EVOLUTIONARY LEARNING METHODS FOR MULTILAYER MORPHOLOGICAL PERCEPTRON

by

Roberto C. Piñeiro

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Engineering

Department of Electrical and Computer Engineering
University of Puerto Rico
Mayagüez Campus
2004

Approved by:

_____	_____
Néstor Rodríguez, Ph.D. Member, Graduate Committee	Date
_____	_____
Domingo Rodríguez, Ph.D. Member, Graduate Committee	Date
_____	_____
Jorge L. Ortiz, Ph.D. President, Graduate Committee	Date
_____	_____
Edgar Acuña, Ph.D. Representative of Graduate Studies	Date
_____	_____
Jorge L. Ortiz, Ph.D. Chairperson of the Department	Date

ABSTRACT

This thesis describes three compressive learning algorithms for multilayer morphological perceptrons. The three algorithms are based on evolutionary algorithms: *direct encoding method*, *indirect encoding method*, and *catesian genetic programming method*. The *direct encoding method* uses adaptive mutation as the genetic algorithm approaches convergence to fine tune network parameters to reach optimal values. In addition, the algorithms use a special fitness function which penalize those networks with redundant neurons. The training of the neural network using the *indirect encoding method* is done by finding the solution without considering the exact connectivity of the network. Looking for the set of connection weights and network architecture in a reduced search space, this simple, but powerful, training algorithm is able to evolve to a feasible solution using up to three layers suficient to perform most pattern classification. The last method uses Cartesian genetic programming to evolve network architecture and connection weights simultaneously. The resulting program consists of the multilayer morphological perceptron, which is able to classify patterns received as the inputs. The algorithm introduces the use of the morphological neuron computational model as the function used by the generated programs. Prototypes were implemented using Matlab, and tested using data sets used previously by other researchers.

RESUMEN

Esta tesis describe en detalle tres algoritmos de aprendizaje para perceptrones morfológicos de múltiples capas. Los tres algoritmos son basados en algoritmos evolutivos: el método de codificación de forma directa, el método de codificación de forma indirecta y el método de programación genética cartesiana. El método de codificación de forma directa utiliza mutación adaptiva según el algoritmo genético se acerca a la convergencia para refinar los parámetros de la red neural para poder conseguir valores óptimos. En adición, el algoritmo utiliza una función de evaluación especial en la que se penalizan aquellas redes neurales con neuronas redundantes de acuerdo a como estas estén colocadas. En el método de codificación de forma indirecta el entrenamiento de la red neural es hecho mediante la búsqueda de soluciones sin considerar la conectividad exacta de la red. Al reducir el espacio de busqueda pesos de las conexiones y la arquitectura de la red, este simple, pero poderoso algoritmo de entrenamiento es capaz de evolucionar soluciones viables usando hasta tres capas las cuales son requeridas para realizar la mayoría de las clasificaciones de patrones. El tercer método, utiliza programación genética cartesiana para evolucionar la arquitectura de la red y los pesos de las conexiones simultáneamente. El programa resultante produce la red neural capaz de clasificar los patrones recibidos como entradas. El metodo introduce el uso del modelo computacional usado por la neurona morfológica como las operaciones utilizadas por los programas generados. Prototipos fueron implementados usando Matlab y probados usando conjuntos de datos presentados por otros investigadores.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Dr. Jorge L. Ortiz for his valuable guidance throughout my graduate studies. I would like to thank my mother Felícita Colón, she always inspired me to achieve my best. I would like to thank my friend Nayda Santiago for her valuable organizational help. I would like to thank my good friends Ariel Mirles and Rafael Cordero for their special advice.

DEDICATORIA

Me gustaría agradecer a mi consejero el Dr. Jorge L. Ortiz por sus consejos en el transcurso de mis estudios graduados. Me gustaría agradecer a mi madre, Felícita Colón, por inspirarme a dar lo mejor. Quisiera agradecer a mi amiga Nayda Santiago por su ayuda organizacional invaluable. Quisiera agradecer a mis amigos Ariel Mirles y Rafael Cordero por sus consejos.

LIST OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Artificial Neural Network (ANN) is a component of artificial intelligence that emulates real brain's neurons. Artificial neural networks are a collection of mathematical models that simulate the connectionism behavior of human's brain. The system performs the computation through the passing of signals within a structured arrangement of connected processing units in response to a given input signal. Although these systems may be applied for prediction, interpretation, diagnosis, planning, and other applications, the most successful uses for artificial neural network are pattern recognition and pattern classification.

Morphological Neural Networks (MNN) (Ritter and Sussner 1996) are a novel class of artificial neural networks based on lattice algebra, in which the operations of multiplication and addition are replaced by addition and maximum or minimum operator, respectively. The algebraic system used by traditional neural network is denoted as $(\Re, +, \times)$, the set of real numbers $\Re$ with the operations of addition and multiplication, and all the laws governing these operators. The computations occurring in the morphological neural network are based on the algebraic lattice structure (Ritter and Sussner 1996) $(\Re_{-\infty}, \vee, +)$ and $(\Re_{\infty}, \wedge, +')$, where $\Re_{-\infty}$ and $\Re_{\infty}$ represent the extended real number systems $\Re_{-\infty} = \Re \cup \{-\infty\}$ and $\Re_{\infty} = \Re \cup \{\infty\}$. The symbol + denotes the usual addition with the additional stipulation that $a + (-\infty) = (-\infty) + a = -\infty;\ \forall a \in \Re_{-\infty}$,

and $+'$ is defined as $a+'b \equiv a+b$ for $a,b \in \Re_{\infty}$, and $a+\infty=\infty+a=\infty;\ \forall a \in \Re_{\infty}$. The symbols $\vee$ and $\wedge$ denote the maximum and minimum operators respectively, with the additional stipulation that $a \vee (-\infty) = (-\infty) \vee a = a;\ \forall a \in \Re_{-\infty}$ and $a \wedge \infty = \infty \wedge a = a\ \forall a \in \Re_{\infty}$. The application of maximum or minimum operations perform a nonlinear operation before the application of the transfer function, resulting in properties completely different from those properties of traditional neural networks. Multilayer Morphological Perceptrons (MLMP) are feed forward morphological neural networks used for pattern classification.

Artificial Neural Networks are able to acquire knowledge from previous experiences and apply the knowledge to similar situations. This process is known as *memorization* and *generalization*. A neural network "learns" how to associate a response pattern to a given input pattern by adjusting the neuron's connection weights and the network architecture. The network architecture includes neurons, layers, neuron's inter-connections, and transfer function.

This thesis explores the use of evolutionary algorithms as an alternative training tool for multilayer morphological neural networks. Evolutionary algorithms (EA) (Fogel 1994) are search and optimization methods inspired on natural selection. Three different encoding schemes were used direct encoding, indirect encoding and Cartesian Genetic Programming (CGP) (Miller 2001). Genetic algorithms are used to train the neural networks using the first two encoding schemes. Cartesian genetic programming encoding scheme was adapted to allow the evolution of the morphological neural network. Prototypes of the algorithms were implemented as a toolbox for Matlab 6.

Multidimensional data sets presented by Ritter and Sussner (Sussner 1998) were used for the tests.

1.1 JUSTIFICATION

Many of different algorithms have been proposed to train artificial neural network, most of them work for a specific kind of artificial neural network, including a specific type of transfer function, and neuron's connections. For example, back propagation is used to update connection weights for a given neural network architecture. Using gradient descent of a continuous error function, connection weights are adjusted in order to minimize this error function. Back propagation can not be used if the error function is not continuous or differentiable. Back propagation may not be able to find the global minimum, because it may be possible for the algorithm to get stuck in a local minimum. In addition, gradient descent adjusts exclusively connection weights for particular network architectures, but the algorithm does not adjust the network architecture to define the optimum neural network for a particular problem.

Recently, evolutionary algorithms are able to evolve connection weights as well as network topology simultaneously. Evolutionary algorithms search for the global maximum in infinite, very complex, multimodal and non-differentiable search space, looking for the best artificial neural network without focusing in a specific problem.

The mathematical model used by the morphological neuron is completely different from the model used by traditional neural network. The maximum and minimum operation results in a non-continue, non-differentiable function, therefore the resulting neural network properties are completely different from those properties of

traditional neural networks. This thesis explores the use of evolutionary algorithms as a learning algorithm for morphological neural networks.

1.2 OBJECTIVE

The objective of this thesis is to explore the use evolutionary algorithms to train multilayer morphological perceptrons. Two different evolutionary approaches are used: Genetic Algorithm and Cartesian Genetic Programming. Different learning approaches are explored including supervised learning and reinforcement learning of the neural network.

1.3 CONTRIBUTIONS

The main contribution of this thesis consists of the introduction of three comprehensive evolutionary learning algorithms for multilayer morphological perceptrons:

a. *Direct Encoding Method.*
    i. The use of genetic algorithms as a learning tool for a (fixed architecture) two layers Morphological Perceptrons.
    ii. The introduction of adaptive mutation for the evolution of MLMP as a technique to speed up the convergence of the evolutionary process.
    iii. Introduction of a penalty function to reduce the number of unnecessary neurons from the neural network.

b. *Indirect Encoding Method*

i. Learning algorithm able to produce a Multilayer Morphological Perceptron which is able to solve most pattern classification problems, without considering patterns distribution.

ii. Evolutionary learning algorithm for a maximum of 3-layers Morphological Perceptron, which is good enough to solve most pattern classification problems.

iii. Indirect evolution of morphological neural network's architecture, including number of neurons and connection weights simultaneously.

c. Cartesian Genetic Programming learning algorithm

i. Learning algorithm able to produce a Multilayer Morphological Perceptron which is able to solve pattern classification problems without considering patterns distribution.

ii. The introduction to the use of Cartesian Genetic Programming as an evolutionary learning tool for Multilayer Morphological Perceptrons.

iii. The introduction to the use of the morphological neuron computational model as the node function using Cartesian Genetic Programming.

iv. Simultaneous evolution of morphological neural network's architecture, including number of neurons, neuron interconnection, and connection weights.

v. Make use of a penalty function to reduce the number of unnecessary neurons.

1.4 OVERVIEW

This thesis is organized as follow:

Chapter 2 introduces the concepts of Artificial Neural Networks. Chapter 3 describes the new paradigm of Morphological Neural Networks. Chapter 4 introduces the concepts of Evolutionary Algorithms, Genetic Algorithms and Cartesian Genetic Programming. Chapter 5 presents a survey of related works organized in two subtopics: i) Evolutionary Artificial Neural Networks; and ii) Morphological Learning Algorithms. Chapter 6 presents the evolutionary learning algorithms for multilayer morphological perceptrons: i) Direct Encoding Method; ii) Indirect Encoding Method; and Cartesian Genetic Programming. Chapter 7 describes the toolbox designed for Matlab to train multilayer morphological neural networks. Chapter 8 presents the performance analysis and results, finally Chapter 9 presents conclusions. Appendix A provides the source code for all the methods used by the Evolutionary Morphological Learning Algorithm Toolbox.

CHAPTER 2

ARTIFICIAL NEURAL NETWORKS

2.1 INTRODUCTION

This chapter describes the paradigm of Artificial Neural Networks. A large assortment of different neural networks had been developed, each of them with the ability to solve a particular problem, and allow its application to various field in science and engineering. This chapter presents an introduction to artificial neural networks and how they can be effectively used in pattern recognition problems, pattern classification, speech recognition, and others.

Human's brain is built of thousand of a specific cell, which provides us with our abilities to remember, think and apply previous experiences. Each of these cells, known as neurons, can be connected with other thousands of neurons. Figure 2.1 shows the components of a neuron, which are the cell body, the branching extensions called dendrites for receiving the inputs, and an axon that carries the neuron's output to the dendrites of other neurons.

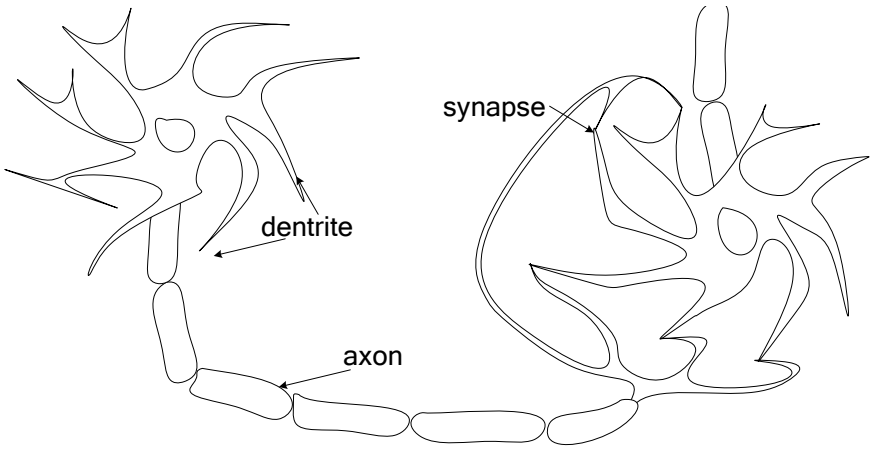


Figure 2.1 Biological neuron

Artificial Neural Networks (ANNs) are a component of artificial intelligence that simulates real brain's neurons. Also known as *parallel distributed processing*, or *connectionist models,* artificial neural networks are information processors inspired by the way the highly interconnected structures of the brain process information. Artificial neural networks are mathematical models that emulate some properties observed from the biological neural network: the knowledge is acquired by the network through a learning process and the synaptic weight is used to store the knowledge. Computations are performed through the passing of signals within a structured arrangement of highly interconnected processing units in response to a given input signal.

The artificial neural network model was introduced by McCulloch and Pitts, after the definition of the computational model for the traditional perceptron in 1943. This is an artificial neuron with a hard-limiting activation function. Since that artificial neural networks have been implemented to solve a variety of problems involving pattern classification and pattern recognition.

2.2 ARTIFICIAL NEURAL NETWORKS COMPONENTS

2.2.1 Artificial Neuron

The basic element of an artificial neural network is the artificial neuron. The artificial neuron simulates some of the operations the natural neuron can perform. The neuron receives as inputs the outputs from other neurons, if the combined strength of the signal reaches a specific threshold; the neuron sends a signal to all the neurons waiting for the output. Figure 2.2 shows an example of an artificial neuron.

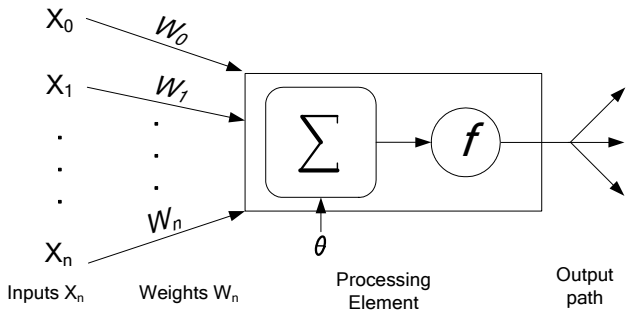


Figure 2.2 The artificial neuron model

The symbols $x_0,\ldots, x_n$, represent the strength of the input signals, $w_0,\ldots w_n$, represent the connection strengths of the given input signal, and the output is represented by the symbol *y*. The computational model for the traditional neuron is given by Equation 2.1.

$$y = f\left(\left[\sum_{i=1}^{n} w_i \cdot x_i\right] - \theta\right), \tag{2.1}$$

where θ is a threshold value and *f* is the neuron's activation function.

The most commonly used activation function (also known as transfer function) is the hard-limiting function shown in Equation 2.2. However, this one can vary from neuron to neuron.

$$f : \Re \to \Re$$
$$x \to \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ else} \end{cases} \tag{2.2}$$

Some of the most common transfer used activation functions shown in Figure 2.3

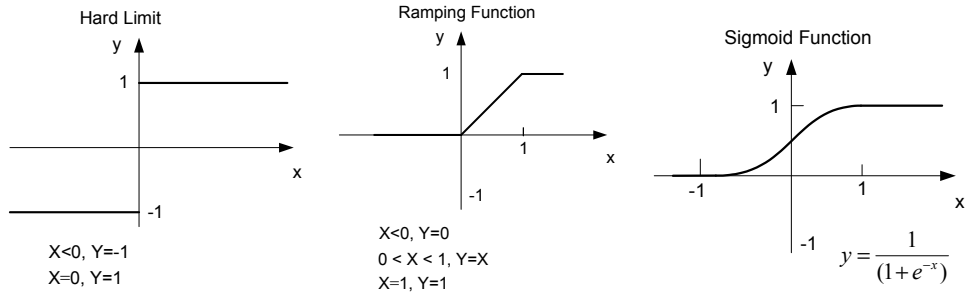


Figure 2.3 Most commonly used transfer functions

2.2.2 Architectural Elements of an Artificial Neural Network

The basic components of neural network architecture are neurons, the layers, and neuron connection. A neural network consists of a set of neurons highly interconnected, grouped into three types of layers: the input layer, output layer and the hidden layers. The behavior of the neural network depends on the interaction between the neurons. Interaction between network components depends on the type of connection that is used to pass messages between neurons. There are four types of synaptic connections: feed forward, feedback, lateral and time-delayed connections. It is important to highlight that synaptic connections may be fully interconnected or partially interconnected.

Feed forward connections are used to propagate the output from the neurons of a lower layer to neurons of an upper layer, as shown in Figure 2.4.

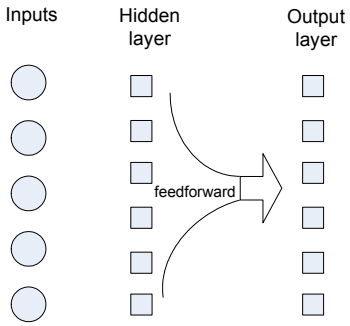


Figure 2.4 Feed forward connections.

Feedback connections are used to send the output from neurons of an upper layer back to neurons of a lower layer, as shown in Figure 2.5.

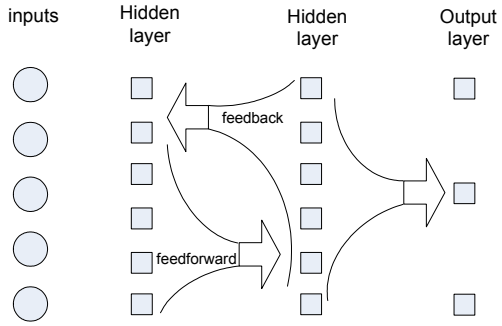


Figure 2.5 Feedback connections.

Lateral Connections are usually used in the output layer, when the output with the higher value predominated over all the other output nodes, as shown in Figure 2.6.

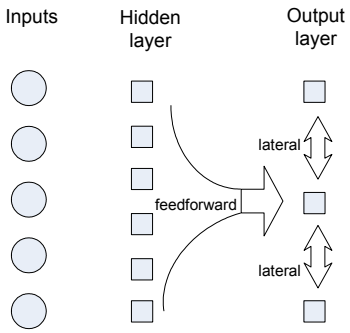


Figure 2.6 Lateral connections.

Time delayed connections add elements to the network to yield temporal dynamics models. These connections are used in recurrent neural networks which are networks that, also, use feedback connections.

2.3 LEARNING PROCESS FOR AN ARTIFICIAL NEURAL NETWORK

The brain learns from knowledge collected from previous experiences, while artificial neural networks the learning process is achieved by changing the connection weights between neurons. The connection strength of two neurons is represented by the specific connection weight for that connection.

Learning algorithm in an artificial neural network are classified into supervised, unsupervised, and reinforcement learning. Supervised learning is based on direct comparison between the actual output of a system and the desired correct output. Unsupervised learning is based on the correlations among input data. No information about the "correct output" is available for learning. In reinforcement learning, the system receives inputs and evaluation actions and the system has to learn how to map the inputs to actions resulting in the best performance.

2.3.1 Back-Propagation

Back propagation is a learning algorithm for feed-forward neural network which minimizes a continuous error function. The error function is the difference between the actual output α_d and the desired output t_d, presented in Equation 2.3.

$$E\left[\vec{w}\right] = \frac{1}{2}\sum_{d=1}^{N}\left(\alpha_d - o_d\right)^2 \tag{2.3}$$

Each pattern is defined as $\langle \vec{a}, t \rangle$ *where* $\vec{a}$ *is the vector of the inputs values and t is the target output value.* α *is the learning rate, p represent the initial conditions (randomly initialized), m represents the layer and M represents the total amount of layers.* W^m *are the connection weights from layer m,* b^m *are the biases for the neurons from layer m,* f^m *is the transfer function for the neurons from layer m, and*

$$\dot{F}^m(n^m) = \begin{bmatrix} \frac{\partial f^m(n_1^m)}{\partial n_1^m} & 0 & \cdots & 0 \\ 0 & \frac{\partial f^m(n_2^m)}{\partial n_2^m} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \frac{\partial f^m(n_{s^m}^m)}{\partial n_{s^m}^m} \end{bmatrix}, \text{ where } \dot{f}(n_j^m)\frac{\partial f^m(n_j^m)}{\partial n_j^m}$$

- *Propagate the input forward through the network*
  - $a^0 = p$
  - $a^{m+1} = f^{m+1}\left(W^{m+1}a^m + b^{m+1}\right)$ for m=M-1,…,2,1
  - $a = a^M$
- *Propagate the sensitivities backward through the network*
  - $s^M = -2F^M(n^M)(t-a)$
  - $s^m = F^M(n^m)\left(W^{m+1}\right)^T s^{m+1}$
- *Update weight and biases using the gradient descent:*
  - $W^m(k+1) = W^m(k) - \alpha s^m\left(a^{m-1}\right)^T$
  - $b^m(k+1) = b^m(k) - \alpha s^m$

Figure 2.7 Back propagation learning algorithm described by Hagan (Hagan and Demuth and Beale 1996)

Back propagation uses gradient information to updates connection weights for fixed network architecture in order to reduce the error in classification. The network architecture uses sigmoid as the transfer function for the hidden layers. Figure 2.7 describes the algorithm used for back propagation.

2.4 TRAINING OF AN ARTIFICIAL NEURAL NETWORK

The training of an artificial neural network can be seen as the search for the best architecture and connection weights in an architectural space where each point represents a neural network architecture. Some neural networks architecture properties, like the number of neurons, number of layers, and total number of misclassified patterns are used to define a surface in the search space. According to Miller (Miller and Todd and Hegde 1989), this surface is infinitely large since the possible number of nodes and connections is not fixed. The surface is not differentiable since the changes in number of neurons, layers and connection is discrete. This is not a continuous function as in traditional optimization problems. In addition, similar architectures may have different performance, but different architectures may have similar performance.

CHAPTER 3

MORPHOLOGICAL NEURAL NETWORKS

3.1 INTRODUCTION

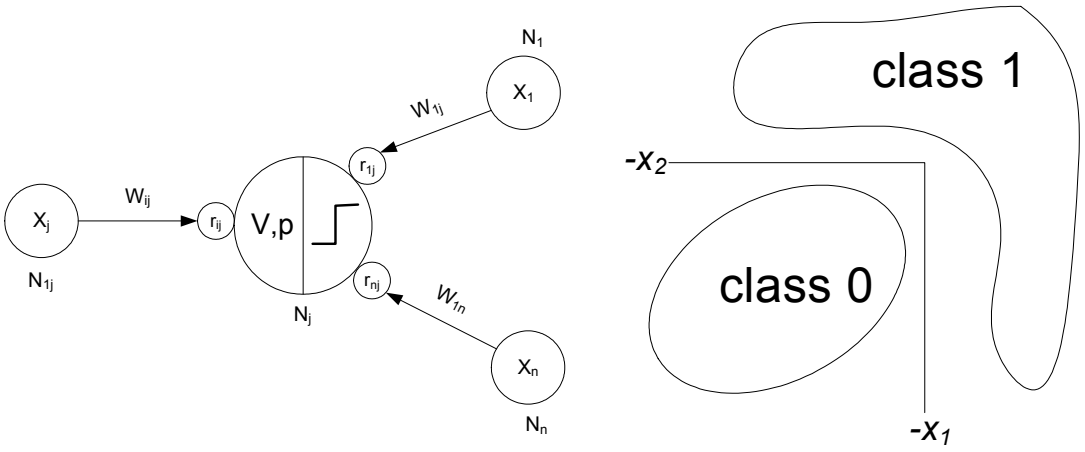
The new paradigm of Morphological Neural Networks (MNNs) was introduced by Ritter, Sussner and Wilson (Ritter and Sussner 1996, Sussner 1998). These neural networks replace the classical operations of multiplication and addition by addition and maximum or minimum operations, respectively. The computations occurring in the morphological neural network are based on the algebraic lattice structure $(\Re_{-\infty}, \vee, +)$ and $(\Re_{\infty}, \wedge, +')$, where $\Re_{-\infty}$ and $\Re_{\infty}$ represent the extended real number systems $\Re_{-\infty} = \Re \cup \{-\infty\}$ and $\Re_{\infty} = \Re \cup \{\infty\}$. The symbol + denotes the usual addition with the additional stipulation that $a + (-\infty) = (-\infty) + a = -\infty;\ \forall a \in \Re_{-\infty}$ and $+'$ is defined by $a +' b \equiv a + b$ for $a, b \in \Re_{\infty}$, and $a + \infty = \infty + a = \infty;\ \forall a \in \Re_{\infty}$. The symbols $\vee$ and $\wedge$ denote the maximum and minimum operators, respectively, with the additional stipulation that $a \vee (-\infty) = (-\infty) \vee a = a;\ \forall a \in \Re_{-\infty}$ and $a \wedge \infty = \infty \wedge a = a;\ \forall a \in \Re_{\infty}$. The maximum and minimum operations allow performing a nonlinear operation before the application of the activation function, resulting in properties completely different from those properties of traditional neural networks.

3.2 MORPHOLOGICAL NEURAL NETWORKS

Morphological neural networks are a new type of neural network introduced by Ritter, Beavers and Sussner. Differing from traditional neural networks, these neural networks replace the operator of multiplication by the operator of addition, and the operator of addition is replaced by the maximum operator or by the minimum operator. The morphological neuron follows the mathematical model described in Equation 3.1:

$$f\left(p_j \cdot \bigvee_{i=1}^{n} r_{ij}\left(x_i + w_{ij} \right) \right) \tag{3.1}$$

where $\vee$ is the maximum operator (or minimum operator $\wedge$), n is the number of dimensions of the pattern to be classified, x_i is the value of the i-th input of the neuron, w_{ij} denotes the synaptic weight associated between the i-th neuron and the j-th neuron, r_{ij} represents the inhibitory or excitatory pre-synaptic values and p_j represents the inhibitory or excitatory post-synaptic value, where r_{ij} and p_j can be set to values of {+1, -1}. Figure 3.1 presents a graphical representation of the morphological model and the decision boundary defined by a morphological neuron in a $\Re^2$ space.



a. b.

Figure 3.1 (a) Computational Model for Morphological Neural Network (b) Morphological Perceptron

Morphological Neural Networks have been used for scene recognition, and self-localization, as part of a vision based navigation framework for mobile robots. Additionally, it has been used in image restoration by reconstruction of patterns from noisy inputs.

3.3 SINGLE LAYER MORPHOLOGICAL PERCEPTRON

Single layer morphological perceptron is a binary pattern classifier like the traditional perceptron. In other words, the patterns forwarded as inputs for the neural network are classified as belonging to either class C_0 or class C_1. The morphological perceptron uses a hard-limit transfer function, as shown in Equation 3.2:

$$f : \mathbb{R} \to 0,1$$
$$x \to \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ else} \end{cases} \tag{3.2}$$

Let $W = [w_1, w_2, \ldots, w_n] \in \Re^n$ represent a set of weights, and θ the threshold. The traditional perceptron assigns a pattern $x \in \Re^n$, to class C_0 if

$$f\left(\left[\sum_{j=1}^{n} x_j \cdot w_j\right] - \theta\right) = 0 \tag{3.3}$$

otherwise the pattern is assigned in class C_1. The morphological perceptron assigns a pattern $x \in \Re^n$ to the class C_0 if

$$f\left(\left[\bigvee_{j=1}^{n} \left(x_j + w_j\right)\right] - \theta\right) = 0 \tag{3.4}$$

otherwise the pattern is assigned to the class C_1. According to the dual nature of the morphological neuron a pattern $x \in \Re^n$ can be assigned to the class C_0 if

$$f\left(\left[\bigwedge_{j=1}^{n}\left(x_j +' w_j\right)\right]-\theta\right)=0 \tag{3.5}$$

otherwise the pattern is assigned in class C_1. Since w_j and θ are constants values, the Equation 3.4 and Equation 3.5 can be rewritten as shown in Equation 3.6 and Equation 3.7, respectively.

$$f\left(\bigvee_{j=1}^{n}\left(x_j + w_j\right)\right)=0 \tag{3.6}$$

$$f\left(\bigwedge_{j=1}^{n}\left(x_j +' w_j\right)\right)=0 \tag{3.7}$$

The decision boundaries defined by Equation 3.4 in a $\Re^2$ space is shown in Figure 3.2a, and the corresponding space for Equation 3.5 is shown in Figure 3.2b.

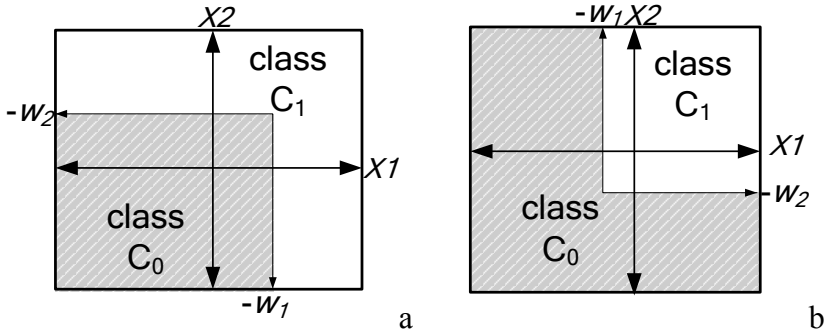


Figure 3.2 Decision boundaries defined by the morphological perceptron. (a) Decision boundary defined a neuron using the mathematical model in Equation 3.6 and (b) decision boundary defined by a neuron using the mathematical model in Equation 3.7 in a $\Re^2$ dimensional space.

Sussner (Sussner 1998) described the effects produced applying different pre-synaptic values to the morphological model shown in Equation 3.1. The value of r represents the pre-synaptic response at i-th synapse. A value of r_i = -1 represents an inhibitory response and a value of r_i = +1 means an excitatory response. Figure 3.3 shows the different resulting effects produced in the decision boundaries defined in a $\Re^2$ space

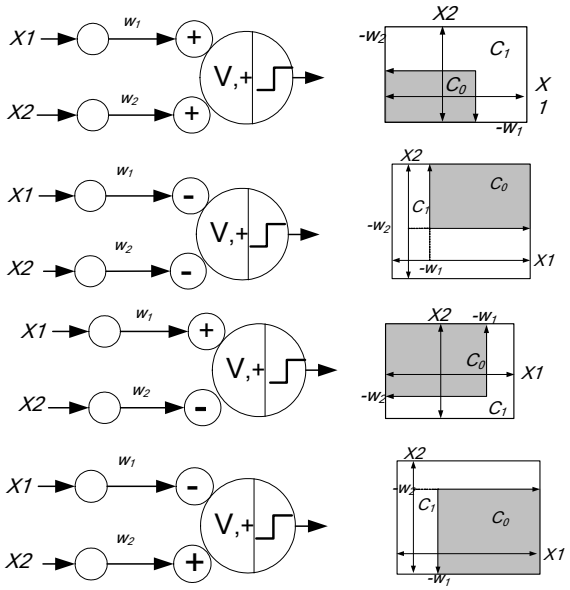


Figure 3.3 Resulting decision boundaries produced by changing pre-synaptic values of a morphological neuron using a maximum operator in a $\Re^2$ space.

Since the morphological neuron can use two different operators, the maximum operator from Equation 3.1 can be replaced by a minimum operator. The resulting decision boundaries assigning different pre-synaptic values are shown in Figure 3.4.

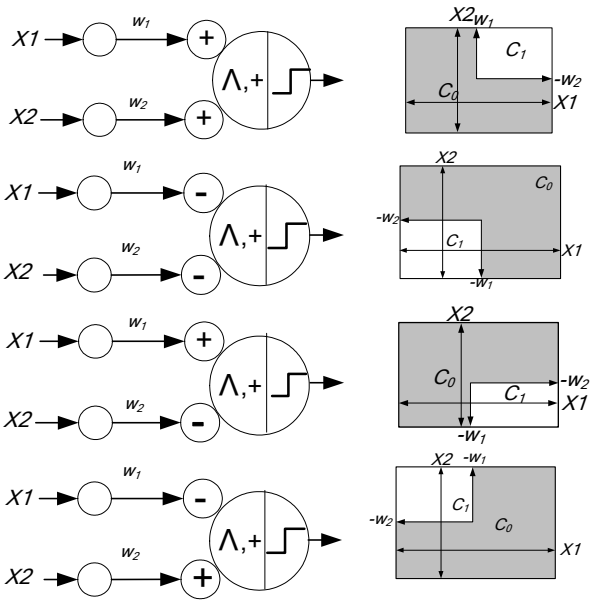


Figure 3.4 Resulting decision boundaries produced by changing pre-synaptic values in a morphological neuron using a minimum operator in a $\Re^2$ space.

3.4 EXAMPLE

Lets define classes C_0= {(0,0) }, C_1= {(1,0), (0,1), (1,1)}, as shown in Figure 3.5. In order to classify these patterns the morphological operator used by the neuron must be a maximum operator ($\vee$), the corresponding connection weights may be $w_1 = -\frac{1}{2}$ and $w_2 = -\frac{1}{2}$, and the pre-synaptic values must be $r_1 = +1$ and $r_2 = +1$.

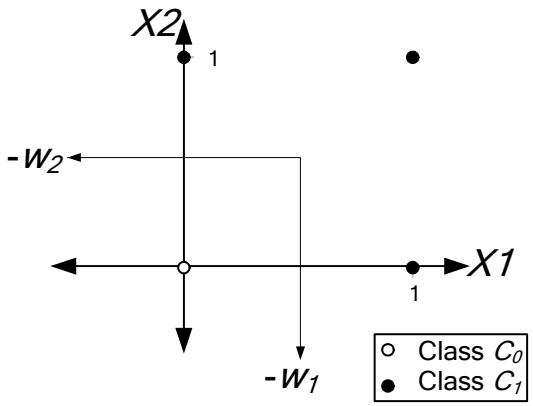


Figure 3.5 Decision boundary of the morphological perceptron

The morphological perceptron can be tested using patterns p_1 = (0, 0) and p_2 = (0,1). Pattern p_1 is classified in class C_0, as shown in Equation 3.7. Pattern p_2 is classified in class C_1, as shown in Equation 3.8.

$$f\left(+1\bullet\left[0-\frac{1}{2}\right]\vee+1\bullet\left[0-\frac{1}{2}\right]\right)=f\left(-\frac{1}{2}\vee-\frac{1}{2}\right)=f\left(-\frac{1}{2}\right)=0 \tag{3.7}$$

$$f\left(+1\bullet\left[1-\frac{1}{2}\right]\vee+1\bullet\left[0-\frac{1}{2}\right]\right)=f\left(\frac{1}{2}\vee-\frac{1}{2}\right)=f\left(\frac{1}{2}\right)=1 \tag{3.8}$$

3.5 MULTILAYER MORPHOLOGICAL PERCEPTRON

The limitations of Single Layer Morphological Perceptron become evident when the patterns are grouped in multiple clusters, and those subsets can no be separated by a single morphological neuron. A traditional example is the XOR logic function. The XOR is a binary operator on $\{0,1\}^2$ such that for all

$$(a,b)\in\{0,1\}^2,\ a \text{ XOR } b=\begin{cases}0 \text{ if } a=b\\ 1 \text{ else}\end{cases} \tag{3.4}$$

Two different classes C_0= {(0,0), (1,1)}, and C_1= {(1,0), (0,1)} are defined as shown in Figure 3.6. The multilayer morphological perceptron is able to overcome this problem by adding additional hidden layers which process the output of the first layer resulting in a nonlinear decision boundary.

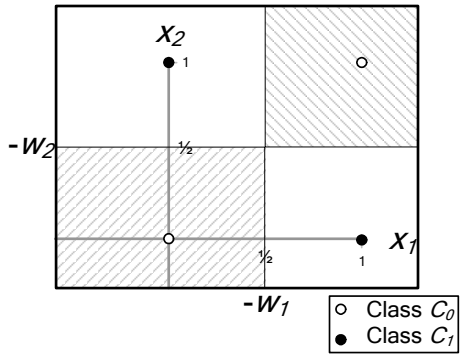


Figure 3.6 Decision boundaries for the XOR classification problem using morphological neurons.

It is not possible to separate patterns from class C_0 and class C_1 with a single morphological neuron, to classify these patterns correctly the output of two neurons in the first layer must be combined, and connected to a neuron in a second layer as shown in Figure 3.7.

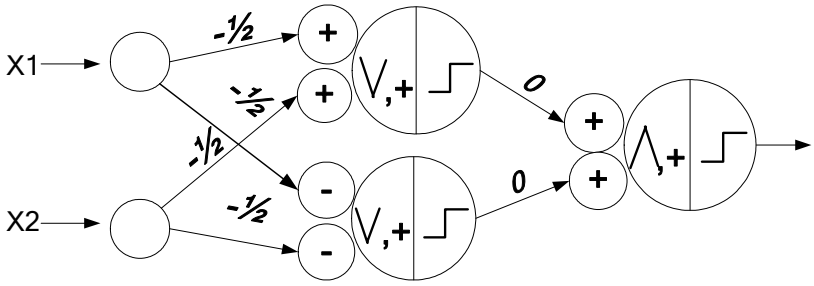


Figure 3.7 Morphological neural network used to solve the XOR classification problem.

CHAPTER 4

EVOLUTIONARY ALGORITHMS

4.1 INTRODUCTION

Evolutionary Algorithms (EA) are based on the idea that basic concepts of biological reproduction and evolution can be used as a model to solve problems using computers to emulate the same process. Evolutionary Algorithms are a robust heuristic search and optimization mechanism which can be applied to problems where normal solutions are not available or generally lead to unsatisfactory results. The most important areas of research in simulated evolution are: evolutionary strategies, evolutionary programming, and genetic algorithms. The three main operators in Evolutionary Algorithms are selection, recombination, and mutation. A population of possible solutions is maintained and encoded into data structures called chromosomes of an organism. Elements of the population are able to mate, mutate, and evolve, directed by the fitness funtion that evaluates the quality of the population with respect to a preset goal.

4.2 SEARCH ALGORITHMS

The goal of optimization problem is to find the best solution where several feasible solutions are available. An evaluation function or fitness function is used for determining how good each particular solution is and the goal is to find the best solution. Given a set of possible solutions, also know as search space, there may be several local

maximum or sub-optimal values, but the over all highest value of the set is considered the optimal value. If the search space is small, all the possible solutions can be examined, but as the search space grows in size, this exhaustive search becomes impractical.

In a particular problem, the search space and the evaluation function for the elements in the search space in terms of performance define a landscape. The landscape consists of hills, valleys, and other geographical features. Figure 4.1 shows some features that may exist in the resulting landscape. Points *a*, *d*, *f,* and *h* are the top of the hills surrounded by points with lower values. Points *c*, *e* and *g* are the bottom of the valleys, surrounded by other points with higher values. Point *b* in the graph is the middle of a plateau. The performance of the points next to the plateu are exactly the same.

