Polynomial Regression using a Perceptron with Axo-axonic Connections

Nuria Gómez Blas, Luis F. de Mingo, Alberto Arteta

Abstract: *Social behavior is mainly based on swarm colonies, in which each individual shares its knowledge about the environment with other individuals to get optimal solutions. Such co-operative model differs from competitive models in the way that individuals die and are born by combining information of alive ones. This paper presents the particle swarm optimization with differential evolution algorithm in order to train a neural network instead the classic back propagation algorithm. The performance of a neural network for particular problems is critically dependant on the choice of the processing elements, the net architecture and the learning algorithm. This work is focused in the development of methods for the evolutionary design of artificial neural networks. This paper focuses in optimizing the topology and structure of connectivity for these networks.*

Keywords: *Social Intelligence, Neural Networks, Grammatical Swarm, Particle Swarm Optimization, Learning Algorithm.*

ACM Classification Keywords: *F.1.1 Theory of Computation - Models of Computation, I.2.6 Artificial Intelligence - Learning.*

Introduction

Neural networks are non-linear systems whose structure is based on principles observed in biological neuronal systems. A neural network could be seen as a system that can be able to answer a query or give an output as answer to a specific input. The in/out combination, i.e. the transfer function of the network is not programmed, but obtained through a training process on empiric datasets.

In practice the network learns the function that links input together with output by processing correct input/output couples. Actually, for each given input, within the learning process, the network gives a certain output which is not exactly the desired output, so the training algorithm modifies some parameters of the network in the desired direction. Hence, every time an example is input, the algorithm adjusts its network parameters to the optimal values for the given solution: in this way the algorithm tries to reach the best solution for all the examples. These parameters we are speaking about are essentially the weights or linking factors between each neuron that forms our network.

Neural Networks' application fields are tipically those where classic algorithms fail because of their unflexibility (they need precise input datasets). Usually problems with unprecise input datasets are those whose number of possible input datasets is so big that they can't be classified. For example in image recognition are used probabilistic algoithms whose efficency is lower than neural networks' and whose caratheristics are low flexibility and high development complexity. Another field where classic algorithms are in troubles is the analisys of those phenomena whose matematical rules are unknown.

There are indeed rather complex algorithms which can analyse these phenomena but, from comparisons on the results, it comes out that neural networks result far more efficient [Hu and Hwang 2001; Katagiri 2000]: these algorithms use *Fourier*'s trasform to decompose phenomena in frequential components and for this reason they result highly complex and they can only extract a limited number of harmonics generating a big number of approximations. A neural network trained with complex phenomena's data is able to estimate also frequential components, this means that it realizes in its inside a *Fourier*'s trasform even if it was not trained for that! One of the most important neural networks' applications is undoubtfully the estimation of complex phenomena such as meteorological, financial, socio-economical or urban events. Thanks to a neural network it's possible to predict, analyzing hystorical series of datasets just as with these systems but there is no need to restrict the problem or use *Fourier*'s tranform. A defect common to all those methods it's to restrict the problem setting certain hypothesis that can turn out to be wrong. We just have to train the neural network with hystorical series of data given by the phenomenon we are studying.

Calibrating a neural network means to determinate the parameters of the connections (synapsis) through the training process. Once calibrated there is need to test the netowrk efficency with known datasets, which has not been used in the learning process. There is a great number of Neural Networks which are substantially distingushed by: type of use, learning model (supervised/non-supervised), learning algorithm, architecture, etc.

This paper focuses on supervised networks with the *backpropagation* learning algorithm and applied to signal analysis with a typical feedforward architecture.

Multilayer Perceptron

Multilayer perceptrons (*MLPs*) are layered feedforward networks [Cover and Tomas 1991] typically trained with static backpropagation. These networks have found their way into countless applications requiring static pattern classification. Their main advantage is that they are easy to use, and that they can approximate any input-output map.

In principle, backpropagation provides a way to train networks with any number of hidden units arranged in any number of layers [Hu 1996; Hu et al. 1994]. In fact, the network does not have to be organized in layers - any pattern of connectivity that permits a partial ordering of the nodes from input to output is allowed. In other words, there must be a way to order the units such that all connections go from "earlier" (closer to the input) to "later" ones (closer to the output). This is equivalent to stating that their connection pattern must not contain any cycles. Networks that respect this constraint are called feedforward networks; their connection pattern forms a directed acyclic graph or dag.

