POLYNOMIAL APPROXIMATION USING PARTICLE SWARM OPTIMIZATION OF LINEAR ENHANCED NEURAL NETWORKS WITH NO HIDDEN LAYERS

Luis F. de Mingo, Miguel A. Muriel, Nuria Gómez Blas, Daniel Triviño G.

Abstract: *This paper presents some ideas about a new neural network architecture that can be compared to a Taylor analysis when dealing with patterns. Such architecture is based on lineal activation functions with an axo-axonic architecture. A biological axo-axonic connection between two neurons is defined as the weight in a connection in given by the output of another third neuron. This idea can be implemented in the so called Enhanced Neural Networks in which two Multilayer Perceptrons are used; the first one will output the weights that the second MLP uses to computed the desired output. This kind of neural network has universal approximation properties even with lineal activation functions. There exists a clear difference between cooperative and competitive strategies. The former ones are based on the swarm colonies, in which all individuals share its knowledge about the goal in order to pass such information to other individuals to get optimum solution. The latter ones are based on genetic models, that is, individuals can die and new individuals are created combining information of alive one; or are based on molecular/celular behaviour passing information from one structure to another. A swarm-based model is applied to obtain the Neural Network, training the net with a Particle Swarm algorithm.*

Keywords: *Neural Networks, Swarm Computing, Particle Swarm Optimization.*

ACM Classification Keywords*: F.1.1 Theory of Computation - Models of Computation, I.2.6 Artificial Intelligence - Learning, G.1.2 Numerical Analysis - Approximation.*

MSC*: 68Q32 Computational learning theory, 68T05 Learning and adaptive systems.*

Introduction

The only free parameters in the learning algorithm are the weights of one MLP since the weights of the other MLP are outputs computed by a neural network. This way the backpropagation algorithm must be modified in order to propagate the Mean Squared Error through both MLPs.

When all activation functions in an axo-axonic architecture are lineal ones ($f(x) = ax + b$) the output of the neural network is a polynomial expression in which the degree n of the polynomial depends on the number m of hidden layers $(n = m + 2)$. This lineal architecture behaves like Taylor series approximation but with a global schema instead of the local approximation obtained by Taylor series. All boolean functions $f(x_1, \cdots, x_n)$ can be interpolated with a axo-axonic architecture with lineal activation functions with n hidden layers, where n is the number of variables involve in the boolean functions. Any pattern set can be approximated with a polynomial expression, degree $n + 2$, using an axo-axonic architecture with n hidden layers. The number of hidden neurons does not affects the polynomial degree but can be increased/decreased in order to obtained a lower MSE.

This lineal approach increases MLP capabilities but only polynomial approximations can be made. If non lineal activation functions are implemented in an axo-axonic network then different approximation schema can be obtained. That is, a net with sinusoidal functions outputs Fourier expressions, a net with ridge functions outputs ridge expressions, and so on. The main advantage of using a net is the a global approximation is achieved instead of a local approximation such as in the Fourier analysis.

A variety of general search techniques can be employed to locate a solution in a feasible solution space, in our case neural network weights. Most techniques fit into one of the three broad classes. The first major class

involves calculus-based techniques. These techniques tend to work quite efficiently on solution spaces with friendly landscapes. The second major class involves enumerative techniques, which search (implicitly or explicitly) every point in the solution space. Due to their computational intensity, their usefulness is limited when solving large problems. The third major class of search techniques is the guided random search. Guided searches are similar to enumerative techniques, but they employ heuristics to enhance the search.

Evolutionary algorithms(EAs) are one of the most interesting types of guided random search techniques. EAs are a mathematical modeling paradigm inspired by Darwin's theory of evolution. An EA adapts during the search process, using the information it discovers to break the curse of dimensionality that makes non-random and exhaustive search methods computationally intractable. In exchange for their efficiency, most EAs sacrifice the guarantee of locating the global optimum. Differential evolution (DE) and Particle Swarm Optimization, see figures 9 and 10, are both stochastic optimization techniques. They produce good results on both real life problems and optimization problems. A simple mixture between those two algorithms, called Differential Evolution - Particle Swarm Optimization (DE-PSO), is also considered. The explanation will no longer use the sine function, but the more frequently used sphere function. Also note that the explanation for this algorithm will not use a single value, but arrays (vectors) to represent particles and velocities. Therefore, it is compatible with more dimensions.

Enhanced Neural Networks

The most usual connection type in neural networks is the axo-dendritic connection. This connection is based on the fact that the axon of an afferent neuron is connected to another neuron via a synapse on a dendrite, and modelized in *ANN* model by a weighted activation transfer function. But, there exists many other connection types as: axo-somatic, axo-axonic and axo-synaptic [Delacour,1987]. This paper is focused on the second kind of connection type *axo-axonic*. Merely, the structure of the axo-axonic connection can be sketched by three neurons with a classical axo-dendritic connection and the synaptic axonal termination of N_3 connected to the synapse S_{12}. The principle consists on propagating the action of neuron N_3 as synapse S_{12}. In order to model previous connection type, two neural networks are required [Mingo,1998]. The first (assistant) one will compute the weight matrix of the second (principal) one. And, the second network will output a response, using the previously computed weight matrix, this architecture is named Enhanced Neural Networks *ENN* [Mingo,1999; Mingo,1999a; Mingo,1999b].

Taylor Approximation

Taylor approximation degree 2 of a function n-differentiable at a point $x = a$ can be obtained using the following expression as a power series:

$$\hat{f}(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + e(\xi) \tag{1}$$

, where ξ belongs to interval $[x, a)$.

If $f'''(x)$ is a continuos function in the closed interval $[a, x]$ then this derivate has a maximun M in such interval, and therefore, the error in the aproximation (equation 1) is measure by [Blum,1991]:

$$\max \left| f'''(x) \right| \leq M \tag{2}$$

$$|e(x)| \leq \frac{1}{6} M \, |x-a|^3 \tag{3}$$

In case an approximation degree n of function $f(x)$ must be obtained, previous equations can be generalized in order to get:

$$\hat{f}(x) = \sum_{i=0}^{n} \frac{f^{i)}(a)(x-a)^i}{i!} + \frac{f^{n+1)}(\xi)(x-a)^{(n+1)}}{(n+1)!} \tag{4}$$

provided following constraints are verified:

1. $f^{i)}(x)$ corresponds to the i-derivate of $f(x)$. Besides $f^{0)}(x) = f(x)$.
2. If $i = 0$ then $i! = 1$.
3. ξ is a point at interval $[x, a)$.

The approximation error, that is $f(x) - \hat{f}(x)$, can be measured if the $(n+1)$-derivate is a continuos function in interval $[a, x)$. Approximation error has a maximum defined by:

$$|e(x)| \leq \frac{1}{(n+1)!} M \, |x-a|^{(n+1)} \tag{5}$$

ENN as Taylor series approximators.

Above section has shown that a function can be approximated with a given error using a polynomial $P(x) = \hat{f}(x)$ with a degree n. The error $f(x) - P(x)$ is measure by equation (5) in such a way that in order to find a suitable approximation (error lower than a known threshold) it is only needed to compute sucessive derivates of function $f(x)$ until a certain degree n.

Enhanced Neural Networks behave as n-degree polynomial approximators depending on the number of hidden layer in the architecture. In order to obtain such behavior all activation functions of the net must be lineal function $f(x) = ax + b$.

