Training Multilayer Networks with Discrete Activation Functions

V.P. Plagianakos[1,3], G.D. Magoulas[2,3], N.K. Nousis[1,3], and M.N. Vrahatis[1,3]

[(1)]Department of Mathematics, University of Patras,
GR-261.10 Patras, Greece
e–mail: {vpp,nousis,vrahatis}@math.upatras.gr

[(2)]Department of Information Systems and Computing, Brunel University,
Uxbridge UB8 3PH, UK.
e–mail: George.Magoulas@brunel.ac.uk

[(3)]University of Patras Artificial Intelligence Research Center–UPAIRC

Abstract

Efficient training of multilayer networks with discrete activation functions is a subject of considerable ongoing research. The use of these networks greatly reduces the complexity of the hardware implementation, provides tolerance to noise and improves the interpretation of the internal representations. Methods available in the literature mainly focus on two-state (binary) nodes and try to train these networks by approximating the gradient and modifying appropriately the gradient descent. However, they exhibit slow convergence speed and low possibility of success compared to networks with continuous activations. In this work, we propose an evolution-motivated approach, which is eminently suitable for networks with discrete output states and compare its performance with four other methods.

1 Introduction

Although many different models of neural networks have been proposed, multilayer neural networks are the most common. These networks consist of many interconnected identical simple processing units, also called neurons. Each node calculates the dot product of the incoming signals with its weights, adds the bias to the resultant, and passes the calculated sum through its activation function. In a multilayer network the neurons are organized into layers with no feedback connections.

Although units with discrete activation functions have been superseded to a large extent by the more computationally powerful units with analog activation function, still multilayer Networks with Discrete Activations (NDAs) are important in that they can handle many of the inherently binary tasks that neural networks are used for. Their internal representations are clearly interpretable, they are computationally simpler to understand than networks with sigmoid units and provide a starting point for the study of neural networks properties. Furthermore, when using hard–limiting units we can understand better the relationship between the size of the network and the complexity of the training [5]. In [2], it has been demonstrated that NDAs with only one hidden layer, can create any decision region that can be expressed as a finite union of polyhedral sets when there is one unit in the input layer. Moreover, artificially created examples were given where these networks create non convex and disjoint decision regions. Finally, discrete activation functions facilitate and reduce the complexity of neural network implementations in digital hardware and are much less costly to fabricate.

Various modifications of the gradient descent have been presented to train NDAs [1, 3, 4, 6, 14, 17]. However, these methods require to a certain degree, depending on the method, that the learning task should be static. Thus, the network is trained by applying various problem–dependent heuristics "off–line" during simulation and, then, the weights are transferred to the hardware. But many real–life applications may not be static, i.e. input data may continue to change even after the hardware implementation. In such cases an algorithm capable for "on–chip" training is needed.

0-7803-7044-9/01/$10.00 ©2001 IEEE

In this paper we propose evolution–motivated strategies that provide the potential advantage of continuing the training process in hardware, when purely threshold activation functions are used.

The paper is organized as follows. In the next sections the training problem of NDAs is formulated and current approaches to solve it are discussed. Then, in Section 3 an alternative method is described. Section 4 presents experiments and comparative results. Finally, Section 5 presents concluding remarks.

2 Training Methods for Networks with Discrete Activations

Consider a multilayer neural network with discrete activations consisting of L layers, in which the first layer denotes the input, the last L is the output, and the intermediate layers are the hidden layers. It is assumed that the (l-1) layer has N_{l-1} units. These units operate according to the following equations :

$$net_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^{l-1,l} y_i^{l-1} + \theta_j^l, \tag{1}$$

$$y_j^l = \sigma^l\Big(net_j^l\Big), \tag{2}$$

where net_j^l is the net input to the jth unit at the lth layer, $w_{ij}^{l-1,l}$ is the connection weight from the ith unit at the $(l-1)$ layer to the jth unit at the lth layer, y_i^l denotes the output of the ith unit belonging to the lth layer, θ_j^l denotes the threshold of the jth unit at the lth layer, and σ is the activation function.

