sweepers
- For online stage, we create a random environment with obstacles for every generation.
- For online stage, we create all necessary graphical requirements to display the objects and mine sweepers.

6.3.1 Offline Learning Stage

Our algorithm uses offline learning stage to provide a better foundation for the online stage by generating networks with at least some knowledge rather than no knowledge. The offline stage is performed only once to speed up the rest of the process. We henceforth explain the step by step processing of this stage with references to the algorithm

i. Random Network Creation:

Initially a random population of neural networks are created and stored in a vector data structure. The information about number of inputs, number of mine sweepers and number of outputs is decided here. Consequently, we carry out the following steps:

- We create a population of genomes. These genomes contain only input and output neurons. They do not have any hidden neurons.
- To explore the search space of architectures with hidden neurons, we modify some of the genomes by inserting random hidden neurons.

```
For a network in the population
      Begin
            ▪   Search for a valid link.
            ▪   If a link is found then split the link into
                two different links.
            ▪   Assign a new innovation number for the new
                links.
            ▪   Divide the old weight into half and assign
                the value as new weights for the two links.
            ▪   Create a hidden neuron and assign a new
                innovation number to it.
            ▪   Set the two new links as incoming and out
                going links to this neuron.
      End
```

- We create and assign a unique innovation number to every neuron gene and link gene.

In this way, we create genomes with and without hidden neurons.

ii. Mapping Genotypes to Phenotypes:

Using the above created genomes, we map the genotypes into phenotypes to create actual networks. This mapping is performed using the information in the genes to build the neural networks from neurons by connecting the links between them. The links are assigned the weight information stored in their genes. These weights are assigned randomly when the genomes are initially created.

```
Procedure Create Network(depth of the network)
  Begin
        ▪ Create the neurons from the Genome information.
        ▪ Create the links from Genome information only for those
          links that are enabled.
        ▪ Create a link between relevant neurons and assign the
          weight stored in the link gene.
        ▪ Set the error status (i.e. the number of outgoing links)
          for each neuron.
  End
```

iii. Hybrid Training for the Networks:

In the offline learning, we use modified backpropagation (MBP) with the genetic algorithm's operators for refining and evaluation of created networks. But since MBP is supervised, it needs guidance to train the networks To provide this guidance, we placed several random mine sweepers with no learning ability in the environment. We evolved them for 50 generations, each generation with 600 time units. At the 50th generation, we stored inputs and outputs of the best performing mine sweepers We edited these input-output data to extract 250 input-output sample set. These samples served as training data for both offline and online MBP. The environment used for collecting the training data was static and was similar to one of the random environments. For a desired number of iterations, we do the following:

1. For each phenotype, we apply modified backpropagation algorithm and store the fitness of each network. The offline MBP returns the corresponding MSE.

```
For each network
  Begin
      Error = Function Offline MBP ( )
      Fitness of Network = 1/Error
      Store Fitness (Fitness of Network)
  End
```

2. We sort the networks according to their fitness values.

3. We apply genetic operators like crossover and mutation on the sorted networks. We generate a random number. Only when this random number is less than standard mutation rate, we perform the mutation. Otherwise, we do not perform the mutation operation. Crossover is also similarly performed.

There are four types of mutations performed in offline stage. They are:

- Add link
- Add Neuron
- Weight perturbation
- Mutation of activation response curve

These mutations are performed as described below.

Add Link: The new link can be either recurrent link or forward link.

```
Procedure Add Link ()
Begin
  Generate a random number
  If (Random Number Generated < Mutation Constant)
  Begin
    Generate a random number
    If (Random Number Generated < Recurrent link Constant)
    Begin
      Get a random neuron
      Add a recurrent link, if the neuron does not have one
      Assign an innovation number to the link
    End
    Find two unlinked random neurons
    Add link between these two neurons
    Assign an innovation number to the link
  End
End
```

Add Neuron· We add neurons only if the total number of neurons is less than the maximum number of neurons allowed.

```
Procedure Add Neuron( )
Begin
  Generate a random number
  If (Random Number Generated < Mutation Constant)
  Begin
    If (Total Number of neurons< Number of neurons allowed)
    Begin
       Search for a valid link.
       If a link is found then split the link into two
        different links.
        Assign a new innovation number for the new links.
        Divide the old weight into half and assign the value
        as new weights for the two links.
        Create a hidden neuron and assign a new innovation
        number to it.
        Set the two new links as incoming and out going
        links to this neuron.
    End
  End
End
```

Weight Perturbation: The mutation of weights is achieved using two different approaches. If a randomly generated value is less than a pre-defined constant, we replace the older weight with completely a new weight, else we perturb the weight by a small amount.

```
Procedure Mutate Weights ( )
Begin
    For each link in the network
    Begin
       Generate a random number
       If (Random Number Generated < Mutation Constant)
       Begin
         Replace the weight with a random value
       End
       Else

       Begin
           Add a small random value to the existing weight
       End
    End
End
```

Mutation of the Activation Response Curve This mutation serves as a preliminary evolution of learning rules.

```
Procedure Mutate Activation Response ( )
Begin
   For each neuron
   Begin
      Generate a random number
      If (Random Number Generated < Mutation Constant)
      Begin
         Add a small random value to the existing
         Activation response value.
      End
    End
 End
```

4. If mutations are not performed then we only perform the crossover operation. The crossover operation is executed only when the generated random number is less than the pre-defined crossover constant.

```
Procedure Crossover (parent1, parent2)
Begin
       Generate a random number
       If (Random Number Generated < Crossover Constant)
       Begin
          Find the Fittest Parent
          Add the Fittest parent's genes to the other parent
          If both parents are equally fit
          Begin
             For every gene in the child
               Begin
                 Select one parent randomly and add gene
                  from that parent
               End
          End
       End
     End
```

5. After performing crossover or mutation operators we once again apply the MBP to filter out the less fit genomes from the next population.

iv. Selection:

The required number of genomes is selected to be included in the new population using tournament selection method. In tournament selection 'n' individuals are selected from the population and the fittest of these genomes is chosen to be added to the new population. This process is repeated as many times as is necessary to complete the requirements of the new population.

v. Perform Iterations:

We repeat steps **i** to **iv** on the new population until we reach the desired iterations.

The required number of networks is passed on to the online stage.

6.3.2 Online Stage

The online stage is the core of our process. It can perform with or without the help of offline learning. The offline stage is only used to give online stage a good foundation with better fit networks in the beginning. Online stage works in two phases known as learning phase and evolutionary phase.

6.3.2.1 Learning Phase:

With the offline stage active, the networks developed in the offline stage are used as the initial brains of mine sweepers. Otherwise, initial random networks are created and inserted as initial brains of mine sweepers. Online stage is the core of the application that improves the mine sweepers' performance in a drifting environment.

In each generation, the mine sweepers search the environment for a number of time units. During each time unit, the ANNs of mine sweepers are constantly fed with the information from the surroundings. Depending on these inputs, the networks are updated using the modified backpropagation (MBP) learning algorithm. This version of MBP differs greatly from the offline version in the following aspects:

- This is an online version, i.e. it is updated after application of the input
- The amount of modification to the network parameters decreases over time
- It can use online gradient descent or can evolve the rule
- It can learn locally or globally

First, the input is processed by all the neurons to produce the outputs. Next the outputs of each neuron are collected. We fetch the desired outputs from the training set using a filter function.

```
Procedure Filter(input from environment)
Begin
   Min= infinite
   Index=0
   For each training sample
   Begin
       Find the distance between training input and the input
        from environment
       If (Min> distance )
            Index= Index + 1
    End
    Get the closest matching sample using Index
     Desired outputs =  Matched Sample Outputs
End
```

This procedure is used with local online learning. We have used 250 input-output training pairs to guide the mine sweepers in the random environment.

For global online learning, we do not use the input-output training set. Instead of using the fitness criterion, we generate the desired outputs for the current inputs from the environment.

In our mine sweeper application, an agent's fitness is determined broadly over three observations.

a) The number of collisions with objects or walls
b) The number of rotations
c) Speed of exploration

We use a simple heuristic function to generate outputs for global online learning. In our application the heuristic function uses the speed of the mine sweeper to produce the target outputs. To keep the function simple, we have used only one parameter (speed of mine sweeper) of the environment fitness criterion.

```
Procedure Online MBP ( )
Begin
      For each neuron
      Begin
            Set error status value to number of outgoing links from
            that neuron
      End
      Calculate the error for output neurons using gradient descent
      rule
      For each hidden neuron
      Begin
           If hidden neuron's error status is zero
           Begin
                  Calculate error using total error from its output
                  neurons
           End
           Else
           Begin
                For each neuron connected to this hidden neuron through
                 outgoing link
                Begin
                     Compute the total error
                     Reduce error status value by 1
                End
          End
     End
      Now update the weights with the error calculated using gradient
      descent rule
End
```

Using the mean squared error from desired and generated outputs, we update the weights with either delta (gradient descent) rule or evolved rule. We have implemented global and local online learning methods with the back propagation principles. The local and global learning methods differ from each other in only one way. Global online generates desired outputs using a heuristic function whereas local online uses a training input-output set. The modified backpropagation algorithm uses a history sensitivity function like f (t) = N/t where 'N' is a constant (typically N=1) and t is the number of time units elapsed. The history sensitivity function acts as a loss function, which preserves the previous knowledge while the networks learn online. If we choose to evolve the learning rule rather than delta rule, we use a linear general equation with five random real variables and a random real variable for scaling.

These five real variables are mutated after each generation depending on the difference between the user defined mutation constant and a random real value generated. The weight change is described by the following function.

```
ΔWeight = F (Weight, Learning Rate, Output, Error)
ΔWeight = p0 * (p1 * weight - p2 * error * learning rate - p3 * weight
          * learning rate + p4 * error + p5 * output * learning rate)
```

p0, p1, p2, p3, p4 and p5 are positive constant real values, typically less than one, that regulate the modification of weights. These constants change from generation to generation. The weight updating process attempts to preserve the previously learned knowledge by including the old weight in the updating process Also by including the error value in the equation we let the network learn new information.

The general equation depends on four important parameters that affect the learning.
They are:

- Error at the neuron
- Output of the neuron
- Old weight of the link
- Learning rate

If we do not wish to evolve the learning rule then the online gradient descent rule is used for the modification of connection weights. This updating rule is applied to all the output and hidden layer neuron in-coming connection weights. It is continued until the desired number of time units per is reached.

6.3.2.2 Evolutionary Phase:

After a desired number of time units per generation has been reached, evolutionary phase begins. Evolutionary phase applies genetic algorithm operators on the current population to produce a better population for next generation.

We start this phase by calculating the fitness for each mine sweeper from the current population. We kill or remove the networks and species that are not improving over past few generations. The rest of the networks in the population are sorted according to their fitness values. Next we apply the genetic algorithm operators mutation, crossover and selection.

We speciate the networks using their architectural differences. Later we copy the best performing networks from each species without any modification into the new population. For the rest of the members of population, we use crossover and/or mutation on the current population.

The mutation and crossover operations are carried out analogously to the offline stage. The crossover, add link, add neuron, mutate learning curve response, and mutate weight use the same methods that are used in the offline stage. Unlike the offline learning, these operators are not iterated but are applied until a desired population size is achieved. The online stage has one extra mutation that is not present in the offline stage. If we choose to evolve the learning rule, then we need to mutate the learning rule parameters. This is carried out by applying mutation on the newly created population from the application of the other GA operators.

```
Procedure Mutate Learning Rule Parameters ( )
Begin
   Generate a random number
    If (Random Number generated < Mutation Constant)
    Begin
       For all learning rule parameters
       Begin
          Add a small quantity of random value
       End
    End
End
```

If there is an underflow of networks due to the rounding error, we apply tournament selection to select the rest of the networks from the old population. The new population of networks are inserted into mine sweepers as their new brains. The learning phase will now resume with these mine sweepers.

6.3.3 Performance Parameters

We can set various performance parameters. Some important parameters with their sample values are shown below.

iNumSensors	5	The number of sensors a mine sweeper can have
iNumSweepers	50	The mine sweeper population size
iNumTicks	300	The number of time units per generation
dLearningRate	0.5	Learning rate for the delta or evolved rule
dLearningParameter1	0.5	Learning rule parameter used in rule evolution
dLearningParameter2	0.5	Learning rule parameter used in rule evolution
dLearningParameter3	0.5	Learning rule parameter used in rule evolution
dLearningParameter4	0.5	Learning rule parameter used in rule evolution
dLearningParameter5	0.5	Learning rule parameter used in rule evolution
dLearningParameter6	0.5	Learning rule parameter used in rule evolution
iOfflineTraining	1	Option for having offline training
iGlobalOnline	0	When value is 1 global online method is chosen
iRuleEvolution	1	When value is 1 learning rule is evolved
iOnlyGAs	0	When value is 1 only GAs are used to update

When the application is launched, the F key speeds up the evolution, the R key resets it, and the B key shows the best four mine sweepers from the previous generation. The B key can only be used from second generation onwards, since it requires ANNs from previous generation. The previous generation's best mine sweeper is designed to leave a trail as it explores. The best sweepers also display their sensors and feelers.

CHAPTER 7 ANALYSIS OF RESULTS

Overview

This chapter discusses the results and improvements achieved by our algorithm using a mine sweeper application. Our algorithm helps in developing intelligent mine sweepers. We analyze the results with the help of screen shots and Excel graphs.

7.1 Introduction

Our algorithm assists the artificially intelligent agents (mine sweepers) by enabling them to learn in a drifting environment with the help of the acquired knowledge. Our experimental results show a considerable improvement in the performance of mine sweepers in a drifting environment. To prove our claim that lifelong learning combined with evolutionary process can boost the intelligence of artificially intelligent agents in a drifting environment, we tested our algorithm on several different scenarios. Our algorithm has two learning stages called offline and online. Offline learning is optional. Online learning is further divided into two more phases called the learning phase and the evolutionary phase The learning phase runs for the desired number of time units for each generation and the evolutionary phase runs between generations.

The following describes the type of experiments we carried out to prove the effectiveness of our algorithm.

- Performance of only evolutionary (genetic) algorithms
- Performance of offline learning and evolutionary (genetic) algorithms
- Performance of offline learning and online learning (learning and evolutionary phases)
- Performance of only online learning (learning and evolutionary phases)
- Performance of offline learning, local online in learning phase with evolutionary phase
- Performance of offline learning, global online in learning phase with evolutionary phase
- Performance of local online in learning phase with evolutionary phase
- Performance of global online in learning phase with evolutionary phase

While we observed interesting results, we also discovered the following influencing factors.

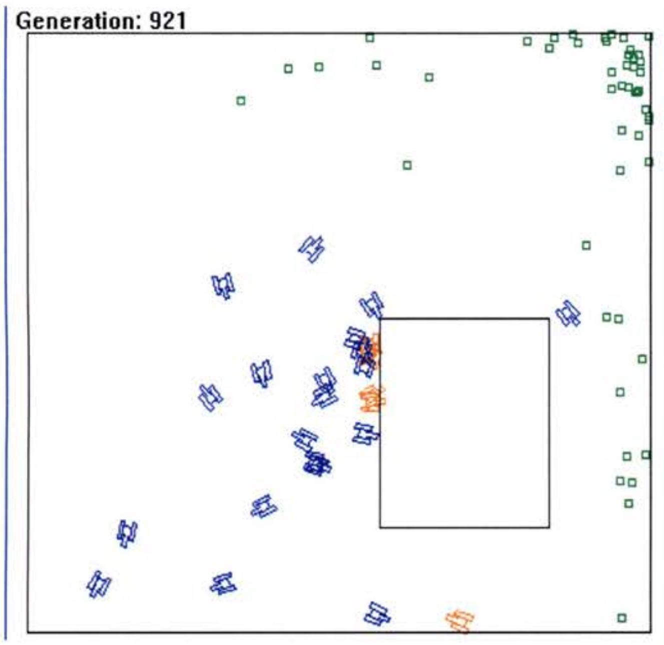
- Number of time units for generation
- Number of sweepers
- Number of generations
- Number of obstacles present
- Fitness criteria

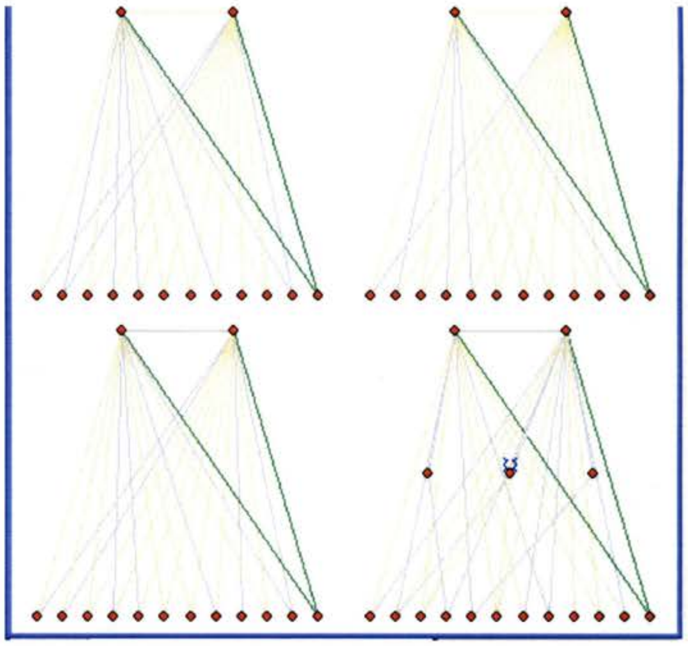
7.2 How Do We Analyze?

The simplest way to determine whether the mine sweepers are adapting to new environments is by looking at them while they perform. However, this method of observation cannot be documented. Hence we used two different fitness readings to assist us in evaluating the performance of the algorithm. The best ever fitness indicates the highest fitness value achieved by any agent (mine sweeper) in any generation until the present one, whereas generation's best fitness value indicates the highest fitness value achieved by an agent (minesweeper) in that generation. In the ideal case for an evolved network both fitness measurements should have the same values. However, these two fitness measurements may not be the same for environments having different number of obstacles and therefore different regions for exploration. In our experiments, there are two ways in which one can recognize an evolved network.

1. The generation's best fitness value should have little variation from the best ever fitness value.
2. The generation's fitness value should maintain its variation consistently from the best ever fitness over several generations.

The former indicates near optimal solution whereas the latter still has room for further improvements.





(a) (b)

Figure 7.1 (a) A screen shot showing the mine sweepers exploring the environment

(b) A screen shot showing previous generations best four networks

7.3 Only Genetic Algorithms

The smart minesweeper application is a combination of GAs and ANNs. In this experiment genetic algorithms are only responsible for the development of the networks and behavior of the mine sweepers. ANNs are merely used to generate outputs for each input. We initially start with a population of neural networks and after every generation GAs are used to generate a better population of networks using genetic operators such as mutation, crossover and selection.

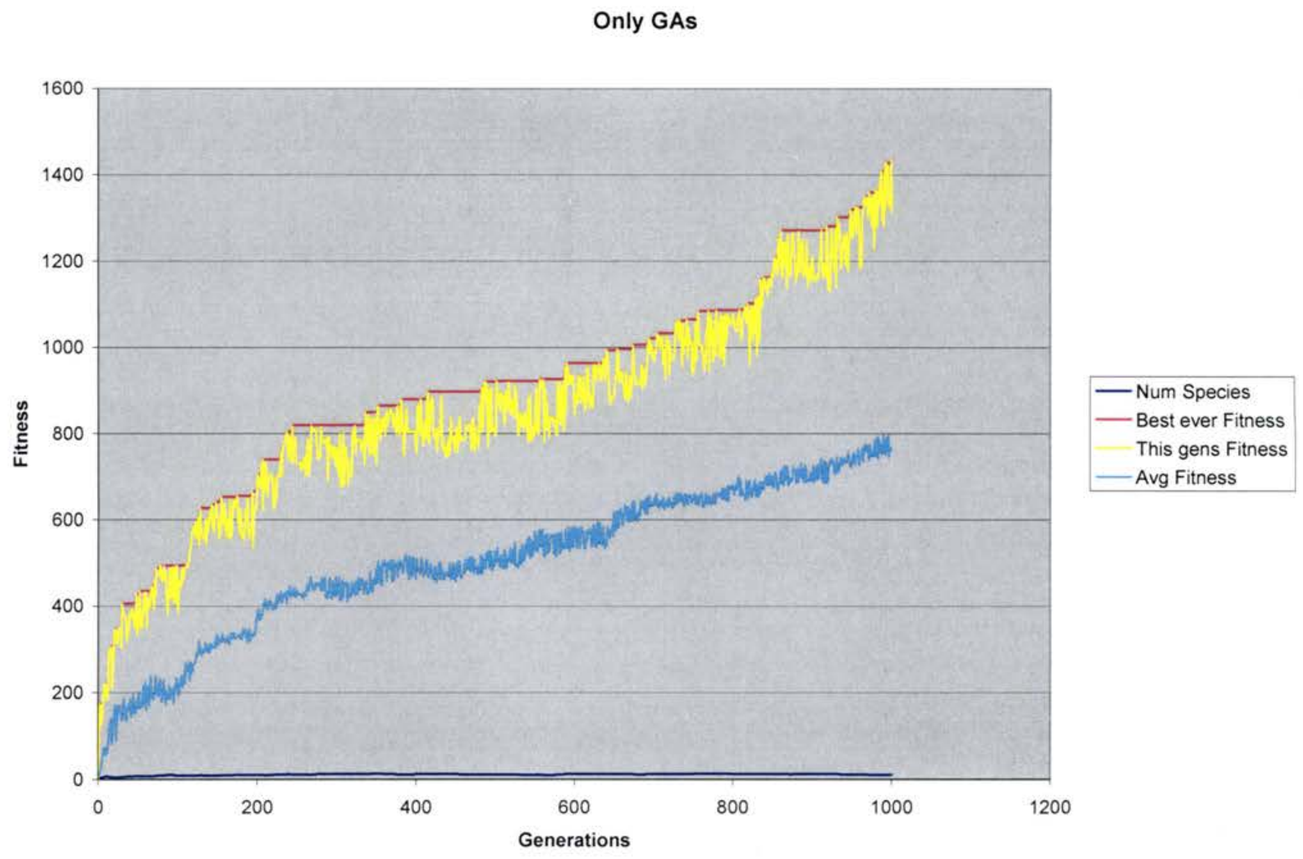


Figure 7.2 The fitness of intelligent agents with only GAs

Figure 7.2 depicts the performance of the intelligent agents equipped with only GAs. For 1000 generations, each generation with 300 time units, the maximum fitness ever achieved is less than 1500 units and the average fitness is under 800 units. Also each generation's fitness (shown in yellow) fluctuates across the best ever fitness (shown in pink) and also these fluctuations are random. This indicates that the GAs have failed to evolve a single best performing network.

7.3.1 Analysis of Performance with Only Evolutionary (Genetic) Algorithms

The performance of mine sweepers equipped with GAs alone was excellent in static environments. Within 300 and 500 generations, a best performing mine sweeper is found. But their performance in a drifting environment was not acceptable. They failed to capture the changes in the environment efficiently and in most cases they did not produce a best performing minesweeper over different generations. In drifting environments, intelligent agents equipped with GAs alone exhibited the following behavior:

- The Changes in the environment prompt the search of the architecture and weight space whenever fitness goes down. This will result in rather complex architectures.

- GAs (Genetic Algorithms) have frequently failed to produce a minesweeper that shows best performance over varying environments. Even when the generation's best fitness is close to best ever fitness, which was observed in several different mine sweepers, they performed well only in their specialized environment.
- GAs improve the population based on their fitness values in the previous environments. They do not consider the fact that the environment may change for the next generation. Hence, they generate better fit population for the environment in which the old population has performed. For drifting environment within 700 to 800 generations, the mine sweepers did not perform well in most of the experiments.
- If the mine sweepers, equipped with GAs alone, search the drifting environment for a long number of generations then the architecture of ANNs gets complicated.

7.4 Offline Learning and Evolutionary (Genetic) Algorithms

We start with offline learning initially and then apply evolutionary algorithms after every generation for further improvement. The offline learning is comprised of genetic algorithms and modified backpropagation algorithm. The genetic algorithms evolve weights and architecture simultaneously and MBP is used to further refine the networks. MBP is also used to test the fitness of the networks. The offline learning provides knowledgeable neural networks. As these networks gain some knowledge about the environment, they tend to reach higher fitness values in less time when compared with only evolutionary algorithms in drifting environments. But characteristics of offline learning and evolutionary algorithms both support only static environments. So offline learning may reduce the number of generations required to reach highest possible fitness, but does not really improve the performance in dynamic environments.

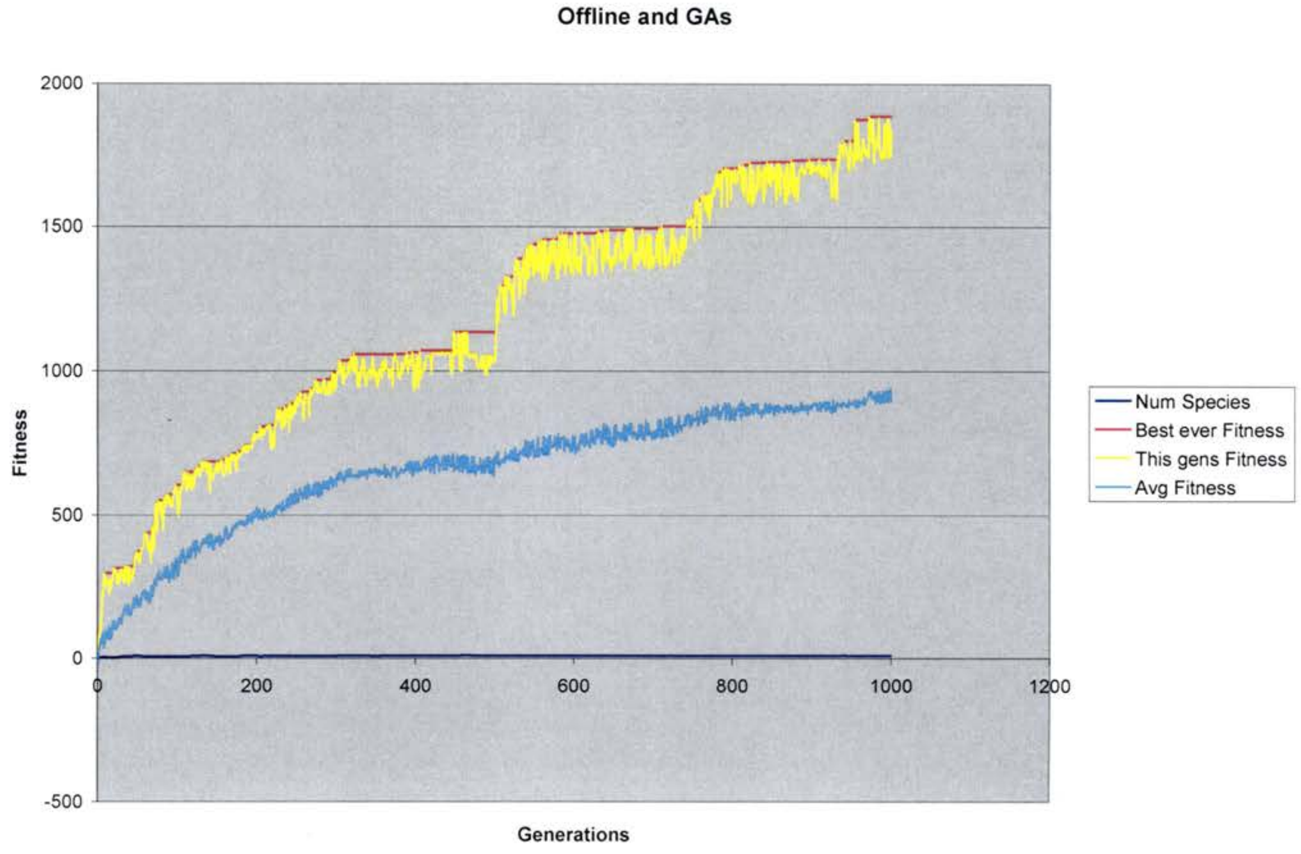


Figure 7.3 The fitness of intelligent agents when offline learning is combined with GAs

Figure 7.3 depicts that in 1000 generations, with 300 time units for each generation, the highest ever fitness reached is below 2000 units and the average fitness is less than 1000 units. We can also observe that each generation's fitness (yellow line in the graph) deviates from the best ever fitness (pink line in the graph) randomly. This indicates the probability of different best performing networks for different environments.

7.4.1 Analysis of Performance with Offline Learning and GAs:

Even with the addition of offline learning, mine sweepers were behaving similarly to those that used genetic algorithm alone. Offline learning is carried out only in the beginning and then genetic algorithms takeover. Therefore, initially the mine sweepers were performing better but as the effect of offline learning fades away their performance becomes similar to those that used GAs alone. Offline learning needs lot of resources. So we can not replace the genetic algorithms stage with offline stage. Both offline learning and GAs perform best in static environments and tend to perform poorly in drifting environments.

7.5 Offline Learning with Online Learning (Learning Phase and Evolutionary Phase)

In this approach, the ANNs benefit from both offline learning and online learning. The mine sweepers are initially equipped with the neural networks that were evolved using offline learning. They explore the environment with the help of their sensors and feelers In the learning phase the mine sweepers learn while they explore the environment. We use MBP to modify the weights of neural networks for every input collected by the mine sweepers from the environment. This helps the mine sweepers adapt to the intrinsic details of the environment when they explore the environment. After completion of every generation in evolutionary phase, genetic algorithms are applied to generate new population from best fit networks of previous generation's population. In learning phase, we have implemented two types of learning methods, namely: local online learning method and global online learning method.

i. Local Online Learning Method:

Local online learning method is a type of online learning method that is carried out using a training set. We compare the inputs obtained from the environment by the sensors and feelers with the training set inputs. When a close match is found, we use the corresponding outputs to guide mine sweepers in the environment. This local online learning method depends on the MBP algorithm, which in turn depends on the learning rule it uses. We can either use standard delta rule or we can evolve the rule. Depending on type of learning rule, local online learning method can be applied either using the delta rule or by evolving a rule. This learning quickly grasps the subtleties of the environments while the mine sweepers perform and improves their fitness considerably.

ii. Global Online Learning Method:

Global online learning method does not use any training set In this type of learning neural networks act as decision-based neural networks. We generate guiding outputs for each input from the environment using a heuristic function of one or more fitness parameters. Similar to local online learning method it is applied in two ways depending on the type of learning rule we use in the MBP. Although global learning method optimizes the performance of the mine sweepers while they explore the environment, it becomes more effective over the generations and works more closely with the genetic algorithms applied in evolutionary phase after each generation.

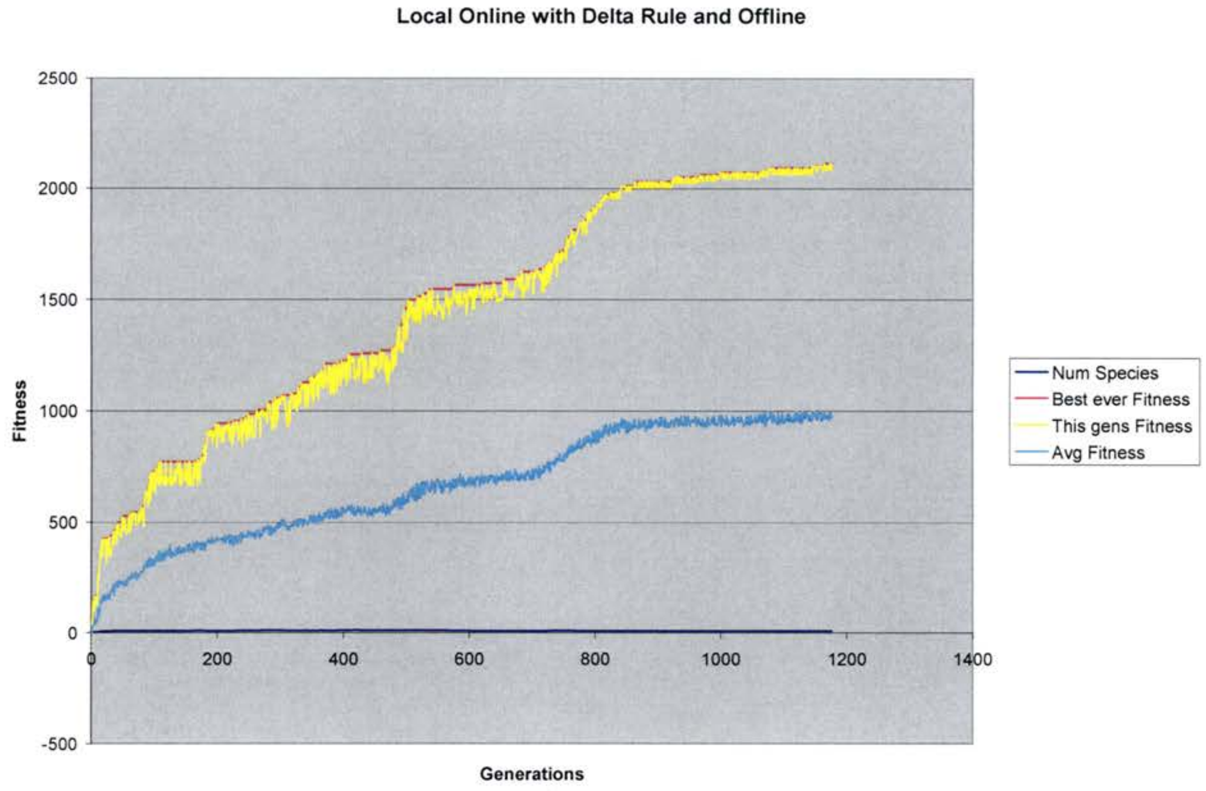


Figure 7.4 The fitness of intelligent agents when offline learning and online learning combined

The highest ever fitness is above 2000 units and the average fitness value is near 1000 units (Figure 7.4). The generation's fitness (shown with yellow line) variation from the best ever fitness (shown with pink line) is decreasing as generations increase. After 750 generations, both yellow and pink lines are close enough to indicate a perfect evolved network i.e. a network performing best in all different environments.

7.5.1 Analysis of Performance with Offline Learning and Online Learning:

The mine sweepers benefit from learning while exploring the environment by adapting to the dynamics of the drifting environment. Through online learning, the mine sweepers modify their previous knowledge to adapt to the subtleties of the new environment. But they should not loose the pre-learned knowledge in the process, hence, we decrease the amount of learning over time. Using either global or local online learning, we can observe the following:

- The mine sweepers show improved behavior from the first generation.
- The mine sweepers capture the dynamics of the environment over time.
- Over the generations mine sweepers exhibit improvements in their fitness because of the close corporation of online learning with genetic algorithm. Online learning and GAs complement each other in producing a better performing mine sweeper over generations.

In most of the experiments, the best performing ANNs have very simple architectures. The performance of all mine sweepers is improved due to the online learning capability.

7.6 Only Online Learning (Learning Phase and Evolutionary Phase)

The absence of offline learning slows down the fitness growth of mine sweepers. However, after few hundred generations mine sweepers become equally efficient to those that used combination of offline and online learning. This proves that online learning is self sufficient.