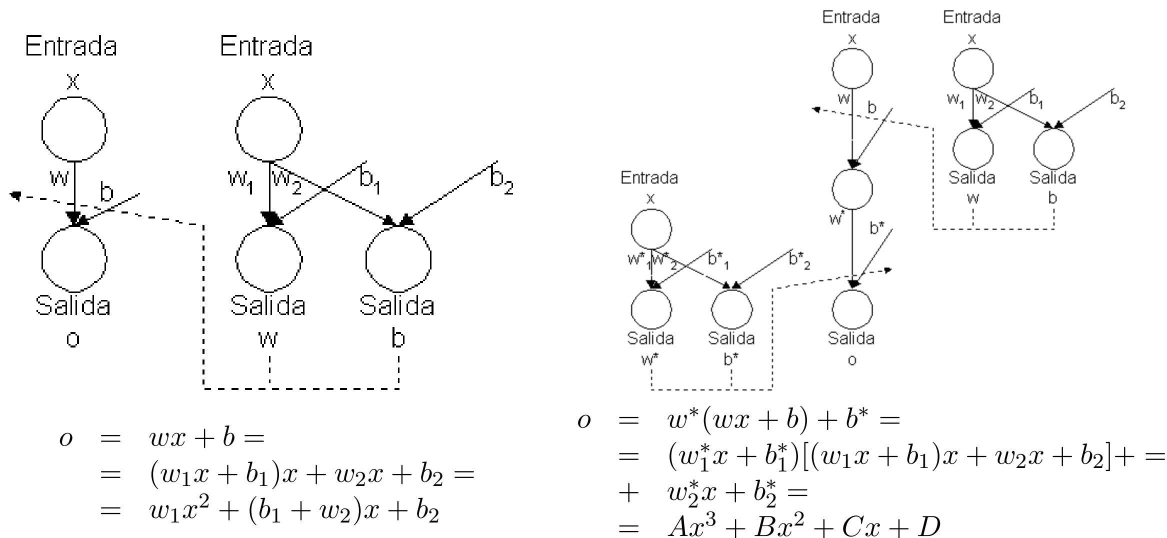


Figure 1: ENN architectures and output expressions

As shown in figure 1 and output equations, the number of hidden layers can be increased in order to increase the degree of the output polynomial, that is, the number n of hidden layers control, in some sense, the degree $n+2$ of output polynomial of the net.

Table 1 shows how the degree of the output polynomial increases according to the number of hidden layers in the net.

Table 1: Number hidden layers vs. degree of output polynomial

Hidden Layers	Degree $P(x)$	Output Polynomial
0	2	$o = a_2x^2 + a_1x + a_0$
1	3	$o = a_3x^3 + a_2x^2 + a_1x + a_0$
...	...	...
n	$n + 2$	$o = \sum_{i=0}^{n+2} a_ix^i$

The only condition that the learning algorithm must verified is that weights must be adjusted to values related with the sucesive derivates of function $f(x)$ that pattern set represents. Usually such function is unkown therefore, if the network converges with a low mean squared error then all weights of the net have converged to the derivates of function $f(x)$ (the pattern set unkown function), and such weights will gather some information about the function and its derivates that the pattern set represents.

As an example, function $f(x) = sen(x)cos(x)$ can be approximated using equation (4), with a given point $a = 0$. Such equation can be reduced to $\tilde{f}(x) = x - \frac{4}{6}x^3$, using a polynomial $P(x)$ degree 3. This is a mathematical approach, but what happens if such function is the pattern set to an enhanced neural network mentioned before?.

A one hidden layer neural network must be used in order to obtain a 3-degree polynomial as the output expression. Figure 2 shows such architecture, after the training stage, the final configuration is shown. Output equation of the net is $o = x - \frac{4}{6}x^3$, equivalent equation with $\tilde{f}(x)$.

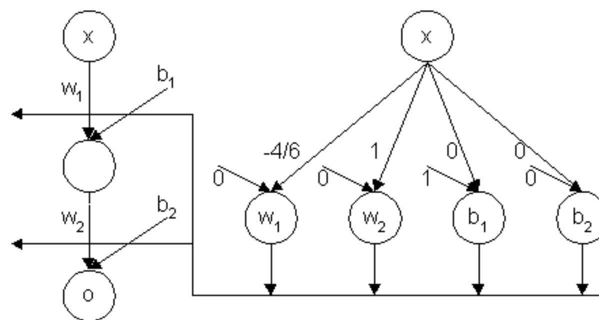


Figure 2: Approximation of $f(x) = sen(x)cos(x)$ with a one hidden layer

The approximation error using net in figure 2 can be computed using equation (5), and therefore $MSE \leq |e(x)|$. Such approximation is not the only one nor the best one, but it can be computed theoretically in order to provide the net some initial weights in order to speed up the learning process and to obtain a better approximation that the initial one with a lower error ratio. In sumary, Enhanced Neural Networks can be initialized to some weights computed using the Taylor Series of the function that the pattern set defines and after this initial stage the learning algorithm must be applied in order to achieved the best solution (the one that improves the Taylor Series error).

Figure 3 shows the surface computed by a net as the number of hidden layers is increased. The mean squared error is decreasing as the number of hidden layers goes up. This figure shows that this kind of neural net is very suitable when approximating functions, a given function or a function defined by the pattern set.

Non-Linear Activation

According to previous ideas, *linear ENNs* are better than linear *MLPs*, or at least, they are able to generate complex regions in order to divide the output space. When working with a *MLP*, only hyperplanes can be obtained. And moreover, the degree of the output equation increases according to the number of hidden layers.

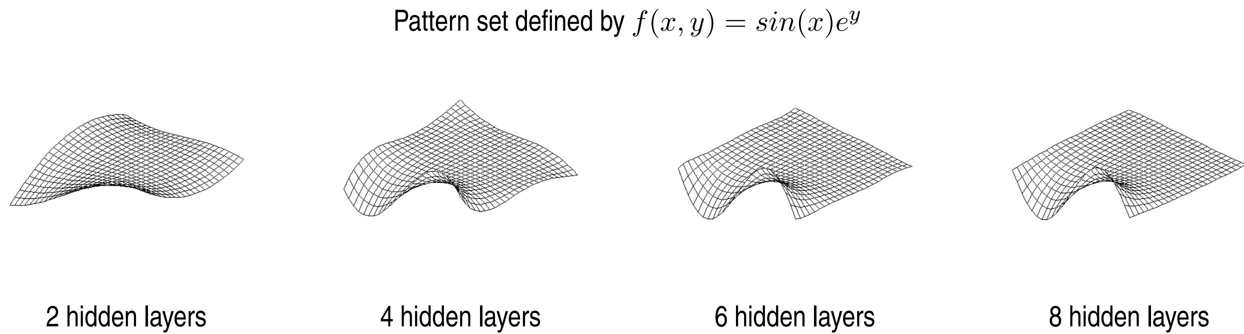


Figure 3: Surface approximation depending on the number of hidden layers

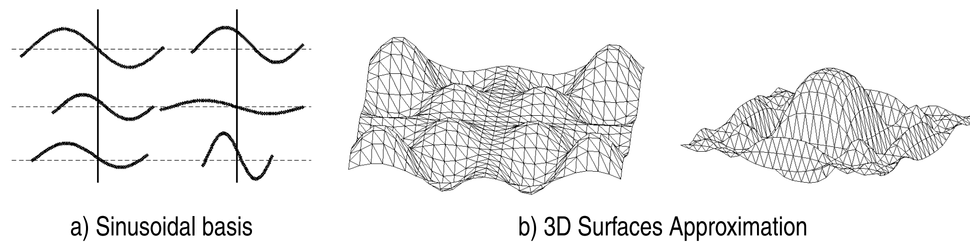


Figure 4: Approximation with sinusoidal activation functions using basis of figure a).

In order to obtain a functional basis, one constraint must be made. It consists on implementing the network architecture with lineal *PE*s except the output neurons of assistant network. These neurons must have an activation function $g(x)$ which is used to computed the functional basis as the application of $g(x)$ to a non lineal combination of inputs. Figure 4 shows an example of a functional basis and the main network ourput.