Backpropagation algorithm can be expressed as equation (1). Note that in order to calculate the error for unit j, we must first know the error of all its posterior nodes (forming the set P_j). Again, as long as there are no cycles in the network, there is an ordering of nodes from the output back to the input that respects this condition. For example, we can simply use the reverse of the order in which activity was propagated forward.

$$\delta_j = f_j^{'}(net_j) \sum_{i \in P_j} \delta_j w_{ij} \tag{1}$$

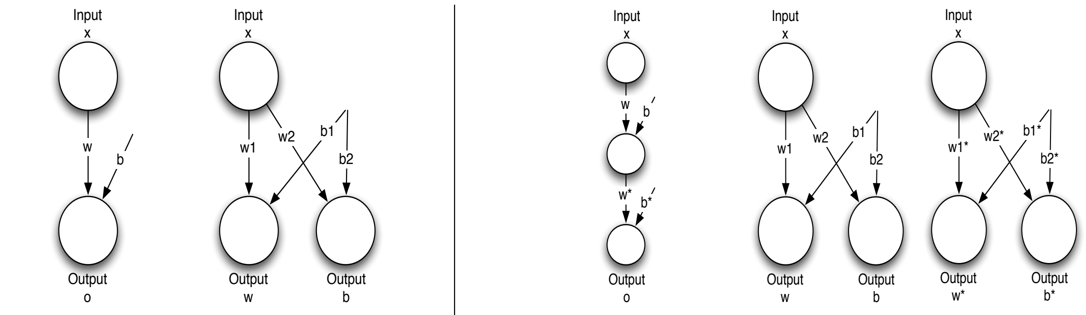
Axo-axonic Neural Networks

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. 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*.

ENN as Taylor series approximators.

It is well known 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 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 $f(x) = ax + b$.



$$\begin{aligned} o &= wx + b = \\ &= (w_1x + b_1)x + w_2x + b_2 = \\ &= Ax^2 + Bx + C \end{aligned}$$

$$\begin{aligned} o &= w^*(wx + b) + b^* = \\ &= (w_1^*x + b_1^*)[(w_1x + b_1)x + w_2x + b_2] + w_2^*x + b_2^* = \\ &= Ax^3 + Bx^2 + Cx + D \end{aligned}$$

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.

Data sets

Previously defined neural network architecture has been used to approximate different data sets generated using a 2-degree polynomial expression of n variables, $\mathcal{P}_n$

Table 2: Sample of randomly generated values using an uniform distribution.

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.97600 -0.50130  0.01855 -0.04053  0.41540  0.98880

SAMPLE DATA --   Standard deviation:  0.5708282 , Variance:  0.3258449

  [1] -0.05100 -0.26798 -0.77103 -0.05086 -0.26436 -0.32416 -0.96345 -0.82661  0.89400  0.86154  0.29179  0.86964  0.72917
 [14] -0.38158 -0.26924  0.07974 -0.42825  0.98878 -0.87016 -0.41886  0.08352  0.25384 -0.39868 -0.97603 -0.79159  0.59038
 [27] -0.96696 -0.69930 -0.95632 -0.42189  0.59004  0.65942  0.40932 -0.77566  0.16400  0.21629  0.45231 -0.39479  0.92717
 [40]  0.36792 -0.49954  0.90206  0.51239 -0.72945 -0.94883 -0.97396  0.54177  0.06722  0.00008  0.59842 -0.69638  0.07971
 [53] -0.50657  0.15915  0.35297 -0.73656  0.05649  0.48023  0.97079  0.17456  0.07224  0.92373 -0.64668  0.63300 -0.30577
 [66] -0.16878 -0.35884 -0.15355 -0.20623 -0.97129 -0.80947  0.44842 -0.61685  0.03132 -0.41215 -0.82795  0.00578  0.98312
 [79] -0.20718 -0.70245  0.27336 -0.92647  0.69454 -0.55756  0.21941 -0.03392  0.43355  0.31659 -0.21409  0.13297  0.65948
 [92] -0.74168 -0.61549  0.10033  0.12152  0.06672  0.68932 -0.04284  0.04052  0.11754  0.52441  0.70085 -0.08563  0.23654
```

$$\mathcal{P}_n = (\bar{C} \times \bar{X})^2 + c_0, \tag{2}$$

where $\bar{C} = \{c_0, c_1, c_2, \cdots, c_n\}$ are the coefficients and $\bar{X} = \{x_1, x_2, \cdots, x_n\}$ are the variables.

A neural network with no hidden layer is able to approximate such data sets with no error at all (or at least with a lower bound), according to theoretical results. Next samples presents different empirical results to justify such theoretical proposition.

All testing data, coefficients and variable values, are generated using an uniform distribution in interval $(-1, 1)$. Table 2 show some generated values and the mean, median, max, min and quartiles of such data just to check the uniform properties of generated values. Figure 2 shows a correlation matrix among 10 variables. The table and figure only show part of the data not all.

Polynomial regression: 2 variables

$$\begin{aligned} f(x, y) &= (Ax + By + C)^2 \\ &= A^2x^2 + B^2y^2 + C^2 + 2ACx + 2BCy + 2ABxy \end{aligned} \tag{3}$$

$$\begin{pmatrix} A^2 & i_1 & j_1 \\ i_2 & B^2 & k_1 \\ j_2 & k_2 & C^2 \end{pmatrix}, \text{where } i_1 + i_2 = 2AB, j_1 + j_2 = -2AC, k_1 + k_2 = -2BC \tag{4}$$

A neural network has been trained using a random data set with an uniform distribution and describing the polynomial function:

$$f(x, y) = ((x, y, 1) \times (-0.7909866, -0.7742045, 0.726699))^2 \tag{5}$$

Mean squared error of the network must be equal to 0, according to the theoretical results. Next listing shows obtained results with the proposed neural network architecture. Note that MSE in the training and cross validation data sets is really low (near 0).