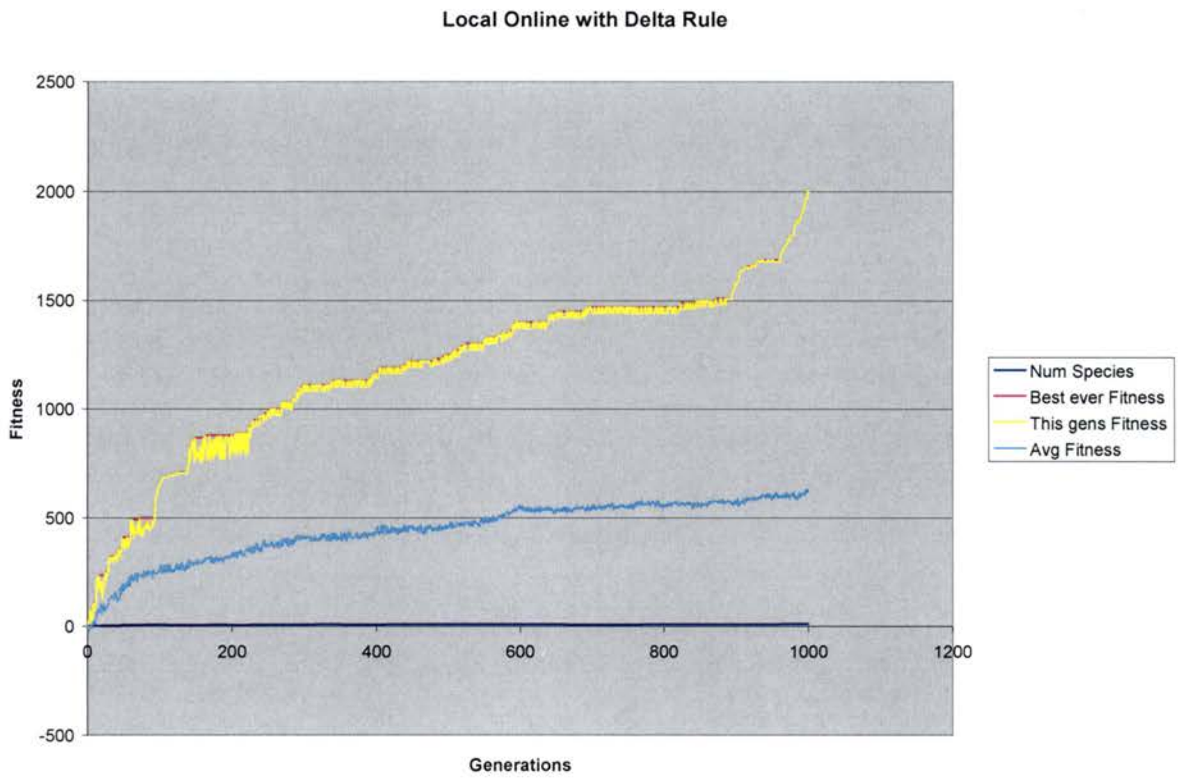


Figure 7.5 The fitness of intelligent agents Online Learning (local online with delta rule in learning phase and evolutionary phase)

Figure 7.5 depicts the performance of online learning (local online with delta rule in the learning phase and evolutionary phase with GAs) alone. For 1000 generations, with 300 time units for each generation, the best fitness ever is 2000 units and maximum average fitness value is around 600 units. The variations between yellow line (each generation's fitness) and the pink line (best ever fitness) decrease as the number of generations increase. Therefore, we can say that the algorithm has successfully evolved a network that can perform best in different environments.

7.6.1 Analysis of Performance with Online Learning Alone

In the absence of offline learning, the initial mine sweepers are too unrefined to produce best fitness. Online learning improves their behavior as early as the first generation. Even though initial fitness values may not be satisfactory, however, as generations increase mine sweepers become more sensitive to the environment with the close corporation of evolutionary phase and learning phase. After a few (200 to 300) generations, mine sweepers' performance is as good as those that included offline learning.

7.7 Analysis of Offline Learning

Figures 7.2, 7.3, 7.4, and 7.5, all exhibit the fact that offline learning significantly improves the average fitness of the population. In few generations, the best fitness values are reached. Although the offline learning considerably improves the fitness for the first few generations, after a large number of generations the effect of offline learning is not noticeable. Hence, we conclude that offline learning is necessary but not essential for the adaptability.

7.8 Comparisons between Different Approaches

The results of all our experiments indicate that the online learning is the key factor for adapting in drifting environments. We have experimented with different types of online learning such as global and local online methods individually as well as in combination with offline learning. Comparisons among these experimental results provide more insight into the online learning.

The Global online learning method does not use a training set to guide the mine sweepers in the new environment but uses a heuristic function to produce guiding outputs. Local online learning uses a small training set to guide the mine sweepers in a drifting environment. The local online learning algorithm performs slightly better than the global online learning But global online learning is faster than local online learning. Global and local online learning methods are further categorized based on the learning rule that is used to update the weights.

i. Global Online and Local Online Methods with Delta Rule (Learning Phase) and Offline Learning:

Figures 7.4 and 7.6 depict the performances of offline learning and the local online method with delta rule, and offline learning and the global online method with delta rule, respectively, in a drifting environment. Figures 7.4 and 7.6 exhibit that both experiments have their highest ever fitness values above 2000 units, but local online method's highest average fitness value is better than that of the global online method. The local online graph (Figure 7.4) is smoother than that of the global one due to its close cooperation with offline learning. Both offline learning and the local online method of learning phase require a training set. The global online method depends on heuristic function and does not benefit from offline learning. In our application global online learning optimizes only the speed of the mine sweepers and does not improve their navigation to avoid hitting the obstacles. Hence, the generation's best fitness deviates from the best ever fitness (Figure 7.6).

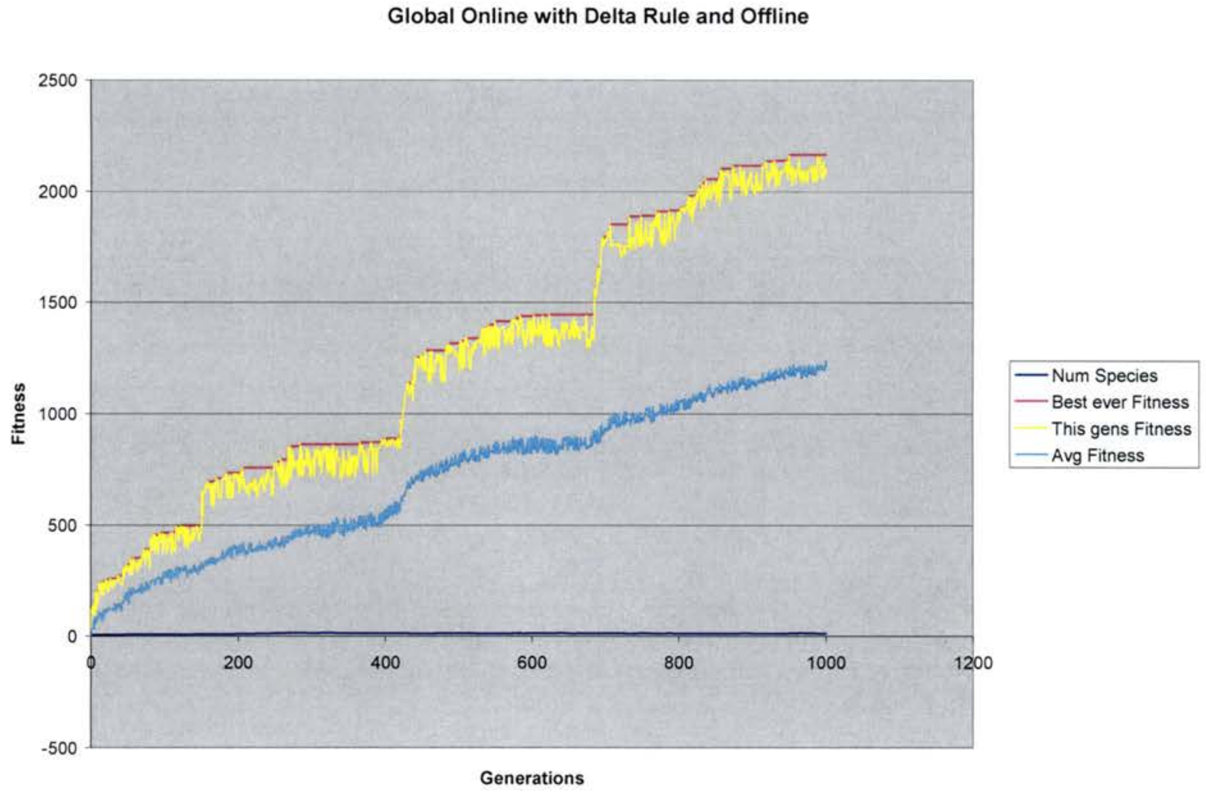


Figure 7.6 The fitness of intelligent agents when global online with delta rule of learning phase, evolutionary phase and offline learning are combined

ii. Global Online and Local Online methods with Delta Rule of Learning Phase:

Figures 7.5 and 7.7 depict the performances of local online and global online methods with delta rule, respectively. In absence of offline learning, global online method with delta rule attained the highest fitness in comparison to the local online method with delta rule. Also, it can be observed that the average fitness of global online learning is far better than the local online learning. But global online learning graph (Figure 7.7) shows more deviations of generation's fitness from the best ever fitness. In fact, local online learning graph is smooth. Absence of offline learning affected the average fitness of local online learning. But the absence of offline learning did not have considerable effect on global online learning method.

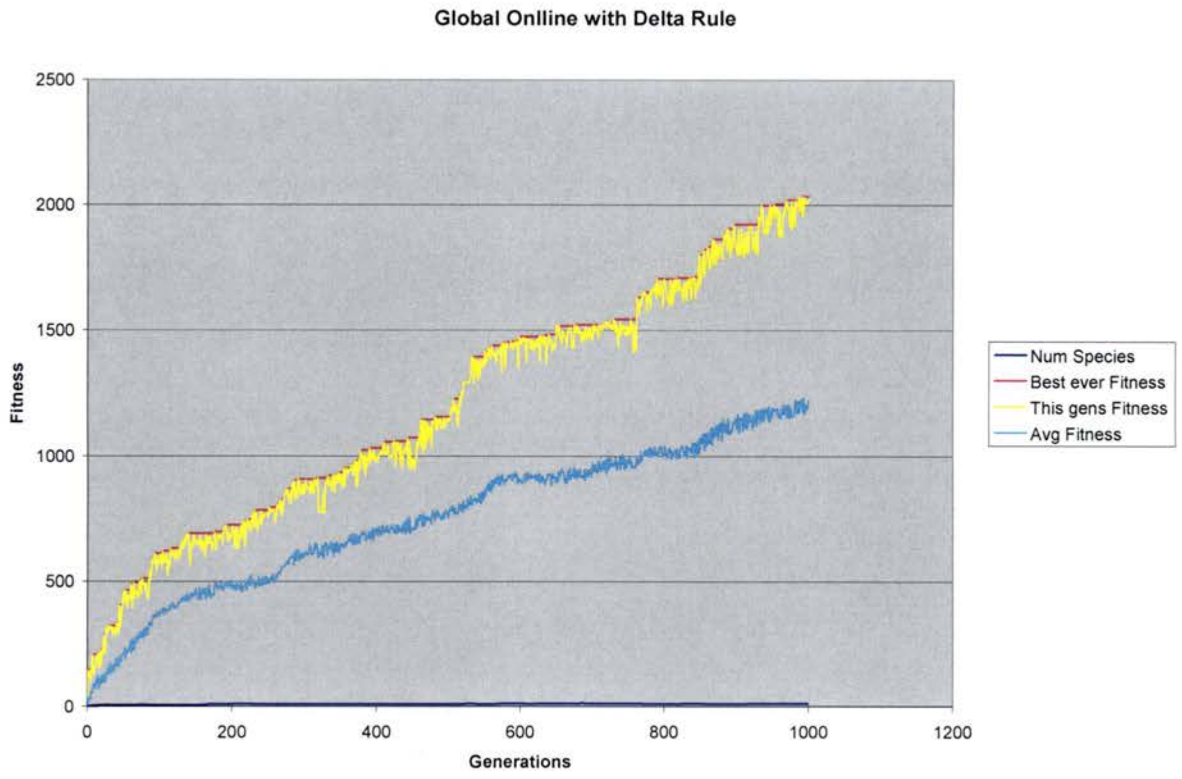


Figure 7.7 The fitness of intelligent agents when global online with delta rule of learning phase and evolutionary phase

iii. Global Online and Local Online methods with Rule Evolution and Offline learning:

Figures 7.8 and 7.9 depict the performances of global online and local online methods of learning phase with rule evolution and offline learning, respectively. Figure 7.9 shows that local online method's performance is superior to global online method's (Figure 7.8) performance. The reason for the decrease in the fitness of global online can be attributed to the randomness introduced by the generalized rule. The learning rule attempts to adapt to provide the best fitness possible. However, the global online learning directs the evolution of learning rule with respect to mine sweeper's speed only. Hence, the low fitness values are noticed for global online learning with rule evolution. But on the contrary, the same reason contributes towards better average fitness for global online learning. Although local online learning attains highest fitness, it still suffers from the deviation of generation's fitness from best ever fitness. These deviations are due to the imperfect nature of the evolving rule. If we continue the experiments for a large number of generations, the deviations at some point tend to decrease and then increase after a certain number of generations. This is due to the evolutionary process of the learning rule. When the general rule approximates the delta rule, in the process of evolution, the performance of mine sweepers is much better for both global and local online methods. Furthermore, in the rule evolution the efficiency of the guidance is observed to play an important role.

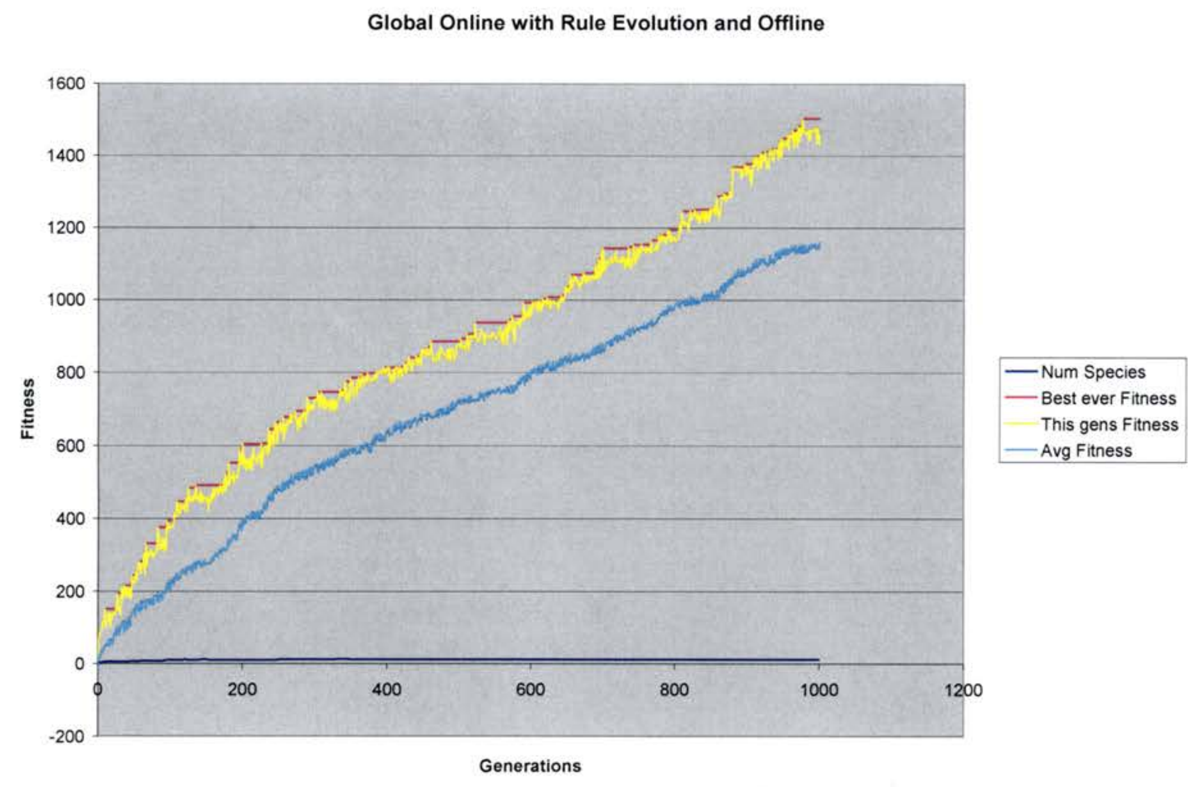


Figure 7.8 The fitness of intelligent agents when global online with rule evolution of learning phase, evolutionary phase and offline learning are combined

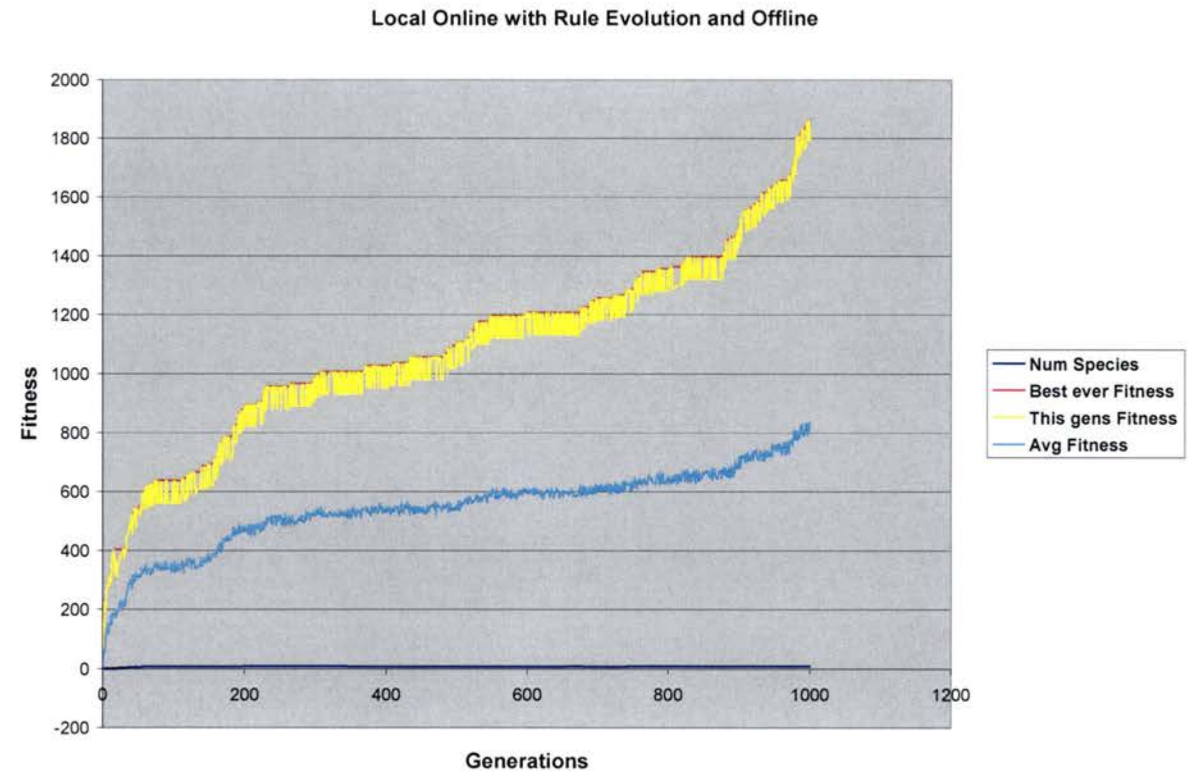


Figure 7.9 The fitness of intelligent agents when local online with rule evolution of learning phase, evolutionary phase and offline learning are combined

iv. Global Online and Local Online Methods with Rule Evolution:

Figures 7.10 and 7.11 depict the performance of global and local online methods with rule evolution, respectively. The absence of offline learning does not have much effect on global online method. Although local online learning has higher fitness than the global online learning, local online learning is greatly affected by the absence of offline learning and its generation's fitness deviates considerably from the best ever fitness. Mine sweeper population's average fitness is higher with global online learning than local online learning. Also, global online learning graph (Figure 7.10) is smoother than local online learning (Figure 7.11).

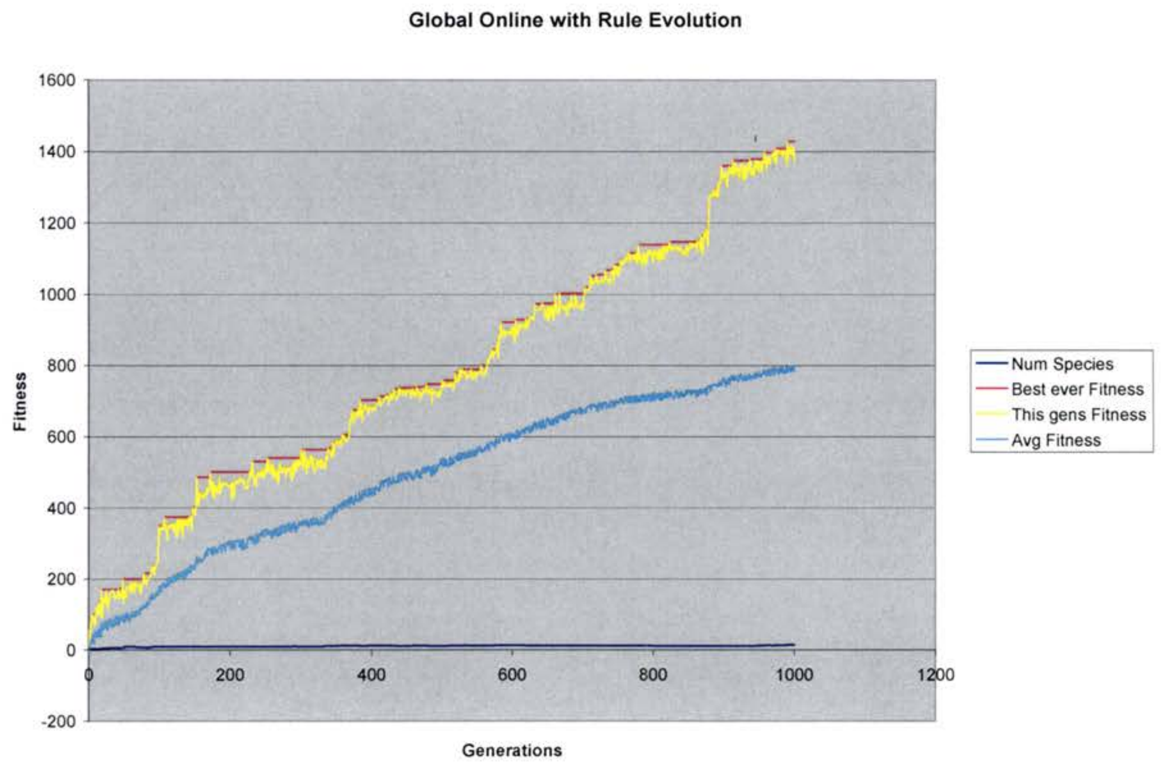
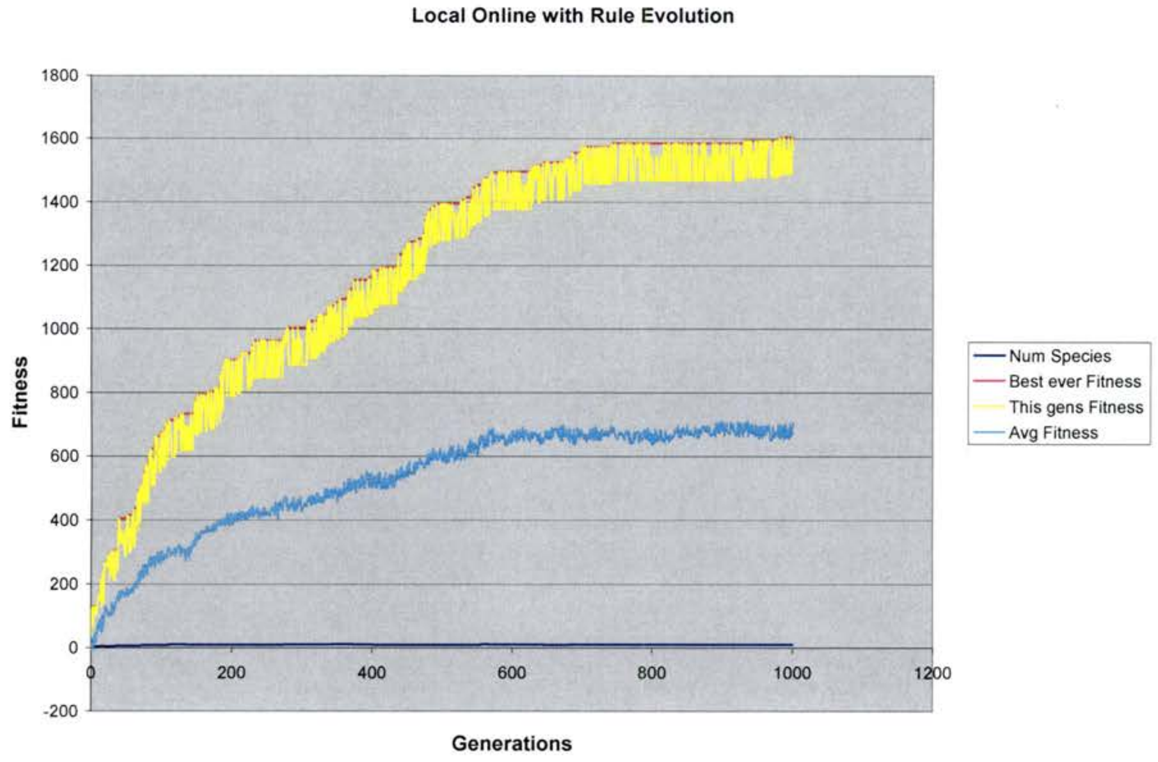


Figure 7.10 The fitness of intelligent agents when global online with rule evolution of learning phase and evolutionary phase

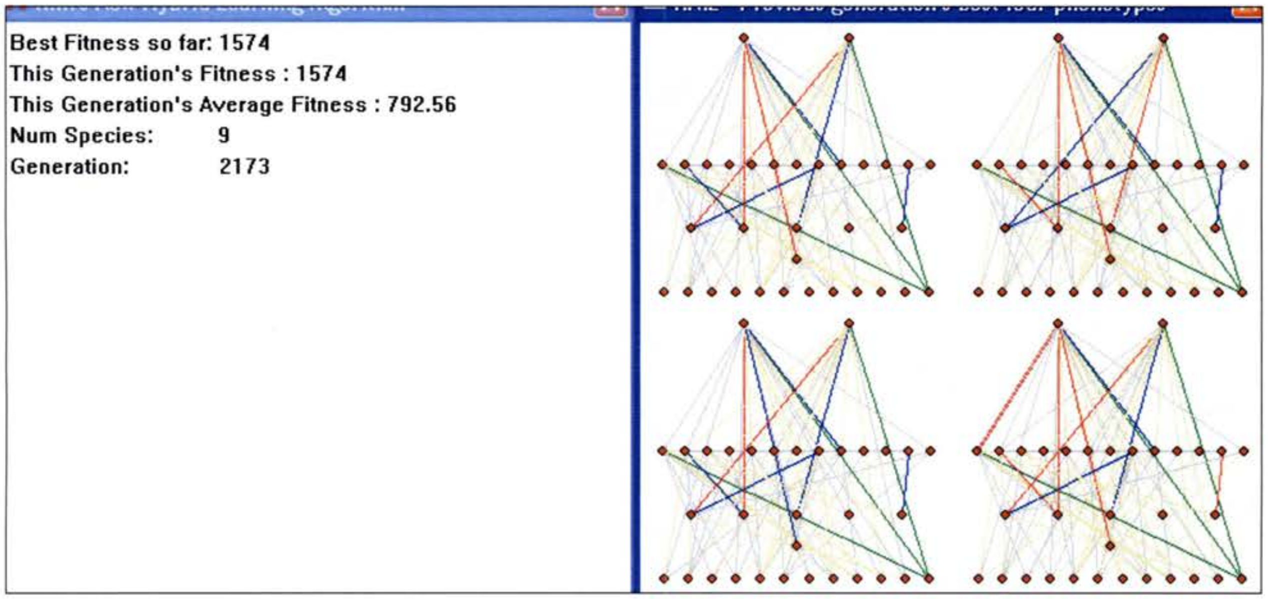


7.11 The fitness of intelligent agents when local online with rule evolution of learning phase and evolutionary phase

v. Online learning vs. GAs only:

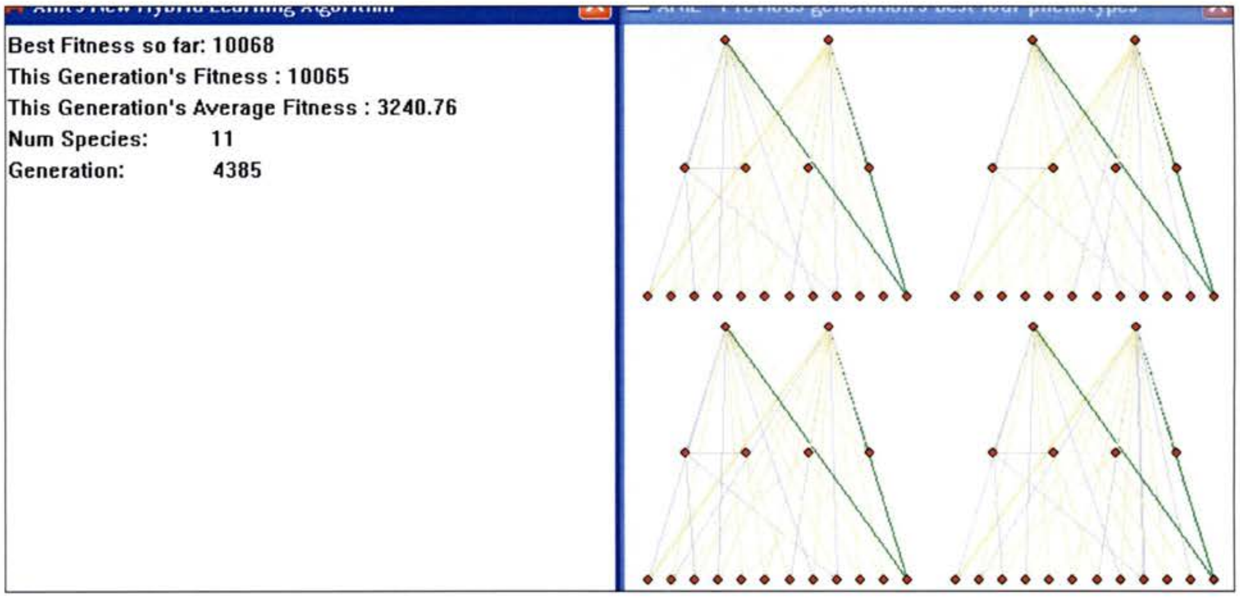
Figures 7.2, 7.5, and 7.7 depict the performances of only GAs, local online learning (learning phase) with GAs (evolutionary phase), and Global online learning (learning phase) with GAs (evolutionary phase), respectively. The graphs clearly depict the superior performance of online learning algorithms over only GAs. The evolutionary phase combined with learning phase show an improved performance in drifting environments. The deviations that are present in Figure 7.2 are due to the fact that the mine sweepers do not learn the changes in the drifting environment. The GAs are guided by learning phase in drifting environments and are not completely random.

Also, only GAs are frequently observed to generate complicated architectures. Figures 7.12 and 7.13 represent the architectures generated by only GAs, and online learning (learning phase and evolutionary phase), respectively. Whenever fitness decreases, only GAs are unable to capture the changes in the environment and therefore attempt to increase the complexity of the architectures in order to improve the fitness. But GAs (evolutionary phase) when combined with the learning phase generate architectures that are tuned to the subtleties of environment with the help of continuous learning from learning phase.



Best Fitness so far: 1574
This Generation's Fitness : 1574
This Generation's Average Fitness : 792.56
Num Species: 9
Generation: 2173

Figure 7.12 A sample run of "GAs Only" showing the complex architectures generated



Best Fitness so far: 10068
This Generation's Fitness : 10065
This Generation's Average Fitness : 3240.76
Num Species: 11
Generation: 4385

Figure 7.13 A sample run of "Only Online Learning" showing the simple architectures generated even after 4000 generations.

7.9 The Factors that influence the performance of Application

Due to the involvement of large number of variables, the application's performance depends on certain important factors. The following provides brief analysis of some important factors.

i. Number of Time Units per Generation:

The time units directly affect the performance. When there are more time units per generation, the mine sweepers can explore more area and thus increase their fitness by finding more mines. In our application, an environment always contains a fixed number of mines. Hence the fitness is directly proportional to the number of time units per generation. In an environment without mines, the fitness is directly proportional to the number of time units per generation until a saturation point and then the fitness stays constant.

ii. Number of Mine Sweepers:

The number of mine sweepers affects the diversity of the population. If there is a large number of mine sweepers, then there is a fair chance of finding an effective architecture in less time. When the population is more diverse, the genetic algorithms can generate better fit offspring.

iii. Number of Generations:

As the generations increase the mine sweepers get better and better. However, if the changes in the environment are not significant, then the MBP over-trains the networks after a large number of generations. Over-training reduces the fitness and the architectures get complicated.

iv. Number and Type of Obstacles Present:

As the environments are of equal size for all generations, number and type of the obstacles present in the environments affect the fitness by either offering more area to explore or by constraining the mine sweepers in the environment. Fitness mainly depends on the amount of area that has been explored. The type of obstacles also affect the fitness. A rectangular obstacle may occupy more space than a triangular obstacle. Hence, the number of obstacles present in the environment is inversely proportional to the fitness value in that environment.

v. Fitness criteria:

The number of generations needed to evolve a network depends on the fitness criteria. If we have simple fitness criteria then we need fewer generations to evolve a network. The number of generations required depends on the complexity of the fitness criteria. Figure 7.14 depicts the graph for simple fitness criteria area exploration.

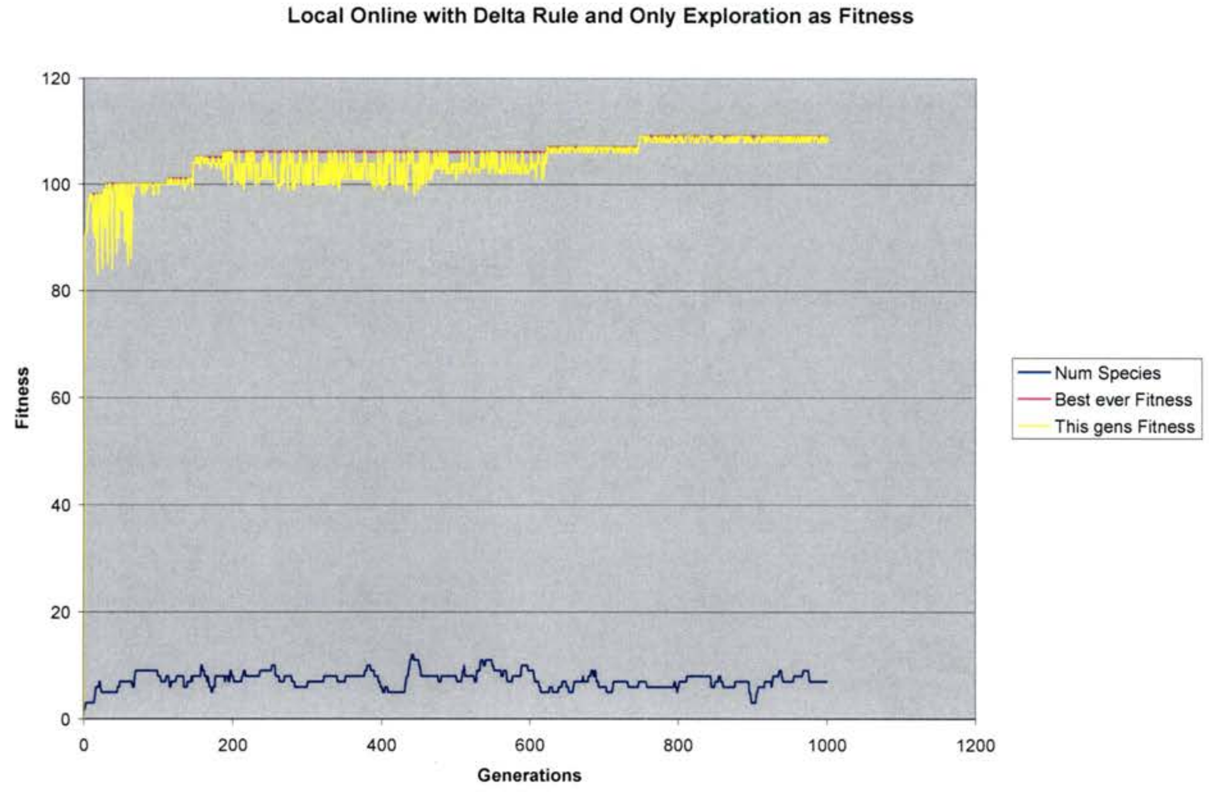


Figure 7.14 The fitness of intelligent agents with online learning and simple fitness criteria

The above graph demonstrates that within 600 generations a best performing network is evolved. If the fitness criterion is complex, a best performing network requires many more generations to evolve.

