Efficient Levenberg-Marquardt Minimization of
the Cross-Entropy Error Function

Amar Sarić and Jing Xiao

Abstract — The Levenberg-Marquardt algorithm is one of the most common choices for training medium-size artificial neural networks. Since it was designed to solve nonlinear least-squares problems, its applications to the training of neural networks have so far typically amounted to using simple regression even for classification tasks. However, in this case the cross-entropy function, which corresponds to the maximum likelihood estimate of the network weights when the sigmoid or softmax activation function is used in the output layer, is the natural choice of the error function and a convex function of the weights in the output layer. It is an important property which leads to a more robust convergence of any descent-based training method. By constructing and implementing a modified version of the Levenberg-Marquardt algorithm suitable for minimizing the cross-entropy function, we aim to close a gap in the existing literature on neural networks. Additionally, as using the cross-entropy error measure results in one single error value per training pattern, our approach results in lower memory requirements for multi-valued classification problems when compared to the direct application of the algorithm.

I. INTRODUCTION

Gradient methods exploit the linear approximation of the function in the neighborhood of the current estimate. For instance, the training of neural networks using least-squares is done by calculating the derivative \( \frac{\partial E(w)}{\partial w} \) of the sum of squared errors \( E(w) = \sum (y_i - f(w,x_i))^2 \), where \( f(w,x_i) \) is the function that we wish to fit to the data, \( y_i \) stands for the observed value associated with the input vector \( x_i \) and \( w \) for the vector of parameters passed to the function. The primary advantage of the gradient descent is its simplicity and applicability to error functions other than least-squares, most notably the cross entropy function. However, it is known for its slow convergence and quite often an outright failure on more complicated functions. On the other hand, the fundamental insight of Marquardt’s [2] approach is not to try to linearize the sum of squares by calculating \( \frac{\partial E(w)}{\partial w} \), but to compute the function \( f(w,x_i) \) itself and its derivative \( \frac{\partial f(w,x_i)}{\partial w} \), and then solve the linear least-squares problem

\[
\min_{\Delta w} \| y - f(w,x) - \left( \frac{\partial f(w,x)}{\partial w} \right) \Delta w \|^2,
\]

instead of the much harder task of finding a solution to the non-linear problem \( \min_w \| y - f(w,x) \|^2 \) directly. Here, the entries of the matrix \( x \) consisting of vectors \( x_i \), and \( f(w,x) \) and its Jacobian, are so defined that they match the corresponding values \( y_i \) of the vector \( y \) for every \( x_i \). At each step of the algorithm the initial estimate of \( w \) is updated by solving the linear least-squares problem. Although it uses only first order derivatives, the algorithm typically performs as well as the second order methods. It is a common choice for training medium-size neural networks (see [6]). The main disadvantage of this approach is that it uses regression instead of the cross-entropy minimization for classification tasks. In this paper we show how the Levenberg-Marquardt algorithm can be adapted to be used for classification tasks based on cross-entropy minimization.

II. AN OVERVIEW OF THE LEVENBERG-MARQUARDT ALGORITHM FOR NONLINEAR LEAST-SQUARES

Let us first examine the linear algebra tools needed to solve the simplified problem (1). There are two common ways to approach the linear least-squares problem for overdetermined systems \( Ax = b \), where \( A \) is an \( m \times n \) matrix with \( m \geq n \) and \( b \) the \( m \) dimensional vector of residuals. One method consists of setting all partial derivatives to zero and solving the set of normal equations \( A'Ax = A'b \). The other, more numerically stable, method uses the QR decomposition and applies a series of transformations to \( \| Ax - b \| \), each one equivalent to the multiplication of \( A \) and \( b \) by an orthonormal matrix, which column by column eliminate the values in the entries below the diagonal, thus enabling us to eventually solve the resulting system of \( n \) linear equations by back-substitution. Here, the Euclidean norm is calculated over all training instances. The method works because the sum of squares is preserved by every transformation. Usually Householder reflections or Givens rotations are used to accomplish the multiplication by the orthonormal matrices, see for instance [13], which set the entries in a given column vector to zero.

