Higher-Order Neural Networks Training Using Differential Evolution

M.G. Epitropakis, V.P. Plagianakos, and **M.N. Vrahatis***

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

Key words Backpropagation Neural Networks, Higher-Order Neural Networks, Pi-Sigma Networks, Differential Evolution.
Subject classification 62M45, 68T10

In this contribution, we study the class of Higher-Order Neural Networks, especially Pi-Sigma Networks. The performance of Pi-Sigma Networks is considered through well known neural network training problems. In our experiments, for the training process, we used Evolutionary Algorithms and more specifically the Differential Evolution algorithm. Preliminary results suggest that this training process is fast, stable and reliable and the trained Pi-Sigma network exhibited good generalization capabilities.

1 Introduction

We study the class of Higher-Order Neural Networks (HONNs), and in particular Pi-Sigma Networks (PSNs) which were introduced by Shin and Ghosh [8,9]. Although PSNs employ fewer weights and processing units than HONNs they manage to incorporate indirectly the capabilities and strengths of HONNs. PSNs have addressed effectively several difficult tasks, such as zeroing polynomials [1–3], and polynomial factorization [5]. In this contribution, we compare PSNs' performance with Feedforward Neural Networks (FNNs) on well known neural network training problems. In our experiments training is performed through an Evolutionary Algorithm, namely the Differential Evolution (DE) [11] algorithm. DE has proved to be an effective and efficient optimization method on numerous hard real-life problems [6, 7, 10, 12, 13]. The remaining of this contribution is organized as follows. Section 2 outlines the mathematical model of FNNs, while Section 3 briefly describes the PSNs. Section 4 is devoted to the presentation of the DE global optimization method. The paper ends with preliminary experimental results and a discussion in Section 5.

2 Feedforward Neural Networks

Although many models of neural networks have been proposed Feedforward Neural Networks (FNNs) are the most commonly encountered. FNNs consist of a large number of interconnected, simple, processing units, called neurons, or nodes. Each node calculates the inner product of the incoming signals with its weights, adds the bias to the result, and passes this sum to its activation function. In a multilayer network, neurons are organized in layers with no feedback connections.

The efficient supervised training of FNNs is a subject of considerable ongoing research and numerous algorithms have been proposed to this end. A common training approach is to minimize the network learning error, which is a measure of network performance. The computation of the learning error is usually based on the difference between the actual output vector of the network and the desired output vector (supervised learning). The rapid computation of a set of weights that minimizes this error function is a difficult task since the number of

* Corresponding author: e-mail: vrahatis@math.upatras.gr, Phone: +302610997374, Fax: +302610992965

network weights is usually large and the error function generates a complicated surface in the weight space, possessing multitudes of local minima and broad flat regions adjoined to narrow steep ones, that need to be searched to locate an "optimal" weight set.

More formally, assuming that the l-th layer of an FNN contains N_l neurons, and $l = 1, \ldots, M$, the operation of the network is described by the following equations:

$$\mathrm{net}_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^{l-1,l} y_i^{l-1}, \qquad y_j^l = f(\mathrm{net}_j^l),$$

where, $w_{ij}^{l-1,l}$ is the weight connecting the i-th neuron of the $(l-1)$-th layer to the j-th neuron of the l-th layer; y_j^l is the output of the j-th neuron of the l-th layer; and $f(\cdot)$ is the j-th's neuron activation function. FNNs typically use analogue activation functions, such as the well known sigmoid function:

$$s(\mathrm{net}_j^l) = \frac{1}{1 + \exp(-\beta\, \mathrm{net}_j^l)},$$

where the factor β (conventionally, $\beta = 1$) is introduced to achieve slope modification. The training process can be realized by minimizing the error function E defined as:

$$E = \frac{1}{2} \sum_{p=1}^{P} \sum_{j=1}^{N_M} (y_{j,p} - t_{j,p})^2 = \sum_{p=1}^{P} E_p, \tag{1}$$

where p is an index over the training input-output pairs, P is the number of these pairs, N_M is the number of neurons in the output layer and $(y_{j,p} - t_{j,p})^2$ is the squared difference between the output $y_{j,p}$ of the j-th neuron in the output layer, for pattern p, and the corresponding target value $t_{j,p}$.

To simplify the formulation of the above equations, let us use a unified notation for the weights. For an FNN with n weights, let w be a column weight vector with components $w_1, w_2, \ldots, w_n$, defined on the n–dimensional Euclidean space $\mathbb{R}^n$, and $w^\star$ be an optimal weight vector with components $w_1^\star, w_2^\star, \ldots, w_n^\star$. Also, let E be the batch error measure defined in Eq. (1), and $\nabla E(w)$ be the gradient vector of the error function E at w. Each pass through the entire training set is called an *epoch*. The batch training of an FNN is consistent with the theory of unconstrained optimization as it uses information from all the training set (i.e. the true gradient is employed) and can be viewed as the minimization of the error function E. This minimization task corresponds to the weight update by epoch and, requires a sequence of weight vectors $\{w^k\}_{k=0}^{\infty}$, where k indicates epochs. Successful training implies that $\{w^k\}_{k=0}^{\infty}$ converges to the point $w^\star$ that minimizes E.

