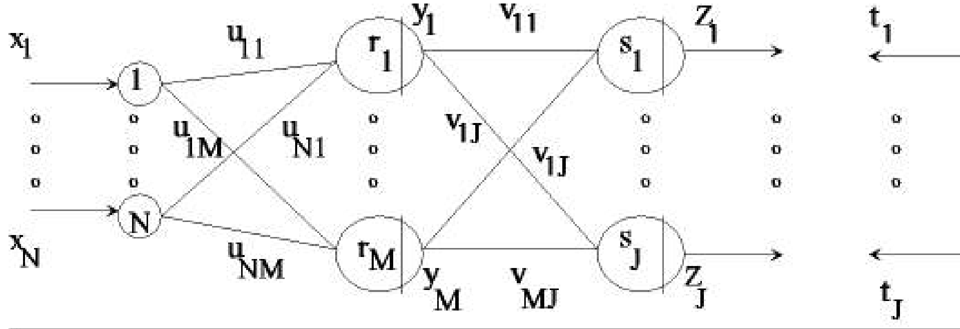
Backpropagation Neural Network Tutorial

The Architecture of BPNN's

A population P of objects that are similar but not identical allows P to be partitioned into a set of K groups, or *classes*, whereby the objects within the same class are more similar and the objects between classes are more dissimilar. The objects have N attributes (called properties or *features*) that can be measured (observed) so that each object can be represented by its N-dimensional feature vector

$$\mathbf{x} = (x_1,...,x_N)$$

We represent the objects by the feature vectors and partition the set of feature vectors into classes. A neural network is a scheme and algorithm that allow such a partition of a set of feature vectors into classes to be learned. We must have a set of Q feature vectors $\{\mathbf{x}^q: q = 1,...,Q\}$ and a set of target codewords that represent the classes, $\{\mathbf{t}^q: q = 1,...,Q\}$ such that for each feature vector there is a codeword that represents the class of that feature vector. The same target codeword may be the codeword for multiple feature vectors because multiple feature vectors can belong to the same class.



The above figure shows a standard BPNN. At the M hidden nodes the incoming feature values $\{x_n\}$ are weighted by the $\{u_{nm}\}$ and summed to form the values

$$r_m = \sum_{(n=1,N)} u_{nm} x_n \quad (1)$$

at each m-th node. Then each r_m value is put through a *sigmoid function* at the m-th node to get

$$y_m = g(r_m) = 1 / [1 + \exp(-\alpha(r_m - a))] \quad (2)$$

The y_m values are weighted by the $\{v_{mj}\}$ and summed at each of the J output nodes to obtain

$$s_j = \sum_{(m=1,M)} v_{mj} y_m \quad (3)$$

This weighted sum is put through the sigmoid function

$$z_j = h(s_j) = 1 / [1 + \exp(-\beta(s_j - b))] \quad (4)$$

This is an *actual output* for each $j = 1,...,J$ that is to be made to approximately equal the target t_j by adjusting the parameters $\{u_{nm}\}$ and $\{v_{mj}\}$.

The Training Method for BPNNs

The BPNN is trained on a set of feature vectors $\{\mathbf{x}^{(q)}: q = 1,...,Q\}$, called *exemplars*, and a paired set of corresponding target vectors $\{\mathbf{t}^{(q)}: q = 1,...,Q\}$ that are the desired outputs. These outputs are codewords that represent the different classes to which the feature vectors belong. This type of *supervised training* adjusts the parameters so that whenever $\mathbf{x}^{(q)}$ (or a vector very close to it) is put into the BPNN, then the actual output vector $\mathbf{z}^{(q)}$ is approximately equal to the target $\mathbf{t}^{(q)}$ for that input feature vector. The actual output is compared with all target vectors to make the decision as to what target, and class (each target represents a class), the input vector belongs.