In order to make it worthwhile to solve a complete linear least-squares problem at every iteration of the Levenberg-Marquardt algorithm, it becomes essential to determine as large a vector \( \Delta w \) as possible for which such a linearization leads to a lower value of the norm in (1). Thus, the optimization at every step has to be constrained, while the radius of \( \Delta w \) should be adapted. The proper insight can be gained by writing out the normal equations of the linearized system. In order to do this we write out the norm in the minimization problem (1), calculate its partial derivatives and set them to zero.
the linear approximation to \( f(w,x) \) is valid. A good estimate for \( \lambda \) can be found by increasing or decreasing its value based on whether there was an improvement resulting from the last update and retracting the unsuccessful steps. Since \( P^T r = P^T (y - f(w,x)) \) is actually the negative gradient of \( \sum_j (y_j - f(w,x_j))^2 \) we resort to gradient descent for large \( \lambda \), as the parameter update vector \( \Delta w \) approaches the direction of the negative gradient. Therefore, see also Levenberg [1], one can interpret this formula as smoothly varying between the unconstrained Gauss-Newton algorithm when \( \lambda \) is small and the simple gradient descent with small step size for a large \( \lambda \), in which case the convergence, albeit slow, is guaranteed. Marquardt [2] gives a rigorous proof of these facts. In order to be able to use the QR decomposition for least-squares minimization, we transform the normal equations \( P^T P + \lambda I \Delta w = P^T r \) one more time (see Moré [3]) by noting that the system is equivalent to

\[
\begin{pmatrix} P, \sqrt{\lambda I} \end{pmatrix}^{\top} \begin{pmatrix} P \sqrt{\lambda I} \end{pmatrix} \Delta w = \begin{pmatrix} P, \sqrt{\lambda I} \end{pmatrix} \begin{pmatrix} r \end{pmatrix},
\]

which are exactly the normal equations of the linear least-squares problem \( \min_w \left\| \begin{pmatrix} P \sqrt{\lambda I} \end{pmatrix} \Delta w - \begin{pmatrix} r \end{pmatrix} \right\| \). Consequently the implementation of the algorithm can be done as follows:

Set \( \lambda = 0.01 \), flag = true
Repeat until convergence criterion is met (e.g. LMS < threshold, number of iteration exceeded, etc.)

Step 1: If flag is true calculate the Jacobian \( P \) and residuals \( r \)
Step 2: Create temporary copies of \( P \) and \( r \)
Step 3: Using QR decomposition solve the problem
\[
\min_w \left\| \begin{pmatrix} P \sqrt{\lambda I} \end{pmatrix} \Delta w - \begin{pmatrix} r \end{pmatrix} \right\| \text{ on the copies.}
\]
Step 4: Test new weight values \( w + \Delta w \) to see if there is an improvement (i.e. reduction in norm).
Step 5: If there is an improvement
\[
\text{set } \lambda = 0.1\lambda \text{ and flag = true}\]
Else
\[
\text{set } \lambda = 10\lambda \text{ and flag = false}
\]
For best performance, the implementation should make use of optimized BLAS libraries and LAPACK (see [12] and [9]), especially on larger datasets.

III. THE CROSS-ENTROPY ERROR FUNCTION
The natural error measure for classification problems is the cross-entropy function. For mutually exclusive classes, let
\( o_j(w,x) \) denote the \( j \text{th} \) output of a classifier for the pattern \( x \), and \( \delta_j \), where \( \delta_j = 1 \) if \( i = j \) and 0 otherwise, indicate that the \( j \text{th} \) class is the observed outcome. If we interpret the \( j \text{th} \) output of the classifier as the probability that a particular data instance is in the \( j \text{th} \) class, then, according to the classifier, the probability of observing the correct outcomes for the whole dataset is \( \prod_{x \in \text{test}} o_j(w,x) \). This quantity has to be maximized. The cross-entropy error function is defined as being equal to the negative logarithm of the expression and can be minimized instead. Therefore, we obtain

\[
-\sum_{x \in \text{test}} \log o_j(w,x) \quad \text{or equivalently} \quad -\sum_{x \in \text{test}} \sum_i \delta_j \log o_j(w,x).
\]

(8)

Now, assuming softmax as the activation function

\[
o_j(w,x) = \frac{\exp(\text{sum}_j(w,x))}{\sum_k \exp(\text{sum}_k(w,x))},
\]

(9)

