Back-propagation

The question is how to compute these slopes of an error function, particularly for multilayer networks. The answer to this question is one of the major contributions in the history of cognitive science. The discovery, now known as the back-propagation technique, was apparently independently achieved several times (Bryson & Ho, 1969; Parker, 1985; Rumelhart et al., 1986; Werbos, 1974).

The basic idea of the back-propagation technique is to compute the slopes for weights going into output units, and then to back-propagate the results to earlier layers in the network. Readers who may need to refresh their knowledge of slopes and how they are computed may want to look at appendix A. In the derivations to follow, there are occasional references to equations from appendix A, each identified by the prefix *A*.

Generalized delta rule

Sometimes it is necessary to find the derivative of a function that is a function of some function of the variable x. This is accomplished using the chain rule for differentiation (equation A.9), which establishes an intermediate variable, typically called u. Differentiation is another term for finding a derivative of a function. The chain rule specifies that the derivative of $f(x)$ with respect to x is the product of the derivative of $f(x)$ with respect to u and the derivative of u with respect to x.

In the back-propagation technique, the desired derivative of the error function with respect to weight is actually computed with two intermediate variables: the activation of the unit j (y_j) and the net input to unit j (x_j):

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \times \frac{dy_j}{dx_j} \times \frac{\partial x_j}{\partial w_{ij}} \qquad (2.6)$$

This is a double application of the chain rule in which the partial derivative of error with respect to weight is equal to the product of three terms:

1. The partial derivative of error with respect to the unit's activity
2. The derivative of unit activity with respect to the unit's net input
3. The partial derivative of the unit's net input with respect to weight

The chain rule makes sense in this context because error is a direct function of output activation (and target activation), unit activation is a direct function of the unit's net input, and net input is a direct function of connection weights.

These three key derivatives are computed from equations that have already been presented. For example, the partial derivative of error with respect to output activation is the sum of differences over output units between actual activations and target activations.

$$\frac{\partial E}{\partial y_j} = \sum_j (y_j - t_j) \tag{2.7}$$

Equation 2.7 derives from equation 2.4, which defined error as the squared difference between actual and target activity. A full derivation of equation 2.7 is provided in appendix B.

Next, the derivative of a unit's activity with respect to its input is the product of the activity and 1 minus the activity.

$$\frac{dy_j}{dx_j} = y_j \times (1 - y_j) \tag{2.8}$$

Although derivation of equation 2.8 is typically presented in a somewhat offhand manner as being "easy" (e.g., Rumelhart et al., 1986), it in fact requires about nine explicit algebraic steps that not everyone can apprehend at a glance. Consequently, this derivation is presented in detail in appendix C for those readers seeking some degree of demystification.

Finally, the third term used in computing the slope of the error function with respect to weight is this:

$$\frac{\partial x_j}{\partial w_{ij}} = y_i \tag{2.9}$$

Equation 2.9 conveys that the partial derivative of net input to a receiving unit, with respect to changes in weight, is simply the level of activity in the sending unit. This follows in only one algebraic step from how net input to a unit is computed (equation 2.1) and the exponent rule for differentiation (equation A.5).

Combining our results from equations 2.6–2.9, we get that the slope of error change with respect to weight change at the output units is the following:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \times \frac{\partial y_j}{\partial x_j} \times \frac{\partial x_j}{\partial w_{ij}} = (y_j - t_j) \times y_j \times (1 - y_j) \times y_i \tag{2.10}$$

In summary, equation 2.10 indicates that computation of the desired slope of error with respect to weight can be accomplished via the chain rule of differentiation. This slope is the triple product of the derivatives of error with respect to activation, of activation with respect to net input, and of net input with respect to weight. Derivations of each of these three component slopes from equations for network error, unit activation, and net input to a unit ensure that the slope of error for a particular output weight can be computed from three respective readily available local values: the output-unit activation (y_j), the target activation for that unit (t_j), and the activation of the sending unit (y_i). Again, the purpose of this computation is to enable weight adjustment to be some small negative proportion of the current error slope, as specified in equation 2.5. This serves to move each weight entering each output unit a tiny step in the right direction to reduce network error, thus enabling the network gradually to learn the training patterns.