```
Number of variables:  2
Coefficients (A, B, C): -0.7909866 -0.7742045 0.726699
Squared coefficients (A*A, B*B, C*C): 0.6256597 0.5993926 0.5280914
Number of patterns:  2000 , Iterations:  102 , Learning rate:  0.05 , Cross val.:  20 %
Mean Squared Error (TRAINING):
 Standard deviation  1.037659e-14 , Variance  1.076736e-28
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-2.442e-14 -1.499e-14 -8.105e-15 -6.531e-15  9.159e-16  2.565e-14
Mean Squared Error (CROSS VALIDATION):
```

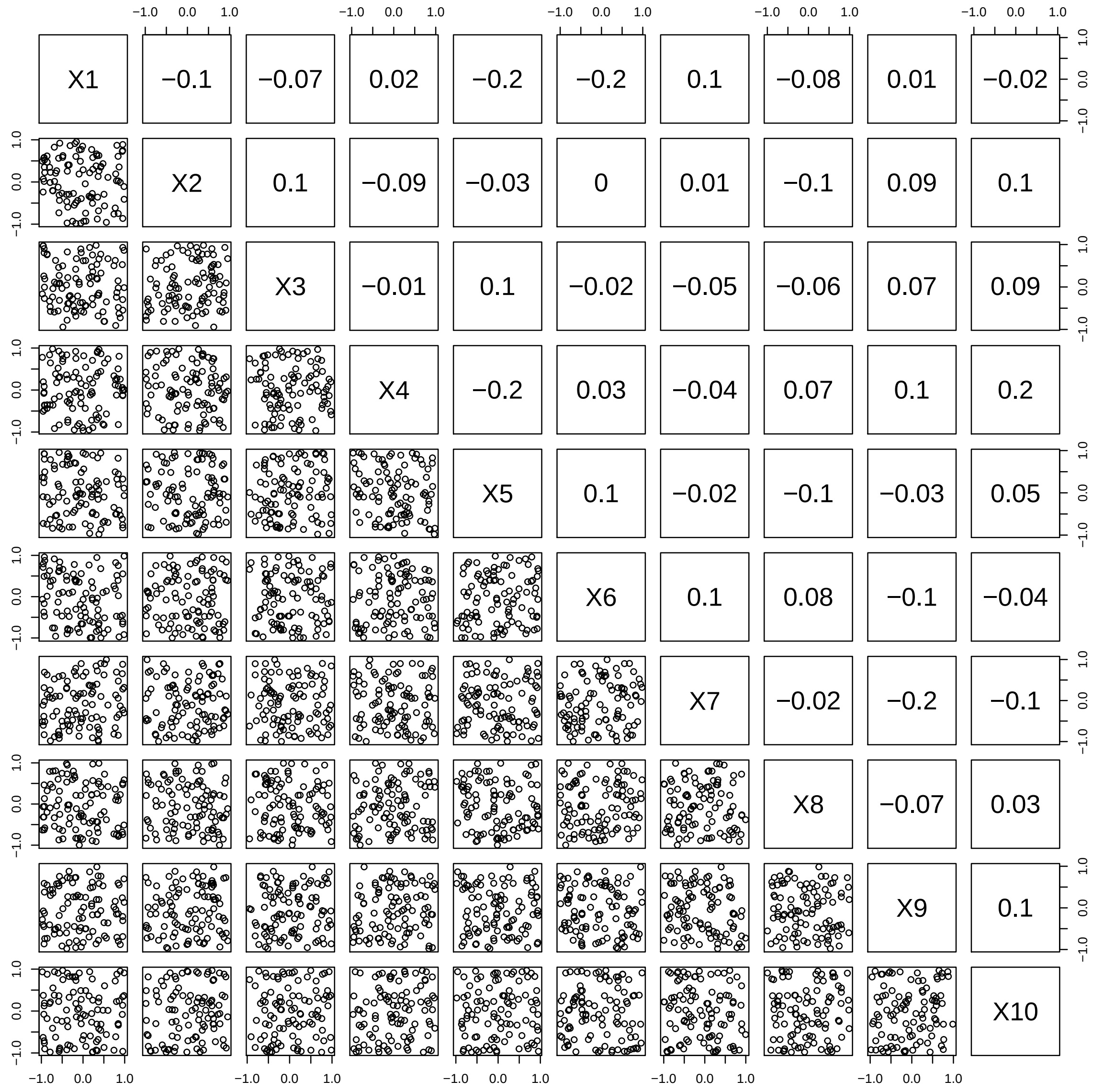


Figure 2: Correlation matrix among 10 variables of the generated data set.