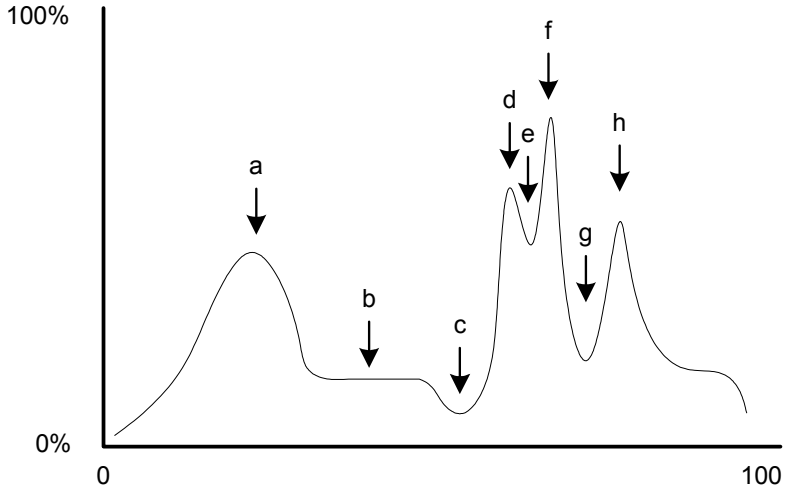


Figure 4.1 Search space's landscape

Traditional search algorithms, such as the gradient descent, examine a point in the search space at the time, and the next point to be examined is obtained based on the current position. Usually, the next point to be examined has better performance than the previous point. The process continues until the top of a hill is reached. This point may be a local maximum, however since the new position is based on the previous one, it may

not be possible to make a drastic move to get onto the slope of a higher hill, this is known as the hill climbing effect. Another deficiency of this algorithm is that it is possible to get stuck on a plateau. This may happen if the algorithm is unable to move far away from the flat region. Another problem with this algorithm comes from the fact that the final result depends on the starting search point, it may be possible that different starting points produce different results. However this can be considered as an advantage because different results may provide the best solution among all local maximum points.

4.3 EVOLUTIONARY COMPUTATION

Evolutionary Algorithms is based on the basic concepts of biological reproduction and evolution that is used as a model to solve problems using computers to emulate the same process. All possible solutions for a problem are represented with a particular genetic representation scheme. A set of solutions or individuals is generated to form the initial population of organisms, as shown in Figure 4.2. Each organism is evaluated using a fitness function specific to the problem. The fitness function measures the performance of the organism according to specific characteristics. Using a particular selection algorithm based on the fitness value, some organisms are chosen to be the parents for the next generation. New organisms, also known as offspring are produced after the information contained in the parents is combined using reproduction operators such as crossover and mutation. Finally, some organisms are selected from the old population and from the new offspring to form the population for the next generation. These steps are repeated until a solution that satisfies the selected criteria is found.

Evolutionary algorithms overcome some of the deficiencies presented by the hill climbing effect by exploring a set of possible solutions at the same time. Since evolutionary algorithms are population based, even if some of the solutions in the initial population are a plateau or a local maximum, the genetic operations may be able to produce a totally different set of possible solutions in the next generation, moving the set of solutions toward the global maximum. In addition, several initial population sets may lead to similar final set if the desired feature is present in the initial population.

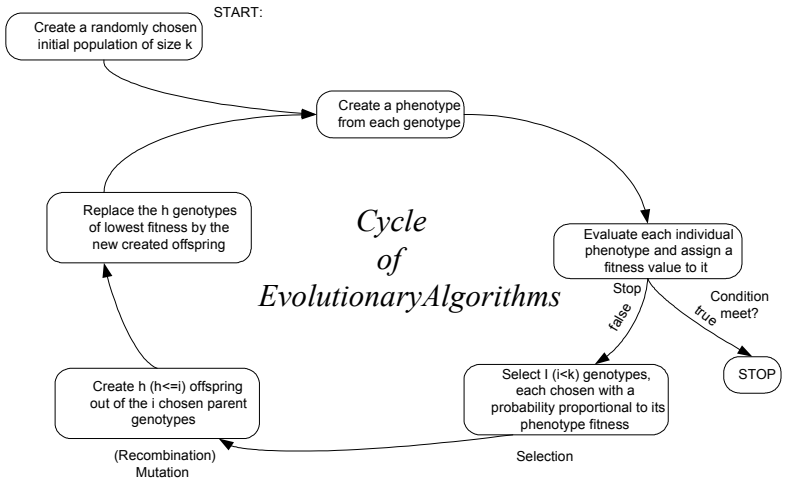


Figure 4.2 Cycle of Evolutionary Algorithms

4.4 GENETIC ALGORITHMS

Genetic Algorithms are robust search and optimization algorithms introduced by Holland in 1970s. GA is one of the most popular areas of research in evolutionary algorithms, particulary useful for multidimensional optimization problems in which the chromosome can encode the values for different variables to be optimized. The most important factors to consider in genetic algorithms as a search mechanism are: representation scheme, fitness function, reproduction operators, and selection methods.

The genotype consists of a set of genes inherited from parents that code a trait. The most common representation scheme uses a fixed length gene string. The gene may be of any size, but usually it is binary. The gene may consist of a discrete set of values represented by integers or by a continuous set of values represented by floating point numbers, or a combination of them. Figure 4.3 shows how a single neuron may be encoded into the genotype using integer numbers to represent the neuron operation and pre-synaptic response, as well as floating point numbers to represent connection weights.

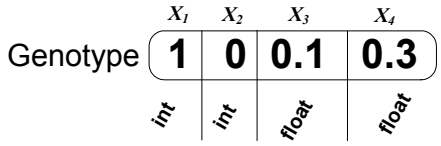


Figure 4.3 Genotype representation using different types of representation for the genes.

The upper and lower bounds describe the valid range for each gene in the chromosome. Each gene may be used individually or combined with other genes during the decoding of the genotype into the *phenotype*. The *phenotype* manifests physical properties of the individual. Each gene is associated to a special mapping function or decoding function which translate the content of the gene into a physical property. Table 4.1 describes the mapping function associated to each gene in a chromosome.

operator		*post-synaptic*		*connection weights*			
x_1	{min,max}	x_2	{-1,1}	$\boldsymbol{x_3}$	double (-10.0, 10.0)	$\boldsymbol{x_4}$	double (0.0, 10.0)
0	min	0	-1		(20.0)* $\boldsymbol{x_3}$-10.0		(10.0)* $\boldsymbol{x_4}$+0.0
1	max	1	1				

Table 4.1 Mapping function for each gene from the genotype shown in Figure 4.3.

The fitness function must be able to evaluate every component in the gene string. The fitness function is always specific to the problem and measures the performance of the organism in terms of how good is the solution for the problem.

The most common reproduction operators are crossover and mutation. The crossover combines the genetic information from parents and produces offsprings that are consistent with the representation scheme. The resulting offspring may be completely different to either parents, slightly different or even the same as the parents. Two crossover mechanism used in this thesis are the arithmetic crossover and order base crossover. Arithmetic crossover selects a random weight $w \in [0,1]$ and creates children from weighted averages of the parents, as shown in Equation 4.1. Figure 4.4 shows an example of crossover of two parents, and the resulting offspring.

$$\begin{aligned} \text{offspring}_1 &= w \cdot \text{parent}_1 + (1\text{-}w) \cdot \text{parent}_2 \\ \text{offspring}_2 &= w \cdot \text{parent}_2 + (1\text{-}w) \cdot \text{parent}_1 \end{aligned} \tag{4.1}$$

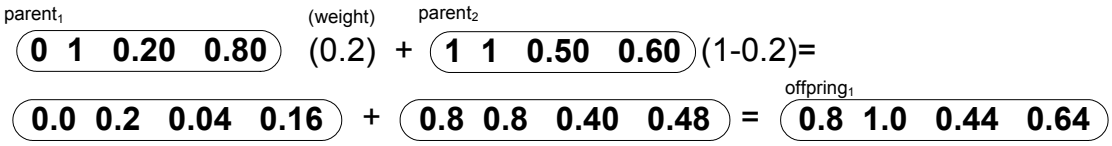


Figure 4.4 Arithmetic crossover of two parents producing one offspring.

It is important to remember that each gene in the resulting offspring needs to enforce the range constrains defined for it, for this reason the first two genes in the resulting offspring are rounded because only integer values are allowed in those genes, as shown in Figure 4.5

Figure 4.5 Gene contrain reinforcement after crossover.

The mutation introduces additional information that the crossover is not able to introduce, since the crossover only recombines the information from the parents. The most commonly used mutation is the single point mutations. Single point mutation selects

a gene randomly and changes its content enforcing the range constrains defined for the gene, as shown in Figure 4.6.

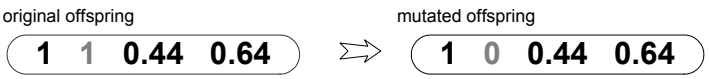


Figure 4.6 Single point mutation

The selection method chooses the organism to be parents based on the fitness value. According to the fitness value the probability of being selected as parent is assigned to the organism. That organism that has non-zero fitness may become a parent. One common selection method is the wheel roulette selection. The wheel roulette method assigns the probability to each organism and the fitness value for that organism is divided by the total sum of the fitness values for the population.

4.5 GENETIC PROGRAMMING

Genetic Programming (GP) (Koza 1992) is an extension of the genetic model for learning into the space of feasible solutions. The objects in genetic programming are programs that represent organisms that when executed are candidate solutions to the problem. These programs are expressed as parse trees, rather than as lines of code. Differing from genetic algorithms these objects are not fixed-length character strings.

4.5.1 Cartesian Genetic Programming

The Cartesian Genetic Programming (CGP) (Miller and Thomson 2000, Miller 2001) method is represented as an indexed directed graph of nodes. Distributed in a rectangular array. The nodes represent operations on the data received by the inputs. Integer values are assigned to inputs, nodes, operations, and outputs. Node operations may be simple operations such addition, subtraction, multiplication or division.

The genotype is represented as a fixed length array of integers, as shown in Figure 4.7. The genotype maintains a list of the inputs and the function associated to a particular node as shown in Figure 4.8. A unique integer value is assigned to all the inputs, nodes and output. The information maintained by the nodes is encoded in the chromosome. The first three integers in the chromosome encode the information from the upper left node shown in Figure 4.8. The first two values represent the inputs received by the nodes, and the third value represents the node operation or function. The last elements from the chromosome indicate the output nodes. Variable length phenotypes are produced by unexpressed genes carried in the genotype, as shown in Figure 4.9. Unexpressed genes are those nodes that are not in the path of nodes that directly connect from the input layer to the output layer.

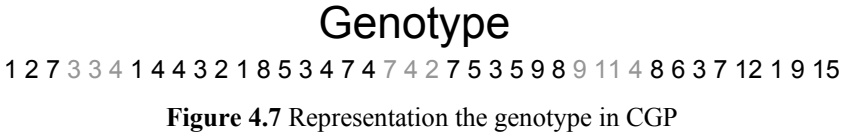


Figure 4.7 Representation the genotype in CGP

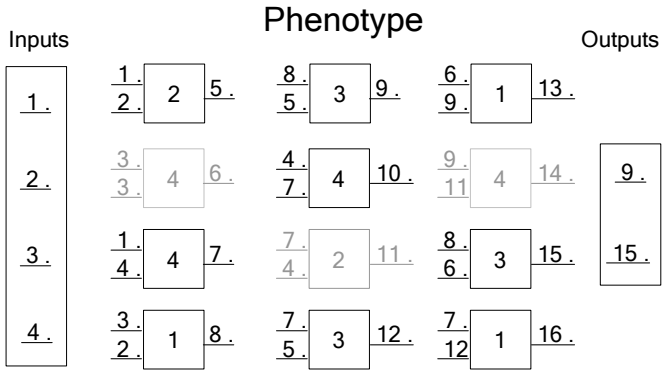


Figure 4.8 Graph of nodes used to represent the phenotype in CGP.

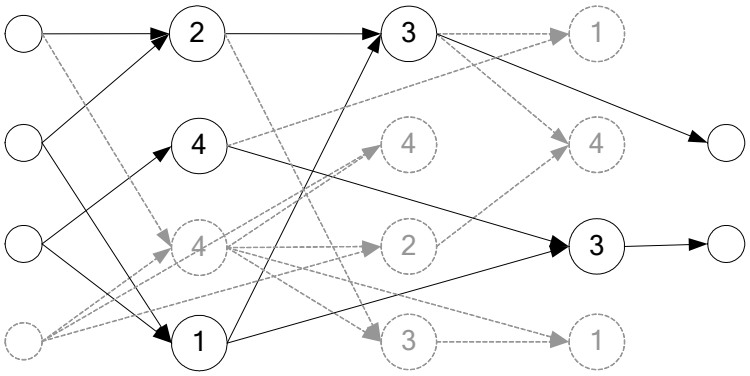


Figure 4.9 Resulting organism with unexpressed nodes

CHAPTER 5

LITERATURE REVIEW AND PREVIOUS WORK

5.1 INTRODUCTION

This chapter describes the use of Evolutionary Learning algorithms as a learning tool for traditional neural networks in addition to different evolution trends in evolutionary artificial neural networks including evolution of connection weights, and evolution of architecture. Finally, this chapter describes the learning algorithms currently used for multilayer morphological perceptrons.

5.2 EVOLUTIONARY ARTIFICIAL NEURAL NETWORKS

The algorithm most widely used to train neural networks is the back propagation algorithm which is a local gradient search method. Convergence is not always obtained and the algorithm may get stuck in a local maxima. On the other hand, evolutionary algorithms (Fogel 1994, Fogel and Fogel 1996, Saravanan and Fogel 1995) usually avoid local maxima by searching in several regions simultaneously. And the only information they need is some performance value that determines how good a given set of weights is and no gradient information is required. Several studies have been conducted in the Evolutionary Artificial Neural Network (EANN) field as an alternative to the gradient information. EANN refers to an Artificial Neural Network that uses Evolutionary Algorithms to evolve connection weights and architecture (Yao 1999). EANN can be

seen as a system that adapts the architecture and rules dynamically without human intervention.

Evolution in artificial neural networks can be found at three different levels: connection weights, architectures, and learning rules (Branke 1995), (Espacia-Alcaza and Sharman 1996), (Fukuda and Kohno and Shibata 1993), (Gruau 1992), (Harp and Samad and Guha 1989), (Hintz and Spofford 1990), (Howard 1995), (Jacob and Rahder 1993), (Miller and Todd and Hegde 1989). Most of the studies focus on three different approaches. The first approach is fixed architecture and the evolutionary algorithm is used to search for a set of weights that best performs on the network. In the second approach, evolutionary algorithms are used to develop, simultaneously, connection weights and network architecture. In the last approach, the evolution of learning rules can be regarded as a process of "learning to learn".

Using fixed architecture method, the architecture of an Artificial Neural Networks is known before the learning process, and it does not changed during the evolution of the connections weights. Evolutionary Algorithms can be used in the evolution to find a sub-optimal set of connection weights globally without computing gradient information. Many research and application has been conducted in evolutionary algorithms (Miller and Todd and Hegde 1989), (Koza 1992), (Kitano 1990), (Gruau 1992), (Yao 1999) because they can deal with very large, complex, not differentiable and multimodal spaces.

Recently, a lot of research has been done to design architecture and weights of the Artificial Neural Network simultaneously (Branke 1995), (Esparcia-Alcaza and Sharman 1996), (Gruau 1992), (Karunanithi and Das and Whithley 1992), (Kitano90), (Koza and

Rice 1991), (Koza 1992), (Jacob and Rehder 1993), (Vonk et al 1995). The architecture of the Artificial Neural Networks includes the topological structure, i.e., connectivity and transfer function of each node. One of the key issues in encoding the Artificial Neural Network is to decide how much information should be encoded into the chromosome. One of the methods is direct encoding of the neural network, where details of the neural network are described in the chromosome in such a way that the gene may be used directly as a working neural network (Whitley and Starkweather and Bogart 1990). Using indirect encoding, only weight and biases details of the neural network are encoded in the chromosome and no details of the connections are used (Kitano 1990), (Koza 1992), (Gruau 1992), (Luke and Spector 1996).

Two different approaches can be taken in the direct encoding: the first separates the evolution of the architecture from that of the connection weights (Howard 1995). The second approach evolves the architecture and the connection weights simultaneously (Koza 1992), (Gruau 1992), (Gruau and Whitley and Pyeatt 1995).

Indirect encoding has been used to reduce the length of the genotype representation of the network architecture (Gruau 1992), (Hussain and Browse 1998), (Kitano90), (Luke and Spector 1996). Different indirect encoding schemes include structural encoding, parametric encoding, and grammar encoding.

Structural encoding defines the structure of the network is embedded in the chromosome. Koza (Koza and Rice 1991), (Koza 1992), applied genetic programming to discover both the architecture and the weights of a neural network. In this work, the neural network was represented as a point-labeled tree. Parametric encoding uses certain important aspects of neural network architecture (such as the number of hidden layers,

the number of hidden nodes in each layer, etc.) and is represented by fixed parameters (Harp and Samad and Guha 1989), (Harp and Samad and Guha 1990).

Another technique is grammatical encoding, where the neural network is represented as a sentence of a special language described by a grammar. Two basic approaches to grammar encoding include *developmental grammar encoding,* and *derivation grammar encoding.* Developmental grammar encoding describes the chromosome by grammar rules that will be used to develop a specific neural network structure (Kitano 1990). Derivation grammar encoding design a single fixed grammar and the chromosome contains the derivation sequence which define the network architecture (Jacob and Rehder 1993), (Gruau 1992).

Gruau (Gruau 1992), Gruau and Whitley (Gruau and Whitley 1993), Gruau and Whitley and Pyeatt (Gruau and Whitley and Pyeatt 1995) and Esparcia-Alcazar and Sharman (Esparcia-Alcazar and Sharman 1996) have used genetic programming to create the topology for recurrent neural networks. Luke and Spector (Luke and Spector 1996) showed that graphs and networks can be evolved using an edge encoding scheme. Hussain and Browse (Hussain and Browse 1998) proposed the use attribute grammars in creating a useful and compact genetic encoding of neural networks.

5.3 MORPHOLOGICAL LEARNING ALGORITHMS

There are very few learning algorithms proposed for Morphological Neural Networks. Ritter and Sussner (Ritter and Sussner 1996), proposed an algorithm to train single layer morphological perceptrons on $\Re^n$ space. Sussner (Sussner 1998), proposed a two layers morphological perceptron training algorithm. Differing from the classical

perceptron learning rule, these algorithms converge in a finite number of steps. Lima (Lima et al. 2001) proposed a hybrid algorithm which combines evolutionary algorithms with a nonlinear optimization method based on gradient information to accelerate the convergence. No comprehensive evolutionary algorithms had been presented at this time to train morphological perceptrons

CHAPTER 6

EVOLUTIONARY LEARNING METHODS FOR MULTILAYER MORPHOLOGICAL PERCEPTRONS

6.1 INTRODUCTION

Evolutionary learning algorithms had proven to be successful training traditional neural networks. The chapter describes how to separate patterns into multiple classes using a multilayer morphological perceptron. In addition, three comprehensive learning algorithms based on evolutionary algorithms are presented. Two learning algorithms are based on genetic algorithms and a third one is based on Cartesian genetic programming.

6.2 CLASSIFICATION OF PATTERNS INTO MULTIPLE CLASSES

In general, a morphological perceptron can separate only two classes. In order to classify multiple classes, a vector that contains a binary pattern is assigned to each class, for example:

$$C_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad C_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ and } \quad C_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

A neural network may be trained for each entry in the classification vector. A neural network used to classify all test patterns for the first entry in the vector correctly, requires to assign test patterns from classes that have the value of 0 to a temporary class C_{t0}, otherwise to class C_{t1}. Those temporary classes will be used during the training process of the neural network. Figure 6.1a shows the set of test patterns, and their

corresponding binary vector. Figure 6.1b shows how all test patterns have been regrouped into temporary classes. A multilayer morphological perceptron is built in such a way that it will be able to separate the patterns in the new classes C_{t0} and C_{t1}. The output of that network is assigned to the first entry in the binary vector. Figure 6.1c shows that the test patterns must be regrouped in order to build the neural network for the second entry in the binary vector.

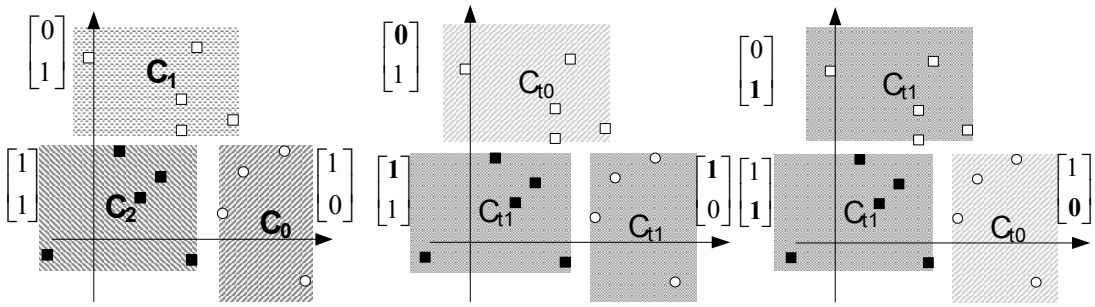


Figure 6.1 Distribution of patterns into temporary groups used during the training process.

6.3 DIRECT ENCODING LEARNING ALGORITHM FOR MULTILAYER MORPHOLOGICAL PERCEPTRONS

The proposed algorithm identifies a set of connection weights, and neuron properties of a two layers feed forward morphological perceptron with only one output.

Genetic algorithms are used to search for the connections weights, pre-synaptic and post-synaptic response values, and neuron operations given a neural network architecture defined before the learning process. The neural network architecture consists of one neuron or two layers feed-forward neural network with only one output node. All the morphological neurons in the neural network use the hard limit transfer function previously defined in Equation 3.2. In addition, the number of neurons and neuron distribution must be known before the training process take place. All the neurons are fully connected, which means that all the neurons in the first layer receive as inputs all

the connections from the inputs layer, and all the neurons in the first layer connect to the neuron in the second layer.

6.3.1 Organism Representation

The phenotype is directly encoded into the genotype, which means that all the genetic information related to the network architecture, connection weights, and neuron information is represented in the genotype. The multilayer morphological perceptron is encoded into the genotype using a tree data structure, where each node in the tree represents a neuron, and each branch represents a connection between two neurons. The terminal nodes of the tree structure represent the input layer of the network. The tree structure representation was selected because it perfectly matches the topology of the multilayer morphological perceptron. Figure 6.2 shows an example of how a multilayer morphological perceptron may be encoded into a tree data structure.

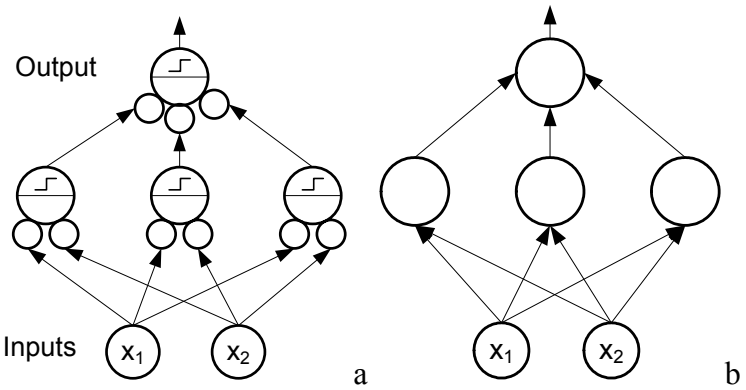


Figure 6.2 Tree based encoding. (a) Morphological neural network, (b) the corresponding representation in a tree structure.

Each node contains special registers that maintains a list of the inputs for the node, in addition to connection weights, synaptic values for each connection, and neuron

operation (maximum or minimum). Since the same transfer function is used for all the neurons in the neural network, it is not necessary to encode it into the genotype.

6.3.1.1 Selection

Rank selection is used to select the group of individual that will become parents for the next generation. A rank selection rank each individual and a fitness value is assigned according to the rank *R* it receives. The worst individual receives the value of 1, the second worst receives the value of 2 up to the best individual that receives the value of *N* (the number of individuals in the population). The probability of an individual *i* to be selected is shown in Equation 6.1.

$$P(i) = \frac{R}{\frac{N(N+1)}{2}} \tag{6.1}$$

6.3.1.2 Recombination

Offspring are produced when the genetic information from two parents is combined by genetic operators such as crossover. The crossover selects two parents randomly, as shown in Figure 6.3, then a node on each parent is randomly selected and all the information about the node is exchanged between the parents. The crossover point may be the root node (the output neuron), or a terminal node of the tree, but only nodes from the same level in both parents can be exchanged.

Arithmetic crossover is used to combine all the information between two nodes. Arithmetic crossover selects a weight at random and creates children from paternts weighted averages. It can be used to combine the floating point values of the connection

weights, as well as the integer values used to represent the pre-synaptic and post-synaptic response and the neuron operation.

This network representation using a tree structure format allows the algorithm to perform operations such as crossover replacing or switching whole neurons between parent networks.

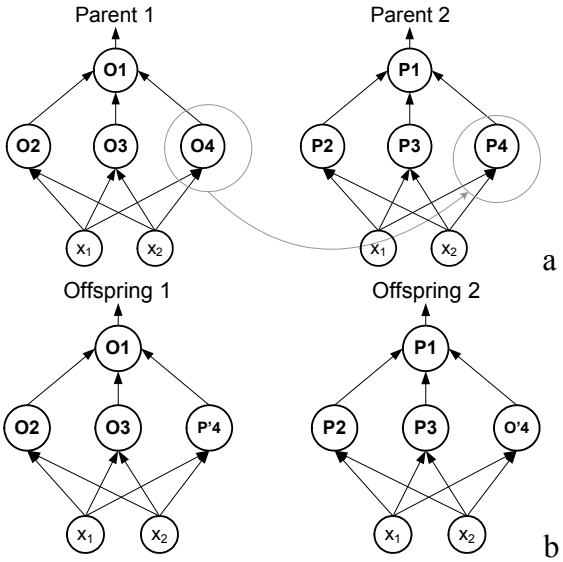


Figure 6.3 Crossover. (a) Initial parents. (b) New individuals formed using syntactically constrained crossover

6.3.1.3 Mutation

Due to the nonlinear nature of the fitness function, adaptive mutation is used in this implementation. The mutation is applied in two different ways. In the first case, a node is selected randomly, and then some of the information in the registers for that node is changed according to specific probabilities. The weights are adjusted by adding or subtracting random values in a predefined range.

On the other hand, as the fitness of the best organism reaches a threshold, the mutation probabilities of most of the MLMP parameters are reduced to minimal values (close to 0%), with the exception of the connection weights mutation probability, which

remains unchanged. In addition, the range in which the connection weights can change is reduced. This approach reduces the chance of an organism to mutate as the problem starts to converge. This approach is very important in this kind problem due the nonlinearity and discontinuity of the fitness function and the neural computational model.

6.3.1.4 Evaluation Function

The fitness function for this application has different components, such as the Mean Square Error of the classified patterns and other parameters related to the network architecture. The ideal scenario would be to get the same number of decision boundaries and number of neurons to be the same. This means optimum performances is obtained avoiding possible decision boundaries overlapping.

The fitness function for an organism that decodes into a single layer morphological perceptron is evaluated according to the performance in classification of the data set used during training, based on the Mean Square Error (MSE) shown in Equation 6.2.

$$MSE = \frac{1}{N} \cdot \left[\sum_{i=1}^{N} (y_i - d_i)^2 \right] \tag{6.2}$$

where N is the total number of patterns used during the training, y_i is the class where pattern x_i belongs and d_i is the class assigned by the neural network.

For a two layers morphological perceptron, the organism is evaluated according to its classification performance in addition to a penalty assigned to the number of redundant perceptrons in the network. A perceptron p_1, shown in Figure 6.4a, is considered to be redundant in relation to perceptron p_2, shown in Figure 6.4b, if the

region defined by the perceptron p_1 in the $\Re^n$ space is equal to the region produced by a perceptron in the second layer which receives the outputs from perceptrons p_1 and p_2 as inputs, as shown in Figure 6.4c. The resulting fitness function used to evaluate the individuals is shown in Equation 6.3.

$$f(o) = k_1 \cdot (1 - MSE) + k_2 \cdot p / C_2^t \tag{6.3}$$

where weighting factors $k_1 = \frac{1}{3}$, $k_2 = \frac{2}{3}$ are used, t is the number of neurons in the first layer, p is the number of non redundant perceptrons in the first layer, and

$$C_m^n = \binom{n}{m} = \frac{n!}{m!(m-n)!} \tag{6.4}$$

that represents total possible neuron-boundary combinations.

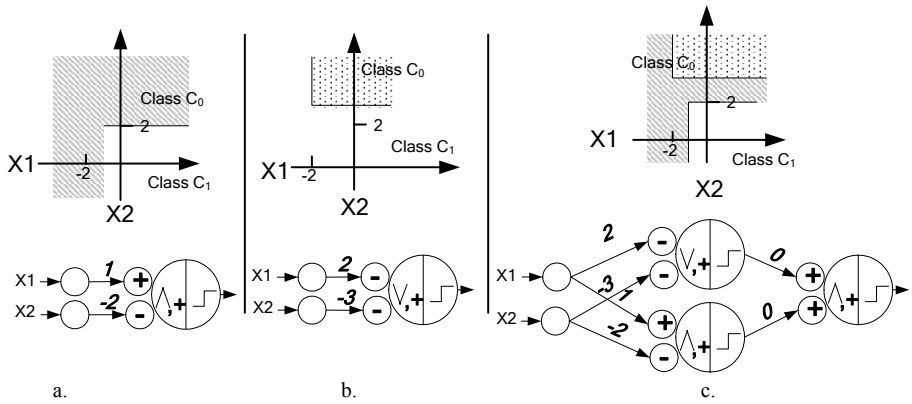


Figure 6.4 Redundant perceptrons. Region produced by two perceptrons (a) and (b), are combined into a perceptron in the second layer (c). The resulting region does not differ from region defined by perceptron (b).

6.4 INDIRECT ENCODING EVOLUTIONARY LEARNING ALGORITHM FOR THE MULTILAYER MORPHOLOGICAL PERCEPTRON

The proposed algorithm identifies the number of necessary neurons needed to perform the classification, connection weights, and defines architecture for multilayer morphological perceptrons used for pattern classification.

Figure 6.5 shows a $\Re^2$ space where patterns are grouped into clusters. The boundaries for these clusters can be approximated by succession of rectangular regions, where the corners of each of these regions can be seen as the decision boundaries of a morphological perceptron. The same concept can be extended to a higher domain space.

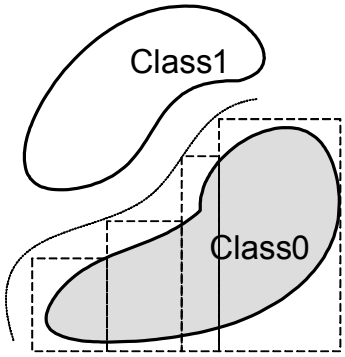


Figure 6.5 The region of the class C_0 is approximated by a succession of rectangles.

The problem is restated in such a way that the solution for the new problem results in a simpler representation. Instead of searching for the optimal architecture, set of connection weights, connection distribution, and neuron properties, the algorithm searches for those hypercubes that enclose all the patterns in class C_{t0}, without including patterns in class C_{t1}. Once a solution is found, the corners of the regions are used as decision boundaries, and a MLMP is built using the indirectly encoded information.

The genotype maintains the necessary information to rebuild a MLMP based on a set of rules. No information about the neuron inter-connections, connection weights, neuron parameters, and neuron distribution in layers is directly represented in the chromosome. All genotypes map to valid phenotypes. The genotype consists of an array of integer values. Each element represents a pattern from the class C_0. In addition, the genotype maintains information that identifies how the patterns are grouped.

It is important to describe the neural network architecture defined by the algorithm. Figure 6.6 shows an example of how the neural network architecture may look. The following rules are used to build MLMP that defines hypercubes enclosing patterns:

1. Resulting MLMP consists of three layers. Two layers MLMP are built when one hypercube enclose all the patterns.
2. All the neurons in the first layer use the maximum operator.
3. Two neurons in the first layer who that forward their output to a neuron in a second layer define a hypercube. The value of +1 is assigned to the all the pre-synaptic value for the neuron that defines the upper-right boundary and -1 is assigned to all the pre-synaptic values for the neuron that defines the lower-left boundary. Both neurons in the first layer use the maximum operator.
4. All the neurons in the second layer use the maximum operator, assign 0 to all the connection weights, and +1 to all the pre-synaptic values.
5. The last layer consists of one morphological neuron, which uses the minimum operator. The neuron receives variable number of inputs depending on the number of hypercubes defined by the genotype. All the connection weights are assigned to 0, and the pre-synaptic values are assigned to +1.
6. All the post-synaptic values for all the neurons in the MLMP are set to +1.

6.4.1 Encoding of the Genotype

The way the problem is encoded into the chromosome affects the performance of the algorithm. Differing from other approaches, in this algorithm nothing regarding to the connection weights or the relationship between the neurons, or the neural network

architecture is encoded into the chromosome. The chromosome keeps only enough information to identify each of the patterns from the class C_{t0}. Inside of the chromosome or genotype, there are groups or set of patterns, each of them represents clusters of patterns. Each set must contain at least one pattern, and no empty groups are allowed.

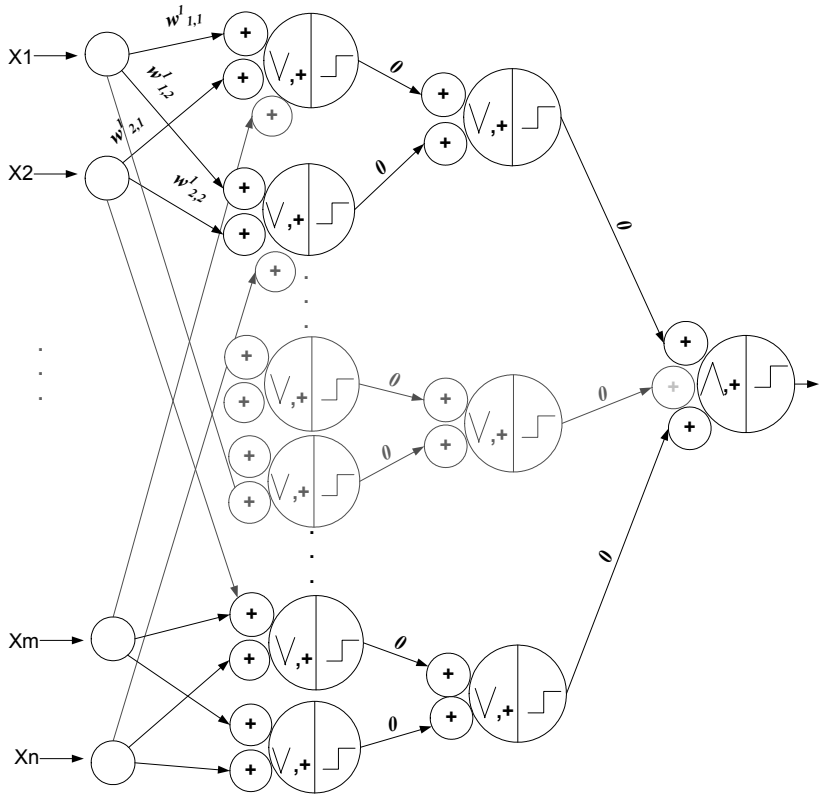


Figure 6.6 Shows how the morphological neural network architecture may look.

Example in Figure 6.7 shows class C_{t0} which contains 10 patterns in a $\Re^2$ space, identified as P# enclosed by a square, where # is an integer value used to identify each pattern. The patterns that do not belong to the class C_{t0} are represented by circles. It is important to mention that only those patterns that belong to class C_{t0} are encoded into the chromosome, using an integer value that corresponds to each pattern. When the initial population is generated, the patterns are randomly distributed into the chromosome in no particular order and the groups are randomly generated. Figure 6.8 shows how patterns

may be encoded in the chromosome, in addition to the graphical representation of the hypercubes that enclose the pattern groups represented in the chromosome

The elements from each group are used to define the limits of the hypercube which are used as the decision boundaries for the MLMP. A hypercube that includes all the elements for that particular group is defined for each group in the chromosome. The boundaries are used as the set of connections weights for neurons in the first layer.

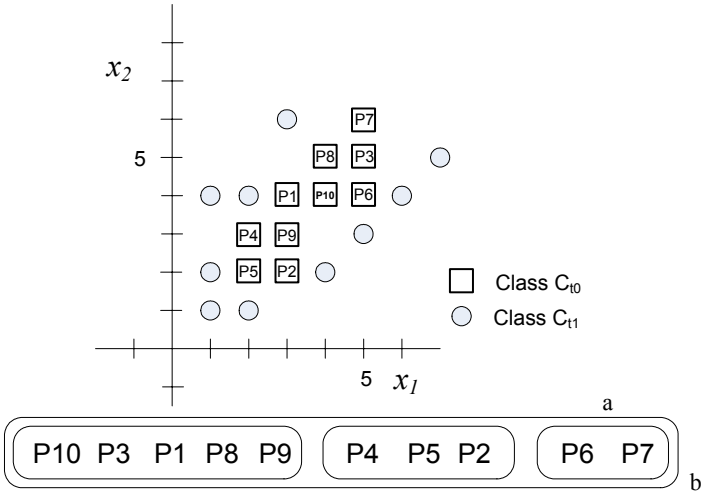


Figure 6.7 An example of how the patterns may be encoded into the chromosome of a randomly generated organism

6.4.1.1 Recombination

The crossover used in the implementation of the algorithm selects a set of *n* elements randomly distributed from different groups defined in the chromosome of the first parent. The selected elements are identified and their positions are exchanged in the first chromosome, according to order they appear in the second parent. The process is repeated again, but this time the exchange of elements is done in the second parent based on the order they appear in the first parent.

Figure 6.8a shows the chromosome of parent 1, with 10 patterns coded on it. Also, Figure 6.8b shows the hypercube for each group defined in the chromosome. Figure

6.9a shows the chromosome of the second parent as it will be used for the crossover process.

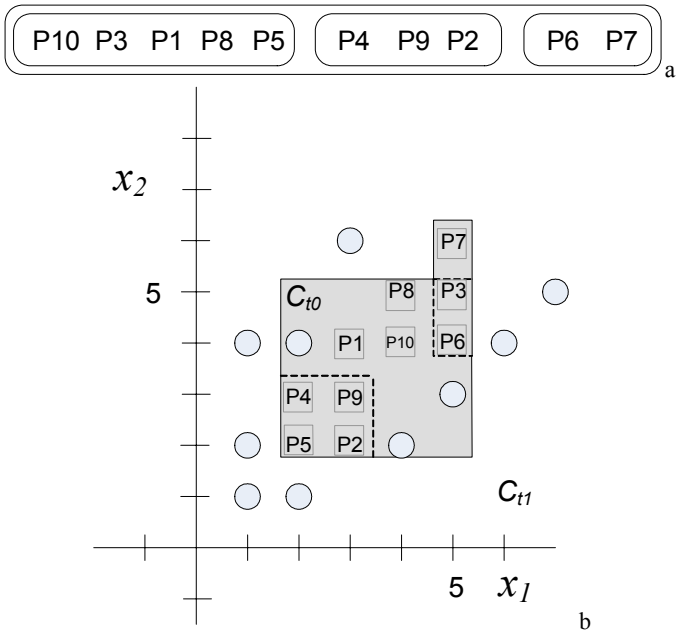


Figure 6.8 (a) Chromosome of first parent and (b) the corresponding set of hypercubes.

