Neural Network Training and Simulation Using a Multidimensional Optimization System

A. Likas, D.A. Karras and I.E. Lagaris
Department of Computer Science
University of Ioannina
P.O. Box 1186 - GR 45110 Ioannina, Greece

Abstract

A new approach is presented to neural network simulation and training that is based on the use of general purpose optimization software. This approach requires that the training problem should be formulated as the minimization of a cost function of the network weights. This cost function is a user written code called by the optimization system, which in turn provides the user with a variety of minimization procedures that can be combined via user programmable minimization strategies. Experimental results concerning several learning paradigms indicate that the approach is very convenient and effective and leads to the discovery of efficient training strategies.

1 Introduction

The increased interest in neural network research has led to the development of many software simulators that provide the experimentation means for training and testing the variety of the existing models. These simulators can be classified into the following categories.

1. *Network specific simulators.* They are specific to a particular neural network type, most often multilayer perceptrons (MLPs), allowing the user a limited choice of network's parameters. Their use is limited to the application areas of the specific model implemented by them.

2. *Template simulators.* A template simulator can be mainly characterized by its ease of use, basically through a well constructed graphical user interface. One can select a network type from a pool of existing ones and after easily specifying its architecture (number of layers, units, activation function type etc.) can select its learning rule

through a multitude of different training algorithms. The only flexibility allowed to the user is to easily modify architectural and training parameters. However, new training rules cannot be incorporated. Defined nets can usually be called from high level languages like C, allowing nets to be incorporated in layered applications. SNNS (although it lacks the feature of C code generation) is a representative example of a powerful template simulator, which probably affords the largest number of models and rules.

3. *Non-template simulators.* These more advanced simulators allow the user to define new ANN models or implement major modification to existing models. For examples new activation functions and learning rules can be specified. A feature of nearly all these simulators is the provision of a well designed graphical interface, which allows unexpected behaviour of a new model to be tracked and understood. The major characteristic of non-template simulators is the lack of a well developed and documented library to provide the means to build new models not from scratch.

The ojective of this paper is to demonstrate a new approach to neural network simulation and training, at least for the models whose training is based on optimization methods. Within this methodology all the architectural characteristics are integrated in the ANN cost function, which should *programmed* by the user. The user may also provide the derivatives of this function. Otherwise, numerical methods for derivative approximation are involved. The variety of optimization techniques incorporated in the system may be invoked and the many subtle implementation details of optimization methods are hidden from the users. This fact constitutes the main advantage of the proposed approach which is based on the use of the MERLIN package for multidimensional minimization [1, 2].

It is well known that several types of neural network training problems can be formulated as optimization problems that aim at minizing a suitably formulated function. This is true for every type of learning, ie supervised, reinforcement and unsupervised learning. MERLIN has given us the opportunity to easily test several minimization algorithms, assess their effectiveness and discover appropriate combinations of methods that exhibit superior minimization performance. More specifically, we have devised effective strategies for supervised training of multilayer perceptrons and for delayed reinforcement problems and, in addition, we also provide preliminary results on the unsupervised training of clustering networks.

In the next section the basic characteristics of the MERLIN optimization environment are described. Section 3 describes a training strategy for multilayer perceptrons while section 4 presents an appropriate strategy for training reinforcement neurocontrollers. Section 5 shows how MERLIN can be used to train clustering networks and describes a set of candidate unsupervised learning problems that may be examined using the proposed approach. Finally section 6 contains conclusions and directions for future work.

2 Merlin Description

MERLIN [1, 2] is a software package for multidimensional minimization that handles the following category of problems:
Find a local minimum of the function:

$$f(\mathbf{x}), \quad \mathbf{x} \in R^N, \quad \mathbf{x} = [x_1, x_2, \cdots, x_N]^T$$

under the conditions:

$$x_i \in [a_i, b_i] \ for \ i = 1, 2, \cdots, N$$

Special merit has been taken for problems where the objective function can be written as a sum of squares ie:

$$f(\mathbf{x}) = \sum_{i=1}^{M} f_i^2(\mathbf{x})$$

MERLIN supports various minimization algorithms that can be divided into two categories:

- A) Methods that use only function values, and
- B) Methods that use gradient information as well.

