Speeding Up Fuzzy Clustering with Neural Network Techniques

Christian Borgelt and Rudolf Kruse

Research Group Neural Networks and Fuzzy Systems
Dept. of Knowledge Processing and Language Engineering, School of Computer Science
Otto-von-Guericke-University of Magdeburg, Universitätsplatz 2, Germany
e-mail:{borgelt,kruse}@iws.cs.uni-magdeburg.de

Abstract— **We explore how techniques that were developed to improve the training process of artificial neural networks can be used to speed up fuzzy clustering. The basic idea of our approach is to regard the difference between two consecutive steps of the alternating optimization scheme of fuzzy clustering as providing a gradient, which may be modified in the same way as the gradient of neural network backpropagation is modified in order to improve training. Our experimental results show that some methods actually lead to a considerable acceleration of the clustering process.**

I. Introduction

The basic idea to train an artificial neural network, especially a multilayer perceptron, is to carry out a gradient descent on the error surface [Haykin 1994], [Zell 1994], [Anderson 1995]. That is, the error of the output of the neural network (w.r.t. a given set of training examples) is seen as a function of its parameters, i.e., connection weights and bias values. Hence the partial derivatives of this function w.r.t. the parameters can be computed, yielding the direction of steepest ascent of the error function. Starting from a random initialization of the parameters, they are changed in the direction opposite to this gradient, since it is the goal to minimize the error. At the new point in the parameter space the gradient is recomputed and another step in the direction of steepest descent is carried out. The process stops if a (local) minimum of the error function is reached.

Fuzzy clustering usually also employs an iterative optimization scheme [Bezdek 1981], [Bezdek and Pal 1992], [Bezdek *et al.* 1999], [Höppner *et al.* 1999]. Starting from a random initialization of the cluster parameters (center coordinates and size and shape parameters), the data points are assigned—with a certain degree of membership—to the different clusters based on their distances to these clusters. Then the parameters of a cluster are recomputed from the data points assigned to it, respecting, of course, the degree of membership of the data points to this cluster. The idea underlying this computation is to minimize the sum of the squared distances of the data points to the cluster centers. The process of assigning data points and recomputing the cluster parameters is iterated until the clusters are stable.

Since both methods rely on a minimization of a function, namely the error function of a neural network or the sum of squared distances for fuzzy clustering, the idea suggests itself to transfer improvements that have been developed for one method to the other. Here we focus on a transfer of the different improvements of the gradient descent training method for neural networks to fuzzy clustering. Our rationale is that the difference between the parameter values of two consecutive steps of the alternating optimization scheme of fuzzy clustering can be seen as a kind of (inverse) gradient. In this way we can apply the same modifications as in neural network training.

II. Neural Network Techniques

In this section we review some of the best-known methods for improving the gradient descent training process of an artificial neural network, including a momentum term, parameter-specific self-adapting learning rates, resilient backpropagation and quick backpropagation.

A. Standard Backpropagation

In order to have a reference point, we state first the weight update rule for standard error backpropagation. This rule reads for a parameter w:

$$w(t+1) = w(t) + \Delta w(t) \quad \text{where} \quad \Delta w(t) = -\eta \nabla_w e(t).$$

That is, the new value of the parameter (in step $t+1$) is computed from the old value (in step t) by adding a weight change, which is computed from the gradient $\nabla_w e(t)$ of the error function $e(t)$ w.r.t. this parameter w. η is a learning rate that influences the size of the steps that are carried out. The minus sign results from the fact that the gradient points into the direction of the steepest *ascent*, but we have to carry out a gradient *descent*. Depending on the definition of the error, the gradient may also be preceded by a factor of $\frac{1}{2}$ in this formula, in order to cancel a factor of 2 that results from differentiating a squared error.

Note that it can be difficult to choose an appropriate learning rate, because a small learning rate can lead to unnecessarily slow learning, whereas a large learning rate can lead to oscillations and uncontrolled jumps.

