Training Neural Networks with Threshold Activation Functions and Constrained Integer Weights

V.P. Plagianakos
University of Patras, Department of Mathematics,
U.P. Artificial Intelligence Research Center (UPAIRC),
GR-26500 Patras, Greece.
e-mail: vpp@math.upatras.gr

M.N. Vrahatis
University of Patras, Department of Mathematics,
U.P. Artificial Intelligence Research Center (UPAIRC),
GR-26500 Patras, Greece.
e-mail: vrahatis@math.upatras.gr

***Abstract*— Evolutionary neural network training algorithms are presented. These algorithms are applied to train neural networks with weight values confined to a narrow band of integers. We constrain the weights and biases in the range $[-2^{k-1}+1, 2^{k-1}-1]$, for $k = 3,4,5$, thus they can be represented by just k bits. Such neural networks are better suited for hardware implementation than the real weight ones.**

Mathematical operations that are easy to implement in software might often be very burdensome in the hardware and therefore more costly. Hardware–friendly algorithms are essential to ensure the functionality and cost effectiveness of the hardware implementation. To this end, in addition to the integer weights, the trained neural networks use threshold activation functions only, so hardware implementation is even easier. These algorithms have been designed keeping in mind that the resulting integer weights require less bits to be stored and the digital arithmetic operations between them are easier to be implemented in hardware. Obviously, if the network is trained in a constrained weight space, smaller weights are found and less memory is required. On the other hand, as we have found here, the network training procedure can be more effective and efficient when larger weights are allowed. Thus, for a given application a trade off between effectiveness and memory consumption has to be considered.

Our intention is to present results of evolutionary algorithms on this difficult task. Based on the application of the proposed class of methods on classical neural network benchmarks, our experience is that these methods are effective and reliable.

1 Introduction

Artificial Feedforward Neural Networks (FNNs) have been widely used in many application areas in recent years and have shown their strength in solving hard problems in Artificial Intelligence. Although many different models of neural networks have been proposed, multilayered FNNs are the most common. FNNs consist of many interconnected identical simple processing units, called neurons. Each neuron 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 feedforward network the neurons are organized into layers with no feedback connections.

FNNs can be simulated in software, but in order to be utilized in real life applications, where high speed of execution is required, hardware implementation is needed. The natural implementation of an FNN – because of its modularity – is a parallel one. The problem is that the conventional multilayer FNNs, which have continuous weights, are expensive to implement in digital hardware. Another major implementation obstacle is the weight storage. FNNs having integer weights and biases are easier and less expensive to implement in electronics as well as in optics and the storage of the integer weights is much easier to be achieved. Finally, the use of threshold activation functions for all the hidden and output neurons, greatly reduces the complexity of the hardware implementation, because there is no need to design and implement complicated non–linear activation functions.

Another advantage of the FNNs with integer weights and threshold activation functions is that the trained neural network is immune to noise in the training data. Such networks only capture the main feature of the training data. Low amplitude noise that possibly contaminates the training data cannot perturb the discrete weights, because those networks require relatively large variations to "jump" from one integer weight value to another.

In a recent publication [5] we have studied neural networks with arbitrarily integer weights. Here, we study neural networks having integer weights constrained in the ranges $[-2^{k-1}+1, 2^{k-1}-1]$, $k = 3,4,5$, which correspond to k–bit integer representation of the weights. This property reduces the amount of memory required for weight storage in digital electronic implementations. Additionally, it simplifies the digital multiplication operation, since

0-7695-0619-4/00 $10.00 © 2000 IEEE

multiplying any number with a k–bit integer requires only the following number of basic instructions: one sign change, $(k-1)(k-2)/2$ one–step left shifts and $(k-2)$ additions. Finally, if inputs are restricted to the set $\{-1,1\}$ (bipolar inputs), the neurons in the first hidden layer require only sign changes during multiplication operations, and only integer additions.

