A Neural-Net Training Program Based on Conjugate-Gradient Optimization

Etienne Barnard and Ronald A. Cole

Technical Report No. CS/E 89-014

A neural-net training program based on conjugate-gradient optimization

Etienne Barnard

Carnegie Mellon University
Department of Electrical and
Computer Engineering
Pittsburgh, PA 15213

Ronald A. Cole

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 89-014

July, 1989

I. Introduction

Neural-net classifiers have been demonstrated to be attractive alternatives to conventional classifiers by numerous researchers [1, 2, 3, 4]. However, there are two reasons why these classifiers have not gained wider acceptance:

1. they have a reputation for being highly wasteful of computational resources during training. In one of the first papers on neural-net classifiers [5], a relatively simple two-class experiment is described that required several thousand iterations to train, and Waibel et al. [2] describe a three-class problem that required more than 20 000 training iterations and several days on a supercomputer.
2. their training has conventionally been associated with the heuristic choice of a number of parameters; if these parameters are chosen incorrectly, poor performance results, yet no theoretical basis for choosing them appropriately for a given problem exists.

In this report we describe a technique that goes some way towards eliminating these disadvantages. To do this, we use an optimization technique extensively studied within the numerical-analysis literature instead of the variants of gradient descent customarily employed with neural-net training. Use of conventional optimization techniques for the training of neural nets has been proposed by numerous authors [6, 7, 3], for reasons which will become clear. Also described in the current report are two programs which implement the training and testing phases of a neural-net classifier based on these ideas.

II. Conjugate-gradient optimization for training neural nets

It is well known [5] that the training of the most popular neural-net classifier (known as "backpropagation" or, less accurately, as the "multilayer perceptron") is accomplished by the optimization of a criterion function E. (The function E depends on the training data according to details that will not concern us here.) When backpropagation was introduced originally [5], it was proposed that the criterion function be optimized using gradient descent. However, it was soon realized that more efficient training techniques can be employed; the "momentum" term [5] is the most popular example of such improvements. In this approach, not only the gradient, but also the previous weight change is used to update the weight vector. Various researchers have shown the usefulness of the momentum approach; its efficacy can be understood by analogy with the quadratic case, where it improves the minimization rate by creating a second-order dynamic system (i.e. a system whose time behavior is characterized by two poles in the frequency domain). The second-order dynamic system can be shown to have much better asymptotic convergence properties than the first-order system that describes gradient descent without momentum.

The momentum algorithm is characterized by two parameters (the "learning rate" and the "momentum constant"), whose values are critical to the success of the optimization process. Although it is not hard to derive optimal values for these constants in the quadratic case, no equivalent result for the more general case - which almost always obtains for neural-net classifiers- exists. A large number of heuristic algorithms that adapt these parameters according to the training set and the status of the neural net have therefore been proposed (see, for instance, [8, 9, 10]). These heuristics are usually derived in ad-hoc fashion, so that neither optimality nor robustness is ensured. In fact, in our experience most adaptive heuristics still leave a number of problem-dependent parameters for the user to choose, and still are rather slow to optimize E.

One way of overcoming these limitations is to utilize the body of knowledge available from

conventional numerical analysis. Unconstrained optimization, which is required for the training we study, is one of the most mature disciplines within numerical mathematics [11], and supplies a wealth of applicable knowledge. The current opinion in the optimization community is [11] that the fastest general-purpose algorithms for unconstrained non-linear optimization are the variable-metric (also called quasi-Newton) methods. Unfortunately, these techniques require $O(N^2)$ storage locations, where N is the number of minimization variables (in our case, the number of weights). Since N is often fairly large in practical neural-net applications (problems with several thousand weights are not unusual), this storage requirement is a major drawback. We therefore use a conjugate-gradient optimization algorithm [12]. The conjugate-gradient algorithm is usually able to locate the minimum of a multivariate function much faster than the gradient-descent procedure that is customarily employed with BP. Furthermore, its memory usage is on the order of N locations. It is important to note that the conjugate-gradient technique eliminates the choice of critical parameters (such as the learning-rate and momentum parameters of BP), making it much easier to use than the conventional BP training algorithm.