Another point to note about equation 2.10 is that the results depend on the particular activation function used for a given unit. Equation 2.8 depends on the use of semilinear activation functions, such as the sigmoid and asigmoid activation functions specified in equations 2.2 and 2.3. A semilinear activation function is one in which a unit's output is a nondecreasing, monotonic, differentiable function of its net input. This definition applies at least to sigmoid, asigmoid, and hyperbolic-tangent functions, all of which roughly resemble an *S* shape.[4] Different activation functions could produce somewhat different results. A more general version of equation 2.10 is this:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \times \frac{\partial y_j}{\partial x_j} \times \frac{\partial x_j}{\partial w_{ij}} = (y_j - t_j) \times f_j'(x_j) \times y_i \tag{2.11}$$

Here $f_j'(x_j)$ is the derivative of unit j's activation function with respect to net input to the unit. Equation 2.11 would apply to learning in networks having any sort of activation function (Rumelhart et al., 1986).

When the activation function happens to be linear ($y_j = x_j$), the resulting, restricted version of equation 2.11 becomes the following:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial w_{ij}} = (y_j - t_j) \times y_i \tag{2.12}$$

This is because $f_j'(x_j) = \partial y_j / \partial x_j = 1$. Equation 2.12 was indeed a precursor of equation 2.11, functioning as a learning rule for simple associative networks with linear output units and without any hidden units. These early neural networks were called *perceptrons* (Rosenblatt, 1962), and the learning rule was known under a variety of names: the delta rule, the adaline rule, the Widrow-Hoff rule, and the least-mean-square rule (Widrow & Hoff, 1960).[5] Later this delta rule was shown to be virtually identical to Rescorla and Wagner's (1972) model of classical conditioning (Sutton & Barto, 1981).

An even more restricted rule for weight adjustment occurs when there is no target activation, i.e., when $t_j = 0$:

$$(y_j - t_j) \times y_i = y_j y_i \tag{2.13}$$

As in the delta rule, the learning-rate parameter r is also typically used here to modulate the amount of weight change.

Equation 2.13 is also known as the *Hebb* rule because it was proposed much earlier in verbal form by Donald Hebb (1949) in speculation about how learning might occur in the brain. "When an axon of cell A is near enough to cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased" (1949, p. 62). In neural-network terms, Hebb can be read as recommending that we strengthen the connection between units A and B whenever A and B are simultaneously active. Equation 2.13 is actually a bit more general than Hebb's proposal because it covers both negative and positive activation ranges and also decrements in connection strength. Essentially, equation 2.13 says to adjust a weight between two units in proportion to the product of their simultaneous activation.

Although the Hebb rule is effective in some learning situations, it is rather severely limited by the fact that the stimulus training patterns must be orthogonal, that is, uncorrelated with each other, for learning to be successful. Correlations between patterns introduce contamination between responses to different patterns, and thus make accurate learning of partially correlated patterns impossible. A further limitation is that, like the delta rule, the Hebb rule cannot cope with hidden units having nonlinear activation functions. Nonetheless, the Hebb learning rule is favored by some modelers because of its biological plausibility (e.g., Kelso, Ganong & Brown, 1986) and the fact that it does not rely on the presence of a training target. It does figure in a few developmental models.

The more general rule derived for equation 2.11 is called the *generalized* delta rule because it can deal with units with nonlinear activation functions and with networks possessing hidden units (Rumelhart et al., 1986). So far, we have covered only the weights entering output units. It is also necessary to consider how the back-propagation method deals with the weights entering hidden units.

Propagating error backwards

To propagate weight changes back to the previous layer in a network, it is necessary to know the partial derivative of error with respect to the activity of a hidden unit (y_i) in that layer:

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial y_i} \tag{2.14}$$

By the chain rule for differentiation (equation A.8), this is the sum across j links to output units of the product of two partial derivatives: error with respect to net input to an output unit j and net input to that output unit with respect to activity of the sending hidden unit i.

