In Proceedings of the *IEEE Symposium on Parallel and Distributed Computing*, pp. 308-315, 1989.

On the Pseudo Multilayer Learning of Backpropagation

Tony Martinez and *Michael Lindsey*
Computer Science Dept., Brigham Young University, Provo, Utah 84602

Abstract

Rosenblatt's convergence theorem for the simple perceptron initiated much excitement about iterative weight modifying neural networks. However, this convergence only holds for the class of linearly separable functions, which is vanishingly small compared to arbitrary functions. With multilayer networks of nonlinear units it is possible, though not guaranteed, to solve arbitrary functions. Backpropagation is a method of training multilayer networks to converge to the solution of arbitrary functions. This paper describes how classification takes place in single and multilayer networks using threshold or sigmoid nodes. It then shows that the current backpropagation method can only do effective learning on one layer of a network at a time.

Introduction

Early work on iterative gradient descent algorithms by Rosenblatt [6] showed that a network of non-linear threshold units with a single layer of weights could always converge to classify any linearly separable problem. This proof is encouraging, but the ratio of linearly separable functions to total functions becomes vanishingly small as the size of the input space increases. It is well known that a mutilayer network of nonlinear units can be made to solve any arbitrary classification. Recent research has sought learning algorithms for multilayer networks which will converge to solve arbitrary classifications. Although not having the feature of guaranteed convergence to a solution, the backpropagation [7] mechanism is one of the most popular connectionist models in use.

This paper discusses how multilayer weight multiplying networks in general, and backpropagation networks in specific, do classification. It is shown that learning in a backpropagation (BP) network only effectively takes place on one layer at a time, rather than simultaneously on all layers.

Linear Classification

In a simple perceptron [6] learning model, each node is a binary threshold unit, such that the output of a node is

$$O = \begin{cases} 0 & if\ \sum x_i w_i < \theta \\ 1 & if\ \sum x_i w_i \geq \theta \end{cases} , \tag{1}$$

where x_i are real valued inputs, w_i are real valued weights, and θ is a threshold value. It follows that a single node does a linear separation of its inputs into two output spaces. A two input node maps a line through the input space such that everything on one side of the line maps to 1, and everything on the other to a 0. Nodes with more inputs linearly separate the multi-dimensional space with a hyperplane.

Learning in these systems is attained by adjusting the weights, and thus changing the position of the hyperplane, until it properly separates the input space. This is typically done by using a training set which contains a set of input vectors and their outputs. Each input vector is iteratively applied to the network. The weights are then adjusted depending on how the real output matched the goal output by an equation of the form

$$\Delta w_{ij} = c\, E_p\, O_i , \tag{2}$$

where Δw_{ij} is the change in the weight from node i to node j, E_p is the error for the given pattern (typically being T_j - O_j, where T_j is the goal output of node j), and c is a learning constant.

Assuming that the goal function can be separated in a linear fashion, this learning model is guaranteed to converge to a proper solution in finite time [6]. The total number of possible functions which a network of n inputs can be programmed to accomplish is 2^{2^n}. However, the number of linearly separable (LS) functions having n inputs grows only as

$$LS(P,n) = \begin{cases} 2\sum_{i=0}^{n} \frac{(P-i)!}{(P-1-i)!\ i!} & if\ P > n \\ 2^P & if\ P \leq n \end{cases} \tag{3}$$

where P is the number of patterns to be classified [5]. For a general classifier P is equal to 2^n. The ratio of linearly separable (LS) functions to arbitrary functions quickly approaches 0 as n increases.

Although the single layer perceptron is limited to learning LS functions, it is well known [1] that a multilayer network of threshold units (where two layers is sufficient for binary classification and three for analog, though not necessarily efficient) can solve any arbitrary function. Numerous efforts have been made to find learning algorithms for multilayer networks with the provable convergence properties of the single layer perceptron. This has met with limited success. Although it does not always converge to a correct network, the *backpropagation* model of iterative learning is perhaps the most popular current model in its class [7].

