Backpropagation

J.G. Makin

February 15, 2006

1 Introduction

The aim of this write-up is clarity and completeness, but not brevity. Feel free to skip to the "Formulae" section if you just want to "plug and chug" (i.e. if you're a bad person). If you're familiar with notation and the basics of neural nets but want to walk through the derivation, just read the "Derivation" section. Don't be intimidated by the length of this document, or by the number of equations! It's only long because it includes even the simplest details, and conceptually it's entirely straighforward.

2 Specification

We begin by specifying the parameters of our network. The feed-forward neural networks (NNs) on which we run our learning algorithm are considered to consist of layers which may be classified as input, hidden, or output. There is only one input layer and one output layer but the number of hidden layers is unlimited. Our networks are "feed-forward" because nodes within a particular layer are connected only to nodes in the immediately "downstream" layer, so that nodes in the input layer activate only nodes in the subsequent hidden layer, which in turn activate only nodes in the *next* hidden layer, and so on until the nodes of the final hidden layer, which innervate the output layer. This arrangement is illustrated nicely in Fig. (1). Note that in the figure, every node of a particular layer is connected to every node of the subsequent layer, but this need not be the case.

A few comments on notation: If a particular layer has $J \in \mathbb{N}$ nodes in it, then we will refer to an arbitrary node in that layer as the j^{th} node, where $j \in \{0, 1, \dots, J\}$. Similarly, the i^{th} node is in a layer with I nodes, and so on, where each layer makes use of its own index variable. We take (abusive) advantage of this notation in the discussion below by referring to *layers* by the node index variable associated with each one. Thus, e.g., the i^{th} node is in the i^{th} layer, which has a total of I nodes.

Secondly, the layers are labeled in reverse alphabetical order (we didn't want to make it too easy on you, gentle reader), so that the further upstream (i.e. the closer to the input layer), the later the letter.

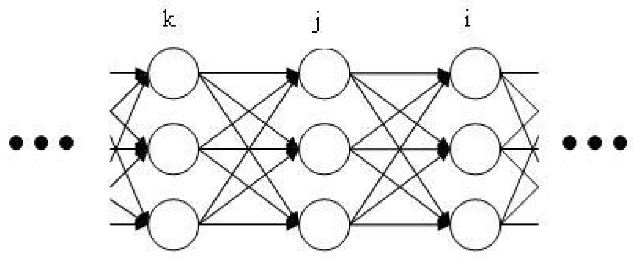

Figure 1: A piece of a neural network. Activation flows from layer k to j to i.

Thirdly and finally: Since the layers are not in general fully connected, the nodes from layer k which innervate the j^{th} node of layer j will in general be only a subset of the K nodes which make up the k^{th} layer. We will denote this subset by K_j, and similarly for the other subsets of input nodes. (Frankly, this distinction has no ramifications for the derivation, and if the reader is confused by it he is advised to ignore it.)

3 The McCulloch-Pitts Neuron

A single McCulloch-Pitts (MP) neuron (Fig. 3), very simply, transforms the weighted sum of its inputs via a function, usually non-linear, into an activation level, alternatively called the "output." In class we broke this up into two parts, the weighted sum and the activation function, largely to stress that all MP neurons perform the weighted sum of the inputs but activation functions vary. Thus, the weighted sum is

$$x_j = \sum_{k \in K_j} w_{kj} y_k, \tag{1}$$

where again K_j is the set of nodes from the k^{th} layer which feed node j (cf. Fig. 2); and the activation is

$$y_j = f(x_j). \tag{2}$$

We discussed four different activation functions, $f(\cdot)$, in class: linear,

$$f(z) = \beta z; \tag{3}$$

threshold,

$$f(z) = \begin{cases} 1 & z \geq \theta \\ 0 & z < \theta; \end{cases} \tag{4}$$

sigmoid,

$$f(z) = \frac{1}{1 + e^{-\gamma z}}; \tag{5}$$

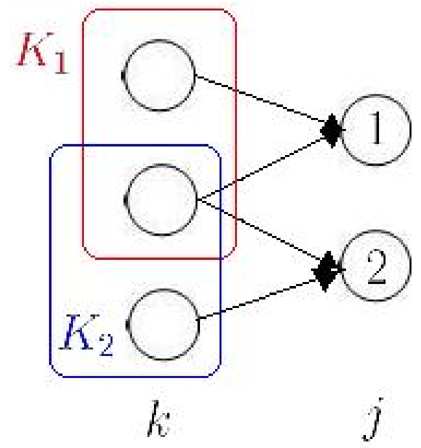

Figure 2: The set of nodes labeled K_1 feed node 1 in the j^{th} layer, and the set labeled K_2 feed node 2.

