Brief Papers

Improving Generalization Performance Using Double Backpropagation

Harris Drucker and Yann Le Cun

Abstract—**In order to generalize from a training set to a test set, it is desirable that small changes in the input space of a pattern do not change the output components. This can be done by including variations of the input space as part of the training set—but this is computationally very expensive. Another method is to force this behavior as part of the training algorithm. This is done in double backpropagation by forming an energy function that is the sum of the normal energy term found in backpropagation and an additional term that is a function of the Jacobian. Significant improvement is shown with different architectures and different test sets, especially with architectures that had previously been shown to have very good performance when trained using backpropagation. It also will be shown that double backpropagation, as compared to backpropagation, creates weights that are smaller thereby causing the output of the neurons to spend more time in the linear region.**

I. Introduction

BACKPROPAGATION [1] has been a popular supervised training algorithm for a number of years. The general procedure is to learn on a training set and see how well the generalization is on a test set. In an attempt to improve generalization performance, one can create different architecture, or for a specific architecture one can impose additional constraints. Two of the latter techniques are weights decay [2]–[4] and training with noise [3], [5], [6]. The normal energy term in backpropagation, here denoted as E_f (f for forward) is of the form

$$E_f = \frac{1}{2}\sum_{j=1}^{m}(d_j - x_j)^2 = \frac{1}{2}(\boldsymbol{D}-\boldsymbol{X})(\boldsymbol{D}-\boldsymbol{X})^t \quad (1)$$

where t indicates transpose, x_j is the jth component of the output (total m output components) and $\boldsymbol{D}$ and $\boldsymbol{X}$ are the desired and actual output row matrices, respectively.

In weight decay we add to E_f a term of the form $\alpha\|\boldsymbol{W}\|^2$, i.e., a constant times the norm squared of the weight vector. The idea here is to force the weights to be small therefore keeping the output of the nonlinear elements out of the saturated regions. Although weight decay does work, it is not always obvious in advance what the value of α should be. Therefore, it usually requires multiple runs to determine α. The rationale of adding noise to the input patterns is to move the search out of a local minimum of the weight space in addition to providing variations of the input. The problem with this approach is to determine how much noise should be added thereby requiring multiple runs to determine the proper amount of noise. Generalization may be increased by picking appropriate architectures or starting with an architecture and then pruning or adding units [7]–[9].

Manuscript received June 11, 1991; revised January 3, 1992.

H. Drucker is with AT&T Bell Laboratories and Monmouth College, Monmouth College, West Long Branch, NJ 07764.

Y. Le Cun is with AT&T Bell Laboratories, Room 3G-332, Crawford Corners Road, Holmdel, NJ 07733.

IEEE Log Number 9106865.

II. Double Backpropagation

In double backpropagation, in addition to the normal energy term E_f we additionally try to minimize a term of the form:

$$E_b = \frac{1}{2}\left(\frac{\partial Ef}{\partial i_1}\right)^2 + \frac{1}{2}\left(\frac{\partial Ef}{\partial i_2}\right)^2 + \cdots \frac{1}{2}\left(\frac{\partial Ef}{\partial i_n}\right)^2$$

where i_j refers to the jth (of a total n) input component. The rationale for this approach is that if the input changes slightly the energy function E_f should not change. One measure of this change is just the derivative of E_f with respect to all the inputs. Therefore, by forcing E_b to be small we force all of the appropriate derivatives to be small.

Let us calculate $(\partial E_f/\partial i_1)$:

$$\frac{\partial E_f}{\partial i_1} = -\left((d_1 - x_1)\frac{\partial x_1}{\partial i_1} + (d_2 - x_2)\frac{\partial x_2}{\partial i_1} + \cdots + (d_m - x_m)\frac{\partial x_m}{\partial i_1}\right) = -(\boldsymbol{D}-\boldsymbol{X})\begin{bmatrix}\frac{\partial x_1}{\partial i_1}\\ \frac{\partial x_2}{\partial i_1}\\ \vdots \\ \frac{\partial x_m}{\partial i_1}\end{bmatrix} = -(\boldsymbol{D}-\boldsymbol{X})\boldsymbol{J}\boldsymbol{C}_1$$

where $\boldsymbol{J}$ is the m by n Jacobian matrix (corresponding to the m output components and n input components) and $\boldsymbol{C}_1$ is a column vector of row size n with unity in the first row but zero, otherwise. Let us use $\boldsymbol{C}_j$ as the column vector of row size n that is zero except for unity in row j.

