A Generalized Feedforward Neural Network Architecture and Its Training Using Two Stochastic Search Methods

Abdesselam Bouzerdoum[1] and Rainer Mueller[2]*

[1] School of Engineering and Mathematics
Edith Cowan University, Perth, WA, Australia
a.bouzerdoum@ieee.org
[2] University of Ulm, Ulm, Germany

Abstract. *Shunting Inhibitory Artificial Neural Networks (SIANNs)* are biologically inspired networks in which the synaptic interactions are mediated via a nonlinear mechanism called *shunting inhibition*, which allows neurons to operate as adaptive nonlinear filters. In this article, The architecture of SIANNs is extended to form a *generalized feedforward neural network* (GFNN) classifier. Two training algorithms are developed based on stochastic search methods, namely *genetic algorithms* (GAs) and a *randomized search method*. The combination of stochastic training with the GFNN is applied to four benchmark classification problems: the XOR problem, the 3-bit even parity problem, a diabetes dataset and a heart disease dataset. Experimental results prove the potential of the proposed combination of GFNN and stochastic search training methods. The GFNN can learn difficult classification tasks with few hidden neurons; it solves perfectly the 3-bit parity problem using only one neuron.

1 Introduction

Computing has historically been dominated by the concept of *programmed computing*, in which algorithms are designed and subsequently implemented using the dominant architecture at the time. An alternative paradigm is *intelligent computing*, in which the computation is distributed and massively parallel and learning replaces *a priori* program development. This new, biologically inspired, intelligent computing paradigm is called *Artificial Neural Networks* (ANNs) [1]. ANNs have been used in many applications where the conventional programmed computing has immense difficulties, such as understanding speech and handwritten text, recognizing objects, etc. However, an ANN needs to learn the task at hand before it can be operated in practice to solve the real problem. Learning is accomplished by a training algorithm. To this end, a number of different training methods have been proposed and used in practice.

* R. Mueller was a visiting student at ECU for the period July 2001 to June 2002.

Another biologically inspired computing paradigm is *genetic* and *evolutionary algorithms* [2],[3]. Evolutionary algorithms are stochastic search methods that mimic the metaphor of natural biological evolution. They operate on population of potential solutions applying the principle of survival of the fittest. The combination of these two biologically inspired computing paradigms is a powerful instrument for solving problems in pattern recognition, signal and image processing, machine vision, control, etc.. The aim in this article is to combine a *Generalized Feedforward Neural Network* (GFNN) architecture with genetic algorithms to design a new class of artificial neural networks that has the potential to learn complex problems more efficiently.

In the next section, the *generalized shunting neuron* and the GFNN architecture are introduced. Two training methods for the GFNN architecture are presented in section 3. First the randomized search method is presented in Subsection 3.1, then the GA technique in Subsection 3.2. The developed training algorithms are tested with some common benchmark problems in Section 4, followed by concluding remarks and future work in Section 5.

2 The Generalized Feedforward Neural Network Architecture

In [4] Bouzerdoum introduced the class of *shunting inhibitory artificial neural networks* (SIANNs) and used them for classification and function approximation. In this section, we extend SIANNs to form a generalized feedforward neural network architecture. But before describing the generalized architecture, we first introduce the elementary building block of the architecture, namely the *generalized shunting inhibitory neuron.*

2.1 Generalized Shunting Inhibitory Neuron

The output of a generalized shunting inhibitory neuron is given by

$$x_j = \frac{f(\sum_i w_{ji} I_i + w_{j0})}{a_j + g(\sum_i c_{ji} I_i + c_{j0})} = \frac{f(\mathbf{w}_j \cdot \mathbf{I} + w_{j0})}{a_j + g(\mathbf{c}_j \cdot \mathbf{I} + c_{j0})} \tag{1}$$

where x_j is the activity (output) of neuron j; I_i is the ith input; c_{ji} is the "shunting inhibitory" connection weight from input i to neuron j; w_{ji} is the connection weight from input i to neuron j; w_{j0} and c_{j0} are bias constants; a_j is a constant preventing the division by zero, by keeping the denominator always positive; f and g are activation functions. The name shunting inhbition comes from the fact that a high term in the denominator tends to supress (or inhibit in a shunting fashion) the activity caused by the term in the numerator of (1).