Therefore, all key factors (number of time units, number of mine sweepers, number of generations, number of obstacles and fitness criteria) have significant effect on the performance of the algorithm. As our algorithm is guided (learning phase) and random (evolutionary phase), these factors play an important role in evolving an adapting network for drifting environments.

CHAPTER 8. CONCLUSIONS AND FUTURE WORK

Overview

We have proposed a hybrid algorithm that can learn to adapt to new environments and suggested possible extensions to the present work.

8.1 Conclusion

Designing neural networks is a tedious process that requires lots of expertise and time, since a large number of variables may be involved. Evolutionary (Genetic) Algorithms have been successful in automatically generating efficient neural networks. The design of neural networks involves three different aspects namely: connection weights, architecture, and learning rules. There are no algorithms present that can evolve a neural network using simultaneous evolution of weights, architecture, and learning rules due to the complexity of the process. Our algorithm provides an efficient way to achieve the simultaneous evolution of all three aspects to generate efficient neural networks for drifting environments. Our algorithm is capable of evolving feedforward as well as recurrent neural networks and focuses on a key issue: Dynamism in the environment. With drifting environments the nature and variables of the environment change over time, emphasizing the importance of adapting to the changes in the environments. We provided a theoretically motivated hybrid adaptive learning algorithm for the drifting environments. Our algorithm design is based on the following evolutionary characteristics.

- Automatic design and generation of dynamic neural networks using evolution.
- A continuous (life-long) learning mechanism for these dynamic networks.

In addition to evolution, we used online learning mechanism to fine-tune the evolved networks for drifting environments. We observed that complete evolution with an online learning mechanism enabled the neural networks to adapt to changing environments efficiently in a short period of time. We designed and successfully used two types of online learning namely heuristic online (global online) and guided online (local online).

Our experimental results demonstrate the ability of our algorithm to evolve efficient neural networks with simple architectures in few hundreds of generations. We have evolved neural networks for mine sweepers in an environment that changes from one generation to the next and our results indicate great improvement in the mine sweepers' behavior. In addition, our results indicate that our algorithm successfully evolved simple and easy-to-fine-tune networks in very few generations.

We have used a variation of backpropagation algorithm, which can adjust the connection weights for a random and dynamic neural network without the need for re-arrangement into layers. Our modified backpropagation (MBP) can handle feedforward and recurrent networks. We successfully evolved learning rules using a simple general linear equation. Our results have shown that the evolved learning rule is as effective as the delta rule. Many real time applications do not have an input-output training set, hence we formulated the heuristic online or global online which uses a heuristic function to improve the agent's (mine sweepers) performance in the environment.

Our results show the performance of the hybrid algorithm with online learning is far superior to the performance of only evolutionary algorithms, even with complete simultaneous evolution. This underlines our basic claim that life-long learning is an important mechanism in adaptation in the drifting environments.

As our algorithm mimics human evolution we have successfully used all genetic operators in the evolutionary process. We have successfully implemented complete evolution and online learning to achieve effective design automation of neural networks with the ability to adapt to the drifting environments. Finally, our algorithm can be effectively used with artificial life as well as artificial agents in computer games.

8.2 Future Work

- We have used supervised learning approach in our algorithm. Some real time applications do not support this approach. Hence future work could be extending the algorithm to use unsupervised learning approach.

- One can extend the algorithm to use online evolution instead of online learning. It would be interesting to compare these two methods.

- Our algorithm generates dynamic networks but the input and output nodes are fixed in the algorithm. Hence it can be an effective extension if we can add or delete input and output nodes as need arises.

- Modified backpropagation can be optimized to run faster. Also, it can be an interesting phenomenon to implement incremental evolution and observe the improvements if any.