```
 Standard deviation  1.06709e-14 , Variance  1.138682e-28
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-2.465e-14 -1.367e-14 -6.852e-15 -5.291e-15  2.276e-15  2.476e-14
MATRIX Network coefficients:
          [,1]      [,2]      [,3]
[1,] 0.6256597 0.4474168 0.4502775
[2,] 0.7773538 0.5993926 0.5274104
[3,] 0.6993407 0.5978168 0.5280914
A*A = 0.6256597 , B*B = 0.5993926 , C*C = 0.5280914
2*A*B = 1.224771 , -2*A*C = 1.149618 , -2*B*C = 1.125227
```

Coefficients of regression polynomial can be obtained using the weights matrix of trained neural network. Previous matrix shows final weights with a cuasi-null MSE, in our case the coefficients are the following ones (according to equation 4):

$$(A^2, B^2, C^2) = (0.6256597, 0.5993926, 0.5280914) \tag{6}$$

$$(2AB, -2AC, -2BC) = (1.224771, 1.149618, 1.125227) \tag{7}$$

Such results are tottally coherent with equation 5, that is, proposed neural network is able to approximate the data set and generate the polynomial function that describes the data set.

High dimension properties

Previous neural network model behaves with the same *MSE* in case using high dimension data sets. Following listing shows the learning result of a polynomial with 50 variables. Table 3 shows a summary using different number of variables.

```
Number of variables:  50
Coefficients (A, B, ...): -0.4198864 0.6027695 0.6039954 -0.1065414 0.9356629 0.9834093 -0.8226473
0.8743879 -0.7042015 -0.3393832 0.7709589 -0.5013933 -0.8844903 -0.699804 -0.6985472 0.5526986
0.5012762 -0.07739273 0.1085922 -0.914291 0.4816206 -0.7921206 -0.3315833 0.1244581 -0.5412277
-0.5013642 -0.9528081 0.2618788 0.2249716 -0.9169567 0.1750446 0.7072091 -0.6856862 0.3719996
-0.1491458 -0.5018547 0.9422977 0.691155 0.9959811 0.2117917 -0.1234416 0.1298864 -0.8055485
0.2700599 0.9574739 0.8468997 0.3655415 0.6998148 -0.2625522 -0.6242939 -0.9633645
Squared coefficients (A*A, B*B, ...): 0.1763046 0.3633311 0.3648104 0.01135107 0.8754651 0.9670938
0.6767485 0.7645542 0.4958998 0.115181 0.5943776 0.2513952 0.7823231 0.4897256 0.4879682
0.3054758 0.2512778 0.005989635 0.01179226 0.835928 0.2319584 0.627455 0.1099475 0.01548981
0.2929274 0.2513661 0.9078433 0.0685805 0.05061223 0.8408095 0.03064062 0.5001447 0.4701656
0.1383837 0.02224446 0.2518581 0.8879249 0.4776952 0.9919784 0.04485574 0.01523784 0.01687047
0.6489085 0.07293237 0.9167562 0.7172391 0.1336206 0.4897407 0.06893364 0.3897428 0.9280712
Number of patterns:  1850 , Iterations:  150 , Learning rate:  0.05 , Cross val.:  20 %
Mean Squared Error (TRAINING):
 Standard deviation  0.08624005 , Variance  0.007437346
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.31370 -0.03923  0.01557  0.01329  0.06721  0.28350
Mean Squared Error (CROSS VALIDATION):
 Standard deviation  0.8489767 , Variance  0.7207615
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2.22700 -0.52780  0.02572  0.02860  0.57930  2.79500
MATRIX Network coefficients:
0.139645 0.2698724 0.4126177 0.03386078 0.8716183 0.8480131 0.795495 0.8117691 0.4728284 0.202625
0.4350294 0.344523 0.9615263 0.4889943 0.5821863 0.4702517 0.3315289 -0.01390491 -0.01994969
0.9191157 0.1723037 0.6412748 0.06657735 0.1184948 0.3114974 0.2422406 0.9148941 0.1689978
-0.03863459 0.9131449 -0.06600995 0.4795042 0.4226703 0.1084583 0.09037618 0.2399482 0.8859146
0.3590639 0.8691869 -0.01292396 0.04054493 0.1078918 0.6508386 0.144514 0.7362382 0.7127247
0.1572646 0.4372879 0.122485 0.4098477 0.8552449
```

Table 3: Results with different number of variables (see figure 3)