We now briefly describe the principles behind conjugate-gradient optimization. For more complete information, the paper by Powell [12] should be consulted. As with most optimization procedures, the principles of conjugate-gradient optimization are derived for a quadratic criterion function. The guiding idea is to maintain "conjugacy" of the successive search directions when such a quadratic is optimized, as we now explain. In Fig. 1, the contours of constant function

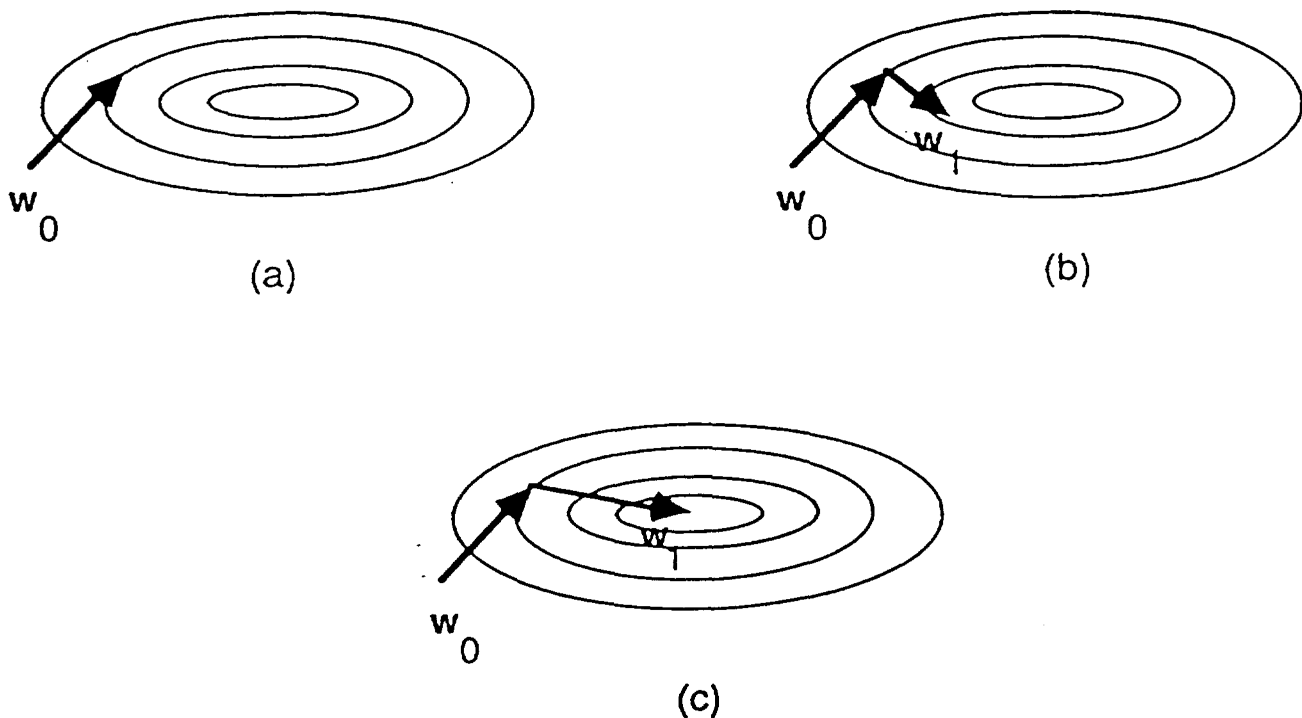

Figure 1: (a) Location of the minimum along the direction of $\mathbf{w}_0$
(b) Subsequent minimization along gradient direction (c) Minimization along direction conjugate to $\mathbf{w}_0$

value are shown for a two-dimensional slice through a quadratic function. Say we have located the minimum along the direction $\mathbf{w}_0$ indicated by the arrow in Fig. 1(a), and we now want to choose a new direction along which to search for a one-dimensional minimum. In Fig. 1(b) the new search direction $\mathbf{w}_1$ is chosen to point in a direction opposite to the gradient at the position of the current minimum; it is clear that it will be necessary to minimize along $\mathbf{w}_0$ again after the minimum along $\mathbf{w}_1$ has been located. The search direction in Fig. 1(c), on the other hand, was chosen to ensure that the minimum along $\mathbf{w}_0$ is maintained (that is, all points along $\mathbf{w}_1$ share the property that their value cannot be decreased by moving in the direction of $\mathbf{w}_0$). This obviates the need for another search along $\mathbf{w}_0$. This example suggests a reasonable criterion for the choice of a new search direction, namely that it should not interfere with the minima found along the previous search directions. It turns out that there is a simple algorithm for constructing such a direction out of the gradient and the previous search direction when the function to be minimized is quadratic [12]. This algorithm requires only $O(N)$ computation per iteration and storage, and is known as the "conjugate gradient" algorithm.