APPENDIX

```
#ifndef C2DMATRIX_H
#define C2DMATRIX_H

//-----------------------------------------------------------------------
//
//        Name: C2DMatrix.h
//
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//               Anil kumar Enumulapally  2005
//
//
// Desc: Matrix class  from the book Game AI Programming with Neural Nets
//      and Genetic Algorithms.
//
//-----------------------------------------------------------------------

#include <stdlib.h>
#include <math.h>
#include <iostream>
#include <vector>

#include "utils.h"

struct SPoint;

using namespace std;


class C2DMatrix
{
private:

 struct S2DMatrix
 {

        double _11, _12, _13;
        double _21, _22, _23;
        double _31, _32, _33;

  S2DMatrix()
        {
                _11=0; _12=0; _13=0;
                _21=0; _22=0; _23=0;
                _31=0; _32=0; _33=0;
        }

  friend ostream &operator<<(ostream& os, const S2DMatrix &rhs)
        {
                os << "\n" << rhs._11 << "  " << rhs._12 << "  " << rhs._13;

                os << "\n" << rhs._21 << "  " << rhs._22 << "  " << rhs._23;

                os << "\n" << rhs._31 << "  " << rhs._32 << "  " << rhs._33;
```

```
                        return os;
                }
        };
        S2DMatrix m_Matrix;

        //multiplies m_Matrix with mIn
        inline void  S2DMatrixMultiply(S2DMatrix &mIn);

public:

        C2DMatrix()
        {
          //initialize the matrix to an identity matrix
          Identity();
        }

        //create an identity matrix
        void Identity();

        //create a transformation matrix
        void    Translate(double x, double y);

        //create a scale matrix
        void    Scale(double xScale, double yScale);

        //create a rotation matrix
        void Rotate(double rotation);

         //applys a transformation matrix to a std::vector of points
        inline void TransformSPoints(vector<SPoint> &vPoints);

};

//multiply two matrices together
inline void C2DMatrix::S2DMatrixMultiply(S2DMatrix &mIn)
{
        S2DMatrix mat_temp;

        //first row
        mat_temp._11 = (m_Matrix._11*mIn._11) + (m_Matrix._12*mIn._21) +
(m_Matrix._13*mIn._31);
        mat_temp._12 = (m_Matrix._11*mIn._12) + (m_Matrix._12*mIn._22) +
(m_Matrix._13*mIn._32);
        mat_temp._13 = (m_Matrix._11*mIn._13) + (m_Matrix._12*mIn._23) +
(m_Matrix._13*mIn._33);

        //second
        mat_temp._21 = (m_Matrix._21*mIn._11) + (m_Matrix._22*mIn._21) +
(m_Matrix._23*mIn._31);
        mat_temp._22 = (m_Matrix._21*mIn._12) + (m_Matrix._22*mIn._22) +
(m_Matrix._23*mIn._32);
        mat_temp._23 = (m_Matrix._21*mIn._13) + (m_Matrix._22*mIn._23) +
(m_Matrix._23*mIn._33);

        //third
        mat_temp._31 = (m_Matrix._31*mIn._11) + (m_Matrix._32*mIn._21) +
(m_Matrix._33*mIn._31);
        mat_temp._32 = (m_Matrix._31*mIn._12) + (m_Matrix._32*mIn._22) +
(m_Matrix._33*mIn._32);
        mat_temp._33 = (m_Matrix._31*mIn._13) + (m_Matrix._32*mIn._23) +
(m_Matrix._33*mIn._33);

        m_Matrix = mat_temp;
}
```

```
//applies a 2D transformation matrix to a std::vector of SPoints
inline void C2DMatrix::TransformSPoints(vector<SPoint> &vPoint)
{
	for (int i=0; i<vPoint.size(); ++i)
	{
		double tempX =(m_Matrix._11*vPoint[i].x) + (m_Matrix._21*vPoint[i].y) +
(m_Matrix._31);

		double tempY = (m_Matrix._12*vPoint[i].x) + (m_Matrix._22*vPoint[i].y) +
(m_Matrix._32);

		vPoint[i].x = tempX;

		vPoint[i].y = tempY;

	}
}

#endif
```

```
#include "C2DMatrix.h"

/////////////////////////////////////////////////////////////////////
//
//        Matrix functions
//
/////////////////////////////////////////////////////////////////////
//create an identity matrix
void C2DMatrix::Identity()
{
        m_Matrix._11 = 1; m_Matrix._12 = 0; m_Matrix._13 = 0;

        m_Matrix._21 = 0; m_Matrix._22 = 1; m_Matrix._23 = 0;

        m_Matrix._31 = 0; m_Matrix._32 = 0; m_Matrix._33 = 1;

        }

//create a transformation matrix
void C2DMatrix::Translate(double x, double y)
{
        S2DMatrix mat;

        mat._11 = 1; mat._12 = 0; mat._13 = 0;

        mat._21 = 0; mat._22 = 1; mat._23 = 0;

        mat._31 = x;    mat._32 = y;    mat._33 = 1;

        //and multiply
  S2DMatrixMultiply(mat);
}

//create a scale matrix
void C2DMatrix::Scale(double xScale, double yScale)
{
        S2DMatrix mat;

        mat._11 = xScale; mat._12 = 0; mat._13 = 0;

        mat._21 = 0; mat._22 = yScale; mat._23 = 0;

        mat._31 = 0; mat._32 = 0; mat._33 = 1;

        //and multiply
  S2DMatrixMultiply(mat);
}


//create a rotation matrix
void C2DMatrix::Rotate(double rot)
{
        S2DMatrix mat;

        double Sin = sin(rot);
        double Cos = cos(rot);

        mat._11 = Cos;  mat._12 = Sin; mat._13 = 0;

        mat._21 = -Sin; mat._22 = Cos; mat._23 = 0;

        mat._31 = 0; mat._32 = 0;mat._33 = 1;

        //and multiply
  S2DMatrixMultiply(mat);
}
```

```
#ifndef CCONTROLLER_H
#define CCONTROLLER_H

//---------------------------------------------------------------------
//
//        Name: CController.h
//
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                               Anil kumar Enumulapally  2005
//
// Desc: Controller class for Anil Smart Sweepers
//
//---------------------------------------------------------------------
#include <vector>
#include <sstream>
#include <string>
#include <windows.h>

#include "CMinesweeper.h"
#include "utils.h"
#include "C2DMatrix.h"
#include "SVector2D.h"
#include "CParams.h"
#include "Cga.h"
//#include <fstream>

using namespace std;



class CController
{

private:

        //storage for the entire population of chromosomes
        Cga*              m_pPop;

        //array of sweepers
        vector<CMinesweeper> m_vecSweepers;

  //and the mines
        vector<SVector2D>          m_vecMines;

 //array of best sweepers from last generation (used for
 //display purposes when 'B' is pressed by the user)
        vector<CMinesweeper> m_vecBestSweepers;

        int                                           m_NumSweepers;

        //vertex buffer for the sweeper shapes vertices
        vector<SPoint>               m_SweeperVB;

          //vertex buffer for objects
        vector<SPoint>       m_ObjectsVB;

          //vertex buffer for the mine shape's vertices
        vector<SPoint>               m_MineVB;

        //stores the average fitness per generation
        vector<double>               m_vecAvFitness;

        //stores the best fitness per generation
        vector<double>              m_vecBestFitness;

  //best fitness ever
        double           m_dBestFitness;
        float            m_dAvgFitness;

        //pens we use for the stats
        HPEN                                m_RedPen;
        HPEN                                m_BluePen;
        HPEN                                m_GreenPen;
        //HPEN                              m_BlackBrush;
        HPEN      m_GreyPenDotted;
        HPEN      m_RedPenDotted;
        HPEN                                m_OldPen;

        HBRUSH     m_RedBrush;
        HBRUSH     m_BlueBrush;
        HBRUSH     m_BlackBrush;
  //HBRUSH     m_BlueBrush;

        //local copy of the handle to the application window
        HWND                                m_hwndMain;

  //local copy of the  handle to the info window
        HWND       m_hwndInfo;

        //toggles the speed at which the simulation runs
        bool                                m_bFastRender;

  //when set, renders the best performers from the
  //previous generaion.
    bool       m_bRenderBest;

        //cycles per generation
        int                                         m_iTicks;

        //generation counter
        int                                         m_iGenerations;

  //local copy of the client window dimensions
        int        m_cxClient, m_cyClient;

  //this is the sweeper who's memory cells are displayed
        int        m_iViewThisSweeper;

        void   PlotStats(HDC surface)const;


        void   RenderSweepers(HDC &surface, vector<CMinesweeper> &sweepers);

        void   RenderSensors (HDC &surface, vector<CMinesweeper> &sweepers);

public:

        CController(HWND hwndMain, int cxClient, int cyClient);

        ~CController();

        void              Render(HDC &surface);
        void WorldTransform1(vector<SPoint> &VBuffer, SVector2D vPos);
        void RenderMines(HDC &surface, vector<SVector2D> &mines);

  //renders the phenotypes of the four best performers from
  //the previous generation
         void   RenderNetworks(HDC &surface);
         void WriteResults();
```

```
          void GeneratePoint(int &x, int &y);
          float AvgFitness()
  {
          return m_dAvgFitness;
  }

        bool                Update();

        //-------------------------------------accessor methods
        bool                        FastRender()const{return m_bFastRender;}
        void                        FastRender(bool arg){m_bFastRender = arg;}
        void                        FastRenderToggle(){m_bFastRender = !m_bFastRender;}

        bool          RenderBest()const{return m_bRenderBest;}
        void          RenderBestToggle(){m_bRenderBest = !m_bRenderBest;}

        void          PassInfoHandle(HWND hnd){m_hwndInfo = hnd;}

        vector<double>  GetFitnessScores()const;

          void          ViewBest(int val)
          {
                if ( (val>4) || (val< 1) )
                {
                  return;
                }

                m_iViewThisSweeper = val-1;
          }
};

#endif
```

```
#include "CController.h"
//#include "file.h"
#include<fstream>
#include <string>
#include <sstream>

#include <stdio.h>


  #define PRINT   OutputDebugString



//these hold the geometry of the sweepers and the mines
const int NumSweeperVerts = 16;
vector<double> sensors;
vector<double> transsensors;
bool bBest;
int iEnv=0;
char *sEnv="";

// Initialize Mine sweeper vertices
const SPoint sweeper[NumSweeperVerts] = {SPoint(-1, -1),
                                         SPoint(-1, 1),
                                         SPoint(-0.5, 1),
                                         SPoint(-0.5, -1),

                                         SPoint(0.5, -1),
                                         SPoint(1, -1),
                                         SPoint(1, 1),
                                         SPoint(0.5, 1),

                                         SPoint(-0.5, -0.5),
                                         SPoint(0.5, -0.5),

                                         SPoint(-0.5, 0.5),
                                         SPoint(-0.25, 0.5),
                                         SPoint(-0.25, 1.75),
                                         SPoint(0.25, 1.75),
                                         SPoint(0.25, 0.5),
                                         SPoint(0.5, 0.5)};


//Initialize Mine vertices
const int NumMineVerts = 4;
const SPoint mine[NumMineVerts] = {SPoint(-1, -1),
                                   SPoint(-1, 1),
                                   SPoint(1, 1),
                                   SPoint(1, -1)};

//Initialize object vertices
int NumObjectVerts;
int NumObjectVerts5=44;
const SPoint objects[44] =                          {
                                                                         SPoint(80, 60),
                                       SPoint(200,60),
                                       SPoint(200,60),
                                       SPoint(200,100),
                                       SPoint(200,100),
                                       SPoint(160,100),
                                       SPoint(160,100),
                                       SPoint(160,200),
                                       SPoint(160,200),
                                       SPoint(80,200),
                    SPoint(80,200),
                    SPoint(80,60),

                    SPoint(250,100),
                    SPoint(300,40),
                    SPoint(300,40),
                    SPoint(350,100),
                    SPoint(350,100),
                    SPoint(250, 100),

                    SPoint(220,180),
                    SPoint(320,180),
                    SPoint(320,180),
                    SPoint(320,300),
                    SPoint(320,300),
                    SPoint(220,300),
                    SPoint(220,300),
                    SPoint(220,180),

                    SPoint(12,15),
                    SPoint(380, 15),
                    SPoint(380,15),
                    SPoint(380,360),
                    SPoint(380,360),
                    SPoint(12,360),
                    SPoint(12,360),
                                                                      SPoint(12,340),
                    SPoint(12,340),
                    SPoint(100,290),
                    SPoint(100,290),
                    SPoint(12,240),
                    SPoint(12,240),
                                                                      SPoint(12,340),
                                                                      SPoint(12,340),
                    SPoint(12,15),
                                                                      SPoint(12,15)};

const int NumObjectVerts1 = 20;
const SPoint objects1[NumObjectVerts1] = {

        SPoint(12,15),

        SPoint(380, 15),

        SPoint(380,15),

        SPoint(380,360),

        SPoint(380,360),

        SPoint(12,360),

        SPoint(12,360),

        SPoint(12,15),

        SPoint(80, 60),

        SPoint(200,60),

        SPoint(200,60),

        SPoint(200,100),

        SPoint(200,100),
          SPoint(160,100),

          SPoint(160,100),

          SPoint(160,200),

          SPoint(160,200),

          SPoint(80,200),

          SPoint(80,200),

          SPoint(80,60)
                                                              };

const int NumObjectVerts2 = 14;
const SPoint objects2[NumObjectVerts2] = {

          SPoint(12,15),

          SPoint(380, 15),

          SPoint(380,15),

          SPoint(380,360),

          SPoint(380,360),

          SPoint(12,360),

          SPoint(12,360),

          SPoint(12,15),

          SPoint(250,100),

          SPoint(300,40),

          SPoint(300,40),

          SPoint(350,100),

          SPoint(350,100),

          SPoint(250, 100)
                                                              };

const int NumObjectVerts3 = 16;
const SPoint objects3[NumObjectVerts3] = {

          SPoint(12,15),

          SPoint(380, 15),

          SPoint(380,15),

          SPoint(380,360),

          SPoint(380,360),

          SPoint(12,360),

          SPoint(12,360),

          SPoint(12,15),
            SPoint(220,180),

            SPoint(320,180),

            SPoint(320,180),

            SPoint(320,300),

            SPoint(320,300),

            SPoint(220,300),

            SPoint(220,300),

            SPoint(220,180)
                                                                                };

const int NumObjectVerts4 = 16;
const SPoint objects4[NumObjectVerts4] = {

            SPoint(12,15),

            SPoint(380, 15),

            SPoint(380,15),

            SPoint(380,360),

            SPoint(380,360),

            SPoint(12,360),

            SPoint(12,360),

            SPoint(12,15),

            SPoint(12,360),

            SPoint(12,340),

            SPoint(12,340),

            SPoint(100,290),

            SPoint(100,290),

            SPoint(12,240),

            SPoint(12,240),

            SPoint(12,15)
                                                                                };
//Create or append the results into a excel file
 fstream store1("Evolution_Rules_Local_No_Off2.xls", fstream::in | fstream::out | fstream::app);


//-------------------------------------constructor---------------------
//
//        initilaize the sweepers, their brains and the GA factory
//
//--------------------------------------------------------------------
CController::CController(HWND hwndMain,
                     int  cxClient,
                     int  cyClient): m_NumSweepers(CParams::iNumSweepers),
m_bFastRender(false),
                         m_bRenderBest(false),
                                                                                          m_iTicks(0),

m_hwndMain(hwndMain),
                         m_hwndInfo(NULL),

m_iGenerations(0),
                         m_cxClient(cxClient),
                         m_cyClient(cyClient),
                         m_iViewThisSweeper(0),

m_dAvgFitness(0)

{

        store1<<"Generation"<<"\t"<<"Num Species"<<"\t"<<"Best ever Fitness"<<"\t"<<"This gens
Fitness"<<"\t"<<"Avg Fitness"<<"\t"<<"Env_no"<<"\t"<<"Env_name"<<endl;
        if(CParams::iOfflineTraining==1)
        {
                //Perform offline stage

                vector<double> fitness;
                double dTempFitness;
                int iIterSize=10;//Number of iterations to be performed

                //Create Random Networks
                for (int i=0; i<m_NumSweepers; ++i)
                        {
                                m_vecSweepers.push_back(CMinesweeper());
                        }

                //Create the population
                m_pPop = new Cga(CParams::iNumSweepers,
                                                  CParams::iNumInputs,
                                                  CParams::iNumOutputs,
                                                  CParams::iOfflineTraining,
                                                  hwndMain,
                                                  10,10);


                 //create the phenotypes
                  vector<CNeuralNet*> pBrains = m_pPop->CreatePhenotypes();

                for(int iIter=0,i_tmp_here=0;iIter<10;iIter++,i_tmp_here++)
                {


                 fitness.clear();

                 for (int i=0; i<pBrains.size(); i++)
                 {
                        //Store the mean squared error from Modified Backpropagation
                        dTempFitness=pBrains[i]->offlineTraining(m_hwndMain);

                        // Fitness is defined as 1/error here
                        fitness.push_back(1/dTempFitness);
                 }//end of i FOR loop

                        // Perform Genetic Operations
                 vector<CNeuralNet*> pBrains = m_pPop->Epoch(fitness,1);


                }//end of iIter FOR loop
        //assign the phenotypes

         for ( i=0; i<m_NumSweepers; i++)
        {
                m_vecSweepers[i].InsertNewBrain(pBrains[i]);
                m_vecSweepers[i].SetStartingPoint(180,200);
         }
                //lets create the random mines
                for(i=0;i<50;i++)
                {
                        int tempx1,tempy1;
                        GeneratePoint(tempx1,tempy1);
                        m_vecMines.push_back(SVector2D(tempx1,tempy1));
                }

        //and the vector of sweepers which will hold the best performing sweepers
        for (i=0; i<CParams::iNumBestSweepers; ++i)
                {
                        m_vecBestSweepers.push_back(CMinesweeper());
                }


}
else
{
                //We are in online learning

                //let's create the mine sweepers
                for (int i=0; i<m_NumSweepers; ++i)
                {
                        m_vecSweepers.push_back(CMinesweeper());
                }

                //lets create the random mines
                for(i=0;i<50;i++)
                {
                        int tempx1,tempy1;
                        GeneratePoint(tempx1,tempy1);
                        m_vecMines.push_back(SVector2D(tempx1,tempy1));
                }

        //and the vector of sweepers which will hold the best performing sweepers
        for (i=0; i<CParams::iNumBestSweepers; ++i)
                {
                        m_vecBestSweepers.push_back(CMinesweeper());
                }



        m_pPop = new Cga(CParams::iNumSweepers,
                                        CParams::iNumInputs,
                                        CParams::iNumOutputs,
                                        CParams::iOfflineTraining,
                                        hwndMain,
                                        10,10);

        //create the phenotypes
         vector<CNeuralNet*> pBrains = m_pPop->CreatePhenotypes();

                //assign the phenotypes
        for (i=0; i<m_NumSweepers; i++)
        {
                m_vecSweepers[i].InsertNewBrain(pBrains[i]);
        }
}//end of offline flag ELSE
```

```
//create a pen for the graph drawing
m_BluePen       = CreatePen(PS_SOLID, 1, RGB(0, 0, 250));
m_RedPen        = CreatePen(PS_SOLID, 1, RGB(255, 100, 0));
m_GreenPen      = CreatePen(PS_SOLID, 1, RGB(0, 180, 0));
m_GreyPenDotted = CreatePen(PS_DOT, 1, RGB(100, 100, 100));
 m_RedPenDotted  = CreatePen(PS_DOT, 1, RGB(200, 0, 0));

m_OldPen        = NULL;

//and the brushes
m_BlueBrush = CreateSolidBrush(RGB(0,0,244));
m_RedBrush = CreateSolidBrush(RGB(150,0,0));
m_BlackBrush= CreateSolidBrush(RGB(0,0,0));

//fill the vertex buffers
for (int i1=0; i1<NumSweeperVerts; ++i1)
{
        m_SweeperVB.push_back(sweeper[i1]);
}

//fill mine vertex buffers
for (int i2=0;i2<NumMineVerts;++i2)
{
        m_MineVB.push_back(mine[i2]);
}

// Randomely generate the objects in the environment
int temp=RandInt(0,15);
if(temp<=2)
{
        sEnv="SquareRect";
        iEnv=1;
        m_ObjectsVB.clear();
        NumObjectVerts=NumObjectVerts1;
        for (int i2=0; i2<NumObjectVerts; ++i2)
        {

                m_ObjectsVB.push_back(objects1[i2]);
        }
}

if((temp>2) && (temp<=5))
{
        sEnv="UpTraingle";
        iEnv=2;
        NumObjectVerts=NumObjectVerts2;
        m_ObjectsVB.clear();
        for (int i=0; i<NumObjectVerts2; ++i)
        {

                m_ObjectsVB.push_back(objects2[i]);
        }
}
if((temp>5) && (temp<=8))
{
        sEnv="Rectangle";
        iEnv=3;
        NumObjectVerts=NumObjectVerts3;
        m_ObjectsVB.clear();
        for (int i=0; i<NumObjectVerts3; ++i)
        {
                m_ObjectsVB.push_back(objects3[i]);
        }
}
```

```
		if((temp>8) && (temp<=10))
		{
			sEnv="VerticalTraingle";
			iEnv=4;
			NumObjectVerts=NumObjectVerts4;
			m_ObjectsVB.clear();
			for (int i=0; i<NumObjectVerts4; ++i)
			{
				m_ObjectsVB.push_back(objects4[i]);
			}
		}

		if((temp>10) && (temp<=15))
		{
			sEnv="Full";
			iEnv=5;
			NumObjectVerts=NumObjectVerts5;
			m_ObjectsVB.clear();
			for (int i=0; i<NumObjectVerts5; ++i)
			{
				m_ObjectsVB.push_back(objects[i]);
			}
		}


}
//--------------------------------------destructor-------------------------------------
//
//-----------------------------------------------------------------------------------
CController::~CController()
{
  if (m_pPop)
  {
    delete m_pPop;
  }

		DeleteObject(m_BluePen);
		DeleteObject(m_RedPen);
		DeleteObject(m_GreenPen);
		DeleteObject(m_OldPen);
		DeleteObject(m_GreyPenDotted);
		DeleteObject(m_RedPenDotted);
		DeleteObject(m_BlueBrush);
		DeleteObject(m_RedBrush);
		DeleteObject(m_BlackBrush);
}


//-------------------------------------Update--------------------------------------
//
//	This is the main workhorse. The entire simulation is controlled from here.
//
//-----------------------------------------------------------------------------------
bool CController::Update()
{
	//run the sweepers through NUM_TICKS amount of cycles. During this loop each
	//sweepers NN is constantly updated with the appropriate information from its
	//surroundings. The output from the NN is obtained and the sweeper is moved.
	if (m_iTicks++ < CParams::iNumTicks)
	{
		for (int i=0; i<m_NumSweepers; ++i)
		{
			bBest=false;
```

```
                    //update the NN and position
                    if (!m_vecSweepers[i].Update(m_ObjectsVB,i,m_iGenerations,bBest,m_iTicks))
                    {
                            //error in processing the neural net
                            MessageBox(m_hwndMain, "Wrong amount of NN inputs!", "Error",
MB_OK);

                            return false;
                    }
                    //see if it's found a mine
                    int GrabHit = m_vecSweepers[i].CheckForMine(m_vecMines,2);

                    if (GrabHit >= 0)
                     {
                            //we have discovered a mine so increase fitness
                            m_vecSweepers[i].incrementmineval();

                            //mine found so replace the mine with another at a random
                                    //position
                            int tempx,tempy;
                            GeneratePoint(tempx,tempy);
                            m_vecMines[GrabHit] = SVector2D(tempx,tempy);
                     }
            }



  //update the NNs of the last generations best performers
  if (m_iGenerations > 0)
  {

       /*       if(m_vecBestSweepers.size()!=4)
                MessageBox(m_hwndMain, "Wrong amount of bests!", "Error", MB_OK);
                */

   for (int i=0; i<m_vecBestSweepers.size(); ++i)
   {
                 bBest=true;
     //update the NN and position
                         if
(!m_vecBestSweepers[i].Update(m_ObjectsVB,i,m_iGenerations,bBest,m_iTicks))
                     {
                              //error in processing the neural net
                              MessageBox(m_hwndMain, "Wrong amount of NN inputs!", "Error",
MB_OK);

                              return false;
                     }
   }
  }
       }

       //We have completed another generation so now we need to run the GA
       else
       {
               float dTempAvg=0;
               //first add up each sweepers fitness scores. (remember for this task
               //there are many different sorts of penalties the sweepers may incur
               //and each one has a coefficient)
               for (int swp=0; swp<m_vecSweepers.size(); ++swp)
               {
                       m_vecSweepers[swp].EndOfRunCalculations();
                       dTempAvg +=m_vecSweepers[swp].Fitness();
                    }
                    m_dAvgFitness=dTempAvg/m_vecSweepers.size();

                            // Writing results to screen
                            WriteResults();

                            //increment the generation counter
                            ++m_iGenerations;

                            //reset cycles
                            m_iTicks = 0;



                    //perform an epoch and grab the new brains
                    vector<CNeuralNet*> pBrains = m_pPop->Epoch(GetFitnessScores(), 0);


                            //insert the new  brains back into the sweepers and reset their
                    //positions
                    for (int i=0; i<m_NumSweepers; ++i)
                            {
                                    pBrains[i]-
>MutateLearningParameters(CParams::dActivationMutationRate,

        CParams::dMaxActivationPerturbation);
                                    m_vecSweepers[i].InsertNewBrain(pBrains[i]);

                                    m_vecSweepers[i].Reset();
                            }

                    //Change the objects in the environment randomely
                            int temp=RandInt(0,15);
                            if(temp<=2)
                            {
                                    sEnv="SquareRect";
                                    iEnv=1;
                                    m_ObjectsVB.clear();
                                    NumObjectVerts=NumObjectVerts1;
                                    for (int i2=0; i2<NumObjectVerts; ++i2)
                                    {

                                            m_ObjectsVB.push_back(objects1[i2]);
                                    }
                            }

                            if((temp>2) && (temp<=5))
                            {
                                    sEnv="UpTraingle";
                                    iEnv=2;
                                    NumObjectVerts=NumObjectVerts2;
                                    m_ObjectsVB.clear();
                                    for (int i=0; i<NumObjectVerts2; ++i)
                                    {

                                            m_ObjectsVB.push_back(objects2[i]);
                                    }
                            }
                            if((temp>5) && (temp<=8))
                            {
                                    sEnv="Rectangle";
                                    iEnv=3;
                                    NumObjectVerts=NumObjectVerts3;
                                    m_ObjectsVB.clear();
                                    for (int i=0; i<NumObjectVerts3; ++i)
```

```
                        {
                                m_ObjectsVB.push_back(objects3[i]);
                        }
                }

                if((temp>8) && (temp<=10))
                {
                        sEnv="VerticalTraingle";
                        iEnv=4;
                        NumObjectVerts=NumObjectVerts4;
                        m_ObjectsVB.clear();
                        for (int i=0; i<NumObjectVerts4; ++i)
                        {
                                m_ObjectsVB.push_back(objects4[i]);
                        }
                }

                if((temp>10) && (temp<=15))
                {
                        sEnv="Full";
                        iEnv=5;
                        NumObjectVerts=NumObjectVerts5;
                        m_ObjectsVB.clear();
                        for (int i=0; i<NumObjectVerts5; ++i)
                        {
                                m_ObjectsVB.push_back(objects[i]);
                        }
                }
        //}

    //grab the NNs of the best performers from the last generation
    vector<CNeuralNet*> pBestBrains = m_pPop->GetBestPhenotypesFromLastGeneration();

    //put them into our record of the best sweepers
    for ( i=0; i<m_vecBestSweepers.size(); ++i)
            {
                    m_vecBestSweepers[i].InsertNewBrain(pBestBrains[i]);

                    m_vecBestSweepers[i].Reset();
            }



      //this will call WM_PAINT which will render our scene
            InvalidateRect(m_hwndInfo, NULL, TRUE);
                    UpdateWindow(m_hwndInfo);

        }

        return true;
}

//----------------------------------- RenderNetworks ----------------------
//
//  Renders the best four phenotypes from the previous generation
//------------------------------------------------------------------------
void CController::RenderNetworks(HDC &surface)
{
  if (m_iGenerations < 1)
  {
    return;
  }

  //draw the network of the best 4 genomes. First get the dimensions of the
   //info window
   RECT rect;
```

```
        GetClientRect(m_hwndInfo, &rect);

        int       cxInfo = rect.right;
        int       cyInfo = rect.bottom;

  //now draw the 4 best networks
  m_vecBestSweepers[0].DrawNet(surface, 0, cxInfo/2, cyInfo/2, 0);
  m_vecBestSweepers[1].DrawNet(surface, cxInfo/2, cxInfo, cyInfo/2, 0);
  m_vecBestSweepers[2].DrawNet(surface, 0, cxInfo/2, cyInfo, cyInfo/2);
  m_vecBestSweepers[3].DrawNet(surface, cxInfo/2, cxInfo, cyInfo, cyInfo/2);
}

//-----------------------------------Render()-------------------------------------
//
//------------------------------------------------------------------------------
void CController::Render(HDC &surface)
{
        //do not render if running at accelerated speed
        if (!m_bFastRender)
        {
   string s = "Generation: " + itos(m_iGenerations);
          TextOut(surface, 5, 0, s.c_str(), s.size());

   //select in the blue pen
   m_OldPen = (HPEN)SelectObject(surface, m_BluePen);

   if (m_bRenderBest)
   {
    //render the best sweepers memory cells
    m_vecBestSweepers[m_iViewThisSweeper].Render(surface);

    //render the best sweepers from the last generation
    RenderSweepers(surface, m_vecBestSweepers);
                // render mines
          RenderMines(surface,m_vecMines);

    //render the best sweepers sensors
    RenderSensors(surface, m_vecBestSweepers);
   }

   else
   {
                 //render the sweepers
    RenderSweepers(surface, m_vecSweepers);

          //Enable the following line to see the sensors and feelers for all minesweepers
          //RenderSensors(surface,m_vecSweepers);

          RenderMines(surface,m_vecMines);
   }

   SelectObject(surface, m_OldPen);
        HBRUSH OldBrush=(HBRUSH)SelectObject(surface,NULL);
        SelectObject(surface,m_BlackBrush);
        POINT * p;
        int itempNum,i99,i_t=0;


   //render the objects
        //Polygon(surface,p,NumObjectVerts);
   for (int i=0; i<NumObjectVerts; i+=2)
   {
    MoveToEx(surface, m_ObjectsVB[i].x, m_ObjectsVB[i].y, NULL);

    LineTo(surface, m_ObjectsVB[i+1].x, m_ObjectsVB[i+1].y);
   }
```

```
          //SelectObject(surface,OldBrush);

          }//end if

  else
  {
   PlotStats(surface);

   RECT sr;
   sr.top    = m_cyClient-50;
   sr.bottom = m_cyClient;
   sr.left   = 0;
   sr.right  = m_cxClient;

  }

}

//------------------------- RenderSweepers -------------------------------
//
//  given a vector of sweepers this function renders them to the screen
//-----------------------------------------------------------------------
void CController::RenderSweepers(HDC &surface, vector<CMinesweeper> &sweepers)
{
 for (int i=0; i<sweepers.size(); ++i)
          {

   //if they have crashed into an obstacle draw
   if ( sweepers[i].Collided())
   {
    SelectObject(surface, m_RedPen);
   }

   else
   {
    SelectObject(surface, m_BluePen);
   }

           //grab the sweeper vertices
           vector<SPoint> sweeperVB = m_SweeperVB;

           //transform the vertex buffer
           sweepers[i].WorldTransform(sweeperVB, sweepers[i].Scale());

           //draw the sweeper left track
                   MoveToEx(surface, (int)sweeperVB[0].x, (int)sweeperVB[0].y, NULL);

                   for (int vert=1; vert<4; ++vert)
                   {
                    LineTo(surface, (int)sweeperVB[vert].x, (int)sweeperVB[vert].y);
                   }

                   LineTo(surface, (int)sweeperVB[0].x, (int)sweeperVB[0].y);

    //draw the sweeper right track
           MoveToEx(surface, (int)sweeperVB[4].x, (int)sweeperVB[4].y, NULL);

                   for (vert=5; vert<8; ++vert)
                   {
            LineTo(surface, (int)sweeperVB[vert].x, (int)sweeperVB[vert].y);
                   }

                   LineTo(surface, (int)sweeperVB[4].x, (int)sweeperVB[4].y);
```

```
                MoveToEx(surface, (int)sweeperVB[8].x, (int)sweeperVB[8].y, NULL);
                LineTo(surface, (int)sweeperVB[9].x, (int)sweeperVB[9].y);

                MoveToEx(surface, (int)sweeperVB[10].x, (int)sweeperVB[10].y, NULL);

                for (vert=11; vert<16; ++vert)
                {
                        LineTo(surface, (int)sweeperVB[vert].x, (int)sweeperVB[vert].y);
                }
        }//next sweeper
}


void CController::RenderMines(HDC &surface, vector<SVector2D> &mines)
{
                                //render the mines
                                for (int i=0; i<mines.size(); ++i)
                                {
                                        SelectObject(surface, m_GreenPen);
                                        //grab the vertices for the mine shape
                                        vector<SPoint> mineVB = m_MineVB;

                                        WorldTransform1(mineVB, mines[i]);

                                        //draw the mines
                                        MoveToEx(surface, (int)mineVB[0].x, (int)mineVB[0].y,
NULL);

                                        for (int vert=1; vert<mineVB.size(); ++vert)
                                        {
                                                LineTo(surface, (int)mineVB[vert].x,
(int)mineVB[vert].y);
                                        }

                                        LineTo(surface, (int)mineVB[0].x, (int)mineVB[0].y);

                                }

}//end of render mines


//---------------------------- RenderSensors ---------------------------
//
//  renders the sensors of a given vector of sweepers
//---------------------------------------------------------------------
void CController::RenderSensors(HDC &surface, vector<CMinesweeper> &sweepers)
{
  //render the sensors
   for (int i=0; i<sweepers.size(); ++i)
   {
     //grab each sweepers sensor data
     vector<SPoint> tranSensors    = sweepers[i].Sensors();
     vector<double> SensorReadings = sweepers[i].SensorReadings();
     vector<double> MemoryReadings = sweepers[i].MemoryReadings();

     for (int sr=0; sr<tranSensors.size(); ++sr)
     {
       if (SensorReadings[sr] > 0)
       {
         SelectObject(surface, m_RedPen);
       }

       else
       {
         SelectObject(surface, m_GreyPenDotted);
       }
```

```
        //make sure we clip the drawing of the sensors or we will get
        //unwanted artifacts appearing
        if (!((fabs(sweepers[i].Position().x - tranSensors[sr].x) >
            (CParams::dSensorRange+1))||
            (fabs(sweepers[i].Position().y - tranSensors[sr].y) >
            (CParams::dSensorRange+1))))
        {

          MoveToEx(surface,
                (int)sweepers[i].Position().x,
                (int)sweepers[i].Position().y,
                NULL);

          LineTo(surface, (int)tranSensors[sr].x, (int)tranSensors[sr].y);

          //render the cell sensors
          RECT rect;
          rect.left  = (int)tranSensors[sr].x - 2;
          rect.right = (int)tranSensors[sr].x + 2;
          rect.top   = (int)tranSensors[sr].y - 2;
          rect.bottom= (int)tranSensors[sr].y + 2;

          if (MemoryReadings[sr] < 0)
          {

            FillRect(surface, &rect, m_BlueBrush);
          }

          else
          {
            FillRect(surface, &rect, m_RedBrush);
          }

        }
      }
    }
}


//--------------------------------Write Results into the excel file---------------
void CController::WriteResults()
{
        store1<<m_iGenerations<<"\t";
        store1<<m_pPop->NumSpecies()<<"\t";
        store1<<m_pPop->BestEverFitness()<<"\t";
        store1<<m_pPop->BestGenFitness()<<"\t";
        store1<<m_dAvgFitness<<"\t";
        store1<<iEnv<<"\t";
        store1<<sEnv<<endl;


}

void CController::WorldTransform1(vector<SPoint> &VBuffer, SVector2D vPos)
{
        //create the world transformation matrix
        C2DMatrix matTransform;

        //scale
        matTransform.Scale(2, 2);

        //translate
        matTransform.Translate(vPos.x, vPos.y);

        //transform the ships vertices
        matTransform.TransformSPoints(VBuffer);
}

//--------------------------PlotStats------------------------------------
//
// Given a surface to draw on this function displays some simple stats
//-----------------------------------------------------------------------
void CController::PlotStats(HDC surface)const
{

   string s = "Generation:             " + itos(m_iGenerations);
          TextOut(surface, 5, 85, s.c_str(), s.size());
          //store1<<m_iGenerations<<"\t";

   s = "Num Species:         " + itos(m_pPop->NumSpecies());
          TextOut(surface, 5, 65, s.c_str(), s.size());
          //store1<<m_pPop->NumSpecies()<<"\t";


   s = "Best Fitness so far: " + ftos(m_pPop->BestEverFitness());
   TextOut(surface, 5, 5, s.c_str(), s.size());
        //store1<<m_pPop->BestEverFitness()<<endl;

        s = "This Generation's Fitness : " + ftos(m_pPop->BestGenFitness());
        TextOut(surface, 5, 25, s.c_str(), s.size());

        s = "This Generation's Average Fitness : " + ftos(m_dAvgFitness);
        TextOut(surface, 5, 45, s.c_str(), s.size());

}


//------------------------------- GetFitnessScores -----------------------
//
// returns a std::vector containing the genomes fitness scores
//-----------------------------------------------------------------------
vector<double> CController::GetFitnessScores()const
{
  vector<double> scores;

  for (int i=0; i<m_vecSweepers.size(); ++i)
  {
    scores.push_back(m_vecSweepers[i].Fitness());
  }
  return scores;
}



//---------------------------------------Generate Point---------------------
//Generates a random point which is not covered by objects and also not out of boundaries
//--------------------------------------------------------------------------
void CController::GeneratePoint(int &x, int &y)
{
        bool bPointFlag=true;
        x=180;
        y=200;
        for(;;)
        {

         bPointFlag=true;
         x=RandFloat() * m_cxClient;
         y=RandFloat() * m_cyClient;

                if((x<=18)||(y<=21)||(x>=370)||(y>=350))
              //        if((x<=12)||(y<=15)||(x>=380)||(y>=360))
```

```
            {
             bPointFlag=false;
            }
    //for 1
            if((x>=80)&&(x<=200))
            {
             if((y<=200)&&(y>=60))
                    bPointFlag=false;

            }

    //for 2
            if((x>=250)&&(x<=350))
                    {
                     if((y<=100)&&(y>=40))
                            bPointFlag=false;

                    }

    //for 3
            if((x>=220)&&(x<=320))
            {
             if((y<=300)&&(y>=180))
                    bPointFlag=false;

            }
    //for 4
            if((x>=12)&&(x<=100))
            {
             if((y<=340)&&(y>=15))
                    bPointFlag=false;

            }

    if(bPointFlag==true) break;
    }//end of for

}
```

```
#ifndef CGA_H
#define  CGA_H

//-----------------------------------------------------------------------
//
//        Name: Cga.h
//
//  Authors:
//  Created by  Mat Buckland 2002
//  Modified by Anil kumar Enumulapally  2004
//                                     Anil kumar Enumulapally  2005
//
//  Desc: The evolutionary algorithm class  used in the implementation
//
//-----------------------------------------------------------------------
#include <vector>
#include <windows.h>

#include "phenotype.h"
#include "genotype.h"
#include "CSpecies.h"
#include "CParams.h"

using namespace std;



//-----------------------------------------------------------------------
//
//  this structure is used in the creation of a network depth lookup
//  table.
//-----------------------------------------------------------------------
struct SplitDepth
{
  double val;

  int    depth;

  SplitDepth(double v, int d):val(v), depth(d){}
};


//-----------------------------------------------------------------------
//
//-----------------------------------------------------------------------
class Cga
{

private:

          //current population
          vector<CGenome>   m_vecGenomes;

          //keep a record of the last generations best genomes. (used to render
 //the best performers to the display if the user presses the 'B' key)
          vector<CGenome>   m_vecBestGenomes;

          //all the species
 vector<CSpecies>  m_vecSpecies;

          //to keep track of innovations
 CInnovation*      m_pInnovation;
```

```
        //current generation
int           m_iGeneration;

        int           m_iNextGenomeID;

int           m_iNextSpeciesID;

        int           m_iPopSize;

//adjusted fitness scores
double          m_dTotFitAdj,
              m_dAvFitAdj;

//index into the genomes for the fittest genome
int           m_iFittestGenome;

double          m_dBestEverFitness;
double          m_dGenBestFitness ;

//local copy of app handle
HWND            m_hwnd;

//local copies of client area
int           cxClient, cyClient;

//this holds the precalculated split depths. They are used
//to calculate a neurons x/y position for rendering and also
//for calculating the flush depth of the network when a
//phenotype is working in 'snapshot' mode.
vector<SplitDepth> vecSplits;


        //used in Crossover
        void    AddNeuronID(int nodeID, vector<int> &vec);

//this function resets some values ready for the next epoch, kills off
//all the phenotypes and any poorly performing species.
void    ResetAndKill();

//separates each individual into its respective species by calculating
//a compatibility score with every other member of the population and
//niching accordingly. The function then adjusts the fitness scores of
//each individual by species age and by sharing and also determines
//how many offspring each individual should spawn.
void    SpeciateAndCalculateSpawnLevels();

//adjusts the fitness scores depending on the number sharing the
//species and the age of the species.
void    AdjustSpeciesFitnesses();

CGenome Crossover(CGenome& mum, CGenome& dad);

CGenome TournamentSelection(const int NumComparisons);

//searches the lookup table for the dSplitY value of each node in the
//genome and returns the depth of the network based on this figure
int     CalculateNetDepth(const CGenome &gen);

//sorts the population into descending fitness, keeps a record of the
//best n genomes and updates any fitness statistics accordingly
void    SortAndRecord();

//a recursive function used to calculate a lookup table of split
//depths
vector<SplitDepth> Split(double low, double high, int depth);
```

```
public:

        Cga(int             size,
         int                inputs,
         int                outputs,
                              int      offline,
         HWND  hwnd,
         int   cx,
         int   cy);

        ~Cga();

        vector<CNeuralNet*>       Epoch(const vector<double> &FitnessScores, int iOffline);

        //iterates through the population and creates the phenotypes
  vector<CNeuralNet*>  CreatePhenotypes();

  //keeps a record of the n best genomes from the last population.
  //(used to display the best genomes)
  void              StoreBestGenomes();

  //renders the best performing species statistics and a visual aid
  //showing the distribution.
  void              RenderSpeciesInfo(HDC &surface, RECT db);

  //returns a vector of the n best phenotypes from the previous generation
  vector<CNeuralNet*>  GetBestPhenotypesFromLastGeneration();


  //---------------------------------------------------accessor methods
  int    NumSpecies()const{return m_vecSpecies.size();}

  double  BestEverFitness()const{return m_dBestEverFitness;}
  double  BestGenFitness()const{return m_dGenBestFitness;}

};


#endif
```

```
#include "Cga.h"


//-----------------------------------------------------------------------
//        this constructor creates a base genome from supplied values and creates
//        a population of 'size' similar (same topology, varying weights) genomes
//-----------------------------------------------------------------------
Cga::Cga(int  size,
         int  inputs,
         int  outputs,
                     int  offline,
         HWND hwnd,
         int  cx,
         int  cy):  m_iPopSize(size),
                m_iGeneration(0),
                m_pInnovation(NULL),
                m_iNextGenomeID(0),
                m_iNextSpeciesID(0),
                m_iFittestGenome(0),
                m_dBestEverFitness(0),
                                          m_dGenBestFitness(0),
                m_dTotFitAdj(0),
                m_dAvFitAdj(0),
                m_hwnd(hwnd),
                cxClient(cx),
                cyClient(cy)
{
         //create the population of genomes
         for (int i=0; i<m_iPopSize; ++i)
         {
                        m_vecGenomes.push_back(CGenome(m_iNextGenomeID++, inputs, outputs));
         }

         //create the innovation list. First create a minimal genome
         CGenome genome(1, inputs, outputs);

         //create the innovations
  m_pInnovation = new CInnovation(genome.Genes(), genome.Neurons());

  //If this constructor is called in offline learning we add hidden neurons
  //to few minimal genomes
  if(offline==1)
  {
           //create minimal genome with hidden neurons
           for (int i_temp=0; i_temp<m_iPopSize/2; ++i_temp)
            {
                     double j_temp=RandFloat();
                     if(j_temp<=RandFloat())
                     {
m_vecGenomes[i_temp].AddNeuron(0.9,*m_pInnovation,CParams::iNumTrysToFindOldLink);
                     }
            }
  }//end of OFFLINE-IF

  //create the network depth lookup table
  vecSplits = Split(0, 1, 0);
}

//------------------------------------ dtor -----------------------------
//
//-----------------------------------------------------------------------
```

```
Cga::~Cga()
{
 if (m_pInnovation)
 {
  delete m_pInnovation;

  m_pInnovation = NULL;
 }
}

//------------------------------CreatePhenotypes------------------------
//
//        cycles through all the members of the population and creates their
// phenotypes. Returns a vector containing pointers to the new phenotypes
//-----------------------------------------------------------------------
vector<CNeuralNet*> Cga::CreatePhenotypes()
{
        vector<CNeuralNet*> networks;

 for (int i=0; i<m_iPopSize; i++)
        {
                //calculate max network depth
                int depth = CalculateNetDepth(m_vecGenomes[i]);

                //create new phenotype
                CNeuralNet* net = m_vecGenomes[i].CreatePhenotype(depth);

  networks.push_back(net);
        }

        return networks;
}

//------------------------- CalculateNetDepth --------------------------
//
// searches the lookup table for the dSplitY value of each node in the
// genome and returns the depth of the network based on this figure
//-----------------------------------------------------------------------
int Cga::CalculateNetDepth(const CGenome &gen)
{
 int MaxSoFar = 0;

 for (int nd=0; nd<gen.NumNeurons(); ++nd)
 {
  for (int i=0; i<vecSplits.size(); ++i)
  {

   if ((gen.SplitY(nd) == vecSplits[i].val) &&
      (vecSplits[i].depth > MaxSoFar))
   {
    MaxSoFar = vecSplits[i].depth;
   }
  }
 }
 return MaxSoFar + 2;
}


//----------------------------------AddNeuronID---------------------------
//
//        just checks to see if a node ID has already been added to a vector of
// nodes. If not  then the new ID  gets added. Used in Crossover.
//-----------------------------------------------------------------------
void Cga::AddNeuronID(const int nodeID, vector<int> &vec)
{
        for (int i=0; i<vec.size(); i++)
        {
                if (vec[i] == nodeID)
                {
                        //already added
                        return;
                }
        }

        vec.push_back(nodeID);

        return;
}

//------------------------------------ Epoch ---------------------------
//
// This function performs one epoch of the genetic algorithm and returns
// a vector of pointers to the new phenotypes
//-----------------------------------------------------------------------
vector<CNeuralNet*> Cga::Epoch(const vector<double> &FitnessScores, int iofflne)
{
        bool bOffline_flag=false;


 //reset appropriate values and kill off the existing phenotypes and
 //any poorly performing species
 ResetAndKill();

 //update the genomes with the fitnesses scored in the last run
 for (int gen=0; gen<m_vecGenomes.size(); ++gen)
 {
   m_vecGenomes[gen].SetFitness(FitnessScores[gen]);
 }

 //sort genomes and keep a record of the best performers
 SortAndRecord();

 //separate the population into species of similar topology, adjust
 //fitnesses and calculate spawn levels
 SpeciateAndCalculateSpawnLevels();

 //this will hold the new population of genomes
 vector<CGenome> NewPop;

 //request the offspring from each species. The number of children to
 //spawn is a double which we need to convert to an int.
 int NumSpawnedSoFar = 0;

 CGenome baby;

 //now to iterate through each species selecting offspring to be mated and
 //mutated
 for (int spc=0; spc<m_vecSpecies.size(); ++spc)
 {
   //because of the number to spawn from each species is a double
   //rounded up or down to an integer it is possible to get an overflow
   //of genomes spawned. This statement just makes sure that doesn't
   //happen
   if (NumSpawnedSoFar < CParams::iNumSweepers)
   {
     //this is the amount of offspring this species is required to
     // spawn. Rounded simply rounds the double up or down.
     int NumToSpawn = Rounded(m_vecSpecies[spc].NumToSpawn());

     bool bChosenBestYet = false;

     while (NumToSpawn--)
```

```
{
 //first grab the best performing genome from this species and transfer
 //to the new population without mutation. This provides per species
 //elitism
 if (!bChosenBestYet)
 {
  baby = m_vecSpecies[spc].Leader();

  bChosenBestYet = true;
 }

 else
 {
  //if the number of individuals in this species is only one
  //then we can only perform mutation
  if (m_vecSpecies[spc].NumMembers() == 1)
  {
   //spawn a child
   baby = m_vecSpecies[spc].Spawn();
  }

  //if greater than one we can use the crossover operator
  else
  {
   //spawn1
   CGenome g1 = m_vecSpecies[spc].Spawn();

   if (RandFloat() < CParams::dCrossoverRate)
   {

    //spawn2, make sure it's not the same as g1
    CGenome g2 = m_vecSpecies[spc].Spawn();

    //number of attempts at finding a different genome
    int NumAttempts = 10;

    while ( (g1.ID() == g2.ID()) && (NumAttempts--) )
    {
     g2 = m_vecSpecies[spc].Spawn();
    }

    if (g1.ID() != g2.ID())
    {
                        if(iofflne==1)
                        {
                                if(bOffline_flag==false)
                                        {