The efficient supervised training of FNNs, i.e. the incremental adaptation of the connection weights that propagate information between the neurons, is a subject of considerable ongoing research and numerous algorithms have been proposed to this end. The majority of those algorithms use the negative of the gradient of the error function, $-\nabla E(w)$, as their descent direction. The gradient $\nabla E(w)$ can be computed by the BackPropagation of the error through the layers of the network. This calculation, however, is computationally expensive and difficult to be implemented in hardware. In this contribution, we propose a new class of training algorithms that do *not* need the gradient of E and can train networks with threshold units.

Other algorithms that train neural networks with threshold units need the learning task to be static, i.e. not to change over time, in order to train the network "off–line" in a software simulation and later transfer it to the hardware [1, 4]. 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 to continue training "on–chip" is needed. The advantage of the proposed strategies is that they are capable of continuing the training process in hardware, when threshold activation functions have been used.

Formally, a typical FNN consists of L layers, where the first layer denotes the input, the last one, L, is the output, and the intermediate layers are the hidden layers. It is assumed that the (l-1) layer has N_{l-1} neurons. The neurons operate according to the following equations

$$net_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^{l-1,l} y_i^{l-1} + \theta_j^l, \quad y_j^l = f^l\left(net_j^l\right),$$

where $w_{ij}^{l-1,l}$ is the integer connection weight from the i-th neuron at the $(l-1)$ layer to the j-th neuron at the l-th layer, y_i^l is the output of the ith neuron belonging to the l-th layer, θ_j^l denotes the integer bias of the j-th neuron at the lth layer, and f is the activation function. The weights in the FNN can be expressed in vector notation. Let the weight vector have the form: $w = (w_1, w_2, \ldots, w_N)$. The weight vector, in general, defines a point in the N–dimensional real Euclidean space $\mathbb{R}^N$, where N denotes the total number of weights and biases in the network. Throughout this paper w is considered to be the *k-bit, $k = 3,4,5$, integer* vector of the weights and biases.

From the optimization point of view, supervised training of an FNN is equivalent to minimizing the corresponding error function, which is a multivariate function that depends on the weights in the network. The square error over the set of input–desired output patterns with respect to every weight, is usually taken as the function to be minimized. Specifically, the error function for an input pattern t is defined as follows:

$$e_j(t) = y_j^L(t) - d_j(t), \qquad j = 1, 2, \ldots, N_L,$$

where $d_j(t)$ is the desired response of an output neuron at the input pattern t. For a fixed, finite set of input-desired output patterns, the square error over the training set which contains T representative pairs is:

$$E(w) = \sum_{t=1}^{T} E_t(w) = \sum_{t=1}^{T} \sum_{j=1}^{N_L} e_j^2(t),$$

where $E_t(w)$ is the sum of the squares of errors associated with the pattern t. Minimization of E is attempted by using a training algorithm to update the weights. Efficient training algorithms have been proposed for trial and error based training, but it is difficult to use them when training with discrete weights [2, 3].

In this work a differential evolution approach, as explained in Section 2, has been utilized to train neural networks with k–bit, $k = 3,4,5$, integer weights and threshold activation functions, suitable for hardware implementation. A brief overview of the most used differential evolution strategies is also presented. In Section 3 techniques for training neural networks with threshold activation functions are proposed. Experiments and computer simulation results are presented in Section 4. The final section contains concluding remarks and a short discussion for future work.

2 Training neural networks with integer weights

In a recent work, Storn and Price [7] have presented a novel minimization method, called Differential Evolution (DE), which has been designed to handle non-differentiable, nonlinear 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, but requires few easily chosen control parameters. Experimental results have shown that DE has good convergence properties and outperforms other well known evolutionary algorithms.

In order to apply DE to neural network training with k-bit integer weights, we start with a specific number (*NP*) of N-dimensional integer weight vectors, as an initial weight population, and evolve them over time. *NP* is fixed throughout the training process. The weight population is initialized with random integers from the interval $[-2^{k-1}+1, 2^{k-1}-1]$, for $k = 3, 4, 5$, following a uniform probability distribution.

