ZESZYTY NAUKOWE POLITECHNIKI BIAŁOSTOCKIEJ 2009
Informatyka – Zeszyt 4

Krzysztof Bandurski[1], Wojciech Kwedlo[1]

TRAINING NEURAL NETWORKS WITH A HYBRID DIFFERENTIAL EVOLUTION ALGORITHM

Abstract: A new hybrid method for feed forward neural network training, which combines differential evolution algorithm with a gradient-based approach is proposed. In the method, after each generation of differential evolution, a number of iterations of the conjugate gradient optimization algorithm is applied to each new solution created by the mutation and crossover operators. The experimental results show, that in comparison to the standard differential evolution the hybrid algorithm converges faster. Although this convergence is slower than that of classical gradient based methods, the hybrid algorithm has significantly better capability of avoiding local optima.

Keywords: neural networks, differential evolution, conjugate gradients, local minima

1. Introduction

Artificial Neural Networks with feedforward structure (ANNs) are widely used in regression, prediction, and classification. The problem of ANN training is formulated as the minimization of an error function in the space of connection weights. Typical ANN training methods e.g. backpropagation and conjugate gradient algorithms are based on gradient descent. The most advanced of them [4,12] are capable of fast convergence. However, like all *local search* methods, they are incapable of escaping from a local minimum of the error function. This property makes the final value of the error function very sensitive to initial conditions of training.

In recent years *global search* methods have received a lot of attention. Examples of such methods include evolutionary algorithms (EAs) [11] and simulated annealing [1]. EAs are stochastic search techniques inspired by the process of biological evolution. Unlike gradient descent methods they simultaneously process a *population* of problem solutions, which gives them the ability to escape from local optima. However this ability comes at the expense of very high computational complexity. This problem is especially important in neural network training where evaluation of each solution requires reading the whole learning set. A possible method for alleviating

[1] Faculty of Computer Science, Bialystok Technical University, Białystok

this drawback is construction of a hybrid algorithm which incorporates the gradient descent into the process of the evolutionary search.

The hybridization of EAs with gradient based methods can be achieved in a number of ways. In one approach, employed e.g. in the commercial DTREG package ([14]), an EA is used to locate a promising region of the weight space. Next, a gradient descent method is used to fine-tune the best solution (or all the solutions from the population) obtained by the EA. A version of this method, in which Levenberg-Marquardt algorithm is employed to refine a solution obtained by the DE was proposed in [16].

In an alternative approach, referred to in [13] as *Lamarckian* and applied for neural network training in [5], a gradient descent procedure is incorporated into an EA as a new search operator. This operator is applied to the population members in each EA iteration, in addition to standard operators such as mutation and crossover. Each application of the operator usually involves more than one iteration of a gradient descent method.

This paper is motivated by the work of Ilonen et al. presented in [9], in which they applied Differential Evolution (DE) proposed in [15] to train the weights of neural networks. They concluded that although the algorithm can converge to a global minimum, the computation time required to achive that goal can be intolerable. In an attempt to speed up the convergence rate of DE we followed the Lamarckian approach and combined it with the Conjugate Gradients ([4]) algorithm.

2. Artificial Neural Networks

A single node (artificial neuron) of an ANN receives a vector of input signals $\mathbf{x}$, augmented by the *bias* signal which is always equal to one. The node then computes the dot-product of this input vector and the vector of weights $\mathbf{w}$ to obtain its *activation*. The output signal y emitted by the node is a usually nonlinear function of the activation, referred to as the *transfer function* or the *activation function* and denoted here as $f(net)$. It may be a simple threshold function, though other functions, like standard sigmoid or hyperbolic tangent, are often chosen for their specific properties, e.g. differentiability. The above operations can be written down as:

$$y = f(net) = f(\mathbf{x} \cdot \mathbf{w}), \tag{1}$$

where *net* is the node's activation, $\mathbf{x}$ represents the vector of input values including bias and $\mathbf{w}$ stands for the vector of weights.

