NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Electrical and Electronic Engineering

GENETIC ALGORITHMS VERSUS NERUAL NETWORKS

Graduation Project
EE – 400

Student: Atique ur Rehman (981262)

Supervisor: Asst. Prof. Dr Adnan Khashman

Nicosia - 2001

Acknowledgements

I am very thankful to the following people:

First of all to **Asst. Prof. Dr Adnan Khashman** who gave me step by step guidance through out the project, I got acquainted with an entirely new and flourishing field of Neural Networks and Genetic Algorithms, I am thankful to my advisor, for his intellectual support, encouragement, enthusiasm which made this thesis possible.

I am also very thankful to my father **Khalil-Ur-Rehman** and my family who have supported, and sponsored me through out my life and encouraged me in every aspect of my life. What I am today is because of them. The greatness of my Father is unable to be explained in words.

I am also thankful to my friends Rihan and Jamal, who helped me a lot in getting the material for the thesis, surely true friends are blessing.

I dedicate my project to all the people mentioned above.

Abstract

Neural Networks and Genetic Algorithms are examples of the latest computer applications, which are under consideration by the scientists over the world. The combination of neural networks and genetic algorithms can be very helpful in creation of artificial intelligent systems. With the development of these two fields, machines that can behave like humans can be invented.

This thesis describes an investigation of combining neural networks and genetic algorithms into real applications.

With the passage of time and development in machine learning a Question Arises?

Will the humans be slaves of machines?

Table of Contents

Introduction

Overview:

Neural Networks is a popular Artificial Intelligent computer systems, inspired by the principles biological neural behavior, this technology is being applied to the computer systems for solving difficult problems, whose solutions require human intelligence. Along with the neural networks another interesting algorithms approach, inspired by the biological genetic behavior, genetic algorithms is being applied in complicated computer systems, along with neural networks.

Research Objectives:

The objectives of the work presented within the thesis are to investigate independently neural networks and genetic algorithms.In addition the benefits that can be achieved by integrating neural networks and genetic algorithms will also be discussed.

Thesis Structure:

Research Objectives include the following:

- In chapter one brief history of neural networks along with biological terminology, with a brief discussion of hybrid intelligent systems will be discussed.
- In chapter two brief history of genetic algorithm along with a brief discussion of search space, genetic operators and application of genetic algorithms will be discussed.
- In chapter three details about neural networks, neural learning, supervised and unsupervised learning, classification of neural networks will be discussed.
- In chapter four details about genetic algorithm, implementation, genetic encoding and selection methods, along with a brief example of prisoner dilemma will be discussed.

- In chapter five a detail example of implementing a genetic algorithm, in a neural network will be discussed.
- In chapter six future directions for implementation of neural networks and genetic algorithms will be discussed.
- Finally the thesis will be concluded, with final remarks.

Chapter One

Neural Networks

1.1 History of Neural Networks:

The progress of neurobiology has allowed researchers to build mathematical models of neurons to simulate neural behavior. This idea dates back to the early 1940s, when one of the first abstract models of a neuron was introduced by McCulloch and Pitts [1], (1943). Hebb [2] in 1949 proposed a learning law that explained how a network of neurons learned, Hebbs law stated that:

"When an axon of cell A is near enough to excite cell B and persistently takes part in firing it some growth process or metabolic changes takes place in one or both cells such that As efficiency increased".

The law proposed by Hebb formed the basis of modern neural network research. Later, Minsky and Papet [1] (1946) pointed out theoretical limitations of single-layer neural network modes in their landmark book Perceptrons. Due to this pessimistic projection, research on artificial neural network lapsed into an eclipse for nearly two decades. Despite the negative atmosphere, some researchers still continued their research and produced meaningful results. For example, Anderson (1977) and Grossberg [1] (1980) did important work on psychological models. Kohonen [1] (1977) developed associative memory models.

In early 1980s the neural network approach was restructured, Hop Field [1] in (1982) introduced the idea of energy minimization in physics into neural networks. In the middle of 1980s, the book Parallel Distributed Processing by Rumelhart [1] and McClelland [1] (1986) generated great impacts on computer, cognitive and biological sciences. The back prorogation-learning algorithm developed by Rumelhart offers a

powerful solution to training a multi-layer neural network and shattered the curse imposed on perceptrons.

1.2 Knowledge-based Information Processing:

Knowledge based information system can be defined as:

" A knowledge-based system is a computer program that acquires, represents, and uses knowledge for a specific purpose".

Its basic structure is as shown in fig below, which consists of a knowledge base which stores knowledge and an inference knowledge engine which makes inference using the knowledge. A conventional computer program is characterized by algorithmic processing data. In this programming paradigm, the knowledge concerning how to do things is enclosed as a bunch of procedures, which are executed step by step to deal with the data entered. In knowledge-based programming, on the other hand, we represent what we know in a declarative manner and knowledge in invoked under a certain inference strategy or driven heuristically.

Another important distinction between the two programming paradigms is the feature of separating knowledge from the control. In knowledge-based systems, knowledge is stored in the knowledge base while control strategies reside in the separate inference engine. This separation benefits the development and maintenance of the system because when knowledge is updated, the inference engine can be left alone, and when the inference process in changed; the knowledge base is not affected. Because of separation, different inference engines can run a knowledge base and an inference engine can drive different knowledge bases. As a consequence, a lot of time and effort can be saved using the knowledge-based approach. The comparison of knowledge and data-oriented information processing is provided in table below.

Knowledge-Based processing	Data-Oriented processing
Declarative knowledge	Procedural knowledge.

Separating control from knowledge.	Integrating control and knowledge.
Strategic and heuristic processing.	Algorithmic processing.
Symbolic processing (dominant).	Numerical processing (dominant).
Explanation capability.	No explanation.

Table 1.1 Comparison of knowledge-based and data-oriented information processing.

1.3 Neural Information Processing:

Biological neurons transmit electrochemical signals over neural pathways. Each neuron receives signals form other neurons through special junctions called ***synapses.*** Some inputs tend to excite neuron; other tends to inhibit it. When the cumulative effect exceeds a threshold, the neuron fires and sends a signal down to other neurons. An artificial neuron models these simple biological characteristics. Each artificial neuron receives a set of inputs. Each input is multiplied by a weight analogous to a synaptic strength. The sum of all weighted inputs determines the degree of firing called the *activation level* (In neural network, connection weights and activations are sometimes reffered to as LTM (long-term memory) and STM (short-term memory), respectively). Notation ally, each input X_i is modulated by weight W_i and the total input is expressed as,

$$\sum X_i W_i$$

or in vector form, **X.W.**

The input signal is further processed by an activation function to produce the output signal, which if not zero, is transmitted along. The activation function can be a threshold function or a smooth function like a sigmoid or a hyperbolic tangent function.

The neural network is represented by a set of nodes and arrows, which is a fundamental concept in graph theory. A node corresponds to a neuron, and an arrow corresponds to a connection along with the direction of signal flow between neurons. As illustrated in

Fig below some nodes are connected to the system input and others are connected to the system output for information processing.

Neural networks solve problems by self-learning and self-organization. They derive their intelligence from the collective behavior of simple computational mechanisms at individual neurons. Computations advantages offered by neural networks include:

- Knowledge acquisition under noise and uncertainty; Neural networks can performs generalization, abstraction, and extraction of statistical properties form the data.
- Flexible knowledge representation: Neural networks can create their own representation by self-organization.
- Efficient knowledge processing: Neural nets can carry out computation in parallel. It is know as parallel-distributed processing, or PDP (Rumelhart [1] and McClelland 1986). Special hardware devices have been manufactured which exploit this advantage. Thus, real-time operation is feasible. Notice that training a neural network may be time-consuming, but once it is trained, it can operate very fast.
- Fault tolerance: Through distributed knowledge representation and redundant information encoding, the system performance degrades gracefully in response to faults (errors).

Neural networks can recognize, classify, convert, and learn patterns. A pattern is a quantitative description of an object or concept or event. A pattern class is a set of pattern sharing some common properties. Pattern recognition refers to the categorization of input data into identifiable classes by recognizing significant features or attributes of the data.

1.4 Brain As A Neural Network:

Human brain is made up of a vast network of computing elements, called neurons, coupled with sensory receptors (affecters) and effectors. The average human brain, roughly three pounds in weight and 90 cubic inches in volume, is estimated to contain

about 100 billion cells of various types. A neuron is a special cell that conducts electrical signal, and there are about 10 billion neurons in the human brain. The remaining 90 billion cells are called glial or glue cell, and these sever as support cells for the neurons. Brain organizes the huge number of neurons (also referred to as cells because glial cells are not of interest here) each with weak computing power, into a massively parallel complex network in which the neurons interact with each other dynamically to produce a powerful information processor.

1.5 Hybrid Intelligence:

Integration of symbolic AI and neural network results in a so-called Hybrid intelligent system. Under this approach, the fundamental assumption on intelligence is as follows:

- Neither the physical symbol system nor the neural network is a necessary means of general intelligent action.
- The symbolic level and the connectionist level represent two different levels of abstraction for intelligent process.
- Knowledge is power. Every intelligent being should have knowledge in one form or another.

Hybrid intelligence is a biological plausible notion. Recall that humans store knowledge in certain complex molecules such as genes and proteins, which determine what we are and how we behave, and at the same time, we have nerve system to coordinate our behavior.

Examples of research in this area include:

- **Knowledge-based neural network**: Neural networks are built based on domain knowledge or theory. In this construct, neural networks model some aspects such as noise and uncertainty which knowledge is not dealing with.

- **Translation of neural network knowledge into symbolic knowledge**: This is important for interpreting neural network, explaining neural network

behavior, and learning knowledge under noise and uncertainty. The idea can also be applied to regularize the neural network and to prevent it form over fitting the data.

- **Learning by combining knowledge and adaptation:** It is concerned with how to build a better learning system that using knowledge or adaptation alone, how to build an incremental learning system, and how to build a useful discovery system. The central idea is to use knowledge as the initial crystal and then grow the crystal by adaptation.

- **Connectionist Symbol processing:** It bears on how to represent symbolic information or knowledge in the framework of connectionists, how to process the information accordingly, and how to retrieve the information. The advantages of this approach include fault tolerance, space sharing, and special processing strategies offered by the distributed representation of connectionists.

- **Hybrid Intelligent System:** Such systems possess knowledge-based components and neural networks, which are integrated in a certain manner so that each component performs the tasks for which it is best, suited.

- **Expert Networks:** They refer to neural networks that can perform as well as human experts. Explanation is an important issue for designing such systems.

Chapter Two
Genetic Algorithms

2.1 Overview:

Science arises from the very human desire to understand and control the world. Over the course of history, we humans have gradually built up the grand edifice of knowledge that enables us to predict, to varying extents, the weather, the motions of the planets, solar and lunar eclipses, the control of diseases, the rise and fall of economic growth, the stages of language development in children, and a vast panorama of other natural, social, and cultural phenomena. Most recently we have even come to understand some fundamental limits to our abilities to predict. Over the eons we have developed increasingly complex means to control many aspects of our lives and our interactions with nature, and we have learned often the hard way, the extent to which other aspects are uncontrollable.

The goal of creating artificial intelligence and artificial life can be traced back to the very beginning of computer age. The earliest computer scientists Alan Turing, John von Neumann, Norbert Wiener [3], and others were motivated in large part by visions of imbuing computer programs with intelligence, with the life-like ability to self-replicate, and with the adaptive capability to learn and to control their environments. These pioneers of computer science were as much interested in biology and psychology as in electronics, and they looked to natural systems as guiding metaphors for how to Brachiates their visions. It should be no surprise, then, that from the earliest days computers were applied not only to calculation missile trajectories and deciphering military codes but also to modeling the brain, mimicking human learning, and simulating biological evolutions.

2.2 Brief History:

Charles Darwin, 1809-1882

Genetic algorithms are appropriate for problems, which require optimization with respect to some computable criterion. This paradigm can also be applied to data mining problems. Here the quantity to be minimized is often the number of classification errors on a training set. Ultragem [4] has developed proprietary techniques for efficiently representing and evolving classification rules using the genetic algorithm paradigm.

Unlike natural evolution, genetic algorithms do not require millions of years to produce results. However, the system may need to run for many hours or even days. Large, complex problems require a fast computer in order to obtain good solutions in a reasonable amount of time. Mining of large datasets by genetic algorithms has only recently become practical due to the availability of affordable high-speed computers such as the DEC Alpha.

In the 1950s and 1960s several computer scientists independently studied evolutionary systems with the idea that evolution could be used as optimization tool for engineering problems. The idea in all these systems was to evolve a population of candidate solution to a given problem, using operators inspired by natural genetic variation and natural selection.

In the 1960s, Rechenberg [3] (1965,1973) introduced "evolutionary strategies", a method he used to optimize real valued parameters for devices such as airfoils. This idea was further developed by Schwefel [3] (1975, 1977). The field of evolution of strategies has remained an active area of research, mostly developing independently from the field of genetic algorithms.

Several other people working in 1950s and the 1960s developed evolution inspired algorithms for optimization and machine learning. Box (1957), Friedman (1959), Bledsoe (1962), Bremermann (1962), and Reed, Toombs, and Baricelli [3] (1967) all worked in this area, though their work has been given little or none attention or follow

up that evolution strategies, evolutionary programming, and genetic algorithms have seen.

Genetic algorithms were invented by John Holland [3] in 1960s and were developed by Holland and his students and colleagues at the University of Michigan in 1960s and 1970s. In contrast with evolution strategies and evolutionary programming, Holland's original goal was not to design algorithms to solve specific problems, but rather to formally study the phenomena of adaptation as it occurs in nature and to develop ways in which the mechanism of natural adaptation might be imported into computer systems.

In the last several years there has been widespread interaction among researchers studying various evolutionary computation methods, and the boundaries between Genetic Algorithms, evolutionary strategies, evolutionary programming, and other evolutionary approaches have been broken down to some extent.

2.3 Biological Terminology:

The evolution of Genetic Algorithms is based on analogy with real biology and can be understood more precisely as:

All living organisms consist of cell, and each cell contains the same set of one or more *chromosomes* - strings of DNA – that serves as "blueprint" for the organism. A chromosome can be conceptually divided into *genes* – functional block of DNA, each of which encodes a particular protein. Very roughly, one can think of gene as encoding a *trait*, such as eye color. The different possible "settings" for a trait (e.g. blue, brown, hazel) are called *alleles*. Each gene is located at a particular locus (position) on the chromosome.

In genetic algorithms, the term chromosome typically refers to candidate solution to a problem, often encoded as a bit string. The "genes" are either single bits or short blocks of adjacent bits that encode a particular element of the candidate solution (e.g. in the context of multi-parameter function optimization the bits encoding a particular

parameter might be considered to be a gene). An allele in a bit string is either 0 or 1; for larger alphabets more alleles are possible at each locus. Crossover typically consists of exchanging genetic material between two single chromosome haploid parents.

2.4 Search Space:

If we are solving some problem, we are usually looking for some solution, which will be the best among others. The space of all feasible solutions (it means objects among those the desired solution is) is called **search space** (also state space). Each point in the search space represents one feasible solution. Each feasible solution can be "marked" by its value or fitness for the problem. We are looking for our solution, which is one point (or more) among feasible solutions - that is one point in the search space.

The looking for a solution is then equal to a looking for some extreme (minimum or maximum) in the search space. The search space can be whole known by the time of solving a problem, but usually we know only a few points from it and we are generating other points as the process of finding solution continues.

2.5 Elements of Genetic Algorithms:

It turns out that there is no rigorous definition of "genetic algorithm" accepted by all in the evolutionary-computation community that differentiates GAs from other evolutionary computation methods. However, it can be said that most methods called "GAs" have at least the following elements in common: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring. Inversion—Holland's fourth element of GAs is rarely used in today's implementations, and its advantages, if any, are not well established.

The chromosomes in a GA population typically take the form of bit strings. Each locus in the chromosome has two possible alleles: 0 and 1.Each chromosome can be thought of as a point in the search space of candidate solutions. The GA processes populations of chromosomes, successively replacing one such population with another. The GA most often requires a fitness function that assigns a score (fitness) to each chromosome in the current population. The fitness of a chromosome depends on how well that chromosome solves the problem at hand.

2.5.1 Examples of Fitness Functions:

One common application of GAs is function optimization, where the goal is to find a set of parameter values that maximize, say, a complex multi-parameter function. As a simple example, one might want to maximize the real-valued one-dimensional function (Riolo [3] 1992). Here the candidate solutions are values of y, which can be encoded as bit strings representing real numbers. The fitness calculation translates a given bit string x into a real number y and then evaluates the function at that value. The fitness of a string is the function value at that point.

$$f(y) = y + |\sin(32.y)|, \qquad 0 < y < \pi$$

As a non-numerical example, consider the problem of finding a sequence of 50 amino acids that will fold to a desired three-dimensional protein structure. A GA could be applied to this problem by searching a population of candidate solutions, each encoded as a 50-letter string such as

IHCCVASASDMIKPVFTVASYLKNWTKAKGPNFEICISGRTPYWDNFPGI,

Where each letter represents one of 20 possible amino acids. One way to define the fitness of a candidate sequence is as the negative of the potential energy of the sequence with respect to the desired structure. The potential energy is a measure of how much physical resistance the sequence would put up if forced to be folded into the desired structure the lower the potential energy, the higher the fitness. Of course one would not want to physically force every sequence in the population into the desired structure and measure its resistance this would be very difficult, if not impossible. Instead, given a sequence and a desired structure (and knowing some of the relevant biophysics), one can estimate the potential energy by calculating some of the forces acting on each amino acid, so the whole fitness calculation can be done computationally.

These examples show two different contexts in which candidate solutions to a problem are encoded as abstract chromosomes encoded as strings of symbols, with fitness

functions defined on the resulting space of strings. A genetic algorithm is a method for searching such fitness landscapes for highly fit strings.

2.6 GA Operators:

The simplest form of genetic algorithm involves three types of operators selection, crossover, and mutation.

1. **Selection:** This operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected.

2. **Crossover:** this operator randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring. For example, the strings 10000100 and 11111111 could be crossed over after the third locus in each to produce the two offspring 10011111 and 11100100. The crossover operator roughly mimics biological recombination between two single-chromosome (haploid) organisms.

3. **Mutation:** This operator randomly flips some of the bits in a chromosome. For example, the string 00000100 might be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probability, usually very small (e.g. 0.001).

2.7 Applications of Genetic Algorithms:

Various kinds of Genetic Algorithms have been applied on different scientific and engineering problems and models. Some examples are:

- **Optimization:** GAs have been used in wide variety of optimization tasks, including numerical optimization and such combinatorial optimization problems as circuit layout and job-shop scheduling.

- **Automatic Programming:** GAs have been used to evolve computer programs for specific tasks, and to design other computational structures such as cellular automata and sorting networks.

- **Machine learning:** GAs have been used for many machine learning applications, including classification and prediction tasks, such as the prediction of weather or protein structure. Gas have also been used to evolve aspects of particular machine learning systems, such as weight for neural networks, rules for learning classifier systems or symbolic production systems, and sensors for robots.

- **Economics:** GAs have been used to model processes of innovation, the development of bidding strategies, and the emergence of economic markets.

- **Immune systems:** GAs have been used to model various aspects of natural immune systems, including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.

- **Ecology:** GAs have been used to model ecological phenomena such as biological arms races, host-parasite co evolution, symbiosis, and resource flow.

- **Population Genetics:** GAs have been used to study how individual learning and species evolution affect one another.

- **Evolution and Learning:** GAs have been used to study how individual learning and species evolution affect one another.

- **Social Systems :** GAs have been used to study evolutionary aspects of social systems, such as the evolution of social behavior in insects colonies, and more generally, the evolution of cooperative and communication in multi-agent systems.

This list gives a brief idea of the flavor of the kinds of things GAs have been used for, both in problem solving and in scientific contexts. Because of their success in these and other areas, interests in GAs has been growing rapidly in the last several years among researchers in many disciplines. The field of GAs has become sub discipline of computer science, with conferences, journals, and scientific society.

Chapter Three

Basic Neural Network Learning and Computational Models

3.1 Overview:

The neural network contains a large number of simple neuron like processing elements and large number of weighted connections between the elements. The weights on the connection encode the knowledge of a network. Though biologically inspired, many of the neural networks developed do not duplicate the operation of human brain. Some computational principles in these models are not even explicable from biological viewpoints.

In many tasks such as recognizing human faces and understanding speech, current AI systems cannot do better than humans. It is conjectured that the structure of brain is somehow suited to these tasks and not suited to tasks such as high-speed arithmetic operation.

The intelligence of a neural network emerges from the collective behavior of neurons, each of which performs only very limited operation. Even though each individual neuron works slowly, they can still quickly find a solution by working in parallel. This fact can explain why humans can recognize a visual scene faster than a digital computer, while an individual brain cell responds much more slowly than a digital cell in VLSI circuit.

3.2 Basic Concepts of Neural Network:

A neural network has a parallel-distributed architecture with a large number of nodes and connections. Each connection points from one node to another and is associated with a weight. A simple view of the network structure and behavior is given in fig 2.1. Construction of a neural network involves the following tasks.

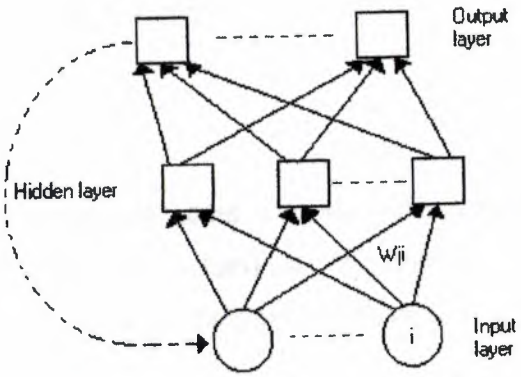


Fig 2.1 A neural network computational model.

- **Determine the network properties:** The network topology (connectivity), types of connections, the order of connections, and the weight range.

- **Determine the node properties:** The activation range and the activation (transfer) function.

- **Determine the system Dynamics:** The weight initialization scheme, the activation-calculating formula, and the learning rule.

1) **Network Properties:** The topology of a neural network refers to its framework as well its interconnection schemes. The framework is often specified by number of layers (or slabs) and the number of nodes per layer. The types of layers include:

- **The input layer:** The nodes in it are called input units, which encode the instance presented to the network for processing. For example, each input unit may be designated by an attribute value possessed by the instance.

- **The hidden layer:** The nodes in it are called the hidden units, which are not directly observable and hence hidden. They provide nonlinear ties for the network.

- **The output layer:** The nodes in it are called output units, which encode possible concepts, (or values) to be assigned to the instance under consideration. For example each output unit represents a class of objects.

The Input units do not process information; they simply distribute the information to the other units. Schematically, input units are drawn as circles as distinguished from processing elements like hidden units and output units, which are drawn as squares.

According to interconnection scheme, a network can either be feed forward or recurrent and its connection either symmetrical as asymmetrical, which are defined below.

- **Feed forward networks:** All connections point in one direction (from the input toward the output layer), or form left to right as shown in figure2.2.

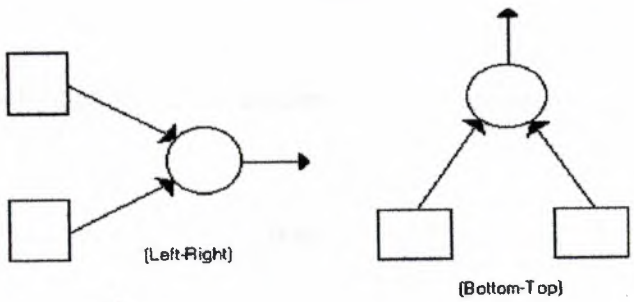


Fig 2.2 Single layer feedforward Perceptron.

- **Recurrent Networks:** There are feedback connections or loops, as shown in Fig 2.1.

- **Symmetrical Connection:** If there is a connection pointing from node 1 to node 2, then there is also a connection from node 2 to node 1, and the weights associated with the two connections are equal, or notationally,

$$W_{12} = W_{21}$$

- **Asymmetrical Connection:** If the connections are not symmetrical as described above then they are asymmetrical.