                                                        bOffline_flag=true;
                                                        baby = Crossover(g1, g2);
                                        }
                        }//end of offline stage check
                        else //it is online
                                baby = Crossover(g1, g2);
    }
   }//end of crossover constant check

   else
   {
    baby = g1;
   }
  }


 ++m_iNextGenomeID;

 baby.SetID(m_iNextGenomeID);
```

```
                if (ioffline==1)
                {
                                //now we have a spawned child lets mutate it! First there is the
                        //chance a neuron may be added
                        if (baby.NumNeurons() < CParams::iMaxPermittedNeurons)
                        {
                                if(bOffline_flag==false)
                                {
                                                                        bOffline_flag=true;

        baby.AddNeuron(CParams::dChanceAddNode,
                                                                        *m_pInnovation,

        CParams::iNumTrysToFindOldLink);
                                }//end of offline flag check;
                        }//emd of add neuron mutation


                                //now there's the chance a link may be added
                                if(bOffline_flag==false)
                                {
                                                                        bOffline_flag=true;
                                                                baby.AddLink(CParams::dChanceAddLink,
                                                                  CParams::dChanceAddRecurrentLink,
                                                                  *m_pInnovation,
                                                                  CParams::iNumTrysToFindLoopedLink,
                                                                  CParams::iNumAddLinkAttempts);
                                }

                                //mutate the weights
                                if(bOffline_flag==false)
                                {
                                                                        bOffline_flag=true;

        baby.MutateWeights(CParams::dMutationRate,

CParams::dProbabilityWeightReplaced,

CParams::dMaxWeightPerturbation);
                                }

                                if(bOffline_flag==false)
                                {
                                                                        bOffline_flag=true;

        baby.MutateActivationResponse(CParams::dActivationMutationRate,

        CParams::dMaxActivationPerturbation);
                                }



                }//end of ioffline==1


                else  //for Online
                {

                        //now we have a spawned child lets mutate it! First there is the
                        //chance a neuron may be added
                        if (baby.NumNeurons() < CParams::iMaxPermittedNeurons)
                        {
                                baby.AddNeuron(CParams::dChanceAddNode,
                                                                *m_pInnovation,
                                                CParams::iNumTrysToFindOldLink);
                          }

                          //now there's the chance a link may be added
                          baby.AddLink(CParams::dChanceAddLink,
                                       CParams::dChanceAddRecurrentLink,
                                       *m_pInnovation,
                                       CParams::iNumTrysToFindLoopedLink,
                                       CParams::iNumAddLinkAttempts);

                          //mutate the weights
                          baby.MutateWeights(CParams::dMutationRate,

CParams::dProbabilityWeightReplaced,

CParams::dMaxWeightPerturbation);

                          baby.MutateActivationResponse(CParams::dActivationMutationRate,

    CParams::dMaxActivationPerturbation);
             }//end of else offline==1;
   }


   //sort the babies genes by their innovation numbers
   baby.SortGenes();

   //add to new pop
   NewPop.push_back(baby);

   ++NumSpawnedSoFar;

   if (NumSpawnedSoFar == CParams::iNumSweepers)
   {
     NumToSpawn = 0;
   }

  }//end while

 }//end if

}//next species


//if there is an underflow due to the rounding error and the amount
//of offspring falls short of the population size additional children
//need to be created and added to the new population. This is achieved
//simply, by using tournament selection over the entire population.
if (NumSpawnedSoFar < CParams::iNumSweepers)
{

  //calculate amount of additional children required
  int Rqd = CParams::iNumSweepers - NumSpawnedSoFar;

  //grab them
  while (Rqd--)
  {
    NewPop.push_back(TournamentSelection(m_iPopSize/5));
  }
}

//replace the current population with the new one
m_vecGenomes = NewPop;

//create the new phenotypes
vector<CNeuralNet*> new_phenotypes;
```

```
  for (gen=0; gen<m_vecGenomes.size(); ++gen)
  {
    //calculate max network depth
    int depth = CalculateNetDepth(m_vecGenomes[gen]);

    CNeuralNet* phenotype = m_vecGenomes[gen].CreatePhenotype(depth);

    new_phenotypes.push_back(phenotype);
  }

  //increase generation counter
  ++m_iGeneration;

  return new_phenotypes;
}

//--------------------------- SortAndRecord-------------------------------
//
// sorts the population into descending fitness, keeps a record of the
// best n genomes and updates any fitness statistics accordingly
//------------------------------------------------------------------------
void Cga::SortAndRecord()
{
  //sort the genomes according to their unadjusted (no fitness sharing)
  //fitnesses
  sort(m_vecGenomes.begin(), m_vecGenomes.end());
  m_dGenBestFitness=m_vecGenomes[0].Fitness();

  //is the best genome this generation the best ever?
  if (m_vecGenomes[0].Fitness() > m_dBestEverFitness)
  {
    m_dBestEverFitness = m_vecGenomes[0].Fitness();
  }

  //keep a record of the n best genomes
  StoreBestGenomes();
}

//---------------------------- StoreBestGenomes -------------------------
//
// used to keep a record of the previous populations best genomes so that
// they can be displayed if required.
//------------------------------------------------------------------------
void Cga::StoreBestGenomes()
{
  //clear old record
  m_vecBestGenomes.clear();

  for (int gen=0; gen<CParams::iNumBestSweepers; ++gen)
  {
    m_vecBestGenomes.push_back(m_vecGenomes[gen]);
  }
}

//----------------- GetBestPhenotypesFromLastGeneration ------------------
//
// returns a std::vector of the n best phenotypes from the previous
// generation
//------------------------------------------------------------------------
vector<CNeuralNet*> Cga::GetBestPhenotypesFromLastGeneration()
{
  vector<CNeuralNet*> brains;

  for (int gen=0; gen<m_vecBestGenomes.size(); ++gen)
  {
    //calculate max network depth
    int depth = CalculateNetDepth(m_vecBestGenomes[gen]);

    brains.push_back(m_vecBestGenomes[gen].CreatePhenotype(depth));
  }

  return brains;
}

//--------------------------- AdjustSpecies -----------------------------
//
//  this functions simply iterates through each species and calls
//  AdjustFitness for each species
//------------------------------------------------------------------------
void Cga::AdjustSpeciesFitnesses()
{
  for (int sp=0; sp<m_vecSpecies.size(); ++sp)
  {
    m_vecSpecies[sp].AdjustFitnesses();
  }
}

//----------------- SpeciateAndCalculateSpawnLevels ---------------------
//
//  separates each individual into its respective species by calculating
//  a compatibility score with every other member of the population and
//  niching accordingly. The function then adjusts the fitness scores of
//  each individual by species age and by sharing and also determines
//  how many offspring each individual should spawn.
//------------------------------------------------------------------------
void Cga::SpeciateAndCalculateSpawnLevels()
{
  bool bAdded = false;

  //iterate through each genome and speciate
  for (int gen=0; gen<m_vecGenomes.size(); ++gen)
  {
    //calculate its compatibility score with each species leader. If
    //compatible add to species. If not, create a new species
    for (int spc=0; spc<m_vecSpecies.size(); ++spc)
    {
      double compatibility = m_vecGenomes[gen].GetCompatibilityScore(m_vecSpecies[spc].Leader());

      //if this individual is similar to this species add to species
      if (compatibility <= CParams::dCompatibilityThreshold)
      {
        m_vecSpecies[spc].AddMember(m_vecGenomes[gen]);

        //let the genome know which species it's in
        m_vecGenomes[gen].SetSpecies(m_vecSpecies[spc].ID());

        bAdded = true;

        break;
      }
    }

    if (!bAdded)
    {
      //we have not found a compatible species so let's create a new one
      m_vecSpecies.push_back(CSpecies(m_vecGenomes[gen], m_iNextSpeciesID++));
    }

    bAdded = false;
  }
  //now all the genomes have been assigned a species the fitness scores
  //need to be adjusted to take into account sharing and species age.
  AdjustSpeciesFitnesses();

  //calculate new adjusted total & average fitness for the population
  for (gen=0; gen<m_vecGenomes.size(); ++gen)
  {
    m_dTotFitAdj += m_vecGenomes[gen].GetAdjFitness();
  }

  m_dAvFitAdj = m_dTotFitAdj/m_vecGenomes.size();

  //calculate how many offspring each member of the population
  //should spawn
  for (gen=0; gen<m_vecGenomes.size(); ++gen)
  {
    double ToSpawn = m_vecGenomes[gen].GetAdjFitness() / m_dAvFitAdj;

    m_vecGenomes[gen].SetAmountToSpawn(ToSpawn);
  }

  //iterate through all the species and calculate how many offspring
  //each species should spawn
  for (int spc=0; spc<m_vecSpecies.size(); ++spc)
  {
   m_vecSpecies[spc].CalculateSpawnAmount();
  }
}


//-------------------------- TournamentSelection -----------------------
//
//----------------------------------------------------------------------
CGenome Cga::TournamentSelection(const int NumComparisons)
{
  double BestFitnessSoFar = 0;

  int ChosenOne = 0;

  //Select NumComparisons members from the population at random testing
  //against the best found so far
  for (int i=0; i<NumComparisons; ++i)
  {
   int ThisTry = RandInt(0, m_vecGenomes.size()-1);

   if (m_vecGenomes[ThisTry].Fitness() > BestFitnessSoFar)
   {
    ChosenOne = ThisTry;

    BestFitnessSoFar = m_vecGenomes[ThisTry].Fitness();
   }
  }

  //return the champion
  return m_vecGenomes[ChosenOne];
}

//---------------------------------Crossover---------------------------
//
//----------------------------------------------------------------------
CGenome Cga::Crossover(CGenome& mum, CGenome& dad)
{

 //helps make the code clearer
 enum parent_type{MUM, DAD,};
```

```
//first, calculate the genome we will using the disjoint/excess
//genes from. This is the fittest genome.
parent_type best;

//if they are of equal fitness use the shorter (because we want to keep
//the networks as small as possible)
if (mum.Fitness() == dad.Fitness())
{
 //if they are of equal fitness and length just choose one at
 //random
 if (mum.NumGenes() == dad.NumGenes())
 {
  best = (parent_type)RandInt(0, 1);
 }

 else
 {
  if (mum.NumGenes() < dad.NumGenes())
  {
   best = MUM;
  }

  else
  {
   best = DAD;
  }
 }
}

else
{
 if (mum.Fitness() > dad.Fitness())
 {
  best = MUM;
 }

 else
 {
  best = DAD;
 }
}

//these vectors will hold the offspring's nodes and genes
vector<SNeuronGene>  BabyNeurons;
vector<SLinkGene>    BabyGenes;

//temporary vector to store all added node IDs
vector<int> vecNeurons;

//create iterators so we can step through each parents genes and set
//them to the first gene of each parent
vector<SLinkGene>::iterator curMum = mum.StartOfGenes();
vector<SLinkGene>::iterator curDad = dad.StartOfGenes();

//this will hold a copy of the gene we wish to add at each step
SLinkGene SelectedGene;

//step through each parents genes until we reach the end of both
while (!((curMum == mum.EndOfGenes()) && (curDad == dad.EndOfGenes())))
{

 //the end of mum's genes have been reached
 if ((curMum == mum.EndOfGenes())&&(curDad != dad.EndOfGenes()))
 {
  //if dad is fittest
  if (best == DAD)
  {
   //add dads genes
   SelectedGene = *curDad;
  }

  //move onto dad's next gene
  ++curDad;
 }

 //the end of dads's genes have been reached
 else if ( (curDad == dad.EndOfGenes()) && (curMum != mum.EndOfGenes()))
 {
  //if mum is fittest
  if (best == MUM)
  {
   //add mums genes
   SelectedGene = *curMum;
  }

  //move onto mum's next gene
  ++curMum;
 }

 //if mums innovation number is less than dads
 else if (curMum->InnovationID < curDad->InnovationID)
 {
  //if mum is fittest add gene
  if (best == MUM)
  {
   SelectedGene = *curMum;
  }

  //move onto mum's next gene
  ++curMum;
 }

 //if dads innovation number is less than mums
 else if (curDad->InnovationID < curMum->InnovationID)
 {
  //if dad is fittest add gene
  if (best == DAD)
  {
   SelectedGene = *curDad;
  }

  //move onto dad's next gene
  ++curDad;
 }

 //if innovation numbers are the same
 else if (curDad->InnovationID == curMum->InnovationID)
 {
  //grab a gene from either parent
  if (RandFloat() < 0.5f)
  {
   SelectedGene = *curMum;
  }

  else
  {
   SelectedGene = *curDad;
  }

  //move onto next gene of each parent
  ++curMum;
  ++curDad;
```

```
    }

    //add the selected gene if not already added
    if (BabyGenes.size() == 0)
    {
      BabyGenes.push_back(SelectedGene);
    }

    else
    {
      if (BabyGenes[BabyGenes.size()-1].InnovationID !=
          SelectedGene.InnovationID)
      {
        BabyGenes.push_back(SelectedGene);
      }
    }

    //Check if we already have the nodes referred to in SelectedGene.
    //If not, they need to be added.
    AddNeuronID(SelectedGene.FromNeuron, vecNeurons);
    AddNeuronID(SelectedGene.ToNeuron, vecNeurons);

  }//end while

  //now create the required nodes. First sort them into order
  sort(vecNeurons.begin(), vecNeurons.end());

  for (int i=0; i<vecNeurons.size(); i++)
  {
    BabyNeurons.push_back(m_pInnovation->CreateNeuronFromID(vecNeurons[i]));
  }

  //finally, create the genome
  CGenome babyGenome(m_iNextGenomeID++,
                     BabyNeurons,
                     BabyGenes,
                     mum.NumInputs(),
                     mum.NumOutputs());

  return babyGenome;
}


//---------------------------- ResetAndKill ------------------------------
//
// This function resets some values ready for the next epoch, kills off
// all the phenotypes and any poorly performing species.
//------------------------------------------------------------------------
void Cga::ResetAndKill()
{
  m_dTotFitAdj = 0;
  m_dAvFitAdj  = 0;

  //purge the species
  vector<CSpecies>::iterator curSp = m_vecSpecies.begin();

  while (curSp != m_vecSpecies.end())
  {
    curSp->Purge();

    //kill off species if not improving and if not the species which contains
    //the best genome found so far
    if ( (curSp->GensNoImprovement() > CParams::iNumGensAllowedNoImprovement) &&
         (curSp->BestFitness() < m_dBestEverFitness) )
    {
     curSp = m_vecSpecies.erase(curSp);
    --curSp;
   }

   ++curSp;
  }

  //we can also delete the phenotypes
  for (int gen=0; gen<m_vecGenomes.size(); ++gen)
  {
    m_vecGenomes[gen].DeletePhenotype();
  }
}

//------------------------------- Split ----------------------------------
//
//  this function is used to create a lookup table that is used to
//  calculate the depth of the network.
//------------------------------------------------------------------------
vector<SplitDepth> Cga::Split(double low, double high, int depth)
{
  static vector<SplitDepth> vecSplits;

  double span = high-low;

  vecSplits.push_back(SplitDepth(low + span/2, depth+1));

  if (depth > 6)
  {
    return vecSplits;
  }

  else
  {
    Split(low, low+span/2, depth+1);
    Split(low+span/2, high, depth+1);

    return vecSplits;
  }
}

//--------------------------- RenderSpeciesInfo -------------------------
//
//  Used to display species information on the screen
//------------------------------------------------------------------------
void Cga::RenderSpeciesInfo(HDC &surface, RECT db)
{
  if (m_vecSpecies.size() < 1) return;

  int numColours = 255/m_vecSpecies.size();

  double SlicePerSweeper = (double)(db.right-db.left)/(double)(CParams::iNumSweepers-1);

  double left = db.left;

  //now draw a different colored rectangle for each species
  for (int spc=0; spc<m_vecSpecies.size(); ++spc)
  {

     //choose a brush to draw with
     HBRUSH PieBrush = CreateSolidBrush(RGB(numColours*spc, 255, 255 - numColours*spc));

     HBRUSH OldBrush = (HBRUSH)SelectObject(surface, PieBrush);

     if (spc == m_vecSpecies.size()-1)
     {
       Rectangle(surface,
            left,
            db.top,
            db.right,
            db.bottom);
    }

    else
    {
      Rectangle(surface,
            left,
            db.top,
            left+SlicePerSweeper*m_vecSpecies[spc].NumMembers(),
            db.bottom);
    }

    left += SlicePerSweeper * m_vecSpecies[spc].NumMembers();

    SelectObject(surface, OldBrush);
    DeleteObject(PieBrush);

    //display best performing species stats in the same color as displayed
    //in the distribution bar
    if ( m_vecSpecies[spc].BestFitness() == m_dBestEverFitness)
    {
      string s = "Best Species ID: " + itos(m_vecSpecies[spc].ID());
      TextOut(surface, 5, db.top - 80, s.c_str(), s.size());

      s = "Species Age: " + itos(m_vecSpecies[spc].Age());
      TextOut(surface, 5, db.top - 60, s.c_str(), s.size());

      s = "Generations  no improvement: " + itos(m_vecSpecies[spc].GensNoImprovement());
      TextOut(surface, 5, db.top - 40, s.c_str(), s.size());
    }
  }

  string s = "Species Distribution Bar";
        TextOut(surface, 5, db.top - 20, s.c_str(), s.size());
}
```

```
#ifndef CINNOVATION_H
#define CINNOVATION_H
//-----------------------------------------------------------------------
//
// Name: CInnovation.h
//
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                                Anil kumar Enumulapally  2005
//
//        Desc: class to handle genome innovations used in the implementation.
//
//
//-----------------------------------------------------------------------
#include <vector>
#include <algorithm>

#include "utils.h"
#include "genotype.h"
#include "phenotype.h"

using namespace std;

struct SLinkGene;



//--------------------Innovation related structs/classes----------------
//
//-----------------------------------------------------------------------
enum innov_type
{
        new_neuron,
        new_link
};

//-----------------------------------------------------------------------
//
// structure defining an innovation
//-----------------------------------------------------------------------
struct   SInnovation
{
 //new neuron or new link?
 innov_type  InnovationType;

 int         InnovationID;

 int         NeuronIn;
 int         NeuronOut;

 int         NeuronID;

 neuron_type NeuronType;

 //if the innovation is a neuron we need to keep a record
 //of its position in the tree (for display purposes)
 double      dSplitY,
          dSplitX;

        SInnovation(int        in,
          int      out,
          innov_type t,
          int      inov_id):NeuronIn(in),
                         NeuronOut(out),
                         InnovationType(t),
                         InnovationID(inov_id),
                         NeuronID(0),
                         dSplitX(0),
                         dSplitY(0),
                         NeuronType(none)
        {}

  SInnovation(SNeuronGene neuron,
          int       innov_id,
          int       neuron_id):InnovationID(innov_id),
                           NeuronID(neuron_id),
                           dSplitX(neuron.dSplitX),
                           dSplitY(neuron.dSplitY),
                           NeuronType(neuron.NeuronType),
                           NeuronIn(-1),
                           NeuronOut(-1)
  {}

  SInnovation(int       in,
          int      out,
          innov_type  t,
          int      inov_id,
          neuron_type type,
          double     x,
          double     y):NeuronIn(in),
                    NeuronOut(out),
                    InnovationType(t),
                    InnovationID(inov_id),
                    NeuronID(0),
                    NeuronType(type),
                    dSplitX(x),
                    dSplitY(y)
        {}
};

//-----------------------------------------------------------------------
//
// CInnovation class used to keep track of all innovations created during
// the populations evolution
//-----------------------------------------------------------------------
class CInnovation
{

private:

  vector<SInnovation> m_vecInnovs;

  int              m_NextNeuronID;

  int              m_NextInnovationNum;


public:

  CInnovation(vector<SLinkGene>   start_genes,
          vector<SNeuronGene> start_neurons);

  //checks to see if this innovation has already occurred. If it has it
  //returns the innovation ID. If not it returns a negative value.
  int   CheckInnovation(int in, int out, innov_type type);
```

```
      //creates a new innovation and returns its ID
  int   CreateNewInnovation(int in, int out, innov_type type);

  //as above but includes adding x/y position of new neuron
  int   CreateNewInnovation(int         from,
                            int         to,
                            innov_type  InnovType,
                            neuron_type NeuronType,
                            double      x,
                            double      y);

  //creates a BasicNeuron from the given neuron ID
  SNeuronGene CreateNeuronFromID(int id);


  //-----------------------------------------------accessor methods
  int   GetNeuronID(int inv)const{return m_vecInnovs[inv].NeuronID;}

  void  Flush(){m_vecInnovs.clear(); return;}

  int   NextNumber(int num = 0)
  {
    m_NextInnovationNum += num;

    return m_NextInnovationNum;
  }
};


#endif
```

```
#include "CInnovation.h"

//---------------------------------- ctor --------------------------------
//
// given a series of start genes and start neurons this ctor adds
// all the appropriate innovations.
//-----------------------------------------------------------------------
CInnovation::CInnovation(vector<SLinkGene>   start_genes,
                  vector<SNeuronGene> start_neurons)
{
         m_NextNeuronID            = 0;
         m_NextInnovationNum = 0;

         //add the neurons
  for (int nd=0; nd<start_neurons.size(); ++nd)
  {
    m_vecInnovs.push_back(SInnovation(start_neurons[nd],
                           m_NextInnovationNum++,
                           m_NextNeuronID++));
  }

  //add the links
  for (int cGen = 0; cGen<start_genes.size(); ++cGen)
         {
                  SInnovation NewInnov(start_genes[cGen].FromNeuron,
                  start_genes[cGen].ToNeuron,
                  new_link,
                  m_NextInnovationNum);

                  m_vecInnovs.push_back(NewInnov);

    ++m_NextInnovationNum;

         }
}

//--------------------------CheckInnovation------------------------------
//
//       checks to see if this innovation has already occurred. If it has it
//       returns the innovation ID. If not it returns a negative value.
//-----------------------------------------------------------------------
int CInnovation::CheckInnovation(int in, int out, innov_type type)
{
 //iterate through the innovations looking for a match on all
 //three parameters
         for (int inv=0; inv<m_vecInnovs.size(); ++inv)
         {
   if ( (m_vecInnovs[inv].NeuronIn == in)   &&
       (m_vecInnovs[inv].NeuronOut == out) &&
       (m_vecInnovs[inv].InnovationType == type))
                  {
                           //found a match so assign this innovation number to id
                           return m_vecInnovs[inv].InnovationID;
                  }
         }
         //if no match, return a negative value
 return -1;
}

//-------------------------CreateNewInnovation--------------------------
```

```
//
//        creates a new innovation and returns its ID
//------------------------------------------------------------------------
int CInnovation::CreateNewInnovation(int in, int out, innov_type type)
{
        SInnovation new_innov(in, out, type, m_NextInnovationNum);

        if (type == new_neuron)
        {
                new_innov.NeuronID = m_NextNeuronID;

                ++m_NextNeuronID;
        }

  m_vecInnovs.push_back(new_innov);

  ++m_NextInnovationNum;

        return (m_NextNeuronID-1);
}

//------------------------------------------------------------------------
//
// as above but includes adding x/y position of new neuron
//------------------------------------------------------------------------
int CInnovation::CreateNewInnovation(int          from,
                                     int          to,
                                     innov_type   InnovType,
                                     neuron_type  NeuronType,
                                     double       x,
                                     double       y)
{
        SInnovation new_innov(from, to, InnovType, m_NextInnovationNum, NeuronType, x, y);

        if (InnovType == new_neuron)
        {
                new_innov.NeuronID = m_NextNeuronID;

                ++m_NextNeuronID;
        }

  m_vecInnovs.push_back(new_innov);

  ++m_NextInnovationNum;

        return (m_NextNeuronID-1);

}

//------------------------------- CreateNeuronFromID -----------------------
//
// given a neuron ID this function returns a clone of that neuron
//------------------------------------------------------------------------
SNeuronGene CInnovation::CreateNeuronFromID(int NeuronID)
{
  SNeuronGene temp(hidden,0,0,0);

  for (int inv=0; inv<m_vecInnovs.size(); ++inv)
  {
    if (m_vecInnovs[inv].NeuronID == NeuronID)
    {
      temp.NeuronType = m_vecInnovs[inv].NeuronType;
      temp.iID      = m_vecInnovs[inv].NeuronID;
      temp.dSplitY  = m_vecInnovs[inv].dSplitY;
      temp.dSplitX  = m_vecInnovs[inv].dSplitX;
      return temp;
    }
  }

  return temp;
}

#ifndef CMAPPER_H
#define CMAPPER_H

#include <vector>
#include <windows.h>

#include "utils.h"
#include "Cparams.h"

using namespace std;

//------------------------------------------------------------------------
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                                  Anil kumar Enumulapally  2005
//
// structure to define a 'cell'. A cell is a RECT in space and keeps
// a track of how many ticks the bot has spent at the cell.
//------------------------------------------------------------------------
struct SCell
{
  int iTicksSpentHere;

  //the coordinates which describe the cell's position
  RECT Cell;

  SCell(int xmin, int xmax, int ymin, int ymax):iTicksSpentHere(0)
  {
    Cell.left   = xmin;
    Cell.right  = xmax;
    Cell.top    = ymin;
    Cell.bottom = ymax;
  }

  void Update()
  {
    ++iTicksSpentHere;
  }

  void Reset()
  {
    iTicksSpentHere = 0;
  }
};


//------------------------------------------------------------------------
//
// This mapper class holds information about a 2d vector of cells
//------------------------------------------------------------------------
class CMapper
{
private:

  //the 2d vector of memory cells
  vector<vector<SCell> > m_2DvecCells;

  int     m_NumCellsX;
  int     m_NumCellsY;
  int     m_iTotalCells;

  //the dimensions of each cell
  double  m_dCellSize;

public:

  CMapper():m_NumCellsX(0),
          m_NumCellsY(0),
          m_iTotalCells(0)
  {}

  //this must be called after an instance of this class has been
  //created. This sets up all the cell coordinates.
  void    Init(int MaxRangeX, int MaxRangeY);

  //this method is called each frame and updates the time spent
  //at the cell at this position
  void    Update(double xPos, double yPos);

  //returns how many ticks have been spent at this cell position
  int     TicksLingered(double xPos, double yPos) const;

  //returns the total number of cells visited
  int     NumCellsVisited()const;

  //returns if the cell at the given position has been visited or
  //not
  bool    BeenVisited(double xPos, double yPos) const;

  //This method renders any visited cells in shades of red. The
  //darker the red, the more time has been spent at that cell
  void    Render(HDC surface);

  void    Reset();

  int     NumCells()const{return m_iTotalCells;}
};


#endif
```

```
#include "CMapper.h"


//-------------------------- Init -------------------------------------
//
// This method needs to be called before you can use the instance.
//---------------------------------------------------------------------
void CMapper::Init(int MaxRangeX, int MaxRangeY)
{
 //if already initialized return
 if(m_NumCellsX) return;

 m_dCellSize = CParams::dCellSize;

 //first calculate how many segments we will require
 m_NumCellsX = (int)(MaxRangeX/m_dCellSize)+1;
 m_NumCellsY = (int)(MaxRangeY/m_dCellSize)+1;

 //create the 2d vector of blank segments
 for (int x=0; x<m_NumCellsX; ++x)
 {
  vector<SCell> temp;

  for (int y=0; y<m_NumCellsY; ++y)
  {
   temp.push_back(SCell(x*m_dCellSize, (x+1)*m_dCellSize, y*m_dCellSize, (y+1)*m_dCellSize));
  }

  m_2DvecCells.push_back(temp);
 }

 m_iTotalCells = m_NumCellsX * m_NumCellsY;

}
//------------------------------------------------------------
void CMapper::Update(double xPos, double yPos)
{
  //check to make sure positions are within range
 if ( (xPos < 0) || (xPos > CParams::WindowWidth) ||
     (yPos < 0) || (yPos > CParams::WindowHeight) )
 {
  return;
 }

 int cellX = (int)(xPos / m_dCellSize );
 int cellY = (int)(yPos / m_dCellSize );

 m_2DvecCells[cellX][cellY].Update();

 return;
}

//-------------------------------------------------------------
int CMapper::TicksLingered(double xPos, double yPos)const
{
  //bounds check the incoming values
 if ( (xPos > CParams::WindowWidth) || (xPos < 0) ||
    (yPos > CParams::WindowHeight)|| (yPos < 0))
  {
   return 999;
  }

  int cellX = (int)(xPos / m_dCellSize);
  int cellY = (int)(yPos / m_dCellSize);

  return m_2DvecCells[cellX][cellY].iTicksSpentHere;
 }

 //------------------------ Visited -------------------------------------
 //
 //----------------------------------------------------------------------
 bool CMapper::BeenVisited(double xPos, double yPos)const
 {
  int cellX = (int)(xPos / m_dCellSize);
  int cellY = (int)(yPos / m_dCellSize);

  if (m_2DvecCells[cellX][cellY].iTicksSpentHere > 0)
  {
   return true;
  }

  else
  {
   return false;
  }
 }
 //-------------------------------- Render ------------------------------
 //
 // renders the visited cells. The color gets darker the more frequently
 // the cell has been visited.
 //----------------------------------------------------------------------
 void CMapper::Render(HDC surface)
 {

  for (int x=0; x<m_NumCellsX; ++x)
  {
   for (int y=0; y<m_NumCellsY; ++y)
   {
    if (m_2DvecCells[x][y].iTicksSpentHere > 0)
    {
     int shading = 2 * m_2DvecCells[x][y].iTicksSpentHere;

     if (shading >220)
     {
      shading = 220;
     }


     HBRUSH lightbrush = CreateSolidBrush(RGB(240,220-shading,220-shading));

     FillRect(surface, &m_2DvecCells[x][y].Cell, lightbrush);

     DeleteObject(lightbrush);
    }
   }
  }
 }

 //---------------------------------- Reset -----------------------------
 void CMapper::Reset()
 {

  for (int x=0; x<m_NumCellsX; ++x)
 {
  for (int y=0; y<m_NumCellsY; ++y)
  {
   m_2DvecCells[x][y].Reset();
  }
 }
}


int CMapper::NumCellsVisited() const
{
 int total = 0;

 for (int x=0; x<m_NumCellsX; ++x)
 {
  for (int y=0; y<m_NumCellsY; ++y)
  {
   if (m_2DvecCells[x][y].iTicksSpentHere > 0)
   {
    ++total;
   }
  }
 }

 return total;
}
```

```
#ifndef CMINESWEEPER_H
#define CMINESWEEPER_H

//-----------------------------------------------------------------------
//
//        Name: CMineSweeper.h
//
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                              Anil kumar Enumulapally  2005
//
// Desc: Class to create a minesweeper object
//
//-----------------------------------------------------------------------
#include <vector>
#include <math.h>

#include "phenotype.h"
#include "utils.h"
#include "C2DMatrix.h"
#include "SVector2D.h"
#include "CParams.h"
#include "collision.h"
#include "CMapper.h"



using namespace std;


class CMinesweeper
{

private:

 CNeuralNet*  m_pItsBrain;

 //its memory
 CMapper     m_MemoryMap;

 //its position in the world
        SVector2D                   m_vPosition;

        //direction sweeper is facing
        SVector2D                   m_vLookAt;

        //how much it is rotated from its starting position
        double                      m_dRotation;

        double                      m_dSpeed;

        //to store output from the ANN
        double                      m_lTrack, m_rTrack;

        //the sweepers fitness score.
        double                      m_dFitness;
        //the scale of the sweeper when drawn
        double                  m_dScale;

        //no of mines found;
        int  m_iMines;

/*      //the inputs from sensors
        double       m_dSensors[5];
        //the inputs from feelers
        double       m_dFeelers[5];
 */



 //fitness parameters
        int m_iCollisions;
        int m_iRotval;
        int m_iSpeedval;


 //to store end vertices of sensor segments
 vector<SPoint>  m_Sensors;
 vector<SPoint>  m_tranSensors;

 //this keeps a record of how far down the sensor segment
 //a 'hit' has occurred.
 vector<double>  m_vecdSensors;

 //the end points of the sensors check their coordinate
 //cell to see how many times the sweeper has visited it.
 vector<double>  m_vecFeelers;

 //if a collision has been detected this flag is set
 bool          m_bCollided;


 void    CreateSensors(vector<SPoint> &sensors,
                   int         NumSensors,
                   double       range);

 int     CheckForHit(vector<SVector2D> &objects, double size);



 void    TestSensors(vector<SPoint> &objects);

 void    TestRange();


public:


        CMinesweeper();

        //updates the ANN with information from the sweepers enviroment
        bool                    Update(vector<SPoint> &objects, int ival,int igen,bool bBest, int
iTicks);

        //used to transform the sweepers vertices prior to rendering
 void                   WorldTransform(vector<SPoint> &sweeper, double scale);

        void                    Reset();
        //checks to see if the minesweeper has 'collected' a mine
 int    CheckForMine(vector<SVector2D> &mines, double size);
 void incrementmineval();
void SetStartingPoint(int x, int y)
{
        m_vPosition = SVector2D(x, y);

}


void      EndOfRunCalculations();

void      RenderStats(HDC surface);

void      Render(HDC surface);

void      DrawNet(HDC &surface, int cxLeft, int cxRight, int cyTop, int cyBot)
{
  m_pItsBrain->DrawNet(surface, cxLeft, cxRight, cyTop, cyBot);
}


        //-------------------accessor functions
        SVector2D               Position()const{return m_vPosition;}

double          Rotation()const{return m_dRotation;}

        float                               Fitness()const{return m_dFitness;}

double          Scale()const{return m_dScale;}

vector<SPoint>&   Sensors(){return m_tranSensors;}

vector<double>&   SensorReadings(){return m_vecdSensors;}

bool            Collided()const{return m_bCollided;}

vector<double>    MemoryReadings(){return m_vecFeelers;}

int             NumCellsVisited(){return m_MemoryMap.NumCellsVisited();}

void            InsertNewBrain(CNeuralNet* brain){m_pItsBrain = brain;}
CNeuralNet*       getBrain(){ return(m_pItsBrain);}

};

#endif
```

```
#include "CMinesweeper.h"
#include "file.h"
int jval=0;
//-----------------------------------constructor-------------------------
//
//-----------------------------------------------------------------------
CMinesweeper::CMinesweeper():
                      m_dRotation(0),
                      m_lTrack(0),
                      m_rTrack(0),
                      m_dFitness(0),
                                                m_dScale(CParams::iSweeperScale),
                      m_bCollided(false),
                                                m_iCollisions(0),
                                                m_iSpeedval(0),
                                                m_iMines(0),
                                                m_iRotval(0)



{
 //create a static start position
 m_vPosition = SVector2D(180, 200);

  //create the sensors
 CreateSensors(m_Sensors, CParams::iNumSensors, CParams::dSensorRange);

 //initialize its memory
 m_MemoryMap.Init(CParams::WindowWidth,
              CParams::WindowHeight);

}


//-------------------------------- CreateSensors ------------------------
//
//        This function returns a vector of points which make up the segments of
// the sweepers sensors.
//-----------------------------------------------------------------------
void CMinesweeper::CreateSensors(vector<SPoint> &sensors,
                        int          NumSensors,
                        double        range)
{
 //make sure vector of sensors is empty before proceeding
 sensors.clear();

 double SegmentAngle = CParams::dPi / (NumSensors-1);

        //going clockwise from 90deg left of position calculate the fan of
        //points radiating out (not including the origin)
        for (int i=0; i<CParams::iNumSensors; i++)
        {
                //calculate vertex position
                SPoint point;

                point.x =          -sin(i * SegmentAngle - CParams::dHalfPi) * range;
                point.y = cos(i * SegmentAngle - CParams::dHalfPi) * range;

                sensors.push_back(point);

        }//next segment

}
//-----------------------------Reset()-----------------------------------
//
//      Resets the sweepers position, fitness and rotation
//
//------------------------------------------------------------------------
void CMinesweeper::Reset()
{
        //reset the sweepers positions
        m_vPosition = SVector2D(180, 200);

        //and the fitness
        m_dFitness = 0;

  //and the rotation
  m_dRotation = 0;

           m_iCollisions=0;
           m_iRotval=0;
           m_iSpeedval=0;

  //reset its memory
  m_MemoryMap.Reset();

}

//------------------------- RenderMemory ---------------------------------
//
//------------------------------------------------------------------------
void CMinesweeper::Render(HDC surface)
{
  //render the memory
  m_MemoryMap.Render(surface);

  string s = itos(m_MemoryMap.NumCellsVisited());
  s = "Num Cells Visited: " + s;
  TextOut(surface, 220,0,s.c_str(), s.size());

}
//---------------------WorldTransform-------------------------------
//
//      sets up a translation matrix for the sweeper according to its
//  scale, rotation and position. Returns the transformed vertices.
//-------------------------------------------------------------------
void CMinesweeper::WorldTransform(vector<SPoint> &sweeper, double scale)
{
        //create the world transformation matrix
        C2DMatrix matTransform;

        //scale
        matTransform.Scale(scale, scale);

        //rotate
        matTransform.Rotate(m_dRotation);

        //and translate
        matTransform.Translate(m_vPosition.x, m_vPosition.y);

        //now transform the ships vertices
        matTransform.TransformSPoints(sweeper);
}

//------------------------------Update()-------------------------------
//
//      First we take sensor readings and feed these into the sweepers brain.
//
//      The inputs are:
//
// The readings from the minesweepers sensors
//
//      We receive two outputs from the brain.. lTrack & rTrack.
//      So given a force for each track we calculate the resultant rotation
//      and acceleration and apply to current velocity vector.
//
//--------------------------------------------------------------------
bool CMinesweeper::Update(vector<SPoint> &objects,int ival,int igen,bool bBest, int iTicks)
{

        //this will store all the inputs for the NN
        vector<double> inputs;

 //grab sensor readings
 TestSensors(objects);

 //input sensors into net
 for (int sr=0; sr<m_vecdSensors.size(); ++sr)
 {
  inputs.push_back(m_vecdSensors[sr]);
        inputs.push_back(m_vecFeelers[sr]);
 }

 inputs.push_back(m_bCollided);

        //update the brain and get feedback
 vector<double> output = m_pItsBrain->Update(inputs, CNeuralNet::active,iTicks);

        //make sure there were no errors in calculating the
        //output
        if (output.size() < CParams::iNumOutputs)
 {
  return false;
 }

        //assign the outputs to the sweepers left & right tracks
        m_lTrack = output[0];
        m_rTrack = output[1];

        //calculate steering forces
        double RotForce = m_lTrack - m_rTrack;

        //if its not rotating too much it gets bonus;
        if((RotForce>0.5)||(RotForce<-0.5))
                m_iRotval++;

//clamp rotation
        Clamp(RotForce, -CParams::dMaxTurnRate, CParams::dMaxTurnRate);

        m_dRotation += RotForce;

        //update Look At
        m_vLookAt.x = -sin(m_dRotation);
        m_vLookAt.y = cos(m_dRotation);

//if the sweepers haven't collided with an obstacle
//update their position
```

```
  if (!m_bCollided)
  {
    m_dSpeed = m_lTrack + m_rTrack;

            //if speed of exploration is more then gets bonus
            if(m_dSpeed>1.5)
                    m_iSpeedval++;


   //update position
   m_vPosition += (m_vLookAt * m_dSpeed);

   //test range of x,y values - because of 'cheap' collision detection
   //this can go into error when using < 4 sensors
   TestRange();
  }
  else
 {
             m_iCollisions++;
  }
  //update the memory map
  m_MemoryMap.Update(m_vPosition.x, m_vPosition.y);

            return true;
}


//----------------------- TestSensors -----------------------------------
//
//  This function checks for any intersections between the sweeper's
//  sensors and the objects in its environment
//-----------------------------------------------------------------------
void CMinesweeper::TestSensors(vector<SPoint> &objects)
{
  m_bCollided = false;

  //first we transform the sensors into world coordinates
  m_tranSensors = m_Sensors;

  WorldTransform(m_tranSensors, 1);  //scale is 1

  //flush the sensors
  m_vecdSensors.clear();
  m_vecFeelers.clear();

  //now to check each sensor against the objects in the world
  for (int sr=0; sr<m_tranSensors.size(); ++sr)
  {
    bool bHit = false;

    double dist = 0;

    for (int seg=0; seg<objects.size(); seg+=2)
    {
      if (LineIntersection2D(SPoint(m_vPosition.x, m_vPosition.y),
                         m_tranSensors[sr],
                         objects[seg],
                         objects[seg+1],
                         dist))
      {
        bHit = true;

        break;
      }
    }
```

```
if (bHit)
{
  m_vecdSensors.push_back(dist);

  //implement very simple collision detection
  if (dist < CParams::dCollisionDist)
  {
    m_bCollided = true;
  }
}

else
{
  m_vecdSensors.push_back(-1);
}

//check how many times the minesweeper has visited the cell
//at the current position
int HowOften = m_MemoryMap.TicksLingered(m_tranSensors[sr].x,
                                         m_tranSensors[sr].y);


//Update the memory info according to HowOften. The maximum
//value is 1 (because we want all the inputs into the
//ANN to be scaled between -1 < n < 1)
if (HowOften == 0)
{
  m_vecFeelers.push_back(-1);

  continue;
}

if (HowOften < 10)
{
  m_vecFeelers.push_back(0);

  continue;
}

if (HowOften < 20)
{
  m_vecFeelers.push_back(0.2);

  continue;
}

if (HowOften < 30)
{
  m_vecFeelers.push_back(0.4);

  continue;
}

if (HowOften < 50)
{
  m_vecFeelers.push_back(0.6);

  continue;
}

if (HowOften < 80)
{
  m_vecFeelers.push_back(0.8);

  continue;
}
```

```
     m_vecFeelers.push_back(1);

   }//next sensor
 }

 //------------------------------- TestRange ----------------------------
 //
 //--------------------------------------------------------------------
 void CMinesweeper::TestRange()
 {
  if (m_vPosition.x < 0)
  {
   m_vPosition.x = 5;
  }

  if (m_vPosition.x > CParams::WindowWidth)
  {
   m_vPosition.x = CParams::WindowWidth-5;
  }

  if (m_vPosition.y < 0)
  {
   m_vPosition.y = 5;
  }

  if (m_vPosition.y > CParams::WindowHeight)
  {
   m_vPosition.y = CParams::WindowHeight+5;
  }
 }


//---------------------------- CheckForMine ----------------------------
//
// this function checks for collision with a random mine
//--------------------------------------------------------------------
int CMinesweeper::CheckForMine(vector<SVector2D> &mines, double size)
{
         for(int i=0; i<mines.size(); i++)
         {
                         SVector2D DistToObject = m_vPosition - mines[i];

                 if (Vec2DLength(DistToObject) < (size + 10))
                 {
                                 return i;
          }
         }

  return -1;
}


//------------------------ EndOfRunCalculations() ----------------------
//
//--------------------------------------------------------------------
void CMinesweeper::EndOfRunCalculations()
{
 m_dFitness += m_MemoryMap.NumCellsVisited()+m_iSpeedval/5-m_iCollisions/5-
m_iRotval/10+m_iMines*10;

         //Another Fitness function
 //m_dFitness += m_MemoryMap.NumCellsVisited()+m_iSpeedval/10-m_iCollisions/2-
m_iRotval/10+m_iMines*5;

  //A simple fitness function
 //m_dFitness += m_MemoryMap.NumCellsVisited()
}

void CMinesweeper::incrementmineval()
{
        m_iMines++;
}




        #ifndef CPARAMS_H
#define CPARAMS_H
//-----------------------------------------------------------------------
//
//      Name: CParams.h
//
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                             Anil kumar Enumulapally  2005
//
// Desc: class to hold all the parameters used in this project. The values
//      are loaded in from an ini file when an instance of the class is
//      created.
//
//
//-----------------------------------------------------------------------
#include <windows.h>
#include <fstream>
//#include "file.h"
using namespace std;

//i-o training pairs
 static double dIop[250][13];

class CParams
{

public:

 //-------------------------------------------------------------------
 // general parameters
 //-------------------------------------------------------------------

 static double dPi;
 static double dHalfPi;
 static double dTwoPi;

 static int    WindowWidth;
 static int    WindowHeight;

 static int    InfoWindowWidth;
 static int    InfoWindowHeight;

 static int    iFramesPerSecond;
 static int    iOfflineTraining;
 static int    iGlobalOnline;
 static int    iRuleEvolution;
 static int    iOnlyGAs;

 //fstream file;

 //-------------------------------------------------------------------
 // used to define the sweepers
 //-------------------------------------------------------------------
```

```
  static int    iNumSweepers;

  //limits how fast the sweepers can turn
  static double dMaxTurnRate;

  //for controlling the size
  static int   iSweeperScale;

  //amount of sensors
  static int    iNumSensors;

  //range of sensors
  static double dSensorRange;

  //distance 0 < d < 1 for collision detection. The smaller the
  //value is the closer the sweeper has to be to an object.
  static double dCollisionDist;


  //-------------------------------------controller parameters

  //number of time steps we allow for each generation to live
  static int    iNumTicks;

  //-------------------------------------------------------------------
  //  used in CMapper.h/cpp
  //-------------------------------------------------------------------

  static double dCellSize;


  //--------------------------------------------------------------------
  // used in phenotype.h/cpp
  //--------------------------------------------------------------------

  static int    iNumInputs;
  static int    iNumOutputs;

  //bias value
  static double dBias;

  //starting value for the sigmoid response
  static double dSigmoidResponse;

//-------------------------------------------------------------------------
  //learning rate and evolutionary parameters of learning rule to use in learning
  //used in phenotype.h/cpp
//--------------------------------------------------------------------------

  static double dLearningRate;
  static double dParam1;
  static double dParam2;
  static double dParam3;
  static double dParam4;
  static double dParam5;
  static double dParam6;
  static double dParam7;
  static double dParam8;
  static double dParam9;
  static double dParam10;
  static double dParam11;
  //static double dParam1;

  //i-o training pairs
  //static double dIop[600][13];
```

```
//--------------------------------------------------------------------
// used in genotype.h/cpp
//--------------------------------------------------------------------

//number of times we try to find 2 unlinked nodes when adding a link.
//see CGenome::AddLink()
static int    iNumAddLinkAttempts;

//number of attempts made to choose a node that is not an input
//node and that does not already have a recurrently looped connection
//to itself. See CGenome::AddLink()
static int    iNumTrysToFindLoopedLink;

//the number of attempts made to find an old link to prevent chaining
//in CGenome::AddNeuron
static int    iNumTrysToFindOldLink;

//the chance, each epoch, that a neuron or link will be added to the
//genome
static double dChanceAddLink;
static double dChanceAddNode;
static double dChanceAddRecurrentLink;

//mutation probabilities for mutating the weights in CGenome::Mutate()
static double dMutationRate;
static double dMaxWeightPerturbation;
static double dProbabilityWeightReplaced;

//probabilities for mutating the activation response
static double dActivationMutationRate;
static double dMaxActivationPerturbation;

//the smaller the number the more species will be created
static double dCompatibilityThreshold;



//--------------------------------------------------------------------
// used in CSpecies.h/cpp
//--------------------------------------------------------------------

//during fitness adjustment this is how much the fitnesses of
//young species are boosted (eg 1.2 is a 20% boost)
static double dYoungFitnessBonus;

//if the species are below this age their fitnesses are boosted
static int    iYoungBonusAgeThreshhold;

//number of population to survive each epoch. (0.2 = 20%)
static double dSurvivalRate;

//if the species is above this age their fitness gets penalized
static int    iOldAgeThreshold;

//by this much
static double dOldAgePenalty;



//--------------------------------------------------------------------
// used in Cga.h/cpp
//--------------------------------------------------------------------

//how long we allow a species to exist without any improvement
static int    iNumGensAllowedNoImprovement;
```

```
//maximum number of neurons permitted in the network
static int    iMaxPermittedNeurons;

//the number of best performing sweepers shown when 'B' is
//selected. (you will see copies from the previous generation
static int    iNumBestSweepers;

static double dCrossoverRate;


//--------------------------------------------

//ctor
CParams(){}

bool Initialize()
{


        if(!LoadInParameters("params.ini"))
 {
   MessageBox(NULL, "Cannot find 'params.ini'", "Error", 0);

   return false;
 }
       // fstream file ("input.rtf", ios::out | ios::app | ios::in);

 dPi     = 3.14159265358979;
 dHalfPi = dPi / 2;
 dTwoPi  = dPi * 2;

 dCollisionDist = (double)(iSweeperScale+1)/dSensorRange;

 iNumInputs  = (iNumSensors * 2) + 1;
 iNumOutputs = 2;
        char *szFileName10="io_training4.txt";
                //fstream grab2("io_training.txt", fstream::in | fstream::out | fstream::app);

        fstream grab2(szFileName10);
        int i_here,j_here;
        //double dtmpsum;

        //populate dIop
        for (i_here=0;i_here<250;i_here++)
        {
           //trainingInputs.clear();
                //targetOutputs.clear();
                //desiredOutputs.clear();
   //dtmpsum=0;
                //Read input from file

                j_here=0;
   grab2>>dIop[i_here][j_here];

                //1
                j_here++;
   grab2>>dIop[i_here][j_here];
   //trainingInputs.push_back(dTmpvar);

                //2
                j_here++;
   grab2>>dIop[i_here][j_here];

                //3
                j_here++;
```

```
    grab2>>dIop[i_here][j_here];

            //4
            j_here++;
    grab2>>dIop[i_here][j_here];

            //5
            j_here++;
    grab2>>dIop[i_here][j_here];

            //6
            j_here++;
    grab2>>dIop[i_here][j_here];

            //7 j_here++;
            j_here++;
    grab2>>dIop[i_here][j_here];

            //8
            j_here++;
    grab2>>dIop[i_here][j_here];

            //9
            j_here++;
    grab2>>dIop[i_here][j_here];

            //10
            j_here++;
    grab2>>dIop[i_here][j_here];

            //11
            j_here++;
    grab2>>dIop[i_here][j_here];

            //12
            j_here++;
    grab2>>dIop[i_here][j_here];

       }//end of for loop

   return true;

 }

 bool LoadInParameters(char* szFileName);
};

#endif
```

```
#include "CParams.h"


double CParams::dPi                    = 0;