We now need as the first term of E_b:

$$\left(\frac{\partial E_f}{\partial i_1}\right)^2 = (\boldsymbol{D}-\boldsymbol{X})\boldsymbol{J}\boldsymbol{C}_1\boldsymbol{C}_1{}^t\boldsymbol{J}^t(\boldsymbol{D}-\boldsymbol{X})^t$$

$(\partial E_f/\partial i_j)$ can be calculated similarly except that $\boldsymbol{C}_1$ is replaced with $\boldsymbol{C}_j$.

Therefore

$$E_b = (\boldsymbol{D}-\boldsymbol{X})\boldsymbol{J}[\boldsymbol{C}_1\boldsymbol{C}_1^t + \boldsymbol{C}_2\boldsymbol{C}_2^t + \ldots + \boldsymbol{C}_n\boldsymbol{C}_n^t]\boldsymbol{J}^t(\boldsymbol{D}-\boldsymbol{X})^t.$$

The term in brackets reduces to the identity matrix and therefore we obtain:

$$E_b = (\boldsymbol{D}-\boldsymbol{X})\boldsymbol{J}\boldsymbol{J}^t(\boldsymbol{D}-\boldsymbol{X})^t.$$

If we now add the normal energy term to this last term we obtain for the total energy term E_t:

$$E_t = \frac{1}{2}(\boldsymbol{D}-\boldsymbol{X})(\boldsymbol{D}-\boldsymbol{X}) + \alpha(\boldsymbol{D}-\boldsymbol{X})\boldsymbol{J}\boldsymbol{J}^t(\boldsymbol{D}-\boldsymbol{X})^t$$
$$= (\boldsymbol{D}-\boldsymbol{X})\left(\frac{1}{2}\boldsymbol{I}_m + \alpha\boldsymbol{J}\boldsymbol{J}^t\right)(\boldsymbol{D}-\boldsymbol{X})^t$$

where $\boldsymbol{I}_m$ is the identity matrix of square size m and α is a multiplicative constant which will be related to the learning rate of the neural network. The constant α is greater than one, the rationale being that near the minimum, $(\boldsymbol{D}-\boldsymbol{X})(\boldsymbol{D}-\boldsymbol{X})^t$ is close to zero and hence $\boldsymbol{J}\boldsymbol{J}^t$ will have small effect near this minimum unless α is large. In practice the optimum solution is fairly insensitive to the choice of α and no time is spent searching for the optimum α. α is related to the choice of learning constant in the neural network.

One of the key ideas in double backpropagation is that the additional energy term E_b can be calculated in one backward pass through a neural network or one forward pass through what we will call an appended network. Let us examine a simple architecture (Fig. 1—below the dashed horizontal line) with three input neurons, two hidden-layer neurons, and two output neurons. The nonlinearity is the hyperbolic tangent saturating at ± 1.7. These values are chosen so that the outputs are ± 1 and the extremes of the second derivative take place when the inputs are ± 1. The targets are always ± 1. The layers are fully connected (to the layer immediately above) but not all weights are shown (nor is the bias). We use letters (rather than the typical weight notation with superscripts and subscripts) to denote the weights. The input layer has linear neurons while the other layers have the nonlinear transfer function. Therefore $a_j^{(r)}$, represents the summed input of neuron number j at layer r and $x_j^{(r)} = f\left(a_j^{(r)}\right)$ where f is the hyperbolic tangent.

In this case the forward energy function is:

$$E_f = \frac{1}{2}(d_1 - x_1^{(2)})^2 + \frac{1}{2}(d_2 - x_2^{(2)})^2. \tag{1}$$

We call this the forward energy function because it is obtained by propagating the states forward through the network. Now, let us look at some of the terms obtained by backpropagating through the network.