3 Higher-Order Neural Networks

Higher-order Neural Networks (HONNs) expand standard FNNs by including nodes at the input layer that provide the network with a more complete understanding of the input patterns and their relations. Basically, the inputs are transformed in a well understood mathematical way so that the network does not have to learn the most basic mathematical functions. These functions do enhance the network's understanding of a given problem, while they transform the inputs via higher-order functions such as squares, cubes, or sines. Although, this technique has been shown to accelerate training on some applications, 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.

A Pi–Sigma Network (PSN) utilizes product cells as the output units to indirectly incorporate the capabilities of HONNs, while using fewer weights and processing units. More 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, 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.

Specifically, 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. We define as the order of the network, the number of neurons in the middle layer.

This type of networks is 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, 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 capabilities of HONNs with a smaller number of weights and processing units.

4 Training Neural Networks Using the Differential Evolutionary Algorithm

Differential Evolution (DE) is a minimization method, capable of handling non-differentiable, discontinuous and multimodal objective functions. To fulfill this requirement, DE has been designed as a stochastic parallel direct search method, which utilizes concepts borrowed from the broad class of evolutionary algorithms. 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. DE has been successfully applied to FNN training [6, 7].

DE maintains a population of potential solutions, *individuals*, to probe the search space. The population of individuals is randomly initialized in the optimization domain with *NP*, n-dimensional vectors, following a uniform probability distribution and is evolved over time to explore the search space and locate the minima of the objective function. *NP* is fixed throughout the training process. At each iteration, called *generation*, new individuals are generated through the combination of randomly chosen individuals of the current population. This operation in our context can be referred to as *mutation*. The resulting vectors are then mixed with other predetermined vectors - the *target* vectors - and this operation can be called *recombination*. We briefly describe the mutation and recombination operations. Starting with a population of *NP* 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 the following equation:

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

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, $r_1, r_2 \in \{1, 2, \ldots, i-1, i+1, \ldots, NP\}$ are random numbers mutually different and different from the running index i. During recombination, for each component j of the 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.

5 Experiments and Discussion

In this study the DE algorithm is applied to train PSNs. Our experimental results on many classical neural network training problems indicate that this is a promising approach. Here, we report preliminary results from the handwritten digits classification problem, which is part of the UCI Repository of Machine Learning Databases [4],

and is characterized by a real-valued training set of approximately 7500 patterns. In this experiment, a digit database has been assembled by collecting 250 samples from 44 independent writers. The samples written by 30 writers are used for training, and the rest are used for testing. The training set consists of 7494 real valued samples and the test set of 3498 samples.

For this problem, we have used the fixed values of $F = 0.5$ and $CR = 0.7$ as the mutation and crossover constants respectively. We have trained one network for each digit and have conducted 250 independent simulations for each network. The termination criterion applied to the learning algorithm was either a training error less than 0.001 or 5000 iterations.

A different PSN is trained to discriminate samples from each problem class. To this end, in order to train and test the network for each class, we used all the patterns from the training, and the test set respectively.

To present the experimental results the following notation is used: Min indicates the minimum generalization capability of the trained PSNs; Max is the maximum generalization capability; $Mean$ is the average generalization capability; $St.D.$ is the standard deviation of the generalization capability. The experimental results exhibited in Table 1 indicate that the training of PSNs using Evolutionary Algorithms (DE in particular) is efficient and promising. The learning process was fast, stable, and reliable and the trained PSNs exhibited good generalization capabilities.

Table 1 Generalization results for the Pendigit Problem.

Network Topology	Generalization (%)			
	Min	Max	$Mean$	$St.D.$
16-2-1	80.475	87.650	84.256	1.4171
16-3-1	80.560	87.993	84.521	1.4165

Acknowledgements The authors would like to thank the European Social Fund, Operational Program for Educational and Vocational Training II (EPEAEK II), and particularly the Program "Pythagoras" for funding this work. This work was also partially supported by the University of Patras Research Committee through a "Karatheodoris" research grant.

References

[1] D.S. Huang and Z. Chi, Neural networks with problem decomposition for finding real roots of polynomials, International Joint Conference on Neural Networks, 25–30 (2001)
[2] D.S. Huang, A Constructive Approach for Finding Arbitrary Roots of Polynomials by Neural Networks, IEEE Trans. On Neural Networks, **15**, no. 2, 477–491 (2004)
[3] D.S. Huang, H.H.S. Ip, K.C.K. Law, and Z. Chi, Zeroing Polynomials Using Modified Constrained Neural Network Approach, IEEE Trans. On Neural Networks, **16**, no. 3, 721–732 (2005)
[4] 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)
[5] 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)
[6] V.P. Plagianakos and M.N. Vrahatis, Neural network training with constrained integer weights, Congress on Evolutionary Computation (CEC'99), (1999)
[7] V.P. Plagianakos and M.N. Vrahatis, Parallel evolutionary training algorithms for 'hardware–friendly' neural networks, Natural Computing, **1**, 307–322 (2002)
[8] 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)
[9] Y. Shin and J. Ghosh, Ridge polynomial networks, IEEE Transactions on Neural Networks, **6**, 610–622 (1995)
[10] R. Storn, System Design by Constraint Adaptation and Differential Evolution, IEEE Transactions on Evolutionary Computation, **3**, 22–34 (1999)
[11] 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)
[12] D.K. Tasoulis, N.G. Pavlidis, V.P. Plagianakos, and M.N. Vrahatis, Parallel Differential Evolution, IEEE 2004 Congress on Evolutionary Computation (CEC2004), (2004)
[13] 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)