From category A, the SIMPLEX method [4], and a pattern search method (termed ROLL [1]) are implemented. From category B, conjugate gradient methods along with Quasi–Newton methods are the chosen ones. Specifically the Fletcher–Reeves [5], Polak–Ribiere [6] and the Generalized Polak–Ribiere [7] are implemented from the conjugate gradient family and the DFP [8] and several versions of the BFGS [9] method from the Quasi–Newton (variable metric) family. The special Sum–of–Squares form, is treated in addition with a Levenberg–Marquardt method [10].

Generaly speaking, methods that use derivatives, are more efficient. However some problems correspond to objective functions that by nature are non-differentiable and hence these methods will not work. Such a case is described in section 4, where the pole-balancing problem is solved by a training strategy based on the SIMPLEX method.

An interesting and useful feature of the package is that it can approximate the derivatives of the objective function numerically. In fact the gradient can be approximated either via a forward difference formula, or more accurately via a central difference two–point formula. An additional numerical estimation option of high accuracy but computationaly expensive uses a six (or more) points in a symmetric finite difference formula to approximate the gradient. The user may also provide his own code for the calculation of the gradient. Moreover, since the calculation of the gradient is often quite complicated, it is very common for the user written code to be erroneous. To help the development of correct code for the gradient, there is a built–in facility that allows the user to compare the results of his code against the finite difference estimates.

The philosophy followed for Merlin construction was similar to that usualy adopted for building operating systems shells. The system idles expecting an input command. Once this is entered (by the user), it is identified and if it is a valid command it is executed. Upon its completion the system idles again and so on so forth. This structure is very important since it permits the programmability of minimization strategies. In fact, a language has been defined [3] to control the MERLIN system and the associated compiler has been implemented. The MERLIN Control Language (MCL) supports all the MERLIN commands plus commands to control loops, conditions and branching. For simple problems one does not need to use MCL, however in problems where an algorithmic strategy is needed, MCL programming is instrumental. Via MCL one can code very easily global optimization procedures (for instance stochastic ones) for problems where local minima do not represent acceptable solutions. In addition using MCL, one can handle non–linear constraints by employing penalty and barrier methods. Other facilities offered that may be useful are one-dimensional plots, confidence intervals for the parameters (for the case of the Sum of squares form, where the maximum likehood notion is meaningful), fixing one or more parameters to a certain value, freeing previously fixed parameters, imposing box constraints on the parameters, etc.

3 Training of MLPs

Minimizing the MLP error function in realistic problems is a difficult task since the many layers, the multitude of training patterns and the variety of categories cast a very complex landscape with wide plateaus and narrow valleys [12]. There is no single algorithm that can be used as a panacea to solve such optimization problems. Algorithms that use gradient information perform well only at regions of the parameter space where the function is smooth, while algorithms using only function values may be effective at regions where the derivatives are not well defined. From this point of view, the main weak point of the existing MLP training procedures is the use of a single optimization algorithm. Through the use of MERLIN we were able to discover a novel *multi-algorithm* optimization procedure governed by a strategy that exploits the virtues and strengths of the participating algorithms. This renders the procedure efficient and robust and although it is an established approach in the field of optimization, it has never been employed in MLP training before. This new methodology has been implemented within the novel simulating approach defined in the previous sections.

The suggested procedure uses three different algorithms, specifically the quasi-Newton BFGS, the Polak-Ribiere (PR) conjugate gradient algorithm and a pattern search method (ROLL) that uses only function values. Pattern search methods have not been used in MLP training so far. Since the above algorithm employs no derivatives it is expected to be effective at the regions with plateaus of the weight space where the BFGS and PR techniques that use gradient information fail to perform. Since the ROLL method is not widely known we provide a brief description for it. Let $E(W_1, W_2, \ldots, W_n)$ be the error function in MLPs with W_j corresponding to the weight variables. Let, also, $W^c = (W_1^c, W_2^c, \ldots, W_n^c)$ be the current point in the optimization process of E and $E_c = E(W^c)$. Finally, let S_i be a step associated with each free variable W_i.

