Stochastic bidirectional training

T.D. Gedeon
E-mail: tom@cse.unsw.edu.au

Department of Information Engineering
School of Computer Science & Engineering
The University of New South Wales
Sydney NSW 2052 AUSTRALIA

ABSTRACT

In this paper we consider connectionist compression schemes using auto-associative networks, demonstrate the advantages gained by imposing different constraints on allowed network weights, and comparison with pruning of the unconstrained auto-associative network.

In this paper we demonstrate the advantages for generalisation performance of constraining weights symmetrically using weight sharing, and by constraining functional symmetry by the use of enhanced backpropagation networks trained bidirectionally [1-2]. In the process, we discover the stochastic bidirectional training algorithm.

KEYWORDS: Autoassociative, Bidirectional, Pruning, Weight sharing.

1. Autoassociative Topology

A normal feedforward network is shown in Figure 1.

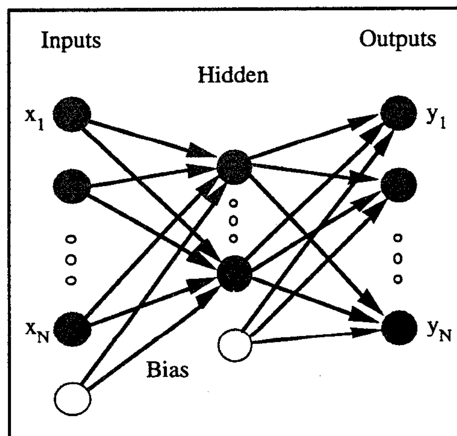

Figure 2. Normal feedforward network

This network has no backward, lateral or layer-skipping connections. All processing neurons have a bias which is implemented as an extra input which is always on. Note that following the usual (unfortunate) convention, input neurons are drawn, however these are not processing neurons, merely switch boxes distributing the single input x_i to the hidden neurons.

This is an auto-associative network, hence each of x_i are the same as the corresponding x_i.

The standard algorithm is as follows:

1. Initialise weights (including bias weights) to small random values.
2. Present input $X = (x_1, x_2, \ldots, x_N)$, desired output D=X.
3. Calculate output $Y = (y_1, y_2, \ldots, y_N)$:

   hidden activations $x'_j = f(\sum_{i=0}^{N} w_{ij} x_i),\ 0 \le j \le N1$

   output activations $y_k = f(\sum_{j=0}^{N1} w'_{jk} x'_j),\ 0 \le k \le N$

   where $f(x) = (1 + e^{-x})^{-1}$
4. Adapt weights recursively, starting with the output layer and working backwards towards the input layer:
   For the output neurons:
   $w_{jk}(t+1) = w_{jk}(t) + \eta \delta_k x'_j$ where $\delta_k = y_k(1-y_k)(x_k - y_k)$
   For the hidden neurons:
   $w_{ij}(t+1) = w_{ij}(t) + \eta \delta_j x_i$ where $\delta_j = x'_j(1-x'_j)\sum_k \delta_k w_{jk}$
5. Repeat from step 2 until finished.

2. Connectionist Compression

The significance of training an auto-associative network for compression is that the network learns a compressed representation for the input patterns (usually images) as well as the decompression function. For transmission over a slow network connection, the neural network weights could be transmitted as a fixed initial cost, while subsequently the compressed representation formed by the hidden neurons could be transmitted using the activation values of the hidden neurons for the particular pattern.

All neural compression is inherently lossy, hence interest in such work is largely for the insights into image compression and into neural networks.
In a standard auto-associative network such as described in the previous section, the first layer of weights from the inputs to the hidden units are seen as implementing a compression function on the input pattern. The second layer of weights from the hidden units to the outputs implement a decompression function on the compressed image. That is, the hidden unit activations are mapped to recreate an approximation of the original image.

3. Shared Weight Topology

The topology of the shared weight auto-associative network is the same as the standard auto-associative feed-forward network, it is the meaning of the weighted connections between units that is different.