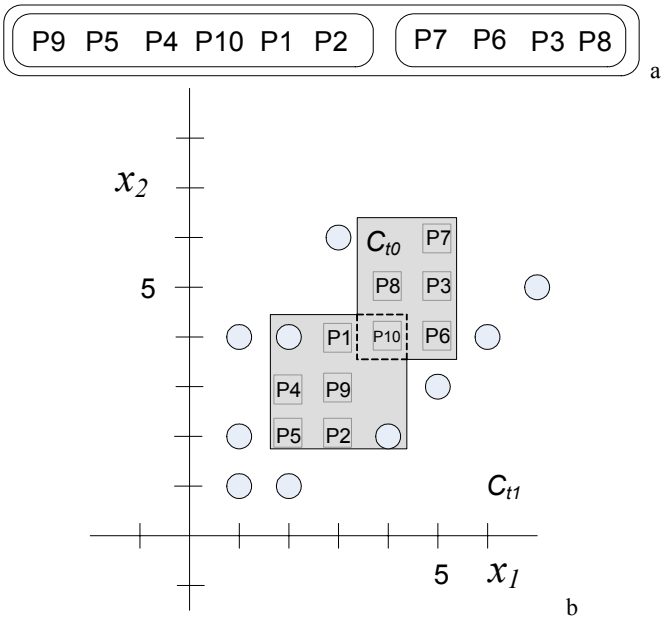


Figure 6.9 (a) Second parent used for the crossover and (b) the corresponding set of hypercubes.

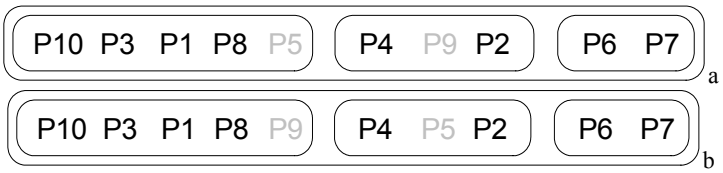


Figure 6.10 (a) First parent before the crossover and (b) the resulting offspring.

Assume P9 and P5 are the selected elements from the first parent, as shown in Figure 6.10a. Now these elements are identified in the second parent and the order is exchanged according the way they appear in the second parent. The final result after the elements are exchanged is shown in Figure 6.10b and the resulting set of hypercubes is shown in Figure 6.11. To obtain the second offspring the process is repeated, but this time the selection and exchange of the elements is done in the second parent according to the order they appear in the first parent.

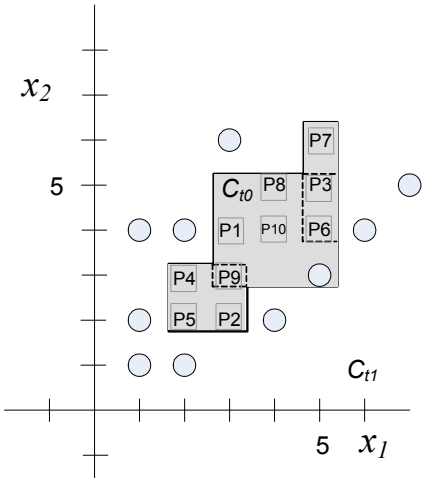


Figure 6.11 Hypercubes for the resulting offspring.

6.4.1.2 Mutation

Mutation operation in the proposed algorithm consists of two possible operations: fusion of two groups or division of a group into two new groups. In fusion of two groups, two groups are randomly selected, and then all the elements of these two groups are combined to create a new group. The other groups in the chromosome remain untouched.

Figure 6.12a shows an example of how the groups are defined before mutation and Figure 6.12b shows the resulting chromosome after mutation. In this type of mutation elements from different hypercubes are regrouped into one hypercube. This mutation operation promotes the combination of elements that may be grouped together into a single hypercube. Figure 6.13 shows the graphical effect of the mutation.

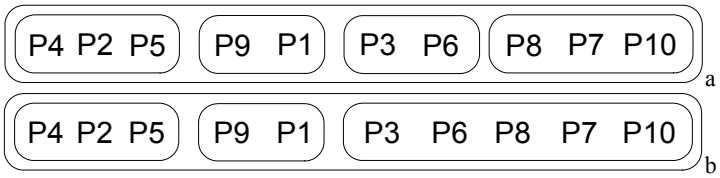


Figure 6.12 (a) Chromosome before mutation and (b) after mutation using group division.

Another possible mutation operation may be the redistribution of the elements in a group into two different groups. In this case, one group must be selected and all the elements of the group are distributed randomly between the two new groups. Figure 6.14a shows an example of a chromosome before mutation and Figure 6.14b shows the resulting chromosome after mutation, where elements of a group have been distributed into two different groups. This mutation operation promotes the separation of elements that should not be in the same group. Figure 6.15 shows mutation effect changes the definition of the hypercubes.

6.4.1.3 Decoding the Genotype

The groups encoded in the chromosome define hypercubes, whose boundaries are used as the connection weights for the neurons in the first layer of the MLMP. Each group in the chromosome defines a hypercube large enough to enclose all the patterns assigned to that particular group. Figure 6.16a shows an example of a chromosome and

Figure 6.16b shows the corresponding maximum and minimum values for the hypercube that enclose all the patterns defined in the first group in the chromosome.

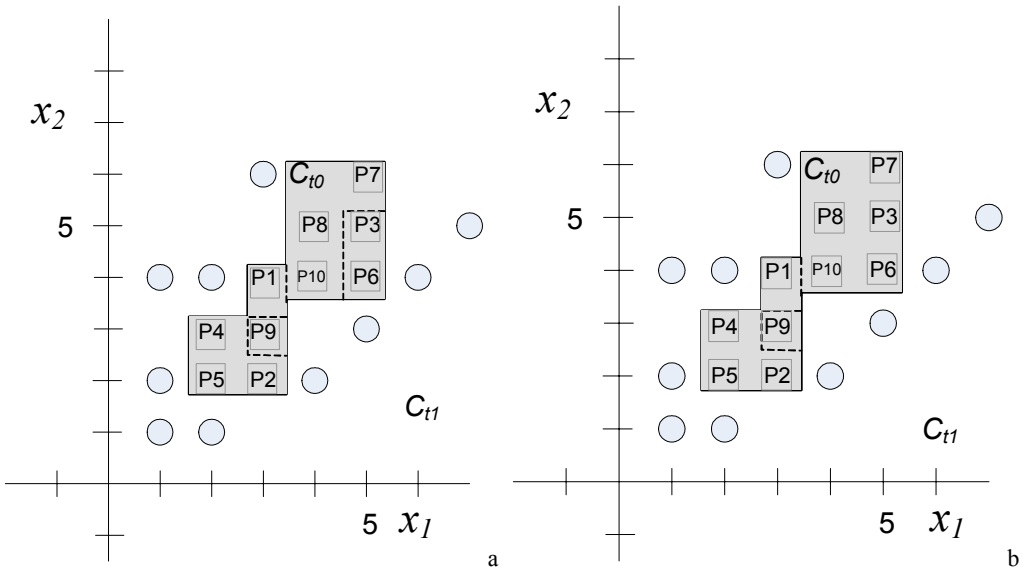


Figure 6.13 (a) The effect in the regions defined by the groups in the chromosome before mutation and (b) after mutation.

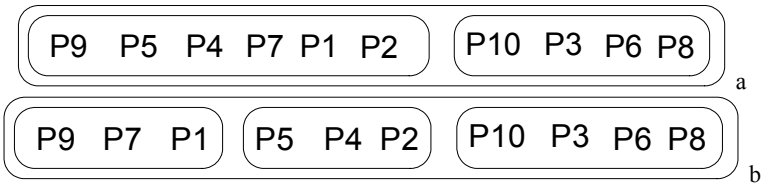


Figure 6.14 (a) Chromosome before mutation and (b) after mutation by combining two groups.

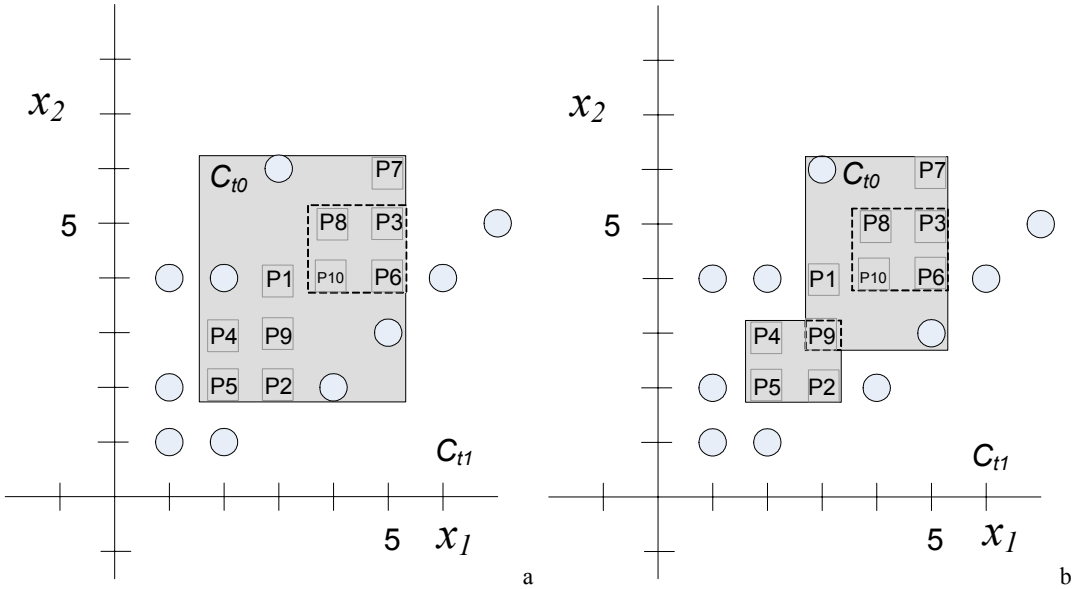


Figure 6.15 (a) Graphical effect of mutation in the regions defined by the groups in the chromosome before mutation and (b) after mutation.

The maximum and minimum values for each dimension of the hypercube are used as the connection weights for neurons in the first layer. The maximum values are assigned to the neuron with all the pre-synaptic values as +1, as shown in Figure 6.17a, and the minimum values are assigned to the neuron with all the pre-synaptic values as -1, as shown in Figure 6.17b. These two neurons are connected to another neuron in the second layer, as shown in Figure 6.18. As can be seen in Figure 6.18, three neurons are needed to define a single hypercube.

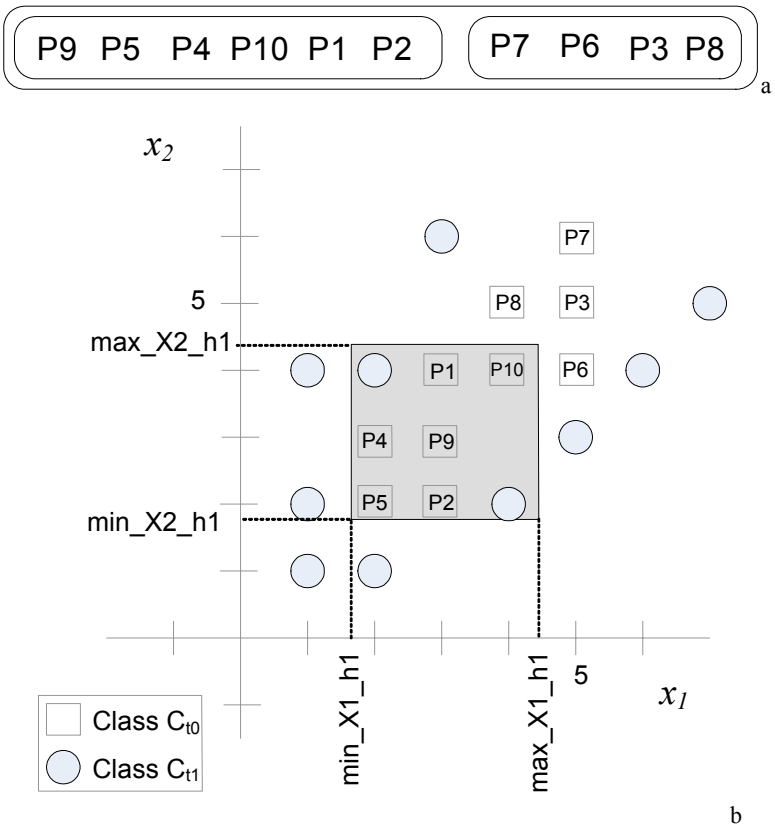


Figure 6.16 (a) An organism encoded into a chromosome and (b) the corresponding hypercube for the first group defined in the chromosome.

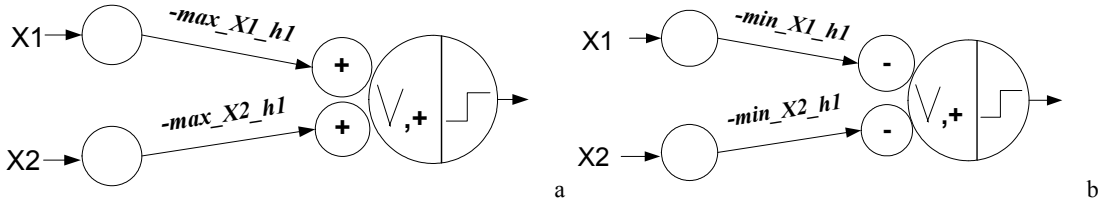


Figure 6.17 (a) Upper-right corner of the hypercube and (b) lower-left corner of the hypercube.

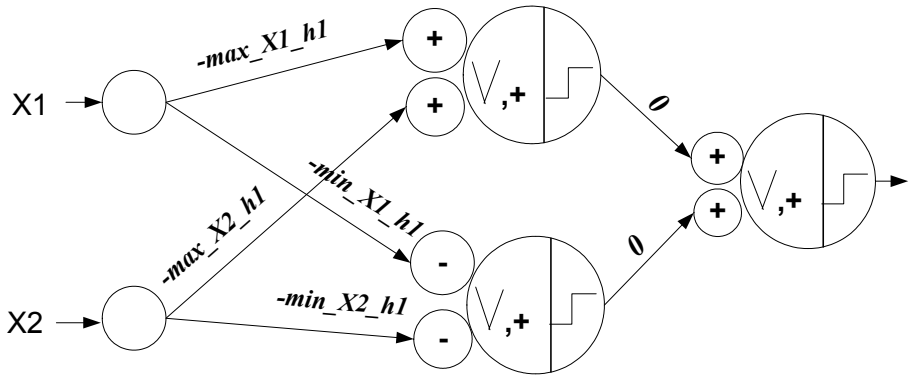


Figure 6.18 Neural network for a single hypercube.

The boundaries of the second hypercube are used to build another branch that will be added to the final neural network. Figure 6.19 shows the region defined by the second hypercube and Figure 6.20 shows the neural network that defines that particular hypercube. As can be seen in Figure 6.21, these two branches are combined as the inputs to a neuron in the third layer.

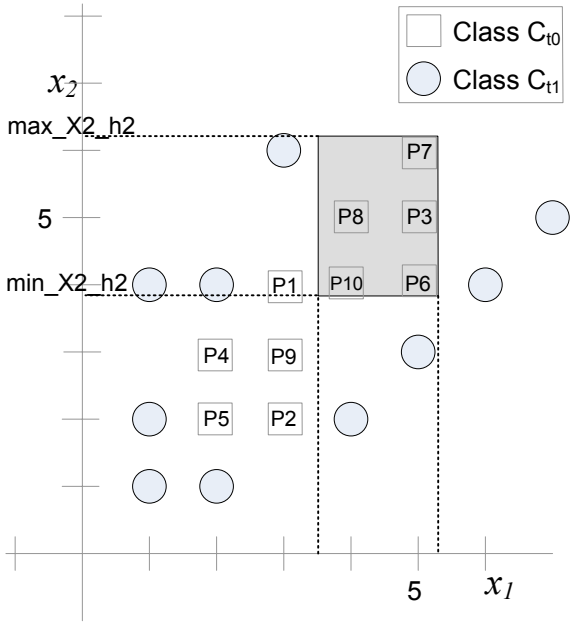


Figure 6.19 Region defined by the second group in the chromosome.

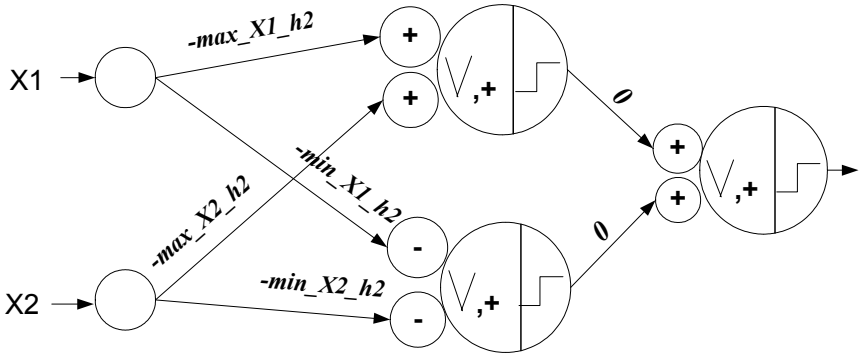


Figure 6.20 Resulting neural network for the second hypercube.

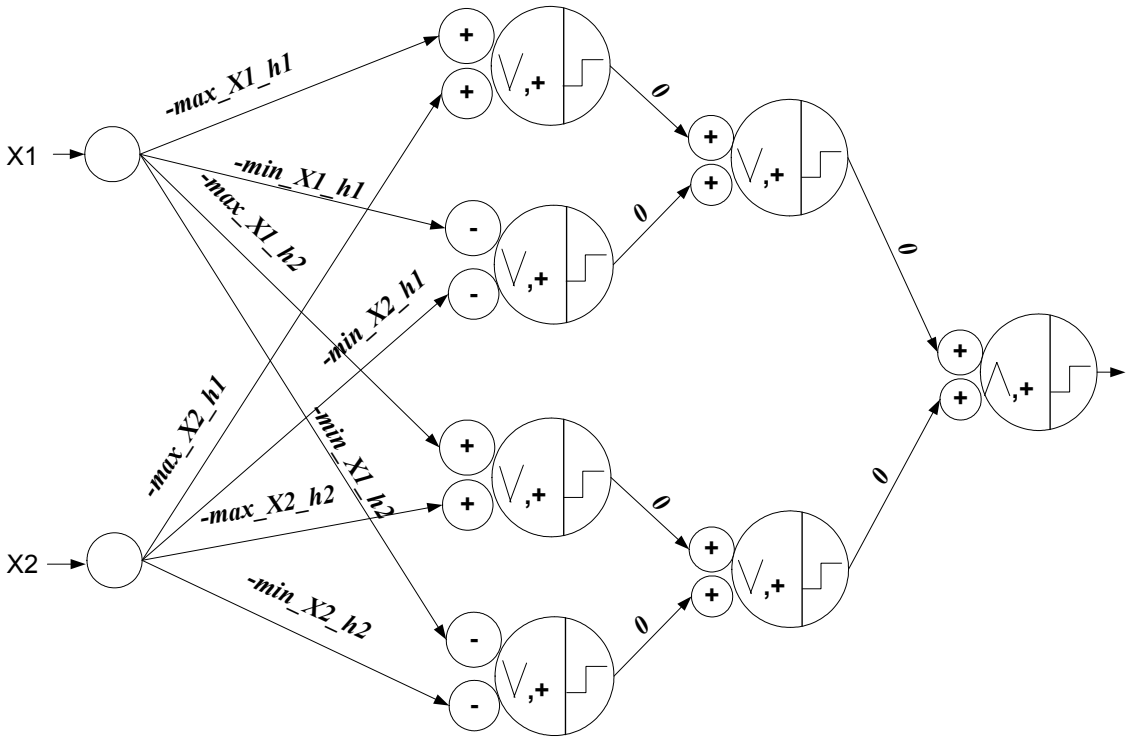


Figure 6.21 Resulting neural network for the chromosome defined in Figure 6.16a.

6.4.1.4 Evaluation Function

Each organism must be evaluated according to the features it has and only those organisms that have the desired features will survive and mate other organisms in order to transmit their own characteristics to the future generations.

One of the most important factors to take in consideration must be the number of misclassified patterns. Another important objective is the reduction of network complexity by using the minimum number of neurons needed to classify all the patterns correctly. This can be achieved by determining the minimum number of hypercubes necessary to enclose all test patterns. When a hypercube is added or removed from the chromosome, the architecture of the neural network changes. Changes are limited to the architectural constrains previously established. New neurons are added or removed from

the first and second layer of the network as a hypercube is added or removed, respectively.

The individual is evaluated according to the fitness function defined in Equation 6.4:

$$f(o) = 1/\left((k+1)^2 \cdot l\right) \tag{6.4}$$

where k is the number of patterns incorrectly classified. The value of l represents the number of neuron groups defined in the chromosome. Each neuron group consists of three neurons as shown in Figure 6.20. The fitness function minimizes the number of misclassified patterns as well as the number of hypercubes or neurons used to solve the problem.

6.4.1.5 Selection

A selection process is used to allow organisms who have higher fitness to transmit their features with higher probability than those who have a lower fitness. In order to consider that an organism is able to transmit their characteristics to future generations, the best 50% of the population that meets the requirements is selected. This accelerates the convergence reducing those members of the population that are not desirable. Wheel roulette is used to select the group of organism that will become parents for the next generation. The probability of an organism to be selected is equal to the fitness of the organism divided by the total fitness of all the organisms.

6.5 TRAINING OF THE MULTILAYER MORPHOLOGICAL PERCEPTRON USING CARTESIAN GENETIC PROGRAMMING

6.5.1 Encoding of the Genotype

The encoding used for this algorithm is based on Cartesian Genetic Programming. Some adjustments have been done to adapt CGP to train morphological networks. The most remarkable difference resides on the function used by the nodes. The proposed algorithm uses the computational model of the morphological neuron described in Equation 2.1 as the basic function for the nodes, contrary to the simple functions used on traditional CPG. The computational model operates over the inputs of the node as if they were the inputs of the neuron, which includes a set of connection weights, pre-synaptic response, and neuron operation.

The number of nodes may vary from layer to layer, as shown in Figure 6.22. Nodes are restricted to pass their outputs exclusively to nodes in the next layer; therefore a layer can not be skipped. The third layer contains only one neuron and it is used as the output node. All the nodes in the first layer are connected to all the inputs nodes, to preserve all the signals from the patterns. These connections are fixed in the chromosome, which means these values should not be changed by the reproduction operators. In traditional CGP, all the nodes has the same number of inputs, but in this approach this number may vary from layer to layer. Each node must have at least two inputs, except the last node. The last node is a special node that can accept variable number of inputs. It maintains a record of connected nodes and disconnected nodes in addition to the node operation. The inputs received by this node may vary from 0 to the total amount of nodes in the preceding layer.

Node operator in traditional Cartesian Genetic Programming can be used by all the nodes in the grid. Differing from traditional Cartesian Genetic Programming, each layer maintains a list of operators available for that particular layer. This means that nodes from the first layer are not allowed to use node operations defined for the first layer. As shown in Figure 6.22, $f_{n,m}$ denotes the function n defined for the layer m.

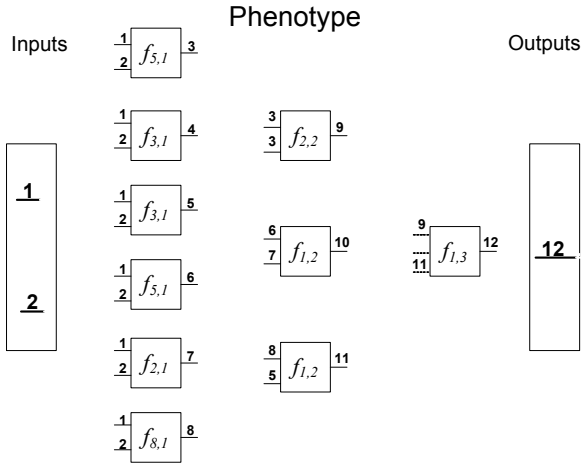


Figure 6.22 Graph of nodes used in the algorithm

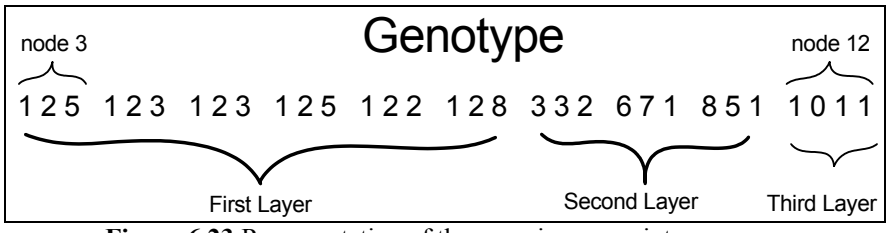


Figure 6.23 Representation of the organism as an integer array.

Each layer of nodes uses a particular set of operators (or functions) defined for each particular layer. The operations used by the algorithm maintain all the information needed to reconstruct the neuron, including neuron operation, pre-synaptic response, and connection weights. The set of operations defined for the first layer consist of all possible combinations of neuron operations (maximum and minimum), pre-synaptic response for each connection (+1, -1), and each test pattern in the class C_0, used for the training. The negative value of each dimension in the pattern is used as the weight for the neuron's

connections. In the second layer, only two operations are used. In both operations all the pre-synaptic values are set to be +1 and all the connection weights are set to be 0. The difference in each neuron consist in the operation, one of the operations uses the maximum operator while the other one uses the minimum operator. In the last layer, there is only one operation available for the node. The operation consists of minimum operator, all the pre-synaptic values are set to be +1, and connection weights are set to be 0.

The chromosome consists of an array of integer, which describes the information contained in a node, as shown in Figure 6.23. The node consists of a set of inputs and an operation. All the inputs and the operations are represented by integer values. The last node consists of an input binary array of integers and the node function. Each binary entry identifies which node from the second layer connects to the last node.

In order to reconstruct the morphological perceptron, the chromosome is analyzed starting from the node in the right section of the chromosome. The last node maintains a binary array that identifies which nodes from the previous layer forward their outputs as inputs to the last neuron. The other neurons are added to the network in the same way as in traditional CGP.

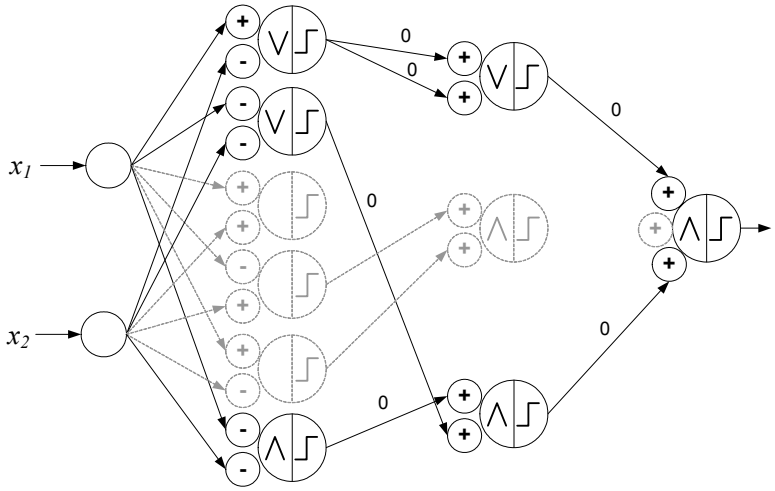


Figure 6.24 Resulting Morphological Neural Network after decoding of the chromosome with unexpressed neurons.

6.5.2 Genetic Operators

The recombination of the genetic information is done using multipoint crossover. The crossover consists of the selection of several nodes from one parent, each node contains the inputs and function associated to that specific node. The first offspring, is obtained exchanging the information contained on each node in the first parent with the information from a node at the same position from the second parent.

Multipoint mutation is used to mutate the chromosome but a specialized mutation rate is used in different regions of the chromosome to promote more changes on particular areas. Multipoint mutation consists of selection and modification of several points in the chromosome. Each point is modified according to the functional constrains or constrains imposed by the levels. The chromosome is divided into three regions: node functions, inputs for the nodes in the second layer, and inputs for the node in the third layer. Three different mutation rates are assigned to each region, to produce independent changes on each region. The mutation rate used for the node function is larger than the mutation rate used on the other regions of the chromosome, promoting faster changes on

connection weights, pre-synaptic response, and neuron operation over network architecture. All these parameters are part of the function encoded for these nodes.

6.5.3 Evaluation Function

Only those individuals that have the desired features will survive and mate other individuals in order to transmit their own characteristics to future generations. An individual is evaluated according to amount of the patterns classified correctly during the training process. In addition to the classification accuracy of the neural network, the inputs received by the last neuron is analyzed. It may be possible that these neurons assigns all the patterns in the search space to one particular class, either class C_0 or class C_1. If this occurs, the organism is penalized by the number of neuron in the second layer that produces these results. The final fitness function is shown in Equation 6.5.

$$f(o) = \left(\frac{c}{C}\right)\left(\frac{m}{M}\right)\left(\frac{n}{N}\right) \tag{6.5}$$

where c is the number of correctly classified patterns, C is the total number of patterns used in the training process, m is the number of neurons without penalty, M is the total number of available nodes in the second layer, n is the number of disconnected nodes in the last layer, and N is the maximum number of nodes that can be connected to neuron in the last layer.

It may be possible that the node in the last layer does not receive any input from the previous layer, if this is the case the individual is discarded by assigning a fitness value of 0. If only one node connects to the last neuron, then this node is used as the output node instead of the node in the last layer, resulting in a two layer morphological perceptron.

6.6 EXAMPLE

Lets consider the XOR problem (Sussner 1998) to illustrate an example of how to use the training algorithm. The XOR is a binary operator on $\{0,1\}^2$ such that for all $(a,b) \in \{0,1\}^2$:

$$a \text{ XOR } b = \begin{cases} 0 \text{ if } a = b \\ 1 \text{ else} \end{cases} \tag{6.9}$$

Two different classes are defined as C_0= {(0,0), (1,1)}, and C_1= {(1,0), (0,1)} as shown in Figure 6.25.

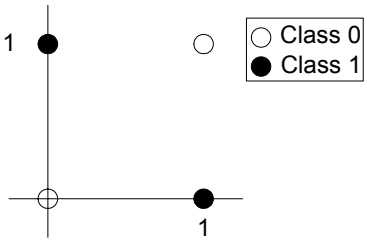


Figure 6.25 Distribution of patterns for the XOR problem.

The function for the nodes in the first layer are constructed from all possible combinations of the patterns defined in class C_0, and all the possible combinations of the values for the parameters in Equation 6.6:

$$f\left(\overset{n}{\underset{i=1}{O}} r_i \left(x_i + w_i \right) \right) \tag{6.6}$$

where O denotes the neuron operator maximum (or minimum), and the other values are the traditional neuron parameters.

Only two functions are defined for the hidden layers, one of these functions uses the maximum operator and the other one uses the minimum operator. The pre-synaptic values are set to +1 and all the connection weights are set to 0. Usually the function in the last layer consists of the minimum operator, because the patterns enclosed by the decision

boundary belong to the class C_0. In the same way, the pre-synaptic values for this neuron are set to 0 and the connection weights are set to 0. Since this node has variable number of inputs, it is important to remember that these values are assigned to all the active connections.

An initial population is randomly generated to start the algorithm. The chromosome consists of the nodes defined in Figure 6.26, appended one after the other in a sequence. Each node maintains a list of values for inputs and function. Inputs and functions for each node are defined as shown in Table 6.1.

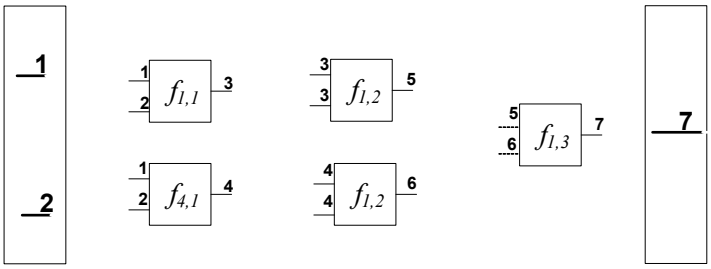


Figure 6.26 Graph of nodes used to represent the organism.

Table 6.1a shows the set of functions available for the nodes in the first layer. Table 6.1b contains the set of functions available for the nodes in the second layer. These values vary in a specific range, determined by the position of the node in the graph, and chromosome representation constrains defined in section *6.5.1 Representation*. Table 6.2 shows how the graph in Figure 6.26 may be encoded in the chromosome, in addition to the lower and upper bounds for each entry in the chromosome. A possible neural network produced by the learning algorithm is shown in Figure 6.27, and its corresponding decision boundary is shown in Figure 6.28.

	O	r1	r2	w1	w2
$f_{1,1}$	∧	+1	+1	0	0
$f_{2,1}$	∨	+1	-1	0	0
$f_{3,1}$	∧	-1	+1	0	0
$f_{4,1}$	∨	-1	-1	0	0
$f_{5,1}$	∧	+1	+1	0	0
$f_{6,1}$	∨	+1	-1	0	0
$f_{7,1}$	∧	-1	+1	0	0
$f_{8,1}$	∨	-1	-1	0	0
$f_{9,1}$	∧	+1	+1	1	1
$f_{10,1}$	∨	+1	-1	1	1
$f_{11,1}$	∧	-1	+1	1	1
$f_{12,1}$	∨	-1	-1	1	1
$f_{13,1}$	∧	+1	+1	1	1
$f_{14,1}$	∨	+1	-1	1	1
$f_{15,1}$	∧	-1	+1	1	1
$f_{16,1}$	∨	-1	-1	1	1

a

	O	r1	r2	w1	w2
$f_{1,2}$	∧	+1	+1	0	0
$f_{2,2}$	∨	+1	+1	0	0

b

Table 6.1 (a) Set of functions available for nodes in the first layer, and (b) functions available for nodes in the second layer

genotype	1	2	1	1	2	9	3	3	2	4	3	1	1	1	1
lower bound	1	2	1	1	2	1	3	3	1	3	3	1	0	0	1
upper bound	1	2	16	1	2	16	4	4	2	4	4	2	1	1	1

Table 6.2 Example of how the organism is encoded, and the lower and upper bounds for each entry in the chromosome.

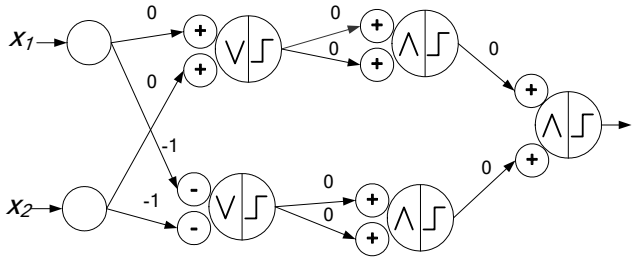


Figure 6.27 Resulting neural network defined for the XOR problem using Cartesian Genetic Programming method.

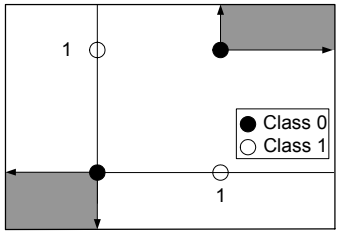


Figure 6.28 Corresponding decision boundary defined by the neural network shown in Figure 6.27

CHAPTER 7

MATLAB TOOLBOX FOR MORPHOLOGICAL PERCEPTRON

7.1 INTRODUCTION

This chapter describes a set of methods implemented on Matlab as a toolbox to create, process, and train multilayer morphological perceptrons. The training methods were implemented using Matlab 6.The chapter describes the configuration parameters used by each training algorithm in addition to a sample code in Matlab.

7.2 TOOLBOX

7.2.1 Common Configuration Parameters

Each training method requires a set of specific configuration parameters. The configuration parameters control the evolutionary process, including initial population size, termination conditions, genetic operators, and evaluation function. These parameters have been added to provide a flexible control over the evolutionary process. Different genetic operators, as well as evaluation functions may be used, producing different results. The source code for all the toolbox can be found on Appendix A does not has to be changed. Table 7.1 presents the common configuration parameters used by all the training methods.

Parameter	Type	Description	Example
param.evalFn	m-file	Specifies the name of the evaluation function used.	['CGPEval3']
param.evalParams	vector of double	Specifies any parameter passed to the evaluation function	[]
param.mutationFn	m-file	Specifies the name of the mutation function.	['CGPMultiPointMutation2']
param.mutationParams	vector of double	Specifies any arguments need by the mutation function.	[0.08 0.08]
param.popSize	integer	Size of the population used during the evolution	20
param.selectFn	m-file	Specifies the name of the selection function, used to select the survivals from a generation to the next one.	['roulette2']
param.selectParams	vector of double	Specifies any parameter passed to the selection function.	[0.33]
param.termParams	integer	Specifies the termination criteria: [max. number of generations, final fitness]	[8000,1.0]
param.xOverFn	m-file	Specifies the name of the crossover function.	['CGPMultipointXover'];
param.xOverParams	vector of double	Specifies any necessary parameter passed to the crossover function.	[0.95 0.80]

Table 7.1 Configuration parameters used by all the training methods

7.2.2 Direct Encoding Toolbox

7.2.2.1 Configuration Parameters

In addition to the parameters presented on Table 7.1, the *Direct Encoding method* requires configuration parameters described on Table 7.2. *Direct Encoding method* requires the number of neurons to be specified prior to the training of the neural network,

each position in the vector specified by *param.layer* represents the number of neuron for each layer.

Parameter	Type	Description	Example
param.layers	vector of integers	Specifies the number of neurons for each layer.	[2 1]
param.opts	vector of double		[1e-6 1 1]

Table 7.2 Configuration parameters used by Direct Encoding Method

7.2.2.2 Training Method

[*net, traceInfo*] = DirectTrainMNN(*patterns*, *classes*, *bounds*, *targets*, *config*);

Description: Trains a multilayer morphological perceptron using the *Direct Encoding method*. The method receives as arguments the patterns used during the training process. *Patterns* are passed to the method as an *M*x*N* matrix, which contains *N* patterns of *M* dimensions. All the patterns from all the classes are appended one after another in the argument *patterns*, starting by patterns from class C_0, then patterns from class C_1 are appended, and finally patterns from class C_T, where *T* is the total number of classes to be trained. The parameter *classes* define a column vector containing the number of patterns defined for each class in the *patterns* matrix. *Bounds* is a 2xM matrix in which each row vector represents the lower and upper bounds for each dimension. The variable *targets* contains a *P*x*Q* matrix of binary elements, where each row represents the binary vector associated to a particular class. The parameter *config* is a data structure that contains the configuration parameters shown in Table 7.1 and Table 7.2. The method returns an object *net* that represents a MNN.