3.3 Node Properties:

The activation levels of nodes can be discrete (e.g., 0 and 1) or continuous across a range (e.g., [0,1] or unrestricted. This depends on the activation (transfer) function chosen. If it is a hard-limiting function, then the activation level are 0 or (–1) and 1. For a sigmoid function, the activation levels are limited to a continuous range of reals [0,1]. Figure 2.3 shows the sigmoid function F:

$$F(x) = \frac{1}{1+e^{-x}}$$

In case of a linear activation function, the activation levels are open.

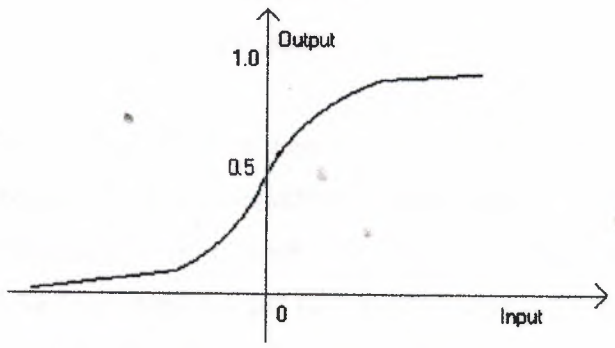


Fig 2.3 The sigmoid activation Function.

3.4 Inference and Learning:

Building an AI system based on the neural network approach will generally involve the following steps.

1. Select a suitable neural network based on the nature of problem.
2. Construct a neural network according to the characteristics of the application domain.
3. Train the neural network with the learning procedure of the selected model.
4. Use the trained network for making inference or solving problems. If the performance is not satisfactory, then go to one of the previous steps.

Familiarity with existing applications will help determine the appropriate network architecture and select the best-suited computational model for learning and inference. Learning is discussed in detail in **supervised** and **unsupervised** learning.

3.5 Learning:

In as much as great variety of human experience can be described as learning, the term machine learning is sometimes obscure. A somewhat more focused definition suggested by Hebert Simon [5] (1983) is based on notion of change.

"Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time".

Learning can refer to either acquiring new knowledge or enhancing or refining skills. Learning new knowledge includes acquisition of significant concepts, understanding of their meanings and relationships to each other and to domain concerned. The new knowledge should be assimilated and put in mentally usable form before it can be called "learned". Thus, knowledge acquisition is defined as learning new symbolic information combined with the ability to use that information effectively.

3.6 Historical Sketch:

Research and development in machine learning have seen several major evolutionary changes. Over the years, different paradigms with different emphasis on objectives have been pursued. Four major periods can be distinguished, each centering around a different paradigm:

Since 1940s and 1950s:

- **Paradigms:** Neural network; decision-theoretical learning.
- **Objectives:** Neural modeling; pattern recognition.
- **Examples:** McCulloch and Pitts (1943); Rosenblatt (1958); Samuel [5] (1959).

Since 1960s:

- **Paradigms:** Symbolic learning.
- **Objectives:** Concept acquisition; building knowledge-based expert systems.
- **Examples:** Winston (1975); Buchanan and Mitchell [5] (1978).

Since 1970s:

- **Paradigms:** Knowledge-intensive learning.
- **Objectives:** Exploration of various learning strategies.
- **Examples:** Mitchell, Keller, and Kedar-Cabelli [5] (1986).

Since 1980s:

- **Paradigms:** Neural network and connectionist learning; hybrid learning.
- **Objectives:** Neural computers; robust learning; massive parallelism.
- **Examples:** Rumelhart, McClelland, and PDP Group (1986); Goldberg [5] (1989).

3.7 Supervised Learning:

In a supervised learning process, the input data and its corresponding output are presented to the neural network. The neural network will according to the defined law

change its weight in order to be able to reproduce the correct output, when an input is applied.

Supervised learning algorithms utilize the information on the class membership of each training instance. This information allows supervised algorithms to detect pattern misclassification as a feedback to themselves. Error information contributes to the learning process by rewarding accurate classification and/or misclassifications – a process known as *credit and blame assignment.* It also helps eliminate implausible hypothesis.

3.8 Unsupervised Learning:

Unsupervised learning process requires only input vectors to train the network. On the input data is presented to the neural network, the weights are adjusted in an ordered way according to some figure of merit.

Unsupervised learning algorithms use unlabeled instances. They blindly or heuristically process them. Unsupervised learning algorithms often have less computational complexity and less accuracy than supervised learning algorithms. Unsupervised learning algorithms can be designed to learn rapidly. This makes unsupervised learning practical in many high-speed, real time environments, where we may not have enough time and information to apply supervised techniques. Unsupervised learning has also been used for scientific discovery.

Unsupervised learning refers to how neural networks modify their parameters in biologically plausible ways. In this learning mode, the neural network does not use the class membership of training instances. Instead, it uses information associated with a group of neurons to modify local parameters.

3.9 Neural Network Learning:

The neural network has been dubbed the "connectionist". It contains large number of simple neuron like processing elements and a large number of weighted connections between the elements. The weights on the connections encode the knowledge of a network. It uses a high parallel, distributed control, and can learn to adjust itself automatically.

3.9.1 Backpropagation:

The backpropagation network is probably the most well known and widely used among the current types of neural network systems available. The learning rule is known as backpropagation, which is a kind of gradient decent technique with backward error (gradient) propagation, as depicted in fig. The training instance set for the network must be presented many times in order for correct classification of input patterns. While the network can recognize patterns similar to those they have learned, they don't have the ability to recognize new patterns. This is true for all supervised learning networks. In order to recognize new patterns network needs to be retrained with these patterns along with previously known patterns. If only new patterns are provided for retraining, then old patterns may be forgotten. In this way, learning is not incremental over time.

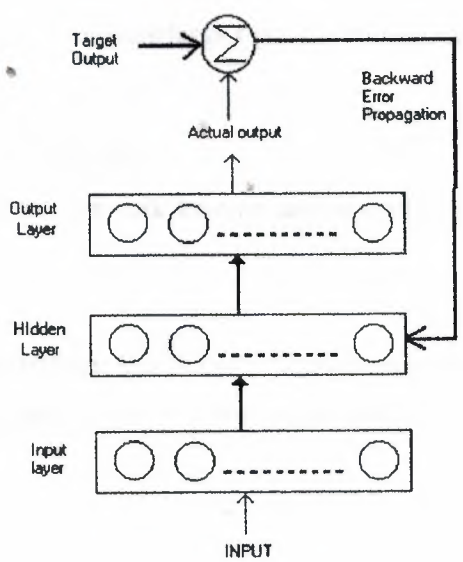


Fig 2.4 The back propagation network.

The backpropagation network is essence learns a mapping from a set of input patterns (e.g. extracted features) to a set of output patterns (e.g. class information). This network can be designed and trained to accomplish a wide variety of mappings. This ability comes form the nodes in hidden layer or layers of the network, which learns to respond to features, found in the input patterns. The features recognized or extracted by hidden units (nodes) correspond to the correlation of activity among different input units. As the network is trained with different examples, the network has the ability to generalize over similar features found in different patterns.

The back propagation network is capable of approximating arbitrary mappings. Furthermore, it can learn to estimate posterior probabilities ($p(w_i/\mathrm{x})$) for classification. The sigmoid function guarantees that the outputs are bounded between 0 and 1.

The back propagation network consists of one input layer, one output layer and one or more hidden layers. If n bits or n values describe the input pattern, then there should be n input units to accommodate it. The number of output units is like wise determined by how many bits or values are involved in output pattern.

The name back propagation comes from the fact that the error (gradient) or hidden units are derived form propagation backward the errors associated with output units. In back propagation network, the activation function chosen is the sigmoid function, which compresses the output value into the range between 0 and 1. The sigmoid function is advantageous in that it can accommodate large signals without saturation

- **Back Propagation Learning:**

The equation that describes the network training and operation can be divided into two categories.

1. **Feed forward Calculations:** Use in both training mode and operation mode.
2. **Error Back Propagation:** Use in training mode only.

Activation Function: Any activation function that is differentiable can be used in Back propagation algorithm.

- Linear Function with adjustable gain.
- Sigmoid Function (Squashing Function).

a) Feed forward Calculations:

Normalization of the input data prior training is necessary. The value of input data into input layer must be in the range (0-1).

Input Layer (*i*): The output of each input neuron is exactly equal to the normalized input.

$$\text{Input-layer} = \text{Output}$$

$$O_i = I_i$$

Hidden Layer (*h*): The signal presented to a neuron in the hidden layer is equal to the sum of all outputs of the input layer multiplied by their associated weights.

Hidden Layer Input:

$$I_h = \sum_i W_{hi} O_i$$

Each output of a hidden nurode is calculated using the SIGMOID function.

Output Layer (*j*): Similar to Hidden Layer Calculation.

Output layer Input (*j*):

$$I_j = \sum_i W_{jh} O_h$$

These equations describe the feedforward calculations, which can be used in both training and running phases.

b) Error Back Propagation Calculations:

Vital elements in these calculations are:

Error Signal: The definition of network error is the difference between the output value that an output neuron is supposed to have (Target value, Tj), and the value it actually has as a result of feed forward calculations (O_j).

$$E_p = \sum_{j=1}^{n_j} (T_{pj} - O_{fp})^2$$

P: denotes what the value is for a given pattern.

The aim of training a Neural Network is to minimize this error over all training patterns. The output of a neuron in the output layer is a function of its input $O_j = f(I_j)$. The first derivative of this function $f(I_j)$ is an important element in error back propagation. For output layer neurons, a quantity called the error signal is represented by Δ_j which is defined as,

$$\Delta_j = f'(I_j)(T_j - O_j)$$
$$\Delta_j = (T_j - O_j)O_j(1 - O_j)$$

The error value is propagated back and weights adjustments are made.

There are two essential parameters that affect the learning of a neural network:

1. Learning co-efficient η which defines the learning power of a neural network.
2. Momentum factor α, which defines the learning power of a neural network.

The effect of these parameters is described by the following equations.

Output Layer Weights Update: The weights that feed the output layer (W_{jh}) are updated using the following equations. This also includes the bias weights at the output layer. However in order to prevent the network getting caught in local minima, the momentum term is also added.

$$W_{jh}(new) = W_{jh}(old) + \eta \Delta_j O_j$$

or with momentum rate

$$W_{jh}(new) = W_{jh}(old) + \eta \Delta_j O_j + \alpha \left[\delta W_{jh}(old) \right]$$

Hidden Layer Weights Update: Similar to output layer weights update but the Delta error will be different. Error for Hidden layer is defined by the following equation.

$$\Delta_h = O_h (1 - O_h) \sum_{j=0}^{n_j} W_{jh} \Delta_j$$

Weights-adjustments:

$$W_{hi}(new) = W_{hi}(old) + \eta \Delta_h O_i + \alpha \left[\delta W_{hi}(old) \right]$$

All the equations describe the mathematical foundations for Back Propagation Learning Algorithm.

3.10 Classification Models:

Neural Networks can be classified according to the way they learn, learning can be performed on a Supervised Or Unsupervised basis.

- **Supervised Learning Models:**
  1. The Perceptron.
  2. The Back Propagation Learning Algorithm.
  3. The Hop Field Algorithm.
  4. The Hamming Algorithm.

- **The Perceptron:** This can be trained and can make decisions. During the training phase, pairs of input and output vectors are used to train the network. While each input vector, the output vector is compared with a desired output (target) as shown in fig 2.4, and the error between the actual and desired output vectors is used to update the weights.

- **Back Propagation:** A multi-layer network can be trained using the back propagation-learning algorithm. This is done by presenting pairs of input and output vectors. The actual output is compared with the target. If there is no difference the weights do not change, otherwise the weights are adjusted to reduce the error difference. This learning algorithm propagates back the error through the multi-layer to update the weights.

- **Hop Field Network:** A Hop field network is essentially used with binary numbers. Weights are initialized using training samples. In the decision making phase, the test data is presented to the net at certain time, following initialization the Hop field Network iterates in discrete time steps using some mathematical function, and the network is considered to have converged when the outputs no longer change on successive iterations.

- **Hamming Network:** It is similar to Hop field network, but it consists of four layers.

L1: Input Layer.
L2: Calculates matching scores.
L3: Feed back as in Hop field.
L4: Output Layer.

- **Unsupervised Learning Models:**
  1. Kohonen's Self-Organizing Maps.
  2. Competitive Learning.

3. Adaptive Learning.

- **Kohonen's Learning:** Kohonen [2] suggested that that one of the important mechanism in the human brain is placement of Neurons in an orderly manner. Kohonen's learning algorithm creates a feature map by adjusting weights from input vectors to output vectors in a two layer network. The first layer is input. the second is competitive layer. The tow layers are fully connected. Input vectors are presented sequentially to layer one. Each unit computes the dot product of its weight with the input vectors. The unit with the highest dot product is declared the winner. This and its neighbors are the only units allowed to learn.

- **Competitive Learning:** The simplest way to implement competitive learning is where each unit in the hidden or output layers receives input from all the units in the preceding layer. With in the layer units are broken down into a set of inhibitory clusters. The units with in the cluster compete with in one another to respond to data appearing at input layer. The more strongly and particular unit responds to incoming stimulus the more it inhibits other units with in the cluster. The unit learns by shifting a fraction of its weight from its inactive lines. The main disadvantage of competitive learning is the loss of previous learning's.

- **Adaptive Resonance Theory (ART):** ART is divided into two methods.

    1. Accepts only binary.
    2. Accepts binary and continuous input.

Chapter Four
Genetic Algorithm Implementation

4.1 How do Genetic Algorithms Works?

Although genetic algorithms are simple to describe and program, their behavior can be complicated, and many open questions exist about how they work and for what types of problems they are best suited. Much work has been done on theoretical foundations of GAs.

The tradition theory of GAs (first formulated in Holland 1975) assumes that, a very general level of description, GAs work by discovering, emphasizing, and recombination 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 – combination of bit values that confer higher fitness on the strings in which they are present.

Holland [6] (1975) introduced the notion of schemas (or schemata) to formalize the information notion of "building blocks". A schema is a set of bit strings that can be described by a template made up of ones, zeros, and asterisks, the asterisks representing wild cards (or "don't cares"). For example, the schema H=1****1 represents the set of all 6-bit strings that begin and end with 1. The strings that fit this template (e.g. 100111 and 110011) are said to be instances of H. The schema H is said to have two defined bits (non-asterisks) or, equivalent, to be or order 2. Its defining length (the distance between its outermost defined bits) is 5. Here the term "schema" is used to donate both a subset of strings represented by such a template itself.

Note that not every possible subset of the set of length-l bit strings can be described as a schema; in fact, the huge majority cannot. There are 2^l possible strings of length l, and thus 2^{2l} possible subsets of strings, but there are only 3^l possible schemas. However, a central tenet of traditional GA theory is that schemas are – implicitly – the building

blocks that the GA processes effectively under the operators of selection, mutation, and single-point crossover.

How do the GA process schemas? Any given bit string of length l is an instance of 2^l different schemas. For example, the string 11 is an instance of **(all four possible bit strings of length 2), *1, 1*, and 11 (the schema that contains only one string, 11). Thus, any given population of n strings contains instances of between 2^l and $n \times 2^l$ different schemas. If all the strings are identical, then there are instances of exactly 2^l different schemas; otherwise, the number is less than or equal to $n \times 2^l$. This means that, at a given generation, while the GA is explicitly evaluating the fitness of the n strings in the population, it is actually implicitly estimating the average fitness of a much larger number of schemas, where the average fitness of a schema is defined to be the average fitness of possible instances of that schema. For example, in a randomly generated population of n strings, on average half strings will be instances of 1***--* and half will be instances of 0***---0. The evaluation of approximately n/2 strings that are instances of 1**---* give an estimate of the average fitness of that schema. Just as schemas are not explicitly represented or evaluated by the GA, the estimates of schema average fitness are not calculated or stored explicitly by the GA. However, as will be seen below, the GAs behavior, in term of the population, can be described as though it actually calculating and storing these averages.

We can calculate the approximate dynamics of this increase decrease in schema instances by using the following equation:

$$E(m(H,t+1)) = \sum_{x \in H} f(x) / \bar{f}(t)$$
$$= (\hat{u}(H,t) / \bar{f}(t)) m(H,t) \qquad (1.1)$$

Where H is considered as schema with at least one instance present in the population at time t, m(H,t) be the number of instances present in the population at time t, and u(H,t) be the observed average fitness of H at time t.

The disruptive effects of mutation can be quantified as follow; let p_m be the probability of any bit being mutated. Then $S_m(H)$, the probability that schema H will survive under and instance of H, is equal to $(1-p_m)^{O(H)}$, where o(H) is order of H. (i.e. the number of defined bits in H).

The disruptive effects can be used to amend equation 1.1, and can be defined as:

$$E(m(H,t+1)) = \frac{\hat{u}(H,t)}{\bar{f}(t)} m(H,t)\left(1 - p_c \frac{d(H)}{l-1}\right)\left[(1-p_m)^{O(H)}\right] \qquad (1.2)$$

This is known as Schema Theorem. It describes the growth of a schema from one generation to the next.

The Schema Theorem and some of its purported implications for the behavior of GAs have been subjected of much critical discussion in the GA community. These criticisms and the new approaches are discussed in detail in the coming sections.

4.2 Implementing a Genetic Algorithm:

The case studied earlier illustrated that when one wants to apply the GA to a particular problem, one faces a huge number of choices about how to proceed, with little theoretical guidance on how to make them.

John Holland's simple GA inspired all subsequent GAs and provided the basis for theoretical analysis of GAs. For real problem solving and modeling, however, it is clear that the simple GA is limited in its power in several respects. Not all problems should use bit-string encodings, fitness-proportionate selection is not always the best method, and the simple genetic operators are not always the most effective or appropriate ones. Furthermore, the simple GA leaves out many potentially useful ideas from real biology, several of which were proposed for use in GAs by Holland (1975) but have not been examined systematically until recently.

In this section we will survey some implementation issues for GAs and some sophisticated GA techniques, including self-adapting GAs. Of course, this survey is by no means complete—although GA researchers speak informally of "the GA," anyone who has little idea about GAs will notice that there are actually as many different GAs as there are GA projects.

4.3 When Should a Genetic Algorithm be used?

The GA literature describes a large number of successful applications, but there are also many cases in which GAs perform poorly. Given a particular potential application, how do we know if a GA is good method to use? There is no rigorous answer, though many researchers share the intuitions that if the space to be searched is large/ is known not to be perfectly smooth and unimodal (i.e., consists of a single smooth "hill"), or is not well understood, or if the fitness function is noisy, and if the task does not require a global optimum to be found—i.e., if quickly finding a sufficiently good solution is enough—a GA will have a good chance of being competitive with or surpassing other "weak" methods (methods that do not use domain-specific knowledge in their search procedure). If a space is not large, then it can be searched exhaustively, and one can be sure that the best possible solution has been found, whereas a GA might converge on a local optimum rather than on the globally best solution. If the space is smooth or unimodal, a gradient-ascent algorithm such as steepest-ascent hill climbing will be much more efficient than a GA in exploiting the space's smoothness. If the space is well understood (as is the space for the well-known Traveling Salesman problem, for example), search methods using domain-specific heuristics can often be designed to outperform any general-purpose method such as a GA. If the fitness function is noisy (e.g., if it involves taking error-prone measurements from a real-world process such as the vision system of a robot), a one-candidate-solution-at-a-time search method such as simple hill climbing might be irrecoverably led astray by the noise, but GAs, since they work by accumulating fitness statistics over many generations, are thought to perform robustly in the presence of small amounts of noise.

4.4 Encoding a Problem for Genetic Algorithm:

As for any search and learning method, the way in which candidate solutions are encoded is a central, if not the central, factor in the success of a genetic algorithm. Most GA applications use fixed-length, fixed-order bit strings to encode candidate solutions. However, in recent years, there have been many experiments with other kinds of encoding.

1. Binary Encodings:

Binary encoding (i.e., bit strings) are the most common encodings for a number of reasons. One is historical: in their earlier work, Holland and his students concentrated on such encodings and GA practice has tended to follow this lead. Much of the existing GA theory is based on the assumption of fixed-length, fixed-order binary encodings. Much of that theory can be extended to apply to nonbinary encodings, but such extensions are not as well developed as the original theory. In addition, heuristics about appropriate parameter settings (e.g., for crossover and mutation rates) have generally been developed in the context of binary encodings.

Holland [3] (1975) gave a theoretical justification for using binary encodings. He compared two encodings with roughly the same information-carrying capacity, one with a small number of alleles and long strings (e.g., bit strings of length 100) and the other with a large number of alleles and short strings (e.g., decimal strings of length 30). He argued that the former allows for a higher degree of implicit parallelism than the latter, since an instance of the former contains more schemas than an instance of the latter (2^{100} versus 2^{30}). (This schema-counting argument is relevant to GA behavior only insofar as schema analysis is relevant, which, as I have mentioned, has been disputed).

In spite of these advantages, binary encodings are unnatural and unwieldy for many problems (e.g., evolving weights for neural networks or evolving condition sets in the manner of Meyer and Packard), and they are prone to rather arbitrary orderings.

2. Many-Character and Real-Valued Encodings:

For many applications, it is most natural to use an alphabet of many characters or real numbers to form chromosomes. Examples include Kitano's [3] many-character representation for graph-generation grammars, Meyer and Packard's [3] real-valued representation for condition sets, Montana and Davis's [3] real-valued representation for neural-network weights, and Schultz-Kremer's [3] real-valued representation for torsion angles in proteins.

Holland's schema-counting argument seems to imply that GAs should exhibit worse performance on multiple-character encodings than on binary encodings. Several empirical comparisons between binary encodings and multiple-character or real-valued encodings have shown better performance for the latter e.g., Janikow and Michalewicz [3]. But the performance depends very much on the problem and the details of the GA being used, and at present there are no rigorous guidelines for predicting which encoding will work best.

3. Tree Encodings:

Tree encoding schemes, such as John Koza's [3] scheme for representing computer programs, have several advantages, including the fact that they allow the search space to be open-ended (in principle, any size tree could be formed via crossover and mutation). This open-endedness also leads to some potential pitfalls. The trees can grow large in uncontrolled ways, preventing the formation of more structured, hierarchical candidate solutions. (Koza's [3] (1992, 1994) "automatic definition of functions" is one way in which GP can be encouraged to design hierarchically structured programs.) Also, the resulting trees, being large, can be very difficult to understand and to simplify. Systematic experiments evaluating the usefulness of tree encodings and comparing them with other encodings are only just beginning in the genetic programming community. Likewise, as yet there are only very nascent attempts at extending GA theory to tree encodings.

These are only the most common encodings; a survey of the GA literature will turn up experiments on several others.

4.5 Adapting the Encoding:

Choosing a fixed encoding ahead of time presents a paradox to the potential GA user: for any problem that is hard enough that one would want to use a GA, one doesn't know enough about the problem ahead of time to come up with the best encoding for the GA. In fact, coming up with the best encoding is almost tantamount to solving the problem itself! The original lexicographic ordering of bits was arbitrary, and it probably impeded the GA from finding better solutions quickly—to find high-fitness rules, many bits spread throughout the string had to be co adapted. If these bits were close together on the string, so that they were less likely to be separated under crossover, the performance of the GA would presumably be improved. But we had no idea how best to order the bits ahead of time for this problem. This is known in the GA literature as the "linkage problem"—one wants to have functionally related loci be more likely to stay together on the string under crossover, but it is not clear how this is to be done without knowing ahead of time which loci are important in useful schemas. Faced with this problem, and having notions of evolution and adaptation already primed in the mind, many users have a revelation: "As long as I'm using a GA to solve the problem, why not have it adapt the encoding at the same time!"

1. Inversion:

Holland [3] (1975) included proposals for adapting the encodings in his original proposal for GAs. Holland, acutely aware that correct linkage is essential for single-point crossover to work well, proposed an "inversion" operator specifically to deal with the linkage problem in fixed-length strings.

Inversion is a reordering operator inspired by a similar operator in real genetics. Unlike simple GAs, in real genetics the function of a gene is often independent of its position in the chromosome (though often genes in a local area work together in a regulatory network), so inverting part of the chromosome will retain much or all of the "semantics" of the original chromosome.