1. Pick a trial point: $W_j^t = W_j^c$ for all $j \neq i$ and $W_i^t = W_i^c + S_i$
2. Calculate $E_+ = E(W^t)$.
3. if $E_+ < E_c$ set $W^c = W^t, E_c = E_+$ and $S_i = aS_i$. Then, go to step 8.
4. if $E_+ >= E_c$ pick another trial point as : $W_j^t = W_j^c$ for all $j \neq i$ and $W_i^t = W_i^c - S_i$
5. Calculate $E_- = E(W^t)$.
6. if $E_- < E_c$ set $W^c = W^t, E_c = E_-$ and $S_i = -aS_i$. Then, go to step 8.
7. if $E_- >= E_c$ calculate an appropriate step by: $S_i = -\frac{1}{2}\frac{(E_+ - E_-)}{(E_+ + E_- - 2E_c)}S_i$.
8. Proceed with step 1 for the next value of i.

In the above, $a > 1$, is a user set factor (in our experiments $a = 3.0$). If after looping over all variables there is no progress, a line search is performed in the direction $S = (S_1, S_2, \ldots, S_n)$. The above procedure is repeated until a preset number of calls to the

objective function is reached.
In what follows we give a rather detailed account of the proposed *MultiALgorithm Optimization (MALO)* strategy that was coded in MCL.

Initialization: Pick at random an initial set of weights all in $[-1, 1]$.

Set the maximum allowed number of calls to the error function.

Set the target value (a satisfactory value for the error function) E_0.

Set the value for the rate of progress r. (We used $r = \frac{1}{100}$).

Step (1): Test the number of calls to decide whether to stop or not.

Step (2): Determine and fix the non-influential weights. These weights w have the property $|\frac{\partial E}{\partial w}| < \epsilon$, where $\epsilon > 0$ a small preset value, i.e the error function is not very sensitive to changes in these weights. This step adds efficiency since at this point these weights are not important.

Step (3): Apply in succession the BFGS and the ROLL algorithms (this adds efficiency and robustness since these two methods are succesful for different types of landscapes).

Step (4): Redetermine the non-influential weights and fix them (temporarily fixing non-influential weights is beneficial since, due to dimensionality reduction, the optimization problem becomes easier).

Step (5): Test if E_0 has been reached to decide whether to stop or not

Step (6): If the relative rate of progress per call $\frac{1}{Noc}\frac{\Delta E}{E} \leq r$ enhance the weight range as $b = min(d, b\,\alpha)$ ($Noc =$ Number of calls).

Step (7): Apply the PR method (Usually it is less efficient than BFGS, but performs less bookkeeping operations).

Step (8): Repeat from step (1).

In order to demonstrate the efficiency of our approach in MLP training we considered two real problems, since the increasing demand for high performance neural networks in real world applications renders obsolete any research based only on artificial benchmarks like XOR etc. Both real problems were selected from the Proben1 real world benchmark collection [14], since they are considered especially difficult and hence suitable for testing. In the first problem the approval of a credit card to a customer should be predicted, while in the second the diabetes of Pima Indians should be diagnosed. There are 51 (8) inputs, 2 outputs and 690 (768) examples divided randomly three times in 345 (384), 173 (192) and 172 (192) patterns for training, validating and testing respectively, hence forming card1, card2 and card3 (diabetes1, diabetes2 and diabetes3) tasks. In Table 1 we compare the results obtained in these six tasks by our Multi-Algorithm Optimization

Problem	Average (60 runs) training/validation/test error		
	MALO	PR-BP	Off-BP
card1	**0.98/8.44/10.10**	8.83/8.75/10.40	9.10/8.88/10.55
card2	**0.78/10.60/14.85**	8.47/10.95/15.10	8.12/11.10/15.22
card3	**0.75/8.55/12.98**	7.50/8.58/13.40	7.96/9.12/13.75
diabetes1	**12.05/15.47/16.25**	14.10/15.80/16.81	14.98/16.40/16.97
diabetes2	**10.35/16.80/17.94**	13.32/17.05/18.40	12.88/17.20/18.60
diabetes3	**10.04/17.47/15.88**	13.79/17.95/16.35	14.01/18.43/16.60

Table 1: Comparative results of different methodologies in MLP training

(MALO) methodology against to those obtained by the offline Backpropagation (Off-BP) ($learning - rate = 0.01, momentum = 0.05$) and the Polak-Ribiere Conjugate gradient method (PR-BP), according to Proben1 specifications concerning architectures, error measures and number of runs. MALO clearly outperforms the other methods as well as the RPROP algorithm used in Proben1 in terms of training average error reduction (notice an improvement of 16-90% regarding the best results obtained in Proben1 [14] with no-shortcut architectures).