B. Momentum Term

The momentum term method [Rumelhart *et al.* 1986b] consists in adding a fraction of the weight change of the previous step to a normal gradient descent step. The rule for changing the weights thus becomes

$$\Delta w(t) = -\eta \nabla_w e(t) + \beta\, \Delta w(t-1),$$

where β is a parameter, which must be smaller than 1 in order to make the method stable. In neural network training β is usually chosen between 0.5 and 0.95.

The additional term $\beta\ \Delta w(t-1)$ is called *momentum term*, because its effect corresponds to the momentum that is gained by a ball rolling down a slope. The longer the ball rolls in the same direction, the faster it gets. So it has a tendency to keep on moving in the same direction (momentum term), but it also follows, though slightly retarded, the shape of the surface (gradient term).

By adding a momentum term training can be accelerated, especially in areas of the parameter space, in which the error function is (almost) flat, but descends in a uniform direction. It also reduces slightly the problem of how to choose the value of the learning rate, although this is not relevant for our current investigation.

C. Self-Adaptive Backpropagation

The idea of (super) self-adaptive backpropagation (SuperSAB) [Jakobs 1988], [Tollenaere 1990] is to introduce an individual learning rate η_w for each parameter of the neural network, i.e., for each connection weight and each bias value. These learning rates are then adapted (before they are used in the current update step) according to the values of the current and the previous gradient. The exact adaptation rule for the learning rates is

$$\eta_w(t) = \begin{cases} \gamma^- \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \quad \cdot \nabla_w e(t-1) < 0, \\ \gamma^+ \cdot \eta_w(t-1), & \text{if } \nabla_w e(t) \quad \cdot \nabla_w e(t-1) > 0 \\ & \quad \wedge\ \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \eta_w(t-1), & \text{otherwise.} \end{cases}$$

γ^- is a shrink factor ($\gamma^- < 1$), which is used to reduce the learning rate if the current and the previous gradient have opposite signs. In this case we have leaped over the minimum, so smaller steps are necessary to approach it. Typical values for γ^- are between 0.5 and 0.7.

γ^+ is a growth factor ($\gamma^+ > 1$), which is used to increase the learning rate if the current and the previous gradient have the same sign. In this case two steps are carried out in the same direction, so it is plausible to assume that we have to run down a longer slope of the error function. Consequently, the learning rate should be increased in order to proceed faster. Typically, γ^+ is chosen between 1.05 and 1.2, so that the learning rate grows slowly.

The second condition for the application of the growth factor γ^+ prevents that the learning rate is increased immediately after it has been decreased in the previous step. A common way of implementing this is to simply set the previous gradient to zero in order to indicate that the learning rate was decreased. Although this also suppresses two consecutive reductions of the learning rate, it has the advantage that it eliminates the need to store $\nabla_w e(t-2)$.

In order to prevent the weight changes from becoming too small or too large, it is common to limit the learning rate to a reasonable range. It is also recommended to use batch training, as online training tends to be unstable.

D. Resilient Backpropagation

The resilient backpropagation approach (Rprop) [Riedmiller and Braun 1993] can be seen as a combination of the ideas of Manhattan training (which is like standard backpropagation, but only the sign of the gradient is used, so that the learning rate determines the step width directly) and self-adaptive backpropagation. For each parameter of the neural network, i.e., for each connection weight and each bias value, a *step width* Δw is introduced, which is adapted according to the values of the current and the previous gradient. The adaptation rule reads

$$\Delta w(t) = \begin{cases} \gamma^- \cdot \Delta w(t-1), & \text{if } \nabla_w e(t) \quad \cdot \nabla_w e(t-1) < 0, \\ \gamma^+ \cdot \Delta w(t-1), & \text{if } \nabla_w e(t) \quad \cdot \nabla_w e(t-1) > 0 \\ & \quad \wedge\ \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \Delta w(t-1), & \text{otherwise.} \end{cases}$$

In analogy to self-adaptive backpropagation, γ^- is a shrink factor ($\gamma^- < 1$) and γ^+ a growth factor ($\gamma^+ > 1$), which are used to decrease or increase the step width. The application of these factors is justified in exactly the same way as for self-adaptive backpropagation. The typical ranges of values also coincide ($\gamma^- \in [0.5, 0.7]$ and $\gamma^+ \in [1.05, 1.2]$).