Multilayer neural networks are usually based on units with analog activation functions, as the well known sigmoid :

$$s(net_j^l) = \frac{1}{1+e^{-\beta net_j^l}}, \tag{3}$$

where the factor β is introduced to achieve slope modification. For high values of β the sigmoid unit approximates the hard-limiting unit [7], i.e. $\sigma^l(net_j^l) =$ "$true$", if $net_j^l \geq 0$, and "$false$" otherwise.

Let us now define the error for a discrete unit as follows: $e_j(t) = d_j(t) - y_j^L(t)$, for $j = 1, 2, ..., N_L$, where $d_j(t)$ is the desired response at the jth unit of the output layer at the input pattern t, and $y_j^L(t)$ is the output at the kth unit of the output layer L. For a fixed, finite set of input-output cases, the square error over the training set which contains T representative cases is:

$$E = \sum_{t=1}^{T} E(t) = \sum_{t=1}^{T} \sum_{j=1}^{N_L} e_j^2(t). \tag{4}$$

The *fixed–increment* rule and the *fractional correction* rule, both described in [8], were the first training methods for training single layer discrete networks. Nowadays, the most common multilayer neural network training algorithm, the back–propagation (BP), [12], that makes use of the gradient descent, cannot be applied directly to networks of units with discrete output states, since discrete activation functions (such as hardlimiters) are non–differentiable.

However, various modifications of the gradient descent have been presented, such as the well known MRII [17]. Another method proposed by Tom [14] uses a hybrid activation function that is a linear combination of analog (sigmoid) and discrete (hard-limiting) functions depending on the values of a heuristic parameter. Thus, for a specific range of values a unit can be purely analog having a sigmoid activation, such as in Relation (3), while for other values it becomes purely binary. For intermediate values the network is a hybrid with activation functions that are differentiable everywhere except at net_j^l.

Bartlett in [1] introduced another approach by defining the weights as random variables with smooth distribution functions and proposed an algorithm that uses an approach that is similar to BP to adjust the parameters of the weights' distributions. In [4], Corwin suggested to train NDAs with progressively steeper analog functions to facilitate training. Thus, in his experiments he used values such as $\beta \in \{2, 3, 5, 10\}$ to alter the shape of the sigmoid from time to time during training.

Finally, Goodman in [3] proposed an approximation to gradient descent, the so-called *pseudo-gradient* training method. The pseudo–gradient assumes that units with two discrete output states are used, i.e. f (or $-f$) for "false" and t (or $+t$) for "true", where f, t are real positive numbers and $f < t$, instead of the classical 0 and 1 (or -1, and $+1$). Real positive values prevent units from saturating, give to the logic "false" some power of influence over the next layer of the network, and help the justification of the approximated gradient value.

Note that the idea of the pseudo-gradient was first introduced in training discrete recurrent neural networks [18, 19] and extented to NDAs [3]. The method approximates the true gradient of the error function with respect to the weights, i.e. $\nabla E(w)$, by introduc-

ing an analog set of values for the outputs of the hidden layer units and the output layer units.

Thus, it is assumed that (2) can be written as :

$$y_j^l = \tilde{\sigma}^l\Big(s(net_j^l)\Big), \tag{5}$$

where $\tilde{\sigma}$(x)= "$true$", if x$\geq$ 0.5, and "$false$" otherwise if $s(\cdot)$ is defined in $[0,1]$. If $s(\cdot)$ is defined in $[-1,1]$, then $\tilde{\sigma}$(x)= "$true$" if x$\geq$ 0, and "$false$" otherwise.

