Brill Academic Publishers
P.O. Box 9000, 2300 PA Leiden,
The Netherlands

Lecture Series on Computer and Computational Sciences
Volume 1, 2006, pp. 1-4

Integer Weight Higher-Order Neural Network Training Using Distributed Differential Evolution

M.G. Epitropakis, **V.P. Plagianakos**, **M.N. Vrahatis**[1]

Computational Intelligence Laboratory (CI Lab), Department of Mathematics,
University of Patras Artificial Intelligence Research Center (UPAIRC),
University of Patras, GR–26110 Patras, Greece.
e-mail: {mikeagn, vpp, vrahatis}@math.upatras.gr

Received 1 July, 2006; accepted in revised form 31 July, 2006

Abstract: We study the class of Higher-Order Neural Networks and especially the Pi-Sigma Networks. The performance of Pi-Sigma Networks is evaluated through several well known neural network training benchmarks. In the experiments reported here, Distributed Evolutionary Algorithms for Pi-Sigma networks training are presented. More specifically the distributed version of the Differential Evolution algorithm has been employed. To this end, each processor is assigned a subpopulation of potential solutions. The subpopulations are independently evolved in parallel and occasional migration is employed to allow cooperation between them. The proposed approach is applied to train Pi-Sigma networks using threshold activation functions. Moreover, the weights and biases were confined to a narrow band of integers, constrained in the range $[-32, 32]$, thus they can be represented by just 6 bits. Such networks are better suited for hardware implementation than the real weight ones. Preliminary results suggest that this training process is fast, stable and reliable and the distributed trained Pi-Sigma network exhibited good generalization capabilities.

Keywords: Backpropagation Neural Networks, Integer Weight Neural Networks, Threshold Activation Functions, 'Hardware-Friendly' Implementations, 'On-chip' Training, Higher-Order Neural Networks, Pi-Sigma Networks, Distributed Differential Evolution.

Mathematics Subject Classification: 62M45, 68T10, 92B20

1 Introduction

In this contribution, we study the class of Higher-Order Neural Networks (HONNs) and in particular Pi-Sigma Networks (PSNs), which were introduced by Shin and Ghosh [6]. Although PSNs employ fewer weights and processing units than HONNs they manage to indirectly incorporate many of the capabilities and strengths of HONNs. PSNs have addressed effectively several difficult tasks, such as zeroing polynomials [1] and polynomial factorization [3]. Here, we compare PSN's performance against Feedforward Neural Networks (FNNs) on several well known neural network training problems. In our experiments, we trained PSNs with small integer weights and threshold activation functions, utilizing a Distributed Evolutionary Algorithm. More specifically, a distributed modified version of the Differential Evolution (DE) [5, 8] algorithm has been used. DE has proved to be an effective and efficient optimization method on numerous hard real-life problems [4, 5, 7, 9, 10]. The distributed DE algorithms has been designed keeping in mind that

[1]Corresponding author: e-mail: vrahatis@math.upatras.gr, Phone: +30 2610 997374, Fax: +30 2610 992965

the resulting integer weights and biases require less bits to be stored and the digital arithmetic operations between them are easier to be implemented in hardware. If the network is trained in a constrained weight space, smaller weights are found and less memory is required. On the other hand, the network training procedure can be more effective and efficient when larger integer weights are allowed. Thus, for a given application a trade off between effectiveness and memory consumption has to be considered.

The remaining of this paper is organized as follows. Section 2 briefly describes the mathematical model of PSNs. Section 3 is devoted to the presentation of the distributed DE optimization algorithm. The paper ends with preliminary experimental results and a discussion in Section 4.

2 Higher-Order Neural Networks

Higher-order Neural Networks (HONNs) expand the capabilities of standard FNNs by including input nodes which provide the network with a more complete understanding of the input patterns and their relations. Basically, the inputs are transformed so that the network does not have to learn the most basic mathematical functions, such as squares, cubes, or sines. The inclusion of these functions do enhance the network's understanding of a given problem and has been shown to accelerate training on some applications. However, typically only second order networks are considered in practice. The main disadvantage of HONNs is that the required number of weights increases exponentially with the dimensionality of the input patterns. On the other hand, a Pi–Sigma Network (PSN) utilizes product (instead of summation) nodes as the output units to indirectly incorporate the some of the capabilities of HONNs, while using fewer weights and processing units. Specifically, PSN is a multilayer feedforward network that outputs products of sums of the input components. It consists of an input layer, a single 'hidden' (or middle) layer of summing units, and an output layer of product units. The weights connecting the input neurons to the neurons of the middle layer are adapted during the learning process by the training algorithm, while those connecting the neurons of the middle layer to the product units of the output layer are fixed. For this reason the middle layer is not actually hidden and the training process can be simplified and accelerated.