To use inversion in GAs, we have to find some way for the functional interpretation of an allele to be the same no matter where it appears in the string. For example, in the chromosome encoding a cellular automaton the leftmost bit under lexicographic ordering is the output bit for the neighborhood of all zeros. We would want that bit to represent that same neighborhood even if its position were changed in the string under an inversion. Holland proposed that each allele be given an index indicating its "real" position, to be used when evaluating a chromosome's fitness. For example, the string 00010101 would be encoded as

{(1,0) (2,0) (3, 0) (4,1) (5, 0) (6,1) (7, 0) (8,1)},

With the first member of each pair giving the "real" position of the given allele. This is the same string as, say,

{(1, 0) (2, 0) (6,1) (5, 0) (4, .1) (3,0) (7,0) (8,1)}.

Inversion works by choosing two points in the string and reversing the order of the bits between them—in the example just given, bits 3-6 were reversed. This does not change the fitness of the chromosome, since to calculate the fitness the string is ordered by the indices. However, it does change the linkages: the idea behind inversion is to produce orderings in which beneficial schemas are more likely to survive. Suppose that in the original ordering the schema 00 * *01 * * is very important. Under the new ordering, that schema is 0010 * * * *. Given that this is a high-fitness schema and will now tend to survive better under single-point crossover, this permutation will presumably tend to survive better than would the original string.

The reader may have noticed a hitch in combining inversion with single-point crossover. Suppose, for example, that

{(1, 0) (2, 0) (6,1) (5,0) (4,1) (3,0) (7, 0) (8,1)}

Crosses with

{(5,1) (2, 0) (3,1) (4,1) (1,1) (8,1) (6,0) (7, 0)}

After the third bit, the offspring are

{(1,0) (2,0) (6,1) (4,1) (1,1) (8,1) (6, 0) (7,0)}

And

{(5,1) (2, 0) (3,1) (5,0) (4,1) (3,0) (7, 0) (8,1)}.

The first offspring has two copies each of bits 1 and 6 and no copies of bits 3 and 5. The second offspring has two copies of bits 3 and 5 and no copies of bits 1 and 6. How can we ensure that crossover will produce offspring with a full set of loci? Holland proposed two possible solutions:

- Permit crossover only between chromosomes with the same permutation of the loci. This would work, but it severely limits the way in which crossover can be done.

- Employ a "master/slave" approach: choose one parent to be the master, and temporarily reorder the other parent to have the same ordering as the master. Use this ordering to produce offspring, returning the second parent to its original ordering once crossover has been performed. Both methods have been used in experiments on inversion.

Inversion was included in some early work on GAs but did not produce any stunning improvements in performance (Goldberg 1989a). More recently, forms of inversion have been incorporated with some success into GAs applied to "ordering problems" such as the DNA fragment assembly problem (Parsons, Forrest, and Burks, in press). However, the verdict on the benefits of inversion to GAs is not yet in; more systematic experimental and theoretical studies are needed. In addition, any performance benefit conferred by inversion must be weighed against the additional space (to store indices for every bit) and additional computation time (e.g., to reorder one parent before crossover) that inversion requires.

2. Evolving Crossover "Hot Spots":

A different approach, also inspired by nature, was taken by Schaffer and Morishima [3] (1987). Their idea was to evolve not the order of bits in the string but rather the positions at which crossover was allowed to occur (crossover "hot spots"). Attached to each candidate solution in the population was a second string—a "crossover template"—that had a 1 at each locus at which crossover was to take place and a 0 at each locus at which crossover was not to take place. For example, 10011111:00010010 (with the chromosome preceding and the crossover template following the colon) meant that crossover should take place after the fourth and seventh loci in that string. Using an exclamation point to denote the crossover markers (each attached to the bit on its left), we can write this as 1001!111!1. Now, to perform multi-point crossover on two parents (say 1001!111!1 and OOOOOO'OO), the !s mark the crossover points, and they get inherited along with the bits to which they are attached:

Parents **1 0 0 I! 1 1 I! 1**
0 0 0 0 0 0! 0 0

Offspring **1 0 0 0! 0 0! I! 0**
0 0 0 0 1 1 0 1

Mutation acts on both the chromosomes and the attached crossover templates. Only the candidate solution is used to determine fitness, but the hope is that selection, crossover, and mutation will not only discover good solutions but also coevolve good crossover templates. Schaffer and Morishima found that this method outperformed a version of the simple GA on a small suite of function optimization problems. Although this method is interesting and is inspired by real genetics (in which there are crossover hot spots that have somehow coevolved with chromosomes), there has not been much further investigation into why it works and to what degree it will actually improve GA performance over a larger set of applications.

3. Messy GAs:

The goal of "messy GAs," developed by Goldberg and his colleagues, is to improve the GA's function-optimization performance by explicitly building up increasingly longer, highly fit strings from well-tested shorter building blocks (Goldberg, Korb, and Deb 1989; Goldberg, Deb, and Korb [3] 1990; Goldberg, Deb, Kargupta, and Harik [3] 1993). The general idea was biologically motivated: "After all, nature did not start with strings of length 5.9×10^9 (an estimate of the number of pairs of DNA nucleotides in the human genome) or even of length two million (an estimate of the number of genes in *Homo sapiens)* and try to make man. Instead, simple life forms gave way to more complex life forms, with the building blocks learned at earlier times used and reused to good effect along the way."

Consider a particular optimization problem with candidate solutions represented as bit strings. In a messy GA each bit is tagged with its "real" locus, but in a given chromosome not all loci have to be specified ("under specification") and some loci can be specified more than once, even with conflicting alleles ("over specification"). For example, in a four-bit problem, the following two messy chromosomes might be found in the population:

{(1,0) (2,0) (4,1) (4,0)}

and

{(3,1) (3, 0) (3,1) (4, 0) (4,1) (3,1)}.

The first specifies no value for locus 3 and two values for locus 4. The second specifies no values for loci 1 and 2, two values for locus 4 and a whopping four values for locus 3. (The term "messy GA" is meant to be contrasted with standard "neat" fixed-length, fixed-population-size GAs.)

Given all this under- and over specification, how is the fitness function to be evaluated? Over specification is easy: Goldberg and his colleagues simply used a left-to-right, first-come-first-served scheme. (e.g., the chosen value for locus 4 in the first chromosome is 1.) Once over specification has been taken care of, the specified bits in the chromosome can be thought of as a "candidate schema" rather than as a candidate solution. For example, the first chromosome above is the schema 00*1. The purpose of messy GAs is to evolve such candidate schemas, gradually building up longer and longer ones until a solution is formed. This requires a way to evaluate a candidate schema under a given fitness function. However, under most fitness functions of interest, it is difficult if not impossible to compute the "fitness" of a partial string. Many loci typically interact non-independently to determine a string's fitness, and in an under specified string the missing loci might be crucial. Goldberg and his colleagues first proposed and then rejected an "averaging" method: for a given under specified string, randomly generate values for the missing loci over a number of trials and take the average fitness computed with these random samples. The idea is to estimate the average fitness of the candidate schema. But, as was pointed out earlier, the variance of this average fitness will often be too high for a meaningful average to be gained from such sampling. Instead, Goldberg [3] and his colleagues used a method they called "competitive templates." The idea was not to estimate the average fitness of the candidate schema but to see if the candidate schema yields an improvement over a local optimum. The method works by finding a local optimum at the beginning of a run by a hill-climbing technique, and then, when running the messy GA, evaluating under specified strings by filling in missing bits from the local optimum and then applying the fitness function. A local optimum is, by definition, a string that cannot be improved by a single-bit change;

thus, if a candidate schema's defined bits improve the local optimum, it is worth further exploration.

The messy GA proceeds in two phases: the "primordial phase" and the "juxtapositional phase." The purpose of the primordial phase is to enrich the population with small, promising candidate schemas, and the purpose of the juxtapositional phase is to put them together in useful ways. Goldberg and his colleagues' first method was to guess at the order k of the smallest relevant schemas and to form the initial population by completely enumerating all schemas of that order. For example, if the size of solutions is $l = 8$ and the guessed k is 3, the initial population will be,

{(1,0) (2,0) (3,0)}
{(1,0) (2,0) (3,1)}

.
.

{(1,1) (2,1) (3,1)} {(1,0) (2,0) (4,0)}
{(1,0) (2,0) (4,1)}

.

{(6,1) (7,1) (8,1)}.

After the initial population has been formed and the initial fitnesses evaluated (using competitive templates), the primordial phase continues by selection only (making copies of strings in proportion to their fitnesses with no crossover or mutation) and by culling the population by half at regular intervals. At some generation (a parameter of the algorithm), the primordial phase comes to an end and the juxtapositional phase is invoked. The population size stays fixed, selection continues, and two juxtapositional operators—"cut" and "splice"—are introduced. The cut operator cuts a string at a random point. For example,

{(2,0) (3,0) (1,1) (4,1) (6,0)}

Could be cut after the second locus to yield two strings: {(2, 0) (3,0)} and {(1,1) (4,1) (6, 0)}. The splice operator takes two strings and splices them together. For example,

{(1.1) (2,1) (3,1)} and {(1, 0) (4,1) (3, 0)}

Could be spliced together to form

{(1,1) (2,1) (3,1) (1,0) (4,1) (3,0)}.

Under the messy encoding, cut and splice always produce perfectly legal strings. The hope is that the primordial phase will have produced all the building blocks needed to create an optimal string, and in sufficient numbers so that cut and splice will be likely to create that optimal string before too long. Goldberg and his colleagues did not use mutation in the experiments they reported.

Unfortunately, even with probabilistically complete initialization, the necessary initial population size still grows exponentially with *k*, so messy GAs will be feasible only on problems in which *k* is small. Goldberg and his colleagues seem to assume that most problems of interest will have small *k*, but this has never been demonstrated. It remains to be seen whether the promising results they have found on specially designed fitness functions will hold when messy GAs are applied to real-world problems. Goldberg, Deb, and Korb [3] have already announced that messy GAs are "ready for real-world applications" and recommended their "immediate application ... to difficult, combinatorial problems of practical import." To my knowledge, they have not yet been tried on such problems.

4.6 Selection Methods:

After deciding on an encoding, the second decision to make in using a genetic algorithm is how to perform selection—that is, how to choose the individuals in the population that will create offspring for the next generation, and how many offspring each will create. The purpose of selection is, of course, to emphasize the fitter individuals in the

population in hopes that their offspring will in turn have even higher fitness. Selection has to be balanced with variation from crossover and mutation (the "exploitation/exploration balance"): too-strong selection means that suboptimal highly fit individuals will take over the population, reducing the diversity needed for further change and progress; too-weak selection will result in too-slow evolution. As was the case for encodings, numerous selection schemes have been proposed in the GA literature. In the following section are described some of the most common methods.

1. **Fitness-Proportionate Selection with "Roulette Wheel" and "Stochastic Universal" Sampling:**

Holland's original GA used fitness-proportionate selection, in which the "expected value" of an individual (i.e., the expected number of times an individual will be selected to reproduce) is that individual's fitness divided by the average fitness of the population. The most common method for implementing this is "roulette wheel" sampling, each individual is assigned a slice of a circular "roulette wheel," the size of the slice being proportional to the individual's fitness. The wheel is spun N times, where N is the number of individuals in the population. On each spin, the individual under the wheel's marker is selected to be in the pool of parents for the next generation. This method can be implemented as follows:

1. Sum the total expected value of individuals in the population. Call this sumT.

2. Repeat N times

Choose a random integer r between 0 and T'. Loop through the individuals in the population, summing the expected values, until the sum is greater than or equal to r. The individual whose expected value puts the sum over this limit is the one selected.

This stochastic method statistically results in the expected number of offspring for each individual. However, with the relatively small populations typically used in GAs, the actual number of offspring allocated to an individual is often far from its expected value (an extremely unlikely series of spins of the roulette wheel could even allocate all offspring to the worst individual in the population). James Baker (1987) proposed a

different sampling method—"stochastic universal sampling" (SUS)—to minimize this "spread" (the range of possible actual values, given an expected value). Rather than spin the roulette wheel N times to select N parents, SUS spins the wheel once—but with N equally spaced pointers, which are used to selected the N parents. Baker (1987) gives the following code fragment for SUS (in C).

SUS does not solve the major problems with fitness-proportionate selection. Typically, early in the search the fitness variance in the population is high and a small number of individuals are much fitter than the others. Under fitness-proportionate selection, they and their descendents will multiply quickly in the population, in effect preventing the GA from doing any further exploration. This is known as "premature convergence." In other words, fitness-proportionate selection early on often puts too much emphasis on "exploitation" of highly fit strings at the expense of exploration of other regions of the search space. Later in the search, when all individuals in the population are very similar (the fitness variance is low), there are no real fitness differences for selection to exploit, and evolution grinds to a near halt. Thus, the rate of evolution depends on the variance of fitnesses in the population.

2. Sigma Scaling:

To address such problems, GA researchers have experimented with several "scaling" methods—methods for mapping "raw" fitness values to expected values so as to make the GA less susceptible to premature convergence. One example is "sigma scaling" (Forrest [3] 1985; it was called "sigma truncation" in Goldberg 1989a), which keeps the selection pressure (i.e., the degree to which highly fit individuals are allowed many offspring) relatively constant over the course of the run rather than depending on the fitness variances in the population. Under sigma scaling, an individual's expected value is a function of its fitness, the population mean, and the population standard deviation. An example of sigma scaling would be

$$ExpVal(i,t) = \begin{cases} 1 + \dfrac{f(i) - \bar{f}(t)}{2\sigma(t)} & \text{if} \sigma(t) \neq 0 \\ 1.0 \rightarrow otherwise \end{cases}$$

where ExpVal (i, t) *is* the expected value of individual *i* at time *t*, *f(i) is* the fitness of *i*, *f(t) is* the mean fitness of the population at time *t*, and $\sigma(t)$ is the standard deviation of the population fitnesses at time *t*. This function, used in the work of Tanese (1989), gives an individual with fitness one standard deviation above the mean 1.5 expected offspring. If ExpVal(i,t) was less than 0, Tanese arbitrarily reset it to 0.1, so that individuals with very low fitness had some small chance of reproducing.

At the beginning of a run, when the standard deviation of fitnesses is typically high, the fitter individuals will not be many standard deviations above the mean, and so they will not be allocated the lion's share of offspring. Likewise, later in the run, when the population is typically more converged and the standard deviation is typically lower, the fitter individuals will stand out more, allowing evolution to continue.

3. Elitism:

"Elitism," first introduced by Kenneth De Jong [3] (1975), is an addition to many selection methods that forces the GA to retain some number of the best individuals at each generation. Such individuals can be lost if they are not selected to reproduce or if they are destroyed by crossover or mutation. Many researchers have found that elitism significantly improves the GA's performance.

4. Boltzmann Selection:

Sigma scaling keeps the selection pressure more constant over a run. But often different amounts of selection pressure are needed at different times in a run—for example, early on it might be good to be liberal, allowing less fit individuals to reproduce at close to the rate of fitter individuals, and having selection occur slowly while maintaining a lot of variation in the population. Later it might be good to have selection be stronger in

order to strongly emphasize highly fit individuals, assuming that the early diversity with slow selection has allowed the population to find the right part of the search space.

One approach to this is "Boltzmann selection" (an approach similar to simulated annealing), in which a continuously varying "temperature" controls the rate of selection according to a preset schedule. The temperature starts out high, which means that selection pressure is low (i.e., every individual has some reasonable probability of reproducing). The temperature is gradually lowered, which gradually increases the selection pressure, thereby allowing the GA to narrow in ever more closely to the best part of the search space while maintaining the "appropriate" degree of diversity. For examples of this approach, see Goldberg 1990, de la Maza and Tidor [3] 1991 and 1993, and Prugel-Bennett and Shapiro [3] 1994. A typical implementation is to assign to each individual; an expected value,

$$ExpVal(i,t) = \frac{e^{f(i)/T}}{\left\langle e^{f(i)/T} \right\rangle T}$$

where T is temperature and $\langle\ \rangle t$ denotes the average over the population at time t. Experimenting with this formula will show that, as T decreases, the difference in ExpVal(i, t) between high and low fitnesses increases. The desire is to have this happen gradually over the course of the search, so temperature is gradually decreased according to a predefined schedule. De la Maza and Tidor [3] (1991) found that this method outperformed fitness-proportionate selection on a small set of test problems. They also (1993) compared some theoretical properties of the two methods.

5. Rank Selection:

Rank selection is an alternative method whose purpose is also to prevent too-quick convergence. In the version proposed by Baker (1985), the individuals in the population are ranked according to fitness, and the expected value of each individual depends on its rank rather than on its absolute fitness. There is no need to scale fitnesses in this case,

since absolute differences in fitness are obscured. This discarding of absolute fitness information can have advantages (using absolute fitness can lead to convergence problems) and disadvantages (in some cases it might be important to know that one individual is far fitter than its nearest competitor). Ranking avoids giving the far largest share of offspring to a small group of highly fit individuals, and thus reduces the selection pressure when the fitness variance is high. It also keeps up selection pressure when the fitness variance is low: the ratio of expected values of individuals ranked i and, $i+1$ will be the same whether their absolute fitness differences are high or low.

6. Tournament Selection:

The fitness-proportionate methods described above require two passes through the population at each generation: one pass to compute the mean fitness (and, for sigma scaling, the standard deviation) and one pass to compute the expected value of each individual. Rank scaling requires sorting the entire population by rank—a potentially time-consuming procedure. Tournament selection is similar to rank selection in terms of selection pressure, but it is computationally more efficient and more amenable to parallel implementation. Two individuals are chosen at random from the population. A random number r is then chosen between 0 and 1. If $r < k$ (where k *is a* parameter, for example 0.75), the fitter of the two individuals is selected to be a parent; otherwise the less fit individual is selected. The two are then returned to the original population and can be selected again.

7. Evolving a Learning Rule:

David Chalmers [3] (1990) took the idea of applying genetic algorithms to neural networks in a different direction: he used GAs to evolve a good learning rule for neural networks. Chalmers limited his initial study to fully connected feedforward networks with input and output layers only, no hidden layers. In general a learning rule is used during the training procedure for modifying network weights in response to the network's performance on the training data. At each training cycle, one training pair is given to the network, which then produces an output. At this point the learning rule is invoked to modify weights. A learning rule for a single-layer, fully connected

feedforward network might use the following local information for a given training cycle to modify the weight on the link from input unit i to output unit j:

a_i : the activation of input unit *i*.

o_j : the activation of output unit *j*.

t_j :the training signal (i.e., correct activation, provided by a teacher) on output unit *j*.

w_{ij} : the current weight on the link from *i* to *j*.

The change to make in weight w_{ij}, Δw_{ij}, is a function of these values:

$$\Delta w_{ij} = f(a_i, o_j, t_j, w_{ij})$$

The chromosomes in the GA population encoded such functions.

Chalmers [3] made the assumption that the learning rule should be a linear function of these variables and all their pair wise products. That is, the general form of the learning rule was

$$\Delta w_{ij} = k_0(k_1 w_{ij} + k_2 a_i + k_3 o_j + k_4 t_j + k_5 w_{ij} a_i + k_6 w_{ij} o_j + k_7 w_{ij} t_j + k_8 a_i o_j + k_9 a_i t_j + k_{10} o_j t_j)$$

The $k_m (1 \le m \le 10)$ are constant coefficients, and k_o is a scale parameter that affects how much the weights can change on any one cycle, (k_o is called the "learning rate.") Chalmers's assumption about the form of the learning rule came in part from the fact that a known good learning rule for such networks the "Widrow-Hoff" or "delta" rule—has the form

$$\Delta w_{ij} = \eta(t_j o_j - a_i o_j)$$

(where η) is a constant representing the learning rate. One goal of Chalmers's work was to see if the GA could evolve a rule that performs as well as the delta rule.

4.7 Genetic Operators:

The third decision to make in implementing a genetic algorithm is what genetic operators to use. This decision depends greatly on the encoding strategy. Where we will discuss crossover and mutation mostly in the context of bit-string encodings, and mention a number of other operators that have been proposed in the GA literature.

1. **Crossover:** It could be said that the main distinguishing feature of a GA is the use of crossover. Single-point crossover is the simplest form: a single cross-over position is chosen at random and the parts of two parents after the crossover position are exchanged to form two offspring. The idea here is, of course, to recombine building blocks (schemas) on different strings. Single-point crossover has some shortcomings, though. For one thing, it cannot combine all possible schemas. For example, it cannot in general combine instances of 11*****1 and ****11** to form an instance of ll**11*1. Likewise, schemas with long defining lengths are likely to be destroyed under single-point crossover. Eshelman, Caruana, and Schaffer [3] (1989) call this "positional bias": the schemas that can be created or destroyed by a crossover depend strongly on the location of the bits in the chromosome. Single-point crossover assumes that short, low-order schemas are the functional building blocks of strings, but one generally does not know in advance what ordering of bits will group functionally related bits together—this was the purpose of the inversion operator and other adaptive operators described above. Eshelman, Caruana, and Schaffer also point out that there may not be any way to put all functionally related bits close together on a string, since particular bits might be crucial in more than one schema. They point out further that the tendency of single-point crossover to keep short schemas intact can lead to the preservation of hitchhikers—bits that are not part of a desired schema but which, by being close on the string, hitchhike along with the beneficial schema as it reproduces. Many people have also noted that single-point crossover treats some loci preferentially, the segments exchanged between the two parents always contain the endpoints of the strings.

Most of the comments above also assume that crossover's ability to re-combine highly fit schemas is the reason it should be useful. Given some of the challenges we have seen to the relevance of schemas as a analysis tool for understanding GAs, one might ask if we should not consider the possibility that crossover is actually useful for some entirely

different reason (e.g., it is in essence a "macro-mutation" operator that simply allows for large jumps in the search space). I must leave this question as an open area of GA research for interested readers to explore. (Terry Jones [3] (1995) has performed some interesting, though preliminary, experiments attempting to tease out the different possible roles of crossover in GAs.) Its answer might also shed light on the question of why recombination is useful for real organisms (if indeed it is)—a controversial and still open question in evolutionary biology.

2. **Mutation:** A common view in the GA community, dating back to Holland's book *Adaptation in Natural and Artificial Systems,* is that crossover is the major instrument of variation and innovation in GAs, with mutation insuring the population against permanent fixation at any particular locus and thus playing more of a background role. This differs from the traditional positions of other evolutionary computation methods, such as evolutionary programming and early versions of evolution strategies, in which random mutation is the only source of variation. (Later versions of evolution strategies have included a form of crossover.)

4.8 Parameters for Genetic Algorithms:

The fourth decision to make in implementing a genetic algorithm is how to set the values for the various parameters, such as population size, crossover rate, and mutation rate. These parameters typically interact with one another nonlinearly, so they cannot be optimized one at a time. There is a great deal of discussion of parameter settings and approaches to parameter adaptation in the evolutionary computation literature—too much to survey or even list here. There are no conclusive results on what is best; most people use what has worked well in previously reported cases. Some of the experimental approaches people have taken to find the "best" parameter settings, are discussed below.

De Jong [3] (1975) performed an early systematic study of how varying parameters affected the GA's on-line and off-line search performance on a small suite of test

functions. Recall from chapter 4 that "on-line" performance at time t is the average fitness of all the individuals that have been evaluated over t evaluation steps. The off-line performance at time t is the average value, over t evaluation steps, of the best fitness that has been seen up to each evaluation step. De Jong's experiments indicated that the best population size was 50-100 individuals, the best single-point crossover rate was ~ 0.6 per pair of parents, and the best mutation rate was 0.001 per bit. These settings (along with De Jong's test suite) became widely used in the GA community, even though it was not clear how well the GA would perform with these settings on problems outside De Jong's test suite. Any guidance was gratefully accepted.

Somewhat later, Grefenstette [3] (1986) noted that, since the GA could be used as an optimization procedure, it could be used to optimize the parameters for another GA! (A similar study was done by Bramlette [3] (1991).) In Grefenstette's experiments, the "meta-level GA" evolved a population of 50 GA parameter sets for the problems in De Jong's test suite. Each individual encoded six GA parameters: population size, crossover rate, mutation rate, generation gap, scaling window (a particular scaling technique that I won't discuss here), and selection strategy (elitist or nonelitist). The fitness of an individual was a function of the on-line or off-line performance of a GA using the parameters encoded by that individual. The meta-level GA itself used De Jong's parameter settings. The fittest individual for on-line performance set the population size to 30, the crossover rate to 0.95, the mutation rate to 0.01, and the generation gap to 1, and used elitist selection. These parameters gave a small but significant improvement in on-line performance over De Jong's settings. Notice that Grefenstette's results call for a smaller population and higher crossover and mutation rates than De Jong's. The meta-level GA was not able to find a parameter set that beat De Jong's for off-line performance. This was an interesting experiment, but again, in view of the specialized test suite, it is not clear how generally these recommendations hold. Others have shown that there are many fitness functions for which these parameter settings are not optimal.

