UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

Estrategias Bioinspiradas para la optimización del aprendizaje de Redes Neuronales Artificiales (RNA)

Manuel Konomi Pilkati
Tutor: Sacha Gómez Moñivas

Mayo 2014

Resumen:

A lo largo de las últimas décadas las Redes Neuronales Artificiales (RNAs) han sido un campo de estudio muy popular en el campo de la computación e inteligencia artificial. Esto es debido a potencia que tiene como herramientas en gran variedad de campos y disciplinas. Junto con las RNAs, otro campo en auge es el de los algoritmos evolutivos, algoritmos extremadamente versátiles y eficaces a la hora de atacar un problema de optimización de forma automatizada.

En un intento de optimizar estas redes y avanzar en el desarrollo de estas tecnologías, hemos creado nuestra versión de RNA en la que las neuronas son independientes y pueden actuar de forma distinta a las demás. Esto nos es útil a la hora de analizar el comportamiento de la red al quitarle neuronas después del entrenamiento o inducir anomalías. Utilizando esta red como base, hemos implementado dos algoritmos de aprendizaje: El clásico Backpropagation y un algoritmo genético. Usando las características de nuestra red neuronal artificial hemos realizado un estudio comparativo donde analizamos la tolerancia a la perdida de neuronas en relación al algoritmo de aprendizaje utilizado.

Hemos usado la experiencia en el estudio comparativo para aplicar las características de nuestra red a un caso real en el que clasificamos curvas simuladas de microscopia de sonda de barrido. A través de una serie de experimentos con la función de fit del algoritmo genético hemos aumentado la tolerancia a fallos o datos perdidos en la red con lo que demostramos que el trabajo que hemos realizado tiene aplicaciones directas en la vida real.

Palabras clave:

Inteligencia artificial, algoritmos evolutivos, algoritmos genéticos, backpropagation, redes neuronales artificiales, nanotecnología, neuronas, bioinspirado, perceptrón.

Abstract:

On the last decades, Artificial Neural Networks (ANNs) have been a widely studied field of computer science and artificial intelligence. This is due to their power as a tool in a large number of disciplines. ANNs can classify data, approximate complex functions, make predictions, recognize patterns and more. There are many types of ANN. In our project we use a modified version of a Feedforward Neural Network (FFNN) for our experiments. Along with ANNs, Evolutionary Algorithms (EAs) are a popular study subject. The reason behind this is that EAs are extremely versatile and effective when it comes to automatic problem optimization.

In an attempt to optimise this kind of networks and contribute on their further development we have added to our feedforward neural network the ability to treat neurons independently. This is useful when it comes to disconnect specific neurons after the training or to induce any kind of malfunctioning to observe the effects. Using this network as our baseline we implemented two training techniques: Backpropagation (BP) and the Genetic Algorithm (GA). Thanks to our network's characteristics we have conducted a comparative study where we analyse the network's tolerance to neuronal failure depending on the training algorithm.

We use the experience gained in the study to apply the network's characteristics to a real problem. We classify simulated Scanning Probe Microscopy (SPM) curves. Through a series of experiments with GA's fit function we increased failure and missing data tolerance in the network. With this we demonstre that the work we have done has real life applications.

Keywords:

Artificial intelligence, Artificial Neural Networks, Feedforward Neural Networks, Backpropagation, Genetic algorithm, Evolutionary algorithm, Multilayer perceptron, nanotechnology, neurons.

Table of contents

Table index:

Figures index:

Equation index:

Glossary:

AI	Artificial Intelligence
AFM	Atomic Force Microscopy
ANN	Artificial Neural Network
BP	Backpropagation Algorithm
CANCER	Wisconsin Breast Cancer DataBase (a FFNN problem)
CARD	Creddit Approval (a FFNN problem)
EA	Evolutionary Algorithm
FFNN	Feedforward Neural Network, a subtype of ANN
GA	Genetic Algorithm, a subtype of EA
HORSE	Horse Colic Database (a FFNN problem)
LF	Learning Factor, a parameter used by BP
MLP	MultiLayer Perceptron
PIMA	Pima Indians Diabetes Database (a FFNN problem)
SEM	Scanning Electron Microscopy
SONAR	Sonar, Mine vs. Rocks (a FFNN problem)
SPM	Scanning Probe Microscopy

1. Introduction

In the last two decades, Artificial Neural Networks (ANNs) have been a widely studied field in computer science. This is due to the power and applications of this networks in many different fields such as medical diagnosis, data mining, pattern recognition, function approximation, data classification and more. Now there is an increasing interest on the field of ANN architecture optimisation and many of the proposals are based on biological mechanisms. They draw their ideas from the behaviour of real neurons in the brain and how problems are solved. In the brain, effects such as the loss of a neuron are diminished because the rest of the neurons tune up their connections to supply the missing neuron. This type of mechanisms are the ones that are being adapted to the ANNs.

1.1 Project Objectives

-**Apply the principles of neuron differentiation** where the behaviour of individual neurons can be altered individually. Neurons can be manipulated in several ways such as turning them of, altering their ability to learn, diminishing their output and more.

-**The use of an evolutionary algorithm** (EA) as the training method for our instance of ANN (i.e. a Feedforward Neural Network). EAs are global search techniques that adopt the principle of natural biological evolution and/or the social behaviour of species. The use of EAs as training mechanisms along the neuron differentiation brings the possibility for new ways of architecture optimisation. It makes possible to reduce the effects of noise in the data used by the FFNN as well.

-**Use the developed mechanisms on real nanotechnology problems**. In the context of data classification for scanning electron microscopy (SEM), the developed techniques are used to try to increase the network tolerance to anomalies in either the data or the network.

1.2 Document structure

This document is organized in the following sections

- **Introduction**: We present the ideas with which we will work through the project and explain the structure of the document

- **Technologies and State of Art**: Here all the knowledge, background, algorithms and technologies we will use are presented. We also explain our modifications and use of such technologies.

- **Design and Development**: In this section we explain the architecture and organization of the tool we implemented to make our experiments. We also analyse the classes of which the tool is composed by.

- **Results**: In this section we present the results of our experiments

- **Conclusion**: This last section summarizes what we have learnt through the project.

- **References**: Bibliography of the document.

2. Technologies and State of Art.

The different technologies used in this project are described in this section. We start with the state of art of these technologies. Then we explaining the main common characteristic of the feedforward network (FFNN). To finish, we introduce two learning techniques used to train the FFNN. This techniques have several modifications specifically developed for a comparative analysis.

2.1 State of Art

2.1.1 FFNN and learning algorithms

Over the last decades, solving classification problems with artificial neural networks such as Feedforward Neural Networks (FFNN) is considered a promising and useful strategy [1] due to their high capability to classify real world complex problems. In classification problems, the network has to be trained (i.e. generate a set of weights that solves the problem properly) with a set of data to correctly classify their desired outputs. Once trained, the network is ready to classify new data [2]. Since training means to optimise the weights of the network, we should decide which optimization algorithms better perform the task. At present, there are a lot of different algorithms used to train a network. One of the most widely used is the backpropagation algorithm (BP) [3] [4]. However, regardless of its popularity, BP has several disadvantages. First, BP converges very slowly when the network's architecture starts being big and complex. Another well-known undesired effect is that BP can easily fall into local minima [5] [6]. It is also very dependent of several parameters. For example, to make BP work properly, learning factor and momentum parameters must be chosen carefully. It is well-known that, once working, any slightly change of them can disturb the networks accuracy [7]. Finally, BP strongly depends on the training set presentation: to achieve the optimal set of weights, BP depends on the sequence of training cases used (i.e. the same cases in a different order may train the network better or worse). It was pointed out by Curry and Morgan [8] that gradient techniques might not always give the best and fastest way to train an ANN.

In this context, evolutionary algorithms (EAs) appear as a promising answer to the necessary improvement in the ANN learning process. EAs are global search techniques adopting the principle of natural biological evolution and/or the social behaviour of species. One of the main advantages of EAs is their ability to escape from local minima [9] since, unlike BP, they start with a wide population of solutions. There are already many studies that show how EAs give accurate and promising results [10] [11]. For example, in [1] [12] we can see comparatives between BP and genetic algorithm learning (for both real and binary coded), as well as a method by Daniel Rivero et al. [13] to use GA to optimise the network architecture.

2.1.2 AI and ANN applied to nanotechnology

Modern scientific and technological development increasingly relies on nano, biological and information sciences. For more than a decade, the thought that the convergence of nanotechnology, artificial intelligence (AI) and biology will promote another technical and scientific revolution has been lingering [14]. However, this expected integration of multidisciplinary research is still in progress. Nanotechnology combines the knowledge of physics, chemistry and engineering [15], while artificial intelligence has heavily relied on biological inspiration to develop some of its most effective paradigms such as Neural Networks or Evolutionary Algorithms (EA) [16]. Bridging the gap between current nanosciences and AI can boost research in these disciplines and provide a new generation of information and communication technologies that will have a large impact in our society (see Figure 1).

In the last few years, there has been an increasing emphasis on AI techniques applied to Nanotechnology. For example, EAs have been applied to automatize the process of imaging in Scanning Probe Microscopy (SPM) with software that is able to tune the precise state of the probe and the associated control parameters [17]. The image treatment in Atomic Force Microscopy (AFM) has been also helped by using EA to select image filters with the proper type and order [18], which significantly improve the quality of the images, helping in the location of nanoparticles. Artificial Neural Networks (ANNs) have also been employed to determine the morphology of Carbon Nanotube turfs by quantifying structural properties such as alignment and curvature [19]. The characterization of different properties of thin films has been solved by the PI’s group using ANNs, where the determination of electrostatic [20] properties has been done by using theoretical simulations for the training set.

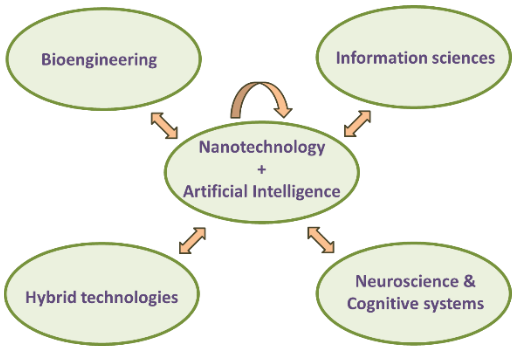
Bioengineering
Information sciences
Nanotechnology + Artificial Intelligence
Hybrid technologies
Neuroscience & Cognitive systems

Figure 1: Convergence of nanotechnology and artificial intelligence has a broad potential impact in many other scientific fields, e.g. bioengineering, novel information sciences based on new computer architectures and data representations, hybrid technologies that use biological entities, nanotechnological devices and research in neuroscience and cognitive systems, to name a few.

These are a few examples of recent applications where Nanotechnology and AI have been combined. However, a deeper interaction between both disciplines is needed. Since the convergence of Nanotechnology and AI is an incipient topic, it is very common to find publications where standard ANNs or EAs are being used without developing new AI paradigms or even without using the full potential of the most standard algorithms.

2.2 Technology

2.2.1 Simple perceptron

First, to start explaining the multilayer perceptron, we should talk about the simple perceptron.

A simple perceptron is a classifying algorithm able to generate criteria to split a set of elements into different subsets or classes. The only condition for this is that the classes have to be divisible by a linear function. The simple perceptron’s structure can be seen in the following diagram:

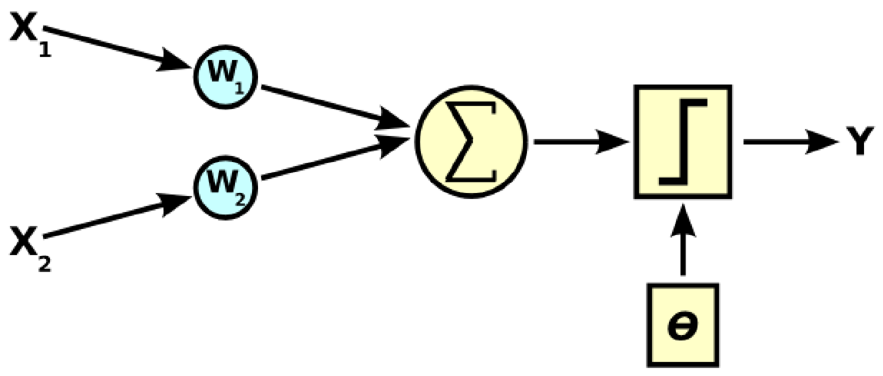

Figure 2: Simple perceptron's mechanism

In Figure 2 we see two inputs, X_1 and X_2, and two weights, W_1 and W_2, associated with them. The perceptron multiplies each input value with its respective weight. Once done with all inputs, it sums up all the results. This sum acts like the input of the threshold activation function. The other input, θ, is compared to the sum. If the sum is bigger than θ the output of the perceptron will be 1. If it is not bigger than θ the output will be 0. This can be seen in the following equation.

$$\begin{cases} if \sum x_i w_i > \theta \; then \; y = 1 \\ if \sum x_i w_i \leq \theta \; then \; y = 0 \end{cases}$$

Equation 1: Simple perceptron's behaviour

When implementing a simple perceptron, the input θ can be moved to the other side of the equation as we see in Equation 2. By doing this it can be treated like another input-weight set where the input will always be 1 and the weight would vary to meet the original value of θ.The reason to do this transformation is to simplify the implementation of the simple perceptron. This way we only need to do a sum and check if the result is either over or under zero.

$$\begin{cases} if \sum x_i w_i - \theta > 0 \; then \; y = 1 \\ if \sum x_i w_i - \theta \leq 0 \; then \; y = 0 \end{cases}$$

Equation 2: Simple perceptrón simplification

2.2.2 Multilayer Perceptron

The Multilayer Perceptron (MLP) is a network formed by several neurons. The neuron originated from the simple perceptron, and therefore has an almost identical behaviour. The difference between a neuron and a simple perceptron lies in their activation function.

This is because the most popular training algorithm (BackPropagation, BP) needs a continuous and derivable activation function. Usually, the sigmoid function is appointed to this task.