Using the chain rule, the pseudo–gradient is computed :

$$\frac{\widetilde{\partial E}}{\partial w_{ij}^{l-1,l}} = \widetilde{\delta_j^l} y_i^{l-1}, \tag{6}$$

$$\widetilde{\delta_j^L} = \Big(d_j - s(net_j^L)\Big) s'(net_j^L), \tag{7}$$

$$\widetilde{\delta_j^l} = s'(net_j^l) \sum_n w_{jn}^{l,l+1} \widetilde{\delta_n^{l+1}}, \text{ for } l \in [2, L-1]. \tag{8}$$

Equations (7) and (8), in which $s'(net_j^l)$ is the derivative of the analog activation function, specify the back-propagating error signal $\tilde{\delta}$ for the output layer and the hidden layers respectively.

By using real positive values for "true" and "false" we ensure that the pseudo–gradient will not reduce to zero when the output is "false". Note also that we do not use σ' which is zero everywhere and non–existent at zero. Instead, we use s' which is always positive, so $\tilde{\delta}_j^l$ gives an indication of the direction and magnitude of a step up or down as a function of net_j^l on the surface of the error function E. The justification of the pseudo–gradient can be found in any one of [3, 18, 19], and is based on the idea of using the gradient of a sigmoid as a heuristic hint instead of the true gradient.

However, as pointed out in [3] the value of the pseudo–gradient is not accurate enough, so gradient descent based training in NDAs is considerably slow when compared with BP training in multilayer neural networks with continuous activations.

Based on the idea of the pseudo-gradient, in [6] an attractive alternative has been proposed. This method exploits the imprecise information regarding the error function and the approximated gradient, like the pseudo–gradient method does, however it has an improved convergence speed and is potentially useful in situations where the pseudo–gradient method fails to converge.

3 Evolution Strategies for training NDAs

In this section we introduce a novel approach based on Evolutionary Algorithms (EAs) for training NDAs with purely threshold units. EAs are adaptive stochastic search methods which mimic the metaphor of natural biological evolution. Differently from other adaptive stochastic search algorithms, evolutionary computation techniques operate on a set of potential solutions, which is called *population*, applying the principle of survival of the fittest to produce better and better approximations to a solution, and, through cooperation and competition among the potential solutions, they find the optimal one. This approach often helps finding optima in complicated optimization problems more quickly than traditional optimization methods.

In Figure 1, a high level description of a general EA is presented.

```
EVOLUTIONARY ALGORITHM MODEL
{
  //initialise the time counter
  t := 0;
  //initialise the population of individuals
  InitPopulation(P(t));
  //evaluate fitness of all individuals
  Evaluate(P(t));
  //test for termination criterion
  //(time, fitness, etc.)
  while not done do
    t := t + 1;
    //select a sub-population for
    //offspring production
    Q(t) := SelectParents(P(t));
    //recombine the "genes" of selected parents
    Recombine(Q(t));
    //perturb the mated population
    //stochastically
    Mutate(Q(t));
    //evaluate the new fitness
    Evaluate(Q(t));
    //select the survivors for the next
    //generation
    P(t + 1) := Survive(P(t), Q(t));
  end
}
```

Figure 1: High level description of a simple Evolutionary Algorithm.

Here, we use the *Differential Evolution* (DE) strategies [13], which have been designed as stochastic parallel direct search methods that can efficiently handle non differentiable, nonlinear and multimodal objective functions, and require few, easily chosen control parameters. Experimental results have shown that DE

algorithms have good convergence properties and outperform other evolutionary methods [9, 10].

To apply DE algorithms to NDAs' learning we start with a specific number (NP) of N–dimensional weight vectors, as an initial weight population, and evolve them over time. The number of individuals NP is kept fixed throughout the learning process and the population is initialized randomly following a uniform probability distribution. As in EAs, at each iteration of the DE algorithm, called *generation*, new weight vectors are generated by the combination of weight vectors randomly chosen from the population using the relation

$$w_{g+1}^{i} = w_{g}^{i} + \xi \left(w_{g}^{\text{best}} - w_{g}^{i} \right) + \xi \left(w_{g}^{r_1} - w_{g}^{r_2} \right), \quad (9)$$

where w_{g}^{best} is the best member of the previous generation, $\xi > 0$ is a real parameter, called mutation constant, which regulates the contribution of the difference between two weight vectors, and

$$r_1, r_2 \in \{1, 2, \ldots, i-1, i+1, \ldots, NP\}$$

are random integers mutually different and different from the running index i.