It is indicative of the analytical complexity of nonlinear optimization that no general results on the relative merits of these two types of algorithms exist, despite their popularity in practical use. Because the variable-metric methods accumulate more knowledge of the error function, one expects that they will generally require fewer iterations to find the minimum than conjugate-gradient techniques; this is indeed what is observed in most benchmarks (see, e.g., Gill et al. [11]). However, we know of no characterization that will enable one to decide which method is most efficient computationally in a given situation. Thus, for a particular type of problem, it is necessary to empirically compare the two types of algorithms to see whether the increased rate of convergence of variable-metric methods outweighs its additional computational and storage requirements. For the case of neural-net criterion functions, such an empirical comparison has been performed [13] using (amongst others) the programs described in the next section. It was found that conjugate-gradient training is indeed slower than variable-metric training, but that the difference is rather small. For various problems it was found that the conjugate-gradient technique required approximately 10-20% more iterations than the variable-metric technique to reach a given level of accuracy. Depending on the problem, this might actually translate into less computer time for conjugate-gradient optimization, because of its reduced complexity with respect to the variable-metric methods. In any case, the $O(N^2)$ storage of the latter methods is apparently not justified for training "multilayer perceptrons".

We do not try to compare conjugate-gradient training with "momentum" training formally, since there is no standard way of choosing the parameters of the latter (as was explained above). However, we do not believe that any of the heuristic algorithms for choosing these parameters can consistently compete with optimized conjugate-gradient training as far as speed and robustness are concerned; the existence of such an algorithm would probably present a major surprise to most of the numerical-analysis community!

III. Programs for training and testing neural classifiers

In this section we describe the usage of a program "opt", that implements the training phase of a neural-net classifier based on conjugate-gradient optimization. The program was designed to allow the experimenter to train a neural-net classifier with minimal attention to the innards of the classifier. It is used in conjunction with the program "optest", that tests the performance of the classifier created with "opt".

The neural net implemented by "opt" is the "multilayer perceptron" shown in Fig. 2. It consists of N layers, each containing m_i neurons ($i=1, 2, ..., N$). Successive layers are completely

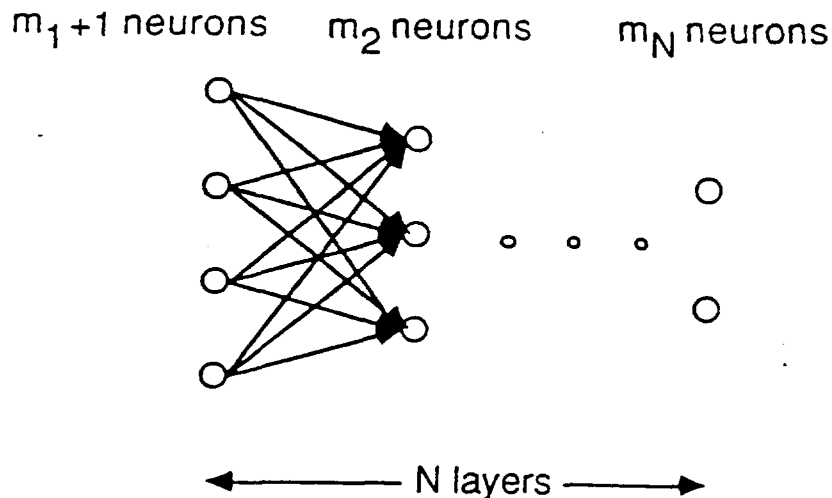

Figure 2: Structure of a neural net with complete interconnection between successive layers, and only feedforward connections

interconnected in the forward direction, as indicated by the arrows in Fig. 2. By choosing this structure, we have eliminated various possibilities which have been investigated in the literature (such as recurrent nets [14], nets with certain weights constrained to be equal [2], or nets with more elaborate connectivity [15]). This was done since the structure we have chosen is known to be computationally complete [16] in the sense that any classification boundary can be approximated to arbitrary accuracy with this structure. Thus, whereas other structures might be useful for reasons such as efficiency, or for applications going beyond conventional classification, the essential properties of neural-net classifiers are all captured by the topology implemented by "opt".

