

Neural learning using orthogonal arrays

Adrian Stoics, Julian Blosiu

Center for Space Microelectronics l'ethnology, Jet Propulsion Laboratory
California Institute of Technology, Pasadena, CA 91109, USA

and Brian Figic

Ballistic Missile Defense Organization, Washington, DC 20301

Abstract

The paper proposes the use of the orthogonal arrays for neural network learning. Learning can be seen as a search for the neural weights that give an optimal network performance. The search/optimization adopted here is inspired from **Taguchi** methods based on Orthogonal Arrays (a special set of Latin Squares), which proved to be a powerful tool in robust design. In its most straightforward implementation the search requires only a few steps, leading to fast neural learning. However the solution obtained in this way is only in the vicinity of an optimum. Getting closer to the optimum can be done by making the search adaptive, at the price of an increased number of iterations.

1. INTRODUCTION

There is a large area of real-world applications, including pattern recognition and intelligent sensor fusion problems, for which neural networks (NN) have proved very efficient, largely due to their learning capability. For a given NN architecture learning refers to modifications of the values of neuron weights, which modulate signal transmission between interconnected neurons. Most learning algorithms determine the weight modifications such as to optimize some cost function. The most commonly used cost function is the sum of squared errors between current NN output and the desired (target) output. Thus, learning in NN can be seen as a search process to find the weight values that generate the smallest cost.

Learning in NN can be a lengthy process. Traditional learning algorithms are based on numerical methods that require a significant amount of iterations. Most popular learning methods are based on Gradient Descent (GD) search/optimization. The classic example for such methods is the backpropagation algorithm. Pure GD is a local search technique and will end up in the closest local minimum. Global searches were also proposed for neural learning, in particular using Genetic Algorithms (GA). Both GD and GA based neural learning are computationally expensive, and hence slow. Thousands of iterations are common for GD based learning, and hundreds of generations with hundreds of individuals in the population are common for GA. This is impractical for real-time application systems, in which systems are expected to quickly learn and adapt to their environments, and for which fast learning methods are in high demand.

In this paper we propose the use of a global search mechanism, based on a modified **Taguchi** technique for robust design. In the proposed approach weights modifications result from an iterative application of robust design optimization using orthogonal arrays (OA) [Tag87]. This investigation was triggered by a study showing that in some optimization problems OA outperform GA searches [Gold96]. Searches based on OA share some of the nice characteristics of GA based searches: no derivatives have to be computed, the algorithm is not too sensitive to starting conditions. Also, large numbers of variables can be handled easily. For details on Taguchi methods and OA the reader is referred to [Tag87].

In a NN the weights are the equivalent of "parameters" in the robust design methodology. Several weight values enter the network evaluation at one iteration step, and are associated with values of levels of the parameters used in robust design. The neural network for which learning is to be performed gets associated with an Orthogonal Array (OA), with the number of parameters equals the number of weights of the NN. The interval containing the solution is gradually shrinking as the result of an iterative algorithm.

11. LEARNING ALGORITHM BASED ON ORTHOGONAL ARRAYS

To facilitate the understanding, the algorithm is explained in relation to a simple example, using the NN illustrated in Figure 1. The network has 7 nodes and 13 weights. Each neuron calculates its output by a weighted sum of its inputs, modulated by a sigmoidal nonlinearity. For example, output Y_1 of a neuron in the hidden layer, is calculated as $Y_1 = f(X_1 * W_1 + X_2 * W_4 + X_3 * W_7)$, where $f(x) = 1/(1 + e^{-x})$. The outputs of the hidden layer, Y_1, Y_2, Y_3 act as inputs for the output node, which produces an output signal calculated as $O = f(Y_1 * W_{10} + Y_2 * W_{11} + Y_3 * W_{12} + 1 * W_{13})$, where $f(x) = 1/(1 + e^{-x})$. The network will learn an input-output dependency from input-output (1/0) examples shown Table 1. The output shown is the target value.