The process of adjusting the parameters to achieve the training goal is to start with a randomly chosen set of initial parametric values for the $\{u_{nm}\}$ and $\{v_{mj}\}$ and then to iteratively adjust these parameters to minimize the mean-square error E over all J outputs and Q training pairs $\mathbf{x}^{(q)}$, $\mathbf{t}^{(q)}$, where

$$E = [1/(QJ)]\sum_{(q=1,Q)} \sum_{(j=1,J)} (t_j^{(q)} - z_j^{(q)})^2 \quad (5)$$

Such adjustment is done by using the method of *steepest descent* from differential calculus. Steepest descent finds the nearest local minimum from a starting point $\{u_{nm}^{(0)}, v_{mj}^{(0)}\}$ in the *parameter space*, for which the miniumum may not be a global minimum. The formulas are

$$u_{nm}^{(k+1)} = u_{nm}^{(k)} - \eta(\partial E/\partial u_{nm}) \quad (6)$$

$$v_{mj}^{(k+1)} = v_{mj}^{(k)} - \lambda(\partial E/\partial v_{mj}) \quad (7)$$

for all n and m, and all m and j, where η and λ are the step sizes that are called *learning rates*.

We will now derive these formulas in terms of the parameters, inputs, sums and sigmoid function values by means of the chain rule for derivatives. But first we need to know the derivative of the sigmoid function

$$w = h(s) = 1 / [1 + \exp(-\beta(s - b))] \quad (8)$$

$$dh(s)/ds = [1 + \exp(-\beta(s - b))]^{-2} [\exp(-\beta(s - b))](-\alpha) = \quad (9)$$

$$[w^2][1 + \exp(-\beta(s - b)) - 1](-\beta) =$$

$$[w^2][(1/w) - 1](-\beta) = [w^2][(1 - w)/w](\beta) = \beta w(1 - w)$$

where

$$w = 1 / [1 + \exp(-\beta(s - a))] \tag{10}$$

Now we proceed with the derivations using the chain rule. We suppress the q superscripts for simplicity (we will sum over q in the final result) and show the derivation for a single input feature vector and target vector.

$$\partial E/\partial v_{mj} = (\partial E/\partial z_j)(\partial z_j/\partial s_j)(\partial s_j/\partial v_{mj}) = \tag{11}$$

$$(\partial/\partial z_j)[\textstyle\sum_{(p=1,J)} (t_p - z_p)^2](\partial z_j/\partial s_j)(\partial s_j/\partial v_{mj}) =$$

$$[(-2)(t_j - z_j)] (\partial z_j/\partial s_j)(\partial s_j/\partial v_{mj}) =$$

$$[(-2)(t_j - z_j)] (dh(s_j)/ds_j)(\partial \textstyle\sum_{(m=1,M)} v_{mj}\, y_m)/\partial v_{mj}) = -2(t_j - z_j)\beta z_j(1 - z_j)y_m$$

Thus, training over all Q training feature vectors and their corresponding targets yields

$$\partial E/\partial v_{mj} = -2\alpha \textstyle\sum_{(q=1,Q)} (t_j^{(q)} - z_j^{(q)})z_j^{(q)}(1 - z_j^{(q)})y_m^{(q)} \tag{12}$$

The process is a little more complex for finding $\partial E/\partial u_{nm}$ because we must express the parts as functions of the key variables. Having done that, we again suppress the q index in the derivation.

$$\partial E/\partial u_{nm} = (\partial E/\partial y_m)(\partial y_m/\partial r_m)(\partial r_m/\partial u_{nm}) = \tag{13}$$

$$[\textstyle\sum_{(p=1,J)}(\partial/\partial z_j) (t_p - z_p)^2][(\partial z_j/\partial s_j)(\partial s_j/\partial y_m)(\partial y_m/\partial r_m)(\partial r_m/\partial u_{nm}) =$$

$$[-2\textstyle\sum_{(p=1,J)}(t_p - z_p)][\beta z_j(1 - z_j)]v_{mj}[\alpha y_m(1 - y_m)]x_n$$

where we have used the sigmoid derivatives for $g(r_m)$ and $h(s_j)$. This gives, over all Q training feature vectors, the following result.