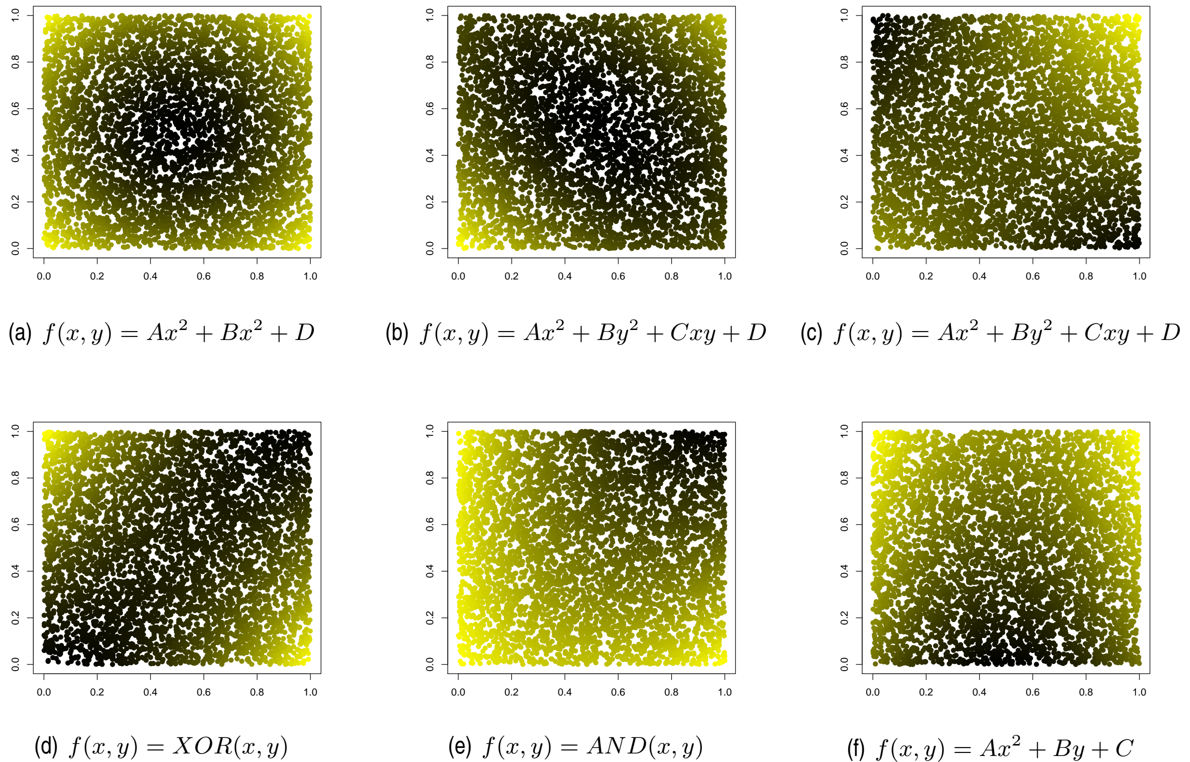
Depending on the activation function of output neurons belonging to assistant network, the main network will output an approximation function based on non lineal combination of elements belonging to the basis. That is if a sinusoidal activation function is implemented, then a cuasi-Fourier approximation is computed by the network; is a Ridge activation function is implemented, then a cuasi-Ridge approximation is computed and so on.

Main advantage of this new approximation method is that is absolutely easy to implement. And moreover, a global approximation to all the pattern set is perform. This way, if there are enough input patterns, then the generalization error will be minimized if there are enough learning iterations.

Enhanced Neural Networks as Universal Approximators

Along the paper [Mingo,1999a], this new architecture has shown that it is very suitable when dealing with any problem. Decision surfaces generated by the net are complex enough to represent any data set. The powerfull of these nets is in the number of hidden layers, that is, in the degree of the output polinomial associated to one output unit.

Funahashi Theorem can be directly apply to *Enhanced Neural Networks* in order to proof the universal approximation property of proposed networks, provided that activation function in hidden and output neurons belongs to a given class of functions stated by *Funahashi*. This way, *ENN* behave as universal approximators, that is, they are able to learn any pattern set.



(a) $f(x,y) = Ax^2 + Bx^2 + D$ (b) $f(x,y) = Ax^2 + By^2 + Cxy + D$ (c) $f(x,y) = Ax^2 + By^2 + Cxy + D$

(d) $f(x,y) = XOR(x,y)$ (e) $f(x,y) = AND(x,y)$ (f) $f(x,y) = Ax^2 + By + C$

Figure 5: Network output corresponding to 2-degree polynomial functions using an enhanced neural network with no hidden layers.

Results using Linear Networks with no hidden layers

The enhanced neural network architecture has the property that the output equation of an output unit is a n-degree polynomial if the activation function is a lineal one. This output can be compared with the Taylor approximation polynomials since the both methods are very similar. A set of data can be approximated by ENNs, computing a n-degree polynomial as the network output. But, the activation function can be a sinusoidal, instead being a lineal one. With this function, the approach of ENN is similar to Fourier series decomposition. This way, the activation function can be changed in order to get a better approximation than in the case of MLPs.

Figure 5 shows that the proposed network is able to learn different surfaces in a 2D space with a low MSE. This is mainly due to the special architecture of the net. The input to the net affects to the weights in the connection, and even changes them in order to optimize the error achieved by the net.

In a more complex pattern set, that is, a high dimension space, the proposed architecture is also stable, see figue 6 and note the correlation among all inputs and the correlation between the real output (OUT.1) and the desired response (OUT). Table 2 shows the final weights of the network.

Particle Swarm Optimization of Enhanced Neural Networks

Starting form general Particle Swarm Optimization algorithms formulas:

$$v_d^{(i)} = v_d^{(i)} + c_1\epsilon_1(p_d^{(i)} - x_d^{(i)}) + c_2\epsilon_2(g_d^{(i)} - x_d^{(i)}) \quad (6)$$