2.2 The Network Architecture

The architecture of the generalized feedforward neural network is similar to that of a *Multilayer Perceptron Network* [1], and is shown in Fig. 1. The network

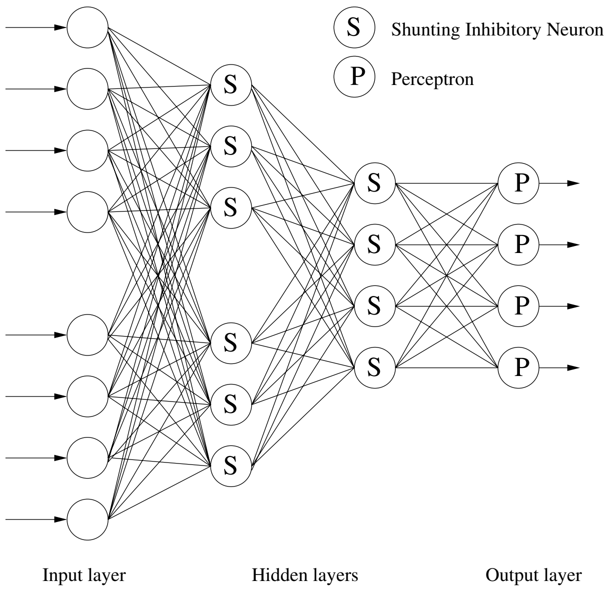


Fig. 1. Generalized Feedforward Neural Network architecture (GFNN).

consists of many layers, each of which has a number of neurons. The input layer only acts as a receptor that receives inputs from the environment and broadcasts them to the next layer; therefore, no processing is done in the input layer. The processing in the network is done by the hidden and output layers. Neurons in each layer receive inputs from the previous layer, process them and then pass their outputs to the next layer. Hidden layers are so named because they have no direct connection with the environment. In the GFNN architecture, the hidden layers consist of only generalized shunting inhibitory neurons. The role of the shunting inhibitory layers is to perform a nonlinear transformation on the input data so that the results can easily be combined by the output neurons to form the correct decision. The output layer, which may be a linear or sigmoidal type (i.e., perceptron), is different from the hidden layers; each output neuron basically calculates the weighted sum of its inputs followed by an appropriate activation function. The response, y, of an output neuron is given by

$$y = h(\mathbf{w}_o \cdot \boldsymbol{x} + b) \tag{2}$$

where $\boldsymbol{x}$ is the input vector $\mathbf{w}_o$ is the weight vector, b is the bias constant, and h is the activation function, which may be a linear or a sigmoid function.

3 Training Methods

An artificial neural network needs to be trained instead of being *a priori* programmed. Supervised learning is a form of learning in which the target values are included part of the training data. During the training phase, the set of training data is repeatedly applied to the network and the weights of the network are adjusted until the difference between the target values and the network output values is within the desired tolerance.

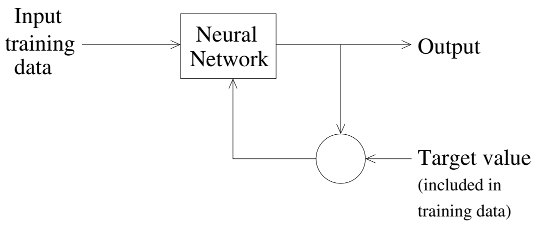


Fig. 2. Supervised learning: the weights are adjusted until the target values are reached.

In this section, two different methods for training the GFNN are described: the Random Optimization Method (ROM) and the GA based method. Since GAs are known for being able to find good solutions for many complex optimization problems, this training method is of particular interest to us.

3.1 Random Optimization Method (ROM)

The ROM is employed because it is a simple method to implement and intuitively appealing. It is used to test the network structure before the GA is applied, and serves as a benchmark for comparing the GA based training method. The ROM searches the weight space by generating randomized vectors in the weight space and testing them. The basic ROM procedure is as follows [1]:

1. Randomly choose a weight vector $\boldsymbol{W}$ and a small vector $\boldsymbol{R}$.
2. If the output of the net $\boldsymbol{Y}(\boldsymbol{W}+\boldsymbol{R})$ is better than $\boldsymbol{Y}(\boldsymbol{W})$ then $\boldsymbol{W}=\boldsymbol{W}+\boldsymbol{R}$.
3. Check for termination criteria, end the algorithm when one of the termination criteria is achieved.
4. Randomly choose a new $\boldsymbol{R}$ and go to step (2)