Parameter	Type	Description	Example
patterns	matrix of integers	Specifies the all the patterns used during the training process.	class0 = [0 0; 1 1]; class1 = [0 1; 1 0]; patterns = [class0; class1];
classes	vector of integers	Specifies the amount of patterns defined on each class	[2; 2]
targets	matrix of integers	Each row represents the binary vector associated to a particular class	[0 ; 1]
params	struct	Configuration parameters for the algorithm.	as shown in Table 7.1 and Table 7.2
net	MNN	MNN trained for the patterns	
traceInfo	Matrix of double	Performance of the evolution. Four column matrix representing: generation number, fitness of the best individual, average fitness of the generation, and standard deviation	

Table 7.3 Parameters passed to the Direct Encoding training method.

7.2.2.3 Sample Code

The code shown in Figure 7.1 defines patterns for two classes C_0={(0,0) ,(1,1)}, and C_1={(0,1), (1,0)}, in a two-dimensional search space for the training algorithm and returns a morphological perceptron which is able to classify these patterns.

```
% param is previously defined
class0 = [0 0; 1 1];
class1 = [0 1; 1 0];
patterns = [class0;class1];
targets = [0 ; 1];
classes = [size(class0,1); size(class1,1)];

% Compute the bounds for each dimension
minVals = min(patterns);
bound = [(max(patterns)-minVals); minVals];

% Expand the boundaries by %25
bound = bound +[ boundaries (1,:)*.125; - bound (1,:)*0.125]
[net,traceInfo] = DirectTrainMNN(patterns, classes, bound, targets, config);
```

Figure 7.1 Example code of how Direct Encoding Method can be used to train MNN

7.2.3 Indirect Encoding Toolbox

7.2.3.1 Configuration Parameters

Configuration parameters used by the Indirect Encoding method are the configurations parameters shown in Table 7.1. No additional configuration parameters are needed.

7.2.3.2 Training Function

[*net, traceInfo*] = IndirectTrainMNN(*patterns, classes, targets, params*)

Description: Trains a multilayer morphological perceptron using the *Indirect Encoding method*. The function receives as arguments the patterns used during the training process. *Patterns* are passed to the function as an *M*x*N* matrix, which contains *N* patterns of *M* dimensions. All patterns from all the classes are appended one after another in the argument *patterns*, starting by patterns from class C_0, then patterns from class C_1 are appended, finally patterns from class C_T, where *T* is the total number of classes to be trained. The parameter *classes* define a column vector containing the number of patterns defined for each class in the *patterns* matrix. The variable *targets* contains a *P*x*Q* matrix of binary elements, where each row represents the binary vector associated to a particular class. The parameter *config* is a data structure that contains the configuration parameters used for the training algorithm. The method returns an object *net* that represents a MNN.

Parameter	Type	Description	Example
patterns	Matrix of integers	Specifies the all the patterns used during the training process.	class0 = [0 1; 1 1]; class1 = [1 0; 0 0]; patterns = [class0; class1];
class_distribution	Vector of integers	Specifies the amount of patterns defined on each class	[2; 2]
targets	Matrix of integers	Each row represents the binary vector associated to a particular class	[0 ; 1]
params		Configuration parameters for the algorithm.	As shown in Table 7.1
net	MNN	MNN trained for the patterns	
traceInfo	Matrix of double	Performance of the evolution. Four column matrix representing: generation number, fitness of the best individual, average fitness of the generation, and standard deviation	

Table 7.4 Parameters passed to the CGP training method

7.2.3.3 Sample Code

The code shown in Figure 7.2 defines patterns for two classes C_0={(0,0) ,(1,1)}, and C_1={(0,1), (1,0)}, is a 2-dimensinal search space for the training algorithm and returns a morphological perceptron which is able to classify these patterns.

```
% params  is previously defined
class0 = [0 1; 1 1];
class1 = [1 0; 0 0];
patterns = [class0;class1];
targets = [0 ; 1];
classes = [size(class0,1); size(class1,1)];

[net,traceInfo]=IndirectTrainMNN(patterns,classes, targets, params);
```

Figure 7.2 Example code of how Indirect Encoding Method can be used to train MNN

7.2.4 Cartesian Genetic Programming Toolbox

7.2.4.1 Configuration Parameters

Configuration parameters used by the training algorithms include the configuration parameters shown in Table 7.1 in addition to the configuration parameters shown in Table 7.5. Configuration parameters from Table 7.5 define some network properties such as the distribution of nodes, and number of inputs received by the neurons for each layer.

Configuration parameter	Type	Description	Example
param.connections	vector of integers	Specifies the number of connections used by the nodes on each layer.	[20 20 1]
param.layers	vector of integers	Specifies the maximum number of nodes defined for each layer.	[4 2 20]

Table 7.5 Additional configuration parameters used by Cartesian Genetic Programming.

7.2.4.2 Training Function

[*net*,*traceInfo*] = *CGPTrainMNN*(*patterns, classes, targets, params*)

Description: Trains a multilayer morphological perceptron based on Cartesian genetic programming. The function receives as arguments the patterns used during the training process. *Patterns* are passed to the function as an *M*x*N* matrix, which contains *N* patterns of *M* dimensions. All the patterns from all the classes are appended one after the other in the argument *patterns*, starting by patterns from class C_0, then patterns from class C_1 are appended, finally patterns from class C_T, where *T* is the total number of classes to be trained. The parameter *classes* define a column vector containing the number of patterns defined for each class in the *patterns* matrix. The variable *targets* contains a *P*x*Q* matrix of binary elements, where each row represents the binary vector associated to a particular

class. The parameter *config* is a data structure that contains the configuration parameters used for the training algorithm. The method returns an object *net* that represents a MNN, and a matrix *traceInfo* which consists of three columns. The first column identify the generation number, the second column corresponds to the fitness value assigned to the best organism for the corresponding generation, the third column corresponds to the average value for the fitness of the population, the value forth column corresponds to the standard deviation.

Parameter	Type	Description	Example

patterns	Matrix of integers	Specifies the all the patterns used during the training process.	class0 = [0 1; 1 1];
			class1 = [1 0; 0 0];
			patterns = [class0;class1];

classes	Vector of integers	Specifies the amount of patterns defined on each class	[2; 2]
targets	Matrix of integers	Each row represents the binary vector associated to a particular class	[0 ; 1]
params		Configuration parameters for the algorithm.	As shown in Table 7.1 and Table 7.5
net	MNN	MNN trained for the patterns	
traceInfo	Matrix of double	Performance of the evolution. Four column matrix representing: generation number, fitness of the best individual, average fitness of the generation, and standard deviation	

Table 7.6 Parameters passed to the CGP training method

7.2.4.3 Sample Code

The code shown in Figure 7.3 defines patterns for two classes C_0={(0,0) ,(1,1)}, and C_1={(0,1), (1,0)}, the search space for the training algorithm, and returns a morphological perceptron which is able to classify the patterns.

```
% params  is previously defined
class0 = [0 1; 1 1];
class1 = [1 0; 0 0];
patterns = [class0;class1];
targets = [0 ; 1];
classes = [size(class0,1); size(class1,1)];

[net,traceInfo] = CGPTrainMNN(patterns, classes, targets, params);
```

Figure 7.3 Example code of how Indirect Encoding Method may be used to train MNN

7.3 COMMON TOOLS

This section describes a set of common tools used by all the training methods to manipulate and control morphological neural networks.

7.3.1 Pattern Classification

[*class*] = evalMorphologicalNet(*net*, *pattern*s)

Description: Classify the patterns defined by the argument *patterns* given a vector of MLMP denoted by the argument *net*. Each entry in the vector *net* represents a Multilayer Morphological Perceptron used to construct the classification vector. Multiple patterns may be classified simultaneously using a single function call as shown in Figure 7.4. In Figure 7.4, three 4 dimensional patterns are assigned to class $\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$. Each row from the *class* matrix denote the corresponding classification for each pattern defined by each row from the *patterns* argument.

```
class1 =

   5.1000   3.5000   1.4000   0.2000
   4.9000   3.0000   1.4000   0.2000
   4.7000   3.2000   1.3000   0.2000

>> evalMorphologicalNet(net,class1)

ans =

   0   1   1
   0   1   1
   0   1   1
```

Figure 7.4 How to use Multilayer Morphological Perceptrons to classify multiple patterns.

7.3.2 Plotting the Network

***plotNetwork2*(*net*)**

Description: Display a graphical representation of the perceptrons denoted by the argument *net* in a 2-dimensinal space, as shown in Figure 7.5.

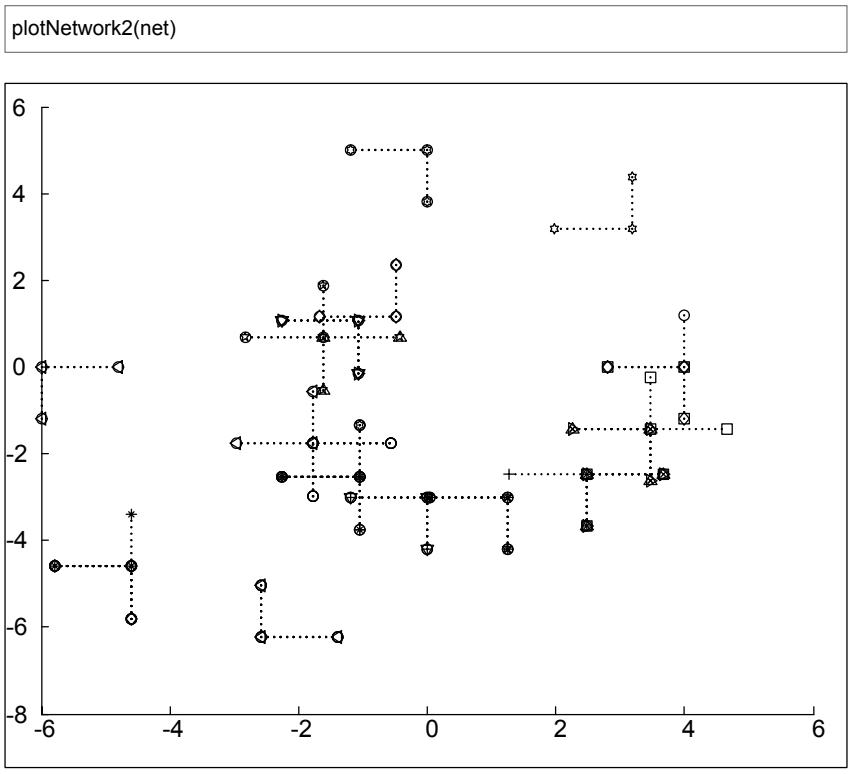


Figure 7.5 Graphical representation of Multilayer Morphological Perceptrons.Tthe morphological perceptrons are represented by two intersecting perperdicular dotted lines.

7.4 ANALYZING PROGRESS OF THE LEARNING PROCESS

After the training of the neural network has been completed, information about the progress of the evolution is returned, in addition to an object which represents a multilayer morphological perceptron. The *traceInfo* is a matrix that contains 4 columns: generation, fitness of the best individual, mean fitness of the population for each generation, and standard deviation of the fitness for each generation. The *traceInfo* provides useful information about the evolutionary progress of the population.The information can be used to evaluate genetic operators, such as crossover, mutation, selection, and in order to select the best parameters for each genetic operator for a data set. Figure 7.6 shows the progress of the fitness as the number of generations increase.

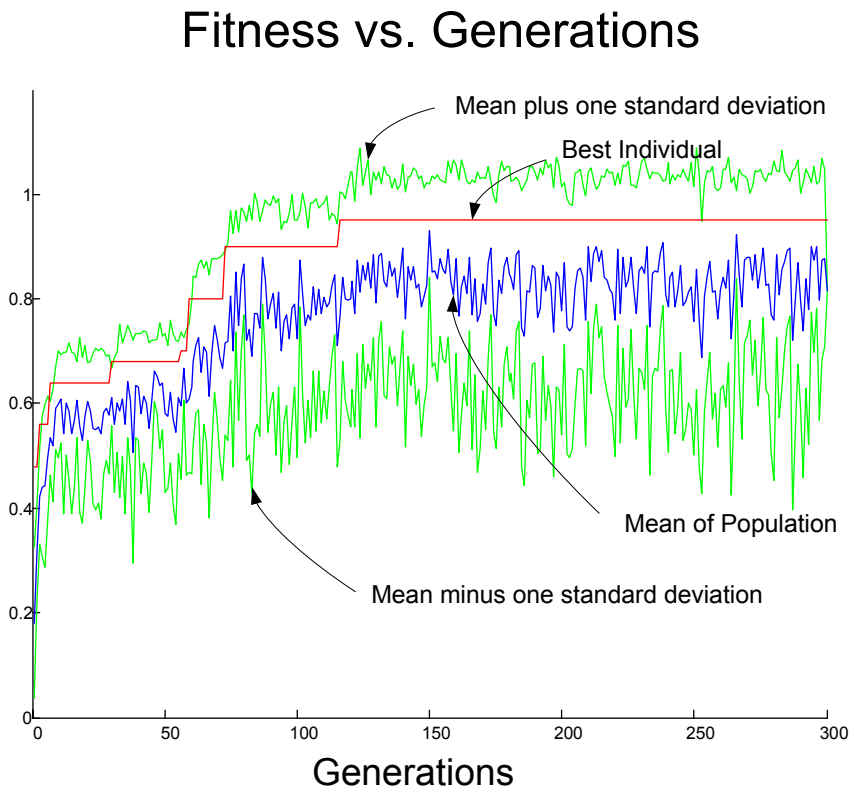


Figure 7.6 Evolutionary progress of the population for Carterian Genetic Programming using the Sussner Data set

The graph shown in Figure 7.6 was produced by the Cartesian Genetic Programming method using Sussner Data set. It is important to point out that the fitness function used in the evolution tries to maximize the performance of the individuals, this means that an individual with higher fitness value is considered the best candidate solution for the problem. Since the fitness function maximizes the performance, the resulting graph of the mean fitness value of the population should increase or remain horizontal. The graph that describes the optimum behaviour of the evolution should look like a logarithmic curve, with an asymptote at 1.0. The graph of the mean fitness may oscilate, due to the diversity introduced by new mutated members, but in general the graph should increase all the time, otherwise there may be something wrong with the evolutionary operators. Possible explanations to this behavior may be high mutation rates, or inapropiate fitness function. High mutation rates may introduce too much diversity in the population, incrementing the oscilation range for the mean fitness value and prolonging the time required to convege into the optimum value. Another explanation to this behaviour may be that the fitness function is not able to differenciate correctly the performance of two individuals.

Inspecting the standard deviation of the population fitness value is another way to determine if the mutation rates are too high. As the mutation rate increases the standard deviation of the fitness increases. The mean fitness value plus one standard deviation and the mean fitness values minus one standard deviation are shown in Figure 7.6. The effects produces by different mutation rates and crossover rates on the progress of the fitness value in terms of generations are shown in Figure 7.7. For example, Figure 7.7g shows how the fitness of the population varies over generations when the genetic operators are

using a mutation rate of 12% and a crossover rate of 80%. Figure 7.7 shows some of the most relevant graphs obtained by changing the crossover and mutation rates.

It is important to observe that when the mutation rate is low, high crossover rate affects the number of generations needed to reach a fitness value of 0.8. When a mutation rate is fixed to 5%, using a crossover rate of 95%, the fitness reach to a value of 0.8 before 50 generations. Using a crossover rate of 80%, it takes more than 100 generations to reach a value of 0.8. Using a crossover rate of 20%, it takes almost 200 generations to reach the value of 0.8. This effect is almost unnoticeable when the mutation rate is high, as for example 24%.

Increasing the mutation rate also affects the time needed to reach a particular fitness value. As the mutation rate increases, the number of generations needed to reach a fitness value of 0.8 decrese. On the opposite side, a high mutation rate increases the time needed to reach the optimum fitness value of 1.0, therefore an optimum evolutionary curve may be produced by high crossover rates and low mutation rates as shown in Figure 7.7b.

Another feature that can be observed from Figure 7.7, is that it may be possible to reach the optimum fitness value using the mutation operator exclusively.

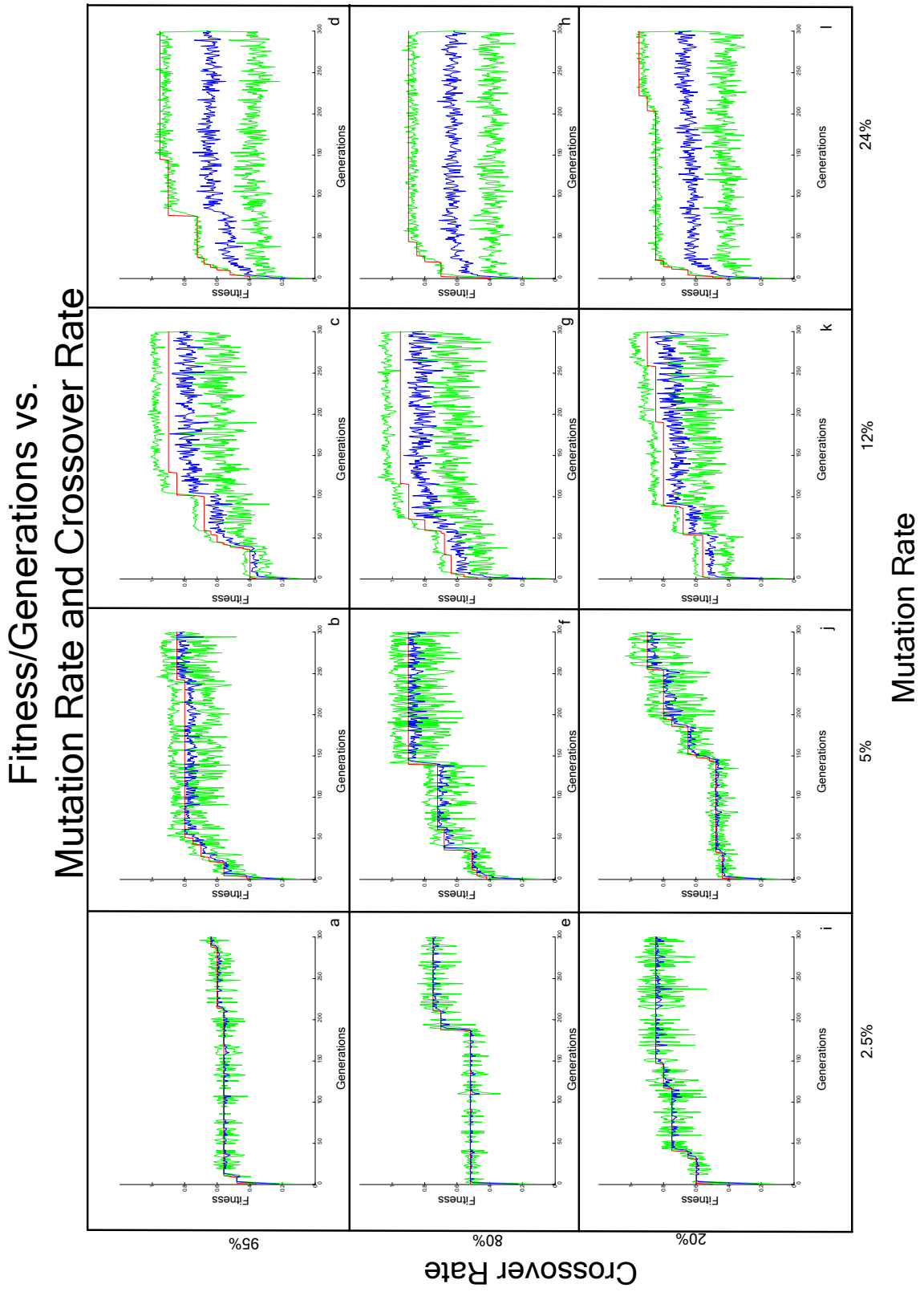


Figure 7.7 Effects produced on the fitness of a population and the number of generations by different crossover and mutation rates.

CHAPTER 8

PERFORMANCE ANALYSIS

8.1 INTRODUCTION

This chapter measures the performance of the evolutionary learning algorithms used for the Multilayer Morphological Perceptrons in terms of *memorization* and *generalization* of the trained neural network for different data sets presented by Sussner and Ritter, and other commonly used data sets such as the Iris Fisher Data and a Spiral data set.

8.2 DATA SETS

8.2.1 Sussner Data Set

The Sussner Data set, shown in Figure 8.1, consists of 20 patterns equally divided among two classes, used as a benchmark for comparison of performance between the evolutionary learning algorithms presented in this thesis and the learning algorithms proposed by Sussner.

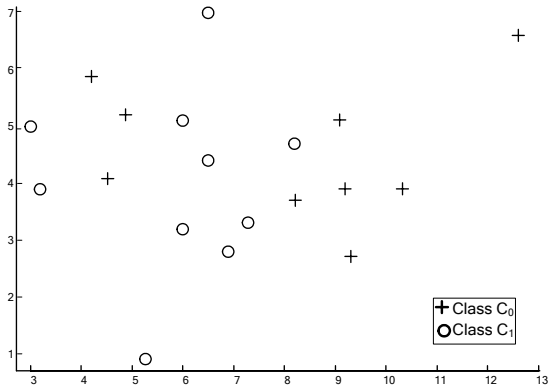


Figure 8.1 Data set used by Sussner (Sussner 1998)

8.2.2 Spiral Data Set

The spiral data set is a two-class set of patterns, each set representing a spiral. The parametric equations for the spiral are:

$$\begin{aligned} x_0(theta) &= theta * \cos(theta) * 2 / pi \\ y_0(theta) &= theta * \sin(theta) * 2 / pi \end{aligned} \tag{8.1}$$

and

$$\begin{aligned} x_1 &= -x_0 \\ y_1 &= -y_0 \end{aligned} \tag{8.2}$$

where theta=0*pi/16+pi/2, 1*pi/16+pi/2, 2*pi/16+pi/2…39*pi/16+pi/2. Half of the patterns were used to train the neural network and the other half were used to test the performance of the resulting neural network as shown in Figure 8.2

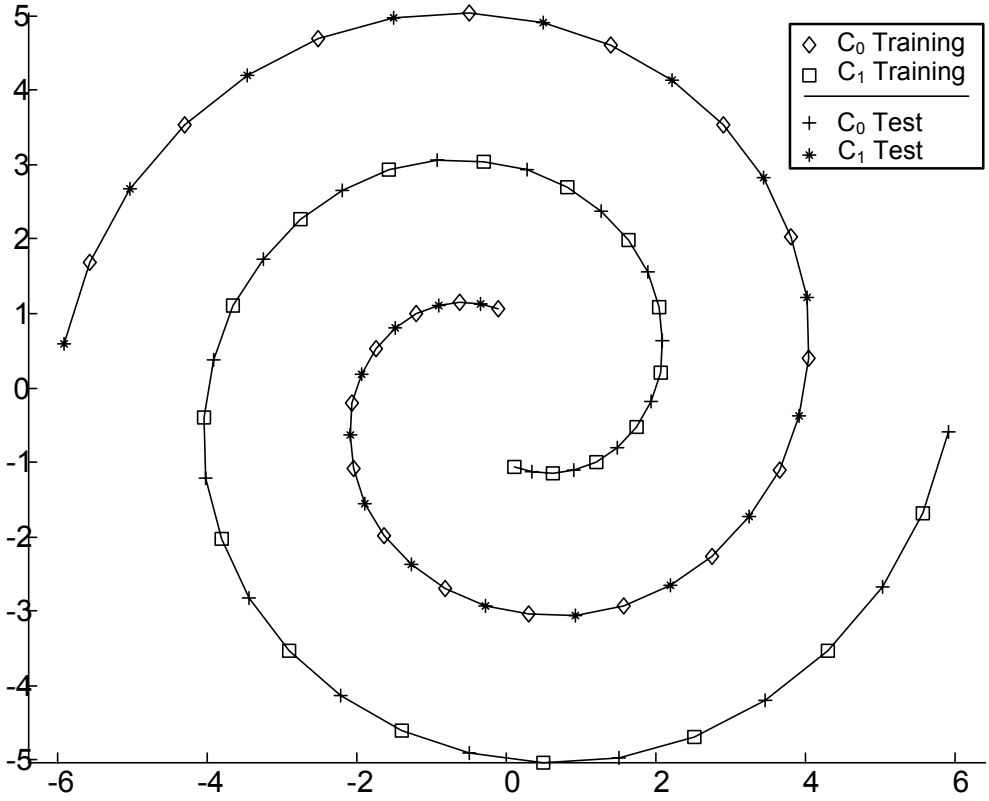


Figure 8.2 Spiral data set used during the training and performance of the resulting neural network.

8.2.3 Iris Fisher Data

The Iris Fisher Data (IFD) is often used as a benchmark in the field of pattern recognition. It consists of 150 patterns equally divided among 3 classes. Each group corresponds to one species of Iris Flower: Iris Sectosa (class C_0), Iris Versicolor (class $C_{1)}$, and Iris Verginica (class C_2). Each class has 4 attributes, representing petal width, petal length, and sepal width and sepal length expressed in inches. Since the IFD contains more than two outputs classes, multiple MLMP had to be trained, one MLMP for each entry in the binary vector associated to each class, $C_0=\begin{bmatrix}0\\0\end{bmatrix}$, $C_1=\begin{bmatrix}1\\0\end{bmatrix}$, and $C_2=\begin{bmatrix}0\\1\end{bmatrix}$.

8.3 PERFORMANCE ANALYSIS

Several tests were conducted using multidimensional data sets such as Iris Fisher Data, Spiral Data set, and the Sussner Data Set. All the learning algorithms were used to train a multilayer morphological perceptrons, then neural network *memorization* and *generalization* was measured in addition to the number of generations needed to reach convergence. Since *Direct Encoding* method can define the connection weights for a neural network with fixed architecture, and no more than two layer morphological perceptron can be trained using this method, the decision boundaries defined by the resulting neural network are very simple. Usually, the resulting neural network is able to separate patterns from two different classes if all the patterns from one of the classes are grouped into a single cluster. The Spiral Data set requires a complex decision boundary that the *Direct Encoding* method is not able to define, for these reason additional data sets were defined to measure the performance of the evolutionary traning algorithm.

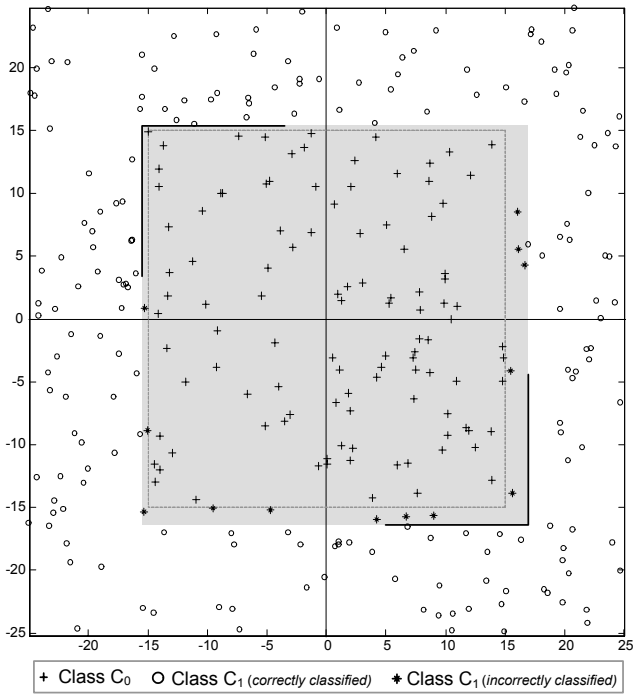
8.3.1 Direct Encoding Method

The network configuration consists of a two-layer morphological perceptron, with a variable number of neurons in the first layer, depending on the distributions of the patterns in the data set. Figure 8.3, Figure 8.4, and Figure 8.5 show how the patterns were distributed.

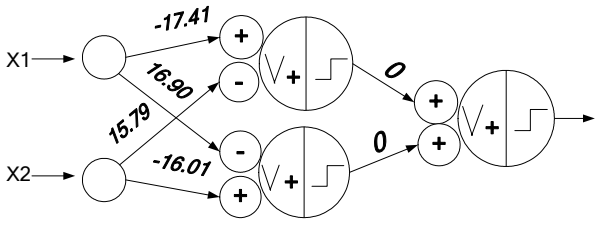
Figure 8.3 shows an example of a $\Re^2$ space and its corresponding decision boundaries defined by the algorithms. The corresponding neural network architecture is shown in Figure 8.3b. Another $\Re^2$ example is shown in Figure 8.4, and Figure 8.5 shows pattern distribution for a $\Re^3$ space and the corresponding morphological perceptrons.

A population of 20 individuals was used for all the tests. Arithmetic crossover and single point mutation were the genetic operators used for the tests. The crossover probability was assigned to be 80%, and the mutation probability varied from 33% to 5%. The evolutionary time was limited to 400 generations for the patterns in Figure 8.3a, Figure 8.5, and the Iris Fisher Data. A total of 500 generations were used for the patterns in Figure 8.4. Experimental results show that in most of the performed tests at least 90% of the patterns were classified correctly.

In addition, the algorithm was tested using the Iris Fisher Data. Half of the patterns were used to training the MLMP, and the other half of patterns ware used to test the performance of the neural network. Table 8.1 summarizes the results obtained from training process for the Iris Fisher Data in addition to the other data sets.



a.



b.

Figure 8.3 (a) 2-Dimension problem and the corresponding architecture Data (b) Two perceptrons are used in the first layer to define its boundaries.

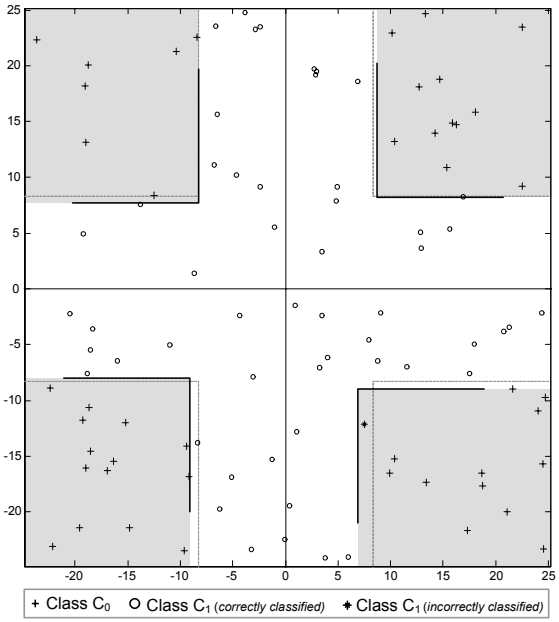


Figure 8.4 Patterns from the class C_0 are distributed among the four corners.

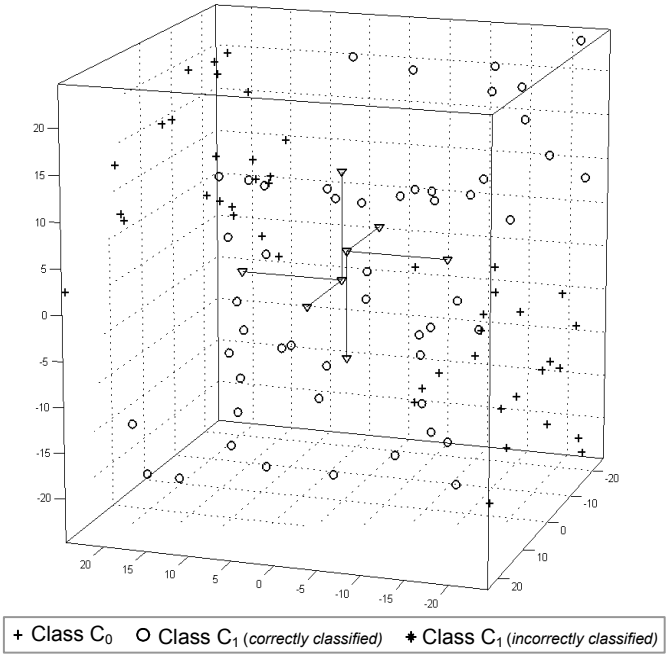


Figure 8.5 A 3-dimensions search space and the corresponding classification boundaries.

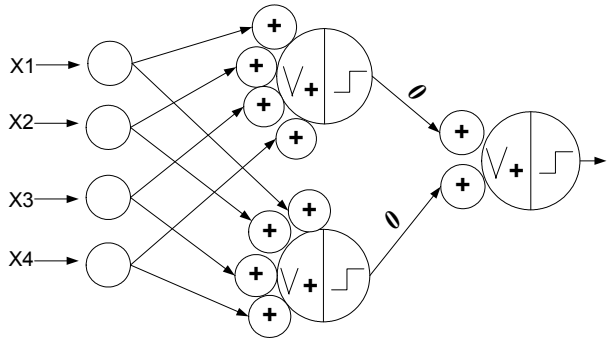


Figure 8.6 Neural network architecture used to produce one of the outputs of the binary vector associated to the class.

	Figure 8.3a	Figure 8.4	Figure 8.5	Iris Fisher Data	
Number of Runs	30	30	30	30	
Population Size	20	20	20	20	
Min. Classification Memorization	85.33%	82.00%	92.71%	93.33%	
Avg. Classification Memorization	89.63%	94.57%	98.72%	96.36%	
Max. Classification Memorization	100.00%	100.00%	100.00%	100.00%	
Min. Classification Generalization	82.00%	79.00%	81.91%	88.80%	
Avg. Classification Generalization	88.15%	89.37%	94.57%	92.04%	
Max. Classification Generalization	100.00%	100.00%	97.34%	97.33%	
Min. Number of Generations	382	91	48	1000	121
Avg. Number of Generations	919	697	196	1000	894
Max. Number of Generations	1000	1000	400	1000	1000

Table 8.1 Summary of results of the tests for the direct encoding training algorithm.

8.3.2 Indirect Encoding Method

Order base crossover and group mutation described in Chapter 6 were used as the genetic operators by the Indirect Encoding Method. The crossover probability was assigned to be 80% and the mutation probability was 3%. The crossover probability was relatively high, but it was not 100%, therefore the resulting offspring produced by the crossover may be identical to the parents.

Using a population of 20 individuals, convergence was typically reached in 100 iterations. Experimental results show that in most of the performed tests at least 96% of the patterns were memorized correctly. Typically all the patterns are memorized in early generations; however network topology improves with more iteration reducing the number of redundant neurons.

Figure 8.7 shows an example of the decision boundaries defined by the algorithm for the Sussner data set. Figure 8.9 shows another example of the decision boundaries defined for Spiral data set.

The algorithm was tested using the Iris Fisher Data. Half of the test patterns were used to train the system. The maximum number of generations used for the test was 200 generations. In most of the tests, the resulting neural network was able to memorize 100% of the patterns. The other half of the patterns were used to test the generalization performance of the resulting network, obtaining up to 77% of the patterns classified correctly. Table 8.2 summarizes the results of the tests for Sussner, Spiral and Iris Fisher data sets.

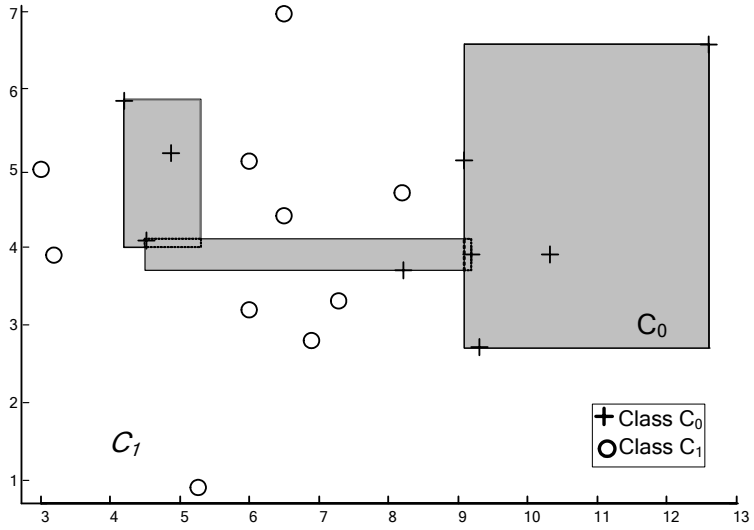


Figure 8.7 Decision boundaries found by indirect encoding method for Sussner Data set.

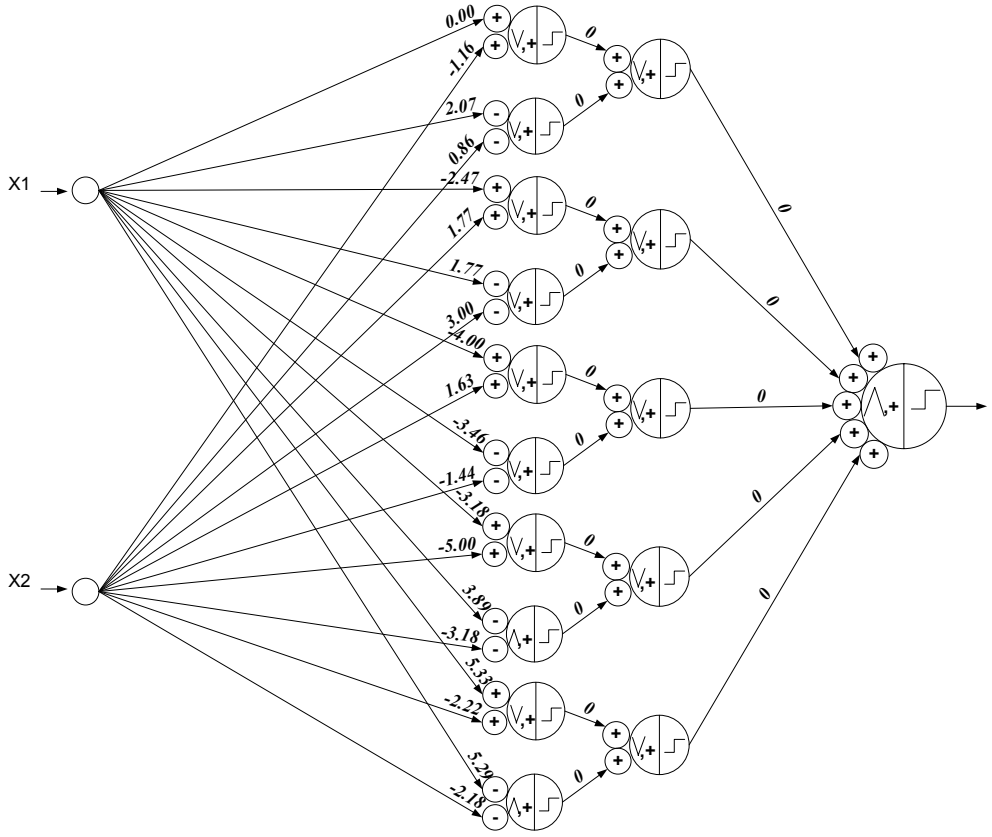


Figure 8.8 Neural network architecture produced by indirect encoding method for Spiral Data set.