Like in self-adaptive backpropagation the step width is restricted to a reasonable range in order to avoid far jumps as well as slow learning. It is also advisable to use batch training as online training can be very unstable.

In several applications resilient backpropagation has proven to be superior to a lot of other approaches (including momentum term, self-adaptive backpropagation, and quick backpropagation), especially w.r.t. training time [Zell 1994]. It is definitely one of the most highly recommendable methods for training multilayer perceptrons.

E. Quick Backpropagation

The idea underlying quick backpropagation (Quickprop) [Fahlman 1988] is to locally approximate the error function by a parabola (see Figure 1) and to change the weight in such a way that we end up at the apex of this parabola, i.e., the weight is simply set to the value at which the apex lies. If the error function is "good-natured", i.e., can be approximated well by a parabola, this enables us to get fairly close to the true minimum in one or very few steps.

The rule for changing the weights can easily be derived from the derivative of the approximation parabola (see Figure 2). Obviously, it is (consider the shaded triangles, both of which describe the ascent of the derivative)

$$\frac{\nabla_w e(t-1) - \nabla_w e(t)}{w(t-1) - w(t)} = \frac{\nabla_w e(t)}{w(t) - w(t+1)}.$$

Solving for $\Delta w(t) = w(t+1) - w(t)$ and exploiting that $\Delta w(t-1) = w(t) - w(t-1)$ we get

$$\Delta w(t) = \frac{\nabla_w e(t)}{\nabla_w e(t-1) - \nabla_w e(t)} \cdot \Delta w(t-1).$$

However, it has to be taken into account that the above formula does not distinguish between a parabola that opens

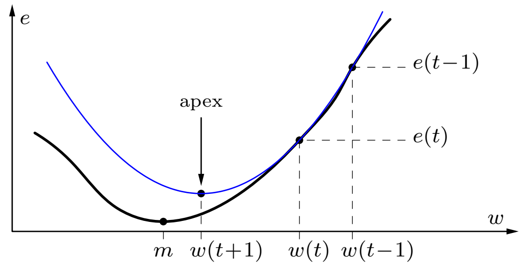

Fig. 1. Quick backpropagation uses a parabola to locally approximate the error function. m is the true minimum.

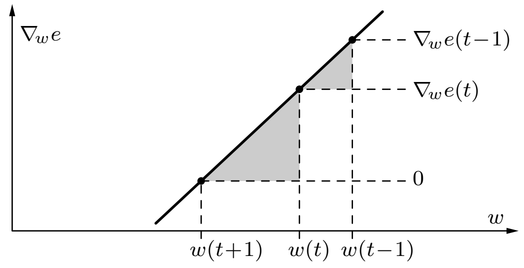

Fig. 2. The formula for the weight change can easily be derived from the derivative of the approximation parabola.

upwards and one that opens downwards, so that a maximum of the error function may be approached. Although this can be avoided by checking whether

$$\frac{\nabla_w e(t-1) - \nabla_w e(t)}{\Delta w(t-1)} < 0$$

holds (parabola opens upwards), this check is often missing in implementations. Furthermore a growth factor is introduced, which limits the weight change relative to the previous step. That is, it is made sure that

$$|\Delta w(t)| \leq \gamma \cdot |\Delta w(t-1)|$$

holds, where γ is a parameter, which is commonly chosen between 1.75 and 2.25.

In addition, neural network implementations of this method often add a normal gradient descent step if the two gradients $\nabla_w e(t)$ and $\nabla_w e(t-1)$ have the same sign, i.e., if the minimum does not lie between the current and the previous weight value. In addition it is advisable to limit the weight change in order to avoid far jumps.