First, the derivative of the forward energy function with respect to the input of the first neuron of the output layer:

$$-\frac{\partial E_f}{\partial a_1^{(2)}} = \left(d_1 - x_1^{(2)}\right)f'\left(a_1^{(2)}\right). \tag{1}$$

where f' is the derivative. Next, the derivatives of the forward energy function with respect to the input of both neurons at the hidden layer:

$$\frac{\partial E_f}{\partial a_1^{(1)}} = \left[A\frac{\partial E_f}{\partial a_1^{(2)}} + C\frac{\partial E_f}{\partial a_2^{(2)}}\right]f'\left(a_1^{(1)}\right)$$
$$\frac{\partial E_f}{\partial a_2^{(1)}} = \left[B\frac{\partial E_f}{\partial a_1^{(2)}} + D\frac{\partial E_f}{\partial a_2^{(2)}}\right]f'\left(a_2^{(1)}\right)$$

The change in weight A is now due to the term $-(\partial E_f/\partial a_1^{(2)})$ multiplied by both $x_1^{(2)}$ and the learning constant. Similarly we can update the other weights. In normal backpropagation, these are all the gradients needed. However, we can proceed one more step and calculate the derivative of the forward energy function with respect to the input (recalling that the input neurons are linear):

$$-\frac{\partial E_f}{\partial i_1} = \left[E\frac{\partial E_f}{\partial a_1^{(1)}} + F\frac{\partial E_f}{\partial a_2^{(1)}}\right].$$

Terms of the form $(\partial E_f/\partial i_j)$, where j ranges over the n input components, are called the *input gradients*. It is important to note that the calculation of the input gradients is a linear operation. We now show that the input gradients can be calculated by appending a network (the top part of Fig. 1) to the original network.

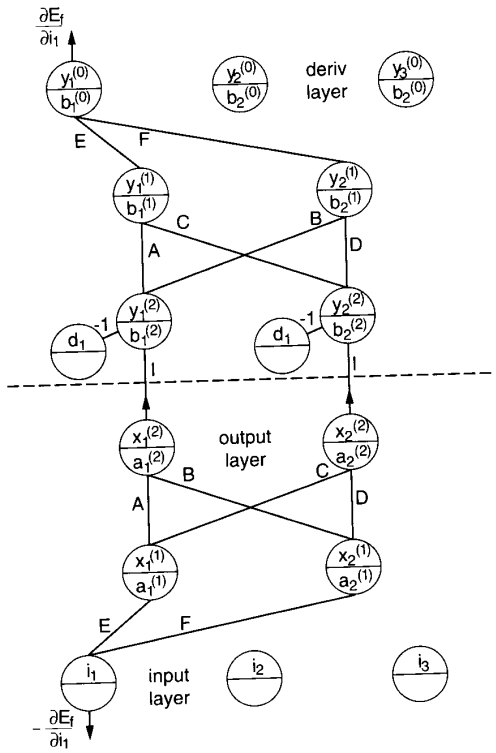

Fig. 1. Backpropagation (below dashed line) and double backpropagation (entire network).

The appended network, which is a "mirror image" about the dashed line, uses linear neurons:

$$y_j^{(r)} = k_j^{(r)} b_j^{(r)}$$
$$k_j^{(r)} = \begin{cases} f'\left(a_j^{(r)}\right) & r > 0 \\ 1 & r = 0 \end{cases}$$

where k is the multiplicative constant whose value is related to the derivative of the input state of a neuron in the lower network. Note that the superscripts decrease as one approaches the top of the appended layer.

Although not all weights are shown consider the input state of the first neuron of the appended layer:

$$b_1^{(2)} = \left(x_1^{(2)} - d_1\right)$$
$$y_1^{(2)} = k_1^{(2)} b_1^{(2)} = f'\left(a_1^{(2)}\right)\left(x_1^{(2)} - d_1\right)$$

which is (2). Proceeding in this fashion, we see that the output of the appended network is just the input gradient of the forward energy function. Note that the bias is not involved in the calculation of the input gradient. Therefore, three steps are involved in calculating the input gradient: 1) forward propagate through the lower network 2) copy the derivatives of the input states from the lower network to the appended network as the multiplicative constants of the linear neurons, and 3) forward propagate through the appended network. Once training is complete, the appended network is no longer needed and is removed.