The outcoming weight vectors are then mixed with another predetermined weight vector, the *target* weight vector. This operation is called *crossover* and it yields the so–called *trial* weight vector. This vector is accepted for the next generation if and only if it reduces the value of the error function E. This last operation is called *selection*.

In a recent work [11], we showed that the DE stategies can efficiently be used to train networks with arbitrary integer weights. However, in [11] we trained networks "off–line" by altering the gain of the sigmoid activations in such a way that we reached the binary threshold at the end of training. Here, we further expand this approach to train "on–chip" NDAs by explicitly using threshold activations.

4 Experiments and Results

In this section, we present comparative results for the DE algorithm, and the algorithms proposed in [4], [14], [3], [6], which are denoted in the tables below as (GLO), (T), (GZ), and (MVGA), respectively.

4.1 The XOR classification problem

Classification of the four XOR patterns in one of two classes, $\{0, 1\}$, is a sensitive to initial weights and presents a multitude of local minima. We used a 2–2–1 NDAs and set the learning stepsize to 0.1. In all instances, 100 simulations were run and the results are summarized in Table 1, where the following notation is used: *mean* indicates the mean number of function evaluations; *s.d.* the standard deviation of function evaluations; *max* the maximum number of function evaluations; *min* the minimum number of function evaluations and *succ.* the percentage of successful runs.

Algorithm	min	mean	max	s.d.	succ.
DE	252	394.0	1170	111.9	100%
GLO	142	335.6	2303	250.4	84%
T	117	192.2	528	66.4	69%
GZ	5202	5202.7	9964	3530.8	5%
MVGA	122	280.6	334	56.1	50%

Table 1: Results for the XOR Problem.

4.2 The three bit parity problem

A 3–3–1 NDAs receives eight, 3–dimensional binary input patterns and must output an "1" if the inputs have an odd number of ones and "0" if the inputs have an even number of ones. This is a very difficult problem for a NDAs because it must determine the proper parity (the value at the output) for input patterns which differ only by Hamming distance 1. The stepsize was equal to 0.5. The results of 100 simulations are summarized in Table 2.

Algorithm	min	mean	max	s.d.	succ.
DE	640	1732.8	9888	1195.8	100%
GLO	77	146.3	372	50.8	96%
T	70	114.9	448	45.3	88%
GZ	–	–	–	–	0%
MVGA	220	502.7	964	330.8	65%

Table 2: Results for the 3–bit Parity Problem.

4.3 Controlling a lathe cutting process

The cutting tool, driven by a servo-motor, is augmented with a force sensor which returns a signal contaminated by tool chatter and unwanted noise to the controller [15, 16] whose object is to maintain a constant force on the tool by varying the material feed rate. The controller generates a signal to the actuator to effect the necessary optimum feed rate in order to assure the desired product quality. Feed rate demand is the input to the device and the cutting force, as measured by the force sensor on the workpiece, is the device output. A 2-4-1 FNN is used for controlling the process. The neural controller is trained using fuzzified values for the control system error and the error change.

The controller provides a correction signal which passes through an integrator to give the control input to the device. In Table 3 we exhibit results regarding the training performance of the neuro–controller.

Algorithm	min	mean	max	s.d.	succ.
DE	1020	4400.6	15390	2716.2	93%
GLO	–	–	–	–	0%
T	–	–	–	–	0%
GZ	–	–	–	–	0%
MVGA	1200	2666.1	11114	917.2	34%

Table 3: Results from training the neuro–controller.

5 Concluding remarks

Various methods for training networks with discrete activations have been proposed in the literature. However, these methods exhibit slow convergence speed and low percentage of success, as well as poor generalisation compared to networks with continuous activations. It is known that the hardware implementation of the backward passes, which compute the gradient of the error function, is more difficult than the implementation of the forward passes. Thus, in this paper, we proposed an evolution strategy method that does not perform gradient evaluations and is able to train units with purely threshold activations. The behavior of the algorithm has been examined by means of simulations and comparative results have been presented. The performance of the algorithm is promising even when compared with other methods that require the gradient approximations of the error function and train the network by progressively altering the shape of the sigmoid function.