Where the \( \text{sum}_k(x_n) \) denotes the weighted sum of the neural network inputs for the \( k \text{th} \) output, the cross entropy can be rewritten in its more familiar form

\[
-\sum_{x \in \text{test}} \sum_i \delta_j (x) \log \left( \sum_k \exp(\text{sum}_k(w,x)) \right) =
\]

\[
-\sum_{x \in \text{test}} \left( \sum_i \delta_j (x) \text{sum}_j(w,x) - \log \left( \sum_k \exp(\text{sum}_k(w,x)) \right) \right) =
\]

\[
\sum_{x \in \text{test}} \left[ \log \left( 1 + \sum_{k \neq l} \exp(\text{sum}_k(w,x) - \text{sum}_l(w,x)) \right) \right.
\]

\[
-\sum_{i \neq j} \delta_j (x) (\text{sum}_j(w,x) - \text{sum}_i(w,x))
\]

\[
\sum_{x \in \text{test}} \left[ \log \left( \sum_k \exp(\text{sum}_k(w,x)) \right) \right]
\]

(10)

where \( l \) is the index corresponding to \( \max_k(\text{sum}_k(w,x)) \).

The last formula is useful in avoiding numerical overflow. The derivative of the expression, on the other hand, is rather simple

\[
\frac{\partial}{\partial \text{sum}_j(w,x)} \left( -\sum_{x \in \text{test}} \left( \sum_i \delta_j (x) \text{sum}_j(w,x) - \log \left( \sum_k \exp(\text{sum}_k(w,x)) \right) \right) \right)
\]

\[
= -\sum_{x \in \text{test}} \left( \delta_j (x) - \frac{\exp(\text{sum}_j(w,x))}{\sum_k \exp(\text{sum}_k(w,x))} \right)
\]

\[
= -\sum_{x \in \text{test}} \left( \delta_j (x) - o_j(w,x) \right).
\]

Moreover, it is well known that \( \log \left( \sum_k \exp(u_k) \right) \) is a convex function of \( u_k \). To see this we calculate its Hessian

\[
H = -\frac{1}{\left( \sum_k \exp(u_k) \right)^2} \begin{pmatrix} \exp(2u_1) & \exp(u_1 + u_n) \\ \exp(u_n + u_1) & \exp(2u_n) \end{pmatrix}
\]

\[
+ \frac{1}{\sum_k \exp(u_k)} \begin{pmatrix} \exp(u_1) & 0 & 0 \\ 0 & \cdots & 0 \\ 0 & 0 & \exp(u_n) \end{pmatrix}
\]

(12)

and rely on the Cauchy-Schwartz inequality to arrive at

\[
v^T Hv = \left( \sum_k \exp(u_k) \right) \left( \sum_k v_k \exp(u_k) \right) - \left( \sum_k v_k \exp(u_k) \right)^2
\]

\[
\geq 0,
\]

(13)

by considering the vectors \( \left( \sqrt{\exp(u_1)}, \ldots, \sqrt{\exp(u_n)} \right)^T \) and \( \left( v_1 \sqrt{\exp(u_1)}, \ldots, v_n \sqrt{\exp(u_n)} \right)^T \) (see [11]). Therefore, the cross entropy is also a convex function. This is important for any descent method including the gradient descent and the related approaches. Unfortunately, convexity does not hold for the weights in the hidden layer(s) of a feed forward neural network, only for those in the output layer. In the case of the hidden layers, nonlinear activation functions from the subsequent layers interfere and the error surface is no longer convex. Consequently, it is much harder to find the global minimum, i.e. the most likely estimate for the values of the network weights.

IV. LEVENBERG-MARQUARDT ALGORITHM FOR CROSS-ENTROPY MINIMIZATION

As previously noted, the main disadvantage of the Levenberg-Marquardt algorithm is that it cannot be used for cross-entropy minimization in its original form, and in the rest of the paper our focus will be on addressing this limitation. If we take a look at the definition of cross-entropy, there is a rather straightforward way to rewrite the expression...
\[
\sum_{x \in \text{data}} f(w,x)^2 = -\sum_{x \in \text{data}} \left( \log \left( \sum_k \exp(\text{sum}_k(w,x)) \right) - \frac{\delta_y(x)\text{sum}_j(w,x)}{\text{sum}_j(w,x)} \right)
\]

\[f(w,x) = \sqrt{\left( \sum_i \delta_y(x)\text{sum}_j(w,x) - \log \left( \sum_k \exp(\text{sum}_k(w,x)) \right) \right)}
\]

to make it amenable to optimization by Gauss-Newton and hence also the Levenberg-Marquardt algorithm – we can simply fit the function

\[
\alpha_{j,k} = \frac{o_j(w,x_k) - \delta_y(x_k)}{\sqrt{-\log \left( \sum_i \delta_y(x_i) o_j(w,x_i) \right)}}
\]

and

\[
r_{c}(w,x) = \sqrt{-\log \left( \sum_i \delta_y(x_i) o_j(w,x_i) \right)}
\]