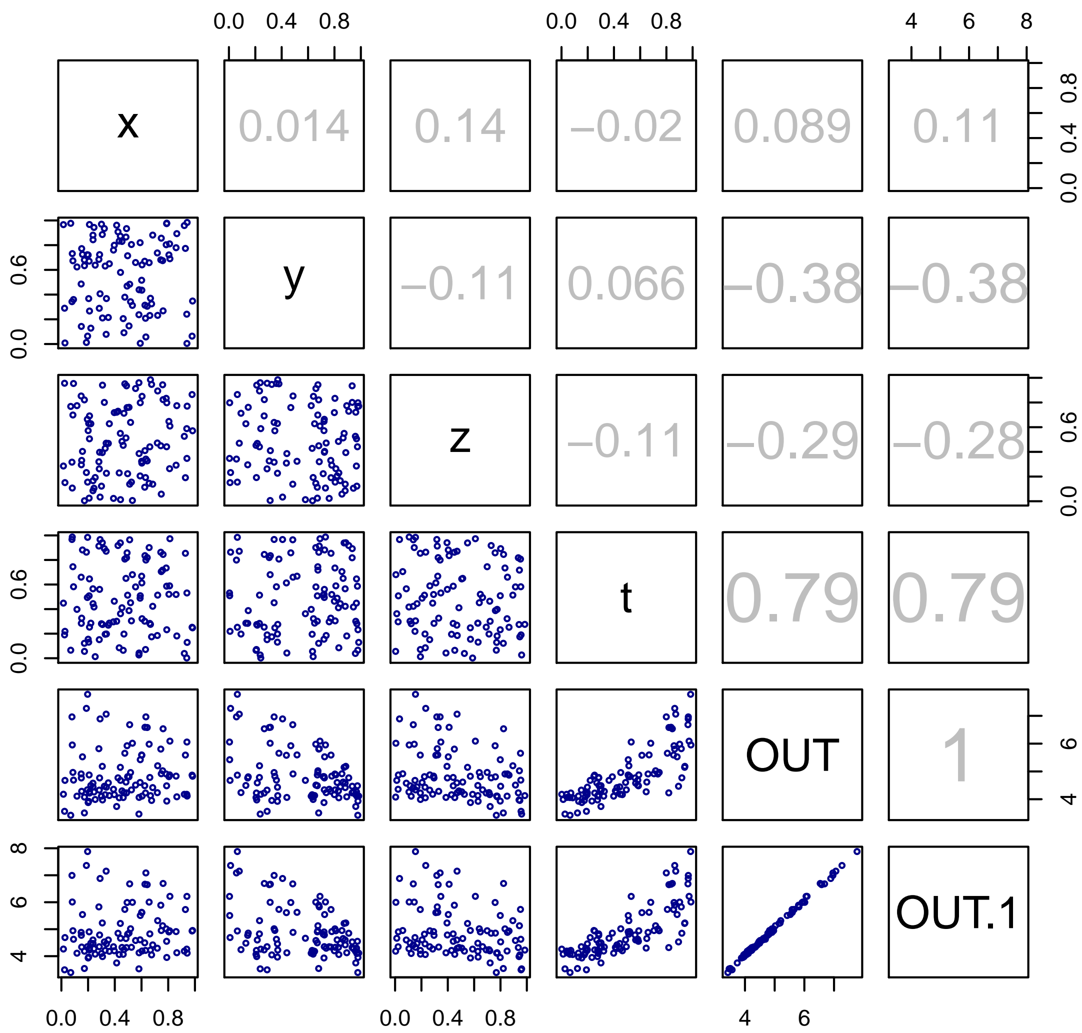


Figure 6: Correlation matrix of $f(x, y, z, t) = x^2 z - (y + 1)^2 t - z^2 + (t + 2)^2$, where $OUT = f(x, y, z, t)$ is the desired output and $OUT.1$ is the network output.

$$x_d^{(i)} = x_d^{(i)} + v_d^{(i)} \quad (7)$$

Regarding the PSO algorithm, different variants have been developed. Most of them aimed at speeding up the convergence of it. In addition to the unconstrained optimization problem in discrete or continuous variable, the multi target problem and the constrained problem have been addressed. We have also developed some hybrid optimization techniques. PSO technique has been tested with good results for training Artificial Neural Networks. When applying the method of Back Propagation, we are able to find appropriate weights that minimize an error function through a succession of iterations. Furthermore, by applying the PSO technique, the weights found are more efficient just by making small modifications to the algorithm. The new guidelines are aimed at avoiding PSO stagnation of the local optimal solutions.

Shi and Eberhart [Shi,1998] proposed adjustments to the velocities of the particles by using a factor w called *inertial weight*. This factor utilizes the inertia of the particles in the process of friction when they are moving. This modification in the algorithm is done to control the search space. In order to do that it must change (8).The large inertia weight makes the global search easier; however small inertia weight does not improve local search. That is why was the initial value is greater than 1.0 to promote global exploration, and then gradually decreases to obtain more refined solutions. The algorithm decreases linearly at each iteration. Moreover, the use of inertial weight removes the restriction V_{max} on the velocity.

$$v_d^{(i)} = wv_d^{(i)} + c_1\epsilon_1(p_d^{(i)} - x_d^{(i)}) + c_2\epsilon_2(g_d^{(i)} - x_d^{(i)}) \quad (8)$$

In each iteration, inertia weight decrease linearly through the following expression:

$$w = w_{max} - (w_{max} - w_{min})\frac{g}{G} \quad (9)$$

g is the index of the generation, G is the maximum number of iterations previously determined, w_{max} is a value greater than 1, and w_{min} a value under 0.5. This variation of the method has proven to accelerate convergence.

Clerc and Kennedy [Clerck,2002] obtain another variation in the speed calculation. A constriction factor χ is introduced with that purpose, This factor depends on the constants that are used when calculating speed and it affects to the formula (6) The aim is to avoid the explosion of velocity:

$$v_d^{(i)} = \chi[v_d^{(i)} + c_1\epsilon_1(p_d^{(i)} - x_d^{(i)}) + c_2\epsilon_2(g_d^{(i)} - x_d^{(i)})] \quad (10)$$

χ is:

$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}, \quad \varphi = c_1 + c_2 = 4.1 \quad (11)$$

Table 2: Weight matrix corresponding to the auxiliary network when learning function $f(x, y, z, t) = x^2z - (y + 1)^2t - z^2 + (t + 2)^2$.

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	0.69392712	-0.03879246	0.636080632	0.1224748	0.3380812
[2,]	-0.07873902	-0.55159000	-0.009702458	-1.9004765	-0.4532731
[3,]	0.54488180	-0.02171687	-1.068343779	0.1584132	0.2621986
[4,]	0.06375846	-1.23761019	-0.131404413	1.0906522	-1.2437301
[5,]	0.36486673	-0.22649428	-0.091435548	-1.8485381	4.0222552

Function	Dim.	Search space.	Name
$f_1(x) = \sum_{i=1}^{d} x_i^2$	30	[-100,100]	Sphere/Parabola
$f_2(x) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	30	[-30,30]	Rosenbrock Generaliz.
$f_3(x) = \sum_{i=1}^{d} (\sum_{j=1}^{i} x_j)^2$	30	[-100,100]	Schwefel 1.2
$f_4(x) = \sum_{i=1}^{d} cos(x_i)^2$	30	$(-\infty, \infty)$	
$f_5(x) = \sum_{i=1}^{d} (x_i^2 - 10cos(2\pi x_i) + 10)$	30	[-5.2, 5.2]	Rastrigin Generaliz.
$f_6(x) = -\sum_{i=1}^{d} x_i \sin(\sqrt{\lvert x_d \rvert})$	30	[-500,500]	Schwefel Generaliz. 2.6

Table 3: Functions tested with PSO algorithm

The results are: $\chi = 0.729$ and c_1=c_2= 2.05. These parameters were obtained by performing several tests.

χ factor is similar to the inertial weight. This means that controlling the velocity with $V(max)$ is not required when χ is used. Bratton and Kennedy [Bratton,2007], analyzed the stability of this algorithm by using these values and by following a comparative study of both PSO algorithms (inertial weight and χ factor). Both of them are mathematically equivalent, in particular the algorithm with constriction factor is a special case of the inertial weight. Moreover, Parsopoulos al [Parsopoulos,2002], combined both for problems with constraints and they obtained equally good results in several tests.

We observe that the convergence always becomes slower when problem size increase, so when it comes to high-dimensional problems, a larger number of iterations occurs. Researchers Hatanaka et al [Hatanaka,2007] developed a PSO model, where velocity values are updated, by considering the rotation of the coordinate system. This model is aimed at problems of high dimensionality and it showed good results when applied to all functions of De Jong, (larger dimensions).

In order to test the standard PSO algorithm and two variants with incorporated and inertial weight factor χ, we have used some unrestricted functions which are commonly referred as *De Jong functions*'. The minimum of these functions is located in the search space. They were originally proposed by de Jong to measure the performance of genetic algorithms. However they have also been used to test the performance in PSO algorithms. Some of the other functions are unimodal and multimodal i.e Ring (*lbest*) and star (*gbest*) topologies. In the ring topology each particle is related to its two neighbours. In the star topology all particles are interconnected. A population of 20 particles was considered. Table 3 functions are tested with the standard PSO algorithm and two variations: inertia weight and constriction factor. The first three functions are unimodal and have the optimal solution $x^* = 0.0^d$ and the minimum value $f_i(x^*) = 0.0$. The following are multimodal functions, the function f_4 has the minimum value 0.0, the optimal solution is $\pm n\frac{\pi}{2}^d$, the function f_5 has optimal solution $x^* = 0.0^d$ and the minimum value of the function: $f(x^*) = 0.0$ and f_6 has the optimal solution $x^* = 420.968^d$ and the minimum value of the function : $f(x^*) = -12.569, 4866$.

Tables (4) and (5) show the results after 20 executions of the standard algorithm. Not only the inertial weight has been modified in this algorithm but also the constriction factor for each functions by using neighbourhood models *lbest* and *gbest*. The algorithm stops when two successive values of the best assessments of the swarm get close to each other. (A ϵ value is prefixed and so it is a maximum number of iterations). NPE (average number of assessments) shows the average number of evaluations for the function when applying PSO and its variants.