Neurons in the MLP are organized in layers as we can see in Figure 3. Each layer generates the input for the next layer. This diagram represents the structure of a MLP where each circle is a neuron:

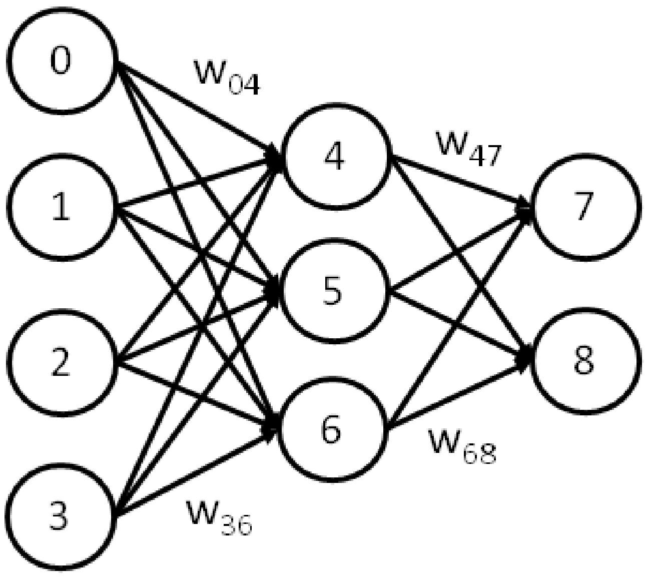

Figure 3: An example of FFNN with four inputs, one hidden layer and two outputs.

This type of artificial neural network (FFNN) solves the simple perceptron's biggest limitation. MLP can classify sets that are not linearly divisible.

As we said before, MLP is arranged in layers. There are three types of them.

-**Input layer**: Composed by the neurons that receives the system's input.

-**Output layer**: The last layer, whose outputs represent the system's output.

-**Hidden layer/s**: all the layers in-between the input and output. The calculations of the network are mostly done here.

2.2.2.1 Characteristics of the network

In this project, we use a particular case of multilayer Feedforward Neural Networks (FFNN). From the most widely used FFNN, we have added context-independent functionality to neurons. Now, every neuron can be individually configured to modify its behaviour in the following ways:

-Neuron Shutdown. A specific neuron can be shut down (i.e. turning its output to 0. The effects of turning off input or hidden neurons are shown in Figure 4. As we can see, a certain number of weights stop working (or being useless since they end in a non-working neuron). Instead of turning off a neuron completely, we can also reduce its performance by a certain percentage, emulating a partial wrong behaviour.

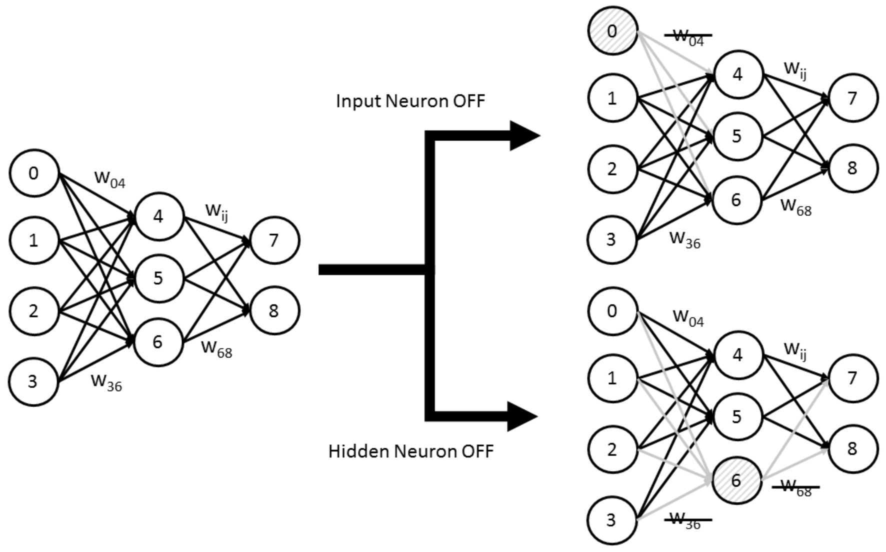

Figure 4: Scheme of the FFNN and the effects on the network performance when an input or hidden layer is turned off. Grey arrows represent the weights that stop having any influence in the final result.

-Possibility to use a different activation function. For the present study, we have just implemented the sigmoid function.

-Sigmoid function modification. When setting up a neuron with a sigmoid function, we can modify its sigmoid function by changing the parameters α, β and γ, as we show in the following equation:

$$s(x) = \frac{\beta}{\gamma + e^{-x\alpha}}$$

Equation 3: Modified Sigmoid function

-Managing the learning factor. Our FFNN allows the user to change the learning factor of a particular neuron. The learning factor had an effect on the speed of the learning process. This gives us the chance to make a neuron learn faster or slower than the others.

2.2.3 Evolutionary Algorithm (EA) as a learning method

Evolutionary Algorithms (EAs) are essentially search functions, i.e. EAs search for the fittest solution of a problem [21]. To decide whether a solution is good or not, the algorithm bases its decision on a pre-set criteria, where the best solution will be the one that better satisfies these criteria. The searching strategy is inspired by the evolutionary process at cellular level. First, EAs generate a set of random solutions we call Chromosomes. Then, they make the chromosomes evolve by means of crossing, mutating and selecting the finest next chromosome generations, created from the previous ones.

In the general EA algorithm strategy that is the origin of our method [22], a chromosome is represented by an array of bits. The process to obtain a solution is divided into the following stages:

1- **Generation of a random set of chromosomes (Initialization).** In this phase the initial set of chromosomes is created. The randomness is needed in order to have the wider variety of elements, which is needed to avoid local minimum and premature wrong convergence of the algorithm

2- **Evaluation of the population.** In this stage, once we have a set of individuals, all of them are evaluated according to the previously selected criteria. This evaluation is done by the fit function. Indeed, depending on the problem we are trying to solve, this function can be very different. Since the fit function drives the search of the algorithm, it is extremely important.

3- **Selection of the finest.** Once the population has been evaluated, the best ones are selected to create a new generation of chromosomes that will inherit their characteristics. There are several selection methods, but all of them mainly follow the same principle: better solutions have better probabilities of being selected. First, a selection probability is calculated with the following equation

$$\mathrm{p}(\mathrm{C_i}) = \frac{\mathrm{f}(\mathrm{C_i})}{\sum_{\mathrm{j}=1}^{\mathrm{N}} \mathrm{f}(\mathrm{C_j})}$$

Equation 4: GA's selection probability

where P is the selection probability of the chromosome C_i and the function f(Ci) is the fit function of a chromosome. When all chromosomes have their respective P(Ci) calculated, a selection procedure called stochastic sampling with replacement [22] is used. All chromosomes are mapped on a roulette wheel (see Figure 5). The roulette is proportionally divided between the chromosomes according to their probabilities. By spinning the roulette, the selected chromosomes are copied for the new population. Because of this mechanism, chromosomes with high probability might be copied several times.

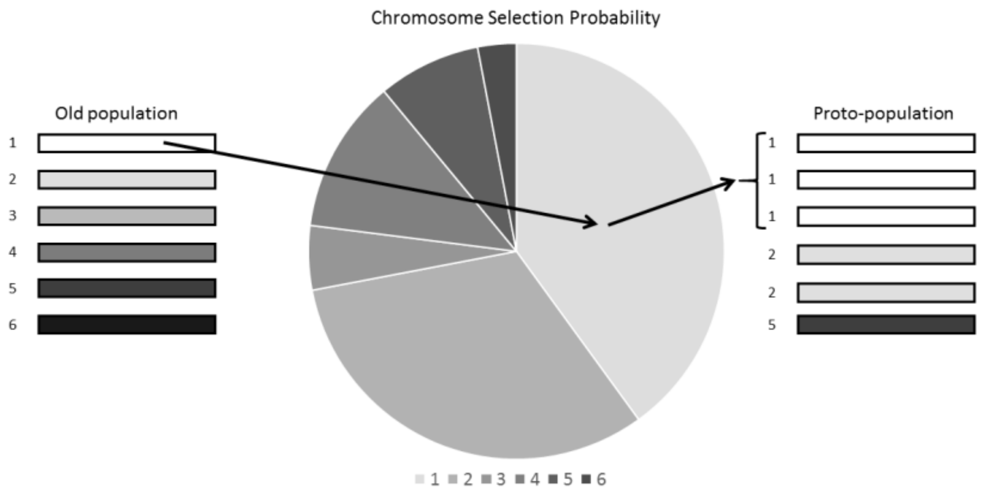

Figure 5: Roulette wheel method mapping the probabilities of the old population (left) and selecting the candidates for the new generation (proto-population on the right side).

Once we have selected enough chromosomes to have a new population, the individuals with the best fit will be repeated in the new population, taking the place of the worst ones.

4- **Individual Crossing.** Crossing emulates genetic recombination that takes place in meiotic cell division. In that process, two homologous chromosomes exchange pieces of genetic material, turning themselves into different chromosomes. The crossing process is done by selecting an initial crossing point along the chromosome and swapping the following bits with their counterpart in the second one. Selecting an initial and final crossing point and swapping only the bits between them is another way of individual crossing. We can see a visual example in the following diagram.

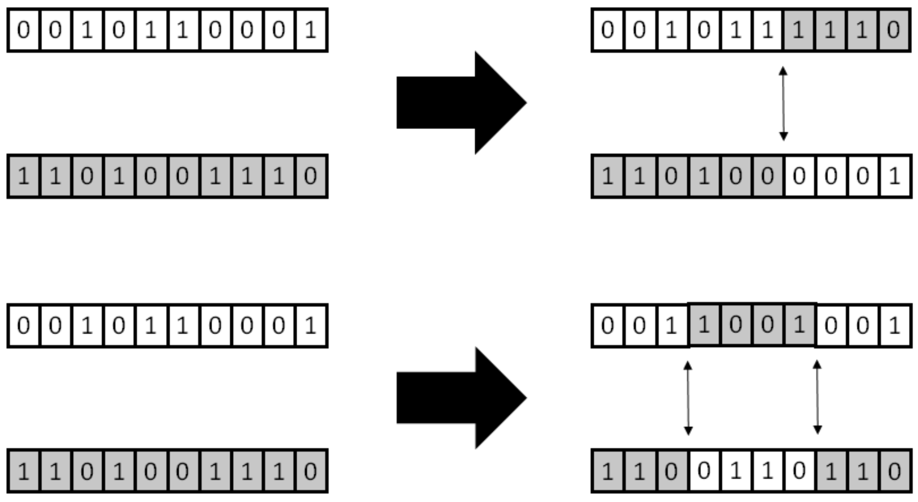

Figure 6: Chromosome crossing process. On the top, one point crossing. On the bottom, two point crossing

5- **Individual Mutation.** At this point, some randomly chosen chromosomes are mutated by changing the value of a bit from 1 to 0 or vice versa. The reason of this step is exploring new possible solutions that were impossible to reach with no more than the information from the parents. Like in natural selection, mutation can be beneficial for the chromosome or harmful (it is likely to be harmful). For this reason, only a few chromosomes are mutated. An example of mutation can be seen in the following diagram.

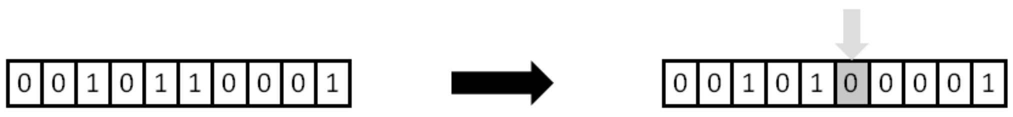

Figure 7: Chromosome mutation process. We see how the sixth bit is mutated.

6- **Repetition.** At this point, our new population is ready to go back to phase 2. This loop will continue until we get a solution good enough to solve the problem, or a certain number of generations are reached. Both conditions must be checked between phases 2 and 3.

Now, we will show how we have adapted this general algorithm for the specific porpoises of this article. First, we need to represent the solution in a way we can apply mutation and crossing in the way we have shown before. In other words, we need to represent the solution of the problem as a chromosome. To use EA in the training of FFNN, our chromosome will be an array of weights, which will correspond to a set of real numbers. Since the original EA was designed with arrays of bits, several modifications are needed to make it work with real numbers.

If we want to keep the classic EA unmodified, we could think as a first approximation a way to transform the real numbers to a bit based representation (binary coded EA). Although this solves the problem initially, it also limits the accuracy of the algorithm because we transform a continuous representation to a discrete one. Since this first approximation lacks the accuracy, we have chosen to adapt the algorithm to a Real Coded Genetic Algorithms (RCGAs) [23] [24] in the following way:

1- **Chromosome representation.** As we have already stated, chromosomes will become arrays of real numbers instead bits. For this reason, we have to change the full structure of the algorithm.

2- **Evaluation of the population.** The original EA does not specify any evaluation function (fit function). The fit function is problem-dependant and different in general for every problem. In our solution the fittest chromosomes are the ones that have the lower value in the fit function. For the training of FFNNs, the value is calculated by adding all the least-square errors between the output of the system and the desired output the way we see in the following formula.

$$fit(x) = 1 - \frac{\sum_j \sum_i (net(i,x) - t_i)^2}{I \cdot J}$$

Equation 5: Real coded GA's Fit function

Where **x** is an individual; **i** represents a training case; **net(i, x)** represents the output of the FFNN for the case **i** and the set of weights of the individual **x**; **t** is the desired output for the case **i**; **j** is one of the output neurons; **I** is the number of cases; and **J** is the number of output neurons. Once the fit of the population is calculated, it is normalised and used for the selection.

3- **Selection of the finest.** Because the evaluation always gives a real value (both in our version of EA and the original) there is no need to change the selection mechanism. It works as previously explained.

4- **Crossing.** This phase of the algorithm is heavily affected by the change of individual representation. If we made it the way it is usually done in the original algorithm, the resulting individuals would be too different from the originals [25] thus making convergence difficult. There are several ways to implement the crossover operator in a RCGA. Here we list some of them.

Let us assume that $C_1 = (c_1^1 \, . \, . \, . \, c_n^1)$ and $C_2 = (c_1^2 \, . \, . \, . \, c_n^2)$ are two chromosomes that are going to be crossed.

-Flat crossover [26]

In this crossover method, the offspring is calculated by simply choosing a random number in the interval $[c_i^1, c_i^2]$.