In a multilayer feed forward ANN nodes are arranged in layers. A single network of this type consists of the *input layer*, which does not perform any computations and

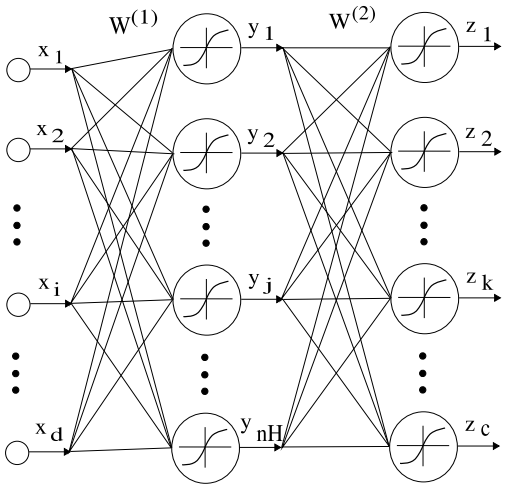

Fig. 1. A feed-forward ANN with two processing layers

only emits output signals, an arbitrary number of *processing layers*, i.e. one or more *hidden layers* and one *output layer*. Each node uses Equation (1) to compute its output signal which is then transmitted to one input of each node in the following layer. An example of an ANN with two processing layers (e.g. one hidden layer and one output layer) is presented in Fig.1. The outputs of the nodes that form the l-th processing layer in such a network can be calculated as follows:

$$\mathbf{y}_l = \mathbf{f_l}(\mathbf{x}_l \mathbf{W}_l), \tag{2}$$

where $\mathbf{x}_l = [1, y_{l-1,1} \ldots y_{l-1,n} \ldots y_{l-1,n_{l-1}}]$ is a vector consisting of n_{l-1} outputs signals emitted by the previous layer $l-1$ augmented with the bias signal equal to 1, $\mathbf{y}_l = [y_{l,1} \ldots y_{l,n} \ldots y_{l,n_l}]$ is the vector of output values yielded by the l-th layer, $\mathbf{f_l}$ is the element-by-element vector activation function used in the l-th layer and $\mathbf{W}_l$ is the matrix of weights assigned to the neurons in the l-th layer, in which a single n-th column contains all the weights of the n-th neuron, $n = 1 \ldots n_l$. When l is the number of the output layer, equation (2) yields the final output vector $\mathbf{z}$.

Before an ANN can be used for prediction it must first undergo *training*. This is usually done using a *training set* T consisting of m pairs (training samples): $T = \{(\mathbf{x}_1, \mathbf{t}_1), (\mathbf{x}_2, \mathbf{t}_2), \ldots, (\mathbf{x}_i, \mathbf{t}_i), \ldots, (\mathbf{x}_m, \mathbf{t}_m)\}$, where $\mathbf{x}_i$ is a vector of d input values and $\mathbf{t}_i$ is a vector of c desired target values corresponding to the i-th input vector. For each input vector $\mathbf{x}_i$ supplied to the input layer the neural network yields an output

vector $\mathbf{z}_i$, which is compared against the desired target vector $\mathbf{t}_i$ using a chosen *error function* E. A commonly used error function is the sum of squared errors:

$$SSE(T,W) = \sum_{i=1}^{m} \sum_{k=1}^{c} (t_{ik} - z_{ik})^2, \tag{3}$$

where W is the set of all weights in the network, t_{ik} denotes the k-th value in the i-th target vector and z_{ik} denotes the k-th output of the network produced in response to the i-th input vector. The *training error* calculated using Equation (3) is then reduced by adjusting the weights. We may thus formulate the problem of learning as the problem of minimizing the error function in the space of weights.

3. Differential Evolution

In this section the most popular *DE/rand/1/bin* differential evolution method is presented. For a more detailed description the reader is referred to [15].