A big question, then, for any adaptive approach to setting parameters— including Davis's—is this: How well does the rate of adaptation of parameter settings match the rate of adaptation in the GA population? The feedback for setting parameters comes from the population's success or failure on the fitness function, but it might be difficult

for this information to travel fast enough for the parameter settings to stay up to date with the population's current state. Very little work has been done on measuring these different rates of adaptation and how well they match in different parameter-adaptation experiments. This seems to me to be the most important research to be done in order to get self-adaptation methods to work well.

4.9 Example of GAs :

As warmups to more extensive discussions of GA applications, here are brief examples of GAs in action on two particularly interesting projects.

1. Using GAs to Evolve Strategies for the Prisoner's Dilemma:

The Prisoner's Dilemma, a simple two-person game invented by Merrill Flood and Melvin Dresher [7] in the 1950s, has been studied extensively in game theory, economics, and political science because it can be seen as an idealized model for real-world phenomena such as arms races (Axelrod 1984; Axelrod and Dion [7] 1988). It can be formulated as follows: Two individuals (call them Alice and Bob) are arrested for committing a crime together and are held in separate cells, with no communication possible between them. Alice is offered the following deal: If she confesses and agrees to testify against Bob, she will receive a suspended sentence with probation, and Bob will be put away for 5 years. However, if at the same time Bob confesses and agrees to testify against Alice, her testimony will be discredited, and each will receive 4 years for pleading guilty. Alice is told that Bob is being offered precisely the same deal. Both Alice and Bob know that if neither testifies against the other they can be convicted only on a lesser charge for which they will each get 2 years in jail.

Should Alice "defect" against Bob and hope for the suspended sentence, risking a 4-year sentence if Bob defects? Or should she "cooperate" with Bob (even though they cannot communicate), in the hope that he will also cooperate so each will get only 2 years, thereby risking a defection by Bob that will send her away for 5 years?

The game can be described more abstractly. Each player independently decides which move to make—i.e., whether to cooperate or defect. A "game" consists of each player's making a decision (a "move"). The possible results of a single game are summarized in a payoff matrix like the one shown in figure 1.3. Here the goal is to get as many points (as opposed to as few years in prison) as possible. (In figure 1.3, the payoff in each case can be interpreted as 5 minus the number of years in prison.) If both players cooperate, each gets 3 points. If player A defects and player B cooperates, then player A gets 5 points and player B gets 0 points, and vice versa if the situation is reversed. If both players defect, each gets 1 point. What is the best strategy to use in order to maximize one's own payoff? If you suspect that your opponent is going to cooperate, then you should surely defect. If you suspect that your opponent is going to defect, then you should defect too. No matter what the other player does, it is always better to defect. The dilemma is that if both players defect each gets a worse score than if they cooperate. If the game is *iterated* (that is, if the two players play several games in a row), both players' always defecting will lead to a much lower total payoff than the players would get if they cooperated. How can reciprocal cooperation be induced? This question takes on special significance when the notions of cooperating and defecting correspond to actions in, say, a real-world arms race (e.g., reducing or increasing one's arsenal).

		Player B	
		Cooperate	Defect
Player A	Cooperate	3,3	0,5
	Defect	5,0	1,1

Figure 1.3 The payoff matrix for the Prisoner's Dilemma (adapted from Axelrod [7] 1987). The two numbers given in each box are the payoffs for players A and B in the given situation, with player A's payoff listed first in each pair.

Robert Axelrod [7] of the University of Michigan has studied the Prisoner's Dilemma and related games extensively. His interest in determining what makes for a good strategy led him to organize two Prisoner's Dilemma tournaments (described in Axelrod 1984). He solicited strategies from researchers in a number of disciplines. Each

participant submitted a computer program that implemented a particular strategy, and the various programs played iterated games with each other. During each game, each program remembered what move (i.e., cooperate or defect) both it and its opponent had made in each of the three previous games that they had played with each other, and its strategy was based on this memory. The programs were paired in a round-robin tournament in which each played with all the other programs over a number of games. The first tournament consisted of 14 different programs; the second consisted of 63 programs (including one that made random moves). Some of the strategies submitted were rather complicated, using techniques such as Markov processes and Bayesian inference to model the other players in order to determine the best move. However, in both tournaments the winner (the strategy with the highest average score) was the simplest of the submitted strategies: TIT FOR TAT. This strategy, submitted by Anatol Rapoport [7], cooperates in the first game and then, in subsequent games, does whatever the other player did in its move in the previous game with TIT FOR TAT. That is, it offers cooperation and reciprocates it. But if the other player defects, TIT FOR TAT punishes that defection with a defection of its own, and continues the punishment until the other player begins cooperating again.

Chapter Five

Genetic Algorithms Versus Neural Network

5.1 Evolving Neural Network using Genetic Algorithm:

Neural networks are biologically motivated approaches to machine learning, inspired by ideas from neuroscience. Recently some efforts have been made to use genetic algorithms to evolve aspects of neural networks.

In its simplest "feed forward" form figure 5.1 shown, a neural network is a collection of connected activatable units ("neurons") in which the connections are weighted, usually with real-valued weights. The network is presented with an activation pattern on its input units, such a set of numbers representing features of an image to be classified (e.g., the pixels in an image of a handwritten letter of the alphabet). Activation spreads in a forward direction from the input units through one or more layers of middle ("hidden") units to the output units over the weighted connections. Typically, the activation coming into a unit from other units is multiplied by the weights on the links over which it spreads, and then is added together with other incoming activation. The result is typically thresholded (i.e., the unit "turns on" if the resulting activation is above that unit's threshold). This process is meant to roughly mimic the way activation spreads through networks of neurons in the brain. In a feed forward network, activation spreads only in a forward direction, from the input layer through the hidden layers to the output layer. Many people have also experimented with "recurrent" networks, in which there are feedback connections as well as feed forward connections between layers.

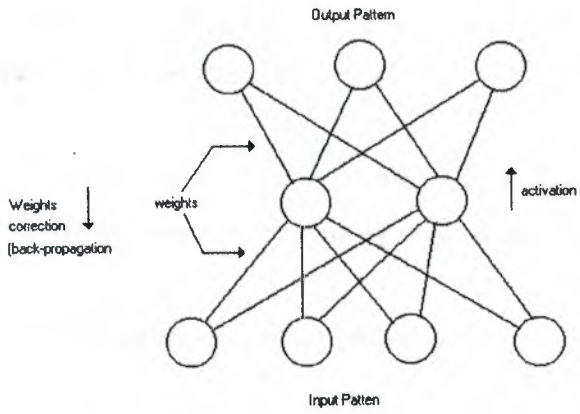


Fig 5.1. A schematic diagram of a simple feed forward neural network and the back-propagation process by which weight values are adjusted.

After activation has spread through a feedforward network, the resulting activation pattern on the output units encodes the network's "answer" to the input (e.g., a classification of the input pattern as the letter A). In most applications, the network learns a correct mapping between input and output patterns via a learning algorithm. Typically the weights are initially set to small random values. Then a set of training inputs is presented sequentially to the network. In the back-propagation learning procedure (Rumelhart, Hinton, [3] and Williams 1986), after each input has propagated through the network and an output has been produced, a "teacher" compares the activation value at each output unit with the correct values, and the weights in the network are adjusted in order to reduce the difference between the network's output and the correct output. Each iteration of this procedure is called a "training cycle," and a complete pass of training cycles through the set of training inputs is called a "training epoch." (Typically many training epochs are needed for a network to learn to successfully classify a given set of training inputs.) This type of procedure is known as "supervised learning," since a teacher supervises the learning by providing correct output values to guide the learning process. In "unsupervised learning" there is no teacher, and the learning system must learn on its own using less detailed (and sometimes less reliable) environmental feedback on its performance.

There are many ways to apply GAs to neural networks. Some aspects that can be evolved are the weights in a fixed network, the network architecture (i.e., the number of units and their interconnections can change), and the learning rule used by the network.

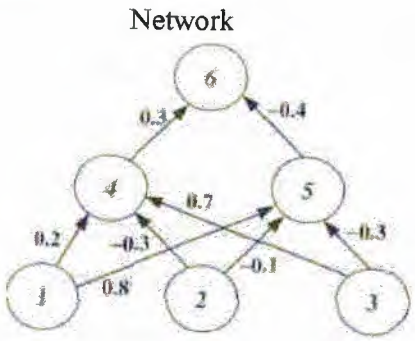
Here some different projects, each of which uses a genetic algorithm to evolve one of these aspects. (Two approaches to evolving network architecture will be described.)

5.2 Evolving Weights in a Fixed Network:

David Montana and Lawrence Davis [3] (1989) took the first approach— evolving the weights in a fixed network. That is, Montana and Davis were using the GA *instead* of back-propagation as a way of finding a good set of weights for a fixed set of connections. Several problems associated with the back-propagation algorithm (e.g., the tendency to get stuck at local optima in weight space, or the unavailability of a "teacher" to supervise learning in some tasks) often make it desirable to find alternative weight-training schemes.

Montana and Davis were interested in using neural networks to classify underwater sonic "lofargrams" (similar to spectrograms) into two classes: "interesting" and "not interesting." The overall goal was to "detect and reason about interesting signals in the midst of the wide variety of acoustic noise and interference which exist in the ocean." The networks were to be trained from a database containing lofargrams and classifications made by experts as to whether or not a given lofargram is "interesting." Each network had four input units, representing four parameters used by an expert system that performed the same classification. Each network had one output unit and two layers of hidden units (the first with seven units and the second with ten units). The networks were fully connected feed forward networks—that is, each unit was connected to every unit in the next higher layer. In total there were 108 weighted connections between units. In addition, there were 18 weighted connections between the non-input units and a "threshold unit" whose outgoing links implemented the thresholding for each of the non-input units, for a total of 126 weights to evolve.

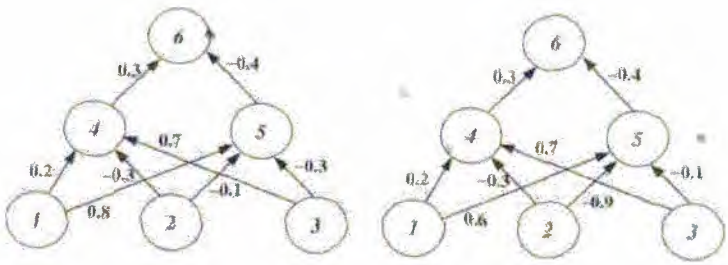
The GA was used as follows. Each chromosome was a list (or "vector") of 126 weights. Figure 5.2 shows (for a much smaller network) how the encoding was done: the weights were read off the network in a fixed order (from left to right and from top to bottom) and placed in a list. Notice that each "gene" in the chromosome is a real number rather than a bit. To calculate the fitness of a given chromosome, the weights in the chromosome were assigned to the links in the corresponding network, the network was run on the training set (here 236 examples from the database of lofargrams), and the sum of the squares of the errors (collected over all the training cycles) was returned. Here, an "error" was the difference between the desired output activation value and the actual output activation value. Low error meant high fitness.



Chromosome: (0.3 -0.4 0.2 0.8 -0.3 -0.1 0.7 -0.3)

Figure 5.2 Illustration of Montana and Davis's encoding of network weights into a list that serves as a chromosome for the GA. The units in the network are numbered for later reference. The real-valued numbers on the links are the weights.

Before mutation **After mutation**

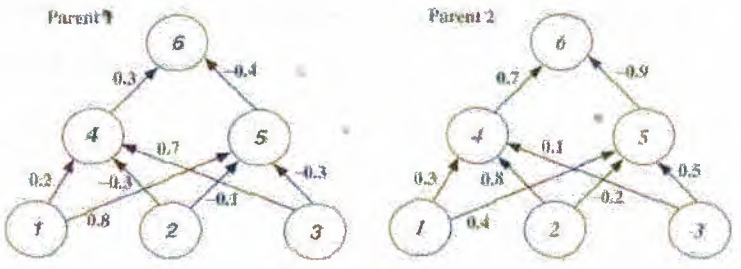


(0.3 -0.4 0.2 0.8 -0.3 -0.1 0.7 -0.3) **(0.3 -0.4 0.2 0.6 -0.3 -0.9 0.7 -0.1)**

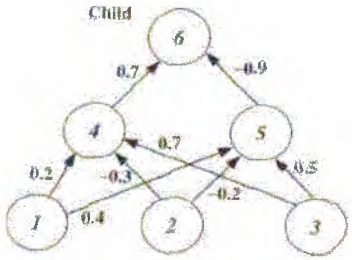
Figure 5.3 Illustration of Montana and Davis's mutation method. Here the weights on incoming links to unit 5 are mutated.

An initial population of 50 weight vectors was chosen randomly, with each weight being between -1.0 and +1.0. Montana and Davis tried a number of different genetic operators in various experiments. The mutation and crossover operators they used for their comparison of the GA with back-propagation are illustrated in figures 5.3 and 5.4. The mutation operator selects *n* non-input units and, for each incoming link to those units, adds a random value between -1.0 and +1.0 to the weight on the link. The crossover operator takes two parent weight vectors and, for each non-input unit in the offspring vector, selects one of the parents at random and copies the weights on the incoming links from that parent to the offspring. Notice that only one offspring is created.

The performance of a GA using these operators was compared with the performance of a back-propagation algorithm. The GA had a population of 50 weight vectors, and a rank-selection method was used. The GA was allowed to run for 200 generations (i.e., 10,000 network evaluations). The back-propagation algorithm was allowed to run for 5000 iterations, where iteration is a complete epoch (a complete pass through the training data). Montana and Davis reasoned that two network evaluations under the GA are equivalent to one back-propagation iteration, since back-propagation on a given training example consists of two parts—the forward propagation of activation (and the calculation of errors at the output units) and the backward error propagation (and adjusting of the weights). The GA performs only the first part. Since the second part requires more computation, two GA evaluations takes less than half the computation of a single back-propagation iteration.



(0.3 -0.4 0.2 0.8 -0.3 -0.1 0.7 -0.3) **(0.7 -0.9 0.3 0.4 0.8 -0.2 0.1 0.5)**



(0.7 -0.9 0.2 0.4 -0.3 -0.2 0.7 0.5)

Figure 5.4 Illustration of Montana and Davis's crossover method.

The results of the comparison are displayed in figure 5.5. Here one backpropagation iteration is plotted for every two GA evaluations. The x axis gives the number of iterations, and the y axis gives the best evaluation (lowest sum of squares of errors) found by that time. It can be seen that the GA significantly outperforms back-propagation on this task, obtaining better weight vectors more quickly.

This experiment shows that in some situations the GA is a better training method for networks than simple back-propagation. This does not mean that the GA will outperform back-propagation in all cases. It is also possible that enhancements of back-propagation might help it overcome some of the problems that prevented it from performing as well as the GA in this experiment. Schaffer, Whitley, and Eshelman (1992) point out that the GA has not been found to outperform the best weight-adjustment methods (e.g., "quickprop") on supervised learning tasks, but they predict that the GA will be most useful in finding weights in tasks where back-propagation and its relatives cannot be used, such as in unsupervised learning tasks, in which the error at each output unit is not available to the learning system, or in situations in which only sparse reinforcement is available. This is often the case for "neurocontrol" tasks, in which neural networks are used to control complicated systems such as robots navigating in unfamiliar environments.

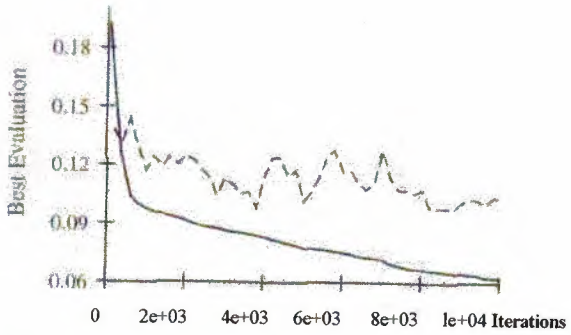
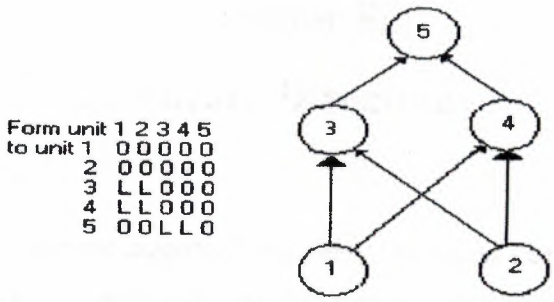


Figure 5.5 Montana and Davis's results comparing the performance of the GA with back-propagation.

5.3 Evolving Network Architectures:

Montana and Davis's [3] GA evolved the weights in a fixed network. As in most neural network applications, the architecture of the network—the number of units and their interconnections—is decided ahead of time by the programmer by guesswork, often aided by some heuristics (e.g., "more hidden units are required for more difficult problems") and by trial and error. Neural network researchers know all too well that the particular architecture chosen can determine the success or failure of the application, so they would like very much to be able to automatically optimize the procedure of designing an architecture for a particular application. Many believe that GAs are well suited for this task. There have been several efforts along these lines, most of which fall into one of two categories: direct encoding and grammatical encoding. Under direct encoding, network architecture is directly encoded into a GA chromosome. Under grammatical encoding, the GA does not evolve network architectures; rather, it evolves grammars that can be used to develop network architectures.



chromosome: 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 0 0 0 0 0 1 1 0

Figure 5.6 An illustration of Miller, Todd, and Hegde's representation scheme. Each entry in the matrix represents the type of connection on the link between the "from unit" (column) and the "to unit" (row). The rows of the matrix are strung together to make the bit-string encoding of the network, given at the bottom of the figure.

Chapter Six
Future Directions

As we have seen that genetic algorithms and neural networks can be a powerful tool for solving problems and for simulating natural systems in a wide variety of scientific fields. In examining the accomplishments of these algorithms, we have also seen that many unanswered questions remain. Finally we, summarize what the field of neural networks and genetic algorithms has achieved, and what are the most interesting and important directions for future research.

From the knowledge of problem-solving, scientific modeling, and theory we come to the following conclusions:

GAs and neural networks are promising methods for solving difficult technological problems, and for machine learning. More generally, they are a part of a new movement in computer science that is exploring biologically inspired approaches to computation. Advocates of this movement believe that in order to create the kinds of computing systems we need systems that are adaptable, massively parallel, able to deal with complexity, able to learn, and even creative—we should copy natural systems with these qualities. Natural evolution is a particularly appealing source of inspiration.

Genetic algorithms and neural networks are also promising approaches for modeling the natural systems that inspired their design. Most models using GAs are meant to be "gedanken experiments" or "idea models" (Roughgarden et al. [3] 1995) rather than precise simulations attempting to match real-world data. The purposes of these idea models are to make ideas precise and to test their plausibility by implementing them as computer programs (e.g., Hinton and Nowlan's [3] model of the Baldwin effect), to understand and predict general tendencies of natural systems (e.g.. Echo), and to see how these tendencies are affected by changes in details of the model (e.g., Collins and Jefferson's variations on Kirkpatrick's [3] sexual selection model). These models can allow scientists to perform experiments that would not be possible in the real world, and

NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Electrical and Electronic Engineering

GENETIC ALGORITHMS VERSUS NERUAL NETWORKS

Graduation Project
EE – 400

Student: Atique ur Rehman (981262)

Supervisor: Asst. Prof. Dr Adnan Khashman

Nicosia - 2001

Acknowledgements

I am very thankful to the following people:

First of all to **Asst. Prof. Dr Adnan Khashman** who gave me step by step guidance through out the project, I got acquainted with an entirely new and flourishing field of Neural Networks and Genetic Algorithms, I am thankful to my advisor, for his intellectual support, encouragement, enthusiasm which made this thesis possible.

I am also very thankful to my father **Khalil-Ur-Rehman** and my family who have supported, and sponsored me through out my life and encouraged me in every aspect of my life. What I am today is because of them. The greatness of my Father is unable to be explained in words.

I am also thankful to my friends Rihan and Jamal, who helped me a lot in getting the material for the thesis, surely true friends are blessing.

I dedicate my project to all the people mentioned above.

Abstract

Neural Networks and Genetic Algorithms are examples of the latest computer applications, which are under consideration by the scientists over the world. The combination of neural networks and genetic algorithms can be very helpful in creation of artificial intelligent systems. With the development of these two fields, machines that can behave like humans can be invented.

This thesis describes an investigation of combining neural networks and genetic algorithms into real applications.

With the passage of time and development in machine learning a Question Arises?

Will the humans be slaves of machines?

Table of Contents

Introduction

Overview:

Neural Networks is a popular Artificial Intelligent computer systems, inspired by the principles biological neural behavior, this technology is being applied to the computer systems for solving difficult problems, whose solutions require human intelligence. Along with the neural networks another interesting algorithms approach, inspired by the biological genetic behavior, genetic algorithms is being applied in complicated computer systems, along with neural networks.

Research Objectives:

The objectives of the work presented within the thesis are to investigate independently neural networks and genetic algorithms.In addition the benefits that can be achieved by integrating neural networks and genetic algorithms will also be discussed.

Thesis Structure:

Research Objectives include the following:

- In chapter one brief history of neural networks along with biological terminology, with a brief discussion of hybrid intelligent systems will be discussed.
- In chapter two brief history of genetic algorithm along with a brief discussion of search space, genetic operators and application of genetic algorithms will be discussed.
- In chapter three details about neural networks, neural learning, supervised and unsupervised learning, classification of neural networks will be discussed.
- In chapter four details about genetic algorithm, implementation, genetic encoding and selection methods, along with a brief example of prisoner dilemma will be discussed.

- In chapter five a detail example of implementing a genetic algorithm, in a neural network will be discussed.
- In chapter six future directions for implementation of neural networks and genetic algorithms will be discussed.
- Finally the thesis will be concluded, with final remarks.

Chapter One

Neural Networks

1.1 History of Neural Networks:

The progress of neurobiology has allowed researchers to build mathematical models of neurons to simulate neural behavior. This idea dates back to the early 1940s, when one of the first abstract models of a neuron was introduced by McCulloch and Pitts [1], (1943). Hebb [2] in 1949 proposed a learning law that explained how a network of neurons learned, Hebbs law stated that:

"When an axon of cell A is near enough to excite cell B and persistently takes part in firing it some growth process or metabolic changes takes place in one or both cells such that As efficiency increased".

The law proposed by Hebb formed the basis of modern neural network research. Later, Minsky and Papet [1] (1946) pointed out theoretical limitations of single-layer neural network modes in their landmark book Perceptrons. Due to this pessimistic projection, research on artificial neural network lapsed into an eclipse for nearly two decades. Despite the negative atmosphere, some researchers still continued their research and produced meaningful results. For example, Anderson (1977) and Grossberg [1] (1980) did important work on psychological models. Kohonen [1] (1977) developed associative memory models.

In early 1980s the neural network approach was restructured, Hop Field [1] in (1982) introduced the idea of energy minimization in physics into neural networks. In the middle of 1980s, the book Parallel Distributed Processing by Rumelhart [1] and McClelland [1] (1986) generated great impacts on computer, cognitive and biological sciences. The back prorogation-learning algorithm developed by Rumelhart offers a

powerful solution to training a multi-layer neural network and shattered the curse imposed on perceptrons.

1.2 Knowledge-based Information Processing:

Knowledge based information system can be defined as:

" A knowledge-based system is a computer program that acquires, represents, and uses knowledge for a specific purpose".

Its basic structure is as shown in fig below, which consists of a knowledge base which stores knowledge and an inference knowledge engine which makes inference using the knowledge. A conventional computer program is characterized by algorithmic processing data. In this programming paradigm, the knowledge concerning how to do things is enclosed as a bunch of procedures, which are executed step by step to deal with the data entered. In knowledge-based programming, on the other hand, we represent what we know in a declarative manner and knowledge in invoked under a certain inference strategy or driven heuristically.

Another important distinction between the two programming paradigms is the feature of separating knowledge from the control. In knowledge-based systems, knowledge is stored in the knowledge base while control strategies reside in the separate inference engine. This separation benefits the development and maintenance of the system because when knowledge is updated, the inference engine can be left alone, and when the inference process in changed; the knowledge base is not affected. Because of separation, different inference engines can run a knowledge base and an inference engine can drive different knowledge bases. As a consequence, a lot of time and effort can be saved using the knowledge-based approach. The comparison of knowledge and data-oriented information processing is provided in table below.