```
Number of variables:  3
Number of patterns:  1200 , Iterations:  20 , Learning rate:  0.05 , Cross validation set:  20 %
Real coefficients:  0.885 0.814 0.114 -0.513
Squared Real coefficients:  0.783 0.663 0.013 0.263
Mean Squared Error (TRAINING): Standard deviation 0.005430024 , Variance 2.948516e-05
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -0.0172000 -0.0030180 0.0009090 0.0006146 0.0048900 0.0106200
Mean Squared Error (CROSS VALIDATION): Stan ard deviation 0.005166362 , Variance 2.669129e-05
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -0.0136000 -0.0026350 0.0009429 0.0006865 0.0046690 0.0107400
Network coefficients:  0.77 0.65 0.008 0.274
```

```
Number of variables:  5
Number of patterns:  2000 , Iterations:  20 , Learning rate:  0.05 , Cross validation set:  20 %
Real coefficients:  -0.633 0.07 -0.428 -0.441 -0.858 -0.946
Squared Real coefficients:  0.401 0.005 0.183 0.194 0.736 0.895
Mean Squared Error (TRAINING): Standard deviation 0.001910566 , Variance 3.650264e-06
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -0.0047460 -0.0015840 -0.0002670 -0.0002031 0.0010550 0.0059510
Mean Squared Error (CROSS VALIDATION): Stan ard deviation 0.001839322 , Variance 3.383107e-06
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -0.0046280 -0.0013950 -0.0003133 -0.0002007 0.0008592 0.0055970
Network coefficients:  0.404 0.007 0.187 0.197 0.739 0.89
```

```
Number of variables:  7
Number of patterns:  2800 , Iterations:  20 , Learning rate:  0.05 , Cross validation set:  20 %
Real coefficients:  -0.085 0.723 -0.362 -0.662 0.049 -0.006 -0.272 -0.725
Squared Real coefficients:  0.007 0.523 0.131 0.438 0.002 0 0.074 0.525
Mean Squared Error (TRAINING): Standard deviation 0.0006043065 , Variance 3.651864e-07
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -0.0017650 -0.0005786 -0.0001700 -0.0001428 0.0002686 0.0020250
Mean Squared Error (CROSS VALIDATION): Stan ard deviation 0.0006502922 , Variance 4.228799e-07
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -1.614e-03 -5.753e-04 -9.671e-05 -8.572e-05 3.291e-04 2.020e-03
Network coefficients:  0.008 0.524 0.131 0.439 0.003 0.001 0.075 0.523
```

```
Number of variables:  9
Number of patterns:  3600 , Iterations:  20 , Learning rate:  0.05 , Cross validation set:  20 %
Real coefficients:  0.601 -0.022 0.753 0.023 0.764 0.589 -0.06 -0.772 0.03 0.777
Squared Real coefficients:  0.361 0 0.567 0.001 0.584 0.347 0.004 0.596 0.001 0.604
Mean Squared Error (TRAINING): Standard deviation 0.0001982064 , Variance 3.928577e-08
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -5.585e-04 -1.581e-04 -2.589e-05 -1.777e-05 1.126e-04 7.886e-04
Mean Squared Error (CROSS VALIDATION): Stan ard deviation 0.0002006681 , Variance 4.026771e-08
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -5.901e-04 -1.647e-04 -3.507e-05 -2.336e-05 9.689e-05 7.056e-04
Network coefficients:  0.361 0.001 0.567 0.001 0.584 0.347 0.004 0.596 0.001 0.603
```