Now we can form the backward energy function, so named because it could be obtained by backpropagating through the lower network:

$$E_b = (1/2)\left(y_1^{(0)}\right)^2 + (1/2)\left(y_2^{(0)}\right)^2 + (1/2)(y_3^{(0)})^2 = (1/2)\left(\frac{\partial Ef}{\partial i_1}\right)^2 + (1/2)\left(\frac{\partial Ef}{\partial i_2}\right)^2 + (1/2)\left(\frac{\partial Ef}{\partial i_3}\right)^2. \quad (3)$$

We will now minimize the sum of a constant times the backward energy function (3) and the forward energy function (1). The general idea is to backpropagate through the lower network to minimize the forward energy function and do another backpropagation starting at the top of the appended network to minimize the backward energy function (hence the description of the training algorithm as double backpropagation).

Backpropagation through the upper network has some subtleties because the weights are shared between the upper network and the lower network. First let us find the derivative of the backward energy function with respect to the state of the first neuron in the top layer recalling both that the neuron is linear and that for the top layer, the multiplicative constant is 1.

$$\frac{\partial Eb}{\partial b_1^{(0)}} = y_1^{(0)}.$$

Now, the gradient with respect to the weight F:

$$\frac{\partial E_b}{\partial F} = \frac{\partial E_b}{\partial b_1^{(0)}}\frac{\partial b_1^{(0)}}{\partial F} + \frac{\partial E_b}{\partial a_2^{(1)}}\frac{\partial a_2^{(1)}}{\partial F} = \frac{\partial E_b}{\partial b_1^{(0)}} y_2^{(1)} + \frac{\partial E_b}{\partial a_2^{(1)}} i_1.$$

The first term of the sum is found in normal backpropagation, the second is not. However, we will get the equivalent result by backpropagating through the whole network. Therefore, the algorithm proceeds as follows:

1) Present the input pattern and propagate it to the output (the top of the lower network).
2) Backpropagate the gradient of the forward energy function through the lower network. Compute the change in weights but do not change the weights yet.
3) Copy the appropriate derivatives from the lower network to the appended network.
4) Propagate forward through the upper network
5) Now backpropagate the backward energy function from the top of the appended network down through the original network, calculating the weights changes but do not change the weights yet.
6) Finally, change the weights using the weight changes calculated in steps 2) and 5).

In our experiments, there is a learning constant associated with each weight. The change in weight is a product of the learning constant, the gradient of the output energy (whether the forward or backward energy) with respect to the summed input of the neuron immediately "above" that weight , and the input to the weight. Effectively, the learning constant in step 2) above is half that of step 5).

III. Experimental Results

Each results presented below is the average error rate on the test set for ten runs for a particular architecture and particular training algorithm. Each run consists of one hundred iterations of the following: a pass through the training data followed by the performance evaluation on the test data. The best test results were retained. The best test performance always occurred before the last pass through the training data.

Five learning algorithms were used:

1) Backpropagation.
2) Full double backpropagation which was described above.
3) Partial double backpropagation is a modified form of the computationally expensive full double backpropagation. In this case we form a backward energy term that is a function of the gradients at the input to the hidden layer: $(\partial E_f / \partial a_j^{(1)})$. The rationale is that small changes in the input to the hidden layer should not affect the output. In this case, the appended network required to calculate these derivatives is smaller (minus the uppermost layer in Fig. 1).
4) Normal backpropagation followed by full double backpropagation. The reasoning is that backpropagation is faster and full double backpropagation following normal backpropagation should require less training cycles. In this case, full double backpropagation starts with the results of the network with the best test score found in 1) above.

TABLE I
ERROR RATE IN PERCENT FOR FIVE ARCHITECTURES AND FIVE TRAINING ALGORITHMS. SIX HUNDRED SAMPLES IN TRAINING SET AND SIX HUNDRED IN TEST SET.

ARCHITECTURE	256–10	256–10–10	256–20–10	256–30–10	256–40–10
BACKPROPAGATION	12.2	7.1	5.8	5.6	5.3
FULL DOUBLE BACKPROPAGATION	7.9	4.7	3.9	3.7	3.4
PARTIAL DOUBLE PROPAGATION	7.9	5.0	3.6	3.1	3.0
FULL DOUBLE BACKPROPAGATION FOLLOWS BACKPROP	7.9	6.7	5.6	5.4	5.0
PARTIAL DOUBLE BACKPROPAGATION FOLLOWS BACKPROP	7.9	6.9	5.6	5.5	5.0

5) Normal backpropagation followed by partial double backpropagation.