After PSO Algorithm and its variants are executed, results are collected; best results are found in approximately equal number of cases regardless the model is used (*lbest* or *gbest*), so we can not assure which topology is optimal. Moreover, we notice that when using the constriction factor, the convergence accelerates and results are better when compared to the exact solution. In some cases, we notice that the region in which the swarm of particles initially are, can affect the results. Regarding the number of particles of the swarm, when increased to more than 20, results did not improve.

Name	PSO Original		Weight Inertial		Factor Constriction	
	Best Solut.	NPE	Best Solut.	NPE	Best Solut.	NPE
f_1	**l**: 3,70889e-07 **g**: 5,9803e-04	190.480 2×10^5	7,68359e-07 2,45656e-20	28.800 151.500	2,69078e-08 3,63055e-20	19.100 30.820
f_2	**l**: 5,60357e-06 **g**: 4,51068e-02	2×10^5 4×10^6	2,2112e-05 9,24376e-12	831.580 4×10^6	4,5089e-06 2,59676e-09	1.257.520 5×10^5
f_3	**l**: 5,76231e-06 **g**: 4,91261e-06	198.280 2×10^5	5,57901e-15 1,81786e-12	1.425.900 2.596.780	2,83076e-16 1,44177e-13	399.140 125.640

Table 4: Results obtained by applying the PSO algorithm to unimodal functions, l indicates the model *lbest*; g refers to model *gbest*.

Name	PSO Original		Weight Inertial		Factor Constriction	
	Best solut.	NPE	Best solut.	NPE	Best solut.	NPE
f_4	**l**: 2.18719e-05 **g**: 3,9543e-12	2×10^5 180.820	4.36373e-19 1.3000e-12	114.560 15.120	6,7306e-19 1.7063e-15	4.174 14.160
f_5	**l**: 1,70688e-14 **g**: 3,00262e-04	216.240 2×10^5	1,81227e-14 6,82288e-12	339.020 12.180	1,07181e-11 2,81599e-12	9.480 11.040
f_6	**l**: -12.568,2 **g**: -12.569,5	2×10^5 6×10^5	-12.569,5 -12.569,5	2×10^5 6×10^5	-12.569,5 12.352,3	2×10^5 2×10^5

Table 5: Results obtained by applying the PSO algorithm to unimodal functions, l indicates the model *lbest*; g refers to model *gbest*.

Particle Swarm Optimization and Neural Networks

Particle swarm optimization can be applied to solve many problems. One of them could be the training of a neural network architecture: Given a neural architecture, the problem is to find weights that minimize the mean squared error of the net. Individuals code weights of the neural network, and the fitness function corresponds to the mean squared error. According to *Kolmogorov* a multilayer perceptron can approximate any function even when the number of hidden neurons is unknown.

Obviously, a neural network with i input neurons, h hidden neurons and o output neurons it has $(i+1)h+(h+1)o$ weights and therefore, individuals of the PSO have $(i+1)h+(h+1)o$ dimensions. By considering such number, any real application with neural networks has at least 20 weights. A classical particle swarm algorithm could be applied however individuals have a high dimension and then convergence depends on the random initialization.

Figure 7 shows the learning curve of the PSO algorithm applied to a XOR neural network. This network has a $2-2-1$ architecture. It can be seen that the random initialization of individuals affect the convergence process (columns of figure). And the number of iterations (100 or 1000, at each row) achieves a lower fitness (mean squared error). Anyway, this simple example is solved with 10 individuals in the population, with dimension 9.

Another example is a binary coding neural network. An exclusive 8-bit vector coded it in a 3-bit vector. This classical problem can be solved by using a multilayer perceptron with 3 hidden neurons. Table below shows the input/output patterns of the neural network and the final weights found applying the PSO algorithm. In this case the dimension of individuals is 39 with a population of 15 individuals.

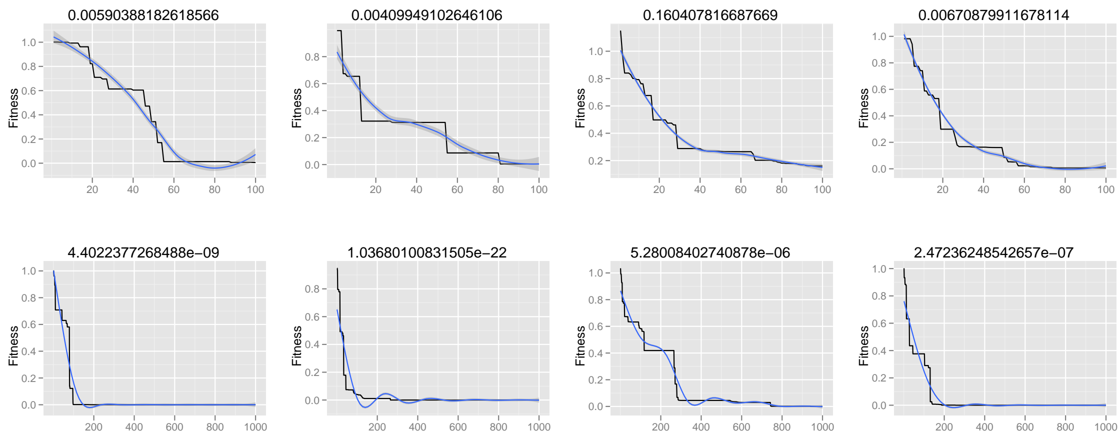


Figure 7: XOR multilayer perceptron with 2 hidden neurons and a particle swarm optimization learning using 10 individuals (individuals have 9 dimensions). Each column represents a different random initialization and each row a number of iterations (100 and 1000).

Input								Output		
1	1	1	1	1	1	1	-1	1	1	1
1	1	1	1	1	1	-1	1	1	1	-1
1	1	1	1	1	-1	1	1	1	-1	1
1	1	1	1	-1	1	1	1	1	-1	-1
1	1	1	-1	1	1	1	1	-1	1	1
1	1	-1	1	1	1	1	1	-1	1	-1
1	-1	1	1	1	1	1	1	-1	-1	1
-1	1	1	1	1	1	1	1	-1	-1	-1

Best fitness value: 5.067e-05

Best neural network weights with a $8-3-3$ architecture:

Input layer $\rightarrow$ Hidden layer

	0.1156528	1.097272	-0.946379977
	-0.3683220	25.945492	-1.703378035
	1.5325933	-9.765752	-0.636187430
	0.4830886	26.536611	0.002121948
	-1.5133460	0.790667	-0.131921926
	-1.7465932	-3.369892	0.987214704
	1.0519552	-4.920479	0.005404300
	0.3397246	-1.924014	3.273439670
Bias:	0.7092471	-5.304714	-0.331283300

Hidden layer $\rightarrow$ Output layer

	1.261584	-4.759320	0.9853185
	148.541038	-1.054461	-3.1161444
	-162.388447	-2.017318	-3.4691962
Bias:	0.090192	1.207254	1.9260548

These two neural examples have shown that the PSO can be successfully applied to the particle swarm algorithm in order to solve, in some way, the convergence of the algorithm when dealing with high dimension individuals.