and radial basis, as in e.g. the Gaussian:

$$f(z) = exp\Big\{-\frac{(z-\mu)^2}{\sigma^2}\Big\}. \tag{6}$$

Here $\beta, \theta, \gamma, \sigma$, and μ are free parameters which control the "shape" of the function.

4 The Sigmoid and its Derivative

In the derivation of the backpropagation algorithm below we use the sigmoid function, largely because its derivative has some nice properties. Anticipating this discussion, we derive those properties here. For simplicity we assume the parameter γ to be unity.

Taking the derivative of Eq. (5) by application of the "quotient rule," we find:

$$\begin{aligned}
\frac{df(z)}{dz} &= \frac{0 \cdot (1-e^{-z}) - (-e^{-z})}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}}\left(\frac{e^{-z}}{1+e^{-z}}\right) \\
&= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) \\
&= f(z)(1-f(z)) \qquad (7)
\end{aligned}$$

This somewhat surprising result will simplify our notation in the derivation below.

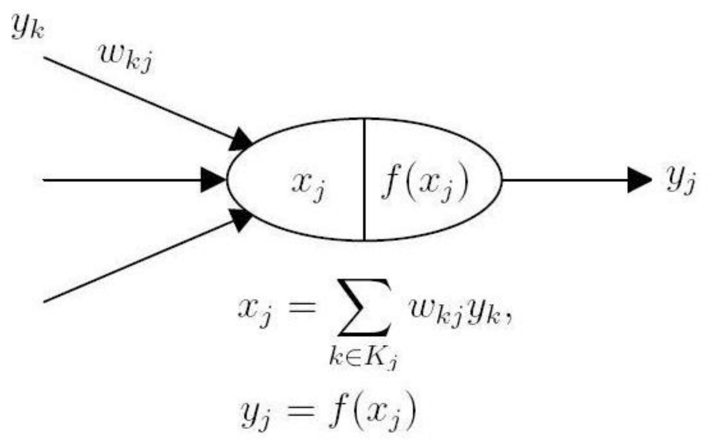

Figure 3: The MP neuron takes a weighted sum (x_j) of the inputs (y_k), and passes it through the activation function $f(\cdot)$ to produce the output y_j.

5 Interpretation of the Algorithm

A supervised learning algorithm attempts to minimize the error between the actual outputs—i.e., the activation at the output layer—and the desired or "target" activation, in this case by changing the values of the weights in the network. Backprop is an iterative algorithm, which means we don't change the weights all at once but rather incrementally. How much should we change each weight? One natural answer is: in proportion to its influence on the error; the bigger the influence of weight w_m, the greater the reduction of error that can induced by changing it, and therefore the bigger the change our learning algorithm should make in that weight, hoping to capitalize on the strength of influence of the weight at this point of the error curve. Of course, this influence isn't the same everywhere: changing any particular weight will generally make all the others more or less influential on the error, including the weight we have changed.

A good way to picture this "influence" is as the steepness of a hill, where the planar dimensions are weights, and the height of the hill is the error. This picture is shown in Fig. 4. (One obvious limitation of this approach is that our imaginations limit us to three dimensions, and hence to only two weights at once; whereas our network may have many, many more than two weights.) Now, given a position in weight space by the current value of the weights, the influence of weight w_m on the error is the steepness of the hill at that point along the direction of the w_m axis. The steeper the hill at that point, the bigger the change in weights. The weights are changed in proportion to these steepnesses, the error recalculated, and the process begun again.* This process is iterated until the error falls below some pre-ordained threshold, at which point the algorthim is considered to have learned the

*It makes intuitive sense to recalculate the error every time a weight has been changed, but in your programming assignment you will probably want to calculate all the errors at once, and then make all the changes at once, without recalculating the error after each change.