References

[1] P.L. Bartlett, and T. Downs, (1992). "Using random weights to train multilayer networks of hard–limiting units", *IEEE Trans. Neural Networks*, **3**, 202–210.

[2] G.J. Gibson and F.N. Cowan, (1990). "On the decision regions of multi-layer perceptrons", *Proc. IEEE*, **78**, 1590–1594.

[3] R. Goodman and Z. Zeng, (1994). "A learning algorithm for multi–layer perceptrons with hard–limiting threshold units", in *Proc. IEEE Neural Networks for Signal Processing*, 219–228.

[4] E.M. Gorwin, A.M. Logar, and W.J.B. Oldham, (1994). "An iterative method for training multilayer networks with threshold functions", *IEEE Trans. Neural Networks*, **5**, 507–508.

[5] S. E. Hampson and D.J. Volper, (1990). "Representing and learning boolean functions of multivalued features", *IEEE Trans. Systems, Man & Cybernetics*, **20**, 67–80.

[6] G.D. Magoulas, M.N. Vrahatis, T. N. Grapsa, and G.S. Androulakis, (1997). "A training method for discrete multilayer neural networks", in *Mathematics of Neural Networks: Models, Algorithms & Applications*, chapter 41, S. W. Ellacot, J. C. Mason, and I. J. Anderson (eds.), Kluwer Academic Publishers, Operations Research/Computer Science Interfaces series.

[7] W. McCullough and W. H. Pitts, (1943). "A Logical Calculus of the Ideas Imminent in Nervous Activity", *Bulletin Mathematical Biophysics*, **5**, 115–133.

[8] N.J. Nilsson, (1965). *Learning Machines*, McGraw–Hill, NY.

[9] V.P. Plagianakos and M.N. Vrahatis, (1999). "Training neural networks with 3–bit integer weights", in *Proceedings of Genetic and Evolutionary Computation Conference (GECCO'99)*, 910–915.

[10] V.P. Plagianakos and M.N. Vrahatis, (1999). "Neural network training with constrained integer weights", in *Proceedings of Congress on Evolutionary Computation (CEC'99)*, 2007-2013, Washington D.C.

[11] V.P. Plagianakos and M.N. Vrahatis, (2000). "Training Neural Networks with Threshold Activation Functions and Constrained Integer Weights", in *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN'2000)*, Como, Italy.

[12] D.E. Rumelhart and J.L. McClelland eds., (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, **1**, 318–362, MIT Press.

[13] R. Storn and K. Price, (1997). "Differential Evolution – A simple and efficient heuristic for global optimization over continuous spaces", *Journal of Global Optimization*, **11**, 341–359.

[14] D.J. Tom, (1990). "Training binary node feed forward neural networks by back–propagation of error", *Ellectonics Letters*, **26**, 1745–1746.

[15] M. Tomizuka and S. Zhang, (1988). "Modeling and conventional/adaptive PI control of a lathe cutting process", *Transactions ASME*, **110**, 305-354.

[16] G. Tsitouras and R. King, (1997). "Rule-based neural control of mechatronic systems", *Int. J. of Intelligent Mechatronics*, **2**, 1-11.

[17] B. Widrow and R. Winter, (1988). "Neural nets for adaptive filtering and adaptive pattern recognition", *IEEE Computer*, March, 25–39.

[18] Z. Zeng, R. Goodman, and P. Smyth, (1993). "Learning finite state machines with self–clustering recurrent networks", *Neural Computation*, **5**, 976–990.

[19] Z. Zeng, R. Goodman, and P. Smyth, (1994). "Discrete recurrent neural networks for grammatical inference", *IEEE Trans. Neural Netwoks*, **5**, 320–330.