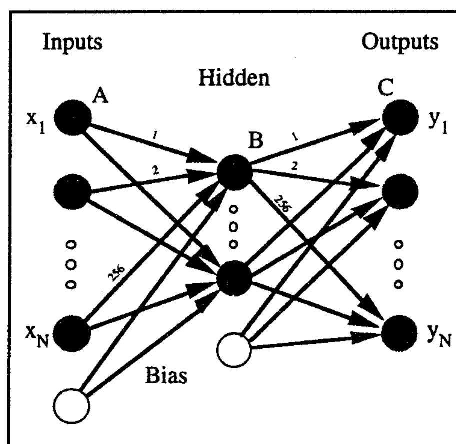

Figure 3. Shared weight network

The connection between input *A* and hidden unit *B* in Figure 2 has the same weight as between hidden unit *B* and output *C*. That is, the number of free parameters is no longer the same as the number of weights.

The reduction in free parameters by about half would be expected to improve generalisation performance

The simplest implementation of shared weights in a simulator is to back-propagate errors and update weights as in standard backpropagation without weight sharing, and then after the values of the A-B weight and B-C weight have diverged to average their values. This has the unfortunate effect of increasing the computation time required, but is much clearer conceptually than the alternatives, and was the approach we followed.

The significance of the weight sharing is that the space of possible network weight configurations is very much reduced or constrained.

In a shared weight network, the constraining of input to hidden and hidden to output weights to be identical is to effectively require that the compression function be directly invertible. This has two consequences. Firstly, it may make finding a compression-decompression function harder for a particular case. Secondly, if an appropriate function can be found, we would expect it to perform better in general than a standard backpropagation network in that the inverse function is 'the' inverse function, rather than any approximation of the inverse function.

In a standard backpropagation network, the first layer of weights may implement a non-invertible compression function. Then, the second layer could only implement some approximation to the ideal decompression function, as this ideal does not exist if the compression function was non-invertible.

The shared weight algorithm is as follows:

1. Initialise weights (including bias weights) to the same small random values as a standard run.
2. Present input $X = (x_1, x_2, \dots, x_N)$, desired output D=X.
3. Calculate output $Y = (y_1, y_2, \dots, y_N)$:

   hidden activations $x'_j = f(\sum_{i=0}^{N} w_{ij} x_i),\ 0 \le j \le N1$

   output activations $y_i = f(\sum_{j=0}^{N1} w_{ij} x'_j),\ 0 \le i \le N$

   where $f(x) = (1 + e^{-x})^{-1}$
4. Adapt weights recursively, starting with the output layer and working backwards towards the input layer:

   For the output neurons:
   $w'_{jk}(t+1) = w_{jk}(t) + \eta \delta_k x'_j$ where $\delta_k = y_k(1 - y_k)(x_k - y_k)$

   For the hidden neurons:
   $w_{ij}(t+1) = w'_{ij}(t) + \eta \delta_j x_i$ where $\delta_j = x'_j(1 - x'_j) \sum_k \delta_k w_{jk}$

   (Where $w'_{ij}(t+1) = w'_{jk}(t+1)$ for shared weights j = k.)
5. Repeat from step 2 until finished.

4. Bidirectional Topology

The topology of the bidirectional weight auto-associative network is still very much the same as the standard auto-associative feed-forward network, and again it is largely the meaning of the weighted connections between units that is different.

In a standard feed-forward network if we consider the weighted links to be for propagating activations, then we use the weights to modulate the activations. From this conventional viewpoint, it is clear that the weights in the reverse direction are all zero.

If we instead consired the weighted links to provide degree of significance measures which are modulated by

other information then the standard model is (implicitly) bidirectional already. That is, in the forward direction the weights are modulated by the activations, while in the reverse direction they are modulated by the error signals. Taking this latter view, our bidirectional training introduces a form of symmetry into this process.

The simplest implementation of bidirectional auto-associative training in an existing simulator is to swap the weights between the first and second layers whenever the direction of training changes. This requires inclusion of dummy biases for the input side. These are actually sometimes used in the simulators anyway, following the practice of the bp program of [3].

