Neural Networks and Evolutionary Algorithms

Diplomarbeit

Universität Rostock, Institut für Informatik

Vorgelegt von	:	Huascar Fiorletta
Geboren am	:	27.11.1980 Roma
Gutachter	:	Prof. Dr. rer. nat. habil. Van Bang Le
		Prof. Dr.-Ing. habil. Peter Forbrig
Abgabedatum	:	16.09.2006

Abstract

This work contains an introduction to the world of Neural Networks followed by the basics of Evolutionary algorithms. An overview of the biological inspiration of both fields as well as their theoretic foundations is presented. A review of the most influential advances in evolutionary algorithms is also offered keeping a critical approach to a field which is extremely broad and in continuous expansion. Finally the JoonePad and JooneFarm projects are presented, introducing the reader to a user friendly tool that aspires to be an experimentation workbench for neural networks and evolutionary algorithms enthusiasts enabling them to evolve neural network topologies.

Keywords: evolution strategies, genetic algorithms, neural networks, Joone, JoonePad, JooneFarm.

Table of contents

List of abbreviations

BP	Backpropagation
EA	Evolutionary algorithm
ES	Evolution strategy
GA	Genetic Algorithm
GUI	Graphical user interface
HW	Hardware
MVC	Model View Controller
NN	Neural network
PDA	Personal digital assistant
RMSE	Root mean squared error
SA	Simulated Annealing
SW	Software
TDS	Training data set
TSP	Traveling salesman problem
XML	Extensible Markup Language
XOR	Exclusive disjunction

Motivation and goals

From time immemorial man has always exploited nature as a source of inspiration in order to solve the challenges he faced. The airplane and the sonar as well as the neural network and the genetic algorithm are only part of this observation-and-analysis process. Consequently the motivation behind this work is a deep interest in two of the latest, most promising and fascinating technologies that natural inspiration has ever provided: evolutionary algorithms and neural networks. Following his beliefs in the principles of open source and free content the ultimate goal that the author had in mind while at work was the creation of a text that might introduce people with little or no knowledge in the world of evolutionary computation and neural networks. In the hope that the reader might become interested and read further about this new technologies, the author tried to develop a stimulating approach to a branch dominated by often very technical publications. The work includes the next steps:

- The reader will be first introduced to the basics of evolutionary algorithms and those of neural networks.

- Subsequently a review of the literature enclosing remarks, comments and comparisons will be presented.

- Finally the author's approach to the development of a framework for the evolution of neural network topologies will be illustrated.

Introduction

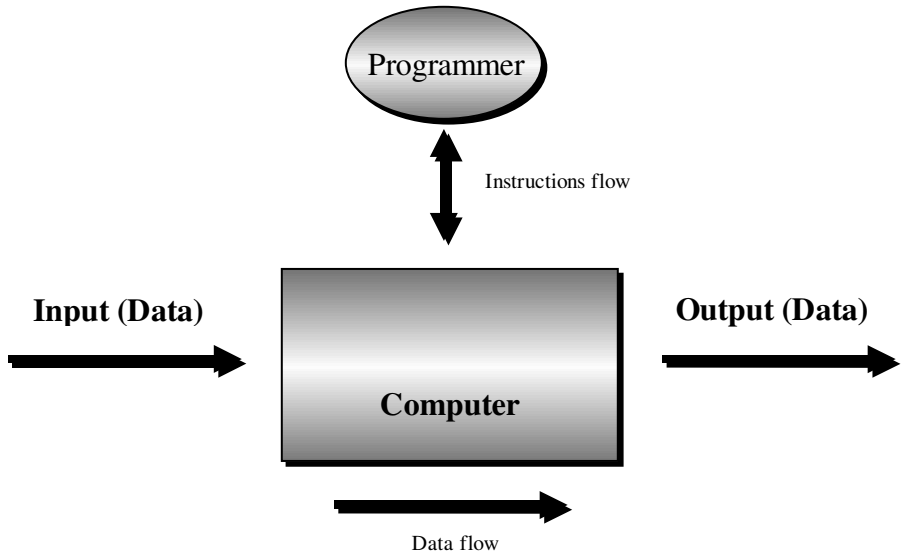
The process of diversification has always been part of any evolving knowledge acquisition. So at the beginnings philosophy embraced physics, rhetoric and theology; medicine divided itself into numerous fields and mathematics saw the birth of informatics.

As history demonstrates, with the passing of time there has always been a subdivision of disciplines into branches that went deeper in examining particular aspects of the original field of study.

We all know computer science is a quite young science and we should be aware that we are not far away from the steam motor or the bow if compared to their respective disciplines. Many say that after the industrial revolution we are entering the digital or electronic revolution, but again, we are just "entering" and it would make sense to realize that computer science is a discipline that it is slowly branching in many domains.

Today's software development is mainly a handmade process. Surely in a so young science the distinction between artisans and intellectuals is still blurred. So what is a programmer? He might be a simple translator, the interpreter between his boss' wishes and the computer. But he might also be an inventor, creating new programs or he might be a scientist, elaborating theories. But in the end he always will have to pass instructions to the computer in a handmade fashion.

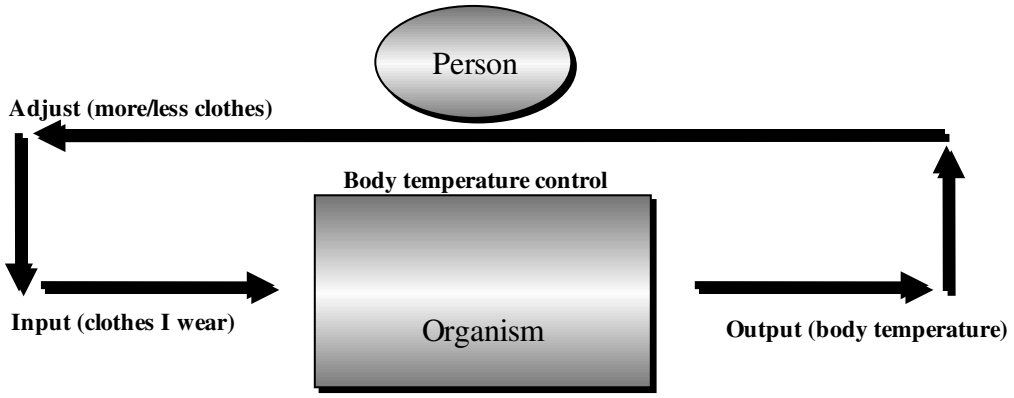
The point is, a computer is a black box which receives inputs from one side and delivers outputs on the other. The programmer intervenes in such a process by giving the computer a second input that could be called the "Instructions flow", to distinguish it from the "Data flow". The instruction flow teaches the computer on how to handle the original input information ("Data flow") in order to obtain the desired output.



To most of us it is clear that there is a great difference between Assembler and Java, this is because with the passing of time higher degrees of abstraction are reached in the Instructions flow, making the programmer's life easier and allowing them to elaborate more complex data flows. So we could consider the ever bigger image files as an increased data flow, while the introduction of Xml interfaces like Xul or SwiXML would represent a more abstract level in the Instructions flow.

In such a model the instructions flow might vary in quality and quantity but remains a constant of this model, meaning that in the end it is always a person who must instruct the computer about how to elaborate the data flow to achieve a certain goal, no matter how easy or hard this task is.

In contrast the animal brain functions in a totally different way. The day by day basic empiric learning process is based on a trial basis (**T**est**O**perate**T**est**E**xit model) where the output is compared with the input, and then the input is modified in order to achieve the desired result. The regulation of our body temperature is a basic example. One person that is dressing himself will check if he feels a comfortable temperature (the test), in case he realized a discomforting feeling because his body's temperature is too high or too low he'd try to put something on or to take something off (operation). After changing his clothes the person would wait again some time in order to check again his body's temperature. If the situation had worsened he would act in the contrary direction he did last time, if instead the temperature improved but not enough, he would again repeat his last action; then another cycle of test and operation would begin until a satisfactory result is reached.



It is from the abstraction of these empirical experiences that a person is in condition to elaborate knowledge in form of generalized statements. Phrases like "put your pullover on if it's snowing outside" are the result of experience, which might come from an empirical experiment ("I got a cold last winter") or information received from someone else ("Mom told me so"). This knowledge can later be used as basis in order to speculate about possible events in the future and to create generalized conclusions like "if a person goes in the snow without sufficient clothes it's almost certain he will get ill". Such conclusions can later be transmitted to other persons through a more or less formal language; so when a doctor communicates with another doctor he uses a specialized vocabulary to describe the same things a common person would describe in simple and often vague words.

This transmission of knowledge will speed up the information acquisition process of the recipient person by eliminating the necessity of an empirical experiment for every concept that they acquire, an example: we BELIEVE in atomic energy because someone explained to us what it is or because we read a book, but in the end very, very few persons have assisted to a demonstration of it.

Often the communication of our knowledge has forms that can be represented by a flow chart, and therefore translated in a sort of meta-programming algorithm (when you...then… and if you…); while for most people the recipient of the communication is represented by another person in the case of a programmer it corresponds with the computer, being programming languages the formalisms used for the transmission in the latter case.

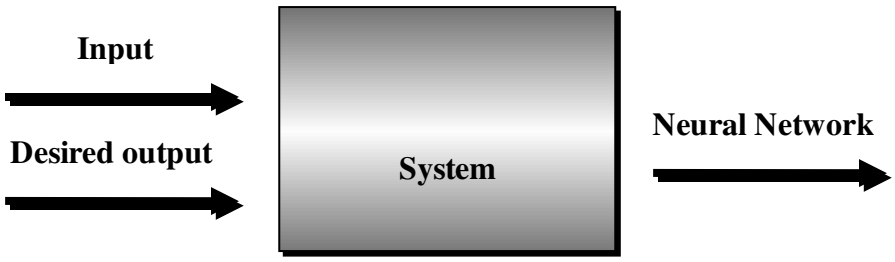
If we ever want a machine to make a work "instead" of us and not "for" us, implying the absence of a human being as instructor or controller and therefore the absence of the "instructions flow", this machine will have to have the same capabilities that we posses or even more. This involves, among others, the potential of learning and solving new unexpected problems. Today's computers are not in the position to completely replace a human being, partly (someone would say) because of their technological level partly because of their

intrinsic structure, so different from a human brain. Neural networks represent a different approach to the need of emulating human behavior and capabilities in machines.

Let us now imagine a system “alpha” that would be able to solve more problems than a system “beta” without the need of additional instructions, we would certainly call alpha more intelligent than beta. Let’s be x the amount of instructions that we give to alpha and beta, x could be measured in programming time, basic operations or code lines but what we need is just a measure for comparison so let’s use the basic operations. Using x instructions it could be possible to teach beta how to resolve a determined number of problems (y combinations of the data flow), but using the same x instructions it could be possible to teach alpha how to learn to solve problems using the information of the data flow. Now, given a sufficiently big data flow, alpha would end up being able to handle more problems than beta, being both given an amount x of instructions; this would render alpha more intelligent than beta according to our previous definition.

We must be aware that also neural networks have an analogy with the classic computer programming, both of them have a “data flow” and an “instructions flow”, that is the necessity of a designer, a person that creates or instructs them. If only it could be possible to eliminate the “instructions flow”, or to restrict it to a minimum by delegating as much knowledge acquisition as possible to the “data flow” we would generate a system with a higher degree of intelligence, a system producing the same results but with a smaller instructions flow (maintaining the precondition of a sufficiently big data flow).

Nowadays the most common practice is to train a neural network with patterns; we could consider the patterns as the data flow and the creation of the neural network as the instructions flow. This technique already represents a step forward in comparison to the classic programming because the neural network is able to determine the best weights for its synapses with help of the input patterns. Never the less the topography of the net must be determined by the user, and this represents a non trivial instructions flow. As said before our goal is to reduce the instructions flow and obtain the same or a better system, there are two choices in this case. The first would be to use a higher formalism that would reduce the amount of work done by the user, for instance the creation of a new neuron type with a more complicated transfer function. The second possibility would be the reduction of the instructions flow by delegating part of the network creation to the data flow. This can be achieved if we have another system that would return us a neural network starting from a determined data flow (input and output).



It is in this second case that genetic algorithms have proved useful and in the following chapters it will become clear why.

Chapter 1

Basics of Neural Networks

1.1 - Why neural networks?

Human vs. computer

Without any doubt computers have achieved in certain areas efficiency levels which are simply inconceivable for human beings; their speed of calculus ridicules the most skillful man, their possibility of storing huge amounts of information, such as images for example, over long periods of time without loss or degradation is something man hasn't reached yet. Scalability is also a very interesting feature, adding memory to a computer is relatively simple, but expanding the capabilities of a human brain is not that easy. Even considering the reaction time of a today's computer memory we would discover that it is in the lap of the nanoseconds while the human brain is still far away from that.

But on the other side humans still have some advantages over computers. Although the speed of a single neuron is quite small (about 100 Hz), the particular structure of our brain enables us to perform much better than computers in certain kinds of operations. Humans are quite good at recognizing patterns; the classic example for this would be the "party conversation". Even if in a party a person will acoustically perceive a large number of voices at the same time, he will still be able to follow a conversation with his immediate neighbor, this because of the filtering capabilities of our brain. A similar case is the distinction between figure and background, where a visual stimulus is elaborated in order to extract information. We perform all these tasks without thinking about it, not even perceiving them as a burden. Only when an optical illusion is presented to us or when we discover a camouflaged animal, we realize the way we analyze our sensorial information. The difficulty of analyzing an optical illusion might let us crack a smile, or, in the case of the camouflaged animal, squeeze our eyes but what is most important is the perception of performing a difficult task of which we were not aware.

The reason behind the differences

The origin of the differences between a human brain and a computer are to be found in their different structures. Our brain works massively parallel, it has something like 100 billion neurons each one of them deeply connected with some other 10 thousand neurons. Each neuron is composed of three parts.

1. A cell body, also called soma, which contains the cell nucleus and is responsible for the creation of the proteins needed by the axon, as well as for the vital activities of the cell.
2. The dendrites which form a branching structure around the cell. They convey electrical stimulations coming from the neighboring cells to the centre of the neuron. It's easy to see them as the "input" of the neuron.
3. An axon. The "output" of the neuron. A thin but very long extension of the cell needed to transport the electrical impulses of the neuron to other cells. It can be longer than a meter, an amazing distance if compared with the diameter of the single neuron (from 0.004 till1 mm).

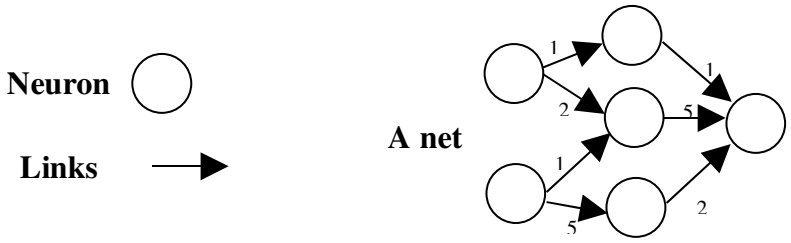
As anybody knows the computer's central processing unit is composed of millions of transistors, which are quite different from a neuron. In addition a computer is composed of specialized peripherals each one of those performing a specific task such as storage or communication. Computer work mainly sequentially or with a quite restricted amount of parallelism.

It is therefore clear how different the structure of a human brain in comparison with a computer can be. In order to efficiently fulfill those tasks where computers still lay behind humans a new approach had to be found. This is the reason why artificial neural networks were created. Such networks are constituted by artificial neurons connected to one another with the final goal of emulating the functioning of a biological network. The purpose of artificial neural networks is not to copy the normal neuron functioning, rather they try to create a model of the behavior of the original neuron. It would be not only impossibly complicate but also senseless to copy every function of a biological neuron; we wish to deal with an essential model only, a model that has more or less the same relations a geographic map has with reality. It is therefore wrong to expect that an artificial neuron is capable of accomplishing the tasks of a biological one. We must always bear in mind that we will be dealing with models.

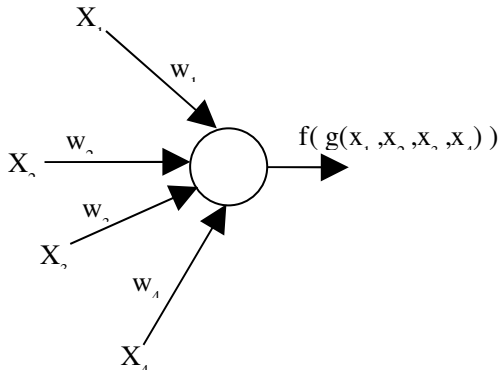
1.2 - What a neural network is

In the literature there are many definitions for a neural network which often vary considerably from one to the other, it is therefore necessary to understand that the following definition of neural network can only be approximate and incomplete; nevertheless it might be useful in order to get at least a rough idea of what neural network is.

A neural network is a model including information processing units called neurons that are connected through links. Such links, also called synapses, have a direction and a weight. While the direction of the propagating signal is determined by the direction of the arrow that represents the link, its weight determines a multiplication factor for the signal transmitted through it and is depicted as a small number over the link.



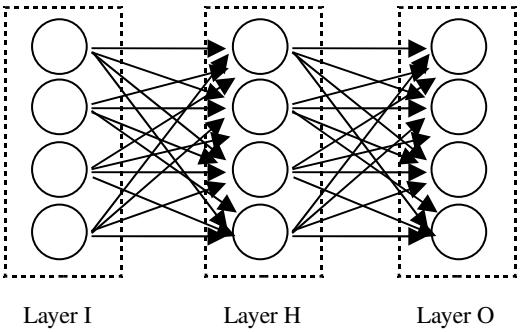
Another important element of a neural network is the integration function, each neuron has a function $g(x_1, x_2, \ldots, x_n)$ where x_n is the n^{th} signal that the neuron receives through a link. This function transforms all the n input values x into a single value.



The activation function (also output function) f() instead, determines the output of the neuron starting from the single value provided by the g()function.

1.3 - Network topology

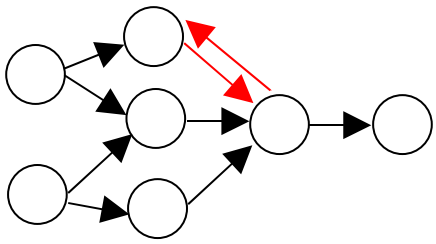
It is obvious that neural networks composed of a single neuron have a quite limited utilization; in order to achieve a certain grade of functionality it is therefore necessary to link several neurons to one another. The term layer denotes a group of neurons that have the same characteristics, the same activation and integration functions as well as the same links to other neurons. This implies that if a neuron of the layer H is connected with one neuron of the O layer all the neurons of the layer H have a link to each of the O layer neurons. Otherwise there would simply be no connection between any of the neurons of the two layers.



Generally neural networks have an input layer, an output layer and one or more hidden layers. The input layer is a layer (here the "I" layer) whose activation function is the input of the net. It is important to note that input layers don't elaborate information, therefore they are not considered when counting the layers that compose a net. The output layer is the last one and receives the results of the net (here shown as the "O" layer). The layers that are found between the input and the output layer are referred to as hidden (hence the "H" name for the middle layer).

There is another criterion for the classification of a network's topology. Nets can be subdivided into feed forward and recurrent networks. Feed forward networks do not contain loops in their graphs; this means that all the signals flow from the input layer towards the output layer. All the nets depicted up to this point were feed forward. If one would follow the direction of the links imaginarily tracing the route of a signal, it will soon be clear that in a feed forward net it is not possible to reach again a neuron belonging to a layer where the signal already transited. To the contrary, a recurrent network is a network that has a cycle

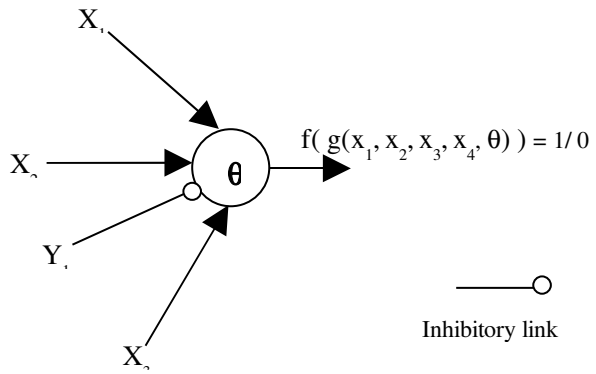
inside of its topology. This means that a signal could pass more than once through the same neuron. The following illustration shows a recurrent network where the red arrows indicate the cycle.



Recurrent networks are clearly more complicated than feed forward networks. In a neural network, each neuron computes a certain function using as arguments all its x_n inputs at a time i, let's call this function $f_i(x_1,x_2,\ldots,x_n)$. Although a loop needs not to be direct and can include many neurons, we will suppose for simplicity reasons, that our neuron has a direct loop that connects its output to its own input. In such case $f_i()$ has as arguments not only the n synapses but also $f_{i-1}(x_1,x_2,\ldots,x_n)$, therefore $f_i()$ becomes then $f_i(x_1,x_2,\ldots,x_n,f_{i-1}(\,)\,)$. This means the existence of recurrent functions. Since recurrent networks have a wider range of problems such as when to stop the computation, we will subsequently focalize only on feed forward networks.

1.4 - The McCulloch-Pitts neuron

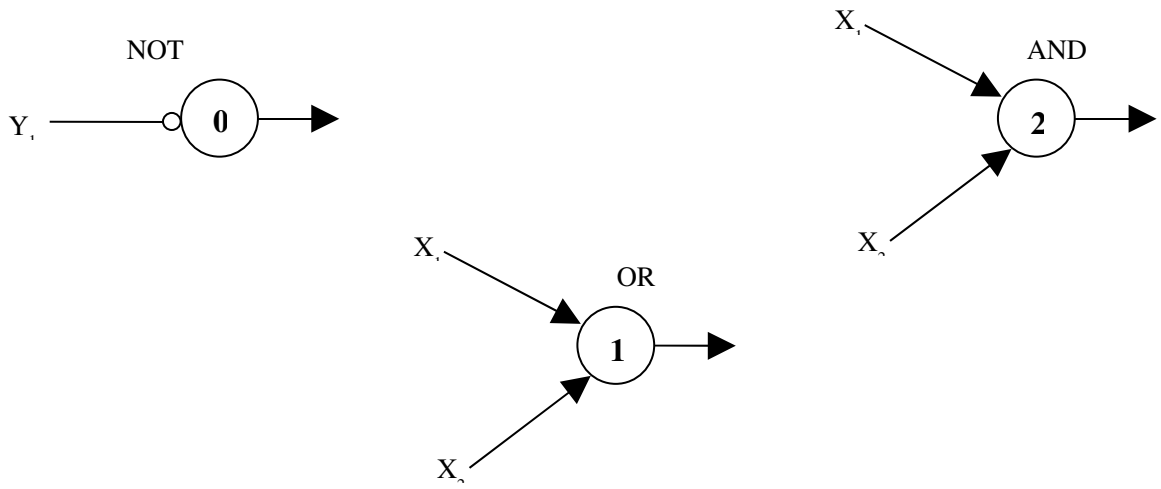
The oldest and probably the easiest neuron to understand is the McCulloch-Pitts neuron. In the first years of the 1940's Warren McCulloch and Walter Pitts introduced what we regard today as the first neural networks. Those networks were composed of neurons that performed a relatively simple logical operation. To start with, the output of a McCulloch-Pitts neuron is either 1 or 0, this is called a binary activation function. All the links of the neuron are directed and not weighted; they are also subdivided into excitatory and inhibitory links. If one or more inhibitory links are active the neuron has an output of 0, said in other words, it doesn't fire. If no inhibitory link is active, all the active input links are summed and compared with the threshold value θ, if their value is bigger than θ then the neuron fires, this is, its output is 1. This is the reason why McCulloch-Pitts neurons are represented by a circle containing the threshold value.



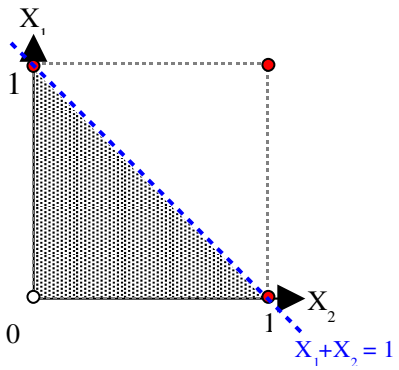
- Excitatory links = X_1, X_2, X_3
- Inhibitory Links = Y_1
- Threshold = θ
- Integration function $g(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i$
- Activation function $f(g(x_1, x_2, x_3, x_4, \theta)) =$

$$\begin{cases} 1 & \text{If } = 0 \text{ and } \geq \theta \\ 0 & \text{If } > 0 \text{ or } < \theta \end{cases}$$

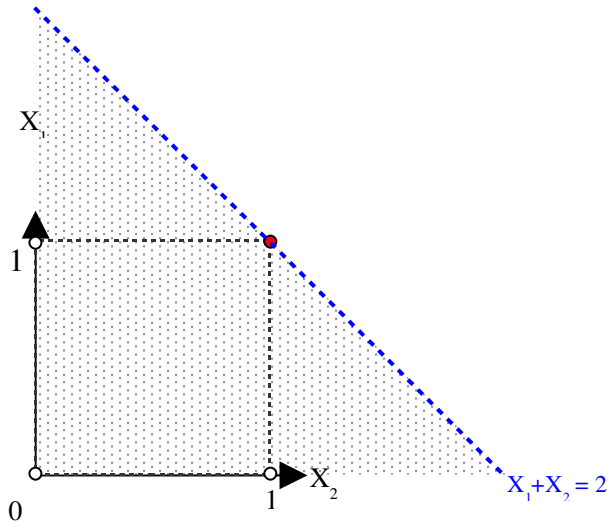
It is easy to recreate basic logical function such as “and”, “or”, “not”. In this case only a neuron is sufficient.



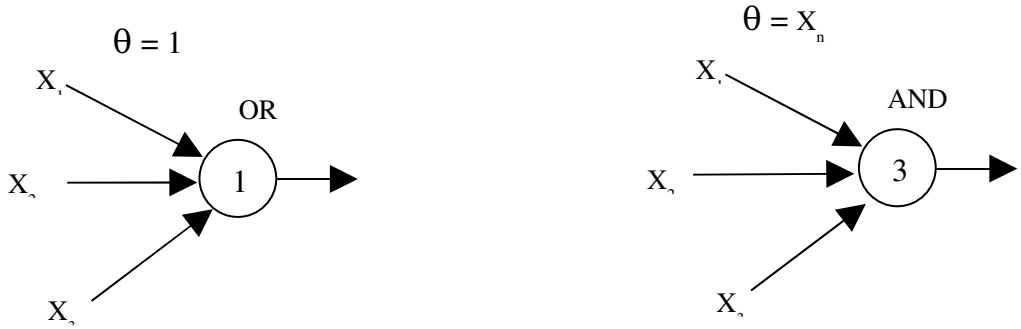
Please note that networks that use McCulloch-Pitts neurons are limited to binary transmission of data, in other words, on its synapses travel only ones and zeros; consequently such nets can only elaborate binary functions. Given that the geometrical interpretation of a McCulloch-Pitts neuron will be referenced quite often it is appropriate to introduce it now. Let's imagine a McCulloch-Pitts neuron with only two input synapses, x_1 and x_2. The threshold value θ divides the space in two according to the equation $\sum_{i=1}^{n} x_i \geq \theta$. The illustration below shows the division of the two dimensional space for a value of $\theta = 1$ which is the case of the "or" function. It is important to note that in this case the diagonal of the square (marked with blue) is part of the upper triangle, which is the area where the neuron fires. In fact the sum of all the input synapses must be bigger or equal to zero. Since the blue line represents the threshold we will call it threshold line from now on.



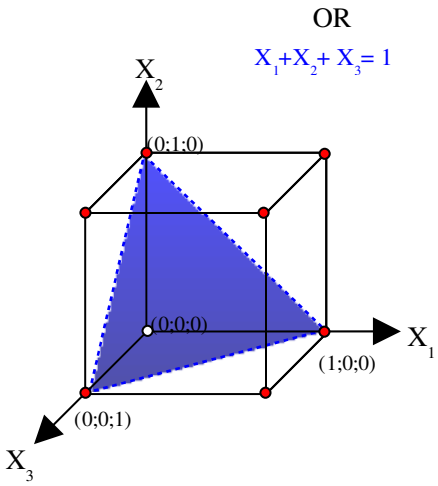
The "and" function is similar to the "or" function but with a threshold value of two, this implies that the threshold line x_1+x_2 has been shifted outwards. Here again the threshold line is part of the area where the neuron fires. The intersection of the firing area and the square representing all the possible inputs of the neuron returns only one point (1;1). This point is the only combination that lets the "and" neuron fire.



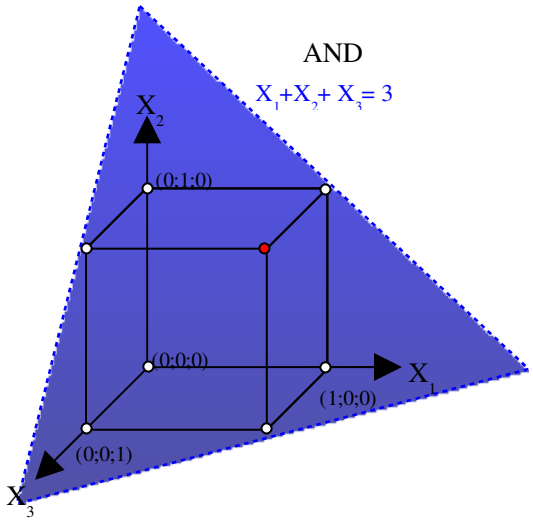
The case of a neuron with three different input synapses sees the threshold value of the "or" function unvaried while in the "and" neuron it becomes 3. It is now possible to determine the generic threshold value for a "or" neuron and for a "and" neuron which have n input synapses. The "or" neuron has always a constant threshold value $\theta = 1$ while the threshold of the "and" neuron is determined by the number of input synapses and is $\theta = X_n$.



This time the graphical interpretation of the McCulloch-Pitts neuron implies a 3D cube whose 8 vertexes represent all the possible combinations of the input synapses. The blue triangle represents the border of the threshold condition $X_1+X_2+X_3 \geq 1$. Analogously to the 2D case, the space is divided into two half-spaces, the firing combinations, depicted by red points, and those combinations that cause the neuron output to be equal zero, depicted by white points.



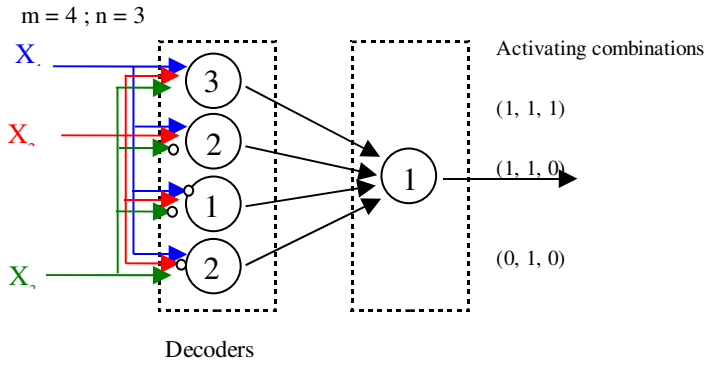
The case of an "and" neuron with three input synapses sees again the threshold condition shifting away from the origin. In this case the only firing solution has coordinates (1;1;1).



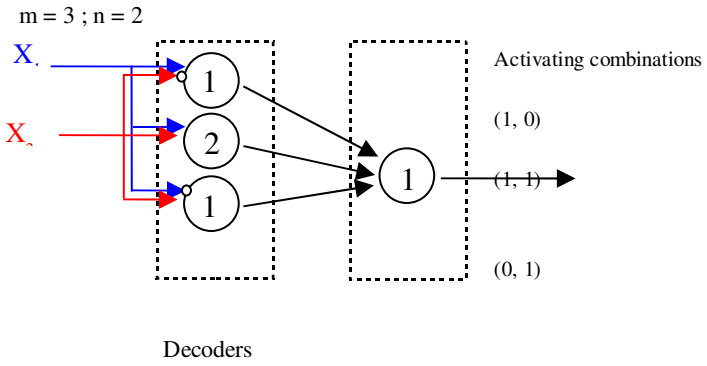
In order to represent the space of the solutions of a neuron with more than three input synapses, it would also be necessary to use a graph with more than three dimensions, unfortunately this is simply not possible. Consequently the concept of hyper plane will be used. A hyperplane is mathematically described as: "any codimension-1 vector subspace of a vector space". That means a vector having one dimension less than the space that contains it. A n dimensional hyperplane is described by a formula that looks like this: $a_1x_1 + a_2x_2 + ... + a_nx_n = b$. A hyperplane always divides the space into two different subspaces, we can

therefore imagine the McCulloch-Pitts neuron as a hyperplane in the n dimensional space where n is the number of input synapses.

It is interesting to note that a network of McCulloch-Pitts neurons with two layers can compute any logical function F: $\{0,1\}^n$ -> $\{0,1\}$. The easiest way to understand this is again graphically. The first layer of the decoding network is composed of m neurons, being m the number of input combinations for which the function F returns the value 1.



Every decoder is connected to a normal synapse if that variable is positive in the combination that the neuron is decoding. E.g. neuron number one decodes the combination (1, 1, 1) this means it has three synapses as input. But if the decoded combination includes zeros, the respective synapse is inhibitory. E.g. neuron number two decodes (1, 1, 0) therefore has an inhibitory synapse connected to the x_3 variable. The threshold value of each neuron is simply equal to the number of positive synapses it has. Each neuron is then connected to a single output neuron with threshold value 1.
The following diagram shows the codification of the "or" function according to this method.

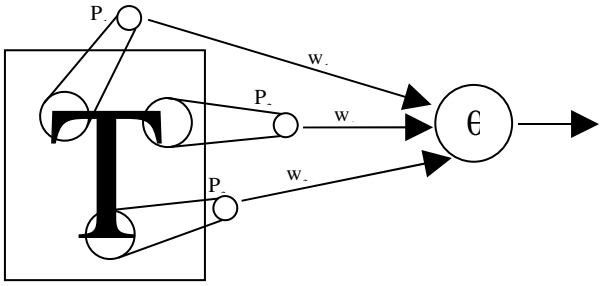


But as we know, the "or" function can also be calculated by a single neuron. In conclusion we have seen that a function can be represented by more than one network, those networks that represent the same function are called equivalent.

1.5 - The Perceptron

As already said neural networks can have weighted edges, such innovation was introduced in 1958 by Rosenblatt when presenting his perceptron model. The perceptron model was later studied and improved in the 60's by Minsky and Papert. While Rosenblatt designed the perceptron as a whole weighted network whose synapses where stochastically determined, and introduced other elements such as bounds to the maximum number of input links; the only difference between the McCulloch-Pitts and the Minsky Papert perceptron model is the fact that synapses are weighted. Please note that when talking about a perceptron we will refer to the Minsky Papert model.

The Minsky Papert Perceptron model was used in experiments where it had to read a "pattern" projected on a surface. In order to do this, some sensors, called predicates, where attached perceptron. The predicates would output only 1's and 0's to the perceptron and this would perform all the computations.



Let's now take a brief look to weighted links. A weighted link can be interpreted as a simplified notation for a branching link. Two synapses are equivalent if the weight of the one is equal to the number of branches that are attached to the neuron of the other. For example, the following two diagrams are equivalent.

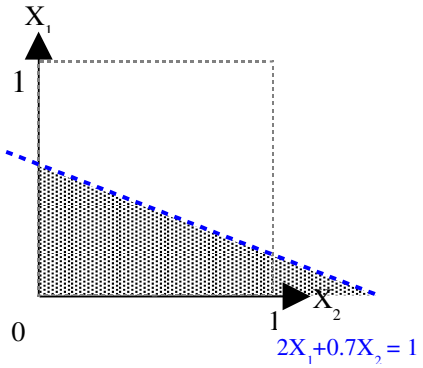
Using this interpretation of weighted link it is possible to understand the exact definition [1] of perceptron:

A simple perceptron is a computing unit with threshold θ *which, when receiving the n real inputs* $x_1, x_2, x_3, \ldots, x_n$ *through edges with the associated weights* $w_1, w_2, w_3, \ldots, w_n$ *outputs 1 if the inequality* $\sum_{i=1}^{n} w_i x_i \geq \theta$ *holds and otherwise 0.*

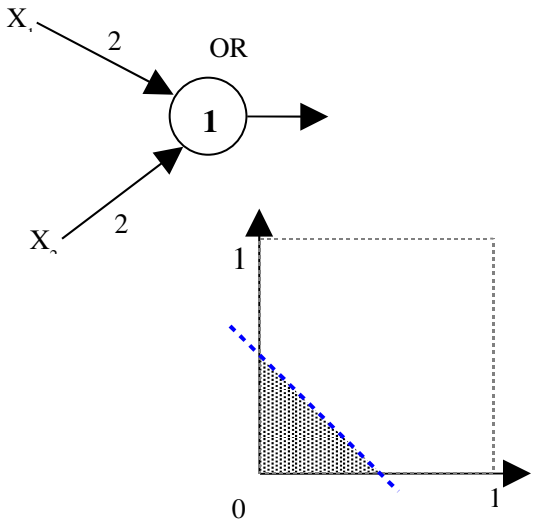
That is:

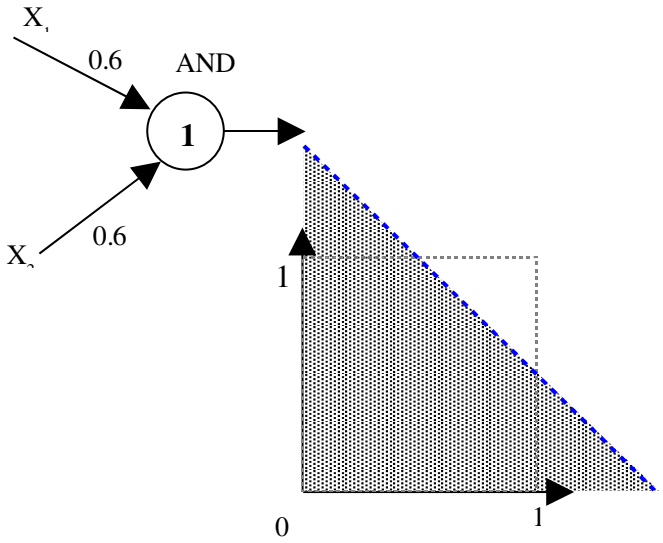
$$Y = \begin{cases} 1 & \text{If} \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

The graphical interpretation of the perceptron is similar to the McCulloch-Pitts neuron but the fact that the weights $w_1, w_2, w_3, \ldots, w_n \in \Re$ makes that the threshold line doesn't necessary pass through the vertexes of the square representing the input space.



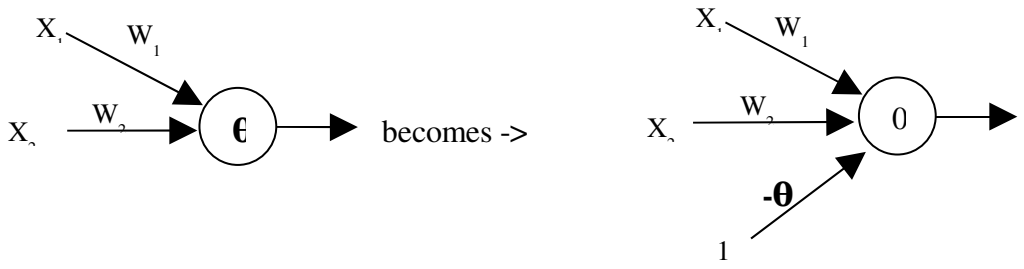
The interesting thing about weighted networks, besides a more compact representation, resides in the fact that a net can "learn" by changing the weights of its links. In fact the "or" perceptron and the "and" perceptron differ in the weights of their links but maintain the same topological structure.





By changing the weights of the links that form the input of the neuron and the threshold value it is possible to shift and rotate the threshold line. Since the weights are real values in the

figure above the threshold line doesn't pass through any of the input space vertexes. These weights can be manually set or even found through an algorithm. In case a learning algorithm is used, the threshold value is often set to zero and a negative link, the so called bias, is added to the perceptron. This simplifies significantly the work of the algorithm that deals with those weights.



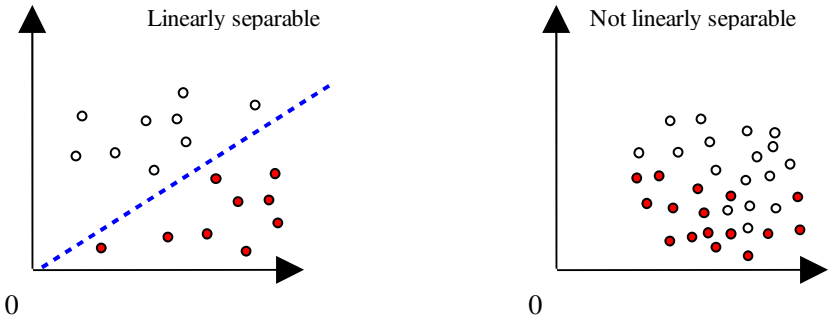
$x_1, x_2, x_3, \ldots, x_n \rightarrow x_1, x_2, x_3, \ldots, x_n, x_{n+1}$ with $x_{n+1} = 1$

$w_1, w_2, w_3, \ldots, w_n \rightarrow w_1, w_2, w_3, \ldots, w_n, w_{n+1}$ with $w_{n+1} = -\theta$

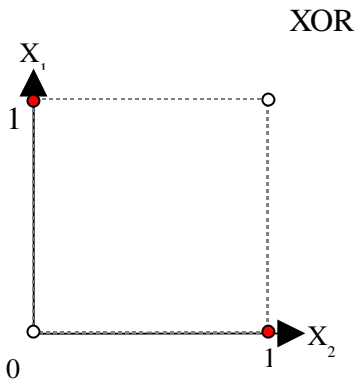
By now it should be clear that a single perceptron divides the solution space into two different areas, the first containing all the firing combinations and the other the remaining ones. Generalizing the graphical interpretation of a perceptron it is easy to see that changing the weights of the input synapses, it is possible to rotate and shift the line that separates the two spaces, the same is possible for the plane and the hyperplane that divide the input space of higher grades. Yet one perceptron is not capable to compute every function, only linear separable functions can be represented through a single perceptron. Linear separable are those functions where all the points that return a 1 can be separated from all the points that return zero respectively through a line, plane or hyperplane.

A more rigorous definition of linear separability [1] would be:

Two sets of points A and B in an n-dimensional space are called linearly separable if $n+1$ *real numbers* $w_1, w_2, w_3, \ldots, w_n, w_{n+1}$ *exist, such that every point* $(x_1, x_2, x_3, \ldots, x_n) \in A$ *satisfies* $\sum_{i=1}^{n} w_i x_i \geq w_{n+1}$ *and every point* $(x_1, x_2, x_3, \ldots, x_n) \in B$ *satisfies* $\sum_{i=1}^{n} w_i x_i < w_{n+1}$.



A typical not linearly separable function is the "xor" a brief look at its input space makes immediately clear that a single perceptron cannot compute such a function. No straight line can separate all the white points from the red ones.



The mathematical demonstration for this is the following:

Value		For the given x_1, x_2	How the result should be in
x_1	X_2	$w_1x_1 + w_2x_2 =$	comparison with θ
0	0	0	$0 < \theta$
1	0	w_1	$W_1 \geq \theta$
0	1	w_2	$W_2 \geq \theta$
1	1	$w_1 + w_2$	$w_1 + w_2 < \theta$

In the first column appear all the possible input combinations for the xor function, in the second column they are substituted in the integration function of the perceptron and finally compared with the threshold value in the third column. The inequalities of the third column that are marked with red contradict eachother, it is impossible to find two weights bigger than zero whose sum is smaller than the value of one of them. In brief, it is not possible to find

two weights w_1, w_2 for the links of the perceptron so that the xor function is correctly calculated.

1.6 - The perceptron learning algorithm

Neural networks do have a computational capacity but they are particularly remarkable because of their learning capabilities, they can be adapted to perform a certain task that they originally weren't able to. We know a neural network can be improved by changing the weights of its synapses. For this purpose all the weights of a neural network could be grouped in a vector of n dimensions, where n represents the number of weights that our neural network has:

$$w = (w_1, w_2, w_3, \ldots, w_n)$$

We can now divide the input space into two sets of vectors P and N. Where all the vectors of P should return 1 and all the vectors of N should return 0:

$$\forall x \in P \Rightarrow f_w(x) = 1$$
$$\forall x \in N \Rightarrow f_w(x) = 0$$

In case of a perceptron $f_w(x)$ is equal to:

$$w \bullet x \geq \theta$$

Starting from this we will now introduce the definition of positive and negative half space [1]. Please note that, like previously described, the notation used in this definition supposes that the threshold value θ has been set to zero and that an ever active link with weight $-\theta$ has been added.

The open (closed) positive half space associated with the n-dimensional weight vector w is the set of all points $x \in \Re^n$ *for which* $w \bullet x > 0$ $(w \bullet x \geq 0)$. *The open (closed) negative half space associated with w is the set of all points* $x \in \Re^n$ *for which* $w \bullet x < 0$ $(w \bullet x \leq 0)$.

In order to calculate how efficient our perceptron is in classifying the input patterns, we need a function that measures the number of errors our perceptron makes. The error function E(W)

tells us how many input patterns have been wrongly classified using the weights of the weight vector w:

$$E(w) = \sum_{x \in A} (1 - f_w(x)) + \sum_{x \in B} f_w(x)$$

Obviously in the best case E(W) is 0. This is the so called global optimum, the weight vector with the lowest error rate at all. It is a good idea to spend some time defining more precisely what a global optimum is and which difference there is between global and local optimums.
First we define a vector $w \in M = (w_1, w_2, w_3, \ldots, w_n)$ where w_n is a real parameter. Then we have to define a goal-function that will assign a real value indicating the quality to each vector $g(w) \rightarrow \Re$, we will use the error function as goal function. If there exists a x* so that:

$g(x^*) \leq g(x)\ \forall x \in M \rightarrow x^*$ is a global minimum
$g(x^*) \geq g(x)\ \forall x \in M \rightarrow x^*$ is a global maximum

The global optimum of a minimization problem corresponds to the global minimum, analogously in case of maximization it is the global maximum. In the case of the error function we are looking for a very low value (few errors) therefore we are searching a global minimum. Besides the concept of global minimum/maximum there is the idea of *local* minimum/maximum. In this case, if there exists some $\varepsilon > 0$ such that:

$g(x^*) \leq g(x)\ \forall x / | x^* - x | < \varepsilon \rightarrow x^*$ is a local minimum
$g(x^*) \geq g(x)\ \forall x / | x^* - x | < \varepsilon \rightarrow x^*$ is a local maximum

That is to consider all the vectors whose distance from x* is less than ε, practically we are looking in the "neighborhood" of x*.

In first instance it could be possible to calculate the weight vector manually, exactly like we have done for the "or" and the "and" perceptrons, unfortunately this might not only result a tedious task but rather contradictory too, as a matter of fact we are interested in neural networks because of their learning capacities, their ability to learn by example and adapt to new tasks. In order to achieve this without having to set the weights manually, a learning algorithm is needed. The first thought would be to randomly choose a new weight vector, but this cannot be defined as learning since all the previous information is not considered.

Then we could "train" the net, in order to achieve this we want to use the positive and negative reinforcement; that is the reinforcement of positive (or correct) behaviors and the suppression of negative (or wrong) behaviors, this notion has been derived from psychology. Children are punished every time they present a behavior that is not desired, such as lying for example, while on the other side they are given recompenses for good conduct like successful

studying; this is a mechanism that is deeply rooted in our society and is directly derived from nature itself. In the same way we desire that every time our net incorrectly recognizes a pattern the probability for this to happen again became lower by adjusting the weights.

The perceptron learning algorithm allows us to train a perceptron on the basis of two sets of vectors, P and N, in an n dimensional input space. All the vectors belonging to P are supposed to belong to the positive half space whereas all the vectors of N lay in the negative half space. Vectors can change over time and therefore the notation takes into account the time variable t.

Perceptron learning algorithm:

START:

The weight vector w_0 is randomly generated.

The time variable t is set to 0.

TEST:

A vector $x \in P \cup N$ is randomly selected

If $x \in P$ and $w_t \bullet x > 0$ go to TEST.

If $x \in P$ and $w_t \bullet x \leq 0$ go to ADD.

If $x \in N$ and $w_t \bullet x < 0$ go to TEST.

If $x \in N$ and $w_t \bullet x \geq 0$ go to SUBTRACT.

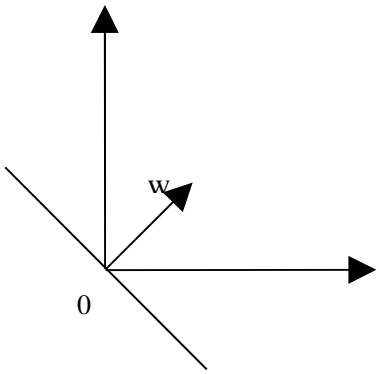
ADD:

Set $w_{t+1} = w_t + x$ and t = t + 1, go to TEST.

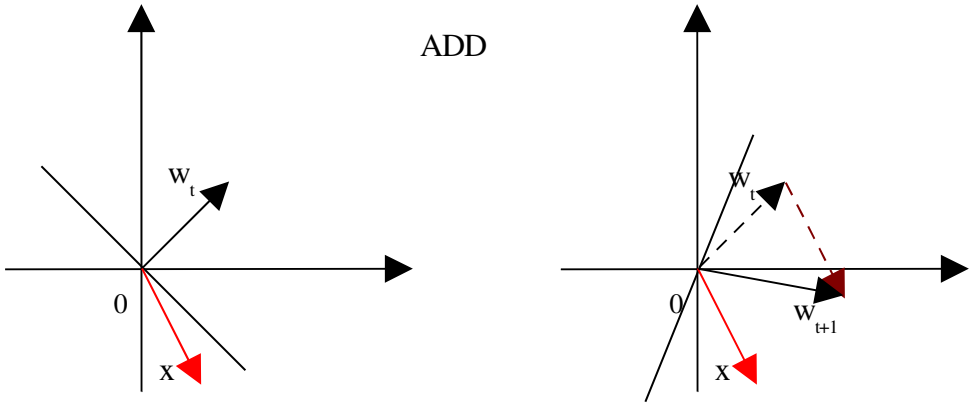
SUBTRACT:

Set $w_{t+1} = w_t - x$ and t = t + 1, go to TEST.

