Gradient Descent Training Rule: The Details

1 For Perceptrons

The whole idea behind gradient descent is to gradually, but consistently, decrease the output error by adjusting the weights. The trick is to figure out HOW to adjust the weights. Intuitively, we know that if a change in a weight will increase (decrease) the error, then we want to decrease (increase) that weight. Mathematically, this means that we look at the derivative of the error with respect to the weight: $\frac{\partial E}{\partial w_{ij}}$, which represents the change in the error given a unit change in the weight.

Once we find this derivative, we will update the weight via the following:

$$\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}} \tag{1}$$

This essentially represents the distance times the direction of change. The distance, η, is a standard parameter in neural networks and often called the learning rate. In more advanced algorithms, this rate may gradually decrease during the epochs of the training phase.

If we update all the weights using this same formula, then this amounts to moving in the direction of steepest descent along the error surface - hence the name, gradient descent.

The above equation is easy, and it captures our basic intuitions: decrease (increase) a weight that positively (negatively) contributes to the error. Unfortunately, computing $\frac{\partial E}{\partial w_{ij}}$ is not so trivial.

For the following derivation, the perceptron depicted in Figure 1 will be used.

First we need to compute the error, and in the case of a perceptron or a row of independent perceptrons, we can focus on the error at any particular output node, i. A standard metric is the sum of squared errors (SSE):

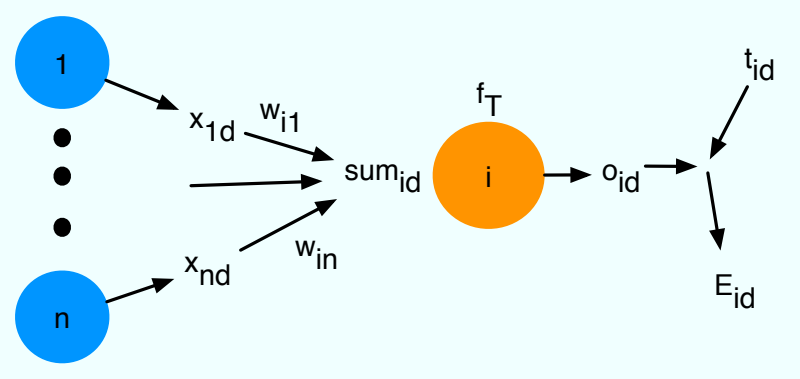

Figure 1: Simple perceptron with n weighted input lines

$$E_i = \frac{1}{2} \sum_{d \in D} (t_{id} - o_{id})^2 \tag{2}$$

where t_{id} and o_{id} are the desired/target and actual outputs, respectively, at node i on example data instance d.

So, to find $\frac{\partial E_i}{\partial w_{ij}}$, we compute:

$$\frac{\partial \left(\frac{1}{2} \sum_{d \in D} (t_{id} - o_{id})^2 \right)}{\partial w_{ij}} \tag{3}$$

Taking the derivative inside the summation and using standard calculus:

$$\frac{1}{2} \sum_{d \in D} 2(t_{id} - o_{id}) \frac{\partial (t_{id} - o_{id})}{\partial w_{ij}} = \sum_{d \in D} (t_{id} - o_{id}) \frac{\partial (-o_{id})}{\partial w_{ij}} \tag{4}$$

The t_{id} term disappears from the derivative, since $\frac{\partial t_{id}}{\partial w_{ij}} = 0$, i.e. the target value is completely independent of the weight. This makes sense, since the target is set from outside the system.

Now, the output value o_{id} is equal to the transfer function for the perceptron, f_T, applied to the sum of weighted inputs to the perceptron (on example instance d), sum_{id}. So we can rewrite as:

$$\sum_{d \in D} (t_{id} - o_{id}) \frac{\partial (-f_T(sum_{id}))}{\partial w_{ij}} \tag{5}$$

where:

$$sum_{id} = \sum_{k=1}^{n} w_{ik} x_{kd} \tag{6}$$

Here, summing over the k means summing over the n inputs to node i. That is, we sum the weighted outputs of all of i's suppliers.