-Simple crossover [27] [28]

This crossover works exactly like the one point crossover seen before, choosing point **i** in the array of values and swapping the following numbers. We can see this in the next equation:

$$\begin{cases} O_1 = (c_1^1, c_2^1, \dots, c_i^1, c_{i+1}^2, \dots, c_n^2) \\ O_2 = (c_1^2, c_2^2, \dots, c_i^2, c_{i+1}^1, \dots, c_n^1) \end{cases}$$

Equation 6: Simple crossover mechanism

-Arithmetical crossover [28]

This mechanism generates two offsprings according to the next equations:

$$\begin{cases} o_i^1 = \alpha c_i^1 + (1-\alpha) c_i^2 \\ o_i^2 = \alpha c_i^2 + (1-\alpha) c_i^1 \end{cases}$$

Equation 7: Arithmetical crossover mechanism

The parameter α can be either constant or vary along the training.

-BLX-α crossover [29]

In general terms, this crossover method is an evolution of the flat crossover where. The value **o$_i$** of the offspring O is selected from the following interval.

$$[c_{i\_min} - I \cdot \alpha, c_{i\_max} + I \cdot \alpha]$$

Equation 8: BLX-α crossover mechanism

In Equation 8 , considering $I = c_{i\_max} - c_{i\_min}$, the parameter α controls the size of the interval. With α=0 the interval would be the same that the one in flat crossover.

-Discrete crossover [30]

H$_i$ is a randomly (uniformly) chosen value from the set [c^1, c^2].

We use a variation of the arithmetical crossover as shown in Equation 9:

$$\begin{cases} I_1' = \propto \cdot\ W_1 + (1-\propto) \cdot\ W_2 \\ I_2' = \beta \cdot W_1 + (1-\beta) \cdot\ W_2 \end{cases}$$

Equation 9: Discrete crossover mechanism

Where **I$_1$** and **I$_2$**, with their respective arrays of weights **W$_1$** and **W$_2$**, are crossed to form **I'$_1$** and **I'$_2$** using two random numbers α and β that exist in the range [0-1]. That's how both new individuals will have a part of their progenitors.

5- **Mutation.** This step has to be modified as well. In the original EA, mutation simply switched a bit to its other possible state, 1 or 0. This cannot be done with real numbers, there is no other state. Like crossover, there are several mutation operators proposed for the RCGA. We will explain some of them. Given the chromosome $C = (c_1, \dots, c_i, \dots, c_{n)}$, a gene $c_i \in [a_i, b_i]$ to be mutated and c_i' the mutated gene.

-Random mutation [28]

c_i' is randomly selected from the interval $[a_i, b_i]$.

-Non-uniform mutation [28]

This mutation mechanism adds or subtracts a value to the gene c_i. As the number of generations increase, the value added or subtracted is smaller.

$$c_i' = \begin{cases} c_i + f(t, b_i - c_i) \; if \; \tau = 0 \\ c_i - f(t, c_i - a_i) \; if \; \tau = 1 \end{cases}$$

Equation 10: Non uniform mutation

$$f(t, y) = y(1 - r^{\left(1 - \frac{t}{g_{max}}\right)^b})$$

Equation 11: Non uniform mutation auxiliar function

Where τ is a random binary number to decide upon subtracting or adding; g_{max} is the maximum number of generations; *r* is a random real number in [0, 1] and *b* a parameter given by the user. By modifying the value of *b* the number of generations can more or less important. Equation 11 returns a value inside [0, y]. The pro of this method is that in an early stage of the algorithm the exploration of the solution space is better.

Following the philosophy of mutation, what we have done is to add or subtract a small number to the weight chosen for mutation. This increment is chosen randomly inside a range defined by a constant [25].

$$\Delta x = \alpha \cdot cte$$

Equation 12: Our implementation of the mutation mechanism

Being α a random number between -1 and 1, this will make the selected weight **x** to vary from **x-cte** to **x+cte**.

2.2.4 Back Propagation learning strategy

BackPropagation (BP) is one of the most popular algorithms to train and make the FFNN learn [7]. It can be divided into two phases. In the first one (i.e. forward phase) BP calculates the output of the FFNN given a set of inputs. Note that this is supervised learning. That means we know the desired output for the cases we are feeding to the FFNN.

In the second phase (i.e. backwards phase) the desired output is used to calculate the error on the output layer's prediction. This output layer error is propagated backwards (thus the name BackPropagation) to calculate the error of every neuron in the FFNN. Once all errors are calculated, the values of the network's weights are updated.

As we said in 2.2.2, BackPropagation needs a derivable function to be the transfer function of the neuron. Such function is usually the sigmoid function. Therefor is good to know how the sigmoid function and its derivate look, for the derivate is used in the formulas in the next section.

In out method, we have used a sigmoid function S as the transfer function of the network:

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

Equation 13: Normal sigmoid function

where x is the sum of the weight-input multiplications that enter in the neuron. This function is commonly used because its derivative can be expressed in terms of itself:

$$\frac{ds(x)}{dx} = s(x)(1 - s(x))$$

Equation 14: Sigmoid function's derivate

To make the neurons work with a specific and unique behaviour, the sigmoid function has been modified to match the one seen in Equation 3. The algorithm is able to change the behaviour of individual units both in the training and validation phases. The momentum factor [7] can also be switched off at any time in both phases.

2.2.4.1 Weight update and error calculation in BP

In this section we explain in detailed way how error in BP is calculated and how the FFNN's weights are updated to reduce it. The following figure will help us to name the components of an FFNN and to explain the equations involved in the process.

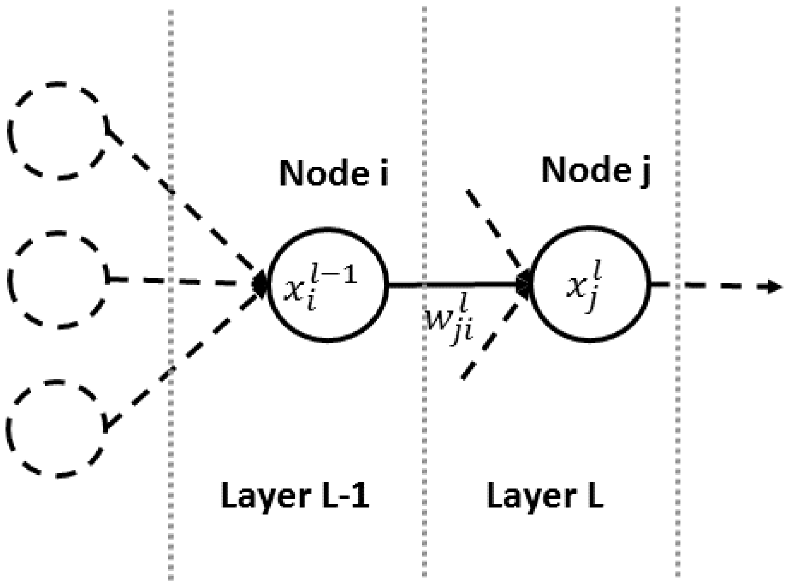

Figure 8: Scheme and notation for the FFNN

According to the diagram's notation, the formula to calculate the increment of the weights looks like this:

$$\Delta w_{ji}^{l} = \eta \delta_j^l x_i^{l-1}$$

Equation 15: Weight increment calculation for BP training

Where η is the learning factor, who controls how much the system learns with every case (e.g. a low learning factor makes the system learn slowly). δ_j^l represents the calculated error for the neuron J in the layer L. x_i^{l-1} is the output of the neuron I in the layer L-1. Summarising, in order to calculate the increment of a weight, the calculated error of the target neuron is multiplied by source neuron's output and the learning factor.

The calculation of the error is made with gradient descent method. That is the reason why the transfer function has to be derivable. This is how the output layer's error is calculated using the sigmoid as the transfer function [7].

$$\delta_j^L = (x_j^L - y_j)x_j^L(1 - x_j^L)$$

Equation 16: Error calculation at the output layer for BP

Where **L** is the output layer and **j** is one of the output neurons. **Y** represents the desired output and **x** the actual output, generated by the system. Note that if we delete the bracket containing **x-y**, we would get the sigmoid's derivate.

When the output layer error is calculated, it is used to calculate the error in following layers with Equation 17:

$$\delta_j^l = x_j^l(1 - x_j^l)(\sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1})$$

Equation 17: Error calculation at the hidden layers for BP

In this equation we are calculating the error in the neuron j of the layer l. The first part of the equation represents the sigmoid's derivate. In the second part we sum the errors of the neurons from the next layer multiplied by their respective weight.

3. Design and Development

In this section we introduce the data representation defined for the FFNN. Also the low and high level architecture will be discussed. We will first see the design of the application and the design choices related to it. Then on the development section, we see all the modules in a detailed way.

3.1 Design

The requisites for our application are to implement a FFNN with the characteristics listed in 2.2.2.1. The language we chose for this application is Java. The reasons for this are that we do not have to manage memory in an active way, the virtual machine takes care of that. Other reason is that Java simplifies the implementation via all of its resources.

It is true that Java has some drawbacks, being efficiency the biggest one, but that is not a problem in our project. All those reasons make Java the perfect language for the implementation.

3.1.1 Architecture

Here we will take a look at the applications architecture. In Figure 9, Figure 10 and Figure 11 we can see the design of the FFNN model (the core functionalities), the input/output system and the package structure with its dependencies respectively. This application is easy to implement with an object oriented language like Java. As we see in Figure 9 the classes are *ANN* (i.e. the neural network) that is composed mainly of *Neuron*s. This is analogue to how the brain works in real life. The brain is the *ANN* and it is composed by neurones. The other important class here is *Input*. It represents the connections between the neurons.

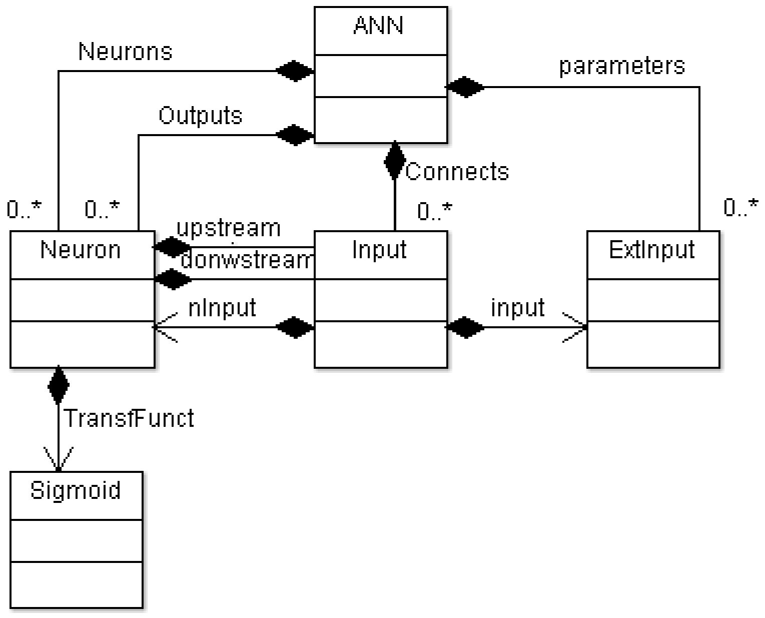

Figure 9: Application model. The main class, ANN is shown with all its components.

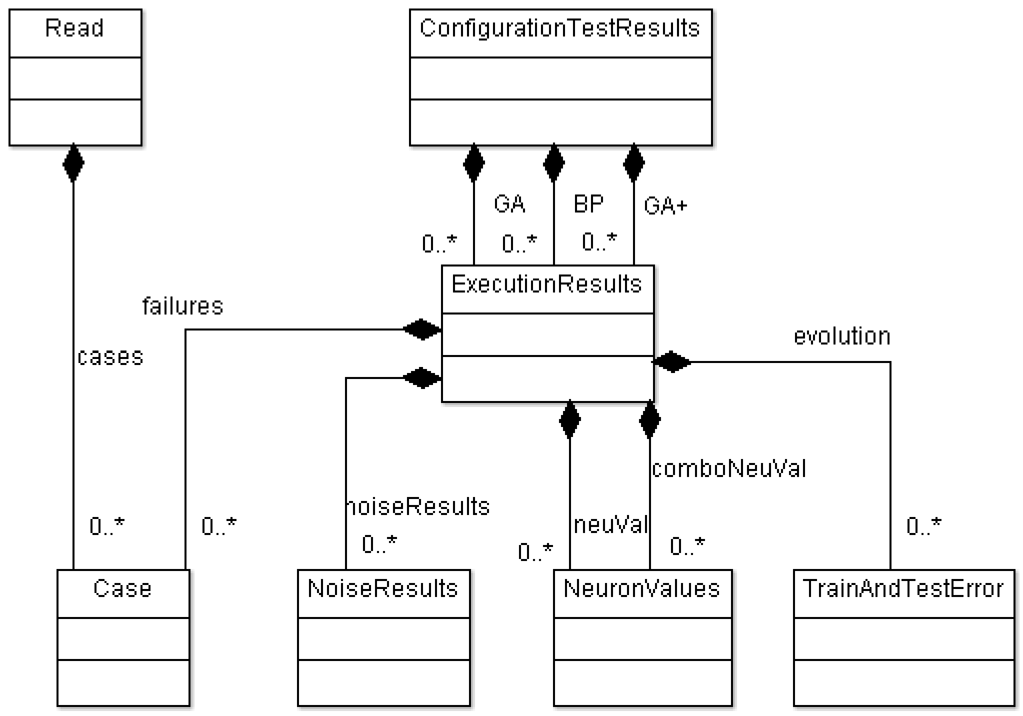

Figure 10: Application's Input and Output system. Read represents the input and ConfigurationTestResults the output.

In Figure 10 we can take a look to the input/output system of the application. *Read* and *ConfigurationTestResults* are the input and the output respectively. *ExecutionResults* is a class used to gather all the information generated during the training of the network with any of the algorithms.