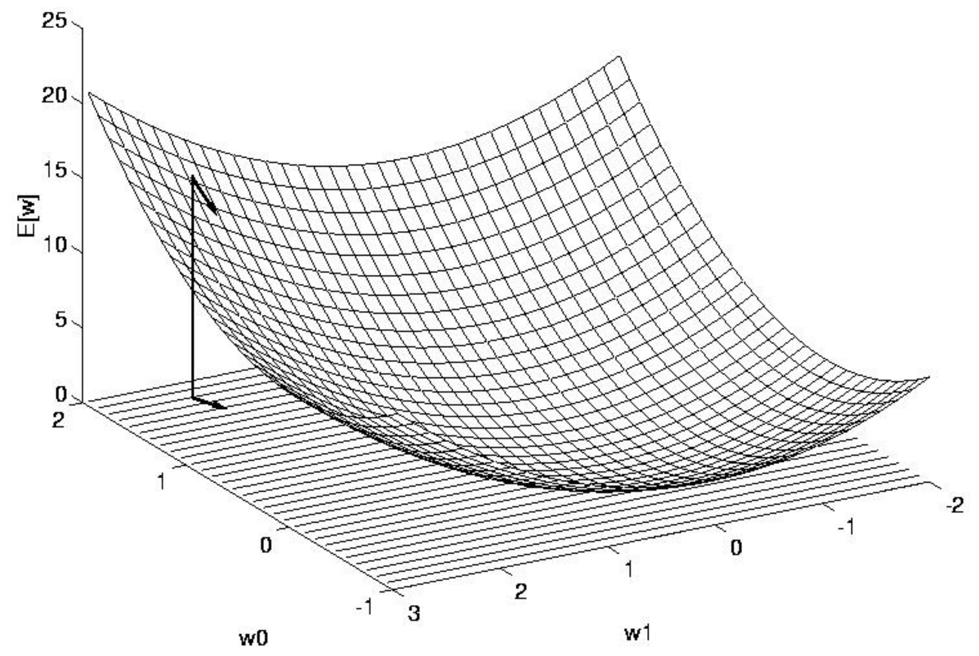

Figure 4: The error as a function of the weight space.

function of interest and the procedure terminates. For this reason, backprop is known as a "steepest descent" algorithm.

Students who have taken a multi-variable calculus course will have long since recognized the quantity which we have variously referred to as the "influence of weight w_m on the error" and the "steepness of the error curve in the direction of weight w_m" as the derivative of the error with respect to weight w_m. All that remains is to calculate this quantity.

6 Derivation

In preparation for the derivation of the algorithm, we need to define a meaure of error. Intuitively, the error is the difference between the actual activation of an output node and the desired ("target") activation (t_j) for that node. The total error is the sum of these errors for each output node. Furthermore, since we'd like negative and positive errors not to cancel each other out, we square these differences before summing. Finally, we scale this quantity by a factor of 1/2 for convenience (which will become clear shortly):

$$E := \frac{1}{2}\sum_{j=1}^{J}(t_j - y_j)^2. \tag{8}$$

It must be stressed that this equation applies only when the j^{th} layer is the output layer, which is the only layer for which the error is defined.

From the considerations mooted in the previous section, the weight change for a weight connecting a node in layer k to a node in layer j is

$$\Delta w_{kj} = -\alpha \frac{\partial E}{\partial w_{kj}}. \tag{9}$$

Here α is a free parameter (the "learning rate") that we set prior to training; it lets us scale our step size according to the problem at hand. In our case, adjustments of α will be *ad hoc*, so we will content ourselves with this intuitive explanation. Note the negative sign: this indicates that weight changes are in the direction of *decrease* in error. Now expanding the partial derivative by the chain rule, we find:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial x_j} \frac{\partial x_j}{\partial w_{kj}}, \tag{10}$$

where we recall from Eq. (1) that x_j is the weighted sum of the inputs into the j^{th} node (cf. Fig. 3), and hence

$$\frac{\partial x_j}{\partial w_{kj}} = y_k. \tag{11}$$

We will find it notationally convenient to treat the first two factors as a single quantity, an "error term":

$$\delta_j := -\frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial x_j}. \tag{12}$$