Backpropagation

Much of the difficulty in finding learning algorithms for multilayer networks lies with the *responsibility* problem. When a node is at least once removed from the output layer, how is it to know if its current output aids or hinders the overall network in solving a function, when only a global error feedback is given. Since its output may be inverted any number of times before effecting the final output, a node does not know whether to increase or decrease its current sensitivity with the only error measure coming from the output/target differential. Backpropagation (BP) overcomes this through an error signal which passes back one layer at a time such that upon arriving at a given node, there *is* a causal connection between the node's current output and the final network output Although this wave is inherently sequential, this is considered to be quite reasonable and is *not* the pseudo multilayer learning addressed in this paper.

In deriving the backpropagation algorithm, the goal is to modify weights such that the modification causes a monotonic decrease in a global error signal. The training set method, mentioned above, is again used, but each weight modification is guaranteed to monotonically decrease this error signal. Since it is purely monotonic, the algorithm can and will get stuck in local minima which are not within the solution space. However, applications to date have shown it to have a relatively good success rate, even though the learning can be quite slow.

Although there are many claimants to the original BP algorithm, the reader is referred to [7] for a lucid derivation of the model. In order to derive the BP algorithm, it is necessary to have a node which executes a differentiable function. The threshold function is not differentiable, so the sigmoid function (which can be considered as a smooth threshold function) is chosen. This has the form of

$$O = \frac{1}{1 + e^{-(\Sigma x_i w_i + \theta)}} . \tag{4}$$

The classifying effect of the two functions will be shown to be quite similar. The learning mechanism, as derived in [7] is based on two simple substitutions of equation 2 above. They differ only the way that the error signal, E_p, is calculated. Nodes at the output layer modify their weights according to

$$\Delta w_{ij} = c\,(T_j - O_j)\,Oj\,(1 - O_j)\,O_i\,, \tag{5}$$

where T_j is the target output of the output node, O_j is the actual output, and O_i is the output of the node connected to node j by the weight. All nodes internal to the network modify their weights, starting from the layer closest to the output and working back, by the equation

$$\Delta w_{ij} = c\,(1 - O_j)\,Oj\,(\Sigma E_{pk}\,w_k)\,O_i\,, \tag{6}$$

where E_{pk} are the calculated error signals of the nodes to which node j outputs.

How Classification Takes Place

We discussed above how classification takes place with threshold units. It will now be show how classification happens with sigmoid units and then how a multilayer network of units is able to classify linearly non-separable functions. To simplify the discussion we will consider only two-input nodes. This allows us to use visualizable figures, since the classifying geometry is a line rather than a hyperplane. Also, no generality is lost since any function can be classified using sufficient layers of two input nodes. Also, we will look at the class of boolean functions, where inputs and outputs are binary.

A single threshold unit can solve the *inclusive OR* function in an infinite number of ways. Figure 1 shows the graphical mapping of 3 possible implementations of the *OR* function with a two input threshold unit.

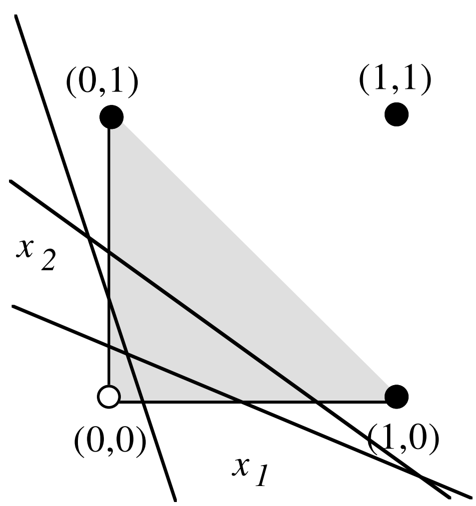

Figure 1 - Threshold unit *OR* function

The solid circles represents inputs for which a 1 should be output. Each line in the figure represents a different setting of the weights and threshold, while the graph shows the possible settings of the inputs x_1 and x_2. Modifications in the weights and/or threshold causes the line to rotate and shift. Any line which passes through the shaded region will classify the *OR* function. Note that movement of the line within the shaded region causes no change in functionality. However, when a line shifts out of the shaded region (by perhaps a very small perturbation) a major change in functionality follows.