The tool is divided into the following five modules:

- Main module
- Elements module
- Input module
- Output module
- Misc. Module

The main module is connected to the other four in the way we can see in Figure 11

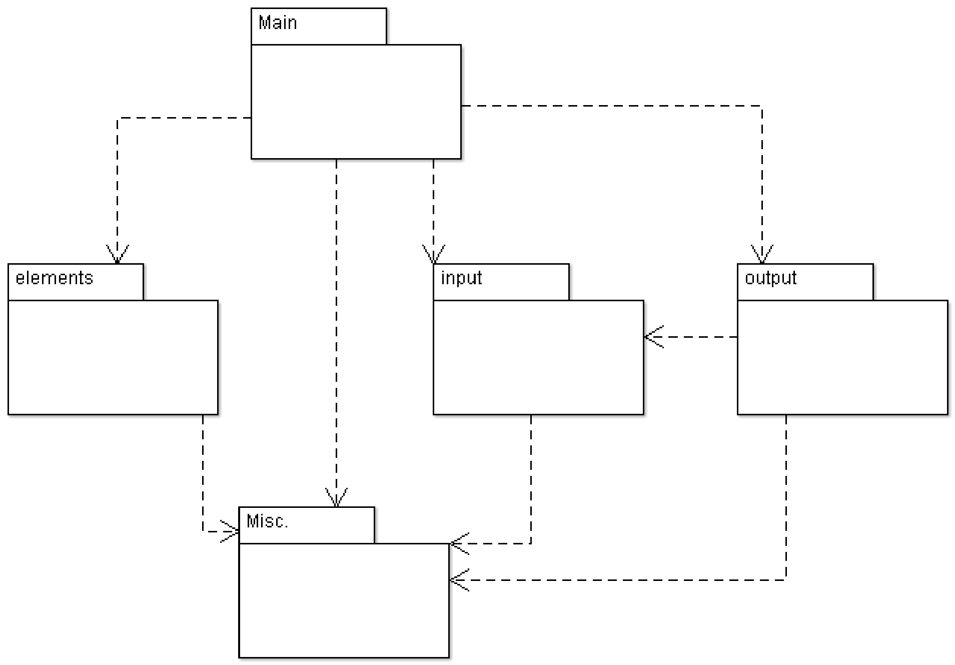

Figure 11: Package organization and their dependences.The classes in each package are explained later.

3.2 Development

In this section we take a detailed look at the modules that conform the application, the classes within them and their functionalities.

3.2.1 Main module

This module is composed exclusively of the FFNN class. This is the most important class of the system. It manages all the neurons of the network and its connections.

The functionalities implemented in this module are the following:

- Reading network specifications from a file
- Building the components of the FFNN and connecting them
- Training the network with the BackPropagation(BP) algorithm
- Training the network with the Genetic Algorithm (GA)
- Changing the input values of the network
- Calculating the output of the network for a given set of input values.
- Activating/deactivating single neurons given a neuron index
- Modifying the weights of all the connections of the network.
- Evaluating a set of problem cases and compare the output with the expected result
- Saving data related to the system evolution/learning
- Adding noise to problem cases before evaluation if needed.

3.2.2 Element module

This module contains the single components that together form a FFNN. These components are the class *Neuron* that implements a Neuron, the class *Input* that implements a connection to or from a neuron, and *ExtInput* that is used as the input of the network.

3.2.2.1 Neuron class

This class's main functionality is to calculate the output of the neuron given a set values in the *Input*s connected to it. The class has several minor functionalities in order to make BP easier to implement. These functionalities are: saving the last output of every neuron, their last error, applying the learning factor and applying the momentum factor. There are two important fields in this class, the downstream and upstream arrays. These arrays hold the connections of the neuron to the next and previous layer respectively. The second one although not being essential for the system does simplify it a lot.

Finally, the neuron has several configurable parameters. These parameters are the transfer function, the learning factor, the momentum factor, activation state and output strength.

3.2.2.2 Input class

As we said before, this class represents a connection to or from a neuron. This class connects neurons with external inputs (*ExtInput*) as well. Its main fields are a reference to the neuron or external input it comes from and a weight.

Its main functionality is to get the value of the external input or the output of the neuron it is connected to. Then it returns this value to whatever object that requested it along with the weight.

Its minor functionality is to propagate the error when using BP as the learning algorithm and to update its own weight.

3.2.2.3 ExtInput class

As we have mentioned several times before, this class represents the input to the FFNN. There will be as many external input classes as parameters in the problem cases. Each external input will keep the value of its respective parameter.

3.2.3 Input module

This module has only two classes and they are responsible for all the reading operations in the system. It performs two tasks: reading the problem case set from a file and reading the configuration of the network neurons, if any.

3.2.3.1 Read class

This is the only class that interacts with external files. Its principal functions are *read*, *read2*, *readAndNormalise* and *readConfig*. The first two functions work almost identically so we will explain just the first.

-*Read* function:

It reads a set of problem cases in a specific format. Both *read* and *read2* cover the same functionality. The difference is the file format that they read. In both cases the function reads al cases in *Case* objects. An array of this objects is returned for the FFNN to use.

-*ReadAndNormalise* function:

This function is very similar to the read functions. The only difference is that after the reading, the attributes of every case are normalised depending on the data read. This function is format-dependet as well. Therefor if the file format is changed, a new function has to be made.

-*ReadConfig* function:

This function reads the specific configurations for individual neurons. By doing so it is possible to modify one or more aspects of a particular neuron's behaviour. Note that it is not necessary to manually configure all neurons (they have a default configuration) or configure all neuron parameters, just the desired ones.

The next example shows the different ways to configure a neuron:

```
ID=6 ACT=1 LF=0.1 ALFA=3

ID=8 B=5 ACT=0
```

The configurations are stored in *NeuronConfig* objects that are later used by the FFNN to generate neurons. The function generates as many *NeuronConfig* as required by the chosen architecture (i.e. the number of neurons in the network), all of them configured by default. The *NeuronConfig* objects will be later modified by the readConfig function if necessary.

3.2.3.1.1 Neuron setup options.

The neuron configuration is written in the following format:

- On each line a neuron is configured.

- On the beginning of the line, the identification of the modified neuron is noted, writing the tag "ID" and the value of the id separated by a space. It is ended with a semicolon.

- Following the ID, the rest of parameters are specified in a similar fashion. Each parameter is separated from the next with a semicolon. The tags which are used to identify each parameter are:

- "LF" for the learning factor
- "FAIL" for the failure rate of the neuron
- "FTYPE" for the failure configuration (failure in learning phase, normal use, both or none of them)
- "ACT" for the activation status (i.e. the neuron either gives or not an output)
- "ALFA", "B" y "G" for alfa, beta and gamma respectively for the parameters seen in Equation 3

3.2.3.2 Case class

This class represents a single case of a problem set. This class is an interface between the external files and the FFNN. Its main functionality is to store the value of the inputs to the

FFNN and the expected outputs for the particular case. This class can be used for training if it stores expected results or for real classification if it doesn't.

3.2.4 Output module

This module has all the necessary methods to gather all the data generated by the FFNN and save said data properly. Its objective is to make possible to launch a batch of tests without supervision and guarantee that all data will be available for analysis.

It contains three classes. *ConfigurationTestResults*, *ExecutionResults*, *noiseResult*, *TrainAndTestError* y *NeuronValues*.

3.2.4.1 ExecutionResults Class

This class stores the information generated in a single execution (i.e. a complete training with one of the algorithms and a verification of the results). The data stored is the following:

- Percentage of classification success after training.
- Percentage of classification success of the FFNN when a single neuron is offline (this is done with every neuron).
- Percentage of classification success of the FFNN when a combination of two hidden neurons are offline
- The *NeuronValues* of each offline neuron and each combination of neurons.
- The evolution of the error of the network during the training with one of the algorithms.
- The execution ID.
- The set of weights of the FFNN after the training process.
- Percentage of classification success when different levels of noise are applied to the study cases.
- A set with the study cases that the FFNN was incapable of classifying.

Other functionality this class has is to write all the stored data into files given a root path. The created files have an ".xlsx" extension (i.e. excel files) and four files are created:

- Main result, with the nomenclature XXY.xlsx where XX can be "AG", "BP" or "APP" and Y is the execution ID. It contains the classification success, the weights of the connections and the *NeuronValues* for every offline neuron
- Evolution results, with the naming XXY_evolution.xlsx. It contains the error percentage obtained when classifying the training set and the test set in every iteration of the training process
- The noise tests file, with the naming XXY_noise_results.xlsx. Contains the classifying success of the network when a given noise level is applied to the study cases.
- The failure file. Its naming is XXY_failure_cases.xlsx. Contains the cases that were wrongly classified.

3.2.4.2 ConfigurationTestResults class

The purpose of this class is to store the variables and attributes of a batch of executions of the FFNN with either learning algorithm. It later writes the stored information.

The stored information is:

- All the setup parameters for the network (Learning factor, number of learning cycles, mutation and crossover probability…)
- The individual tests (*ExecutionResult*s) of each learning algorithm. (it can be more than one test per algorithm)
- A set with all the results from all the executions of the performance with a neuron offline.
- A set with all the results from all the executions of the performance with a combination of neurons offline.
- The mean, max and min classification success of each learning method.
- The root folder where all the files will be created.
- The architecture of the network for all the tests.

Once all tests are over and the data gathering has finished, the class creates the root folder if it didn't exist before. Then it creates a text file where network setup, architecture and mean/max/min performance of the algorithms are written. After that, two other files are created. One showing the network performance through the different executions with the different neurons offline. Similarly to the first, the second contains the performance of the network with different combinations of neurons are offline. To finish, the class creates a folder for every execution for them to write their data on.

3.2.4.3 TrainingAndTestError class

This class stores de value of the network performance in a given iteration of its learning process. It contains the performance with the training data set and the validation (i.e. test) data set.

3.2.4.4 NoiseResult class

The only task of this class is to store the performance of the FFNN when random noise is applied to the test cases. It also saves the noise level used on the cases.

3.2.4.5 NeuronValues class

This class contains a series of values concerning an offline neuron. As said before, after the network training, the system shuts down all the neurons one at a time to check the performance with that neuron lost.

The values saved in this class are:

- The ID of the offline neuron.
- The performance of the network with the offline neuron.
- The average of the input connections (i.e. upstream) weights.
- The average of the output connections (i.e. downstream) weights.
- Max and min weight for both upstream and downstream.

3.2.5 Miscellaneous module.

This module contains a series of classes that perform auxiliary tasks and doesn't clearly belong to any of the other modules. This classes are *Individual*, *ANNConfig*, *NeuronConfig* and *Sigmoid*.

3.2.5.1 Individual class

This class is used exclusively by the GA. It represents a chromosome or individual. The individual is an array of *double* where each double represents a weight of the FFNN.

Aside from the weights, the individual has a fit value associated as well. This fit is used to determine whether or not it is selected for the new generation. It is a measure of the quality of the individual as a solution.

To finish with this class, this function is responsible for the individual crossing as well. Given a second individual, it randomly crosses both while modifying the originals.

3.2.5.2 Sigmoid class

It represents a sigmoid function that can be altered by modification of the parameters ALFA, BETA and GAMMA seen before.

3.2.5.3 ANNConfig class

This class simply holds the parameters for the network architecture (e.g. inputs, outputs, hidden layers…) plus the default momentum and learning factors. It is used to store the configurations and give them to the network builder method.

3.2.5.4 NeuronConfig class

This class has a similar objective than the last one but with neurons on its scope. It stores all the configurations relative to the neuron we explained in 3.4.1 As we said, a set of this objects are created while reading the configuration. This objects initially have default values for each neuron but if the configuration file specifies something different, this values are changed. This objects are directly used by the neuron builder.

4. Tests and Results

In this section we present the results gathered after a series of experiments. First we assert both training algorithms behave properly. For this we use a series of problem sets to test the network accuracy. We also do a comparative analysis of both algorithms. Later we test the effects of neuron failure on the FFNN's output. And to finish, we apply the developed techniques on a real nanotechnology problem.

4.1 Backpropagation and Genetic Algorithm training

We have studied 5 different sets, where we have applied both BP and EA training. In both cases we have execute the simulation 20 times where the training phase started with different randomly selected initializations. The parameters for the BP training are the following: at least 20000 learning iterations and learning factor 0.1. For the EA training we have used 1000-17000 generations, mutation factor = 0.4, mutation and crossover probability = 0.8 and population size = 10. We have chosen these values after different simulations with a wide variety of values for all the parameters. The final decision is the one that guarantees a good performance for all the sets under study. In Table 1 we show the FFNN geometry selected for every set. In Figure 12 we show an example of the learning rate for three representative sets (CANCER, CARD and PIMA) for both BP and EA training.

Set	Inputs	Hidden	Outputs	Class 1	Class 2	Class 3
CANCER	9	8	2	458/65.52%	241/34.48%	
CARD	51	6	2	307/44.49%	383/55.51%	
PIMA	8	6	2	268/34.76%	503/65.24%	
HORSE	58	12	3	224/61.54%	88/24.18%	52/14.29%
SONAR	60	12	2	152/48.72%	160/51.28%	

Table 1: Geometry used in the simulations for the different sets. It is also shown the number of cases that correspond to the different classes in the set. These numbers are given in absolute value and percentage.

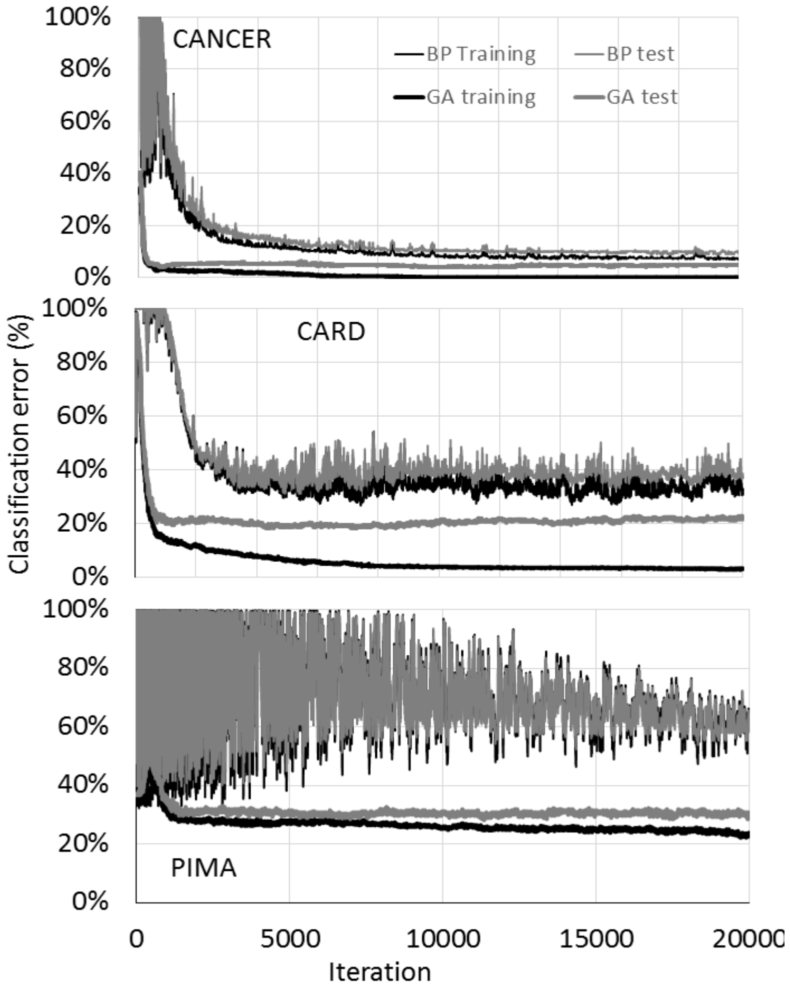

Figure 12: Training evolution for the CANCER, CARD and PIMA sets for both Genetic Algorithm and Backpropagation methods. The error is shown as the percentage of cases that are not correctly classified.