The graphical interpretation of this algorithm is best visualized when the weight vector has only two components. Reminiscent of the algorithm, our first weight vector will be randomly chosen. The threshold condition $w \bullet x \geq 0$ is true for all those vectors whose angle with w is smaller than 90°, this is the half space that in our graph is marked red.

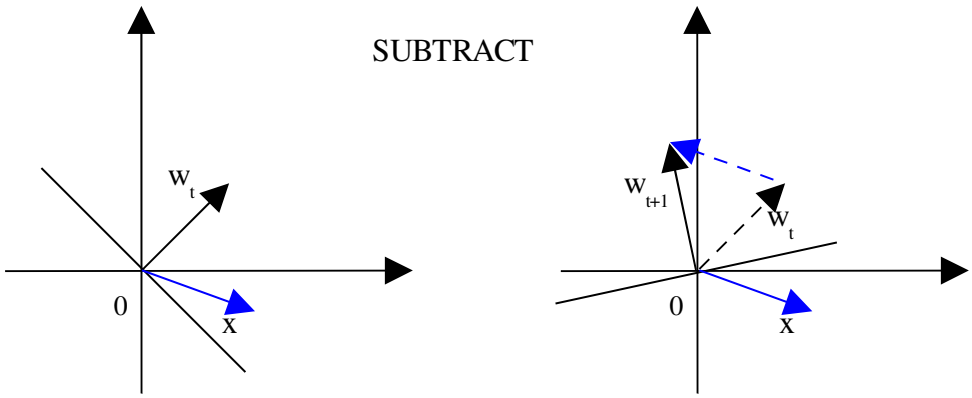


The 1st and 3rd conditions of TEST are those cases where the weight vector w already correctly classifies the input patterns; in that case w is not changed at all. But whenever the input pattern is wrongly identified (2nd and 4th conditions) w is changed by correspondingly adding or subtracting the erroneously recognized pattern. This rotates w and its perpendicular linear separation.



$x \in P$

After the addition of w and x into $w_{t+1,}$ x is correctly recognized, now it lays in the positive half space.



$x \in N$

Also here x is correctly recognized after the subtraction. There is a proof that if P and N are finite and linearly separable, our algorithm will converge, after a finite number of steps it will reach a weight vector w that correctly recognizes P and N. At first a couple of assumptions must be made, this will render the demonstration easier without loss of generality.

1. A new set P is defined as the union of the vectors of the original P and the negate elements of N. $\rightarrow P = P \cup N^{-}$
2. The vectors of the new set P are normalized. This is possible because if there is a vector w so that $w \bullet x > 0$ then this is also valid for a vector ηx where η is constant.
3. Also the weight vector w is normalized and labeled as w*.

δ is defined as a fixed positive number so small that:

$$\forall x \in P \rightarrow w^* \bullet x > \delta$$

The cosine of the angle ρ formed by w_{t+1} and w* is:

$$\text{(a)} \quad \cos \rho = \frac{w^* \bullet w_{t+1}}{| w_{t+1} |}$$

Since on the basis of the 1st assumption all the input vectors are positive follows that the weight vector can be corrected only through ADD. Let's now consider the numerator of equation (a) when after using ADD w_t has been transformed into w_{t+1}.

$$w^* \bullet w_{t+1} = w^* \bullet (w_t + x)$$
$$= w^* \bullet w_t + w^* \bullet x$$
$$\geq w^* \bullet w_t + \delta$$

This means that after n steps:

$$w^* \bullet w_{t+1} \geq w^* \bullet w_0 + (t+1)\delta \quad \text{(b)}$$

Let's now consider:

$$| w_{t+1} |^2 = w_{t+1} \bullet w_{t+1}$$
$$= (w_t + x) \bullet (w_t + x)$$
$$= | w_t |^2 + 2w_t \bullet x + | x_t |^2$$

Since |w|, the denominator of (a), is always positive we can conclude that $w \bullet x$ must be negative, otherwise the ADD routine would not be executed. Moreover |x| =1 because all the vectors have been normalized. Therefore:

$$| w_{t+1} |^2 \leq | w_t |^2 + 1$$

Which means:

$$| w_{t+1} |^2 \leq | w_0 |^2 + (t+1);$$
$$| w_{t+1} | \leq \sqrt{| w_0 |^2 + (t+1)} \quad \text{(c)}$$

Substituting (b) and (c) in (a):

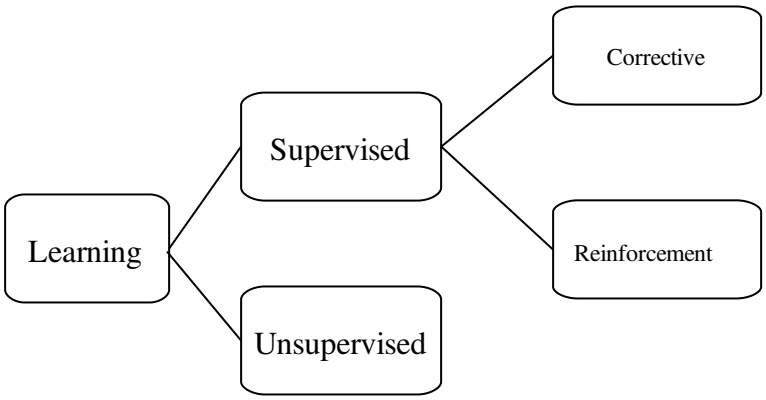
$$\cos \rho \geq \frac{w^* \bullet w_0 + (t+1)\delta}{\sqrt{| w_0 |^2 + (t+1)}}$$

Which for t → ∞:

$$\lim_{t \to \infty} \frac{w^* \bullet w_0 + (t+1)\delta}{\sqrt{| w_0 |^2 + (t+1)}} = \infty$$

But since $\cos \rho \leq 1$, we deduce t must have an upper bound. So the steps (t) of our learning algorithm must be finite.

The perceptron learning algorithm is classified as supervised learning with reinforcement.



It is supervised because the net is trained with some examples for which the desired output is already known, the weights are then corrected according to the correct or wrong classification of the input pattern. Unsupervised learning instead, implies that the net must find a solution for a problem but is given no example about how to solve it.
Corrective learning changes the weight vector with a standard step, using only the information whether the input pattern has been correctly recognized or not. Reinforcement learning instead, considers the extent of the error and changes the weights proportionally to this measure and this is the case of the perceptron learning algorithm, the wider the angle between w_t and x is, the bigger the distance between w_t and w_{t+1} (the correction) will be.

1.7 - Convergence to optimum

The original error function for the perceptron was:

$$E(w) = \sum_{x \in A} (1 - f_w(x)) + \sum_{x \in B} f_w(x)$$

If we want to generalize the error function of in supervised learning, we could say that the error function of the network is the sum of all the single errors during the network training. And we could define the recognition error of the single pattern as proportional to the distance between the desired and the real output vectors. In order to calculate the error for a

net that has been trained with a supervised learning algorithm the sum squared error (SSE) function is generally used. Every input x is coupled to the desired output t so that L is the set of all the pairs (x;t), while the actual output of the net for x is y. We can therefore indicate the error of the network when recognizing a single input pattern(x;t) as:

$$E_{x,t} = \sum_i | t_i - y_i |^2$$

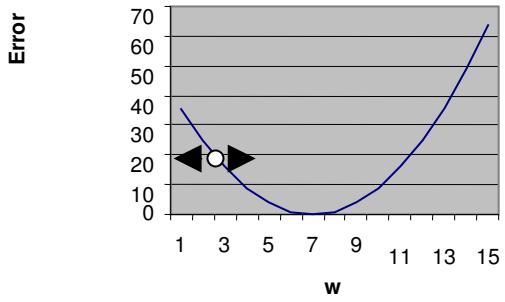
The total error is the sum of the errors that the net made while recognizing all the input patterns of L:

$$E = \sum_{(x,t)\in L} E_{x,t} = \sum_{(x,t)\in L} \sum_i | t_i - y_i |^2$$

Essentially the weight vector that we are looking for is the vector w that minimizes the error function. But how to find it is not exactly a trivial question, let's start to explain one way of diminishing the network error through an example. We will assume for a moment that we have only one weight w whose initial random value is 3 and that our error function is:

$$E = (w-7)^2$$

The graph of this error function looks like this:



Now, if we want to decrease the network error should w increase or decrease? How much? Or is it already the best solution? To answer these questions we have to find derivative the error function. The function derivative will give us an idea about the slope of the error function in the current point. In order to reach a lower error we wish the point representing the weight value to descent along this slope.

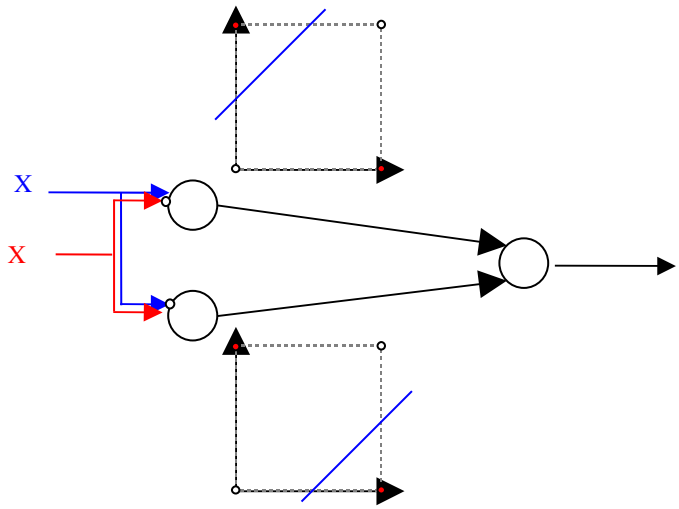
$$\frac{\partial E}{\partial w} = 2w - 14$$

That for w = 3 it is equal to -8. The negative sign of -8 indicates us that at the current point the slope goes down, so we have to increase our weight w if we want to achieve a smaller error. The amount of increase is often called “step” and depends on the implementation of the learning algorithm, nevertheless the direction in which we have to move the weight is clear.

In future we will say that our algorithm is converging when the current solution is reaching a stable solution, a global or local optimum, for the optimization problem. Relating to the previous example, if w would follow the slope downwards we can say that it is converging to 7.

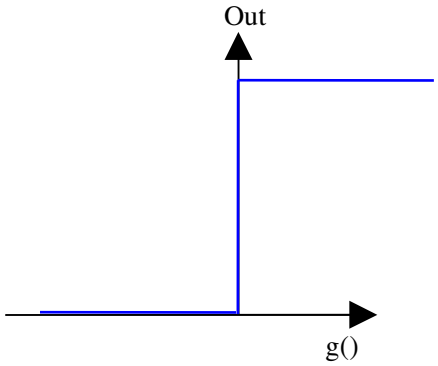
1.8 - The multilayer perceptron

In this chapter we have seen that the xor problem cannot be solved by a single perceptron, but a network with more than a single perceptron might be able to. A possible solution would be a multilayered network where each perceptron of the first layer should decode a linearly separable section of the input space. The result should be then combined by an output perceptron.

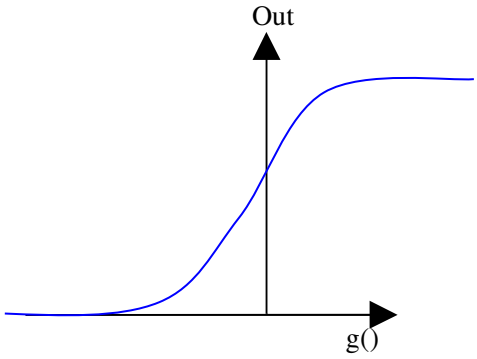


Yet this network is not capable to learn, the problem is that the output perceptron doesn’t receives the actual inputs, instead he receives only the binary outputs from the hidden layer perceptrons. This means that the synapse’s weights cannot be properly adjusted; the learning algorithm is not capable to determine the amount of correction that is necessary on the basis of 0’s and 1’s. This is called the credit assignment problem. The threshold function of the neurons of the first layer eliminates the information needed to adjust the weights of the

following layers. The key is to change the activation function of the perceptron. The step function has to be substituted for the sigmoid function.



Step function



Sigmoid function

The sigmoid function is practically equal to the step function when the integration function g() is very small or very big. The difference is that in the neighborhood of the threshold value the sigmoid function, unlike the step function, assumes the whole gamma of values between one and zero. This allows information about the input to be transmitted to the following layer. The mathematical reason because the threshold function has to be changed is that we actually need a continuous and differentiable error function. Since the step function is not continuous it is necessary to substitute it for the continuous sigmoidal function $s_c : \Re \rightarrow (0,1)$:

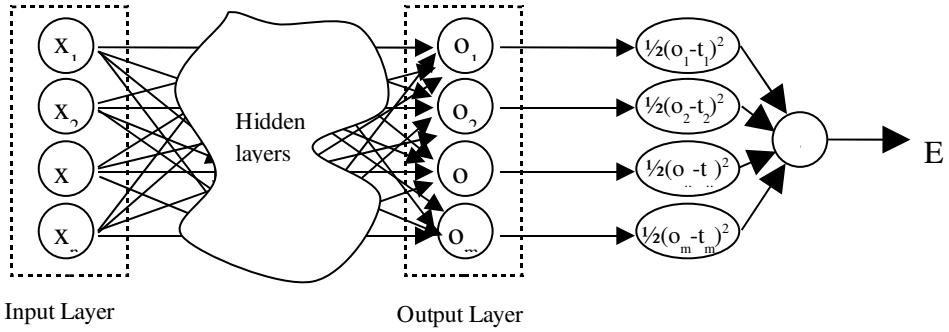
$$s_c(x) = \frac{1}{1+e^{-cx}}$$

Where the constant c determines the shape of the sigmoidal curve; in fact $\lim_{c\rightarrow\infty} s_c$ is equal to the step function.

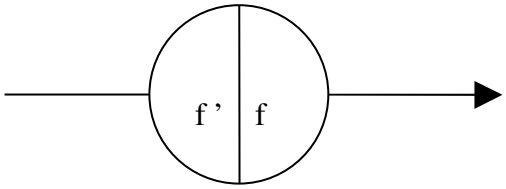
As the activation function of the neuron changes so the learning algorithm must be adapted. The new learning algorithm is called "backpropagation" and has an interesting story. Although presented in 1986 by Rumelhart, McCelland and Williams, it was later found that in 1982 Parker and even as early as in 1974 Werbos had similar conclusions. Backpropagation is another example of supervised learning and it is based on the same principles of the perceptron learning algorithm. A series of input patterns whose desired output is known are given to the net, as next the actual output is compared with the desired one, the derivative of the error function is calculated and a correction for the weight vector is decided. Subsequently it all starts with the definition of the error function. After all a network is nothing more than a graph containing computing nodes, each node calculates a primitive function on its inputs and transmits the result to the nodes that are attached to it through

edges. In general every input pattern is elaborated by the net and transformed into an output pattern, so the network is one of the possible implementations for a complex function composed of many primitive functions. This complex function is called "network function". Learning means adapting the weights of the net so that the network function is as similar as possible to another function f. Often it happens that the user doesn't exactly knows the formula of f, this is the reason why in the case of supervised learning we are not given the function f itself rather examples of inputs and desired outputs of this function in form of pairs (x;t).

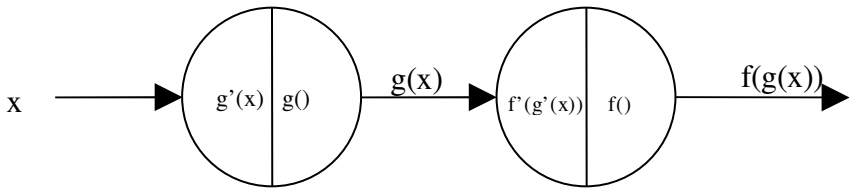
Having substituted the activation function we will assume that every primitive function of the network is continuous and differentiable. To start we will extend the network so that it will calculate our error function. This is done by adding two additional layers that compute the difference between the output and the t vector.



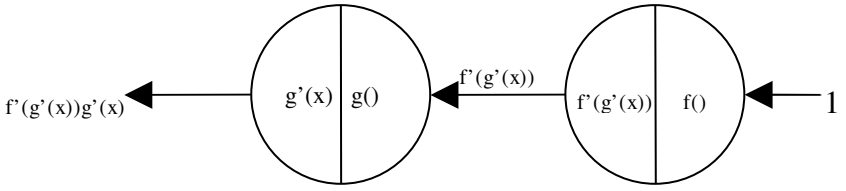
Moreover each neuron of the network will be modified. Neurons will be divided into two parts, one part will calculate the activation function, as usual, while the other will store the derivative of the activation function.



As said before, combining different nodes that implement basic functions we can obtain a network that calculates a complex function through function composition. Let's graphically examine the case of two nodes that are connected together.



On the other side Backpropagation is exactly the contrary of function composition; a one is fed from the output towards the input layer. This enables us to calculate the derivative of the network function.



If feed forward, the new modified network calculates the error function E and since it is composed of single continuous and differentiable functions also E is continuous and differentiable. This means that using backpropagation it is possible to calculate the derivative of network function.

Now that the general idea has been explained, it is time to describe the backpropagation algorithm in detail.

If $\frac{\partial E}{\partial w_k}$ is the partial derivative of E with respect to w_k, we define the gradient of E as:

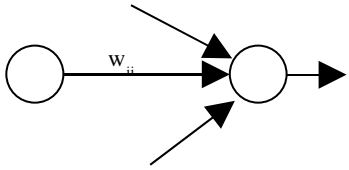
$$\nabla E = \left(\frac{\partial E}{\partial w_1}; \frac{\partial E}{\partial w_2}; \ldots; \frac{\partial E}{\partial w_k}; \right)$$

∇E lets us know how the error changes when a single weight w_k is changed. From here we can calculate the weight correction Δw_i, since it is necessary to move along the error function surface going in the direction of decreasing error a minus has been added:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_k} \qquad \text{for } i = 1,\ldots,k$$

Where η is the learning constant, and determines how strong the correction is.
Our network has n inputs an m outputs and we can define:

1. The network input vector: $x = (x_1, x_2, x_3 \ldots x_n)$
2. The target vector: $t = (t_1, t_2, t_3 \ldots t_m)$
3. The weight of the link from neuron i to neuron j: w_{ij}
4. The input of the neuron j: $net_j = \sum_{i:i \to j} o_i w_{ij}$



5. The output of the neuron i: $o_i = f(net_i)$
6. The set $L = \{(x_1, t_1), \ldots (x_p, t_p)\}$ of all training patterns
7. The set $A = \{(o_1), \ldots (o_m)\}$ of all output neurons
8. The error function $E = \sum_{(x,t) \in L} E_{x,t}$ with $E_{x,t} = \frac{1}{2} \sum_{k \in A} | o_k - t_k |^2$ which is directly derived from the SSE

The steps of the backpropagation learning algorithm are:

1. Initialize the network weights randomly
2. Select a couple (x,t) belonging to L
3. Feed forward computation: by using the input x and the target function t calculate the net output E. At each node the derivative is stored into the left side.
4. Back propagation: feeding a 1 into the output neuron and using the derivatives stored in each neuron calculate $\frac{\partial E_{x,t}}{\partial w_{ij}}$.
5. Adapt the weight $w_{ij} = w_{ij} + \eta \left(-\frac{\partial E_{x,t}}{\partial w_{ij}} \right)$
6. If the error is small enough stop otherwise repeat from number 2

Point number four determines the name of the whole algorithm, and it is not by chance! For that reason it is worth to take a closer look to its mathematical fundaments.

- We know that $E_{x,t}$ depends from net_j and that net_j depends from w_{ij}; using the chain rule we can therefore say that:
  (a) $\frac{\partial E_{x,t}}{\partial w_{ij}} = \frac{\partial E_{x,t}}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$.
  - The second part of (a) is $\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_{i:i \to j} o_i w_{ij} = \sum_{i:i \to j} \frac{\partial w_{ij}}{\partial w_{ij}} o_i = o_i$

- The first part of (a) is the variation of error by changing the input of the input of the neuron $-\frac{\partial E_{x,t}}{\partial net_j} = \delta_j$.

- Now we can write (a) as $-\frac{\partial E_{x,t}}{\partial w_{ij}} = \delta_j o_i$ and from that follows: $\Delta w_i = \eta \delta_j o_i$.
- It is possible to calculate δ_j, we know that $E_{x,t}$ depends from o_j and that o_j depends form net_j then:
  (b) $\delta_j = -\frac{\partial E_{x,t}}{\partial net_j} = -\frac{\partial E_{x,t}}{\partial o_j}\frac{\partial o_j}{\partial net_j}$
  - Considering the second term of (b) $\frac{\partial o_j}{\partial net_j} = f_j'(net_j)$
  - In case of an output neuron the first term of (b) becomes

    $$\frac{\partial E_{x,t}}{\partial o_j} = \frac{\partial \frac{1}{2}\sum_{k\in A} | o_k - t_k |^2}{\partial o_j}$$
    $$= 2 \cdot \frac{1}{2}(o_j - t_j)$$
    $$= -(t_j - o_j)$$

    and therefore δ_j becomes then $\delta_j = f_j'(net_j)(t_j - o_j)$
  - In case of a neuron j belonging to a hidden layer the first term of (b) becomes

    $$\frac{\partial E_{x,t}}{\partial o_j} = \sum_{k:j\to k} \frac{\partial E_{x,t}}{\partial net_k}\frac{\partial net_k}{\partial o_j}$$
    $$= \sum_{k:j\to k} \frac{\partial E_{x,t}}{\partial net_k}\frac{\partial}{\partial o_j}\sum_i w_{ij} o_i$$
    $$= -\sum_{k:j\to k} \delta_k w_{jk}$$

    that substituted in (b) gives $\delta_j = f_j'(net_j)\sum_{k:j\to k} \delta_k w_{jk}$

- For the sigmoidal activation function

$$f_j'(net_j) = s'(net_j)$$
$$= s(net_j)(1 - s(net_j))$$
$$= o_j(1 - o_j)$$

In conclusion:

$$\text{(d)}\ \delta_j = o_j(1-o_j)(t_j - o_j) \quad \text{for an output neuron.}$$

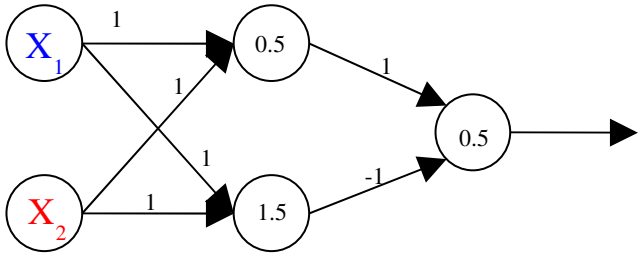
$$\delta_j = o_j(1-o_j)\sum_{k:j\to k}\delta_k w_{jk} \quad \text{for the hidden neurons.}$$

where the weight correction is:

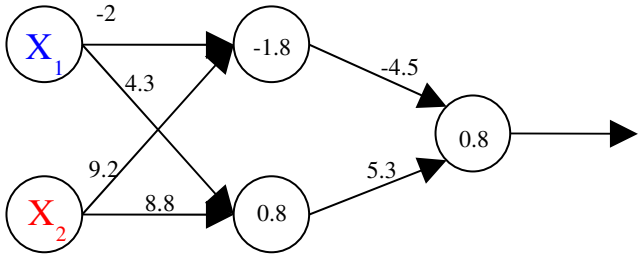
$$w_{ij} = w_{ij} + \eta\delta_j o_i .$$

To correct the weight of a link we must first calculate the value of δ, from (d) it easy to see that δ is a function of all the δ of the next layer k with the only exception of the output neuron. This means that all the weight corrections must be done starting from the output neuron and back propagating through the net as we have seen during the introduction.

Back to the original xor problem we can now see that there is a solution that enables our network to decode the xor function.



Nevertheless there is a chance that the learning algorithm might converge to a solution that is doesn't correctly decode the xor function.



Fortunately this local minimum solution occurs only in something like 1 % of the training cases.

Chapter 2

Basics of Genetic Algorithms

2.1- The beginning

It might be hard to believe but the scientific basis of genetic algorithms lay in a new world view originated during the nineteenth century. It was Charles Darwin (1809 – 1882), a British naturalist, who in 1858 published “On the Perpetuation of Varieties and Species by Natural Means of Selection” introducing the nowadays widely accepted “Evolution Theory”. A fundamental principle of the evolution theory is that species are mutable, they originate evolve and even extinguish. This revolutionary world view upset all the beliefs of his time, as the catholic creationism was the predominant belief in the western society of the nineteenth century. Creationism depicted the world as created by a God or another intelligent being. In such world view, species were immutable and ever existing. For thousands years before Charles Darwin introduced his theory, this had been the explanation for the origin of the animals. Man, instead, was seen as a special species, not belonging to the animal world, he was rather regarded as the elected race in the universe, the centre of it. Beyond the philosophical considerations on the impact that Darwin's theory had in the society and the way man sees himself, it's important to understand why the observations made by Darwin constitute the base for today’s research in the informatics field. The validity of our future algorithms will depend on the validity of the principles on which they are based, and, since such principles derive directly from the evolution theory, it is fundamental to understand it, especially in a period where still exist people that, on the grounds of religious beliefs, question such a theory.

A first observation that Darwin made was that individuals in a species are similar but not identical; this means that they have differences which, although small, would determine the relation between the individual and its environment. Darwin also noted how the features of individuals that successfully adapted to the environment were present in the following generations with an increased rate. His conclusions were that better adapted organisms had bigger chances of survival and as a consequence, of reproducing, on the other hand organisms that weren't good adapted tended to disappear because of an increased dying probability.

These two factors would shape the features of a species from generation to generation, transforming the fundamental characteristics of an individual belonging to that species.

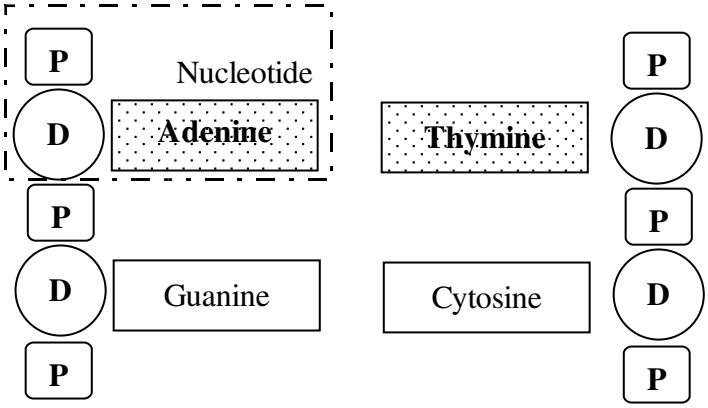
Another fundamental point of Darwin's theory was that a big part of the animals belonging to a species die before they can reproduce themselves; this is the consequence of the population growth in animals. As already Malthus noticed back in 1798, animals grow exponentially while the vital resources at their disposition grow in the best of all cases linearly. This is traduced in a high rate of deaths that will keep the population constant and produce a fight for the resources. In such a fight is again the individuals that adapted best have a higher probability to survive and therefore to pass their genetic features to the following generations. In the case of the emperor penguin the survival rate of chicks is 20%, 60% of them dies during the first year because of hunger, cold and accidents while the residual 20% never gets out of the egg.

2.2- A little bit of genetics

Every individual has a genotype that we could describe as the "construction map" of the creature. On the other side there is the phenotype that could be regarded as one of the possible results of the "map". All the inheritable information of an individual represents his genotype. To clear this it could be helpful to make an example. Every person has a determined genotype at birth; this is the information that will determine his features: the color of his hair, the color of his eyes and even the proteins his body will synthesize. But the external appearance of a person is not only to be ascribed to his genotype. There are environmental variables that influence the development of an individual. So the weight of a person is deeply determined by the diet he follows, and the diet is deeply correlated to the geographic region where such person lives. If the same person would have a diet based on fish instead of meat, this would dramatically change his outer appearance, although in both cases his genotype (features inherited from his parents) would be the same. These two different outcomes of the person's genotype are the so called Phenotypes. The phenotype is nothing more than the manifestation of the organism based on his genotype and the influences of the environment (genotype + environment → phenotype).

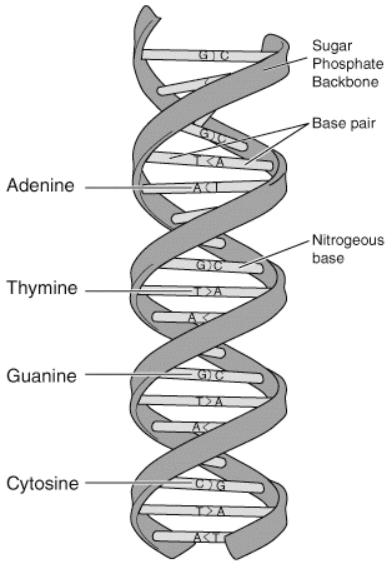
There are various structures that scientist identified as involved in the transmission of the hereditary information, they have been ordered in a hierarchical structure, they will be introduced starting from the smallest element (Bases) and following toward the macro structure (DNA).

All the information of the genotype is stored using an alphabet of four different chemical bases. These bases are called: adenine, guanine, cytosine and thymine.



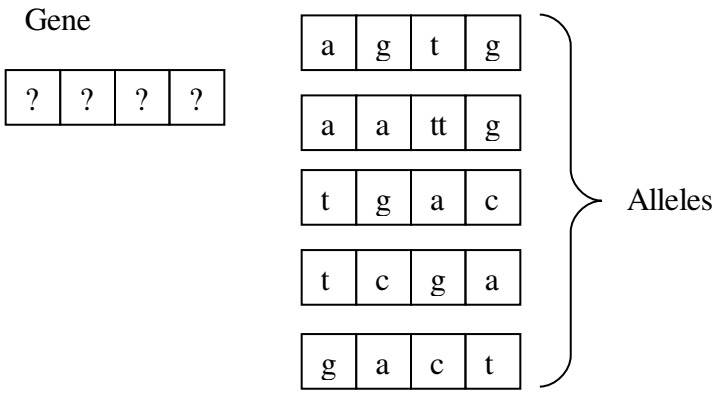
D = Deoxyribose
P = Phosphoric acid

Every base is linked to a sugar element called deoxyribose, which is itself connected to another deoxyribose element through a phosphoric acid molecule. A single deoxyribose molecule together with a single phosphoric acid molecule and a base constitute a nucleotide. All nucleotides are linked together forming a chain which is linked to another identical chain to form a spiral structure called the DNA (deoxyribonucleic acid). The DNA is the container of the entire individual's genetic information and is directly correlated to the genotype.

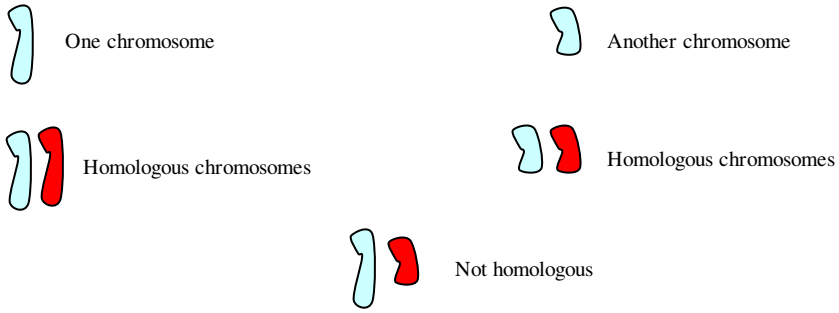


In order to link the two chains of nucleotides and form the DNA the bases are connected among themselves but not every combination is allowed. The only possible combinations are: (adenine, thymine), (thymine, adenine), (guanine, cytosine) and (cytosine, guanine).
It is known that the bases are grouped in triplets, being a triplet a sequence of three bases representing the code for an amino acid. The proteins that we synthesize are composed of amino acids, molecules that contain amino (NH_2) and carboxylic acid (COOH) ends. So in the end the sequence of the triplets is nothing more than a coding for the synthesis of proteins. It might be confusing to realize that all the possible combinations of the 4 DNA bases in groups of 3 (the triplet) are equal to $4^3 = 64$ while in reality only 20 amino acids are synthesized. The question is what happens with the remaining 44 combinations (also called codons)? Three of them are the so called "nonsense codons", they are sequences that indicate when to stop the synthesis of the proteins. The rest of the possible combinations are simply other forms of codifying the same amino acids. This is possible because those triplets that share the first two bases represent the same amino acid independently from the third base that composes the triplet; so GCC and GCA codify the same amino acid: Alanine. This renders the genetic code redundant and as a consequence also the DNA is a redundant code, said in other words, DNA is not the shortest form to describe a genotype.

The triplets can be then grouped in genes. A single gene establishes the structure of a protein and controls when and where the protein will be produced, in this way a gene is directly responsible for a determined hereditary characteristic of the individual. It is interesting to know that genes have variable lengths, they can range from hundreds of bases up to thousands and they might even overlap each other. The word gene refers to a well defined place in the DNA that determines a protein. The actual content of that place is called allele and can be different from individual to individual.

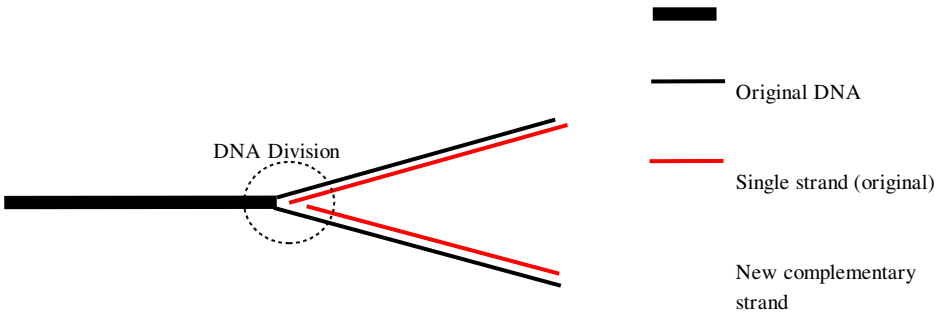


The containers for genes are the chromosomes, whose name was determined by the fact that they are visible under the microscope once properly dyed. When an individual inherits the features of his parents he can't take one gene from here and another from there, he inherits chromosomes as a whole and therefore the inheritance of certain characteristics coded in a gene is associated with the inheritance of the characteristics of the other genes that are present in the same chromosome. This means we inherit blocs of features and not single ones. It is important to note that the number of chromosomes is identical for all the individuals in a species and does not depend on the complexity of the organism, in the case of human beings there are 23 different chromosomes. Since every chromosome is present two times, there are a total of 46 chromosomes (23 pairs). The union of all chromosomes results in the already cited DNA which is to be found in the nucleus of every somatic cell of the body. Conventionally two chromosomes in the same pair are called homologous, they code the same information and while one comes from the father the other comes from the mother. Which of the two chromosomes is actually used to synthesize the proteins is established by the mendelian inheritance rules, for the moment it is only important to know that for every gene there are two copies, one on each chromosome.

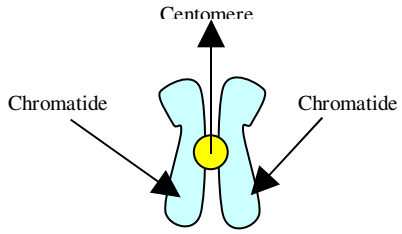


There are fundamentally two different processes where the genetic material of a cell is transmitted to another. They are called mitosis and meiosis.

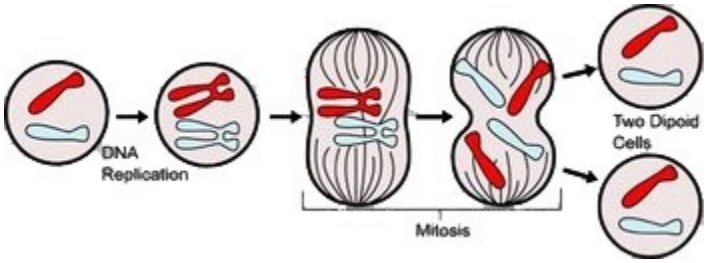
Mitosis is the process where a single cell divides itself into two cells that will share the same genetic patrimony. This is the mechanism that enables organisms to regenerate their tissues whenever they are wounded; the growth of a lost tail in the lizard is an incredible example of this process. The first step is the DNA replication, this is called the S phase. The DNA is now uncoiled and the two nucleotide chains are divided; on each one of these two strands a new one is built and, since for each base there is only another complementary base, the two new double strands result identical to the original one.



In this way every chromosome is replicated. The two identical chromosomes (also called chromatids) produced during the S phase are still joined together in the center by a centromere.



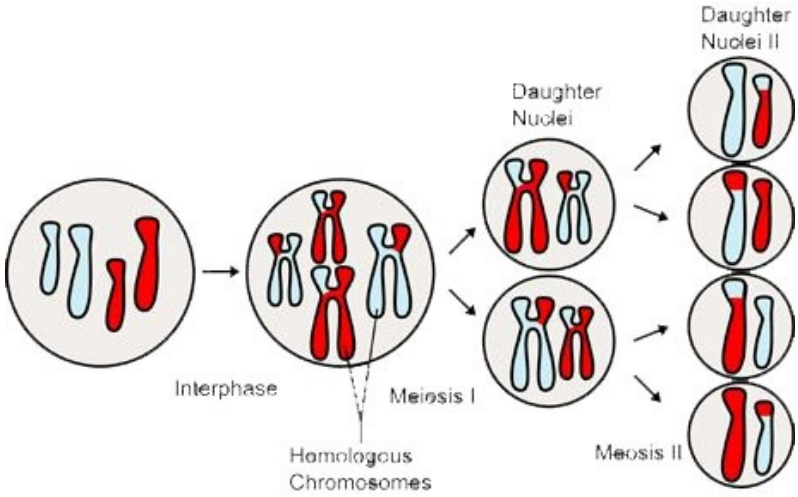
In the following phases (Prophase, Prometaphase, Metaphase, Anaphase) the chromatides will align with the cell equatorial plane and then divide themselves into two identical chromosomes. Each one of these chromosomes will then travel to its own cell pole.



The last phase (Telophase) sees the cell membrane dividing in two equal parts thus producing two independent cells that share the same identical genetic code. Mitosis is the mechanism that nature has found in order to transmit the genetic information of a cell without changes.

Meiosis alternatively poses as the process that creates the cells needed for the reproduction. As said before there are some cells that are called gametes, they are special cells needed for the sexual reproduction of organism, these cells have a peculiarity, they have only one set of chromosomes, the half of a normal DNA. If a cell has two sets of chromosomes is called "diploid" while if it has only one set of chromosomes it is called "haploid". Haploid cells are not an error, rather a necessity. If two cells having a complete DNA would join together they would create a cell with two DNA, meaning that every time two individuals would reproduce themselves the number of DNA molecules of their offspring would double with catastrophic consequences within a few generations.

The first step of meiosis is, like in the mitosis, the replication of the chromosomes into chromatides. The so called meiosis I phase sees the separation of the homologous chromosomes into two cells. Meiosis I is followed by meiosis II, here again a division takes place, but this time are the sister chromatides that are separated and sent to a separate cell. At the end of the process there will be four haploid cells.



In meiosis there are two important differences if compared to mitosis. The first is the creation of haploid cells; during reproduction two haploid cells (the gametes) will join to create a new individual. This has as consequence that the genetic material of the new born will be a combination of his mother's and father's genetic materials.

The second is crossover. During the prophase homologous chromosomes exchange genes; this means that a part of the mother's chromosome is inserted into the father's chromosome and vice versa. This is a second mixing of the genetic material and it is especially important because a brand new chromosome is created by mixing the available ones.

2.3- Evolution as an optimization algorithm

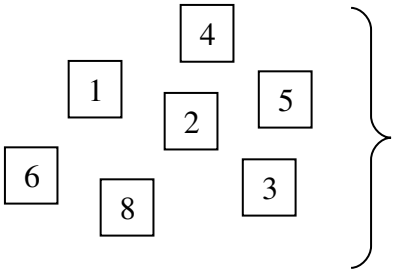
It is astonishing to realize the perfect functioning of even the simplest organism. The complexity of multicellular organisms is something that we haven't yet completely understood. In fact until today we haven't been able to replicate such a wonderful structure as the human brain can be. Even a fly is such a complex organism that scorns our most complex airplane. This is what many call "the mystery of life", a so complex variety of organisms, species and mechanism present in nature to remain incomprehensible for us in many of its parts. It is undeniable that the natural selection has performed an astonishing task in the evolution of species. Selection has produced very complex organisms that have successfully adapted to the changing environments encountered in different times, eras and geographical locations; it must therefore pose as a valid model for the construction of optimization algorithms.

As said before all the genetic information of an organism is stored in the DNA and is coded with the famous four bases: adenine, guanine, cytosine and thymine. We could see this

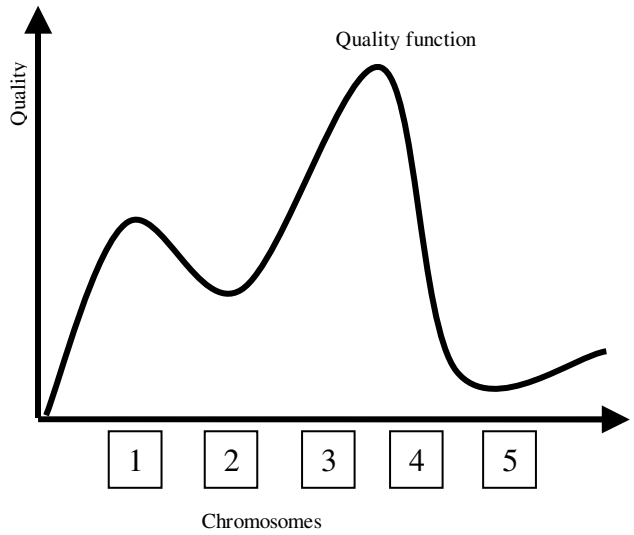
as a code in base 4. We could then consider each organism (or at least its genotype) as nothing more than a combination of the four bases. Please note that being the DNA a redundant code this relation is injective, a relation of N elements to 1; that means that there are several codifications of the DNA that will result in the same genotype.

At this point it is possible to abstract the evolution into a more or less generic algorithm. At first we will define the gene pool as the set of alleles available among the members of a population. Please remember that in biology the gene indicates a region of the DNA while the allele is the actual information that is contained in that region for a particular individual. So it could be possible to imagine the chromosome as the name and length of a variable, while the allele is actual value for that variable, in such metaphor the gene pool is the set of all "variable values" actually present.

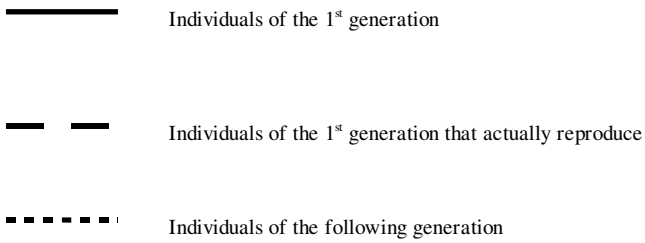
In light of this, evolution can be seen as a non deterministic procedure whose goal is to optimize the gene pool of a species according to the given environmental variables. The genotype of a single individual is here represented by a vector that in computer science is often referred to as chromosome, in reminder of its organic counterpart. Each chromosome vector is one of the possible combinations of values in the search space (the set of all possible solutions) and is given a certain quality value by the quality function.



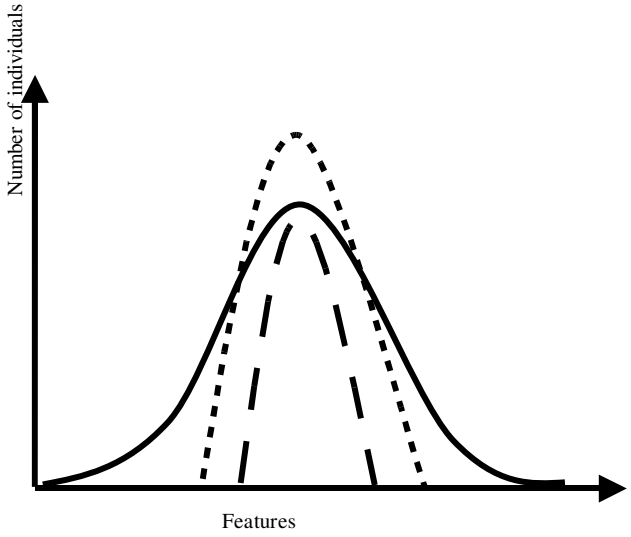
Chromosomes in the mono dimensional search space of



The value provided by the quality function is used to make a selection of the chromosomes that will, create the next generation of chromosomes, thus altering the gene pool in respect to the starting point. We are also aware that in the quality function there are local optimums and we know that there are generally a huge number of combinations in the search space. In fact the human genome is estimated to be composed of something between 20,000 and 25,000 genes. Even if this represents an immense search space, evolution has nevertheless found successful solutions. The most important operators in the evolution are: crossover, mutation and fitness determination. We will examine them in detail later on, now it will be shown how the combination of all these operators in the evolution process can shape a population from a statistical point of view. According to Kattman [12] there are mainly four possible action courses that evolution can take while influencing a population. In the following graphs used to describe these courses the same notation is always used:

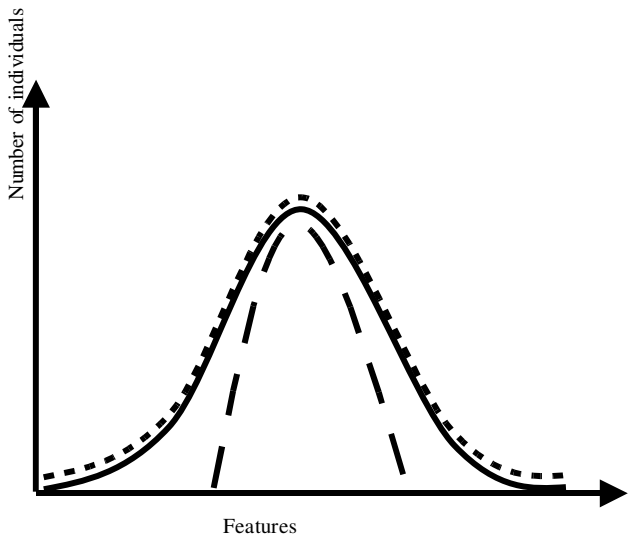


The first case is called reducing selection.



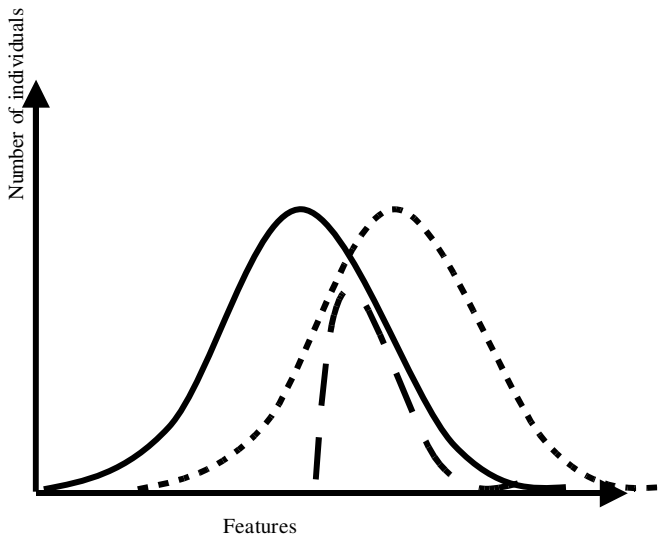
In this case the variance of the gene pool is reduced and the amount of individuals sharing the fittest genotype is increased (homogenization). This is a transitory step of the evolution in the process of search for the optimal gene pool.

The second case is represented by the stabilizing selection.



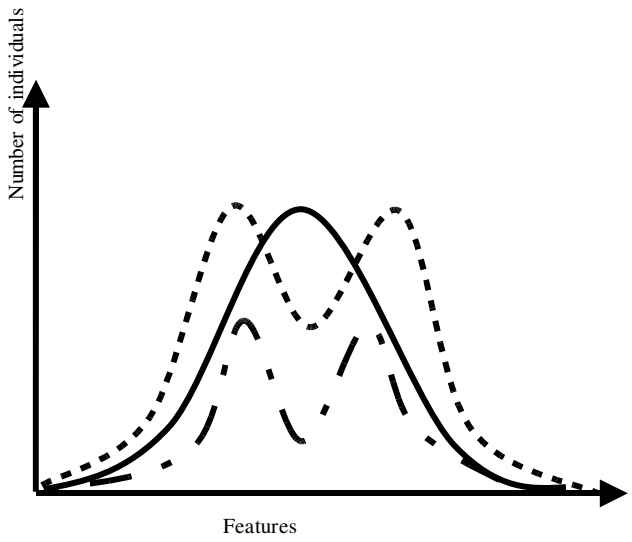
Here the diversity of the genetic patrimony has stabilized and converged to a (maybe sub) optimum. Since the population changes are cyclical there is no further evolution of the gene pool.

The third case is the transforming selection.



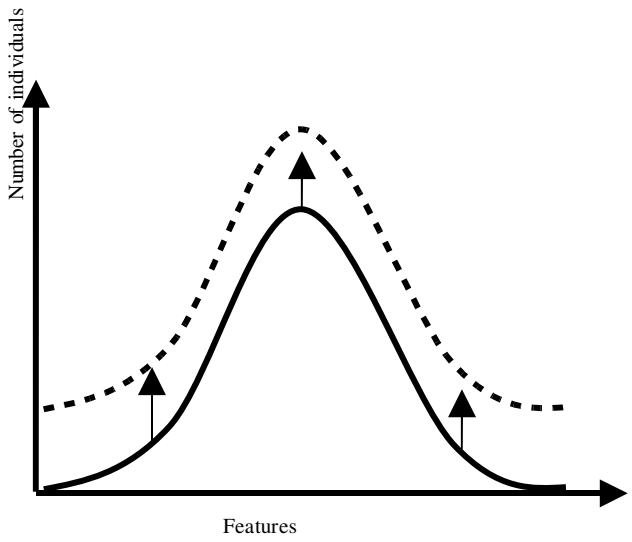
The gene pool of the population is here slowly shifting towards a more favorable solution. Again, this is a transitory step towards the optimum gene pool.

The fourth and last case is the disruptive selection.