From equations 2.7 and 2.8 and the chain rule (equation A.8), we can compute the derivative of error with respect to net input:

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \times \frac{dy_j}{dx_j} = (y_j - t_j) \times y_j \times (1 - y_j) \tag{2.15}$$

It is the product of the partial derivative of error with respect to activity of the output unit (y_j) and the derivative of activity of the output unit with respect to net input to the output unit (x_j).

The next derivative term in equation 2.14 is the derivative of net input to the output unit (x_j) with respect to activity of the sending hidden unit (y_i). This can be obtained in one algebraic step by applying the exponent derivative rule (equation A.5) to the equation for computing net input to a unit (equation 2.1).

$$\frac{dx_j}{dy_i} = w_{ij} \tag{2.16}$$

Combining our results from equations 2.15 and 2.16, we get the following:

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \times \frac{dx_j}{dy_i} = \sum_j (y_j - t_j) \times y_j \times (1 - y_j) \times w_{ij} \tag{2.17}$$

In words, the derivative of error with respect to sending activation from a hidden unit is the sum across links to the output units of the product of four terms:

1. The difference between output and target activation
2. Output activation
3. 1 minus output activation
4. The connection weight between the hidden unit and the output unit

This is a key step in the back-propagation of error, but still not the whole story. To obtain the desired derivative of error with respect to incoming weights for a hidden unit, we use equation 2.10 but substitute the result of equation 2.17 for the original $\partial E/\partial y_j$ and simplify by combining like terms:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \times \frac{dy_j}{dx_j} \times \frac{\partial x_j}{\partial w_{ij}} = \sum_j (y_j - t_j) \times y_j^2 \times (1 - y_j)^2 \times w_{ij} y_i \tag{2.18}$$

The strategy used in equations 2.17 and 2.18 can then be applied recursively back to each successive layer in the network until an error slope has been computed for every output and hidden unit and the connection weights have been adjusted accordingly. Hence the name *back-propagation* of error. This works to train a multilayer feed-forward network, except that the error signal does become a bit diluted with increasing numbers of layers of hidden units. Please keep in mind that equations 2.14 to 2.18 refer to error at the hidden-unit layers, unlike previous equations, which dealt with error at the output units.

Summary of back-propagation

In summary, the back-propagation method for training multilayer feed-forward networks is conducted in two phases. There is first a feed-forward phase in which input patterns are presented and the network computes output values, producing an error value for each pattern at each output unit. This is followed by a feed-backward phase in which error derivatives at each unit are computed and weights are adjusted to reduce error.

Pattern and batch training

These phases can be conducted once for each and every presentation of the pair of training vectors that constitute a single example. This is known as *pattern* training because weight adjustment occurs after each training pattern. Alternatively, the entire set of training patterns can be presented and the errors accumulated before any back-propagation and weight adjustment is done. This is known as *batch* training because the patterns are processed in a single batch. In the case of batch training, the equations would be written with pattern subscripts and the results summed across patterns.

There is some controversy about the relative merits of batch versus pattern training in terms of both psychological plausibility and learning effectiveness. At first glance, it might be thought that pattern training is more plausible. However, there is some psychological (Oden, 1987) and physiological (Dudai, 1989; Squire, 1987) evidence for batch learning. For example, the hippocampus processes information in batch mode in order to relay its information to relevant cortical areas at some later time. In terms of learning effectiveness, batch learning is potentially more computationally efficient than pattern learning because it avoids making and unmaking redundant weight changes that might result from the sequential processing of pattern learning. Even in batch learning, however, outputs are compared to their targets independently of other patterns. Thus, the learning system never has to process more than one pattern at a time, although it does need to keep a running sum of network error, which is eventually used to adjust the weights.