A. Using "opt"

To use "opt", the user supplies two free-format text files. The first, "opt.da*x*", contains data to specify the neural net, and the other, "clas.da*x*", contains the training data. In both cases, x is the same integer in the range 0 - 99; it is used to label the particular data set. The program assumes that these files are located in the current directory (that is, the directory from which execution is initiated).

The program is invoked with the command

opt *x continue_flag print_flag*

with x the integer characterizing the data set, *continue_flag* a binary digit specifying whether the net should start from random values (*continue_flag*=0) or use a previously trained net as starting point (*continue_flag*=1), and *print_flag* a binary digit indicating whether the value of the criterion function should be reported after every iteration (*print_flag*=1) or not (*print_flag*=0). If *continue_flag*=1, the file "opt.o*x*" (see below) should exist in the current directory. For

print_flag=1, the function values are printed to the standard output - it is often convenient to redirect these values to a file for inspection after training has terminated.

In "opt.da*x*", the following quantities are specified (in the order given below):

1. N, the number of layers in the neural net. The input and output layers are included in the count of N, and it is assumed that N is not less than 3; that is, the net has at least one hidden layer.
2. *nit*, the maximum number of iterations of the conjugate-gradient procedure that should be performed.
3. ε, the stopping criterion - if the norm of the gradient becomes less than ε, training is terminated; otherwise, it continues until *nit* iterations have been completed.
4. ρ, a seed with which the random weights are initialized.
5. m_i (i=1, 2, .., N), the size of each of the N layers. m_1 should equal the number of input features, and m_N should equal the number of classes into which the samples are to be classified. The number of output neurons equals the number of classes since each class is represented by a separate neuron. That is, if the k-th neuron in the output layer is most active, the input is classified into class k. Note that the input layer will actually contain m_1+1 neurons, since an "always-on" neuron is included to allow for a variable threshold [17]. The activity of this neuron is fixed at +1.

All these quantities are integers, except for ε, which is a floating-point number.

The assumption that $N \geq 3$ ensures that we have a "true" neural classifier, not a linear classifier. For N=2, the program "opt_l" - that trains such a linear classifier - should be used. Except for the fact that "opt_l" assumes N=2, "opt" and "opt_l" are identical. Thus, "opt_l" will not be discussed any further.

In "clas.da*x*" we include the following data:

1. *ntrain*, an integer equal to the number of training samples in the file;
2. for each of the *ntrain* training samples,
    - an integer in the range 1,2,..,m_N, specifying the class to which this sample belongs, and
    - m_1 floating-point numbers, each representing a different feature value. To minimize local-minimum problems, these features should be scaled so that all or almost all values are between -1 and 1.

After training, "opt" creates a binary file "opt.o*x*" (*x* as above) that contains the trained weights. This file is used for further training (when *continue_flag*=1) or testing, as is explained below.

B. Using "optest"

To test the performance of the trained net, "optest" is used with the command

optest *x y*

with *x* specifying which trained net is to be tested, and *y* specifying the test set to be employed. Thus, both *x* and *y* are integers in the range 0, 1, .., 99; the trained net is contained in the file "opt.o*x*", and the set of test samples is contained in "clas.da*y*", the format of which is explained above.

This command produces a text file "test.y", containing the percentage of test samples correctly classified and the confusion matrix (rows correspond to the true classes and columns to the classes assigned to samples by the neural net).

C. Using "wread"

The program "wread" converts the binary file "opt.ox" into a text file, to allow visual inspection of the weights produced by training. It is invoked by

wread x

with x the integer labeling the training and specification files used to produce the neural net. This command produces the text file "wread.ox", containing the weights in a format which is explained below.

D. Example

We now give examples of the files described in the previous subsections. Consider the exclusive-or problem, as shown in Fig. 3. It consists of four training samples from two classes. To train a three-layered net with three hidden neurons on this problem, we use the file "opt.da0" shown in Fig. 4.