Knowledge-Based processing	Data-Oriented processing
Declarative knowledge	Procedural knowledge.

Separating control from knowledge.	Integrating control and knowledge.
Strategic and heuristic processing.	Algorithmic processing.
Symbolic processing (dominant).	Numerical processing (dominant).
Explanation capability.	No explanation.

Table 1.1 Comparison of knowledge-based and data-oriented information processing.

1.3 Neural Information Processing:

Biological neurons transmit electrochemical signals over neural pathways. Each neuron receives signals form other neurons through special junctions called ***synapses.*** Some inputs tend to excite neuron; other tends to inhibit it. When the cumulative effect exceeds a threshold, the neuron fires and sends a signal down to other neurons. An artificial neuron models these simple biological characteristics. Each artificial neuron receives a set of inputs. Each input is multiplied by a weight analogous to a synaptic strength. The sum of all weighted inputs determines the degree of firing called the *activation level* (In neural network, connection weights and activations are sometimes reffered to as LTM (long-term memory) and STM (short-term memory), respectively). Notation ally, each input X_i is modulated by weight W_i and the total input is expressed as,

$$\sum X_i W_i$$

or in vector form, **X.W.**

The input signal is further processed by an activation function to produce the output signal, which if not zero, is transmitted along. The activation function can be a threshold function or a smooth function like a sigmoid or a hyperbolic tangent function.

The neural network is represented by a set of nodes and arrows, which is a fundamental concept in graph theory. A node corresponds to a neuron, and an arrow corresponds to a connection along with the direction of signal flow between neurons. As illustrated in

Fig below some nodes are connected to the system input and others are connected to the system output for information processing.

Neural networks solve problems by self-learning and self-organization. They derive their intelligence from the collective behavior of simple computational mechanisms at individual neurons. Computations advantages offered by neural networks include:

- Knowledge acquisition under noise and uncertainty; Neural networks can performs generalization, abstraction, and extraction of statistical properties form the data.
- Flexible knowledge representation: Neural networks can create their own representation by self-organization.
- Efficient knowledge processing: Neural nets can carry out computation in parallel. It is know as parallel-distributed processing, or PDP (Rumelhart [1] and McClelland 1986). Special hardware devices have been manufactured which exploit this advantage. Thus, real-time operation is feasible. Notice that training a neural network may be time-consuming, but once it is trained, it can operate very fast.
- Fault tolerance: Through distributed knowledge representation and redundant information encoding, the system performance degrades gracefully in response to faults (errors).

Neural networks can recognize, classify, convert, and learn patterns. A pattern is a quantitative description of an object or concept or event. A pattern class is a set of pattern sharing some common properties. Pattern recognition refers to the categorization of input data into identifiable classes by recognizing significant features or attributes of the data.

1.4 Brain As A Neural Network:

Human brain is made up of a vast network of computing elements, called neurons, coupled with sensory receptors (affecters) and effectors. The average human brain, roughly three pounds in weight and 90 cubic inches in volume, is estimated to contain

about 100 billion cells of various types. A neuron is a special cell that conducts electrical signal, and there are about 10 billion neurons in the human brain. The remaining 90 billion cells are called glial or glue cell, and these sever as support cells for the neurons. Brain organizes the huge number of neurons (also referred to as cells because glial cells are not of interest here) each with weak computing power, into a massively parallel complex network in which the neurons interact with each other dynamically to produce a powerful information processor.

1.5 Hybrid Intelligence:

Integration of symbolic AI and neural network results in a so-called Hybrid intelligent system. Under this approach, the fundamental assumption on intelligence is as follows:

- Neither the physical symbol system nor the neural network is a necessary means of general intelligent action.
- The symbolic level and the connectionist level represent two different levels of abstraction for intelligent process.
- Knowledge is power. Every intelligent being should have knowledge in one form or another.

Hybrid intelligence is a biological plausible notion. Recall that humans store knowledge in certain complex molecules such as genes and proteins, which determine what we are and how we behave, and at the same time, we have nerve system to coordinate our behavior.

Examples of research in this area include:

- **Knowledge-based neural network**: Neural networks are built based on domain knowledge or theory. In this construct, neural networks model some aspects such as noise and uncertainty which knowledge is not dealing with.

- **Translation of neural network knowledge into symbolic knowledge**: This is important for interpreting neural network, explaining neural network

behavior, and learning knowledge under noise and uncertainty. The idea can also be applied to regularize the neural network and to prevent it form over fitting the data.

- **Learning by combining knowledge and adaptation:** It is concerned with how to build a better learning system that using knowledge or adaptation alone, how to build an incremental learning system, and how to build a useful discovery system. The central idea is to use knowledge as the initial crystal and then grow the crystal by adaptation.

- **Connectionist Symbol processing:** It bears on how to represent symbolic information or knowledge in the framework of connectionists, how to process the information accordingly, and how to retrieve the information. The advantages of this approach include fault tolerance, space sharing, and special processing strategies offered by the distributed representation of connectionists.

- **Hybrid Intelligent System:** Such systems possess knowledge-based components and neural networks, which are integrated in a certain manner so that each component performs the tasks for which it is best, suited.

- **Expert Networks:** They refer to neural networks that can perform as well as human experts. Explanation is an important issue for designing such systems.

Chapter Two
Genetic Algorithms

2.1 Overview:

Science arises from the very human desire to understand and control the world. Over the course of history, we humans have gradually built up the grand edifice of knowledge that enables us to predict, to varying extents, the weather, the motions of the planets, solar and lunar eclipses, the control of diseases, the rise and fall of economic growth, the stages of language development in children, and a vast panorama of other natural, social, and cultural phenomena. Most recently we have even come to understand some fundamental limits to our abilities to predict. Over the eons we have developed increasingly complex means to control many aspects of our lives and our interactions with nature, and we have learned often the hard way, the extent to which other aspects are uncontrollable.

The goal of creating artificial intelligence and artificial life can be traced back to the very beginning of computer age. The earliest computer scientists Alan Turing, John von Neumann, Norbert Wiener [3], and others were motivated in large part by visions of imbuing computer programs with intelligence, with the life-like ability to self-replicate, and with the adaptive capability to learn and to control their environments. These pioneers of computer science were as much interested in biology and psychology as in electronics, and they looked to natural systems as guiding metaphors for how to Brachiates their visions. It should be no surprise, then, that from the earliest days computers were applied not only to calculation missile trajectories and deciphering military codes but also to modeling the brain, mimicking human learning, and simulating biological evolutions.

2.2 Brief History:

Charles Darwin, 1809-1882

Genetic algorithms are appropriate for problems, which require optimization with respect to some computable criterion. This paradigm can also be applied to data mining problems. Here the quantity to be minimized is often the number of classification errors on a training set. Ultragem [4] has developed proprietary techniques for efficiently representing and evolving classification rules using the genetic algorithm paradigm.

Unlike natural evolution, genetic algorithms do not require millions of years to produce results. However, the system may need to run for many hours or even days. Large, complex problems require a fast computer in order to obtain good solutions in a reasonable amount of time. Mining of large datasets by genetic algorithms has only recently become practical due to the availability of affordable high-speed computers such as the DEC Alpha.

In the 1950s and 1960s several computer scientists independently studied evolutionary systems with the idea that evolution could be used as optimization tool for engineering problems. The idea in all these systems was to evolve a population of candidate solution to a given problem, using operators inspired by natural genetic variation and natural selection.

In the 1960s, Rechenberg [3] (1965,1973) introduced "evolutionary strategies", a method he used to optimize real valued parameters for devices such as airfoils. This idea was further developed by Schwefel [3] (1975, 1977). The field of evolution of strategies has remained an active area of research, mostly developing independently from the field of genetic algorithms.

Several other people working in 1950s and the 1960s developed evolution inspired algorithms for optimization and machine learning. Box (1957), Friedman (1959), Bledsoe (1962), Bremermann (1962), and Reed, Toombs, and Baricelli [3] (1967) all worked in this area, though their work has been given little or none attention or follow

up that evolution strategies, evolutionary programming, and genetic algorithms have seen.

Genetic algorithms were invented by John Holland [3] in 1960s and were developed by Holland and his students and colleagues at the University of Michigan in 1960s and 1970s. In contrast with evolution strategies and evolutionary programming, Holland's original goal was not to design algorithms to solve specific problems, but rather to formally study the phenomena of adaptation as it occurs in nature and to develop ways in which the mechanism of natural adaptation might be imported into computer systems.

In the last several years there has been widespread interaction among researchers studying various evolutionary computation methods, and the boundaries between Genetic Algorithms, evolutionary strategies, evolutionary programming, and other evolutionary approaches have been broken down to some extent.

2.3 Biological Terminology:

The evolution of Genetic Algorithms is based on analogy with real biology and can be understood more precisely as:

All living organisms consist of cell, and each cell contains the same set of one or more *chromosomes* - strings of DNA – that serves as "blueprint" for the organism. A chromosome can be conceptually divided into *genes* – functional block of DNA, each of which encodes a particular protein. Very roughly, one can think of gene as encoding a *trait*, such as eye color. The different possible "settings" for a trait (e.g. blue, brown, hazel) are called *alleles*. Each gene is located at a particular locus (position) on the chromosome.

In genetic algorithms, the term chromosome typically refers to candidate solution to a problem, often encoded as a bit string. The "genes" are either single bits or short blocks of adjacent bits that encode a particular element of the candidate solution (e.g. in the context of multi-parameter function optimization the bits encoding a particular

parameter might be considered to be a gene). An allele in a bit string is either 0 or 1; for larger alphabets more alleles are possible at each locus. Crossover typically consists of exchanging genetic material between two single chromosome haploid parents.

2.4 Search Space:

If we are solving some problem, we are usually looking for some solution, which will be the best among others. The space of all feasible solutions (it means objects among those the desired solution is) is called **search space** (also state space). Each point in the search space represents one feasible solution. Each feasible solution can be "marked" by its value or fitness for the problem. We are looking for our solution, which is one point (or more) among feasible solutions - that is one point in the search space.

The looking for a solution is then equal to a looking for some extreme (minimum or maximum) in the search space. The search space can be whole known by the time of solving a problem, but usually we know only a few points from it and we are generating other points as the process of finding solution continues.

2.5 Elements of Genetic Algorithms:

It turns out that there is no rigorous definition of "genetic algorithm" accepted by all in the evolutionary-computation community that differentiates GAs from other evolutionary computation methods. However, it can be said that most methods called "GAs" have at least the following elements in common: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring. Inversion—Holland's fourth element of GAs is rarely used in today's implementations, and its advantages, if any, are not well established.

The chromosomes in a GA population typically take the form of bit strings. Each locus in the chromosome has two possible alleles: 0 and 1.Each chromosome can be thought of as a point in the search space of candidate solutions. The GA processes populations of chromosomes, successively replacing one such population with another. The GA most often requires a fitness function that assigns a score (fitness) to each chromosome in the current population. The fitness of a chromosome depends on how well that chromosome solves the problem at hand.

2.5.1 Examples of Fitness Functions:

One common application of GAs is function optimization, where the goal is to find a set of parameter values that maximize, say, a complex multi-parameter function. As a simple example, one might want to maximize the real-valued one-dimensional function (Riolo [3] 1992). Here the candidate solutions are values of y, which can be encoded as bit strings representing real numbers. The fitness calculation translates a given bit string x into a real number y and then evaluates the function at that value. The fitness of a string is the function value at that point.

$$f(y) = y + |\sin(32.y)|, \qquad 0 < y < \pi$$

As a non-numerical example, consider the problem of finding a sequence of 50 amino acids that will fold to a desired three-dimensional protein structure. A GA could be applied to this problem by searching a population of candidate solutions, each encoded as a 50-letter string such as

IHCCVASASDMIKPVFTVASYLKNWTKAKGPNFEICISGRTPYWDNFPGI,

Where each letter represents one of 20 possible amino acids. One way to define the fitness of a candidate sequence is as the negative of the potential energy of the sequence with respect to the desired structure. The potential energy is a measure of how much physical resistance the sequence would put up if forced to be folded into the desired structure the lower the potential energy, the higher the fitness. Of course one would not want to physically force every sequence in the population into the desired structure and measure its resistance this would be very difficult, if not impossible. Instead, given a sequence and a desired structure (and knowing some of the relevant biophysics), one can estimate the potential energy by calculating some of the forces acting on each amino acid, so the whole fitness calculation can be done computationally.

These examples show two different contexts in which candidate solutions to a problem are encoded as abstract chromosomes encoded as strings of symbols, with fitness

functions defined on the resulting space of strings. A genetic algorithm is a method for searching such fitness landscapes for highly fit strings.

2.6 GA Operators:

The simplest form of genetic algorithm involves three types of operators selection, crossover, and mutation.

1. **Selection:** This operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected.

2. **Crossover:** this operator randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring. For example, the strings 10000100 and 11111111 could be crossed over after the third locus in each to produce the two offspring 10011111 and 11100100. The crossover operator roughly mimics biological recombination between two single-chromosome (haploid) organisms.

3. **Mutation:** This operator randomly flips some of the bits in a chromosome. For example, the string 00000100 might be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probability, usually very small (e.g. 0.001).

2.7 Applications of Genetic Algorithms:

Various kinds of Genetic Algorithms have been applied on different scientific and engineering problems and models. Some examples are:

- **Optimization:** GAs have been used in wide variety of optimization tasks, including numerical optimization and such combinatorial optimization problems as circuit layout and job-shop scheduling.

- **Automatic Programming:** GAs have been used to evolve computer programs for specific tasks, and to design other computational structures such as cellular automata and sorting networks.

- **Machine learning:** GAs have been used for many machine learning applications, including classification and prediction tasks, such as the prediction of weather or protein structure. Gas have also been used to evolve aspects of particular machine learning systems, such as weight for neural networks, rules for learning classifier systems or symbolic production systems, and sensors for robots.

- **Economics:** GAs have been used to model processes of innovation, the development of bidding strategies, and the emergence of economic markets.

- **Immune systems:** GAs have been used to model various aspects of natural immune systems, including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.

- **Ecology:** GAs have been used to model ecological phenomena such as biological arms races, host-parasite co evolution, symbiosis, and resource flow.

- **Population Genetics:** GAs have been used to study how individual learning and species evolution affect one another.

- **Evolution and Learning:** GAs have been used to study how individual learning and species evolution affect one another.

- **Social Systems :** GAs have been used to study evolutionary aspects of social systems, such as the evolution of social behavior in insects colonies, and more generally, the evolution of cooperative and communication in multi-agent systems.

This list gives a brief idea of the flavor of the kinds of things GAs have been used for, both in problem solving and in scientific contexts. Because of their success in these and other areas, interests in GAs has been growing rapidly in the last several years among researchers in many disciplines. The field of GAs has become sub discipline of computer science, with conferences, journals, and scientific society.

Chapter Three

Basic Neural Network Learning and Computational Models

3.1 Overview:

The neural network contains a large number of simple neuron like processing elements and large number of weighted connections between the elements. The weights on the connection encode the knowledge of a network. Though biologically inspired, many of the neural networks developed do not duplicate the operation of human brain. Some computational principles in these models are not even explicable from biological viewpoints.

In many tasks such as recognizing human faces and understanding speech, current AI systems cannot do better than humans. It is conjectured that the structure of brain is somehow suited to these tasks and not suited to tasks such as high-speed arithmetic operation.

The intelligence of a neural network emerges from the collective behavior of neurons, each of which performs only very limited operation. Even though each individual neuron works slowly, they can still quickly find a solution by working in parallel. This fact can explain why humans can recognize a visual scene faster than a digital computer, while an individual brain cell responds much more slowly than a digital cell in VLSI circuit.

3.2 Basic Concepts of Neural Network:

A neural network has a parallel-distributed architecture with a large number of nodes and connections. Each connection points from one node to another and is associated with a weight. A simple view of the network structure and behavior is given in fig 2.1. Construction of a neural network involves the following tasks.

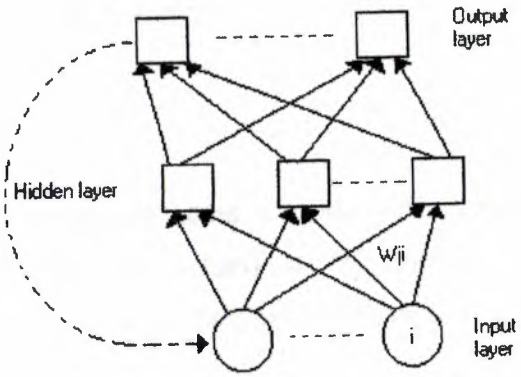


Fig 2.1 A neural network computational model.

- **Determine the network properties:** The network topology (connectivity), types of connections, the order of connections, and the weight range.

- **Determine the node properties:** The activation range and the activation (transfer) function.

- **Determine the system Dynamics:** The weight initialization scheme, the activation-calculating formula, and the learning rule.

1) **Network Properties:** The topology of a neural network refers to its framework as well its interconnection schemes. The framework is often specified by number of layers (or slabs) and the number of nodes per layer. The types of layers include:

- **The input layer:** The nodes in it are called input units, which encode the instance presented to the network for processing. For example, each input unit may be designated by an attribute value possessed by the instance.

- **The hidden layer:** The nodes in it are called the hidden units, which are not directly observable and hence hidden. They provide nonlinear ties for the network.

- **The output layer:** The nodes in it are called output units, which encode possible concepts, (or values) to be assigned to the instance under consideration. For example each output unit represents a class of objects.

The Input units do not process information; they simply distribute the information to the other units. Schematically, input units are drawn as circles as distinguished from processing elements like hidden units and output units, which are drawn as squares.

According to interconnection scheme, a network can either be feed forward or recurrent and its connection either symmetrical as asymmetrical, which are defined below.

- **Feed forward networks:** All connections point in one direction (from the input toward the output layer), or form left to right as shown in figure2.2.

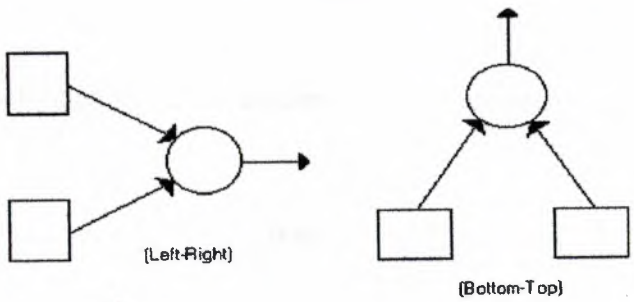


Fig 2.2 Single layer feedforward Perceptron.

- **Recurrent Networks:** There are feedback connections or loops, as shown in Fig 2.1.

- **Symmetrical Connection:** If there is a connection pointing from node 1 to node 2, then there is also a connection from node 2 to node 1, and the weights associated with the two connections are equal, or notationally,

$$W_{12} = W_{21}$$

- **Asymmetrical Connection:** If the connections are not symmetrical as described above then they are asymmetrical.

3.3 Node Properties:

The activation levels of nodes can be discrete (e.g., 0 and 1) or continuous across a range (e.g., [0,1] or unrestricted. This depends on the activation (transfer) function chosen. If it is a hard-limiting function, then the activation level are 0 or (–1) and 1. For a sigmoid function, the activation levels are limited to a continuous range of reals [0,1]. Figure 2.3 shows the sigmoid function F:

$$F(x) = \frac{1}{1+e^{-x}}$$

In case of a linear activation function, the activation levels are open.

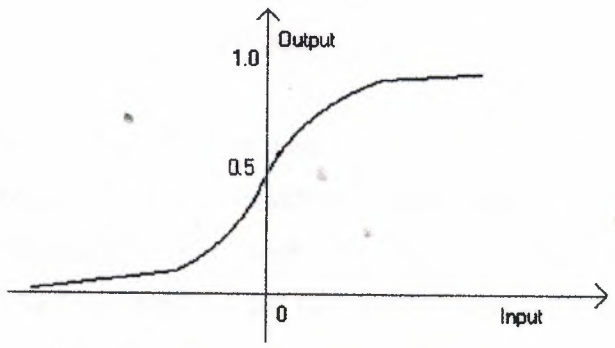


Fig 2.3 The sigmoid activation Function.

3.4 Inference and Learning:

Building an AI system based on the neural network approach will generally involve the following steps.

1. Select a suitable neural network based on the nature of problem.
2. Construct a neural network according to the characteristics of the application domain.
3. Train the neural network with the learning procedure of the selected model.
4. Use the trained network for making inference or solving problems. If the performance is not satisfactory, then go to one of the previous steps.

Familiarity with existing applications will help determine the appropriate network architecture and select the best-suited computational model for learning and inference. Learning is discussed in detail in **supervised** and **unsupervised** learning.

3.5 Learning:

In as much as great variety of human experience can be described as learning, the term machine learning is sometimes obscure. A somewhat more focused definition suggested by Hebert Simon [5] (1983) is based on notion of change.

"Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time".

Learning can refer to either acquiring new knowledge or enhancing or refining skills. Learning new knowledge includes acquisition of significant concepts, understanding of their meanings and relationships to each other and to domain concerned. The new knowledge should be assimilated and put in mentally usable form before it can be called "learned". Thus, knowledge acquisition is defined as learning new symbolic information combined with the ability to use that information effectively.

3.6 Historical Sketch:

Research and development in machine learning have seen several major evolutionary changes. Over the years, different paradigms with different emphasis on objectives have been pursued. Four major periods can be distinguished, each centering around a different paradigm:

Since 1940s and 1950s:

- **Paradigms:** Neural network; decision-theoretical learning.
- **Objectives:** Neural modeling; pattern recognition.
- **Examples:** McCulloch and Pitts (1943); Rosenblatt (1958); Samuel [5] (1959).

Since 1960s:

- **Paradigms:** Symbolic learning.
- **Objectives:** Concept acquisition; building knowledge-based expert systems.
- **Examples:** Winston (1975); Buchanan and Mitchell [5] (1978).

Since 1970s:

- **Paradigms:** Knowledge-intensive learning.
- **Objectives:** Exploration of various learning strategies.
- **Examples:** Mitchell, Keller, and Kedar-Cabelli [5] (1986).

Since 1980s:

- **Paradigms:** Neural network and connectionist learning; hybrid learning.
- **Objectives:** Neural computers; robust learning; massive parallelism.
- **Examples:** Rumelhart, McClelland, and PDP Group (1986); Goldberg [5] (1989).

3.7 Supervised Learning:

In a supervised learning process, the input data and its corresponding output are presented to the neural network. The neural network will according to the defined law

change its weight in order to be able to reproduce the correct output, when an input is applied.

Supervised learning algorithms utilize the information on the class membership of each training instance. This information allows supervised algorithms to detect pattern misclassification as a feedback to themselves. Error information contributes to the learning process by rewarding accurate classification and/or misclassifications – a process known as *credit and blame assignment.* It also helps eliminate implausible hypothesis.

3.8 Unsupervised Learning:

Unsupervised learning process requires only input vectors to train the network. On the input data is presented to the neural network, the weights are adjusted in an ordered way according to some figure of merit.

Unsupervised learning algorithms use unlabeled instances. They blindly or heuristically process them. Unsupervised learning algorithms often have less computational complexity and less accuracy than supervised learning algorithms. Unsupervised learning algorithms can be designed to learn rapidly. This makes unsupervised learning practical in many high-speed, real time environments, where we may not have enough time and information to apply supervised techniques. Unsupervised learning has also been used for scientific discovery.

Unsupervised learning refers to how neural networks modify their parameters in biologically plausible ways. In this learning mode, the neural network does not use the class membership of training instances. Instead, it uses information associated with a group of neurons to modify local parameters.

3.9 Neural Network Learning:

The neural network has been dubbed the "connectionist". It contains large number of simple neuron like processing elements and a large number of weighted connections between the elements. The weights on the connections encode the knowledge of a network. It uses a high parallel, distributed control, and can learn to adjust itself automatically.