4 A Strategy for Training Reinforcement Neurocontrollers

Another learning category where the MERLIN optimization system has been proved very useful is the case of delayed reinforcement learning. In this framework, a system receives input from its environment, selects and executes a sequence of actions, and at the end, receives a reinforcement signal, namely a grade for the made decision. A broad class of reinforcement problems is related with the task of controling a system in such a way, so that its state variables always remain within prescribed ranges. In the case where one or more state variables violate this restriction, the action selection system is penalized by receiving a "penalty" reinforcement signal. Examples of such kinds of problems are the pole balancing problem, teaching an autonomous robot to avoid obstacles, the ball and beam problem [13] etc.

A category of reinforcement learning techniques are the *direct* ones that consider only the action model (in order to provide the action policy) and optimization methods must be employed to adjust the parameters of the action model so that a stochastic integer-valued function is maximized. This function is actually proportional to the number of successful decisions (i.e. actions that do not lead to the receipt of penalty signal). In our case the action model has the architecture of a multilayer perceptron with input units accepting the system state at each time instant, and sigmoid output units providing output values p_i in the range $(0, 1)$. Based on these values the specification of the action to be taken is made either stochastically or deterministically.

Training is performed in cycles with each cycle starting with the system placed at a random initial position and ending with a failure signal. The number of time steps of the cycle constitutes the performance measure to be optimized by appropriately adjusting the parameters of the action network. In practice, when the length of a cycle exceeds a preset maximum number of steps, we consider that the controller has been adequately trained. This is used as a criterion for terminating the training process. There is also the possibility of unsuccessful training termination which occurs when the number of unsuccessful cycles (i.e. function evaluations without reaching maximum value) exceeds a preset upper bound.

Since the function to be optimized is integer-valued, gradient-based optimization techniques cannot be employed. A previous reinforcement learning approach that follows the direct strategy uses genetic algorithms to perform optimization with very good results in terms of training speed (required number of cycles) [15]. In our case, we have considered the derivative-free optimization procedures provided by MERLIN . Among them, the SIMPLEX method has been found to be very effective. The simplex algorithm (or polytope algorithm) starts with an initial simplex, which is subsequently adapted in order to reach the area of a minimum and, finally, it is shrinked around the minimum point.

The initial simplex may be constructed in various ways. At this point MERLIN has been found very useful since it gave us the capability to test several construction schemes of the initial polytope. The approach we followed was to pick the first vertex at random. The rest of the vertices were obtained by line searches originating at the first vertex, along each of the n directions. This initialization scheme proved to be very effective for the pole balancing problem. Other schemes such as, random initial vertices or constrained random vertices on predefined directions, etc, did not work well.

4.1 Solving the Pole Balancing Problem

The simplex-based delayed reinforcement training scheme was tested on the well-studied pole balancing problem. In this problem a single pole is hinged on a cart that may move left or right on a horizontal track of finite length. The pole has only one degree of freedom (rotation about the hinge point). The control objective is to push the cart either left or right with a force so that the pole remains balanced and the cart is kept within the track limits. At each time instant the status of the system is described by the following variables: the horizontal position of the cart (x), the cart velocity ($\dot{x}$), the angle of the pole (θ) and the angular velocity ($\dot{\theta}$) and the action network decides the direction and magnitude of force F to be exerted to the cart. It is assumed that a failure occurs when $|\theta| > 12$ degrees or $|x| > 2.4m$ and that training has been successfully completed if the pole remains balanced for more than 120000 consequtive time steps. We are concerned with the case where the magnitude is fixed ($|F| = 10N$) and the controller must decide only the direction of the force at each time step. Obviously the control problem is more difficult comprared to the case where any value for the magnitude is allowed. Details concerning the equations of motion of the cart-pole system can be found in [11, 15, 13]. These motion equations are unknown to the controller.