$$\partial E/\partial u_{nm} = [-2\alpha\beta \textstyle\sum_{(q=1,Q)} \sum_{(p=1,J)}(t_j^{(q)} - z_j^{(q)})][z_j^{(q)}(1 - z_j^{(q)})]v_{mj}[y_m^{(q)}(1 - y_m^{(q)})]x_n^{(q)} \tag{14}$$

Putting these results into Equations (6, 7) yields the iterative algorithms on the (k+1)-st iteration to be

$$u_{nm}^{(k+1)} = u_{nm}^{(k)} + \eta \textstyle\sum_{(q=1,Q)} [\sum_{(j=1,J)}(t_j^{(q)} - z_j^{(q)})][z_j^{(q)}(1 - z_j^{(q)})]v_{mj}[y_m^{(q)}(1 - y_m^{(q)})]x_n^{(q)} \tag{15}$$

$$v_{mj}^{(k+1)} = v_{mj}^{(k)} + \lambda \textstyle\sum_{(q=1,Q)} (t_j^{(q)} - z_j^{(q)})z_j^{(q)}(1 - z_j^{(q)})y_m^{(q)} \tag{16}$$

where we have let η absorb $2\alpha\beta$ and let λ absorb 2β in Equations (13) and (12) respectively.

Improving the Training

In practice we use values less than or equal to unity in magnitude to prevent errors from growing and this applies to feature values, output codeword values and parametric weights. We draw the initial parameters randomly to be between -0.5 and 0.5 (empirical studies show that values of smaller magnitude are better to start the training). We first adjust the v_{mj} for all m and j and then adjust the u_{nm} for all n and m. This constitutes one iteration. We repeat this a given number of iterations, possibly thousands or tens of thousands. We train multiple times from different random initial parameter sets to assure a solution set of parameters that provide a good local minimum.

There are methods for helping the convergence to be smoother because the method usually is jumpy (the parameters jump around too much). The most commonly used method is to use a ter *momentum term* to smooth the direction of descent (or ascent on some steps): instead of using only the new increments to add to $u_{nm}^{(k)}$ and $v_{mj}^{(k)}$ on iteration k+1, we add an average of the new increments with the previous increments that we save. The increments from Equations (15, 16) are

$$\Delta u_{nm}^{(k)} = \eta \sum_{(q=1,Q)} [\sum_{(j=1,J)}(t_j^{(q)} - z_j^{(q)})][z_j^{(q)}(1 - z_j^{(q)})]v_{mj}[y_m^{(q)}(1 - y_m^{(q)})]x_n^{(q)} \quad (17)$$

$$\Delta v_{mj}^{(k)} = \lambda \sum_{(q=1,Q)} (t_j^{(q)} - z_j^{(q)})z_j^{(q)}(1 - z_j^{(q)})y_m^{(q)} \quad (18)$$

Thus the smoothed increments added onto the parameters on the (k+1)-st iteration are

$$w_1\Delta u_{nm}^{(k)} + w_2\Delta u_{nm}^{(k-1)} \quad (19)$$

$$w_3\Delta v_{mj}^{(k)} + w_4\Delta v_{mj}^{(k-1)} \quad (20)$$

instead of those in Equations (17, 18), where the positive weights satisfy

$$w_1 + w_2 = 1, \quad w_3 + w_4 = 1 \quad (21)$$

There is also the *en route* method of adjusting the learning rates. We start small so the convergence starts. If an adjustment of the v_{mj} parameters makes the MSE decrease, then we increase the learning rate λ (converging so speed it up with bigger steps). For any increase in E we decrease λ (we have gone past the local minimum and need to proceed with smaller steps). This technique is also applied to adjustments to η when the u_{nm} are adjusted.

A High Level Algorithm

We can implement an algorithm for BPNN's by the following steps.