Like all evolutionary algorithms, differential evolution maintains a population $U = \{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_s\}$ of s solutions to the optimization problem. Usually each solution takes the form of a D-dimensional real-valued vector, i.e. $\mathbf{u}_i \in \mathcal{R}^D$. At the beginning all members of the population are initialized randomly. The algorithm advances in *generations*. Each generation involves three consecutive phases: *reproduction* (creation of a temporary population), computing of the objective function (called the *fitness* in the EA terminology) for each temporary population member, and *selection*.

Reproduction in differential evolution creates a temporary population $V = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_s\}$ of *trial vectors*. For each solution $\mathbf{u}_i$ a corresponding trial vector $\mathbf{v}_i$ is created. Each element v_{ij} (where $j = 1 \ldots D$) of the trial vector $\mathbf{v}_i$ is generated as:

$$v_{ij} = \begin{cases} u_{aj} + F * (u_{bj} - u_{cj}) & \text{if } \texttt{rnd()} < CR \\ u_{ij} & \text{otherwise} \end{cases}. \tag{4}$$

In the above expression $F \in [0,2]$ is a user supplied parameter called the *mutation coefficient*. $a, b, c \in 1, \ldots, s$ are randomly selected in such a way, that $a \neq b \neq c \neq i$. `rnd()` denotes a random number from the uniform distribution on $[0,1)$, which is generated independently for each j. $CR \in [0,1]$ is another user supplied parameter called the *crossover factor*. The parameters F and CR influence the convergence speed and robustness of the optimization process. The choice of their optimal values is application dependent [15]. To alleviate the problem of finding optimal F and CR values we turned to [3] where Brest et al. proposed a self-adaptation

scheme for these parameters. The values of F and CR are stored with each individual $\mathbf{u}_i$. Before they are used to generate a candidate solution $\mathbf{v}_i$ they are changed as follows:

$$F_{i,G+1} = \begin{cases} F_l + \texttt{rnd()} * F_u & \text{if } \texttt{rnd()} < \tau_1 \\ F_{i,G} & \text{otherwise} \end{cases}. \tag{5}$$

$$CR_{i,G+1} = \begin{cases} \texttt{rnd()} & \text{if } \texttt{rnd()} < \tau_2 \\ CR_{i,G} & \text{otherwise} \end{cases}. \tag{6}$$

where $\tau_1 = \tau_2 = 0.1$, $F_l = 0.1$ and $F_u = 0.9$, whereas $F_{i,G}$ and $CR_{i,G}$ denote the F and CR values assigned to the i-th individual in the G-th generation. The new values $F_{i,G+1}$ and $CR_{i,G+1}$ are stored with the candidate solution which may replace the original one.

The remaining two phases of a single DE generation are the computation of fitness for all members of the trial population V and the selection. The selection in differential evolution is very simple. The fitness of each trial solution $\mathbf{v}_i$ is compared to the fitness of the corresponding original solution $\mathbf{u}_i$. The trial vector replaces the original in U if its fitness is better. Otherwise the trial vector is discarded.

To apply DE to ANN training [9] the weights of all neurons are stored in a real-valued solution vector $\mathbf{u}$. The algorithm is used to minimize the sum of squared errors (SSE) or a similar criterion function. The evaluation of this function requires iterating through all elements of the training set T and summing all the partial results (squared errors in the case of SSE) obtained for all the elements of T. In the terminology of gradient-based methods, a similar approach, in which the weights are updated after the presentation of all the elements of T to the network is called *the batch training protocol* [6].

4. Conjugate Gradient Descent

The conjugate gradient algorithm (CG) was originally proposed in [8] and applied to minimize of n-dimensional functions in [7]. In [4] it was used for neural network learning as a replacement for the classical backpropagation algorithm (BP). In classical BP the vector of weights being the current estimate of the desired minimum is updated in each step by shifting it along the gradient, but in the opposite direction, according to the following formula:

$$W^{(k+1)} = W^{(k)} + \eta[-\bigtriangledown E(W^{(k)})], \tag{7}$$

where $W^{(k)}$ is the set of all m weights in the network (including biases), η is an arbitrarily chosen learning coefficient and $\bigtriangledown E(W^{(k)})$ is the gradient of E in $W^{(k)}$. In the CG algorithm, however, the gradient is used only to determine the first error descent direction, whereas each subsequent direction is chosen to be *conjugate* with all the previous ones, i.e. one along which the gradient changes only its magnitude, and not direction (in practical applications, though, the algorithm is restarted every m iterations). Another feature that differs the CG algorithm from BP is that it employs a line search algorithm in order to find a "sufficient" approximation of a minimum of the error function along a given direction. The CG algorithm works as follows:

Let $W^{(0)}$ be the initial estimate of the minimum W^* of $E(W)$, m be the size of W, $k = 0$.

[step 0]: if $k \bmod m == 0$ then

$$D^{(k)} = - \bigtriangledown E(W^{(k)}) \tag{8}$$

else

$$D^{(k)} = -\bigtriangledown E(W^{(k)}) + \beta_k D^{(k-1)}, \tag{9}$$

where β_k is the coefficient governing the proportion of the previous search directions in the current one. It may be one of several expressions. In our work we chose the one suggested Polak-Ribiere, which, according to [6], is more robust in non-quadratic error functions:

$$\beta_k = \frac{[\bigtriangledown E(W^{(k)})]^T [\bigtriangledown E(W^k) - \bigtriangledown E(W^{(k-1)})]}{[\bigtriangledown E(W^{(k-1)})]^T \bigtriangledown E(W^{(k-1)})} \tag{10}$$

[step 1]: Perform a line search starting at $W^{(k)}$ along direction $D^{(k)}$ to determine a step length α_k that will sufficiently approximate the minimum of the single variable function given by:

$$F(\alpha) = E(W^{(k)} + \alpha D^{(k)}) \tag{11}$$

[step 2]: Update the estimate of the minimum of $E(W)$:

$$W^{(k+1)} = W^{(k)} + \alpha_k D^{(k)} \tag{12}$$

[step 3]: $k = k + 1$, go to *[step 0]*

The above procedure is repeated until a chosen termination criterion is met. It is clear that the selected line search algorithm has a significant impact on the overall performance of the algorithm presented above. We used an algorithm developed by

Charalambous, which is based on cubic interpolation. For a detailed description of this algorithm the reader is referred to [4].

5. Hybridization

As indicated in the introduction, our method of combining DE and the CG algorithm consisted in applying the latter to each candidate solution $\mathbf{v}_i$ obtained according to (4) before the computation of its fitness. The number of CG iterations is set by the user and remains constant throughout the entire experiment. By "fine-tuning" each candidate before comparing it with its predecessor we were hoping to speed up the convergence rate of DE.

6. Experiments

We tested our hybrid algorithm on 3 artificial problems and 1 real-life dataset that are described in the following paragraphs. Two versions of each of the artificial problems were tackled, each differing in the size of a single training sample. The convergence properties of our method were compared against the results yielded by self-adaptive DE with no local optimization, the Polak-Ribiere variant of Conjugate Gradient. As the computational complexity of one epoch is different in each of these algorithms, we decided to follow [9] and present our results on timescales. Each algorithm was run 30 times for each dataset. The weights of each neuron were initialized with random values from the range $\langle -\frac{1}{n_{in}}, \frac{1}{n_{in}} \rangle$, where n_{in} is the number of inputs of the neuron, including bias. Each neuron uses standard sigmoid as the activation function, whereas the error is measured by SSE. The population size in DE was set to 32. The experiments were run under Linux 2.6 on machines fitted with 64-bit Xeon 3.2GHz CPUs (2MB L2 cache) and 2GB of RAM.

6.1 Synthetic datasets

The bit parity problem. Two networks were examined: one consisting of 6 input nodes, 6 hidden nodes and 1 output node (6-6-1) and one consisting of 12 input nodes, 12 hidden nodes and 1 output node (12-12-1). The output should be set to 1 if the number of 1s in the input vector is even. The training sets consisted of 2^6 and 2^{12} samples, respectively

The encoder-decoder problem. One network consisted of 10 input nodes, 5 hidden nodes and 10 output nodes (loose encoder), whereas the other consisted of 64 input nodes, 5 hidden nodes and 64 output nodes (tight encoder). The task was to recreate the unary represention of a number presented in the input layer.