double CParams::dHalfPi                 = 0;
double CParams::dTwoPi                  = 0;
int CParams::WindowWidth                = 400;
int CParams::WindowHeight               = 400;
int CParams::iFramesPerSecond           = 0;
int CParams::iNumInputs                 = 0;
int CParams::iNumOutputs                = 0;
double CParams::dBias                   = -1;
double CParams::dMaxTurnRate            = 0;
int CParams::iSweeperScale              = 0;
int CParams::iNumSensors                = 0;
double CParams::dSensorRange            = 0;
int CParams::iNumSweepers               = 0;
int CParams::iNumTicks                  = 0;
double CParams::dCollisionDist          = 0;
double CParams::dCellSize               = 0;
double CParams::dSigmoidResponse        = 1;
int CParams::iNumAddLinkAttempts        = 0;
int CParams::iNumTrysToFindLoopedLink   = 5;
int CParams::iNumTrysToFindOldLink      = 5;
double CParams::dYoungFitnessBonus      = 0;
int CParams::iYoungBonusAgeThreshhold   = 0;
double CParams::dSurvivalRate           = 0;
int CParams::InfoWindowWidth            = 400;
int CParams::InfoWindowHeight           = 400;
int CParams::iNumGensAllowedNoImprovement = 0;
int CParams::iMaxPermittedNeurons       = 0;
double CParams::dChanceAddLink          = 0;
double CParams::dChanceAddNode          = 0;
double CParams::dChanceAddRecurrentLink  = 0;
double CParams::dMutationRate           = 0;
double CParams::dMaxWeightPerturbation   = 0;
double CParams::dProbabilityWeightReplaced= 0;

double CParams::dActivationMutationRate  = 0;
double CParams::dMaxActivationPerturbation= 0;

double CParams::dCompatibilityThreshold  = 0;
int CParams::iNumBestSweepers           = 4;
int CParams::iOldAgeThreshold           = 0;
double CParams::dOldAgePenalty          = 0;
double CParams::dCrossoverRate          = 0;
double CParams::dLearningRate           = 0.01;
double CParams::dParam1                 = 4;
double CParams::dParam2                 = 0;
double CParams::dParam3                 = 0;
double CParams::dParam4                 = 0;
double CParams::dParam5                 = 0;
double CParams::dParam6                 = 0;
double CParams::dParam7                 = 0;
double CParams::dParam8              = 0;
double CParams::dParam9              = -2;
double CParams::dParam10             = 2;
double CParams::dParam11             = 0;
//double CParams::dParam1             = 0;
int    CParams::iOfflineTraining     =0;
int    CParams::iGlobalOnline        =0;
int    CParams::iRuleEvolution       =0;
int    CParams::iOnlyGAs            =0;



//this function loads in the parameters from a given file name. Returns
//false if there is a problem opening the file.
bool CParams::LoadInParameters(char* szFileName)
{
  ifstream grab(szFileName);

  //check file exists
  if (!grab)
  {
    return false;
  }

  //load in from the file
  char ParamDescription[40];

  grab >> ParamDescription;
  grab >> iFramesPerSecond;
  grab >> ParamDescription;
  grab >> dMaxTurnRate;
  grab >> ParamDescription;
  grab >> iSweeperScale;
  grab >> ParamDescription;
  grab >> iNumSensors;
  grab >> ParamDescription;
  grab >> dSensorRange;
  grab >> ParamDescription;
  grab >> iNumSweepers;
  grab >> ParamDescription;
  grab >> iNumTicks;
  grab >> ParamDescription;
  grab >> dCellSize;
  grab >> ParamDescription;
  grab >> iNumAddLinkAttempts;
  grab >> ParamDescription;
  grab >> dSurvivalRate;
  grab >> ParamDescription;
  grab >> iNumGensAllowedNoImprovement;
  grab >> ParamDescription;
  grab >> iMaxPermittedNeurons;
  grab >> ParamDescription;
  grab >> dChanceAddLink;
  grab >> ParamDescription;
  grab >> dChanceAddNode;
  grab >> ParamDescription;
  grab >> dChanceAddRecurrentLink;
  grab >> ParamDescription;
  grab >> dMutationRate;
  grab >> ParamDescription;
  grab >> dMaxWeightPerturbation;
  grab >> ParamDescription;
  grab >> dProbabilityWeightReplaced;
  grab >> ParamDescription;
  grab >> dActivationMutationRate;
  grab >> ParamDescription;
  grab >> dMaxActivationPerturbation;
  grab >> ParamDescription;
  grab >> dCompatibilityThreshold;
  grab >> ParamDescription;
  grab >>iOldAgeThreshold;
  grab >> ParamDescription;
  grab >>dOldAgePenalty;
  grab >> ParamDescription;
  grab >> dYoungFitnessBonus;
  grab >> ParamDescription;
  grab >> iYoungBonusAgeThreshhold;
  grab >> ParamDescription;
  grab >> dCrossoverRate;
  grab >> ParamDescription;
  grab >> dLearningRate;
  grab >> ParamDescription;
  grab >> dParam1;
  grab >> ParamDescription;
  grab >> dParam2;
  grab >> ParamDescription;
  grab >> dParam3;
  grab >> ParamDescription;
  grab >> dParam4;
  grab >> ParamDescription;
  grab >> dParam5;
  grab >> ParamDescription;
  grab >> dParam6;
  grab >> ParamDescription;
  grab >> dParam7;
  grab >> ParamDescription;
  grab >> dParam8;
  grab >> ParamDescription;
  grab >> dParam9;
  grab >> ParamDescription;
  grab >> dParam10;
  grab >> ParamDescription;
  grab >> dParam11;
 //grab >> dParam1;
  grab >> ParamDescription;
  grab >> iOfflineTraining;
  grab >> ParamDescription;
  grab >> iGlobalOnline;
  grab >> ParamDescription;
  grab >> iRuleEvolution;
  grab >> ParamDescription;
  grab >> iOnlyGAs;


  return true;
}
```

```
#ifndef CSPECIES_H
#define CSPECIES_H
//-----------------------------------------------------------------------
//
// Name: CSpecies.h
//
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                                   Anil kumar Enumulapally  2005
//
//        Desc: Class to handle species distribution and maintenance
//-----------------------------------------------------------------------
#include <vector>
#include <math.h>
#include <iomanip>
#include <iostream>

#include "genotype.h"

using namespace std;


//-----------------------------------------------------------------------
//
// class to hold all the genomes of a given species
//-----------------------------------------------------------------------
class CSpecies
{

private:

 //keep a local copy of the first member of this species
 CGenome          m_Leader,

 //pointers to all the genomes within this species
 vector<CGenome*>  m_vecMembers;

 //the species needs an identification number
 int              m_iSpeciesID;

 //best fitness found so far by this species
 double           m_dBestFitness;

 //generations since fitness has improved, we can use
 //this info to kill off a species if required
 int              m_iGensNoImprovement;

 //age of species
 int              m_iAge;

 //how many of this species should be spawned for
 //the next population
```

```
  double          m_dSpawnsRqd;


 public:

  CSpecies(CGenome &FirstOrg, int SpeciesID);

  //this method boosts the fitnesses of the young, penalizes the
  //fitnesses of the old and then performs fitness sharing over
  //all the members of the species
  void   AdjustFitnesses();

  //adds a new individual to the species
  void   AddMember(CGenome& new_org);

  void   Purge();

  //calculates how many offspring this species should
  void   CalculateSpawnAmount();

  //spawns an individual from the species selected at random
  //from the best CParams::dSurvivalRate percent
  CGenome Spawn();


  //------------------------------------accessor methods
  CGenome  Leader()const{return m_Leader;}

  double   NumToSpawn()const{return m_dSpawnsRqd;}

  int      NumMembers()const{return m_vecMembers.size();}

  int      GensNoImprovement()const{return m_iGensNoImprovement;}

  int      ID()const{return m_iSpeciesID;}

  double   SpeciesLeaderFitness()const{return m_Leader.Fitness();}

  double   BestFitness()const{return m_dBestFitness;}

  int      Age()const{return m_iAge;}


  //so we can sort species by best fitness. Largest first
  friend bool operator<(const CSpecies &lhs, const CSpecies &rhs)
  {
    return lhs.m_dBestFitness > rhs.m_dBestFitness;
  }

};

#endif
```

```
#include "CSpecies.h"


//--------------------------------------------------------------------------
//
// this ctor creates an instance of a new species. A local copy of
// the initializing genome is kept in m_Leader and the first element
// of m_vecMembers is a pointer to that genome.
//--------------------------------------------------------------------------
CSpecies::CSpecies(CGenome  &FirstOrg,
                int      SpeciesID):m_iSpeciesID(SpeciesID),
                             m_dBestFitness(FirstOrg.Fitness()),
                             m_iGensNoImprovement(0),
                             m_iAge(0),
                             m_Leader(FirstOrg),
                             m_dSpawnsRqd(0)

{
  m_vecMembers.push_back(&FirstOrg);

  m_Leader = FirstOrg;
}

//------------------------ AddMember ------------------------------------
//
// this function adds a new member to this species and updates the member
// variables accordingly
//--------------------------------------------------------------------------
void CSpecies::AddMember(CGenome &NewMember)
{

   //is the new member's fitness better than the best fitness?
  if (NewMember.Fitness() > m_dBestFitness)
  {
    m_dBestFitness = NewMember.Fitness();

    m_iGensNoImprovement = 0;

    m_Leader = NewMember;
  }


  m_vecMembers.push_back(&NewMember);

}

//-------------------------- Purge ---------------------------------------
//
// this functions clears out all the members from the last generation,
// updates the age and gens no improvement.
//--------------------------------------------------------------------------
void CSpecies::Purge()
```

```
{
 m_vecMembers.clear();

 //update age etc
 ++m_iAge;

 ++m_iGensNoImprovement;

 m_dSpawnsRqd = 0;
}

//--------------------------- AdjustFitness ------------------------------
//
// This function adjusts the fitness of each individual by first
// examining the species age and penalising if old, boosting if young.
// Then we perform fitness sharing by dividing the fitness by the number
// of individuals in the species. This ensures a species does not grow
// too large
//-----------------------------------------------------------------------
void CSpecies::AdjustFitnesses()
{
 double total = 0;

 for (int gen=0; gen<m_vecMembers.size(); ++gen)
 {
  double fitness = m_vecMembers[gen]->Fitness();

  //boost the fitness scores if the species is young
  if (m_iAge < CParams::iYoungBonusAgeThreshhold)
  {
   fitness *= CParams::dYoungFitnessBonus;
  }

  //punish older species
  if (m_iAge > CParams::iOldAgeThreshold)
  {
   fitness *= CParams::dOldAgePenalty;
  }

  total += fitness;

  //apply fitness sharing to adjusted fitnesses
  double AdjustedFitness = fitness/m_vecMembers.size();

  m_vecMembers[gen]->SetAdjFitness(AdjustedFitness);

 }
}

//------------------------ CalculateSpawnAmount --------------------------
//
// Simply adds up the expected spawn amount for each individual in the
// species to calculate the amount of offspring this species should
// spawn
//-----------------------------------------------------------------------
void CSpecies::CalculateSpawnAmount()
{
 for (int gen=0; gen<m_vecMembers.size(); ++gen)
 {
  m_dSpawnsRqd += m_vecMembers[gen]->AmountToSpawn();

 }
}


//------------------------ Spawn ----------------------------------------
```

```
//
// Returns a random genome selected from the best individuals
//-----------------------------------------------------------------
CGenome CSpecies::Spawn()
{
 CGenome baby;

 if (m_vecMembers.size() == 1)
 {
  baby = *m_vecMembers[0];
 }

 else
 {
  int MaxIndexSize = (int) (CParams::dSurvivalRate * m_vecMembers.size())+1;

  int TheOne = RandInt(0, MaxIndexSize);

  baby = *m_vecMembers[TheOne];
 }

 return baby;
}
```

```
#ifndef CTIMER_H
#define CTIMER_H
//-----------------------------------------------------------------------
//
// Name: CTimer.h
//
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                              Anil kumar Enumulapally  2005
//
//        Desc: Windows timer class
//
//-----------------------------------------------------------------------

#include <windows.h>


class CTimer
{

private:

        LONGLONG        m_CurrentTime,
                m_LastTime,
                                  m_NextTime,
                                  m_FrameTime,
                                  m_PerfCountFreq;

        double          m_TimeElapsed,
                                  m_TimeScale;

        float            m_FPS;


public:

 //ctors
        CTimer();
        CTimer(float fps);


        //whatdayaknow, this starts the timer
 void           Start();

        //determines if enough time has passed to move onto next frame
 bool           ReadyForNextFrame();

        //only use this after a call to the above.
        double   GetTimeElapsed(){return m_TimeElapsed;}
```

```
          double   TimeElapsed();

};


#endif







#include "CTimer.h"

//---------------------- default constructor ------------------------------
//
//-----------------------------------------------------------------------

CTimer::CTimer(): m_FPS(0),
                                                m_TimeElapsed(0.0f),
                                                m_FrameTime(0),
                                                m_LastTime(0),
                                                m_PerfCountFreq(0)
{
          //how many ticks per sec do we get
          QueryPerformanceFrequency( (LARGE_INTEGER*) &m_PerfCountFreq);

          m_TimeScale = 1.0f/m_PerfCountFreq;
}

//---------------------- constructor -------------------------------------
//
//        use to specify FPS
//
//-----------------------------------------------------------------------

CTimer::CTimer(float fps): m_FPS(fps),
                                                               m_TimeElapsed(0.0f),
                                                               m_LastTime(0),
                                                               m_PerfCountFreq(0)
{

          //how many ticks per sec do we get
          QueryPerformanceFrequency( (LARGE_INTEGER*) &m_PerfCountFreq);

          m_TimeScale = 1.0f/m_PerfCountFreq;

          //calculate ticks per frame
          m_FrameTime = (LONGLONG)(m_PerfCountFreq / m_FPS);
}


//------------------------Start()----------------------------------------
//
//        call this immediately prior to game loop. Starts the timer (obviously!)
//
//-----------------------------------------------------------------------
void CTimer::Start()
{
          //get the time
          QueryPerformanceCounter( (LARGE_INTEGER*) &m_LastTime);
```

```
        //update time to render next frame
        m_NextTime = m_LastTime + m_FrameTime;

        return;
}

//------------------------ReadyForNextFrame()------------------------------
//
//      returns true if it is time to move on to the next frame step. To be used if
//      FPS is set.
//
//--------------------------------------------------------------------------
bool CTimer::ReadyForNextFrame()
{
        if (!m_FPS)
  {
    MessageBox(NULL, "No FPS set in timer", "Doh!", 0);

    return false;
  }

  QueryPerformanceCounter( (LARGE_INTEGER*) &m_CurrentTime);

        if (m_CurrentTime > m_NextTime)
        {

                m_TimeElapsed   = (m_CurrentTime - m_LastTime) * m_TimeScale;
                m_LastTime              = m_CurrentTime;

                //update time to render next frame
                m_NextTime = m_CurrentTime + m_FrameTime;

                return true;
        }

        return false;
}

//-------------------------- TimeElapsed -------------------------------
//
//      returns time elapsed since last call to this function. Use in main
//      when calculations are to be based on dt.
//
//-------------------------------------------------------------------------
double CTimer::TimeElapsed()
{
        QueryPerformanceCounter( (LARGE_INTEGER*) &m_CurrentTime);

        m_TimeElapsed   = (m_CurrentTime - m_LastTime) * m_TimeScale;

        m_LastTime              = m_CurrentTime;

        return m_TimeElapsed;

}
```

```
#ifndef NEATGENOTYPE_H
#define NEATGENOTYPE_H
//-----------------------------------------------------------------------
//
// Name: genotype.h
//
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                                 Anil kumar Enumulapally  2005
//
//        Desc: Genome description
//
//-----------------------------------------------------------------------
#include <vector>

#include "phenotype.h"
#include "utils.h"
#include "CInnovation.h"
#include "Genes.h"

using namespace std;


class Cga;
class CInnovation;


//-----------------------------------------------------------------------
//
// CGenome class definition. A genome basically consists of a vector of
// link genes, a vector of neuron genes and a fitness score.
//-----------------------------------------------------------------------
class CGenome
{

private:

  //its identification number
  int                  m_GenomeID;

  //all the neurons which make up this genome
  vector<SNeuronGene>     m_vecNeurons;

  //and all the the links
  vector<SLinkGene>       m_vecLinks;

  //pointer to its phenotype
  CNeuralNet*            m_pPhenotype;
  //its raw fitness score
  double              m_dFitness;

  //its fitness score after it has been placed into a
  //species and adjusted accordingly
  double              m_dAdjustedFitness;

  //the number of offspring this individual is required to spawn
  //for the next generation
  double              m_dAmountToSpawn;

  //keep a record of the number of inputs and outputs
  int                m_iNumInputs,
                     m_iNumOutPuts;

  //keeps a track of which species this genome is in (only used
  //for display purposes)
  int                m_iSpecies;

  //returns true if the specified link is already part of the genome
  bool    DuplicateLink(int NeuronIn, int NeuronOut);

  //given a neuron id this function just finds its position in
  //m_vecNeurons
  int     GetElementPos(int neuron_id);

  //tests if the passed ID is the same as any existing neuron IDs. Used
  //in AddNeuron
  bool    AlreadyHaveThisNeuronID(const int ID);


public:

  CGenome();

  //this constructor creates a minimal genome where there are output &
  //input neurons and every input neuron is connected to each output neuron
  CGenome(int id, int inputs, int outputs);

  //this constructor creates a genome from a vector of SLinkGenes
  //a vector of SNeuronGenes and an ID number
  CGenome(int               id,
        vector<SNeuronGene> neurons,
        vector<SLinkGene>   genes,
        int               inputs,
        int               outputs);

  ~CGenome();

  //copy constructor
  CGenome(const CGenome& g);

  //assignment operator
  CGenome& operator =(const CGenome& g);

  //create a neural network from the genome
  CNeuralNet*           CreatePhenotype(int depth);

  //delete the neural network
  void              DeletePhenotype();
  CNeuralNet*         GetPhenotype()
  {
          return(m_pPhenotype);
  }
  //add a link to the genome dependent upon the mutation rate
  void              AddLink(double      MutationRate,
                         double      ChanceOfRecurrent,
                         CInnovation &innovation,
                         int         NumTrysToFindLoop,
                         int         NumTrysToAddLink);

  //and a neuron
  void              AddNeuron(double      MutationRate,
                          CInnovation &innovation,
                          int         NumTrysToFindOldLink);

  //this function mutates the connection weights
  void              MutateWeights(double  mut_rate,
                            double  prob_new_mut,
                            double  dMaxPertubation);

  //perturbs the activation responses of the neurons
  void              MutateActivationResponse(double mut_rate,
                                  double MaxPertubation);
  // this function mutates the learning algorithm parameters
  void              MutateLearningParameters(double mut_rate,
                                                                    double
 MaxPertubation);

  //calculates the compatibility score between this genome and
  //another genome
  double             GetCompatibilityScore(const CGenome &genome);

  void              SortGenes();

  //overload '<' used for sorting. From fittest to poorest.
  friend bool operator<(const CGenome& lhs, const CGenome& rhs)
  {
    return (lhs.m_dFitness > rhs.m_dFitness);
  }


        //--------------------------------accessor methods
        int           ID()const{return m_GenomeID;}
  void   SetID(const int val){m_GenomeID = val;}

        int     NumGenes()const{return m_vecLinks.size();}
  int    NumNeurons()const{return m_vecNeurons.size();}
  int    NumInputs()const{return m_iNumInputs;}
  int    NumOutputs()const{return m_iNumOutPuts;}

  double  AmountToSpawn()const{return m_dAmountToSpawn;}
  void   SetAmountToSpawn(double num){m_dAmountToSpawn = num;}

  void   SetFitness(const double num){m_dFitness = num;}
  void   SetAdjFitness(const double num){m_dAdjustedFitness = num;}
  double  Fitness()const{return m_dFitness;}
  double  GetAdjFitness()const{return m_dAdjustedFitness;}

  int    GetSpecies()const{return m_iSpecies;}
  void   SetSpecies(int spc){m_iSpecies = spc;}

  double  SplitY(const int val)const{return m_vecNeurons[val].dSplitY;}

  vector<SLinkGene>        Genes()const{return m_vecLinks;}
  vector<SNeuronGene> Neurons()const{return m_vecNeurons;}

  vector<SLinkGene>::iterator StartOfGenes(){return m_vecLinks.begin();}
  vector<SLinkGene>::iterator EndOfGenes(){return m_vecLinks.end();}
};
```

```
#endif
```

```
#include "genotype.h"


//------------------------------------------------------------------------
//
//  default ctor
//------------------------------------------------------------------------
CGenome::CGenome():m_pPhenotype(NULL),
                m_GenomeID(0),
                m_dFitness(0),
                m_dAdjustedFitness(0),
                m_iNumInputs(0),
                m_iNumOutPuts(0),
                m_dAmountToSpawn(0)
{}


//-----------------------------constructor--------------------------------
//      this constructor creates a minimal genome where there are output +
//      input neurons and each input neuron is connected to each output neuron.
//------------------------------------------------------------------------
CGenome::CGenome(int id, int inputs, int outputs):m_pPhenotype(NULL),
                                          m_GenomeID(id),
                                          m_dFitness(0),
                                          m_dAdjustedFitness(0),
                                          m_iNumInputs(inputs),
                                          m_iNumOutPuts(outputs),
                                          m_dAmountToSpawn(0),
                                          m_iSpecies(0)

{
 //create the input neurons
 double InputRowSlice = 1/(double)(inputs+2);

         for (int i=0; i<inputs; i++)
         {
                 m_vecNeurons.push_back(SNeuronGene(input, i, 0, (i+2)*InputRowSlice));
         }

 //create the bias
 m_vecNeurons.push_back(SNeuronGene(bias, inputs, 0, InputRowSlice));

         //create the output neurons
 double OutputRowSlice = 1/(double)(outputs+1);

         for (i=0; i<outputs; i++)
         {
                m_vecNeurons.push_back(SNeuronGene(output, i+inputs+1, 1,
(i+1)*OutputRowSlice));
        }

        //create the link genes, connect each input neuron to each output neuron and
        //assign a random weight -1 < w < 1
        for (i=0; i<inputs+1; i++)
        {
                for (int j=0; j<outputs; j++)
                {
                        m_vecLinks.push_back(SLinkGene(m_vecNeurons[i].iID,
                          m_vecNeurons[inputs+j+1].iID,
                          true,
                          inputs+outputs+1+NumGenes(),
                          RandomClamped()));
                }
        }

}

//---------------------------------------------------------------------
//
// this constructor creates a genome from a vector of SLinkGenes, a
// vector of SNeuronGenes and an ID number.
//---------------------------------------------------------------------
CGenome::CGenome(int              id,
           vector<SNeuronGene> neurons,
           vector<SLinkGene>   genes,
           int             inputs,
           int             outputs):m_GenomeID(id),
                               m_pPhenotype(NULL),
                               m_vecLinks(genes),
                               m_vecNeurons(neurons),
                               m_dAmountToSpawn(0),
                               m_dFitness(0),
                               m_dAdjustedFitness(0),
                               m_iNumInputs(inputs),
                               m_iNumOutPuts(outputs)

{}

//------------------------------dtor----------------------------------------------------
//
//-------------------------------------------------------------------------------------
CGenome::~CGenome()
{
 if (m_pPhenotype)
 {
                delete m_pPhenotype;

   m_pPhenotype = NULL;
 }
}

//--------------------------------copy ctor--------------------------------------------
//
//------------------------------------------------------------------------------------
CGenome::CGenome(const CGenome& g)
{
   m_GenomeID   = g.m_GenomeID;
   m_vecNeurons  = g.m_vecNeurons;
   m_vecLinks   = g.m_vecLinks;
   m_pPhenotype = NULL;           //no need to perform a deep copy
   m_dFitness   = g.m_dFitness;
   m_dAdjustedFitness = g.m_dAdjustedFitness;
   m_iNumInputs  = g.m_iNumInputs;
```

```
  m_iNumOutPuts = g.m_iNumOutPuts;
  m_dAmountToSpawn = g.m_dAmountToSpawn;
}

//--------------------------------assignment operator----------------------------------
//
//-------------------------------------------------------------------------------------
CGenome& CGenome::operator =(const CGenome& g)
{
  //self assignment guard
        if (this != &g)
        {
    m_GenomeID        = g.m_GenomeID;
    m_vecNeurons       = g.m_vecNeurons;
    m_vecLinks        = g.m_vecLinks;
    m_pPhenotype      = NULL;       //no need to perform a deep copy
    m_dFitness        = g.m_dFitness;
    m_dAdjustedFitness = g.m_dAdjustedFitness;
    m_iNumInputs       = g.m_iNumInputs;
    m_iNumOutPuts      = g.m_iNumOutPuts;
    m_dAmountToSpawn   = g.m_dAmountToSpawn;
  }

  return *this;
}

//------------------------------CreatePhenotype-------------------------
//
//        Creates a neural network based upon the information in the genome.
//        Returns a pointer to the newly created ANN
//---------------------------------------------------------------------
CNeuralNet* CGenome::CreatePhenotype(int depth)
{
 //first make sure there is no existing phenotype for this genome
 DeletePhenotype();

 //this will hold all the neurons required for the phenotype
 vector<SNeuron*>  vecNeurons;

 //first, create all the required neurons
 for (int i=0; i<m_vecNeurons.size(); i++)
 {
   SNeuron* pNeuron = new SNeuron(m_vecNeurons[i].NeuronType,
                        m_vecNeurons[i].iID,
                        m_vecNeurons[i].dSplitY,
                        m_vecNeurons[i].dSplitX,
                        m_vecNeurons[i].dActivationResponse
                                                              );


   vecNeurons.push_back(pNeuron);
 }

 //now to create the links.
 for (int cGene=0; cGene<m_vecLinks.size(); ++cGene)
 {
   //make sure the link gene is enabled before the connection is created
   if (m_vecLinks[cGene].bEnabled)
   {
    //get the pointers to the relevant neurons
    int element        = GetElementPos(m_vecLinks[cGene].FromNeuron);
    SNeuron* FromNeuron = vecNeurons[element];

    element          = GetElementPos(m_vecLinks[cGene].ToNeuron);
    SNeuron* ToNeuron = vecNeurons[element];
    //create a link between those two neurons and assign the weight stored
    //in the gene
    SLink tmpLink(m_vecLinks[cGene].dWeight,
            FromNeuron,
            ToNeuron,
            m_vecLinks[cGene].bRecurrent);

    //add new links to neuron
    FromNeuron->vecLinksOut.push_back(tmpLink);
    ToNeuron->vecLinksIn.push_back(tmpLink);
   }
  }
        for(int i_temp=0; i_temp<vecNeurons.size(); i_temp++)
        {
                //setting the error status for each neuron
                vecNeurons[i_temp]->iErrorStatus=vecNeurons[i_temp]->vecLinksOut.size();
        }
  //now the neurons contain all the connectivity information, a neural
  //network may be created from them.
  m_pPhenotype = new CNeuralNet(vecNeurons, depth);

  return m_pPhenotype;
}

//--------------------------- DeletePhenotype ----------------------------
//
//----------------------------------------------------------------------
void CGenome::DeletePhenotype()
{
 if (m_pPhenotype)
 {
  delete m_pPhenotype;
 }

 m_pPhenotype = NULL;
}

//---------------------------- GetElementPos -----------------------------
//
//        given a neuron ID this little function just finds its position in
//  m_vecNeurons
//----------------------------------------------------------------------
int CGenome::GetElementPos(int neuron_id)
{
 for (int i=0; i<m_vecNeurons.size(); i++)
        {
                if (m_vecNeurons[i].iID == neuron_id)
   {
    return i;
   }
        }

 MessageBox(NULL, "Error in CGenome::GetElementPos", "Problem!", MB_OK);

        return -1;
}

//-----------------------------DuplicateLink----------------------------
//
// returns true if the link is already part of the genome
//----------------------------------------------------------------------
bool CGenome::DuplicateLink(int NeuronIn, int NeuronOut)
{
        for (int cGene = 0; cGene < m_vecLinks.size(); ++cGene)
        {
                if ((m_vecLinks[cGene].FromNeuron == NeuronIn) &&
      (m_vecLinks[cGene].ToNeuron == NeuronOut))
                  {
                        //we already have this link
                        return true;
                  }
          }

          return false;
}

//-------------------------------AddLink--------------------------------
//
// create a new link with the probability of CParams::dChanceAddLink
//----------------------------------------------------------------------
void CGenome::AddLink(double      MutationRate,
                  double      ChanceOfLooped,
                  CInnovation  &innovation,
                  int         NumTrysToFindLoop,
                  int         NumTrysToAddLink)
{
 //just return dependent on the mutation rate
 if (RandFloat() > MutationRate) return;

 //define holders for the two neurons to be linked. If we have find two
 //valid neurons to link these values will become >= 0.
 int ID_neuron1 = -1;
 int ID_neuron2 = -1;

 //flag set if a recurrent link is selected (looped or normal)
 bool bRecurrent = false;

 //first test to see if an attempt shpould be made to create a
 //link that loops back into the same neuron
 if (RandFloat() < ChanceOfLooped)
 {
  //YES: try NumTrysToFindLoop times to find a neuron that is not an
  //input or bias neuron and that does not already have a loopback
  //connection
  while(NumTrysToFindLoop--)
  {
    //grab a random neuron
    int NeuronPos = RandInt(m_iNumInputs+1, m_vecNeurons.size()-1);

    //check to make sure the neuron does not already have a loopback
    //link and that it is not an input or bias neuron
    if (!m_vecNeurons[NeuronPos].bRecurrent &&
       (m_vecNeurons[NeuronPos].NeuronType != bias) &&
       (m_vecNeurons[NeuronPos].NeuronType != input))
    {
     ID_neuron1 = ID_neuron2 = m_vecNeurons[NeuronPos].iID;

     m_vecNeurons[NeuronPos].bRecurrent = true;

     bRecurrent = true;

     NumTrysToFindLoop = 0;
    }
  }
 }

 else
 {
  //No: try to find two unlinked neurons. Make NumTrysToAddLink
  //attempts
  while(NumTrysToAddLink--)
  {
    //choose two neurons, the second must not be an input or a bias
    ID_neuron1 = m_vecNeurons[RandInt(0, m_vecNeurons.size()-1)].iID;

    ID_neuron2 =
    m_vecNeurons[RandInt(m_iNumInputs+1, m_vecNeurons.size()-1)].iID;

    if (ID_neuron2 == 2)
    {
     continue;
    }

    //make sure these two are not already linked and that they are
    //not the same neuron
    if ( !( DuplicateLink(ID_neuron1, ID_neuron2) ||
          (ID_neuron1 == ID_neuron2)))
    {
     NumTrysToAddLink = 0;
    }

    else
    {
     ID_neuron1 = -1;
     ID_neuron2 = -1;
    }
  }
}

//return if unsuccessful in finding a link
if ( (ID_neuron1 < 0) || (ID_neuron2 < 0) )
{
  return;
}

//check to see if we have already created this innovation
int id = innovation.CheckInnovation(ID_neuron1, ID_neuron2, new_link);

//is this link recurrent?
if (m_vecNeurons[GetElementPos(ID_neuron1)].dSplitY >
   m_vecNeurons[GetElementPos(ID_neuron2)].dSplitY)
{
 bRecurrent = true;
}

if ( id < 0)
{
 //we need to create a new innovation
 innovation.CreateNewInnovation(ID_neuron1, ID_neuron2, new_link);

 //then create the new gene
 int id = innovation.NextNumber() - 1;

 SLinkGene NewGene(ID_neuron1,
                   ID_neuron2,
                   true,
                   id,
                   RandomClamped(),
                   bRecurrent);

 m_vecLinks.push_back(NewGene);
}

else
{
 //the innovation has already been created so all we need to
 //do is create the new gene using the existing innovation ID
 SLinkGene NewGene(ID_neuron1,
                    ID_neuron2,
                    true,
                    id,
                    RandomClamped(),
                    bRecurrent);

   m_vecLinks.push_back(NewGene);
  }

  return;
}

//--------------------------------AddNeuron------------------------------
//
//        this function adds a neuron to the genotype by examining the network,
//        splitting one of the links and inserting the new neuron.
//---------------------------------------------------------------------
void CGenome::AddNeuron(double       MutationRate,
                    CInnovation &innovations,
                    int          NumTrysToFindOldLink)
{
  //just return dependent on mutation rate
  if (RandFloat() > MutationRate) return;

  //if a valid link is found into which to insert the new neuron
  //this value is set to true.
  bool bDone = false;

  //this will hold the index into m_vecLinks of the chosen link gene
  int  ChosenLink = 0;

  //first a link is chosen to split. If the genome is small the code makes
  //sure one of the older links is split to ensure a chaining effect does
  //not occur. Here, if the genome contains less than 5 hidden neurons it
  //is considered to be too small to select a link at random
  const int SizeThreshold = m_iNumInputs + m_iNumOutPuts + 5;

  if (m_vecLinks.size() < SizeThreshold)
  {
   while(NumTrysToFindOldLink--)
   {
    //choose a link with a bias towards the older links in the genome
    ChosenLink = RandInt(0, NumGenes()-1-(int)sqrt(NumGenes()));

    //make sure the link is enabled and that it is not a recurrent link
    //or has a bias input
    int FromNeuron = m_vecLinks[ChosenLink].FromNeuron;

    if ( (m_vecLinks[ChosenLink].bEnabled)    &&
         (!m_vecLinks[ChosenLink].bRecurrent) &&
         (m_vecNeurons[GetElementPos(FromNeuron)].NeuronType != bias))
     {
       bDone = true;

       NumTrysToFindOldLink = 0;
     }
   }

   if (!bDone)
   {
    //failed to find a decent link
    return;
   }
  }

  else
```

```
{
  //the genome is of sufficient size for any link to be acceptable
  while (!bDone)
  {
    ChosenLink = RandInt(0, NumGenes()-1);

    //make sure the link is enabled and that it is not a recurrent link
    //or has a BIAS input
    int FromNeuron = m_vecLinks[ChosenLink].FromNeuron;

    if ( (m_vecLinks[ChosenLink].bEnabled) &&
         (!m_vecLinks[ChosenLink].bRecurrent) &&
         (m_vecNeurons[GetElementPos(FromNeuron)].NeuronType != bias))
    {
      bDone = true;
    }
  }
}

//disable this gene
m_vecLinks[ChosenLink].bEnabled = false;

//grab the weight from the gene (we want to use this for the weight of
//one of the new links so that the split does not disturb anything the
//NN may have already learned...
double OriginalWeight = m_vecLinks[ChosenLink].dWeight;

//identify the neurons this link connects
int from =  m_vecLinks[ChosenLink].FromNeuron;
int to   =  m_vecLinks[ChosenLink].ToNeuron;

//calculate the depth and width of the new neuron. We can use the depth
//to see if the link feeds backwards or forwards
double NewDepth = (m_vecNeurons[GetElementPos(from)].dSplitY +
                   m_vecNeurons[GetElementPos(to)].dSplitY) /2;

double NewWidth = (m_vecNeurons[GetElementPos(from)].dSplitX +
                   m_vecNeurons[GetElementPos(to)].dSplitX) /2;

//Now to see if this innovation has been created previously by
//another member of the population
int id = innovations.CheckInnovation(from,
                                     to,
                                     new_neuron);



/*
This function must check to see if a neuron ID is already
being used. If it is then the function creates a new innovation
for the neuron. */
if (id >= 0)
{
  int NeuronID = innovations.GetNeuronID(id);

  if (AlreadyHaveThisNeuronID(NeuronID))
  {
    id = -1;
  }
}

if (id < 0)
{
  //add the innovation for the new neuron
  int NewNeuronID = innovations.CreateNewInnovation(from,
                                                    to,
                                          new_neuron,
                                          hidden,
                                          NewWidth,
                                          NewDepth);

  //create the new neuron gene and add it.
  m_vecNeurons.push_back(SNeuronGene(hidden,
                              NewNeuronID,
                              NewDepth,
                              NewWidth));

  //Two new link innovations are required, one for each of the
  //new links created when this gene is split.

  //----------------------------------first link

  //get the next innovation ID
  int idLink1 = innovations.NextNumber();

  //create the new innovation
  innovations.CreateNewInnovation(from,
                            NewNeuronID,
                            new_link);

  //create the new link gene
  SLinkGene link1(from,
                 NewNeuronID,
                 true,
                 idLink1,
                 (1.0-OriginalWeight/2.0));

  m_vecLinks.push_back(link1);

  //----------------------------------second link

  //get the next innovation ID
  int idLink2 = innovations.NextNumber();

  //create the new innovation
  innovations.CreateNewInnovation(NewNeuronID,
                            to,
                            new_link);

  //create the new gene
  SLinkGene link2(NewNeuronID,
                 to,
                 true,
                 idLink2,
                 OriginalWeight/2.0);

  m_vecLinks.push_back(link2);
}

else
{
  //this innovation has already been created so grab the relevant neuron
  //and link info from the innovation database
  int NewNeuronID = innovations.GetNeuronID(id);

  //get the innovation IDs for the two new link genes.
  int idLink1 = innovations.CheckInnovation(from, NewNeuronID, new_link);
  int idLink2 = innovations.CheckInnovation(NewNeuronID, to, new_link);

  //this should never happen because the innovations *should* have already
  //occurred
  if ( (idLink1 < 0) || (idLink2 < 0) )
```

```
    {
     MessageBox(NULL, "Error in CGenome::AddNeuron", "Problem!", MB_OK);

     return;
    }

    //now we need to create 2 new genes to represent the new links
    SLinkGene link1(from, NewNeuronID, true, idLink1, 1.0);
    SLinkGene link2(NewNeuronID, to, true, idLink2, OriginalWeight);

    m_vecLinks.push_back(link1);
    m_vecLinks.push_back(link2);

    //create the new neuron
    SNeuronGene NewNeuron(hidden, NewNeuronID, NewDepth, NewWidth);

    //and add it
    m_vecNeurons.push_back(NewNeuron);
  }

  return;
}


//--------------------------- AlreadyHaveThisNeuronID ----------------------
//
// tests to see if the parameter is equal to any existing neuron ID's.
// Returns true if this is the case.
//------------------------------------------------------------------------
bool CGenome::AlreadyHaveThisNeuronID(const int ID)
{
 for (int n=0; n<m_vecNeurons.size(); ++n)
 {
  if (ID == m_vecNeurons[n].iID)
  {
   return true;
  }
 }

 return false;
}
//------------------------------- MutateWeights--------------------------
//        Iterates through the genes and purturbs the weights given a
//  probability mut_rate.
//
//        prob_new_mut is the chance that a weight may get replaced by a
//  completely new weight.
//
//        dMaxPertubation is the maximum perturbation to be applied.
//
//        type is the type of random number algorithm we use
//------------------------------------------------------------------------
void CGenome::MutateWeights(double mut_rate,
                          double prob_new_mut,
                          double MaxPertubation)
{
          for (int cGen=0; cGen<m_vecLinks.size(); ++cGen)
          {
                    //do we mutate this gene?
                    if (RandFloat() < mut_rate)
                    {
                              //do we change the weight to a completely new weight?
                              if (RandFloat() < prob_new_mut)
                              {
                                        //change the weight using the random distribtion defined by 'type'
                                        m_vecLinks[cGen].dWeight = RandomClamped();
```

```
                        }

                        else
                        {
                                //perturb the weight
                                m_vecLinks[cGen].dWeight -= RandomClamped() * MaxPertubation;
                        }
                }
        }

        return;
}

void CGenome::MutateActivationResponse(double mut_rate,
                        double MaxPertubation)
{
 for (int cGen=0; cGen<m_vecNeurons.size(); ++cGen)
 {
  if (RandFloat() < mut_rate)
  {
   m_vecNeurons[cGen].dActivationResponse += RandomClamped() * MaxPertubation;
  }
 }
}


void CGenome::MutateLearningParameters(double mut_rate,double MaxPertubation)
{







}

//------------------------- GetCompatibilityScore -------------------------
//
//  this function returns a score based on the compatibility of this
//  genome with the passed genome
//------------------------------------------------------------------------
double CGenome::GetCompatibilityScore(const CGenome &genome)
{
 //travel down the length of each genome counting the number of
 //disjoint genes, the number of excess genes and the number of
 //matched genes
 double NumDisjoint = 0;
 double NumExcess   = 0;
 double NumMatched  = 0;

 //this records the summed difference of weights in matched genes
 double WeightDifference = 0;

 //position holders for each genome. They are incremented as we
 //step down each genomes length.
 int g1 = 0;
 int g2 = 0;

 while ( (g1 < m_vecLinks.size()-1) || (g2 < genome.m_vecLinks.size()-1) )
 {
  //we've reached the end of genome1 but not genome2 so increment
  //the excess score
  if (g1 == m_vecLinks.size()-1)
  {
      ++g2;
      ++NumExcess;

      continue;
    }

    //and vice versa
    if (g2 == genome.m_vecLinks.size()-1)
    {
      ++g1;
      ++NumExcess;

      continue;
    }

    //get innovation numbers for each gene at this point
    int id1 = m_vecLinks[g1].InnovationID;
    int id2 = genome.m_vecLinks[g2].InnovationID;

    //innovation numbers are identical so increase the matched score
    if (id1 == id2)
    {
      ++g1;
      ++g2;
      ++NumMatched;

      //get the weight difference between these two genes
      WeightDifference += fabs(m_vecLinks[g1].dWeight - genome.m_vecLinks[g2].dWeight);
    }

    //innovation numbers are different so increment the disjoint score
    if (id1 < id2)
    {
      ++NumDisjoint;
      ++g1;
    }

    if (id1 > id2)
    {
      ++NumDisjoint;
      ++g2;
    }

  }//end while

  //get the length of the longest genome
  int longest = genome.NumGenes();

  if (NumGenes() > longest)
  {
    longest = NumGenes();
  }

  //these are multipliers used to tweak the final score.
  const double mDisjoint = 1;
  const double mExcess   = 1;
  const double mMatched  = 0.4;

  //finally calculate the scores
  double score = (mExcess * NumExcess/(double)longest) +
               (mDisjoint * NumDisjoint/(double)longest) +
               (mMatched * WeightDifference / NumMatched);


  return score;
}
```

```
//--------------------------- SortGenes ---------------------------------
//
// sorts the genes
//--------------------------------------------------------------------
void CGenome::SortGenes()
{
  sort (m_vecLinks.begin(), m_vecLinks.end());
}







#ifndef PHENOTYPE_H
#define PHENOTYPE_H

//--------------------------------------------------------------------
//
// Name: phenotype.h
//
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                                  Anil kumar Enumulapally  2005
//
//         Desc: definitions required for the creation of a neural network.
//
//--------------------------------------------------------------------

#include <vector>
#include <math.h>
#include <windows.h>
#include <algorithm>

#include "utils.h"
#include "CParams.h"
#include "genes.h"


using namespace std;



struct SNeuron;

//--------------------------------------------------------------------
//
// SLink structure
//--------------------------------------------------------------------
struct SLink
{
 //pointers to the neurons this link connects
 SNeuron*  pIn;
 SNeuron*  pOut;

 //the connection weight
 double  dWeight;

 //is this link a recurrent link?
 bool    bRecurrent;

 SLink(double dW, SNeuron* pIn, SNeuron* pOut, bool bRec):dWeight(dW),
                                              pIn(pIn),
                                              pOut(pOut),
                                              bRecurrent(bRec)
 {}
};


//-----------------------------------------------------------------------
//
// SNeuron
//-----------------------------------------------------------------------
struct SNeuron
{
public:
          //all the links coming into this neuron
 vector<SLink> vecLinksIn;

 //and out
 vector<SLink> vecLinksOut;

 //sum of weights x inputs
 double       dSumActivation;

 //the output from this neuron
 double       dOutput;

 //what type of neuron is this?
 neuron_type   NeuronType;

 //its identification number
 int          iNeuronID;

 //sets the curvature of the sigmoid function
 double       dActivationResponse;

 //indicates status of error i.e. whether the error is collected from all output neurons or not
 int         iErrorStatus;

 //sets the learning rate for backpropagation and gradient descent
 double      dLearningRate;

 //stores the error for this neuron
 double    dError;

 //stores the desired output for the neuron. only valid for output neurons
 double    dDesiredOutput;


 //double    dBpParam1;

 //used in visualization of the phenotype
 int          iPosX,   iPosY;
 double       dSplitY, dSplitX;

 //--- ctors
 SNeuron(neuron_type type,
        int        id,
        double      y,
        double      x,
        double      ActResponse):NeuronType(type),
                           iNeuronID(id),
                           dSumActivation(0),
                           dOutput(0),
dLearningRate(CParams::dLearningRate),
                                                                      dError(1),
                                                                      dDesiredOutput(-1),
                                 iPosX(0),
                                                                      iErrorStatus(vecLinksOut.size()),

                                 iPosY(0),
                                 dSplitY(y),
                                 dSplitX(x),
                                 dActivationResponse(ActResponse)


          {}
};

//--------------------------------------------------------------------
//
// CNeuralNet
//--------------------------------------------------------------------
class CNeuralNet
{

private:

  vector<SNeuron*>  m_vecpNeurons;

  //the depth of the network
  int               m_iDepth;


public:

              double    dBpParam1;
              double    dBpParam2;
              double    dBpParam3;
              double    dBpParam4;
              double    dBpParam5;
              double    dBpParam6;
              double    dBpParam7;
              double    dBpParam8;
              double    dBpParam9;
              double    dBpParam10;
              double    dBpParam11;


  CNeuralNet(vector<SNeuron*> neurons,
             int             depth);


  ~CNeuralNet();

   //you have to select one of these types when updating the network
  //If snapshot is chosen the network depth is used to completely
  //flush the inputs through the network. active just updates the
  //network each timestep
  enum run_type{snapshot, active};

  //update network for this clock cycle
  vector<double>  Update(const vector<double> &inputs, const run_type type, const int iTicks);

  //offline training for a pre defined scenario
 double offlineTraining(HWND hwnd1);

  //mutating learning algorithm's parameters
  void MutateLearningParameters(double mut_rate,
                                                                    double MaxPertubation);
  //backpropagation routine called in offline training
  void  Backprop();
  void hiddenneuronerror(SNeuron*);
  inline vector<double> similaritymeasure(const vector<double> &input);

  //draws a graphical representation of the network to a user speciefied window
  void          DrawNet(HDC &surface,
                       int cxLeft,
                       int cxRight,
                       int cyTop,
                       int cyBot);

};


#endif



#include "phenotype.h"

//------------------------------------Sigmoid function------------------------
//
//-------------------------------------------------------------------------

float Sigmoid(float netinput, float response)
{
          return ( 1 / ( 1 + exp(-netinput / response)));
}


//--------------------------------- ctor --------------------------------
//
//-----------------------------------------------------------------------
CNeuralNet::CNeuralNet(vector<SNeuron*> neurons,
                  int            depth):m_vecpNeurons(neurons),
                                     m_iDepth(depth),

dBpParam1(CParams::dParam1),

dBpParam2(CParams::dParam2),

dBpParam3(CParams::dParam3),

dBpParam4(CParams::dParam4),

dBpParam5(CParams::dParam5),

dBpParam6(CParams::dParam6),

dBpParam7(CParams::dParam7),

dBpParam8(CParams::dParam8),

dBpParam9(CParams::dParam9),

dBpParam10(CParams::dParam10),

dBpParam11(CParams::dParam11)

{}


//--------------------------------- dtor --------------------------------
```

```
//
//-----------------------------------------------------------------------
CNeuralNet::~CNeuralNet()
{
  //delete any live neurons
  for (int i=0; i<m_vecpNeurons.size(); ++i)
  {
    if (m_vecpNeurons[i])
    {
      delete m_vecpNeurons[i];

      m_vecpNeurons[i] = NULL;
    }
  }
}




// This implments the offline modified back propagation algorithm
double CNeuralNet::offlineTraining(HWND hwnd1)
{
	int i_local,i_iter;
	int iLastinputneuron;
	double dTmpvar;
	int iTrainingSize=250;//number of training examples
	int iIterationSize=1000 ;//Numer of times we iterate
	char *szFileName1="io_training3.txt";

	ifstream grab1(szFileName1);
	vector<double> trainingInputs;
	vector<double> targetOutputs;
	vector<double> errorVector1,errorVector2;
for(i_iter=0;i_iter<1;i_iter++)
{
	int size1=m_vecpNeurons.size();
	//MessageBox(hwnd1,"iter_of_offline","progress",MB_OK);
	errorVector1.clear();
	errorVector2.clear();
	for (i_local=0;i_local<iTrainingSize;i_local++)
	{
	   trainingInputs.clear();
			targetOutputs.clear();
			//errorVector1.clear();
			//errorVector2.clear();

			//Read input from file

    //grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][0]);

			//grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][1]);

			//grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][2]);

			//grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][3]);

			//grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][4]);

			//grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][5]);
```

```
                //grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][6]);

                //grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][7]);

                //grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][8]);

                //grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][9]);

                //grab1>>dTmpvar;
    trainingInputs.push_back(dIop[i_local][10]);

                //Read desired output

                //grab1>>dTmpvar;
    targetOutputs.push_back(dIop[i_local][11]);

                //grab1>>dTmpvar;
    targetOutputs.push_back(dIop[i_local][12]);

                //MessageBox(hwnd1,"In phenotype offlinetraingig,after reading
vals","progress3",MB_OK);

                        //this is an index into the current neuron
                int cNeuron = 0;

                //first set the outputs of the 'input' neurons to be equal
                //to the values passed into the function in inputs
                while (cNeuron<10)
                {
                  m_vecpNeurons[cNeuron]->dOutput = trainingInputs[cNeuron];

                  ++cNeuron;
                }
                //MessageBox(hwnd1,"In phenotype offlinetraingig,after setting 10 i/p
vals","progress4",MB_OK);