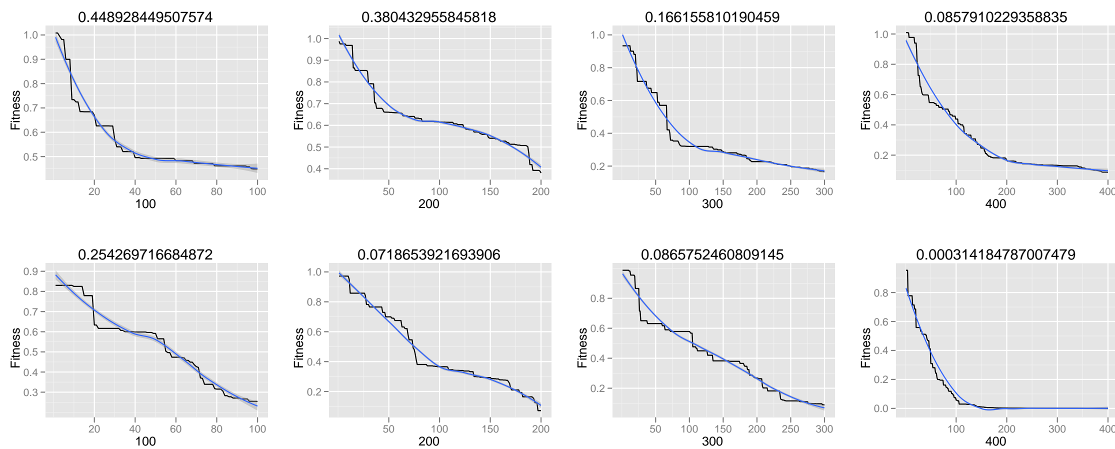


Figure 8: Binary coding neural network form 8 inputs to 3 outputs. First row is a neural network with 3 hidden neurons and second row a neural network with 5 hidden neurons. Mean squared error (fitness of the PSO) decreases as the number of iterations (100 to 400) increases.

The XOR example with dimension 9 and the binary-coding example with dimension 39 are a good starting point to combined classical neural networks with swarm intelligence.

Conclusion

The problem of nonlinear constrained optimization arises frequently in engineering. In general it does not have a deterministic solution. In the past, nonlinear optimization methods were developed and now it is a challenge to work with differentiable functions. Before gradient methods were used successfully for solving some problems. Evolutionary methods provide a new possibility for solving such problems. The PSO technique has been used successfully in optimizing real functions without restrictions, but it has been little used for problems with restrictions. This has happened mainly because there are no mechanism to incorporate restrictions on the *fitness* function. Evolutionary Computation has tried to solve the constrained optimization problem, either by bypassing nonfeasible solutions sequences, or by using a penalty function for nonfeasible sequences. Some researchers suggest to use two subfunctions of *fitness*. One helps to evaluate feasible elements and the other one evaluates the unfeasible one. In this regard, there are many criteria. Moreover, some special self adaptive functions have been designed to implement the penalty technique.

Hu and Eberhart [Hu,2002] presented a PSO algorithm. This algorithm bypasses nonfeasible sequences. it also creates a random initial population, in which nonfeasible sequences are bypassed until the entire population has only feasible particles. By upgrading the positions of the particles nonfeasible sequences are bypassed automatically. The cost of the technique that creates the initial populations is high; especially when it comes to problems with nonlinear constraints because then it must create an entire population of feasible individuals. In his work, Cagnina et al [Cagnina,2008] proposed the following strategies for implementing the PSO into problems with restrictions: **a)** If two particles are feasible, select the one with the best *fitness*. **b)** When a particle is feasible and the other is not, the feasible one is chosen. **c)** If two particles are nonfeasible, the one with the lowest degree of nonfeasibility is selected. These strategies are applied when the particles *gbest* and *lbest* are selected. The same authors also proposed an update in (6). This update considers three elements:

1) $p_d^{(i)}$ which is the best position reached by the particle i in its history. 2) $g_d^{(i)}$ which is the best position reached by the particles in its neigborhood and t_d which is the best position achieved by any particle in the whole swarm.

$$v_d^{(i)} = w(v_d^{(i)} + c_1\epsilon_1(p_d^{(i)} - x_d^{(i)}) + c_2\epsilon_2(g_d^{(i)} - x_d^{(i)}) + c_3\epsilon_3(t_d - x_d^{(i)})) \tag{12}$$

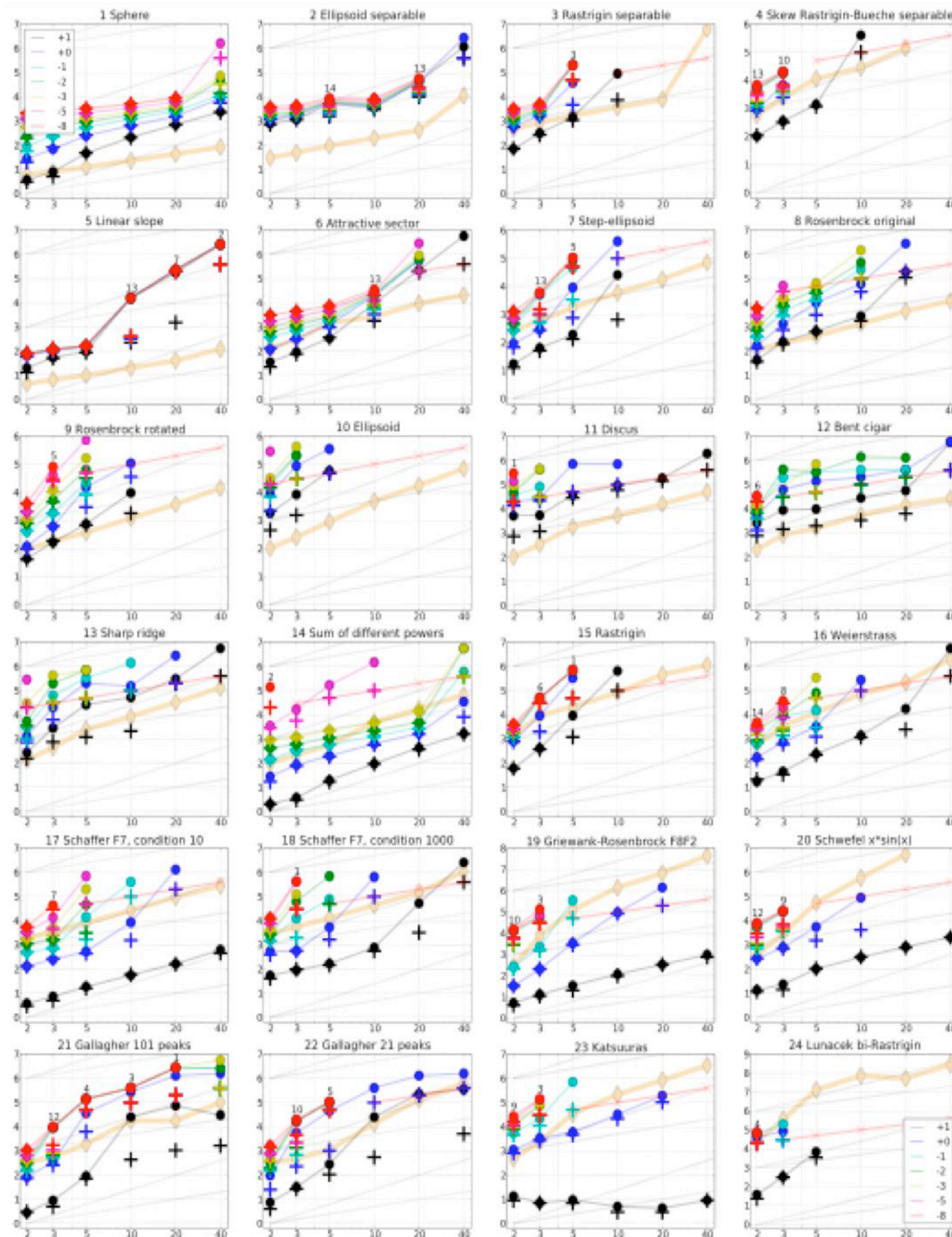


Figure 9: PSO Expected Running Time (ERT, ●) to reach $f_{\mathrm{opt}} + \Delta f$ and median number of f-evaluations from successful trials (+), for $\Delta f = 10^{\{+1,0,-1,-2,-3,-5,-8\}}$ (the exponent is given in the legend of f_1 and f_{24}) versus dimension in log-log presentation. For each function and dimension, $\mathrm{ERT}(\Delta f)$ equals to $\#\mathrm{FEs}(\Delta f)$ divided by the number of successful trials, where a trial is successful if $f_{\mathrm{opt}} + \Delta f$ was surpassed. The $\#\mathrm{FEs}(\Delta f)$ are the total number (sum) of f-evaluations while $f_{\mathrm{opt}} + \Delta f$ was not surpassed in the trial, from all (successful and unsuccessful) trials, and $f_{\mathrm{opt}}$is the optimal function value. Crosses ($\times$) indicate the total number of f-evaluations, $\#\mathrm{FEs}(-\infty)$, divided by the number of trials. Numbers above ERT-symbols indicate the number of successful trials. Y-axis annotations are decimal logarithms. The thick light line with diamonds shows the single best results from BBOB-2009 for $\Delta f = 10^{-8}$. Additional grid lines show linear and quadratic scaling.