At each iteration, called *generation*, new weight vectors are generated by the combination of weight vectors randomly chosen from the population and the outcome is rounded to the nearest integer. Moreover, we force the new vectors to be in the range $[-2^{k-1}+1, 2^{k-1}-1]^N$. This operation is called *mutation*. The outcoming k-bit integer weight vectors are then mixed with another predetermined integer weight vector – the *target* weight vector – and this operation is called *crossover*. This operation yields the so-called *trial* weight vector, which is an integer vector in the range $[-2^{k-1}+1, 2^{k-1}-1]^N$. The trial 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*. We now briefly review the two basic DE operators used for integer weight FNN training.

1. The mutation operator

The first DE operator we consider is mutation. Specifically, for each weight vector w_g^i, $i = 1, \ldots, NP$, where g denotes the current generation, a new vector v_{g+1}^i (mutant vector) is generated according to one of the following relations:

$$v_{g+1}^i = w_g^{r_1} + \mu\left(w_g^{r_1} - w_g^{r_2}\right), \tag{1}$$

$$v_{g+1}^i = w_g^{\text{best}} + \mu\left(w_g^{r_1} - w_g^{r_2}\right), \tag{2}$$

$$v_{g+1}^i = w_g^{r_1} + \mu\left(w_g^{r_2} - w_g^{r_3}\right), \tag{3}$$

$$v_{g+1}^i = w_g^i + \mu\left(w_g^{\text{best}} - w_g^i\right) + \mu\left(w_g^{r_1} - w_g^{r_2}\right), \tag{4}$$

$$v_{g+1}^i = w_g^{\text{best}} + \mu\left(w_g^{r_1} - w_g^{r_2}\right) + \mu\left(w_g^{r_3} - w_g^{r_4}\right), \tag{5}$$

$$v_{g+1}^i = w_g^{r_1} + \mu\left(w_g^{r_2} - w_g^{r_3}\right) + \mu\left(w_g^{r_4} - w_g^{r_5}\right), \tag{6}$$

where w_g^{best} is the best member of the previous generation, $\mu > 0$ is a real parameter, called mutation constant, which controls the amplification of the difference between two weight vectors and

$$r_1, r_2, r_3, r_4, r_5 \in \{1, 2, \ldots, i-1, i+1, \ldots, NP\},$$

are random integers 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 $[-2^{k-1}+1, 2^{k-1}-1]^N$, we take:

$$v_{g+1}^i = \operatorname{sign}(v_{g+1}^i) \times \left(\left|v_{g+1}^i\right| \bmod 2^{k-1}\right).$$

Relation (1) has been introduced as crossover operator for genetic algorithms and is similar to relations (2) and (3). The remaining relations are modifications which can be obtained by the combination of (1), (2) and (3). It is clear that more such relations can be generated using the above ones as building blocks. In previous works [5, 6], we have shown that the above relations can efficiently be used to train FNNs with arbitrary as well as constrained integer weights.

2. The crossover operator

To increase further the diversity of the rounded mutant weight vector, the crossover operator is applied. Specifically, for each integer component j ($j = 1, 2, \ldots, N$) of the mutant weight vector v_{g+1}^i, we randomly choose a real number r from the interval $[0, 1]$. Then, we compare this number with ρ (crossover constant), and if $r \leq \rho$ we select, as the j-th component of the trial vector u_{g+1}^i, the corresponding component j of the mutant vector v_{g+1}^i. Otherwise, we pick the j-th component of the integer target vector w_{g+1}^i. It must be noted that the result of this operation is again a k-bit integer vector.

3 Training with Threshold Activation Functions

The proposed class of algorithm does not need the activation function to be differentiable and is suitable for training with threshold units. In the first phase of our approach, the DE algorithms are used to train a neural network "off–line", using sigmoid activation functions, such as:

$$f_1(x) = \frac{1}{1+e^{-\lambda x}}, \qquad \text{or} \qquad f_2(x) = \frac{2}{1+e^{-\lambda x}} - 1,$$

where λ is the gain parameter. This seems to be a good practice since the network is trained much faster with sigmoid functions. In the second phase we alter the gain of the sigmoid function in such a way that allows a mapping to a threshold unit network.