Step 1. Input N, M, J, Q, $\mathbf{x}^{(q)}$ and $\mathbf{t}^{(q)}$ for q = 1,..,Q, and number of iterations I (set i = 0)

Step 2. Randomly draw all parameters of $\{u_{nm}\}$ and $\{v_{mj}\}$ where each is between -0.5 and 0.5

Step 3. Compute all r_m, y_m, s_j, z_j and E

Step 4. Update all parameters v_{mj} using momentum to smooth the direction of steepest descent and compute s_j, z_j and E. If new E is smaller than previous E then increase learning parameter λ else decrease it

Step 5. Update all parameters u_{nm} using momentum to smooth the direction of steepest descent and compute r_m, y_m, s_j, z_j and E. If new E is smaller than previous E then increase learning parameter η, else decrease it

Step 6. Increment iteration number i

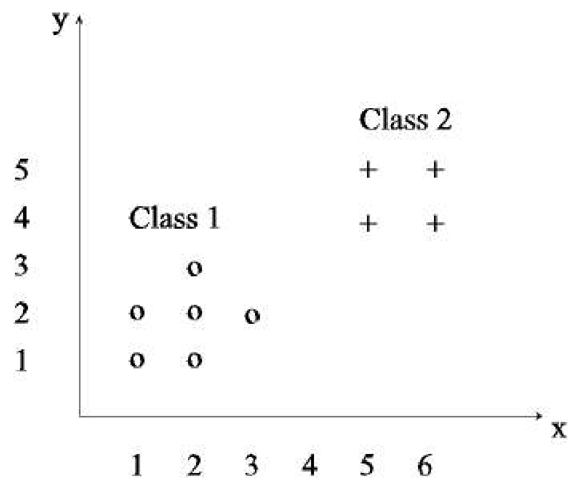
Step 7. If I > I then stop, else go to Step 4

The Ins and Outs of BPNN's

The question arises naturally as to how we choose the targets to pair with the feature vector inputs for the purpose of training the BPNN. A simple example here will show a general way to do this. Suppose we have the 2-dimensional vectors shown in the figure below. There are two classes. The vectors and their classes are give below.

Class 1:
(1,1), (2,1), (1,2), (2,2), (3,2), (2,3)

Class 2:
(5,4), (6,4), (5,5), (6,5)



These can be put in any order in the file of feature vectors, but for each feature vector we must also have a target vector that provides its class. Let us choose the following target vectors.

Target 1: (0,1) [represents Class 1] *Target 2*: (1,0) [represents Class 2]

Thus we can put all of this information in a file of rows in the form shown below. We includes notes to the right that are not in the file to explain the data.

2	(N = dimension of feature vectors)
2	(M = number of hidden nodes)
2	(J = dimension of target vectors)
10	(Q = number of feature vectors)
1, 1, 0, 1	(first feature vector (1,1), target (0,1))
2, 1, 0, 1	(second feature vector (2,1), target (0,1)
1, 2, 0, 1	:
2, 2, 0, 1	
3, 2, 0, 1	
2, 3, 0, 1	
5, 4, 1, 0	
6, 4, 1, 0	
5, 5, 1, 0	:
6, 5, 1, 0	(Qth feature vector (6,5), target (1,0)

When we read this data into the computer, we have values for the input feature vectors and their target vectors. For example, the 10 feature vectors and their paired targets are

$$x_1^{(1)} = 1, \quad x_2^{(1)} = 1, \quad t_1^{(1)} = 0, \quad t_2^{(1)} = 1$$
$$\vdots \qquad\qquad \vdots$$
$$x_1^{(10)} = 6, \quad x_2^{(10)} = 5, \quad t_1^{(10)} = 1, \quad t_2^{(10)} = 0$$

In the usual case we would process the feature vectors to be used for training the BPNN by dividing all of the feature values for the first feature by the maximum value of that feature, i.e., by 6, so all of its values would be between 0 and 1. Similarly we would divide the second feature values by 5. This is a good principle to keep the numerical errors from repeated multiplications well behaved.