The first database consists of a "clean" set of digits: 600 for training and 600 for testing generated by twelve users generating ten examples of each of the ten digits. The input is on a 16 by 16 pixel array with -1's for the background and $+1$ for the foreground. One architecture was a (256–10), i.e., 256 input units and ten output units, fully connected. For this architecture full and partial double backpropagation are equivalent. The other architectures were of the form (256–x–10), i.e, x hidden units, once again fully connected.

The results for this database are shown in Table I. As can be seen by examining the first three rows, both full and partial double backpropagation improves performance significantly, especially for those cases where the error rates using regular backpropagation were small anyhow. Because full double backpropagation requires one backward pass through the original network and another backward pass through both the appended network and the original network, full double backpropagation is slower than normal backpropagation. In the initial implementation of double backpropagation the rate was four times slower. However, in subsequent implementations double backpropagation took only 30% longer using matrix manipulations for linear operations. This includes the forward pass through the appended network and all backward passes.

The convergence rate of double backpropagation compared to normal backpropagation is approximately the same, although the variance is large in both cases. Partial double backpropagation is faster than full double backpropagation and achieves approximately the same results. For this size database and networks, the training time using a Sun SPARCstation is inconsequential in either case.

Examining the last two rows of Table I, we see that double backpropagation following normal backpropagation does increase performance but insignificantly except for the 256–10 network. The advantage of this approach is the much faster convergence, typically in one or two passes through the training set after the completion of the normal backpropagation.

Of particular interest is the histogram of weights. For each of the ten 256–10–10 networks trained using normal backpropagation or full double backpropagation we compiled a histogram of the weights from the input to the hidden units (Fig. 2). Since there are 2570 such weights for each

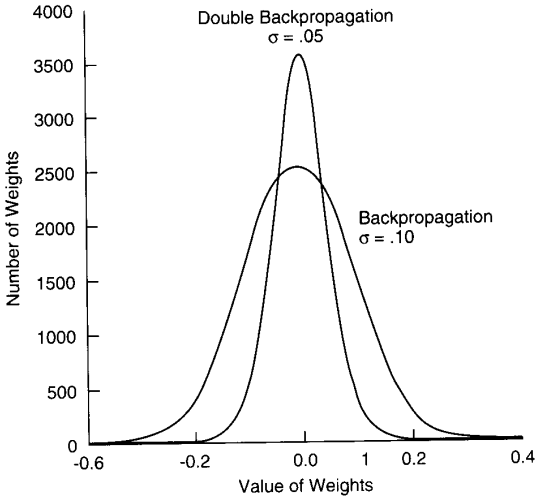

Fig. 2. Histogram of weights.

network (including the bias), the histogram represents the distribution of 25 700 weights. The standard deviation is much smaller for the double backpropagation case appearing to give results similar to what would happen with weight decay. Another histogram (Fig. 3) is that of the summed inputs to the hidden layer, essentially corresponding to the $a_j^{(1)}$ of Fig. 1. This histogram was the result of examining the ten units in the hidden layer for ten 256–10–10 networks for all of the 600 training patterns, a total of 60 000 samples. For both double backpropagation and backpropagation, we obtain a bimodal distribution. However, for double backpropagation, the standard deviation is much smaller. For the sigmoid transfer function used here, the input-output relationship is almost linear when the input ranges over ± 0.7. Calculations show that the histogram is 19.2% of the time in this range for double backpropagation and only 4.0% of the time for normal backpropagation indicating that double backpropagation forces the input into the linear region. Examination of similar histograms (not shown) for the output layer shown that double backpropagation and normal backpropagation do not show this type of dissimilar behavior. Thus the improved performance can be attributed to the weight distribution of the hidden layers, rather than the output layer.

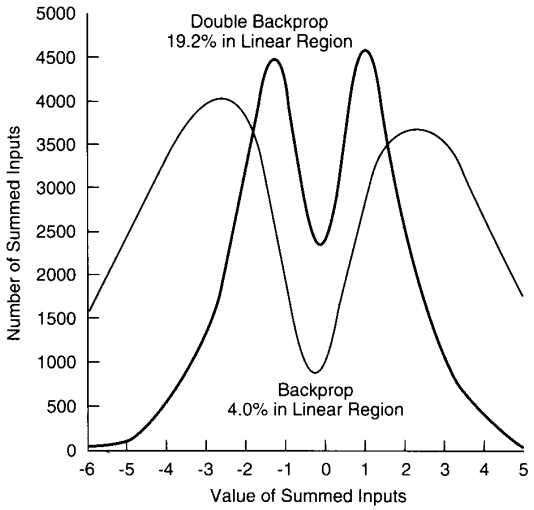

Fig. 3. Histogram of summed inputs.