There are some obvious extensions to the above algorithm which we have implemented. The first one implements reverse side checking. This means instead of checking only $\boldsymbol{W}+\boldsymbol{R}$ we check $\boldsymbol{W}-\boldsymbol{R}$ as well. Furthermore, an orthogonal vector $\boldsymbol{R}^*$ is also checked in both directions. That alone wouldn't improve the algorithm much, but there is another extension. If there is an improvement in any of the four previous directions, simply extend the search in the same direction, instead of just generating another value of $\boldsymbol{R}$. The idea is that if $\boldsymbol{W}+\boldsymbol{R}$ gives an improved output $\boldsymbol{Y}$, then another scaled step $k \cdot \boldsymbol{R}$ in the same direction might be in a "downhill" direction, and hence a successful direction. All these extensions have been implemented to train the GFNN.

3.2 Genetic Algorithms (GAs)

The GAs are used as a training method because they are known for their ability to perform well on complex optimization problems. Furthermore, they are less likely to get trapped in local minima, a problem suffered by traditional gradient based training algorithms. GAs are stochastic search methods that mimic the metaphor of natural biological evolution. They operate on a population of potential solutions applying the principle of survival of the fittest to produce an improved approximation to a solution. At each generation, a new set of approximations is created by the process of selecting individuals according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural evolution. This process leads to the evolution of populations of individuals that are better suited to their environment than the individuals they were created from, just as in natural adaptation.

GAs model natural evolutionary processes, such as selection, recombination, mutation, migration, locality and neighborhood. They work on populations of individuals instead of single solutions. Furthermore, simple GAs can be extended to multipopulation GAs. In multipopulation GAs several subpopulations are introduced, which evolve independently over few generations before one or more individuals are exchanged between the subpopulations. Figure 3 shows the structure of an extended multipopulation genetic algorithm.

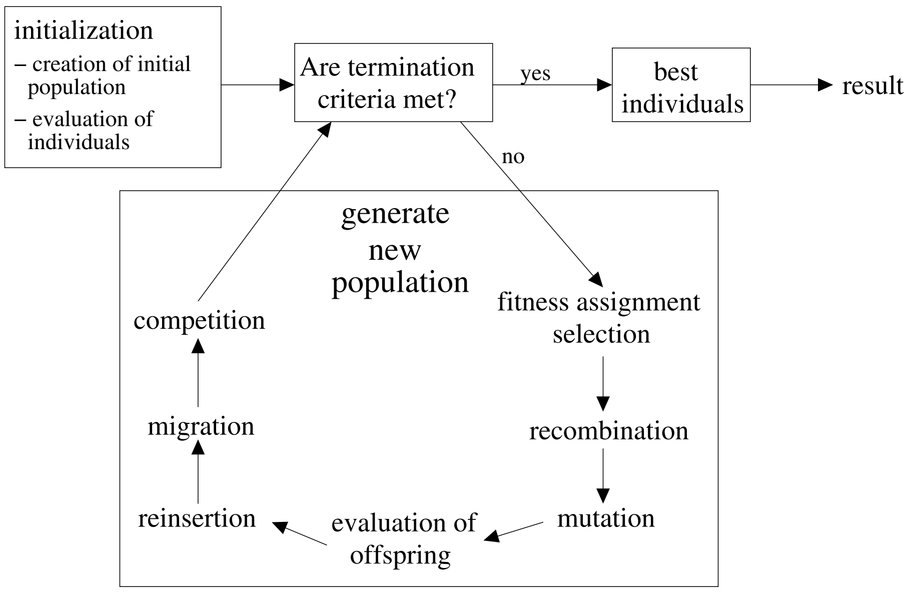


Fig. 3. Structure of an extended multipopulation genetic algorithm (adapted from [5]).

The genetic operators that can be applied to evolve the population depend on the variable representation in the GA: binary, integer, floating point (real), or symbolic. In this research, we employed the real variable representation because it is the most natural representation for weights and biases of neural networks. Furthermore, it has been shown that the real-valued GA is more efficient than the binary GA [3]. Some of the most common GA operators are described below.

Selection. Selection determines the individuals which are chosen for mating (recombination) and how many offsprings each selected individual produces. Each individual in the selection pool receives a reproduction probability depending on its own objective value and the objective values of all other individuals in the population. There are two fitness-based assignment methods: proportional fitness assignment and rank-based fitness assignment. The proportional fitness assignment assigns a fitness value proportional to the objective value, whereas the fitness value in a rank-based assignment depends only on the rank of the individual in a list sorted according to the objective values.