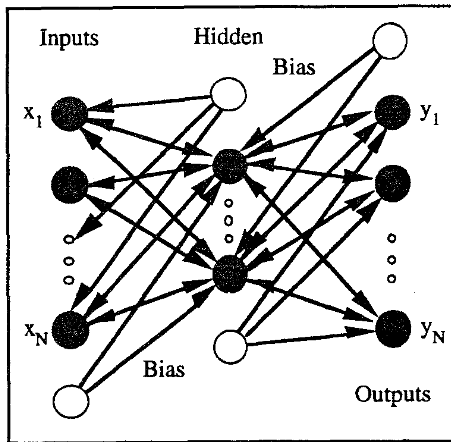

Figure 4. Bidirectional network

Note that in normal backpropagation feedforward networks we use the weights in standard back-propagation in both the forward direction and in the backward direction when calculating errors. By convention we say that values flow from the inputs to the outputs, nevertheless it is clear that information flows in both directions.

The bidirectional algorithm in the forward training direction is identical to the standard algorithm given in section 1. In the reverse training direction the bidirectional algorithm is as follows:

1. Initialise weights (including bias weights) to small random values.
2. Present input $Y = (y_1, y_2, \dots, y_N)$, desired output D=Y.
3. Calculate output $X = (x_1, x_2, \dots, x_N)$:

   hidden activations $y'_j = f(\sum_{k=0}^{N} w_{jk} y_k),\ 0 \le k \le N1$

   output activations $x_i = f(\sum_{j=0}^{N1} w'_{ij} y'_j),\ 0 \le i \le N$

   where $f(y) = (1 + e^{-y})^{-1}$
4. Adapt weights recursively, starting with the output layer and working backwards towards the input layer:

   For the output neurons:
   $w_{ij}(t+1) = w_{ij}(t) + \eta \delta_i y'_j$ where $\delta_i = x_i(1 - x_i)(y_i - x_i)$

   For the hidden neurons:
   $w_{jk}(t+1) = w_{jk}(t) + \eta \delta_j y_k$ where $\delta_j = y'_j(1 - y'_j) \sum_i \delta_i w_{ij}$
5. Repeat from step 2 until finished.

5. Method And Results

We tried to improve the performance of the bidirectional technique, using our observation that the total sum of squares for the bidirectional method oscillates. Since the bidirectional method does switch between directions of training, we attempted to find some relationship between these changes of direction and the oscillation of tss that we observed.

While we found no such relationship, we did discover [4-5] that if the network training switches direction after every pattern instead of every epoch, the results are much improved. The improvement in image quality is substantial. In retrospect, the improvement seems to be analogous to the improvement in performance on real problems of using stochastic or pattern by pattern updating of the weights as compared to epoch or batch mode updating of weights. Hence we call this method the "bidirectional stochastic" method.

6. Conclusion

The advantages for generalisation for both shared weights and bidirectional training probably derive from the reduction in free parameters, and the faster training of the input to hidden weights. That is, for both techniques these weights receive stronger feedback than under the standard back-propagation topology / training algorithm.

We expect advantages to ensue for rule extraction from the various forms of increased symmetry enforced. For shared weights, we rigidly enforce an invertible compression function, while for bidirectional training, a functional symmetry is imposed which is a less rigid symmetry in terms of the weight matrix, which at the limit tends towards such invertible compression functions.

References

[1] Nejad, A.F. and Gedeon, T.D. "BiDirectional MLP Neural Networks," *Proc. International Symposium on Artificial Neural Networks*, pp. 308-313, Taiwan, (1994).

[2] Nejad, A.F. and Gedeon, T.D. "Bidirectional Neural Networks Reduce Generalisation Error," in Mira, J. and Sandoval, F., (eds.), *From Natural to Artificial Neural Computation*, pp. 543-550, Springer Verlag,

Lecture Notes in Computer Science, vol. 930, (1995).
[3] McClelland, J.L. and Rumelhart, D.E., *Explorations in parallel distributed processing*, Cambridge, MA, MIT Press, (1988).
[4] Gedeon, T.D., Catalan, J.A. and Jin, J. "Image Compression using Shared Weights and Bidirectional Networks," *Proc. 2nd International ICSC Symposium on Soft Computing (SOCO'97)*, pp. 374-381, Nîmes, (1997).
[5] Gedeon, T.D., Catalan, J.A. and Jin, J. "A Comparison of some Connectionist Compression Schemes," *Proc. IEEE International Conference on Intelligent Engineering Systems (INES'97)*, pp. 283-288, Budapest, (1997).