The bit counting problem. Two networks, (5-12-6) and (10-16-11) were trained to generate a unary representation of the number of bits that are set in the input vector.

6.2 Real-life dataset

The real-life dataset that we used was the *optdigits* database available in the UCI Machine Learning Repository [2]. It consists of preprocessed, normalized bitmaps of handwritten digits contributed by a total of 43 people. Each sample consists of 64 input values, representing a matrix of 8x8 where each element is an integer in the range 0..16, and 1 output value in the range of 0..9 representing the digit. The entire training set consists of 3823 samples. We modified the dataset and made each training sample contain a unary representation of the relevant digit, consisting of 10 binary values. The network that we trained consisted of 64 input nodes, 20 hidden nodes and 10 output nodes.

6.3 Results

Results are presented in tables and graphs. In the case of the synthetic datasets, each table consist of three columns. The first one contains the names of algorithms that were compared in our study: cgpr stands for Conjugate Gradients described in section 4., DE denotes the Differential Evolution algorithm presented in section 3., whereas DE-cgpr-z*xxx* corresponds to Differential Evolution augmented with the conjugate gradient algorithm as described in section 5., where *xxx* denotes the number of iterations of Conjugate Gradients that were applied to each individual before the selection phase. The two other columns contain the results obtained for each variant of the dataset (the respective network configuration and the duration of each run are given in the header of each column). These results were averaged over 30 independent runs and include: the sum of squared errors divided by 2 (SSE), the standard deviation (σ), the number of forward passes of the entire training set through the network per second (fp/s) and the number of backward passes of the error (bp/s) and finally the mean number of DE generations (gen).

Below each table there are two graphs, each presenting 6 mean error curves reflecting the progress of the tested algorithms.

The results obtained for the real-life dataset are presented in a similar manner, with the difference being that only one network configuration was used.

Table 1. Results obtained for the bit parity problems

algorithm	(6-6-1) - 5 min					(12-12-1) - 4 h				
	SSE	σ	fp/s	bp/s	gen	SSE	σ	fp/s	bp/s	gen
cgpr	0.457	1.423	11398.3	11398.3	n/a	512.00	0.00	75.7	75.7	n/a
DE	2.609	0.593	18826.2	0.0	176494.7	321.80	175.97	133.5	0.0	60063.3
DE-cgpr-z008	0.285	0.229	11585.3	11585.2	2591.3	108.20	35.27	76.0	76.0	884.2
DE-cgpr-z016	0.044	0.113	11578.3	11578.2	1324.6	56.39	20.10	76.0	76.0	439.7
DE-cgpr-z032	0.000	0.000	11595.8	11595.7	671.1	26.62	13.91	76.0	76.0	217.0
DE-cgpr-z064	0.000	0.000	11605.4	11605.3	342.1	26.56	10.63	76.0	76.0	96.5