Roulette-wheel selection, also called "stochastic sampling with replacement" [6], maps the individuals to contiguous segments of a line, such that each individual's segment is equal in size to its fitness [5]. The individual whose segment spans a generated random number, is selected.

In *stochastic universal sampling*, the individuals are mapped to N contiguous segments of a line (N being the number of individuals), each segment having a length proportional to its fitness. Then N equally spaced Pointers are placed above the line and the position of the first pointer is given by a randomly generated number in the range $[0, 1/N]$. Every pointer indicates a selected individual.

In *local selection* every individual interacts only with individual residing in its local neighborhood [5].

In *truncation selection* individuals are sorted according to their fitness and only the best individuals are selected as parents.

The *tournament selection* chooses randomly a number of individuals from the population and the best individual from this group is selected as parent. The process is repeated until enough mating individuals are found.

Recombination. The process of recombination produces new individuals by combining the information contained in the parents. There are different recombination methods depending on the variable representation. Discrete recombination can be used with all representations. In addition, there are two specific methods for real valued recombination, the *intermediate recombination* and the *line recombination*. In intermediate recombination the variables of the offspring are chosen somewhere around and between the variable values of the parents. Line recombination, on the other hand, generates the offspring on a line defined by the variable values of the parents.

Mutation. After recombination, every offspring undergoes a mutation, like in nature. Small perturbations mutate the offspring variables with low probability. Mutation of real variables means that randomly generated values are added to the offspring variables with low probability. Thus, the probability of mutating a variable (mutation rate) and the size of change for each mutated variable (mutation step) must be defined. In our simulations, the mutaion rate is inversely proportional to the number of variables; the more variables an individual has, the smaller is the mutation rate.

Reinsertion. After an offspring is produced it must be inserted into the population. There are two different situations. First, the size of the offspring population produced is less than the size of the original population. In this case, the whole offspring population has to be inserted to maintain the size of the original population. Second more offsprings are generated than there are individuals in the original population. In this case, the reinsertion scheme determines which individuals should be reinserted into the new population and which individuals should be replaced by the offsprings. There are different schemes for reinsertion. *Pure reinsertion* produces as many offsprings as parents and replaces all parents by the offspings. *Uniform reinsertion* produces fewer offsprings than parents and replaces parents uniformly at random. *Elitist reinsertion* produces fewer offsprings than parents and replaces the worst parents. *Fitness based reinsertion* produces more offsprings than needed and reinserts only the best offsprings.

After reinsertion, one needs to verify if a termination criteria is met. If a criteria is met, then the cycle can be stopped; otherwise, the cycle will be repeated until a termination criteria is met. The GA parameters used in the simulations are presented in Table 1 below.

Table 1. Evolutionary algorithm parameters used in the simulations.

subpopulations	individuals	50 30 20 20 10
variable format	real values	
selection	function	selsus (stochastic universal sampling)
	pressure	1.7
	gen. gap	0.9
reinsertion	rate	1
recombination	name	discrete and line recombination
	rate	1
mutation	name	mutreal (real-valued mutation)
	rate	0.00826
	range	0.1 0.03 0.01 0.003 0.001
	precision	12
regional model		
migration	rate	0.1
competition	rate	0.1

The objective function to be minimized here is the mean squared error.

$$MSE = \frac{1}{N_p} \sum_{j=1}^{N_p} (y_j - d_j)^2 \tag{3}$$

where y_j is the output of the GFNN, d_j the desired output for input pattern $\boldsymbol{x}_j$, and N_p is the number of training patterns.

4 Experimental Results

Experiments were conducted to assess the ability of the proposed NN architecture to learn some difficult classification tasks. Four benchmark problems were selected to test the network architecture: two Boolean functions, the Exclusive-OR (XOR) and the 3-bit parity, and two medical diagnosis problems, the heart disease and diabetes. The heart disease and diabetes data sets were obtained from UCI Machine Learning Repository [7].