Let the input $x = (1, x_1, x_2, \ldots, x_N)^\top$, be an $(N+1)$-dimensional vector, with x_k denoting the k-th component of x. Each neuron in the middle layer computes the sum of the products of each input with the corresponding weight. Thus, the output of the j-th neuron in the middle layer is given by the sum: $h_j = w_j^\top x = \sum_{k=1}^{N} w_{kj} x_k + w_{0j}$, where $j = 1, 2, \ldots, K$ and w_{0j} denotes a bias term. Output neurons compute the product of the aforementioned sums and apply an activation function on this product. An output neuron returns $y = \sigma\left(\prod_{j=1}^{K} h_j\right)$, where $\sigma(\cdot)$ denotes the activation function. The number of neurons in the middle layer defines the order of the PSN. This type of networks are based on the idea that the input of a K-th order processing unit can be represented by a product of K linear combinations of the input components. Assuming that $(N+1)$ weights are associated with each summing unit, there is a total of $(N+1)K$ weights and biases for each output unit. If multiple outputs are required (for example, in a classification problem), an independent summing layer is required for each one. Thus, for an M-dimensional output vector y, a total of $\sum_{i=1}^{M}(N+1)K_i$ adjustable weight connections are needed, where K_i is the number of summing units for the i-th output. This allows great flexibility as the output layer indirectly incorporates the some of the capabilities of HONNs with a smaller number of weights and processing units.

Although FNNs and HONNs can be simulated in software, hardware implementation is required in real life applications, where high speed of execution is necessary. The natural implementation of FNNs or HONNs (because of their modularity) is a distributed (or parallel) one. In the next section we briefly review the distributed DE algorithm.

3 Neural Network Training Using the Distributed DE Algorithm

Differential Evolution (DE) is a minimization method, capable of handling non-differentiable, discontinuous and multimodal objective functions. The method requires few, easily chosen, control parameters. Extensive experimental results have shown that DE has good convergence properties and outperforms other well known evolutionary algorithms. The original DE algorithm as well as its distributed implementation have been successfully applied to FNN training [4, 5]. Distributed Differential Evolution (DDE) for Pi-Sigma networks training is presented here. More specifically the distributed version of the Differential Evolution algorithm has been employed. To this end, each processor is assigned a subpopulation of potential solutions. The subpopulations are independently evolved in parallel and occasional migration is employed to allow cooperation between them. The migration of the best individuals is controlled by the migration constant. A good choice for the migration constant is one that allows each subpopulation to evolve for some iterations independently before the migration phase actually occur. Extensive description of the DDE can be found in [5, 9].

The modified DDE maintains a population of potential integer solutions, *individuals*, to probe the search space. The population of individuals is randomly initialized in the optimization domain with *NP*. At each iteration, called *generation*, new individuals are generated through the combination of randomly chosen individuals of the current population. Starting with a population of *NP* integer weight vectors, $w_g^i, i = 1, \ldots, NP$, where g denotes the current generation, each weight vector undergoes mutation to yield a mutant vector, u_{g+1}^i. The mutant vector is obtained through one of the the following equations:

$$u_{g+1}^i = w_g^{\text{best}} + F(w_g^{r_1} - w_g^{r_2}), \tag{1}$$

$$u_{g+1}^i = w_g^{r_1} + F(w_g^{r_2} - w_g^{r_3}), \tag{2}$$

where w_g^{best} denotes the best member of the current generation and $F > 0$ is a real parameter, called *mutation constant* that controls the amplification of the difference between two weight vectors. Moreover, $r_1, r_2, r_3 \in \{1, 2, \ldots, i-1, i+1, \ldots, NP\}$ are random numbers mutually different and different from the running index i. Obviously, the mutation operator results in a real weight vector. As our aim is to maintain an integer weight population at each generation, each component of the mutant weight vector is rounded to the nearest integer. Additionally, if the mutant vector is not in the range $[-32, 32]^N$, we take: $u_{g+1}^i = \text{sign}(u_{g+1}^i) \times \left(\left|u_{g+1}^i\right| \mod 32\right)$. During recombination, for each component j of the integer mutant vector, u_{g+1}^i, a random real number, r, in the interval $[0, 1]$ is obtained and compared with the *crossover constant*, *CR*. If $r \leqslant CR$ we select as the j-th component of the trial vector, v_{g+1}^i, the corresponding component of the mutant vector, u_{g+1}^i. Otherwise, we pick the j-th component of the target vector, w_g^i. It must be noted that the result of this operation is a 6-bit integer vector.