As we can see, the EA based training converges much faster and has a better performance for all the cases under study. Similar results are found for the HORSE and SONAR sets and are not explicitly shown in the figure. In terms of computation time, BP needs 2.29 times more iterations than the EA training, in average. In the CARD case, we have found a small overtraining effect in EA training that is not distinguishable for BP. Since this effect is very small and the training set is clearly improved, we have decided to use the values for 20000 iterations in all the figures. As we can see in Figure 12 this value is a very good choice for any other set and training method since overtraining is not present. Only in the PIMA set we have used a higher value (200000) since the BP method did not converge well at 20000 iterations.

4.2 Neuron failure analysis

Let us study now the effect of losing individual neurons or units. To do it, we are going to use the ability of our algorithm to configure different performances of the individual neurons. First, we are going to switch off a single neuron in the hidden layer by setting b=0 in the sigmoid function. It is worth noting that this change is only applied after the training phase, i.e. the network has been trained with a full and correct performance of every single neuron in the system. After that, we are going to turn off sets of two hidden neurons. Since the neurons in the hidden layer are initially indistinguishable, all the possible combinations should be studied for a full statistical analysis.

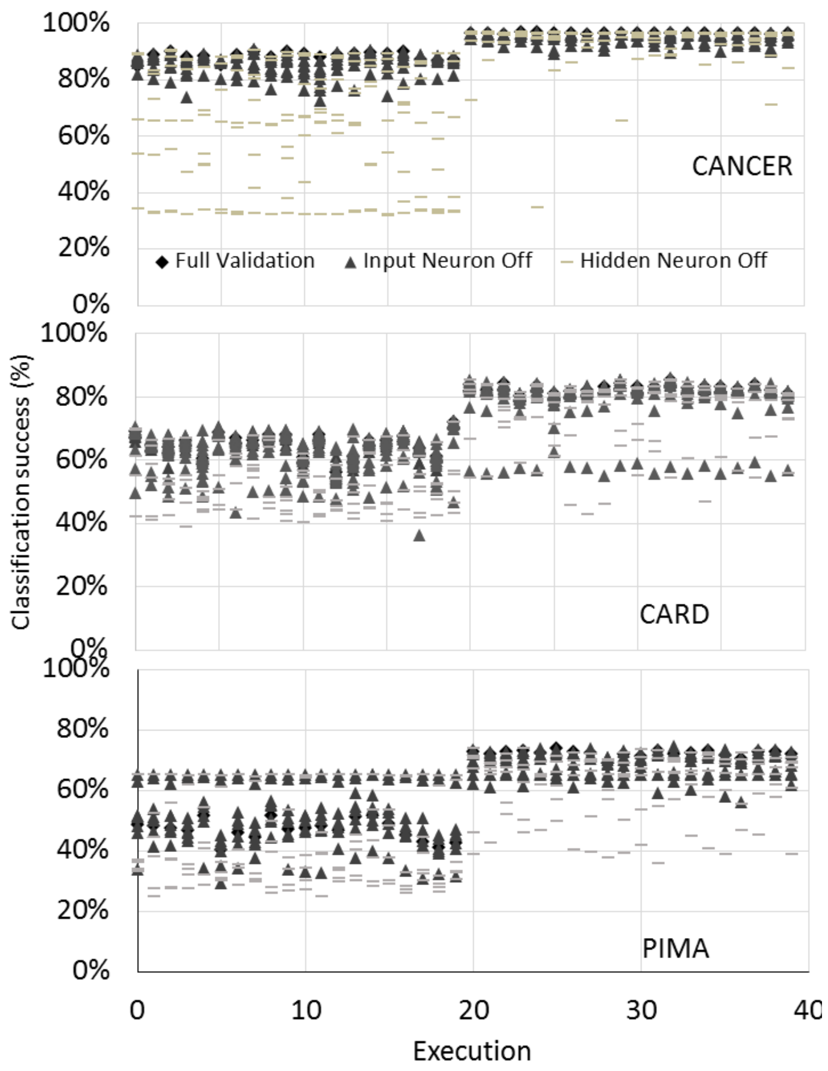

Figure 13: Classification success (defined as the number of cases that are correctly classified) for CANCER, CARD and PIMA sets. The figure shows simultaneously the values for 20 executions for both Backpropagation training (from 0 to 19) and Genetic Algorithm training (from 20 to 39). In all figures, we have a data set for every hidden and input neuron that are included in the FFNN geometry selected for every set under study.

Results of turning off a single hidden neuron are shown in Figure 13 and Figure 14. As we expected, BP trained networks are in general affected by the loss of many hidden neurons. Indeed, there are others that induce a much smaller effect when stopping their activity. This different influence of the hidden neurons has been extensively studied, for example, in several articles related to the pruning effect. Much more remarkable is the extremely high tolerance that is observed in the case of EA trained networks. In this case, the network is still having very similar ratios no matter which neuron is turned off. In Table 2 and Table 3 we show the average and standard deviation respectively for all the cases under study. These values have been obtained by averaging the values from the suppression of every neuron in the 20 cases under study. As we can see, in all the cases we obtain better average values and smaller standard deviations in the EA trainings. The higher robustness against a non-working neuron makes EA training a better choice for the simulation of real biological systems. These kinds of systems must be stable against many effects, like the death of a certain number of neurons, which could imply the loss of the signal from any unit. Although biological systems have many different ways to prevent a wrong network behaviour when the individual units are not working properly, an adequate weight distribution could be also an effective method to prevent the loss of information. In that sense, EA training have demonstrated to be a method with an effective capability of keep working practically in the same effective way when neurons are not having a perfect performance.

There are a few effects in the figures and tables that must be mentioned in the analysis. First,

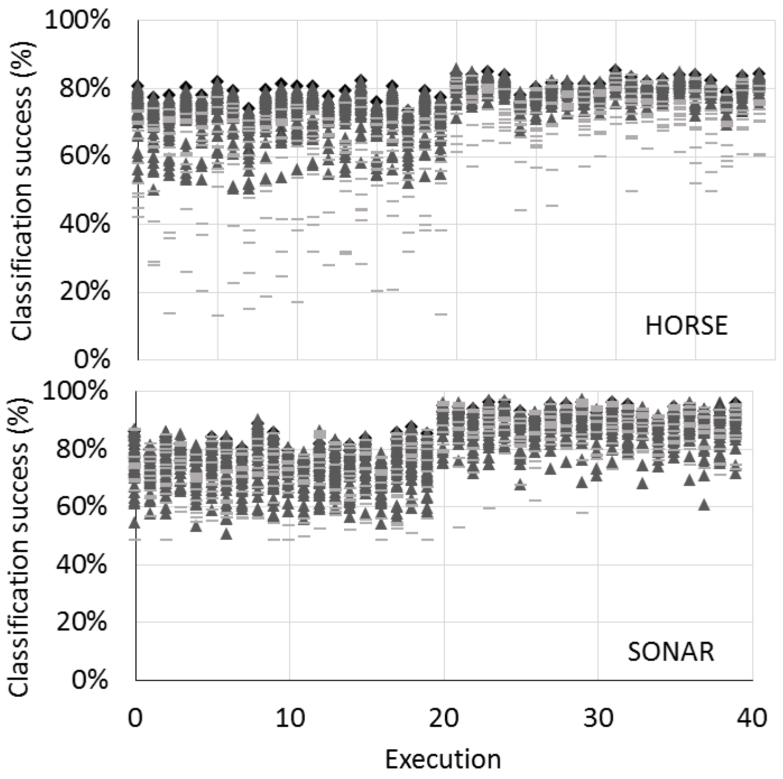

Figure 14: Classification success (defined as the number of cases that are correctly classified) for HORSE and SONAR sets. The figure shows simultaneously the values for 20 executions for both Backpropagation training (from 0 to 19) and Genetic Algorithm training (from 20 to 39). In all figures, we have a data set for every hidden and input neuron that are included in the FFNN geometry selected for every set under study.

the PIMA set seems to give better results when a hidden or input neuron is switched off. In Figure 13 we can see the original set having values around 50%, which is approximately the value obtained from Table 2. However, in many cases, we can see that this value can be increased up to 65% when neurons are switched off after the training. This effect is not really an improvement since this set has 503/771 inputs that correspond to a certain class. As we show in Table 1, this numbers correspond to a relative value of 65.24%, which correspond to the maximum classification success percentage achieved if Figure 12. What is really happening is that, sometimes, the modification included in the system makes it fail and the FFNN always answers with the second class, making it increase the classification success to 65.24% accidentally.

	Original		Hidden		Input	
Set:	BP	EA	BP	EA	BP	EA
CANCER	88.35	96.52	62.18	94.27	85.97	95.3
CARD	65.29	82.86	52.54	75.67	64.67	82.17
PIMA	48.09	72.22	41.58	62.55	50.9	68.82
HORSE	78.89	82.57	58.26	72.67	73.3	80.59
SONAR	82.45	94.55	70.45	86.94	76.63	90.75

Table 2: Average values of the classification success for all the cases under study and both BP and EA training. Results are shown for the case where no modifications are given (original) and the cases where input and hidden neurons have been turned off.

	Original		Hidden		Input	
Set:	BP	EA	BP	EA	BP	EA
CANCER	1.51	0.19	21.58	6.44	3.58	1.6
CARD	3.69	1.18	8.57	10.07	4.6	3.83
PIMA	3.35	1.06	13.96	10.12	9.59	3.77
HORSE	2.37	1.94	15.03	7.43	5.53	2.66
SONAR	4.02	1.53	9.58	7.53	7	5.2

Table 3: Standard deviation values of the classification success for all the cases under study and both BP and EA training. Results are shown for the case where no modifications are given (original) and the cases where input and hidden neurons have been turned off.

From Figure 13 and Figure 14, and Table 2 and Table 3 we can see that both training methods are much more stable when we switch off an input neuron. In Figure 15 we show the average and standard deviation obtained for both BP and EA trainings in all the cases under study. We can see that losing an input neuron induces a minimum effect in the effectiveness of the ANNs for both BP and EA. Interestingly, losing a hidden neuron does not have either a significant effect in the average value for EA in the CANCER set. Moreover, the effect is also smaller for the other 4 sets than in the case of BP. Analysing the SD, we see, however, that it has a significant increase even when the average seems to be similar. We can see the reason of this effect in Figure 12, where some losing neurons reduce significantly the classification success for EA training.

Let us now make the problem worse by switching off two neurons from the hidden layer at the

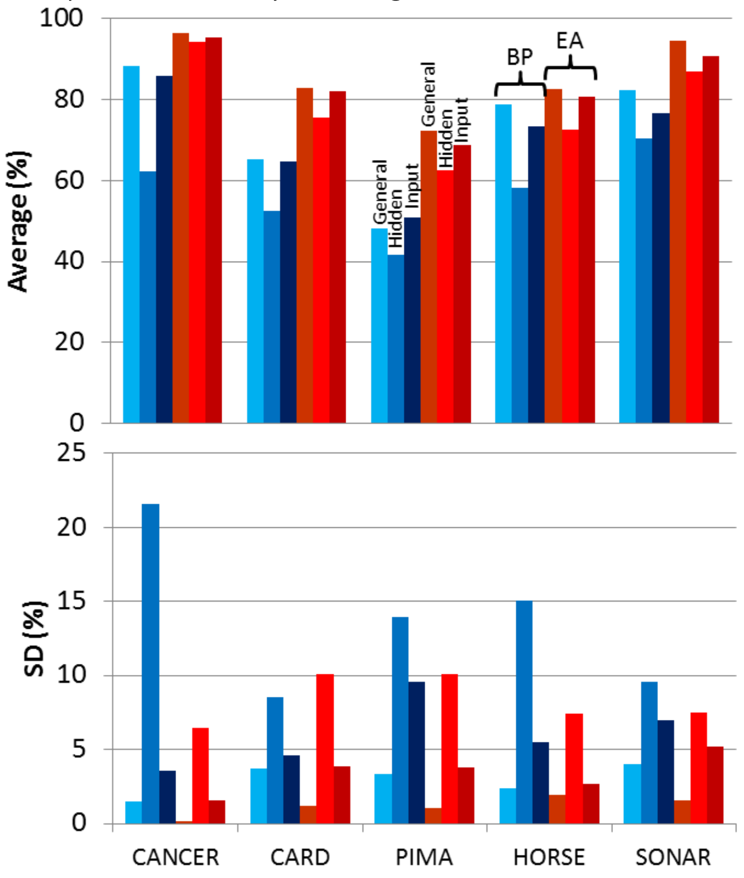

Figure 15: Average value and standard deviation of the classification success (%) for the five sets under study in the case of turning off one hidden or input neuron. The statistical calculations include the 20 different executions and all the hidden and input neuron modifications in the columns that represents the modification of the neurons in those layers.

same time. In Figure 16 and Figure 17 we show the performance of the same sets than in Figure 13 and Figure 14 with the only exception that, in this case, we switch off all the possible combinations of two hidden neurons. Since the sets used in this article have very different number of input neurons, we did not include the effect of turning them off for clarity.

As expected, the performance is worse than in Figure 13 and Figure 14 for both BP and EA training. However, we can see that EA training is still able to give better results. In this case, it is not so clear to see the effect in the figures, due to the big amount of information. In this case, we need the help of the absolute values of average and standard deviation from Table 4 and Table 5.