Nevertheless, we would like to calculate this term. We recall that $y_j = f(x_j)$ is the sigmoid function, the derivative of which we calculated in Eq. (7), so that in the present set of variables:

$$\frac{\partial y_j}{\partial x_j} = y_j(1 - y_j). \tag{13}$$

Finally we consider the first partial derivative in the error term. When j is an output layer, this quantity is easy to calculate; it is just the derivative of Eq. (8) with respect to y_j, i.e.,

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j). \tag{14}$$

(Notice how the scaling factor was neatly eliminated in taking the derivative.) Putting Eqs. (11), (13), and (14) into Eq. (10), we have:

$$\frac{\partial E}{\partial w_{kj}} = -(t_j - y_j) y_j (1 - y_j) y_k. \tag{15}$$

In the case where j is a hidden layer, $\partial E/\partial y_j$ is not quite as simple. Intuitively, we need to see how the error caused by y_j has propagated into the activations of the next (i^{th}) layer. Mathematically, this amounts to another application of the chain rule. Recalling our multi-variable calculus:

$$\frac{\partial E}{\partial y_j} = \sum_{i \in I_j} \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial x_i} \frac{\partial x_i}{\partial y_j}. \tag{16}$$

The first two partial derivatives are, from Eq. (12), just the error term for the *next* (i^{th}) layer, δ_i. The final term is simply the derivative of Eq. (1) with respect to the input:

$$\frac{\partial x_i}{\partial y_j} = w_{ji}. \tag{17}$$

Making these substitutions yields:

$$\frac{\partial E}{\partial y_j} = -\sum_{i \in I_j} \delta_i w_{ji}, \tag{18}$$

so that in the case of hidden layers,

$$\frac{\partial E}{\partial w_{kj}} = -\sum_{i \in I_j} (\delta_i w_{ji}) y_j (1 - y_j) y_k. \tag{19}$$

Notice that even though this formula differs from the output-layer case, Eq. (15), we can write a single formula by employing our definition of the error term, Eq. (12):

$$\frac{\partial E}{\partial w_{kj}} = -\delta_j y_k. \tag{20}$$

We haven't done anything magical here, we've just buried the differences in the equations for the output and hidden layers in the error term. This is essentially what you will do in your programming assignment: create a single function that calculates the weight change for the weight connecting a paricular pair of nodes, given the activation of the upstream node and the error term of the downstream node. Each node will calculate its own error term, and this will vary according to whether the node lives in a hidden or output layer. We now put all these pieces together.

7 Formulae

For a weight connecting a node in layer k to a node in layer j (see Fig. 1), the change in weight is given by

$$\Delta w_{kj}(n) = \alpha \delta_j y_k + \eta \Delta w_{kj}(n-1) \tag{21}$$

where:

- α is the learning rate, a real value on the interval (0,1];
- y_k is the activation of the node in layer k, i.e. the activation of the presynaptic node, the one *upstream* of the weight;
- n and $n - 1$ refer to the iteration through the loop (i.e., the "epoch");
- η is the momentum, a real value on the interval [0,1); and

- δ_j is the "error term" associated with the node *after* the weight, i.e. the postsynaptic node.

This is simply a combination of Eqs. (9) and (20), plus a "momentum term" which will be explained shortly, and application of the iterative procedure discussed in Section 5. Now, if j is the ouput layer,

$$\delta_j := (t_j - y_j)y_j(1 - y_j); \tag{22}$$

if j is a hidden layer, then

$$\delta_j := (\sum_{i \in I_j} \delta_i w_{ji})y_j(1 - y_j). \tag{23}$$

One can see from Eq. (23) that calculation of the error term for a node in a hidden layer requires the error term from nodes in the subsequent (i.e. downstream) layer, and so on until the output layer error terms are calculated using (22). Thus, computation of the error terms must proceed backwards through the network, beginning with the output layer and terminating with the first hidden layer. It is this backwards propagation of error terms after which the algorithm is named.