We think that this smaller variance is important for better generalization. If the input weights are too large, then the hidden layer output tends to saturate and the outputs of the network will be insensitive to large changes in the input with subsequent poor inter-class discrimination. If the input weights are too small so that the hidden layer is always in the linear region, then small changes in the input will create larger changes (compared to the saturated case) in the outputs which increases the sensitivity to the input and thereby aversely affecting intraclass clustering. Double backpropagation seems to reach a compromise between the extremes of linearity and saturation.

A second database consisting of 320 training examples and 160 test samples were used on four locally constrained architectures that had been previously shown to give good results using backpropagation [10]. Twelve examples of each of the ten digits were hand drawn by a single person on a 16 by 13 bitmap using a mouse. Each image was then used to generate four examples by putting the original image in four consecutive horizontal positions on a 16 by 16 bitmap. Thus the architectures and training algorithms will be specifically tested against a database consisting strictly of translations.

The four architectures (Fig. 4):

1) local-net: A locally connected architecture with two hidden layers. The input layer is 16×16 and the first hidden layer is of size 8×8 with each neuron on the hidden layer taking its input from a 3×3 square on the input layer. For units in the hidden layer that are one unit apart, their receptive fields (in the input layer) are two pixels apart. Thus the receptive fields in the hidden layer overlap by one column and one row. The second hidden layer is of size 4×4 with a receptive field of size 5×5, overlapping again by one column and one row. The second hidden layer is fully connected to the output. The net effect is 1226 connections, 1226 weights, and 357 neurons.

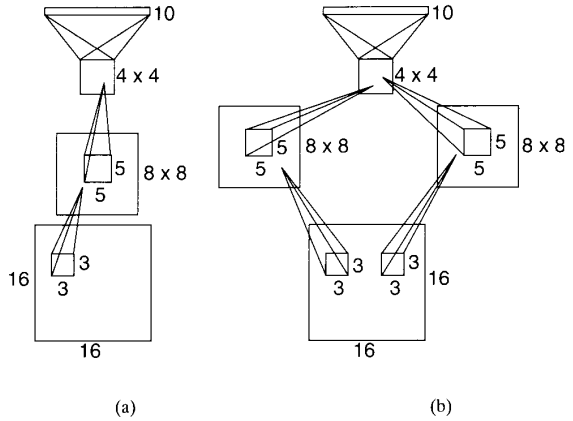

Fig. 4. (a) Local architecture (b) local21 and local2 architecture.

2) local2-net: A network with two hidden layers and weight sharing. There are two hidden layers with the first hidden layer consisting of two 8×8 feature maps, each feature map examining a 3×3 neighborhood on the input layer. All units in a feature map share the same weights. The second hidden layer is again of size 4×4 with receptive fields of size 5×5 and no weight sharing. The output of the second hidden layer is fully connected to the output. The net effect was 2266 connections, 1132 weights (free parameters), and 421 neurons.
3) local21-net: Same as local2-net except that the weights in going from the first to second hidden layer are shared. Thus the number of connections and neurons is the same but the number of free weights is now 382.
4) local4-net: Two hidden layers, the first of which consists of four 8×8 feature maps and the second four 4×4 feature maps. All the units in a feature map share the same weights, the receptive fields being of size 3×3 in going from the first to second hidden layer and of size 5×5 going from the first to second hidden layer. The net effect is 9674 connections, 1406 free weights, and 597 neurons.

In these cases of multiple hidden layers, partial double backpropagation means that the appended network consists of two layers: the mirror image (around the dashed line of Fig. 1) of the output layer and the hidden layer closest to the output. The resultant error rates for the four architectures and five learning algorithms are shown in Table II. Except for two cases of the local21 architecture double backpropagation improves performance. Our local21 architecture was never able to learn the training set in those cases where double backpropagation gives worse results. Of special interest is that fact that the local4 architecture which gives the best results using normal backpropagation has the most significant increase in performance (to 2.2%) using double backpropagation. In some cases, partial double backpropagation gives better results than full double backpropagation.