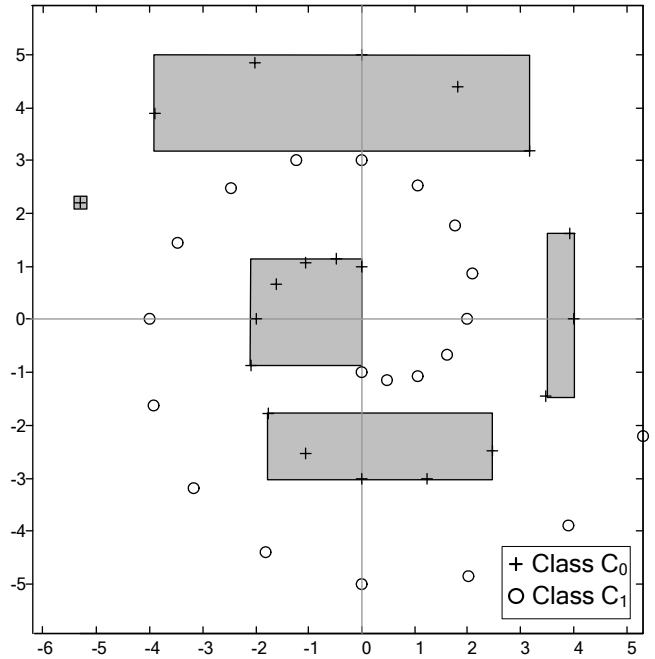


Figure 8.9 Decision boundaries defined by the network architecture shown in Figure 8.8

	Sussner	Spiral	Iris Fisher Data	
Number of Runs	30	30	30	
Population Size	20	20	20	
Min. Classification Memorization	100.00%	100.00%	100.00%	
Avg. Classification Memorization	100.00%	100.00%	100.0%0	
Max. Classification Memorization	100.00%	100.00%	100.00%	
Min. Classification Generalization	n/a	52.50%	50.00%	
Avg. Classification Generalization	n/a	62.25%	66.36%	
Max. Classification Generalization	n/a	72.50%	77.33%	
Min. Number of Generations	2	2	2	9
Avg. Number of Generations	3	10	2	27
Max. Number of Generations	3	30	2	52

Table 8.2 Summary of results for indirect encoding training algorithm.

8.3.3 Cartesian Genetic Programming

Several tests were conducted with the available data sets. An initial population of 20 individuals was used in the entire test. The multipoint mutation and multipoint crossover described in the previous section were used as the genetic operations. A probability of 80% was used for the crossover operator and a maximum probability of 8% was used for the mutation of the neuron operator, and 3% was used for the mutation of the network topology.

The algorithm was tested with the data set used by Sussner. A matrix of nodes was used with three columns. The first column had 20 nodes, the second columnd had 10 nodes, and the third and last column had 1 node. The nodes in the second layer were configured to receive two inputs from the first layer. Convergence was usually archived in 500 generations. Figure 8.10 shows the corresponding neural network architecture for a multilayer morphological perceptron defined by the learning algorithm, and Figure 8.11 shows the corresponding decision boundary with opened regions.

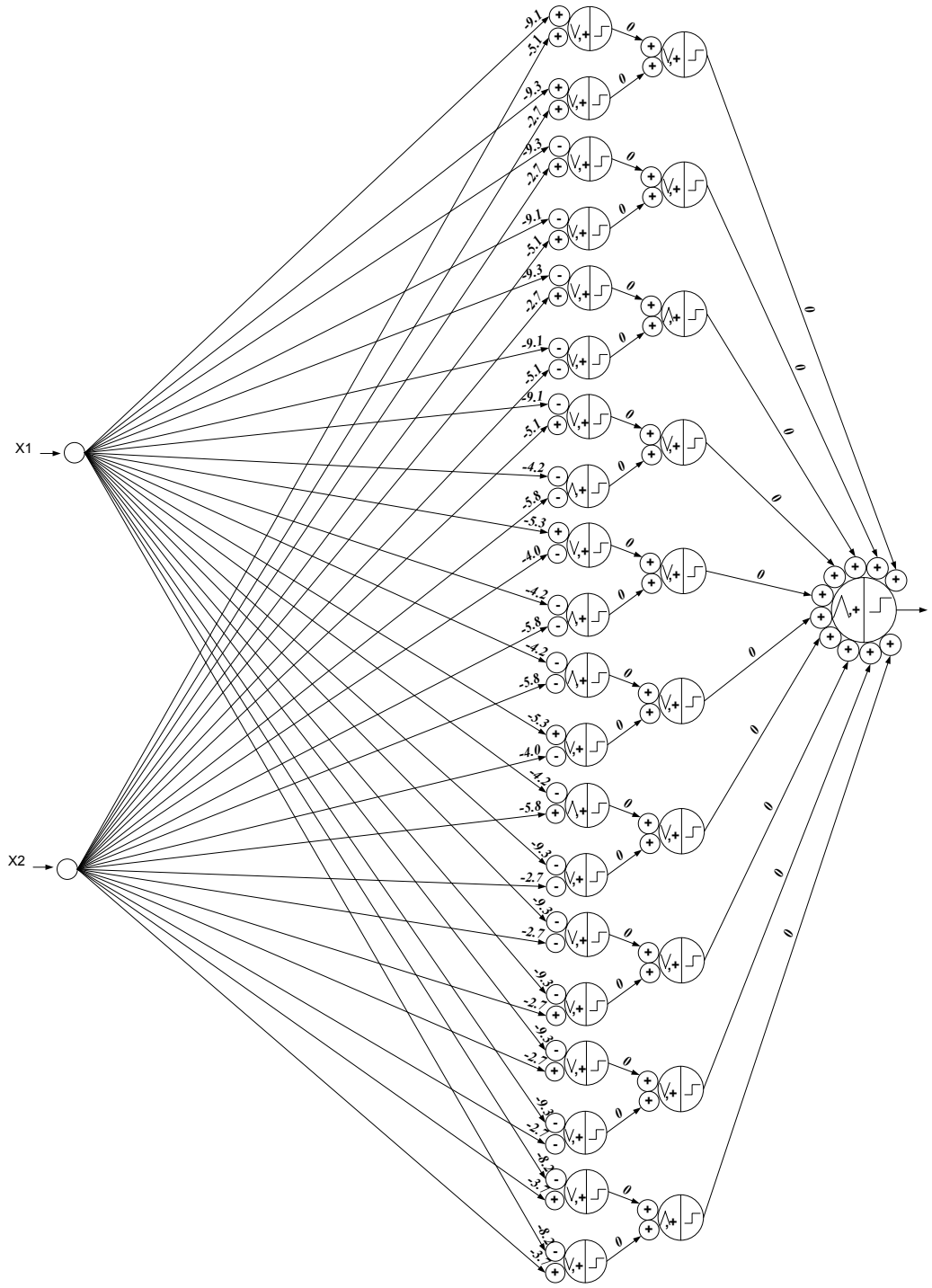


Figure 8.10 Multilayer morphological perceptron defined by the Cartesian Genetic Programming method for Sussner Data Set. Corresponding decision boundary is shown in Figure 8.11

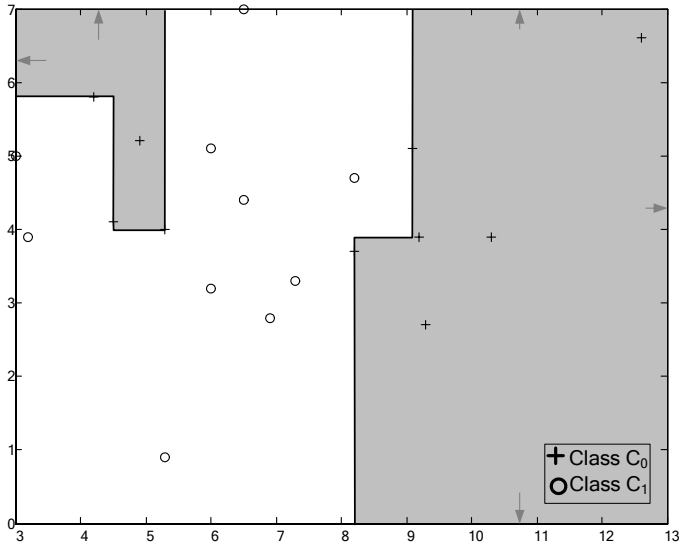


Figure 8.11 Decision boundaries defined by CGP with opened decision boundaries.

In the second example, the previous data set was modified in such a way that only closed decision boundaries were able to classify patterns correctly. Three layers were used with 20 nodes in the first layer, 10 nodes in the second layer and 1 node in the third layer. To force the algorithm to produce compact regions, the nodes in the second layer were configured to receive three inputs. Convergence was usually achieved in 1200 generations. Figure 8.12 shows an example of the decision boundary created by the learning algorithm with closed regions.

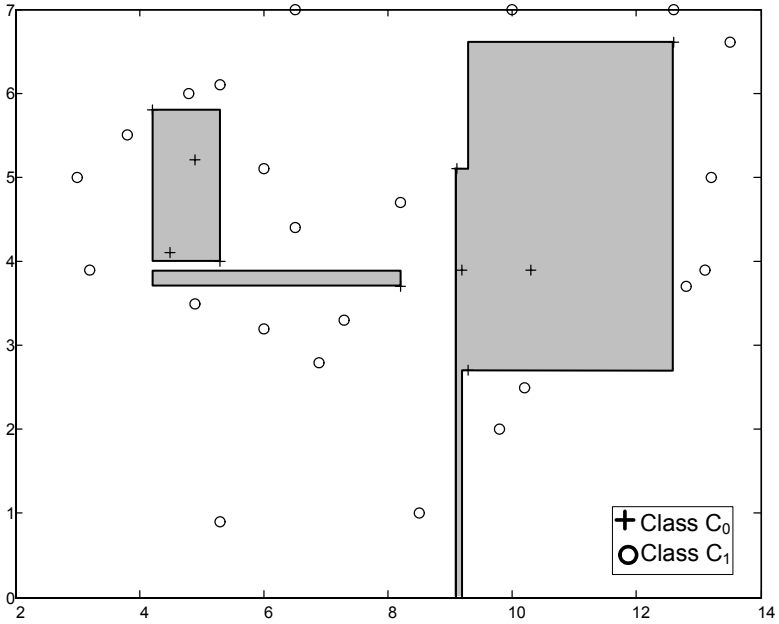


Figure 8.12 Decision boundaries defined by CGP method with closed regions.

In the third example, 40 patterns of 2 dimensions were distributed to form two different spirals, as shown in Figure 8.13. The nodes were distributed among three layers: 30 nodes in the first layer, 20 nodes in the second layer and 1 node in the third layer. The nodes in the second layer were configured to receive two inputs. Convergence was usually achieved by 1200 generations. Figure 8.13 shows an example of the decision boundary defined by the learning algorithm, including opened as well as closed regions.

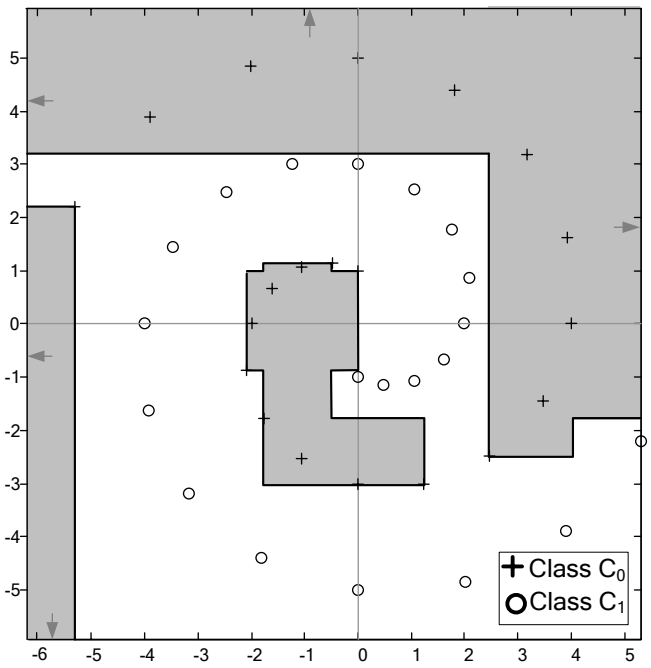


Figure 8.13 Decision boundaries defined for the spiral data set.

In addition, the algorithm was tested with the Iris Fisher Data. During the training process 50% of the patterns were used, and the other 50% was used to measure the generalization of the resulting neural network. The nodes were distributed among three layers: 4 nodes in the first layer, 2 nodes in the second layer and 1 node in the third layer. This simple layer configuration was used to the nature of the distribution of Iris Fisher Data Set. Convergence of the algorithm was typically achieved by 800 generations and 100% of the patterns used during the training process were memorized by the neural network.

	Sussner (Figure 8.11)	**Sussner** (Figure 8.12)	**Spiral**	**Iris Fisher Data**	
Number of Runs	30	30	30	30	
Population Size	20	20	20	20	
Min. Classification Memorization	95.00%	96.97%	90.00%	97.33%	
Avg. Classification Memorization	95.50%	99.60%	96.50%	99.64%	
Max. Classification Memorization	100.00%	100.00%	100.00%	100.00%	
Min. Classification Generalization	*n/a*	*n/a*	72.50%	77.33%	
Avg. Classification Generalization	*n/a*	*n/a*	79.17%	87.78%	
Max. Classification Generalization	*n/a*	*n/a*	85.00%	94.67%	
Min. Number of Generations	*156*	*101*	1497	99	198
Avg. Number of Generations	*486*	*1970*	3846	949	3316
Max. Number of Generations	*500*	*5000*	4000	2630	8000

Table 8.3 Summary of results for the Cartesian Genetic Programming method.

CHAPTER 9

CONCLUSION

9.1 INTRODUCTION

This thesis explores the use of evolutionary algorithms a training tool for Multilayer Morphological Perceptrons. The learning algorithms described on this thesis are based on evolutionary techniques such as: Genetic Algorithms, and Cartesian Genetic Programming. These learning algorithms may be used to train single output feed forward Multilayer Morphological Perceptron for multidimensional/multi-classes pattern classification problems.

9.2 DISCUSSION OF RESULTS

The Iris Fisher Data set and the Spiral data set were used as benchmark to measure the performance of the learning algorithms in terms memorization, that is, the ability to classify correctly the training data set; generalization, that is, predictions for new inputs patterns, and number of generations needed to reach convergence, as shown in Table 9.1.

The performance results of the morphological neural network defined by all the evolutionary learning algorithms were similar to the results presented by Ritter's algorithm (Ritter and Beavers 1999). *Indirect encoding method* was the only training algorithm able to overcome Ritter's algorithm in terms of time needed to reach convergence, but this happened at expenses of a high number of redundant neurons. Direct Encoding method completed most of the tests in lest than 4 seconds running on a

Pentium M 1.6 Ghz, using Matlab 6 and Windows XP. The time needed to reach convergece by *Direct Encoding method* and *Cartesian Genetic Programming method,* vary depending on how complex should be the decision boundaries needed in order to classify the training set correctly. The *CGP* method is able to define complex decision boundaries at expences of additional evolutionary time. The time needed to traing half of the patterns of the Iris Fisher Data set using the Cartesian Genetic Programing method extends over 1 hour. Similar results were obtained training the Spiral Data set.

	CARTESIAN GENETIC PROGRAMMING METHOD			**INDIRECT ENCODING METHOD**			**DIRECT ENCODING METHOD**	
	Spiral	**Iris Fisher Data**		**Spiral**	**Iris Fisher Data**		**Iris Fisher Data**	
Num. of Runs	30	30		30	30		30	
Pop. Size	20	20		20	20		20	
Min. Memorization.	90.00	97.33		100.00	100.00		93.33%	
Avg. Memorization.	96.50	99.64		100.00	100.0%		96.36%	
Max. Memorization	100.00	100.00		100.00	100.00		100.00%	
Min. Generalization	72.50	77.33		52.50	50.00		88.80%	
Avg. Generalization	79.17	87.78		62.25	66.36		92.04%	
Max. Generalization	85.00	94.67		72.50	77.33		97.33%	
Min. Generations	1497	99	2	2	1000	121	9	198
Avg. Generations	3846	949	3	2	1000	894	27	3316
Max. Generations	4000	2630	3	2	1000	1000	52	8000

Table 9.1 Summary of results for the Cartesian Genetic Programming method.

Contrary to the training method presented by Sussner (Sussner 1998), which is limited to train up-to two layers morphological perceptron, the *Indirect Encoding method* and the *CGP* method are able to define three layers morphological neural networks, which is able to solve most pattern classification problems. The CGP method virtually can trains multilayer morphological perceptron of any number of layers. Similary to the algorithm presented by Sussner, the evolutionary learning algorithms are able to determine the number of nodes needed in the hidden layers. Usually neural network

architectural flexibility is achieved with an additional cost in the time needed to reach converge.

For the particular case of the spiral data set, all the patterns were correctly memorized after 10 generations during the training stage, but the resulting neural network was not able to predict correctly 100% of the patterns that were not used during the training. This happened due to the fact that a single pattern can be assigned to one group exclusively, which means that a pattern may be enclosed by a single hypercube, therefore there may be hypercubes which do not overlap, this may result in an incorrect prediction for the classification of those patterns that falls between two non-overlapping hypercubes as shown in Figure 9.1.

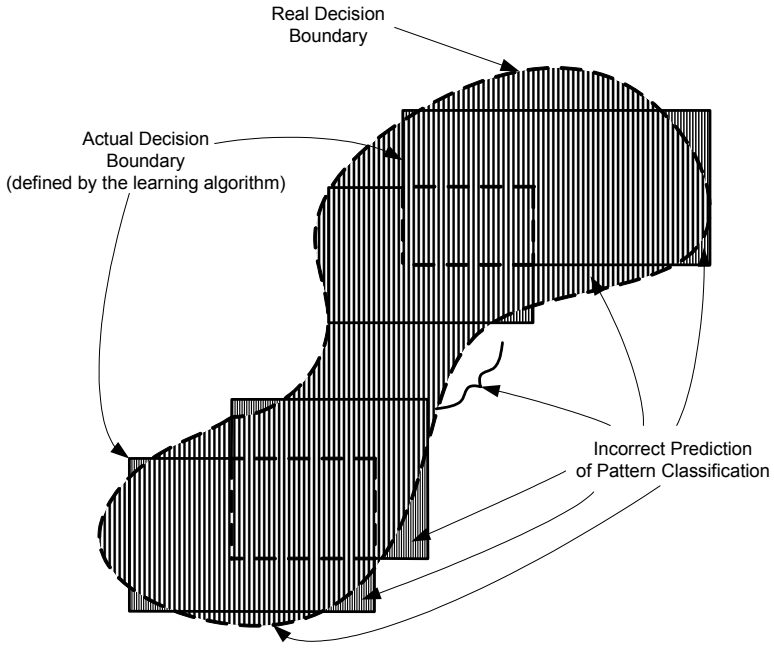


Figure 9.1 Incorrect generalization of the neural network.

Differences in number of generations are due to the increased complexity in the search space for different data sets. Different node configurations are used for different data sets. When the number of nodes is incremented, complex decision boundaries can be defined to perform correct pattern classification at the expenses of additional

evolutionary time. Due to the nature of the morphological neuron, a single neuron can classify most of the Iris Fisher data. For this reason a small number of nodes are used to search for the correct network architecture. Other data sets such as the spiral data set shown in Figure 8.13 or the Sussner data set shown in Figure 8.12 require a higher number of neurons to define complex decision boundaries.

9.3 COMPARISON OF THE LEARNING ALGORITHMS

9.3.1 Direct Encoding Method

The *Direct Encoding* method lacks of scalability for very large problems, due to the fact that the entire space of solutions can not be mapped in detail by the genetic code. The architecture is fixed during the training of the neural network, and a maximum of two layer morphological perceptron can be trained which can be used to solve simple pattern classification problems. The algorithm needs information about architecture of the neural network to be trained, such as number of layers, and number of neurons in the first layer. This information is provided prior the training, after inspection of the data set distribution.

The number of neurons must be specified before the training, and that number is fixed during the evolution. To achieve better utilization of the neurons, the algorithm introduces the use of a penalty function in the evaluation function. A penalty is assigned to those individuals with redundant neurons. Redundant neurons are those neurons that if were removed from the neural network, the decision boundary of the resulting neural network remains unchanged. This is the way the evolutionary algorithm promotes those neural networks that represent a better solution placing neuron on the right location.

The learning algorithm is able to define opened decision boundaries as well as close decision boundaries. The resulting neural network is limited to define very simple decision boundaries which may be able to separate patterns grouped into a single cluster of from other patterns, therefore may not be used to train classify patterns dispersed into several clusters.

In terms of the evolutionary time, the learning method performs an exhaustive computational search over a continuous space of values, looking for the optimum set of connection weights, resulting in a very time consuming task. The evolutionary learning method introduces the use adaptive mutation for the evolution of morphological neural networks to speed up convergence and reduce the evolutionary time.

9.3.2 Indirect Encoding Method

The evolutionary method is able to evolve some architectural elements. This trade off was made to improve convergence speed. The convergence speed of the *Indirect Encoding* method is faster compared to the other two evolutionary learning algorithms, due to fact that the search for the best architecture is done in a reduced and simplified search space. Connection weights are searched in a discrete space and the network architecture is limited to some architectural constrains, such as fixed neuron inter-connection, connection weights for the second and third layer are predefined and fixed during the training, reducing the number of unknown connection weights. Architectural constrains limit the neural network to a maximum of 3 layers, which are needed most pattern classification problems where the patterns can be grouped into clusters.

Usually the algorithm is able to classify most of the patterns at early generations, but network architecture improves with more iteration, reducing the number of redundant

neurons. The evaluation function is used to guide the evolutionary process to create as many hypercubes as needed to correctly classify all the patterns in early generations, and improve the individuals by removing redundant neurons.

Differing from the *Direct Encoding* method and the *Cartesian Genetic Programming* method, the algorithm needs no information about pattern distribution. The evolutionary learning method determines the number of neurons needed in the first layer and the correct number of layers by itself. The resulting neural network is able to define exclusively close decision boundaries. Close decision boundaries results in a better prediction of clustered data set of patterns, but in an inaccurate prediction of non-clustered data sets, which is the major drawback of the learning algorithm. Patterns may be distributed in many different clusters, defining complex decision boundaries.

9.3.3 Cartesian Genetic Programming Method

The *Cartesian Genetic Programming* method introduces the use of Cartesian Genetic Programming as a training tool for Morphological Neural Networks. The method may be able to train multi-layer morphological perceptrons of any number of layers. The learning algorithm provides flexible evolution of neural network architecture, resulting in flexible neural networks solutions. The method needs information about the neural network architecture such as the number of layer, the maximum number of neurons for each layer; and the number of inputs received by the intermediate layers. This information is provided to the algorithm after inspection of the data sets distribution.

This algorithm introduces the use of the morphological neuron computational model as the basic node operation for Cartesian Genetic Programming, allowing the evolutionary training of Morphological Neural Networks. The resulting neural networks

may be able to produce complex decision boundaries, including opened decision boundaries, as well as closed decision boundaries.

In terms of the evolutionary time, the algorithm tends to take longer to define the corresponding decision boundaries, due to the fact that the algorithm search for the network architecture in an open space of architectures, limited only by the maximum number of nodes for each layer and the total number of layers. The search space gets even more complex as additional patterns are used during the training process, due to the addition of patterns the connection weights the search space increases in an exponential order. To speed up convergence, the evolutionary method makes use of two different mutations rates which affect different regions in the chromosome: network architecture and neurons operation.

Similar to Indirect Encoding method, the learning method make use of a special penalty function. This penalty function promotes the reduction of redundant or unnecessary neurons from the neural network.

9.3.4 Summary of Differences

A list of advantages and disadvantages presented by the evolutionary algorithms is shown in Table 9.2.

Training Method	Advantages	Disadvantages
Direct Encoding Method	1. Define open and closed decision boundaries 2. Reduce number of redundant neurons	1. Rigid topology 2. Open search space for connection weights 3. Long time needed to reach convergence
Indirect Encoding Method	1. Some degree of topological flexibility 2. Discrete search space for connection weights 3. Reduced search space results in fast convergence 4. Neural architecture evolves by itself definining nodes for hidden layers	1. High number of redundant neurons 2. Define exclusively closed decision boundaries
Cartesian Genetic Programming Method	1. High degree of topological flexibility 2. Discrete search space for connection weights 3. Define open and closed decision boundaries 4. Neural architecture evolves by itself definining nodes for hidden layers	1. Exponential grow in the search space 2. Long time needed to reach convergence

Table 9.2 Summary of advantages and disadvantages of the evolutionary learning algorithms.

9.4 PROTOTYPES LIMITATIONS

The implementation of the prototype for the *Indirect Encoding Method* is restricted to train a maximum of 50 patterns. This limitation is due to the encoding representation scheme for the groups of patterns defined for the chromosome in Matlab. The actual representation uses the bits of an integer to define group boundaries. The limitation may be overcome using a different representation scheme for the groups of patterns, maybe using a complex data structure or a different programming language. All the other prototypes may be able to train a neural network using as many patterns as needed.

9.5 FUTURE WORK

After the completion of this thesis, new interesting research topics had appeared. Possible research areas may focus on the development of a learning algorithm which may be able to produce opened decision boundaries as well as closed decision boundaries without considering the actual pattern distribution. In addition, the learning algorithms may be able to evolve a morphological neural network which produces multiple outputs simultaneously. Different indirect encoding schemes, such as attribute grammar, or the cellular encoding maybe used to improve performance while reducing complexity of the problem. Additionally, it may be interesting to explore how to train morphological neural networks that use different transfer functions, such as *log sigmoid*, and the exploration of possible uses for those neural networks.

9.6 CONCLUSION

Multilayer Morphological Perceptrons provide simple and scalable solutions for new generations of patterns classifiers. At the same time, evolutionary algorithms provide robust search mechanism to explore in an extensive number of possible solutions. At this point, it is concluded that evolutionary learning algorithms may be used as an alternative training method for Multilayer Morphological Perceptrons. Additional research needs to be conducted to explore different evolutionary learning algorithms that may be capable to define decision boundaries that improves neural network generalization abilities without considering exact pattern distribution, while, at the same time, reduce the search space of the neural network architecture, connection weights, and increases convergence speed.

REFERENCES

BISHOP, C.M., 1996, "Neural Networks for Pattern Recognition", Oxford University Press, Oxford, UK

BRANKE, J., 1995, "Evolutionary Algorithms for Neural Network Design and Training", In Proceedings of the 1st Nordic Workshop Genetic Algorithms Applications. T. Talander.

DUDA, R.O., HART, P.E., STORK, D.G., 2001, "Pattern Classification", 2nd Ed., John Wiley & Sons, Canada.

ESPARCIA-ALCAZAR, A.I., SHARMAN, K.C., 1996, "Genetic Programming Techniques that Evolve Recurrent Neural Network Architectures for Signal Processing", Neural Networks for Signal Processing VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop. pp. 139 -148

FOGEL, D.B, 1994, "An introduction to simulated evolutionary optimization", IEEE Trans. On Neural Networks, vol. 5, no. 1, pp. 3-14

FOGEL, D.B., FOGEL, L.J., 1996, "An Introduction to Evolutionary Programming", LNCS vol. 1063, pp. 21-33

FUKUDA, T., KOHNO, T., AND SHIBATA, T., 1993, "Learning Scheme for Recurrent Neural Network by Genetic Algorithm", Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems. vol. 3, pp. 1756-1761.

GRUAU, F., 1992, "Genetic Synthesis of Boolean Neural Networks with a Cell Rewriting Developmental Process." In *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92),* L. Darrell Whitley and J. David Schaffer, editors. pp. 55–74. Los Alamitos: IEEE Press.

GRUAU, F., WHITLEY, L.D., 1993, "Adding Learning to the Cellular Development of Neural Networks: Evolution and the Baldwin Effect." Evolutionary Computation 1(3): 213-233

GRUAU F., WHITLEY, L.D, PYEATT, L., 1995, "Cellular encoding applied to neurocontrol." In Proceedings of the Sixth International Conference on Genetic Algorithms

HAGAN, M. T, DEMUTH, H. B., BEALE, M., 1996, *Neural Network Design*. Boston, MA: PWS.

HARP, S.A., SAMAD, T., GUHA, A., 1989, “Towards the genetic synthesis of neural networks”, Proc. Of the Third Int’l Conf. on Genetic Algorithms and Their Applications, pp.360-369. Morgan Kaufmann, San Mateo, CA.

HARP, S.A., SAMAD, T., GUHA, A., 1990, “Designing application specific using genetic algorithms”, in Advances in Neural Information Processing Systems 2, pp. 447-454, Morgan Kaufmann, San Mateo, CA.

HINTZ, K.J., SPOFFORD, J.J., 1990, "Evolving a neural network", Proceedings of the 5th IEEE International Symposium on Intelligent Control, pp. 479 -484

HOWARD, L.M. AND D’ANGELO D.J., 1995, “The GA-P: a Genetic Algorithm & Genetic Programming hybrid”, IEEE Expert, pp. 11-15

HUSSAIN, J.E,. BROWSE, R.A., 1998, “Attribute Grammars for Genetic Representations of Neural Networks and Syntactic Constrains of Genetic Programming”, In AIVIGI’98, Workshop on Evolutionary Computation

JACOB,C., REHDER, J., 1993, "Evolution of neural networks architectures by a hierarchical grammar-based genetic system", ANNGA'93, Proc. of the International Conference on Artificial Neural Networks and Genetic Algorithms, Innsbruck, pp. 72 - 79.

KARUNANITHI, N., DAS, R., AND WHITHLEY, D., 1992, “Genetic cascade learning for neural networks.” In Schaffer and Whitley, editors, Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks, pages 134-144.

KITANO, H., 1990, “Designing Neural Networks Using Genetic Algorithms with Graph Generation System”, Complex Systems, vol. 4, no. 4, pp., 461-476.

KOZA, J.R., 1992, “Genetic Programming: On the Programming of Computer by Means of Natural Selection”, MIT Press, Cambridge

KOZA, J.R., AND RICE, J.P., 1991, “Genetic Generation of both the Weights and Architecture for a Neural Network”, IEEE International Joint Conference on Neural Networks.

LIMA, C.A.M., COELHO, A.L.V., SILVA, M.E.S., GUDWIN, R.R, ZUBEN F J.V: 2001 “Hybrid Training of Morphological Neural Networks: A Comparative Study”, Procs. of National Meeting of Artificial Intelligence (ENIA), Congress of Brazilian Computing Society (SBC), pp. 1499-1507, Fortaleza, Brazil.

LUKE, S., SPECTOR, L., 1996, “Evolving Graphs and Networks with Edge Encoding Preliminary Report”, Late Breaking Papers of the Genetic Programming, Stanford

MARSHALL, S.J., HARRISON, R.F., 1991, “Optimization and Training of Feedforward Neural Networks by Genetic Algorithms”, Second International Conference on Artificial Neural Networks. pp. 39-43.

MILLER, G.F., TODD, P.M., HEGDE, S.U., 1989, "Designing Neural Networks using Genetic Algorithms", Proc. Of the Third Int'l Conf. on Genetic Algorithms and Their Applications, pp. 379-384. Morgan Kaufmann, San Mateo, CA.

MILLER, J., THOMSON, P. 2000. "Cartesian genetic Programming in the Genetic Programming." Proceedings of the Third European Conference on Genetic Programming, vol. 1802 of Lecture Notes in Computer Science, 121-132.

MILLER, J., 2001, "What bloat? Cartesian genetic programming on boolean problems", in Proceedings of the 2001 Genetic and Evolutionary Computation Conference, 295–302.

MONTANA, D.J., DAVIS, L., 1989, "Training feedforward neural networks using genetic algorithms." In Proceedings of the International Joint Conference on Artificial Intelligence, pages 762-767.

ORTIZ, J.L., PIÑEIRO, R.C, 2003, "Evolutionary Learning Algorithm for Morphological Perceptron", The Third International Association of Science and Technology for Development (IASTED) International Conference on Artificial Intelligence and Applications, Benalmádena, Spain.

ORTIZ, J.L., PIÑEIRO, R.C., 2004A, "Indirect Encoding Evolutionary Learning Algorithm for the Multilayer Morphological Perceptron", Proceedings of the 17th International Florida AI Research Society (FLAIRS) Conference, Fl, USA, May 17

ORTIZ, J.L., PIÑEIRO, R.C., 2004B, "Evolutionary Learning Methods for Multilayer Morphological Perceptron", Transactions on Neural Networks, IEEE (submitted).

RITTER, G.X., BEAVERS, T.W., 1999, "An Introduction to Morphological Perceptrons", Proceedings of the ANNIE '99 (Artificial Neural Networks in Engineering), St. Louis, MO.

RITTER, G.X., SUSSNER, P., 1998, "Morphological Associative Memories", IEEE Transactions on Neural Networks, Vol. 9, No. 2, pp. 281-293.

RITTER, G. X., SUSSNER, P., 1997, "Morphological Perceptrons" in the proceeding of International Conference on Intelligent Systems and Semiotics - A Learning Perspective, Gaithersburg, Maryland.

RITTER, G.X., SUSSNER, P., 1996A, "An Introduction to Morphological Neural Networks", Proceedings of the 13th International Conference on Pattern Recognition, Vol. IV, Track D, pp. 709-717, Austria.

RITTER, G.X. SUSSNER, P., 1996B, Morphological associative memories and perceptrons. Technical Report TR 96-02, CCVR, University of Florida.

SARAVANAN, N., AND FOGEL, D.B., 1995, "Evolving Neural Control Systems", IEEE Expert,Vol. 10, No. 3, pp. 23-27.

SUSSNER, P., 1998, "Morphological Perceptron Learning", Proceedings of the 1998 IEEISIC/CIRA/ISAS Joint Conference, Gaithersburg, MD.

VONK, E., JAIN, L.C., VEELENTURF, L.P.J., JOHNSON, R., 1995, "Automatic Generation of Neural Network Architecture Using Evolutionary Computation", IEEE.

WHITLEY,D., STARKWEATHER,T., BOGART,C., 1990, "Genetic algorithms and Neural Networks: optimizing connections and connectivity", Parallel Computing, vol. 14, no. 3, pp. 347-361.

YAO, X., 1999, "Evolving Artificial Neural Networks", Proceedings of the IEEE, 87(9):1423-1447.

APPENDIX A

EVOLUTINARY LEARNING ALGORITHMS TOOLBOX FOR MATLAB

A.1 INTRODUCTION

This appendix comprends of the used guide for the Evolutionary Learning Algorithms Toolbox defined for Matlab 6 and source code

A.2 USER GUIDE FOR MULTILAYER MORPHOLOGICAL PERCELTRON

A.2.1 Common Configuration Parameters

Each training method requires a set of specific configuration parameters. The configuration parameters control the evolutionary process, including initial population size, termination conditions, genetic operators, and evaluation function. These parameters have been added to provide a flexible control over the evolutionary process. Different genetic operators, as well as evaluation functions may be used, producing different results. Table A.1 presents the common configuration parameters used by all the training methods.

Parameter	Type	Description	Example
param.evalFn	m-file	Specifies the name of the evaluation function used.	['CGPEval3']
param.evalParams	vector of double	Specifies any parameter passed to the evaluation function	[]
param.mutationFn	m-file	Specifies the name of the mutation function.	['CGPMultiPointMutation2']
param.mutationParams	vector of double	Specifies any arguments need by the mutation function.	[0.08 0.08]
param.popSize	integer	Size of the population used during the evolution	20
param.selectFn	m-file	Specifies the name of the selection function, used to select the survivals from a generation to the next one.	['roulette2']
param.selectParams	vector of double	Specifies any parameter passed to the selection function.	[0.33]
param.termParams	integer	Specifies the termination criteria: [max. number of generations, final fitness]	[8000,1.0]
param.xOverFn	m-file	Specifies the name of the crossover function.	['CGPMultipointXover'];
param.xOverParams	vector of double	Specifies any necessary parameter passed to the crossover function.	[0.95 0.80]

Table A.1 Configuration parameters used by all the training methods

A.2.2 Direct Encoding Toolbox

A.2.2.1 Configuration Parameters

In addition to the parameters presented on Table A.1, the *Direct Encoding method* requires configuration parameters described on Table A.2. *Direct Encoding method* requires the number of neurons to be specified prior to the training of the neural network,

each position in the vector specified by *param.layer* represents the number of neuron for each layer.

Parameter	Type	Description	Example
param.layers	vector of integers	Specifies the number of neurons for each layer.	[2 1]
param.opts	vector of double		[1e-6 1 1]

Table A.2 Configuration parameters used by Direct Encoding Method

A.2.2.2 Training Method

[*net, traceInfo*] = DirectTrainMNN(*patterns*, *classes*, *bounds*, *targets*, *config*);

Description: Trains a multilayer morphological perceptron using the *Direct Encoding method*. The method receives as arguments the patterns used during the training process. *Patterns* are passed to the method as an *M*x*N* matrix, which contains *N* patterns of *M* dimensions. All the patterns from all the classes are appended one after another in the argument *patterns*, starting by patterns from class C_0, then patterns from class C_1 are appended, and finally patterns from class C_T, where *T* is the total number of classes to be trained. The parameter *classes* define a column vector containing the number of patterns defined for each class in the *patterns* matrix. *Bounds* is a 2xM matrix in which each row vector represents the lower and upper bounds for each dimension. The variable *targets* contains a *P*x*Q* matrix of binary elements, where each row represents the binary vector associated to a particular class. The parameter *config* is a data structure that contains the configuration parameters shown in Table A.1and Table A.2. The method returns an object *net* that represents a MNN.

Parameter	Type	Description	Example
patterns	matrix of integers	Specifies the all the patterns used during the training process.	class0 = [0 0; 1 1]; class1 = [0 1; 1 0]; patterns = [class0; class1];
classes	vector of integers	Specifies the amount of patterns defined on each class	[2; 2]
targets	matrix of integers	Each row represents the binary vector associated to a particular class	[0 ; 1]
params	struct	Configuration parameters for the algorithm.	as shown in Table 7.1 and Table 7.2
net	MNN	MNN trained for the patterns	
traceInfo	Matrix of double	Performance of the evolution. Four column matrix representing: generation number, fitness of the best individual, average fitness of the generation, and standard deviation	

Table A.3 Parameters passed to the Direct Encoding training method.

A.2.2.3 Sample Code

The code shown in Figure A.1 defines patterns for two classes C_0={(0,0) ,(1,1)}, and C_1={(0,1), (1,0)}, in a two-dimensional search space for the training algorithm and returns a morphological perceptron which is able to classify these patterns.

```
% param is previously defined
class0 = [0 0; 1 1];
class1 = [0 1; 1 0];
patterns = [class0;class1];
targets = [0 ; 1];
classes = [size(class0,1); size(class1,1)];

% Compute the bounds for each dimension
minVals = min(patterns);
bound = [(max(patterns)-minVals); minVals];

% Expand the boundaries by %25
bound = bound +[ boundaries (1,:)*.125; - bound (1,:)*0.125]
[net,traceInfo] = DirectTrainMNN(patterns, classes, bound, targets, config);
```

Figure A.1 Example code of how Direct Encoding Method can be used to train MNN

A.2.3 Indirect Encoding Toolbox

A.2.3.1 Configuration Parameters

Configuration parameters used by the Indirect Encoding method are the configurations parameters shown in Table A.1. No additional configuration parameters are needed.