```
Number of variables:  11
Number of patterns:  4400 , Iterations:  20 , Learning rate:  0.05 , Cross validation set:  20 %
Real coefficients:  -0.915 -0.737 -0.706 -0.744 0.656 -0.332 0.981 -0.112 0.744 0.59 0.026 0.797
Squared Real coefficients:  0.837 0.543 0.498 0.553 0.43 0.11 0.962 0.013 0.554 0.348 0.001 0.636
Mean Squared Error (TRAINING): Standard deviation 4.617442e-05 , Variance 2.132077e-09
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -1.404e-04 -4.247e-05 -1.064e-05 -9.864e-06 2.066e-05 1.620e-04
Mean Squared Error (CROSS VALIDATION): Stan ard deviation 4.979714e-05 , Variance 2.479755e-09
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -1.413e-04 -4.654e-05 -1.265e-05 -1.095e-05 2.230e-05 1.527e-04
Network coefficients:  0.837 0.543 0.498 0.554 0.43 0.11 0.962 0.013 0.554 0.348 0.001 0.635
```

```
Number of variables:  13
Number of patterns:  5200 , Iterations:  20 , Learning rate:  0.05 , Cross validation set:  20 %
Real coefficients:  -0.101 0.877 -0.38 -0.081 0.022 0.13 0.035 -0.428 0.076 0.364 -0.156 -0.52 0.225 -0.893
Squared Real coefficients:  0.01 0.769 0.145 0.007 0 0.017 0.001 0.183 0.006 0.133 0.024 0.271 0.05 0.798
Mean Squared Error (TRAINING): Standard deviation 0.0001415502 , Variance 2.003647e-08
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -4.066e-04 -1.160e-04 -1.987e-05 -1.510e-05 7.840e-05 5.004e-04
Mean Squared Error (CROSS VALIDATION): Stan ard deviation 0.0001463287 , Variance 2.141208e-08
Min.  1st Qu.  Median Mean 3rd Qu.  Max.  -4.326e-04 -1.208e-04 -2.816e-05 -2.235e-05 7.465e-05 4.004e-04
Network coefficients:  0.01 0.769 0.145 0.007 0.001 0.017 0.001 0.183 0.006 0.133 0.025 0.271 0.051 0.798