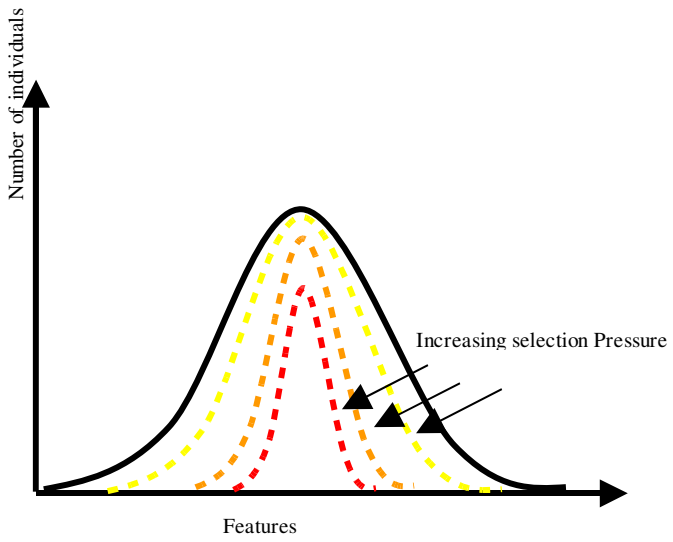
The gene pool is here dividing itself into two different pools that drift each towards its own optimum. This is a starting point for the creation of two different races starting from one and a transitory state.

Under the light of these four different influences of selection on a gene pool it is possible to determine the role played by two variables, the time and the number of individuals that compose a generation. The time a generation lives represents the vertical dimension of the search algorithm, it is the depth of search of our algorithm. Time influences in first instance the number of litters a generation can produce, and the persistence in time of its gene pool. The longer a generation lives the longer its genes are present in the population gene pool. Increasing the living time of a generation increases the number of individuals present in a population (the area under the curve) and shifts our curve upwards.



The life span of the average individual is a very important factor; it is known that in living creatures with very short life cycles the adaptation to the environment converges faster. This is in part consequence of a shorter lifecycle that determines a fast loss of the oldest genetic information, in part the consequence of a short sexual maturity and an increased rate at which an individual is capable of reproduce; factors that are accountable for the constant introduction of new genetic information. So, in case of insects or bacteria it is possible to observe in a relatively short time space the developing of resistances against those chemical agents that men use to combat them. The pulse at which the optimum solution moves in the search space corresponds to the time between one generation and the next, it is therefore clear how the life span of a species is a fundamental variable for our optimization problem.

A bigger temporal overlapping of subsequent generations might maintain a larger base of genetic information, which is useful to avoid local optima but might also slow down convergence. Moreover the population size varies with generation overlapping too; in those cases where resources are limited this could lead to an exacerbated fight for surviving. Every time the number of individuals that actually reproduce is reduced we speak of an increased selection pressure. Increased selection pressure lowers the curve of the reproducing individuals and restricts its variance of features, thus a steeper curve is achieved.



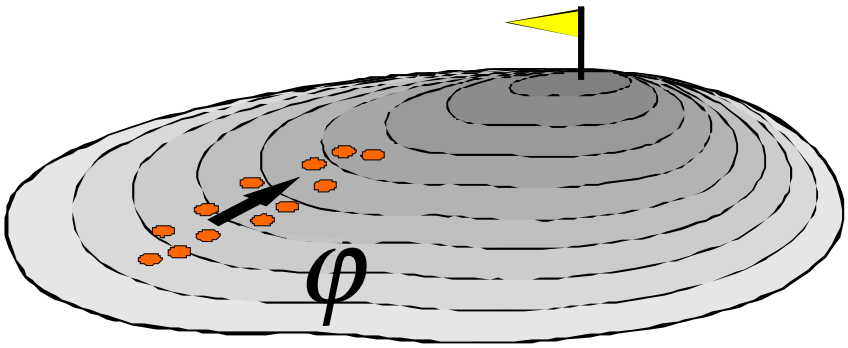
The number of individuals is, on the other side, the horizontal dimension of the research, it is the parallelism grade of our algorithm. This variable drastically determines the convergence to an optimum. Large litters produce a larger gene pool and an augmented broadness in the search for an optimum. Animals that have a small brood, like pandas, have a reduced adaptation capability and so, as nature demonstrates, they are quite prone to extinguish.

2.4- Quality function, mutation and crossover

There are some mechanisms in the transmission of the genetic information of an individual that are fundamental for natural selection. Considering their essential role in our future genetic algorithms it is worth to analyze them a little more in detail.

The so called quality function (in evolution strategies) or evaluation function (in genetic algorithms) assigns a value to each chromosome indicating its quality according to a certain criteria. It will be the nature of the problem we want to solve that will determine the quality function, so the same solution will generally have different quality values in different problems. Now, the point of our optimization is to find a vector that returns a maximum or a minimum value for such quality function. Since the quality function is nothing more than a function that returns a value indicating how good the current chromosome approximates the optimum, it is such function that defines the optimization goal, in that a determined vector is

a “good” solution for a certain quality function and not for another one. E.g. a quality function with the form of a parabola has an optimum solution (minimum) in a completely different point (the chromosome vector) from the one of a logarithmic quality function. We can then imagine the quality function as a surface with hills and valleys where each chromosome vector “v” represents a set of coordinates coupled with a quality value “Q(v)” indicated by the corresponding point for those coordinates on the surface of the function. The genetic algorithm will move the points representing the chromosomes along the quality function surface using a certain strategy, usually shaped like a path. The mechanisms that determine the path take advantage of the available information regarding the points that have been covered up to the moment. In this way the mutation and the crossover operators are used on the available data to create the next step of the path.



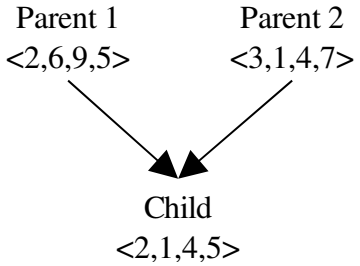
Rechenberg [7]

As seen before, crossover indicates the exchange of chromatides’ pieces among two chromosomes during the prophase of the meiosis. This has as a consequence that the genetic information is recombined mixing parts of the genetic information of one parent with parts of the other one. It is in this way that the new individual will present fresh characteristics, revealing a new solution for the problem of adaptation to the ambient; evidently this has as consequence the development of the species through the introduction of individuals that represent the mix of the features of their parents. If there wouldn’t be any crossover, natural selection would simply be restricted to a search for the best solution among all those already available. Alternatively, new solutions that lay between the parent solutions are always introduced in the gene pool with the use of crossover.

Crossover can be simulated in many different ways. A first rough subdivision would between dominant/recessive or co-dominant crossover, exactly like it happens in humans. Dominant/recessive genes mean that the child will show the features of only one of both parents. As seen before DNA has always two copies for each chromosome, one coming from the father and the other from the mother. This means that the same feature will be coded by two different genes, again, one coming from the mother and the other from the father.

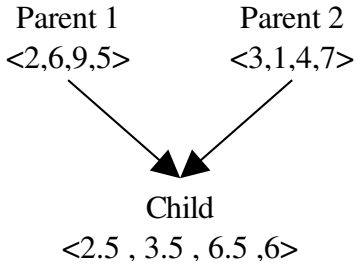
However it often happens that only one the two alleles that code a certain gene is actually used to produce the phenotype, said in other words: in dominant/recessive genes only the father or only the mother information is used. The allele that is actually used is called dominant while the other one is called recessive. So the individual will in the end show only the traits of one of its parents. This is the case of many inherited illnesses like hemophilia, the individual is either affected or not but it can't be affected "just a little".

This can be replicated in an algorithm by creating a new solution starting using the information of its parents' vectors by copying their values according to a certain schema.



There are a lot of different schemata to determine which value should be chosen in order to be copied in the child chromosome, since their efficiency is determined by the context in which they are used they will be introduced together with their respective algorithms afterwards.

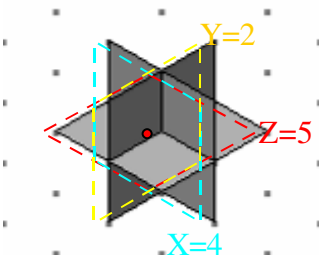
Co-dominant means that both the father's and the mother's genes are used at the same time, the child will then show a mix of its parent features. The blood type of a person a good example; people can have the A blood type gene ,the B blood type gene and the i blood type gene. The "i" gene is recessive, that means that if coupled with an A the individual will have A blood and if coupled with B will produce blood of type B. But when a person has both A and B genes will present the so called AB blood type, this is co-dominance! For a real vector this can be translated with average values.



Dominant/recessive crossover is by far the most used in genetic algorithms, therefore it is often implied that when talking about crossover it is a dominant/recessive crossover. Taking a more mathematical approach to crossover we could say that crossover can be graphically represented as an operator that inserts a new point in one of the vertexes of a

hypercube. To construct such a hypercube we have to use two stars of hyperplanes, one for each chromosome. Each star is composed of all the hyperplanes that have the following form: $x_i = c_i$ where x_i is the i^{th} component of the search space and c_i is the value of the chromosome for that component. Intersecting the hyperplane stars of the two chromosomes involved in crossover we obtain a hypercube. The crossover simply chooses one of the vertexes of this hypercube. Let's make an example in the three dimensional space, we will use the following two chromosomes A:(2,5,4) and B:(4,7,6).
The hyperplane star that passes through A is composed of three planes:

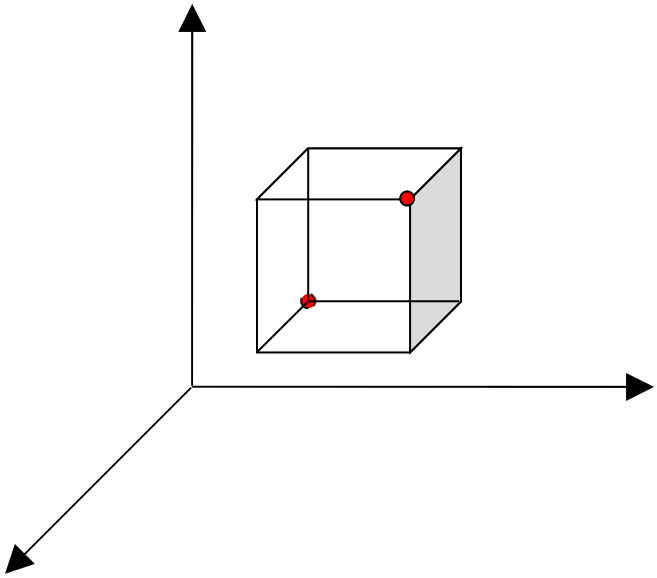
X = 2
Y = 5
Z = 4



Likewise the star for B will be:

X = 4
Y = 7
Z = 6

Now, A and B are two vertexes of the cube created by these planes,

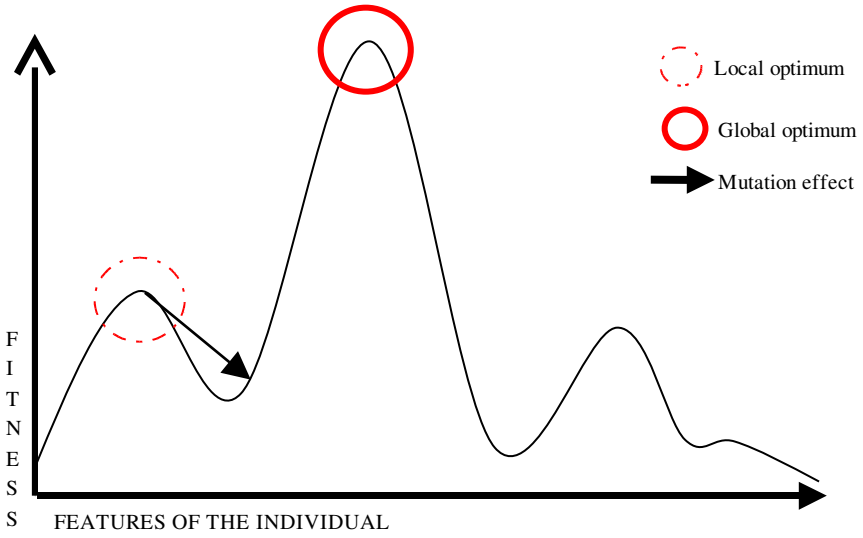


Crossover will generate a new solution that lays on any of the vertexes of the cube.

Mutations instead, are changes in the code of the DNA; in nature they can be produced by chemical agents or by radiations such as the X-rays. Although big mutations in the genetic material render the cell unable to live, it isn't correct to consider mutations as a purely negative factor in the evolution process. In fact these changes play a fundamental role in evolution; they introduce variation in the genetic patrimony of a population. According to Darwin's theory only the fittest survive while the biggest part of a population never manages to reproduce. If we see every individual as a DNA, and therefore as a combination of all the possible sequences of the four bases, it is obvious that together with the death of the biggest part of the population an enormous amount of genetic information goes lost. Supposing the environmental variables are fixed, the genetic information present in a population will tend to become uniform through the generations. The DNAs will "converge" to an optimum solution. Under the influence of the natural selection and with the passing of a large number of generations, the gene pool variance of a population tends to reduce itself. The individuals of such a population tend to have the same features, the ones of the fittest individual. Theoretically all the possible sequences of bases would have converged to one single combination, the fittest for that environment. But what would happen if the environmental variables would suddenly change and render that combination not suitable for survival? The whole population would be unable to adapt to the new environment and would consequently extinguish. Mutations introduce new variants in the converging genetic combinations; they represent a random variable that is needed to avoid a risky grade of homogeneity among the individuals of a species. The importance of such a random value in optimization problems has been recognized long before the introduction of genetic algorithms. The use of a properly weighted random factor is essential for the avoidance of local optimums, in fact a random

value can be often found in classic algorithms such as the simulated annealing. The use of a random value introduces the chance (generally quite small) of jumping to a neighboring solution that is normally worse than the best one found up to date. The advantage in this is that the neighborhood of the new solution might contain another solution that is better than the previous local optimum.

In the following graph we suppose to have an algorithm that follows blindly the slop towards a maximum, of the starting point was x=0 then the algorithm would converge to the local optimum circled with a dotted line and would stay there. If a proper random factor would be introduced the algorithm might jump following the arrow and then converge to the global optimum.



Therefore mutation increases the chances of finding the global optimum. It is important to remember here that the DNA is a redundant code, fact that attenuates the actual effects of such mutations, never the less the chances of a mutation must be kept to a minimum; in fact in nature they occur between once every 10^5 and once every 10^9 genes. The reason behind the necessity of a low rate of mutations can be easily understood. A mutation moves the current optimum to a solution that is (with a very high probability) worse. If our algorithm gets stuck in a local optimum and keeps it as the best solution for a large number of cycles, it is probable that a mutation will occur on that solution. This would initiate again the search for an optimum. But if the mutation would occur with a very high rate, let's suppose once every two iterations, the temporary optimum found would constantly move to a random solution thus preventing any convergence towards a better solution and rendering the algorithm useless.

2.5- The role of chance

The theory of evolution, at first glance, is deterministic. This means that given the same genetic, environmental and temporal variables the evolution outcome is the same. But a closer look reveals that chance plays a fundamental role in the evolution. Firstly, chance is a big factor that manipulates the survival in the real world. It is not always true that the fittest is the survivor and there are many factors in addition to the genotype of an individual that determine its survival. Let's make an example. If a group of animals is escaping from a predator, they will all have different speeds, being the fastest the fittest for survival. But dying because of a landslide is not something that has to do with the DNA, it is an event in which the speed of an animal or its reaction time are negligible. In this case the surviving chances of an animal are proportional to the distance from the landslide center, a factor that can be considered as casual. Even though this case doesn't takes in consideration the fitness of an animal but it can surely influence the gene pool of a population.

As already said, mutations are factors that occur randomly and, as if this wouldn't be enough, they do not always really cause an effect. A mutation might occur in a recessive allele or it might change the third base of a triplet, influencing in no way the production of amino acids, and this case alone represents already the 33% of mutations. Sometimes a mutation might even reverse the effects of a previous mutation thus returning to the original state. There are two ways for this to happen: the first is the alteration of the same base for two times, the other is a mutation of a second gene that will cause the inhibition of the effects of the first mutation. Last but not least the influence of chance during the cross over. Considering the maximal amount of reproductive cells produced during the average life of a human being, it is statistically impossible that two of these cells would have the same genetic code. This gives us an idea of the real recombination effect that cross over has in the meiotic process.

In consideration of all this it would be more appropriate to rephrase the natural selection definition with "the fittest has a higher chance of surviving".

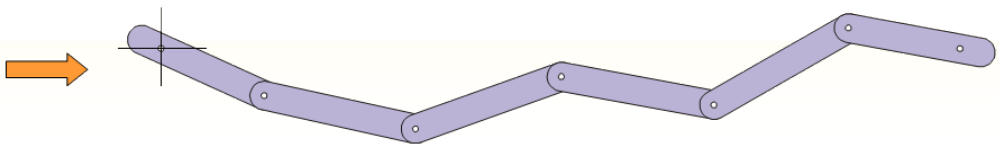
Chapter 3

Evolution Strategies

3.1- Rechenberg, the pioneer

The first attempts to emulate evolution in order to solve problems took place during the 60's at the Technischen Universität in Berlin. At those times Ingo Rechenberg labelled the new methods as Evolution strategies (ESs), the main idea behind the evolution strategies was to abstract the basic mechanisms of evolution and use them to improve technical applications. "The island of the crabs", a science fiction book of Anatolij Dnjeprow, inspired in Rechenberg the idea of natural selection being applied to machines. Rechenberg's plan was to copy from nature only those methods that were fundamental, indeed he affirmed that it is nonsense to copy every single detail of the evolution process in a 1:1 scale because in that case the goal would disappear in the gigantic effort of imitating evolution in its finest detail. The first optimization problems for which Rechenberg used his evolution strategies were typical engineering problems. They where mostly based in finding the best shape for a specific use, for example: the best form for a given surface and air flow, the best heat dissipating form for metal plates or the best conjunction between two fluid transporting tubes that laid at a 90° angle. The results that Rechenberg achieved where astonishing, even more surprising if considered that some of the first experiments were calculated by hand due to the lack of computers, and that even those experiments that used a computer could count only on such limited resources that would let them last entire weeks.

The first experiment, called "Darwin im Windkanal", was done with an articulated metal plate where each of its five pivots was adjustable in 2° steps. The ES was used to calculate the optimal form of the plate so that it would offer the least resistance to the air flow (represented by the orange arrow). The idea was to simulate the evolution of river animals. The transformation from zigzag shaped prehistoric worms into today's flow optimally formed fishes.



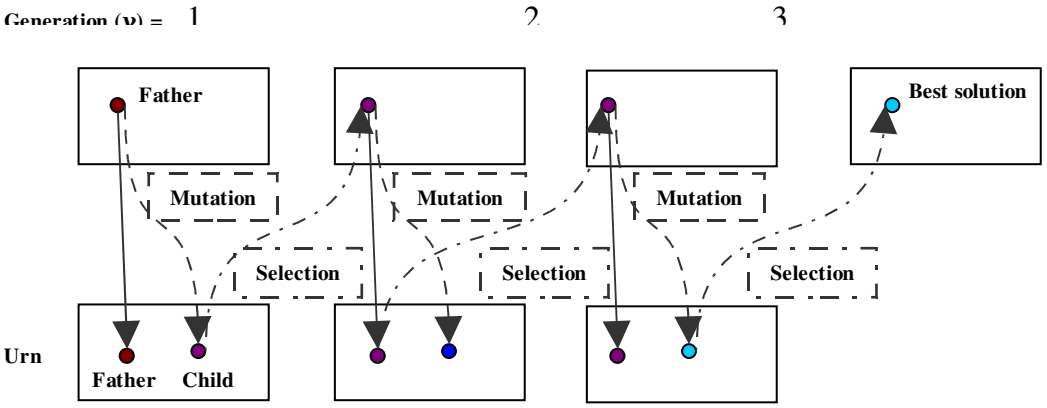
All the possible combinations of the pivots in 2° steps where $51^5 = 345.025.251$ a huge search space. During the experiments was found that after 200 iterations of the ES, the metal plate had reached the optimal form, which was a totally straight surface. This proved the validity of ESs but was no real breakthrough since the use of a straight plate in such environment was very well known and easily predictable. In order to probe the adaptability of ESs to different environmental variables Rechenberg made another experiment changing the wind flow direction. This time the results where less trivial, the plate assumed an oblique form that was optimal but not calculable using the fluid mechanics known in the sixties. Throughout successive experiments Rechenberg succeeded in developing heat dissipating plates with a 97% higher heat dissipation coefficient and a conjunction with 10% less flow deflection loss in respect to their starting designs. Several other experiments followed, demonstrating the strengths of ESs and starting a deep interest in the field of ESs.

ESs are a series of optimization algorithms that share the same approach to the problem. There are individuals that form a population, to be more exact each off those individuals represents a possible solution for the given problem and is coded in a vector of real parameters, the chromosome. The reason for the use of real values in the codification is the direct consequence of the engineering problems that Rechenberg originally tried to solve with help of ESs, in fact the chromosomes employed in the first experiments used to represent angles between surfaces, lengths and other physical dimensions that had to be coded with real values in order to obtain usable results. Such chromosomes generally undergo cycles of reproduction, mutation and selection until a certain criterion is reached. In the next paragraphs we will examine the different structures of this cycles and how they differ one from the other.

3.2- The ($\mu + \lambda$) ES

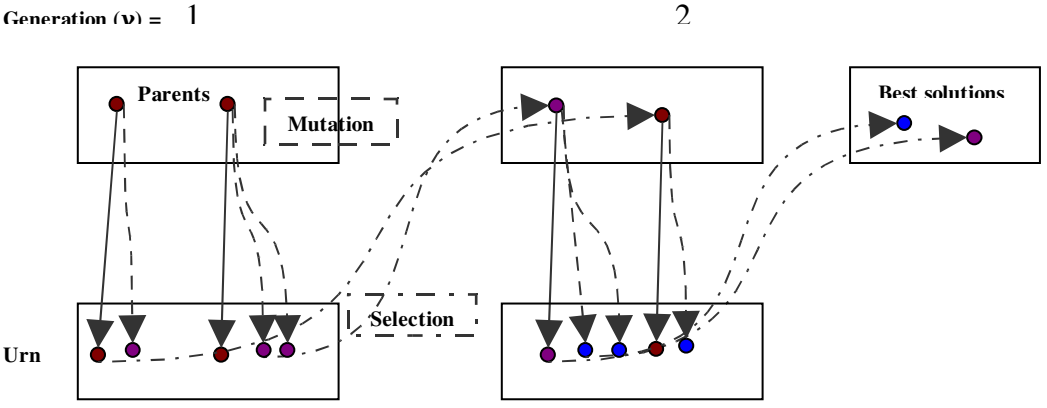
The first evolution strategies were masterpieces of simplicity. In 1964 Rechenberg had already experimented with the so defined "(1 + 1)-ES", starting from a given chromosome, a second one is created through duplication, this "child" chromosome is then mutated and thrown into an urn together with the original vector, the so called "father" chromosome. The solution that proves to be the both of both is extracted from the urn, the other one discarded, and then a new cycle (generation) begins. Hereby the name 1+1, the first 1 indicates that there is one father and the plus sign the fact that it is thrown into the urn together with the child chromosome, the second 1. This procedure would go on until the break criterion is reached, this might be quality of the current chromosome or the number of generations, some temporal constraint might also be used to control the maximum runtime of the ES.

(1 + 1) ES



A generalized version of the (1 + 1) ES would be the (μ + λ) ES, where μ denotes the number of chromosomes that survive the selection process to become the parent chromosomes of the next generation, while λ designates the number of child chromosomes that are created at each generation. For instance, (2 + 3) ES means that 3 new vectors are created by randomly cloning the original 2 parent vectors with uniformly distributed probability; then the so obtained child chromosomes are mutated with normally distributed probability. All the 5 vectors are then thrown into the urn and from there only the 2 best are selected to be the parents of the next generation.

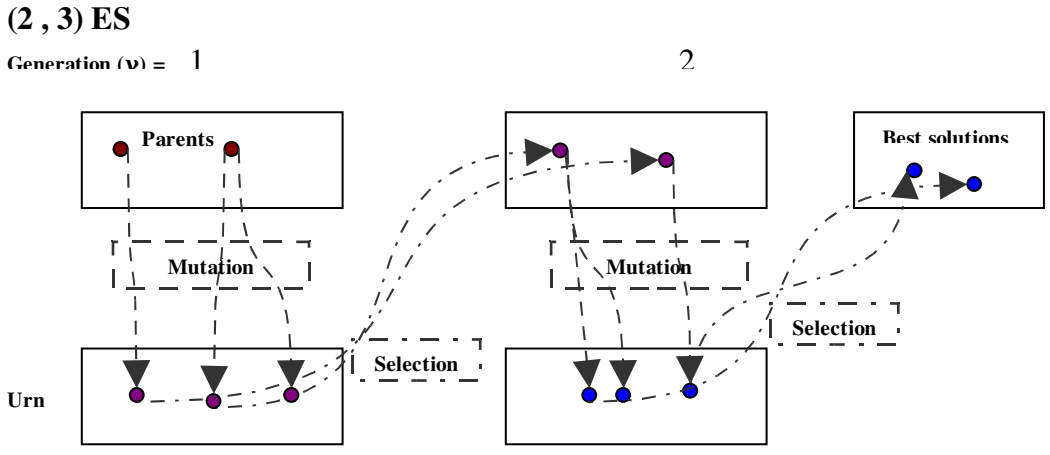
(2 + 3) ES



The "Darwin im Windkanal" experiment earlier described, used a (1 + 1)-ES after the initial approach with a (1 + 10)-ES was interrupted because it revealed to be too time consuming. As it can easily seen (μ + λ) ES are monotonic non-decreasing functions, where quality(b) $\geq$ quality(a) $\forall b > a$, in other words the best solution never gets worse than the best solution of a previous cycle.

3.3- The (μ , λ) ES

We mentioned before some quality functions may have some local optimum, although in ESs a mutation operator is used, this might not be sufficient to avoid premature convergence in such a local optimum. Since a (μ + λ) ES hands the best solution over the generations it is quite prone to get stuck in local optimums. This is something that doesn't happens in nature, where even the fittest individual has a limited life span and, no matter how long it will live, it will eventually die, consequently removing its genotype from the pool. In nature there is no immortal individual, although cloning is a hot topic these days it hasn't had yet succeeded in making a particular genetic material eternal. Although it would be interesting to simulate the usefulness and impact of cloning on a gene pool we have to come back to our (μ , λ) ES. A (μ , λ) ES emulates more accurately natural selection in that only the children chromosomes are thrown in the urn. All the parents are automatically discarded, no matter how good or bad, each chromosome lasts one and only one generation. The remaining procedures such as cloning, mutation or quality calculation are identical to a normal (μ + λ) ES.



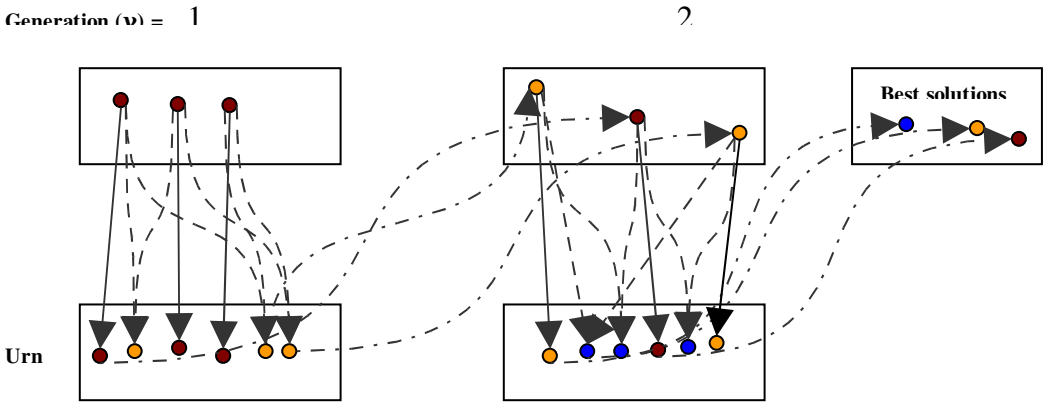
The consequence of switching from a "+" to a "," ES is that the quality function of the best solution is not anymore non-decreasing; practically there is no guarantee that the best solution of the $(n + 1)^{th}$ generation will be better than the one of the n^{th} generation, but this is the price to pay for the prevention of premature convergence.

3.4- ES generalization

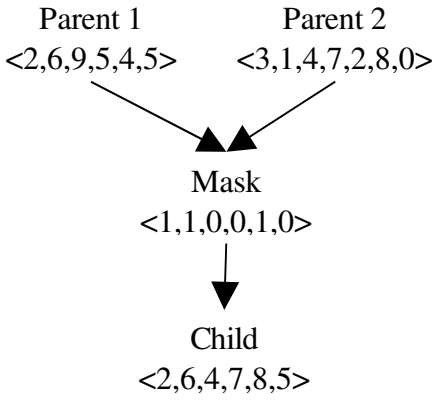
From now on the two ESs previously described will often be referred to as $(\mu \# \lambda)$, where "#" is a wildcard symbol representing either "," or "+"; this generalized reference is often used in those fields where both families of ESs share many common properties. There are many possibilities to vary a $(\mu \# \lambda)$, for example simulating population size fluctuations by dynamically changing the values μ and λ. In fact in nature most populations do not have a steady size, the human race is, for example, exponentially increasing while other populations, such as snowshoe hares (some kind of rabbit), undergo observable cyclical fluctuations. It might be interesting to establish the μ or λ values at each cycle with such functions as sine (cyclical population sizes), exponential functions or even functions that are dependant on other parameters of the ES. Please note that changing μ and λ it is possible to control selection pressure. Selection pressure is mathematically represented by the quotient $s = \mu / \lambda$, that is the fraction of individuals of the entire population that eventually reproduce. Nevertheless there is a restriction that must be kept in mind when setting the parameters of an ES, for obvious reasons μ must always be smaller than λ.

3.5- Recombination in ESs

The ESs presented up to this point use only the mutation and quality calculation functions; it is time to introduce the recombination (crossover) operator. We know recombination tries to emulate the chromosomal crossover in cells, but, if we want to use it, we have first to add a new parameter ρ to the so far used notation. A $(\mu / \rho \# \lambda)$ ES is a $(\mu \# \lambda)$ ES where chromosomes are chosen in groups of ρ with uniformly distributed probability in order to be recombined and produce one child chromosome. It is extremely important to note that the recombination is done by choosing two parents with uniformly distributed probability; this means that the quality of the chromosome has absolutely no influence on the number of children a chromosome produces, nor is it taken into consideration when two parent chromosomes must be selected to produce a new child chromosome.



Since animals have only two parents, ρ is usually set to 2, nevertheless it mustn't be forgotten that possibilities are much wider. Both dominant/recessive and co-dominant crossover have been used in ESs. In the case of a d/r crossover a "mask vector" of the same length of the parents is created. The mask vector is a series of 0 and 1 that are randomly chosen, the child is then created substituting all the 0's with the respective values of the first parent and all the 1's with the values of the second parent.



3.6- Isolated populations

Sometimes it is useful to simulate the existence of populations, which are chromosome groups that are divided from one another. The idea of simulating population has two different reasons. The first is the speed advantage that is achieved when using parallel computing. In case of a distributed environment (several computers connected together) or a single multi-processor system, it is possible to achieve a higher speed through the parallelization of the calculations. If each population is assigned to a processor the grade of parallelization is higher in comparison to the case where a single bigger population runs on all the machines/processors at the same time, therefore tasks are solved more rapidly. Besides in case of a distributed system where network communication and data transfer among computers are slow in comparison with the processing times, the isolation model provides a successful method to reduce communications to the bone, augmenting therefore the overall efficiency. The second reason behind the introduction of populations is the need for a more accurate simulation of nature. In fact most species in nature are bound to a certain geographical zone resulting in the different races typical of each place. This has often had as consequence the creation of "biological niches", very peculiar gene pools that would rapidly extinguish if confronted with the mainstream gene pool but that nevertheless present interesting features worth analyzing. A typical example for gene pool niche would be the case of native Australian animals, such as the Tasmanian tiger, which have encountered enormous survival difficulties once in competition with species imported by humans.

Rechenberg used the [μ' # λ' (μ # λ)] notation to indicate the use of populations. Each population is a detached set of chromosomes that develops in an independent fashion, without communicating with the other existing populations (island model). The notation indicates that each population uses a (μ # λ) cycle and that the μ' best out of λ' populations are selected to become the parent populations of the next cycle.

Let's make an example with a [1, 4 (2, 3)]-ES:

Following a normal (2,3)-ES schema, each population consists of 2 parent chromosomes that are mutated in order to produce 3 child chromosomes. The 2 chromosomes with highest quality values are then chosen to be the parents of the new generation. The previously described (2, 3)-ES schema is applied to 4 (λ') different populations, at the end of the process the best population (μ'=1) is chosen as the parent population of the next generation. This single population of 2 elements will be replicated 4 times in the next cycle in order to create the needed child populations which will, again, follow the (2, 3)-ES schema.

There is no fixed way to establish the quality of a population, common methods are: median quality of the individuals, the quality of the best individual or even the quality variance of the population.

Now it is time to introduce an additional parameter, γ, representing the number of generations that a population remains isolated. This means that in a [1, 4 $(2, 3)^5$] -ES each of the 4 child populations will go through 5 cycles of a (2, 3)-ES totally isolated from the other

populations. After the fifth generation the parents of each population will be finally compared with their "brother" populations in order to select the best population, as it would happen in a normal [1, 4 (2, 3)]-ES.

3.7- Rechenberg-Schwefel notation for ES

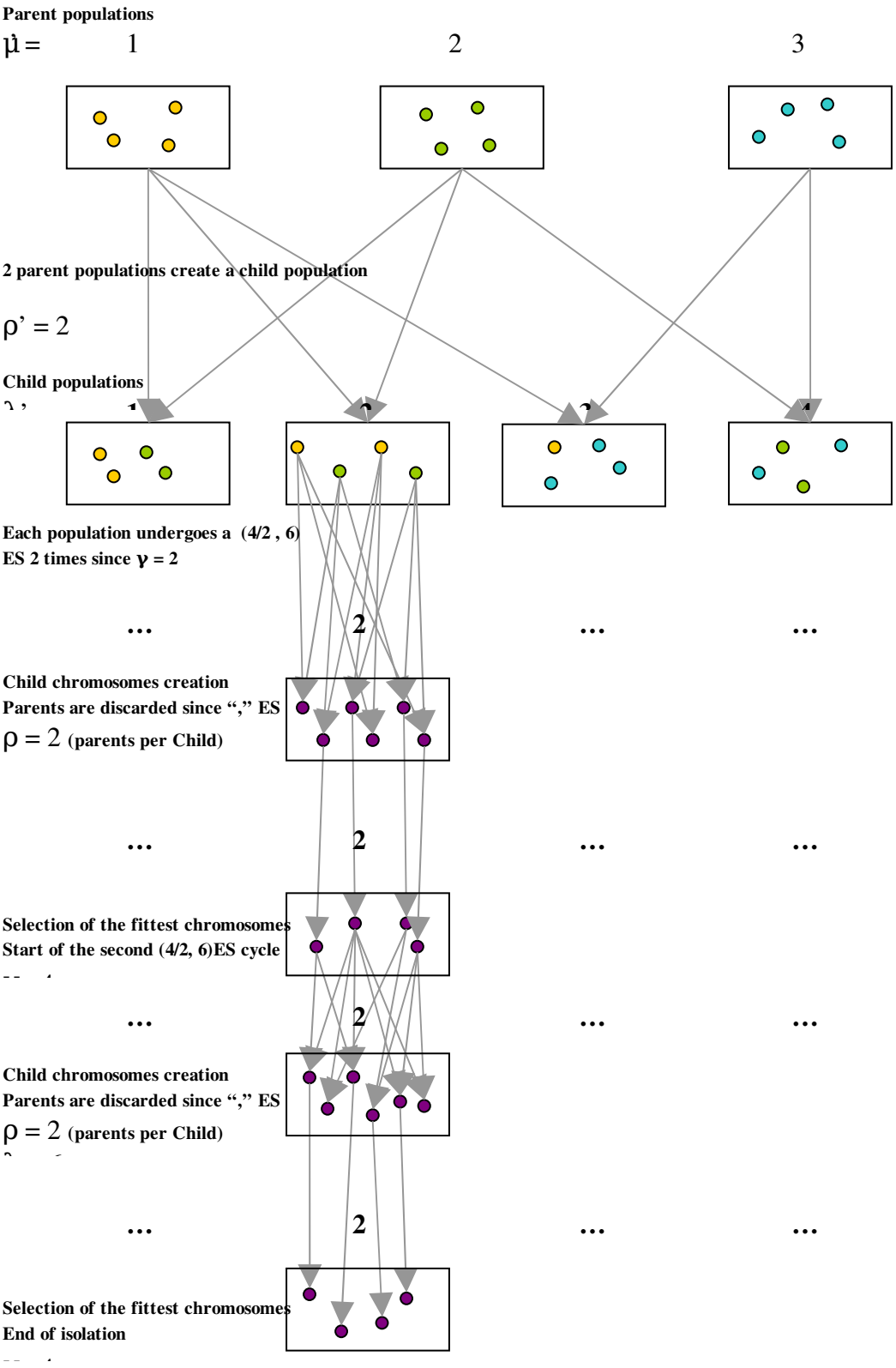
At this point it is possible to determine a general notation for Evolution Strategies.

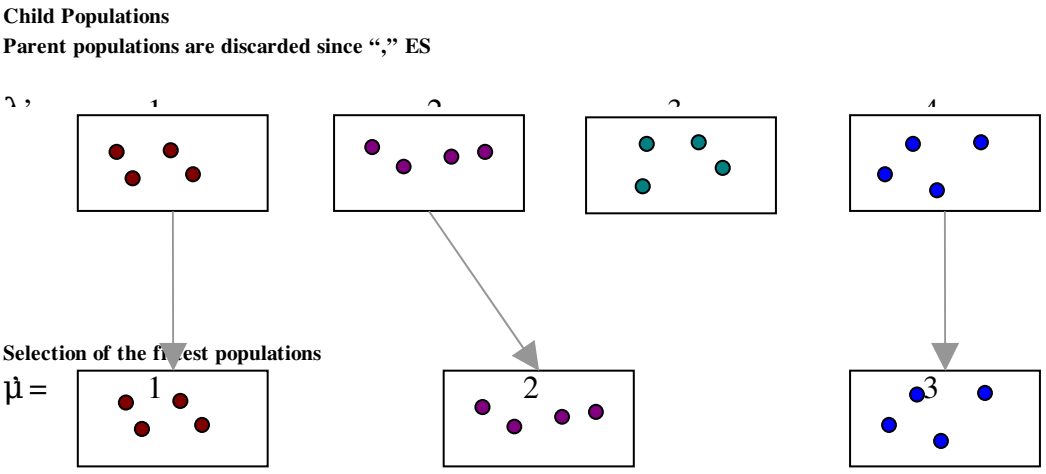
$$[\mu'/\rho' \# \lambda' (\mu/\rho \# \lambda)^{\gamma}]^{\gamma'} - ES$$

#	= represents either a "+" or a ","
μ'	= number of Parent-Populations
λ'	= number of Child-Populations
ρ'	= Recombination Index for Populations
γ'	= number of population generations
μ	= number of Parent-Individuals
λ	= number of Child-Individuals
ρ	= Recombination Index for Individuals
γ	= generations in isolation

Let's make a graphical example:

$$[\mu'/\rho' \# \lambda' (\mu/\rho \# \lambda)^{\gamma}]^{\gamma'} - ES$$
$$=$$
$$[3/2 , 4 (4/2 , 6)^{2}]^{1} - ES$$


Parent populations
μ = 1 2 3
2 parent populations create a child population
ρ' = 2
Child populations
Each population undergoes a (4/2 , 6)
ES 2 times since γ = 2
…
2
…
…
Child chromosomes creation
Parents are discarded since "," ES
ρ = 2 (parents per Child)
…
2
…
…
Selection of the fittest chromosomes
Start of the second (4/2, 6)ES cycle
…
2
…
…
Child chromosomes creation
Parents are discarded since "," ES
ρ = 2 (parents per Child)
…
2
…
…
Selection of the fittest chromosomes
End of isolation



It is possible to make a couple of additional remarks on this notation: nesting in orders higher than two {..[..(..)..]..} is theoretically possible but it doesn't actually deliver real advantages [3]. It has been noted [3] that unfortunately this notation is not apt to describe all the possible combinations for an ES. In fact all populations run for the same amount γ of cycles while it would be very interesting to have a "control group", a population that would continue in isolation while the other populations follow the standard ES. Similarly the / operator doesn't define a specific recombination method.

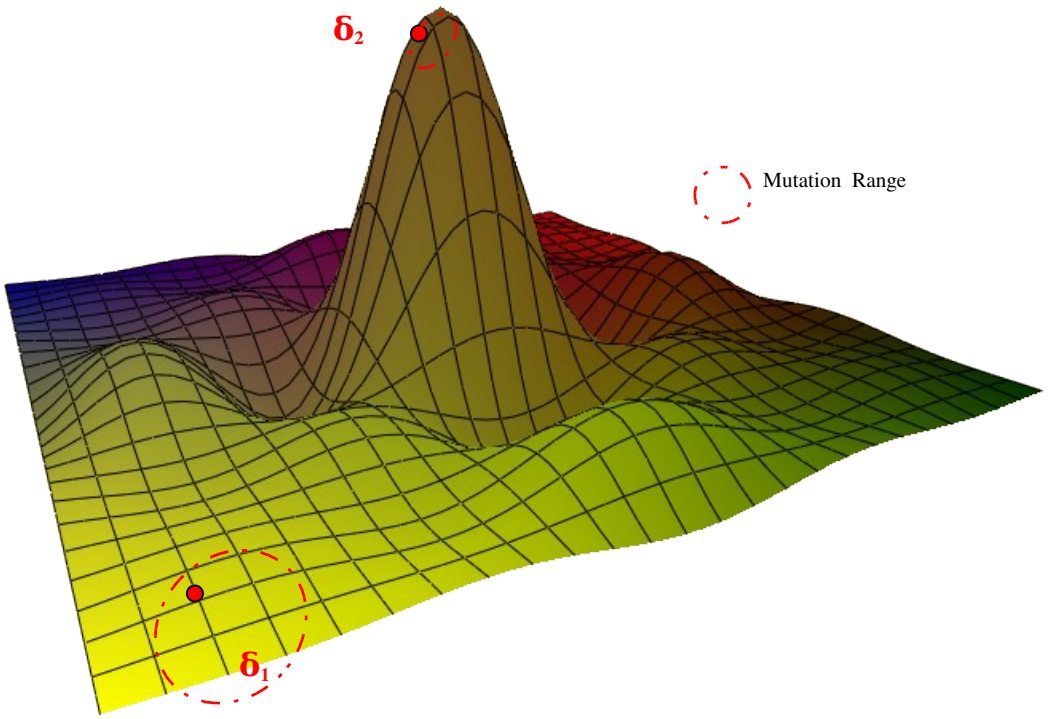
3.8- Progress window and adaptive mutations

The basis mutation algorithm that Rechenberg used mutates the j^{th} parent vector x_{Pj} of the g^{th} generation by adding a random vector z_i in order to obtain the i^{th} child vector x_{Ci}. The formula he used was:

$$(a)\quad x_{Ci}^{g} = x_{Pj}^{g} + \delta \cdot z_i$$

Being δ a multiplication factor that is used to determine the step of the ES and the z_i vector components $<z_1, z_2, \ldots z_n>$ normally distributed random numbers with $\sigma = \frac{1}{\sqrt{n}}$ variance. Such variance is needed to ensure the length of the |z| of the random vector remains centered on 1 independently from the number n of components that compose the z vector.

The problem of finding the suitable value for δ is anything but trivial since a good ES with the wrong δ is most likely to fail its goal. We can represent all the possible mutations obtained by adding the random vector z to our chromosome x with a hypersphere centered on x. Since the components of z are normally distributed the nearer we get to the centre the denser the hypersphere is. By changing δ we actually change the radius of our hypersphere, this means the influence range of the mutation operator. If we want to achieve an optimal search we must then adjust the value of δ to the quality function that we are dealing with. Moreover even inside the same quality function it might be necessary to regulate δ according to the current position on the quality function's surface. Let's make an example, imagine that we have a relatively flat quality function with a very steep hill in the centre.

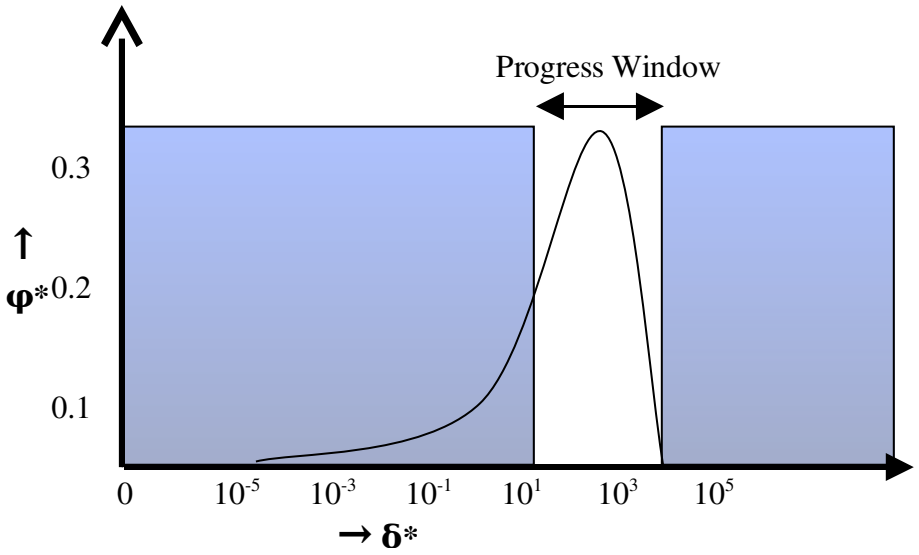


Z = 8*(sin(x)/x)*(sin(y)/y)

It is evident that δ_1 should have a bigger value to accelerate convergence while δ_2 should be small enough to allow a thorough search of the immediately surrounding space. Rechenberg has analytically examined various quality functions such as the sphere model and the corridor model (an elongated cube) and found functions that determine the optimal value for δ. However this task is difficult, especially for complicated quality functions, and it must be considered that most quality functions are very far away from a simple cube or sphere, not to talk about those cases where the quality function is not explicitly known. As a conclusion we

can regard δ as part of the optimization problem itself, with the consequence that it must be constantly changed according to the current state of the optimization algorithm.

To visualize this Rechenberg defined the evolution window showing the influence of δ on convergence speed. The progress window graphically displays the efficiency of the different δ values, in terms of the advance-speed φ.



Where φ* = φ/ω is the universal rate of progress and δ * = δ/ω the universal mutation step size. The denominator ω is the median radius of curvature of the contours of the quality function hills in neighborhood. In other words, ω is a local measure of the non linearity of the search space, a gauge for the measurements. However, without going too much in detail, what interests us is the meaning of the graph; it shows that the amount of mutation applied during the cloning process (x axis) determines the speed of the progress (y axis), which traduced is how fast we will reach an optimum solution. If the mutation is too big (right blue square area) there will be no progress at all, while if the mutation is too small (left blue square area) the ES might not be efficient enough to obtain results in acceptable times. The space among the two blue squares is the "evolution window" and represents the step of mutation that will produce the best results. The question is then, how is it possible to obtain a suitable δ?

In the case of a (μ , λ) the simple kind of mutation previously introduced can be extended like indicated from Schwefel in order to obtain a much more performing mutation that evolves the increment variable δ. In the g^{th} generation each child vector x_C is created mutating a random parent vector x_{Pi} by adding a random vector z formed like previously described.

(a) becomes then:

$$x_{C1}^{g} = x_{Pi}^{g} + \delta_{C1}^{g} \cdot z_1 \qquad x_{C2}^{g} = x_{Pj}^{g} + \delta_{C2}^{g} \cdot z_2 \quad \ldots x_{C\lambda}^{g} = x_{Pk}^{g} + \delta_{C\lambda}^{g} \cdot z_\lambda$$

Where i,j,k,... are random numbers [1,2,... λ] and

$$\delta_{C1}^{g} = \delta_{Pi}^{g} \cdot \xi_1 \quad \delta_{C2}^{g} = \delta_{Pj}^{g} \cdot \xi_2 \quad \ldots \qquad \delta_{C\lambda}^{g} = \delta_{Pk}^{g} \cdot \xi_\lambda$$

With ξ established either randomly or deterministically. For the determination of ξ Rechenberg [5] recommends:

"IF RND < 0.5 THEN $\xi_i = \alpha$ ELSE $\xi_i = \frac{1}{\alpha}$ " with α = 1,3 for n < 100 and lower values of α for n > 100

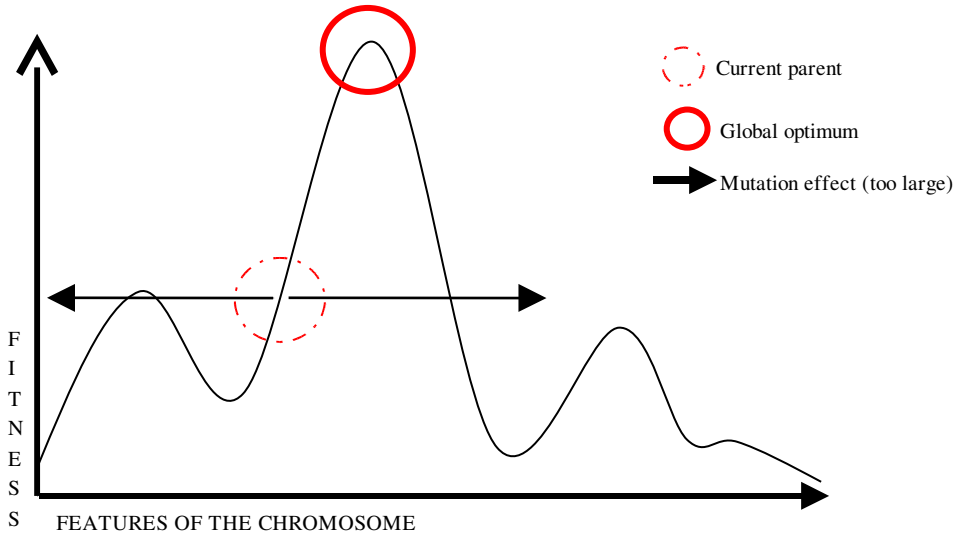
Similarly to the chromosomes, the mutation steps δ undergo in this way a selection process where each δ is coupled with a determined chromosome, almost extending it with a "mutation amount parameter" that is refined through the generations. This happens because the δ of the $(g+1)^{th}$ derives from the best of the δ belonging to the g^{th} generation. The δ dependency on the parent's δ follows the selection schema[5]:

$$x_{P1}^{g+1} = x_{CB1}^{g} \qquad x_{P2}^{g+1} = x_{CB2}^{g} \qquad \ldots \qquad x_{P\mu}^{g+1} = x_{CB\mu}^{g}$$

$$\delta_{P1}^{g+1} = \delta_{CB1}^{g} \qquad \delta_{P1}^{g+1} = \delta_{CB1}^{g} \qquad \ldots \qquad \delta_{P\mu}^{g+1} = \delta_{CB\mu}^{g}$$

Where the Bi index indicates the i^{th} best child chromosome.