More specifically, when the inputs are correctly classified and the network error is small, the value of λ is increased in the sequence $(1, 10, 20, 30, 40, 50, \infty)$. Additional training might be necessary after each increase of λ. This procedure is analogous to taking the limit of the sigmoid function as the gain parameter λ goes to infinity. Finally, the trained network uses only threshold activation functions and thus the complexity of the hardware implementation is greatly reduced. If new input data are introduced, training *can* be continued "on–chip" using the DE algorithms.

4 Functionality tests

Two classical learning test problems – the eXclusive–OR (XOR) and the 3-Bit Parity problems – have been used for testing the functionality, and computer simulations have been developed to study the performance of the DE training algorithms for various values of k. We call DE1 the algorithm that uses relation (1) as mutation operator, DE2 the algorithm that uses relation (2), and so on. Table 1 summarizes the performance of the DE algorithms using different mutation rules when sigmoid activation functions are used. For all the simulation results in Table 1, we have used bipolar input and output vectors and hyperbolic tangent activation functions in both the hidden and output layer neurons. In Table 2 we exhibit the performance of the best DE algorithms, namely DE3 and DE4, to the same test problems, when the training has been performed as described in Section 3 and leads to a trained network that uses only threshold functions.

The reported parameters in the Tables for simulations that have reached solution are: *min* the minimum number of error function evaluations, *mean* the mean value of error function evaluations, *max* the maximum number of error function evaluations, *s.d.* the standard deviation of error function evaluations, and *succ.* simulations succeeded out of 1000 within the generation limit *maxgen*. When an algorithm fails to converge within the *maxgen* limit, it is considered that it fails to train the FNN and its error function evaluations are not included in the statistical analysis of the algorithms. We must note here that a key feature of the DE algorithms is that *only* error function values are needed. No gradient information is required, so there is no need of backward passes. For the test problems considered, we made no effort to tune the mutation and crossover parameters, μ and ρ respectively, in order to obtain optimal or at least nearly optimal convergence speed. Default fixed values ($\mu = 0.5$ and $\rho = 0.7$) have been used instead and we have made no effort to tune them. It is obvious that one can try to tune the μ, ρ and *NP* parameters to achieve better results, i.e. less error function evaluations and/or exhibit higher success rates. The weight population has been initialized with random integers from the interval $[-2^{k-1}+1, 2^{k-1}-1]$ and the weight population size *NP* has been chosen to be twice the dimension of the problem, i.e. *NP*$= 2N$, for all the simulations. Some experimental results have shown that a good choice for *NP* is $2N \leq NP \leq 4N$. It is obvious that the exploitation of the weight space is more effective for large values of *NP*, but sometimes more error function evaluations are required. On the other hand, small values of *NP* make the algorithm inefficient and more generations are required in order to converge to the minimum.

a) The eXclusive–OR problem. The first test problem we will consider is the eXclusive–OR (XOR) Boolean function problem, which historically has been considered as a good test of a network model and learning algorithm. A 2–2–1 FNN (six weights, three biases) has been used for these simulations and the training has been stopped when the value of the error function E, has been $E \leq 0.1$ within *maxgen*=100 generations. The population size is *NP*=18. A typical 3-bit weight vector after the end of the training process is w = (3, 3, 2, 3, 2, −2, 1, −3, −2) and the corresponding value of the error function is $E = 0.0221$. The six first components of the above vector are the weights and the remaining three are the biases.

b) The 3-Bit parity problem. The second test problem is the 3–bit parity problem, which can be considered as a generalized XOR problem but is more difficult. The task is to train a neural network to produce the sum, mod 2, of 3 binary inputs – also known as computing the "odd parity" function. We use a 3–3–1 FNN (twelve weights, four biases) in order to train the 3–Bit Parity problem. The initial population consists of 32 weight vectors. A typical 3-bit weight vector after the end of the training process is $w = (3, 3, 2, 3, -1, -1, 2, -2, -2, -3, 3, -3, 1, 0, 1, 1)$ and the corresponding value of the error function is $E = 0.0257$.