                //set the output of the bias to 1
                m_vecpNeurons[10]->dOutput = 1;

                cNeuron=11;
                //then we step through the network one neuron at a time
                //MessageBox(hwnd1,"In phenotype offlinetraingig,bef while loop","progress5",MB_OK);
                while (cNeuron < m_vecpNeurons.size())
                {
                 //this will hold the sum of all the inputs x weights
                 double sum = 0;

                 //sum this neuron's inputs by iterating through all the links into
                 //the neuron
                 for (int lnk=0; lnk<m_vecpNeurons[cNeuron]->vecLinksIn.size(); ++lnk)
                 {
                        //get this link's weight
                        double Weight = m_vecpNeurons[cNeuron]->vecLinksIn[lnk].dWeight;

                        //get the output from the neuron this link is coming from
                        double NeuronOutput =
                        m_vecpNeurons[cNeuron]->vecLinksIn[lnk].pIn->dOutput;

                        //add to sum
                        sum += Weight * NeuronOutput;
                 }
```

```
                    //now put the sum through the activation function and assign the
                    //value to this neuron's output
                    m_vecpNeurons[cNeuron]->dOutput = Sigmoid(sum, m_vecpNeurons[cNeuron]-
>dActivationResponse);



                    //next neuron
                    ++cNeuron;
                  }//end of while loop
                  //MessageBox(hwnd1,"In phenotype offlinetraingig,after the while loop of o/p
vals","progress6",MB_OK);

  //calculate error


                  //the following sets error status for output neurons to zero
                  cNeuron=0;
                  int iOutputindex=0;
                  bool flag_output=false;

                  while(cNeuron<m_vecpNeurons.size())
                  {
                        if (m_vecpNeurons[cNeuron]->NeuronType == output)
                        {
                              m_vecpNeurons[cNeuron]->iErrorStatus=0;

                          if(flag_output==false)//then it is 1st output neuron
                          {
                                m_vecpNeurons[cNeuron]->dDesiredOutput=targetOutputs[0];
                                flag_output=true;
                          }
                          else //itis 2nd output neuron
                          {
                                m_vecpNeurons[cNeuron]->dDesiredOutput=targetOutputs[1];
                          }//end of inside IF else loop


                        }                         //end of outside if loop
                        if (m_vecpNeurons[cNeuron]->NeuronType == hidden)
                        {
                              m_vecpNeurons[cNeuron]->iErrorStatus=m_vecpNeurons[cNeuron]-
>vecLinksOut.size();
                        }
                        //next neuron
                        ++cNeuron;

                  }
                  //MessageBox(hwnd1,"after setting desired ops","progress",MB_OK);


                  cNeuron=0;
                  iLastinputneuron=0;
                  while(m_vecpNeurons[cNeuron]->NeuronType == input)
                  {
                        iLastinputneuron++;
                        cNeuron++;
                  }


                  //error propagation routine
                  cNeuron=m_vecpNeurons.size()-1;
                  int flag_out=0;


                  while(cNeuron>iLastinputneuron)
		{
			//output neuron error & weight adjustment
			if(m_vecpNeurons[cNeuron]->NeuronType == output)
			{
				m_vecpNeurons[cNeuron]->dError=(m_vecpNeurons[cNeuron]-
>dDesiredOutput-m_vecpNeurons[cNeuron]->dOutput)* m_vecpNeurons[cNeuron]->dOutput*(1-
m_vecpNeurons[cNeuron]->dOutput);
				if(flag_out==1)
				errorVector1.push_back(m_vecpNeurons[cNeuron]->dError);
				else
				{
					errorVector2.push_back(m_vecpNeurons[cNeuron]->dError);
					flag_out=1;
				}
				//Updating the weights
			/*comment-begin here for normal error prop*/
				/*
				for(int lnk1=0;lnk1< m_vecpNeurons[cNeuron]-
>vecLinksIn.size();lnk1++)
				{
					m_vecpNeurons[cNeuron]->vecLinksIn[lnk1].dWeight -=
0.5*m_vecpNeurons[cNeuron]->dLearningRate*m_vecpNeurons[cNeuron]-
>dError*m_vecpNeurons[cNeuron]->vecLinksIn[lnk1].pIn->dOutput;


				}//end of for loop
				*/
		/*comment-end here for normal error prop */

			}//end of if loop for output neurons


		//	MessageBox(hwnd1,"In phenotype offlinetraingig,after setting error for
o/p","progress8",MB_OK);

			/*comment-begin here for normal error prop*/
			/*
			if(m_vecpNeurons[cNeuron]->NeuronType == hidden)
			{
				//MessageBox(hwnd1,"before calling hidden
neuronerror","progress",MB_OK);
				hiddenneuronerror(m_vecpNeurons[cNeuron]);
				//MessageBox(hwnd1,"after calling hidden
neuronerror","progress",MB_OK);

				//Updating the error
				for(int lnk3=0,lnk3<m_vecpNeurons[cNeuron]-
>vecLinksIn.size();lnk3++)
				{
					//update weights
					m_vecpNeurons[cNeuron]->vecLinksIn[lnk3].dWeight -=
m_vecpNeurons[cNeuron]->dLearningRate*m_vecpNeurons[cNeuron]-
>dError*m_vecpNeurons[cNeuron]->vecLinksIn[lnk3].pIn->dOutput;


				}//end of for loop for updating weights



			}//end of if loop for hidden neurons
			*/
			/*comment-end here for normal error propagation */


			cNeuron--;//next iteration
```

```
                }//end of while loop



        }// end of i_local loop


        //A different approach in averaging the error
        double dAvgErr1=0.0,dAvgErr2=0.0;
   for(int g1=0;g1<errorVector1.size();g1++)
        {
                dAvgErr1+=errorVector1.at(g1);
                dAvgErr2+=errorVector2.at(g1);
        }
        dAvgErr1= dAvgErr1/errorVector1.size();
        dAvgErr2= dAvgErr2/errorVector2.size();
   //Updating the error
                int cNeuron1=size1-1;
                int flag_out1=0;

/*comment-begin here for other error prop */
                while(cNeuron1>10)
                {
                        if(m_vecpNeurons[cNeuron1]->NeuronType == output)
                        {
                                if(flag_out1==0)
                                {
                                        m_vecpNeurons[cNeuron1]->dError=dAvgErr2;
                                        flag_out1=1;
                                }
                                else
                                        m_vecpNeurons[cNeuron1]->dError=dAvgErr1;


                                for(int lnk1=0;lnk1< m_vecpNeurons[cNeuron1]-
>vecLinksIn.size();lnk1++)
                                {
                                        m_vecpNeurons[cNeuron1]->vecLinksIn[lnk1].dWeight -=
0.5*m_vecpNeurons[cNeuron1]->dLearningRate*m_vecpNeurons[cNeuron1]-
>dError*m_vecpNeurons[cNeuron1]->vecLinksIn[lnk1].pIn->dOutput;


                                }//end of for loop
                        }//end of output neuron IF

//MessageBox(hwnd1,"In phenotype offlinetraingig,after o/p error vals","progress8",MB_OK);

                                if(m_vecpNeurons[cNeuron1]->NeuronType == hidden)
                        {
                                //MessageBox(hwnd1,"before calling hidden
neuronerror","progress9",MB_OK);
                                hiddenneuronerror(m_vecpNeurons[cNeuron1]);
                                //MessageBox(hwnd1,"after calling hidden
neuronerror","progress10",MB_OK);

                                //Updating the error
                                for(int lnk3=0;lnk3<m_vecpNeurons[cNeuron1]-
>vecLinksIn.size();lnk3++)
                                {
                                        //update weights
                                        m_vecpNeurons[cNeuron1]->vecLinksIn[lnk3].dWeight -=
m_vecpNeurons[cNeuron1]->dLearningRate*m_vecpNeurons[cNeuron1]-
>dError*m_vecpNeurons[cNeuron1]->vecLinksIn[lnk3].pIn->dOutput;
```

```
                                    }//end of for loop for updating weights


                            }//end of if loop for hidden neurons
                                    cNeuron1--;

                    }//end of while


                    /*comment-end here for other error propagation */


                    //errorVector1.clear();
                    //errorVector2.clear();
}//end of i_iter loop


        int cNeuron2=0;
        int iNo_output_Neurons=0;
        double dAvgError=0.0;
        while(cNeuron2<m_vecpNeurons.size())
                        {
                                if (m_vecpNeurons[cNeuron2]->NeuronType == output)
                                {
                                        iNo_output_Neurons++;
                                        dAvgError+=m_vecpNeurons[cNeuron2]->dError;
                                }
                                cNeuron2++;
                        }//end of while

        dAvgError=dAvgError/(double)iNo_output_Neurons;

return(dAvgError);

}//end of offline training function


// A recursive function that finds the error for hidden neurons
void  CNeuralNet::hiddenneuronerror(SNeuron* hiddenneuron)
{
        if(hiddenneuron->vecLinksOut.size()>0)
        {
                for(int lnk2=0;lnk2< hiddenneuron->vecLinksOut.size();lnk2++)
                {
                        if(hiddenneuron->vecLinksOut[lnk2].pOut->iErrorStatus==0)
                        {
                                hiddenneuron->dError+=hiddenneuron->dOutput* (1 - hiddenneuron-
>dOutput)* hiddenneuron->vecLinksOut[lnk2].pOut->dError *  hiddenneuron-
>vecLinksOut[lnk2].dWeight;
                                if(hiddenneuron->iErrorStatus>0)
                                hiddenneuron->iErrorStatus--;//we have calculated error from 1
output neuron so update the status
                                else break;

                        }//end of if errorstatus=0
                        else if(hiddenneuron->vecLinksOut[lnk2].pOut->iErrorStatus>0)
                        {
                                if(!hiddenneuron->vecLinksOut[lnk2].bRecurrent)
                                {
                                        hiddenneuronerror(hiddenneuron->vecLinksOut[lnk2].pOut);
                                        //MessageBox(m_hwndMain, "Wrong amount of NN inputs!",
"Error", MB_OK);
                                }
                                else
```

```
                                        {
                                                continue;
                                        }
                                }//end of else if i.e. error status is not zero
                                else
                                {
                                        hiddenneuron->vecLinksOut[lnk2].pOut->iErrorStatus=0;
                                        continue;
                                }

                        }//end of for lnk2
                }
}//end of  hiddenneuronerror function


//----------------------------------Update--------------------------------
//        takes a list of doubles as inputs into the network then steps through
//  the neurons calculating each neurons next output.
//
//        finally returns a std::vector of doubles as the output from the net.
//------------------------------------------------------------------------
vector<double> CNeuralNet::Update(const vector<double> &inputs,
                          const run_type        type,
                                                                      const int            iTicks)
{
 //create a vector to put the outputs into
 vector<double> outputs;
 vector<double> DesiredOutputs;
 double p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10;


 //if the mode is snapshot then we require all the neurons to be
 //iterated through as many times as the network is deep. If the
 //mode is set to active the method can return an output after
 //just one iteration
 int FlushCount = 0;
 bool flag_output1=false;

            p0= dBpParam1;
                                    p1= dBpParam2;
                                    p2= dBpParam3;
                                    p3= dBpParam4;
                                    p4= dBpParam5;
                                    p5= dBpParam6;
                                    p6= dBpParam7;
                                    p7= dBpParam8;
                                    p8= dBpParam9;
                                    p9= dBpParam10;
                                    p10= dBpParam11;

 if (type == snapshot)
 {
  FlushCount = m_iDepth ;
 }
 else
 {
  FlushCount = 1;
 }

 //iterate through the network FlushCount times
 for (int i=0; i<m_iDepth; ++i)
 {
  //clear the output vector
  outputs.clear();
 // DesiredOutputs.clear();
```

```
    //this is an index into the current neuron
    int cNeuron = 0;

    //first set the outputs of the 'input' neurons to be equal
    //to the values passed into the function in inputs
    while (m_vecpNeurons[cNeuron]->NeuronType == input)
    {
      m_vecpNeurons[cNeuron]->dOutput = inputs[cNeuron];

      ++cNeuron;
    }//end of input while loop

    //set the output of the bias to 1
    m_vecpNeurons[cNeuron++]->dOutput = 1;


          //DesiredOutputs.push_back(0.9789);
          //DesiredOutputs.push_back(0.9897);


    //then we step through the network a neuron at a time
    while (cNeuron < m_vecpNeurons.size())
    {
      //this will hold the sum of all the inputs x weights
      double sum = 0;

      //sum this neuron's inputs by iterating through all the links into
      //the neuron
      for (int lnk=0; lnk<m_vecpNeurons[cNeuron]->vecLinksIn.size(); ++lnk)
      {
        //get this link's weight
        double Weight = m_vecpNeurons[cNeuron]->vecLinksIn[lnk].dWeight;

        //get the output from the neuron this link is coming from
        double NeuronOutput =
        m_vecpNeurons[cNeuron]->vecLinksIn[lnk].pIn->dOutput;

        //add to sum
        sum += Weight * NeuronOutput;
      }//end of for loop

      //now put the sum through the activation function and assign the
      //value to this neuron's output
      m_vecpNeurons[cNeuron]->dOutput =
      Sigmoid(sum, m_vecpNeurons[cNeuron]->dActivationResponse);
            if (m_vecpNeurons[cNeuron]->NeuronType == output)
                  {

                                          outputs.push_back(m_vecpNeurons[cNeuron]->dOutput);

                  }//end of if output loop

                  //next neuron
                  ++cNeuron;
                }//end of while loop

//-----------------------------------------------
/* comment-begin for no online learning */
//--------------------------------------------------

          if(CParams::iOnlyGAs==0)
          {
          //the following sets error status for output neurons to zero
                  cNeuron=0;
                  int iOutputindex=0;
                  bool flag_output=false;
```

```
		DesiredOutputs.clear();

		if(CParams::iGlobalOnline==0)
		{
			DesiredOutputs = similaritymeasure(inputs);//If Local Online then get desired
o/ps from the training set using filter function
		}
		else
		{
			//	MessageBox(NULL, "in global desired", "Error", 0);

			//If Global Online then we use heuristic of fitness parameters. Here we supply
highest speed possible as desired outputs
			DesiredOutputs.push_back(0.9789);
			DesiredOutputs.push_back(0.9897);
		}

		if(CParams::iGlobalOnline==0)
		{
			//if speed value is less then teach minesweepers to spped up
			if(DesiredOutputs[0]<0.75) DesiredOutputs[0]=DesiredOutputs[0]+0.15;
			if(DesiredOutputs[1]<0.75) DesiredOutputs[1]=DesiredOutputs[1]+0.15;
		}

		if(DesiredOutputs.size()==0)
				MessageBox(NULL, "Error Desired opsize=0!", "Error", 0);

		//the following will set error status and desired outputs for output neurons
		while(cNeuron<m_vecpNeurons.size())
		{
				if (m_vecpNeurons[cNeuron]->NeuronType == output)
				{
						m_vecpNeurons[cNeuron]->iErrorStatus=0;

					if(flag_output==false)//then it is 1st output neuron
					{
						m_vecpNeurons[cNeuron]-
>dDesiredOutput=DesiredOutputs[0];
						flag_output=true;//set the flag
					}
					else //it is 2nd output neuron
					{
						m_vecpNeurons[cNeuron]-
>dDesiredOutput=DesiredOutputs[1];
						flag_output=false;//reset the flag
					}//end of else


				}				//end of outside if loop
				if(m_vecpNeurons[cNeuron]->NeuronType == hidden)
				{
					m_vecpNeurons[cNeuron]-
>iErrorStatus=m_vecpNeurons[cNeuron]->vecLinksOut.size();
				}

				//next neuron
				++cNeuron;

		}//end of error and desired op setting WHILE loop


		cNeuron=0;
		int iLastinputneuron=0;
		while(m_vecpNeurons[cNeuron]->NeuronType == input)
		{
			iLastinputneuron++;
```

```
                    cNeuron++;
            }
            //iLastinputneuron++;//to include bias neuron
            if(iLastinputneuron==10)
                            MessageBox(NULL, "10 input neurons", "Anil", 0);

            m_vecpNeurons[iLastinputneuron]->dOutput = 1;


            //error propagation routine
            cNeuron=m_vecpNeurons.size()-1;

            while(cNeuron>iLastinputneuron)
            {
                    //output neuron error & weight adjustment
                    if(m_vecpNeurons[cNeuron]->NeuronType == output)
                    {

                            m_vecpNeurons[cNeuron]->dError=(m_vecpNeurons[cNeuron]-
>dDesiredOutput-m_vecpNeurons[cNeuron]->dOutput)* m_vecpNeurons[cNeuron]->dOutput*(1-
m_vecpNeurons[cNeuron]->dOutput);
                            for(int lnk1=0;lnk1< m_vecpNeurons[cNeuron]-
>vecLinksIn.size();lnk1++)
                            {
                                    double wij1,aj1;
                                    wij1=m_vecpNeurons[cNeuron]->vecLinksIn[lnk1].dWeight;
                                    aj1=m_vecpNeurons[cNeuron]->dLearningRate;
                                    if(CParams::iRuleEvolution==0)
                                    {

                                            //          m_vecpNeurons[cNeuron]-
>vecLinksIn[lnk1].dWeight -= (1/iTicks)*0.25*m_vecpNeurons[cNeuron]-
>dLearningRate*m_vecpNeurons[cNeuron]->dError*m_vecpNeurons[cNeuron]->vecLinksIn[lnk1].pIn-
>dOutput;
                                            m_vecpNeurons[cNeuron]-
>vecLinksIn[lnk1].dWeight += (1/iTicks)*m_vecpNeurons[cNeuron]-
>dLearningRate*m_vecpNeurons[cNeuron]->dError*m_vecpNeurons[cNeuron]->vecLinksIn[lnk1].pIn-
>dOutput;
                                            //m_vecpNeurons[cNeuron]-
>vecLinksIn[lnk1].dWeight += (1/iTicks)*0.25*m_vecpNeurons[cNeuron]-
>dError*m_vecpNeurons[cNeuron]->vecLinksIn[lnk1].pIn->dOutput;
                                    }
                                    else
                                    {
                                            m_vecpNeurons[cNeuron]-
>vecLinksIn[lnk1].dWeight -= p0*(p1*wij1-p2*aj1*m_vecpNeurons[cNeuron]->dError-
p3*wij1*aj1+p4*m_vecpNeurons[cNeuron]->dError+p5*aj1*m_vecpNeurons[cNeuron]->dOutput);
                                    }
                                    //m_vecpNeurons[cNeuron]-
>dLearningRate+=m_vecpNeurons[cNeuron]->dLearningRate*m_vecpNeurons[cNeuron]->dError;

                            }//end of for loop

                    }//end of if loop for output neurons

            if(m_vecpNeurons[cNeuron]->NeuronType == hidden)
                    {
                            hiddenneuronerror(m_vecpNeurons[cNeuron]);
                            for(int lnk3=0;lnk3<m_vecpNeurons[cNeuron]-
>vecLinksIn.size();lnk3++)
                            {
                                    double wij,aj;
                                    wij=m_vecpNeurons[cNeuron]->vecLinksIn[lnk3].dWeight;
                                    aj=m_vecpNeurons[cNeuron]->dLearningRate;
                                    //update weights, if not bias
                                    if(CParams::iRuleEvolution==0)
```

```
                                        {
                                                //m_vecpNeurons[cNeuron]-
>vecLinksIn[lnk3].dWeight -= (1/iTicks)*0.25*m_vecpNeurons[cNeuron]-
>dLearningRate*m_vecpNeurons[cNeuron]->dError*m_vecpNeurons[cNeuron]->vecLinksIn[lnk3].pIn-
>dOutput;
                                            m_vecpNeurons[cNeuron]->vecLinksIn[lnk3].dWeight +=
(1/iTicks)*m_vecpNeurons[cNeuron]->dLearningRate*m_vecpNeurons[cNeuron]-
>dError*m_vecpNeurons[cNeuron]->vecLinksIn[lnk3].pIn->dOutput;
                                                //m_vecpNeurons[cNeuron]-
>vecLinksIn[lnk3].dWeight += (1/iTicks)*0.25*m_vecpNeurons[cNeuron]-
>dError*m_vecpNeurons[cNeuron]->vecLinksIn[lnk3].pIn->dOutput;
                                        }
                                        else
                                        {
                                                m_vecpNeurons[cNeuron]-
>vecLinksIn[lnk3].dWeight -= p0*0.5*(p1*wij-p2*aj*m_vecpNeurons[cNeuron]->dError-
p3*wij*aj+p4*m_vecpNeurons[cNeuron]->dError+p5*aj*m_vecpNeurons[cNeuron]->dOutput);
                                        }
                                        //m_vecpNeurons[cNeuron]-
>dLearningRate+=m_vecpNeurons[cNeuron]->dLearningRate*m_vecpNeurons[cNeuron]->dError;
                                   }//end of for loop for updating weights
                              }//end of if loop for hidden neurons

                              cNeuron--;//next iteration

                         }//end of while loop



                         //set the output of the bias to 1
       m_vecpNeurons[iLastinputneuron]->dOutput = 1;

          }//end of only GA If loop
          //------------------------------------------------
          /* comment-end for no Online learning*/
          //------------------------------------------------




   }//next iteration through the network




   //the network needs to be flushed if this type of update is performed
   //otherwise it is possible for dependencies to be built on the order
   //the training data is presented

   if (type == snapshot)
   {
     for (int n=0; n<m_vecpNeurons.size(); ++n)
     {
       m_vecpNeurons[n]->dOutput = 0;
     }
   }

             //return the outputs
return outputs;

}

//Find the similar i-o pair in the training set
vector<double> CNeuralNet::similaritymeasure(const vector<double> &input)
{
        char *szFileName2="io_training5.txt";
        //fstream grab2("io_training.txt", fstream::in | fstream::out | fstream::app);

        fstream grab2(szFileName2);
        vector<double> trainingInputs;
        vector<double> targetOutputs;
        vector<double> desiredOutputs;
        //double trainingInputs[11];
        //double targetOuputs[2];
        vector<double> dSum;
        int iTrainingSize=250;
        double dMin=1000.0;
        int iMinIndex;
    double dTmpvar;
        int i_here;
        double dtmpsum;

        for (i_here=0;i_here<iTrainingSize;i_here++)
        {
                trainingInputs.clear();
                targetOutputs.clear();
                //desiredOutputs.clear();
                dtmpsum=0;
                //dMin=100.0;
                //Read input from file

                for(int j_here=0;j_here<10;j_here++)
                {
                        dTmpvar=dIop[i_here][j_here];
                        trainingInputs.push_back(dTmpvar);
                }

                dTmpvar=dIop[i_here][11];
                targetOutputs.push_back(dTmpvar);
                dTmpvar=dIop[i_here][12];
                targetOutputs.push_back(dTmpvar);

                //Find the distance between the inputs
                for(int k_here=0;k_here<trainingInputs.size();k_here++)
                {
                        double dDiff=input[k_here]-trainingInputs[k_here];
                        dtmpsum+=fabs(dDiff);
                }

                dSum.push_back(dtmpsum);

                //Update the minimum distance and store the corresponding training input
                if(dMin>dtmpsum)
                {
                        desiredOutputs.clear();
                        dMin=dtmpsum;
                        iMinIndex=i_here;
                        desiredOutputs=targetOutputs;
                }//end of if

        }//end of i_here for loop

        vector<double> output;//=desiredOutputs;
        output=desiredOutputs;
        //output[1]=desiredOutputs[1];
        return(output);
}//end of function


void CNeuralNet::MutateLearningParameters(double mut_rate,double MaxPertubation)
```

```
{
   if (RandFloat() < mut_rate)
  {
               /*if(RandFloat()>0.9)
               {
                      dBpParam1 += -1*((rand()%6)+1);
               }
               else*/
               dBpParam1    += RandomClamped()*MaxPertubation*0.025;
               dBpParam2    += RandomClamped()*MaxPertubation*0.025;
               dBpParam3    += RandomClamped()*MaxPertubation*0.025;
               dBpParam4    += RandomClamped()*MaxPertubation*0.025;
               dBpParam5    += RandomClamped()*MaxPertubation*0.025;
               dBpParam6    += RandomClamped()*MaxPertubation*0.025;
               dBpParam7    += RandomClamped()*MaxPertubation*0.025;
               dBpParam8    += RandomClamped()*MaxPertubation*0.025;
               dBpParam9    += RandomClamped()*MaxPertubation*0.025;
               dBpParam10   += RandomClamped()*MaxPertubation*0.025;
               dBpParam11   += RandomClamped()*MaxPertubation*0.025;

               //dBpParam1    += RandomClamped()*MaxPertubation;
  }
}



//----------------------------- TidyXSplits -----------------------------
//
// This is a fix to prevent neurons overlapping when they are displayed
//-----------------------------------------------------------------------
void TidyXSplits(vector<SNeuron*> &neurons)
{
 //stores the index of any neurons with identical splitY values
 vector<int>    SameLevelNeurons;

 //stores all the splitY values already checked
 vector<double> DepthsChecked;


 //for each neuron find all neurons of identical ySplit level
 for (int n=0; n<neurons.size(); ++n)
 {
  double ThisDepth = neurons[n]->dSplitY;

  //check to see if we have already adjusted the neurons at this depth
  bool bAlreadyChecked = false;

  for (int i=0; i<DepthsChecked.size(); ++i)
  {
   if (DepthsChecked[i] == ThisDepth)
   {
    bAlreadyChecked = true;

    break;
   }
  }

  //add this depth to the depths checked.
  DepthsChecked.push_back(ThisDepth);

  //if this depth has not already been adjusted
  if (!bAlreadyChecked)
  {
   //clear this storage and add the neuron's index we are checking
```

```
      SameLevelNeurons.clear();
      SameLevelNeurons.push_back(n);

      //find all the neurons with this splitY depth
      for (int i=n+1; i<neurons.size(); ++i)
      {
        if (neurons[i]->dSplitY == ThisDepth)
        {
          //add the index to this neuron
          SameLevelNeurons.push_back(i);
        }
      }

      //calculate the distance between each neuron
      double slice = 1.0/(SameLevelNeurons.size()+1);


      //separate all neurons at this level
      for (i=0; i<SameLevelNeurons.size(); ++i)
      {
        int idx = SameLevelNeurons[i];

        neurons[idx]->dSplitX = (i+1) * slice;
      }
    }

  }//next neuron to check

}
//----------------------------- DrawNet ---------------------------------
//
//  creates a representation of the ANN on a device context
//
//-----------------------------------------------------------------------
void CNeuralNet::DrawNet(HDC &surface, int Left, int Right, int Top, int Bottom)
{
  //the border width
  const int border = 10;

  //max line thickness
  const int MaxThickness = 5;

  TidyXSplits(m_vecpNeurons);

  //go through the neurons and assign x/y coords
  int spanX = Right - Left;
  int spanY = Top - Bottom - (2*border);

  for (int cNeuron=0; cNeuron<m_vecpNeurons.size(); ++cNeuron)
  {
    m_vecpNeurons[cNeuron]->iPosX = Left + spanX*m_vecpNeurons[cNeuron]->dSplitX;
    m_vecpNeurons[cNeuron]->iPosY = (Top - border) - (spanY * m_vecpNeurons[cNeuron]->dSplitY);
  }

  //create some pens and brushes to draw with
  HPEN GreyPen  = CreatePen(PS_SOLID, 1, RGB(200, 200, 200));
  HPEN RedPen   = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
  HPEN GreenPen = CreatePen(PS_SOLID, 1, RGB(0, 200, 0));
  HPEN OldPen   = NULL;

  //create a solid brush
  HBRUSH RedBrush = CreateSolidBrush(RGB(255, 0, 0));
  HBRUSH OldBrush = NULL;

  OldPen =   (HPEN)  SelectObject(surface, RedPen);
  OldBrush = (HBRUSH)SelectObject(surface, GetStockObject(HOLLOW_BRUSH));
```

```
//radius of neurons
int radNeuron = spanX/60;
int radLink = radNeuron * 1.5;

//now we have an X,Y pos for every neuron we can get on with the
//drawing. First step through each neuron in the network and draw
//the links
for (cNeuron=0; cNeuron<m_vecpNeurons.size(); ++cNeuron)
{
  //grab this neurons position as the start position of each
  //connection
  int StartX = m_vecpNeurons[cNeuron]->iPosX;
  int StartY = m_vecpNeurons[cNeuron]->iPosY;

  //is this a bias neuron? If so, draw the link in green
  bool bBias = false;

  if (m_vecpNeurons[cNeuron]->NeuronType == bias)
  {
    bBias = true;
  }

  //now iterate through each outgoing link to grab the end points
  for (int cLnk=0; cLnk<m_vecpNeurons[cNeuron]->vecLinksOut.size(); ++ cLnk)
  {
    int EndX = m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].pOut->iPosX;
    int EndY = m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].pOut->iPosY;

    //if link is forward draw a straight line
    if( (!m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].bRecurrent) && !bBias)
    {
      int thickness = (int)(fabs(m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].dWeight));

      Clamp(thickness, 0, MaxThickness);

      HPEN Pen;

      //create a yellow pen for inhibitory weights
      if (m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].dWeight <= 0)
      {
        Pen  = CreatePen(PS_SOLID, thickness, RGB(240, 230, 170));
      }

      //grey for excitory
      else
      {
        Pen  = CreatePen(PS_SOLID, thickness, RGB(200, 200, 200));
      }

      HPEN tempPen = (HPEN)SelectObject(surface, Pen);

      //draw the link
      MoveToEx(surface, StartX, StartY, NULL);
      LineTo(surface, EndX, EndY);

      SelectObject(surface, tempPen);

      DeleteObject(Pen);
    }

    else if( (!m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].bRecurrent) && bBias)
    {
      SelectObject(surface, GreenPen);
  //draw the link
  MoveToEx(surface, StartX, StartY, NULL);
  LineTo(surface, EndX, EndY);
}

//recurrent link draw in red
else
{
  if ((StartX == EndX) && (StartY == EndY))
  {

    int thickness = (int)(fabs(m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].dWeight));

    Clamp(thickness, 0, MaxThickness);

    HPEN Pen;

    //blue for inhibitory
    if (m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].dWeight <= 0)
    {
      Pen  = CreatePen(PS_SOLID, thickness, RGB(0,0,255));
    }

    //red for excitory
    else
    {
      Pen  = CreatePen(PS_SOLID, thickness, RGB(255, 0, 0));
    }

    HPEN tempPen = (HPEN)SelectObject(surface, Pen);

    //we have a recursive link to the same neuron draw an ellipse
    int x = m_vecpNeurons[cNeuron]->iPosX ;
    int y = m_vecpNeurons[cNeuron]->iPosY - (1.5 * radNeuron);

    Ellipse(surface, x-radLink, y-radLink, x+radLink, y+radLink);

    SelectObject(surface, tempPen);

    DeleteObject(Pen);
  }

  else
  {
    int thickness = (int)(fabs(m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].dWeight));

    Clamp(thickness, 0, MaxThickness);

    HPEN Pen;

    //blue for inhibitory
    if (m_vecpNeurons[cNeuron]->vecLinksOut[cLnk].dWeight <= 0)
    {
      Pen  = CreatePen(PS_SOLID, thickness, RGB(0,0,255));
    }

    //red for excitory
    else
    {
      Pen  = CreatePen(PS_SOLID, thickness, RGB(255, 0, 0));
    }


    HPEN tempPen = (HPEN)SelectObject(surface, Pen);

    //draw the link
```

```
        MoveToEx(surface, StartX, StartY, NULL);
        LineTo(surface, EndX, EndY);

        SelectObject(surface, tempPen);

        DeleteObject(Pen);
      }
    }

  }
 }

 //now draw the neurons and their IDs
 SelectObject(surface, RedBrush);
 SelectObject(surface, GetStockObject(BLACK_PEN));

 for (cNeuron=0; cNeuron<m_vecpNeurons.size(); ++cNeuron)
 {
  int x = m_vecpNeurons[cNeuron]->iPosX;
  int y = m_vecpNeurons[cNeuron]->iPosY;

  //display the neuron
  Ellipse(surface, x-radNeuron, y-radNeuron, x+radNeuron, y+radNeuron);
 }

 //cleanup
 SelectObject(surface, OldPen);
 SelectObject(surface, OldBrush);

 DeleteObject(RedPen);
 DeleteObject(GreyPen);
 DeleteObject(GreenPen);
 DeleteObject(OldPen);
 DeleteObject(RedBrush);
 DeleteObject(OldBrush);
}
```

```
#ifndef COLLISION_H
#define COLLISION_H

#include "utils.h"
#include <math.h>


//--------------------2LinesIntersection2D-------------------------
// Authors:
// Created by  Mat Buckland 2002
// Modified by Anil kumar Enumulapally  2004
//                                  Anil kumar Enumulapally  2005
//
//         Given 2 lines in 2D space AB, CD this returns true if an
//         intersection occurs and sets dist to the distance the intersection
// occurs along AB
//
//-----------------------------------------------------------------
inline bool LineIntersection2D(const SPoint A,
                          const SPoint B,
                          const SPoint C,
                          const SPoint D,
                          double &dist)
{
 //first test against the bounding boxes of the lines
 if ( (((A.y > D.y) && (B.y > D.y)) && ((A.y > C.y) && (B.y > C.y))) ||
     (((B.y < C.y) && (A.y < C.y)) && ((B.y < D.y) && (A.y < D.y))) ||
     (((A.x > D.x) && (B.x > D.x)) && ((A.x > C.x) && (B.x > C.x))) ||
     (((B x < C.x) && (A.x < C.x)) && ((B.x < D.x) && (A.x < D.x))) )
 {
   dist = 0;

   return false;
 }

 double rTop = (A.y-C.y)*(D.x-C.x)-(A.x-C.x)*(D.y-C.y);
         double rBot = (B.x-A.x)*(D.y-C.y)-(B.y-A.y)*(D.x-C.x);

         double sTop = (A.y-C.y)*(B.x-A.x)-(A.x-C.x)*(B.y-A.y);
         double sBot = (B.x-A.x)*(D.y-C.y)-(B.y-A.y)*(D.x-C.x);


 double rTopBot = rTop*rBot;
 double sTopBot = sTop*sBot;

 if ((rTopBot>0) && (rTopBot<rBot*rBot) && (sTopBot>0) && (sTopBot<sBot*sBot))
 {
    dist = rTop/rBot;

    return true;
  }


  else
  {
    dist = 0;

    return false;
  }

}


#endif

#include <windows.h>
#include <time.h>

#include "utils.h"
#include "CController.h"
#include "CTimer.h"
#include "resource.h"
#include "CParams.h"



///////////////////////GLOBALS ////////////////////////////////////

char*                   szApplicationName     = "Anil's New Hybrid Learning Algorithm";
char*                   szWindowClassName     = "sweeper";
char*                   szInfoWindowClassName = "Info Window";


//The controller class for this simulation
CController*     g_pController      = NULL;

CParams   g_Params;

//global handle to the info window
HWND g_hwndInfo = NULL;

//global handle to the main window
HWND g_hwndMain = NULL;

//---------------------------- Cleanup ----------------------------------
//
//        simply cleans up any memory issues when the application exits
//-----------------------------------------------------------------------
void Cleanup()
{
        if (g_pController)

                delete g_pController;
}
//-----------------------------------WinProc-----------------------------
//
//-----------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hwnd,
                                                              UINT msg,
                        WPARAM wparam,
                        LPARAM lparam)
{
        //these hold the dimensions of the client window area
        static int cxClient, cyClient;

        //used to create the back buffer
        static HDC                  hdcBackBuffer;
        static HBITMAP    hBitmap;
        static HBITMAP    hOldBitmap;

        switch(msg)
        {
                case WM_CREATE:
                {
                        //seed the random number generator
                        srand((unsigned) time(NULL));

                        //get the size of the client window
                        RECT rect;
                        GetClientRect(hwnd, &rect);

                        cxClient = rect.right;
                        cyClient = rect.bottom;

                        //setup the controller
                        g_pController = new CController(hwnd, cxClient, cyClient);

                                //create a surface for us to render to(backbuffer)
                        hdcBackBuffer = CreateCompatibleDC(NULL);

                        HDC hdc = GetDC(hwnd);

                        hBitmap = CreateCompatibleBitmap(hdc,

cxClient,

cyClient);
                        ReleaseDC(hwnd, hdc);

                        hOldBitmap = (HBITMAP)SelectObject(hdcBackBuffer, hBitmap);
                }

                break;

                //check key press messages
                case WM_KEYUP:
                {
                        switch(wparam)
                        {

                                case VK_ESCAPE:
                                {
                                        PostQuitMessage(0);
                                }

                                break;

                                case 'F':
                                        {
                                                g_pController->FastRenderToggle();
                                        }

                                        break;

    case 'B':
                                        {
                                                g_pController->RenderBestToggle();
                                        }
                                        break;

  case 'R':
   {
     if (g_pController)
     {
       delete g_pController;
     }

     //setup the new controller
                         g_pController = new CController(hwnd, cxClient, cyClient);

     //give the info window's handle to the controller
     g_pController->PassInfoHandle(g_hwndInfo);

     //clear info window
     InvalidateRect(g_hwndInfo, NULL, TRUE);
                         UpdateWindow(g_hwndInfo);
   }

   break;

                /*
  case 'Z':
                      pTimer= SetTimer(10000);
                      break;

       case 'Y':
                      KillTimer(pTimer);
                      break;
                      */

  case '1':
   {
     g_pController->ViewBest(1);
   }

  break;

  case '2':
   {
     g_pController->ViewBest(2);
   }

  break;

  case '3':
   {
     g_pController->ViewBest(3);
   }

  break;

  case '4':
   {
     g_pController->ViewBest(4);
   }

  break;


                      }//end WM_KEYUP switch
                 }
                break;

                //has the user resized the client area?
                case WM_SIZE:
                {
                        cxClient = LOWORD(lparam);
                        cyClient = HIWORD(lparam);
                }

                break;

                case WM_PAINT:
                {
    PAINTSTRUCT ps;

                 BeginPaint(hwnd, &ps);

                        //fill our backbuffer with white
                        BitBlt(hdcBackBuffer,
        0,
        0,
        cxClient,
        cyClient,
        NULL,
        NULL,
        NULL,
        WHITENESS);

                        //render the sweepers
                        g_pController->Render(hdcBackBuffer);

                        //now blit backbuffer to front
                        BitBlt(ps.hdc, 0, 0, cxClient, cyClient, hdcBackBuffer, 0, 0, SRCCOPY);

                        EndPaint(hwnd, &ps);
                }

                break;

                case WM_DESTROY:
                {
                        SelectObject(hdcBackBuffer, hOldBitmap);

                        //clean up our backbuffer objects
                        DeleteDC(hdcBackBuffer);
                        DeleteObject(hBitmap);

    // kill the application, this sends a WM_QUIT message
                        PostQuitMessage(0);
                }

                break;

                default:break;

        }//end switch

        // default msg handler
        return (DefWindowProc(hwnd, msg, wparam, lparam));

}//end WinProc

//-----------------------------------InfoWinProc-----------------------------
//
//--------------------------------------------------------------------------
```

```
LRESULT CALLBACK InfoWindowProc(HWND hwnd,
                                                                        UINT msg,
                  WPARAM wparam,
                  LPARAM lparam)
{
       //these hold the dimensions of the client window area
       static int cxClient, cyClient;

       switch(msg)
       {
              case WM_CREATE:
              {

                     //get the size of the client window
                     RECT rect;
                     GetClientRect(hwnd, &rect);

                     cxClient = rect.right;
                     cyClient = rect.bottom;
              }

              break;

              //has the user resized the client area?
              case WM_SIZE:
              {
                     cxClient = LOWORD(lparam);
                     cyClient = HIWORD(lparam);
              }

              break;

              case WM_PAINT:
              {
   PAINTSTRUCT ps;

               BeginPaint(hwnd, &ps);

   g_pController->RenderNetworks(ps.hdc);

                     EndPaint(hwnd, &ps);
              }

              break;


              default:break;

       }//end switch

       // default msg handler
       return (WindowProc(hwnd, msg, wparam, lparam));

}//end WinProc
//---------------------------------CreateInfoWindow---------------------------
//
//     creates and displays the info window
//
//--------------------------------------------------------------------------
void CreateInfoWindow(HWND hwndParent)
{
       // Create and register the window class
 WNDCLASSEX wcInfo = {sizeof(WNDCLASSEX),
             CS_HREDRAW | CS_VREDRAW,
             InfoWindowProc,
             0,
                0,
                GetModuleHandle(NULL),
                                                NULL,
                                                NULL,
                                                (HBRUSH)(GetStockObject(WHITE_BRUSH)),
                                                NULL,
                                                "Info",
                                                NULL };

 RegisterClassEx( &wcInfo );

        // Create the application's info window
 g_hwndInfo = CreateWindow("Info",
                "ANIL - Previous generation's best four phenotypes",
                                                              WS_OVERLAPPED |
WS_VISIBLE | WS_CAPTION | WS_SYSMENU,
                GetSystemMetrics(SM_CXSCREEN)/2,
                GetSystemMetrics(SM_CYSCREEN)/2 - CParams::WindowHeight/2,
                CParams::InfoWindowWidth,

CParams::InfoWindowHeight,
                                                              hwndParent,
                                                              NULL,
                                                              wcInfo.hInstance,
                                                              NULL );

        // Show the info
        ShowWindow(g_hwndInfo, SW_SHOWDEFAULT);
        UpdateWindow(g_hwndInfo);

 //give the info window's handle to the controller
 g_pController->PassInfoHandle(g_hwndInfo);

        return;
}

//----------------------------------WinMain----------------------------------------
//      Entry point for our windows application
//------------------------------------------------------------------------------------
int WINAPI WinMain(     HINSTANCE hinstance,
                                        HINSTANCE hprevinstance,
                                        LPSTR lpcmdline,
                                        int ncmdshow)
{

        WNDCLASSEX winclass;
        HWND      hwnd;
        MSG                 msg;

 //load in the parameters for the program
 if (!g_Params.Initialize())
 {
  return false;
 }

        // first fill in the window class stucture
        winclass.cbSize       = sizeof(WNDCLASSEX);
        winclass.style                     = CS_HREDRAW | CS_VREDRAW;
        winclass.lpfnWndProc       = WindowProc;
        winclass.cbClsExtra                = 0;
        winclass.cbWndExtra                = 0;
        winclass.hInstance                 = hinstance;
        winclass.hIcon                       = LoadIcon(hinstance, MAKEINTRESOURCE(IDI_ICON1));
        winclass.hCursor             = LoadCursor(NULL, IDC_ARROW);
        winclass.hbrBackground= NULL;
        winclass.lpszMenuName    = NULL;
        winclass.lpszClassName= szWindowClassName;
        winclass.hIconSm      = LoadIcon(hinstance, MAKEINTRESOURCE(IDI_ICON1));


        // register the window class
        if (!RegisterClassEx(&winclass))
        {
                MessageBox(NULL, "Error Registering Class!", "Error", 0);
   return 0;
        }

        // create the window (one that cannot be resized)
        if (!(hwnd = CreateWindowEx(NULL,

                                                                    szWindowClassName,

                                                                    szApplicationName,

                                                                    WS_OVERLAPPED |
WS_VISIBLE | WS_CAPTION | WS_SYSMENU,

GetSystemMetrics(SM_CXSCREEN)/2 - CParams::WindowWidth,
                    GetSystemMetrics(SM_CYSCREEN)/2 - CParams::WindowHeight/2,

                                                                    CParams::WindowWidth,
                    CParams::WindowHeight,
                                                                    NULL,

                                                                    NULL,

                                                                    hinstance,

                                                                    NULL)))
        {
  MessageBox(NULL, "Error Creating Window!", "Error", 0);
                return 0;
        }

 //keep a global record of the window handle
 g_hwndMain = hwnd;

 //create and show the info window
 CreateInfoWindow(hwnd);

 //Show the window
        ShowWindow(hwnd, SW_SHOWDEFAULT );
        UpdateWindow(hwnd);

        //create a timer
        CTimer timer(CParams::iFramesPerSecond);

        //start the timer
        timer.Start();

        // Enter the message loop
        bool bDone = FALSE;

        while(!bDone)
        {

                while( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
                {
                        if( msg.message == WM_QUIT )
                        {
                                // Stop loop if it's a quit message
                                bDone = TRUE;
```

```
				}

				else
				{
					TranslateMessage( &msg );
					DispatchMessage( &msg );
				}
			}

			if (timer.ReadyForNextFrame() || g_pController->FastRender())
			{
			 if(!g_pController->Update())
				{
					//we have a problem, end app
					bDone = TRUE;
				}

				//this will call WM_PAINT which will render our scene
				InvalidateRect(hwnd, NULL, TRUE);
				UpdateWindow(hwnd);
  }

	}//end while


  // Clean up everything and exit the app
  Cleanup();
  UnregisterClass( szWindowClassName, winclass.hInstance );

	return 0;

} // end WinMain
```

```
#ifndef S2DVECTOR_H
#define S2DVECTOR_H

#include <math.h>

/////////////////////////////////////////////////////////////////////
//
//	2D Vector structure and methods
//
/////////////////////////////////////////////////////////////////////
struct SVector2D
{
	double x, y;

	SVector2D(double a = 0, double b = 0):x(a),y(b){}


	//we need some overloaded operators
	SVector2D &operator+=(const SVector2D &rhs)
	{
		x += rhs.x;
		y += rhs.y;

		return *this;
	}

	SVector2D &operator-=(const SVector2D &rhs)
	{
		x -= rhs.x;
		y -= rhs.y;

		return *this;
	}

	SVector2D &operator*=(const double &rhs)
	{
		x *= rhs;
		y *= rhs;

		return *this;
	}

	SVector2D &operator/=(const double &rhs)
	{
		x /= rhs;
		y /= rhs;

		return *this;
```

```
          }
};

//overload the * operator
inline SVector2D operator*(const SVector2D &lhs, double rhs)
{
 SVector2D result(lhs);
 result *= rhs;
 return result;
}

inline SVector2D operator*(double lhs, const SVector2D &rhs)
{
 SVector2D result(rhs);
 result *= lhs;
 return result;
}

//overload the - operator
inline SVector2D operator-(const SVector2D &lhs, const SVector2D &rhs)
{
 SVector2D result(lhs);
 result.x -= rhs.x;
 result.y -= rhs.y;

 return result;
}
//------------------------ Vec2DLength ----------------------------
//
//        returns the length of a 2D vector
//----------------------------------------------------------------
inline double Vec2DLength(const SVector2D &v)
{
        return sqrt(v.x * v.x + v.y * v.y);
}

//------------------------ Vec2DNormalize ----------------------------
//
//        normalizes a 2D Vector
//----------------------------------------------------------------
inline void Vec2DNormalize(SVector2D &v)
{
        double vector_length = Vec2DLength(v);

        v.x = v.x / vector_length;
        v.y = v.y / vector_length;
}

//------------------------ Vec2DDot -------------------------
//
//        calculates the dot product
//----------------------------------------------------------------
inline double Vec2DDot(SVector2D &v1, SVector2D &v2)
{
        return v1.x*v2.x + v1.y*v2.y;
}

//----------------------- Vec2DSign -------------------------------
//
// returns positive if v2 is clockwise of v1, minus if anticlockwise
//----------------------------------------------------------------
inline int Vec2DSign(SVector2D &v1, SVector2D &v2)
{
 if (v1.y*v2.x > v1.x*v2.y)
 {
  return 1;
  }
  else
  {
    return -1;
  }
}

#endif
```

```
#ifndef UTILS_H
#define UTILS_H

#include <stdlib.h>
#include <math.h>
#include <sstream>
#include <string>
#include <iostream>
#include <vector>

using namespace std;

//----------------------------------------------------------------------------
//          UTIL.H
//         some random number functions.
//----------------------------------------------------------------------------

//returns a random integer between x and y
inline int   RandInt(int x,int y) {return rand()%(y-x+1)+x;}

//returns a random float between zero and 1
inline double RandFloat()                  {return (rand())/(RAND_MAX+1.0);}

//returns a random bool
inline bool   RandBool()
{
          if (RandInt(0,1)) return true;

          else return false;
}

//returns a random float in the range -1 < n < 1
inline double RandomClamped()       {return RandFloat() - RandFloat();}


//-----------------------------------------------------------------------
//
//         some handy little functions
//-----------------------------------------------------------------------

//converts an integer to a string
inline string itos(int arg)
{
   ostringstream buffer;

          //send the int to the ostringstream
   buffer << arg;
```

```
        //capture the string
    return buffer.str();
}



//converts a float to a string
inline string ftos(float arg)
{
    ostringstream buffer;

        //send the int to the ostringstream
    buffer << arg;

        //capture the string
    return buffer.str();
}

//clamps the first argument between the second two
inline void Clamp(double &arg, double min, double max)
{
        if (arg < min)
        {
                arg = min;
        }

        if (arg > max)
        {
                arg = max;
        }
}

inline void Clamp(int &arg, int min, int max)
{
        if (arg < min)
        {
                arg = min;
        }

        if (arg > max)
        {
                arg = max;
        }
}

//rounds a double up or down depending on its value
inline int Rounded(double val)
{
  int    integral = (int)val;
  double mantissa = val - integral;

  if (mantissa < 0.5)
  {
    return integral;
  }

  else
  {
    return integral + 1;
  }
}

//rounds a double up or down depending on whether its
//mantissa is higher or lower than offset
inline int RoundUnderOffset(double val, double offset)
{
  int    integral = (int)val;
  double mantissa = val - integral;

  if (mantissa < offset)
  {
    return integral;
  }

  else
  {
    return integral + 1;
  }
}




/////////////////////////////////////////////////////////////////////
//
//          Point structure
//
/////////////////////////////////////////////////////////////////////
struct SPoint
{
          float x, y;

          SPoint(){}
          SPoint(float a, float b):x(a),y(b){}
};



#endif
```

```
//------------------------------------------------
//Global Parameter file
//parameter.ini
//------------------------------------------------


iFramesPerSecond 60
dMaxTurnRate 0.1
iSweeperScale 5
iNumSensors 5
dSensorRange 25
iNumSweepers 25
iNumTicks 600
dCellSize 20
iNumAddLinkAttempts 10
dSurvivalRate 0.2
iNumGensAllowedNoImprovement 10
iMaxPermittedNeurons 100
dChanceAddLink 0.07
dChanceAddNode 0.03
dChanceAddRecurrentLink 0.03
dMutationRate 0.5
dMaxWeightPerturbation 0.5
dProbabilityWeightReplaced 0.1
dActivationMutationRate 0.5
dMaxActivationPerturbation 0.8
dCompatibilityThreshold 0.25
iOldAgeThreshold 50
dOldAgePenalty 0.9
dYoungFitnessBonus 1.3
iYoungBonusAgeThreshhold 10
dCrossoverRate 0.7
dLearningRate  0.05
dLearningParameter1 0.5
dLearningParameter2 0.02
dLearningParameter3 0.02
dLearningParameter4 0.02
dLearningParameter5 0.02
dLearningParameter6 0.02
iOfflineTraining 0
iGlobalOnline 0
iRuleEvolution 1
iOnlyGAs 0
```