According to the specifications of [15, 11] the action network is a multilayer perceptron with four input units (accepting the sytem state), one hidden layer with five sigmoid units and one sigmoid unit in the output layer. There are also direct connections from the input units to the output unit. The specification of the applied force characteristics from the output value $y \in (0, 1)$ was performed in the following way. To intorduce a degree of randomness in the function evaluation process, at the first ten steps of each cycle the specification was probabilistic, i.e. $F = 10N$ with probability equal to y. At the remaining steps the specification was deterministic, i.e., if $y > 0.5$ then $F = 10N$, otherwise $F = -10N$.

The simplex algorithm was very effective being able to balance the pole in a relative few number of cycles (function evaluations) which was less than 1000 in many cases. Since the algorithm is deterministic its effectiveness depends partly on the initial weight values. For this reason we have employed an optimization strategy that is based on the simplex algorithm with random restarts. The following strategy was implemented in MCL: First simplex initialization takes place and then the simplex algorithm is run for up to 100 function evaluations (cycles) and the optimization progress is monitored. If a cycle has been found lasting more than 100 steps, application of the polytope algorithm continues for additional 750 cycles, otherwise we consider that the initial polytope was not proper and a random restart takes place. A random restart is also performed when after the additional 750 function evaluations a solution was not encountered. Moreover, a maximum of 15 restarts was allowed.

For comparison purposes we have also implemented the Adaptive Heuristic Critic (AHC) [11] method which belongs to the category of *critic-based* methods and assumes two separate models: an *action model* that selects the action to be taken at each step and the *evaluation model* which provides as output a prediction of the evaluation of the current state. Both models are implemented using feedforward neural networks that are trained on-line through backpropagation with the error being determined using the method of temporal differences [11]. A series of 50 experiments were conducted for each method and the average number of training cycles (function evaluations) was 2250 for the proposed technique and 6175 for the AHC method. Moreover, according to the results reported in [15], the avaerage number of cycles for genetic reinforcement approach is 4097. It is clear that the simplex-based training strategy implemented using the MERLIN optimization environment exhibits significantly better performance with respect to the AHC case and it also outperforms the genetic approach.

5 Unsupervised Training of Clustering Networks

Another neural network area of significant importance where MERLIN facilities can offer great convenience is the area of unsupervised training of neural networks and especially the problem of training a neural network to perform clustering, i.e. to organize unlabeled feature vectors into natural groups and represent them compactly with one or more prototypes. Almost any kind of unsupervised learning problem can be stated as an error minimization problem where the quantity to be minimized is appropriately formulated in order to satisfy the training objective.

Let $X = \{x_1, x_2, \ldots, x_n\}$ $(x_i \in R^p)$ denote the set of unlabeled data and c denote the number of clusters. In the case of *hard clustering*, clustering networks are winner-take-all networks (Figure 1), where each network unit i $(i = 1, \ldots, c)$ in the competitive layer corresponds to a cluster center $v_i = (v_{i1}, \ldots, v_{ip})$ whose coordinates v_{ij} can be considered as the weights of the p inputs to unit i. The objective of training is to adjust the weights of each unit (receptive fields) so that the clustering error J is minimized [16]:

$$J = \sum_{i=1}^{n} min_r d(x_i, v_r)$$

where $d(x_i, u_r)$ is the distance (usually Euclidean) between the vectors x_i and v_r. Many training algorithms have been developed to minimize the above clustering error. Most of them belong to the competitive learning framework with appropriate modifications in