If the assumptions underlying the quick backpropagation method, namely that the error function can be approximated locally by a parabola that opens upwards and that the parameters can be changed fairly independent of each other, and if batch training is used, it is one of the fastest learning methods for multilayer perceptrons. Otherwise it tends to be unstable and is susceptible to oscillations.

III. Fuzzy Clustering

Fuzzy clustering is an objective function based method to divide a dataset into a set of groups or clusters. In contrast to standard (crisp) clustering, fuzzy clustering offers the possibility to assign a data point to more than one cluster, so that overlapping clusters can be handled conveniently. Each cluster is represented by a prototype, which consists of a cluster center and maybe some additional information about the size and shape of the cluster. The degree of membership, to which a data point belongs to a cluster, is computed from the distances of the data point to the clusters center w.r.t. the size and shape information.

Formally, given a dataset $\mathbf{X}$ of n data points $\vec{x}_j \in \mathbb{R}^p$ that is to be divided into c clusters, the clustering result is obtained by minimizing the objective function

$$J(\mathbf{X}, \mathbf{U}, \mathbf{B}) = \sum_{i=1}^{c} \sum_{j=1}^{n} u_{ij}^m d^2(\beta_i, \vec{x}_j)$$

subject to

$$\forall i \in \{1, \ldots, c\}: \qquad \sum_{j=1}^{n} u_{ij} > 0 \qquad \text{and}$$

$$\forall j \in \{1, \ldots, n\}: \qquad \sum_{i=1}^{c} u_{ij} = 1.$$

Here $u_{ij} \in [0, 1]$ is the membership degree of the data point $\vec{x}_j$ to the i-th cluster, β_i is the prototype of the i-th cluster, and $d(\beta_i, \vec{x}_j)$ is the distance between data point $\vec{x}_j$ and prototype β_i. $\mathbf{B}$ denotes the set of all c cluster prototypes $\beta_1, \ldots, \beta_c$. The $c \times n$ matrix $\mathbf{U} = [u_{ij}]$ is called the fuzzy partition matrix and the parameter m is called the fuzzifier. This parameter determines the "fuzziness" of the clustering result. With higher values for m the boundaries between the clusters become softer, with lower values they get harder. Usually $m = 2$ is chosen. The two constraints ensure that no cluster is empty and that the sum of the membership degrees for each datum equals 1, so that each data point has the same weight.

Unfortunately, the objective function J cannot be minimized directly. Therefore an iterative algorithm is used, which alternatingly optimizes the cluster prototypes and the membership degrees. The update formulae are derived by differentiating the objective function (extended by Lagrange multipliers to incorporate the constraints) w.r.t. the parameter to optimize and setting this derivative equal to zero. For the membership degrees we thus obtain the update rule

$$u_{ij} = \frac{1}{\sum_{k=1}^{c} \left(\frac{d^2(\vec{x}_j, \beta_i)}{d^2(\vec{x}_j, \beta_k)} \right)^{\frac{1}{m-1}}}.$$

This formula is independent of what kind of cluster prototype and what distance function is used. However, w.r.t. the update formulae for the cluster parameters we have to distinguish between algorithms that use different kinds of prototypes and, consequently, different distance functions.

The most common fuzzy clustering algorithm is the fuzzy c-means algorithm [Bezdek 1981]. It uses only cluster centers as prototypes, i.e., $\beta_i = (\vec{c}_i)$. Consequently, the distance function is

$$d_{\mathrm{fcm}}^2(\vec{x}_j, \beta_i) = (\vec{x}_j - \vec{c}_i)^T (\vec{x}_j - \vec{c}_i),$$

which leads to the update rule

$$\vec{c}_i = \frac{\sum_{j=1}^{n} u_{ij}^m \vec{x}_j}{\sum_{j=1}^{n} u_{ij}^m}$$

for the cluster centers. The fuzzy c-means algorithm searches for spherical clusters of equal size.