Our next trial was on a very large database consisting of 9709 training samples and 2007 test samples taken from handsegmented zip code data obtained from the U.S. Postal

TABLE II
ERROR RATE IN PERCENT FOR FOUR ARCHITECTURES AND FIVE TRAINING ALGORITHMS. 320 ITEMS IN TRAINING SET AND 160 IN TEST SET

ARCHITECTURE	local	local21	local2	local4
BACKPROPAGATION	8.6	8.2	4.5	3.8
FULL DOUBLE BACKPROPAGATION	5.0	6.2	3.3	3.1
PARTIALDOUBLE BACKPROPAGATION	6.9	8.5	2.8	2.2
FULL DOUBLE BACKPROPAGATION FOLLOWS BACKPROP	5.8	6.1	3.2	2.9
PARTIALDOUBLE BACKPROPAGATION FOLLOWS BACKPROP	6.5	9.5	3.7	2.6

Service. The architecture is fully explained in [11], [12] but in summary consists of the following:

1) A 16 by 16 input surrounded by a six pixel border set to -1 (the background level) to give a 28 by 28 input layer
2) A first hidden layer consisting of four independent 24 by 24 feature maps. Each unit in a feature map examines a 5 by 5 region of the input. The twenty-six weights (including the bias) are shared with every set of twenty-six weights connected to the other neurons within the same feature map.
3) A second hidden layer which does averaging/subsampling. It is composed of four planes of size 12 by 12. Each neuron in one of these planes takes inputs on four units on the corresponding plane in the first hidden layer. Receptive fields do not overlap. All the weights are constrained to be equal, even within a single unit.
4) A third hidden layer consisting of twelve feature maps arranged in 8 by 8 planes, each neuron examining a 5 by 5 field on some of the cells in the second hidden layer and weights constrained to be the same.
5) A final hidden layer consisting of twelve 4 by 4 averaging/subsampling maps.
6) An output layer fully connected to the last hidden layer.

The network has 4645 neurons, 2578 different weights, and 98 442 connections. Because of the size of the training set and the size of the architecture, one pass through the training data initially took on the order of eleven hours using a partially appended network. Using matrix multiplications for the appended network and other enhancements, we now require three hours for partial double backpropagation and four hours for full double backpropagation on this size data set and architecture. Performance is reported as two numbers: the percent of patterns that must be rejected in order to achieve a 1% error rate and the raw error rate. The best results using a Newton version of regular backpropagation had previously been reported [11,12] to be a 9.6% reject rate and 5.03% error rate, i.e., 9.6% of the patterns must be rejected to obtain a 1% error rate on the patterns not rejected, and there is a 5.03% error rate with no rejects. Patterns are rejected if the difference in output levels between the two neurons with the largest outputs is smaller than some critical value (adjusted to get the 1% error rate). Convergence is obtained in approximately thirty passes through the training data.

The following performances were obtained using double backpropagation:

Partial double backpropagation starting from the best network: 4.68% error rate and 10.5% reject.
Partial double backpropagation starting from random weights: 4.68% error rate and 8.9% reject.
Full double backpropagation starting from random weights: 4.68% error rate and 9.1% reject.

Convergence takes approximately twenty-three to thirty iterations. Therefore, partial double backpropagation starting from random weights gives the best improvement over regular backpropagation.

Possible alternatives to double backpropagation are weight decay and noise injection. In weight decay, when we change the weights using backpropagation, we additionally multiply each weight by (1−decay) where the decay is some small number. In noise injection, we add to each pixel of the input pattern a normally distributed, zero mean, random variable. On this architecture and training set, we tried both techniques. In the noise injection case, we varied the standard deviation from 0.1 to 1.0 and in weight decay, we varied the decay factor from 10^{-6} to 10^{-4} using regular backpropagation starting from random weights. Each run takes approximately three days on a Sun SPARCstation.

The following results were obtained:

Decay of 10^{-6}: 5.03% error rate and 11% reject rate.
Decay of 10^{-5}: 5.68% error rate and 13% reject rate.
Decay of 10^{-4}: 7.80% error rate and 22% reject rate.

Noise of 0.1: error rate of 5.2% and 10.3% reject rate.
Noise of 0.2: error rate of 5.3% and 9.7% reject rate.
Noise of 0.5: error rate of 5.2% and 11.3% reject rate.
Noise of 1.0: error rate of 7.5% and 20.0% reject rate.