In [5] and [6] we have shown that DE3 and DE4 are the more suitable algorithms for integer weight training, when sigmoid activation functions are used. From the results shown in Table 1 it is evident that the success rates of some of these strategies are better than other well-known continuous weight training algorithm, such as BackPropagation (BP), adaptive BP or BP with momentum. In Table 2 we exhibit the results of the DE3 and DE4 strategies when threshold nodes are used. The results are promising and indicate that this new class of training algorithms can be used to train neural networks with threshold units effectively and efficiently.

In addition to training speed and efficiency, the generalization performance of the DE algorithms have also been evaluated [5, 6]. The best of them (DE3 and DE4) in their most constrained form ($k = 3, w \in [-3, 3]^N$) have been tested on the MONK's problems [8]. These difficult binary classification tasks rely on the artificial robot domain and have been used for comparing the generalization performance of learning algorithms. Our algorithms have been tested against the BackPropagation (BP), the BackPropagation with Weight Decay (BPWD), and the Cascade Correlation (CC) algorithms and the results have been satisfactory. The DE algorithms had generated FNNs, which are at least as capable as the best generated by real-weight learning algorithms. Moreover, the trained networks seem to have learned the concept embedded in the training data. When the other methods have failed to capture the concept and fit to the noise instead, the DE algorithms had classified all the test data correctly.

		The XOR problem					**The 3–bit Parity problem**				
k	Algorithm	min	mean	max	s.d.	succ.	min	mean	max	s.d.	succ.
3	DE1	54	191.9	810	89.7	63%	96	809.2	2016	313.9	88%
	DE2	90	836.3	1782	371.7	94%	704	1966.9	3072	769.2	2%
	DE3	90	300.5	1584	171.2	83%	320	1123.4	3168	461.0	98%
	DE4	54	364.5	1676	222.6	93%	160	2072.1	3168	631.9	76%
	DE5	36	1047.8	1782	422.6	63%	1344	2057.1	3072	703.9	1%
	DE6	54	931.5	1782	431.1	73%	160	1890.0	3168	907.7	3%
4	DE1	90	182.9	612	97.5	69%	96	762.7	2624	515.1	90%
	DE2	72	266.7	1188	162.4	88%	192	2056.0	3072	711.4	28%
	DE3	198	647.8	1566	246.3	99%	320	978.3	3136	555.5	96%
	DE4	126	764.1	1656	357.2	96%	288	1333.0	3104	652.6	97%
	DE5	72	316.9	1602	300.4	92%	192	1959.1	2816	981.6	10%
	DE6	108	660.0	1566	368.7	93%	1056	2153.4	3168	619.7	18%
5	DE1	54	192.9	684	124.7	75%	160	622.6	3072	522.1	91%
	DE2	72	284.9	1332	216.2	80%	576	1994.1	3168	657.6	61%
	DE3	144	583.9	1314	256.3	97%	224	896.3	2688	450.6	99%
	DE4	180	706.1	1764	343.7	98%	256	1060.2	3168	716.6	98%
	DE5	72	300.5	1584	250.2	85%	672	2112.0	3104	644.9	26%
	DE6	90	482.9	1368	264.9	93%	352	2062.5	3168	794.8	44%
		NP=18, $\mu = 0.5$, $\rho = 0.7$, $maxgen$=100					NP=32, $\mu = 0.5$, $\rho = 0.7$, $maxgen$=100				

Table 1: Simulation Results with Sigmoid Activation Functions

5 Concluding remarks and discussion

Differential evolution based algorithms for k–bit, $k = 3, 4, 5$, integer weight neural networks with threshold activation functions are studied in this contribution. This is an interesting kind of neural networks, because the amount of memory required for the storage of their weights is significantly reduced compared to networks with real weights and non–linear (sigmoid) activation functions and the digital arithmetic operations required are simplified. Moreover, this kind of networks are based on neurons whose output can be in a particular state and are important,