A sigmoid unit is used to solve the *OR* function in a similar way. The difference is that the change over is smooth rather than discontinuous. Figure 2a shows a possible implementation of the *OR* function with a two input sigmoid unit.

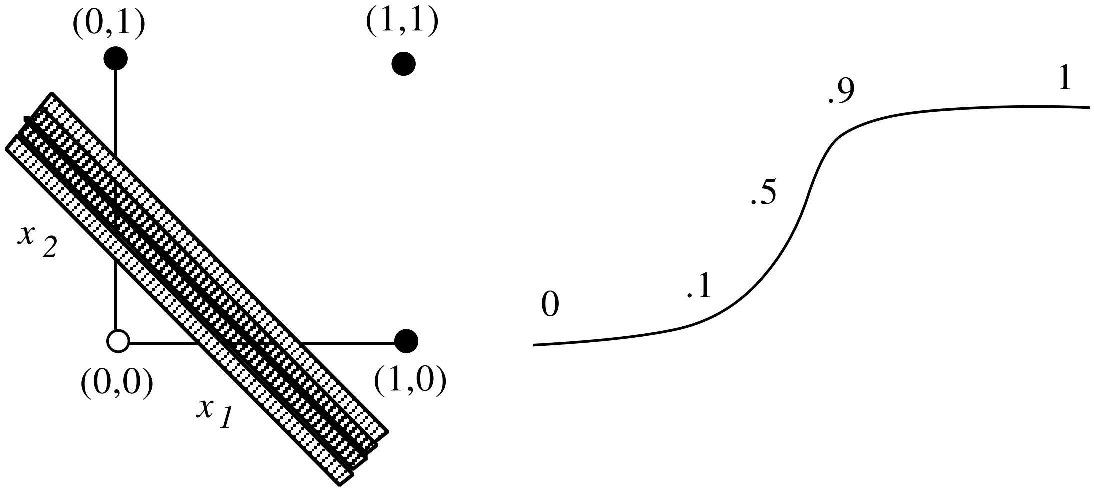

Figure 2a - Sigmoid Implementation of *OR*, 2b - Side view

Rather than a line discriminator as in figure 1, the sigmoid unit has what one might call a *fuzzy* line. In figure 1, all values below the line are 0, and all above are 1. In figure 2a it is the same except when one gets close to the line. (Note that for the sigmoid unit the values never quite reach 1 or 0, but any output approximating a 1 or 0 are pragmatically interpreted as if they were.) Figure 2b shows a side view of the fuzzy step function. Note that points close to the middle of the line are considered to be *between* 1 and 0. This has important effects during the learning algorithm, and a line lying close to training points will be most radically modified. The network stabilizes once the lines have shifted to locations where inputs do not fall upon the middle unstable portion. After learning has terminated, the backpropagation network classifies inputs in the same manner as threshold units, and for boolean functions, the sigmoid units could be replaced by threshold units, without effecting the networks ability to properly classify.

Non-Linearly Separable Classification

After learning, both the threshold and sigmoid units do classification by linearly separating the input space into two subspaces. To simplify the discussion of how to solve linearly non-separable functions, the threshold unit is used. This is reasonable, since we have already shown that they classify equivalently, except in the case where the sigmoid fuzzy line is close to a border, and this is a case which is rectified during the BP learning phase.

The simplest example of a linearly non-separable function is the two input *XOR*. This cannot be solved in a single layer since there is no way a line can be placed such that (1,1) and (0,0) are on one side and (0,1) and (1,0) are on the other. The way such a problem is solved is by having units in lower layers transform the input space into one which is linearly separable. Only at this point can the output unit classify the problem. Since convergence is proven for a linearly separable function, the real problem is whether the lower layers can be trained to transform the input space into a linearly separable one. Figures 3, 4, and 5 show how this is done to solve the *XOR* function with 2 hidden nodes and one output node. Figure 3 shows the simple network.