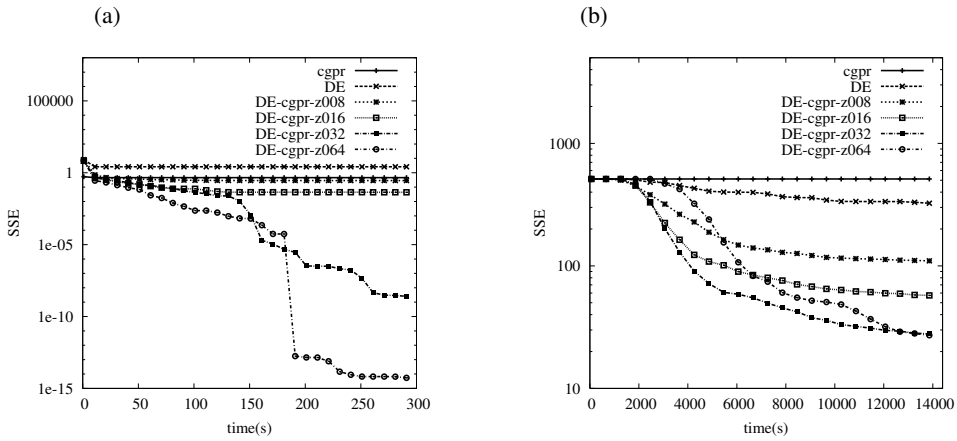

Fig. 2. SSE curves for the 6-bit (a) and 12-bit (b) parity problems

7. Conclusions

The results presented above demonstrate that the hybrid algorithm converges significantly faster than the original version of DE used in [9], provided that the number of iterations of the local optimization algorithm is sufficient. In two cases (namely 64-5-64 encoder-decoder, DE-cgpr-z008 and 10-bit counting, DE-cgpr-z008) the hybrid solution achieved a greater error value than the one achieved by the unmodified version of DE, but as the number of iterations of conjugate gradient descent increased, the hybrid version proved to be superior. This supports the findings presented by Cortez in [5], who advocated the use of the Lamarckian approach.

Our solution's capability to escape local minima is visible the most in figures 2 (a), 2 (b), 3 (a), 5, which also demonstrate its superiority in comparison to the

Table 2. Results obtained for the encoder-decoder problems

algorithm	(10-5-10) - 5 min					(64-5-64) - 4 h				
	SSE	σ	fp/s	bp/s	gen	SSE	σ	fp/s	bp/s	gen
cgpr	0.017	0.090	35535.8	35535.8	n/a	1.85	6.59	1048.7	1048.7	n/a
DE	0.075	0.169	44603.3	0.0	418154.8	15.61	0.88	1546.9	0.0	696124.0
DE-cgpr-z008	0.000	0.000	35569.4	35569.3	7141.7	17.99	1.24	1044.3	1044.3	12602.6
DE-cgpr-z016	0.000	0.000	35615.8	35615.7	3785.0	13.15	2.29	1044.9	1044.9	6463.8
DE-cgpr-z032	0.000	0.000	35662.8	35662.7	1949.1	7.72	1.60	1045.2	1045.2	3056.6
DE-cgpr-z064	0.000	0.000	35696.9	35696.8	1003.9	1.66	1.00	1045.2	1045.2	1045.5

(a) (b)

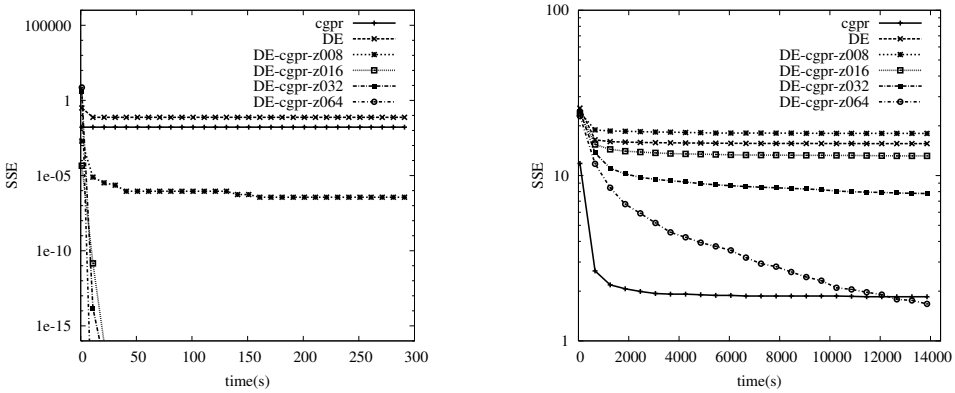

Fig. 3. SSE curves for the 10-5-10 (a) and 64-5-65 (b) encoder-decoder problems

(a) (b)