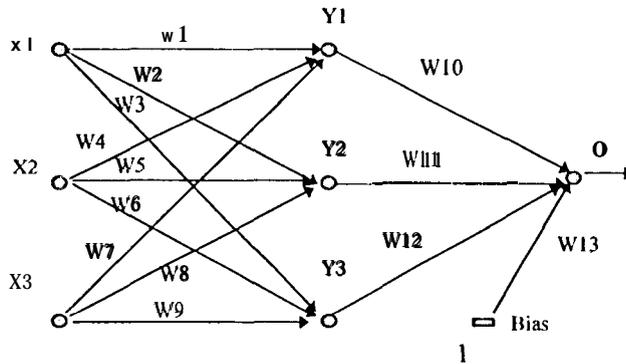


Figure 1. Neural network with 13 weights (parameters)

Table 1. Input-output examples

Input			Output
x_1	x_2	x_3	
0	0	0	0.28
0	0	1	0.33
0	1	0	0.32
0	1	1	0.37
1	0	0	0.26
1	0	1	0.31
1	1	0	0.30
1	1	1	0.35

The algorithm can be expressed as follows:

Initialization: choose an OA with number of parameters = number of weights.

Do until the stopping condition is satisfied:

Step 1. Generate a test set

Step 2. Evaluate a cost function for the test set

Step 3. Evaluate the effect of each level on the cost function

Step 4. Modify each weight in the direction of the level that produced least cost

For example, the stopping condition is when the weight modification is less than a given value.

initialization

The fact that the network has 13 weights (W_1 to W_{13}) determines the choice of an OA with 13 parameters. In this case, an L_{27} orthogonal array was chosen [Tag87], in which each parameter (weight) has 3 levels, thus 3 weight values will be considered at each step of the search. L_{27} is illustrated in Table 2.

Table 2. Orthogonal Array L27 with 13 Parameters (Weights) at Three Levels Each

Test	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	2	2	2	2	2	2	2	2	2
3	1	1	1	1	3	3	3	3	3	3	3	3	3
4	1	2	2	2	1	1	1	2	2	2	3	3	3
5	1	2	2	2	2	2	2	3	3	3	1	1	1
6	1	2	2	2	3	3	3	1	1	1	2	2	2
7	1	3	3	3	1	1	1	3	3	3	2	2	2
8	1	3	3	3	2	2	2	1	1	1	3	3	3
9	1	3	3	3	3	3	3	2	2	2	1	1	1
10	2	1	2	3	1	2	3	1	2	3	1	2	3
11	2	1	2	3	2	3	1	2	3	1	2	3	1
12	2	1	2	3	3	1	2	3	1	2	3	1	2
13	2	2	3	1	1	2	3	2	3	1	3	1	2
14	2	2	3	1	2	3	1	3	1	2	1	2	3
15	2	2	3	1	3	1	2	1	2	3	2	3	1
16	2	3	1	2	1	2	3	3	1	2	2	3	1
17	2	3	1	2	2	3	1	1	2	3	3	1	2
18	2	3	1	2	3	1	2	2	3	1	1	2	3
19	2	1	3	2	1	3	2	1	3	2	1	3	2
20	3	1	3	2	2	1	3	2	1	3	2	1	3
21	3	1	3	2	3	2	1	3	2	1	3	2	1
22	3	2	1	3	1	3	2	2	1	3	3	2	1
23	3	2	1	3	2	1	3	3	2	1	1	3	2
24	3	2	1	3	3	2	1	1	3	2	2	1	3
25	3	3	2	1	1	3	2	3	2	1	2	1	3
26	3	3	2	1	2	1	3	1	3	2	3	2	1
27	3	3	2	1	3	2	1	2	1	3	1	3	2

Iterative steps

Step 1. Generate a set of tests (a “design of experiments“ in terms of robust design terminology). The set of tests consist of a number of NN with different weights. Each NN simulated parametric run is equivalent with an experimental test run. The number of NN equals the number of rows of the OA (which indicates the number of experiments). The weights are obtained from the Orthogonal Array by replacing the (indexed) level of each parameter (weight) with its current value in the search interval.

Suppose that the search starts at the [-1 ,1] interval for all weights, The 3 levels (level 1,2, 3) in the OA could be associated with the extremes and the middle of the search interval, {-1, 0, 1}. Thus, we are replacing the level number in Table 2 with the actual value of the weight at that level, which leads to Table 3. In this example, for weight W1, at the first iteration, level 1 from Table 2 (1” level of weight W1) is replaced by -1 as shown in Table 3, level 2 is replaced by 0, and level 3 is replaced by 1.