A more flexible variant is the Gustafson-Kessel algorithm [Gustafson and Kessel 1979]. It can find (hyper-)ellipsoidal clusters by extending the prototype with a fuzzy covariance matrix. That is, $\beta_i = (\vec{c}_i, \mathbf{C}_i)$, where $\mathbf{C}_i$ is the fuzzy covariance matrix, which is normalized to determinant 1 to unify the cluster size. The distance function is

$$d_{\mathrm{gk}}^2(\vec{x}_j, \beta_i) = (\vec{x}_j - \vec{c}_i)^T \mathbf{C}_i^{-1} (\vec{x}_j - \vec{c}_i).$$

This leads to the same update rule for the cluster centers as for the fuzzy c-means algorithm (see above). The covariance matrices are updated according to

$$\mathbf{C}_i = |\Sigma_i|^{-\frac{1}{p}} \Sigma_i,$$

where

$$\Sigma_i = \sum_{j=1}^{n} u_{ij}^m (\vec{x}_j - \vec{c}_i)(\vec{x}_j - \vec{c}_i)^T.$$

There is also a restricted version of the Gustafson-Kessel algorithm, in which only the variances of the individual dimensions are taken into account [Klawonn and Kruse 1995]. That is, this variant uses only the diagonal elements of the matrix Σ_i and all other matrix elements are set to zero. Since this means that the clusters are axes-parallel (hyper-)ellipsoids we refer to this variant as the axes-parallel Gustafson-Kessel algorithm.

In order to be able to apply the modifications reviewed above for neural networks, we simply identify the (negated) gradient $-\nabla_w e(t)$ w.r.t. a parameter w with the difference between two consecutive values of a cluster parameter (center coordinate or matrix element) that are computed with the standard update methods as we described them above. In this way all modifications of gradient descent we discussed in the preceding section can be applied.

Note that standard backpropagation also yields a modification, because we may introduce a learning rate, which influences the step width, into the clustering process as well. Of course, in fuzzy clustering this learning rate should be no less than 1, because a factor of 1 simply gives us the standard update step. That is, we only use this factor to expand the step width ($\eta \geq 1$), never to reduce it.

IV. Experimental Results

We tested the methods described above on four well-known datasets from the UCI machine learning repository [Blake and Merz 1998]: abalone (physical measurements of abalone clams), breast (Wisconsin breast cancer, tissue and cell measurements), iris (petal and sepal length and width of iris flowers), and wine (chemical analysis of Italian wines from different regions). In order to avoid scaling effects, the data was normalized, so that in each dimension the expected value was 0 and the standard deviation 1.

TABLE I
Best parameter values for the different datasets.

	FCM		a.p. GK		GK	
dataset	exp.	mom.	exp.	mom.	exp.	mom.
abalone 3	1.50	0.30	1.40	0.20	1.90	0.65
abalone 6	1.60	0.50	1.40	0.25	1.90	0.70
breast 2	1.20	0.05	1.30	0.10	1.80	0.55
iris 3	1.40	0.15	1.20	0.05	1.80	0.50
wine 3	1.40	0.10	1.50	0.15	1.80	0.60
wine 6	1.60	0.75	1.80	0.60	1.90	0.55

Since the listed datasets are originally classified (although we did not use the class information), we know the number of clusters to find (abalone: 3, breast: 2, iris: 3, wine: 3), so we ran the clustering using these numbers. In addition, we ran the algorithm with 6 clusters for the abalone and the wine dataset.

The clustering process was terminated when a normal update step changed no center coordinate by more than 10^{-6}. That is, regardless of the modification employed, we used the normal update step to define the termination criterion in order to make the results comparable. Note that for the Gustafson-Kessel algorithm (normal and axes-parallel version) we did not consider the change of the matrix elements for the termination criterion.