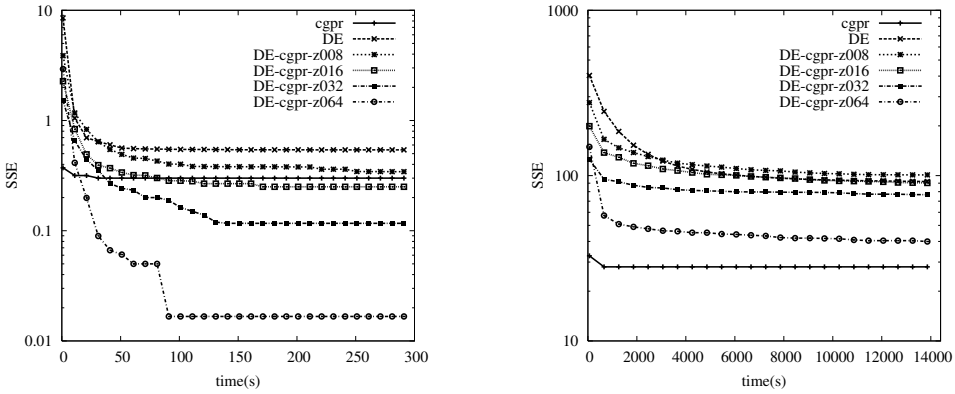

Fig. 4. SSE curves for the 5 bit (a) and 10-bit (b) counting problems

Table 3. Results obtained for the bit counting problems

algorithm	(5-12-6) - 5 min					(10-16-11) - 4 h				
	SSE	σ	fp/s	bp/s	gen	SSE	σ	fp/s	bp/s	gen
cgpr	0.300	0.355	9996.4	9996.4	n/a	28.07	34.67	152.4	152.4	n/a
DE	0.541	0.262	14971.5	0.0	140356.9	91.93	10.26	269.6	0.0	121326.9
DE-cgpr-z008	0.343	0.295	9893.8	9893.7	2255.8	100.73	12.47	153.8	153.8	1877.8
DE-cgpr-z016	0.250	0.250	9904.6	9904.5	1181.0	89.89	10.36	153.8	153.8	909.7
DE-cgpr-z032	0.117	0.211	9912.5	9912.4	579.0	76.71	6.66	153.8	153.8	434.0
DE-cgpr-z064	0.017	0.090	9917.7	9917.6	278.9	39.94	8.00	153.8	153.8	216.7

Table 4. Results obtained for the optdigits dataset

	(64-20-10) - 16 h				
algorithm	SSE	σ	fp/s	bp/s	gen
cgpr	54.75	129.9	10.112	10.112	n/a
DE	36.55	3.56	17.3	0.000	31121.0
DE-cgpr-z008	8.15	5.51	10.0	10.0	510.9
DE-cgpr-z016	4.33	1.90	10.0	10.0	261.9
DE-cgpr-z032	1.34	0.64	10.0	10.0	130.6
DE-cgpr-z064	0.35	0.40	10.0	10.0	64.0

conjugate gradient method. Figures 3 (b) and 4 (b) may suggest, however, that for some problems the hybrid algorithm performs much worse. In these cases it is worth to pay attention to the standard deviation values computed on the basis of all 30 runs performed for each dataset. They are much higher for the conjugate gradient method, which implies that under pre-defined time constraints the hybrid algorithm could require a smaller number of longer runs to yield results at an acceptable level of certainty.

Another advantage of our algorithm is its potential parallelizability. In [10] the authors proposed a method for parallelizing the original version of DE that is based on the decomposition of not only the dataset, but also the population of solutions. The algorithm presented here can be parallelized in a similar manner, with some additional mechanisms to support backpropagation. This subject is currently being investigated and will be presented in a following paper.