Using the Chain Rule from calculus, which states that:

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g(x)} \times \frac{\partial g(x)}{\partial x} \tag{7}$$

we can calculate the derivative of the transfer function with respect to the weight as follows:

$$\frac{\partial (f_T(sum_{id}))}{\partial w_{ij}} = \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}} \tag{8}$$

Now, the first term on the right-hand-side will vary, depending upon the transfer function. Shortly, we will compute it for a few different functions. But first, we can (easily) compute the second term of the product:

$$\frac{\partial sum_{id}}{\partial w_{ij}} = \frac{\partial \left(\sum_{k=1}^{n} w_{ik} x_{kd}\right)}{\partial w_{ij}} = \frac{\partial \left(w_{i1}x_{1d} + w_{i2}x_{2d} + \ldots + w_{ij}x_{jd} + \ldots + w_{in}x_{nd}\right)}{\partial w_{ij}} \tag{9}$$

$$= \frac{\partial(w_{i1}x_{1d})}{\partial w_{ij}} + \frac{\partial(w_{i2}x_{2d})}{\partial w_{ij}} + \ldots + \frac{\partial(w_{ij}x_{jd})}{\partial w_{ij}} + \ldots + \frac{\partial(w_{in}x_{nd})}{\partial w_{ij}} = 0 + 0 + \ldots + x_{jd} + \ldots + 0 = x_{jd} \tag{10}$$

This makes perfect sense: the change in the sum given a unit change in a particular weight, w_{ij}, is simply x_{jd}.

Next, we calculate $\frac{\partial f_T(sum_{id})}{\partial sum_{id}}$ for different transfer functions, f_T.

1.1 The Identity Transfer Function

First, if f_T is the identity function, then $f_T(sum_{id}) = sum_{id}$, and thus:

$$\frac{\partial f_T(sum_{id})}{\partial sum_{id}} = 1 \tag{11}$$

So in that simple case:

$$\frac{\partial(f_T(sum_{id}))}{\partial w_{ij}} = \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}} = 1 \times x_{jd} = x_{jd} \tag{12}$$

So putting everything together, for the identity transfer function, the derivative of the error with respect to weight w_{ij} is:

$$\frac{\partial E_i}{\partial w_{ij}} = \frac{\partial \left(\frac{1}{2}\sum_{d\in D}(t_{id} - o_{id})^2\right)}{\partial w_{ij}} = \sum_{d\in D}(t_{id} - o_{id})\frac{\partial(-o_{id})}{\partial w_{ij}} = \sum_{d\in D}(t_{id} - o_{id})\frac{\partial(-f_T(sum_{id}))}{\partial w_{ij}} \tag{13}$$

$$= -\sum_{d\in D}\left((t_{id} - o_{id})\frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}}\right) = -\sum_{d\in D}\left((t_{id} - o_{id})(1)x_{jd}\right) = -\sum_{d\in D}(t_{id} - o_{id})x_{jd} \tag{14}$$

Thus, the weight update for a neural network using identity transfer functions is:

$$\Delta w_{ij} = -\eta\frac{\partial E_i}{\partial w_{ij}} = \eta\sum_{d\in D}(t_{id} - o_{id})x_{jd} \tag{15}$$

One of the early types of simple neural units, called an Adaline, used this update rule even though step functions were used as transfer functions. So even though the step function is not everywhere differentiable, by treating it like an identity function, you can still get a good estimate of $\frac{\partial E_i}{\partial w_{ij}}$.

1.2 The Sigmoidal Transfer Function

The sigmoidal function is very popular for neural networks, because it performs very similar to a step function, but it is everywhere differentiable. Thus, we can compute $\frac{\partial f_T(sum_{id})}{\partial sum_{id}}$ for the sigmoidal, although we cannot for a standard step function (due to the discontinuity at the step point).

The standard form of the sigmoidal is:

$$f_T(sum_{id}) = \frac{1}{1 + e^{-sum_{id}}} \tag{16}$$

So

$$\frac{\partial f_T(sum_{id})}{\partial sum_{id}} = \frac{\partial \left((1 + e^{-sum_{id}})^{-1}\right)}{\partial sum_{id}} = (-1)\frac{\partial(1 + e^{-sum_{id}})}{\partial sum_{id}}(1 + e^{-sum_{id}})^{-2} \tag{17}$$

$$= (-1)(-1)e^{-sum_{id}}(1 + e^{-sum_{id}})^{-2} = \frac{e^{-sum_{id}}}{(1 + e^{-sum_{id}})^2} \tag{18}$$

But, this is the same as:

$$f_T(sum_{id})(1 - f_T(sum_{id})) = o_{id}(1 - o_{id}) \tag{19}$$

Summarizing the result so far:

$$\frac{\partial f_T(sum_{id})}{\partial sum_{id}} = \frac{e^{-sum_{id}}}{(1 + e^{-sum_{id}})^2} = f_T(sum_{id})(1 - f_T(sum_{id})) = o_{id}(1 - o_{id}) \tag{20}$$

Now we can compute $\frac{\partial E_i}{\partial w_{ij}}$ for neural networks using sigmoidal transfer functions:

$$\frac{\partial E_i}{\partial w_{ij}} = \sum_{d \in D}(t_{id} - o_{id})\frac{\partial(-f_T(sum_{id}))}{\partial w_{ij}} = -\sum_{d \in D}\left((t_{id} - o_{id})\frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}}\right) = -\sum_{d \in D}(t_{id} - o_{id})o_{id}(1 - o_{id})x_{jd} \tag{21}$$

So, the weight update for networks using sigmoidal transfer functions is:

$$\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}} = \eta \sum_{d \in D} (t_{id} - o_{id}) o_{id} (1 - o_{id}) x_{jd} \tag{22}$$

1.3 Batch versus Incremental Updating

Notice that the previous update rules for w_{ij}, for both the identity and sigmoidal functions, involve batch processing in that we compute the total error for all the training instances ($d \in D$) before we update the weights.