3.9.1 Backpropagation:

The backpropagation network is probably the most well known and widely used among the current types of neural network systems available. The learning rule is known as backpropagation, which is a kind of gradient decent technique with backward error (gradient) propagation, as depicted in fig. The training instance set for the network must be presented many times in order for correct classification of input patterns. While the network can recognize patterns similar to those they have learned, they don't have the ability to recognize new patterns. This is true for all supervised learning networks. In order to recognize new patterns network needs to be retrained with these patterns along with previously known patterns. If only new patterns are provided for retraining, then old patterns may be forgotten. In this way, learning is not incremental over time.

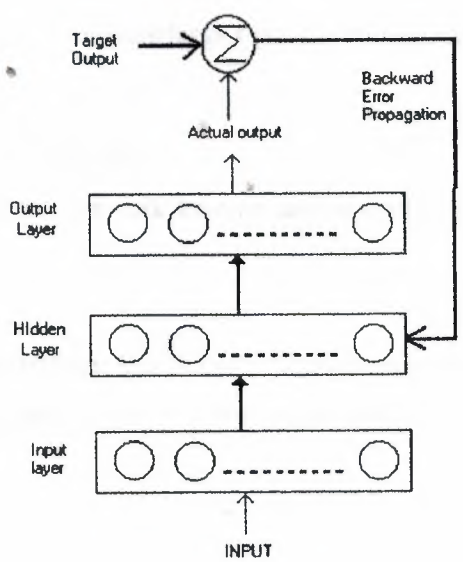


Fig 2.4 The back propagation network.

The backpropagation network is essence learns a mapping from a set of input patterns (e.g. extracted features) to a set of output patterns (e.g. class information). This network can be designed and trained to accomplish a wide variety of mappings. This ability comes form the nodes in hidden layer or layers of the network, which learns to respond to features, found in the input patterns. The features recognized or extracted by hidden units (nodes) correspond to the correlation of activity among different input units. As the network is trained with different examples, the network has the ability to generalize over similar features found in different patterns.

The back propagation network is capable of approximating arbitrary mappings. Furthermore, it can learn to estimate posterior probabilities ($p(w_i/\mathrm{x})$) for classification. The sigmoid function guarantees that the outputs are bounded between 0 and 1.

The back propagation network consists of one input layer, one output layer and one or more hidden layers. If n bits or n values describe the input pattern, then there should be n input units to accommodate it. The number of output units is like wise determined by how many bits or values are involved in output pattern.

The name back propagation comes from the fact that the error (gradient) or hidden units are derived form propagation backward the errors associated with output units. In back propagation network, the activation function chosen is the sigmoid function, which compresses the output value into the range between 0 and 1. The sigmoid function is advantageous in that it can accommodate large signals without saturation

- **Back Propagation Learning:**

The equation that describes the network training and operation can be divided into two categories.

1. **Feed forward Calculations:** Use in both training mode and operation mode.
2. **Error Back Propagation:** Use in training mode only.

Activation Function: Any activation function that is differentiable can be used in Back propagation algorithm.

- Linear Function with adjustable gain.
- Sigmoid Function (Squashing Function).

a) Feed forward Calculations:

Normalization of the input data prior training is necessary. The value of input data into input layer must be in the range (0-1).

Input Layer (*i*): The output of each input neuron is exactly equal to the normalized input.

$$\text{Input-layer} = \text{Output}$$

$$O_i = I_i$$

Hidden Layer (*h*): The signal presented to a neuron in the hidden layer is equal to the sum of all outputs of the input layer multiplied by their associated weights.

Hidden Layer Input:

$$I_h = \sum_i W_{hi} O_i$$

Each output of a hidden nurode is calculated using the SIGMOID function.

Output Layer (*j*): Similar to Hidden Layer Calculation.

Output layer Input (*j*):

$$I_j = \sum_i W_{jh} O_h$$

These equations describe the feedforward calculations, which can be used in both training and running phases.

b) Error Back Propagation Calculations:

Vital elements in these calculations are:

Error Signal: The definition of network error is the difference between the output value that an output neuron is supposed to have (Target value, Tj), and the value it actually has as a result of feed forward calculations (O_j).

$$E_p = \sum_{j=1}^{n_j} (T_{pj} - O_{fp})^2$$

P: denotes what the value is for a given pattern.

The aim of training a Neural Network is to minimize this error over all training patterns. The output of a neuron in the output layer is a function of its input $O_j = f(I_j)$. The first derivative of this function $f(I_j)$ is an important element in error back propagation. For output layer neurons, a quantity called the error signal is represented by Δ_j which is defined as,

$$\Delta_j = f'(I_j)(T_j - O_j)$$
$$\Delta_j = (T_j - O_j)O_j(1 - O_j)$$

The error value is propagated back and weights adjustments are made.

There are two essential parameters that affect the learning of a neural network:

1. Learning co-efficient η which defines the learning power of a neural network.
2. Momentum factor α, which defines the learning power of a neural network.

The effect of these parameters is described by the following equations.

Output Layer Weights Update: The weights that feed the output layer (W_{jh}) are updated using the following equations. This also includes the bias weights at the output layer. However in order to prevent the network getting caught in local minima, the momentum term is also added.

$$W_{jh}(new) = W_{jh}(old) + \eta \Delta_j O_j$$

or with momentum rate

$$W_{jh}(new) = W_{jh}(old) + \eta \Delta_j O_j + \alpha \left[\delta W_{jh}(old) \right]$$

Hidden Layer Weights Update: Similar to output layer weights update but the Delta error will be different. Error for Hidden layer is defined by the following equation.

$$\Delta_h = O_h (1 - O_h) \sum_{j=0}^{n_j} W_{jh} \Delta_j$$

Weights-adjustments:

$$W_{hi}(new) = W_{hi}(old) + \eta \Delta_h O_i + \alpha \left[\delta W_{hi}(old) \right]$$

All the equations describe the mathematical foundations for Back Propagation Learning Algorithm.

3.10 Classification Models:

Neural Networks can be classified according to the way they learn, learning can be performed on a Supervised Or Unsupervised basis.

- **Supervised Learning Models:**
  1. The Perceptron.
  2. The Back Propagation Learning Algorithm.
  3. The Hop Field Algorithm.
  4. The Hamming Algorithm.

- **The Perceptron:** This can be trained and can make decisions. During the training phase, pairs of input and output vectors are used to train the network. While each input vector, the output vector is compared with a desired output (target) as shown in fig 2.4, and the error between the actual and desired output vectors is used to update the weights.

- **Back Propagation:** A multi-layer network can be trained using the back propagation-learning algorithm. This is done by presenting pairs of input and output vectors. The actual output is compared with the target. If there is no difference the weights do not change, otherwise the weights are adjusted to reduce the error difference. This learning algorithm propagates back the error through the multi-layer to update the weights.

- **Hop Field Network:** A Hop field network is essentially used with binary numbers. Weights are initialized using training samples. In the decision making phase, the test data is presented to the net at certain time, following initialization the Hop field Network iterates in discrete time steps using some mathematical function, and the network is considered to have converged when the outputs no longer change on successive iterations.

- **Hamming Network:** It is similar to Hop field network, but it consists of four layers.

L1: Input Layer.
L2: Calculates matching scores.
L3: Feed back as in Hop field.
L4: Output Layer.

- **Unsupervised Learning Models:**
  1. Kohonen's Self-Organizing Maps.
  2. Competitive Learning.

3. Adaptive Learning.

- **Kohonen's Learning:** Kohonen [2] suggested that that one of the important mechanism in the human brain is placement of Neurons in an orderly manner. Kohonen's learning algorithm creates a feature map by adjusting weights from input vectors to output vectors in a two layer network. The first layer is input. the second is competitive layer. The tow layers are fully connected. Input vectors are presented sequentially to layer one. Each unit computes the dot product of its weight with the input vectors. The unit with the highest dot product is declared the winner. This and its neighbors are the only units allowed to learn.

- **Competitive Learning:** The simplest way to implement competitive learning is where each unit in the hidden or output layers receives input from all the units in the preceding layer. With in the layer units are broken down into a set of inhibitory clusters. The units with in the cluster compete with in one another to respond to data appearing at input layer. The more strongly and particular unit responds to incoming stimulus the more it inhibits other units with in the cluster. The unit learns by shifting a fraction of its weight from its inactive lines. The main disadvantage of competitive learning is the loss of previous learning's.

- **Adaptive Resonance Theory (ART):** ART is divided into two methods.

    1. Accepts only binary.
    2. Accepts binary and continuous input.

Chapter Four
Genetic Algorithm Implementation

4.1 How do Genetic Algorithms Works?

Although genetic algorithms are simple to describe and program, their behavior can be complicated, and many open questions exist about how they work and for what types of problems they are best suited. Much work has been done on theoretical foundations of GAs.

The tradition theory of GAs (first formulated in Holland 1975) assumes that, a very general level of description, GAs work by discovering, emphasizing, and recombination 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 – combination of bit values that confer higher fitness on the strings in which they are present.

Holland [6] (1975) introduced the notion of schemas (or schemata) to formalize the information notion of "building blocks". A schema is a set of bit strings that can be described by a template made up of ones, zeros, and asterisks, the asterisks representing wild cards (or "don't cares"). For example, the schema H=1****1 represents the set of all 6-bit strings that begin and end with 1. The strings that fit this template (e.g. 100111 and 110011) are said to be instances of H. The schema H is said to have two defined bits (non-asterisks) or, equivalent, to be or order 2. Its defining length (the distance between its outermost defined bits) is 5. Here the term "schema" is used to donate both a subset of strings represented by such a template itself.

Note that not every possible subset of the set of length-l bit strings can be described as a schema; in fact, the huge majority cannot. There are 2^l possible strings of length l, and thus 2^{2l} possible subsets of strings, but there are only 3^l possible schemas. However, a central tenet of traditional GA theory is that schemas are – implicitly – the building

blocks that the GA processes effectively under the operators of selection, mutation, and single-point crossover.

How do the GA process schemas? Any given bit string of length l is an instance of 2^l different schemas. For example, the string 11 is an instance of **(all four possible bit strings of length 2), *1, 1*, and 11 (the schema that contains only one string, 11). Thus, any given population of n strings contains instances of between 2^l and $n \times 2^l$ different schemas. If all the strings are identical, then there are instances of exactly 2^l different schemas; otherwise, the number is less than or equal to $n \times 2^l$. This means that, at a given generation, while the GA is explicitly evaluating the fitness of the n strings in the population, it is actually implicitly estimating the average fitness of a much larger number of schemas, where the average fitness of a schema is defined to be the average fitness of possible instances of that schema. For example, in a randomly generated population of n strings, on average half strings will be instances of 1***--* and half will be instances of 0***---0. The evaluation of approximately n/2 strings that are instances of 1**---* give an estimate of the average fitness of that schema. Just as schemas are not explicitly represented or evaluated by the GA, the estimates of schema average fitness are not calculated or stored explicitly by the GA. However, as will be seen below, the GAs behavior, in term of the population, can be described as though it actually calculating and storing these averages.

We can calculate the approximate dynamics of this increase decrease in schema instances by using the following equation:

$$E(m(H,t+1)) = \sum_{x \in H} f(x) / \bar{f}(t)$$
$$= (\hat{u}(H,t) / \bar{f}(t)) m(H,t) \qquad (1.1)$$

Where H is considered as schema with at least one instance present in the population at time t, m(H,t) be the number of instances present in the population at time t, and u(H,t) be the observed average fitness of H at time t.

The disruptive effects of mutation can be quantified as follow; let p_m be the probability of any bit being mutated. Then $S_m(H)$, the probability that schema H will survive under and instance of H, is equal to $(1-p_m)^{O(H)}$, where o(H) is order of H. (i.e. the number of defined bits in H).

The disruptive effects can be used to amend equation 1.1, and can be defined as:

$$E(m(H,t+1)) = \frac{\hat{u}(H,t)}{\bar{f}(t)} m(H,t)\left(1 - p_c \frac{d(H)}{l-1}\right)\left[(1-p_m)^{O(H)}\right] \qquad (1.2)$$

This is known as Schema Theorem. It describes the growth of a schema from one generation to the next.

The Schema Theorem and some of its purported implications for the behavior of GAs have been subjected of much critical discussion in the GA community. These criticisms and the new approaches are discussed in detail in the coming sections.

4.2 Implementing a Genetic Algorithm:

The case studied earlier illustrated that when one wants to apply the GA to a particular problem, one faces a huge number of choices about how to proceed, with little theoretical guidance on how to make them.

John Holland's simple GA inspired all subsequent GAs and provided the basis for theoretical analysis of GAs. For real problem solving and modeling, however, it is clear that the simple GA is limited in its power in several respects. Not all problems should use bit-string encodings, fitness-proportionate selection is not always the best method, and the simple genetic operators are not always the most effective or appropriate ones. Furthermore, the simple GA leaves out many potentially useful ideas from real biology, several of which were proposed for use in GAs by Holland (1975) but have not been examined systematically until recently.

In this section we will survey some implementation issues for GAs and some sophisticated GA techniques, including self-adapting GAs. Of course, this survey is by no means complete—although GA researchers speak informally of "the GA," anyone who has little idea about GAs will notice that there are actually as many different GAs as there are GA projects.

4.3 When Should a Genetic Algorithm be used?

The GA literature describes a large number of successful applications, but there are also many cases in which GAs perform poorly. Given a particular potential application, how do we know if a GA is good method to use? There is no rigorous answer, though many researchers share the intuitions that if the space to be searched is large/ is known not to be perfectly smooth and unimodal (i.e., consists of a single smooth "hill"), or is not well understood, or if the fitness function is noisy, and if the task does not require a global optimum to be found—i.e., if quickly finding a sufficiently good solution is enough—a GA will have a good chance of being competitive with or surpassing other "weak" methods (methods that do not use domain-specific knowledge in their search procedure). If a space is not large, then it can be searched exhaustively, and one can be sure that the best possible solution has been found, whereas a GA might converge on a local optimum rather than on the globally best solution. If the space is smooth or unimodal, a gradient-ascent algorithm such as steepest-ascent hill climbing will be much more efficient than a GA in exploiting the space's smoothness. If the space is well understood (as is the space for the well-known Traveling Salesman problem, for example), search methods using domain-specific heuristics can often be designed to outperform any general-purpose method such as a GA. If the fitness function is noisy (e.g., if it involves taking error-prone measurements from a real-world process such as the vision system of a robot), a one-candidate-solution-at-a-time search method such as simple hill climbing might be irrecoverably led astray by the noise, but GAs, since they work by accumulating fitness statistics over many generations, are thought to perform robustly in the presence of small amounts of noise.

4.4 Encoding a Problem for Genetic Algorithm:

As for any search and learning method, the way in which candidate solutions are encoded is a central, if not the central, factor in the success of a genetic algorithm. Most GA applications use fixed-length, fixed-order bit strings to encode candidate solutions. However, in recent years, there have been many experiments with other kinds of encoding.

1. Binary Encodings:

Binary encoding (i.e., bit strings) are the most common encodings for a number of reasons. One is historical: in their earlier work, Holland and his students concentrated on such encodings and GA practice has tended to follow this lead. Much of the existing GA theory is based on the assumption of fixed-length, fixed-order binary encodings. Much of that theory can be extended to apply to nonbinary encodings, but such extensions are not as well developed as the original theory. In addition, heuristics about appropriate parameter settings (e.g., for crossover and mutation rates) have generally been developed in the context of binary encodings.

Holland [3] (1975) gave a theoretical justification for using binary encodings. He compared two encodings with roughly the same information-carrying capacity, one with a small number of alleles and long strings (e.g., bit strings of length 100) and the other with a large number of alleles and short strings (e.g., decimal strings of length 30). He argued that the former allows for a higher degree of implicit parallelism than the latter, since an instance of the former contains more schemas than an instance of the latter (2^{100} versus 2^{30}). (This schema-counting argument is relevant to GA behavior only insofar as schema analysis is relevant, which, as I have mentioned, has been disputed).

In spite of these advantages, binary encodings are unnatural and unwieldy for many problems (e.g., evolving weights for neural networks or evolving condition sets in the manner of Meyer and Packard), and they are prone to rather arbitrary orderings.

2. Many-Character and Real-Valued Encodings:

For many applications, it is most natural to use an alphabet of many characters or real numbers to form chromosomes. Examples include Kitano's [3] many-character representation for graph-generation grammars, Meyer and Packard's [3] real-valued representation for condition sets, Montana and Davis's [3] real-valued representation for neural-network weights, and Schultz-Kremer's [3] real-valued representation for torsion angles in proteins.

Holland's schema-counting argument seems to imply that GAs should exhibit worse performance on multiple-character encodings than on binary encodings. Several empirical comparisons between binary encodings and multiple-character or real-valued encodings have shown better performance for the latter e.g., Janikow and Michalewicz [3]. But the performance depends very much on the problem and the details of the GA being used, and at present there are no rigorous guidelines for predicting which encoding will work best.

3. Tree Encodings:

Tree encoding schemes, such as John Koza's [3] scheme for representing computer programs, have several advantages, including the fact that they allow the search space to be open-ended (in principle, any size tree could be formed via crossover and mutation). This open-endedness also leads to some potential pitfalls. The trees can grow large in uncontrolled ways, preventing the formation of more structured, hierarchical candidate solutions. (Koza's [3] (1992, 1994) "automatic definition of functions" is one way in which GP can be encouraged to design hierarchically structured programs.) Also, the resulting trees, being large, can be very difficult to understand and to simplify. Systematic experiments evaluating the usefulness of tree encodings and comparing them with other encodings are only just beginning in the genetic programming community. Likewise, as yet there are only very nascent attempts at extending GA theory to tree encodings.

These are only the most common encodings; a survey of the GA literature will turn up experiments on several others.

4.5 Adapting the Encoding:

Choosing a fixed encoding ahead of time presents a paradox to the potential GA user: for any problem that is hard enough that one would want to use a GA, one doesn't know enough about the problem ahead of time to come up with the best encoding for the GA. In fact, coming up with the best encoding is almost tantamount to solving the problem itself! The original lexicographic ordering of bits was arbitrary, and it probably impeded the GA from finding better solutions quickly—to find high-fitness rules, many bits spread throughout the string had to be co adapted. If these bits were close together on the string, so that they were less likely to be separated under crossover, the performance of the GA would presumably be improved. But we had no idea how best to order the bits ahead of time for this problem. This is known in the GA literature as the "linkage problem"—one wants to have functionally related loci be more likely to stay together on the string under crossover, but it is not clear how this is to be done without knowing ahead of time which loci are important in useful schemas. Faced with this problem, and having notions of evolution and adaptation already primed in the mind, many users have a revelation: "As long as I'm using a GA to solve the problem, why not have it adapt the encoding at the same time!"

1. Inversion:

Holland [3] (1975) included proposals for adapting the encodings in his original proposal for GAs. Holland, acutely aware that correct linkage is essential for single-point crossover to work well, proposed an "inversion" operator specifically to deal with the linkage problem in fixed-length strings.

Inversion is a reordering operator inspired by a similar operator in real genetics. Unlike simple GAs, in real genetics the function of a gene is often independent of its position in the chromosome (though often genes in a local area work together in a regulatory network), so inverting part of the chromosome will retain much or all of the "semantics" of the original chromosome.

To use inversion in GAs, we have to find some way for the functional interpretation of an allele to be the same no matter where it appears in the string. For example, in the chromosome encoding a cellular automaton the leftmost bit under lexicographic ordering is the output bit for the neighborhood of all zeros. We would want that bit to represent that same neighborhood even if its position were changed in the string under an inversion. Holland proposed that each allele be given an index indicating its "real" position, to be used when evaluating a chromosome's fitness. For example, the string 00010101 would be encoded as

{(1,0) (2,0) (3, 0) (4,1) (5, 0) (6,1) (7, 0) (8,1)},

With the first member of each pair giving the "real" position of the given allele. This is the same string as, say,

{(1, 0) (2, 0) (6,1) (5, 0) (4, .1) (3,0) (7,0) (8,1)}.

Inversion works by choosing two points in the string and reversing the order of the bits between them—in the example just given, bits 3-6 were reversed. This does not change the fitness of the chromosome, since to calculate the fitness the string is ordered by the indices. However, it does change the linkages: the idea behind inversion is to produce orderings in which beneficial schemas are more likely to survive. Suppose that in the original ordering the schema 00 * *01 * * is very important. Under the new ordering, that schema is 0010 * * * *. Given that this is a high-fitness schema and will now tend to survive better under single-point crossover, this permutation will presumably tend to survive better than would the original string.

The reader may have noticed a hitch in combining inversion with single-point crossover. Suppose, for example, that

{(1, 0) (2, 0) (6,1) (5,0) (4,1) (3,0) (7, 0) (8,1)}

Crosses with

{(5,1) (2, 0) (3,1) (4,1) (1,1) (8,1) (6,0) (7, 0)}

After the third bit, the offspring are

{(1,0) (2,0) (6,1) (4,1) (1,1) (8,1) (6, 0) (7,0)}

And

{(5,1) (2, 0) (3,1) (5,0) (4,1) (3,0) (7, 0) (8,1)}.

The first offspring has two copies each of bits 1 and 6 and no copies of bits 3 and 5. The second offspring has two copies of bits 3 and 5 and no copies of bits 1 and 6. How can we ensure that crossover will produce offspring with a full set of loci? Holland proposed two possible solutions:

- Permit crossover only between chromosomes with the same permutation of the loci. This would work, but it severely limits the way in which crossover can be done.

- Employ a "master/slave" approach: choose one parent to be the master, and temporarily reorder the other parent to have the same ordering as the master. Use this ordering to produce offspring, returning the second parent to its original ordering once crossover has been performed. Both methods have been used in experiments on inversion.

Inversion was included in some early work on GAs but did not produce any stunning improvements in performance (Goldberg 1989a). More recently, forms of inversion have been incorporated with some success into GAs applied to "ordering problems" such as the DNA fragment assembly problem (Parsons, Forrest, and Burks, in press). However, the verdict on the benefits of inversion to GAs is not yet in; more systematic experimental and theoretical studies are needed. In addition, any performance benefit conferred by inversion must be weighed against the additional space (to store indices for every bit) and additional computation time (e.g., to reorder one parent before crossover) that inversion requires.

2. Evolving Crossover "Hot Spots":

A different approach, also inspired by nature, was taken by Schaffer and Morishima [3] (1987). Their idea was to evolve not the order of bits in the string but rather the positions at which crossover was allowed to occur (crossover "hot spots"). Attached to each candidate solution in the population was a second string—a "crossover template"—that had a 1 at each locus at which crossover was to take place and a 0 at each locus at which crossover was not to take place. For example, 10011111:00010010 (with the chromosome preceding and the crossover template following the colon) meant that crossover should take place after the fourth and seventh loci in that string. Using an exclamation point to denote the crossover markers (each attached to the bit on its left), we can write this as 1001!111!1. Now, to perform multi-point crossover on two parents (say 1001!111!1 and OOOOOO'OO), the !s mark the crossover points, and they get inherited along with the bits to which they are attached:

Parents **1 0 0 I! 1 1 I! 1**
0 0 0 0 0 0! 0 0

Offspring **1 0 0 0! 0 0! I! 0**
0 0 0 0 1 1 0 1

Mutation acts on both the chromosomes and the attached crossover templates. Only the candidate solution is used to determine fitness, but the hope is that selection, crossover, and mutation will not only discover good solutions but also coevolve good crossover templates. Schaffer and Morishima found that this method outperformed a version of the simple GA on a small suite of function optimization problems. Although this method is interesting and is inspired by real genetics (in which there are crossover hot spots that have somehow coevolved with chromosomes), there has not been much further investigation into why it works and to what degree it will actually improve GA performance over a larger set of applications.

3. Messy GAs:

The goal of "messy GAs," developed by Goldberg and his colleagues, is to improve the GA's function-optimization performance by explicitly building up increasingly longer, highly fit strings from well-tested shorter building blocks (Goldberg, Korb, and Deb 1989; Goldberg, Deb, and Korb [3] 1990; Goldberg, Deb, Kargupta, and Harik [3] 1993). The general idea was biologically motivated: "After all, nature did not start with strings of length 5.9×10^9 (an estimate of the number of pairs of DNA nucleotides in the human genome) or even of length two million (an estimate of the number of genes in *Homo sapiens)* and try to make man. Instead, simple life forms gave way to more complex life forms, with the building blocks learned at earlier times used and reused to good effect along the way."