For the weights in the hidden layers of a multilayer feedforward neural network the matrix entries of \(2P_c(w,x)\) can be calculated by a modified backpropagation algorithm, in which \(\frac{o_j(w,x) - \delta_y(x)}{\sqrt{-\log(o_{\text{correct}}(w,x))}}\) is used as the error term of the output \(o_j(x)\). In a single layer network \(\frac{\partial \text{sum}_j(w,x)}{\partial w_i}\) is equal to the \(l^{th}\) input of the \(j^{th}\) output unit

\[
\sum_{j \text{out}} \frac{o_j(w,x) - \delta_y(x)}{\sqrt{-\log \left( \sum_i \delta_y(x) o_j(w,x) \right)}} \left( l^{th} \text{ input} \right)
\]

In the case of a perceptron-like single layer network there will be no change in indexing. The matrix \(P_c(w,x)\) is subsequently formed by appending the row vectors. The back-propagation formulas can be given explicitly for multilayer, feed-forward networks with a sigmoid activation function, but the indices have to be rearranged into a single row vector to form the Jacobian matrix. This can be done by copying the back-propagated gradient values in some predetermined way, say layer by layer starting with the output layer, into a single memory array. It is necessary in order to be able to use linear algebra methods and LAPACK software package ([9]) to solve the linear squares problem.

The formula for the output layer is

\[
\sum_{j \text{out}} \Delta_j(x) \left( l^{th} \text{ input} \right), \quad \Delta_j(x) = \frac{o_j(w,x) - \delta_y(x)}{\sqrt{-\log \left( \sum_i \delta_y(x) o_j(w,x) \right)}}
\]

The values are back-propagated by applying the update formula

\[
\Delta(x) = o(w,x)(1-o(w,x)) \\
\cdot \sum_{\text{last layer}} \Delta_i(x) \cdot \text{(connecting weight)}
\]
and the gradient for the node in question is calculated as 
\[ \Delta(x) \] (corresponding input). We intentionally do not use \( w \) to denote the weights since the indexing (and depending on implementation possibly also the whole memory layout) are not identical. Obviously, in order to calculate the outputs using the updated weights the new values have to be moved from the array back into the memory layout used for the network (i.e. layer by layer if we chose to do so previously). One possible C/C++ implementation is to assign a single large chunk of contiguous memory for all weights needed to store the neural network and treat the pointer to that array as \( w \), and thus avoid copying. Single arrays needed to organize the network into separate layers correspond in this case to fixed offset values from the pointer \( w \).

Alternatively, and partially motivated by the least-squares approach of Wilamowski et. al. [5], instead of trying to minimize the cross-entropy function we can turn our attention to the square of the cross-entropy and sum over the training instances as follows

\[
\sum_{s \in \text{data}} g(w,x)^2 = \sum_{s \in \text{data}} \left( \log \left( \frac{1}{\sum \exp(s(x))} \right) \right)^2
\]

where \( g(x) \) is simply the cross-entropy function for the training pattern \( x \). The function is obviously again convex. Furthermore, one obtains the entries of the Jacobian matrix \( P_s(w,x) \) as

\[
P_s(w,x) = \frac{\partial}{\partial \text{sum}_j(x)} g(x) = o_j(x) - \delta_j.
\]

Hence,

\[
P_s(w,x) = \begin{bmatrix}
\sum_{j \in \text{out}} \beta_{j,1} \frac{\partial \text{sum}_j(w,x_1)}{\partial w_1} & \ldots & \sum_{j \in \text{out}} \beta_{j,1} \frac{\partial \text{sum}_j(w,x_n)}{\partial w_1} \\
\sum_{j \in \text{out}} \beta_{j,2} \frac{\partial \text{sum}_j(w,x_1)}{\partial w_2} & \ldots & \sum_{j \in \text{out}} \beta_{j,2} \frac{\partial \text{sum}_j(w,x_n)}{\partial w_2} \\
\sum_{j \in \text{out}} \beta_{j,m} \frac{\partial \text{sum}_j(w,x_1)}{\partial w_m} & \ldots & \sum_{j \in \text{out}} \beta_{j,m} \frac{\partial \text{sum}_j(w,x_n)}{\partial w_m}
\end{bmatrix}
\]

where

\[
\beta_{j,k} = o_j(x_k) - \delta_j(x_k)
\]

and the residue vector is given by

\[
r_s(w,x) = \begin{bmatrix}
\log \left( \sum_i \delta_i(x) o_j(w,x_i) \right) \\
\ldots \\
\log \left( \sum_i \delta_i(x_m) o_j(w,x_m) \right)
\end{bmatrix}
\]