The reason because the adaptive step is not used in a (μ + λ) is that, it might happen that all the parent chromosomes have a too large step value. What happens when the step is so large is that the mutations "jump over" the optimum. In such case all the children chromosomes produced would have a lower quality than the parents, resulting in infinite generations with the same set of parents.



In contrast (μ , λ) ESs don't suffer from this problem since no parent is ever lasting.

Now let's see a similar alternative step adaptation method also belonging to Schwefel. This time the chromosome <x> must be expanded to <x,σ> where x represents the original chromosome and σ is the vector of the standard deviations. This means that each chromosome becomes two times longer and carries the information of its mutability. For the recombination of two vectors the normal techniques of vector mask or median value can be used:

$<(x_1, x_{2,\dots} x_n), (\sigma x_1, \sigma x_{2,\dots} \sigma x_n)>$ recombined with $<(y_1, y_{2,\dots} y_n), (\sigma y_1, \sigma y_{2,\dots} \sigma y_n)>$

Gives $<(z_1, z_{2,\dots} z_n)(\sigma z_1, \sigma z_{2,\dots} \sigma z_n)>$ where z_n σz_n are randomly chosen from one of the parents or represent the median value of their respective parent components.

In this strategy mutation follows the schema:

$$\sigma_{new} = \sigma_{old} * e^{N(0,D)}$$

$$X_{new} = X_{old} + N(0, \sigma_{new})$$

Where N(0,D) is a random number with normal distribution, centered on zero and with standard deviation D. This is basically an alternative to Rechenberg's step adaptation

mechanism; in this case it is D that determines the step of the adaptation for the mutation parameter. The term $e^{N(0,D)}$ ensures for an always positive standard deviation. Because of the normal distribution $e^{N(0,D)}$ returns values that are very near to 1, slightly smaller for negative exponents, and slightly bigger for positive ones, this has as consequence that the step is quite small.

Chapter 4

Genetic Algorithms

4.1- Holland the father of genetic algorithms

Genetic algorithms (from now on GA) have originated from the work of John Henry Holland, whom, although contemporary of Rechenberg, worked independently and somehow starting from other premises. Even if Holland's best known book "Adaptation in Natural and Artificial Systems" [15] is dated 1975 he had been working on GAs since at least 1962. Holland had studied at the MIT and as an expert computer programmer, worked for the IBM on what would become the first commercial scientific computer the "701". Like Rechenberg also Holland got his inspiration from a book, "The Genetic Theory of Natural Selection" which provided him with the idea that evolution is training animals to adapt to the environment; the subsequent logical step was to solve the question if this mechanism would also be successful with computer programs. What really made Holland's work interesting was the idea of crossover, in Holland's own words [14]:

"*... in the early 1960s, Hans J. Bremermann (1926 Bremen - 1996) of the University of California at* Berkeley *added a kind of mating: the characteristics of offspring were determined by summing up corresponding genes in two parents. This mating procedure was limited, however, because it could apply only to characteristics that could be added together in a meaningful way.*".

Therefore Holland would proceed to develop what he called Genetic Algorithms, a programming technique that used mutation and crossover, the introduction of which proved to be a breakthrough. Later on he worked on the representation of computer programs in a suitable genetic code and introduced the idea of schemata for the construction of a more rigorous GA theory.

4.2- Similarities and differences wit ESs

The first difference between evolution strategies and genetic algorithms is that Holland started its research with a deep interest on what GA could make for computers, something quite different from the original Rechenberg's approach, a manual effort to solve a practical engineering problem. Consequently the problem codification and the solving techniques reflect the kind of problems that the each author originally intended to solve. A second difference is that ESs use only the quality function to build the ranking list of the chromosomes and subsequently proceed to the creation of the next generation following the schemata that employ the quality value previously assigned to each chromosome. On the other side, GAs take a slightly different strategy. First it must be said that the function assigning the "quality" values to each chromosome is not anymore called "quality function" rather *evaluation function*. Then a new function must be introduced, the *fitness function*; this function measures, on the basis of the value returned from the evaluation function, which is the probability that a certain chromosome takes part in the creation of the next generation. This reflects a different approach in comparison with ESs, while in ESs the possibility for a chromosome to be chosen as parent was equal for all the chromosomes (uniformly distributed p=1/n) now it can be determined by the fitness function and its independent input variable the evaluation of the chromosome. It follows that in many cases the evaluation of a chromosome is NOT equal to its fitness. Is important to note that originally Rechenberg only mutation as operator, a technique not directly criticized but strongly opposed by Holland. Indeed Holland compared a GA using exclusively mutation to an enumerative algorithm. GAs use instead a variety of operators that will be examined later on but out of which stands crossover.

4.3- A framework for GAs

In order to allow a rigorous mathematical study of GAs Holland developed a mathematical framework and introduced it in his first book [15]. The construction of a formal framework was needed to define a generic outline that could be found in adaptation problems belonging to different fields. For example, every time we consider an evolving system there will be an environment to which a certain structure adapts. The problem of identifying such system is a valid question both in a game theory problem and in an artificial intelligence one. So Holland tried to include all the salient features of a common adaptation problem into a single common framework that could eventually permit to compare and possibly translate the results found in a determined field into another field of adaptation problems.

Holland recognized that independently from the particular adaptation problem to solve there are always determined *structures* that are modified with the passing of time; e.g. the well known chromosomes in the field of genetics or the mixes of goods in an economic planning problem, maybe the structure of a network in AI. As said before, such structures vary over time and to do this, some *operators* that change the structure's state are needed, for

chromosomes these operators might be the already well known mutation and recombination while for economic planning the production activities are responsible for the change of the good mixes. The sequence with which such operators are applied to the structures generally responds to a well defined *plan* and such plan can be seen as a trajectory through the set of all possible structures. Generally it is implied that the plan acts on a discrete time basis, to indicate this, the generational variable t = 1, 2, 3... is used. Although it might be seem confusing to use the same designation for cycles that could represent seconds as well as centuries the positive aspect of this is the abstractness of the framework from its actual environmental values, an advantage for the previously advocated comparisons among different fields. Now we are faced with the problem to identify whether the plan's use of an operator is steering our structures' evolution in the right direction or not, if we wish to measure this it is important to define a *performance measure* that will indicate how "well" our structure(s) is performing. However the performance of a certain structure depends on its *environment* too, this might be certain socio-economic circumstances, the current cell type or even the input/output connections of a neural network. To make a couple of examples [15]:

Field	**Structures**	**Operators**	**Performance Meas.**
Genetics	Chromosomes	Mutation, etc..	Fitness
Economic Planning	Mixes of goods	Production activities	Utility
Physiological psychology	Cell assemblies	Synapse modifications	Performance rate
Game theory	Strategies	Rules for iterative approximation of optimal strategy	Payoff
Artificial Intelligence	Programs	Learning rules	Comparative efficiency

The kind of problems that an adaptation process is faced with mostly share common features such as:

- A large set of possible structures.
- A very complicated performance measure showing local optimums, discontinuities, non linearity....
- Time variable performance measures and environments.
- The filtering of huge amounts of data in order to find the relevant information needed by the plan.

At this point we can start to formally introduce Holland's framework:

- E, is the environment of the system undergoing adaptation.

- τ, is the adaptive plan which determines successive structural modifications in response to the environment.
- μ, is the measure of performance of different structures in the given environment.
- $\Lambda = \{A_1, A_2, ...\}$, is the (non empty) set of attainable structures.
- $\Omega = \{\omega_1, \omega_2, ...\}$ is the set of all operators used for modifying the structures.
- I, is the set of all possible inputs to the system from the environment.

The structures belonging to A can be composed of sub-structures, exactly like chromosomes that are made of genes. If each element of Λ is composed of n genes and the i^{th} gene g_i can be occupied with one allele of the set $\{a_{i1}, a_{i2}, ... a_{iki}\}$ then all the possible structures of Λ are the set of all possible combinations of alleles,

$$\Lambda = g_1 \text{ x } g_2 \text{ x } ....\text{x } g_n = \prod_{i=1}^{n} g_i$$

The adaptive plan τ generally produces a different trajectories of structures for different environments, let's then be $\Lambda(t) \in \Lambda$ the structure examined at time t, in order to explore the environment responses to $\Lambda(t)$ the plan uses a set of "sensors" δ_i. The symbol I represents then the range of *stimuli* that τ can receive, this range is obtained in the following manner:

$$I = I_1 \text{ x } I_2 \text{ x } ... \text{ x } I_n = \prod_{i=1}^{n} I_i$$

Being $I_{\Lambda(t)} \subset I$ the single stimulus perceived at time t by trying the structure $\Lambda(t)$ in the environment. Let's make the example of a neural network with k binary inputs, then

$I = I_1 \text{ x } I_2 \text{ x } ... \text{ x } I_k$ with $I_i = \{0, 1\}$ and $I_i(t) \in I(t)$ will represent the actual inputs state of the δ_i sensor at cycle t. Let's $<I(1), I(2), ..., I(t-1)>$ represent the temporal sequence of the sensors' inputs, and I(t) the current signal. It is often helpful to retain information in the *memory* M about the input history $<I(1), I(2), ..., I(t-1)>$. Such memory information will be then used by the plan, in conjunction with the current structure and input in order to determine the next structure and memory state:

$$\tau: I \text{ x } (\Lambda_1 \text{ x } M) \rightarrow (\Lambda_1 \text{ x } M)$$

But mostly τ will be referred to in its simplified form:

$$\tau: I \text{ x } \Lambda \rightarrow \Lambda$$

This version of τ is clearly deterministic but, as already seen, evolution is a non deterministic procedure that assigns a certain reproduction probability to each individual. Therefore a stochastic version of the simplified τ is:

$$\tau: I \times \Lambda \rightarrow P$$

Where *P* is the set of admissible probability distributions over Λ. That means that Λ(t + 1) is chosen randomly from Λ using the probability distribution *P*(t + 1). A more detailed description of the previous plan would be:

$$\tau: I \times \Lambda \rightarrow \Omega$$

Where the plan doesn't return a structure but rather an operator to be used on the structure following the interpretation:

$$\tau(I(t),\Lambda(t)) = \omega_t \quad \Omega$$

and

$$\omega_t(\Lambda(t)) = P(t+1)$$

In this case the plan deterministically chooses an operator ω_t which determines the modification of the structure stochastically. Such interpretation includes the previously stochastic and deterministic versions. As a matter of fact the deterministic version of τ can be regarded as a particular stochastic version that returns a probability distribution with value 1 on a single structure.

τ is called adapting plan for the reason that it has to interact with an unknown environment and modify the structures so that they suit best the current environment.

The influence of the environment on the plan is determined by the information that the plan receives, that is the set I of its inputs. Among such inputs there is the so called *payoff* function $\mu_E(\Lambda(t))$, which calculates the quality of the current structure Λ(t) in the environment E.

$$\mu_E: \Lambda \rightarrow$$

Those plans that have the payoff function as only input:

$$I(t) = \mu_E(\Lambda(t))$$

Are called payoff-only and when it comes to plan comparison they represent the efficiency lower bound. As a matter of fact any plan that has additional inputs besides the payoff function is supposed to perform at least as well as a plan that receives the payoff function as only input. At this point the adaptation problem has shifted one level up, we don't search anymore a single structure that performs well in a determined environment but rather a plan that is capable to efficiently evolve structures. We must therefore introduce a *criterion* χ to compare the efficiency of the different plans in different environments. Let's first define de *cumulative payoff* for a deterministic plan as:

$$U_{\tau,E}(T) = \sum_{t=1}^{T} \mu_E(\Lambda(\tau,t))$$

With:

$\Lambda(\tau,t)$ representing the structure selected by the plan τ at time t

$\mu_E(\Lambda(\tau,t))$ returning the payoff of the structure above

On the other side, the cumulative payoff of a stochastic plan is:

$$U_{\tau,E}(T) = \sum_{t=1}^{T} \overline{\mu}_E(\tau,t)$$

With:

$\overline{\mu}_E(\tau,t) = \sum_j P(A_j,t)\mu_E(A_j)$ representing the sum of payoffs multiplied by the probability of that payoff.

If :

$U^*_E(T) = \operatorname{lub}_{\tau \in \Gamma} U_{\tau,E}(T)$ designates the lowest upper bound of all $U_{\tau,E}(T)$

And

ε designates the set of all possible environments

Γ designates the set of all possible plans

We can say that a plan τ is robust in ε with respect to the asymptotic optimal rate criterion for Γ when

(1) $\underset{E\ \varepsilon}{\text{glb}}\ \underset{T}{\lim}[\frac{U_{\tau,E}(T)}{U^*_E(T)}] = 1$ (glb = greatest lower bound)

And we can *say* that τ is robust in ε with respect to the interim behavior criterion $<c_T>$ for Γ when, for each T

(2) $\underset{E\ \varepsilon}{\text{glb}}[\frac{U_{\tau,E}(T)}{U^*_E(T)}] > (1 - c_t)$

Equation (1) indicates that in the limit the cumulative payoff $U_{\tau,E}$ of the chosen plan *increases* with the same rate of the best possible plan U*.

Equation (2) is a more restrictive criterion that uses the c_t series. The c_t is an arbitrary series that tends to 0, e.g. $\frac{k}{(k+T)^j}$. Therefore criterion (2) forces the series $U_{\tau,E}$ to converge to the lub with a higher rate than the one for which c_t converges to 0.

4.4- Chromosomes as samples of schemata

With help of the previously defined framework Holland explored the GAs in order to find the *mechanism* at the source of their efficiency. He found out that GAs where intrinsically parallel, to explain this he consequently introduced the notion of schemata. Each structure is composed of a set of values which are returned by the detectors δ_i:

$$\{\ \delta_i : \Lambda \to V_i, i = 1,2,...,l\ \}$$

So each *structure* A will be represented by a vector of l attributes given by the detector values:

$$A \in \Lambda = (\delta_1(A), \delta_2(A), ..., \delta_l(A))$$

with

$$\delta_i(A) \in V_i, i = 1,...,l$$

At this point we will introduce a "wildcard" symbol * which stands for any possible value. Let's make an example where the admissible values of δ_i are $V_i = \{v_1, v_2\}$, an the length l of

the vector 3. Then: $A_1<v_1, v_2, v_2>$, $A_2<v_2, v_1, v_2>$, $A_3<v_1, v_1, v_2>$ represent three different structures. And $S_1<v_1,*,*>$ represents the set of all the structures that have the value v_1 at the first position, therefore A_1 and A_3 belong to the S_1 schemata. If on the contrary we consider $S_2<*,*,v_2>$, A_1 as well as A_2 and A_3 belong to this schema. More formally schemata are those l-tuples belonging to Ξ with:

$$\Xi = \prod_{i=1}^{l} \{V_i \quad \{*\}\}$$

an l-tuple $A <a_1, a_2,..., a_l>$ belongs to a particular schemata $\xi <\Delta_1, \Delta_2,..., \Delta_l>$when:

(I) For each $\Delta_i = * \rightarrow a_i \in V$
(II) For each $\Delta_i = v \in V \rightarrow a_i = v$

And we say that a certain *schema* is of order k when it has k defined values, $\Delta_i \neq *$. E.g:

$S_1<v_1, *, *>$ is of order 1
$S_2<v_1, *, v_2>$ is of order 2

Basically, the lower the order of a schema the more chromosomes it comprehends; as a matter of fact it is possible to imagine each vector as a point in the search space and each schema as a subspace of that space. It is obvious that the lower the schema order is the bigger the subspace will be; in a binary search space <1, *, *, *> will collect half of all the possible solutions, <1, 0, 1, *> just two of them.

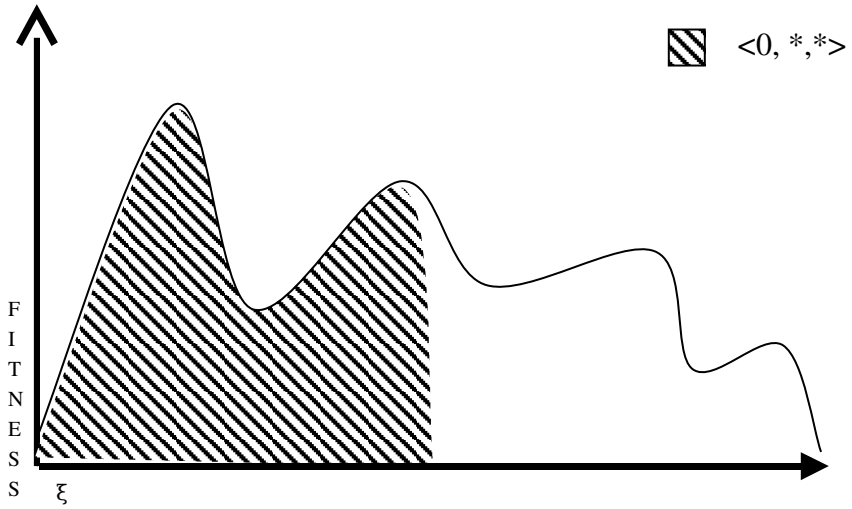
Another significant measure for schemata is the *defining length*, said in other words the distance among the outermost defined values. E.g.: <1, 0, 1, *> has a defining length of 2, and <1, 1, *, 0, *> has a defining length of 3. The defining length of a schema has notable implications in the possibility of destroying a schema during crossover, the lower it is (i.e. the defining bits are compactly packed) the hardest it will be for crossover to separate them and the higher the chance for the schema to survive will be.

Once defined a schemata ξ, it is possible to calculate its *observed payoff* μ_ξ (the average fitness of its trials) and compare it to the current average $\bar{\mu}(T)$ in order to determine whether the structures of ξ show an above the average payoff. If $\mu_\xi > \bar{\mu}(T)$ then new instances of ξ will be created; with the result that the estimation of μ_ξ will be more precise and that the average payoff of the whole population $\bar{\mu}(T)$ is augmented.

Collecting data for the calculus of many μ_ξ can be done by analyzing relatively few structures since each one of them is a sample of a great number of different schemata, this is where the intrinsic parallelism lays. To be exact each structure A_i is instance of 2^l different schemata and it is easy to see why. Given a structure A_i represented by a vector of length l $<v_1, v_2, \ldots v_l>$ it is possible to create a binary mask vector of length l <1, 0, …, 0> where each 1 means that the n^{th} value of the A_i vector must be substituted for the wildcard symbol , it is immediately clear that all the possible combinations of the mask vector are 2^l each one corresponding to a different schema. This means that the performance calculus for a single structure gives us information about a myriad of different schemata. In this way what is being evaluated is not the single structure A_i, rather the attributes of the schemata of which it is instance and their correlations; it is like this that the GA finds and samples with higher frequency the so called "building blocks", allele combinations that present a higher performance. E.g. examining the chromosomes

<0, 0, 0> <0, 0, 1> <0, 1, 0> <0, 1, 1>

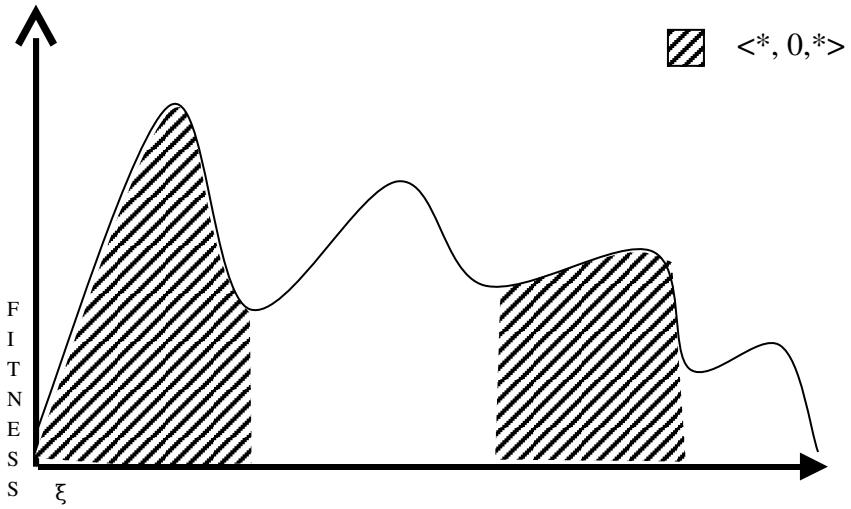
We will implicitly gain information about <0, *,*>



While

<0, 0, 0> <0, 0, 1> <1, 0, 0> <1, 0, 1>

Are instances of <*, 0,*> thus providing information about that schema.



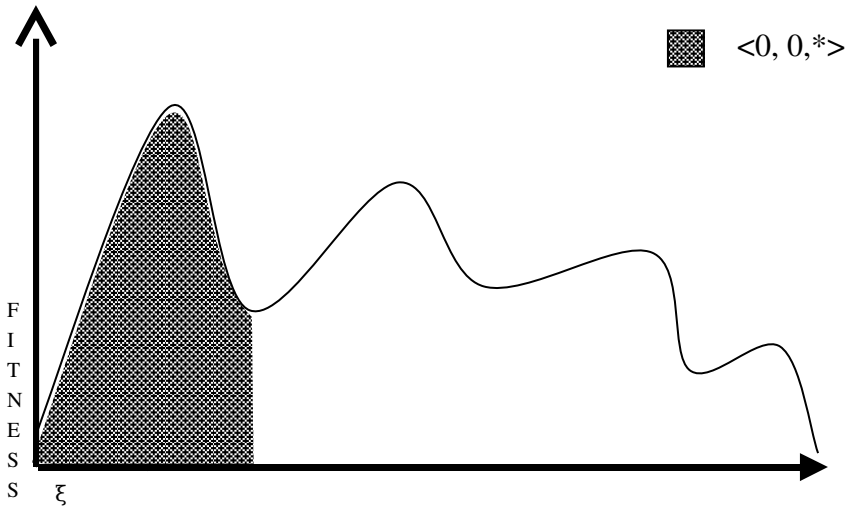
And finally, the original 6 trials:

<0, 0, 0> <0, 0, 1> <0, 1, 0> <0, 1, 1> <1, 0, 0> <1, 0, 1>

Have also intrinsically sampled all the instances of the following 7 different 2nd order schemata:

<0, 0, *> <0, 1, *> <1, 0, *> <0, *, 0> <0, *, 1> <*, 0, 0> <*, 0, 1>

Which represent the 7 of the 12 possible 2nd order schemata. Based on this information it is possible to identify the <0, 0,*> schema, resulting in the detection of a "building block" of superior performance.



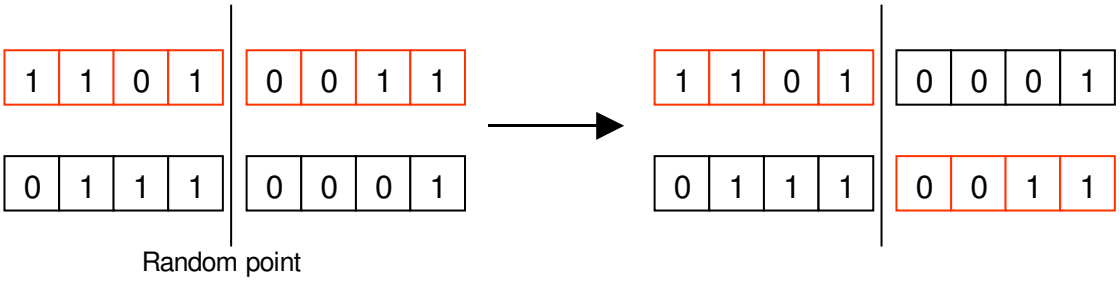
At this point it could be possible to see a genetic algorithm as a biased Monte Carlo method where the whole space is sampled with different frequencies, each determined by the average fitness of the vectors that describe that particular subspace. It is the fitness function's duty to determine how such schemata will influence the following generation composition, or in other words, the next sampling cycle.

4.5- Some operators

Simple Crossover:

Crossover could be described as the fundamental operator in GAs, in fact it is the operator that enables the exploration of the search space by mixing the available solutions into new ones. Crossover has basically two possible outcomes, the first is the creation of an instance belonging to a schema that is already present among the population, which means another sample for that schema and the subsequent refinement of its expected fitness. The second possible outcome is the creation of a chromosome whose schema is not present in the population, thus exploring new combinations in the search space. In GAs crossover is a little different if compared with the mask-crossover of ES in that it doesn't exchange information on a gene by gene basis, instead it swaps groups of contiguous genes. The following explanation refers to the one point crossover, i.e. where each chromosome is divided in two groups of genes. Obviously this is the simplest version though it is easy to modify it by augmenting the number of gene groups that compose each chromosome.

1. Given two structures A = g_1, g_2, g_3, ..., g_l and A' = g'_1, g'_2, g'_3, ..., g'_1 composed of l genes, select a random number x from {1, 2, …, l -1}
2. Create two new structures by exchanging among the original structures all the genes on the right of x resulting in:
   A = g_1, g_2, g_3, ..., g_x, g'_{x+1}, …, g'_1 and A' = g'_1, g'_2, g'_3, ..., g'_x, g_{x+1}, …, g_l



Mutation:

Although the first experiments that Rechenberg made obtained astonishing results with the only help of mutation, Holland is quite critic of an algorithm using the mutation operator only. That kind of plan is, Holland says, nothing more than an enumerative plan with a little bit of "memory effect" thus with a very limited applicability. In fact mutation only plans do not choose structures absolutely at random they rather explore the neighborhood of the best structures found up to date but they do not take into consideration the observed fitness when deciding which of the structures must be modified. In GAs mutation serves an absolutely different purpose, it doesn't represents an operator for the exploration of the search space instead a modifier if the genetic diversity. As a matter of fact Holland says that mutation is a background operator whose only goal is to provide crossover with the necessary variety of chromosomes. If the population has a small size when compared to the set of all possible structures it is evident that many combinations will remain untested if only crossover is being used. Mutations try to supply to this lack of alleles by randomly changing some of them. The other role mutation is the avoidance of the local optima in which the population might converge if the grade of homogenization of the population is too high. The mutation operator proposed by Holland has a quite simple functioning:

1. Each gene of the chromosome A = g_1, ..., g_x, ..., g_l has a small chance of being mutated.
2. The alleles of those genes that are mutated are randomly selected with uniform probability from the V_i possible values that can be assigned to the gene, the remaining alleles remain untouched.

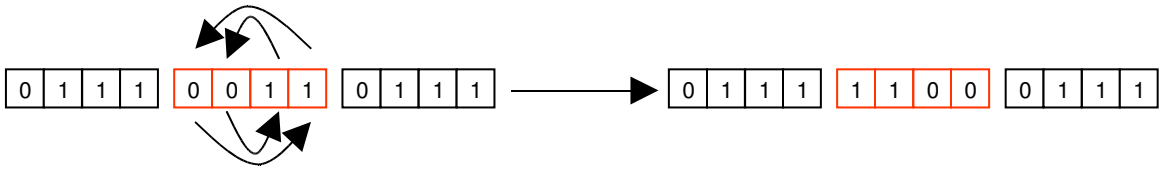
Inversion:

The term linkage is used to describe the compactness of some alleles in the chromosome, as we know that the closer two alleles are the lower the probability that some genetic operator separates them is, so the linkage of two alleles has notable consequences for their evolution. Two adjacent alleles are strongly linked together, thus proving hard to be separated; if the combination of these alleles is particularly effective the "linkage" effect, which favors the transmission of such features, might be desired. It might also be advisable to deliberately introduce a linkage effect in alleles that produce a particularly fit solution but that are not actually close to each other. To do this Holland introduced a new operator called inversion whose goal is to change the length of a schema by modifying its instances. To allow inversion to take place the coding of the genes must be changed, as a matter of fact the meaning of an allele is determined by its position on the gene, for example: <…, …, 4> is not the same as <4, …, …>. This is an obstacle to the relocation of alleles inside the chromosome and to circumvent this, the codification of genes must be altered so that alleles have the same interpretation no matter where they occur in the chromosome. Up to now the chromosome has been composed of single values whose meaning was determined by their location, now each gene will be composed of a pair of values g = (i, a) where the i is the index value, which identifies the interpretation of the couple, and a the actual value of the allele. So the <5, 6, 9, 3, 4, 5> vector becomes <(1,5), (2,6), (3,9), (4,3), (5,4), (6,5)> which is equivalent to the <(4,3), (1,5), (6,5), (3,9), (2,6), (5,4) > vector because every index is associated with the same allele value, thus producing the same interpretation for both chromosomes. The functioning of inversion is the following:

1. Given a structure $A = g_1, g_2, g_3, ..., g_l$ composed of l pairs of values $g = (i, a)$

2. Select two numbers from $\{0, 1, 2, ..., l+1\}$ using uniform random selection, label the smaller as x_1 and the bigger as x_2

3. Invert the segment of A that lays among x_1 and x_2 by setting
$A = g_1, ..., g_{x1}, g_{x2-1}, g_{x2-2}, ..., g_{x1+1}, g_{x2}, ..., g_l$



The case of g_{x1} and g_{x2-1} is significant as two alleles that were originally far away from each other are brought close together after inversion has taken place. This doesn't only transform the single chromosome but also the different schemata of which it is an instance. It follows

that also inversion is intrinsically parallel and modifies the defined values of those schemata that produced the inverted instance. As compact schemata of lower order tend to reproduce more often, to be less subject to crossover and inversion, it is evident that there is a meta-selection process acting on schemata where inversion is randomly moving the contents of the schemata. The fact of having changed the coding of a chromosome to couples of values has some significant consequences for the GA; for instance two chromosomes whose index sequence is different such as:

<(1,5), (2,6), (3,9), (4,3), (5,4), (6,5)> and <(4,3), (1,5), (6,5), (3,9), (2,6), (5,4) >

cannot be directly crossed over without the risk of obtaining two child chromosomes with some alleles duplicated and some other missing such as: <(1,5), (2,6), (3,9), (3,9), (2,6), (5,4) > to avoid this Holland proposes two possibilities. The first is to allow only homologous chromosomes to crossover; that is, to consent to crossover only those chromosomes that share the same sequence of index values (therefore coded in the same way). This method faces the difficulty of finding two homologous chromosomes in the current population by randomly selecting them, there is, however, another way. The second possibility is to temporary adapt the coding of one chromosome to the other obtaining in this way two homologous chromosomes capable of crossing over and, once the crossover has taken place, to return to the original coding.

4.6- A basic GA

The first and simplest kind of plan that Holland introduced was labeled as R_1 plan, it is a generalized type of plan that presented with help of the previously introduced framework.

- B(t) is the set of M structures manipulated by the plan at time t
- Λ_1 is the set of all available structures
- ρ assigns an operator to a structure A in order to modify it
- ω is an arbitrary operator which, using the selected structure $i(t)$ B, returns the distribution P over Λ_1

Holland's [15] Plan type R_1:

1. Set t = 0 and initialize B by selecting M structures at random from Λ_1 to form B(0) = $\{A_h(0), h = 1, \ldots, M\}$
2. Observe and store the performances $\{ \mu_E(A_h(0)), h = 1, \ldots, M \}$ – go to point 4

3. Observe the performance of A'(t) and replace $\mu_E(A_{j(t)}(t))$ by $\mu_E(A'_h(t))$
4. Increment t by 1
5. Select one structure $A_{i(t)}(t)$ from B(t) by taking one sample of B(t) using the probabilities

$$\text{Prob}(A_h(t)) = \frac{\mu_E(A_h(t))}{\sum_{h'=1}^{M} \mu_E(A_{h'}(t))}, \quad h = 1, \ldots, M$$

6. Determine the operator ω_t Ω to be applied to $A_{i(t)}$, $\omega_t = \rho(A_{i(t)}(t))$ and then use ω_t to determine a new structure A'(t) by taking a sample of Λ_1 according to the probability distribution $P_t = \omega_t(i(t), A_1(t), ..., A_M(t))$ P
7. Assign probability $\frac{1}{M}$ to each number 1, …, M, select one number 1 $j(t)$ M accordingly, and replace $A_{j(t)}(t)$ by A'(t) – return to point 3

Plans of type $R_1(P_C, P_I, {}_1P_M, \langle c_t \rangle)$ are a slightly more complicated version of the previous plan, the difference lays in that instead of the generic operator ω, they explicitly use crossover, inversion and mutation operators at each cycle with a determined probability. This kind of plan is the one used in the schema theorem later on, and will be therefore the reference for the demonstration of the GAs' robustness.

[15] GENERIC Plan $R_1(P_C, P_I, {}_1P_M, \langle c_t \rangle)$:

- Pc as the constant probability of applying crossover to a selected individual
- Pi as the constant probability of applying simple inversion to a selected individual
- ^{1}Pm as the initial probability of mutation of an allele (alternative alleles uniformly distributed)
- c_t a sequence wit the following conditions:
  - 0 c_t 1
  - c_t 0
  - $_t c_t$

  And it used to decrease the mutation rate as time passes by.

1. Set t = 0 and initialize B by selecting M structures at random from Λ_l to form B(0) = $\{A_h(0), h = 1, \ldots, M\}$
2. Observe and store the performances $\{ \mu_E(A_h(0)), h = 1, \ldots, M \}$ to form U(0)
3. Calculate $\mu(0) = \frac{\sum_{h=1}^{M} \mu_h(0)}{M}$ - go to 7
4. Observe the performance $\mu_E(A'(t))$
5. Update $\mu(t)$ by calculating $\mu(t) + \frac{\mu_E(A'(t))}{M} - \frac{\mu_{j(t)}(t)}{M}$
6. Update U(t) by replacing $\mu_{j(t)}(t)$ with $\mu_E(A'(t))$
7. Increment t by 1
8. Define the random variable $Rand_t$ on $\Omega_M = \{1, \ldots, M\}$ by assigning the probability $\frac{\mu_h(t)}{\mu(t)}$ to $h \in \Omega_M$. Make one trial of $Rand_t$ and designate the outcome i(t).
9. Apply simple crossover to $A_{i(t)}(t)$ and $A_{i'(t)}(t)$ with probability Pc where $A_{i'(t)}(t)$ is determined by a second trial of $Rand_t$. Select one of the resultants at random (uniform distr.) and designate it as ${}^1A_{i(t)}(t)$
10. Apply simple inversion with probability Pi, yielding ${}^2A_{i(t)}(t)$
11. Apply mutation to ${}^2A_{i(t)}(t)$ with probability $c_t{}^1Pm$ yielding A'(t)
12. Make a random trial with uniform distribution to each $h \in \Omega_M$, designate the outcome j(t)
13. Update B(t) by replacing $A_{j(t)}(t)$ with A'(t) - go to 4

The actual grounds for the efficacy of this kind of algorithm will be analyzed in the following chapters using again the framework introduced by Holland.

4.7- The 2-armd bandit

A GA is initially faced with the absolute uncertainty about the search space and, as it explores the current chromosomes, it gains more and more information about different schemata. But one problem remains, whether it should allocate the trials of the next generation to those already known schemata that present an above-the-average payoff or to use those trials to explore the remaining schemata in search for an even better schema. This problem of exploitation vs. exploration tradeoff is commonly known as the "Multi-armed bandit" and it is a problem that has been widely studied in many fields. If we wish to demonstrate the quality of genetic algorithms we must understand whether they allocate the samples correctly or not.

The Multi-armed bandit problem can be imagined as follows, there are X variables with different probability distributions and we want to obtain the maximum payoff from the sum of our N trials but we have no information about which variable has the highest mean. If we knew which variable has the highest mean the solution to the problem would be trivial, assign all the N trials to that variable but unfortunately this is not the scenario for a GA. So we have to find a strategy to assign the N trials maximizing the payoff or, in other words, minimize the losses. Holland himself explored first the 2 armed bandit problem and then its multi variable generalization, as for the first problem he hypothesized two schemata ξ and ξ' where $\mu_\xi > \mu_{\xi'}$. Here the difficulty is that if the two probability distributions overlap, as it generally happens, we can't be absolutely sure that after N trials the variable currently showing the highest payoff is actually the variable that has the highest mean. Now let's call $\xi 1$ the variable currently showing the highest payoff mean and $\xi 2$ the one with the lowest, where n trials have been assigned to $\xi 2$ and (N-n) to $\xi 1$. What we are actually looking for is n*, the value of n which reduces the payoff losses to a minimum; there are two possible sources of loss:

1. The observed worse ($\xi 2$) is actually the best (ξ), this means that all the (N-n) trials assigned to $\xi 1$ should have been assigned to $\xi 2$, the total loss is therefore $(N - n)*| \mu_\xi - \mu_{\xi'}|$.

2. The observed worse ($\xi 2$) is actually the worse variable (ξ'), this means that all the n trials assigned to $\xi 2$ should have been assigned to $\xi 1$, in this case the total loss is $n*| \mu_\xi - \mu_{\xi'}|$.

Let's now define q as the probability that ξ and ξ' are wrongly classified, respectively to $\xi 2$ and $\xi 1$. Then the total losses whit N trials are:

$$L(N-n,n) = q*(1) + (1-q)*(2) =$$
$$= q(N-n)(\mu_{\xi} - \mu_{\xi'}) + (1-q)n(\mu_{\xi} - \mu_{\xi'})$$

To find the minimum of the loss function L we must find where its derivative is equal to zero:

$$\frac{dL}{dn} = (\mu_{\xi} - \mu_{\xi'}) \left[-q + (N-n)\frac{dq}{dn} + 1 - q - n\frac{dq}{dn} \right] =$$
$$= (\mu_{\xi} - \mu_{\xi'}) \left[1 - 2q + (N-2n)\frac{dq}{dn} \right] = 0$$

Where dq/dn must be expressed in terms of n. To do this we define:

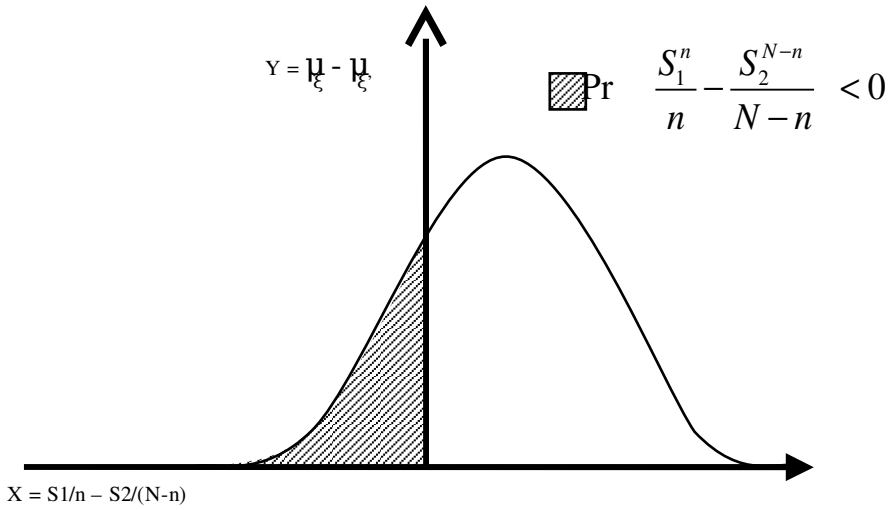
S_1^n as the sum of the n payoffs assigned to $\xi 1$

S_2^{N-n} as the sum of the (N-n) payoffs assigned to $\xi 2$

It follows that:

$$q = \Pr\left[\frac{S_1^n}{n} - \frac{S_2^{N-n}}{N-n} < 0 \right]$$

That means that q is expressed as the probability that the median payoff of $\xi 1$ is smaller than the median payoff of $\xi 2$, in other words that ξ and ξ' have been wrongly classified respectively as $\xi 2$ and $\xi 1$. Being q the result of two random variables it is also a random variable and to calculate the value of the previous expression means we must basically determine the area of its distribution curve that lays behinds zero.



Holland explored this theme and proposed an optimal allocation of trials which was later corrected by Dan Frantz [15] [16] reaching the conclusion that:

$$n^* \quad c_1 \ln \frac{c_2 N^2}{\ln(c_3 N^2)}$$

And

$$N - n^* \quad e^{n^*/2c_1} \sqrt{\frac{\ln(c_3 N^2)}{c_2}} - n^*$$

Where c_1, c_2 and c_3 are constants defined by Frantz. Since as n grows $e^{n^*/2c_1}$ tends to dominate, we can approximate the previous expression for a large n to:

$$N - n^* \quad e^{cn^*}$$

Where

$$c = \frac{1}{2c_1}$$

Summing up we can affirm that for the amount of trials allocated to the variable showing the highest mean must grow exponentially in relation to the trials assigned to the variable with the observed lowest mean.

4.8- The schema theorem

It is now our duty to show, in the light of the previous chapter, that genetic algorithms allocate trials to schemata in an optimal fashion. First we want to calculate the variation, from one generation to the next, of the number of instances belonging to a certain schema. Let's define:

- m(ξ, t) as the number of instances for the schema ξ at time t.
- E(m(ξ, t+1)) as the expected number of instances for the schema ξ at time t+1.

If the selection follows the $R_1(P_C, P_I, {}_1P_M, <c_t>)$ ([15]pag 125) then the expected offspring for each chromosome is proportional to its fitness reported to the entire population fitness:

$$\mathrm{E}(\mathrm{m}(\xi, \mathrm{t+1})) = \frac{f(\xi)}{\overline{f}(t)}$$

Where:

- $f(A)$ is the fitness of the chromosome A.
- $\overline{f}(t)$ is the average fitness of the whole population at time t.

Let:

$$f_{\xi}(\xi, t) = \frac{\sum_{A \in \xi} f(A)}{m(\xi, \mathrm{t})}$$

Be the observed average fitness of the instances of ξ at time t. It follows that the expected offspring for a chromosome is the product of the average fitness of all its instances and the number of instances in the current generation:

$$\text{(a)}\ \ \mathrm{E}(\mathrm{m}(\xi, \mathrm{t+1})) = \frac{\sum_{A \in \xi} f(A)}{\overline{f}(t)} = \frac{f_{\xi}(\xi, t)}{\overline{f}(t)}\ m(\xi, \mathrm{t})$$

It is obvious that the GA, working on a "one chromosome a time" basis, doesn't explicitly calculate the fitness and offspring of each schema; it is again the implicit parallelism that makes the little magic here. As many readers would have already noticed, (a) doesn't take the effects of mutation and crossover into account. The two operators can actually augment and diminish the number of instances that our schema has, let's then examine the *crossover survival probability* S_c. S_c indicates the lower bound for the probability that after undergoing simple crossover in one of its chromosomes, the schema is still present in the population. In this case S_c is considered in a probabilistic way and only destructive effects of crossover are being taken into account, in fact there are two additional possibilities that are not represented here. The first is the chance of acquiring new instances through crossover and the second is the case of two chromosomes belonging to the same schema crossed over, which obviously produces an offspring compliant with the parents' schema. However for simplicity we will calculate the lower bound S_c only and that will suffice for our demonstration purposes.

If

p_c is the crossover occurrence probability.
$d(\xi)$ is the defining length of ξ.
l is the length of the chromosome vector.

Then:

$$S_c(\xi) \quad 1 - p_c \quad \frac{d(\xi)}{l-1}$$

This means that the crossover survival probability depends on the crossover probability and the fraction of the schema that is occupied by the defined values of the schema. Now it's time to determine the disruptive effects of the mutation operator, once established that our mutation operator always changes the allele (i.e. an allele can't mutate into itself) we will define the *mutation survival probability* S_m in the same way we did for crossover: the possibility for schema H to survive after a mutation takes place in one of its instances. Likewise:

p_m is the mutation occurrence probability.
o(H) is the schema order (number of defined values)

Then:

$$S_m(\xi) = (1 - p_m)^{o(H)}$$

In other words, the mutation survival probability is the probability for a bit not to be mutated $(1 - p_m)$ raised to the o(H) power. In conclusion it's easy to note that the lower order the schema has the higher S_m is.

Finally we can complement (a) with the two survival probabilities:

$$\text{E}(\text{m}(\xi, \text{t+1})) \qquad \frac{f_\xi(\xi, t)}{\overline{f}(t)} \; m(\xi, \text{t}) S_c S_m;$$

$$\text{(b)} \;\; \text{E}(\text{m}(\xi, \text{t+1})) \qquad \frac{f_\xi(\xi, t)}{\overline{f}(t)} \; m(\xi, \text{t}) \;\; 1 - p_c \;\; \frac{d(\xi)}{l - 1} \qquad (1 - p_m)^{o(H)}$$

The interpretation of (b) is that the GA samples the different schema, exactly like the multi armed bandit solution suggests. This is the confirmation that GAs allocate trials in a nearly optimal way thus posing a valid solution method.

Let's make a rough example and imagine that our current schema has fitness 15% above the population average for the next x generations. Then:

$$\frac{f_\xi(\xi, t)}{\overline{f}(t)} = 1.15$$

$$m(\xi, \text{t}) = \text{E}(\text{m}(\xi, \text{t-1})) \text{ with } m(\xi, 0) = 1$$

Let's simulate a case by setting the mutation probability and crossover survival probability:

$$S_c(\xi) = 0.98$$
$$S_m(\xi) = 0.99$$

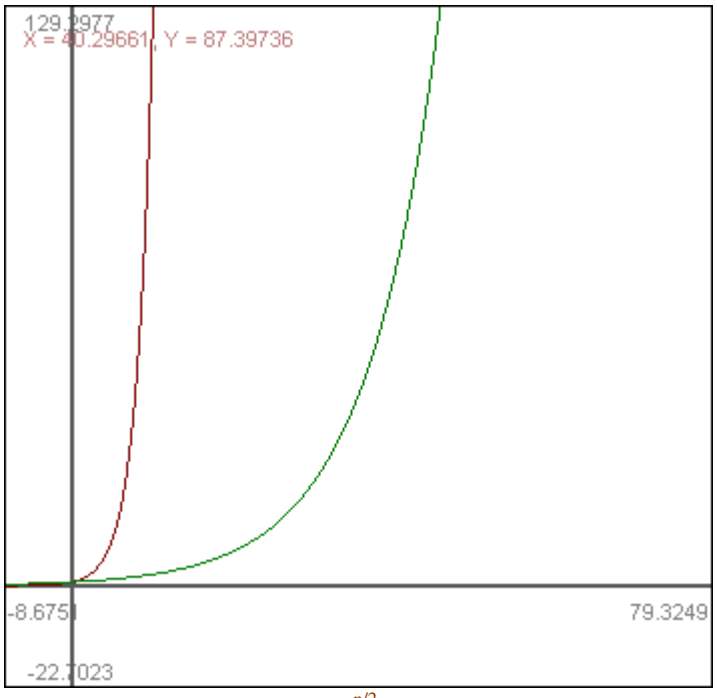
It is now easy to draw the graph indicating the number of instances belonging to ξ.

$$Y = m(\xi, t);$$

$$X = t;$$

$$m(\xi, t+1) \qquad \frac{f_{\xi}(\xi,t)}{\overline{f}(t)} \; m(\xi, t) S_c S_m$$

$$Y = \left[(1.15)(0.98)(0.99) \right]^{x-1}$$



$e^{n/2}$

$[(1.15)(0.98)(0.99)]^{x-1}$

From the graph it is possible to compare the growth of the schema and the solution to the multi armed bandit.

4.9- The genetic traveling salesman problem, an example

The traveling salesman problem, abbreviated TSP, is a classic NP-hard combinatorial problem that is an abstraction of a real life challenge. A man working as a traveling salesman must tour through n cities, sell some products and then return home. Obviously the salesman

wants to travel the shortest distance; which implies that the goal is to find the route that has the minimum distance, passes through all cities that, ends up in the starting city and doesn't passes through the same city more than once. We can basically represent the cities as numbers from 0 to n and the trip as a vector containing the sequence in which the cities are visited:

1. Milan
2. Berlin
3. Vienna
4. Zurich
5. Paris
6. Amsterdam
7. Madrid
8. London
9. Prague
10. Rostock

Trip: <0, 6, 5, 8, 4, 7, 2, 1, 9, 3>

Where obviously the last trip needed to come back home is 3 to 0. All the feasible trips for the poor salesman correspond to the n! possible permutations of n variables, which is a huge number for an already small number of cities. Excluded the possibility of using enumerative algorithms for obvious reasons of runtime we will approach this problem with a genetic algorithm (Appendix A - The genetic traveling salesman problem). Once the values of the problem are established:

```
006     static int PopulationSize = 10;
007     static int NumberOfCities = 10;
008     static int Generations = 1000;
```

It is easy to calculate that all the possible trips are 3.628.800 that compared with the 1000 steps (one individual per step is substituted) are a huge number.

The basic structure of the algorithm is given by the main function:

```
019     public static void main(String[] args) {
020         initDistances();
021         initPopulation();
022         NormalizedMaxFlipFitness();
```

That first randomly creates the table of distances, with the assumption that the maximum distance among two cities is 100 km and that the distance covered from x to y and the one from y to x are the same.