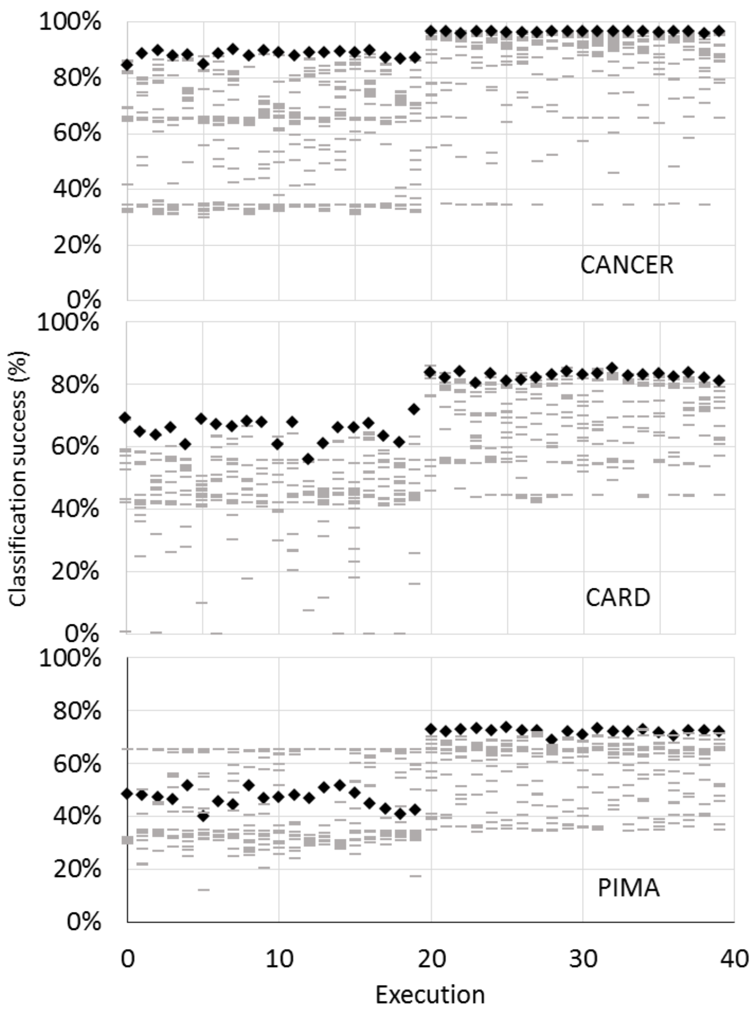

Figure 16: Classification success (defined as the number of cases that are correctly classified) for CANCER, CARD and PIMA sets. The figure shows simultaneously the values for 20 executions for both Backpropagation training (from 0 to 19) and Genetic Algorithm training (from 20 to 39). In all figures, we have a data set for every combination of two hidden neurons included in the FFNN geometry selected for every set under study.

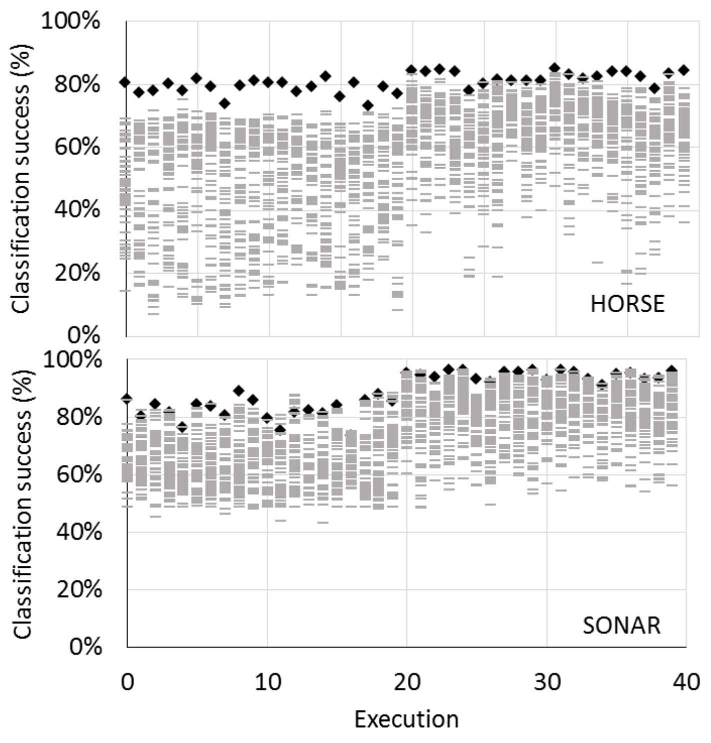

Figure 17: Classification success (defined as the number of cases that are correctly classified) for HORSE and SONAR sets. The figure shows simultaneously the values for 20 executions for both Backpropagation training (from 0 to 19) and Genetic Algorithm training (from 20 to 39). In all figures, we have a data set for every combination of two hidden neurons included in the FFNN geometry selected for every set under study.

From Table 4 and Table 5 we can see that switching off two neurons from the hidden layer in the case of EA training still gives better average results than the original case of BP in the CANCER, CARD and PIMA sets (being pretty similar in the SONAR set and clearly worse in HORSE). However, we must be very careful when comparing these results since removing a couple of hidden neurons always increases the standard deviation dramatically. This implies that, even when the average results are good, we cannot trust EA training in this case since there is a huge variation in the results depending on random factors such as the initialization of the weights. In Figure 18 we show an image with the results from Table 4 and Table 5. In this figure we can see the small difference in the average value and, at the same time, the big increasing of the standard deviation.

	Original		Hidden	
Set:	BP	EA	BP	EA
CANCER	88.35	96.52	55.71	89.69
CARD	65.29	82.86	46.33	68.86
PIMA	48.09	72.22	43.95	57.79
HORSE	78.89	82.57	48.22	64.76
SONAR	82.45	94.55	63.99	81.57

Table 4: Average values of the classification success for all the cases under study and both BP and EA training. Results are shown for the case where no modifications are given (original) and the case where two hidden neurons have been turned off.

	Original		Hidden	
Set:	BP	EA	BP	EA
CANCER	1.51	0.19	19.38	12.67
CARD	3.69	1.18	11.69	12.34
PIMA	3.35	1.06	15.25	11.63
HORSE	2.37	1.94	16.22	11.03
SONAR	4.02	1.53	9.54	9.41

Table 5: Standard deviation values of the classification success for all the cases under study and both BP and EA training. Results are shown for the case where no modifications are given (original) and the case where two hidden neurons have been turned off.

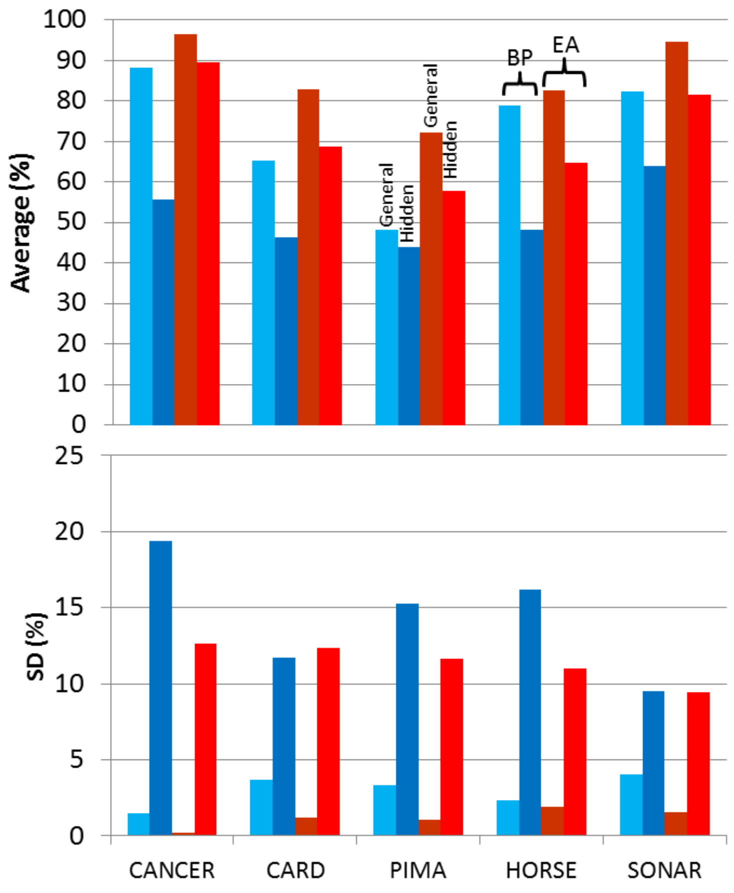

Figure 18: Average value and standard deviation of the classification success (%) for the five sets under study in the case of turning off two hidden neurons. The statistical calculations include the 20 different executions and all the hidden and input neuron modifications in the columns that represent the modification of the neurons in those layers.

4.2.1 Fit function modifications

After this analysis of BP and GA, we have experimented with GA's fit function. This is done in order to lead GA's search to a solution with certain desired characteristics. This characteristics where having low value weights to allow the removal of a neuron (or more than one) with minor consequences. This modification adds a penalty to every chromosome's fit according to their weight values. The penalty will be high if the chromosome's weights are high and so on.

This initial penalty system proved not to work as expected. The main reason for this is our inability to choose the range of values where the penalty is applied. Even the slightest variations in that criterion have great consequences on the GA's training process.

In order to solve this problems, we changed the way we understand the weights in a chromosome. From individual weights to subsets of weights with a source or destination neuron in common. By means of this, we evaluate the influence of a neuron in the network and not just single weights. If the input weights of a neuron are negative (e.g. every weight is around -3) the output will be near zero. With an output near zero its contribution is very small and the neuron can be removed safely. If the output weights are near zero, the output of the neuron (that will go through those weights) will also be near zero. Following the same logic than before, this neuron can be safely removed as well.

This approach simplifies the targeting for the penalty system. It will strive to lower the weights related to a neuron. Anyway this penalty system upgrade shares the same problems of the old one: the inability to define the penalization range and the penalization itself.

4.3 SPM samples classification

To apply what we have learnt on the other tests, we have a set of Scanning Probe Microscopy (SPM) simulated samples. A training set consists of a series of force values at different Tip-Sample distances (D). These samples have two possible classes based in two attributes: the relative dielectric constant ε and the screening length λ. When $\lambda \to \infty$, the sample does not have any free charge and is considered a dielectric, i.e. the electric field is only partially compensated by the local polarization of the sample's molecules. The first class includes the samples within this limit. The second class includes all the samples with finite λ value. To simplify the problem, in this second class we have fixed ε=5. In Figure 19 and Figure 20 we can see some samples of each class.

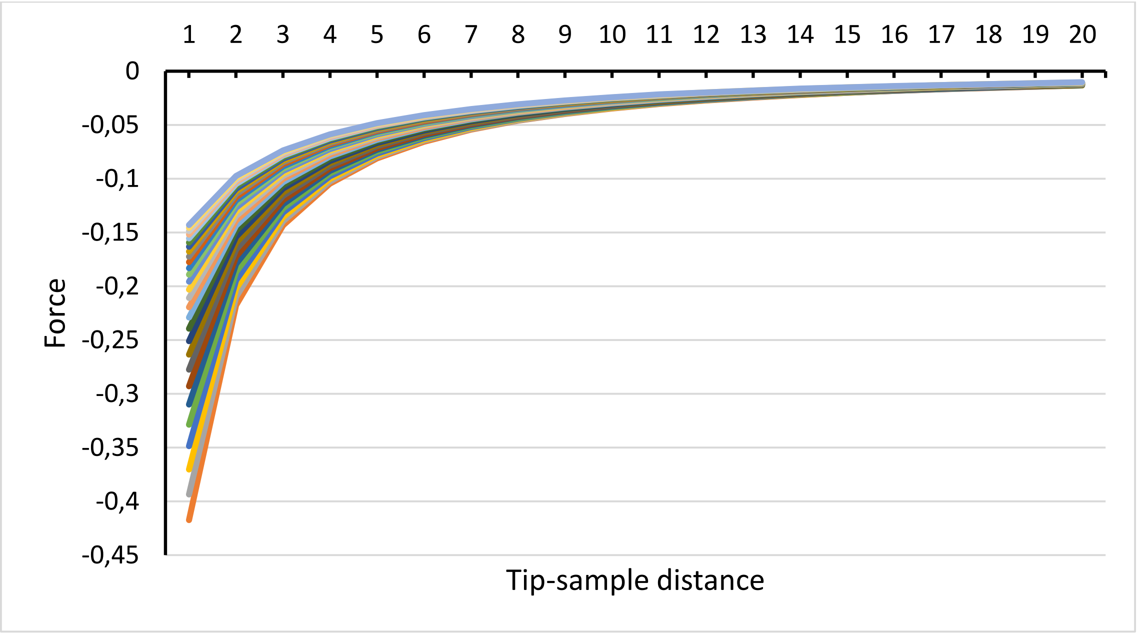

Figure 19: A set of simulated sets belonging to the LAMBDA class. The X axis represent the distance between the Tip and the sample, the Y axis represents the force between them.