Momentum

There is one additional technique that is commonly used to speed back-propagation learning, and that is the addition of a momentum term. The basic idea is to give each weight some relative degree of inertia or momentum, so that it will change less when its last change was small and change more when its last change was large. This could perhaps induce larger weight changes when the weight is far from the minimum error, and small weight changes when the weight is closing in on the minimum error. A modification of equation 2.5 shows how this works:

$$\Delta w_{ij}(t+1) = -r\frac{\partial E}{\partial w_{ij}} + m\Delta w_{ij}(t) \tag{2.19}$$

Here m is a momentum parameter between 0 and 1. Typically, m is set to about 0.9. The new weight change at $t+1$ is negatively related to slope scaled by learning rate r, plus the amount of weight change on the previous time step t scaled by momentum m. In practice, the use of a momentum term allows a programmer to increase the learning rate a bit without increasing the danger of oscillations across the valley defined by the error minimum. Indeed, the two main parameters that are manipulated and reported in back-propagation research are learning rate and momentum.

Evaluating Back-propagation

There are good reasons for the fact that back-propagation is far and away the most popular algorithm for training neural networks, whether on developmental problems or more generally. It is basically robust and often effective in learning a wide variety of problems, it uses only local computations (which is considered biologically plausible), and it is widely available in software packages. Despite the fact that there is no mathematical guarantee that the back-propagation algorithm will find the global error minimum, it often does a fairly good job of getting close (J. A. Anderson, 1995). However, it is also apparent that back-propagation is not a panacea for all applications, because it is plagued by some significant limitations. Some of these limitations are important and have led to improvements of various kinds. The basic limitations of back-propagation have to do with its slowness, the static nature of network design, an inability to learn some difficult problems, occasional overfitting of training data, catastrophic interference, an inability to scale up well to large problems, and biological implausibility.

Learning speed

In terms of learning speed, it is well known that back-propagation can take many thousands of epochs to learn even fairly simple problems, even with a momentum term. (An epoch is a pass through all of the training patterns.) From a psychological point of view, this often seems far too slow for a plausible model of human learning. There appear to be two principal reasons for back-propagation being so slow. One is the step-size problem, and the other is the moving-target problem (Fahlman, 1988; Fahlman & Lebiere, 1990). The step-size problem has to do with an issue presented earlier: uncertainty about how large a step to take in making a weight change. After computing the slope of the error surface at a given size weight, the back-propagation algorithm increases the weight if the slope is negative and decreases the weight if the slope is positive (refer to figure 2.7). Large steps might get to the minimum error faster than small ones, but if they are too large, they might create an oscillation across the valley that never settled into the right weight for minimizing error. To avoid such oscillations, back-propagation tends to take rather small steps, governed by the learning rate parameter, typically set to a moderate proportion such as 0.5. As noted earlier, these difficulties are ameliorated to some extent by the use of a momentum parameter. Other algorithms, such as cascade-correlation, try to improve further on this by using second- as well as first-order derivatives of the error surface. More on this later.

The moving-target problem has to do with the fact that each hidden unit in a back-propagation network is trying to become a feature detector to contribute to the overall solution of the problem it is trying to learn. The difficulty is that all of the hidden units are changing at once in this attempt. Instead of each unit moving quickly and decisively to adopt some useful role in the overall solution, there is a rather complex dance among the hidden units, which can take a long time to sort out.

One manifestation of the moving-target problem is the so-called herd effect. Imagine that there are two subtasks involved in an overall solution to a hypothetical learning problem. As always, the hidden units must decide which of these subtasks to solve. If subtask *A* generates a larger or more coherent error signal, then the hidden units initially tend to converge on subtask *A* and ignore other subtasks. Once subtask *A* is solved, they move on to subtask *B*, with the result that subtask *A* reappears as the major source of error. If you have watched very young children play hockey or soccer, you know what this is like. The players surround the puck (or ball) until it finally squirts out of the group and across the ice (or field). The whole group reconverges on the puck, and the cycle begins again. Eventually the herd (whether players or hidden units) splits up, with each one learning to mind its own job, but this can take considerable time and effort to achieve. Some newer algorithms, such as cascade-correlation, have sought to improve on this performance by encouraging hidden units to specialize in certain parts of the problem from the beginning of learning.

It has also been noted that back-propagation learning slows exponentially with an increasing number of layers of hidden units. This is due to attenuation of the error signal as it propagates back through the layers of the network. Algorithms that can adjust weights only one layer at a time, like cascade-correlation, can avoid this slowdown as depth of the network increases.