Most of the methods we explore here take parameters that influence their behavior. Our experiments showed that the exact values of these parameters are important only for the step width expansion and the momentum term, for which the best value seems to depend on the dataset. Therefore we ran the clustering algorithm for different values of the expansion factor and the momentum factor. In the following we report only the best results we obtained. The values we employed are listed in Table I. For the self-adaptive learning rate as well as the resilient approach we used a growth factor of 1.2 and a shrink factor of 0.5.

In order to remove the influence of the random initialization of the clusters, we ran each method 20 times for each dataset and averaged the number of iterations needed. The standard deviation of the individual results from these averages is fairly small, though. Note that the modifications only affect the recomputation of the cluster centers and not the distance computations for the data points, the latter of which accounts for the greater part of the computational costs of fuzzy clustering. Therefore the increase in computation time for one iteration is negligible and consequently it is justified to compare the different approaches by simply comparing the number of iterations needed to reach a given accuracy of the cluster parameters.

Table II shows the results for the fuzzy c-means algorithm. Each column shows the result for a method, each line the results for a dataset. The most striking observation to be made about this table is that the analogs of resilient backpropagation and quick backpropagation almost always *increase* the number of iterations needed. Step width expansion, momentum term, and self-adaptive learning rate,

TABLE II

Results for the fuzzy c-means algorithm.

dataset	std.	exp.	mom.	adp.	res.	qck.
abalone 3	46.2	31.9	27.5	28.8	56.9	56.0
abalone 6	114.8	72.8	53.1	64.6	87.6	153.2
breast 2	13.2	9.9	11.4	10.6	46.8	13.8
iris 3	26.3	17.8	18.1	16.1	48.1	27.9
wine 3	27.4	20.7	20.6	17.6	52.0	28.9
wine 6	490.2	299.4	112.8	259.4	306.9	428.2

TABLE III

Results for the axes-parallel Gustafson-Kessel algorithm.

dataset	std.	exp.	mom.	adp.	res.	qck.
abalone 3	51.2	38.0	39.3	30.8	75.7	59.0
abalone 6	140.6	110.0	127.6	82.4	143.2	158.1
breast 2	32.6	23.7	26.2	21.4	67.2	31.8
iris 3	26.6	21.9	27.3	18.9	74.1	29.9
wine 3	32.6	22.7	24.1	20.4	60.3	34.1
wine 6	406.0	226.8	173.4	215.8	288.3	370.2

TABLE IV

Results for the normal Gustafson-Kessel algorithm.

dataset	std.	exp.	mom.	adp.	res.	qck.
abalone 3	201.5	107.6	74.0	126.4	fail	fail
abalone 6	370.1	196.6	98.9	222.8	fail	fail
breast 2	163.1	84.2	54.5	74.8	144.5	122.2
iris 3	112.2	59.0	47.6	57.9	fail	238.6
wine 3	173.2	94.5	75.2	94.5	171.8	179.0
wine 6	320.2	175.4	143.2	170.8	320.1	301.4

however, yields very good results, sometimes even cutting down the number of iterations to less than half of what the standard algorithm needs. Judging from the numbers in this table, the momentum term approach and the self-adapting learning rate appear to perform best. However, it should be kept in mind that the best value of the momentum factor is not known in advance and that these results were obtained using the optimal value. Hence the adaptive learning rate, which does not need such tuning, has a clear edge over the momentum term approach.

Table III shows the results for the axes-parallel Gustafson-Kessel algorithm and Table IV the results for the normal Gustafson-Kessel algorithm. Here the picture is basically the same. The analogs of resilient and quick backpropagation are almost complete failures (the entry "fail" means that no stable clustering result could be reached even with 10000 iterations). Step width expansion, momentum term, and self-adaptive learning rate, however, yield very good results. Especially for the normal Gustafson-Kessel algorithm, the momentum term approach seems to be the clear winner, but again it has to be kept in mind that the optimal value of the momentum factor is not known in advance. Therefore we recommend the self-adaptive learning rate, which consistently leads to considerable improvements in the number of iterations.