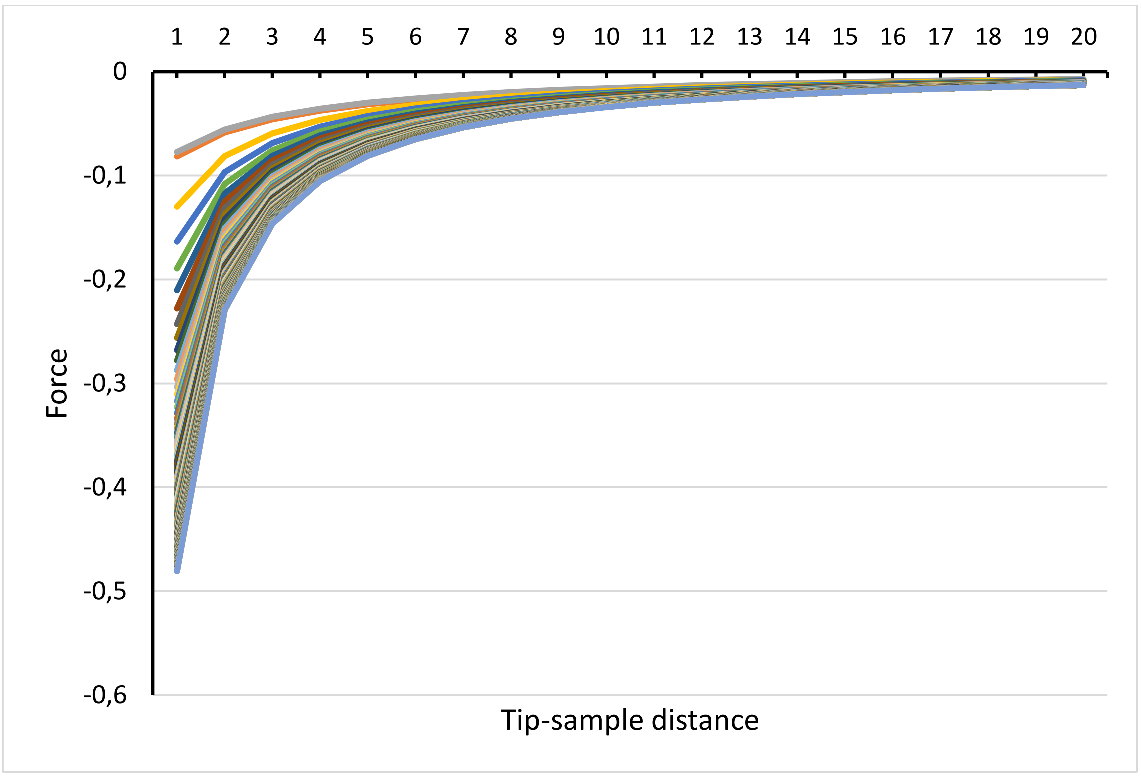

Figure 20: A set of simulated sets belonging to the LAMBDA class. The X axis represent the distance between the Tip and the sample, the Y axis represents the force between them.

4.3.1 Standard classification results

Before trying to optimize the network for this problem we have first checked that it can learn to classify the data. In order to do this, we have followed the same procedure we shown in section 4.1. In this case we have only used GA as the training method. The parameters chosen for the GA training are 10000 generations, mutation factor = 0.4, mutation and crossover probability = 0.8 and population size = 10. In Table 6 the chosen architecture and the class distribution of the set is shown. In Figure 21we show the error rate through the training process with GA.

Set	Inputs	Hidden	Outputs	Class 1	Class 2
SPM	20	8	2	201/50%	201/50%

Table 6: Geometry used in the simulations for the Scanning Probe Microscopy set. It is also shown the number of cases that correspond to the different classes in the set. These numbers are given in absolute value and percentage.

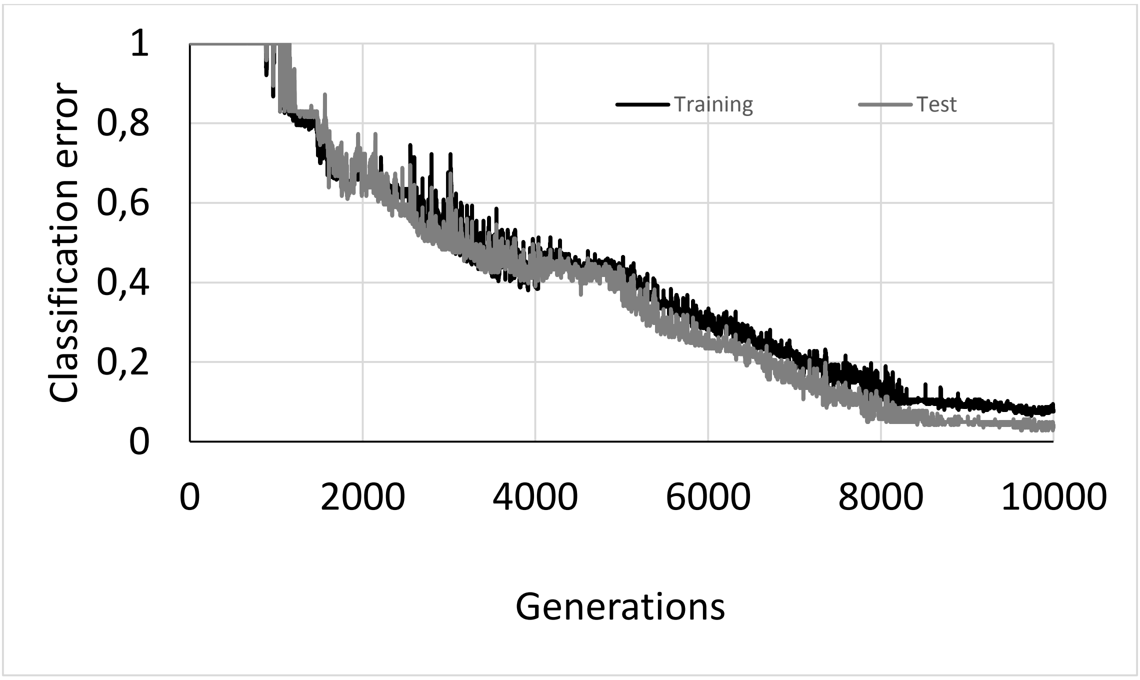

Figure 21: Training evolution for the SPM set for the Genetic Algorithm method. The error is shown as the percentage of cases that are not correctly classified.

As we can see, the GA has a good performance when it comes to classify the SPM samples. Once we have checked its proper behaviour, in Figure 22 we can see the effect on the success rate of the system when a single input neuron is offline. We can establish a relation between this figure and Figure 19 or Figure 20. When the offline neuron represents a short probe-material distance the effects on the output are huge, and as we go to neurons with a bigger distance, the effect on the output becomes imperceptible.

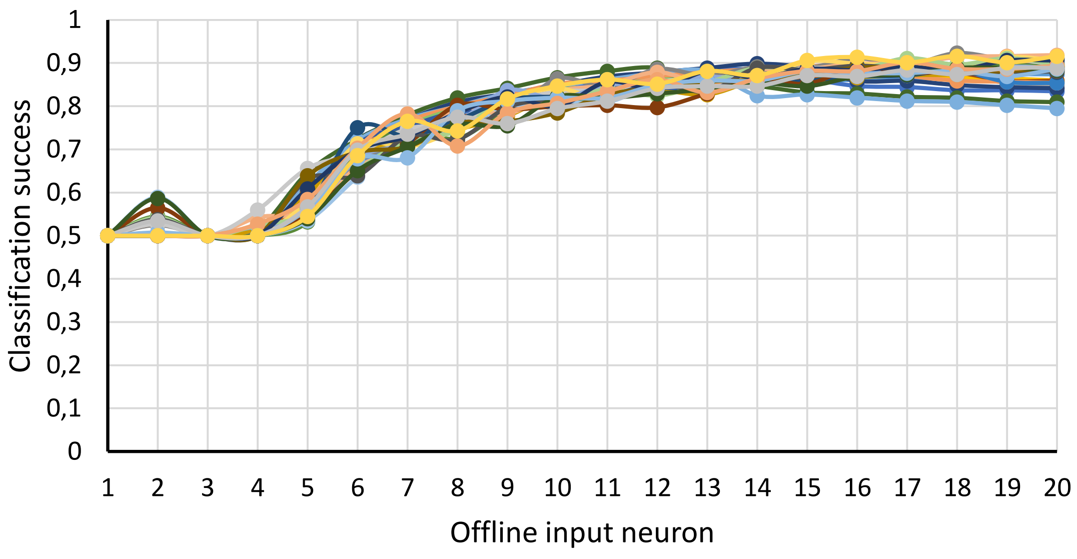

Figure 22: The effect of an input neuron failure is shown. The X axis corresponds to the offline input neuron. The Y axis represents the network's accuracy to classify the samples. The bigger de distance a neuron characterizes, the lesser the effect on the overall output.

The reason behind this effect is that is easier for the network to classify using the data coming from the first 4-5 neurons. This is because the data (if we see Figure 19 and Figure 20) coming from those distances are more spaced.

4.3.2 SPM fit function optimization

In order to distribute the classification importance of the neurons better, and to improve the network's tolerance to neuron failure, we implemented four new fit functions with this sole purpose. All these fit functions use the same data displayed in Figure 22 to rise those curves.

4.3.2.1 Summatory fit function

This first approach calculates the fit of a chromosome in the standard way and then, by disconnecting the input neurons one by one, calculates the fit for every offline neuron. Once it has finished calculating the fits, it sums them all. This sum is the fit for the individual. It is reflected in the following equation:

$$fit'(x) = fit(x) + \sum_i fit(x_i)$$

Equation 18: Summatory fit

Where *i* is the input neuron to be disconnected, *x* is the chromosome and x_i is the fit of the chromosome when the *i* neuron is offline. To study this fit function we trained the network with the following parameters: 50000 generations, mutation and crossover probability 0.8, mutation constant 2.0 and chromosomes 10. In Figure 23 we can see the success values when each neuron is disconnected, and in Figure 24 we can see the evolution of this case.

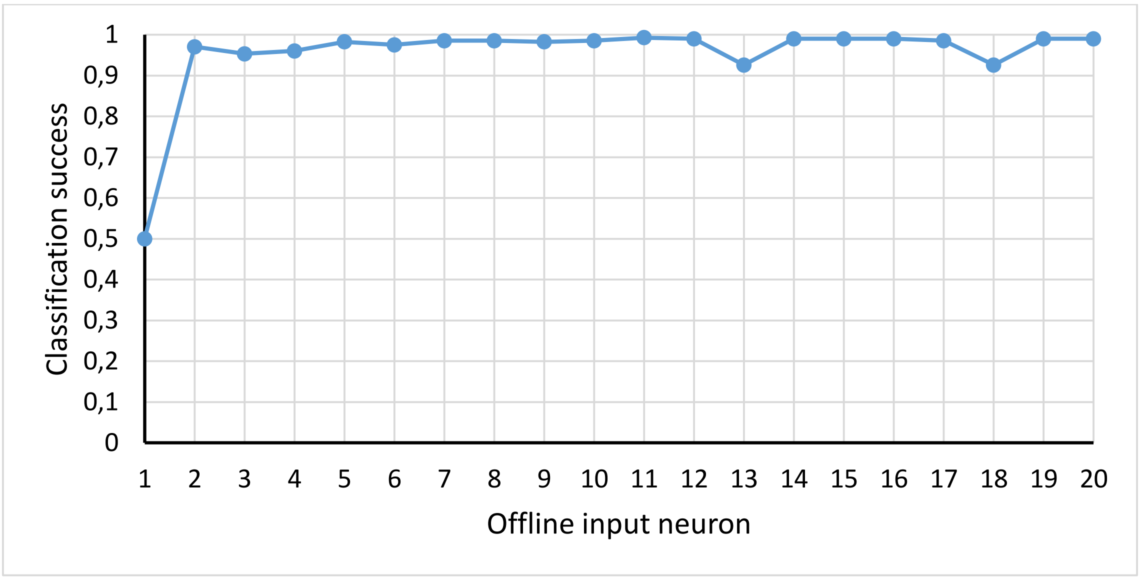

Figure 23: The effect of an input neuron failure is shown. The X axis corresponds to the offline input neuron. The Y axis represents the network's accuracy to classify the samples. The bigger de distance a neuron characterizes, the lesser the effect on the overall output.

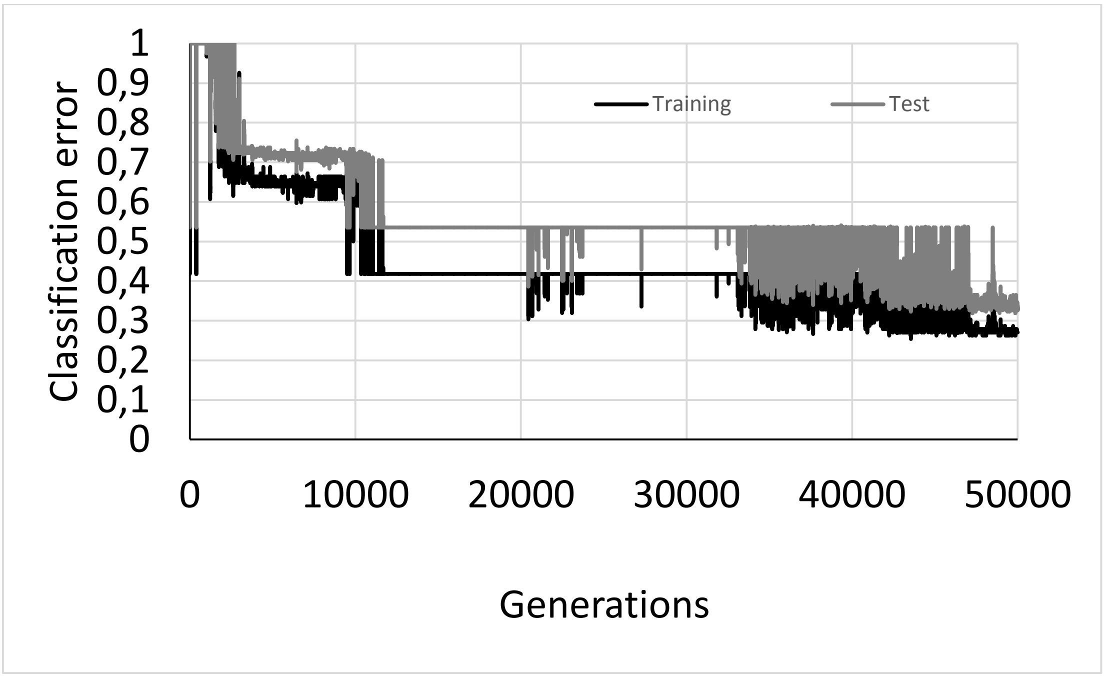

Figure 24: Training evolution for the SPM set for the GA and Summatory fit. The error is shown as the percentage of cases that are not correctly classified.

A curious effect can be observed. While the success rate of the network has raised dramatically, except for the first one, the overall performance of the network with all the neurons online is around 20% worse than before. The reason for this is that the fit of the chromosome with all neurons online is of little significance in comparison to the other twenty fits calculated. The network tunes its weights to work better with only 19 neurons (except in one case) and therefore activating all of them harms its performance.

4.3.2.2 Average fit function

In order avoid the problem of the overall performance, the Summatory fit function is modified the way we see en Equation 19.

$$fit'(x) = fit(x) + \frac{\sum_i^n fit(x_i)}{n}$$

Equation 19: Average fit function

Where *n* is the number of input neurons. By doing the average of the input neurons fit we give equal importance to the overall fit and to the offline neurons fit. To study this fit function we use the following parameters: 37000 generations, mutation and crossover probability 0.8, mutation constant 2.0 and chromosomes 10. In Figure 25 we can see the performance of the disconnected neurons. Figure X shows the evolution of the network.