```

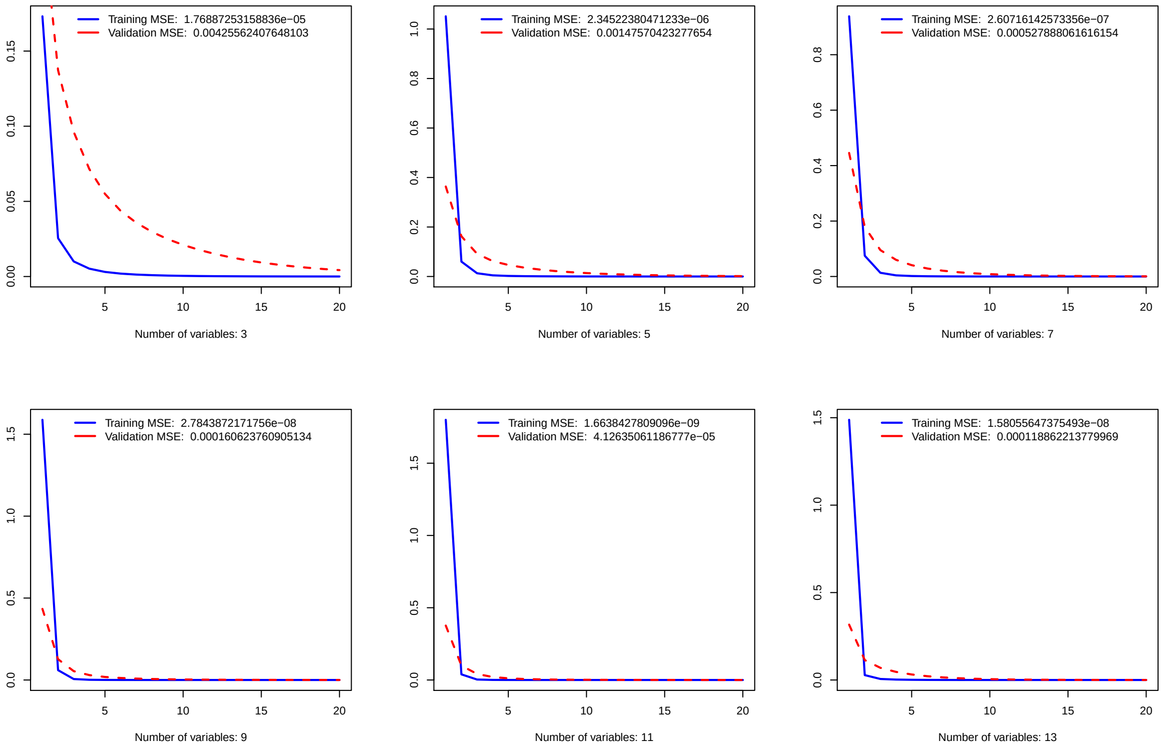


Figure 3: Training and cross-validation MSE with different number of variables (see table 3)

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 4 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.

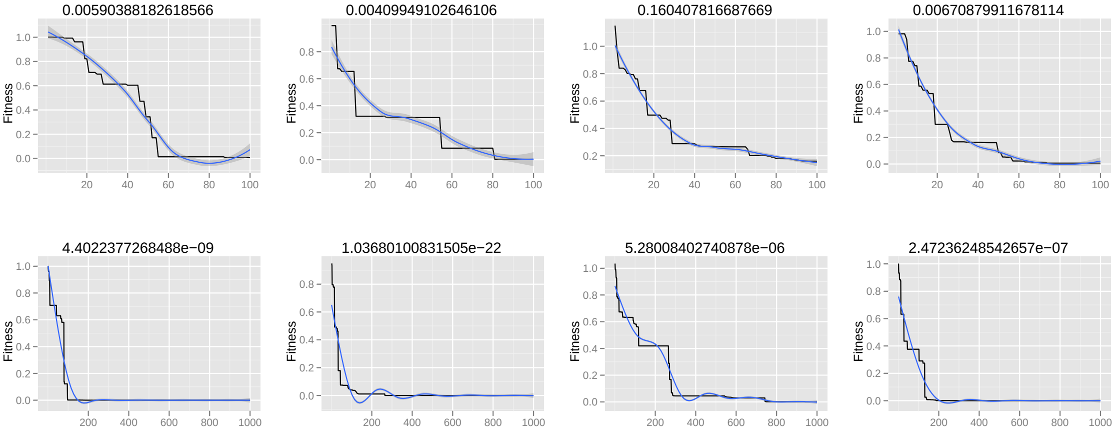


Figure 4: XOR multilayer perceptron with 2 hidden neurons and a particle swarm optimization learning using 10 individuals (individuals have 9 dimensions). Each column represents a different random initialization and each row a number of iterations (100 and 1000).

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 → 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 → 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.

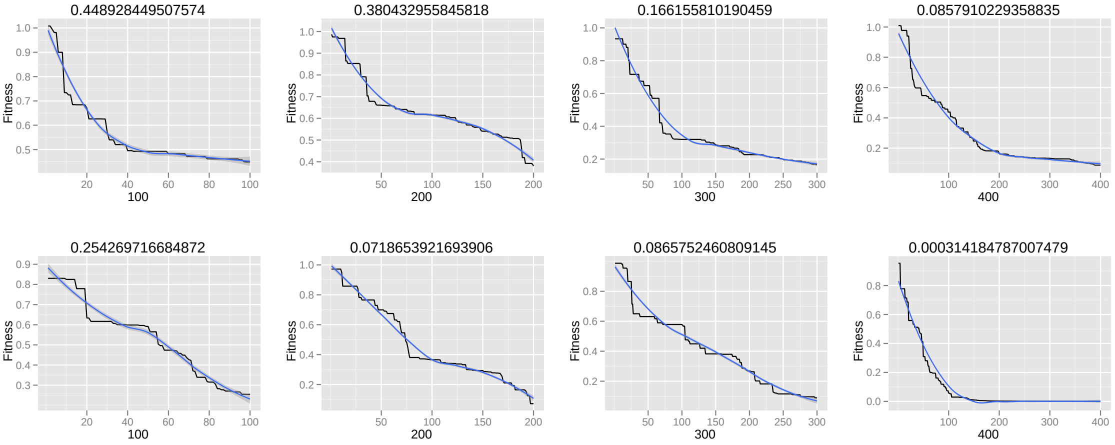


Figure 5: Binary coding neural network form 8 inputs to 3 outputs. First row is a neural network with 3 hidden neurons and second row a neural network with 5 hidden neurons. Mean squared error (fitness of the PSO) decreases as the number of iterations (100 to 400) increases.

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 and Eberhart 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 et al. 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.

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:

$$eval(x) = \begin{cases} \text{f(x)} & x \in \mathcal{F} \\ \text{f(x) + } \textit{penalty}\text{(x)} & eoc \end{cases} \tag{8}$$

$\mathcal{F}$ 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 $\mathcal{F}$.

Acknowledgements

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

Appendix A: Implementation in R