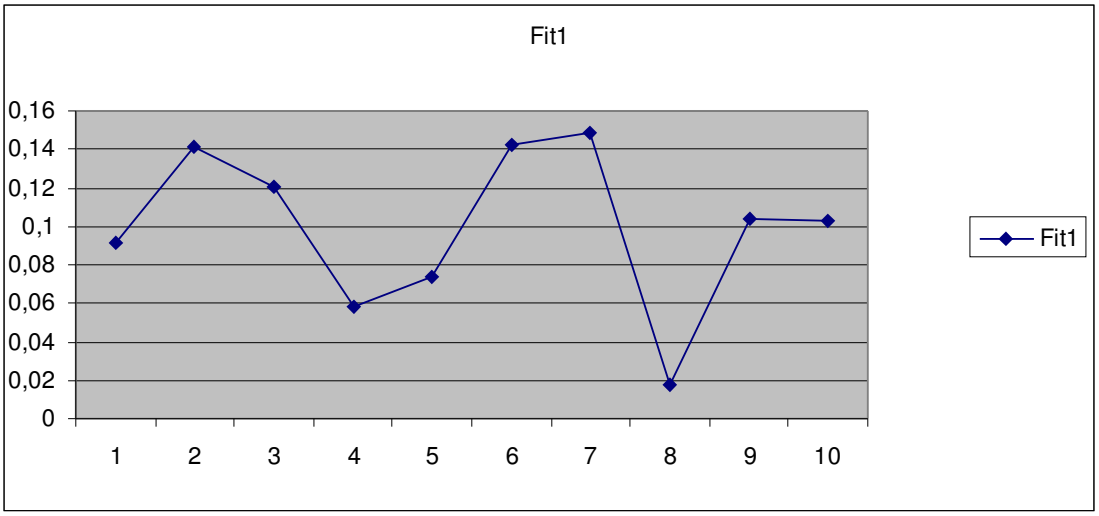
```
221                     DistancesArray[i][j] =  1 + (int)(generator.nextDouble()*100);
222                     DistancesArray[j][i] =  DistancesArray[i][j];
```

The main also initializes the population of solutions by randomly creating 10 permutations in the range 0-9, then it uses the `NormalizedMaxFlipFitness()` function to calculate the fitness of each chromosome. It is necessary to enter a little bit more in detail about how the fitness is calculated. Calculating a fitness value directly proportional with the covered distance is not particularly difficult:

$$Fit_l = \frac{\mu_E(A_h(t))}{\sum_{h'=1}^{M} \mu_E(A_{h'}(t))}$$

The problem that the Tsp poses when calculating fitness is that the higher the mileage is the lower the fitness should be, that is exactly the contrary of Fit_l. But the inverse of Fit_l is not unique, therefore two different routines out of the many possible have been implemented, and, although only `NormalizedMaxFlipFitness()` is used, it is easy to substitute it for `OneOverXFitness()`. The best way to explain the functioning of both fitness calculations is graphically. As already seen our starting point is Fit_l that assigns a fitness value to each individual equal to the fraction of the total kilometers it represents.

	Value	Chromo Percentage
	0,614218384	0,091813205
	0,946370304	0,141463188
	0,804219642	0,120214543
	0,391125169	0,058465289
	0,492689548	0,073647106
	0,952634002	0,142399484
	0,990746306	0,148096501
	0,115194547	0,017219251
	0,692666925	0,103539672
	0,690004958	0,103141762
Total	6,689869786	1

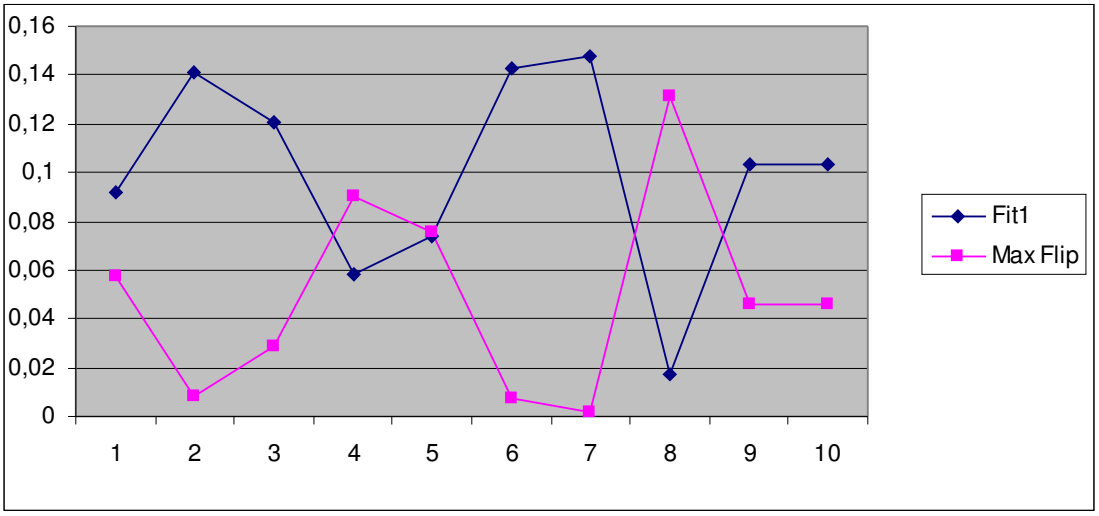


The `NormalizedMaxFlipFitness()` calculates the fitness of each chromosome by taking the highest value of Fit_i and subtracting to it the fitness of the current chromosome multiplied by a scaling factor in the order of 0.99.

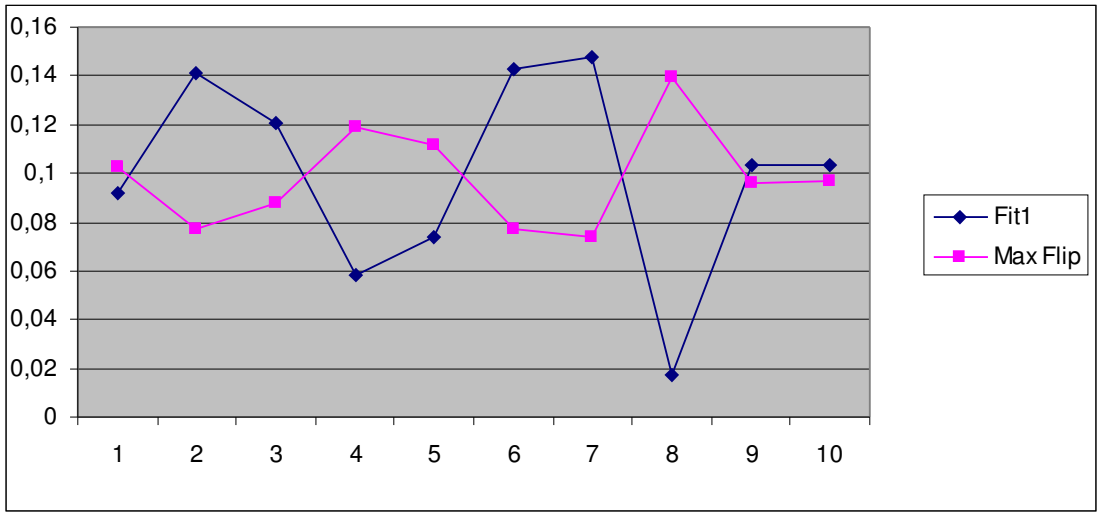
```
311 ChromoFitness[i] = max(ChromoPercentage) -(ChromoPercentage[i]*scalingFactor);
```

Then it normalizes all the values:

```
316             ChromoFitness[i] = (double)ChromoFitness[i] /(double)totalFitness;
```



The point of all this is to flip the fitness curve so that the value of the worst chromosomes lays a little above 0; as a matter of fact, the `scalingFactor` indicates in which measure the proportions among the original chromosomes' fitness are maintained. The lower the factor is the more squashed the curve becomes and the less fitness variance is showed. The previous illustration used a scaling factor of 0.99, that means that the proportions of the original Fit_1 remain almost unaffected, while the following graph shows the effect of setting the scaling factor to 0.5

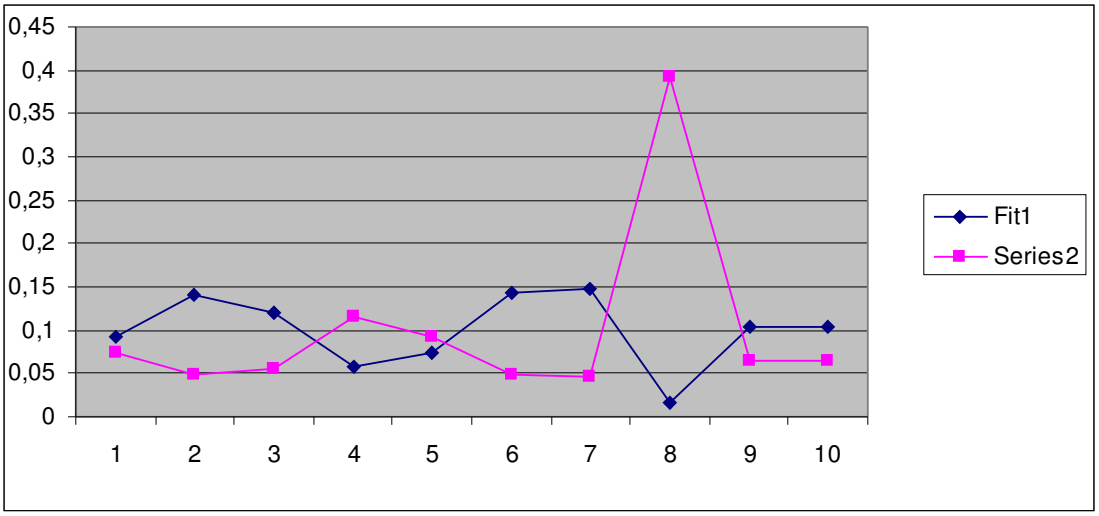


Values of `scalingFactor` that are equal or bigger than 1 return fitness values equal or lower than zero which are simply not acceptable.

On the other side the `OneOverXFitness()` method provides a much higher selection pressure by assigning to each chromosome a fitness equal to $1/Fit_1$ and then normalizing all the values.

```
326             ChromoFitness[i] = 1/ChromoPercentage[i];
331             ChromoFitness[i] = (double)ChromoFitness[i] /(double)totalFitness;
```

With the resulting graph clearly showing that chromosome's quality is generously rewarded in the fitness function and that there is a subsequent tendency to have a super-individual in the population.



Once the initialization part of main has been completed the program enters the cycle that will be repeated for `Generations` times

```
026         do {
027             int i = selectElement();
028             int j = selectElement();
029 
030             int[] ChildChromo = crossover(ChromoArray[i],ChromoArray[j]);
031             mutation(ChildChromo);
```

The cycle is very simple, first two elements are selected using the probability distribution previously calculated by the current fitness determination method. To do this the interval 0-1 is divided in contiguous subintervals of length equal to the chromosome fitness `ChromoFitness[i]` and finally a random number on this interval is extracted. The selected variable will be the one in whose interval the random number falls.

```
295         double rnd = generator.nextDouble();

299         for (int i = 0;i < PopulationSize ;i++ ){
300             CurrentProbability = CurrentProbability + ChromoFitness[i];
301             if (CurrentProbability >= rnd) return i;
302         }
```

Chromo	Fitness	From	To
0	0,116508157	0	0,116508
1	0,016392095	0,116508157	0,1329
2	0,059238647	0,132900252	0,192139
3	0,183752128	0,192138899	0,375891
4	0,15313895	0,375891027	0,52903
5	0,014504113	0,529029977	0,543534

6	0,003016437	0,543534091	0,546551
7	0,266922164	0,546550528	0,813473
8	0,092862474	0,813472691	0,906335
9	0,093664835	0,906335	1

Rnd = 0,19 -> 2 is selected; Rnd= 0,47 -> 4 is selected…

After being chosen the two chromosomes are given to the crossover operator. In this case the edge recombination crossover (ERX) has been implemented, which is, among the various possible crossover operators, the one that tends to preserve the highest number of connections among the cities. In fact the codification that we used for the TSP is not a standard chromosome coding in that the fitness of the chromosome doesn't depends on the value of each allele and its position, instead the fitness is calculated by considering pairs of two contiguous chromosomes. Let's make an example:

The fitness of A1: <1, 2, 3,….> is determined by the distances from city 1 to city 2 and from city 2 to city 3.

The fitness of A2: <1, 4, 3,….> is determined by the distances from city 1 to city 4 and from city 4 to city 3.

The fitness of A3: <3, 2, 1,….> is determined by the distances from city 3 to city 2 and from city 2 to city 1.

That basically means that although A1 and A2 differ only by a chromosome the interpretation of the first three values brings a completely different fitness value, while A1 and A3, that in a normal GA codification would be absolutely different, are simply equivalent. To preserve the major number of "edges" (couples of values) the ERX crossover acts as follows:

1. For each node (city) create a list (`boolean connectedTo[Start Node][End Node]`) of all the nodes it is connected to
2. Randomly choose as first node of the child chromosome one the starting parents nodes, make it the current node
3. Delete the all the mentions to the selected node in the list of connections
4. Calculate which node of those connected to the current one has the lowest number of connections
5. Among the nodes inspected in the previous step select as the next node (the) one with the lowest number of connections and insert it in the child chromosome, make it the current node
6. If the current node has no connections left then randomly select one of the nodes that haven't been inserted yet

7. Unless the chromosome is ready return to point 3

It is easy to appreciate the effects of ERX by examining a couple of crossover examples returned by the program:

```
1) 6 7 0 5 9 8 3 1 4 2   Sum   = 573
3) 5 3 0 4 7 2 6 1 9 8   Sum   = 337
1 + 3 -> 6 2 7 4 1 9 8 3 0 5
```

It is clear that the color couples found in the child chromosome originate from the parents so:

(6 2) is to be found in 1)
(2 7) in 3)
(7 4) in 3) and so on….

In this particular case all the edges are inherited from the parents and point 6. of the algorithm is never used. Now a couple more examples:

```
3) 1 5 4 6 2 3 8 7 9 0   Sum   = 376
4) 8 3 7 6 2 0 4 9 5 1   Sum   = 418
3 + 4 -> 1 5 4 0 9 7 8 3 2 6

5) 5 8 0 6 3 2 4 7 1 9   Sum   = 313
6) 9 7 2 5 0 4 1 6 3 8   Sum   = 261
5 + 6 -> 5 2 3 6 0 8 9 1 7 4
```

After the child chromosome is returned it is mutated with probability `ChromoMutationProb` (=1%) by simply exchanging two of its values:

```
089         boolean mutate =  (ChromoMutationProb >= generator.nextDouble());
090         if (mutate) {
...
093             int tmpValue = childChromo[Allele1];
094             childChromo[Allele1] = childChromo[Allele2];
095             childChromo[Allele2] = tmpValue;
```

This actually changes up to 4 edges and is therefore quite destructive.

Finally a random chromosome is selected for substitution using the uniform probability distribution suggested by Holland.

```
035             int Substituted =  (int)(generator.nextDouble()* PopulationSize);
...
```

```
046             ChromoArray[Substituted] = ChildChromo;
```

And then various recalculations take place to adapt fitness and statistics to the new inserted chromosome.

It is often interesting to note the effects of changing the three basic parameters of the problem:

- PopulationSize
- NumberOfCities
- Generations
- ChromoMutationProb

Discovering how different values produce different results, e.g. increasing the population size augments the sampling of the search space (random search) and increases the quality of the best solution found in the first generation, reducing the distance among starting and final best. However the drawback is that the expected number of crossovers that an individual undergoes diminishes as there will be "Generations" crossovers for a bigger population. The solution is therefore to increment Generations too maintaining the ratio Generations/ PopulationSize constant, this basically produces a more refined search. Since the GeneticTSP doesn't maintains the current best forever it is possible to see a deterioration in the current best solution. Since the substitution does not takes into consideration the chromosome fitness it is not possible to completely eliminate this effect, in fact there the best chromosome has always a chance of 1/n to be substituted but it is possible to attenuate it by increasing the population size or diminishing the mutation rate. As a matter of fact mutation is another factor disrupting the schema of the best chromosome that might contribute to the deterioration of the best solution by augmenting the fluctuations of the fitness values. In the end it is clear that being the number of possible elements in the search space a function of n! a slight change in the number of cities hugely affects the dimension of the search. However the best way for the reader to find out how a parameter value or a combination of them shapes the final result is to experiment with the program until the basic functioning of the GeneticTSP is understood.

Chapter 5

Evolution and Neural Networks

5.1- Introduction

The chance of applying evolutionary mechanisms to computer science has led to a growing interest in optimization algorithms based on the same principles. Within neural networks applications, in particular, evolution-based algorithms have shown to be suitable for solving several problems ranging from weight training to topology determination passing through determination of the learning rule. From now on the term evolutionary algorithms (EAs) will be used to refer to the whole branch of stochastic search algorithms such as GAs and ESs. EAs present some different points of strength in comparison to classic optimization problems that use gradient information; in particular they don't tend to get stuck in local optima. Most experiments demonstrate that EAs are quite good at finding regions that exhibit good performances in big and complex search spaces, even in presence of noisy quality functions. Nonetheless It is important to understand that EAs are not the final solution for each and every optimization challenge posed by a NN, in fact the best solution is always linked with the nature of the problem that must be solved, and consequently the use of heuristics that exploit knowledge of the particular problem is often crucial in order to obtain quality results.

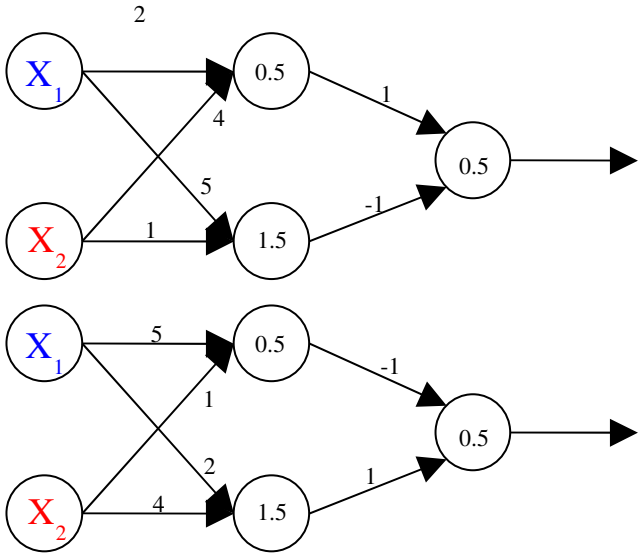
The range of possible applications for EAs is quite wide; EAs could basically be applied to any parameter of a NN that has to be optimized. It is generally possible to distinguish two different approaches; the first contemplates the use of an EA where a classic algorithm was generally used, while the second approach tends to combine the classic algorithm, e.g. backpropagation (BP), with an EA so as to compensate the weaknesses of both methods.

5.2- Weight determination in NNs

The BP method has already been introduced in chapter 1 and it represents one of the most used techniques for weight training in a feed-forward neural network. However it does have its own drawbacks, BP is in effect known for its tendency to get stuck in local optima

and, in case of large problems, it is not very fast [17][19]. As a matter of fact BP loses efficiency whenever the gradient information is hard to obtain, not to talk when it is faced with a non differentiable or multimodal function. We could see BP as a "local" optimization method that is heavily dependant on the initial state of the network weights, as a matter of fact BP will reach only those optima that are in the vicinity of the starting point. EAs are, the other way round, "global" [17][23] search methods that examine broad parts of the search space for higher fitness averages. One of the main obstacles in the race for the best solution is that the correct set of parameters for a BP algorithm is problem dependant [17][18][24] and it deeply determines the likelihood of successful learning. E.g. the step size determination poses a problem of compromise between convergence speed and convergence monotony; a BP using a small step will require more time in order to converge while a big step might jump over the optimum resulting in a possibly worse solution and the consequent inversion of "direction", which would lead to an oscillatory behavior. An additional advantage that EAs have over BP is the broad applicability to any kind of network. While classic weight training methods often impose some limitations about the topology and the transfer functions, EAs can actually be used to train every sort of network, feedforward, recurrent, fuzzy and even higher order[18][23].

The already mentioned advantages that EAs have over BP might be exploited by combining the two methods so that EAs would supply for some of the BP deficiencies; this is especially advantageous when treating discontinuous, non differentiable error functions within a large search space. This is in effect the case of most real world problems. However EAs are not trouble free either. Among those problems in which EAs often incur there is the so called convention/permutation problem [17] [18], which originates by the non unique relation between NN and decoded function. That means that given a NN topology and a vector containing its weights there might be more than one vector which results in equivalent NNs.



E.g. the two previously illustrated networks perform the same task but have two different chromosomes, respectively:

<2,5,4,1,1,-1> and <5,2,1,4,-1,1>

This renders the mapping of networks into functions multimodal and heavily affects the crossover operator efficacy. In fact crossing over two networks (like those illustrated above) that show a similar functionality and a good RMSE, often produces an inferior quality offspring. Although there are some exceptions where it is possible to perform successful crossover, like nearest neighbor multilayer perceptrons and RBF's [18], most NN don't benefit from it. The cause of this can to be found in the encoding method, networks that perform similar functions and as a consequence have a similar error, are not codified in chromosomes that are similar or next to each other. This kind of codification is not intentional and unfortunately the creation of a functional "network to chromosome" relationship, which would encode all equivalent networks into the same chromosome, is until now beyond reach; thus the common approach to avoid some of the permutation problem consequences is to rely exclusively on mutation leaving crossover in disuse[17] [18][23].

Like in any other problem that is solved through EAs, the representation of the chromosomes plays a fundamental role. The simplest way of coding the weight information related to a NN is obviously in binary format, as this has the advantage of rendering crossover and mutation implementations quite simple. On the other hand a binary representation presents some difficulties too, first the choice of how many bits will describe a weight. The more bits are used the more precise the weight approximation can be, however if each value is represented by too few bits the net will not properly adjust to the problem due to a lack of fineness and if too many bits are employed this will adversely bloat the chromosomes resulting in a performance degradation of the algorithm. Another problem connected with the standard binary codification in general is that numbers very far from each other are coded with a very short hamming distance. Let's explain this a little bit more in detail. The hamming distance is the number of ciphers that are different between two strings:

100 and 000 have a hamming distance of 1 because only the first cipher differ.
000 000 000 and 100 000 001 have a hamming distance of two since their first and last ciphers are different.

If we consider a binary coding using only 8 bits we can create two strings with hamming distance 1 (the smallest):

0000 0001 = 1 and

1000 0001 = 129

By looking at both values it is immediately clear that even though their representations are very similar they correspond to numbers that lay very far from each other, and the more bits a binary string has the bigger the distance among two strings with hamming distance 1 can be. It follows that mutation can be very disruptive for such representations. In order to solve this problem binary representations often use alternative binary numeral systems such as the Gray code. In the Gray code two successive numbers are coded in two binary strings whose Hamming distance is always 1. E.g. all 3-bit binary strings coded in Gray code are:

000 = 0
001 = 1
011 = 2
010 = 3
110 = 4
111 = 5
101 = 6
100 = 7

This codification system renders mutation evidently less disruptive and favors EAs.

A real valued coding of the chromosomes generally poses as the alternative to binary representation. Usually that means that for every weight in the network there is a real value in the chromosome. Unfortunately this approach suffers from the permutation problem too, prompting again a shift towards mutation as main operator, and it is here that the use of ESs in conjunction with the aforementioned coding turns out to be quite versatile. As seen before, ESs were born to be applied on real values and used to have mutation as main operator, indeed their use of normally distributed mutation with adaptive step renders them the algorithm of choice when handling chromosomes made of real.

Such a use of an EA was made for the first time by D.J. Montana and L. Davis [19] who used an elitist (best solution always survives) GA to train a network that had to categorize the noisy data of an underwater sonar into two classes: “interesting” and “not interesting”. The used network was feedforward and fully connected, having four input units, two hidden layers and one output for a total of 126 weights to be trained. The chosen chromosome encoding was a vector of real values; whilst the sum of the square errors was used as the inverse fitness function and the initial weights were randomly chosen using the probability distribution $e^{-|x|}$, which was the result of problem specific observations. The population was set to 50 and the generations to 200. A large number of different operators were used to find out which would disclose the best results and in the end particular versions of the mutation and crossover were used; these customized operators worked in a slightly modified fashion that helped to maintain the relationships between a certain weight and the

weights of all its ingoing links (again a problem specific information being used). The interesting result that Montana and Davis obtained was that the GA performed better than BP. This was obtained letting the GA run for 200 generations (that is 10.000 network evaluations) and the BP for 5000 iterations in consideration of the fact that BP executes two passes for each weight adjustment (feedforward and then back propagate), taking the sole backpropagation step more time than two GA evaluations.

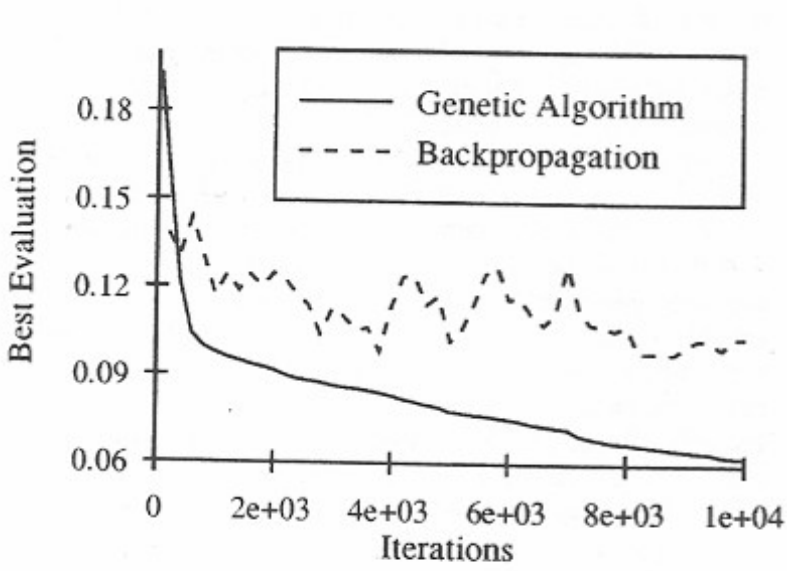


Figure 8: Results of Experiment 5

Even if such results might seem encouraging it must be said that as Schaffer, Whitley, and Eshelman [17] noted that the comparison made by Montana and Davis referred to the simple version of BP. As a matter of fact there also exists a modified version of BP called quickprop (1990) which uses the information obtained from the second order derivatives of the error function to speed up convergence. As a consequence Schaffer and co. concluded, albeit without experimental proof, that GA could not compete with quickprop and that they should be used instead of BP only whenever gradient information is hard or not obtainable at all. This might be the case where neurons use complicated transfer functions different from the classic sigmoid or when training fully recurrent networks.

Also Porto et al. [23] have used EAs to train the weights of a NN and compared the performance of EAs, BP and simulated annealing. Coincidentally also Porto et al. wanted to categorize the data of a sonar, trying to distinguish those waveforms created by a human made spherical object from those originated from natural rocks and ocean floor. The used network was the classic three layered perceptron with a single output neuron and two hidden layers containing 4 neurons each, the training set was composed of 238 patterns while some extra 237 patterns were used as test. It must be said that the EA used did not take crossover in consideration; as a matter of fact it was quite similar to Rechenberg's ESs in that it only a

Gaussian mutation proportional to the square error of the chromosome was used. Selection was "tournament style", which means that for each vector x another vector y is randomly selected, the two chromosomes are then compared and the best receives a "tournament victory", this routine is repeated n times for each chromosome and, at the end of all tournaments, those chromosomes with most victories are selected. The initial population of the EA was composed of 50 individuals with standard normal distribution; the total generations were 500 whilst BP and SA accomplished 5000 iterations each. During the experiments it emerged that, although slower, simulated annealing was overall the best method. When taking into consideration the best case scenario, BP and SA obtained the same results in the number correctly recognized patterns (~95%) while the EA lagged some 3-4% behind. In the worst case scenario things change dramatically, with the worst case of pattern recognition for BP very far behind (~60%) those of the equivalently good EA and SA. This analysis clearly marks a preference for stochastic methods. Porto concludes saying that for a parallel processing computer the computational costs of EA and SA are equivalent, but he also indicates that EAs might have a better performance because of their use of multiple solutions.

Sexton et al. [24] have conducted an extensive and accurate comparison of the NN training capabilities of BP and GAs. They did not only compare one kind of algorithm with the other but also experimented with the various learning parameters, computer architectures, error functions and software implementations in order to find the best version of both algorithm types. At first the different trainings of NN were tested using seven different mathematical functions, and then the problem of classification using NNs was examined. The conclusions were that:

- BP has a wider variability in the quality of its results
- BP produces worse results even if running for longer time
- GAs approximate better the function in the training region
- GAs can easily avoid over training the network

There have been several attempts to mix classic techniques with EAs [18], sometimes applying one to the results of the other, sometimes using them collaboratively; results are mixed, depending on:

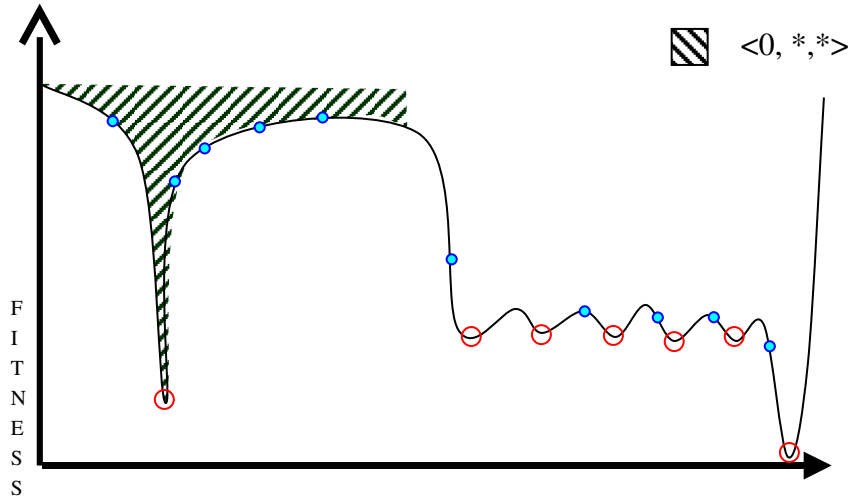
- the particular implementation of the algorithms
- the specific problem
- the starting conditions

So the question is, whether it makes sense to integrate EAs with classic training algorithms or not. In order to obtain an answer to such question we will have a look at a very famous theorem. The "No free lunch" theorem of Wolpert and MacReady proves that the average

performance of two different algorithms across all the possible problems is identical. This basically means that no algorithm outperforms another algorithm, not even random search or enumerative techniques, on each and every possible problem [20]. Or, seeing it from another point of view, the average quality of every search algorithm in the space of all problems is constant. So, the "No free lunch" theorem is the demonstration of what experience already suggested to many, as it states that without knowledge of the problem and its quality function the expected efficiency of the chosen algorithm is no higher than a random search, in other words: the best solution is always problem dependant. The fact that sometimes one search algorithm performs better than another depends on the fact that each algorithm has a part in the space of all possible problems where it is particularly efficient, consequently given a determined problem it is important to find out which algorithm is the best for that problem-space. Considering the conclusions of the theorem it makes sense to mix different algorithms so as to obtain a hybrid training algorithm that yields better results. The basic idea is to sequentially execute one algorithm after the other in such a way to maximize their combined effect. For instance, since EAs are good at finding regions of space with higher fitness it would be wise to start the training using an EA and then pass the obtained results to a classic algorithm such as BP, compensating in this way the fine tuning deficiency presented by the EA and the local minimum problem of BP. Although possible, inverting the order of application probably wouldn't deliver the same results; in fact the idea is to let the EA explore the search space until the current optimum is in the attraction basin of a good (sub) optimum and starting with BP only to later use an EA wouldn't achieve this. In order to determine where the EA should stop and where the BP should start it could be possible, for example, to set a lower bound for the chromosome diversity. Once the elements of a population have converged, they present common features belonging to a particular schema, therefore it could be possible to stop the EA once the percentage difference of all chromosomes is lower than the one set in the threshold and continue the weight optimization with BP. In doing this weights would converge to the optimum that lays at the bottom of the attraction basin much faster and finely. Observing this technique from another point of view we could basically say that instead of randomizing the initial weights of the network, from which the BP result is heavily dependent, we use an EA determine them and then the net is simply trained with the classic BP.

The work of Seong-Whan Lee [22] can be taken as an example of hybrid training for the problem of unconstrained numeral recognition. Lee proposed a three layered network, where the input and the hidden layer were composed of 5 clusters, being a cluster a 4x4 grid of neural units. There were a total number of 170 units, including the output layer, using altogether 2080 parameters. The network was trained using circa 4000 numerals and then tested with 2000, this happened on three different databases. Although Lee provides the precise settings of the GA (mutation: 0.007% crossover: 0.0007% population: 20) he doesn't enter in detail when explaining the use of the GA. However the relation among BP and GAs is clearly indicated, each individual of the initial population is first trained with BP and then

the typical GA loop is entered. At each cycle selection takes place forming the new population, then a BP training is done on all its elements and their evaluation follows. No indication about the exact kind of mutation or crossover is given nor about the cycle termination criterion. Lee claimed not only to have achieved a network topology capable of learning faster and overall better than a fully connected NN, but also demonstrated that the mix of GAs and BP is slightly better than training with only BP. As a matter of fact the combined use of GAs and BP produced, for all three databases, results that were something between 0.2-0.6 % better than simple BP. Surely the performance improvement using GAs is not astonishing but this might depend on the kind of algorithm implemented and without doubt the use of a population with only 20 individuals over 2080 parameters might represent a bottleneck. It is also worth noting the extremely low crossover rate used by Lee, which makes the algorithm more similar to an ES rather than to a fully accomplished GA. Another important point in the combination between BP and EAs made by Lee is that BP calculations are not made after the ES has stopped, instead they are sequentially executed at each cycle, the author thinks that the problem in this is that if a region of space has a very bad average fitness but a deep local optimum (a "foxhole") all the samplings will slide to the local optimum through the use of BP at each cycle, this means that the calculated fitness of a schema is not anymore the average fitness of its instances but rather the fitness of its best "BP reachable instance". Let's see it graphically:



The red circles indicate the basin lower points, also called the attractors; starting from a blue spot, BP generally converges to the next basin attractor. In the image above both schemata <0,*,*> and <1,*,*> have received 5 samplings (5 blue spots); all five samplings of the "0" schema clearly converge to the unique attractor of the first schema, while almost each of the "1" schema samplings has converged to a different attractor. Using BP (like Lee) over each EA sampling, would confer the "0" schema a higher fitness even if its average and its

local optimum are much worse than those of the "1" schema. Although it would be interesting to compare both methods, first ES and then BP vs. ES and BP at each cycle, it is easy to see that Lee stripped the ES of its global searching capabilities, which is somehow questionable if we want to compensate the disadvantages of BP with ES and vice-versa.

After the review of several examples in the literature Yao [18] concludes that there is no clear winner among BP and ESs (as the No free lunch theorem states), and that the discrepancies in the results presented in the literature, showing sometimes good sometimes bad outcomes are not only determined by the many implementations and parameters, but also by the nature of the problem that has to be solved. The best methodology to ensure better results is therefore to use combinations of both.

5.3- Network topology determination

It would be nice if every problem a NN faces could be solved with a simple and repetitive structure, but unfortunately this is nothing but a dream. During the initial surge of interest in multilayer perceptrons they were often been regarded as networks capable of approximating an arbitrary function; indeed it is possible to adapt the approximation capabilities these networks by augmenting or diminishing the number of neurons in the hidden layer. But unfortunately not everything is not as easy as it might initially seem; on the contrary, it is quite hard to determine the proper number of neurons, often resulting in an over trained network or one that inadequately approximates the desired function. Already in the sixties, the discovery of Marvin Minsky and Seymour Papert about the inability of a single layer perceptron to approximate the XOR function showed that a unique network topology capable of solving all kinds of problems was a mirage. However, if it is true that there is no single network topology capable of solving all problems, this doesn't necessary imply that taking each single problem it is impossible to find a topology capable of solving it. An empirical argument that shows how EAs can be successfully used to find a problem-specific network topology has been presented by Roberts and Turega [26], who demonstrated how a standard single layered perceptron with variable number of nodes always proved worse than the architectures found with the different EAs. This demonstrates that the quest for the best network topology is still open.

Finding the optimal network topology for a certain problem means to determine the number of nodes as well as the connections that compose a network capable of solving the given problem with a reasonably low error (RMSE) in an acceptable time. The idea would be to automate the NN creation so that little or no human effort at all would be needed, which is exactly the contrary of what nowadays happens.
NN are generally used to solve a determined kind of problem, which means that although the exact values of the problem variables (inputs) are initially unknown there is enough

information about the problem to determine the number and type of variables which will represent the input of the neural network. E.g. if we want to create a network that computes the sum of two numbers we do know that there are two different input variables (the numbers) and a single output variable (the total) but we don't know the exact values that such variables will have. In this case the specific question is how to solve a fixed problem given unknown variables. GAs help taking a step further in the abstraction scale, trying to answer the question of how to solve an unknown problem, how to create a network for a problem which is initially unknown. Nonetheless this last assertion has to be reinterpreted in the light of the No free lunch theorem. It is incorrect to seek an algorithm capable of finding the best solution for every single problem as there is none. The focus point is the process of problem specific information gain and its interpretation in order to find a solving algorithm, which is what the average NN designer actually does. It is not the universal solution method rather the methodology to find the appropriate method.

But let's come back to the problem of finding a suitable NN topology, this is not a particularly simple task; in fact the search space has the following characteristics:

- It is formed by all the possible networks and is therefore infinite.
- It is not contiguous since the minimum distance among two networks is one neuron or a connection, a discrete step.
- It is multimodal because of the conventions problem (different architectures can solve the same problem).
- It has a weak causality (topologies that are neighbors in the search space can have very different quality).

With the only exception of the weak causality an EA should be well suited to solve such a problem. Weak causality (Rechenberg [7]) can be resumed in "small changes in the causes have big effect changes". Already Rechenberg noted that in a totally chaotic environment a strategy is absolutely useless. Now the question is: when is an environment chaotic? Do chaotic environments exist at all? The connections among chaos and genetic algorithms are quite interesting and would require a very long dissertation, for those interested Schöneburg [3] might be a very inspiring starting point.
Coming back to chaos, there are two forms chaos, deterministic and not deterministic, deterministic chaos means that the environment behaves chaotically because we are unable to see connection among cause and effect, which is also what Rechenberg called a very weak causality. This is the kind of chaos we are faced within a computer simulation, for the true randomness (by definition absolutely unpredictable) required by non deterministic chaos cannot be obtained by the computer, instead it can only be deterministically emulated. So in this case we can trace back the definition of weak causality to the one of deterministic chaos, definition that actually coincides with the lack of knowledge about the problem that is being

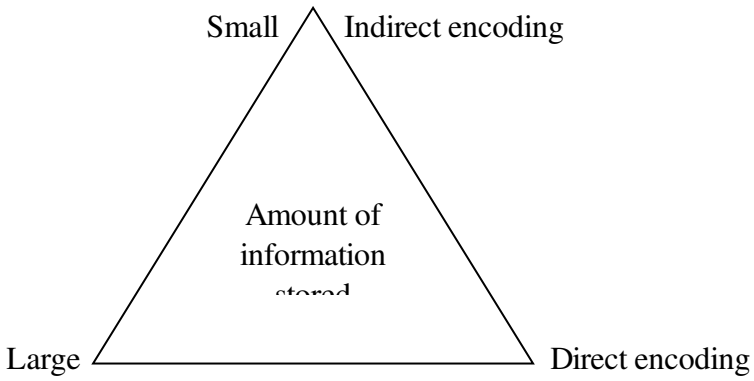
analyzed cited by the "No free lunch theorem". Under this light also Rechenberg was aware of the consequences of the "No free lunch theorem" even though he did not explicitly formulate it.

As for the fact that the NN topology search space is multimodal a mention goes to Thierens [29] [28] one of the few who tried to render the topology encoding non-redundant. With redundancy Thierens indicates the possibility of constructing two different networks that have the same functionality, his efforts were therefore oriented at eliminating the permutation problem which rendered crossover unusable in NN topology determination EAs. Although his early work resulted promising [29], despite the few experimental results[18] , he later found out [28] that for Bayesian networks the use of a non redundant encoding added no benefit. There are also other critiques to the fact that the permutation problem is a real obstacle for EAs [18] but in the end the truth is that such a question has not yet been properly answered.

Like in every EA related problem, during the evolution of topologies we are faced with the (in evolution omnipresent) question of the right encoding as well. Which information should be encoded? How should it be encoded? The question of the best encoding technique is not a trivial one and has been often analyzed.

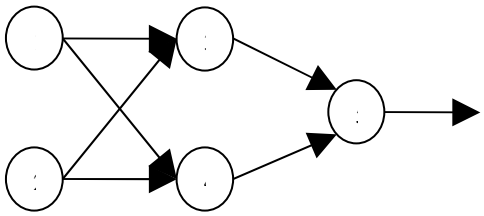
Roberts and Turega [26] demonstrated through different tests on the "parity problem" and the "two or more clumps predicate" that the network encoding not only affects the speed performance of the EA but also the quality of its results. They also showed that the best encoding method varies according to the kind of problem and its size. Altough Roberts and Turega, like many others, tried to categorize all known encodings into three categories (strong, grammar and weak) the horizon of all solutions might comprise very subtle shades. Therefore any categorization that clearly divides all encoding styles into well defined groups must be accepted with a critic eye and it must remain clear that it can only be an approximate classification. Bearing such forewarnings in mind it is possible to make a first rough classification here: encodings are subdivided into direct and indirect encoding. It is like a pyramid where at the bottom lays the bulkiest encoding, the one that stores all the available information: direct encoding while indirect encoding, which stores far less information, lays on the top.

Pyramid of encoding abstraction



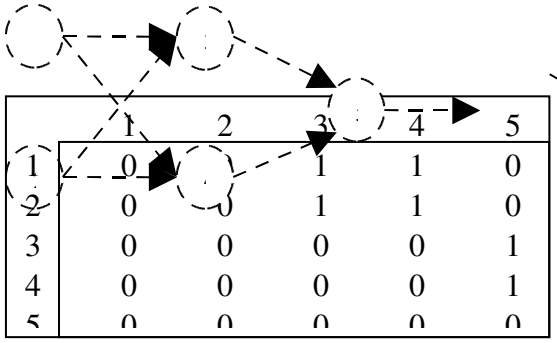
Direct encoding means to store the whole NN into a quite unambiguous way, that is to codify the structure of the network into a chromosome so that once the chromosome is decoded a network almost identical to the starting one is obtained. The most straightforward implementation of direct encoding contemplates the use of the net itself as chromosome. This kind of network encoding is clearly dependant on the implementation of the NN simulator engine and the available parameters for describing each network; in fact the network chromosome will coincide with the way the neural network is stored in memory by the program engine. Another traditional technique which is referred to as direct encoding is the creation of a binary connection matrix [18][25], which is a NxN square matrix where N is the number of neurons and the (x, y) position is true if there is a link going from the neuron x to the neuron y. E.g.:

Original network



Chromosome

Encoding



	1	2	3	4	5
1	0	0	1	1	0
2	0	0	1	1	0
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0

Decoding

Decoded network

This technique is a step over the direct NN-engine-native encoding in the pyramid of abstraction; as a matter of fact it encodes the connections among nodes but doesn't actually provide information about weights and other important parameters of the network. Starting from this kind encoding, it is easy to modify it and make new ones; it could be possible, for example, to replace the binary values with real ones in order to include information about the link weights or to simply prepend to the chromosome the learning rate, the mutation probability and other settings of the network. Among those encodings that use a connection matrix there is a common variation or, more precisely restriction, which is the use of the right upper half of the matrix only. In this way the construction of a feedforward network is enforced. The main advantages of the connection matrix encoding consist in the ease of implementation of the evolutionary operators and its understandability. However, chromosomes that use a matrix direct encoding bloat quickly and consequently tend to scale very badly; in fact as the number of neurons N increases this is traduced in an exponential growth of the chromosome size, which is equal to N^2, something that is not really desirable for large networks.

Those encodings of a neural network that lay on the top of the abstraction pyramid are referred to as indirect encoding, in this case not all the available information about the network is stored into the chromosome, and this causes the mapping of the networks into the chromosome space not to be a one-to-one relation. The amount of information about the NN that goes lost during the encoding must be replaced in the successive decoding process; this is generally done by setting some generic developing rules. E.g. if the chromosome encodes the sequence of layers and their number of nodes only, information about the edges goes lost and is therefore necessary to implement rules for the creation of layer connections. Hence it could

be possible to assume that all layers are fully connected among each other and decode the network accordingly. Since it is often the case that a big part of each network is specified through developing rules, which are common for all chromosomes, indirect codifications have the obvious advantage of being more compact, but also the disadvantage of excluding relevant regions of the NN topology search space. As a matter of fact, in the moment a developing rule is implemented also a restriction on the possible decoded network topologies is made. If, for example, all layers are assumed to be fully connected then all direct connected network architectures are automatically infeasible in that particular encoding and this might exclude the solution for the given problem.

Obviously there are countless possible indirect encodings but it is still possible to categorize them in groups of roughly similar techniques. The parametric_representation assumes that the chromosome stores only certain parameters about the network such as: number of layers, neurons per layer or its learning parameters, these parameters are later interpreted with the help of developing rules. Since few variables are used, parametric encodings are mostly successful if there is an adequate amount of problem specific information to create the developing rules, indeed the topology search space of this approach is considerably reduced. Said in other words, the kind of network we are searching for must be known in order to create some effective rules and, in most cases, only the size and some parameters of such NNs will be evolved.

A second branch of indirect encoding is represented by the developmental rules encoding, this means that the chromosome stores the very rules for the creation of the network and not its structure features. In order to explain this kind of encoding we will look at the famous left-hand grammar chromosome example of Kitano [27][18][16].

Kitano was the first in proposing a chromosome which did not contain information about the NN, but rather a chromosome that would contain information about the rules used to create the network. Each rule had on its left side a non-terminal symbol and a 2x2 matrix of symbols on its right side.

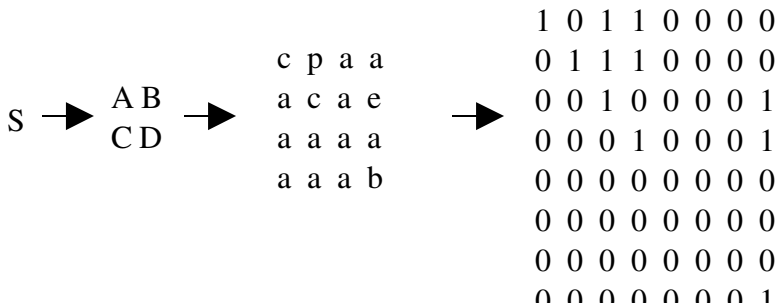
$$S \rightarrow \begin{matrix} A & B \\ C & D \end{matrix}$$

$$A \rightarrow \begin{matrix} c & p \\ a & c \end{matrix} \quad B \rightarrow \begin{matrix} a & a \\ a & e \end{matrix} \quad C \rightarrow \begin{matrix} a & a \\ a & a \end{matrix} \quad D \rightarrow \begin{matrix} a & a \\ a & b \end{matrix}$$

$$a \rightarrow \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix} \quad b \rightarrow \begin{matrix} 0 & 0 \\ 0 & 1 \end{matrix} \quad c \rightarrow \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix} \quad e \rightarrow \begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix} \quad p \rightarrow \begin{matrix} 1 & 1 \\ 1 & 1 \end{matrix}$$

In order to form the network the rules had to be applied in the following manner: starting at the symbol S and following the arrows, the symbol on the left side of a rule was substituted

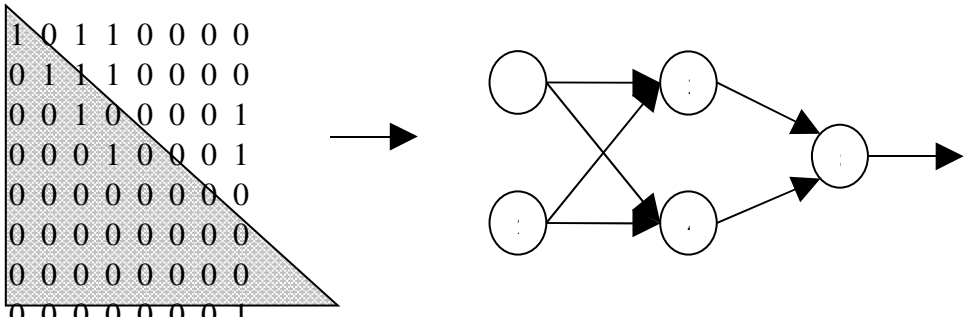
for the symbols on the right hand, continuing until the binary matrix representing the network's connections would be reached.



The rules were then sequentially encoded to create a string containing the whole graph generation grammar and such string would represent the chromosome.

||S|A|B|C|D||A|c|p|a|c||B|a|a|a|e||C|a|a|a|a||D|a|a|a|b ….

Since the networks that Kitano intended to encode were feedforward, only the upper right half of the binary connection matrix was interpreted during the network creation process, in this way a node could only be connected to nodes with a higher index resulting in a feed forward network. Still such an interpretation of the connection matrix also means that more than the half of the used bits is redundant.



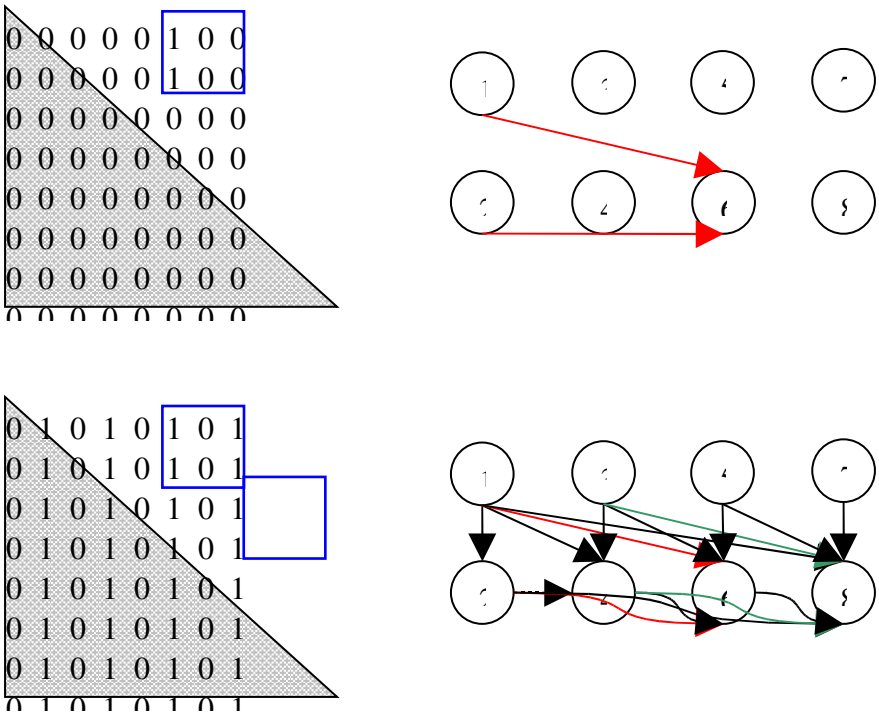
One advantage of Kitano's codification method is that it strongly attenuates the disruption of the chromosome caused by the use of classic genetic operators. As a matter of fact the crossing over of two chromosomes always creates a valid chromosome and in case a rule is specified more than once in the same chromosome (e.g. ||A|c|p|a|c||A|a|a|a|c||), then only its first version is applied so that the chromosome is valid. Mutation is also easily applicable, since, in order to create a valid chromosome, we have only to consider that the changed letter must belong to the alphabet of the current allele (i.e. A-Z or a-p). Given that a small mutation in one of the highest level rules would duplicate a whole branch of the network, resulting in a

wide change of topology, Kitano affirmed that his encoding was particularly suited for the creation of networks that presented a recurring connection structure, a phenomenon caused by the multiple applications of the same rule. It is commonly assumed that the repetition of connection patterns is favorable as it tends to create building blocks (patterns of cells) which are repeated throughout the network. An even more interesting approach since it is also known that the repetition of many similar sub structures is often the key for the creation of powerful NNs.