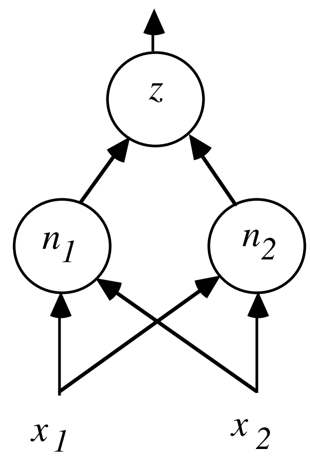

Figure 3 - Network Layout

Figure 4 shows possible setting for nodes n_1 and n_2 which transform the input into the linearly separable space. The arrows point towards the side of the line which outputs 1. The input space has been transformed such that when both n_1 and n_2 are 1, node z should output l. This is simply the linearly separable *AND* function which is implemented by node z (figure 5).

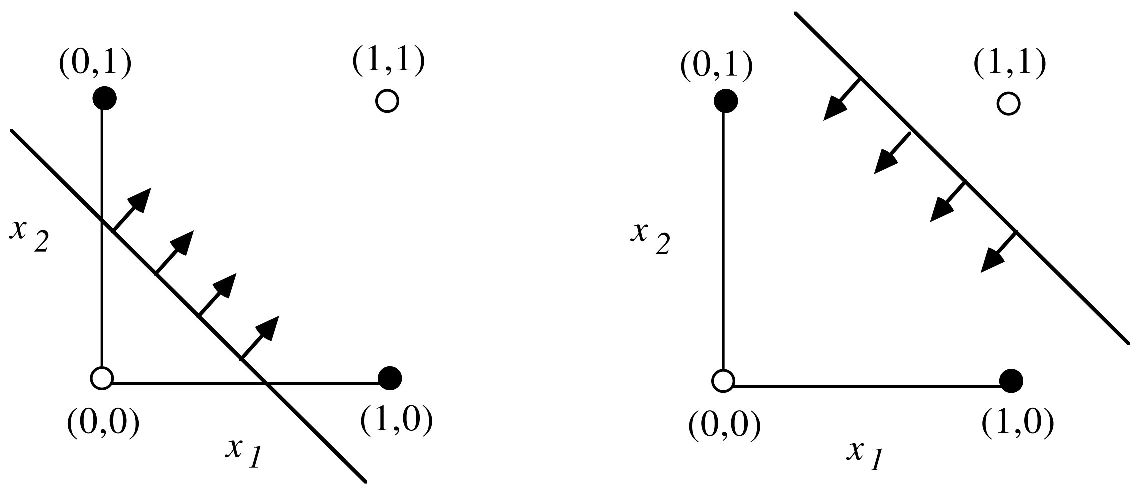

Figure 4 - Possible setups for Nodes 1 and 2

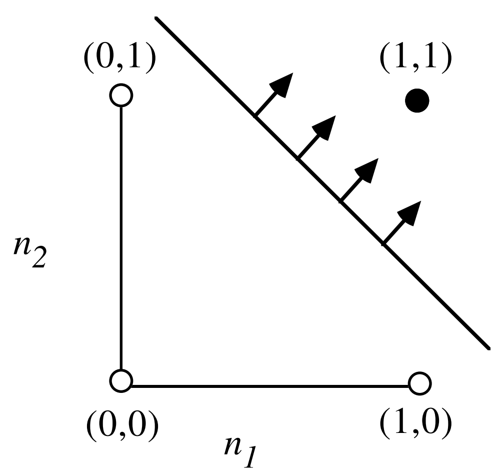

Figure 5 - Final Setting of Output Node *z*.

Why Backpropagation Learning is Pseudo Multilayer

Having shown the actual mechanisms for classifying arbitrary functions with multilayer sigmoid and threshold networks, we can now state why we claim backpropagation to be only pseudo multilayer. In BP learning, changes take place at all layers of the network during each iteration. However, for output nodes, until the preceding layer has transformed the input space into a linearly separable space, no amount of modification and changing can make the output node converge to a correct solution. Also, where more than two layers are used (a choice of topology to minimize the number of nodes in a single layer, and where this minimizing *requires* multiple layers to accomplish the separability transformation), for any layer, until the preceding layer has transformed the input into a more linearly separable space, no amount of modification of that and consequent layers can produce a linearly separable space for the output nodes.