An alternative is to update the weights after EACH example has passed through the network. In this case, the update rules are:

$$\eta (t_{id} - o_{id}) x_{jd} \tag{23}$$

for the identity function, and for the sigmoidal:

$$\eta (t_{id} - o_{id}) o_{id} (1 - o_{id}) x_{jd} \tag{24}$$

In these equations, the *error terms* associated with each **node**, i, on instance d are:

$$(t_{id} - o_{id}) \tag{25}$$

for the identity function, and for the sigmoidal:

$$(t_{id} - o_{id}) o_{id} (1 - o_{id}) \tag{26}$$

Clearly, one should choose a much lower value of the learning rate, η, for incremental than for batch processing.

For batch processing with a single layer of output perceptrons, and thus only one layer of weights to learn, it suffices to simply sum the products of x_{jd} and the error term over all the training data, and then to add Δw_{ij} to w_{ij} at the end of the training epoch.

The main drawback of incremental updating is that the final weight values can be dependent upon the **order** of presentation of the examples. As explained in a later section, the computational effort involved in incremental and batch processing are approximately equal.

2 Gradient Descent for Multi-Layer Neural Networks

For multi-layer networks, the relationship between the error term and ANY weight ANYWHERE in the network needs to be calculated. This involves propagating the error term at the output nodes backwards

through the network, one layer at a time. At each layer, m, an error term (similar to those discussed above) is computed for each node,i, in the layer. Call this δ_{id}. It represents the negative of the effect of sum_{id} on the output error for example d, that is:

$$\delta_{id} = -\frac{\partial E_d}{\partial sum_{id}} \tag{27}$$

Then, going backwards to layer m-1, for each node, j, in that layer, we want to compute $\frac{\partial E_d}{\partial sum_{jd}}$. This is accomplished by computing the product of:

- the influence of j's input sum upon j's output: $\frac{\partial f_T(sum_{jd})}{\partial sum_{jd}}$, which is just $o_{jd}(1 - o_{jd})$ for a sigmoidal f_T.
- the sum of the contributions of j's output value to the total error via each of j's downstream neighbors.

Thus, the complete equation for computing node j's contribution to the error is:

$$\frac{\partial E_d}{\partial sum_{jd}} = \frac{\partial o_{jd}}{\partial sum_{jd}} \sum_{k=1}^{n} \frac{\partial sum_{kd}}{\partial o_{jd}} \frac{\partial E_d}{\partial sum_{kd}} \tag{28}$$

In this section, the variable o is used to denote the output values of all nodes, not just those in the output layer.

The relationships between the different partial derivatives is easiest to see with the diagram in Figure 2:

Just as $\frac{\partial sum_{kd}}{\partial w_{kj}} = o_{jd}$, since most of the terms in the summation are zero (as shown in equations 9 and 10), we also have:

$$\frac{\partial sum_{kd}}{\partial o_{jd}} = w_{kj} \tag{29}$$

Substituting equations 27 and 29 into equation 28 yields :

$$\delta_{jd} = -\frac{\partial E_d}{\partial sum_{jd}} = -\frac{\partial o_{jd}}{\partial sum_{jd}} \sum_{k=1}^{n} w_{kj}(-\delta_{kd}) = \frac{\partial o_{jd}}{\partial sum_{jd}} \sum_{k=1}^{n} w_{kj}\delta_{kd} \tag{30}$$

This illustrates how the error terms at level m-1 (i.e. the δ_{jd}) are based on backpropagated error terms from level m (i.e., the δ_{kd}), with each layer-m error adjusted by the weight along the arc from node j to node k.