Consider a particular optimization problem with candidate solutions represented as bit strings. In a messy GA each bit is tagged with its "real" locus, but in a given chromosome not all loci have to be specified ("under specification") and some loci can be specified more than once, even with conflicting alleles ("over specification"). For example, in a four-bit problem, the following two messy chromosomes might be found in the population:

{(1,0) (2,0) (4,1) (4,0)}

and

{(3,1) (3, 0) (3,1) (4, 0) (4,1) (3,1)}.

The first specifies no value for locus 3 and two values for locus 4. The second specifies no values for loci 1 and 2, two values for locus 4 and a whopping four values for locus 3. (The term "messy GA" is meant to be contrasted with standard "neat" fixed-length, fixed-population-size GAs.)

Given all this under- and over specification, how is the fitness function to be evaluated? Over specification is easy: Goldberg and his colleagues simply used a left-to-right, first-come-first-served scheme. (e.g., the chosen value for locus 4 in the first chromosome is 1.) Once over specification has been taken care of, the specified bits in the chromosome can be thought of as a "candidate schema" rather than as a candidate solution. For example, the first chromosome above is the schema 00*1. The purpose of messy GAs is to evolve such candidate schemas, gradually building up longer and longer ones until a solution is formed. This requires a way to evaluate a candidate schema under a given fitness function. However, under most fitness functions of interest, it is difficult if not impossible to compute the "fitness" of a partial string. Many loci typically interact non-independently to determine a string's fitness, and in an under specified string the missing loci might be crucial. Goldberg and his colleagues first proposed and then rejected an "averaging" method: for a given under specified string, randomly generate values for the missing loci over a number of trials and take the average fitness computed with these random samples. The idea is to estimate the average fitness of the candidate schema. But, as was pointed out earlier, the variance of this average fitness will often be too high for a meaningful average to be gained from such sampling. Instead, Goldberg [3] and his colleagues used a method they called "competitive templates." The idea was not to estimate the average fitness of the candidate schema but to see if the candidate schema yields an improvement over a local optimum. The method works by finding a local optimum at the beginning of a run by a hill-climbing technique, and then, when running the messy GA, evaluating under specified strings by filling in missing bits from the local optimum and then applying the fitness function. A local optimum is, by definition, a string that cannot be improved by a single-bit change;

thus, if a candidate schema's defined bits improve the local optimum, it is worth further exploration.

The messy GA proceeds in two phases: the "primordial phase" and the "juxtapositional phase." The purpose of the primordial phase is to enrich the population with small, promising candidate schemas, and the purpose of the juxtapositional phase is to put them together in useful ways. Goldberg and his colleagues' first method was to guess at the order k of the smallest relevant schemas and to form the initial population by completely enumerating all schemas of that order. For example, if the size of solutions is $l = 8$ and the guessed k is 3, the initial population will be,

{(1,0) (2,0) (3,0)}
{(1,0) (2,0) (3,1)}

.
.

{(1,1) (2,1) (3,1)} {(1,0) (2,0) (4,0)}
{(1,0) (2,0) (4,1)}

.

{(6,1) (7,1) (8,1)}.

After the initial population has been formed and the initial fitnesses evaluated (using competitive templates), the primordial phase continues by selection only (making copies of strings in proportion to their fitnesses with no crossover or mutation) and by culling the population by half at regular intervals. At some generation (a parameter of the algorithm), the primordial phase comes to an end and the juxtapositional phase is invoked. The population size stays fixed, selection continues, and two juxtapositional operators—"cut" and "splice"—are introduced. The cut operator cuts a string at a random point. For example,

{(2,0) (3,0) (1,1) (4,1) (6,0)}

Could be cut after the second locus to yield two strings: {(2, 0) (3,0)} and {(1,1) (4,1) (6, 0)}. The splice operator takes two strings and splices them together. For example,

{(1.1) (2,1) (3,1)} and {(1, 0) (4,1) (3, 0)}

Could be spliced together to form

{(1,1) (2,1) (3,1) (1,0) (4,1) (3,0)}.

Under the messy encoding, cut and splice always produce perfectly legal strings. The hope is that the primordial phase will have produced all the building blocks needed to create an optimal string, and in sufficient numbers so that cut and splice will be likely to create that optimal string before too long. Goldberg and his colleagues did not use mutation in the experiments they reported.

Unfortunately, even with probabilistically complete initialization, the necessary initial population size still grows exponentially with *k*, so messy GAs will be feasible only on problems in which *k* is small. Goldberg and his colleagues seem to assume that most problems of interest will have small *k*, but this has never been demonstrated. It remains to be seen whether the promising results they have found on specially designed fitness functions will hold when messy GAs are applied to real-world problems. Goldberg, Deb, and Korb [3] have already announced that messy GAs are "ready for real-world applications" and recommended their "immediate application ... to difficult, combinatorial problems of practical import." To my knowledge, they have not yet been tried on such problems.

4.6 Selection Methods:

After deciding on an encoding, the second decision to make in using a genetic algorithm is how to perform selection—that is, how to choose the individuals in the population that will create offspring for the next generation, and how many offspring each will create. The purpose of selection is, of course, to emphasize the fitter individuals in the

population in hopes that their offspring will in turn have even higher fitness. Selection has to be balanced with variation from crossover and mutation (the "exploitation/exploration balance"): too-strong selection means that suboptimal highly fit individuals will take over the population, reducing the diversity needed for further change and progress; too-weak selection will result in too-slow evolution. As was the case for encodings, numerous selection schemes have been proposed in the GA literature. In the following section are described some of the most common methods.

1. **Fitness-Proportionate Selection with "Roulette Wheel" and "Stochastic Universal" Sampling:**

Holland's original GA used fitness-proportionate selection, in which the "expected value" of an individual (i.e., the expected number of times an individual will be selected to reproduce) is that individual's fitness divided by the average fitness of the population. The most common method for implementing this is "roulette wheel" sampling, each individual is assigned a slice of a circular "roulette wheel," the size of the slice being proportional to the individual's fitness. The wheel is spun N times, where N is the number of individuals in the population. On each spin, the individual under the wheel's marker is selected to be in the pool of parents for the next generation. This method can be implemented as follows:

1. Sum the total expected value of individuals in the population. Call this sumT.

2. Repeat N times

Choose a random integer r between 0 and T'. Loop through the individuals in the population, summing the expected values, until the sum is greater than or equal to r. The individual whose expected value puts the sum over this limit is the one selected.

This stochastic method statistically results in the expected number of offspring for each individual. However, with the relatively small populations typically used in GAs, the actual number of offspring allocated to an individual is often far from its expected value (an extremely unlikely series of spins of the roulette wheel could even allocate all offspring to the worst individual in the population). James Baker (1987) proposed a

different sampling method—"stochastic universal sampling" (SUS)—to minimize this "spread" (the range of possible actual values, given an expected value). Rather than spin the roulette wheel N times to select N parents, SUS spins the wheel once—but with N equally spaced pointers, which are used to selected the N parents. Baker (1987) gives the following code fragment for SUS (in C).

SUS does not solve the major problems with fitness-proportionate selection. Typically, early in the search the fitness variance in the population is high and a small number of individuals are much fitter than the others. Under fitness-proportionate selection, they and their descendents will multiply quickly in the population, in effect preventing the GA from doing any further exploration. This is known as "premature convergence." In other words, fitness-proportionate selection early on often puts too much emphasis on "exploitation" of highly fit strings at the expense of exploration of other regions of the search space. Later in the search, when all individuals in the population are very similar (the fitness variance is low), there are no real fitness differences for selection to exploit, and evolution grinds to a near halt. Thus, the rate of evolution depends on the variance of fitnesses in the population.

2. Sigma Scaling:

To address such problems, GA researchers have experimented with several "scaling" methods—methods for mapping "raw" fitness values to expected values so as to make the GA less susceptible to premature convergence. One example is "sigma scaling" (Forrest [3] 1985; it was called "sigma truncation" in Goldberg 1989a), which keeps the selection pressure (i.e., the degree to which highly fit individuals are allowed many offspring) relatively constant over the course of the run rather than depending on the fitness variances in the population. Under sigma scaling, an individual's expected value is a function of its fitness, the population mean, and the population standard deviation. An example of sigma scaling would be

$$ExpVal(i,t) = \begin{cases} 1 + \dfrac{f(i) - \bar{f}(t)}{2\sigma(t)} & \text{if} \sigma(t) \neq 0 \\ 1.0 \rightarrow otherwise \end{cases}$$

where ExpVal (i, t) *is* the expected value of individual *i* at time *t*, *f(i) is* the fitness of *i*, *f(t) is* the mean fitness of the population at time *t*, and $\sigma(t)$ is the standard deviation of the population fitnesses at time *t*. This function, used in the work of Tanese (1989), gives an individual with fitness one standard deviation above the mean 1.5 expected offspring. If ExpVal(i,t) was less than 0, Tanese arbitrarily reset it to 0.1, so that individuals with very low fitness had some small chance of reproducing.

At the beginning of a run, when the standard deviation of fitnesses is typically high, the fitter individuals will not be many standard deviations above the mean, and so they will not be allocated the lion's share of offspring. Likewise, later in the run, when the population is typically more converged and the standard deviation is typically lower, the fitter individuals will stand out more, allowing evolution to continue.

3. Elitism:

"Elitism," first introduced by Kenneth De Jong [3] (1975), is an addition to many selection methods that forces the GA to retain some number of the best individuals at each generation. Such individuals can be lost if they are not selected to reproduce or if they are destroyed by crossover or mutation. Many researchers have found that elitism significantly improves the GA's performance.

4. Boltzmann Selection:

Sigma scaling keeps the selection pressure more constant over a run. But often different amounts of selection pressure are needed at different times in a run—for example, early on it might be good to be liberal, allowing less fit individuals to reproduce at close to the rate of fitter individuals, and having selection occur slowly while maintaining a lot of variation in the population. Later it might be good to have selection be stronger in

order to strongly emphasize highly fit individuals, assuming that the early diversity with slow selection has allowed the population to find the right part of the search space.

One approach to this is "Boltzmann selection" (an approach similar to simulated annealing), in which a continuously varying "temperature" controls the rate of selection according to a preset schedule. The temperature starts out high, which means that selection pressure is low (i.e., every individual has some reasonable probability of reproducing). The temperature is gradually lowered, which gradually increases the selection pressure, thereby allowing the GA to narrow in ever more closely to the best part of the search space while maintaining the "appropriate" degree of diversity. For examples of this approach, see Goldberg 1990, de la Maza and Tidor [3] 1991 and 1993, and Prugel-Bennett and Shapiro [3] 1994. A typical implementation is to assign to each individual; an expected value,

$$ExpVal(i,t) = \frac{e^{f(i)/T}}{\left\langle e^{f(i)/T} \right\rangle T}$$

where T is temperature and $\langle\ \rangle t$ denotes the average over the population at time t. Experimenting with this formula will show that, as T decreases, the difference in ExpVal(i, t) between high and low fitnesses increases. The desire is to have this happen gradually over the course of the search, so temperature is gradually decreased according to a predefined schedule. De la Maza and Tidor [3] (1991) found that this method outperformed fitness-proportionate selection on a small set of test problems. They also (1993) compared some theoretical properties of the two methods.

5. Rank Selection:

Rank selection is an alternative method whose purpose is also to prevent too-quick convergence. In the version proposed by Baker (1985), the individuals in the population are ranked according to fitness, and the expected value of each individual depends on its rank rather than on its absolute fitness. There is no need to scale fitnesses in this case,

since absolute differences in fitness are obscured. This discarding of absolute fitness information can have advantages (using absolute fitness can lead to convergence problems) and disadvantages (in some cases it might be important to know that one individual is far fitter than its nearest competitor). Ranking avoids giving the far largest share of offspring to a small group of highly fit individuals, and thus reduces the selection pressure when the fitness variance is high. It also keeps up selection pressure when the fitness variance is low: the ratio of expected values of individuals ranked i and, $i+1$ will be the same whether their absolute fitness differences are high or low.

6. Tournament Selection:

The fitness-proportionate methods described above require two passes through the population at each generation: one pass to compute the mean fitness (and, for sigma scaling, the standard deviation) and one pass to compute the expected value of each individual. Rank scaling requires sorting the entire population by rank—a potentially time-consuming procedure. Tournament selection is similar to rank selection in terms of selection pressure, but it is computationally more efficient and more amenable to parallel implementation. Two individuals are chosen at random from the population. A random number r is then chosen between 0 and 1. If $r < k$ (where k *is a* parameter, for example 0.75), the fitter of the two individuals is selected to be a parent; otherwise the less fit individual is selected. The two are then returned to the original population and can be selected again.

7. Evolving a Learning Rule:

David Chalmers [3] (1990) took the idea of applying genetic algorithms to neural networks in a different direction: he used GAs to evolve a good learning rule for neural networks. Chalmers limited his initial study to fully connected feedforward networks with input and output layers only, no hidden layers. In general a learning rule is used during the training procedure for modifying network weights in response to the network's performance on the training data. At each training cycle, one training pair is given to the network, which then produces an output. At this point the learning rule is invoked to modify weights. A learning rule for a single-layer, fully connected

feedforward network might use the following local information for a given training cycle to modify the weight on the link from input unit i to output unit j:

a_i : the activation of input unit *i*.

o_j : the activation of output unit *j*.

t_j :the training signal (i.e., correct activation, provided by a teacher) on output unit *j*.

w_{ij} : the current weight on the link from *i* to *j*.

The change to make in weight w_{ij}, Δw_{ij}, is a function of these values:

$$\Delta w_{ij} = f(a_i, o_j, t_j, w_{ij})$$

The chromosomes in the GA population encoded such functions.

Chalmers [3] made the assumption that the learning rule should be a linear function of these variables and all their pair wise products. That is, the general form of the learning rule was

$$\Delta w_{ij} = k_0(k_1 w_{ij} + k_2 a_i + k_3 o_j + k_4 t_j + k_5 w_{ij} a_i + k_6 w_{ij} o_j + k_7 w_{ij} t_j + k_8 a_i o_j + k_9 a_i t_j + k_{10} o_j t_j)$$

The $k_m (1 \le m \le 10)$ are constant coefficients, and k_o is a scale parameter that affects how much the weights can change on any one cycle, (k_o is called the "learning rate.") Chalmers's assumption about the form of the learning rule came in part from the fact that a known good learning rule for such networks the "Widrow-Hoff" or "delta" rule—has the form

$$\Delta w_{ij} = \eta(t_j o_j - a_i o_j)$$

(where η) is a constant representing the learning rate. One goal of Chalmers's work was to see if the GA could evolve a rule that performs as well as the delta rule.

4.7 Genetic Operators:

The third decision to make in implementing a genetic algorithm is what genetic operators to use. This decision depends greatly on the encoding strategy. Where we will discuss crossover and mutation mostly in the context of bit-string encodings, and mention a number of other operators that have been proposed in the GA literature.

1. **Crossover:** It could be said that the main distinguishing feature of a GA is the use of crossover. Single-point crossover is the simplest form: a single cross-over position is chosen at random and the parts of two parents after the crossover position are exchanged to form two offspring. The idea here is, of course, to recombine building blocks (schemas) on different strings. Single-point crossover has some shortcomings, though. For one thing, it cannot combine all possible schemas. For example, it cannot in general combine instances of 11*****1 and ****11** to form an instance of ll**11*1. Likewise, schemas with long defining lengths are likely to be destroyed under single-point crossover. Eshelman, Caruana, and Schaffer [3] (1989) call this "positional bias": the schemas that can be created or destroyed by a crossover depend strongly on the location of the bits in the chromosome. Single-point crossover assumes that short, low-order schemas are the functional building blocks of strings, but one generally does not know in advance what ordering of bits will group functionally related bits together—this was the purpose of the inversion operator and other adaptive operators described above. Eshelman, Caruana, and Schaffer also point out that there may not be any way to put all functionally related bits close together on a string, since particular bits might be crucial in more than one schema. They point out further that the tendency of single-point crossover to keep short schemas intact can lead to the preservation of hitchhikers—bits that are not part of a desired schema but which, by being close on the string, hitchhike along with the beneficial schema as it reproduces. Many people have also noted that single-point crossover treats some loci preferentially, the segments exchanged between the two parents always contain the endpoints of the strings.

Most of the comments above also assume that crossover's ability to re-combine highly fit schemas is the reason it should be useful. Given some of the challenges we have seen to the relevance of schemas as a analysis tool for understanding GAs, one might ask if we should not consider the possibility that crossover is actually useful for some entirely

different reason (e.g., it is in essence a "macro-mutation" operator that simply allows for large jumps in the search space). I must leave this question as an open area of GA research for interested readers to explore. (Terry Jones [3] (1995) has performed some interesting, though preliminary, experiments attempting to tease out the different possible roles of crossover in GAs.) Its answer might also shed light on the question of why recombination is useful for real organisms (if indeed it is)—a controversial and still open question in evolutionary biology.

2. **Mutation:** A common view in the GA community, dating back to Holland's book *Adaptation in Natural and Artificial Systems,* is that crossover is the major instrument of variation and innovation in GAs, with mutation insuring the population against permanent fixation at any particular locus and thus playing more of a background role. This differs from the traditional positions of other evolutionary computation methods, such as evolutionary programming and early versions of evolution strategies, in which random mutation is the only source of variation. (Later versions of evolution strategies have included a form of crossover.)

4.8 Parameters for Genetic Algorithms:

The fourth decision to make in implementing a genetic algorithm is how to set the values for the various parameters, such as population size, crossover rate, and mutation rate. These parameters typically interact with one another nonlinearly, so they cannot be optimized one at a time. There is a great deal of discussion of parameter settings and approaches to parameter adaptation in the evolutionary computation literature—too much to survey or even list here. There are no conclusive results on what is best; most people use what has worked well in previously reported cases. Some of the experimental approaches people have taken to find the "best" parameter settings, are discussed below.

De Jong [3] (1975) performed an early systematic study of how varying parameters affected the GA's on-line and off-line search performance on a small suite of test

functions. Recall from chapter 4 that "on-line" performance at time t is the average fitness of all the individuals that have been evaluated over t evaluation steps. The off-line performance at time t is the average value, over t evaluation steps, of the best fitness that has been seen up to each evaluation step. De Jong's experiments indicated that the best population size was 50-100 individuals, the best single-point crossover rate was ~ 0.6 per pair of parents, and the best mutation rate was 0.001 per bit. These settings (along with De Jong's test suite) became widely used in the GA community, even though it was not clear how well the GA would perform with these settings on problems outside De Jong's test suite. Any guidance was gratefully accepted.

Somewhat later, Grefenstette [3] (1986) noted that, since the GA could be used as an optimization procedure, it could be used to optimize the parameters for another GA! (A similar study was done by Bramlette [3] (1991).) In Grefenstette's experiments, the "meta-level GA" evolved a population of 50 GA parameter sets for the problems in De Jong's test suite. Each individual encoded six GA parameters: population size, crossover rate, mutation rate, generation gap, scaling window (a particular scaling technique that I won't discuss here), and selection strategy (elitist or nonelitist). The fitness of an individual was a function of the on-line or off-line performance of a GA using the parameters encoded by that individual. The meta-level GA itself used De Jong's parameter settings. The fittest individual for on-line performance set the population size to 30, the crossover rate to 0.95, the mutation rate to 0.01, and the generation gap to 1, and used elitist selection. These parameters gave a small but significant improvement in on-line performance over De Jong's settings. Notice that Grefenstette's results call for a smaller population and higher crossover and mutation rates than De Jong's. The meta-level GA was not able to find a parameter set that beat De Jong's for off-line performance. This was an interesting experiment, but again, in view of the specialized test suite, it is not clear how generally these recommendations hold. Others have shown that there are many fitness functions for which these parameter settings are not optimal.

A big question, then, for any adaptive approach to setting parameters— including Davis's—is this: How well does the rate of adaptation of parameter settings match the rate of adaptation in the GA population? The feedback for setting parameters comes from the population's success or failure on the fitness function, but it might be difficult

for this information to travel fast enough for the parameter settings to stay up to date with the population's current state. Very little work has been done on measuring these different rates of adaptation and how well they match in different parameter-adaptation experiments. This seems to me to be the most important research to be done in order to get self-adaptation methods to work well.

4.9 Example of GAs :

As warmups to more extensive discussions of GA applications, here are brief examples of GAs in action on two particularly interesting projects.

1. Using GAs to Evolve Strategies for the Prisoner's Dilemma:

The Prisoner's Dilemma, a simple two-person game invented by Merrill Flood and Melvin Dresher [7] in the 1950s, has been studied extensively in game theory, economics, and political science because it can be seen as an idealized model for real-world phenomena such as arms races (Axelrod 1984; Axelrod and Dion [7] 1988). It can be formulated as follows: Two individuals (call them Alice and Bob) are arrested for committing a crime together and are held in separate cells, with no communication possible between them. Alice is offered the following deal: If she confesses and agrees to testify against Bob, she will receive a suspended sentence with probation, and Bob will be put away for 5 years. However, if at the same time Bob confesses and agrees to testify against Alice, her testimony will be discredited, and each will receive 4 years for pleading guilty. Alice is told that Bob is being offered precisely the same deal. Both Alice and Bob know that if neither testifies against the other they can be convicted only on a lesser charge for which they will each get 2 years in jail.

Should Alice "defect" against Bob and hope for the suspended sentence, risking a 4-year sentence if Bob defects? Or should she "cooperate" with Bob (even though they cannot communicate), in the hope that he will also cooperate so each will get only 2 years, thereby risking a defection by Bob that will send her away for 5 years?

The game can be described more abstractly. Each player independently decides which move to make—i.e., whether to cooperate or defect. A "game" consists of each player's making a decision (a "move"). The possible results of a single game are summarized in a payoff matrix like the one shown in figure 1.3. Here the goal is to get as many points (as opposed to as few years in prison) as possible. (In figure 1.3, the payoff in each case can be interpreted as 5 minus the number of years in prison.) If both players cooperate, each gets 3 points. If player A defects and player B cooperates, then player A gets 5 points and player B gets 0 points, and vice versa if the situation is reversed. If both players defect, each gets 1 point. What is the best strategy to use in order to maximize one's own payoff? If you suspect that your opponent is going to cooperate, then you should surely defect. If you suspect that your opponent is going to defect, then you should defect too. No matter what the other player does, it is always better to defect. The dilemma is that if both players defect each gets a worse score than if they cooperate. If the game is *iterated* (that is, if the two players play several games in a row), both players' always defecting will lead to a much lower total payoff than the players would get if they cooperated. How can reciprocal cooperation be induced? This question takes on special significance when the notions of cooperating and defecting correspond to actions in, say, a real-world arms race (e.g., reducing or increasing one's arsenal).

		Player B	
		Cooperate	Defect
Player A	Cooperate	3,3	0,5
	Defect	5,0	1,1

Figure 1.3 The payoff matrix for the Prisoner's Dilemma (adapted from Axelrod [7] 1987). The two numbers given in each box are the payoffs for players A and B in the given situation, with player A's payoff listed first in each pair.

Robert Axelrod [7] of the University of Michigan has studied the Prisoner's Dilemma and related games extensively. His interest in determining what makes for a good strategy led him to organize two Prisoner's Dilemma tournaments (described in Axelrod 1984). He solicited strategies from researchers in a number of disciplines. Each

participant submitted a computer program that implemented a particular strategy, and the various programs played iterated games with each other. During each game, each program remembered what move (i.e., cooperate or defect) both it and its opponent had made in each of the three previous games that they had played with each other, and its strategy was based on this memory. The programs were paired in a round-robin tournament in which each played with all the other programs over a number of games. The first tournament consisted of 14 different programs; the second consisted of 63 programs (including one that made random moves). Some of the strategies submitted were rather complicated, using techniques such as Markov processes and Bayesian inference to model the other players in order to determine the best move. However, in both tournaments the winner (the strategy with the highest average score) was the simplest of the submitted strategies: TIT FOR TAT. This strategy, submitted by Anatol Rapoport [7], cooperates in the first game and then, in subsequent games, does whatever the other player did in its move in the previous game with TIT FOR TAT. That is, it offers cooperation and reciprocates it. But if the other player defects, TIT FOR TAT punishes that defection with a defection of its own, and continues the punishment until the other player begins cooperating again.

Chapter Five

Genetic Algorithms Versus Neural Network

5.1 Evolving Neural Network using Genetic Algorithm:

Neural networks are biologically motivated approaches to machine learning, inspired by ideas from neuroscience. Recently some efforts have been made to use genetic algorithms to evolve aspects of neural networks.