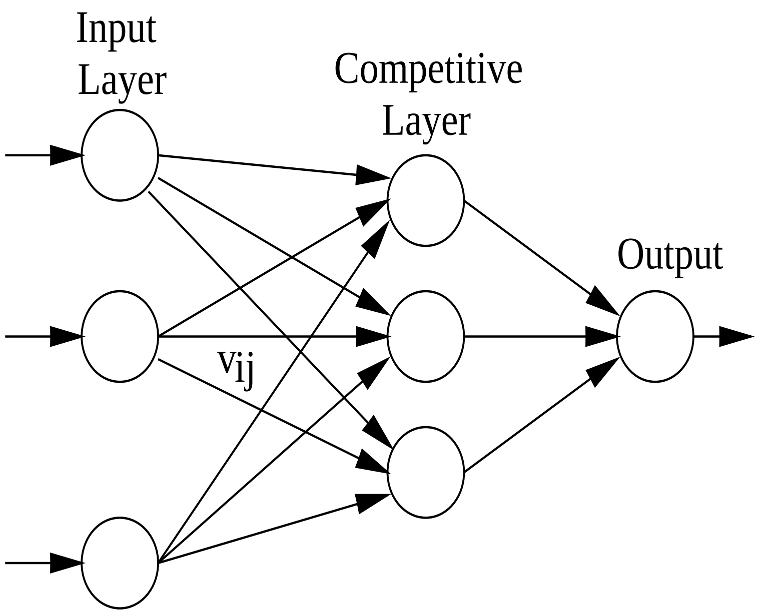

Figure 1: Clustering network

order to overcome certain training difficulties. Those algorithms operate on-line, in the sense that patterns are presented to the network one at a time, and the appropriate weight modifications take place at each step.

Since we deal with an optimization problem it is straightforward to consider the optimization capabilities provided by MERLIN . We consider as many network units as the desirable number of clusters and employ optimization strategies to minimize J, with the adjustable parameters being the input weights to each cluster unit. In fact, this is the approach proposed in [18] where simplex optimization procedure (provided by the Matlab optimization package) was employed. A disadvantage of this approach is that it does not fully exploit problem information since it is possible for the training algorithm to move cluster centers outside the domain area where the data points have been gathered. If m_i and M_i denote the minimum and maximum values at each dimension i ($i = (1, \ldots, p)$), then the problem can be transformed to a *constrained minimization* one:

$$\text{minimize } J \text{ with } v_{ij} \in [m_j, M_j].$$

The above problem can be easily handled by MERLIN by initially restricting the range of each adjustable parameter using the *margin specification* commands. In analogy with the supevised case, each function evaluation required a pass through the training set in order for the value of J to be computed. Therefore the approach can be classified as a batch training one.

Method	Function Evaluations
BFGS	635
SIMPLEX	1850
ROLL	822
CONGRA	750

Table 2: Average number of function evaluations to solve the IRIS clustering problem.

The technique was applied to the well-studied IRIS data [17] which is a set of 150 data points in R^4. Each point corresponds to one of three classes and there are 50 points of each class in the data set. Of course in this case class information was not taken into account during training. When three clusters are considered, the minimum error value is $J_{min} = 78.9$ [18] in the case where the Euclidean distance is considered. To tackle this clustering problem we considered a network with three clustering units, each having four inputs, ie. there were 12 training parameters. Each parameter value v_{ij} was initialized in the range $[m_j, M_j]$ according to the initialization scheme proposed in [19]:

$$v_{ij} = m_j + (i-1)\frac{M_j - m_j}{c-1}$$

To perform minimization we considered both derivative-free and derivative-based methods. In the latter case *numerical computation of the gradient* was used.

All tested algorithms exhibited very good performance being able to easily locate the global minimum. The number of function evaluations required for each of the used optimization methods is presented in Table 2. It can be observed that the gradient-based methods are faster despite the fact that the gradient is numerically computed.

The above work consitutes only a first attempt to treat unsupervised learning problem using MERLIN . There are many cases to be examined, like for example the employement of a different distance metric (for example the Mahalanobis distance), the use of different types of clustering (fuzzy, possibilistic clustering), the use of reference vectors other than points (for example lines or spheres) and the examination of MERLIN effectiveness in training Radial Basis Function (RBF) networks. All these cases can be easily treated with MERLIN and successful optimization strategies may be devised to tackle difficult clustering cases.

6 Conclusions

An approach has been proposed to neural network experimentation and training that is based on the employment of the Merlin general purpose optimization software [1, 2]. Such an approach requires that the user defines training as a minimization problem that is subsequently solved by invoking the procedures provided by the optimization environment. The user need not know exact details concerning the implementation of the procedures. In addition the capability of programming appropriate combinations of methods in terms of minimization strategies provides a very convenient way to implement and experiment with multialgorithm methods. The approach has been used in supervised, reinforcement and unsupervised learning problems with very good results. In the first two cases it has led to the development of novel effective training strategies, while in the unsupervised case it has very easily provided the optimal solution to a classical benchmark problem.