		The XOR problem					The 3–bit Parity problem				
k	Algorithm	min	mean	max	s.d.	succ.	min	mean	max	s.d.	succ.
3	DE3	270	442.4	756	87.1	100%	1024	2142.4	5376	760.0	100%
	DE4	270	513.2	1386	204.4	100%	1056	3459.8	6240	1137.6	100%
4	DE3	270	449.6	4104	375.5	100%	512	1348.2	3808	577.4	100%
	DE4	252	385.0	918	91.7	100%	608	1901.1	4960	903.2	100%
5	DE3	270	432.4	1062	106.6	100%	672	1423.8	7488	825.6	100%
	DE4	252	394.0	1170	111.9	100%	640	1732.8	9888	1195.8	100%
		NP=18, $\mu = 0.5$, $\rho = 0.7$, $maxgen$=100					NP=32, $\mu = 0.5$, $\rho = 0.7$, $maxgen$=100				

Table 2: Simulation Results with Threshold Activation Functions

since 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 the neural network properties [4].

Furthermore, 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. That is why all the proposed algorithms require only forward passes resulting in the value of the error function.

Obviously, the smaller the k, the less the memory required. On the other hand, as we have observed, the network training procedure can be more effective and efficient when more bits are used. Thus, for a given application a trade off between effectiveness and memory consumption has to be considered.

In this paper, customized differential evolution operators have been applied on the population of k–bit integer weight vectors, in order to evolve them over time and exploit the constrained weight space as wide as possible. The performance of these algorithms has been examined and simulation results from some classical test problems have been presented. Summarizing the simulations, we have concluded that the DE3 and DE4 algorithms are definitely the best choices for all the constrained weight spaces tested. On the other hand, even the algorithm DE1, based on the simple strategy (1), has performed very well. An interesting observation is that all the algorithms considered increase their performance when the k value is increased. The results indicate that this new class of algorithms is promising and effective, even when compared with other recently proposed algorithms that require the gradient of the error function and train the network with real weights.

References

[1] E.M. Corwin, A.M. Logar and W.J.B. Oldham, An Iterative Method for Training Multilayer Networks with Threshold Functions, *IEEE Transactions on Neural Networks*, **5**, 507–508, (1994).

[2] A.H. Khan, *Feedforward Neural Networks with Constrained Weights*, Ph.D. Thesis, Univ. of Warwick, Dept. of Engineering, (1996).

[3] A.H. Khan and E.L. Hines, Integer-weight neural nets, *Electronics Letters*, **30**, 1237–1238, (1994).

[4] G.D. Magoulas, M.N. Vrahatis, T.N. Grapsa and G.S. Androulakis, A training method for discrete multilayer neural networks, In: *Mathematics of Neural Networks, Models, Algorithms and Applications*, S.W. Ellacott, J.C. Mason and I.J. Anderson Eds., Kluwer Academic Publishers, 250–254, (1997).

[5] V.P. Plagianakos and M.N. Vrahatis, Training Neural Networks with 3–bit Integer Weights, In: *Proceedings of Genetic and Evolutionary Computation Conference (GECCO'99)*, Orlando, 910–915, (1999).

[6] V.P. Plagianakos and M.N. Vrahatis, Neural Network Training with Constrained Integer Weights, In: *Proceedings of the Congress On Evolutionary Computation (CEC'99)*, Washington D.C., 2007–2013, (1999).

[7] R. Storn and K. Price, Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous spaces, *Journal of Global Optimization*, **11**, 341–359, (1997).

[8] S.B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski, S.E. Fahlman, D. Fisher, R. Hamann, K. Kaufmann, S. Keller, I. Kononenko, J. Kreuziger, R.S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek and J. Zhang, *The MONK's Problems: A performance comparison of different learning algorithms*, Technical Report, Carnegie Mellon University, CMU-CS-91-197, (1991).