A.2.3.2 Training Function

[*net, traceInfo*] = IndirectTrainMNN(*patterns, classes, targets, params*)

Description: Trains a multilayer morphological perceptron using the *Indirect Encoding method*. The function receives as arguments the patterns used during the training process. *Patterns* are passed to the function as an *M*x*N* matrix, which contains *N* patterns of *M* dimensions. All patterns from all the classes are appended one after another in the argument *patterns*, starting by patterns from class C_0, then patterns from class C_1 are appended, finally patterns from class C_T, where *T* is the total number of classes to be trained. The parameter *classes* define a column vector containing the number of patterns defined for each class in the *patterns* matrix. The variable *targets* contains a *P*x*Q* matrix of binary elements, where each row represents the binary vector associated to a particular class. The parameter *config* is a data structure that contains the configuration parameters used for the training algorithm. The method returns an object *net* that represents a MNN.

Parameter	Type	Description	Example
patterns	Matrix of integers	Specifies the all the patterns used during the training process.	class0 = [0 1; 1 1]; class1 = [1 0; 0 0]; patterns = [class0; class1];
class_distribution	Vector of integers	Specifies the amount of patterns defined on each class	[2; 2]
targets	Matrix of integers	Each row represents the binary vector associated to a particular class	[0 ; 1]
params		Configuration parameters for the algorithm.	As shown in Table 7.1
net	MNN	MNN trained for the patterns	
traceInfo	Matrix of double	Performance of the evolution. Four column matrix representing: generation number, fitness of the best individual, average fitness of the generation, and standard deviation	

Table A.4 Parameters passed to the CGP training method

A.2.3.3 Sample Code

The code shown in Figure A.2 defines patterns for two classes C_0={(0,0) ,(1,1)}, and C_1={(0,1), (1,0)}, is a 2-dimensinal search space for the training algorithm and returns a morphological perceptron which is able to classify these patterns.

```
% params  is previously defined
class0 = [0 1; 1 1];
class1 = [1 0; 0 0];
patterns = [class0;class1];
targets = [0 ; 1];
classes = [size(class0,1); size(class1,1)];

[net,traceInfo]=IndirectTrainMNN(patterns,classes, targets, params);
```

Figure A.2 Example code of how Indirect Encoding Method can be used to train MNN

A.2.4 Cartesian Genetic Programming Toolbox

A.2.4.1 Configuration Parameters

Configuration parameters used by the training algorithms include the configuration parameters shown in Table A.1 in addition to the configuration parameters

shown in Table A.5. Configuration parameters from Table A.5 define some network properties such as the distribution of nodes, and number of inputs received by the neurons for each layer.

Configuration parameter	Type	Description	Example
param.connections	vector of integers	Specifies the number of connections used by the nodes on each layer.	[20 20 1]
param.layers	vector of integers	Specifies the maximum number of nodes defined for each layer.	[4 2 20]

Table A.5 Additional configuration parameters used by Cartesian Genetic Programming.

A.2.4.2 Training Function

[*net*,*traceInfo*] = *CGPTrainMNN*(*patterns, classes, targets, params*)

Description: Trains a multilayer morphological perceptron based on Cartesian genetic programming. The function receives as arguments the patterns used during the training process. *Patterns* are passed to the function as an *M*x*N* matrix, which contains *N* patterns of *M* dimensions. All the patterns from all the classes are appended one after the other in the argument *patterns*, starting by patterns from class C_0, then patterns from class C_1 are appended, finally patterns from class C_T, where *T* is the total number of classes to be trained. The parameter *classes* define a column vector containing the number of patterns defined for each class in the *patterns* matrix. The variable *targets* contains a *P*x*Q* matrix of binary elements, where each row represents the binary vector associated to a particular class. The parameter *config* is a data structure that contains the configuration parameters used for the training algorithm. The method returns an object *net* that represents a MNN, and a matrix *traceInfo* which consists of three columns. The first column identify the generation number, the second column corresponds to the fitness value assigned to the best organism for the corresponding generation, the third column corresponds to the

average value for the fitness of the population, the value forth column corresponds to the standard deviation.

Parameter	Type	Description	Example

patterns	Matrix of integers	Specifies the all the patterns used during the training process.	class0 = [0 1; 1 1];
			class1 = [1 0; 0 0];
			patterns = [class0;class1];

classes	Vector of integers	Specifies the amount of patterns defined on each class	[2; 2]
targets	Matrix of integers	Each row represents the binary vector associated to a particular class	[0 ; 1]
params		Configuration parameters for the algorithm.	As shown in Table 7.1 and Table 7.5
net	MNN	MNN trained for the patterns	
traceInfo	Matrix of double	Performance of the evolution. Four column matrix representing: generation number, fitness of the best individual, average fitness of the generation, and standard deviation	

Table A.6 Parameters passed to the CGP training method

A.2.4.3 Sample Code

The code shown in Figure A.3 defines patterns for two classes C_0={(0,0) ,(1,1)}, and C_1={(0,1), (1,0)}, the search space for the training algorithm, and returns a morphological perceptron which is able to classify the patterns.

```
% params  is previously defined
class0 = [0 1; 1 1];
class1 = [1 0; 0 0];
patterns = [class0;class1];
targets = [0 ; 1];
classes = [size(class0,1); size(class1,1)];

[net,traceInfo] = CGPTrainMNN(patterns, classes, targets, params);
```

Figure A.3 Example code of how Indirect Encoding Method may be used to train MNN

A.3 COMMON TOOLS

This section describes a set of common tools used by all the training methods to manipulate and control morphological neural networks.

A.3.1 Pattern Classification

[*class*] = evalMorphologicalNet(*net*, *patterns*)

Description: Classify the patterns defined by the argument *patterns* given a vector of MLMP denoted by the argument *net*. Each entry in the vector *net* represents a Multilayer Morphological Perceptron used to construct the classification vector. Multiple patterns may be classified simultaneously using a single function call as shown in Figure A.4. In Figure A.4, three 4 dimensional patterns are assigned to class $\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$. Each row from the *class* matrix denote the corresponding classification for each pattern defined by each row from the *patterns* argument.

```
class1 =

   5.1000   3.5000   1.4000   0.2000
   4.9000   3.0000   1.4000   0.2000
   4.7000   3.2000   1.3000   0.2000

>> evalMorphologicalNet(net,class1)

ans =

    0    1    1
    0    1    1
    0    1    1
```

Figure A.4 How to use Multilayer Morphological Perceptrons to classify multiple patterns.

A.3.2 Plotting the Network

***plotNetwork2*(*net*)**

Description: Display a graphical representation of the perceptrons denoted by the argument ***net*** in a 2-dimensinal space, as shown in Figure A.5.

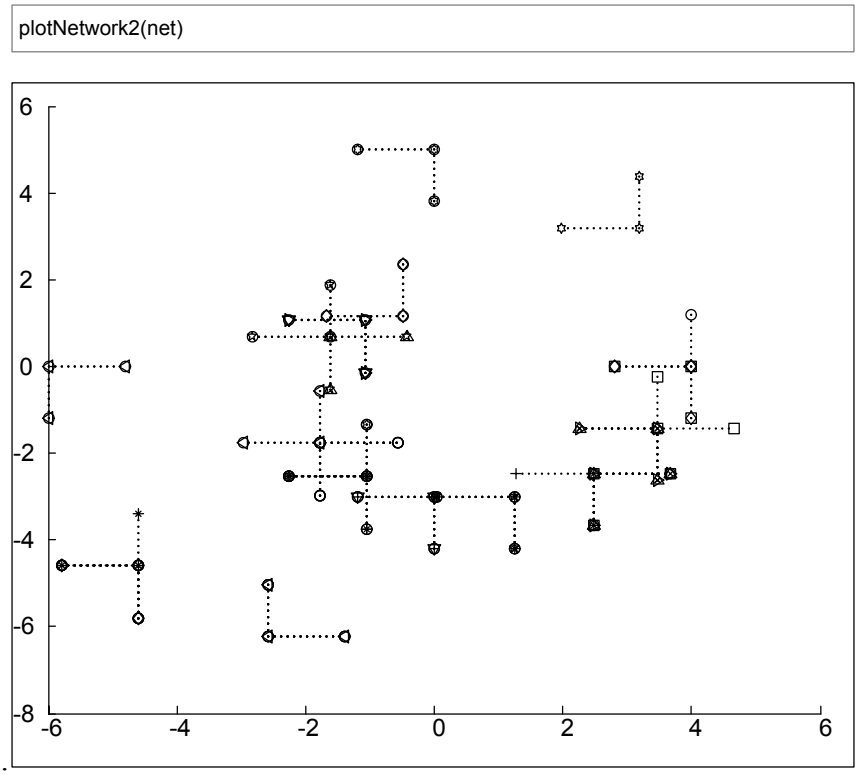
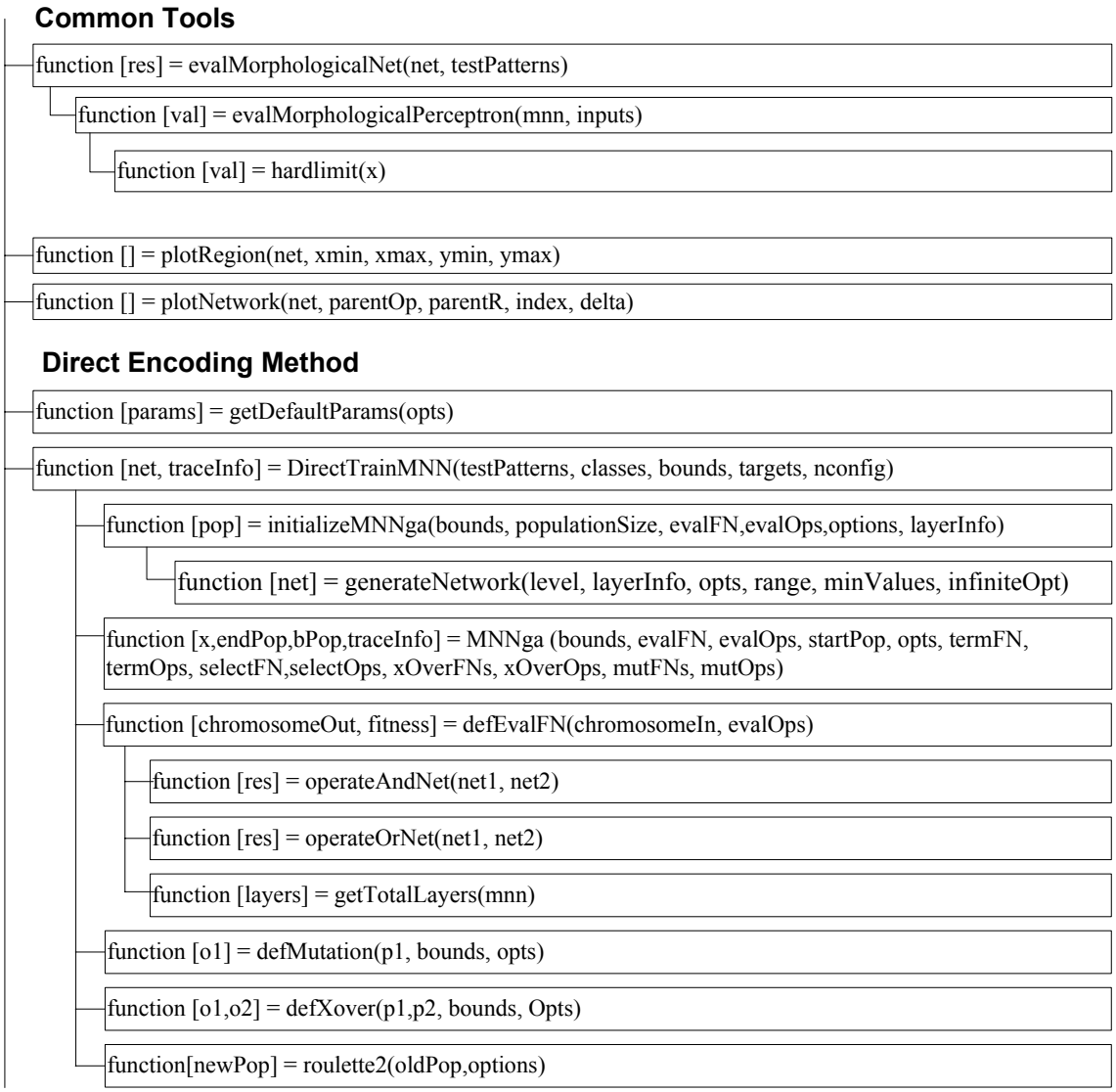


Figure A.5 Graphical representation of Multilayer Morphological Perceptrons.Tthe morphological perceptrons are represented by two intersecting perperdicular dotted lines.

A.4 DEPENDENCY STRUCTURE OF METHOD IN THE TOOLBOX

This section describes the dependency structure followed by the different methods implemented in the training algorithms toolbox for Matlab. All functions are enclosed by a rectangle and linked to other sub-functions needed to complete the required task, as shown in Figure A.6


Common Tools
function [res] = evalMorphologicalNet(net, testPatterns)
function [val] = evalMorphologicalPerceptron(mnn, inputs)
function [val] = hardlimit(x)
function [] = plotRegion(net, xmin, xmax, ymin, ymax)
function [] = plotNetwork(net, parentOp, parentR, index, delta)
Direct Encoding Method
function [params] = getDefaultParams(opts)
function [net, traceInfo] = DirectTrainMNN(testPatterns, classes, bounds, targets, nconfig)
function [pop] = initializeMNNga(bounds, populationSize, evalFN,evalOps,options, layerInfo)
function [net] = generateNetwork(level, layerInfo, opts, range, minValues, infiniteOpt)
function [x,endPop,bPop,traceInfo] = MNNga (bounds, evalFN, evalOps, startPop, opts, termFN, termOps, selectFN,selectOps, xOverFNs, xOverOps, mutFNs, mutOps)
function [chromosomeOut, fitness] = defEvalFN(chromosomeIn, evalOps)
function [res] = operateAndNet(net1, net2)
function [res] = operateOrNet(net1, net2)
function [layers] = getTotalLayers(mnn)
function [o1] = defMutation(p1, bounds, opts)
function [o1,o2] = defXover(p1,p2, bounds, Opts)
function[newPop] = roulette2(oldPop,options)

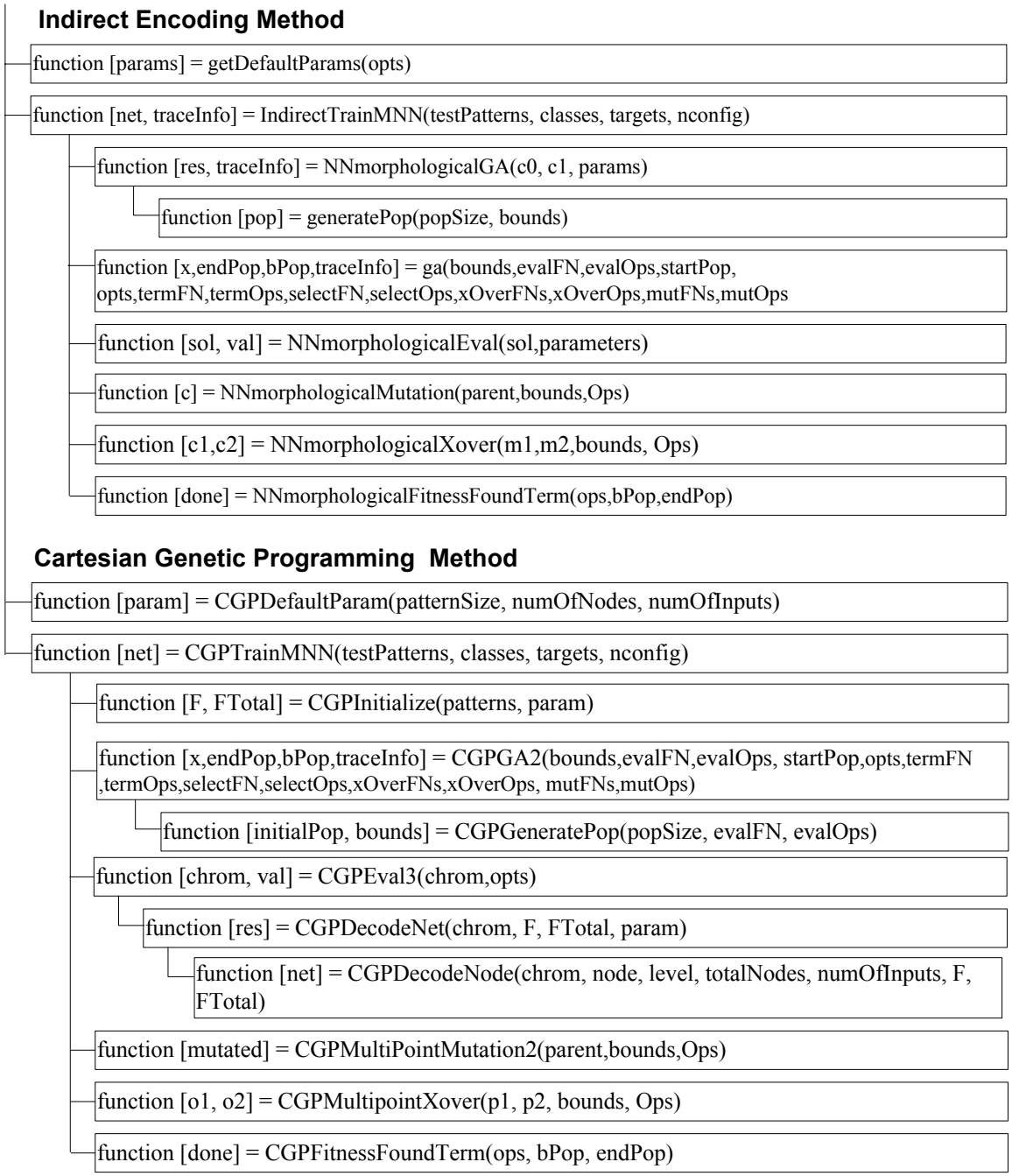


Figure A.6 Interdependency of functions for th e Matlab toolbox.

A.5 MATLAB TOOLBOX FOR MORPHOLOGICAL PERCEPTRON

This section provides all the commonly used functions defined for all the learning methods:

function [res] = evalMorphologicalNet(net, testPatterns)

```
% Version 1.0
%
function [res] = evalMorphologicalNet(net, testPatterns)
res = [];
sz= size(net,2);
for i=1:sz
   res = [res evalMorphologicalPerceptron(net(i), testPatterns)];
end
```

function [val] = evalMorphologicalPerceptron(mnn, inputs)

```
% Version 1.1
% Evaluate the morphological perceptron
% mnn - takes a morphological neural network
% inputs - a matrix containing the input patterns
function [val] = evalMorphologicalPerceptron(mnn, inputs)

numberOfInputs = size(mnn.inputs,2);
val = 0;
if (numberOfInputs ~= 0)
   res = [];
   for i=1:numberOfInputs            % Evaluate branches of the tree recursively
      res = [res evalMorphologicalPerceptron(mnn.inputs(1,i),inputs)];
   end

   %Evaluate the resulting outputs of the neurons
   totalTestPatterns = size(inputs,1);
   weights = [];
   r = [];
   for n=1:totalTestPatterns
      weights = [weights; mnn.weights];
      r       = [r;       mnn.r];
   end
   if (mnn.op == 0)
      val = hardlimit(min((r.*(res-weights))')');
   else
      val = hardlimit(max((r.*(res-weights))')');
   end
else
   % Evaluate a single neuron
   totalTestPatterns = size(inputs,1);
   weights = [];
   r = [];
   for n=1:totalTestPatterns
      weights = [weights; mnn.weights];
      r       = [r;       mnn.r];
   end
   if (mnn.op == 0)
      val = hardlimit(min((r.*(inputs-weights))')');
   else
      val = hardlimit(max((r.*(inputs-weights))')');
   end
end
```

function [val] = hardlimit(x)

```
% Version 1.1
% hardLimit used in Morphological Neural Network
% x -
```

```
function [val] = hardlimit(x)

val = x>0;
```

function [] = plotNetwork(net, parentOp, parentR, index, delta)

```
% Version
% Draw a 2D/3D representation of a neural network
function [] = plotNetwork(net, parentOp, parentR, index, delta)

if (nargin < 5)
   delta = 1;
end
if (nargin < 4)
   index = 0;
end

%set up default values
color =['b', 'g', 'r', 'c', 'm', 'y', 'k'];
symbol = ['o', 'x', '+','*', 's', 'd', 'v', '^','<','>','p','h', '.'];
connection = ['-', ':', '-.', '--'];
%  -  / r=+1, op=0
%  :  / r=+1, op=1
% -.  / r=-1, op=0
% --  / r=-1, op=1
   hold on;
   sz = size(net.inputs,2);
   if (sz == 0)

      deltas = delta*(-net.r);

      con = parentOp+1;
      if (parentR ~= 1)
         con = con+2;
      end
      if (con == 1)
         c = '-';
      elseif (con ==2)
         c = ':';
      elseif (con == 3)
         c = '-.';
      elseif (con == 4)
         c = '--';
      end

      if (index > 13)
         index = 1;
      end
      if (net.op == 1)
         color1 = strcat('b', symbol(index), c);
      else
         color1 = strcat('r', symbol(index), c);  % or o minimo
      end

      totalWeights = size(net.weights,2);
      x = net.weights(1);
      y = net.weights(2);
      %Check how many dimension contains the pattern and plot it
      if (totalWeights == 2)
         plot([x,x,x+1.2*deltas(1)],[y+1.2*deltas(2),y,y], color1);
      elseif (totalWeights == 3)
         z = net.weights(3);
         plot3([x,x,x+1.2*deltas(1)],[y+1.2*deltas(2),y,y], [z,z,z], color1);
         plot3([x,x], [y,y], [z,z+1.2*deltas(3)], color1);
      end
   else
      if (index == 0)
```

```
        for i=1:sz
            plotNetwork2(net.inputs(i), net.op, net.r(i), i);
        end
    else
        for i=1:sz
            plotNetwork2(net.inputs(i), net.op, net.r(i), index);
        end
    end
end
```

function [] = plotRegion(net, xmin, xmax, ymin, ymax)

```
function [] = plotRegion(net, xmin, xmax, ymin, ymax)

if (nargin < 4)
    xmin = -2;
    xmax = 4;
    ymin = -2;
    ymax = 4;
end

class = [];
for n=1:2000
    x = rand*(xmax-xmin)+xmin;
    y = rand*(ymax-ymin)+ymin;
    class = [class; x y];
end
r = evalMorphologicalNet(net, class);
c0 = find(r==0);
c1 = find(r==1);
hold on;
plot(class(c0,1),class(c0,2), 'ro');
plot(class(c1,1),class(c1,2), 'bs');
```

A.6 DIRECT ENCODING METHOD

This section provides all the necessary functions used by the Direct Encoding Method.

function [net, traceInfo] = DirectTrainMNN(testPatterns, classes, bounds, targets, nconfig)

```
% Version 1.1
% Train a Morphological Neural Network
% testPatterns - a M by N matrix, it contains M patterns of dimension N
% classes      - a M by 2 matrix where M is the number of classes.
%              Each element in the first column is the number of test patterns that belongs to the
%                class at the corresponding index
%              The second row contains the dimension of each test pattern class
%              NOTE: all the test patterns must contains be of the same dimension
% targets      an matrix
% networkConf   see getDefaultConfig()

function [net, traceInfo] = DirectTrainMNN(testPatterns, classes, bounds, targets, nconfig)
global class0;
```

```
global class1;
global config;

config = nconfig;
% Validate the inputs
sz = size(testPatterns);
if (sz(1) < 2)
   error('\nERROR: Insuficient number of \"testPatterns\". At least two test patterns are needed\n');
end
if (sz(2) < 2)
   error('\nERROR: Invalid dimension of the \"testPatterns\". The minimum dimenison should be 2\n');
end
[numberOfOutputs, totalClassesTargets] = size(targets);

if (numberOfOutputs < 2)
   error(sprintf('\nERROR: Insuficient number of outputs specified in \"targets\". At least two outputs are
required.\n\tCurrent value: %d',numberOfOutputs));
end

% Verify the number of classes must be less than or equal to 2*(number of outputs)
totalClasses = size(classes,1);
if (totalClasses > numberOfOutputs*2)
   error(sprintf('\nERROR: The number of outputs defined in the \"targets\" parameter must be %d',(totalClasses+1)/2));
end
if (2*totalClassesTargets < 2)
   error(sprintf('\nERROR: Insuficient number of classes in the target definition. \n\tTAt least two classes are
required.\n\tCurrent value: %d',totalClassesTargets));
end
if (totalClasses < 2)
   error(sprintf('\nERROR: Insuficient number of classes in the \"classes\" definition.\n\tAt least two classes are
required.\n\tCurrent value: %d',totalClasses));
end

% Sum all the elements in the class
pos = [cumsum(classes(:,1))];
% Add append a 0 value at the begining of the vector and remove the last one.
pos = [0; pos(1:end-1,1)]+1;

% Initialize the resulting net to null
net = [];

% Add the number of inputs to the layer information
config.layers = [size(testPatterns,2) config.layers];

% Compute the bounds for each dimension
maxVals = bounds(2,:);
minVals = bounds(1,:);
bounds = [(maxVals-minVals); minVals];


% For each output, define the network
net = struct('op', cell(1,totalClassesTargets), 'r', {[1]}, 'weights', {[1]}, 'inputs', {[]});
traceInfo = struct('trace', {[]});
for output=1:totalClassesTargets

   % Regroup the test patterns
   fprintf('  Training output: %d\n',output);
   groups = find(targets(:,output)==0);
   totalGroups = size(groups,1);
   class0 = [];

   % Verify the class with the output value of 0 at the output index(output) exist!.
   if (totalGroups == 0)
      error('ERROR: Invalid class definition. The target with 0 on evey index must be defined');
   end;
   for class=1:totalGroups
      begn = pos(groups(class));
      final = begn+classes(groups(class))-1;
      class0 = [class0; testPatterns(begn:final,:)];
   end
```

```
    groups = find(targets(:,output)==1);
    totalGroups = size(groups,1);

    class1 = [];
    for class=1:totalGroups
        begn = pos(groups(class));
        final = begn+classes(groups(class))-1;
        class1 = [class1; testPatterns(begn:final,:)];
    end
    % Generate the initial population
    pop = initializeMNNga(bounds, config.popSize, config.evalFn, config.evalParams, ...
        [config.variableArchitecture config.allowInfinite], config.layers);
    gaFN =  ['[x, endPop, bPop, traceInf] = ' config.defaultGA '(bounds, config.evalFn, config.evalParams, pop, config.opts,'
...
        'config.termFn, config.termParams, config.selectFn, config.selectParams,' ...
        'config.xOverFn, config.xOverParams,config.mutFn, config.mutParams);'];

    eval(gaFN);

    traceInfo(output).trace = traceInf;
    net(output) = x.chromosome;
end
```

function [net] = generateNetwork(level, layerInfo, opts, range, minValues, infiniteOpt)

```
% Generate a random neuron based in the configuration
% level  - indicates the layer+1 of the network
% layerInfo = [num_of_inputs  num_of_neurons_1rst_layer  num_of_neurons_2nd_layer(optional)
num_of_neurons_3thrd_layer(optional)]
% opts      = [variableArchitecture]
% infiniteOpt = [allowInfinite inifiniteOps]

function [net] = generateNetwork(level, layerInfo, opts, range, minValues, infiniteOpt)

if nargin < 3
    error('ERRORO: Invalid number of arguments in generateNetwork\n');
end
allowInfinite = 0;
if (nargin >= 4)
    if (size(infiniteOpt,2) >= 2)
        [allowInfinite infiniteProb] = infiniteOpt;
    end
end

totalOpts = size(opts,2);
if (totalOpts > 1)
    variableArchitecture = 0;
else
    variableArchitecture = opts(1);                    % Is the architecture fixed or variable?
end

if (variableArchitecture)

else
    if (level == 4)                                % level 2?
        neuron.op = 0;
        neuron.r = ones(1,layerInfo(level-1));
        neuron.weights = zeros(1,layerInfo(level-1));
        neuron.inputs = [];
        for i = 1:layerInfo(level-1)
            [mnn] = generateNetwork(level-1,layerInfo,opts, range, minValues, infiniteOpt);
            neuron.inputs = [neuron.inputs mnn];
        end;
```

```
   elseif (level == 3)                          % level 2?
      neuron.op = round(rand);
%       neuron.p = [1];
      neuron.r = ones(1,layerInfo(level-1));          % what happend if r = -1?
      neuron.weights = zeros(1,layerInfo(level-1));
      neuron.inputs = [];
      for i = 1:layerInfo(level-1)
         [mnn] = generateNetwork(level-1,layerInfo,opts, range, minValues, infiniteOpt);
         neuron.inputs = [neuron.inputs mnn];
      end;
   elseif (level == 2)                          % level 2?
      neuron.op = round(rand);                     % Generate random [max min] operator
%       neuron.p = round(rand);
      neuron.r = 2*round(rand(1,layerInfo(level-1)))-1; % Generate a vector of [-1 1] values

      if (allowInfinite)
         if (rand < infiniteProb)
         end
      else
         neuron.weights = ((rand(1,layerInfo(level-1))).*range)+minValues;       % Generate weights
      end
      neuron.inputs = [];
   end
end
net = neuron;
```

function [pop] = initializeMNNga(bounds, populationSize, evalFN,evalOps,options, layerInfo)

```
% Version 1.0
function [pop] = initializeMNNga(bounds, populationSize, evalFN,evalOps,options, layerInfo)
global class0;
global class1;

% options(1) 0 - fixed architecture
%            1 - variable architecture
% options(2) 0 dont allow -Inf and +Inf
%            1 - allow -Inf and +Inf
% options(3) inf probability


%Validate parameters
pop = [];

if nargin<6
   error('ERROR: Missing layer configuration options in parameters.');
   layerInfo = [2 1];
else
   % validate layerInfo
   totalLayers = size(layerInfo,2)-1;
   totalInputs = layerInfo(1);
   if (totalLayers > 3)
      error('Invalid number of layers\n');
   end
   if (size(find(layerInfo<=0),2) > 0)
      error('Invalid layer configuration. None of the parameters can be 0');
   end
   if (totalLayers == 3)
      if (layerInfo(4) > 1 )
         error('Invalid number of neurons in layer 3');
      end
      if (layerInfo(3) < 2)
         error('Invalid number of neurons in layer 2');
      end
      if (layerInfo(2) < 2)
         error('Invalid number of neurons in layer 1');
```

```
        end
    end
    if (totalLayers == 2)
        if (layerInfo(3) > 1)
            error('Invalid number of neurons in layer 2');
        end
        if (layerInfo(2) < 2)
            error('Invalid number of neurons in layer 2');
        end
    end
    if (totalLayers == 1)
        if (layerInfo(2) > 1)
            error('Invalid number of neurons in layer 2');
        end
    end

end
if nargin<5
%  options=[1e-6 1];
   options=[1 0 0];
end
if nargin<4
  evalOps=[];
end

if any(evalFN<48) %Not a .m file
  estr=['x=pop(i).chromosome; pop(i).fitness=', evalFN ';'];
else %A .m file
  estr=['[ pop(i).chromosome pop(i).fitness]=' evalFN '(pop(i).chromosome,[0 evalOps]);'];
end


% Generate random population

pop = struct('fitness', cell(1,populationSize), 'chromosome', {[]});

for i=1:populationSize
    neuron.fitness = 0;
    neuron.chromosome = generateNetwork(size(layerInfo,2), layerInfo, options(1), bounds(1,:), bounds(2,:),
[options(2:end)]);
%     if (isEmptyArea(neuron.chromosome))
%         neuron.chromosome.op = 0;
%     end

    pop(i) = neuron;
    eval(estr);
end

return;
```

function [x,endPop,bPop,traceInfo] = MNNga (bounds, evalFN, evalOps, startPop, opts, termFN, termOps, selectFN,selectOps, xOverFNs, xOverOps, mutFNs, mutOps)

```
% Version 1.0
% MNNga
function [x,endPop,bPop,traceInfo] = MNNga(bounds,evalFN,evalOps,startPop,opts,...
termFN,termOps,selectFN,selectOps,xOverFNs,xOverOps,mutFNs,mutOps)
global class0;
global class1;
global config;
%global range;
%global minValues;
% GA run a genetic algorithm
% function [x,endPop,bPop,traceInfo]=ga(bounds,evalFN,evalOps,startPop,opts,
```

```
%                               termFN,termOps,selectFN,selectOps,
%                               xOverFNs,xOverOps,mutFNs,mutOps)
%
% Output Arguments:
%   x           - the best solution found during the course of the run
%   endPop      - the final population
%   bPop        - a trace of the best population
%   traceInfo   - a matrix of best and means of the ga for each generation
%
% Input Arguments:
%   bounds      - a matrix of upper and lower bounds on the variables
%   evalFN      - the name of the evaluation .m function
%   evalOps     - options to pass to the evaluation function ([NULL])
%   startPop    - a matrix of solutions that can be initialized
%                 from initialize.m
%   opts        - [epsilon prob_ops display] change required to consider two
%                 solutions different, prob_ops 0 if you want to apply the
%                 genetic operators probabilisticly to each solution, 1 if
%                 you are supplying a deterministic number of operator
%                 applications and display is 1 to output progress 0 for
%                 quiet. ([1e-6 1 0])
%   termFN      - name of the .m termination function (['maxGenTerm'])
%   termOps     - options string to be passed to the termination function
%                 ([100]).
%   selectFN    - name of the .m selection function (['normGeomSelect'])
%   selectOpts  - options string to be passed to select after
%                 select(pop,#,opts) ([0.08])
%   xOverFNS    - a string containing blank seperated names of Xover.m
%                 files (['arithXover heuristicXover simpleXover'])
%   xOverOps    - A matrix of options to pass to Xover.m files with the
%                 first column being the number of that xOver to perform
%                 similiarly for mutation ([2 0;2 3;2 0])
%   mutFNs      - a string containing blank seperated names of mutation.m
%                 files (['boundaryMutation multiNonUnifMutation ...
%                       nonUnifMutation unifMutation'])
%   mutOps      - A matrix of options to pass to Xover.m files with the
%                 first column being the number of that xOver to perform
%                 similiarly for mutation ([4 0 0;6 100 3;4 100 3;4 0 0])


if (0)          % For debuging pourpose, change it to 1 to validate parameters
   n=nargin;
   if n<13
     disp('Insufficient arguements')
   end
   if n<3 %Default evalation opts.
     evalOps=[];
   end
   if n<5
     opts = [1e-6 1 0];
   end
   if isempty(opts)
     opts = [1e-6 1 0];
   end

   if n<12 %Default muatation information
       mutFNs=['defMutation'];
       mutOps=[size(startPop,2) 1 1 1 1];
   end
   if n<10 %Default crossover information
      xOverFNs=['defXover'];
      xOverOps=[size(startPop,2) 0.6];

   end
   if n<9 %Default select opts only i.e. roullete wheel.
     selectOps=[];
   end
   if n<8 %Default select info
     selectFN=['normGeomSelect'];
     selectOps=[0.08];
```

```
    end
    if n<6 %Default termination information
      termOps=[100];
      termFN='maxGenTerm';
    end
    if n<4 %No starting population passed given
      startPop=[];
    end
    if isempty(startPop) %Generate a population at random
      %startPop=zeros(80,size(bounds,1)+1);
    %  startPop=initializega(80,bounds,evalFN,evalOps,opts(1:2));
       fprintf('ERROR: Empty initial population\n');
       return;
    end
end

if any(evalFN<48) %Not using a .m file
    e1str=['x=c1; c1.fitness=', evalFN ';'];
    e2str=['x=c2; c2.fitness=', evalFN ';'];
else %Are using a .m file
    e1str=['[c1.chromosome c1.fitness]=' evalFN '(c1.chromosome, [gen evalOps]);'];
    e2str=['[c2.chromosome c2.fitness]=' evalFN '(c2.chromosome, [gen evalOps]);'];
end

%minValues = bounds(:,1)';
%range = (bounds(:,2)-bounds(:,1))';
if (config.dbg.plotEvolution)
    if (config.dbg.fixedAxis)
        close all;
        figure(1);
        cla;
        if (size(bounds,2) == 2)
            a = [bounds(2,1), bounds(1,1)+bounds(2,1), bounds(2,2), bounds(1,2)+bounds(2,2)];
        elseif (size(bounds,2) == 3)
            a = [bounds(2,1), bounds(1,1)+bounds(2,1), bounds(2,2), bounds(1,2)+bounds(2,2) , bounds(2,3),
bounds(1,3)+bounds(2,3)];
        end
        axis(a);
        axis square;
        axis manual;
%       box;
%       grid;

        if (config.dbg.showTheBest)
            figure(2);
            cla;
            axis(a);
            axis square;
            axis manual;
%           box;
%           grid;
        end
    else
        close all;
    end
end


popSize      = size(startPop,2);                %Number of individuals in the pop
%endPop       = zeros(popSize,xZomeLength); %A secondary population matrix
endPop       = struct('fitness', cell(1,popSize), 'chromosome', {[]});

c1.fitness   = 0;
c1.chromosome= [];
c2.fitness   = 0;
c2.chromosome= [];
epsilon      = opts(1);                %Threshold for two fittness to differ
oval         = max([startPop.fitness]); %Best value in start pop
bFoundIn     = 1;                                        %Number of times best has changed
done         = 0;                    %Done with simulated evolution
```

```
gen         = 1;                                        %Current Generation Number
collectTrace = (nargout>3);                    %Should we collect info every gen
floatGA      = opts(2)==1;            %Probabilistic application of ops
display      = opts(3);               %Display progress

subPopSize = 2*fix(selectOps(1)*popSize/2);

while(~done)
  %Elitist Model
  [bval,bindx] = max([startPop.fitness]); %Best of current pop
  best =  startPop(bindx);

  if collectTrace
    traceInfo(gen,1) = gen;                                %current generation
    traceInfo(gen,2) = startPop(bindx).fitness;          %Best fittness
    traceInfo(gen,3) = mean([startPop.fitness]);         %Avg fittness
    traceInfo(gen,4) = std([startPop.fitness]);
  end

  if ( (abs(bval - oval)>epsilon) | (gen==1)) %If we have a new best sol
    if display
      fprintf(1,'\n%d %f\n',gen,bval);          %Update the display
    end
    stat.generation = gen;
    stat.organism = startPop(bindx);
    bPop(bFoundIn)=[stat]; %Update bPop Matrix

    bFoundIn=bFoundIn+1;                    %Update number of changes
    oval=bval;                            %Update the best val
  else
    if display
      fprintf(1,'%d ',gen);                       %Otherwise just update num gen
    end
  end

  endPop = feval(selectFN,startPop,[gen selectOps]); %Select
  totalOrg = subPopSize+1;
  totalFitness = sum([endPop.fitness]);
  fit = [endPop.fitness]/totalFitness;
  fit = cumsum(fit);


    mutationParams = ((1.0-(0.4).*(startPop(bindx).fitness))).* mutOps(1,:);

    mutationParams = [gen, startPop(bindx).fitness, mutationParams];

    while totalOrg < popSize

        val1 = find(fit-rand>=0);
        val2 = find(fit-rand>=0);
              [c1.chromosome c2.chromosome] =
feval(xOverFNs,endPop(val1(1)).chromosome,endPop(val2(1)).chromosome,bounds,[gen xOverOps(1,:)]);

                  c1.chromosome = feval(mutFNs,c1.chromosome,bounds,mutationParams);
                  c2.chromosome = feval(mutFNs,c2.chromosome,bounds,mutationParams);

        eval(e1str);
        eval(e2str);


              endPop(totalOrg)=c1;
              endPop(totalOrg+1)=c2;
        totalOrg = totalOrg+2;
    end

    % maxGen = termOps(1);


    %((maxGen-gen)/maxGen)
    %mutationParams = (3*(1.1-startPop(bindx).fitness)).* mutOps(i,:);
```

```
%    mutationParams = [gen, startPop(bindx).fitness, mutOps(i,:)];

  gen=gen+1;
  done=feval(termFN,[gen termOps],bPop,endPop); %See if the ga is done
  startPop=endPop;                              %Swap the populations

  [bval,bindx] = min([startPop.fitness]); %Keep the best solution
  startPop(bindx) = best;                       %replace it with the worst

  if (config.dbg.plotEvolution)                % Plot the evolution of the population
     if (config.dbg.showTheBest)                % Plot the best organism
        figure(2);
        cla;
        hold on;
        if (size(class0,2) == 2)
           plot(class0(:,1),class0(:,2),'ks');
           plot(class1(:,1),class1(:,2),'go');
        elseif (size(class0,2) == 3)
           plot3(class0(:,1),class0(:,2),class0(:,3),'ks');
           plot3(class1(:,1),class1(:,2),class1(:,3),'go');
        end
        plotNetwork(best.chromosome);
     end
     figure(1);
     cla;
     hold on;

     if (size(class0,2) == 2)
        plot(class0(:,1),class0(:,2),'ks');
        plot(class1(:,1),class1(:,2),'go');
     elseif (size(class0,2) == 3)
        plot3(class0(:,1),class0(:,2),class0(:,3),'ks');
        plot3(class1(:,1),class1(:,2),class1(:,3),'go');
     end

     sz = size(startPop,2);
     for n=1:sz
        plotNetwork(startPop(n).chromosome);          % plot a single network
     end

     if (config.dbg.delay > 0)
        pause(config.dbg.delay);                      % Delay the output
     end
  end
end

[bval,bindx] = max([startPop.fitness]);
if display
  fprintf(1,'\n%d %f\n',gen,bval);
end

x=startPop(bindx);
stat.generation = gen;
stat.organism = startPop(bindx);
bPop(bFoundIn)=stat;


if collectTrace
  traceInfo(gen,1)=gen;                         %current generation
  traceInfo(gen,2)=startPop(bindx).fitness; %Best fittness
  traceInfo(gen,3)=mean([startPop.fitness]); %Avg fittness
  traceInfo(gen,4)=std([startPop.fitness]);
end
```