The programs, the datasets, and a shellscript that have been used to carry out these experiments are available at

http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html

V. Conclusions

In this paper we explored how ideas from artificial neural network training can be used to speed up fuzzy clustering. Our experimental results show that there is indeed some potential for accelerating the clustering process, especially if a self-adaptive learning rate or a momentum term are used. With these approaches the number of iterations needed until convergence can sometimes be reduced to less than half the number of the standard approach. We recommend the self-adaptive learning rate, but the step width expansion and the momentum term approach also deserve strong interest, especially, because they are so simple to implement. Their drawback is, however, that the best value for their parameter depends heavily on the dataset.

References

[Anderson 1995] J.A. Anderson. *An Introduction to Neural Networks.* MIT Press, Cambridge, MA, USA 1995

[Bezdek 1981] J.C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms.* Plenum Press, New York, NY, USA 1981

[Bezdek *et al.* 1999] J.C. Bezdek, J. Keller, R. Krishnapuram, and N.R. Pal. *Fuzzy Models and Algorithms for Pattern Recognition and Image Processing.* Kluwer, Dordrecht, Netherlands 1999

[Bezdek and Pal 1992] J.C. Bezdek and S.K. Pal, eds. *Fuzzy Models for Pattern Recognition — Methods that Search for Structures in Data.* IEEE Press, Piscataway, NJ, USA 1992

[Blake and Merz 1998] C.L. Blake and C.J. Merz. UCI Repository of Machine Learning Databases. Department of Information and Computer Science, University of California, Irvine, CA, USA 1998. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[Fahlman 1988] S.E. Fahlman. An Empirical Study of Learning Speed in Backpropagation Networks. In: [Touretzky *et al.* 1988].

[Gustafson and Kessel 1979] E.E. Gustafson and W.C. Kessel. Fuzzy Clustering with a Fuzzy Covariance Matrix. *(IEEE CDC, San Diego, CA)*, pp. 761-766, IEEE Press, Piscataway, NJ, USA 1979

[Haykin 1994] S. Haykin. *Neural Networks — A Comprehensive Foundation.* Prentice-Hall, Upper Saddle River, NJ, USA 1994

[Höppner *et al.* 1999] F. Höppner, F. Klawonn, R. Kruse, and T. Runkler. *Fuzzy Cluster Analysis.* J. Wiley & Sons, Chichester, United Kingdom 1999

[Jakobs 1988] R.A. Jakobs. Increased Rates of Convergence Through Learning Rate Adaption. *Neural Networks* 1:295–307. Pergamon Press, Oxford, United Kingdom 1988

[Klawonn and Kruse 1995] F. Klawonn and R. Kruse. Automatic Generation of Fuzzy Controllers by Fuzzy Clustering. *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (Vancouver, Canada)*, 2040–2045. IEEE Press, Piscataway, NJ, USA 1995

[Riedmiller and Braun 1993] M. Riedmiller and H. Braun. A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. *Int. Conf. on Neural Networks (ICNN-93, San Francisco, CA)*, 586–591. IEEE Press, Piscataway, NJ, USA 1993

[Rumelhart *et al.* 1986b] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning Representations by Back-Propagating Errors. *Nature* 323:533–536. Nature Publishing Group, Basingstoke, United Kingdom 1986

[Tollenaere 1990] T. Tollenaere. SuperSAB: Fast Adaptive Backpropagation with Good Scaling Properties. *Neural Networks* 3:561–573. Pergamon Press, Oxford, United Kingdom 1990

[Touretzky *et al.* 1988] D. Touretzky, G. Hinton, and T. Sejnowski, eds. *Proc. Connectionist Models Summer School (Carnegie Mellon University, Pittsburgh, PA).* Morgan Kaufman, San Mateo, CA, USA 1988

[Zell 1994] A. Zell. *Simulation Neuronaler Netze.* Addison-Wesley, Bonn, Germany 1994