In no case was the error performance improved and in most cases the reject rate was severely degraded. Thus, although the literature reports that weight decay and noise injection does work in many cases, there is no evidence of improvement for this large and sophisticated architecture.

Two larger networks were formed in the following manner: The weights for the network previously trained to a 4.68% error rate and the 8.9% reject rate were fixed, the output layer

removed (with its connections) and a network added to the output of the penultimate layer and trained. The penultimate layer has 192 outputs. In network 1, a 192–75–10 fully connected network was added. In network 2, a 192–75–40–10 fully connected network was added. The advantage of this technique is that training is much faster since one can store and then train on the output of the penultimate layer.

The following results were obtained:

Network 1 using regular backpropagation: 4.68% error rate and 9.4% reject rate.
Network 1 using full double backpropagation: 4.13% error rate and 7.7% reject rate.

Thus backpropagation slightly degrades performance (over the original network) in this case and double backpropagation significantly improves performance.

Network 2 using regular backpropagation: 4.58% error rate and 7.7% reject rate.
Network 2 using full double backpropagation: 4.43% error rate and 7.0% reject rate.

Therefore, in all these networks performance is improved using double backpropagation.

IV. Conclusions

Double backpropagation has been shown to be a technique that improves performance by forcing the output to be insensitive to incremental changes in the input. The improvements are especially significant for those architectures which show very good performance when trained using backpropagation. The penalty paid is an increased running time which is not too large a penalty if partial double propagation is used. Double backpropagation can be used following normal backpropagation but generally does not give as good results as double backpropagation alone. It was furthermore shown that double backpropagation creates a weight distribution at the input to the first hidden layer that has a smaller variance than that generated using backpropagation.

References

[1] D. E. Rumelhart *et al.*, "Learning internal representations by error propagation," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, 1* Rumelhart and McClelland, Eds. Cambridge MA: MIT Press, 1986, pp. 318–362.

[2] G. E. Hinton, "Learning distributed representations of concepts," in *Proc. Eight Annual Conf. Cognitive Science Society*, Amherst, MA. Hillsdale, NJ: Erlbaum, 1986, pp. 1–12.

[3] R. Scaletter and A. Zee, "Emergence of grandmother memory in feed forward networks: Learning with noise and forgetfulness, in *Connectionist Models and Their Implications: Readings from Cognitive Science*, D. Waltz and J. A. Feldman, Eds. Norwood, MA: Ableex, 1988, pp. 309–332.

[4] A. H. Kramer and A. Sangiovanni-Vincentelli, "Efficient parallel learning algorithms for neural networks, in *Advances in Neural Information Processing Systems I* , Denver, CO, 1988, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1989.

[5] A. von Lehman, *et al.*, "Factors influencing learning by back propagation," presented at the *IEEE Int. Conf. Neural Networks*, San Diego, CA, 1988, vol. I, New York, IEEE, pp. 335–341.

[6] J. Sietsma and R. J. F. Dow, "Neural net pruning—why and how," presented at the *IEEE Int. Conf. Neural Networks*, San Diego, CA, 1988, vol. I, pp. 325–333.

[7] S. E. Fahlman and C. Lebiere, "The Cascade-Correlation learning architecture, in *Advances in Neural Information Processing Systems II*, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1990, pp. 524–532.

[8] M. Mezard and J. -P. Nadal, "Learning in feedforward layered networks: the tiling algorithm, *J. Phys. A 22*, pp. 2191–2204, 1989.

[9] S. I. Gallant, "Optimal linear discriminants, presented at the *Eighth Int. Conf. Pattern Recognition*, Paris, France, 1986, pp. 849–852.

[10] Y. LeCun, "Generalization and network design strategies, in *Connectionism in Perspective*, Pfeifer *et al.*, Eds. 19 Zurich, Switzerland: Elsevier

[11] Y. Le Cun *et al.*, "Backpropagation applied to handwritten zip code recognition, *Neural Computation 1*, 1989, pp. 541–551.

[12] Y. Le Cun *et al.*, "Handwritten digit recognition with a back-propagation network," in *Advances in Neural Information Processing Systems*, Denver, CO, 1989, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, pp. 396–404.