function [chromosomeOut, fitness] = defEvalFN(chromosomeIn, evalOps)

```
% Version 1.0
% Fitness Function
% Inputs-
%   chromosomeIn - chromosome to be evaluated
%   evalOps -
% Outputs:
%   chromosomeOut - (must be the same as the input)
%   fitness  = how good is the organism
%   fitness = (1/N)(total patterns classified correctly)/total Test Patterns

function [chromosomeOut, fitness] = defEvalFN(chromosomeIn, evalOps)
global class0;
global class1;


chromosomeOut = chromosomeIn;
% Evaluate patterns of class0
evalClass0 = evalMorphologicalPerceptron(chromosomeIn, class0);
totalCorrectClass0 = size(find(evalClass0==0),1)/size(evalClass0,1);

% Evaluate patterns of class1
evalClass1 = evalMorphologicalPerceptron(chromosomeIn, class1);
totalCorrectClass1 = size(find(evalClass1==1),1)/size(evalClass1,1);

%totalTestPatterns = size(class0,1)+size(class1,1);
fitness = (totalCorrectClass0+totalCorrectClass1)/2;

if (getTotalLayers(chromosomeIn)> 1)

   totalNeurons = size(chromosomeIn.inputs,2);
   cumSum = 0;
   if (chromosomeIn.op == 0)
      total = 0;
      for n=1:totalNeurons
         for m=1:totalNeurons
            if (n ~= m)
               total = total+operateOrNet(chromosomeIn.inputs(n),chromosomeIn.inputs(m));
               cumSum = cumSum +1;
            end
         end
      end
   else
      total = 0;
      for n=1:totalNeurons
         for m=1:totalNeurons
            if (n ~= m)
               total = total+operateAndNet(chromosomeIn.inputs(n),chromosomeIn.inputs(m));
               cumSum = cumSum +1;
            end
         end
      end
   end
   fitness = (fitness+2*(cumSum-total)/cumSum)/3;
end
```

function [o1] = defMutation(p1, bounds, opts)

```
% Version 1.0
% opts = [gen, bestFitness, totalMut, mutProb, mutOpProb, mutWeightProb,mutRProbm mutRange]
function [o1] = defMutation(p1, bounds, opts)
global class0;
global class1;
```

```
o1 = p1;
if (size(opts) ~= 8)
   error('Invalid number of parameters');
end
bestFitness = opts(2);
mutProb  = opts(4);
mutOpProb = opts(5);
mutWeightProb = opts(6);
mutRProb = opts(7);
mutRange = opts(8);

%Adjust mutation parameters according to the fitness
if (bestFitness >= 0.96)
   mutProb = mutProb*.05;
   mutOpProb = mutOpProb*.05;
   mutRProb = mutRProb*0.05;
   mutRange = mutRange*0.10;
end

%if (rand > mutProb)
%    return;c
%end


totalLayers = getTotalLayers(p1);
if     (totalLayers == 1)
   if (rand < mutOpProb)
      o1.op = abs(p1.op-1);
   end


   if (rand < mutWeightProb)
      rangeMin = bounds(1,:)*mutRange;
      rangeMax = rangeMin;

      sz = size(o1.weights,2);
%       o1.weights = o1.weights+(bounds(1,:).*(0.5*(rand(1,sz)-0.25)));
      rangeMax = min(p1.weights+rangeMax , (bounds(1,:)+bounds(2,:)))-p1.weights;
      rangeMin = max(p1.weights-rangeMin,bounds(2,:))-p1.weights;
      mutation = (rangeMax-rangeMin).*rand(1,sz)+rangeMin;

      o1.weights = p1.weights+mutation;
   end

%    if (rand < mutRProb)
      sz = size(o1.r,2);
%       o1.r = 2*round(rand(1,sz))-1;
      o1.r = o1.r .*(2*(rand(1,sz)<mutRProb)-1);
%    end

elseif (totalLayers == 2)
   % To be implemented
   level = round(rand*0.80+0.20)+1;
   if (level == 1)
%       pos11 = round(rand*(size(o1.inputs,2)-1))+1;
%       if (rand < mutOpProb)
         o1.op = abs(p1.op-1);
%          o1.op = rand*.6;
%       end
   elseif (level == 2)
      sz = size(o1.inputs,2);
      totalMutBranches = round(rand*(sz-1))+1;

      totalInputs1 = size(o1.inputs,2);
      for n=1:totalMutBranches
         pos11 = round(rand*(totalInputs1-1))+1;

         if (rand < mutOpProb)
            o1.inputs(pos11).op = abs(p1.inputs(pos11).op-1);
```

```
        end

        if (rand < mutWeightProb)
            rangeMin = bounds(1,:)*mutRange;
            rangeMax = rangeMin;

            sz = size(p1.inputs(pos11).weights,2);

            rangeMax = min(p1.inputs(pos11).weights+rangeMax, (bounds(1,:)+bounds(2,:)))-p1.inputs(pos11).weights;
            rangeMin = max(p1.inputs(pos11).weights-rangeMin,  bounds(2,:))-p1.inputs(pos11).weights;
            mutation = (rangeMax-rangeMin).*rand(1,sz)+rangeMin;                    % this should be optimized


            o1.inputs(pos11).weights = p1.inputs(pos11).weights+mutation;
        end

        if (rand < mutRProb)
            sz = size(o1.inputs(pos11).r,2);
            o1.inputs(pos11).r = 2*round(rand(1,sz))-1;
        end
    end
  end
elseif (totalLayers == 3)
  level = round(rand)+2;

  if (level == 3)
    pos11 = round(rand*(size(o1.inputs,2)-1))+1;
    pos12 = round(rand*(size(o1.inputs(pos11).inputs,2)-1))+1;

    if (rand < mutOpProb)
      o1.inputs(pos11).inputs(pos12).op = round(rand);
    end

    if (rand < mutWeightProb)
      sz = size(o1.inputs(pos11).inputs(pos12).weights,2);
      o1.inputs(pos11).inputs(pos12).weights = o1.inputs(pos11).inputs(pos12).weights+(bounds(1,:).*(0.5*(rand(1,sz)-
0.25)));
    end

    sz = size(o1.inputs(pos11).inputs(pos12).r,2);
    for i=1:sz
      if (rand < mutRProb)
        o1.inputs(pos11).inputs(pos12).r(i) = 2*round(rand)-1; % Generate a vector of [-1 1] values
      end
    end
  else
    pos11 = round(rand*(size(o1.inputs,2)-1))+1;
    if (rand < mutOpProb)
      o1.inputs(pos11).op = round(rand);
    end

  end
%   if (isEmptyArea(o1))
%       o1.op = 0;
%   end
end;
```

function [o1,o2] = defXover(p1,p2, bounds, Opts)

```
% Version 2.0
% Crossover function
%
function [o1,o2] = defXover(p1,p2, bounds, Opts)
global class0;
global class1;
```

```
global config;
if (config.dbg.showCrossover)
   figure(10);
   cla;
   plot(class0(:,1),class0(:,2),'g*');
   plot(class1(:,1),class1(:,2),'kx');
   plotNetwork(p1);

   figure(11);
   cla;
   plot(class0(:,1),class0(:,2),'g*');
   plot(class1(:,1),class1(:,2),'kx');
   plotNetwork(p2);
end
o1 = p1;
o2 = p2;
totalLayers = min(getTotalLayers(p1), getTotalLayers(p2));


if (totalLayers == 1)
      % arithmetic crossover from matlab toolbox

   a = rand;

   o1.op       = round (p1.op*a+p2.op*(1-a));
   o1.r        = 2*((p1.r*(a)+(p2.r*(1-a)))>=0)-1;
   o1.weights  = p1.weights*a+p2.weights*(1-a);
   o1.inputs   = p1.inputs;

   o2.op       = round (p2.op*a+p1.op*(1-a));
   o2.r        = 2*((p2.r*(a)+(p1.r*(1-a)))>=0)-1;
   o2.weights  = p2.weights*a+p1.weights*(1-a);
   o2.inputs   = p2.inputs;
elseif (totalLayers == 2)
   cutPoint = round(rand+1);

   if (cutPoint == 1)
%      fprintf('\tcutPoint: 1 \t');
%      if (rand < 0.25)
         rnd = round(rand*2);
         rnd = bitor(rnd, 2^round(rand));

%          fprintf('cross: [');
         if (bitand(rnd, 1))                          %cross the operation
%             fprintf('op');
            a = rand;
            o1.op       = round (p1.op*a+p2.op*(1-a));
            o2.op       = round (p2.op*a+p1.op*(1-a));
         end

         if (bitand(rnd, 2))                          % cross a branch
            %totalBranches = round(rand*(size(p1.inputs,2)-1))+1;
            totalBranches = size(p1.inputs,2);
            cnt = 1;
            branchInfo = round(rand*(2^totalBranches-1));
            for n=1:totalBranches
               if (bitand(branchInfo, cnt))
%                  fprintf(' swap-branch');
                  % Find a neuron in the first parent
                  pos11 = n;
                  pos21 = n;
                  %sz = size(p1.inputs,2);
                  %pos11 = round(rand*(sz-1))+1;

                  % Find the neuron in the second parent
                  %sz = size(p2.inputs,2);
                  %pos21 = round(rand*(sz-1))+1;

                  neuron = p1.inputs(pos11);
                  o1.inputs(pos11) = p2.inputs(pos21);
```

```
                o2.inputs(pos21) = neuron;
            end
            cnt = cnt*2;
        end
    end
%       fprintf('] ');

elseif (cutPoint == 2)
    totalNeurons = size(p1.inputs,2);

%    fprintf('\tcutPoint: 2 \t totalNeurons: %d', totalNeurons);


    branchInfo = round(rand*(2^totalNeurons-1));
    cnt = 1;
    for n=1:totalNeurons

        if (bitand(branchInfo, cnt))
        % Find the neuron in the first parent
        pos11 = n;
        pos21 = n;
        %sz = size(p1.inputs,2);
        %pos11 = round(rand*(sz-1))+1;
        % neuron1 = mnn1.inputs(pos11);

        % Find the neuron in the second parent
        %sz = size(p2.inputs,2);
        %pos21 = round(rand*(sz-1))+1;
        %   neuron2 = mnn2.inputs(pos21);

        rnd = round(rand*3);
        rnd = bitor(rnd, 2^(round(rand*2)));

        a = rand;

%       fprintf('\t\tcross:%d [',n);

        if (bitand(rnd, 1))
 %          fprintf('op');
            o1.inputs(pos11).op       = round (p1.inputs(pos11).op*a+p2.inputs(pos21).op*(1-a));
            o2.inputs(pos21).op       = round (p2.inputs(pos21).op*a+p1.inputs(pos11).op*(1-a));
        end

        if (bitand(rnd, 2))
%           fprintf(' weights');
            o1.inputs(pos11).weights   = p1.inputs(pos11).weights*a+p2.inputs(pos21).weights*(1-a);
            o2.inputs(pos21).weights   = p2.inputs(pos21).weights*a+p1.inputs(pos11).weights*(1-a);
        end
        if (bitand(rnd, 4))
%           fprintf(' r');
            sz = size(p1.inputs(pos11).r,2);
            aa = rand(1,sz);
            o1.inputs(pos11).r        = 2*(((p1.inputs(pos11).r).*aa+((p2.inputs(pos21).r).*(1-aa)))>=0)-1;
            o2.inputs(pos21).r        = 2*(((p2.inputs(pos21).r).*aa+((p1.inputs(pos11).r).*(1-aa)))>=0)-1;
%             o1.inputs(pos11).r       = 2*((p1.inputs(pos11).r*(a)+(p2.inputs(pos21).r*(1-a)))>=0)-1;
%             o2.inputs(pos21).r       = 2*((p2.inputs(pos21).r*(a)+(p1.inputs(pos11).r*(1-a)))>=0)-1;
        end
%       fprintf('] ');
        end
        cnt = cnt*2;

    end % for

    if (isEmptyArea(o1))
%       fprintf('[fixed o1] ');
        o1.op = 0;
    end
    if (isEmptyArea(o2))
%       fprintf('[fixed o2] ');
        o2.op = 0;
```

```
        end
%       fprintf('\n');
    end  % elseif (cutPoint == 2)

elseif (totalLayers == 3)
    % Find the neuron in the first parent
    mnn1 = p1;
    sz = size(mnn1.inputs,2);
    pos11 = round(rand*(sz-1))+1;
    mnn1 = mnn1.inputs(pos11);

    sz = size(mnn1.inputs,2);
    pos12 = round(rand*(sz-1))+1;
    neuron1 = mnn1.inputs(pos12);

    % Find the neuron in the second parent
    mnn2 = p2;
    sz = size(mnn2.inputs,2);
    pos21 = round(rand*(sz-1))+1;
    mnn2 = mnn2.inputs(pos21);

    sz = size(mnn2.inputs,2);
    pos22 = round(rand*(sz-1))+1;
    neuron2 = mnn2.inputs(pos22);

    % Swap the neuron content
    o1.inputs(pos11).inputs(pos12).op = neuron2.op;
    o1.inputs(pos11).inputs(pos12).r = neuron2.r;
    o1.inputs(pos11).inputs(pos12).weights = neuron2.weights;
    o1.inputs(pos11).inputs(pos12).inputs = neuron2.inputs;

    o2.inputs(pos21).inputs(pos22).op = neuron1.op;
    o2.inputs(pos21).inputs(pos22).r = neuron1.r;
    o2.inputs(pos21).inputs(pos22).weights = neuron1.weights;
    o2.inputs(pos21).inputs(pos22).inputs = neuron1.inputs;
end

if (config.dbg.showCrossover)
    figure(20);
    cla;
    plot(class0(:,1),class0(:,2),'g*');
    plot(class1(:,1),class1(:,2),'kx');
    plotNetwork(o1);

    figure(21);
    cla;
    plot(class0(:,1),class0(:,2),'g*');
    plot(class1(:,1),class1(:,2),'kx');
    plotNetwork(o2);
    pause;
end
```

function [res] = operateAndNet(net1, net2)

```
% Version 1.0
% Determine of the hyperspace of a network is empty
function [res] = operateAndNet(net1, net2)

res = 0;
totalInputs = size(net1.inputs,2);
equal = 1;
    Sxi = net1.op*2-1;
    Syi = net2.op*2-1;
    inside = 1;
    for i=1:totalInputs
        if (net1.weights(i) > net2.weights(i))
            if ((Sxi*net1.r(i) < 0) & (Syi*net2.r(i) > 0))
```

```
            res = 1;
            return;
        end
    end
    if (net1.weights(i) < net2.weights(i))
        if ((Sxi*net1.r(i) > 0) & (Syi*net2.r(i) < 0))
            res = 1;
            return;
        end
    end

    if (net1.r(i) ~= net2.r(i))
        equal = 0;
    end
    if (equal)
        if (Sxi*net1.r(i) > 0 & net2.weights(i) < net1.weights(i))
            inside = 0;
        elseif (Sxi*net1.r(i) < 0 & net2.weights(i) > net1.weights(i))
            inside = 0;
        end
    end

end

if (~equal & net1.op == net2.op)
    res = 1;
end
if (inside)
    res = 1;
end
```

function [res] = operateOrNet(net1, net2)

```
function [res] = operateOrNet(net1, net2)

%Defermine if the hyper space of two vectors is different
totalInputs = size(net1.r,2);
%net1.r = net1.r*(net1.op*2-1);
%net2.r = net2.r*(net2.op*2-1);
if (net1.op == 0)       % or
    if (net2.op ==0)
        inside = 1;
        allSameDirection = 1;
        intersection = 0;
        for n=1:totalInputs
            if (net1.r(n) == net2.r(n))
                if (net1.r(n) < 0)
                    if (net2.weights(n) < net1.weights(n))
                        inside = 0;
                    end
                else
                    if (net2.weights(n) > net1.weights(n))
                        inside = 0;
                    end
                end
            else
                allSameDirection = 0;
                if (net1.r(n) < 0)
                    if (net2.weights(n) < net1.weights(n))
                        intersection = 1;
                    end
                else
                    if (net2.weights(n) > net1.weights(n))
                        intersection = 1;
                    end
                end
            end;
```

```
        end
        if (allSameDirection & inside)
            res = 1;
            return;
        end
        if (allSameDirection & ~inside)
            res = 0;
            return;
        end
        if (intersection)
            res = 0;
            return;
        end
        res = 1;
        return;
    else
        allInside = 1;
        allOutside = 1;
        allSameDirection = 1;
        intersection = 0;

        for n=1:totalInputs
            if (net1.r(n) == net2.r(n))
                allSameDirection = 0;
            end
            if (net1.r(n) > 0)
                if (net2.weights(n) > net1.weights(n))
                    allOutside = 0;
                else
                    allInside = 0;
                end
                if (net2.weights(n) < net1.weights(n) & net2.r(n) < 0)
                    intersection = 1;
                end
            else
                if (net2.weights(n) < net1.weights(n))
                    allOutside = 0;
                else
                    allInside = 0;
                end
                if (net2.weights(n) > net1.weights(n) & net2.r(n) > 0)
                    intersection = 1;
                end
            end
        end
        if (allInside)
            res = 0;
            return;
        end
        if (allSameDirection & allOutside)
            res = 1;
            return;
        end
        if (~allInside & ~allOutside & intersection)
            res = 0;
            return;
        end
        res = 1;
    end
elseif(net1.op == 1)        % and
    if (net2.op == 1)       % and
        inside = 1;
        allSameDirection = 1;
        for n=1:totalInputs
            if (net1.r(n) ~= net2.r(n))
                allSameDirection = 0;
            end
            if (net1.r(n) < 0)
                if (net2.weights(n) <= net1.weights(n))
                    inside = 0;
```

```
            end
        else
            if (net2.weights(n) >= net1.weights(n))
                inside = 0;
            end
        end
    end
    if (inside & allSameDirection)
        res = 1;
        return
    end
    res = 0;
    return;
  else
    inside = 1;
    for n=1:totalInputs
        if (net1.r(n) ~= net2.r(n))
            inside = 0;
        end
        if ((net1.r(n)>0) & (net1.weights(n) < net2.weights(n)))
            inside = 0;
            break;
        elseif((net1.r(n)<0) & (net1.weights(n) > net2.weights(n)))
            inside = 0;
            break;
        end
    end
    res = inside;
    return;
  end
end
```

function[newPop] = roulette2(oldPop,options)

```
function[newPop] = roulette2(oldPop,options)
%roulette is the traditional selection function with the probability of
%surviving equal to the fittness of i / sum of the fittness of all individuals
%
%function[newPop] = roulette(oldPop,options)
%newPop  - the new population selected from the oldPop
%oldPop  - the current population
%options - [gen] options

if (size(options) < 2)
   error('Incorrect options');
end
%Get the parameters of the population
totalIn = 2*fix(options(2)*size(oldPop,2)/2);

numSols = size(oldPop,2);
totalFit = sum([oldPop.fitness]);
fit = [oldPop.fitness]';
%fit = [oldPop.fitness]' / totalFit;

x = zeros(numSols,2);
x(:,1) =[numSols:-1:1]';
[y x(:,2)] = sort(fit);

totalIn = numSols-totalIn;
newIn = 1;
for n=numSols:-1:totalIn
   newPop(newIn) = oldPop(x(n,2));
   newIn = newIn+1;
end
```

function [layers] = getTotalLayers(mnn)

```
% Version 1.0
% Returns the total number of layers of a morphological neural network

function [layers] = getTotalLayers(mnn)

numberOfInputs = size(mnn.inputs,2);
layers = 1;
if (numberOfInputs ~= 0)
   subLayers = 0;
   for i=1:numberOfInputs
      subLayers = max(getTotalLayers(mnn.inputs(i)), subLayers);
   end;
   layers = layers+subLayers;
end
```

function [params] = getDefaultParams(opts)

```
% Version 2.0
% Default parameter configuration for the training algorithm
function [params] = getDefaultParams(opts)

% Genetic Algorithm Parameters
params.defaultGA = 'MNNga';
params.popSize = 10;                                % Default population size
params.xOverFn = 'defXover';                        % Default crossover function
params.xOverParams = [params.popSize];              % Total number of crossover applied to
                                                    %   the population
params.mutFn = 'defMutation';                       % Default mutation option
params.mutParams = [params.popSize 1.0 1.0 1.0 1.0];      % Mutation options
                    % [numOfMutations, mutProb, mutOpProb, mutWeightProb,mutRProb, mutRange]
                    %  numOfMutations - total number of mutation operations applied
                    %       over the population
                    %  mutProb - global probability of changing an organism
                    %  mutWeightProb - prob. of changing the weights
                    %  mutRProb - prob. of changing the R values
                    %  mutRange - a percentage of the range in which the weights can change
params.evalFn = 'defEvalFN';                        % Default evaluation function
params.evalParams = [];                             % Evaluation function's parameters
params.termFn = 'maxGenTerm';                       % Default temination function
params.termParams = [100 1.0];                      % Termination fucntion parameters
                    % [numOfGenerations minProbRequired]
                    % numOfGenerations - max. number of generations
                    % minProbRequired - min. prob. requiered to complete the evolution
params.selectFn = 'normGeomSelect';             % Default selection function
params.selectParams = [0.33];                   % selecti
                    % [normProb] - normal distribution parameter
params.opts = [1e-6 1 0];                       %

% Mophological Neural Network Parameters
params.variableArchitecture = 0;        % Variable or fixed architecture? (not used)
params.allowInfinite = 0;               % Allow infinite weights?
params.infiniteOps = 0;                 % [infProb] - probability that a weight could be inf.
params.layers = [1];                    % Layer configuration
                    %   Each entry represents a layer level
                    %   The value of the entries represent the number of neurons
                    %     connected to parent in the next level
                    %     Ex.  [ 3, 3, 1]

% Debuging Options
params.dbg.plotEvolution = 0;           % Plot all organism of the population
```

```
params.dbg.showTheBest = 0;            % Show the best organism
params.dbg.delay = 0.5;               % Delay between snapshots
params.dbg.fixedAxis = 1;             % Draw the test patterns using fixed axes
params.dbg.showCrossover = 0;         % Draw the evolution of the organism during the crossover
```

A.7 INDIRECT ENCODING METHOD

This section provides all the necessary functions used by the Indirect Encoding Method.

function [net, traceInfo] = IndirectTrainMNN(testPatterns, classes, targets, nconfig)

```
% Version 1.1
% Train a MLMP using Indirect Encoding
% testPatterns - a M by N matrix, it contains M patterns of dimension N
% classes      - a M by 2 matrix where M is the number of classes.
%              Each element in the first column is the number of test patterns that belongs to the
%                class at the corresponding index
%              The second row contains the dimension of each test pattern class
%              NOTE: all the test patterns must contains be of the same dimension
% targets      an matrix
% networkConf   see getDefaultConfig()

function [net, traceInfo] = IndirectTrainMNN(testPatterns, classes, targets, nconfig)
global class0;
global class1;
global config;

config = nconfig;
% Validate the inputs
sz = size(testPatterns);
if (sz(1) < 2)
   error('\nERROR: Insuficient number of \"testPatterns\". At least two test patterns are needed\n');
end
if (sz(2) < 2)
   error('\nERROR: Invalid dimension of the \"testPatterns\". The minimum dimenison should be 2\n');
end
[numberOfOutputs, totalClassesTargets] = size(targets);

if (numberOfOutputs < 2)
   error(sprintf('\nERROR: Insuficient number of outputs specified in \"targets\".At least two outputs are
required.\n\tCurrent value: %d',numberOfOutputs));
end

% Verify the number of classes must be less than or equal to 2*(number of outputs)
totalClasses = size(classes,1);
if (totalClasses > numberOfOutputs*2)
   error(sprintf('\nERROR: The number of outputs defined in the \"targets\"parameter must be %d',(totalClasses+1)/2));
end
if (2*totalClassesTargets < 2)
   error(sprintf('\nERROR: Insuficient number of classes in the target definition. \n\tAt least two classes are
required.\n\tCurrent value: %d',totalClassesTargets));
end
if (totalClasses < 2)
   error(sprintf('\nERROR: Insuficient number of classes in the \"classes\" definition.\n\tAt least two classes are
required.\n\tCurrent value: %d',totalClasses));
end
```

```
% Sum all the elements in the class
pos = [cumsum(classes(:,1))];
% Add append a 0 value at the begining of the vector and remove the last one.
pos = [0; pos(1:end-1,1)]+1;

% Initialize the resulting net to null
net = [];

% For each output, define the network
net = struct('op', cell(1,totalClassesTargets), 'r', {[1]}, 'weights', {[1]}, 'inputs', {[]});

traceInfo = struct('trace', {[]});
for output=1:totalClassesTargets

   % Regroup the test patterns
   fprintf('  Training output: %d\n',output);
   groups = find(targets(:,output)==0);
   totalGroups = size(groups,1);
   class0 = [];

   % Verify the class with the output value of 0 at the output index(output) exist!.
   if (totalGroups == 0)
      error('ERROR: Invalid class definition. The target with 0 on evey index must be defined');
   end;
   for class=1:totalGroups
      begn = pos(groups(class));
      final = begn+classes(groups(class))-1;
      class0 = [class0; testPatterns(begn:final,:)];
   end
   groups = find(targets(:,output)==1);
   totalGroups = size(groups,1);

   class1 = [];
   for class=1:totalGroups
      begn = pos(groups(class));
      final = begn+classes(groups(class))-1;
      class1 = [class1; testPatterns(begn:final,:)];
   end

   [res, trace] = NNmorphologicalGA(class0, class1, nconfig);
   traceInfo(output).trace = trace;
   net(output) = res;
end

return
```

function [res, traceInfo] = NNmorphologicalGA(c0, c1, params)

```
function [res, traceInfo] = NNmorphologicalGA(c0, c1, params)
% Declaracion de variable globales
res = 0;
global asciiString;
global Range;
global numOfBits;
global binString;       %
global maxValue;
global numOfChars;
global varBounds;
global numOfPoints;
global minDist;
global gPoints;
global bPoints;

global worstEval;
populationSize =params.popSize;

gPoints = c0;
```

```
bPoints = c1;

if (size(gPoints,1) > 50)
   gPoints = gPoints(1:50,:);
end

numOfPoints = size(gPoints,1);

hold off;
plot(gPoints(:,1),gPoints(:,2),'r.');%'r.'
hold on;
plot(bPoints(:,1),bPoints(:,2),'k.');
%hold on;


tPoints = [gPoints; bPoints];

gPointsSz = size(gPoints,1);
bPointsSz = size(bPoints,1);

minDist = [];
for z =1:size(tPoints,2)
   dist = 1e10;
   for m= 1:size(tPoints,1)-1
      for n =m+1:size(tPoints,1);
         cDist = abs(tPoints(m,z) - tPoints(n,z));
         if ((cDist > 0) && (cDist < dist) )
            dist = cDist;
         end
      end;
   end
   minDist(1,z) = dist;
end
minDist = minDist/2;


varBounds = [];
for n = 1:numOfPoints
   varBounds(n,1) = 0;
   varBounds(n,2) = 2^(ceil(log2(numOfPoints)))-1;
end
varBounds(numOfPoints+1,1) = 0;
varBounds(numOfPoints+1,2) = 2^(numOfPoints-1)-1;

pop = generatePop(populationSize, varBounds);


[sol, pop,bPop,traceInfo] = ga(varBounds, params.evalFn, params.evalParams, pop, params.opts, params.termFn,...
   params.termParams, params.selectFn, params.selectParams, params.xOverFn, params.xOverParams,params.mutFn,
params.mutParams);

solSize = size(sol,2)-1;
sol(1:solSize) = sol(1:solSize)+1;
groups = sol(end);
groups = double(dec2bin(groups,numOfPoints-1)-48);

limits = find(groups>0);

if (size(limits,2) == 0)
   limits = [limits (solSize)];
else
   if (limits(size(limits,2)) ~= solSize)
      limits = [limits (solSize)];
   end
end;
totalGroups = size(limits,2);


subGroup = [];
```

```
totalPointsInside = 0;
prev = 1;

root.op = 0;        %or
root.r = [];
root.weights = [];
root.inputs = [];

for i = 1:totalGroups
   subGroup = sol(prev:limits(i));
   fprintf('%i ',subGroup);
   fprintf('-');


   testpts = subGroup;

   minDim = gPoints(testpts(1),:);
   maxDim = minDim;

   vecDimension = size(gPoints,2);

   for n=1:size(testpts,2)
      for vs=1:vecDimension
         if (gPoints(testpts(n),vs) < minDim(1,vs))
            minDim(1,vs) = gPoints(testpts(n),vs);
         end;
         if (gPoints(testpts(n),vs) > maxDim(1,vs))
            maxDim(1,vs) = gPoints(testpts(n),vs);
         end;
      end;
   end;
   eq = find(minDim == maxDim);
   maxDim(eq) = maxDim(eq)+minDist(eq);
   minDim(eq) = minDim(eq)-minDist(eq);

   mnn1.op = 1;        %and
   mnn1.r = -ones(1,vecDimension);
   mnn1.weights = minDim;
   mnn1.inputs = [];

   mnn2.op = 1;        %and
   mnn2.r = ones(1,vecDimension);
   mnn2.weights = maxDim;
   mnn2.inputs = [];

   mnn3.op = 1;        %and
   mnn3.r = [1 1];
   mnn3.weights = [0 0];
   mnn3.inputs = [mnn1 mnn2];

   root.r = [root.r 1];
   root.weights = [root.weights 0];
   root.inputs = [root.inputs mnn3];
%    if ((size(subGroup,2) > 1) && (maxDim(1,2)-minDim(1,2)>0) && (maxDim(1,1)-minDim(1,1)>0))
%       square = [minDim(1,1) minDim(1,2); maxDim(1,1) minDim(1,2); maxDim(1,1) maxDim(1,2); minDim(1,1)
maxDim(1,2); minDim(1,1) minDim(1,2)];
%    else
%       square = [minDim(1,1)-minDist(1) minDim(1,2)-minDist(2); maxDim(1,1)+minDist(1) minDim(1,2)-minDist(2);
maxDim(1,1)+minDist(1) maxDim(1,2)+minDist(2); minDim(1,1)-minDist(1) maxDim(1,2)+minDist(2); minDim(1,1)-
minDist(1) minDim(1,2)-minDist(2)];
%    end


%    plot(square(:,1), square(:,2),'-');
%    hold on;

   prev = limits(i)+1;
end

if (size(root.inputs,2) <2)
```

```
    root = root.inputs(1);
end
res = root;
```

function [pop] = generatePop(popSize, bounds)

```
function [pop] = generatePop(popSize, bounds)
global numOfPoints;

%numOfPoints = 4;
bits = calcbits(bounds,1);

pts = [];
for n=1:numOfPoints
    pts = [pts (n-1)];
end;

pop = [];
popp = [];

for n=1:popSize
    totalPts = numOfPoints;
    res = [];
    for m=1:numOfPoints
        pos = round(rand*(totalPts-1))+1;
        val = pts(pos);
        pts(:,pos) = [];
        pts = [pts val];

        res = [res val];
        totalPts = totalPts-1;
    end;

    res = [res round(bounds(end,2)*rand)];
    [r, val] = NNmorphologicalEval(res,[]);
%    res = res-1;
    res = [res val];
    popp = [popp; res];

    resBits = [];
    for m=1:size(bits,2)
        resBits = [resBits double(dec2bin(res(m),bits(m))-48)];
    end
    resBits = [resBits res(end)];
    pop = [pop; resBits];
end
%popp
```

function [c] = NNmorphologicalMutation(parent,bounds,Ops)

```
function [c] = NNmorphologicalMutation(parent,bounds,Ops)
bits = calcbits(bounds,1);
%fprintf('mutation\n');
%for n=1:size(parent,2)
%    fprintf('%i ',parent(n));
%end

n = 0;
for i=1:size(bits,2)-1
    n = n+bits(i);
end
value = parent(n+1:n+bits(end));
for i=1:size(value,2)
```

```
    if (rand<0.08)
        if (value(i) == 1) value(i) =0;
        else
            value(i) = 1;
        end
    end
end
parent(n+1:n+bits(end)) = value(1:end);

bits = calcbits(bounds,1);

p1 = [];
sz = size(bits,2);
pos = 1;
for b=1:sz-1
    num1 = 0;
    for n=1:bits(b)
        num1 = num1*2+parent(pos);
        pos = pos+1;
    end
    p1 = [p1 num1];
end

i1 = fix(size(p1,2)*rand)+1;
i2 = fix(size(p1,2)*rand)+1;
e = p1(i1);
p1(i1) = p1(i2);
p1(i2) = e;

rc1 = [];
for b=1:size(bits,2)-1
    rc1 = [rc1 int8(dec2bin(p1(b),bits(b))-48)];
end
parent(1:size(rc1,2)) = rc1(1:end);

parent = removeOverlap(parent,bits);
c = parent;
```

function [c1,c2] = NNmorphologicalXover(m1,m2,bounds, Ops)

```
function [c1,c2] = NNmorphologicalXover(m1,m2,bounds, Ops)
% con
% m1 - array of bits
%
%c1=m1;
%c2=m2;
%c1
%c2
%return;

%fprintf('morphologicalXover\n');
bits = calcbits(bounds,1);

p1 = [];
p2 = [];
sz = size(bits,2);
pos = 1;
for b=1:sz-1
    num1 = 0;
    num2 = 0;
    for n=1:bits(b)
        num1 = num1*2+m1(pos);
        num2 = num2*2+m2(pos);
        pos = pos+1;
    end
    p1 = [p1 num1];
    p2 = [p2 num2];
```

```
end

%p1 = p1+1;
%p2 = p2+1;

c1 = p1;
c2 = p2;

%return;
sz=size(p1,2);
n=floor(sz/2);
%cut1 = round(rand*(n/2-1))+1;         %Generate random cut point U(1,n/2);
%cut2 = round(rand*(sz-cut1-1))+cut1;  %Generate random cut point U(cut1+1,n-1);
cut1 = round(rand*(n-1)+0.5); %Generate random cut point U(1,n/2);
cut2 = round(rand*(sz-cut1-1)+1+cut1); %Generate random cut point U(cut1+1,n-1);
pm1=p1(1:end);
pm2=p2(1:end);
c1=p1;
c2=p2;
for i=[1:cut1 (cut2+1):sz]
  pm1=replace(pm1,p2(i),-1);
  pm2=replace(pm2,p1(i),-1);
end

c1((cut1+1):cut2)=p2(find(pm2>=0));
c2((cut1+1):cut2)=p1(find(pm1>=0));

%c1 = c1-1;
%c2 = c2-1;
%fprintf('out c1 and c2');

rc1 = [];
rc2 = [];
for b=1:size(bits,2)-1
   rc1 = [rc1 double(dec2bin(c1(b),bits(b))-48)];
   rc2 = [rc2 double(dec2bin(c2(b),bits(b))-48)];
end
c1 = m1;
c2 = m2;

sz = size(rc1,2);
c1(1:sz) = rc1(1:end);
c2(1:sz) = rc2(1:end);
```

function [sol, val] = NNmorphologicalEval(sol,parameters)

```
function [sol, val] = NNmorphologicalEval(sol,parameters)

global numOfPoints;
global worstEval;
%sol = sol;
%val = 10;
%return;
%numOfPoints = 8;

solSize = size(sol,2)-1;
sol(1:solSize) = sol(1:solSize)+1;
groups = sol(end);
groups = double(dec2bin(groups,numOfPoints-1)-48);

limits = find(groups>0);

if (size(limits,2) == 0)
   limits = [limits (solSize)];
else
   if (limits(size(limits,2)) ~= solSize)
      limits = [limits (solSize)];
```

```
   end
end;
totalGroups = size(limits,2);


subGroup = [];

totalPointsInside = 0;
prev = 1;
for i = 1:totalGroups
   subGroup = sol(prev:limits(i));
%    fprintf('%i ',subGroup);
%    fprintf('-');

   totalPointsInside = totalPointsInside + testPoints(subGroup);

   prev = limits(i)+1;
end

%sol = sol;
val = 1.0/(((totalPointsInside+1)^2)*(totalGroups));
%if (totalPointsInside >= 0)
%    fprintf(' %i\n',val);
%end
sol(1:end-1) = sol(1:end-1)-1;
```

function [done] = NNmorphologicalFitnessFoundTerm(ops,bPop,endPop)

```
function [done] = NNmorphologicalFitnessFoundTerm(ops,bPop,endPop)
global numOfChars;
global maxValue;
%bPop
[x,y] = size(bPop);
%fprintf('El mejor string: %s\n',char(bPop(x, 2:numOfChars+1)));
currentGen = ops(1);
maxGen = ops(2);
%done = currentGen >= maxGen  | maxValue <= bPop(x,y);
done = currentGen >= maxGen;
%bPop
%endPop
```