REFERENCES

[1] Adrian Agogino, Kenneth Stanley, and Risto Miikkulainen (2000). Online Interactive Neuro-Evolution, *Neural Processing Letters* 11:29-37, 2000.

[2] Amarı, S. (1967). *A Theory of Adaptive Pattern Classifiers*. IEEE Transactions on Electronic Computers, Vol. EC-16, No. 3, pp. 299-307

[3] Anil Kumar Enumulapally, Ligguo Bu, and Khosrow Kaıkhah (2004). Backpropagatıon: In Search of Performance Parameters, *WSEAS Transactions on Systems, Issue 2, Vol. 3, April 2004.*

[4] Antonia J. Jones (1993). Genetic algorıthms and their applications to the desıgn of neural networks. *Neural Computing & Applications*, **1**(1):32-45.

[5] A. Likartsis, I.Vlachavas, and L.H.Tsoukalas (1997). A New Hybrid Neural-Genetic Methodology for Improving Learning. *Proc. of 9th International Conference on Tools with Artificial Intelligence(ICTAI '97).*

[6] Baldwin, Mark J(1896). A New Factor in Evolutıon. *Adaptive Individuals in the evolving Populations: Models and Algorithms*. Addison-Wesley, Reading, MA.

[7] Bottou, L. (1998). Onlıne Algorıthms and Stochastic Approximatıons, 9-42. *In Saad, D., editor, Online Learning in Neural Networks*. Cambridge Unıversity Press, Cambridge, UK.

[8] Chalmers, D. J. (1990) The evolutıon of learning: An experiment ın genetıc connectionism. In D. S. Touretzky, editor, *Proceedings of the 1990 Connectionist Models Summer School*, 81-90. San Mateo, CA: Morgan Kaufmann.

[9] Dara Curran, Colm O'Riordan (2002). Applying Evolutionary Computation to Designing Neural Networks: A Study of the State of the Art.

[10] Darpa Neural Network Study, (1998). *AFCEA International Press.*

[11] Davıd B. Fogel, Evolutionary computatıon: toward a new philosophy of machıne intelligence, IEEE Press, Pıscataway, NJ, 1995

[12] David E. Goldberg (1989). Genetic Algorithms in Search, Optimizatıon, and Machıne Learning.

[13] Haykın. S, *Neural Networks: a Comprehensive Foundation*, 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

[14] Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press: Ann Arbor, MI.

[15] Hollestein R. B. (1979) Artificial genetic adap-tation in computer control systems. *PhD dissertation, University of Michigan.*

[16] Jurgen Branke (1995). Evolutionary Algorithms for Neural Network design and Training.

[17] Kenneth O. Stanley and Risto Miikkulainen (2002). Efficient Evolution Of Neural Network Topologies, *Proceedings of the 2002 Congress on Evolutionary Computation* (CEC '02). Piscataway, NJ: IEEE, 2002.

[18] Kenneth O. Stanley and Risto Miikkulainen (2002). Efficient Evolution Of Neural Network Topologies, *Proceedings of the 2002 Congress on Evolutionary Computation* (CEC '02) Piscataway, NJ: IEEE, 2002.

[19] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen (2003). Evolving Adaptive Neural Networks with and Without Adaptive Synapses, To appear in *Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC-2003).*

[20] Kenneth O. Stanley and Risto Miikkulainen (2002). Evolving Neural Networks Through Augmenting Topologies, *Evolutionary Computation* 10(2):99-127, 2002.

[21] Kim W.C. Ku, M.W.Mak, and W.C.Siu (2003). Approaches to Combining Local and Evolutionary Search for Training Neural Networks: A Review and Some New Results.

[22] Kitano, H. (1990). Empirical studies on the speed of convergence of neural network training using genetic algorithms. Proc. of the Eighth National Conf. on Artificial Intelligence.

[23] Klaus-Robert Müller, Andreas Ziehe, Noboru Murata, Shun-ichi Amari (1998). On-line Learning in Switching and Drifting Environments with Application to Blind Source Separation.

[24] Magoulas,G.D., Plagianakos,V.P., and Vrahatis,M.N., Hybrid methods using evolutionary algorithms for on-line training, in *Proceedings of the INNS-IEEE International Joint Conference on Neural Networks*, Washington DC, 14-19 July 2001, USA.

[25] Mat Buckland (2002). AI Techniques for Game Programming, Premier press inc.

[26] Mehrotra,K.,Mohan,C.K, and Ranka,S. (2000). Elements of Artificial Neural Networks. *The MIT press. Cambridge*, Massachusetts

[27] Nikola Kasabov (2003). Evolving connectionist systems: methods and applications in *bioinformatics, brain study and intelligent machines.* London ; New York : Springer publications.

[28] Nolfi, S., Elman, J. L., & Parisi, D. (1990). Learning and evolution in neural networks. *CRL Techn. Rep. 9019.* Center for Research in Language, University of California, San Diego.

[29] Nolfi, S., & Parisi, D. (1991). Growing neural networks. Techn. Rep. PCIA{91{15 *Department of Cognitive Processes and Artificial Intelligence*, C.N.R. Rome, Italy.

[30] Parisi, D., Nolfi, S., and Cecconi, F. (1992). Learning, behavior, and evolution *In Proceedings of the First European Conference on Artificial Life,* Cambridge, MA, MIT Press/Bradford Books.

[31] Philipp Köhn (1996). Genetic Encoding Strategies for Neural Networks, *Proceedings, Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Granada, Spain, Volume II, pages 947-950.

[32] Rıchard S Sutton and Steven D Whıtehead (1993) Onlıne Learnıng wıth Random Representatıons, *Proceedings of the Tenth Int. Conf on Machine Learnıng pp 314-321* Morgan Kaufmann

[33] Robert M French and Adam Messınger (1994) Genes, phenes and the Baldwın Effect Learnıng and Evolutıon ın a sımulated populatıon, *Artificıal Lıfe IV, 277-282*

[34] Rumelhart D E , Hınton G E, and Wıllıams R J Learnıng ınternal representatıons by error propagatıon In *Parallel Distributed Processing*, volume 1, pages 318-362 MIT Press, 1986

[35] Saad,D , edıtor (1998) On-lıne Learnıng ın Neural Networks, *Publıcations of Newton Institute, Cambrıdge Unıversıty Press,* Cambrıdge, UK

[36] Sompolınsky H, Barkaı N and Seung HS (1995) *On-line Learnıng of Dichotomıes Algorıthms and Learnıng Curves* In Advances ın Neural Informatıon Processıng Systems 7 Cowan J D, Tesauro G, and Alspector J, Eds

[37] Tom Mıtchell (1997) Machıne Learnıng, McGraw Hıll publıcatıon

[38] Talıb S Hussaın (1997) Methods of Combınıng Neural Networks and Genetıc Algorıthms Queen's Unıversıty

[39] V Petrıdıs, S Kazarlıs, A Papaıkonomu and A Fılelıs (1992) A Hybrıd Genetıc algorıthm for traınıng Neural networks *Artıficial Neural Networks, 2, 953-956*

[40] Yao, X (1999) Evolvıng artıficıal neural networks *Proceedıngs of the IEEE*, 87(9) 1423-1447

[41] Yao, X , Lıu, Y (1997) A new evolutıonary system for evolvıng artıficıal neural networks *IEEE Transactıons on Neural Networks*, 8(3) 694-713

Web References

[42] http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/cs11/article1.html

[43] http://www.dacs.dtic.mil/techs/neural/neural3.html

[44] http://lslwww.epfl.ch/~anperez/NN_tutorial/NNdemo_intro.html

[45] http://www.gc.ssr.upm.es/inves/neural/ann1/concepts/Suunsupm.htm

[46] http://ai-junkie.com/ai-junkie.html

[47] http://www.gel.ulaval.ca/~beagle/refmanual/a01116.html

[48] http://www.evalife.dk/ToEC2002

[49] http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#Contents

VITA

Anil Kumar Enumulapally was born in Jagtial, India, on January 10, 1980, the son of Waman Rao Enumulapally and Surekha Enumulapally. After completing his work at Sharada Vidya Nilayam, Jagtial, India, in 1994, he entered Chaitanya Jr. College, Jagtial, India. In September of 1997 he entered into Bapuji Institute of Engineering and Technology, Davangere, India, where he remained until his graduation with a Bachelor of Engineering in Computer Science. He also secured a Higher Diploma in Software Engineering in February, 2000, from Aptech Educational Center, India. In August, 2002 he entered the graduate college of Wichita State University. Later in January, 2003 he transferred to Texas State University-San Marcos to pursue a Master of Science in Computer Science. During his education in the computer science department, he received Academic Excellence awards for years 2003 and 2004 and published a paper titled Backpropagation: In Search of Performance Parameters in WSEAS Transactions on Systems, Issue 2, Vol. 3, April 2004 with Lingguo Bu and Khosrow Kaikhah. During his study he was employed by the English and Finance & Economics Departments as a computer support assistant and web master. He also worked for the Alkek Library as a student assistant.

E-mail: anilkumar.e @gmail.com

This thesis was typed by Anil Kumar Enumulapally.