Assuming sigmoidal transfer functions:

$$\frac{\partial o_{jd}}{\partial sum_{jd}} = \frac{\partial f_T(sum_{jd})}{\partial sum_{jd}} = o_{jd}(1 - o_{jd}) \tag{31}$$

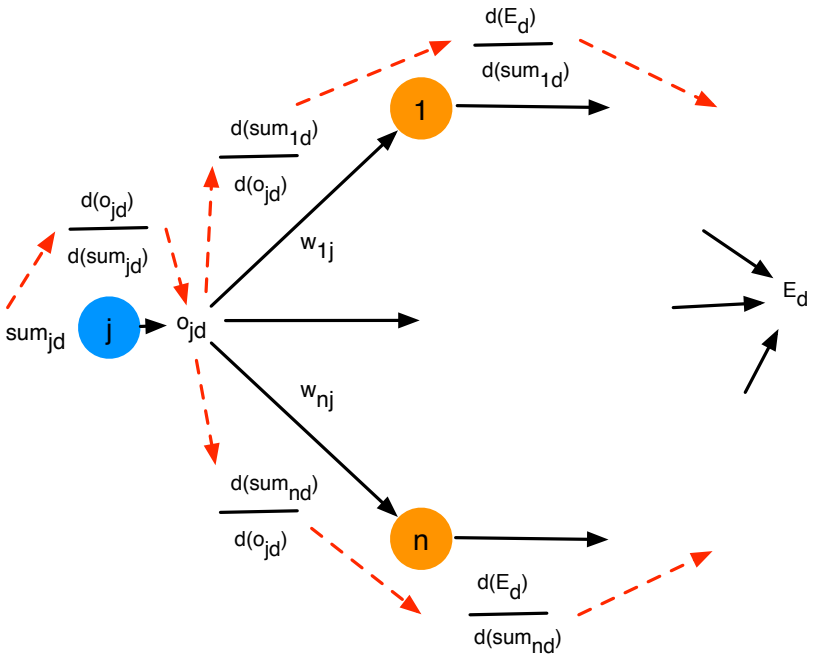

Figure 2: Multi-layer network displaying basic backpropagation terms and their relationships

Thus,

$$\delta_{jd} = o_{jd}(1 - o_{jd}) \sum_{k=1}^{n} w_{kj}\delta_{kd} \tag{32}$$

This provides an operational definition for implementing the recursive backpropagation algorithm for networks using sigmoidal transfer functions:

1. For each output node, i, compute its error term using:

$$\delta_{id} = (t_{id} - o_{id})o_{id}(1 - o_{id}) \tag{33}$$

2. Working backwards, layer by layer, compute the error terms for each internal node, j, as:

$$\delta_{jd} = o_{jd}(1 - o_{jd}) \sum_{k=1}^{n} w_{kj}\delta_{kd} \tag{34}$$

   where the k are the downstream neighbor nodes.

Once we know $\frac{\partial E_d}{\partial sum_{id}} = -\delta_i$, the computation of $\frac{\partial E_d}{\partial w_{ij}}$ is quite easy. This stems from the fact that the only effect of w_{ij} upon the error is via its effect upon sum_{id}:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial sum_{id}}{\partial w_{ij}} \times \frac{\partial E_d}{\partial sum_{id}} = \frac{\partial sum_{id}}{\partial w_{ij}} \times (-\delta_{id}) = -o_{jd}\delta_{id} \tag{35}$$

So, given an error term, δ_i, for node i, the update of w_{ij} for all nodes j feeding into i is simply:

$$\Delta w_{ij} = -\eta \frac{\partial E_d}{\partial w_{ij}} = -\eta(-o_{jd}\delta_{id}) = \eta \delta_{id} o_{jd} \tag{36}$$

Obviously, the procedure is similar for other types of transfer functions, such as the linear or inverse hyperbolic tangent function. It is only $\frac{\partial f_T(sum_{jd})}{\partial sum_{jd}}$ that will vary.

3 Batch Processing for Multilayer Neural Networks

As shown in equations 33 and 34, when performing incremental weight updates, the error terms for nodes are a function of the output values of those nodes on a particular training case. Then, as shown in equations 35 and 36, the contribution of w_{ij} to the error (which determines Δw_{ij}) is also a function of the output value of node j on that same training case.

Hence, for batch processing, it will not suffice to update the node error terms after each training case and then, at the end of the epoch, update the weights based on the node errors. Instead, we need to keep track of the Δw_{ij} after each training case, *but we do not actually update the weight until the end of the epoch.* So we sum the recommended weight changes after each training instance, but we only modify the weight at epoch's end. Formally, we are computing:

$$\sum_{d \in D} \Delta_d w_{ij} \tag{37}$$

Notice that the recommended changes can be positive or negative, so they may partially cancel one another out.

Unfortunately, this means that batch processing does not save computational effort, since we are essentially performing full backpropagation after each training instance. Although we do not update w_{ij} on each instance, we update the sum of the changes (equation 37), which is just as much work, computationally.

However, practically speaking, it makes more sense to change w_{ij} once, at the end of an epoch, than to raise and lower it many times during the course of that same epoch. Also, changing weights only at the end of an epoch eliminates any sensitivity to training order in the learning procedure.