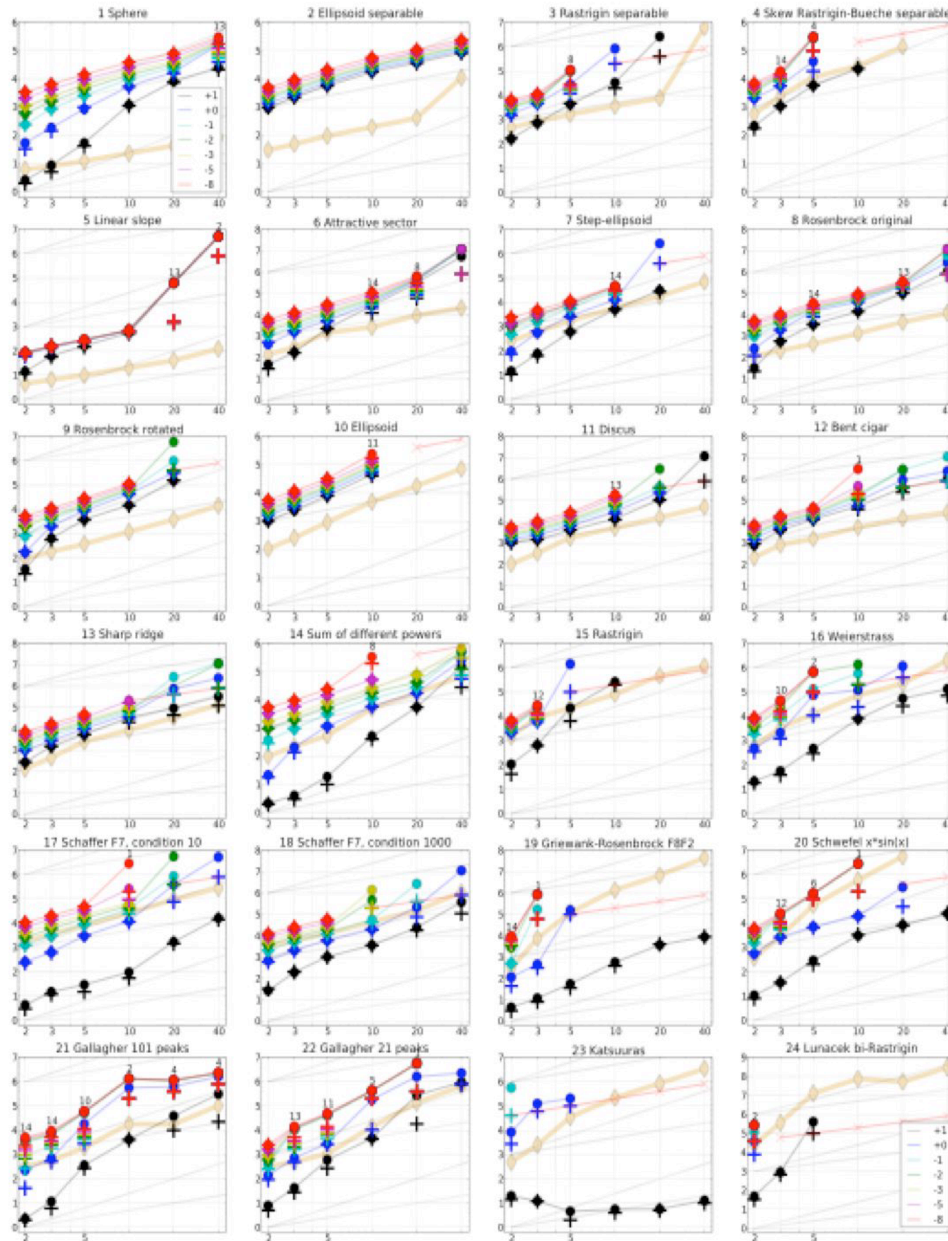


Figure 10: DE-PSO Expected Running Time (ERT, ●) to reach $f_{\mathrm{opt}} + \Delta f$ and median number of f-evaluations from successful trials ($+$), for $\Delta f = 10^{\{+1,0,-1,-2,-3,-5,-8\}}$ (the exponent is given in the legend of f_1 and f_{24}) versus dimension in log-log presentation. For each function and dimension, $\mathrm{ERT}(\Delta f)$ equals to $\#\mathrm{FEs}(\Delta f)$ divided by the number of successful trials, where a trial is successful if $f_{\mathrm{opt}} + \Delta f$ was surpassed. The $\#\mathrm{FEs}(\Delta f)$ are the total number (sum) of f-evaluations while $f_{\mathrm{opt}} + \Delta f$ was not surpassed in the trial, from all (successful and unsuccessful) trials, and f_{opt} is the optimal function value. Crosses ($\times$) indicate the total number of f-evaluations, $\#\mathrm{FEs}(-\infty)$, divided by the number of trials. Numbers above ERT-symbols indicate the number of successful trials. Y-axis annotations are decimal logarithms. The thick light line with diamonds shows the single best results from BBOB-2009 for $\Delta f = 10^{-8}$. Additional grid lines show linear and quadratic scaling.

c_1 is the personal learning factor and c_2 and c_3 are the social learning factors. According to Michalewicz et al [Michalewicz,1998] and [Michalewicz,1996] constrained optimization methods are classified as:

1. Methods based on preserving feasibility of solutions.
2. Methods based on penalty functions
3. Methods that make a clear distinction between feasible solutions and infeasible sequences.
4. Methods based on decoders
5. Hybrid methods

We propose to analyze the penalty methods under E.A. perspective (Evolutionary Algorithms). The penalty methods use functions (penalty functions) that degrade the quality of the nonfeasible solution. In this way the constrained problem becomes a problem without constraints by using a modified evaluation function:

$$eval(x) = \begin{cases} \text{f(x)} & x \in \mathcal{F} \\ \text{f(x) + } \textit{penalty}\text{(x)} & eoc \end{cases} \tag{13}$$

$\mathcal{F}$ is the set created by the intersection of all sets that are the restrictions of the problem (Feasible region). The penalty is zero if no violation occurs and it is positive otherwise. The penalty function is based on the distance between a nonfeasible sequence and the feasible region $\mathcal{F}$, It also works for repairing solutions outside of the feasible region $\mathcal{F}$.

There are many penalty methods. The main difference between the methods is the way the penalty function is designed and applied to the nonfeasible sequences. Some methods associate a penalty function f_j, $(j = 1, \ldots, m)$ with a constraint, which measures the violation of the restriction j as follows:

$$f_j(x) = \begin{cases} max\{0, g_j(x)\}, & si \;\; 1 \le j \le p \\ |h_j(x)| & si \;\; p+1 \le j \le m \end{cases} \tag{14}$$

Acknowledgements

The research was supported by the Spanish Research Agency projects TRA2010-15645 and TEC2010-21303-C04-02.

Bibliography

[Shi,1998] Shi, Y., Eberhart, R., A Modified Particle Swarm Optimizer, Proc. IEEE World Congr. Comput. Intell., 1998, pp. 69

[Clerck,2002] Clerc, M., Kennedy, J.,*The particle Swarm: Explosion, Stability, and Convergence in a Multidimensional Complex Space*,IEEE Trans. Evol. Comput., vol. 6, no. 1, pp. 5873, Feb. 2002.

[Bratton,2007] Bratton, D., Kennedy, J.,*Defining a Standard for Particle Swarm Optimization*.Proceedings of the 2007 IEEE Swarm Intelligence Symposium (SIS 2007).