Step 2. Evaluate a cost function for each network in the test set. For example, this could be the sum of squared errors between the network output and target output value. The cost to be minimized is the sum of cost functions for all networks.

For the first iteration the actual cost value for each NN simulation run (calculated as sum of squared errors for the given I/O set) is shown in the last column in Table 3.

Step3. Evaluate the effect that each individual level had on the cost function. This is done by summing the cost functions for the networks (experiments) in which that level was present.

Consider for example W 1. Level 1 (of W 1) enters the first 9 networks (experiments) of L27, see the 1st column of Table 2. The cumulated effects of Level 1 of W 1 (E_{L1}^{W1}) are given by the sum of costs for the networks in which L1 intervenes (costs are written in the last column in Table 3). Thus $E_{L1}^{W1} = cost1 + cost2 + \dots + cost9$.

For this example, after the first iteration, the effect of each level value on the cost is synthesized in Fig. 2.

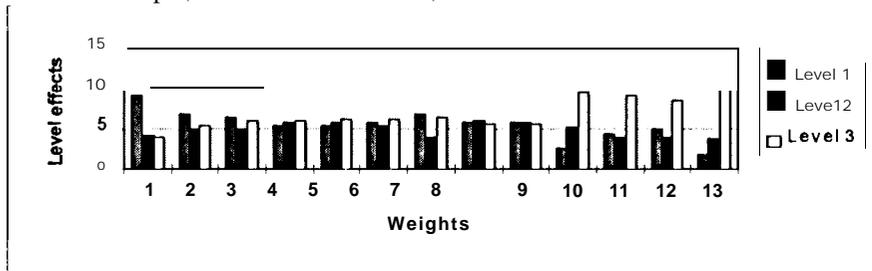


Figure 2. Cost effects of levels 1,2,3 of the 13 weights after the 1st iteration

Table 3. The set of test for the first iteration: 27 NN with their particular weights

Test	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	cost
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0.19
2	-1	-1	-1	-1	0	0	0	0	0	0	0	0	0	0.26
3	-1	-1	-1	-1	1	1	1	1	1	1	1	1	1	2.97
4	-1	0	0	0	-1	-1	-1	0	0	0	1	1	1	2.18
5	-1	0	0	0	0	0	0	1	1	1	-1	-1	-1	0.28
6	-1	0	0	0	1	1	1	-1	-1	-1	0	0	0	0.07
7	-1	1	1	1	-1	-1	-1	1	1	1	0	0	0	0.84
8	-1	1	1	1	0	0	0	-1	-1	-1	1	1	1	1.75
9	-1	1	1	1	1	1	1	0	0	0	-1	-1	-1	0.45
10	0	-1	0	1	-1	0	1	-1	0	1	-1	0	1	1.2
11	0	-1	0	1	0	1	-1	0	1	-1	0	1	-1	0.01
12	0	-1	0	1	1	-1	0	1	-1	0	1	-1	0	0.35
13	0	0	1	-1	-1	0	1	0	1	-1	1	-1	0	0.07
14	0	0	1	-1	0	1	-1	1	-1	0	-1	0	1	0.76
15	0	0	1	-1	1	-1	0	-1	0	1	0	1	-1	0.27
16	0	1	-1	0	-1	0	1	1	-1	0	0	1	-1	0.08
17	0	1	-1	0	0	1	-1	-1	0	1	1	-1	0	1.12
18	0	1	-1	0	1	-1	0	0	1	-1	-1	0	1	0.1
19	0	-1	1	0	-1	1	0	-1	1	0	-1	1	0	0.26
20	1	-1	1	0	0	-1	1	0	-1	1	0	-1	1	1.47
21	1	-1	1	0	1	0	-1	1	0	-1	1	0	-1	0.04
22	1	0	-1	1	-1	1	0	0	-1	1	1	0	-1	0.39
23	1	0	-1	1	0	-1	1	1	0	-1	-1	1	0	0.1
24	1	0	-1	1	1	0	-1	-1	1	0	0	-1	1	0.87
25	1	1	0	-1	-1	1	0	1	0	-1	0	-1	1	0.12
26	1	1	0	-1	0	-1	1	-1	1	0	1	0	-1	0.02
27	1	1	0	-1	1	0	-1	0	-1	1	-1	1	0	0.75