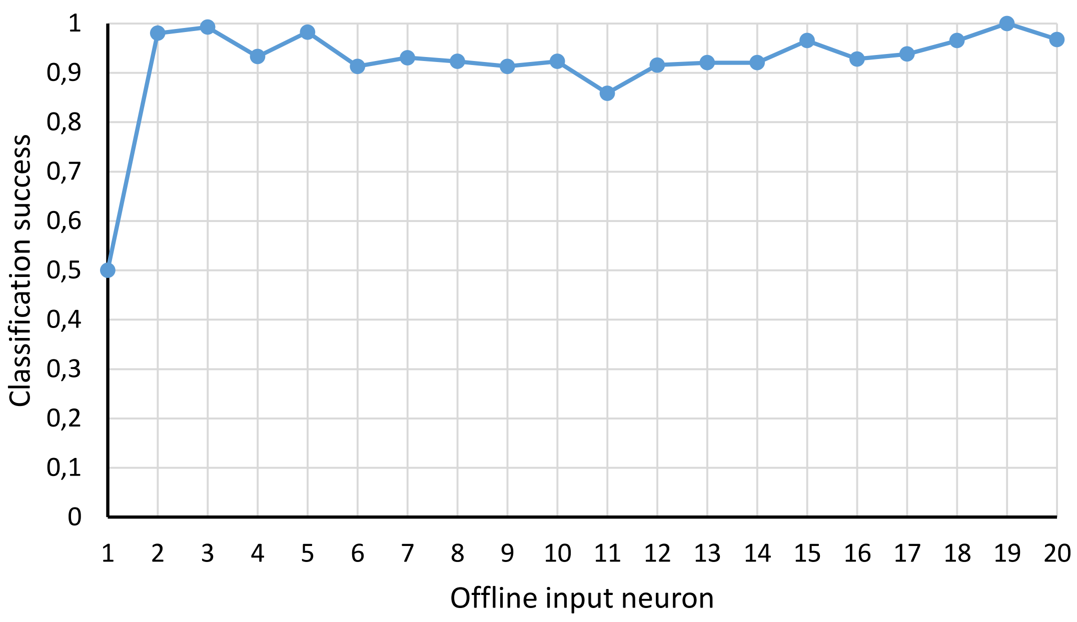

Figure 25: The effect of an input neuron failure is shown. The X axis corresponds to the offline input neuron. The Y axis represents the network's accuracy to classify the samples. The bigger de distance a neuron characterizes, the lesser the effect on the overall output.

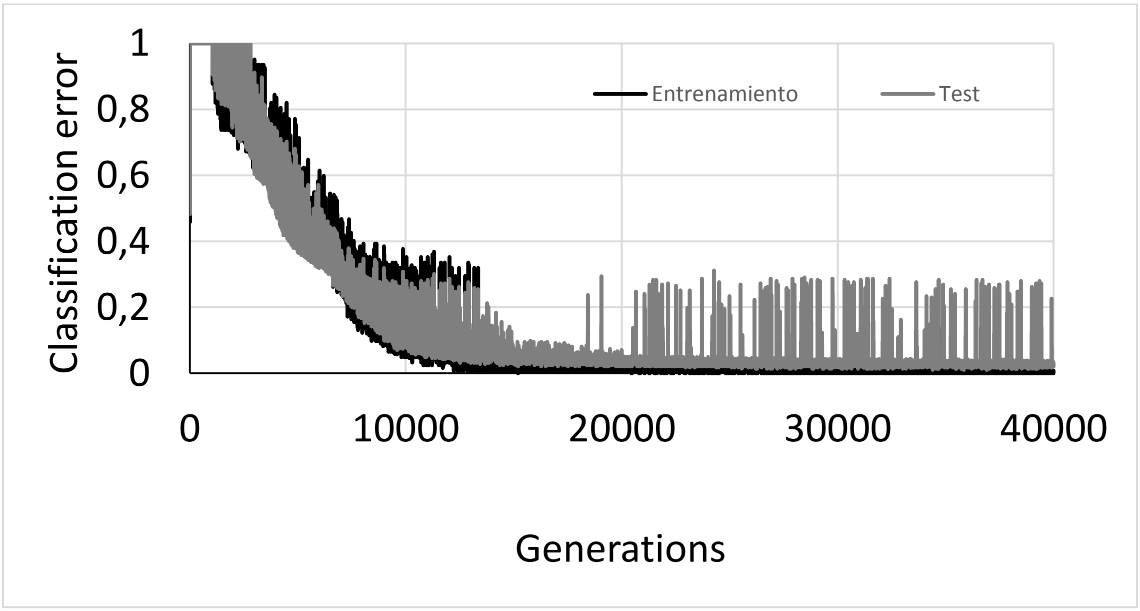

Figure 26: Training evolution for the SPM set for the GA and Average fit. The error is shown as the percentage of cases that are not correctly classified.

Although we might notice that the values of the offline neurons are not as good as they were with the Summatory fit function, it's worth noting that the overall performance is as good as it was with the standard fit function.

4.3.2.3 Power fit function

The two last fit functions had a problem in common. Even if both of them get to raise the accuracy of most input neurons, the two of them fail to do so with the first one as we can see in Figure 23 and Figure 25. In an attempt to solve this we designed the power fit function and the next one, the elimination fit function. The power fit function is shown in the next equation:

$$fit'(x) = f(x) + \sum_i f(x_i)$$
$$f(x) = fit(x)^\gamma$$

Equation 20: Power fit function

Where γ is a number that satisfies the following criteria:

$$A^\gamma + 20 \cdot B^\gamma < C^\gamma \leftrightarrow A < C < B$$

Equation 21: Power fit criteria

If such γ is found, chromosomes with homogeneous individual fits (i.e. without input neurons that harms the network's performance when disconnected) would have a better fit that functions that have very high individual fits but one or more undesired one. Unfortunately we couldn't find a good value for γ and there are no results to show.

4.3.2.4 Elimination fit function

This last fit function has a different approach. When the other ones where mathematical solutions for the problem, the elimination fit function goes with a computational approach. The fit is calculated with the Summatory fit function but while it is calculating the fit of every disconnected neuron the function checks if it is too low. If that is the case, it simply sets the fit of the chromosome to zero. This is the pseudo code for this function:

```
BEGIN
Calculate overall fit
While still input neurons
    Disconnect next input neuron
    Calculate fit for the neuron
    If fit is too low set a flag
    Reconnect the neuron
Loop
If flag is true
    Set fit 0
    Return
Else
    Set fit (overall + input neuron fits)
    Return fit
END
```

Sadly, this function didn't give any results either.

4.3.3 Summary

In conclusion, we have developed a method that is able to reduce the strong dependence of the classification with the presence of the whole data set. This is very important in this case because experimental results may be measured in very different conditions. Specifically, the tip-sample distance is a parameter that cannot be easily estimate and, sometimes, the experiment is done in an environment that does not allow to get data from the whole set of distances. By forcing the FFNN with the EA fit function described before, we have reduced the error in the case of losing some experimental data. This technique is the natural evolution of the previous application, that only analyzed the effect of changing the training method and demonstrates that EA training is also able to tune the behavior of the trained network, without reducing the classification ratio.

5. Conclusion

In this project we have developed a bioinspired method to train FFNN that allows us to discriminate and use individually the different neurons of the network. We have used two different training methods to improve the training performance. We have applied the technique in different applications. The first one is related to a basic FFNN development, and theoretical concepts of both FFNN and EA. We apply our methods just to observe and measure the behavior of the training algorithms. This gives us a new point of view to compare them, taking into account aspects as neuron failure tolerance of the algorithms.

The second one is related to a realistic experimental problem in Nanotechnology. In this application we increase considerably the network's tolerance to missing values. This is of vital importance in this field because, as we said in the last section, is rather difficult to get all the data of a set. We have also demonstrated that this failure tolerance increase does not have any negative effects whatsoever on the classification ratio of the network. This opens the door to lots of possibilities when it comes to combine the special characteristics of our FFNN with the fit function of the EA.

5.1 Future work

Because of the versatility of FFNN and the power of the FFNN+EA combo, the future of the project is open to thousands of possibilities in several fields, from biology to statistics and more. If we chose to deeper in the field of nanotechnology and SPM a good option is the characterization of individual atoms/molecules from SPM images. Images from SPM have a degree of uncertainty when it comes to identifying atoms and molecules. It can be interesting to use AI paradigms to solve this problem in the future.

6. References

[1] B. Mahji and M. Panda P. P. Sarangi, "Performance Analysis of Neural Networks Training using Real Coded Genetic Algorithm," *International Journal of Computer Applications*, no. 51, pp. 30-36, 2012.

[2] S. B. Kotsiantis, "Supervised Machine Learning: A Review of Classification Techniques," *Informatica*, no. 31, pp. 249-268, 2007.

[3] R.S. Sexton J.N.D. Gupta, "Comparing backpropagation with a genetic algorithm for neural network training," *Omega*, no. 27, pp. 679-684, 1999.

[4] X. Yao, "Evolving artificial neural networks," *Proc. IEEEE*, vol. 9, no. 87, pp. 1423-1447, 1999.

[5] X. Fu, Y. Mao, M.I. Menhas, M. Fei L. Wang, "A novel modified binary differential evolution algorithm and its applications," *Neurocomputing*, no. 98, pp. 55-75, 2012.

[6] M. Castellani, "Evolutionary generation of neural network classifiers-An empirical comparison," *Neurocomputing*, no. 99, pp. 214-229, 2013.

[7] M. Hajmeer I. A. Basheer, "Artificial Neural Networks: fundamentals, computing, design and application," *Journal of Microbiological methods*, no. 43, pp. 3-31, 2000.

[8] P. Morgan B. Curry, "Neural networks: a need for caution," *Omega, Internationa Journal of Management sciences*, no. 25, pp. 123-133, 1997.

[9] H. Rowlands M. Castellani, "Evolutionary Artificial Neural Network Design and Training for wood veneer classification," *Engineering Applications of Artificial Intelligence*, no. 22, pp. 732-741, 2009.

[10] D. Whitley, "Applying Genetic Algorithms to Neural Network Problems," *International Neural Networks*, p. 230, 1988.

[11] D. Whitley, L.J. Eshelman J.D. Schaffer, "Combinations of Genetich Algorithms and Neural Networks: A Survey of the State of the Art," *IEEE Computer Society on Combinations of Genetic Algorithms and Neural Networks*, 1992.

[12] H. Bal H.H. Örkcü, "Comparing performances of backpropagation and genetic algorithms in the data classification," *Expert Systems with Applications*, no. 38, pp. 3703-3709, 2011.

[13] H. Shao, Y. Li C. Zhang, "Particle Swarm Optimization for Evolving Artificial Neural Network," *IEEE International Conference on System, Man, and Cybernetics*, no. 4, pp. 2487-2490, 2000.

[14] G M Sacha and P Varona, "Artificial intelligence in nanotechnology," *Nanotechnology*, vol. 24, no. 45, 2013.

[15] Pablo Pou, Oscar Custance, Pavel Jelinek, Masayuki Abe, Ruben Perez, Seizo Morita Yoshiaki Sugimoto, "Complex Patterning by Vertical Interchange Atom Manipulation Using Atomic Force Microscopy," *Science*, vol. 322, no. 5900, pp. 413-417, 2008.

[16] Pablo Varona, Allen I. Selverston, and Henry D. I. Abarbanel Mikhail I. Rabinovich, "Dynamical principles in neuroscience," *Reviews of Modern Physics*, vol. 78, no. 4, pp. 1213-1265, 2006.

[17] Shuqi and Cheng, Hua and Yang, Haifang and Li, Junjie and Duan, Xiaoyang and Gu, Changzhi and Tian, Jianguo Chen, "Polarization insensitive and omnidirectional broadband near perfect planar metamaterial absorber in the near infrared regime," *Applied Physics Letters*, vol. 99, no. 25, 2011.

[18] S. Esmaeilzadehha M. H. Korayem, "Virtual reality interface for nano-manipulation based on enhanced images," *The International Journal of Advanced Manufacturing Technology*, vol. 63, no. 9-12, pp. 1153-1166, 2012.

[19] C Pezeshki, J L McHale and F J Knorr M A Al-Khedher, "Quality classification via Raman identification and SEM analysis of carbon nanotube bundles using artificial neural networks," *Nanotechnology*, vol. 18, no. 35, 2007.

[20] F B Rodríguez and P Varona G M Sacha, "An inverse problem solution for undetermined electrostatic force microscopy setups using neural networks," *Nanotechnology*, vol. 20, no. 8, 2009.

[21] T. B. Ludermir L. M. Alameida, "A multi-objective memetic and hybrid methodology for optimizing the Parameters and performance of artificial neural networks," *Neurocomputing*, no. 73, pp. 1438-1450, 2010.

[22] J.H. Holland, "Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial inteligence," *U Michigan Press*, 1975.

[23] M. Lozano F. Herrera, "Editorial Real Coded genetic algorithms," *Soft. Comput.*, no. 9, pp. 223-224, 2005.

[24] T.B. Ludermir L.M. Alameida, "An Evolutionary Approach for Tuning Artificial Neural Network Parameters, Providing dynamic instructional adaptation in learning programming," *HAIS*, pp. 156-163, 2008.

[25] M. Lozano, J.L. Verdegay F. Herrera, "Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis," *Artificial Intelligence Review*, no. 12, pp. 265-319, 1998.

[26] Radcliffe N.J., "Equivalence Class Analysis of Genetic Algorithms," *Complex Systems*, vol. 5, no. 2, pp. 183-205, 1991a.

[27] A. Wright, , G.J.E Rawlin, Ed., 1991, pp. 205-218.

[28] Z. Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs," *Springer-Verlag*, 1992.

[29] Eshelman L.J.&Schaffer J.D., "Real-Coded Genetic Algorithms and Interval-Schemata," in *Foundation of Genetic Algorithms 2*, L.Darrell Whitley (Ed.), Ed.: Morgan Kaufmann Publishers,San mateo, 1993, pp. 187-202.

[30] Mühlenbein H. & Schlierkamp-Voosen D., "Predictive Models for the Breeder Genetic Algorithm I. Continuous Parameter Optimization," *Evolutionary Computation*, vol. 1, pp. 25-49, 1993.