4.1 The XOR and 3-Bit Parity Problems

A two-layer network architecture consisting of two inputs, one or two hidden units, and an output unit is trained with XOR problem. For every network configuration, ten training runs, with different intializations, were performed using both the GA- and the ROM-based training algorithms. If during the training a network reaches an error of zero, training is halted. Table 2 summarizes the results: the first column indicates the f/g combination of activation functions (see Eq. (1)), along with the training algorithm. In all the simulations f was *hyperbolic tangent sigmoid* activation function, `tansig`, and g was either the *exponential* function, `exp`, or the *logarithmic sigmoid* activation function, `logsig`. The GA uses a search space ranging from -128 to 128, and hence is labeled GA128. The second column shows the number of training runs that achieved zero error. The "Best case error" column shows the lowest test error of trained networks. Note that even when an error of zero is not reached during training, the network can still learn the desired function after thresholding its output.

Table 2. Training with the XOR problem.

	Runs w. E=0	Aver. generation to reach E=0	Aver. time to reach E=0	Best case error	Mean error	Std
No. of neurons: 1 (hidden layer), 9 weights						
tansig/logsig GA128	1	620	15.89	0.00	25.50	790
tansig/logsig ROM	4	4423	4.56	0.00	15.00	1290
tansig/exp GA128	10	21	0.51	0.00	0.00	000
tansig/exp ROM	6	488	0.47	0.00	10.00	1290
No. of neurons: 2 (hidden layer), 17 weights						
tansig/logsig GA128	8	68	2.02	0.00	5.00	1054
tansig/logsig ROM	10	393	0.52	0.00	0.00	000
tansig/exp GA128	10	13	0.37	0.00	0.00	000
tansig/exp ROM	10	845	1.05	0.00	0.00	000

The best results were obtained using two neurons in the hidden layer with the exponential activation function, `exp`, in the denominator. Note that both training algorithms, GA and ROM, reached an error of zero at least once during training. The GA was slightly faster with 0.37 minutes average time to reach an error of zero than the ROM, which needed 1.05 minutes. Figure 4 displays the percentage mean error vs. training time for the best combination of activation functions (tansig/exp). More importantly, however, is the fact that even with one hidden neuron and tansig/exp combination, ten out of ten runs reached an error of zero, with the GA as training algorithm. However, the time to reach an error of 0 was 0.51 minutes slightly longer than the time of the two neuron network. Also, we can observe that both the ROM and GA perform well in the sense of reaching runs with error zero. Furthermore, all trained were able to classify the XOR problem correctly.

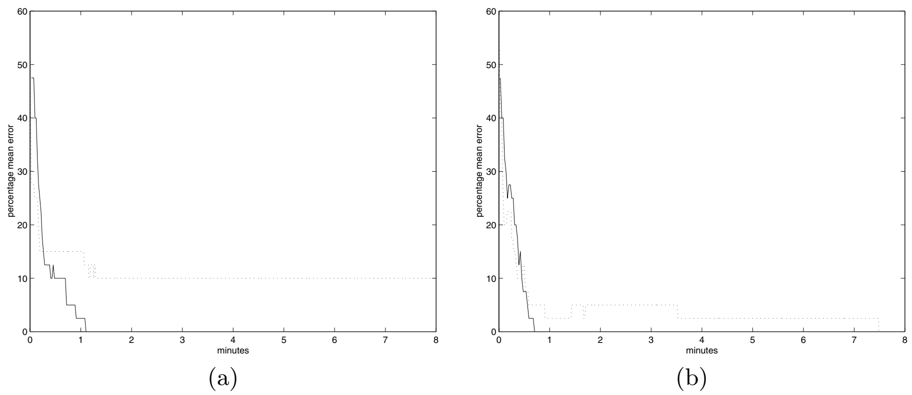


Fig. 4. Percentage mean error over time with tansig/exp as activation functions: (a) 1 hidden unit, (b) 2 hidden units. The dotted line is the result of the ROM and the solid line is the result of the GA.

For the 3-bit partiy problem, the network architecture consists of three inputs, one hidden layer and one ouptput unit of the perceptron type; the hidden layer comprises one, two or three shunting neurons. The same experiments as with the XOR problem were conducted with 3-bit parity; that is, ten runs for each architecture are performed with tansig/logsig or tansig/exp activation functions. Table 3 presents the result of the ten runs. None of the networks with `logsig` activation function in the denominator reach an error of zero during training. However, using the exponential activation function in the denominator, some networks with one hidden unit reach zero error during training and most networks, even those that do not reach zero error during training, learn to classify the even-parity correctly.