Much of this occurrence is due to the BP learning equations. According to these equations, part of a weight's change is determined by the output of its preceding node, Nj. If the input into Nj is "incorrect" because the previous level of weights are not yet linearly separable, the output of Nj does properly represent the desired mapping. Thus the change in the weight from Nj is incorrect. Therefore, unlearning can take place on this level of weights until the previous level of weights has approached a state of linear separability.

This hypothesis was tested through simulation. The question is, what type of weight changes take place at layers following layers which have not yet done their linearly separable mappings. It was felt that weights at these hidden and output layers could follow three possible scenarios.

1. Converge towards a correct final state (proper multilayer learning).
2. Diverge for a while and then converge after the earlier layer(s) has made a linearly separable mapping (potentially harmful learning).
3. Randomly change and then converge after the earlier layer(s) has made a linearly separable mapping (inneffective learning).

The decision was made to start with a small network and check the hidden and output weights after each iteration of backpropagation to see what the weights were doing. We chose a three input, two output network with one hidden layer and a four input one output network with two hidden layers for our tests. After running these two tests we found that several of the hidden and outer layer weights diverged from their final state until their preceding layers were close to linear separability. (See appendix for complete simulation data).

Thus, later layers in a BP network can basically "spin their wheels" until the lower layers have made a proper transformation. It follows then that the upper layers do not do efficient work until that time. Also, after a lower layer has made a proper transformation, it continues to be modified, and may leave the proper transformation before (or even after) the upper layers have had sufficient iterations to converge and take advantage of it.

Conclusion and Alternatives

The conclusion is that in BP only one layer does *effective* modification at a time, while other upper layers are not yet doing effective modification and lower layers which continue to modify may be undoing the efforts of consequent layers. This is not meant as a criticism of BP, but rather as an effort to better understand its learning mechanisms while seeking to improve its speed and effectiveness.

We are currently looking at possible alternatives which either obviate the problem or seek to use BP learning in a more efficient manner. One is to explicitly guide learning through a layered approach. In this case output nodes are held steady in a set function (*AND, OR*, etc.) while the hidden layer(s) learn. Lower levels can in turn be held stable while the output or upper levels modify. Also, mechanisms to measure the extent to which layers transform the input space into a more linearly separable one are being looked at to form heuristics for learning at single layers. These measures do not always look at a global error signal, and have the potential of escaping minima which could not otherwise be avoided. In depth study of how networks really classify can also lead to divergent learning mechanisms which do exhibit simultaneous multilayer learning [2,3,4].

Bibliography

1. Lippmann, R. P., An Introduction to Computing with Neural Nets, *IEEE ASSP*, Vol. 3, No.4, pp. 4-22, (1987).
2. Martinez T. R., Models of Parallel Adaptive Logic, *Proceedings of the 1987 IEEE Systems Man and Cybernetics Conference*, pp. 290-296, (October, 1987).
3. Martinez, T. R. and J. J. Vidal, Adaptive Parallel Logic Networks, *Journal of Parallel and Distributed Computing*, Vol. **5**, pp. 26-58, (1988).
4. Martinez, T.R., Digital Neural Networks, *Proceedings of the 1988 IEEE Systems Man and Cybernetics Conference*, (August, 1988).
5. Nilsson, N., *Learning Machines*, McGraw-Hill, (1965).
6. Rosenblatt, F. *Principles of Neurodynamics*, Spartan Books, Washington, D.C., (1962).
7. Rumelhart, D. and McClelland, J., *Parallel Distributed Processing:*, Vol. I, pp. 318-362, MIT Press, (1986).

Appendix

The simulation was initiated with the idea to start with a small network and then work up to a larger network to test the BP multi-layer learning mechanism. The first test was performed using a small network with three input nodes, three hidden nodes, and two output nodes (figure 6).