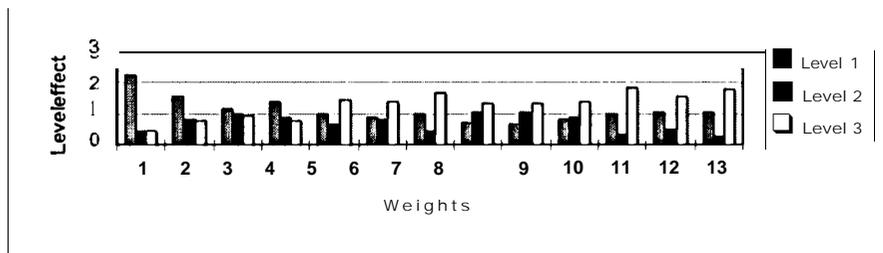


Figure 3. Cost effects of levels 1,2,3 of the 13 weights after the 2nd iteration

Step 4. Modify the values of each weight (shrink the search interval for the weight) in the direction of the weight value that produced the minimum cost. From Figure 2, comparing the effects of the 3 levels of W1, one notices that Level 3 of weight W1 produces minimum cost, therefore the weights will move in its direction. The new levels of W1, in the 2nd iteration, could be calculated as:

$$W1_1^{new} = W1_2$$

$$W1_2^{new} = W1_2 + (W1_3 - W1_2)/2$$

$$W1_3^{new} = W1_3$$

Figure 3 indicates the cost effect after the new weight value was calculated. Reducing the search interval at half of its value and selecting to continue the search in the half that is closest to the level of the weight that produced minimum cost is a straightforward approach to calculate the new weight levels. By dividing the interval in 2 at each iteration, the search converges very quickly (the search interval for a weight shrinks 3 orders of magnitude in 10 steps). Figure 4 illustrates the fast convergence to the target value of a given cost, thus resulting in a quick learning process.

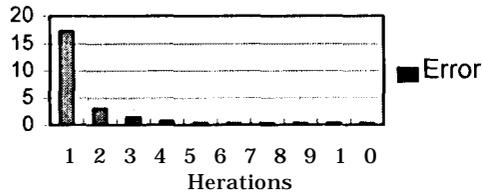


Figure 4 Learning expressed as decreasing modeling error

III. DISCUSSION

Shrinking the search interval in half at every step, narrows down the search space very rapidly. In the particular way the decision is made as of which **half** to choose, only “wide valleys” in the cost function space would guide the search to the optimal solution. Any “narrow valley” would be missed. A solution would be to divide the search into more levels, increasing thus the size of the array, or shrinking at lower pace, increasing thus the power of the search at the price of an increased number of iterations. To improve the result a local search (e.g. gradient-based) may follow, e.g. as in [Gold96], where such a combination outperforms Simulated Annealing and Genetic Algorithms.

Another aspect that needs further work is obtaining the appropriate orthogonal array for the problem posed. Tables of Orthogonal Arrays are available, however they include some particular OA only. For the general case of arbitrary size NN, a generative mechanism to produce the OA of appropriate size is needed.

IV. CONCLUSION

The paper proposed the use of orthogonal arrays to guide the search for weight values which minimize a certain cost objective function for a neural network. The idea of applying orthogonal arrays to guide the search appears promising as the search is global, done in parallel, in multiple points, similarly to searches guided by genetic algorithms. In the straightforward application of the method (as in the example illustrated here), the interval in which the solution is searched is reduced in half at every iteration, this leading to very rapid convergence to an approximate solution (a shrink of 3 orders of magnitude every 10 iterations) which however is non-optimal. Modifications of the technique could get solutions closer to optimal, at the price of an increased number of iterations.

V. ACKNOWLEDGEMENT

The research described in this paper was performed by the Center for Space Microelectronics Technology, Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the Ballistic Missile Defense Organization, through an agreement with the National Aeronautics and Space Administration.

VI. REFERENCES

- [Gold96] Gold, S. Comparison of Global Optimization Methods, TR MIE Dept. U. Mass., 1996
- [Kota93] Kota, S. and Chiou, S. Use of orthogonal arrays in mechanism synthesis, Mechanical Machine Theory, Vol. 28, pp 777-794, 1993
- [Tag87] Taguchi, G. and Konishi, S. *Orthogonal Arrays and Linear Graphs*, ASI Press, 1987