4.2 Diabetes Problem

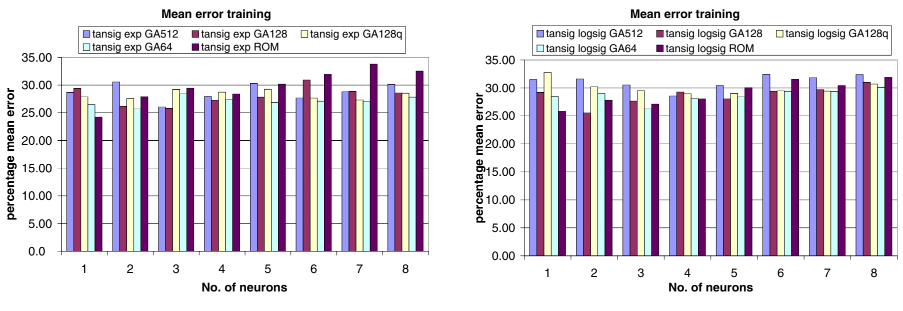
The diabetes dataset has 768 samples with 8 input parameters and two output classes: presence (1) or absence (0) of diabetes. The dataset was partitioned into two sets: 50% of the data points were used for training and the other 50% for testing. The network architecture consisted of 8 input units, one hidden layer

Table 3. Training with the 3-bit even parity.

	Runs w. E=0	Aver. generation to reach E=0	Aver. time to reach E=0	Best case error	Mean error	Std
No. of neurons: 1 (hidden layer), 11 weights						
tansig/logsig GA128	0	NaN	NaN	12.50	20.00	6.45
tansig/logsig ROM	0	NaN	NaN	12.50	28.75	11.86
tansig/exp GA128	2	629	7.13	0.00	17.50	12.08
tansig/exp ROM	0	2720	1.36	0.00	20.00	10.54
No. of neurons: 2 (hidden layer), 21 weights						
tansig/logsig GA128	0	NaN	NaN	12.50	22.50	5.27
tansig/logsig ROM	0	7320	4.99	0.00	18.75	8.84
tansig/exp GA128	6	243	3.33	0.00	6.25	8.84
tansig/exp ROM	4	11180	6.56	0.00	7.50	6.45
No. of neurons: 3 (hidden layer), 31 weights						
tansig/logsig GA128	3	753	12.58	0.00	12.50	10.21
tansig/logsig ROM	3	4770	6.59	0.00	13.75	10.94
tansig/exp GA128	8	57	0.92	0.00	2.50	5.27
tansig/exp ROM	7	9083	12.04	0.00	3.75	6.04

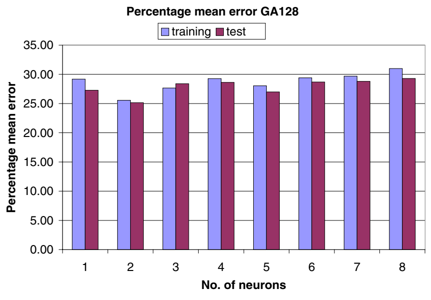
of shunting neurons, and one output unit. The number of hidden units varied from one to eight. The size of the search space is also varied: $[-64, 64]$ (GA64), $[-128, 128]$ (GA128), $[-512, 512]$ (GA512). Again ten training runs for each architecture and each algorithm, GA and ROM, were performed. The network GA128 was also trained on a reduced data set (a quarter of the total data); this network is denoted GA128q. After training is completed, the generalization ability of each network is tested by evaluating its performance on the test set.

Figure 5 presents the percentage mean error of the training dataset. It can be observed that the tansig/exp activation function combination performs slightly better than the tansig/logsig. The ROM gets worse with increasing number of neurons, what we expected. The reason is that the one hidden-neuron configuration has 21 weights/biases whereas the 8 hidden-neuron configuration has 161

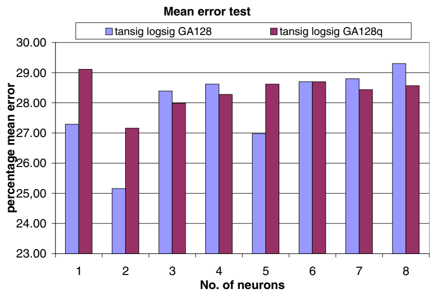


(a) (b)

Fig. 5. Percentage mean error (train dataset) of the 10 runs: (a) tansig/exp, (b) tansig/logsig configuration.