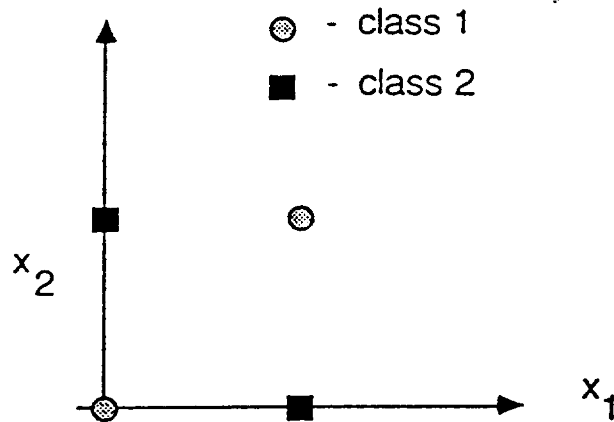

Figure 3: The exclusive-or classification problem

```
3 30 0.0000001 1989
2 3 2
```

Figure 4: Contents of "opt.da0" to train on exclusive-or problem

The sample vectors used for training are contained in the file "clas.da0", as is shown in Fig. 5.

```
4
1 0 0
2 0 1
2 1 0
1 1 1
```

Figure 5: Contents of "clas.da0" to train on exclusive-or problem

By typing

```
opt 0 0 1 > e_log
```

we will let the conjugate-gradient optimizer run for 30 iterations starting from random initial weights (whose values are determined by the seed number 1989). The values of the criterion function on successive iterations are directed to the file "e_log" (for a typical example see Fig. 6), and the binary output file "opt.o0" contains the resulting weights of the classifier. In Fig. 6 it can be seen how the value of the criterion function is gradually decreased as the training procedure iterates.

```
0.2910
0.2933
0.2139
0.1967
0.1693
0.2407
0.1650
0.1641
0.1622
0.1590
0.1912
0.1581
0.1579
0.1579
0.1623
0.1579
0.1574
0.1538
0.1533
0.1527
0.1509
0.1500
0.1461
0.1444
0.1373
0.1307
0.1302
0.1197
0.1131
0.1051
```

Figure 6: Output produced by command "opt 0 0 1"

For the exclusive-or problem, we wish to test on the same samples we trained on, so "optest" is invoked with the arguments "0 0". This produces the file "test.0", shown in Fig. 7. We see that 3 of the 4 samples are correctly classified after 30 iterations (a "1" was called a "2"). (If we had wanted to use a different test set - contained in "clas.da50", say - we would have given the command "optest 0 50").

To inspect the weoghts leading to this classification, we run "wread 0". The contents of the file "wread.o0" produced by this are shown in Fig. 8. As can be seen, the first several lines of

```
 # of its.: 30; % correct= 75.00
label    1    2
----- ---- ----
   1:    1    1
   2:    0    2
```

Figure 7: Contents of "test.0" after execution of "optest 0 0"

"wread.o0" describe the size of the neural net used. (Note that the number of input neurons is three, one more than the number of features, because of the "always-on" neuron. By convention, this neuron is neuron m_1+1 in the input layer, with the first m_1 neurons in the first layer corresponding to the m_1 features.) Thereafter, the weights are listed, with all weights leading to a given neuron on the same line. For a neuron in layer i, the weights from the m_{i-1} neurons in layer $i-1$ are listed in order. Thus, in our example, the second neuron in layer 2 is connected to the first feature by a weight of 6.569, to the second feature with a weight of -1.950, and to the always-on neuron by a weight of 0.972.

```
# of layers: 3
layer 1 has 3 neurons
layer 2 has 3 neurons
layer 3 has 2 neurons

Weights:
***Layer 1 to layer 2***
To neuron 1:
 -9.432522 10.613865 -7.209504
To neuron 2:
 6.569661 -1.950402 0.972534
To neuron 3:
 1.260390 -6.663808 3.632652
***Layer 2 to layer 3***
To neuron 1:
 1.109862 -0.393785 0.336293
To neuron 2:
 -1.055528 0.674493 -0.387741
Value of criterion function: 0.105076
```

Figure 8: Contents of "wread.o0" after execution of "wread 0"

Further training can be initiated by changing the number of iterations in "opt.da0" if desired, and running "opt" with *continue_flag* equal to 1. The criterion-function values output by the program are useful to determine whether it is sensible to continue iterating. For the output shown in Fig. 6, further minimization is clearly required, since the criterion function values were still decreasing rapidly when optimization was terminated after 30 iterations. Note that the number of iterations does not accumulate over separate runs, so the program will execute *nit* iterations after being started regardless of the number of iterations previously completed.

IV. Comments on the classifier implemented

The network implemented by "opt" is fairly standard, so that the details given in e.g. [5] are mostly applicable here. However, a number of points should be noted:

- we have implemented a training "cutoff" - instead of using desired values of 0 and 1 for the output neurons, we have chosen 0.3 and 0.7, respectively. When the output

exceeds 0.7 for a desired value of 0.7, or when the output is less than 0.3 for a desired value of 0.3, the error is set to 0. This cutoff procedure is useful since it counteracts overtraining. The values of 0.3 and 0.7 were chosen heuristically; the classifier is reasonably insensitive to changes in these values.