Therefore, the linear problem to be solved by the QR decomposition at every step of the Levenberg-Marquardt algorithm becomes

\[
\min_{\Delta w} \left\| P_s(w,x) \Delta w - \begin{bmatrix} r_s(w,x) \\ 0 \end{bmatrix} \right\|.
\]

Note that we arrive at almost exactly the same minimization problem as previously, except that the equations have been rescaled by the square root of the negative logarithm of the correct output \( \frac{1}{2} \log \left( \sum_i \delta_i(w) o_j(w,x) \right) \), and that the factor \( \frac{1}{2} \) is missing. This suggests removing the factor from the cross-entropy minimization formulas (18) as well, especially since the direction of the update does not change for \( \lambda = 0 \), only the step size decreases. An alternative way to think of it is that we can minimize the cross-entropy itself by rescaling the equations given in (24) or, equivalently, by solving

\[
\min_{\Delta w} \left\| 2P_s(w,x) \Delta w - \begin{bmatrix} r_s(w,x) \end{bmatrix} \right\|
\]

using the QR decomposition. This is our preferred formula. In order to motivate it further, we will give another justification for multiplying the Jacobian matrix by 2 in terms of the location of the minimum obtained by the linear approximations. If we restrict ourselves to the special case of a binary classification task for which the target value is equal to 0, then for a single training instance we can explicitly calculate the location of the minimum of the functions

\[
l_i(w) = \frac{\sqrt{\log(1 + \exp(\text{sum}(w_i)))}}{2(1 + \exp(\text{sum}(w_i)))} \log(1 + \exp(\text{sum}(w_i)))
\]

and

\[
l_i(\text{sum}(w)) = \frac{\sqrt{\log(1 + \exp(\text{sum}(w)))}}{1 + \exp(\text{sum}(w))} \log(1 + \exp(\text{sum}(w)))
\]

which are the squares of the linear estimates of \( \sqrt{\log(1 + \exp(\text{sum}(w_i)))} \) – the square root of the cross-entropy function – around the point \( \text{sum}(w_i) \). Here \( l_i(w) \) corresponds to the formulas (16) – (18) and \( l_i(\text{sum}(w)) \) to the modified update rule which uses \( 2P \) instead of \( P \), as in the formula (25). The resulting approximations are displayed in fig. 1 and fig. 2. We see that a better local approximation, even better than that resulting from the second order Taylor series, is obtained for \( l_i(w) \) around both \( \text{sum}(w_i) = 0 \) and \( \text{sum}(w_i) = 10 \). However, we also see that the update suggested by this formula takes the value of the linear sum
the farther into the negative region, the larger the margin by which the point was previously misclassified. On the other hand the formula $l_2(w)$ results in a smaller absolute value of the linear sum of the outputs. The exact solution can be obtained by taking the derivative of the expressions and solving for $\text{sum}(w)$. It is displayed in the plot in fig. 3. The minimum corresponding to formula $l_1(w)$ is plotted against the value of the linear sum of the output using the dashed line, and that of $l_2(w)$ using the solid line. We observe that by minimizing $l_2(w)$ we are in fact attempting to make the linear sum negative and small in magnitude for the points which are currently misclassified by a large margin, whereas the formula $l_1(w)$ is less predictable, since it will try to push the value of the linear sum into the negative region of the linear classifier by a magnitude which is roughly proportional to the misclassification margin of the data point. Although the formula $l_1(w)$ better approximates the function locally, such behavior can ruin the convergence. The total update is calculated as the sum of errors of individual training patterns. Therefore, $l_2(w)$ is clearly preferable. In this paper we will not attempt to give a theoretical justification for more than two classes and outputs; it is left as an open research question.

Based on the above considerations, we propose that the formulas (25), i.e. the linearization corresponding to $l_1(w)$, be used as the standard method. Additionally, we obtain
good perceptron/adaline training method, since the Levenberg-Marquardt algorithm is known for its fast convergence, and the convexity of the error function for all weights enables us to find the global minimum.

V. NUMERICAL RESULTS

We shall demonstrate the performance of the training procedure on two commonly used datasets – the Wisconsin Diagnostic Breast Cancer (WDBC) and the letter-recognition problem from the Machine Learning Repository of the University of Irvine.