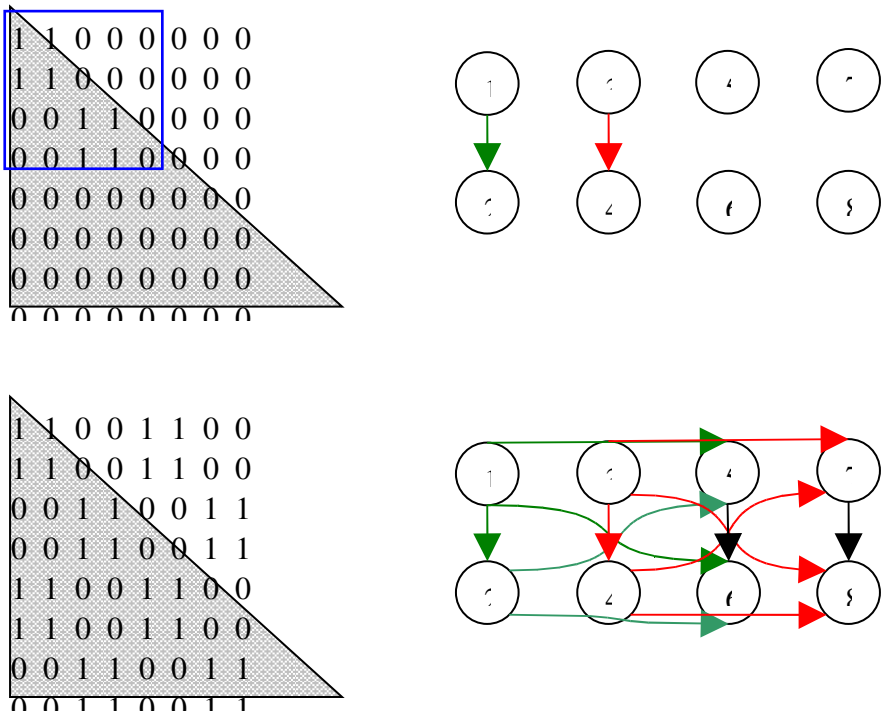
Kitano benchmarked his grammar encoding against direct encoding techniques in order to evolve encoder/decoder networks obtaining favorable results that indicated a higher convergence speed for the grammar encoding and overall better results than direct encoding.

However Siddiqi et al. [30] questioned the possibility for grammar encodings to create such repetitive sub structures in a NN. Siddiqi indicated that the repetition of a pattern at different points the binary connection matrix isn't traduced in the creation of sub-circuits for the NN. As the illustrations below show, the patterns enclosed in the blue squares are repeated allover their respective matrixes. However no sub-structure, in the form of a repeated sub-network, is present.

Example 1:



Example 2:



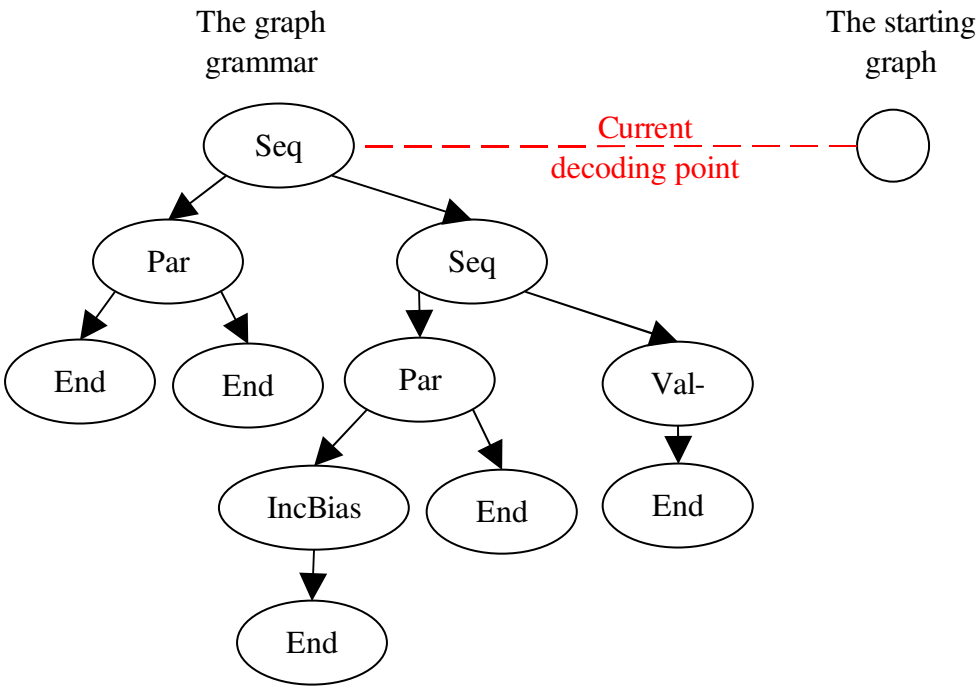
This definitively shows that Kitano was wrong when claiming that grammar encoding would be capable of "…copying the meaningful sub-circuits involved…". Nonetheless the connections follow a distinguishable pattern which connects each node with nodes whose relative "index distance" is constant. E.g. in drawing 2:

	Relative distance	+1	+4	+5
Node	1	2	5	6
	3	4	7	8

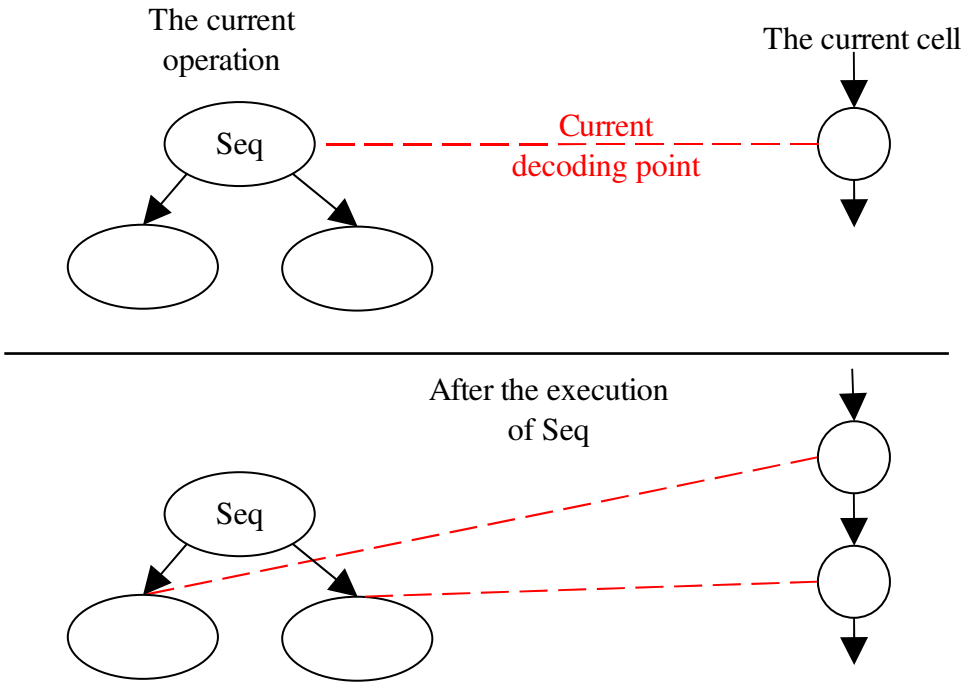
A feature of the grammar encoding that Siddiqi did not mention at all. We could therefore say that the grammar encoding is the best choice for those networks whose functionality is determined by the relative connections among neurons, yet remains the open question whether such networks could have an application. Siddiqi et al. also extended the original set of tests Kitano used to compare grammar encoding with direct encoding. The use of different starting parameters proved that direct encoding produced results practically as good as those

of grammar encoding. Siddiqi consequently formulated the hypothesis that the good results obtained by Kitano in his original experiments were caused by the fact that the grammar encoding might be particularly well suited for the encoder/decoder problem and the fact that the initial parameters of the direct encoding created a starting population whose nets contained too few connections.
Another critique the author moves to Kitano's first work was the use of the so called "adaptive mutation"; it must be said that the name's choice was quite unlucky given that the mutation amount was proportional to the hamming distance of the parents, the less differences the bigger mutation. This is actually no adaptive mutation at all as it tends to maintain a fixed rate of chromosome diversity and evolves in no way together with the network fitness. In fact the population would never converge as every time the chromosomes would be similar enough they would be mutated with a higher frequency, which makes the EA a random search uniformly distributed in the search space. It would have been much better to implement a technique similar to Rechenberg's adaptive mutation where the mutation rate is indirectly linked to the fitness of the chromosome.
Even if it seems that the comparison methods and problems used by Kitano are somehow questionable [16][30] Kitano's technique remains in the literature one of the most inspiring and interesting, worth exploring and modifying.

Taking inspiration from Kitano's idea of an encoding a grammar, Gruau [31] tried a more ambitious method of grammar encoding called cell encoding. This time the plan was to encode a graph grammar in the chromosome. Now, a graph grammar is a grammar that creates a graph, and since a neural network is a graph it is possible to encode a NN topology by encoding its creating graph grammar. Gruau represented the graph grammar in a tree where each node is an operation (terminal or non-terminal) that modifies the current graph, respecting the clause that terminal operations correspond to the leaves of the graph grammar tree. The initial graph is composed by a single cell. Please note that a cell does not correspond to a neuron, it is a kind of placeholder for a sub-graph which might contain a single neuron as well as many. The final NN is achieved by modifying the starting cell following the operations that are stored in the nodes of the graph grammar tree.

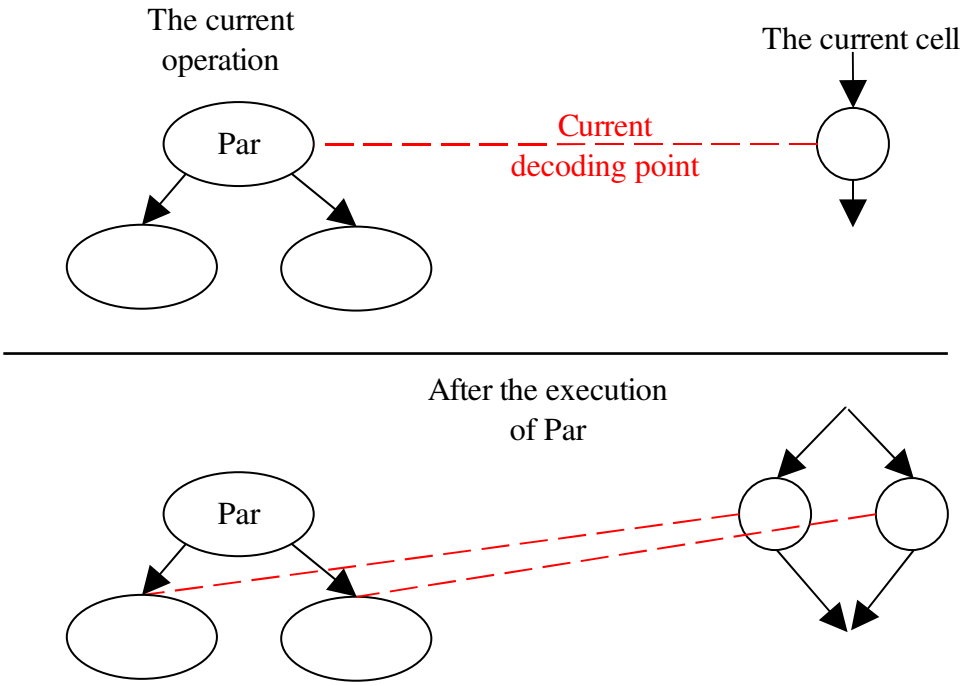


The previous illustration shows a typical starting point for the chromosome decoding, the graph is composed by a single cell and the current decoding point for that cell is the top of the graph grammar. Starting here the tree containing the graph grammar is traversed in preorder, leftmost-child first In order to construct the NN. Let's see how it works, the first operation stored in the graph grammar is SEQ (sequential division) and can be illustrated as follows:

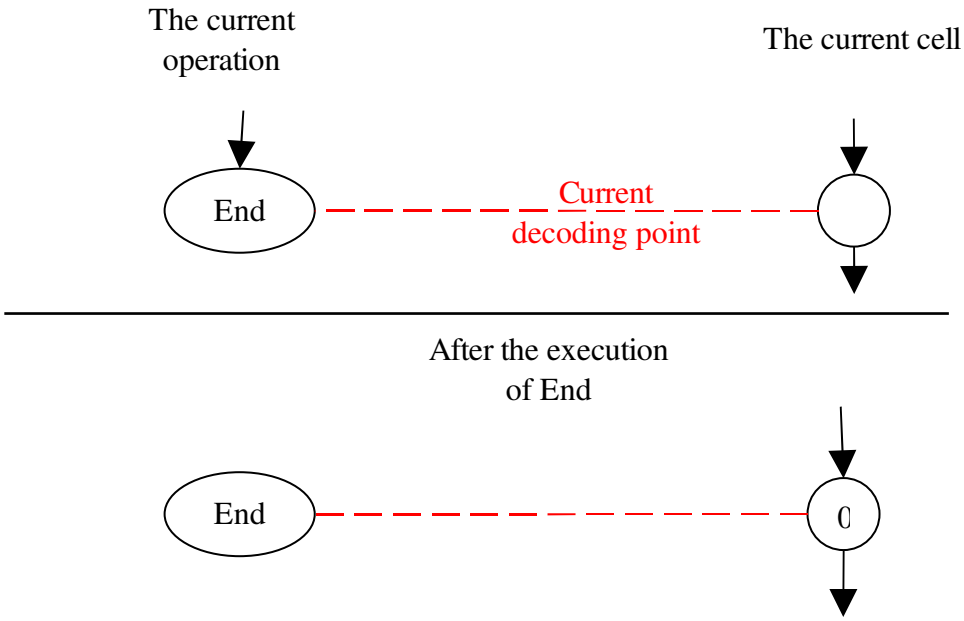


SEQ simply divides the current cell into two connected cells, the first of which inherits the parent inputs while the other one its outputs. The current decoding point is also divided in two each one of whom is assigned to its respective child operation and cell.

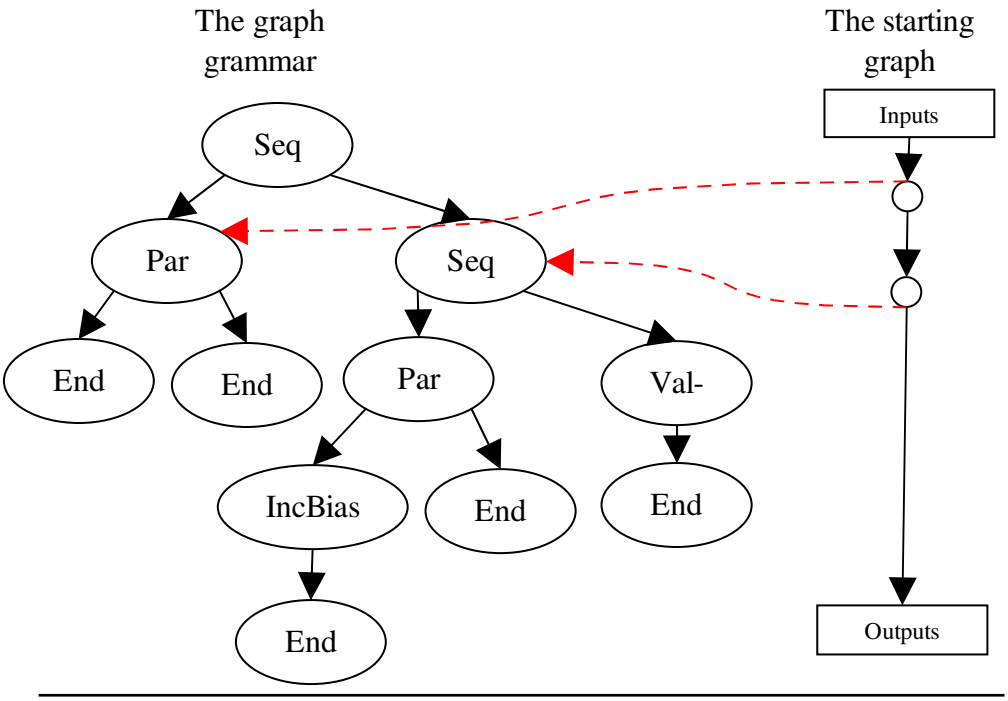
The PAR operation is another example of a non terminal operation (parallel division) which works as following:

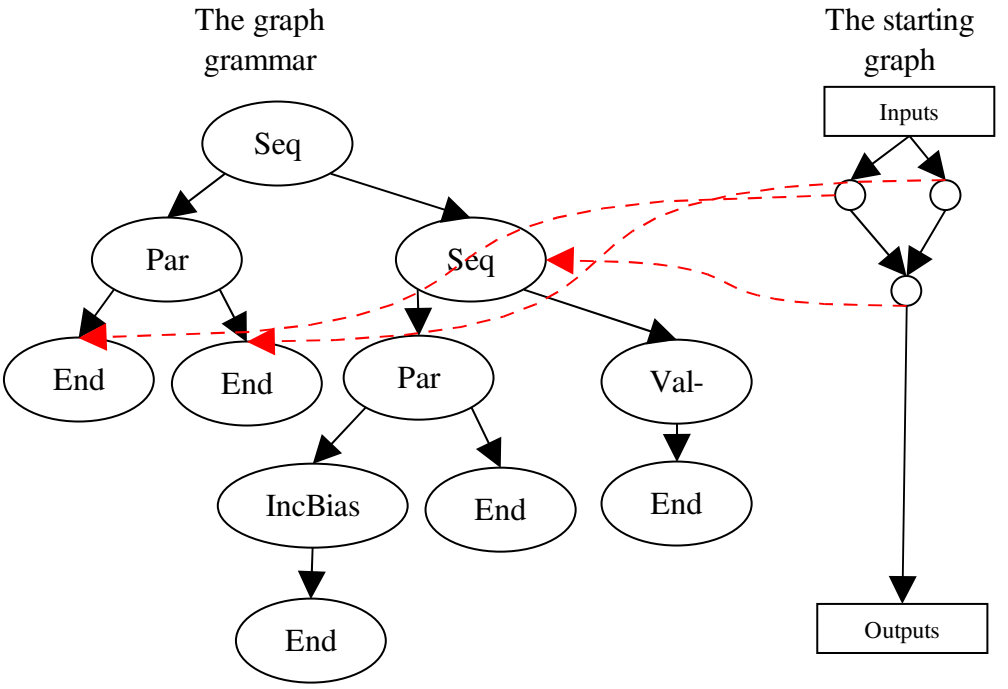


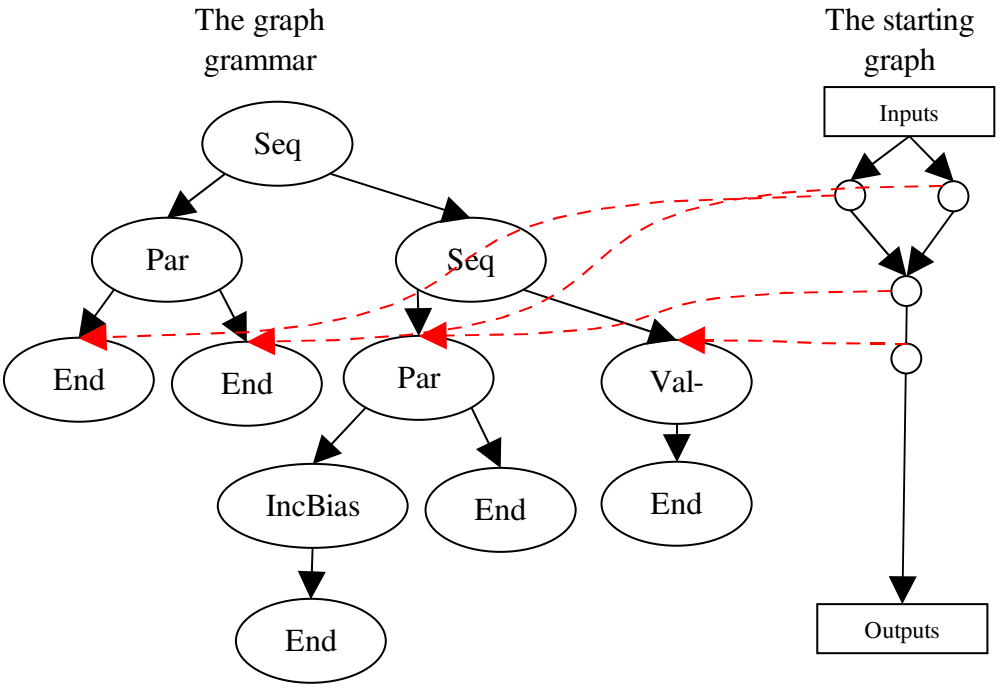
The current cell is subdivided into two cells, each one of which inherits all the inputs and outputs of the original cell, the first child cell moves its current decoding point to the left child of the PAR operation and the second to the right child. And now it is important to specify that the actual execution of the operations is FIFO, which means that once the current operation is executed the next child operation is stored at the end of the queue. Contrary to the previous operations, the END operation is a terminal and converts the current cell into a neuron, distinguishable from the cell for its threshold value inscription:

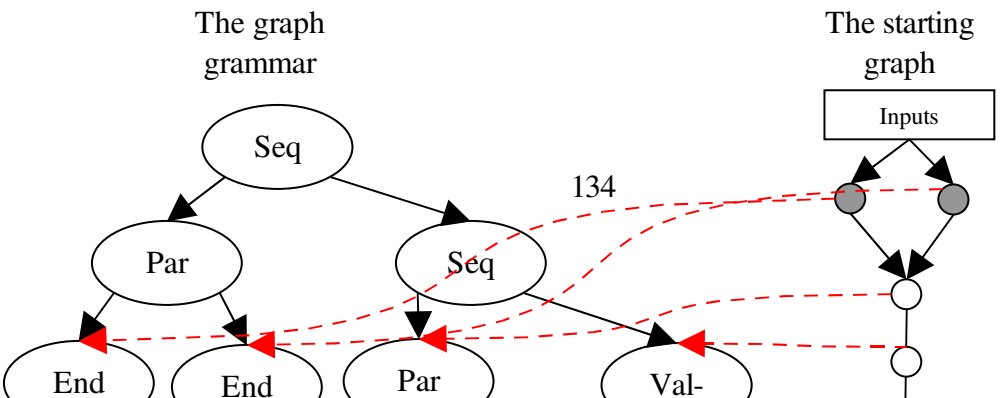


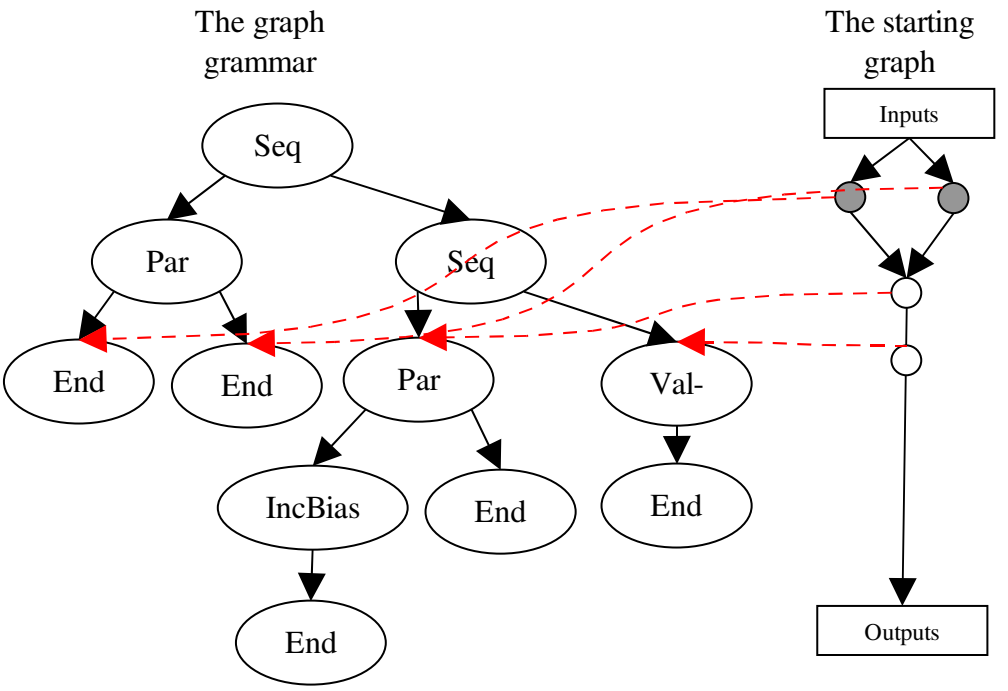
Now that the basic mechanisms have been explained, let's see how the graph grammar of the first example would be decoded:

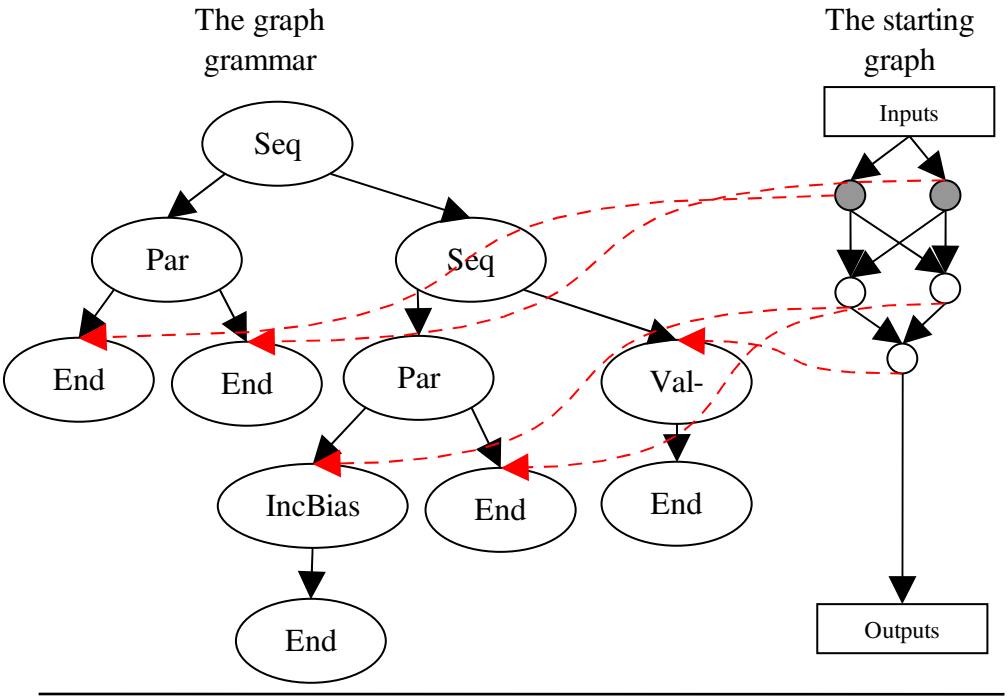



The graph grammar
The starting graph
Inputs
Seq
Par
Seq
End
End
Par
Val-
IncBias
End
End
End
Outputs

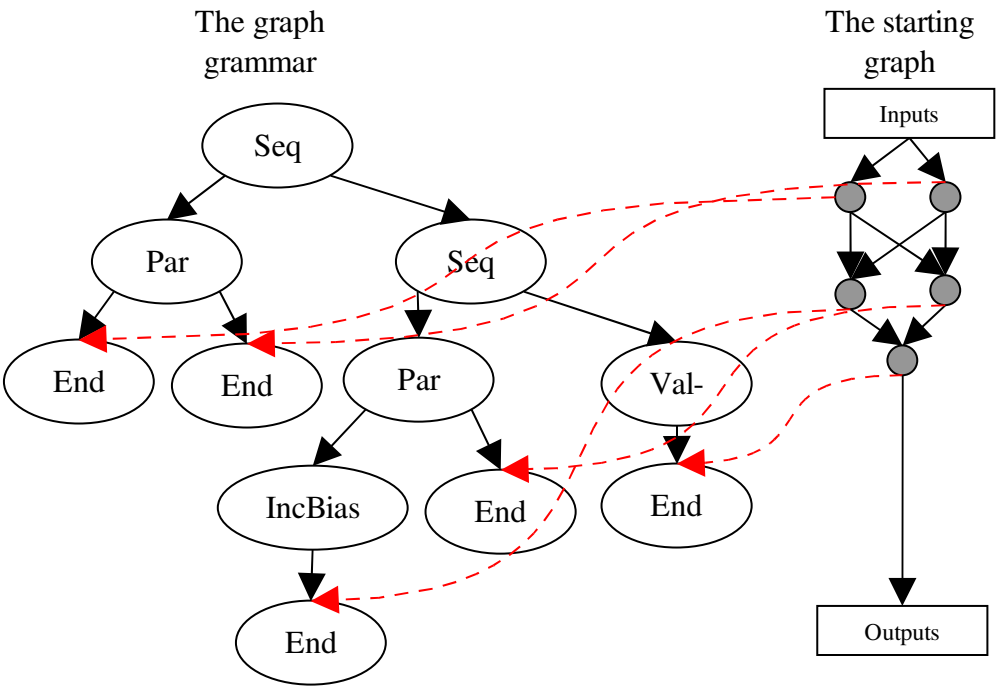

The graph grammar
The starting graph
Inputs
Seq
Par
Seq
End
End
Par
Val-
IncBias
End
End
End
Outputs


The graph grammar
The starting graph
Inputs
Seq
Par
Seq
End
End
Par
Val-


The graph grammar
The starting graph
Seq
Par
Seq
End
End
Par
Val-
IncBias
End
End
End
Inputs
Outputs


The graph grammar
The starting graph
Seq
Par
Seq
End
End
Par
Val-
IncBias
End
End
End
Inputs
Outputs

[…]



Since all cell decoding positions point to terminals the parsing of the grammar tree stops and the NN is considered complete. The non terminal operations VAL- and IncBias respectively modify the weights and the bias of their respective cells.

The possibilities of such encoding are evidently determined by the alphabet (the set of operators) used by the grammar; Gruau defined the minimal alphabet capable of describing all NN (complete alphabet) as being composed of four operations only: {CYC, END, PAR, CUT}. While END and PAR have already been introduced the remaining operations deserve a short introduction. CUT n means that the current cell has to eliminate its nth link while CYC stands for cyclic, and indicates the presence of a recurrent link in the ancestor cell. Another significant achievement of Gruau's work was the definition of a set of properties an encoding should respect. While traditionally the different encodings' features were left to the fantasy of the author and only their experimental results were used to validate their quality, Gruau described those properties that would distinguish a quality chromosomal codification [31]:

1. (Completeness): "*An encoding scheme is complete with respect to a set of architectures (resp. neural networks) if any element of the set can be encoded.*"

2. (Compactness): "*An architecture encoding scheme E is said to be topologically more compact than another encoding F if every architecture encoded by F with a code of size s, can also be encoded by E with a code of size O(s). If the scheme also encodes weights, E*

is said to be functionally more compact than F if for every Neural Network encoded by F with a code of size s, there exists a neural network encoded by E with a codes size of O(s) that implements the function associated to N."

3. (Closure): "*An architecture encoding scheme is closed with respect to a set of architectures, if every possible code develops an architecture in this set.*"

4. (Modularity): "*Consider a network N_1 that includes at many different places, a copy of the same sub-network N_2. The code of N_1 is modular if it includes the code of N_2, a single time.*"

5. (Power of expression): "*A parametrized neural network encoding has a power of expression greater than a programming language if for every program P written with this language, there exists a code of a parametrized neural network that simulates the program P. Moreover, the size of the code is O(s(P)), where s(P) is the number of bits needed to store the source code of P.*"

6. (Scalability): "*Let (f_L) be a family of functions on a finite domain, where L is an integer index. A parametrized encoding is scalable with respect to (f_L) if there exists a code such that for any L the neural network encoded with a parameter L computes the function f_L*"

7. (Abstraction): "*An encoding is abstract if it has a power of expression greater than a high level language*."

8. (Grammaticality): "*A coding scheme is grammatical if the object that is encoded can be translated into a grammar, a rewriting system.*"

As far as for the first four principles there is nothing new, just the fact that Gruau explicitly identified and specified as requirements those features that other successful encoding schemata presented. Although Power of expression is listed as a property it actually is a term of comparison and as a consequence the author thinks it shouldn't be listed here. Abstraction instead, is the real property and it is based on the definition of Power of expression. Consequently both 5th and 7th properties basically say the same; they indicate that the encoding should be compact enough by stating that its expressions must be more powerful than those of a programming language used as term of comparison. Scalability means that the NN complexity and the complexity of the problem it solves are both linked to an integer L; as the problem size grows the network's code remains constant and only its parameter L changes. E.g. in the AND network with n inputs, L determines the threshold value and the number of input connections, however the network remains composed of a single neuron no matter if it has 1 or 100 inputs, it is always the same. The 8th and last property simply defines when an encoding should be categorized as grammar encoding.

Concluded the review of the possible chromosome codifications it is time to face the next determinant aspect of an EA. We know that besides the chromosome codification the fitness and error functions used in the network topology play a central part in a successful search. Those error functions intended for network topology search are generally more complicated than the ones used for weight training. In fact they obviously have to take the RMSE of the network into account, as it happened during network training, but they must also take into account some secondary goals that have to be pursued when searching for an appropriate topology. Clearly one of those is to obtain a network capable of performing the given task with the least complex topology; this means that the number of connections and nodes has to be included in the fitness calculation as to avoid an uncontrolled growth of the evolved NNs. The measurement of the network's complexity and its integration in the error function are fundamental steps in the creation of networks that are not only capable of doing a job, but also to do it efficiently. The most common versions of complexity control are implemented by simply counting the number of neurons or connections that are present in the NN, and then applying a proportional penalization (also called handicap) value to the original error. One classic example of a fitness function that considers the complexity of the network can be observed in Mayer [32][34] who derived his fitness function from that of Rumelhart and that of Harp and Samad:

$$\varepsilon_c = \sum_{i \in C} \frac{w_i^2}{1 + w_i^2}$$

Where C is the set of all network links and w_i the weight of one of them.

$$\varepsilon_m = \frac{e_v}{n_v}$$

Where e_v is the number of misclassifications of the NN and n_v is the validation set size.

$$F = \alpha_1 (1 - \varepsilon_m) + \alpha_2 \frac{1}{1 + \varepsilon_c}$$

Where F is the fitness value and the sum of both α is 1.

The author stresses the fact that in this kind of fitness functions the final value is composed of two completely separated terms, the network's error and the network's complexity. The problem with this classical approach is that the α values are problem dependant; as a matter of fact some difficult problems might require large networks while simpler problems might need much smaller networks to achieve the same error rate, let's just remember the XOR network (4 links) with a 4-bit parity function (18 links) network. This means that if we wish not to let one term overweight the other we must carefully adjust the α values and to do this we must also know the expected network size and error, something that it isn't necessarily obvious. On the contrary if the first term happened to be dominant because of its large value, this would render the search orientated only in the direction of the low-error networks which often coincide with very, very large nets. If instead the second term was extremely large the network would search very simple architectures regardless of their error. As a consequence the author suggests a method where the error of the network is multiplied for a factor exponentially proportional to the network's complexity, such error (and its related fitness) function will be described later on when introducing JooneFarm. The author whishes to point out that both methods presented up to this point refer to networks where the complexity value of each neuron is fixed; said in other words, the transfer function of the neuron and the kind of synapse are not considered in the network complexity calculus. This is an assumption that makes sense as far as the kinds of neurons and synapses allowed are restricted and very similar. In effect both the direct encoding and the Kitano's encoding previously presented assume all the neurons to be of sigmoid type. If on the contrary, such assumption wouldn't be true and the EA had to work on neurons with very different transfer functions the author indicates a complexity calculus based on the training time of the network as the most appropriate. The use of time for reasons of complexity calculus makes sense as the network complexity calculus is generally done because of time constraints and not computer memory constraints. So given the following three prerequisites:

- If neurons and synapses with transfer functions which have significant performance differences can be used in the evolved net
- If the complexity calculus is done because networks that perform a job quickly are required (no memory constraint on network size)
- If all the computers which train the networks that compose a generation have (almost) identical training performance

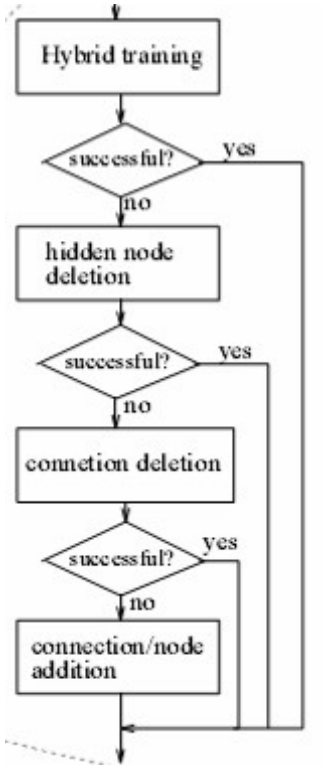
It is meaningful to use the training time (probably milliseconds) as a complexity indicator. Please note that point three would be false in case of a distributed training environment which uses computers whose training times for the same network differ significantly. An example of a possible error function that considers the training time could be the following:

$$E*H^{\mathrm{Log}_{\sqrt{x}}t}$$

Where:

E is the RMSE error.
t is the network training time.
H is a user defined handicap parameter generally in the range [1,5 ; 3]
X is the training time of the smallest valid network.

Let's take EPNet, a network evolving system developed by Yao and Liu, as another example of complexity control. This time it is the mutation operator used in EPNet that takes care of the complexity control and not the fitness function. The idea is to attempt different series of mutation one after the other until an offspring with a fitness higher than the parent's one is found.



After the network is trained (Hybrid because it is a mix of BP and SA) there is an attempt to remove a neuron, if this doesn't yield any improvement then a connection is deleted and the resulting network tested in search for improvements. If a reduction of complexity doesn't turns out to be advantageous then EPNet tries to augment the size of the network by trying to add a connection first and a neuron then.

5.4- Hybrid methods and other application fields

Whoever has ever tried to train the same NN several times probably knows that it is often the case that during the learning phase a neural network trained with BP can achieve an RMSE value which is much below the average of all other learning sessions. This is the most straight forward example to show that a network is capable of performing a certain job only if the set of its weights is properly adjusted; consequently a network topology without appropriate weights might not be of much use. Sometimes it even happens that BP just won't converge to such optimal weights at training time. It is therefore important to treasure the weight vector of a good network as it represents the information gained during previous searches and experiments. As if this wouldn't be enough we should not forget about the many other parameters, for example the learning rate, which affect the learning capabilities and convergence of a NN. In order to include all this variables into the evolutionary process several methods that take into account weight information as well as other parameters alongside with topology information have been developed and they are generally referred to as Hybrid methods. Even if a definition would result restrictive we could say that a hybrid method is an EA whose optimization efforts are not directed in just one direction (weights, topology, parameters, training set...) but in many, alternatively a hybrid method is one that uses a combination of different optimization techniques.

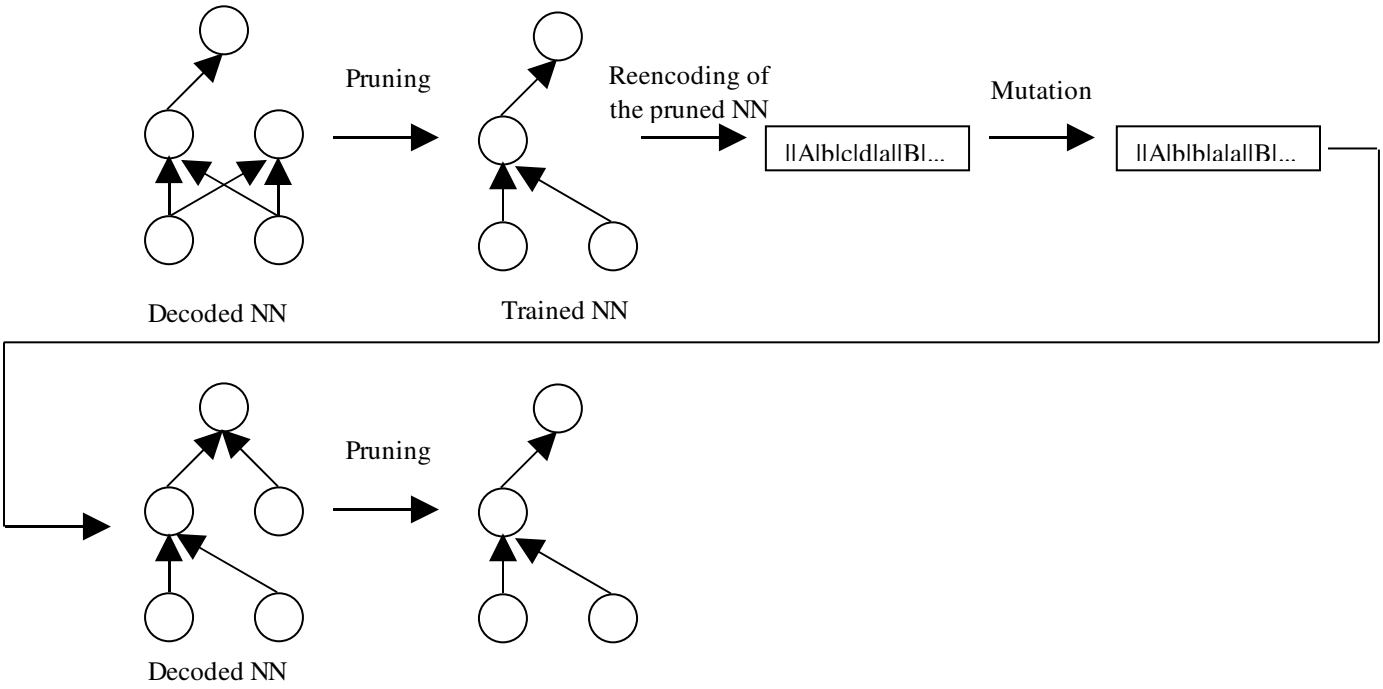
The previously described problem where BP doesn't always converge to the same weighs has been named by Yao [18] as "noisy fitness evaluation" and is a major problem for all those who try to create a fitness function based on the NN's RMSE. In fact the fitness of a chromosome varies according to the network's RMSE and this is non-monotonic. As we have seen BP might not converge to a good set of weights causing the RMSE to assume a much higher value than the one obtained during previous trainings. This damages the selection process as the chromosome's fitness values are sometimes bogus, prompting the selection of chromosomes that actually aren't the best ones. While the most obvious method to overcome such problem is training the network several times and/or using more training cycles each time starting from different random weights, this can only arbitrarily reduce the evaluation error at cost of exponentially slowing down the genetic search. A very simple alternative to solve the "noisy fitness evaluation", at least for the parent chromosomes, would be to keep the best RMSE value obtained during the previous training sessions and compare it with the value of the last training session; this would be enough to render its fitness function of the parent chromosomes monotonic. Even if the two previous methods can attenuate the problem the real solution is the inclusion of the network weights into the chromosome so that they are evolved along with the network topology.

While the first experiments of Kitano [16] focused on network topology determination only, he later investigated methods for a combined determination of network topologies and their weights[27]. This step was taken as Kitano considered the fact that the functionality of a network is deeply based on its weights; as a consequence, he concluded that trying to establish the fitness of a topology using weights that randomly changed at each encoding/decoding of the same wasn't appropriate. The idea was that convergence could be faster if the weight determination problem would be considered during the network designing stage. In this subsequent work Kitano went on using fixed length chromosomes (180 binary values) and a pretty standard GA with a two point crossover, point mutation and proportional reproduction with elitism; as for the training of the networks the standard BP was used. Although Kitano had gained knowledge about the importance of the context for a topology evolution by the time, he kept the context free grammar encoding of his previous work without modifying it. On the other hand, the original mutation management changed becoming probabilistic and ranging from a minimum mutation probability of 5% to a maximum one of 30%. The used crossover rate was set to 50% which actually is a very high rate for a topology search application, a field where generally crossover is not used at all. The network's population was composed of 20 chromosomes and their fitness was set equal to 1/TSS (total sum square error) a method that augments the fitness variance in comparison to those that use RMSE. However the GA used by Kitano was only a part of the NN evolution system, a system which was articulated in 5 stages.

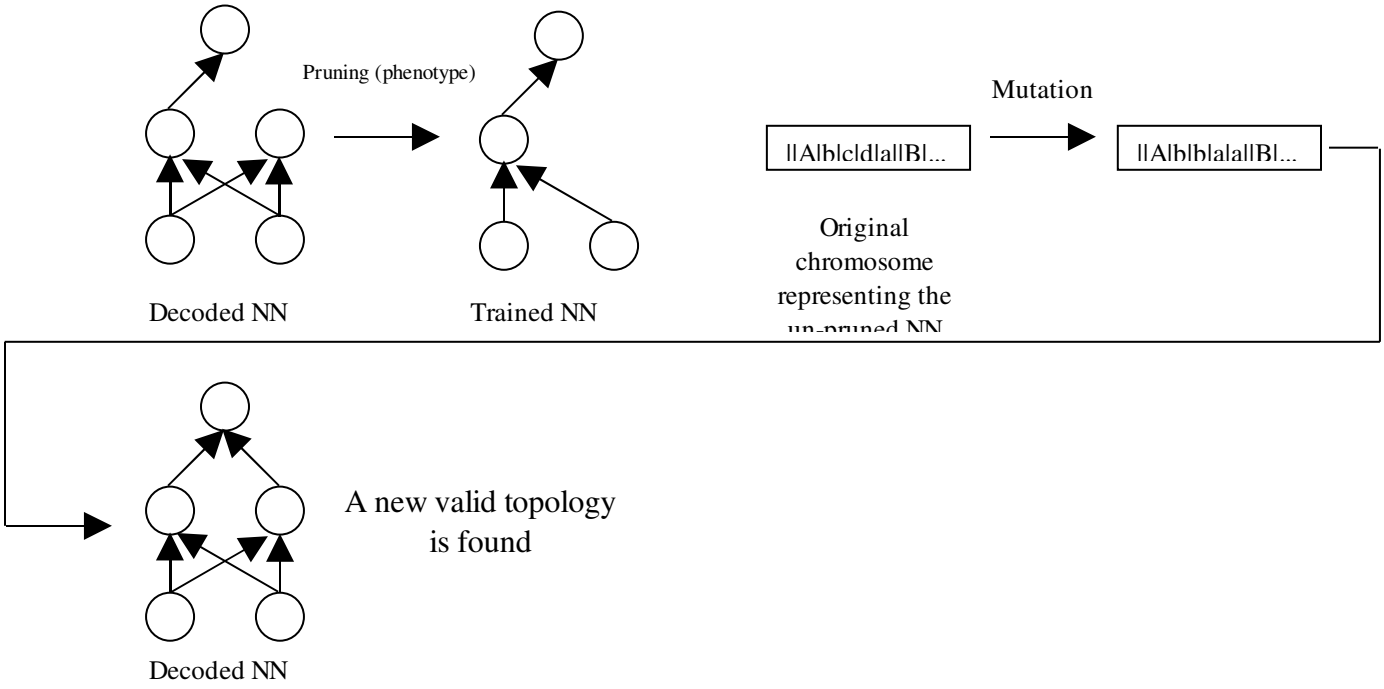
1. Evolution Stage
2. Developmental stage
3. Weight distribution stage
4. Network pruning stage
5. BP training stage

During stage 1 the described GA was applied to the chromosomes while stage 2 represented the interpretation of the grammar stored in each to chromosome with the goal of network creation. Stage 3 assigned the weights to each link while stage 4 (not used during tests) cleared the NN of all those nodes that had no functional meaning. Finally stage 5 trained the network. Once again the problems Kitano used to test his EA were XOR and decoder/encoder, however this time considerations about the scalability of the encoding techniques in respect to the problem complexity were made; as network sizes increased the direct encoding without weight learning definitively proved much worse than grammar encodings with weight learning, at least as far as fitness and chromosome size were concerned. The mechanism that Kitano used to implement weight learning used a matrix, called CAM. Several encoding versions of such matrix were tested but the basic idea remained the same, given a link among two nodes its weight was determined by the value in the CAM matrix that was at the coordinates (x,y), where x and y were determined by the starting and ending node. Kitano experimented with three different CAM versions, which

differed from one another in the form in which the node to row/column association was made and how the values of the matrix where determined. The so called “binary CAM” proved to be the best, in this case each node was coupled with a three bit pattern and the union of the starting node pattern with the one of the end node provided a 6 bit string which was then interpreted as a real value weight, further calculations on the same 6 bit string provided the possibility to determine if the connection existed or not according to a connection matrix where, again, rows and columns where assigned to the nodes’ patterns. The explanation Kitano gave for the good results of this hybrid encoding method was that it rendered the copy and moving of network substructures easy, that it successfully coupled weights with structures that would have otherwise proven inefficient. Kitano also considered that the redundancy of the grammar encoding helped creating a sort of diploidism, where a rule could be specified more than once but having only one of its implementations interpreted during the network’s decoding; in case mutation disrupted the main rule a second one would be still available. In the end Kitano proposed a method to interpret rules coming from networks whose chromosomes had different sizes, thus theoretically enabling to use a variable chromosome size. Another suggestion made by Kitano is to prune the originally decoded network so as to obtain a more computationally efficient network, this could be done by eliminating all those nodes without input or output links and those nodes with a single input link. Unfortunately no indication is made of how this process influences the original chromosome. In fact the author agrees with the necessity of trimming the decoded network so as to improve the EA global performance, but also wishes to warn the reader that this should be done only for fitness calculations scopes. Changing the original chromosome in order to reflect the current trimmed network would eliminate much of the genetic diversity making it much more difficult to generate new architectures.