- the initial weight choices are determined by two contradictory factors: to break the symmetry between the various hidden neurons, the weights should be large, but to place the initial decision boundaries in reasonable places (to minimize local-minimum problems) the weights should not be too large. "Large" and "small" are determined by the scale of the input features, since the output of a given hidden neuron depends on the product of the weight and feature values. In the problems we have studied, we have scaled the inputs in the range [0,1] or [-1,1]. With the random weights to a node with n_to incoming connections scaled to be uniformly distributed between $-3/\sqrt{n_to}$ and $3/\sqrt{n_to}$, satisfactory performance was obtained. (Thus, the sum of the weights leading to a particular node is a random variable with zero mean and a variance of 3, independent of the number of connections leading to it.) This scaling is used in "opt".
- as described by Powell [12], the conjugate-gradient algorithm (with restarts) depends on a number of parameters - e.g. line-search accuracy, the criterion for restarting, etc. Reasonable values for these parameters were determined by comparing various implementations; in any case, the algorithm is fairly insensitive to their exact values.

References

[1] R.P.Gorman and T.J.Sejnowski, "Analysis of hidden units in a layered network trained to classify sonar signals," *Neural Networks*, Vol. 1, pp. 75-90, 1988.

[2] A.Waibel,T.Hanazawa,G.Hinton,K.Shikano and K.J.Lang, "Phoneme Recognition Using Time-Delay Neural Networks," *IEEE Trans. Acoust., Speech, Signal Processing*, Vol. ASSP-37, pp. 328-339,March 1989.

[3] E.Barnard and D.Casasent, "A comparison between criterion functions for linear classifiers, with an application to neural nets," *IEEE Trans. Syst., Man, Cybern.*, Submitted, 1988.

[4] D.J.Burr, "Experiments on neural net recognition of spoken and written text," *IEEE Trans. ASSP*, Vol. ASSP-36, pp. 1162-1168,July 1988.

[5] D. E. Rumelhart,G. E. Hinton and R. J. Williams, "Learning internal representations by error propagation, " in *Parallel Distributed Processing*. Cambridge MA: MIT press, pp. 318-362,ch. 8,1986.

[6] R.L.Watrous, "Learning algorithms for connnectionist networks: applied gradient methods of non-linear optimization," *International Conference on Neural Networks*, IEEE, San Diego, July 1987, pp. II-619 - II-627.

[7] A.Lapedes and R.Farber, "How neural nets work," *Neural Information Processing Systems*, D.Z.Anderson,ed., AIP, Denver, Co, 1988, pp. 442-456.

[8] J.P.Cater, "Successfully using peak learning rates of 10 (and greater) in Back-propagation networks with the heuristic learning algorithm," *International Conference on Neural Networks*, IEEE, San Diego, July 1987, pp. II-645 - II-651.

[9] T.P.Vogl, J.K.Mangis, A.K.Zigler, W.T.Zink and D.L.Alkon, "Accelerating the convergence of the back-propagation method," *Biological Cybernetics*, Vol. 59, pp. 257-264,Sept. 1988.

[10] R.A.Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, Vol. 1, pp. 295-308, 1988.

[11] P.E.Gill, W.Murray and M.H.Wright, *Practical optimization*. London: Academic Press,

1981.

[12] M.J.D.Powell, "Restart Procedures for the Conjugate Gradient Method," *Mathematical Programming*, Vol. 12, pp. 241-254,Apr. 1977.

[13] E.Barnard, "Optimization for training neural nets," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Submitted, March 1989.

[14] F.J.Pineda, "Dynamics and architecture for neural computation," *J. of Complexity*, Vol. 4, pp. 216-245,Sept. 1988.

[15] K.J.Lang and G.E.Hinton, "The development of the time-delay neural network architecture for speech recognition", Tech. report CMU-CS-88-152, Department of Computer Science, Carnegie Mellon University, 1988.

[16] K.Hornik, M.Stinchcombe and H.White, "Multilayer feedforward networks are universal approximators", Discussion Paper 88-45, UCSD Department of Economics, Feb. 1989.

[17] R.P.Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, Vol. 2, pp. 4-22,April 1987.