Future work will focus on the development of graphical user interfaces that will provide a convenient way for the specification of the training architecture and will automatically generate the code for the cost function to be minimized. In addition we will continue to experiment with other kinds of training problems as for example the training of recurrent neural networks and fuzzy neural networks.

One of us (I. E. L.) acknowledges partial support from the General Secretariat of Research and Technology under contract PENED 91 ED 959.

References

[1] Evangelakis GA, Rizos JP, Lagaris IE, Demetropoulos IN, *Merlin - A Portable System for Multidimensional Minimization*, Computer Physics Communications, vol. 46 pp. 402-412, 1987.

[2] Papageorgiou DG, Chassapis CS., Lagaris IE, *MERLIN-2.0 - Enhanced and programmable version*, Computer Physics Communications vol. 52, 241-247, 1989.

[3] Chassapis CS, Papageorgiou DG, Lagaris, IE., *MCL - Optimization Oriented Programming Language*, Computer Physics Communications vol. 52 pp. 223-239, 1989.

[4] Nelder JA, and Mead R., *A Simplex Method for Function Minimization*, Computer Journal vol. 7 pp. 308-313, 1965.

[5] Fletcher, R. and Reeves, C.M., *Function Minimization by Conjugate Gradients*, Computer Journal, vol. 7, pp. 149-154, 1964.

[6] Polak, E. and Ribiere, G., *Note sur la convergence de methodes de directions conjugees*, Revue Fr. Inf. Rech. Oper., vol. 16-R1, pp.35-43, 1969.

[7] Khoda, K.M., Liu, Y. and Storey, C., *Generalized Polak-Ribiere Algorithm*, J. of Optimization Theory and Applications, vol. 75, pp. 345-354, 1992.

[8] Fletcher, R. and Powell, M.J.D., *A rapidly convergent descent method for minimization*, Computer Journal, vol. 6, pp. 163-168, 1963.

[9] Fletcher, R., *A new approach to variable metric algorithms*, Computer Journal, vol. 13, pp. 317-322, 1970.

[10] More, J., *The Leveberg-Marquardt Algorithm: Implementation and Theory*, Lecture Notes in Mathematics, vol. 630, pp. 105-116, Springer, 1977.

[11] Anderson CW., *Learning to Control an Inverted Pendulum Using Neural Networks*, IEEE Control Systems Magazine, vol. 9, no. 3, pp. 31-37, 1989.

[12] Haykin, S., *Neural Networks: A Comprehensive Foundation*, Macmillan Publishing Company, 1994.

[13] Lin C-J, Lin C-T. *Reinforcement Learning for an ART-Based Fuzzy Adaptive Learning Control Network*, IEEE Trans. on Neural Networks, vol. 7 no. 3, pp. 709-731, 1996.

[14] Prechelt, L., *PROBEN1: A set of Neural Network Benchmark Problems and Benchmarking Rules*, Tech. Report 21/94, September 30, 1994, Fakultat fur Informatik, Universitat Karlshruhe, Germany.

[15] Whitley D, Dominic S, Das R. and Anderson CW, *Genetic Reinforcement Learning for Neurocontrol Problems*, Machine Learning vol. 13 pp. 259-284, 1993.

[16] Kohonen, T., *Self-Organization and Associative Memory*, 3rd ed. Berlin: Springer-Verlag, 1989.

[17] Anderson, E., *The IRISes of the Gaspe Peninsula*, Bull. Amer. IRIS Soc., vol. 59, pp. 381-406, 1935.

[18] Hathaway, R. J., *Optimization of Clustering Criteria by Reformulation*, IEEE Trans. on Fuzzy Systems, vol. 3, no. 2, pp. 241-245, 1995.

[19] Karayiannis, N., Bezdek, J.C., Pal, N.R., Hathaway, R.J. and Pai, P., *Repairs to GLVQ: A New Family of Competitive Learning Schemes*, IEEE Trans. on Neural Networks, vol. 7, no. 5, pp. 1062-1071, 1996.