The point is that a node without outgoing links might seem insensate and its trimming would save precious time during the network's training, however if the chromosome is modified by eliminating such node, also the possibility of creating another valid network with help of small mutation step is discarded. In fact if the chromosome stores nodes without outgoing links it might happen that successive mutations could create the missing outgoing link ending in a new valid architecture. If on the contrary all nodes without outgoing (incoming) links are systematically removed, mutation should create: a node, at least one outgoing link and at least two incoming links in order to create an acceptable chromosome that wouldn't be reverted to its pre-mutation state by pruning it. This would be a quite fortuitous event, and would have the consequence of raising the mutation rate to a point that the original network could easily be transfigured by a single mutation, thus rendering the relation between a network and its mutated clone unrecognizable.



E.g. The two previous diagrams clearly show the results of the re-encoding of pruned network; to reach the new valid topology of the second diagram staring from the pruned network of the first diagram three different links have to be contemporary mutated. A thing that implicates no building block can be established for a starting network containing solely three links. This problem is particularly evident for small size networks and is attenuated as larger networks are used, however even in a very large network a more refined search of its sub-structures wouldn't be bad. The biological confirmation of such proceedings might be

found in the existence of many animal organs that do not have an immediate use. The wings of a penguin or those of a chicken are the classic example for a middle step in the evolution of a species as those species are not capable to fly but still present features that allow us to trace back their evolutionary process.

Michel and Biondi [26] presented a hybrid weight-topology optimization method that took additional inspiration from biology in order to develop NN using a method similar to cellular protein synthesis. The idea of copying nature has not always produced useful strategies when it comes to emulate each and every detail, this mostly because the nature of the solving problems are quite different from those of the inspiration source and because the complexity and depth of the problems that can be analyzed by means of modern computing is tremendously simple in comparison to the evolution process in nature. However Michel and Biondi successfully translated the context mechanism of protein synthesis into EAs, providing a biologic foundation for its effectiveness. The original observation was that, during protein synthesis, the DNA decoding is influenced by the presence of other proteins called activators and inhibitors; if an activator protein is present then the synthesis of the current protein will go on while if an inhibitor protein is present the current protein won't be synthesized. This basically means that also protein creation obtained through the decoding of DNA, and consequently all cell activities, is context aware. Michel et al. defined a chromosome as a set of rules whose application was dependant on two "message" sets. The first set specified those messages that had to be present in the context before the rule could be applied, while the second set specified those messages that mustn't be present in case the same rule was to be applied. Each time a rule is carried out one or more messages can be added to the current list of messages. The actions that are executed by the rules resemble those of biologic cells: creating connections to another cell, creating another cell or specifying the kind of cell (transfer function). Although the description of the GA used is rather superficial Michel and Biondi clearly indicate the use of one point crossover and mutation for the creation of unconstrained networks, this means that the networks created with this methodology can be recurrent as well as non recurrent. Finally the evolved networks were used in a simulated Khepera mobile robot that imitated a biologic organism in search for food whose movements decreased its simulated energy reserves. The conclusions were that after 200 generation the network fitted the inputs/outputs of the robot and that after 300 generations some sort of "search" behavior emerged in that robots did not move in lines or circles anymore but were attracted by "food".

Bae and Shimohara [21] developed a modular system for developing neural networks which might is an interesting mix of different techniques and application fields. Instead of evolving the architecture of a net whose nodes are represented by single neurons Bae and Shimohara used modules as the basic building block. Each module being composed of excitatory, inhibitory, control and communications neurons grouped together to form a single categorizing unit that will assign input patterns to discrete categories in a winner take it all

fashion. The number of nodes each module has, the learning parameters of the Hebb rule used to train the weights of the inter-module connections and the network structure were encoded in the chromosome. This means that the EA used in this kind of experiments is hybrid as it not only modifies the architecture of the net but also other important factors such as learning parameters and the size of the modules. The network encoding used by Bae and Shimohara was similar to the grammar encoding of Kitano in that chromosomes were composed of a set of left side generation rules, but with the difference that this time they were context-aware. The context awareness implies that the application of the production rule is determined by a condition that has the form of an expression and by the context in which the alphabet symbol is placed. An example of a rule:

B(x1,y1) <B(x,y)> C(x2,y2) : x > 10 -> C(x/2,y) C(x/2,y-1)

Where the terms enclosed in <...> are replaced by the part of the rule following the -> symbol if and only if the condition (x >10) is true and the context (B(x1,y1) , C(x2,y2)) is given. E.g. in:

A(100,1) [B(8,0) B(12,-1) C(10,-1)]

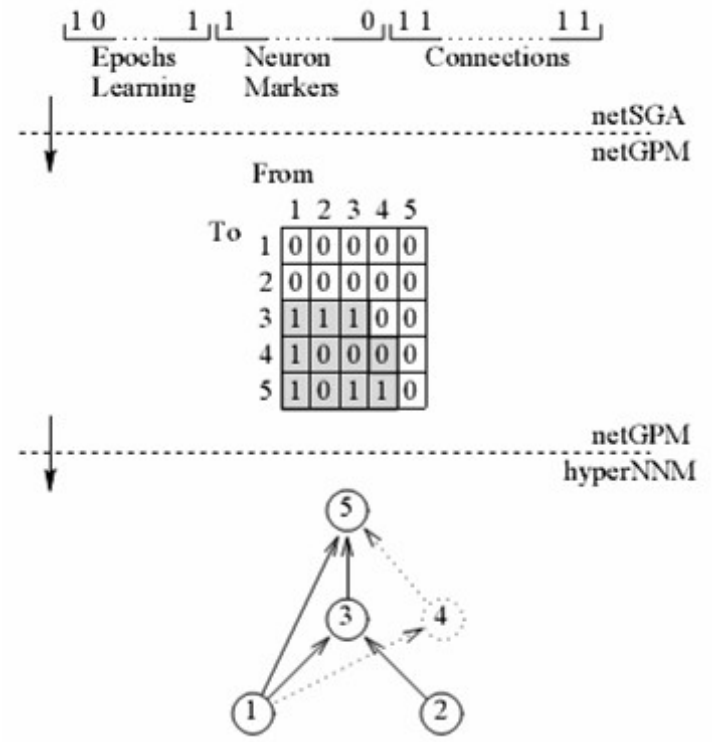
The underlined term fulfills all the requisites for the application of the rule as its first value is greater than 10 and the B term has a right neighbor of type C and a left neighbor of type B. So the previous expression becomes:

A(100,1) [B(8,0) C(6,-1) C(6,-2)]C(10,-1)

Using such rules and an initial population of 50, Bae and Shimohara evolved networks that were subsequently trained on 300 hand written digits, each rasterized in a 10x10 grayscale image. Since the idea was to obtain a pattern recognizer, a multilayer perceptron with 20 hidden nodes was used as term of comparison; ten different sets of patterns, each with 300 patterns differing from those used in the training phase were used as test. The results were favorable for the networks evolved by Bae and Shimohara whose performance was never surpassed by the perceptron achieving an average of 97.33 correct recognitions every 100 patterns. The multilayer perceptron instead, achieved only an average recognition rate of 95.33%. Beyond the positive results obtained by Bae and Shimohara it is interesting to compare their method with the one used by Kitano, who actually was the first in encoding a network according to its production grammar. Bae and Shimohara not only used EAs to evolve the various aspects of the net, such as learning parameters, that were beyond topology but also recognized the relationship between the elements inside a neural network. They acknowledged twice the fact that the functionality of a neural network element is heavily determined by its context, the first time when they constrained the network topology to a series of building blocks, in fact the building blocks they used were formed by a group of

neurons bounded together in a fixed pattern of connections which maintained its functionality even when the number of neurons changed. The second time Bae and Shimohara showed the importance of context is by the use of context-aware production rules, as the production rules ensure that the neighborhood of a building block is of a well defined sort. Another interesting point is that the production rules just presented include variables (x, y, z...), these variables are handed down during the production of the network as they are included both in the left side and in the right side of a rule, which implies that the starting symbol carries some parameters that will influence the network creation. Just for comparison Kitano's starting symbol was fixed and unique, influencing in no way the development of the network. It is however obvious that the structure of the building block is the product of deep problem knowledge, and that networks evolved with this method might be absolutely unusable in problems for which the building block wasn't originally meant for. Under this aspect the EA is in charge of the scalability and the tuning of the components but does not actually find a topology that represents the solution of the problem.

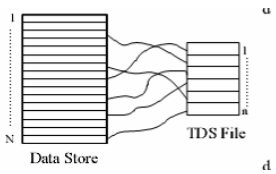
The work of Mayer [32] represents a case of what is generally defined as co-evolution. Co-evolution takes into account the fact that all those structures subject to evolutionary processes, which as we have seen might be NNs as well as strings coding a traveling route, are deeply correlated with an environment that determines their fitness. Up to now the environment that used to determine the quality of a chromosome has been presented as immutable, e.g. the distance among the cities of the TSP and the training patterns for the NNs. However this assumption of a fixed environment might not be the case of many applications that take place in the real world. Mayer introduces the example of a robot equipped with infrared sensors which return distance information about the environment's objects. It is obvious that in this scenario we wish to evolve a NN capable of guiding the robot in a variety of environments containing different obstacles (kitchen, living room, other robots, humans…) and not in just a single environment. In order to do this the robot controlling NN shouldn't be designed and trained with the data belonging to a single environment as it'd probably result unable to adapt to other situations.
So as to achieve this, Mayer used to encode the evolving neural networks in a string containing the relevant parts of a serialized connection matrix (a modified Miller Matrix).



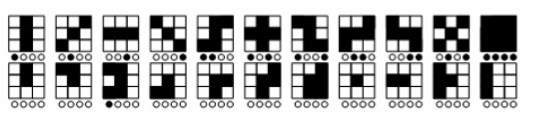
There are a couple of interesting things about this technique. The first is the notable reduction of the chromosome size, given that more than half of the Miller Matrix remains unused in a fashion that resembles a binary connection matrix, the inclusion in the chromosomal string of only those parts of the matrix that are actually used renders the final chromosome much more compact. In addition Mayer introduced the convention that all those bits found on the diagonal indicate the presence (x <> 0) or absence (x = 0) of the neuron, independently from its number of connections. The author points out that such use of the matrix's diagonal bits increases the chances that simpler structures are found by the evolutionary process. One of the main problems of those encodings based on a binary connection matrix (Kitano, Direct Encoding) is that it is relatively easy for them to add a neuron to the net but it is not as easy to remove it. As a matter of fact (according to the specific implementation) one (two) link suffices for a new neuron to appear in the network. While on the other hand, the removal of a neuron is a much harder task as it requires that all the input and (or) output connections to the neuron are removed before the neuron is detached from the network. This clearly renders the previously presented encodings constructive, which means that they are very well suited for the exploration of networks that are bigger than the one encoded in the current chromosome but they are hardly capable of simplifying it. Mayer's encoding doesn't have this limitation, being equally suited to expand as well as to simplify a NN topology.

Still Mayer's most notable achievement regards the co-evolution of networks and TDS (training data set). A TDS is nothing more than a set of patterns that is used to train NN, until

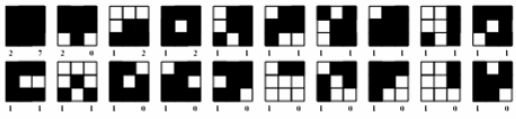
now nothing new, but the remarkable thing is that Mayer created a population of TDS where each element was a series of pointers, each one indicating a training pattern of the central repository. This means that by evolving the TDSs it is the set of patterns that changes and not the content of the patterns.



The goal was to train a network capable of recognizing lines in a 3x3 matrix of B&W pixels using a fixed NN structure. Two fixed network topologies were used; the first was the minimal NN topology capable of solving this problem, which was known in the literature, while the second was a 9-10-4 perceptron. The first of Mayer's conclusion is that evolving a population of TDSs it is possible to obtain a set of patterns that train a NN much more efficiently. Mayer evolved the TDS using fixed NN architectures and assigning a fitness value to each TDS based on the error of the NN after it had been trained with the very same TDS. He compared a human created DTS which clearly showed the lines that had to be recognized.



With the TDS evolved with a classic GA (50 individuals, 50 Generations, 2 point crossover, mutation, tournament) which did not present resemblances with the patterns that the NN had to recognize.

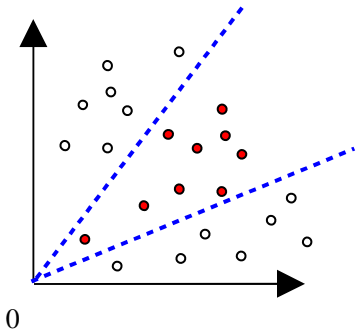


Mayer reports that the human selected TDS proved unsatisfactory in training both NN architectures while evolved TDS was extremely efficient, with 100% recognition rate in case it was used to train the minimal topology network. In the following experiments Mayer used real co-evolution meaning that not only the TDSs evolved but also the NN topologies. The first experiment saw several TDSs evolved using the minimal NN topology already known as capable of solving the above mentioned problem. After 50 generations the best TDS evolved through the use of that fixed topology was extracted and used as fixed environment for the evolution of a network topology. This first case could be described as sequential co-evolution.

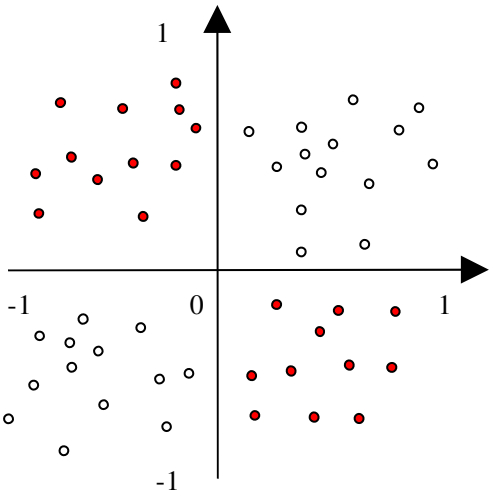
In the second experiment both NNs and TDSs were contemporary evolved. Each NN

was randomly assigned to a TDS, after the NN was trained using the selected TDS a fitness value was calculated on the base of the NN error. Such fitness was then assigned to both the NN and the TDS. What Mayer found out after several repetitions of the two experiments, was that with the only exception of a couple of synapses more, both experiments found the topology of the minimal NN. And, as the reader can easily imagine, the evolved topologies had a recognition rate of 100%.

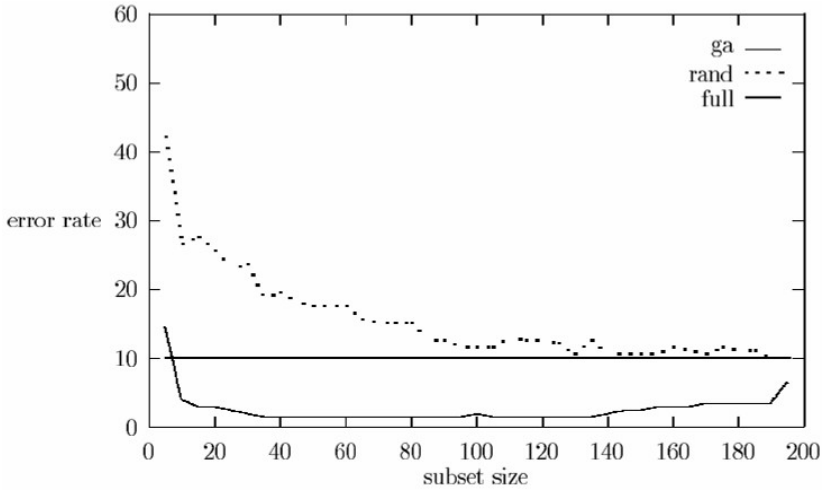
Another interesting work that analyzes the connections between TDSs and NN during the learning phase has been presented by Reeves and Taylor [33]. The importance of selecting the correct TDS is recognized in this paper, while the fact that TDSs often contain superfluous patterns which are not vital for the training is stressed. In effect, if we visualize the patterns as points on a surface and their categorization by the NN as lines dividing them into categories, it is known that those patterns that are nearer to the boundary lines are much more influential for the determination of the boundary itself than those in inner areas.



That means that given a set of training patterns not all of them have the same importance in determining the network classification capabilities of future patterns, consequently some patterns might be eliminated while others, which are fundamental, should be kept. Reeves and Taylor tried to determine relevant TDSs by using a GA, the goal was to optimize the training of a NN by finding the most appropriate set of patterns. The first experiment consisted in a continuous version of the XOR problem [-1 ; 1] where the patterns had to be classified according to the Cartesian plane quadrant they belonged to.

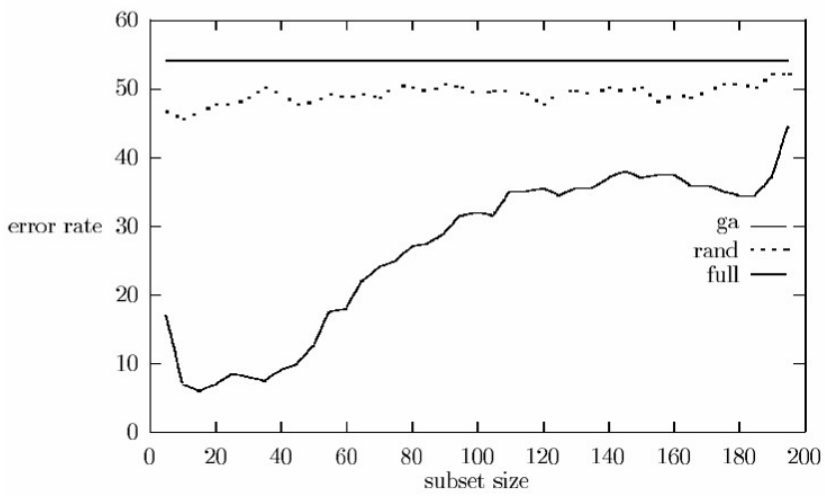


The experiment included the creation of TDSs of various sizes, ranging from 5 up to 195 (out of 200) patterns, and the subsequent training of a net 2 centre radial basis function NN with such TDSs. For each TDS the network was also trained with a TDS of the same size but whose patterns were randomly selected in order to have a term of comparison. The conclusions were that the same network once trained with those TDS selected by the GA always presented a lower error than when trained with the randomly selected patterns or even with the full set of patterns.

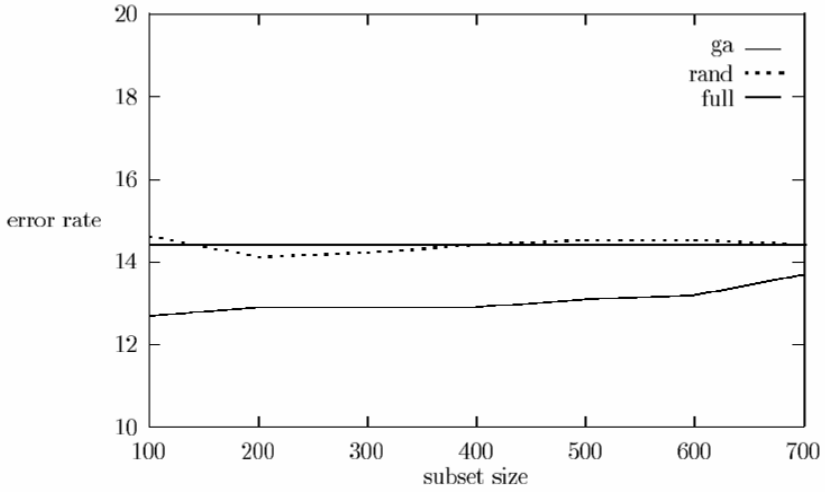


The same experiment was then run on a network whose topology was not optimal (only two RBF elements), the conclusions in this case were that a carefully selected TDS can train a

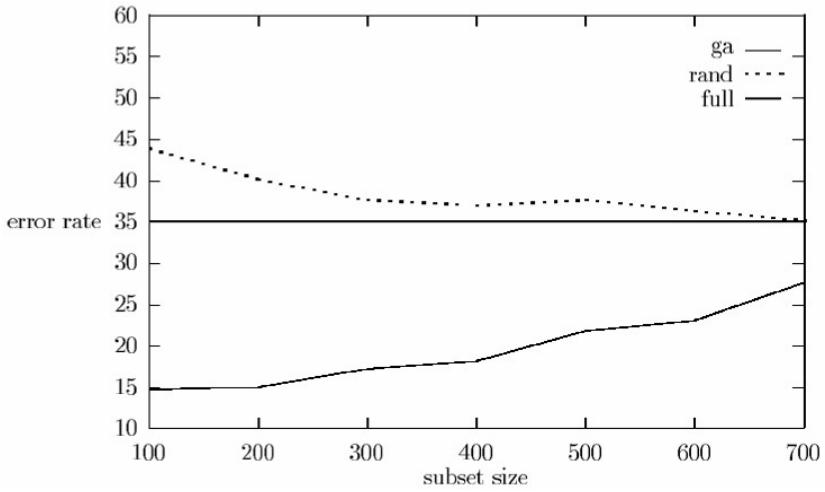
deficient network supplying, at least in part, for its topological faults. The smaller the number of patterns is the higher the advantage of training the network with a GA selected TDS is.



In order to prove these "laboratory results" had a real application, Reeves and Taylor used the data of 2403 customers of a major UK mortgage loan company in order to train a network that would classify customers in reliability classes. The network took advantage of 28 different parameters assigned to every customer and each of the assignable classes was supposed to include those customers whose estimated paying reliability was similar. The patterns obtained from past customer data were subdivided in three groups: a training set, a validation set and a validation set for the best network. The executed tests were similar to the "lab tests", as the first used the best known network topology and confirmed the advantage (though this time smaller) of training a network with a good TDS, especially when the TDS is composed of few patterns.



The second test used a network reduced in topology containing only one RBF element out of the optimal 6. As in the second "laboratory test" it showed that a sub optimal topology accentuates the benefits of a genetically selected TDS.



Both papers [32,33] show an impressive potential of improvement in the NN training field, however it must be said that those experiments that show the largest gains in the final error reduction were those experiments where the optimal topology was already known and used to train the TDS. If we consider that one of the most interesting aspects of having a good TDS is that the search for the optimal topology is much more efficient, the necessity of the optimal topology in order to find the best TDS results in a "Chicken And Egg" dilemma. However such techniques could give an advantage in case the maximum creatable network is smaller than the optimal one and nonetheless we wish to find a sub optimal NN. Surely the selection of TDSs allows us to train networks more accurately, which is always an advantage, but the biggest potential would be the co evolution of NNs and TDSs. Another potential research field could be the exploration of alternative topologies for those NN where the best known topology allows the creation of an efficient TDS.

One last field where EAs have been applied is the search for an optimal learning rule. It has already been indicated that the results of a NN training change according to the particular learning parameters and that it is possible to include these parameters in the chromosome of a network (Mayer [32]). Given that the change of such parameters effectively modifies the NN learning behavior, this could be seen as the first step in the evolution of the learning mechanism of a NN. There have been many attempts to evolve the parameters of a BP learning rule and the initial weights of a NN [17], sometimes obtaining values that are in line with those traditionally used by "human" and sometimes obtaining extreme values; it seems however that the number of cycles used to train the network and the size as well as the kind of

network used have a deep impact on the resulting training variables. This is yet another confirmation of the already known view that the right BP parameters are problem dependant.

It is nonetheless possible to evolve not only the parameters of a learning rule, but the actual formula too. Such task makes sense as different NN architectures require different learning rules in order to obtain the best results if not to be able to train the NN at all (e.g. BP and XOR with step perceptrons); which means that even if we would be capable of exploring all possible architectures and all TDSs, the search might be fruitless if the learn rule is not appropriate for the problem and its kind of network. In such scenario it is obvious that the roles of population and training set change dramatically. As a matter of fact we need a set of NN and their relatives input patterns so as to determine the quality of each learning rule. This because the population that has to be evolved is composed by a set of chromosomes representing the learning rules, chromosomes whose fitness values are calculated with help of NNs and their TDSs. Generally the average of the error values obtained by training each network with the given rule is used to determine the fitness assigned to each learning rule chromosome. Since the search space of all learning rules is simply huge it is often the case that some assumption are made, something that resembles the NN indirect encoding methods previously described. Such limitations might regard the information available for the learning rule and constraint it to be local, or the use of a pre-defined learning function where only some parameters evolve, or even the use of a single neuron transfer function [18][35]. An interesting approach in this problem has been taken by Runarsson et al [35]; in fact they used a second neural network in order to work as a trainer of the original network instead of just evolving a formula.

Chapter 6

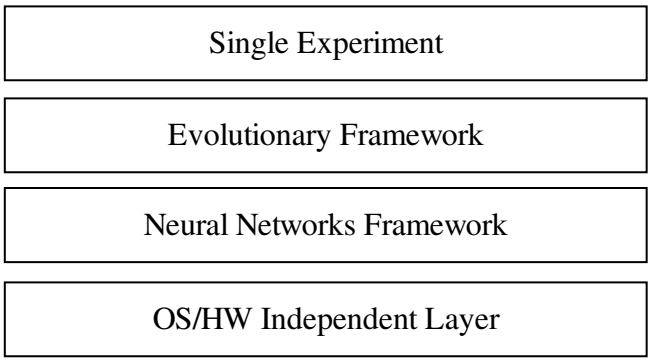
JoonePad and JooneFarm

6.1- Introduction

The current document is only part of the whole Dilplomarbeit work which contemplated the developing of the JoonePad and JooneFarm programs in collaboration with their original author Paolo Marrone. In the following chapter both programs will be introduced, it will follow an insight in their architectures and a brief explanation of how to use them.

6.2- The Frameworks

By reading the available literature on a theme such as evolutionary computation it quickly becomes clear that there are serious difficulties in comparing the results of the various papers with one another. The first and most banal problem is that in the field of genetic algorithms (as in IT in general) there might well be five, ten or even fifteen years between one report and the other, which according to Moore's law are real ages in the architecture of a computer and its related software. Other difficulties that are often found while comparing the different approaches to EAs is that the authors hardly ever describe in detail the implementations used for their experiments, mostly they tend to completely omit the part regarding the NN simulation. As a consequence the only assumption that can be made is that the NN simulator is an experiment-tailored piece of software, which further augments the uncertainty about the general validity of some results. It is clear that such shortcomings represent one of the main causes of the sometimes so different results[18]. The ideal would be to define a standard implementation (as far as this is possible in the IT branch) of a neural network framework on top of which all the experiments could be made. Clearly such framework MUST be open source if quality, transparency and expandability are to be secured. Another point is that such framework should be easily portable from one architecture to the other in order to satisfy multiple needs and secure, at least in part, its independency from a single hardware or software architecture.



While choosing the lowest layer it was pretty soon clear that the most suitable programming language would be java and, given that building a solid NN framework from scratch would have needed years of development, there was no doubt that an already available and established framework had to pose as the NN Framework layer. This restricted the choice pretty much and looking on the internet the Joone framework proved to be the best candidate to form the cornerstone.

"Joone consists of a modular architecture based on linkable components that can be extended to build new learning algorithms and neural networks architectures. All the components have specific features, like persistence, multithreading, serialization and parameterization that guarantee scalability, reliability and expansibility, all mandatory features to make Joone suitable for commercial applications and to reach the final goal to represent the future standard of the AI world." [36]

And still:

http://www.developer.com/java/other/article.php/10936_1546201_1
http://www.google.com/Top/Computers/Artificial_Intelligence/Neural_Networks/Software/
http://www.heatonresearch.com/articles/13/page1.html

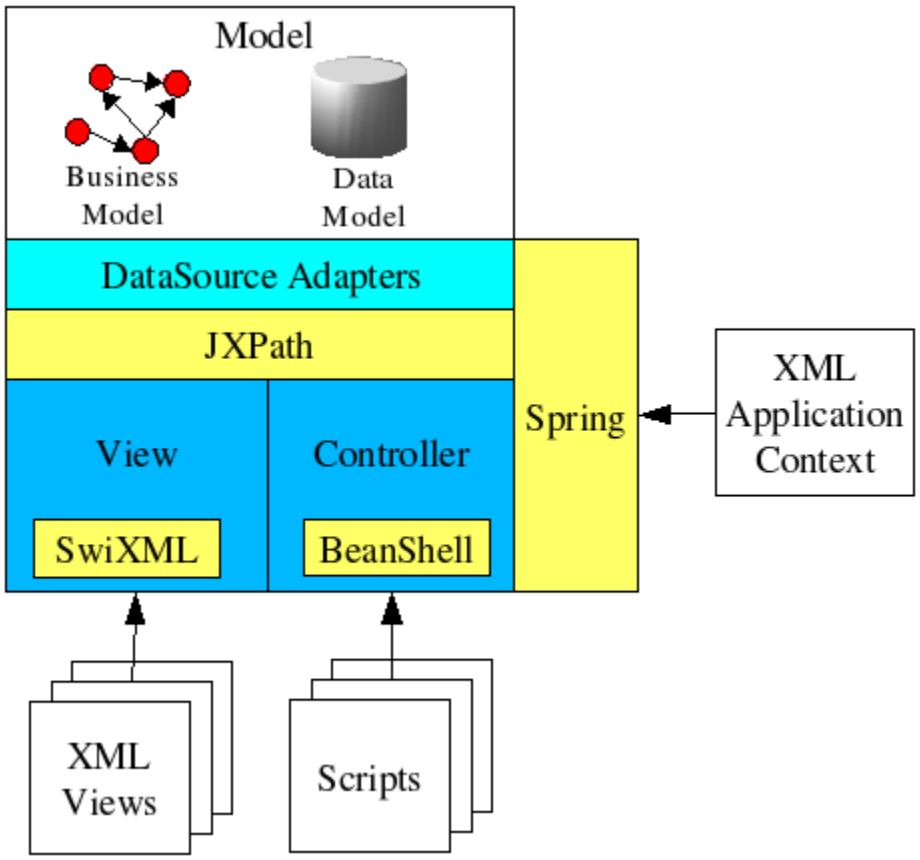
6.3- The implementation of the evolutionary framework

At the moment this Diplomarbeit started Joone was (and still is) going through a transition phase, in fact the original editor that was traditionally included with the framework had already been abandoned and its successor JoonePad only existed in the form of an idea.

Consequently the first step was the implementation of a NN editor that used the original Joone framework. Throughout the development of the new editor and all subsequent programs the only two active members of the project were Paolo Marrone and the author.

The new developed GUI is distinguishable because of its new fresh look and several interesting software development techniques. The most outstanding of these techniques is the implementation of the whole JoonePad interface in xml files. As a matter of fact JoonePad is no real java program but only a series of xml files interpreted with help of SwiXat, another tool created by Paolo Marrone. In order to understand what SwiXat is [37]:

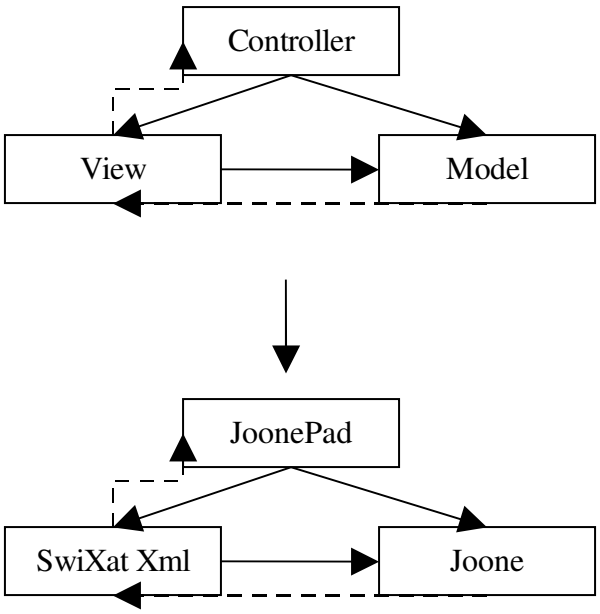
"SwiXAT is the name of the framework built with the aim of providing a powerful User Interface framework based on Swing and XML. It allows new Java applications to be build simply by writing XML parameters and scripted Java code."



"It takes a lot of work to develop a Swing application, laying out and configuring GUI components, and then integrating them with the application functionality. SwiXAT addresses both of these issues by providing a framework based on a complete implementation of the MVC architectural pattern.

The benefits […] are the following:
1. Architectural Correctness : By adopting a true MVC (Model View Controller) based framework, […] clean separation between the view and the application logic.
2. Development Speed : […] use of XML to define the user interface, […]'Code&Test' development style, where the compilation time is reduced to zero.
3. Code Reuse : The net separation between the view and the control logic permits to write reusable modules […]. The developer is naturally induced to modularize […] and write reusable code, minimizing the effort of building new applications or adding new functionality to existing ones."

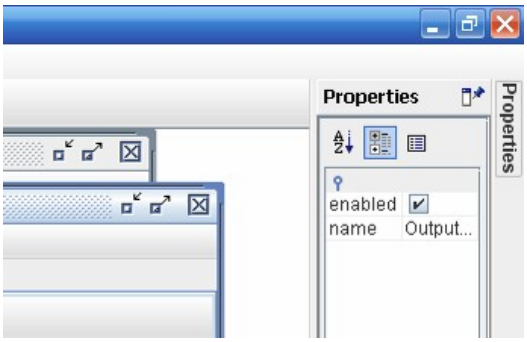
Following the pattern of Model View Controller JoonePad has been implemented respecting the divisions among the three different modules.



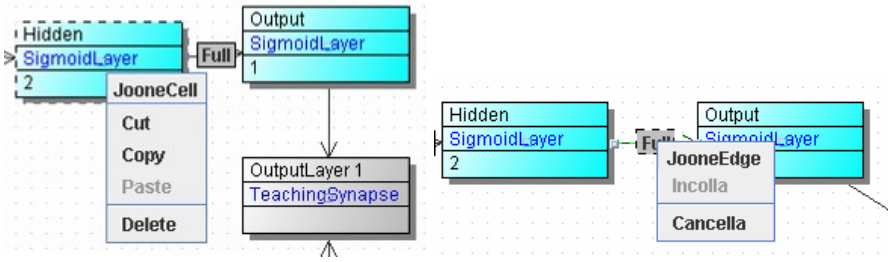
While the model, the machinery that manages the NN simulation, can be identified with Joone, the view is composed of the SwiXat framework and all the XML descriptors that compose the JoonePad Interface. The controller instead has been implemented in standard Java code and is contained in the JoonePad project too.

Beyond the principles of good software development one of the goals in JoonePad was creating a user friendly environment. If it is true that there are lots of NN related programs available out on the net it is also true that their usability leaves a bad taste in the mouth.

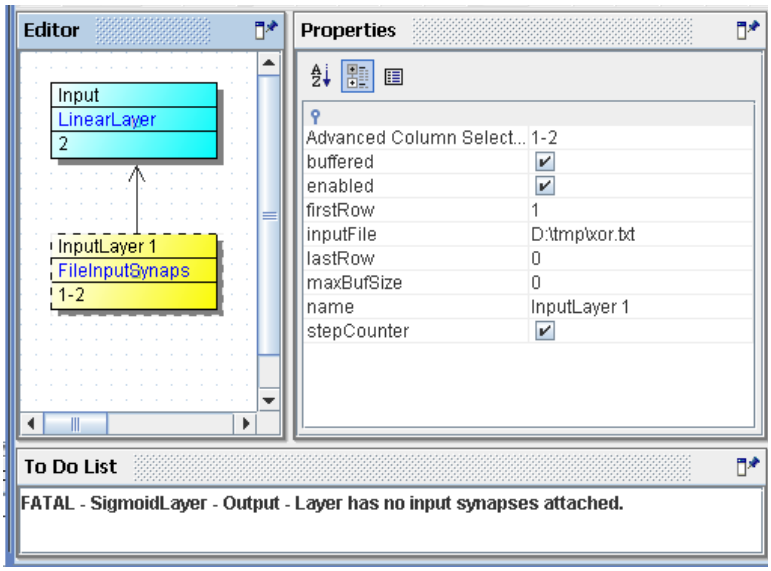
JoonePad is therefore provided with an easy to use tabbed interface that supports auto hiding panels (Flexdock).



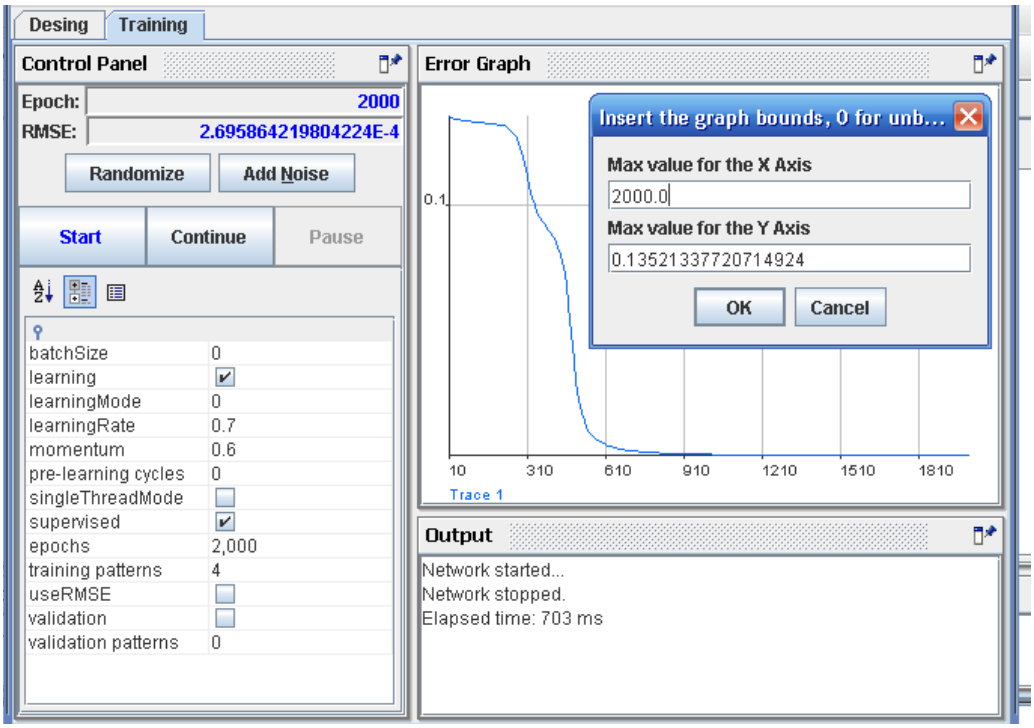
Internationalized interface and context menus. JoonePad also provides classic drag methods on its layers and synapses as well as copy and paste actions on them, all basic features that the normal user expects to find in a user friendly application.



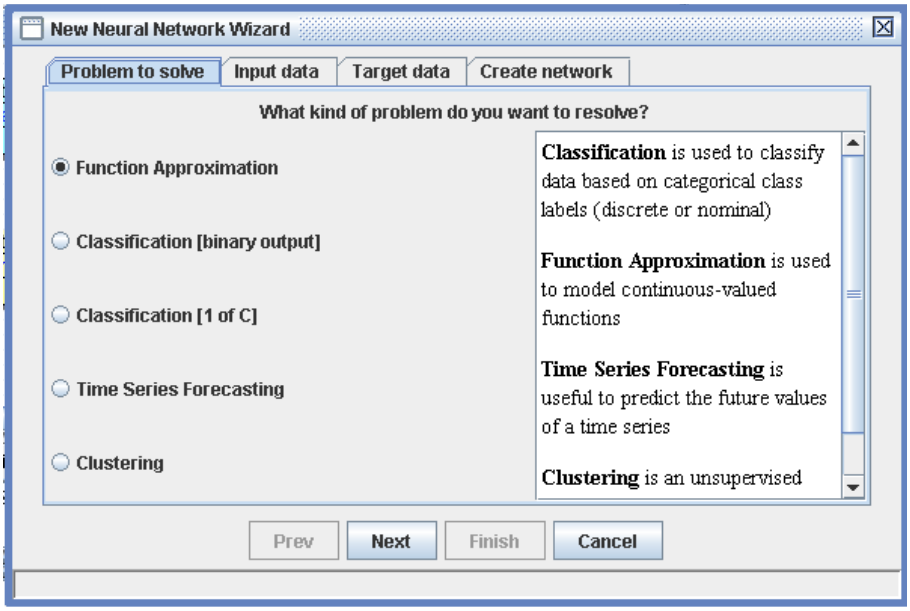
There is even a layer properties panel that provides fast access to all the layer properties and a ToDo list that will help beginners understand the basic steps needed to create a NN.

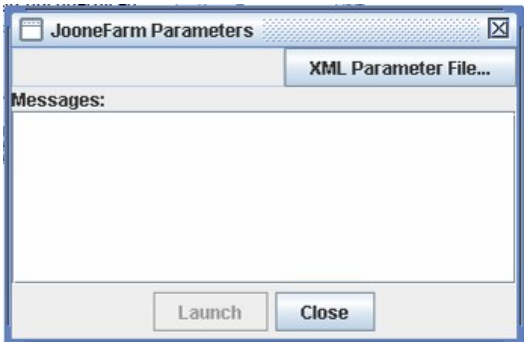


A zoom cable graph panel showing the NN learning is available in the training view.

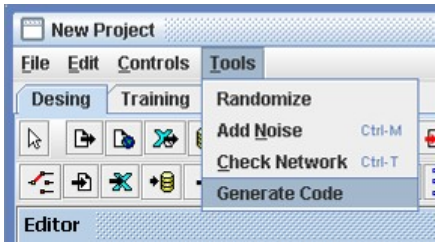


A wizard that guides the user in the creation of general use NNs and a basic control that helps launching JooneFarm will turn helpful for those that just started using Joonepad.





JoonePad even offers the possibility of generating Java source code that simulates the current NN so as to run the current network even outside the editor (e.g. on a PDA).

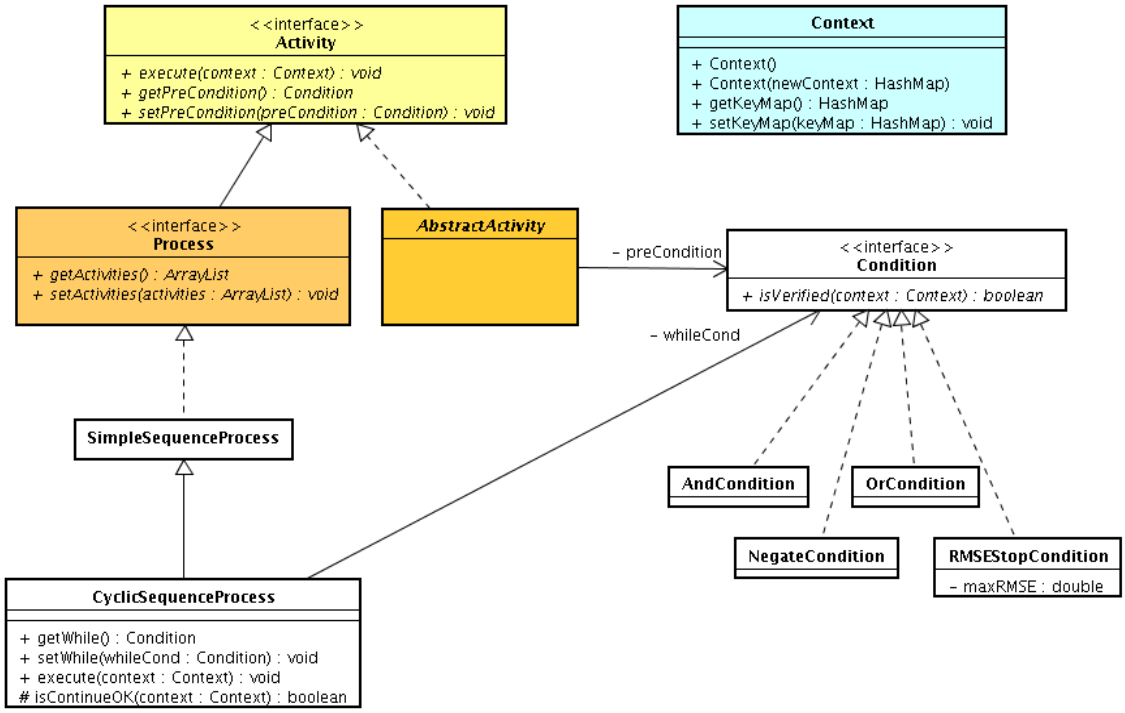


Although the work done in JoonePad took almost half of the whole available Diplomarbeit time this was done under the light of the usability and ease of use principle that inspired the whole work. As soon as the basic NN handling capabilities where present the

author moved the center of the developing effort to the implementation of the Evolutionary Framework. The starting point for the Evolutionary Framework was the JooneFarm manifesto [36]:

"Our dream is to provide a professional framework to permit everyone to build neural networks that could be trained on a lot of common problems of the real life suitable to be resolved by the neural networks (spoken and handwriting recognition, stock quotes forecasting, credit loan assessment, etc.), and then distribute these trained nets to interested users (final users, sales force, etc.) that can use the trained neural networks on their devices (desktops, palmtops, PDA, etc.). Thanks to this framework it would be possible to originate a 'Neural Network Farm' that could fabricate neural networks by training them in parallel on several machines, and finally distribute them to the final users (e.g. for free, or as prepaid service), to 'train once, run anywhere' a neural network without being worried about the HW or SW platform owned by the user."

So the development of the framework was a precondition as well as a parallel goal to the development of a genetic toolkit. The workflow engine of JooneFarm can be resumed in the following diagram.



Activity: this interface represents the main element of the workflow. All the inheriting classes implement the execute method, which is invoked by the framework when the corresponding

workflow's step must be executed. Each Activity has a parameter named preCondition that accepts a class of type Condition, which defines the condition that must be verified in order to execute the Activity. In fact the isVerified method of the defined preCondition (if not null) should be invoked within the execute method. Only if this method returns true, the execution of the Activity is performed.

AbstractActivity: this is the abstract implementation of a generic Activity. Normally all the implemented activities extend this abstract class. It implements a canExecute method that returns the result of the preCondition.isVerified method. In order to implement a new Activity, we need to write in its execute method:

```
public void execute(Context context) throws DTEException {
    if (canExecute(context)) {
        // Write here the activity's execution code
        ...
    }
}
```

Process: while the Activity interface represents a simple step that must be performed in the workflow, the Process interface represents a complex activity, where multiple activities are executed.

SimpleSequenceProcess: this class implements a simple Process where multiple activities are executed in sequence. It's possible to define a list of Activities by the setActivities method. The execute method of this class will invoke the execute method on all the defined Activities. Of course, as the Process interface extends the Activity interface, wherever we can use an Activity, we can also use a Process, hence it's very simple to define very complex processes composed by a sequence of Activities and/or Processes (composed by nested Activities/Processes... and so on).

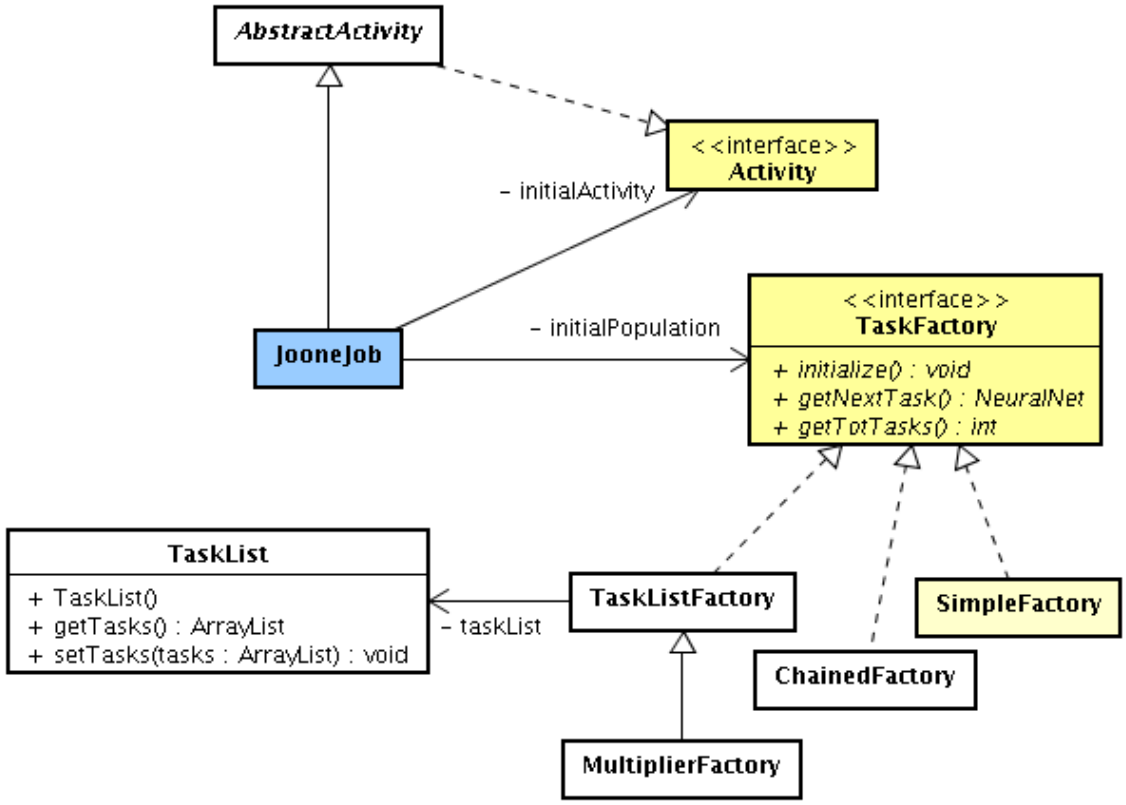
CyclicSequenceProcess: while the SimpleSequenceProcess class executes only once the defined list of Activities that compose the Process, this class permits to define a loop, where the Activities are executed in sequence many times, until a predefined condition (set by the setWhileCondition method) is verified. The loop will stop when the whileCondition will not be satisfied.