```
train <- function (iter, alpha, patrones_in, patrones_out, test, verbose) {
  patrones_in <- as.matrix(patrones_in)
  patrones_out <- as.matrix(patrones_out)
  entradas <- ncol(patrones_in)
  salidas <- length(patrones_out[1,])
  num_patrones <- round(nrow(patrones_in)*(1-test),digits=0)
  num_patrones_test <- 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)
  mse <- (1:iter)
  mse_test <- (1:iter)
  for ( i in 1:iter) {
    mse[i] <- 0.0
    for (id_patron in 1:num_patrones) {
        patron_in <- c((patrones_in[id_patron,]),-1)
        patron_out <- c((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[i] <- mse[i] + 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((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))
    }
    salida_test <- test(matriz_pesos_auxiliar, patrones_in[(num_patrones+1):num_patrones_test, ],
      (patrones_out[(num_patrones+1):num_patrones_test, ]))
    mse_test[i] <- (sum(abs(salida_test - (patrones_out[(num_patrones+1):num_patrones_test, ])))/
      (num_patrones_test-num_patrones))/salidas
    if ((verbose>0) && (i%%verbose)==0) {
        cat("Iteration␣", i, "␣-->␣MSE",(mse[i]/num_patrones)/salidas, "\t\t␣-->␣CV␣", mse_test[i], "\n")
    }
    mse[i] <- (mse[i]/num_patrones)/salidas
  }
  train <- list(mpa=matriz_pesos_auxiliar,mse=mse,mse_test=mse_test)
}

test <- function  (matriz_pesos_auxiliar,patrones_in, patrones_out) {
    patrones_in <- as.matrix(patrones_in)
    patrones_out <- as.matrix(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((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 = 1.5 * ( abs(cor(x, y))) + 2 )
}

plot_correlation <- function(patrones_in, patrones_out, network_output) {
    pairs(data.frame(patrones_in, patrones_out, network_output),upper.panel=panel.cor,
      main="Relationships␣between␣characteristics␣of␣data", col="gray", cex=0.5)
}

plot_mse <- function(mpa, title="", xlabel="", ylabel="") {
  iteraciones <- length(mpa$mse)
  plot(c(1:iteraciones),mpa$mse,lty=1,type="l",main=c(title),xlab=xlabel,ylab=ylabel, col="blue", lwd=2)
  lines(c(1:iteraciones),mpa$mse_test,lty=2, col="red", lwd=2)
  legend("top", bty="n", cex=1, legend=c(paste("Training␣MSE:␣", mpa$mse[iteraciones]),
    paste("Validation␣MSE:␣", mpa$mse_test[iteraciones])), col=c("blue","red"), lty=1,lwd=2)
}
```

Bibliography

Cagnina, L. C., Esquivel, S. C., and Coello, C. (2008). Solving engineering optimization problems with the simpel constrained particle swarm optimizer. *Informatica*, 32:319–326.

Cover, T. M. and Tomas, J. A. (1991). *Elements of information theory*. John Willey Sons.

Delacour, J. (1987). *Apprentissage et memoire: Une approache neurobiologique*. Masson.

Hu, X. and Eberhart, R. (2002). Solving constrained nonlinear optimization problems with particle swarm optimization. In *6th world Multiconference on Systemics, Cybernetics and Informatics.*, volume 5.

Hu, Y. H. (1996). Pattern classification with multiple classifiers. Technical report, Department of Electrical Computing Engineering. University of Wisconsin, Madison.

Hu, Y. H. and Hwang, J.-N. (2001). *Handbook of neural network signal processing VE profiling*. CRC Press.

Hu, Y. H., Tompkins, W. J., Urrusti, J. L., and Afonso, V. X. (1994). Applications of artificial neural networks for ecg. *Journal of electrocardiology*, 26:66–73.

Katagiri, S. (2000). *Handbook of neural networks for speech processing*. Artech House.

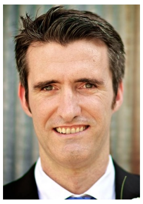
Authors' Information

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

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



Alberto Arteta *- 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