In its simplest "feed forward" form figure 5.1 shown, a neural network is a collection of connected activatable units ("neurons") in which the connections are weighted, usually with real-valued weights. The network is presented with an activation pattern on its input units, such a set of numbers representing features of an image to be classified (e.g., the pixels in an image of a handwritten letter of the alphabet). Activation spreads in a forward direction from the input units through one or more layers of middle ("hidden") units to the output units over the weighted connections. Typically, the activation coming into a unit from other units is multiplied by the weights on the links over which it spreads, and then is added together with other incoming activation. The result is typically thresholded (i.e., the unit "turns on" if the resulting activation is above that unit's threshold). This process is meant to roughly mimic the way activation spreads through networks of neurons in the brain. In a feed forward network, activation spreads only in a forward direction, from the input layer through the hidden layers to the output layer. Many people have also experimented with "recurrent" networks, in which there are feedback connections as well as feed forward connections between layers.

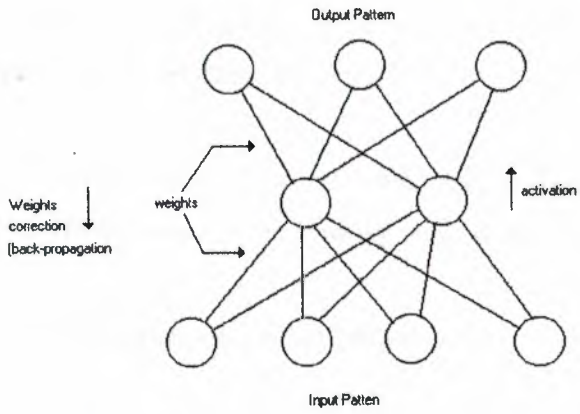


Fig 5.1. A schematic diagram of a simple feed forward neural network and the back-propagation process by which weight values are adjusted.

After activation has spread through a feedforward network, the resulting activation pattern on the output units encodes the network's "answer" to the input (e.g., a classification of the input pattern as the letter A). In most applications, the network learns a correct mapping between input and output patterns via a learning algorithm. Typically the weights are initially set to small random values. Then a set of training inputs is presented sequentially to the network. In the back-propagation learning procedure (Rumelhart, Hinton, [3] and Williams 1986), after each input has propagated through the network and an output has been produced, a "teacher" compares the activation value at each output unit with the correct values, and the weights in the network are adjusted in order to reduce the difference between the network's output and the correct output. Each iteration of this procedure is called a "training cycle," and a complete pass of training cycles through the set of training inputs is called a "training epoch." (Typically many training epochs are needed for a network to learn to successfully classify a given set of training inputs.) This type of procedure is known as "supervised learning," since a teacher supervises the learning by providing correct output values to guide the learning process. In "unsupervised learning" there is no teacher, and the learning system must learn on its own using less detailed (and sometimes less reliable) environmental feedback on its performance.

There are many ways to apply GAs to neural networks. Some aspects that can be evolved are the weights in a fixed network, the network architecture (i.e., the number of units and their interconnections can change), and the learning rule used by the network.

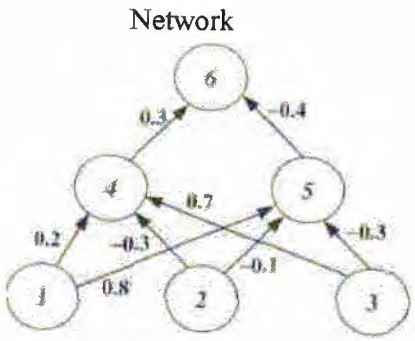
Here some different projects, each of which uses a genetic algorithm to evolve one of these aspects. (Two approaches to evolving network architecture will be described.)

5.2 Evolving Weights in a Fixed Network:

David Montana and Lawrence Davis [3] (1989) took the first approach— evolving the weights in a fixed network. That is, Montana and Davis were using the GA *instead* of back-propagation as a way of finding a good set of weights for a fixed set of connections. Several problems associated with the back-propagation algorithm (e.g., the tendency to get stuck at local optima in weight space, or the unavailability of a "teacher" to supervise learning in some tasks) often make it desirable to find alternative weight-training schemes.

Montana and Davis were interested in using neural networks to classify underwater sonic "lofargrams" (similar to spectrograms) into two classes: "interesting" and "not interesting." The overall goal was to "detect and reason about interesting signals in the midst of the wide variety of acoustic noise and interference which exist in the ocean." The networks were to be trained from a database containing lofargrams and classifications made by experts as to whether or not a given lofargram is "interesting." Each network had four input units, representing four parameters used by an expert system that performed the same classification. Each network had one output unit and two layers of hidden units (the first with seven units and the second with ten units). The networks were fully connected feed forward networks—that is, each unit was connected to every unit in the next higher layer. In total there were 108 weighted connections between units. In addition, there were 18 weighted connections between the non-input units and a "threshold unit" whose outgoing links implemented the thresholding for each of the non-input units, for a total of 126 weights to evolve.

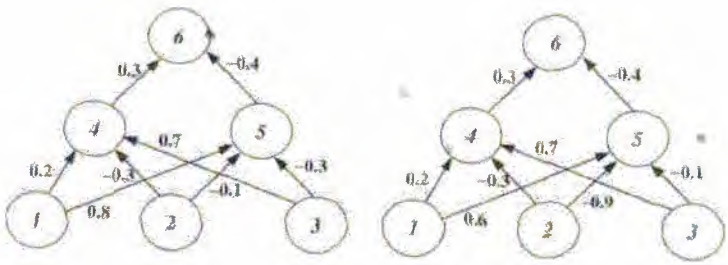
The GA was used as follows. Each chromosome was a list (or "vector") of 126 weights. Figure 5.2 shows (for a much smaller network) how the encoding was done: the weights were read off the network in a fixed order (from left to right and from top to bottom) and placed in a list. Notice that each "gene" in the chromosome is a real number rather than a bit. To calculate the fitness of a given chromosome, the weights in the chromosome were assigned to the links in the corresponding network, the network was run on the training set (here 236 examples from the database of lofargrams), and the sum of the squares of the errors (collected over all the training cycles) was returned. Here, an "error" was the difference between the desired output activation value and the actual output activation value. Low error meant high fitness.



Chromosome: (0.3 -0.4 0.2 0.8 -0.3 -0.1 0.7 -0.3)

Figure 5.2 Illustration of Montana and Davis's encoding of network weights into a list that serves as a chromosome for the GA. The units in the network are numbered for later reference. The real-valued numbers on the links are the weights.

Before mutation **After mutation**

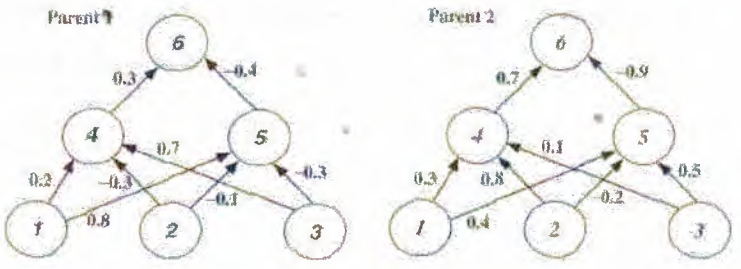


(0.3 -0.4 0.2 0.8 -0.3 -0.1 0.7 -0.3) **(0.3 -0.4 0.2 0.6 -0.3 -0.9 0.7 -0.1)**

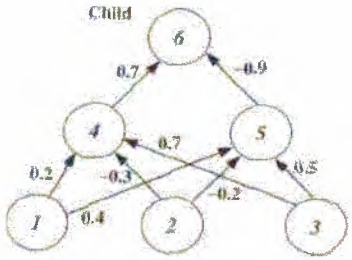
Figure 5.3 Illustration of Montana and Davis's mutation method. Here the weights on incoming links to unit 5 are mutated.

An initial population of 50 weight vectors was chosen randomly, with each weight being between -1.0 and +1.0. Montana and Davis tried a number of different genetic operators in various experiments. The mutation and crossover operators they used for their comparison of the GA with back-propagation are illustrated in figures 5.3 and 5.4. The mutation operator selects *n* non-input units and, for each incoming link to those units, adds a random value between -1.0 and +1.0 to the weight on the link. The crossover operator takes two parent weight vectors and, for each non-input unit in the offspring vector, selects one of the parents at random and copies the weights on the incoming links from that parent to the offspring. Notice that only one offspring is created.

The performance of a GA using these operators was compared with the performance of a back-propagation algorithm. The GA had a population of 50 weight vectors, and a rank-selection method was used. The GA was allowed to run for 200 generations (i.e., 10,000 network evaluations). The back-propagation algorithm was allowed to run for 5000 iterations, where iteration is a complete epoch (a complete pass through the training data). Montana and Davis reasoned that two network evaluations under the GA are equivalent to one back-propagation iteration, since back-propagation on a given training example consists of two parts—the forward propagation of activation (and the calculation of errors at the output units) and the backward error propagation (and adjusting of the weights). The GA performs only the first part. Since the second part requires more computation, two GA evaluations takes less than half the computation of a single back-propagation iteration.



(0.3 -0.4 0.2 0.8 -0.3 -0.1 0.7 -0.3) **(0.7 -0.9 0.3 0.4 0.8 -0.2 0.1 0.5)**



(0.7 -0.9 0.2 0.4 -0.3 -0.2 0.7 0.5)

Figure 5.4 Illustration of Montana and Davis's crossover method.

The results of the comparison are displayed in figure 5.5. Here one backpropagation iteration is plotted for every two GA evaluations. The x axis gives the number of iterations, and the y axis gives the best evaluation (lowest sum of squares of errors) found by that time. It can be seen that the GA significantly outperforms back-propagation on this task, obtaining better weight vectors more quickly.

This experiment shows that in some situations the GA is a better training method for networks than simple back-propagation. This does not mean that the GA will outperform back-propagation in all cases. It is also possible that enhancements of back-propagation might help it overcome some of the problems that prevented it from performing as well as the GA in this experiment. Schaffer, Whitley, and Eshelman (1992) point out that the GA has not been found to outperform the best weight-adjustment methods (e.g., "quickprop") on supervised learning tasks, but they predict that the GA will be most useful in finding weights in tasks where back-propagation and its relatives cannot be used, such as in unsupervised learning tasks, in which the error at each output unit is not available to the learning system, or in situations in which only sparse reinforcement is available. This is often the case for "neurocontrol" tasks, in which neural networks are used to control complicated systems such as robots navigating in unfamiliar environments.

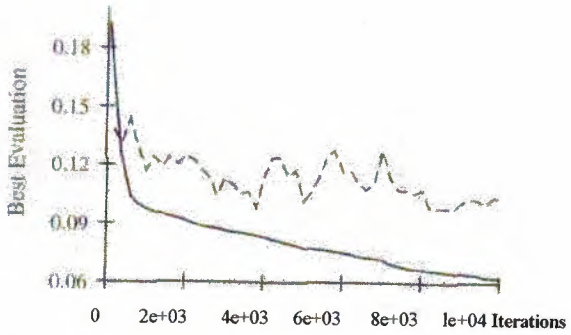
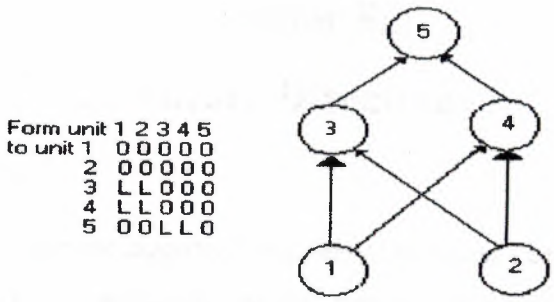


Figure 5.5 Montana and Davis's results comparing the performance of the GA with back-propagation.

5.3 Evolving Network Architectures:

Montana and Davis's [3] GA evolved the weights in a fixed network. As in most neural network applications, the architecture of the network—the number of units and their interconnections—is decided ahead of time by the programmer by guesswork, often aided by some heuristics (e.g., "more hidden units are required for more difficult problems") and by trial and error. Neural network researchers know all too well that the particular architecture chosen can determine the success or failure of the application, so they would like very much to be able to automatically optimize the procedure of designing an architecture for a particular application. Many believe that GAs are well suited for this task. There have been several efforts along these lines, most of which fall into one of two categories: direct encoding and grammatical encoding. Under direct encoding, network architecture is directly encoded into a GA chromosome. Under grammatical encoding, the GA does not evolve network architectures; rather, it evolves grammars that can be used to develop network architectures.



chromosome: 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 0 0 0 0 0 1 1 0

Figure 5.6 An illustration of Miller, Todd, and Hegde's representation scheme. Each entry in the matrix represents the type of connection on the link between the "from unit" (column) and the "to unit" (row). The rows of the matrix are strung together to make the bit-string encoding of the network, given at the bottom of the figure.

Chapter Six
Future Directions

As we have seen that genetic algorithms and neural networks can be a powerful tool for solving problems and for simulating natural systems in a wide variety of scientific fields. In examining the accomplishments of these algorithms, we have also seen that many unanswered questions remain. Finally we, summarize what the field of neural networks and genetic algorithms has achieved, and what are the most interesting and important directions for future research.

From the knowledge of problem-solving, scientific modeling, and theory we come to the following conclusions:

GAs and neural networks are promising methods for solving difficult technological problems, and for machine learning. More generally, they are a part of a new movement in computer science that is exploring biologically inspired approaches to computation. Advocates of this movement believe that in order to create the kinds of computing systems we need systems that are adaptable, massively parallel, able to deal with complexity, able to learn, and even creative—we should copy natural systems with these qualities. Natural evolution is a particularly appealing source of inspiration.

Genetic algorithms and neural networks are also promising approaches for modeling the natural systems that inspired their design. Most models using GAs are meant to be "gedanken experiments" or "idea models" (Roughgarden et al. [3] 1995) rather than precise simulations attempting to match real-world data. The purposes of these idea models are to make ideas precise and to test their plausibility by implementing them as computer programs (e.g., Hinton and Nowlan's [3] model of the Baldwin effect), to understand and predict general tendencies of natural systems (e.g.. Echo), and to see how these tendencies are affected by changes in details of the model (e.g., Collins and Jefferson's variations on Kirkpatrick's [3] sexual selection model). These models can allow scientists to perform experiments that would not be possible in the real world, and

to simulate phenomena that are difficult or impossible to capture and analyze in a set of equations. These models also have a largely unexplored but potentially interesting side that has not so far been mentioned here: by explicitly modeling evolution as a computer program, we explicitly cast evolution as a computational process, and thus we can think about it in this new light. For example, we can attempt to measure the "information" contained in a population and attempt to understand exactly how evolution processes that information to create structures that lead to higher fitness.

Holland's [3] *Adaptation in Natural and Artificial Systems,* in which GAs were defined, was one of the first attempts to set down a general framework for adaptation in nature and in computers. Holland's work has had considerable influence on the thinking of scientists in many fields, and it set the stage for most of the subsequent work on GA theory. However, Holland's theory is not a complete description of GA behavior. Recently a number of other approaches, such as exact mathematical models, statistical-mechanics-based models, and results from population genetics, have gained considerable attention. GA theory is not just academic; theoretical advances must be made so that we can know how best to use GAs and how to characterize the types of problems for which they are suited. Theoretical advances will also filter back to the evolutionary biology community. Though it hasn't happened yet, there is a very good chance that proving things about these simple models will lead to new ways to think mathematically about natural evolution.

Evolutionary computation is far from being an established science with a body of knowledge that has been collected for centuries. It has been around for little more than 30 years, and only in the last decade have a reasonably large number of people been working on it. Almost all the projects discussed in this book still can be considered "work in progress." Here is a brief list of some of the directions, which are the most important and promising.

6.1 Incorporating Ecological Interactions:

In most neural and GA applications the candidate solutions in the population are assigned fitnesses independent of one another and interact only by competing for selection slots via their fitnesses. However, some of the more interesting and successful

applications have used more complicated "ecological" interactions among population members. Hillis's [3] host-parasite co evolution was a prime example; so was Axelrod's [3] experiment in which the evolving strategies for the Prisoner's Dilemma played against one another and developed a cooperative symbiosis. These methods (along with other examples in the GA literature) are not understood very well; much more work is needed, for example, on making host-parasite co evolution a more generally applicable method and understanding how it works. In addition, other types of ecological interactions, such as individual competition for resources or symbiotic cooperation in collective problem solving, can be utilized in GAs.

6.2 Incorporating New Ideas from Genetics:

Haploid [4] crossover and mutation are only the barest bones of real-world genetic systems. Some extensions, including diploidy, inversion, gene doubling, and deletion. Other GA researchers have looked at genetics-inspired mechanisms such as dominance, translocation, sexual differentiation and introns (Levenick [4] 1991). These all are likely to have important roles in nature, and mechanisms inspired by them could potentially be put to excellent use in problem solving with GAs. As yet, the exploration of such mechanisms has only barely scratched the surface of their potential. Perhaps even more potentially significant is genetic regulation. In recent years a huge amount has been learned in the genetics community about how genes regulate one another—how they turn one another on and off in complicated ways so that only the appropriate genes get expressed in a given situation. It is these regulatory networks that make the genome a complex but extremely adaptive system. Capturing this kind of genetic adaptivity will be increasingly important as GAs are used in more complicated, changing environments.

6.3 Incorporating Development and Learning:

In typical GA applications evolution works directly on a population of candidate solutions, in nature there is a separation between genotypes (encodings) and phenotypes (candidate solutions). There are very good reasons for such a separation. One is that, as organisms become more complex, it seems to be more efficient and tractable for the operators of evolution to work on a simpler encoding that develops into the complex organism. Another is that environments are often too unpredictable for appropriate

behavior to be directly encoded into a genotype that does not change during an individual's life. In nature, the processes of development and learning help "tune" the behavioral parameters defined by the individual's genome so that the individual's behavior will become adapted for its particular environment. These reasons for separating genotype from phenotype and for incorporating development and learning have been seen in several of our case studies. Kitano pointed out that a grammatical encoding followed by a development phase allows for the evolution of more complex neural networks than is possible using a direct encoding. Incorporating development in this way has been taken further by Gruau [4] (1992) and Belew (1993), but much more work needs to be done if we are to use GAs to evolve large, complex systems (such as computational "brains"). The same can be said for incorporating learning into evolutionary computation—we have seen how this can have many advantages, even if what is learned is not directly transmitted to offspring—but the simulations we have seen are only early steps in understanding how to best take advantage of interactions between evolution and learning.

6.4 Adapting Encodings and using Encodings that Permit Hierarchy and Open-Endedness:

Evolution in nature not only changes the fitnesses of organisms, it also has mechanisms for changing the genetic encoding. Some reordering operators that occur in nature (e.g., inversion and translocation). In addition, genotypes have increased in size over evolutionary time. The ability to adapt their own encodings is important for GAs. Several methods have been explored in the GA literature. If we want GAs eventually to be able to evolve complex structures, the most important factors will be open-endedness (the ability for evolution to increase the size and complexity of individuals to an arbitrary degree), encapsulation (the ability to protect a useful part of an encoding from genetic disruption and to make sure it acts as a single whole), and hierarchical regulation (the ability to have different parts of the genome regulate other parts, and likewise the ability to have different parts of the phenotype regulate other parts). Some explorations of open-endedness and encapsulation in genetic programming were discussed; to me these types of explorations seem to be on the right track, though the

specific type of encoding used in GP may not turn out to be the most effective one for evolving complex structures.

6.5 Adapting Parameters:

Natural evolution also adapts its own parameters. Crossover and mutation rates are encoded (presumably in some rather complicated way) in the genomes of organisms, along with places where these operators are more likely to be applied (e.g., crossover hot spots). Likewise, population sizes in nature are not constant but are controlled by complicated ecological interactions. To find similar ways of adapting the parameters for GAs as part of the evolutionary process. There have been several explorations of this already, but much more work needs to be done to develop more general and sophisticated techniques. One of the major difficulties is having the time scale of adaptation for the parameters appropriately match the time scale of adaptation of the individuals in the population.

6.6 Extension of Statistical Mechanics Approaches:

Approaches similar to that taken by Prugel-Bennett and Shapiro [3] are promising for better understanding the behavior of GAs. That is, rather than construct exact mathematical models that in effect take into account every individual in a population; it is more useful to understand how macroscopic population structures change as a result of evolution. Ultimately we would like to have a general theory of the evolution of such macroscopic structures that will predict the effects of changes in parameters and other details of the GA. There is much more to be mined from the field of statistical mechanics in formulating such theories.

6.7 Identifying and Overcoming Impediments to the Success of GAs:

In the case studies and in the theoretical discussion we came across many potential impediments to the success of GAs, including deception, hitchhiking, symmetry breaking, over fitting, and inadequate sampling. GA researchers do not yet have anywhere near a complete understanding of the precise effects of these and other impediments on the performance of GAs, or of the precise conditions under which they come about, or of how to overcome them if that is possible.

6.8 Understanding the Role of Crossover:

Crossover is the primary operator distinguishing GAs from other stochastic search methods, but its role in GAs needs to be better understood.

Under what conditions does it indeed recombine building blocks to form high-fitness solutions, and under what conditions is it instead serving only as a "macro-mutation" operator, simply making larger jumps in the search space than a simple mutation operator can make? What is its role during various stages of the search? How can we quantify its ability to construct good solutions? Much theoretical work on GAs is aimed at answering these questions but precise answers are still lacking.

6.9 Theory of GAs with Endogenous Fitness:

In many of the scientific models we have looked at, "fitness" is not externally imposed but instead arises endogenously; it is reflected, for example, by the longevity and the fertility of an individual. Up to now, almost all work in GA theory has assumed exogenous rather than endogenous fitness functions. Holland (1994) has recently done some theoretical work on GA behavior with endogenous fitness in the context of Echo, using notions such a "flow-matrix" for describing the transmission of useful genetic building blocks from generation to generation. This is only a first step in theoretically analyzing such systems.

Conclusion

The integration of Neural Networks and Genetic Algorithms can be applied in many scientific complicated research areas, which can lead to the solution of many unanswered questions that require human intelligence. Systems can be trained to work and think like humans, e.g. modeling of natural systems, encoding and decoding of parameters, machine learning, evolution of strategies as in the example of Prisoner Dilemma etc.

Chapter one was a brief discussion on history of neural networks, with their evolution in biological terminology, along with a brief discussion of hybrid intelligent systems was discussed. In chapter two brief history of genetic algorithm along with a brief discussion of search space, genetic operators and application of genetic algorithms was discussed. Chapter three was a detail discussion about neural networks, neural learning, supervised and unsupervised learning, and classification of neural networks was discussed. In chapter four details about genetic algorithm, implementation, genetic encoding and selection methods, along with a brief example of prisoner dilemma was discussed. In chapter five a detail example of implementing a genetic algorithm, in a neural network was discussed. In chapter six future directions for implementation of neural networks and genetic algorithms was discussed.

However though Genetic Algorithms and Neural Networks can offer great benefits for mankind in terms of simulating human perception in machine, there are potential problems related to this novel approach in the Artificial Intelligence. These problems have so for been ethical.

Ethical Problems:

Although the integration of Neural Networks and Genetic Algorithms can have a lot of benefits, but on the other hand when it comes to machine learning, (which is an important topic not only because it would be an indispensable element in an intelligence system but, also because it holds a great promise in scientific discoveries) machine can

be trained but machines don't have feelings as humans do, this can lead to the invention of machines of massive destruction, which can have negative effect on the human race. On the other hand such machines can turn on their own inventors during their training periods.

References

[1] LiMin Fu., *Neural Networks in Computer Intelligence*, Mc Graw-Hill, Inc., New York NY, 1994.

[2] Asst. Prof. Dr Adnan Kashman., *Neural Networks Lecture Notes*.

[3] Melanie Mitchell., *An Introduction to Genetic Algorithms*, The MIT Press MA, Cambridge, Massachusetts, London, England, 1996.

[4] Davis L., *Handbook of Genetic Algorithms*, Morgan Kaufmann, 1991.

[5] Zeidenberg, Neural Networks in Artificial Intelligence, Ellis Horwood, New York NY, 1990.

[6] Goldberg De., *Genetic Algorithms*, Addison-Welsey MA, 1989.

[7] Liepins Ge., and Vose, *Foundations of Genetic Algorithms*, MIT Press MA, England, 1991.