Condition: as already described in the previous classes, a mechanism to define some conditions that will be used to modify/control the flow of execution of our workflow is needed. In order to do this, the interface Condition has been defined. Its isVerified method, when invoked, returns true only if the underlying condition is verified.

Context: all the Activities, when executed, receive a context that contains the current state of the workflow. In order to implement a class that is able to contain the workflow's state, the class Context has been declared. It contains a Map – accessible by the get/setKeyMap

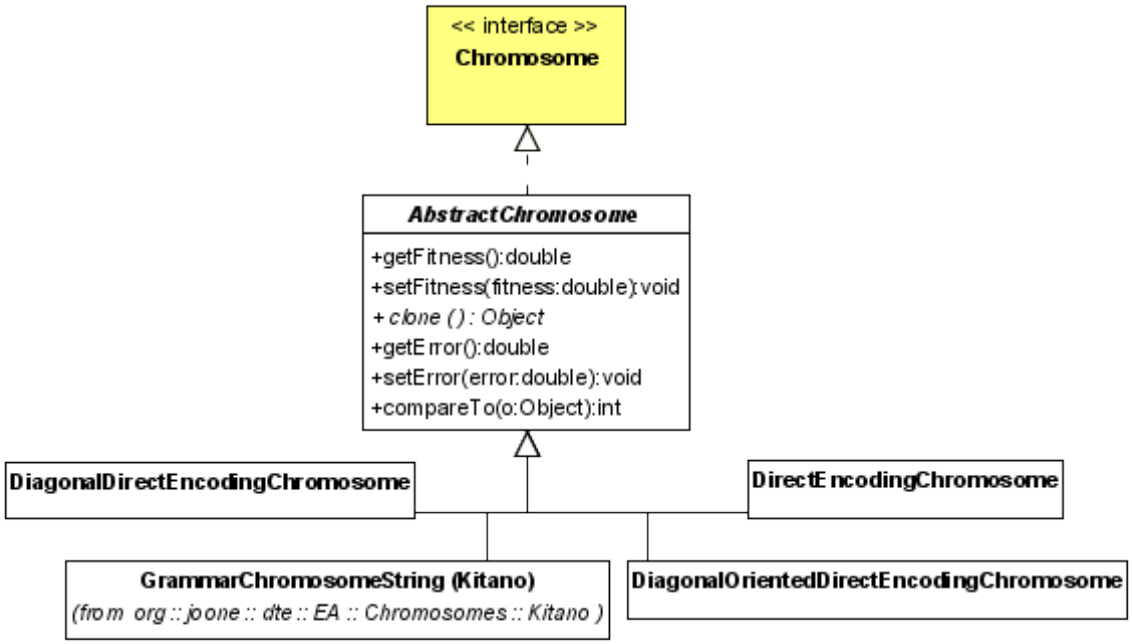
methods - where the user can store/retrieve whatever key/value pair. The Context is passed as parameter both to the Activity.execute and the Condition.isVerified methods.

The execution model of JooneFarm can be resumed with the following UML diagram:



JooneJob: is the main class that implements the processing phase. It owns two main properties: the *initialPopulation*, that represents the initial set of neural networks that must be processed by the defined workflow, and the *initialActivity*, that, as the name says, points to the Activity instance that will be executed. The JooneJob class extends the AbstractActivity.

On top of JooneFarm we can find the evolutionary framework, which is stored in the EA package. The most basic and fundamental point of the framework is doubtlessly the Chromosome package.



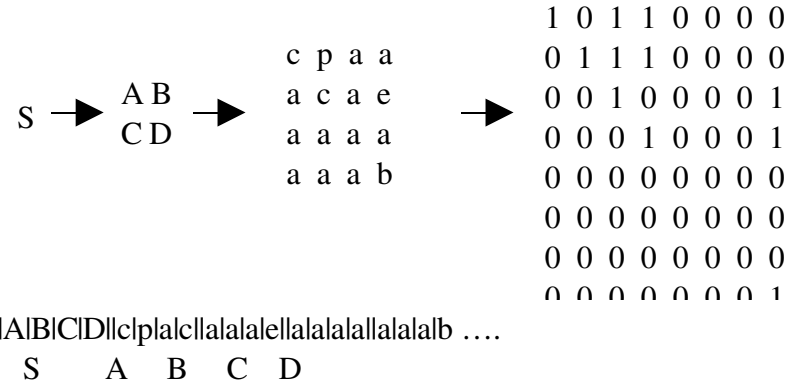
Chromosome: defines a common interface which all different chromosomal implementations must met in order to be accepted by the EA package. Being a chromosome an arbitrary codification of a neural network it must provide the possibility to import a NN into a chromosome as well as to export a chromosome to a NN. This is done through the getPhenotype() and createGenotype(NeuralNet nnet) methods. The developer can then implement a very wide range of chromosomes maintaining the compatibility with the rest of the framework. Being error, mutation and crossover operators heavily coupled with the particular encoding of a chromosome they form also part of this interface.

AbstractChromosome: Implements some basic mechanisms that most chromosomes share. E.g. compareTo(Object o).

DirectEncodingChromosome: This chromosome is the implementation of a standard direct encoding chromosome with a redundant binary connection matrix. It is a basic implementation useful for historic and comparison reasons. The number of mutated alleles is proportional to an adaptive mutationProbability parameter and the size of the maximum constructible network. This renders the mutation step proportional to the search space, prompting faster search but also a lack of fine tuning. Crossover is single point and is executed on the two serialized binary connection matrixes of the parents. As any other chromosome implemented in the EA package it is cloneable, serializable and comparable. This chromosome allows the user to constraint the maximum mutability and size of the

network in order to enhance performance. The closure (Gruau – each chromosome produces a net) of the chromosome is guaranteed with use of phenotypical pruning and in rare cases with the connection of layers too. The genotype remains always untouched from such adjustments.

GrammarChromosome: It is the implementation of a Kitano's grammar encoding chromosome with improved error handling. It shares with the DirectEncodingChromosome the basic mechanism that transforms a NN into a binary connection matrix and vice versa. However, following Kitano's procedure, the connection matrix is encoded in a grammar tree containing all the rules needed for the reconstruction of the connection matrix starting from a fixed non terminal symbol. Given that the maximum NN size is constrained by a user-defined parameter the connection matrix of each chromosome resembles the maximum size in that it has a row for each possible neuron. Consequently the grammar tree is capable of auto adjusting its height; this enables it to encode variable size networks maintaining the defined number of children per node constant. The probabilities of mutation and crossover are handled like in DirectEncodingChromosome but their implementations act upon the grammar encoding tree instead of the binary connection matrix. The only difference that this particular implementation has with the original Kitano encoding is that the grammar rules do not have a header but are simply sequentially encoded, thus preventing the duplication and the elimination of rules. This method ensures that the encoded grammar tree is always valid and interpretable (Gruau - Closure).



DiagonalDirectEncodingChromosome: It is a modification of the direct encoding chromosome following the advice of Mayer [32]. The bits that lay on the diagonal of the matrix indicate the presence of the layer, independently of whether it is connected to other layers or not. Its mutation is specially tailored to treat the diagonal bits in a particular way, the number of modified layers (added or deleted) is equal to:

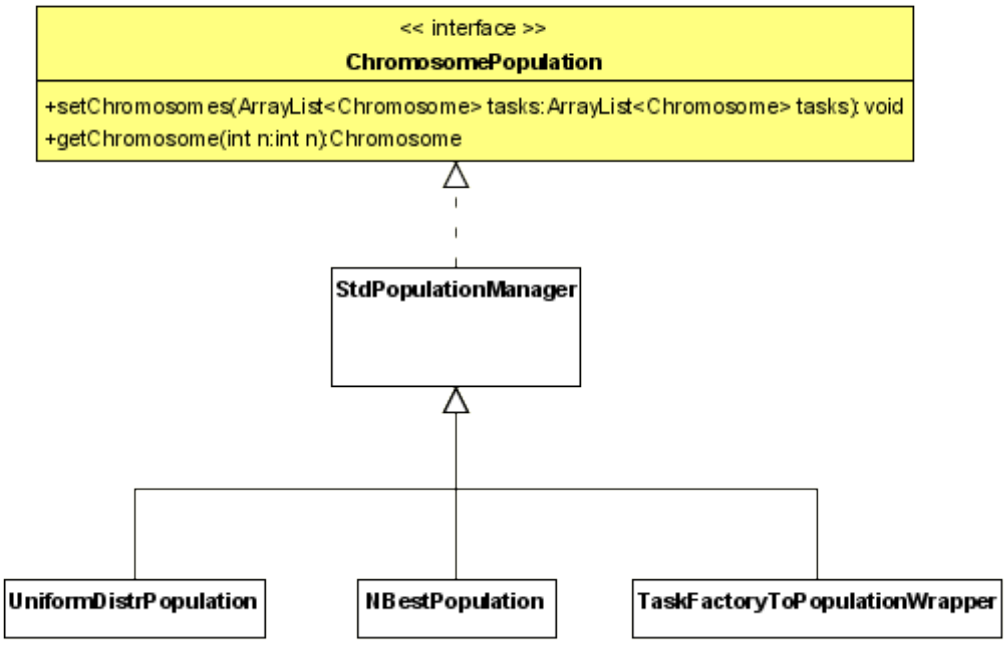
adaptiveMutationStep*Math.abs(generator.nextGaussian()*Math.sqrt(diagonalTrueAlleles));

Such formula is directly derived from Recheberg's adaptive mutation. In fact the second term (Math.abs(…)) is a positive normal distributed value with center on 0 and variance equal to the number of existing layers. The Gaussian (normal) distribution prompts for a thorough search of the architectures that are in the near of the current NN. While the fact that the variance is linked to the NN size decouples mutation from the size of the search space and puts its focus on the size of the network which is being mutated. This means that the number of mutated layers is always a proportional fraction of all those in the NN; a bigger network prompts a bigger mutation, a smaller network a smaller mutation.
The first term of the formula adaptiveMutationStep, is another parameter which is evolved alongside with the NN chromosome. Its main function is the fine tuning of mutation. If all the networks share the same architecture (which is the case of a converged population) the value obtained from the second term of the previously introduced formula is pretty much the same for each network. But let's imagine that all those networks are very large, this means that mutation is very large too. How is it possible to progressively reduce mutation in order to reach that good architecture which is only one neuron away but that has always been sprang over because of the large mutation? adaptiveMutationStep is a parameter evolved on the ground of a network fitness and influences the mutation step. It basically helps networks with higher fitness to set the mutation step for the next generation, further steering the search in the direction of a particular reduced\augmented mutation which proved successful during the previous generations. This kind of mutation is "User free" as there is no parameter that the user should set. Nonetheless it proved much less problem dependant and provided far better results than the previously introduced mutation mechanisms.

DiagonalOrientedDirectEncodingChromosome: Is the expansion of the previous chromosome. It has the peculiarity of using an "oriented" mutation, which is a mutation that either adds or removes elements from the NN but not both. Previous mutations simply flipped the value of a random selected bit on the binary connection matrix, repeating this procedure for the number of bits that had to be mutated. Such mechanisms caused that number of bits changed to one was proportional to the number of bits in the matrix which were set to zero, vice versa the number of bits that mutation set to zero was proportional to the number of bits that were one. This caused the matrix to slowly tend to a state with 50% of the bits set to 1 and 50% set to 0, which coincides (for direct encoding chromosome) with a NN of the maximum size and a 50% chance that each neuron is connected to a neuron of a successive layer. In order to avoid this search bias the DiagonalOrientedDirectEncodingChromosome chooses a direction (either 1 or 0) with a 66% probability of inheriting it from its parent chromosome. The selected direction is used to calculate the number of bits that have to be mutated; first the number of bits in the connection matrix that are equal to the direction is calculated and stored in numberOfDirectedAlleles. Then the amount of bits that have to be mutated is calculated according to the already known formula:

```
adaptiveMutationStep *Math.abs(generator.nextGaussian()*Math.sqrt(numberOfDirectedAlleles+1))
```

Finally only those bits which are not equal to the direction are flipped. This mechanism creates a "mutation trend" where each chromosome adds or removes layers and synapses. The trend is strongly determined by the parent chromosome's trend while the size of mutation adapts to the current NN size.
Since every EA works on populations a mechanism to wrap chromosomes together had to be implemented:



ChromosomePopulation: Is the common interface for a population of chromosomes. It states the mandatory methods for adding and getting single chromosomes as well as groups of them.

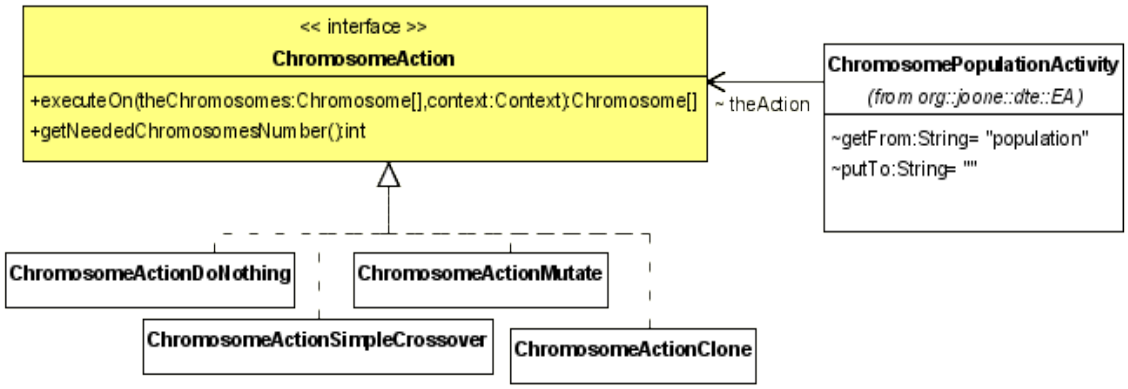
StdPopulationManager: The basic implementation of a chromosome population. Stores all the chromosomes in an ArrayList and retrieves them in sequential order. Chromosome addition can take place in different times. It is generally used to store the current population of chromosomes as they undergo mutation, crossover and other activities.

NBestPopulation: A population that n best chromosomes. It is mostly used during the chromosome selection process.

TaskFactoryToPopulationWrapper: It is a wrapper class that transforms a JooneFarm factory, which returns NNs, into a chromosome population, which returns chromosomes.

UniformDistrPopulation: A population that returns one random chromosome a time selecting it with uniform distribution. This is mainly used for the creation of the child population.

The actions package contains a series of plug-ins that together with the ChromosomePopulationActivity class allow the user to execute operations on a population of chromosomes a time:



ChromosomePopulationActivity: Is an extension of the standard AbstractActivity class. It takes all the chromosomes returned by the getFrom (E.g an NBestPopulation) population and executes the specified activity on them (E.g. mutation) then stores the resulting chromosomes in the population specified by the putTo variable. The exchange of chromosome populations between one ChromosomePopulationActivity and the next is done registering them in the Context of the current JooneJob.

ChromosomeAction: Is the interface implemented by all those classes which represent an action that is intended to be executed on one or more chromosomes a time. The getNeededChromosomesNumber returns the number of chromosomes that the action requires, e.g. mutation needs only one chromosome while crossover employs two or more chromosomes.

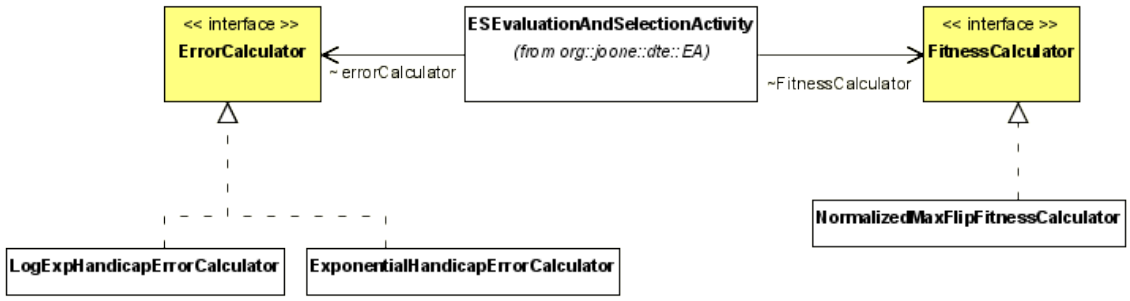
ChromosomeActionClone: Returns a clone of the current chromosome.

ChromosomeActionDoNothing: It performs no action on the chromosomes. Used whenever a ChromosomePopulationActivity should simply transfer the chromosomes of the getFrom ChromosomePopulation to the putTo population (E.g. from a StdPopulationManager to a NBestPopulation).

ChromosomeActionMutate: Mutates the chromosomes.

ChromosomeActionSimpleCrossover: Returns the two chromosomes obtained by calling the simpleCrossover method on the first chromosome using the second chromosome as argument and vice versa. Although simpleCrossover depends on the particular implementation of the chromosome it is always assumed that it is as similar as possible to a single point crossover.

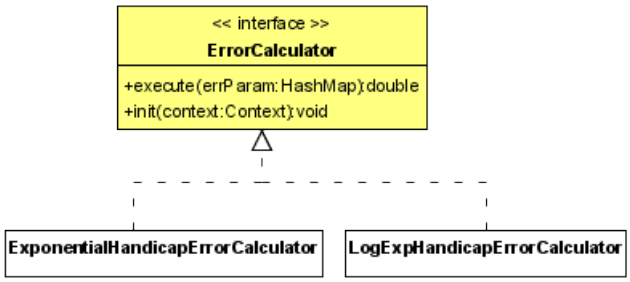
The final part of the EA package is composed by the evaluation and reproduction mechanisms:



ESEvaluationAndSelectionActivity: It is the most complex Activity of the EA package. Its first step consists in the creation of a population following the value of to the ESstyle parameter. Accordingly to Rechenberg's notation the two possible values are "+" and ";" being the default variant the elitist "+" version. Each chromosome of the so obtained population is then trained for a user defined number of learning sessions, each time for an arbitrary amount cycles. Statistical information about the chromosome such as: average RMSE, NN size and mutation, amount among others, is collected and inserted in the JooneJob context for fitness calculation, error determination and charting purposes. Part of such data is passed together with the current training network to a class that implements the errorCalculator interface, which sets the error value for the relevant chromosome. The second step consists in calling a FitnessCalculator implementing class in order to assign a fitness

value to each chromosome. The best chromosome is then stored and compared with "theChampion" the best chromosome ever found throughout the current JooneJob. Setting the appropriate parameters, the current population, the parent population, the generation's best chromosome and the champion can be saved to disk in JoonePad compatible format, allowing posterior inspection of the EA behavior. Further options include the possibility to re-encode the trained network in the chromosome so as to preserve the weight values obtained during training (requires weight encoding chromosomes).

NormalizedMaxFlipFitnessCalculator: It is a slightly adapted version of the `NormalizedMaxFlipFitness` function presented in the Traveling Salesman Problem of chapter 4. Its main function is the storage of the fitness value for each chromosome. Fitness is interpreted as the probability [0 ; 1] that a chromosome has to be choosen for reproduction and it is calculated on the base of the chromosome error and the error of all the other chromosomes in the population. A threshold value has been included so that all those chromosomes which are 100 times worse than the best chromosome automatically receive a 0 fitness value. This might obviously restrict diversification so the threshold is user adjustable, nonetheless the author points out that allowing the reproduction of chromosomes which are more than 100 times worse might maintain chromosomal diversity but isn't a convergence friendly technique.



ExponentialHandicapErrorCalculator: It is the basic error calculator and uses the following formula:

$$E*H^S$$

Where:

E = (AverageRMSE*x + LowestRMSE(1-x)) with x [0 ; 1]
S = Number of neurons in the NN.
H = User defined handicap parameter generally in the range [1,5 ; 3]

The handicap parameter is interpreted as following:

If H is 2 for each neuron added to a network its RMSE must become half in order to maintain the same error level. If H is n for each neuron added to a network its RMSE must become n times smaller in order to maintain the same error level. Let's mathematically analyze how the error of two different networks with the same performance but different sizes compare:

$$\frac{E_1}{E_2} = \frac{RMSE * H^{S_1}}{RMSE * H^{S_2}} = H^{S_1 - S_2}$$

This exponential growing error has been adopted as a linear growing error does not respect the rate of time and resources consumption that a growing neural network has. We expect a neural network that doubles its performance to be much less than twice as big.

RMSE	Size	Error	networkSizeHandicap
8,51E-04	2	0,0034033	2,00
8,51E-04	3	0,0068067	
8,51E-04	4	0,0136133	
8,51E-04	5	0,0272267	
8,51E-04	6	0,0544533	
8,51E-04	7	0,1089067	
8,51E-04	8	0,2178134	
8,51E-04	9	0,4356268	
8,51E-04	10	0,8712535	
8,51E-04	11	1,7425071	
8,51E-04	12	3,4850142	
8,51E-04	13	6,9700283	
8,51E-04	14	13,9400566	
8,51E-04	15	27,8801132	
8,51E-04	16	55,7602264	
8,51E-04	17	111,5204528	
8,51E-04	18	223,0409056	
8,51E-04	19	446,0818112	
8,51E-04	20	892,1636224	
8,51E-04	30	913575,5493394	
8,51E-04	31	1827151,0986788	

LogExpHandicapErrorCalculator: It is the evolution of the previous error calculation. The problem that ExponentialHandicapErrorCalculator presents is that when a population

contains networks substantially different sizes (~20 times bigger) the difference in error values becomes extreme. This can often mislead fitness calculation, especially with normalized fitness functions, as approximation errors tend to blend the difference among the best networks if they are compared with very bad ones. E.g. :

RMSE	Size	Error
8,51E-04	2	0,0034033
8,51E-04	4	0,0136133
8,51E-04	31	1827151,0986788

Although the two first networks (2 and 4 neurons) present a substantial difference in their error, the last network (31 neurons) has such a high error value that a normalization of the error "squeezes" the first two networks at the same quality level. Such approximation errors are very damaging for evolution because seriously retard convergence and often tend discard very good solutions (in this case by choosing size 4 instead of 2).

In order to avoid such side effects the error calculation formula has been modified to:

$$E*H^{\mathrm{Log}_{\sqrt{2}}S}$$

Where:

E = (AverageRMSE*x + LowestRMSE(1-x)) with x [0 ; 1]
S = y*Number of neurons + z*Number of synapses. Generally y = 1, z = 0,1
H = User defined handicap parameter generally in the range [1,5 ; 3]

The first change consists in the fact that synapses are considered for the determination of the network size, directing the search towards even more simplified topologies.
The second change is the use of a logarithm in the calculus of the handicap exponent. The choice of a logarithm with a root square two base is because it tends to maintain the values in the near of 2 (which is the minimum NN size) while strongly curbs extreme values.

RMSE	Size	Error	$\mathrm{Log}_{\sqrt{2}}S$
8,51E-04	2	0,0034033	2,0000000
8,51E-04	3	0,0076575	3,1699250
8,51E-04	4	0,0136133	4,0000000
8,51E-04	5	0,0212708	4,6438562
8,51E-04	6	0,0306300	5,1699250
8,51E-04	7	0,0416908	5,6147098
8,51E-04	8	0,0544533	6,0000000
8,51E-04	9	0,0689175	6,3398500
8,51E-04	10	0,0850834	6,6438562

8,51E-04	11	0,1029509	6,9188632
8,51E-04	12	0,1225200	7,1699250
8,51E-04	13	0,1437909	7,4008794
8,51E-04	14	0,1667634	7,6147098
8,51E-04	15	0,1914375	7,8137812
8,51E-04	16	0,2178134	8,0000000
8,51E-04	17	0,2458909	8,1749257
8,51E-04	18	0,2756701	8,3398500
8,51E-04	19	0,3071509	8,4958550
8,51E-04	20	0,3403334	8,6438562
8,51E-04	30	0,7657502	9,8137812
8,51E-04	31	0,8176510	9,9083926
8,51E-04	40	1,3613337	10,6438562
8,51E-04	50	2,1270838	11,2877124
8,51E-04	60	3,0630007	11,8137812
8,51E-04	70	4,1690843	12,2585660
8,51E-04	80	5,4453346	12,6438562
8,51E-04	90	6,8917516	12,9837062
8,51E-04	100	8,5083353	13,2877124
8,51E-04	200	34,0333413	15,2877124
8,51E-04	201	34,3745256	15,3021034
8,51E-04	512	223,0409056	18,0000000
8,51E-04	1024	892,1636224	20,0000000
8,51E-04	2048	3568,6544896	22,0000000

This clearly allows a sufficiently wide range of network sizes to be included in the same population.

6.4- Writing a JooneJob

The previously introduced JooneJobs are defined within XML files which are written according to the Spring framework syntax.

Here is an example of a simple workflow composed of two activities – ActivityA and ActivityB - that must be executed in sequence:

```
<bean id="myWorkflow" class="org.joone.dte.SimpleSequenceProcess ">
   <property name="activities">
      <list>
         <ref bean="ActivityA"/>
         <ref bean="ActivityB"/>
      </list>
```

```
   </property>
</bean>
```

Of course the ActivityA and ActivityB references must point to two beans (declared in the same XML file) that implement either the Activity or the Process interfaces.

If we wish to use this workflow within a Job, we must declare a JooneJob object:

```
<bean id="myJob" class="org.joone.dte.JooneJob">
   <property name="initialActivity">
      <ref bean="myWorkflow"/>
   </property>
   ...
</bean>
```

In order to complete the above declaration, we should also to declare the JooneJob's ititialPopulation property, by referencing a class that implements the TaskFactory interface.

As the TaskFactory implementations are the same of the old DTE, you can read the DTE Guide to learn more about them.

Of course we can define a more complex workflow, by nesting several kinds of Activities/Processes, as in the following example:

```
<bean id="myComplexWorkflow" class="org.joone.dte.SimpleSequenceProcess ">
   <property name="activities">
      <list>
         <bean class="org.joone.dte.CyclicSequenceProcess">
            <property name="activities">
                  <list>
                     <ref bean="ActivityA"/>
                     <ref bean="ActivityB"/>
                  </list>
            </property>
            <!-- Loops until wileCondition is False -->
           <property name="whileCondition">
             <bean class="org.joone.dte.control.AndCondition">
                  <property name="leftCondition">
                        <ref bean="ConditionA"/>
                  </property>
                  <property name="rightCondition">
                        <ref bean="ConditionB"/>
                  </property>
              </bean>
            </property>
         </bean>
         <ref bean="ActivityC"/>
      </list>
   </property>
</bean>
```

Here we have defined a SimpleSequenceProcess composed by two Activities: the first one is a CyclicSequenceProcess where the ActivityA and ActivityB will be executed several times, until (ConditionA AND ConditionB) become false (note the AndCondition bean declaration within the 'whileCondition' property).
Once the CyclicProcess has finished, the ActivityC will be executed (only once).
As the reader can see, both the Activities/Processes and the Conditions can be nested and combined in several manners, in order to build whatever complex workflow is needed.

Chapter 7

Final considerations

7.1- Conclusion and outlook

It is clear that in the field of EAs there is substantial space for further improvement. Nonetheless the application of EAs as optimization techniques is justified by a sufficient theoretical basis derived from the investigation of their basic mechanisms (e.g. the building block hypotheses). And even though their correctness has been demonstrated by means of the schema theorem it is always important, like for any other search method, to evaluate their actual usefulness in relation to the particular nature of the problem that has to be solved. The efficiency of an EA might otherwise result equal to that of an enumerative search (No free lunch theorem).

Although in the praxis EAs are not as widely known or used as other optimization techniques, their peculiarities render them a valid alternative that should always be taken into consideration. Judging from the different implementations found in the literature, it can be said that even if a minimal algorithm, which comprises mutation, selection and reproduction, is at the basis of almost every implementation, there is no universally accepted standard algorithm. Generally the detailed features of an EA are left to the developer's choice, his problem knowledge, his experience and his imagination. In this circumstances the work of Gruau might deserve some particular attention, because if it is true that the principles he exposed were not presented in the most rigorous way, it is also true that the idea of establishing implementation independent quality principles for chromosomal codifications has been seriously neglected in the literature despite the evidence of repeating patterns in the different implementations.

At this point the reader should have acquired a new perspective about the optimization possibilities of weights, topologies, training data sets and parameters of a neural network as well as the problems he might face when dealing with evolutionary algorithms. It should be now possible for him to make a critic review of the different implementations, distinguishing the biological plausibility of and approach from the actual advantages it offers.

The reader has also been introduced to the user friendly, powerful and extensible frameworks of the Joone project, thus enabling him to further experiment with neural networks and improve his practical knowledge in the interesting field of genetic algorithms.

The application possibilities in this field are almost endless. Just to make an example, a very interesting experiment would be the validation of the biological theory of evolution with help of evolutionary computation. Given that evolution is still considered a theory and no absolute truth, it would be interesting to experiment with evolved robot behavior more in depth, especially under the light of the promising literature examples previously analyzed.

The last word wants to be an exhortation to experiment the mix of neural networks and EAs, since with a little bit of experience everybody develops a sort of “green thumb” for chromosomes, fitness functions and all the correlated parameters. It is an amazing feeling to see an algorithm at work which automatically finds a neural network capable of solving a problem.

Appendix A

Examples' Code

- ***– The genetic traveling salesman problem***

Included in the alleged file GeneticTSP.java

```
001 import java.util.Random;
002
003 public class GeneticTSP {
004
005
006     static int PopulationSize = 10;
007     static int NumberOfCities = 10;
008     static int Generations = 1000;
009     static double ChromoMutationProb = NumberOfCities * 0.01;
010
011     static int[][] DistancesArray = new int[NumberOfCities][NumberOfCities];
012     static int[][] ChromoArray = new int[PopulationSize][NumberOfCities];
013     static int[] ChromoLength = new int[PopulationSize];
014     static double[] ChromoPercentage = new double[PopulationSize];
015     static double[] ChromoFitness = new double[PopulationSize];
016     static int TotalLength = 0;
017
018
019     public static void main(String[] args) {
020         initDistances();
021         initPopulation();
022         NormalizedMaxFlipFitness();
023
024         //**Generational cycle**\\
025
026         do {
027             int i = selectElement();
028             int j = selectElement();
029
030             int[] ChildChromo = crossover(ChromoArray[i],ChromoArray[j]);//creates a chr
omosome by crossing over
031             mutation(ChildChromo);
032
033             //Replace and Update
034             Random generator = new Random();
035             int Substituted =  (int)(generator.nextDouble()* PopulationSize);
036
037             if (Generations%10 ==0) {
038                 //printChromosomes();//Display the chromosomes
039                 System.out.print(i + " + " + j + " -> ");printChromosome(ChildChromo);
040                 int Best = findBest();
041                 System.out.println(" Mean Length " + TotalLength/PopulationSize + " Tot
" + TotalLength + " Best for the " + Generations + " generation is n°" + Best + " with " + C
hromoLength[Best]);
```

```
042                 System.out.println("----------------------------------------");
043             }
044 
045             TotalLength = TotalLength - ChromoLength[Substituted]; //remove old length
046             ChromoArray[Substituted] = ChildChromo;
047             ChromoLength[Substituted] = calculateChromoLength(ChildChromo);
048             TotalLength = TotalLength + ChromoLength[Substituted]; //add new length
049             calculateChromoPercentage();
050             NormalizedMaxFlipFitness();  //Update Chromo fitness
051 
052             Generations--;
053         } while(Generations >= 0);
054     }
055 
056 
057 
058     private static void printChromosomes() {
059         for (int i = 0; i<= PopulationSize -1 ;i++){
060             System.out.print(i + ") ");
061             printChromosome(ChromoArray[i]);
062             ChromoLength[i] = calculateChromoLength(ChromoArray[i]);
063             System.out.print(" Sum  = " + ChromoLength[i]);
064             System.out.println(" Fit  = " + ChromoFitness[i]);
065         }
066     }
067 
068     private static int findBest() {
069         int Best = 0;
070         int min = ChromoLength[0];
071         for (int i=1; i<ChromoLength.length; i++) {
072             if (ChromoLength[i] < min) {
073                 min = ChromoLength[i];
074                 Best = i;
075             }
076         }
077         return Best;
078     }
079 
080     private static void calculateTotalLength() {
081         TotalLength = 0;
082         for (int i = 0; i<= PopulationSize -1 ;i++){
083             TotalLength = ChromoLength[i];
084         }
085     }
086 
087     private static void mutation(int[] childChromo) {
088         Random generator = new Random();
089         boolean mutate =  (ChromoMutationProb >= generator.nextDouble());
090         if (mutate) {
091             int Allele1 = (int)(generator.nextDouble()* (NumberOfCities));
092             int Allele2 = (int)(generator.nextDouble()* (NumberOfCities));
093             int tmpValue = childChromo[Allele1];
094             childChromo[Allele1] = childChromo[Allele2];
095             childChromo[Allele2] = tmpValue;
096         }
097         //System.out.println(" Mutated? = " + mutate);
098     }
099 
100 
101     /**
102      *
103      * @param ch1
104      * @param ch2
105      * @return The crossovered child
```

```
106      */
107     private static int[] crossover(int[] ch1, int[] ch2) {
108 
109         Random generator = new Random();
110 
111         //randomly chooses one of both children
112         int[] RightParentChromo, LeftParentChromo;
113         int[] ChildChromo = new int[NumberOfCities];
114 
115         if (generator.nextBoolean()) {
116             LeftParentChromo = ch1;
117             RightParentChromo = ch2;
118         } else {
119             LeftParentChromo = ch2;
120             RightParentChromo = ch1;
121         }
122 
123         boolean[][] connectedTo = new boolean[NumberOfCities][NumberOfCities];//stores t
he nodes connected to the index node
124         int[] connectionsNumber = new int[NumberOfCities];
125 
126         for (int h = 0; h < NumberOfCities; h++){//creates the connections matrix
127             int nexth = h + 1; if (nexth == NumberOfCities) nexth = 0;
128             int previoush = h - 1;if (previoush == -1) previoush = NumberOfCities - 1;
129             //for left parent
130             connectedTo[LeftParentChromo[h]][LeftParentChromo[nexth]] = true;
131             connectedTo[LeftParentChromo[h]][LeftParentChromo[previoush]] = true;
132             //for right parent
133             connectedTo[RightParentChromo[h]][RightParentChromo[nexth]] = true;
134             connectedTo[RightParentChromo[h]][RightParentChromo[previoush]] = true;
135         }
136         for (int h = 0; h < NumberOfCities; h++){//store the number for each node
137             int sumConnections = 0;
138             for (int i = 0; i < NumberOfCities; i++){
139                 if (connectedTo[h][i]) sumConnections++;
140             }
141             connectionsNumber[h] = sumConnections;
142         }
143 
144         //First node is random from parents
145         ChildChromo[0] = LeftParentChromo[0];
146 
147         for (int h = 0; h < NumberOfCities; h++){//erase the node from the list
148             if (connectedTo[h][ChildChromo[0]]) {
149                 connectedTo[h][ChildChromo[0]]= false;
150                 connectionsNumber[h]--;
151             }
152         }
153         //printArray(connectedTo);
154         //end First node
155 
156         for( int currentAllele = 0;currentAllele < NumberOfCities - 1;currentAllele++){/
/for all but the last allele
157             int minConnections = NumberOfCities + 1;
158             for (int i = 0; i < NumberOfCities; i++){//calculate the node connected to c
urrent allele that has min connections
159                 if (connectedTo[ChildChromo[currentAllele]][i]){
160                     if (connectionsNumber[i]< minConnections) minConnections = connectio
nsNumber[i];//the number of connection of the node with less connections
161                 }
162             }
163 
```

```
164             int rnd,rnd2;
165             if (connectionsNumber[ChildChromo[currentAllele]] == 0){//if no connection f
rom current node is available
166                 do{
167                     //printArray(connectedTo);
168                     rnd = (int)(generator.nextDouble()* NumberOfCities); //choose a rand
om node not already used
169                     rnd2= (int)(generator.nextDouble()* NumberOfCities);
170                 } while (connectedTo[rnd2][rnd] == false);
171             } else {//randomly choose one of the nodes with min connections
172                 do{
173                     rnd = (int)(generator.nextDouble()* NumberOfCities);
174                 } while (!((connectedTo[ChildChromo[currentAllele]][rnd])
175                         && (connectionsNumber[rnd]== minConnections)));//random among
those with the min number of connections
176             }
177             ChildChromo[currentAllele + 1] = rnd;
178             //erase all true for rnd in connectedTo
179             for (int h = 0; h < NumberOfCities; h++){//erase the node from the list
180                 if (connectedTo[h][rnd]) {
181                     connectedTo[h][rnd]= false;
182                     connectionsNumber[h]--;
183                 }
184             }
185             //System.out.println();
186             //printArray(connectedTo);
187         }
188         return ChildChromo;
189     }
190
191
192     private static void printArray(boolean[][] connectedTo) {
193         for (int j = 0; j< NumberOfCities;j++){
194             System.out.print(j + ") ");
195             for (int i = 0; i< NumberOfCities;i++){
196                 if (connectedTo[j][i]) System.out.print("1 "); else System.out.print("0
");
197             }
198             System.out.println();
199         }
200     }
201
202
203
204     private static int indexOf(int i, int[] ParentChromo) {
205         for (int j = 0; j<=ParentChromo.length -1;j++){
206             if (ParentChromo[j] == i) return j;
207         }
208         return -1; //not found
209     }
210
211
212
213     public static void initDistances(){
214         System.out.println("Table of distances");
215         Random generator = new Random();
216         int startPoint = 0;
217         for (int i = 0; i<=NumberOfCities -1;i++){
218             for (int j = startPoint; j<= NumberOfCities -1;j++){
219                 if (i == j) DistancesArray[i][j] = 0;
220                 else {
221                     DistancesArray[i][j] =  1 + (int)(generator.nextDouble()*100);
```

```
222                     DistancesArray[j][i] =  DistancesArray[i][j];
223                 }
224             }
225             startPoint++;
226         }
227 
228 
229         for (int i = 0; i<=NumberOfCities -1;i++){
230             for (int j = 0; j<= NumberOfCities -1;j++){
231                 System.out.print(DistancesArray[i][j] + " ");
232             }
233             System.out.println();
234         }
235         System.out.println("-----------------------------------------");
236     }
237 
238     public static void initPopulation(){
239         System.out.println("First Population");
240         Random generator = new Random();
241         int tmpAllele;
242         for (int i = 0; i<= PopulationSize -1 ;i++){
243             System.out.print(i + ") ");
244             for (int j = 0; j<=NumberOfCities -1;j++){
245                 do {
246                     tmpAllele = (int)(generator.nextDouble()* NumberOfCities);
247                 } while (isNotValid(tmpAllele,ChromoArray[i], j));
248                 ChromoArray[i][j] = tmpAllele;
249             }
250             printChromosome(ChromoArray[i]);
251             ChromoLength[i] = calculateChromoLength(ChromoArray[i]);
252             System.out.println(" Sum  = " + ChromoLength[i]);
253             TotalLength = TotalLength + ChromoLength[i];
254         }
255         System.out.println("Best = " + ChromoLength[findBest()] + " Total Length = " + T
otalLength);
256         System.out.println("-----------------------------------------");
257         calculateChromoPercentage();
258     }
259 
260     public static void printChromosome(int[] Chromosome){
261         for (int j = 0; j<=NumberOfCities -1;j++){
262             System.out.print( Chromosome[j] + " ");
263         }
264     }
265 
266     /**
267      *
268      * @param chromo
269      * @return The fitness of chromo
270      */
271     public static int calculateChromoLength(int[] chromo) {
272         int tmpFitness = 0;
273         for (int j = 0; j<=NumberOfCities - 2;j++){
274             tmpFitness = tmpFitness  + DistancesArray[chromo[j]][chromo[j+1]];
275         }
276         tmpFitness = tmpFitness  + DistancesArray[chromo[9]][chromo[0]]; //from last loc
ation to first
277         return tmpFitness;
278     }
279 
280     /**
281      * @param value The new value to be inserted
282      * @param chromo The chromosome
283      * @return Whether the value is acceptable and can be inserted in chromo
```

```
284      */
285     public static boolean isNotValid(int value, int[] chromo,int length){
286         for (int i = 0; i<length ;i++){
287             if (chromo[i] == value) return true;
288         }
289         //if the value is not present in the chromosome it can be inserted
290         return false;
291     }
292 
293     private static int selectElement() {
294         Random generator = new Random();
295         double rnd = generator.nextDouble();
296         double CurrentProbability = 0;
297 
298         //System.out.println(rnd);
299         for (int i = 0;i < PopulationSize ;i++ ){
300             CurrentProbability = CurrentProbability + ChromoFitness[i];
301             if (CurrentProbability >= rnd) return i;
302         }
303         return PopulationSize - 1;
304     }
305 //---------------------------------------------------------------------------------

306     private static void NormalizedMaxFlipFitness() {//Less variance
307         double totalFitness = 0;
308         double scalingFactor = 0.99; // compresses or expands the distr. curve, 0.99 avo
ids prob = 0 for the smallest chromo
309 
310         for (int i = 0; i<PopulationSize;i++){
311             ChromoFitness[i] = max(ChromoPercentage) -
(ChromoPercentage[i]*scalingFactor);
312             totalFitness = totalFitness + ChromoFitness[i];
313         }
314         //Normalization
315         for (int i = 0; i<PopulationSize ;i++){
316             ChromoFitness[i] = (double)ChromoFitness[i] /(double)totalFitness;
317         }
318     }
319     /**
320      * Calculates the fitness of all chromosomes
321      */
322     private static void OneOverXFitness() {//More aggressive
323         double totalFitness = 0;
324 
325         for (int i = 0; i < PopulationSize; i++){
326             ChromoFitness[i] = 1/ChromoPercentage[i];
327             totalFitness = totalFitness + ChromoFitness[i];
328         }
329         //Normalization
330         for (int i = 0; i < PopulationSize; i++){
331             ChromoFitness[i] = (double)ChromoFitness[i] /(double)totalFitness;
332         }
333     }
334 //---------------------------------------------------------------------------------
335 
336     /**
337      * Stores in ChromoPercentage the percentual length
338      * of the chromosome in comparison to the total
339      */
340     private static void calculateChromoPercentage() {
341         for (int i = 0; i< PopulationSize; i++){
342             ChromoPercentage[i] = ((double)ChromoLength[i]/(double)TotalLength);
```

```
343         }
344     }
345 
346     public static double max(double[] t) {
347         double maximum = t[0];
348         for (int i=1; i<t.length; i++) {
349             if (t[i] > maximum) {
350                 maximum = t[i];
351             }
352         }
353         return maximum;
354     }
355 
```

Bibliography

[1] R. Rojas, Neural Networks, *A systematic Introduction*
Springer (1996)

[2] R. Beale, T. Jackson, *Neural Computing, an introduction*
IOP Publishing Ltd (1990)

[3] E. Schöneburg, F. Heinzmann, S. Feddersen, *Genetische Algorithmen und Evolutionsstrategien*
Addison Wesley (1994)

[4] D. Whitley, *A Genetic Algorithm Tutorial* [Online]
Colorado University (1994)
Available: http://www.cs.colostate.edu/~genitor/MiscPubs/tutorial.pdf

[5] I. Rechenberg, *Evolutionsstrategie '94*
Frommann-holzboog (1994)

[6] I. Rechenberg, „Biologische Evolution auf dem experimentellen Prüfstand“ in *Vorlessung Bionik I im Winter 00/01* [Online]
TU Berlin (2000)
Available: www.bionik.tu-berlin.de/institut/skript/bibu2.pdf

[7] I. Rechenberg, „PowerPoint-Folien zur Vorlesung „Evolutionsstrategie I“ *(WS 05/06)* [Online]
TU Berlin (2000)
Available: http://www.bionik.tu-berlin.de/institut/skript/Fo05ES1.htm

[8] John R. Koza, *Genetic Programming - On the Programming of Computers by Means of Natural Selection*
A Bradford Book, MIT Press (1998)

[9] John R. Koza, *Genetic Programming as a Means for Programming Computers by Natural Selection* [Online]
Stanford University (1994)

Available:http://citeseer.ist.psu.edu/rd/51610382%2C145594%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/1987/http:zSzzSzwww.genetic-programming.comzSzSCJ.pdf/koza94genetic.pdf

[10]P. Marrone, *The Joone Complete Guide* [online]
The Joone Project (2005)
Available:http://prdownloads.sourceforge.net/joone/JooneCompleteGuide.pdf?download

[11]F. Nerdinger, *Grundlagen des Verhaltens in Organisationen*
Kohlhammer (2003)

[12]Kattman, „Gene & Evolution“ in *Unterricht Biologie - heft 260*
(? 2000)

[13]Adam Marczyk, *“A brief history of GAs”* in *Genetic Algorithms and Evolutionary Computation* (2004) [online]
Available: http://www.talkorigins.org/faqs/genalg/genalg.html#history

[14]John H. Holland, *Genetic Algorithms* (2004) [online]
Available: http://www.arch.columbia.edu/DDL/cad/A4513/S2001/r7/

[15]John H. Holland, *Adaptation in Natural and Artificial Systems* A
Bradford Book, MIT Press(2004)

[16]Mitchell Melanie, *An Introduction to Genetic Algorithms* A
Bradford Book, MIT Press(1999)

[17]Schaffer, Whitley, Eshelman , *Combinations of Genetic Algorithms and Neural Networks: A survey of the state of the Art* in COGANN-92 Int. Workshop on Combinations of Genetic Algorithms and Neural Networks pp. 1-37 (1993)

[18]Xin Yao, *Evolving Artificial Neural Networks* in
Proceedings of the IEEE vol. 87 n° 9 pp. 1423-1439 (1999)

[19]D. Montana and L. Davis, *Training Feedforward Neural Networks Using Genetic Algorithms* in Proceedings of the International Joint Conference on Artificial Intelligence, pp. 762-767 (1989)

[20]Karsten Weicker, *Evolutionäre Algorithmen* B.G. Teubner (2002)

[21]Sung-Bae Cho and Katsunori Shimohara, *Grammatical Development of Evolutionary Modular Neural Networks* in Simulated evolution and Learning, pp. 413-420 Springer (1998)

[22]Seong-Whan Lee, *Off-Line Recognition of Totally Unconstrained Handwritten Numerals Using Multilayer Cluster Neural Network* in IEEE Transactions Pattern Analysis Machine Intelligence, vol 18 pp. 648-652 (1996)

[23]V. W. Porto, D. B. Fogel and L. J. Fogel, *Alternative neural network training methods* in IEEE Expert vol. 10 n°3 pp. 16-22 (1995)

[24]Randall S. Sexton, Robert E. Dorsey, John D. Johnson, *Toward Global Optimization Of Neural Networks: A Comparison Of The Genetic Algorithm And Backpropagation* (1998) [online] Available: http://citeseer.ist.psu.edu/cache/papers/cs/11136/http:zSzzSzwww.bsu.eduzSzclasseszSzsextonzSzgabpdss.pdf/sexton98toward.pdf

[25]Kenneth O. Stanley, Risto Miikkulainen, *Evolving Neural Networks through Augmenting Topologies* in Evolutionary Computation Volume 10, Number 2, The MIT Press Journals (2002)

[26]D.W. Pearson, N.C. Steele, R.F. Albrecht , *Artificial Neural Nets and Genetic Algorithms*, proceeding of the international conference in Alès, Springer (1995)

[27]Hiroaki Kitano*, Neurogenetic learning: an integrated method of designing and training neural networks using genetic algorithms* in Physica D 75 pp. 225-238 Elsevier Science B .V. (1994)

[28]Steven van Dirk, Dirk Thierens, *On the use of a non redundant encoding for learning Bayesian networks from data with GA*, (2004) [online] http://citeseer.ist.psu.edu/rd/0%2C655055%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/30569/http:zSzzSzwww.cs.uu.nlzSzpeoplezSzstevenzSzdownloadzSzppsn2004.pdf/vandijk04use.pdf

[29]Dirk Thierens, *Non-redundant genetic coding of neural networks*, in Proceedings of the1996 IEEE International Conference on Evolutionary Computation, ICEC 96, Piscataway pp. 571-575 IEEE Press 1996. [online] http://citeseer.ist.psu.edu/rd/0%2C191973%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/3326/ftp:zSzzSzftp.cs.uu.nlzSzpubzSzRUUzSzCSzSztechrepszSzCS-1998zSz1998-46.pdf/non-redundant-genetic-coding.pdf

[30]A.A. Siddiqi, S.M. Lucas, *A comparison of matrix rewriting versus direct encoding for evolving neural networks*, in Proc. of the 1998 IEEE International Conference on Evolutionary Computation (Piscataway, NJ, USA) pp. 392-397 IEEE Press (1998) [online] http://citeseer.ist.psu.edu/rd/0%2C98648%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/1779/ftp:zSzzSztarifa.essex.ac.ukzSzimageszSzsmlzSzreportszSzencoding.pdf/siddiqi98comparison.pdf

[31]F. Gruau, *Neural network synthesis using cellular encoding and the genetic algorithm*, Thesis [online]

[32]H. Mayer, *Symbiotic Coevolution of Artificial Neural Networks and Training Data Sets*, in A.E. Eiben, T. Bäck, M. Schoenauer, Parallel Problem Solving from Nature, Lecture Notes in Computer Science, Vol. 1498, pp. 511-520, Springer (1998).

[33]C. Reeves, S. Taylor, *Selection of Training Data for Neural Networks by a Genetic Algorithm*, in A.E. Eiben, T. Bäck, M. Schoenauer, Parallel Problem Solving from Nature, Lecture Notes in Computer Science, Vol. 1498, pp. 633-642, Springer (1998).

[34]H. Mayer, R. Huber, R. Schwaiger, *Lean Artificial Networks Regularization Helps Evolution,* [Online] http://citeseer.ist.psu.edu/rd/0%2C132842%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/3652/http:zSzzSzwww.cosy.sbg.ac.atzSz%7EreinizSzmayer_huber_schwaiger_step_96.pdf/mayer96lean.pdf

[35]T. Runarsson, M. Jonsson, *Evolution and design of distributed learning rules*, (2000) [Online] http://citeseer.ist.psu.edu/rd/0%2C481498%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/24412/http:zSzzSzcerium.raunvis.hi.iszSz%7EtprzSzpaperszSzRuJo00.pdf/runarsson00evolution.pdf

[36]P. Marrone, *Joone Documentation*, (2006) [Online] http://www.jooneworld.com/site.pdf#search=%22huascar%20fiorletta%22

[37]P. Marrone, *SwiXAT User Guide*, (2006) [Online]
http://prdownloads.sourceforge.net/swixat/SwiXATUserGuide.pdf?download

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 16.09.2006 _____

Huascar Fiorletta