function [params] = getDefaultParams(opts)

```
% Version 2.0
% Default parameter configuration for the training algorithm
function [params] = getDefaultParams(opts)

% Genetic Algorithm Parameters
params.popSize = 10;                                % Default population size
params.xOverFn = 'NNmorphologicalXover';                                % Default crossover function
params.xOverParams = [0.8 3];                   % Total number of crossover applied to
                                                %   the population
params.mutFn = 'NNmorphologicalMutation';                               % Default mutation option
params.mutParams = [0.07 7];       % Mutation options
                                % [numOfMutations, mutProb, mutOpProb, mutWeightProb,mutRProb, mutRange]
                                %  numOfMutations - total number of mutation operations applied
                                %         over the population
                                %  mutProb - global probability of changing an organism
                                %  mutWeightProb - prob. of changing the weights
                                %  mutRProb - prob. of changing the R values
                                %  mutRange - a percentage of the range in which the weights can change
```

```
params.evalFn = 'NNmorphologicalEval';                          % Default evaluation function
params.evalParams = [];                                  % Evaluation function's parameters
params.termFn = 'NNmorphologicalFitnessFoundTerm';                          % Default temination function
params.termParams = [100 1.0];                           % Termination fucntion parameters
                                    % [numOfGenerations minProbRequired]
                                    % numOfGenerations - max. number of generations
                                    % minProbRequired - min. prob. requiered to complete the evolution
params.selectFn = 'roulette';             % Default selection function
params.selectParams = [0.08];                     % selecti
                                  % [normProb] - normal distribution parameter
params.opts = [1 0 1];                    %

% Mophological Neural Network Parameters
params.variableArchitecture = 0;        % Variable or fixed architecture?
params.allowInfinite = 0;               % Allow infinit weights?
params.infiniteOps = 0;                 % [infProb] - probability that a weight could be inf.
params.layers = [1];                   % Layer configuration
                                  %   Each entry represents a layer level
                                  %   The value of the entries represent the number of neurons
                                  %     connected to parent in the next level
                                  %     Ex.  [ 3, 3, 1]

% Debuging Options
params.dbg.plotEvolution = 0;           % Plot all organism of the population
params.dbg.showTheBest = 0;              % Show the best organism
params.dbg.delay = 0.5;                % Delay between snapshots
params.dbg.fixedAxis = 1;               % Draw the test patterns using fixed axes
params.dbg.showCrossover = 0;           % Draw the evolution of the organism during the crossover
```

function [x,endPop,bPop,traceInfo] = ga(bounds,evalFN,evalOps,startPop, opts,termFN,termOps,selectFN,selectOps,xOverFNs,xOverOps,mutFNs,mutOps)

```
function [x,endPop,bPop,traceInfo] = ga(bounds,evalFN,evalOps,startPop,opts,...
termFN,termOps,selectFN,selectOps,xOverFNs,xOverOps,mutFNs,mutOps)
global minDist;
% GA run a genetic algorithm
% function [x,endPop,bPop,traceInfo]=ga(bounds,evalFN,evalOps,startPop,opts,
%                                termFN,termOps,selectFN,selectOps,
%                                xOverFNs,xOverOps,mutFNs,mutOps)
%
% Output Arguments:
%   x           - the best solution found during the course of the run
%   endPop      - the final population
%   bPop        - a trace of the best population
%   traceInfo   - a matrix of best and means of the ga for each generation
%
% Input Arguments:
%   bounds      - a matrix of upper and lower bounds on the variables
%   evalFN      - the name of the evaluation .m function
%   evalOps     - options to pass to the evaluation function ([NULL])
%   startPop    - a matrix of solutions that can be initialized
%                 from initialize.m
%   opts        - [epsilon prob_ops display] change required to consider two
%                 solutions different, prob_ops 0 if you want to apply the
%                 genetic operators probabilisticly to each solution, 1 if
%                 you are supplying a deterministic number of operator
%                 applications and display is 1 to output progress 0 for
%                 quiet. ([1e-6 1 0])
%   termFN      - name of the .m termination function (['maxGenTerm'])
%   termOps     - options string to be passed to the termination function
%                 ([100]).
%   selectFN    - name of the .m selection function (['normGeomSelect'])
%   selectOpts  - options string to be passed to select after
%                 select(pop,#,opts) ([0.08])
```

```
%   xOverFNS     - a string containing blank seperated names of Xover.m
%                  files (['arithXover heuristicXover simpleXover'])
%   xOverOps     - A matrix of options to pass to Xover.m files with the
%                  first column being the number of that xOver to perform
%                  similiarly for mutation ([2 0;2 3;2 0])
%   mutFNs       - a string containing blank seperated names of mutation.m
%                  files (['boundaryMutation multiNonUnifMutation ...
%                        nonUnifMutation unifMutation'])
%   mutOps       - A matrix of options to pass to Xover.m files with the
%                  first column being the number of that xOver to perform
%                  similiarly for mutation ([4 0 0;6 100 3;4 100 3;4 0 0])

% Binary and Real-Valued Simulation Evolution for Matlab
% Copyright (C) 1996 C.R. Houck, J.A. Joines, M.G. Kay
%
% C.R. Houck, J.Joines, and M.Kay. A genetic algorithm for function
% optimization: A Matlab implementation. ACM Transactions on Mathmatical
% Software, Submitted 1996.
%
% This program is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 1, or (at your option)
% any later version.
%
% This program is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU General Public License for more details. A copy of the GNU
% General Public License can be obtained from the
% Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

%%$Log: ga.m,v $
%Revision 1.10  1996/02/02  15:03:00  jjoine
% Fixed the ordering of imput arguments in the comments to match
% the actual order in the ga function.
%
%Revision 1.9  1995/08/28  20:01:07  chouck
% Updated initialization parameters, updated mutation parameters to reflect
% b being the third option to the nonuniform mutations
%
%Revision 1.8  1995/08/10  12:59:49  jjoine
%Started Logfile to keep track of revisions
%


n=nargin;
if n<2 | n==6 | n==10 | n==12
  disp('Insufficient arguements')
end
if n<3 %Default evalation opts.
  evalOps=[];
end
if n<5
  opts = [1e-6 1 0];
end
if isempty(opts)
  opts = [1e-6 1 0];
end

if any(evalFN<48) %Not using a .m file
  if opts(2)==1 %Float ga
    e1str=['x=c1; c1(xZomeLength)=', evalFN ';'];
    e2str=['x=c2; c2(xZomeLength)=', evalFN ';'];
  else %Binary ga
    e1str=['x=b2f(endPop(j,:),bounds,bits); endPop(j,xZomeLength)=',...
              evalFN ';'];
  end
else %Are using a .m file
  if opts(2)==1 %Float ga
    e1str=['[c1 c1(xZomeLength)]=' evalFN '(c1,[gen evalOps]);'];
```

```
    e2str=['[c2 c2(xZomeLength)]=' evalFN '(c2,[gen evalOps]);'];
  else %Binary ga
    e1str=['x=b2f(endPop(j,:),bounds,bits);[x v]=' evalFN ...
              '(x,[gen evalOps]); endPop(j,:)=[f2b(x,bounds,bits) v];'];
  end
end


if n<6 %Default termination information
  termOps=[100];
  termFN='maxGenTerm';
end
if n<12 %Default muatation information
  if opts(2)==1 %Float GA
  mutFNs=['boundaryMutation multiNonUnifMutation nonUnifMutation unifMutation'];
    mutOps=[4 0 0;6 termOps(1) 3;4 termOps(1) 3;4 0 0];
  else %Binary GA
    mutFNs=['binaryMutation'];
    mutOps=[0.05];
  end
end
if n<10 %Default crossover information
  if opts(2)==1 %Float GA
    xOverFNs=['arithXover heuristicXover simpleXover'];
    xOverOps=[2 0;2 3;2 0];
  else %Binary GA
    xOverFNs=['simpleXover'];
    xOverOps=[0.6];
  end
end
if n<9 %Default select opts only i.e. roullete wheel.
  selectOps=[];
end
if n<8 %Default select info
  selectFN=['normGeomSelect'];
  selectOps=[0.08];
end
if n<6 %Default termination information
  termOps=[100];
  termFN='maxGenTerm';
end
if n<4 %No starting population passed given
  startPop=[];
end
if isempty(startPop) %Generate a population at random
  %startPop=zeros(80,size(bounds,1)+1);
  startPop=initializega(80,bounds,evalFN,evalOps,opts(1:2));
end

if opts(2)==0 %binary
  bits=calcbits(bounds,opts(1));
end

xOverFNs=parse(xOverFNs);
mutFNs=parse(mutFNs);

xZomeLength  = size(startPop,2);          %Length of the xzome=numVars+fittness
numVar       = xZomeLength-1;                    %Number of variables
popSize      = size(startPop,1);          %Number of individuals in the pop
endPop       = zeros(popSize,xZomeLength); %A secondary population matrix
c1           = zeros(1,xZomeLength);      %An individual
c2           = zeros(1,xZomeLength);      %An individual
numXOvers    = size(xOverFNs,1);          %Number of Crossover operators
numMuts      = size(mutFNs,1);                   %Number of Mutation operators
epsilon      = opts(1);              %Threshold for two fittness to differ
epsilon      = 1e-6;
oval         = max(startPop(:,xZomeLength)); %Best value in start pop
bFoundIn     = 1;                        %Number of times best has changed
done         = 0;                  %Done with simulated evolution
gen          = 1;                        %Current Generation Number
```

```
collectTrace = (nargout>3);                  %Should we collect info every gen
floatGA      = opts(2)==1;            %Probabilistic application of ops
display      = opts(3);               %Display progress

while(~done)
  %Elitist Model
  [bval,bindx] = max(startPop(:,xZomeLength)); %Best of current pop
  best =  startPop(bindx,:);

  if collectTrace
    traceInfo(gen,1)=gen;                              %current generation
    traceInfo(gen,2)=startPop(bindx,xZomeLength);      %Best fittness
    traceInfo(gen,3)=mean(startPop(:,xZomeLength));    %Avg fittness
    traceInfo(gen,4)=std(startPop(:,xZomeLength));
  end

  if ( (abs(bval - oval)>epsilon) | (gen==1)) %If we have a new best sol
    if display
      fprintf(1,'\n%d %f\n',gen,bval);         %Update the display
    end
    if floatGA
      bPop(bFoundIn,:)=[gen startPop(bindx,:)]; %Update bPop Matrix
    else
      bPop(bFoundIn,:)=[gen b2f(startPop(bindx,1:numVar),bounds,bits)...
                startPop(bindx,xZomeLength)];
    end
    bFoundIn=bFoundIn+1;                    %Update number of changes
    oval=bval;                           %Update the best val
  else
    if display
      fprintf(1,'%d ',gen);                      %Otherwise just update num gen
    end
  end

  endPop = feval(selectFN,startPop,[gen selectOps]); %Select

  if floatGA %Running with the model where the parameters are numbers of ops
    for i=1:numXOvers,
      for j=1:xOverOps(i,1),
              a = round(rand*(popSize-1)+1);             %Pick a parent
              b = round(rand*(popSize-1)+1);             %Pick another parent
              xN=deblank(xOverFNs(i,:));    %Get the name of crossover function
              [c1 c2] = feval(xN,endPop(a,:),endPop(b,:),bounds,[gen xOverOps(i,:)]);

              if c1(1:numVar)==endPop(a,(1:numVar)) %Make sure we created a new
                c1(xZomeLength)=endPop(a,xZomeLength); %solution before evaluating
              elseif c1(1:numVar)==endPop(b,(1:numVar))
                c1(xZomeLength)=endPop(b,xZomeLength);
              else
                %[c1(xZomeLength) c1] = feval(evalFN,c1,[gen evalOps]);
                eval(e1str);
              end
              if c2(1:numVar)==endPop(a,(1:numVar))
                c2(xZomeLength)=endPop(a,xZomeLength);
              elseif c2(1:numVar)==endPop(b,(1:numVar))
                c2(xZomeLength)=endPop(b,xZomeLength);
              else
                %[c2(xZomeLength) c2] = feval(evalFN,c2,[gen evalOps]);
                eval(e2str);
              end

              endPop(a,:)=c1;
              endPop(b,:)=c2;
      end
    end

    for i=1:numMuts,
      for j=1:mutOps(i,1),
              a = round(rand*(popSize-1)+1);
              c1 = feval(deblank(mutFNs(i,:)),endPop(a,:),bounds,[gen mutOps(i,:)]);
```

```
            if c1(1:numVar)==endPop(a,(1:numVar))
              c1(xZomeLength)=endPop(a,xZomeLength);
            else
              %[c1(xZomeLength) c1] = feval(evalFN,c1,[gen evalOps]);
              eval(e1str);
            end
            endPop(a,:)=c1;
    end
   end

  else %We are running a probabilistic model of genetic operators
   for i=1:numXOvers,
    xN=deblank(xOverFNs(i,:));          %Get the name of crossover function
    cp=find(rand(popSize,1)<xOverOps(i,1)==1);
%     cp
    if rem(size(cp,1),2) cp=cp(1:(size(cp,1)-1)); end

%     cp
    cp=reshape(cp,size(cp,1)/2,2);
    for j=1:size(cp,1)
            a=cp(j,1); b=cp(j,2);
            [endPop(a,:) endPop(b,:)] = feval(xN,endPop(a,:),endPop(b,:),...
              bounds,[gen xOverOps(i,:)]);
    end
   end
   for i=1:numMuts
    mN=deblank(mutFNs(i,:));
    for j=1:popSize
            endPop(j,:) = feval(mN,endPop(j,:),bounds,[gen mutOps(i,:)]);
            eval(e1str);
    end
   end
  end

  gen=gen+1;
  done=feval(termFN,[gen termOps],bPop,endPop); %See if the ga is done
  startPop=endPop;                      %Swap the populations

  [bval,bindx] = min(startPop(:,xZomeLength)); %Keep the best solution
  startPop(bindx,:) = best;             %replace it with the worst
end

[bval,bindx] = max(startPop(:,xZomeLength));
if display
  fprintf(1,'\n%d %f\n',gen,bval);
end

x=startPop(bindx,:);
if opts(2)==0 %binary
  x=b2f(x,bounds,bits);
  bPop(bFoundIn,:)=[gen b2f(startPop(bindx,1:numVar),bounds,bits)...
     startPop(bindx,xZomeLength)];
else
  bPop(bFoundIn,:)=[gen startPop(bindx,:)];
end

if collectTrace
  traceInfo(gen,1)=gen;                 %current generation
  traceInfo(gen,2)=startPop(bindx,xZomeLength); %Best fittness
  traceInfo(gen,3)=mean(startPop(:,xZomeLength)); %Avg fittness
end
```

A.8 CARTESIAN GENETIC PROGRAMMING METHOD

This section provides all the necessary functions used by the Cartesian Genetic Programming method.

function [net] = CGPTrainMNN(testPatterns, classes, targets, nconfig)

```
% Version 1.1
% Train a Morphological Neural Network
% testPatterns - a M by N matrix, it contains M patterns of dimension N
% classes      - a M by 2 matrix where M is the number of classes.
%                Each element in the first column is the number of test patterns that belongs to the
%                 class at the corresponding index
%                The second row contains the dimension of each test pattern class
%                NOTE: all the test patterns must contains be of the same dimension
% targets      an matrix
% networkConf   see getDefaultConfig()

function [net, traceInfo] = CGPTrainMNN(testPatterns, classes, targets, nconfig)

global class0;
global class1;
global F;
global FTotal;
global param;

ga = nconfig;
% Validate the inputs
sz = size(testPatterns);
if (sz(1) < 2)
   error('\nERROR: Insuficient number of \"testPatterns\". At least two test patterns are needed\n');
end
if (sz(2) < 2)
   error('\nERROR: Invalid dimension of the \"testPatterns\". The minimum dimenison should be 2\n');
end
[numberOfOutputs, totalClassesTargets] = size(targets);

if (numberOfOutputs < 2)
   error(sprintf('\nERROR: Insuficient number of outputs specified in \"targets\". At least two outputs are
required.\n\tCurrent value: %d',numberOfOutputs));
end

% Verify the number of classes must be less than or equal to 2*(number of outputs)
totalClasses = size(classes,1);
if (totalClasses > numberOfOutputs*2)
   error(sprintf('\nERROR: The number of outputs defined in the \"targets\" parameter must be %d',(totalClasses+1)/2));
end
if (2*totalClassesTargets < 2)
   error(sprintf('\nERROR: Insuficient number of classes in the target definition. \n\tTAt least two classes are
required.\n\tCurrent value: %d',totalClassesTargets));
end
if (totalClasses < 2)
   error(sprintf('\nERROR: Insuficient number of classes in the \"classes\" definition.\n\tAt least two classes are
required.\n\tCurrent value: %d',totalClasses));
end

% Sum all the elements in the class
pos = [cumsum(classes(:,1))];
% Add append a 0 value at the begining of the vector and remove the last one.
pos = [0; pos(1:end-1,1)]+1;

% Initialize the resulting net to null
net = [];

% For each output, define the network
net = struct('op', cell(1,totalClassesTargets), 'r', {[1]}, 'weights', {[1]}, 'inputs', {[]});
traceInfo = struct('trace', {[]});
```

```
for output=1:totalClassesTargets

    % Regroup the test patterns
    fprintf('  Training output: %d\n',output);
    groups = find(targets(:,output)==0);
    totalGroups = size(groups,1);
    class0 = [];

    % Verify the class with the output value of 0 at the output index(output) exist!.
    if (totalGroups == 0)
       error('ERROR: Invalid class definition. The target with 0 on evey index must be defined');
    end;
    for class=1:totalGroups
       begn = pos(groups(class));
       final = begn+classes(groups(class))-1;
       class0 = [class0; testPatterns(begn:final,:)];
    end
    groups = find(targets(:,output)==1);
    totalGroups = size(groups,1);

    class1 = [];
    for class=1:totalGroups
       begn = pos(groups(class));
       final = begn+classes(groups(class))-1;
       class1 = [class1; testPatterns(begn:final,:)];
    end

    param = CGPDefaultParam(size(class0,2), ga.layers, ga.connections);

    [F, FTotal] = CGPInitialize(class0, param);

    [initialPopulation, bounds] = CGPGeneratePop2(ga.popSize, ga.evalFn, ga.evalParam);

    [sol, pop,bPop,trace] = CGPga2(bounds, ga.evalFn, ga.evalParam, initialPopulation, [0.000001 1 1],
['CGPFitnessFoundTerm2'],...
       [ga.maxGen], ga.selectFn, ga.selectParam, ga.xOverFn, ga.xOverParam, ga.mutationFn, ga.mutationParam);
    res = CGPDecodeNet(sol(1,1:end-1), F, FTotal, param);

    traceInfo(output).trace = trace;
   % trace(output) = traceInfo;
    net(output) = res;
end
```

function [x,endPop,bPop,traceInfo] = CGPGA2(bounds,evalFN,evalOps, startPop,opts,termFN,termOps,selectFN,selectOps,xOverFNs,xOverOps ,mutFNs,mutOps)

```
function [x,endPop,bPop,traceInfo] = CGPGA2(bounds,evalFN,evalOps,startPop,opts,...
termFN,termOps,selectFN,selectOps,xOverFNs,xOverOps,mutFNs,mutOps)
% GA run a genetic algorithm
% function [x,endPop,bPop,traceInfo]=ga(bounds,evalFN,evalOps,startPop,opts,
%                                termFN,termOps,selectFN,selectOps,
%                                xOverFNs,xOverOps,mutFNs,mutOps)
%
% Output Arguments:
%   x           - the best solution found during the course of the run
%   endPop      - the final population
%   bPop        - a trace of the best population
%   traceInfo   - a matrix of best and means of the ga for each generation
%
% Input Arguments:
%   bounds      -
```

```
%   evalFN      -
%   evalOps     -
%   startPop    -
%   opts        - [epsilon prob_ops display] change required to consider two
%                 solutions different, prob_ops 0 if you want to apply the
%                 genetic operators probabilisticly to each solution, 1 if
%                 you are supplying a deterministic number of operator
%                 applications and display is 1 to output progress 0 for
%                 quiet. ([1e-6 1 0])
%   termFN      -
%   termOps     -
%   selectFN    -
%   selectOpts  -
%   xOverFNS    -
%   xOverOps    -
%   mutFNs      -
%   mutOps      -

global F;
global FTotal;
global param;
global class0;
global class1;

n=nargin;
if n<2 | n==6 | n==10 | n==12
  disp('Insufficient arguements')
end
if n<3 %Default evalation opts.
  evalOps=[];
end
if n<5
  opts = [1e-6 1 0];
end
if isempty(opts)
  opts = [1e-6 1 0];
end

  if opts(2)==1 %Float ga
    e1str=['[c1 c1(xZomeLength)]=' evalFN '(c1,[gen evalOps]);'];
    e2str=['[c2 c2(xZomeLength)]=' evalFN '(c2,[gen evalOps]);'];
  end

if n<6 %Default termination information
  termOps=[100];
  termFN='maxGenTerm';
end
if n<12 %Default muatation information
  if opts(2)==1 %Float GA
    mutFNs=['binaryMutation'];
    mutOps=[0.05];
  end
end
if n<10 %Default crossover information
  if opts(2)==1 %Float GA
    xOverFNs=['simpleXover'];
    xOverOps=[0.6];
  end
end
if n<9 %Default select opts only i.e. roullete wheel.
  selectOps=[];
end
if n<8 %Default select info
  selectFN=['normGeomSelect'];
  selectOps=[0.08];
end
if n<6 %Default termination information
  termOps=[100];
  termFN='maxGenTerm';
end
```

```
if n<4 %No starting population passed given
  startPop=[];
end
if isempty(startPop) %Generate a population at random

end


xOverFNs=parse(xOverFNs);
mutFNs=parse(mutFNs);

xZomeLength  = size(startPop,2);              %Length of the xzome=numVars+fittness
numVar       = xZomeLength-1;                 %Number of variables
popSize      = size(startPop,1);              %Number of individuals in the pop
endPop       = zeros(popSize,xZomeLength); %A secondary population matrix
c1           = zeros(1,xZomeLength);          %An individual
c2           = zeros(1,xZomeLength);          %An individual
numXOvers    = size(xOverFNs,1);              %Number of Crossover operators
numMuts      = size(mutFNs,1);                %Number of Mutation operators
epsilon      = opts(1);               %Threshold for two fittness to differ
oval         = max(startPop(:,xZomeLength)); %Best value in start pop
bFoundIn     = 1;                             %Number of times best has changed
done         = 0;                    %Done with simulated evolution
gen          = 1;                             %Current Generation Number
collectTrace = (nargout>3);                   %Should we collect info every gen
floatGA      = 1;                     %Probabilistic application of ops
display      = opts(3);               %Display progress

while(~done)
  pause(0.05);
  %Elitist Model
  [bval,bindx] = max(startPop(:,xZomeLength));          %Best of current pop
  best =  startPop(bindx,:);

  if collectTrace
    traceInfo(gen,1)=gen;                                %current generation
    traceInfo(gen,2)=startPop(bindx,xZomeLength);       %Best fittness
    traceInfo(gen,3)=mean(startPop(:,xZomeLength));     %Avg fittness
    traceInfo(gen,4)=std(startPop(:,xZomeLength));
  end

  if ( (abs(bval - oval)>epsilon) | (gen==1))          %If we have a new best sol
    if display
      fprintf(1,'\n%d %f\n',gen,bval);                %Update the display
    end

    bPop(bFoundIn,:)=[gen startPop(bindx,:)];          %Update bPop Matrix

    bFoundIn=bFoundIn+1;                            %Update number of changes
    oval=bval;                                   %Update the best val
  else
    if display
      fprintf(1,'%d ',gen);                          %Otherwise just update num gen
    end
  end

  if (0==1)
     sz = size(startPop,1);
     for o=1:sz
        figure(o);
        hold off;
        plot(class1(:,1),class1(:,2),'bo');
        hold on;
        plot(class0(:,1),class0(:,2),'gs');

        plotNetwork2(CGPDecodeNet(startPop(o,1:end-1), F, FTotal, param));
%         plotMorphologicalPerceptron(endPop(o,1:end-1));
      end
     % pause;
   end
```

```
  endPop = feval(selectFN,startPop,[gen selectOps]);    %Select
  totalOrg = size(endPop,1);
  fit = endPop(:,end);
  totalFitness = sum(fit);
  if (totalFitness == 0)
     totalFitness = 1;
  end
  fit = cumsum(fit/totalFitness);


  while (totalOrg < popSize)
     a = find(fit-rand>=0);
     b = find(fit-rand>=0);
     a = a(1);
     b = b(1);

     xN=deblank(xOverFNs(1,:));                          %Get the name of crossover function
              [c1 c2] = feval(xN,endPop(a,:),endPop(b,:),bounds,[gen xOverOps(1,:)]);

     c1 = feval(mutFNs(1,:),c1,bounds,[gen mutOps(1,:)]);
              c2 = feval(mutFNs(1,:),c2,bounds,[gen mutOps(1,:)]);
     eval(e1str);
     eval(e2str);


     endPop(totalOrg+1,:)=c1;
              endPop(totalOrg+2,:)=c2;
     totalOrg = totalOrg+2;
  end

  gen=gen+1;
  done=feval(termFN,[gen termOps],bPop,endPop); %See if the ga is done
  startPop=endPop;                          %Swap the populations

  [bval,bindx] = min(startPop(:,xZomeLength)); %Keep the best solution
  startPop(bindx,:) = best;                 %replace it with the worst
end

[bval,bindx] = max(startPop(:,xZomeLength));
if display
  fprintf(1,'\n%d %f\n',gen,bval);
end

x=startPop(bindx,:);
if opts(2)==1 %binary
  bPop(bFoundIn,:)=[gen startPop(bindx,:)];
end

if collectTrace
  traceInfo(gen,1)=gen;                     %current generation
  traceInfo(gen,2)=startPop(bindx,xZomeLength); %Best fittness
  traceInfo(gen,3)=mean(startPop(:,xZomeLength)); %Avg fittness
end
```

function [res] = CGPDecodeNet(chrom, F, FTotal, param)

```
function [res] = CGPDecodeNet(chrom, F, FTotal, param)
% Decodes a chormosome
% chrom - an integer array that contains the genes
% F - matrix that contains all the node functions
% FTotal - a vector that contains the number of function available for a layer
% param - default parameters for the GA
```

```
% Excample of the chromosome
%chrom = [ 1, 2, 1, 1, 2, 2, 1, 2, 3, 1, 2, 4, 1, 2, 5, 1, 2, 6, 1, 2, 7, 1, 2, 8, 1, 2, 9, 1, 2, 10,   4, 5, 1, 3, 4, 2, 5, 6, 3, 7, 8, 4,
9, 1, 5, 2, 3, 6, 4, 5, 7, 6, 7, 8, 8, 1, 8, 8, 1, 2,   1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1];
totalNodes = param.numOfNodes;
totalLayers = param.numOfLayers;
numOfInputs = param.numOfInputs;

%lastNode = sum(totalNodes(1:end-1))*(numOfInputs+1);
lastNode = sum(totalNodes(1:end-1).*(numOfInputs(1:end-1)+1));
%%num of Inptus
%lastNode = totalNodes*(numOfInputs+1)*(totalLayers-1);

connections = chrom(lastNode+1:end-1);
numOfConnections = size(connections,2);

fnet = F(1,3);  %hard coded

t = 0;
for n=1:numOfConnections
   if (connections(n) == 1)
      net = CGPDecodeNode(chrom, n, totalLayers-1, totalNodes, numOfInputs, F, FTotal);

      fnet.inputs = [fnet.inputs net];
      fnet.weights =[fnet.weights 0];
      fnet.r = [fnet.r 1];
   end
end
if (size(fnet.inputs,2) == 0)
   res = fnet;
elseif (size(fnet.inputs,2) == 1)
   res = fnet.inputs(1);
else
   res = fnet;
end
```

function [net] = CGPDecodeNode(chrom, node, level, totalNodes, numOfInputs, F, FTotal)

```
function [net] = CGPDecodeNode(chrom, node, level, totalNodes, numOfInputs, F, FTotal)

%pos = 0;
%for n=1:level-1
%    pos = pos+totalNodes(n)*(numOfInputs(n)+1);
%end
pos = sum(totalNodes(1:level-1).*(numOfInputs(1:level-1)+1))+(node-1)*(numOfInputs(level)+1);

func = chrom(pos+(numOfInputs(level)+1));


net = F(func,level);
if (level > 1)
   for n=1:numOfInputs(level)
      neuron = CGPDecodeNode(chrom, chrom(pos+n),level-1, totalNodes, numOfInputs, F, FTotal);
      net.inputs = [net.inputs neuron];
   end
end
```

function [param] = CGPDefaultParam(patternSize, numOfNodes, numOfInputs)

```
function [param] = CGPDefaultParam(patternSize, numOfNodes, numOfInputs)
```

```
%
%

if (nargin < 1)
   patternSize = 2;
end
if (nargin < 2)
   numOfNodes = [20 10 1];
end
if (nargin < 3)
   numOfInputs = [patternSize, 2, numOfNodes(2)];
end;
param.numOfNodes =  numOfNodes;
param.numOfLayers = 3;
param.numOfInputs = numOfInputs;
```

function [chrom, val] = CGPEval3(chrom,opts)

```
% Version 1.0
% Fitness Function
% Inputs-
%   chromosomeIn - chromosome to be evaluated
%   evalOps -
% Outputs:
%   chromosomeOut - (must be the same as the input)
%   fitness  = how good is the organism
%   fitness = (1/N)(total patterns classified correctly)/total Test Patterns

function [chrom, val] = CGPEval3(chrom,opts)
global class0;
global class1;

global F;
global FTotal;
global param;

net = CGPDecodeNet(chrom(1:end-1), F, FTotal, param);

if (size(net.inputs,2) == 0)
   val = 0.0;
   return ;
end

evalClass0 = evalMorphologicalPerceptron(net, class0);
totalCorrectClass0 = size(find(evalClass0==0),1);

% Evaluate patterns of class1
evalClass1 = evalMorphologicalPerceptron(net, class1);
totalCorrectClass1 = size(find(evalClass1==1),1);


fitness = (totalCorrectClass0+totalCorrectClass1)/(size(class0,1)+size(class1,1));

maxBranches = param.numOfNodes(end-1);

totalLayers = getTotalLayers(net);
%if (totalLayers== 3)
   totalUsedBranches = 0;
   totalBranches = size(net.inputs,2);
   for in=1:totalBranches
      chromIn = net.inputs(in);

      evalClass0 = evalMorphologicalPerceptron(chromIn, class0);
      totalCorrectClass0 = size(find(evalClass0==0),1);
```

```
    if (totalCorrectClass0 ~= 0 && totalCorrectClass0 ~= size(class0,1))
       totalUsedBranches = totalUsedBranches +1;
    end
  end

  branchPercent = totalUsedBranches/totalBranches;

  fitness = (fitness)*branchPercent *(totalBranches/maxBranches);

%elseif (totalLayers == 2)
%end

val = fitness;
```

function [done] = CGPFitnessFoundTerm(ops, bPop, endPop)

```
function [done] = CGPFitnessFoundTerm(ops, bPop, endPop)
currentGen = ops(1);
maxGen = ops(2);

done = (currentGen >= maxGen) || (bPop(end,end) == 1.0);

%bPop(end,end)
```

function [mutated] = CGPMultiPointMutation2(parent,bounds,Ops)

```
function [mutated] = CGPMultiPointMutation2(parent,bounds,Ops)
global param;

mutated = parent;

mutProb = Ops(2);
mutProb_Branches = 0.24;
mutProb_Inputs = 0.94;
mutProb_Weights = 0.90;

%if (rand < mutProb)

  %------------------------------ Mutate Operation/Weights

  if (mutProb_Weights > rand)
    %totalMut = round(rand*6);
    sz = size(parent,2)-1;
    totalNodes = sum(param.numOfNodes(1:end-1));
    if (size(Ops,2) < 3)
       totalMut = fix(totalNodes*0.08);
    else
       totalMut = fix(totalNodes*Ops(3));
    end

    for n=1:totalMut

       pos = round(rand*(totalNodes-1))+1;
       %pos = round(rand*(6-1))+1;

       if (pos > param.numOfNodes(1))
          t = pos;
          pos = param.numOfNodes(1)*(param.numOfInputs(1)+1);
          pos = pos + (t - param.numOfNodes(1))*(param.numOfInputs(2)+1);
       else
           pos = pos*(param.numOfInputs(1)+1);
       end
       %pos = pos*(param.numOfInputs+1);
```

```
        %cs = cumsum(numOfNodes);
        %p = find(cs < pos)
        %pos = sum(param.numOfNodes(p).*param.numOfInputs(p))+  (pos -
cs(size(p,2)+1)*param.numOfInputs(size(p,2)+1));



        mutated(pos) = round(rand*(bounds(2,pos) - bounds(1,pos))) + bounds(1,pos);
      end
   end

   %-------------------------- Mutate neuron inputs
   if (mutProb_Inputs > rand)
      totalNodes = sum(param.numOfNodes(2));
      if (size(Ops,2) < 4)
         totalMut = fix(totalNodes*0.15);
      else
         totalMut = fix(totalNodes*Ops(4));
      end
      for n=1:totalMut

         pos = round(rand*(totalNodes-1));
         pos = (pos)*(param.numOfInputs(2)+1) + (param.numOfInputs(1)+1)*param.numOfNodes(1);
         pos = 1+pos + round(rand *(param.numOfInputs(2)-1));

         mutated(pos) = round(rand*(bounds(2,pos) - bounds(1,pos))) + bounds(1,pos);
      end
   end
   %------------------------------ Mutate branches
   if (mutProb_Branches > rand)
      totalNodes = sum(param.numOfNodes(1:end-1));
      if (size(Ops,2) < 4)
         totalMut = fix(totalNodes*0.15);
      else
         totalMut = fix(totalNodes*Ops(4));
      end
      sz = param.numOfNodes(end-1);
      for n=1:totalMut

         pos = round(rand*(sz-1))+1;
         pos = pos+sum((param.numOfInputs(1:end-1)+1).*param.numOfNodes(1:end-1));

         mutated(pos) = round(rand*(bounds(2,pos) - bounds(1,pos))) + bounds(1,pos);
      end
   end
%end
return
```

function [o1, o2] = CGPMultipointXover(p1, p2, bounds, Ops)

```
function [o1, o2] = CGPMultipointXover(p1, p2, bounds, Ops)
global param;

%o1 = p1;
%o2 = p2;

xRate = Ops(3);
numOfBits = 3;
%numVar = size(p1,2)-1;
numVar = sum(param.numOfNodes(1:end-1));
xRatePerGen = xRate/numVar;
%fprintf('xRatePerGen prob %d\n',xRatePerGen);

pos = 0;
o1 = [];
o2 = [];
for n = 1: numVar
```

```
    if (rand < xRatePerGen)
        s = p1((pos*numOfBits)+1:(pos+1)*numOfBits);
        o1 = [o1 s];
        o2 = [o2 p2((pos*numOfBits)+1:(pos+1)*numOfBits)];
    else
        s = p1((pos*numOfBits)+1:(pos+1)*numOfBits);
        o2 = [o2 s];
        o1 = [o1 p2((pos*numOfBits)+1:(pos+1)*numOfBits)];
    end
    pos = pos +1;
end
if (rand < xRatePerGen)
    s = p1((numVar*numOfBits)+1:end-1);
    o1 = [o1 s];
    o2 = [o2 p2((numVar*numOfBits)+1:end-1)];
else
    s = p1((numVar*numOfBits)+1:end-1);
    o2 = [o2 s];
    o1 = [o1 p2((numVar*numOfBits)+1:end-1)];
end
%c1 = [c1 p1(numVar+1)];
%c2 = [c2 p2(numVar+1)];
o1 = [o1 0];
o2 = [o2 0];
```

function [F, FTotal] = CGPInitialize(patterns, param)

```
function [F, FTotal] = CGPInitialize(patterns, param)

dim = size(patterns,2);
F1 = CGPGenerateNodesForPatterns(patterns);
if (0)
F2 = CGPGenerateNodesForPatterns(zeros(1,dim));
else
net.op = 1;  % max
net.r = ones(1,param.numOfInputs(2));
net.weights = zeros(1,param.numOfInputs(2));
net.inputs = [];
F2 = net;
net.op = 0;  % min
net.r = ones(1,param.numOfInputs(2));
net.weights = zeros(1,param.numOfInputs(2));
net.inputs = [];
F2 = [F2; net];
end

net.op = 0;  % min
net.r = [];
net.weights = [];
net.inputs = [];
F3 = net;

F = F1;
FTotal = [size(F1,1), size(F2,1), size(F3,1)];

F2 = [F2; struct('op', cell(size(F1,1)-FTotal(2),1), 'r', {[1]}, 'weights', {[1]}, 'inputs', {[]})];
F3 = [F3; struct('op', cell(size(F1,1)-FTotal(3),1), 'r', {[1]}, 'weights', {[1]}, 'inputs', {[]})];
F = [F1 F2 F3];
```

function [initialPop, bounds] = CGPGeneratePop(popSize, evalFN, evalOps)

```
function [initialPop, bounds] = CGPGeneratePop(popSize, evalFN, evalOps)
```

```
global class0;
global class1;
global F;
global FTotal;
global param;

%chromosomeSize = (param.numOfLayers-1)*(param.numOfNodes*(param.numOfInputs+1))+param.numOfNodes+1;

initialPop = [];
bounds = [];

% compute boudaries for each variable in the chromosome
   pos = 1;
   layer = 1;
   for nodes=1:param.numOfNodes(1)
      i = 1;
      for inputs=1:param.numOfInputs(1)
         bounds(1,pos) = inputs;
         bounds(2,pos) = inputs;
         pos = pos+1;
         i = i+1;
      end
      bounds(1,pos) = 1;
      bounds(2,pos) = FTotal(layer);
      pos = pos+1;
   end
   layer = layer+1;
   i =1;
   for lyr=layer:param.numOfLayers-1
      for node=1:param.numOfNodes(lyr)
         for input=1:param.numOfInputs(lyr)
            if (0)
               bounds(1,pos) = i;
               bounds(2,pos) = i;
               i = i+1;
               if (i > param.numOfNodes(lyr))
                  i = 1;
               end
            else
               bounds(1,pos) = 1;
               bounds(2,pos) = param.numOfNodes(lyr-1);
            end
            pos = pos+1;
         end
         bounds(1,pos) = 1;
         bounds(2,pos) = FTotal(lyr);
         pos = pos+1;
      end
      layer = layer +1;
   end

   for nodes=1:param.numOfNodes(end-1)
      if (0)
         if (nodes <=3)
            bounds(1,pos) = 1;
            bounds(2,pos) = 1;
         else
            bounds(1,pos) = 0;
            bounds(2,pos) = 0;
         end
      else
         bounds(1,pos) = 0;
         bounds(2,pos) = 1;
      end
      pos = pos +1;
   end
   bounds(1,pos) = 1;
   bounds(2,pos) = FTotal(layer);
```

```
% generate population
for n=1:popSize
    pos = 1;
    v = size(bounds,2);
    for n=1:v
        org(pos) = round((bounds(2,pos)-bounds(1,pos))*rand +bounds(1,pos));
        pos = pos+1;
    end
    org(pos) = 0;   % temporary fitness value
    e1str = ['[org, fitness]=' evalFN '(org,[0 evalOps]);'];
    eval(e1str);
    org(pos) = fitness;

    initialPop = [initialPop; org];
end

return
```