The WDBC data is known to be linearly separable and the classifier can therefore be implemented without a hidden layer. Hence the minimization problem is convex and a good indicator of performance of the training procedure in the case of a perceptron/adaline network for the binary classification task. Approximately the same number of training iterations is necessary for the proposed formula (25) based on minimizing the sum of squared entropy values (24). The algorithm does indeed exhibit excellent convergence properties, as seen in fig. 4, which shows the improvement in classification accuracy on the training set against the number of the iterations (i.e. training epochs).

![Fig. 4. Training of a simple perceptron on the Wisconsin Diagnostic Breast Cancer dataset](image)

Difficulties occur, however, with the unmodified formula (18) in which the factor $\frac{1}{2}$ has been kept. Although it can be used to train the adaline network the convergence suffers, as could be expected based on our previous considerations for binary classification. The fig. 4 shows the accuracy of the trained classifier on the training set after every training epoch. We see that the unmodified formula oscillates in an erratic manner. Moreover, several restarts of the algorithm might be needed for a successful run. Also, it should be noted that the number of classification errors itself is not used as the minimization criterion, but rather the cross-entropy (or alternatively sum of squared entropy values), which is steadily decreasing during the whole training process for all three cases as expected.

The letter-recognition classification task demonstrates the numerical convergence properties of our training method for a multi-class problem, which turn out to be surprisingly good even after the introduction of a hidden layer and the resulting non-convexity. The problem consists of 26 classes (one for each letter). The plot in fig. 5 was obtained for 100 hidden layer units.

![Fig. 5. Letter-recognition task. 26 classes, 100 hidden units, 20000 training examples](image)

We see that the unmodified formula (18), which features the factor $\frac{1}{2}$, performs poorly. It also requires several restarts to produce reasonable classification accuracy. On the other hand, we were quite pleased with both the effectiveness and the convergence properties of the proposed formula (25). Halving the step size is justified since our numerical tests indicate that it has even more of an impact for multi-valued classification problems. Alternatively, one can also use the sum of squares of entropy values as the performance function (24). The main disadvantage is the increase in the computational time which is proportional to the time necessary to solve the linear least-squares problem. This is typical of the Levenberg-Marquardt algorithm – whereas the training can be performed in the matter of seconds for the
WDBC dataset, an optimized BLAS library is necessary for the letter-recognition dataset containing 20000 instances.

A deeper insight of the optimization process itself can be gained by looking at the cross-entropy values after every training epoch as shown in the fig. 6. The algorithm needs to adjust the parameter $\lambda$ when there is no improvement. The dashed line indicates the cross-entropy of the point considered at that particular iteration (i.e. training epoch), which is then used to update the current solution only if it results in a decrease of the cross-entropy error measure. A good discussion of the issues involved is given by Lampton [7], but there is generally little to be gained by a more elaborate search mechanism. An important point is that multiplicative damping does not protect against a singular Jacobian matrix. Consequently, additive damping has been our preferred choice throughout. It is important to initially choose a small $\lambda$. Here 0.01 or 0.001 is appropriate in most cases.

Moreover, both formulas aggregate the outputs of the neural network for different classes into one error value per training instance and reduce the memory usage for multiclass problems as compared to the standard training procedure which uses Levenberg-Marquardt algorithm and the sum of squared errors (e.g. [4]). In our approach the number of classes does not affect the memory requirements, which makes it comparable in this respect to that of Wilamowski et. al. ([5]). For perceptron/adaline the convexity of the error function results in stable convergence. Even with the non-convexity introduced by an additional hidden layer(s) the proposed algorithm offers an effective training mechanism. Additional research is, of course, needed to fully examine the performance of the formulas, however, there are theoretical reasons for the algorithm to converge to the global minimum for single layer neural networks, and to outperform the gradient descent based algorithms for networks with hidden layers. The limitation of our training method, and that of the related approaches (see [4], [5], and [6]), is that Levenberg-Marquardt algorithm requires a least-squares solution to the set of linear equations resulting from the individual training instances at each iteration, which is an issue for large networks and training sets.

REFERENCES


![Fig. 6. Entropy minimization using the proposed formula for the letter-recognition dataset.](image)

The plot also illustrates how the algorithm defensively explores the parameter space.

VI. CONCLUSION

We have shown how the Levenberg-Marquardt algorithm can be successfully modified in order to train neural networks using the cross-entropy error function, or, alternatively, using the sum of squares of the entropy values.