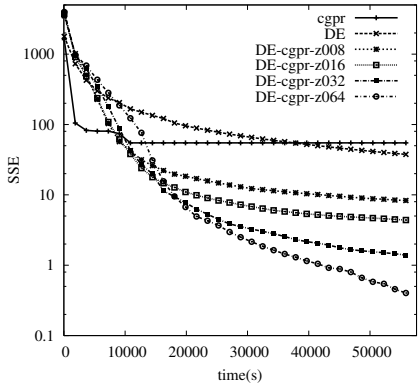

Fig. 5. SSE curves for the optdigits problem

References

[1] E. H. L. Aarst and J. Korst: Simulated Annealing and Boltzmann Machines, John Wiley, 1989.

[2] C. Blake, E. Keogh, and C. J. Merz: UCI repository of machine learning databases, University of California, Dept. of Computer Science, http://www.ics.uci.edu/∼mlearn/MLRepository.html, 1998.

[3] J. Brest, S.Greiner, B. Boskovic, M. Mernik, and V.Zumer: Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems, IEEE Transactions on Evolutionary Computation, 10(6):646–657, 2006.

[4] C. Charalambous: Conjugate gradient algorithm for efficient training of artificial neural networks, Circuits, Devices and Systems, IEE Proceedings G, 139:301–310, 1992.

[5] Paulo Cortez, Miguel Rocha, and Jos Neves: A lamarckian approach for neural network training Neural Processing Letters, 15:105–116, 2002.

[6] R. O. Duda, P. E. Hart, and D. G. Stork: Pattern Classification, John Wiley and Sons, 2001.

[7] R. Fletcher and C. M. Reeves: Function minimization by conjugate gradients, The Computer Journal, 7:149–154, 1964.

[8] M. R. Hestenes and E. Stiefel: Methods of conjugate gradients for solving linear systems, Journal of Research of the National Bureau of Standards, 49:409–436, 1952.

[9] J. Ilonen, J. K. Kamarainen, and J. Lampinen: Differential evolution training algorithm for feed-forward neural networks, Neural Processing Letters, 17:93–105, 2003.

[10] W. Kwedlo and K. Bandurski: A parallel differential evolution algorithm for neural network training, In Parallel Computing in Electrical Engineering, 2006. PARELEC 2006. International Symposium on, pages 319–324. IEEE Computer Society Press, 2006.

[11] Z. Michalewicz: Genetic Algorithms + Data Structures = Evolution Programs, Springer Verlag, 1996.

[12] M. F. Møller: A scaled conjugate gradient algorithm for fast supervised learning, Neural Networks, 6(4):525–533, 1993.

[13] Brian J. Ross: A lamarckian evolution strategy for genetic algorithms, In Lance D. Chambers, editor, Practical Handbook of Genetic Algorithms: Complex Coding Systems, volume 3, pages 1–16. CRC Press, Boca Raton, Florida, 1999.

[14] Phillip H. Sherrod: Dtreg - predictive modeling software, 2008.

[15] 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.

[16] B. Subudhi and D. Jena: Differential evolution and Levenberg Marquardt trained neural network scheme for nonlinear system identification, Neural Processing Letters, 27(3):285–296, 2008.

UCZENIE SIECI NEURONOWYCH HYBRYDOWYM ALGORYTMEM OPARTYM NA DIFFERENTIAL EVOLUTION

Streszczenie: W artykule przedstawiono nową, hybrydową metodę uczenia sieci neuronowych, łączcą w sobie algorytm Differential Evolution z podejściem gradientowym. W nowej metodzie po każdej generacji algorytmu Differential Evolution każde nowe rozwiązanie, powstałe w wyniu działania operatorów krzyżowania i mutacji, poddawane jest kilku iteracjom algorytmu optymalizacji wykorzystującego metodę gradientów sprzężonych. Wyniki eksperymentów wskazują, że nowy, hybrydowy algorytm ma szybszą zbieżność niż standardowy algorytm Differential Evolution. Mimo, iż zbieżność ta jest wolniejsza, niż w przypadku klasycznych metod gradientowych, algorytm hybrydowy potrafi znacznie lepiej unikać minimów lokalnych.

Słowa kluczowe: sieci neuronowe, differential evolution, gradienty sprzężone, minima lokalne