(a)



(b)

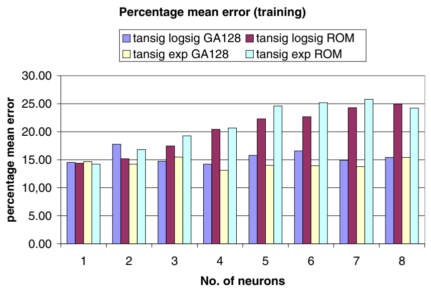
Fig. 6. (a) Percentage mean error of the GA128 on the training and test sets. (b) Generalization performance of GA128 and GA128q on the test set.

weights/biases. With increasing number of weights/biases the dimension of the search space increases, which leads to worse performance by the ROM.

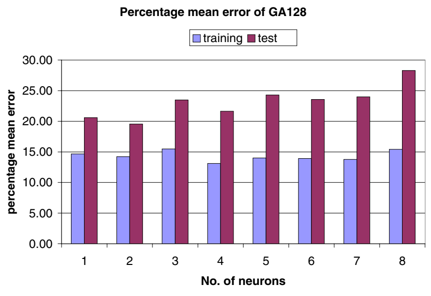
In Fig. 6 the percentage mean error of the training dataset is compared with the percentage mean error of the test set; both are almost equal for all the different number of neurons. This shows that overfitting is not a serious problem.

4.3 Heart Disease Problem

The experimental procedure was the same as for the diabetes diagnoses problem, except that the data set has only 270 samples with 13 input parameters. This increases the number of parameters of the network and slows down the training process. To avoid being bogged down by the training process, only GA128 was trained on the Heart dataset. Figure 7(a) presents the mean error rates on the training set. Not surprising, the mean error rate of the ROM increases with increasing number of neurons. Figure 7(b) compares the performances of the GA on the training and test sets. The results of the heart disease problem are



(a)



(b)

Fig. 7. Percentage mean error: (a) training set, (b) training set compared totest set.

similar to those of the diabetes diagnoses problem, except the errors are much lower; it is well known that the Diabetes problem is harder to learn that the Heart Disease problem.

5 Conclusions and Future Work

In this article we presented a new class of neural networks and two training methods: the ROM and the GA algorithms. As expected, the ROM works well for a small number of weights/biases but becomes worse as the number of parameters increases. The experimental results show that the presented network architecture, with the proposed learning schemes, can be a powerful tool for solving problems in prediction, forecasting and classification. It was shown that the proposed architecture can learn a Boolean function perfectly with a small number of hidden units. The tests on the two medical diagnosis problems, diabetes and heart disease, proved that the proposed architecture can learn complex tasks with good generalization ability and hardly any overfitting.

Some further work needs to be considered to improve the learning performance of the proposed architecture. Firstly, a suitable termination criteria must be found to stop the algorithm, which could be the classification error on a validation set. Secondly, the settings of the GA should be optimized. In this project only different sizes of the search space were used. To get better results other settings, e.g. size of population, mutation methods, should be optimized. Finally a combination of the GA and, e.g., gradient descent method can improve the results further. GAs are known for their global search and gradient methods for their local search; by combining the two, we should expect better results.

References

1. Schalkoff, R. J.: *Artificial Neural Networks.* McGraw-Hill 1997.
2. Goldberg, D. E.: *Genetic Algorithms in search, Optimization and Machine Learning.* Addison-Wesley, 1989.
3. Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Programs,* (2nd edition). Berlin, Heidelberg, New York: Springer-Verlag, 1994.
4. Bouzerdoum, A.: “Classification and function approximation using feed-forward shunting inhibitory artificial neural networks,” *Proc. IEEE/INNS Int. Joint Conf. Neural networks* (IJCNN-2000), Vol. VI, pp. 613–618, 24–27 July 2000, Como, Italy.
5. Pohlheim, H.: *Genetic and Evolutionary Algorithms: Principles, Methods and Algorithms*, 1999. http://www.geatbx.com.
6. Baker, J. E.: “Reducing bias and inefficiency in the selection algorithms,” *Proc. Second Int. Conf. on Genetic Algorithms*, pp. 14–21, 1987.
7. Blake, C. L., Merz, C. J.: “UCI Repository of Machine Learning Databases,” Dept. Information and Computer Science, University of California, Irvine, 1998.
8. Rooij, Jain and Johnson: *Neural Network Training using Genetic Algorithms.* World Scientific, 1996.