Static network design

In classic back-propagation learning, a network is typically designed by hand, by the programmer, and then remains static as learning proceeds. The programmer must decide how many layers of hidden units to use, how many hidden units per layer, and what the weight connection scheme is like. Some researchers are better at this than others, and so network design is often more of an art than a science. It has more to do with intuitions and experience-based rules of thumb than with principles of mathematics, neuroscience, or psychology. Generally speaking, the computational power of a neural network is proportional to the number of hidden units that it possesses. If a network is designed with too few hidden units, it could fail to learn the training set. But if it is designed with too many hidden units, it might overfit the training data, essentially memorizing them by rote and failing to develop any useful generalizations that could help with novel examples of the same problem. This is somewhat analogous to the preferences expressed by Goldilocks in her fairy tale. It can't be too much or too little—it has to be just right. But how does nature or evolution anticipate the right size and connectivity of networks for each of the various problems that a child might confront over the course of a lifetime, particularly in rapidly changing environments. Later we consider arguments that, indeed, nature likely cannot accurately anticipate such things. An alternative, employed by generative networks such as cascade-correlation, is to start a network in an underpowered state and recruit as many hidden units as needed to solve the problem being learned.

Catastrophic interference

A further problem with back-propagation networks is that new learning catastrophically interferes with old learning (McCloskey & Cohen, 1989; Ratcliff, 1990). In human memory, new learning interferes with old knowledge a bit, but not catastrophically (e.g., Barnes & Underwood, 1959). Ironically, the fact that such interference is catastrophic in back-propagation networks derives from the same properties that make neural learning so desirable for modeling, that is, that knowledge is stored on shared connection weights. Because the problem is created by overlapping hidden-unit representations, it is not too surprising to find that it can be avoided by a variety of techniques that minimize representational overlap (French, 1999). These solutions tend to require extreme changes to back-propagation learning, such as mixing in old training pairs with new ones or using two different network modules, one for new learning and another for long-term storage.

Scaling up

In an updated version of their influential book on perceptrons, neural networks without hidden units, Minsky and Papert (1988) criticized even hidden-layered back-propagation networks for not being able to scale up to large and complex problems. If back-propagation networks do have difficulty scaling up to large problems, this could be due to the use of a single, homogeneous network on problems that humans would solve in a modular fashion, by breaking the overall problem down into natural subproblems, each solved by a distinct module. Some promising inroads have been made into modular neural networks. In contrast to larger and more homogeneous networks, modular networks restrict complexity to be proportional to problem size, generalize effectively, learn multiple tasks, easily incorporate prior knowledge, perform robustly, and are easily extended or modified (Gallinari, 1995). The solutions of modular networks should also be easier to analyze than the solutions of homogeneous networks.

Biological plausibility

It has become customary to criticize back-propagation for being biologically implausible because the brain is not known to send error signals back through a feed-forward network of neurons (Crick, 1989). Activation does flow in both forward and backward directions in biological networks, but the backward connections carry activation, not error information. There have been a number of proposals for building more plausible versions of back-propagation learning by using activation instead of error back-propagation (Fausett, 1990; Hecht-Neilson, 1989; O'Reilly, 1996).

From a developmental point of view, there is another sense in which back-propagation learning is biologically implausible. This concerns the fact that the brain generates new synapses and even new neurons, and does so under the pressure of learning, not only in infancy but throughout life (Eriksson, Perfilieva, Bjork-Eriksson, Alborn, Nordborg, Peterson & Gage, 1998; Gould, Tanapat, Hastings & Shors, 1999; Gould, Reeves, Graziano & Gross, 1999; Kempermann, Kuhn & Gage, 1997; Quartz & Sejnowski, 1997). These processes of synaptogenesis and neurogenesis may be responsible for the qualitative increases in computational and representational power shown by children at various points in their development. Generative algorithms like cascade-correlation make extensive use of these generative processes by recruiting new hidden units and new links to access them. Neural algorithms for static networks that fail to implement such growth processes are not mirroring what goes on in the brain.

For further discussion of the properties of and suggested improvements to back-propagation, see Haykin (1999, chap. 4).