[Parsopoulos,2002] Parsopoulos, K.E., Vrahatis, M. N.,Particle Swarm Optimization Method for Constrained Optimization Problems, 2002.

[Hatanaka,2007] Hatanaka, T., Korenaga, T., Kondo, N., Uosaki, K.*Search Performance Improvement for PSO in High Dimensional Space*,2007.

[Hu,2002] Hu, X., Eberhart, R., *Solving Constrained Nonlinear Optimization Problems wirh Particle Swarm Optimization*. Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informaticsics, SCI2002 , Vol. 5, IIIS, July 2002.

[Cagnina,2008] Cagnina, L.C., Esquivel,S. C., Coello, C.,*Solving Engineering Optimization Problems with the Simple Constrained Particle Swarm Optimizer*, 2008.

[Michalewicz,1998] Michalewicz, Z. , Fogel, D., *How to solve it. Modern Heuristics*. Springer,1998.

[Michalewicz,1996] Michalewicz, Z. , Shoenauer, M., *Evolutionary Algorithms for Constrained Parameter Optimization Problems*. Evolutionary Computation, 1996, Vol. 4, pp. 132.

[Delacour,1987] Delacour, J.; Apprentissage et Memoire: Une Approache Neurobiologique. Masson(Ed.) September. (1987).

[Mingo,1999] Mingo L.F., Arroyo F., Luengo C., Castellanos J.; *Learning HyperSurfaces with Neural Networks*. 11th Scandinavian Conference on Image Analysis. SCIA'99. June 7-11. Kangerlussuaq, Greenland. Pp: 731-737. 1999.

[Mingo,1999a] Mingo L.F., Arroyo F., Luengo C., Castellanos J.; Enhanced Neural Networks and Medical Imaging. 8th International Conference on Computer Analysis of Images and Patterns. CAIP'99. September 1-3. Ljubljana, Slovenia. 1999.

[Mingo,1999b] Mingo L.F., Giménez V., Castellanos J.; Interpolation of Boolean Functions with Enhanced Neural Networks. Second Conference on Computer Science and Information Technologies. CSIT'99. August 17-22. Yerevan, Armenia. 1999.

[Mingo,1998] Mingo L.F., Castellanos J., Giménez V.; A New Kind of Neural Networks and Its Learning Algorithm. Information Processing and Management of Uncertainty in Knowledge Based Systems. IPMU'98. Paris, France. July 6-10. Pp: 1913-1914. 1998.

[Blum,1991] Blum, E. K. & Leong, L.: Approximation Theory and Feedforward Networks. Neural Networks, 4. Pp. 511-515. 1991.

Authors' Information

Luis Fernando de Mingo López *- Dept. Organización y Estructura de la Información, Escuela Univesitaria de Informática, Universidad Politécnica de Madrid, Crta. de Valencia km. 7, 28031 Madrid, Spain; e-mail: lfmingo@eui.upm.es*
Major Fields of Scientific Research: Artificial Intelligence, Social Intelligence

Nuria Gómez Blas *- Dept. Organización y Estructura de la Información, Escuela Univesitaria de Informática, Universidad Politécnica de Madrid, Crta. de Valencia km. 7, 28031 Madrid, Spain; e-mail: ngomez@eui.upm.es*
Major Fields of Scientific Research: Bio-inspired Algorithms, Natural Computing

Miguel A. Muriel *- Dept. Tecnología Fotónica, ETSI Telecomunicación, Universidad Politécnica de Madrid, Avenida Complutense 30, Ciudad Universitaria, 28040 Madrid, Spain; e-mail: m.muriel@upm.es*
Major Fields of Scientific Research: Theoretical Computer Science, Microwave Photonics

Daniel Triviño García *- Natural Computing Group, Facultad de Informática, Universidad Politécnica de Madrid, Campus de Montegancedo s.n., 28660, Boadilla del Monte, Madrid, Spain; e-mail: d.trivino@fi.upm.es*
Major Fields of Scientific Research: Artificial Intelligence, Computer Science

Appendix A - Linear Enhanced Neural Networks (with no hidden layers) Implementation in R

```
train <- function (iter, alpha, patrones_in, patrones_out, verbose) {
    entradas <- ncol(patrones_in)
    salidas <- length(patrones_out[1,])
    num_patrones <- nrow(patrones_in)
    num_pesos <- (entradas+1)*salidas
    matriz_pesos_auxiliar <- matrix(runif((entradas+1)*num_pesos),nrow=(entradas+1),ncol=num_pesos)
    matriz_pesos_principal <- matrix(runif((entradas+1)*salidas),nrow=(entradas+1),ncol=salidas)
    for ( i in 1:iter) {
        mse <- 0.0
        for (id_patron in 1:num_patrones) {
            patron_in <- c(as.matrix(patrones_in[id_patron,]), -1)
            patron_out <- c(as.matrix(patrones_out[id_patron,]))
            salida_red_auxiliar <- patron_in %*% matriz_pesos_auxiliar
            matriz_pesos_principal <- (matrix(salida_red_auxiliar,nrow=(entradas+1),ncol=salidas))
            salida_red_principal <- patron_in %*% matriz_pesos_principal
            error <- (salida_red_principal - patron_out)
            mse <- mse + sum(error*error*0.5)
            variacion_pesos <- -alpha * (error)
            matriz_pesos_principal <- matriz_pesos_principal + t(matrix(variacion_pesos,
                    nrow=(salidas),ncol=(entradas+1))) * (matrix(patron_in,entradas+1,salidas))
            vector_salida_red_auxiliar <- c(as.matrix(matriz_pesos_principal))
            error_auxiliar <- (salida_red_auxiliar - vector_salida_red_auxiliar)
            variacion_pesos_auxiliar <- -alpha * error_auxiliar
            matriz_pesos_auxiliar <- matriz_pesos_auxiliar + t(matrix(variacion_pesos_auxiliar,
                    nrow=(num_pesos),ncol=(entradas+1))) * (matrix(patron_in,entradas+1,num_pesos))
        }
        if (((i%%10)==0)&& verbose) {
            cat("Iteration␣", i, "␣-->␣MSE",(mse/num_patrones)/salidas,"\n")
        }
    }
    train <- matriz_pesos_auxiliar
}

test <- function (matriz_pesos_auxiliar,patrones_in, patrones_out) {
    entradas <- ncol(patrones_in)
    salidas <- length(patrones_out[1,])
    num_patrones <- nrow(patrones_in)
    num_pesos <- (entradas+1)*salidas
    salida_red <- patrones_out
    for (id_patron in 1:num_patrones) {
        patron_in <- c(as.matrix(patrones_in[id_patron,]), -1)
        salida_red_auxiliar <- patron_in %*% matriz_pesos_auxiliar
        matriz_pesos_principal <- (matrix(salida_red_auxiliar,nrow=(entradas+1),ncol=salidas))
        salida_red_principal <- patron_in %*% matriz_pesos_principal
        salida_red[id_patron,] <- (salida_red_principal)
    }
    test <- salida_red
}

panel.cor <- function(x, y, ...) {
    par(usr = c(0, 1, 0, 1))
    txt <- as.character(format(cor(x, y), digits=2))
    text(0.5, 0.5, txt, cex = 1.4 * ( abs(cor(x, y))) + 2, col="grey" )
}

plot_correlation <- function(patrones_in, patrones_out, network_output) {
    pairs(data.frame(patrones_in, patrones_out, network_output),upper.panel=panel.cor,
        main="", col="darkblue", cex=0.5)
}

plot2d_interval <- function(matriz_pesos_auxiliar, patrones_out) {
    patrones_in <- data.frame(runif(5000), runif(5000))
    sal <- test(matriz_pesos_auxiliar, patrones_in, patrones_out)
    color <- (sal[,1]-min(sal[,1]))/max(sal[,1]-min(sal[,1]))
    plot(patrones_in,col=rgb(color,color,0),xlab="",ylab="",pch=19,cex=1)

}
```