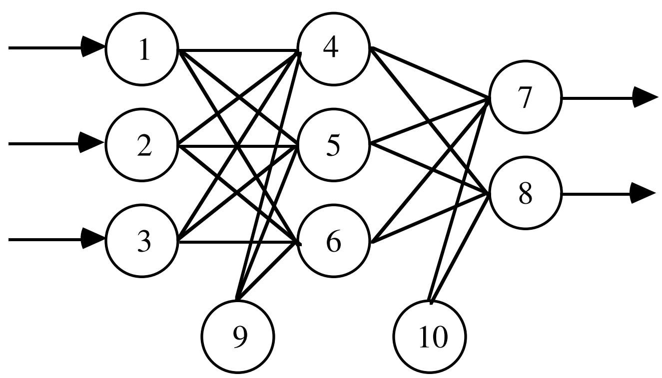

Figure 6 - 3-3-2 Network

The learning constant was 5.0, momentum of 0, and the training set was:

000->00, 011->11, 111->00, 010->11, 101->01, and 001->10

The initial weights were set to:

$w_{1,4} =$ -.11	$w_{1,5} =$ -.1	$w_{1,6} =$ .15	$w_{4,7} =$ -.1	$w_{4,8} =$ .01
$w_{2,4} =$ .3	$w_{2,5} =$ 0	$w_{2,6} =$ -.81	$w_{5,7} =$ .1	$w_{5,8} =$ .05
$w_{3,4} =$ -.2	$w_{3,5} =$ -.2	$w_{3,6} =$ .1	$w_{6,7} =$ .01	$w_{6,8} =$ -.1
$w_{9,4} =$ 0	$w_{9,5} =$ .78	$w_{9,6} =$ -.2	$w_{10,7} =$ -.13	$w_{10,8} =$ .4

The network converged after 136 iterations, with several interesting results. The observations were:

1. $w_{5,7}$ climbed to 5.81 after 83 iterations at which time it began a gradual decline to its final state of 4.81.
2. $w_{6,7}$ climbed to 1.29 after 35 iterations at which time it started to decline to -1.47.

3. $w_{10,7}$ performed similar to the above as it climbed to 1.7 after 9 iterations and then declined to -6.71.

4. $w_{5,8}$ rose to .29 after 46 iterations and then fell to -5.58.

A larger network (4->2->2->1) was then chosen for the second test (figure 7).

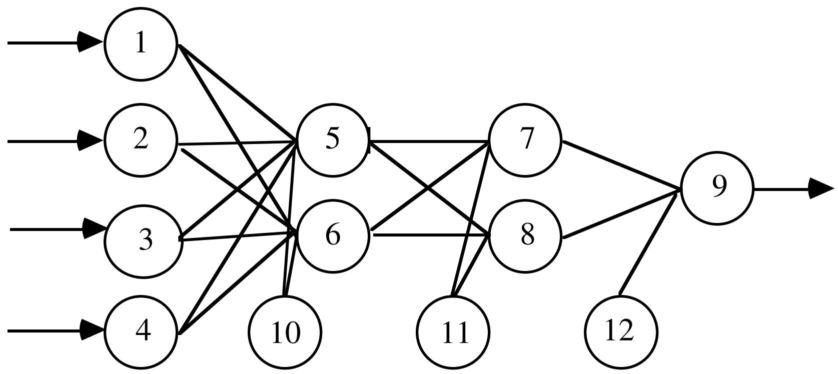

Figure 7 - 4-2-2-1 Network

The learning constant was 10.0 and the training set was:

1111->1, 0000->1, 0111->0, and 0110->0 (This is part of the 4-way XOR problem.)

The initial weights were set to:

$w_{1,5} = .2$	$w_{1,6} = .17$	$w_{5,7} = -.1$	$w_{5,8} = .3$	$w_{7,9} = .25$
$w_{2,5} = -1.5$	$w_{2,6} = 1.11$	$w_{6,7} = .1$	$w_{6,8} = -.15$	$w_{8,9} = .1$
$w_{3,5} = 1.2$	$w_{3,6} = -.3$	$w_{11,7} = 0$	$w_{11,8} = .81$	$w_{12,9} = -.1$
$w_{4,5} = 1.9$	$w_{4,6} = -2.2$			
$w_{10,5} = -.3$	$w_{10,6} = 0$			

The network took 99 iterations to converge and the following weights produced interesting results:

1. $w_{5,7}$ decreased to -2.45 after 75 iterations and then increased back up to -1.44.
2. $w_{5,8}$ dropped to -3.53 after 72 iterations and the rose to -2.19.
3. $w_{11,7}$ dropped to -1.58 after 39 iterations and then proceeded to climb to a final state of .86.