4 Experiments and Discussion

In this study the DDE algorithm is applied to train PSNs with integer weights and threshold activation functions. Here, we report preliminary results on the MONK's problem [11]. These three problems from the UCI Machine Learning Repository [2] are difficult binary classification tasks which have been used for comparing the generalization performance of learning algorithms. We call DDE_1 the distributed DE algorithm that uses Relation (1) as mutation operator and DDE_2 the algorithm that uses Relation (2). We have compared the DDE_1 and DDE_2 algorithms utilizing threshold functions and 6-bit integer weights.

For the experiments, we have conducted 1000 independent simulations for each algorithm, using a distributed computation environment consisting of 16 nodes. We have used fixed values for the

mutation, crossover and migration constants, $F = 0.5$, $CR = 0.7$, and $\phi = 0.1$, respectively. The termination criterion applied to the learning algorithm was either a training error less than 0.01 or 5000 iterations. The generalization capability of the DDE trained integer weight PSNs is exhibited in Table 1. The results indicate that the training of PSNs with integer weights and thresholds, using the modified DDE is efficient and promising. The learning process was fast and reliable, and the performance of the DDE stable. Additionally, the trained PSNs exhibited good generalization capabilities.

Table 1: Generalization results for the MONK's Problems

Problem	Network Topology	Algorithm	Generalization (%) Min	Max	$Mean$	$St.D.$
MONK-1	17-2-1	DDE_1	81	100	94.5	3.2
MONK-1	17-2-1	DDE_2	81	100	94.6	3.2
MONK-2	17-2-1	DDE_1	89	100	96.4	1.7
MONK-2	17-2-1	DDE_2	87	100	96.0	2.0
MONK-3	17-2-1	DDE_1	79	99	92.2	2.8
MONK-3	17-2-1	DDE_2	76	99	91.3	3.5

References

[1] D.S. Huang, H.H.S. Ip, K.C.K. Law, and Z. Chi, Zeroing Polynomials Using Modified Constrained Neural Network Approach, *IEEE Transactions on Neural Networks*, **16**, no. 3, 721–732 (2005)

[2] P.M. Murphy and D.W. Aha, UCI Repository of machine learning databases, Irvine, CA: University of California, Department of Information and Computer Science, (1994)

[3] S. Perantonis, N. Ampazis, S. Varoufakis, and G. Antoniou, Constrained Learning in Neural Networks: Application to Stable Factorization of 2D Polynomials, *Neural Processing Letters*, **7**, 5–14 (1998)

[4] V.P. Plagianakos and M.N. Vrahatis, Neural network training with constrained integer weights, *Congress on Evolutionary Computation (CEC'99)*, (1999)

[5] V.P. Plagianakos and M.N. Vrahatis, Parallel evolutionary training algorithms for 'hardware–friendly' neural networks, *Natural Computing*, **1**, 307–322 (2002)

[6] Y. Shin and J. Ghosh, The pi-sigma network: An efficient higher-order neural network for pattern classification and function approximation, *International Joint Conference on Neural Networks*, vol. 1, 13–18 (1991)

[7] R. Storn, System Design by Constraint Adaptation and Differential Evolution, *IEEE Transactions on Evolutionary Computation*, **3**, 22–34 (1999)

[8] R. Storn and K. Price, Differential evolution – a simple and efficient adaptive scheme for global optimization over continuous spaces, *Journal of Global Optimization*, **11**, 341–359 (1997)

[9] D.K. Tasoulis, N.G. Pavlidis, V.P. Plagianakos, and M.N. Vrahatis, Parallel Differential Evolution, *IEEE 2004 Congress on Evolutionary Computation (CEC2004)*, (2004)

[10] D.K. Tasoulis, V.P. Plagianakos, and M.N. Vrahatis, Clustering in Evolutionary Algorithms to Efficiently Compute Simultaneously Local and Global Minima, *Congress on Evolutionary Computation (CEC 2005)*, (2005)

[11] S.B. Thrun *et al.*, The MONKs Problems: A performance comparison of different learning algorithms. Technical Report, Carnegie Mellon University, CMU-CS-91-197, (1991).