Back Propagation Algorithm with Proofs

Luke Hunsberger

This document presents the back propagation algorithm for neural networks along with supporting proofs. The notation sticks closely to that used by Russell & Norvig in their Artificial Intelligence textbook (3rd edition, chapter 18). The web page at `http://www.speech.sri.com/people/anand/771/html/node37.html` provided the general framework for the proofs.

1 Training by Gradient Descent

Think of a feed-forward neural network as implementing a function. In particular, it is a function of however many inputs are in the input layer to however many outputs are in the output layer. Such functions are called *vector* functions because the set of inputs can be collected into an *input vector* and the set of outputs can be collected into an *output vector*. The weights on the edges in the neural network can be thought of as parameters that determine the function implemented by a neural network. When we change the weights on the edges, the neural network typically implements a different function. Typically, our goal is to generate a set of weights that will enable the neural network to closely simulate a desired function. Typically, we don't know everything about the desired function. Instead, we only have a set of *input-output* pairs, called *training examples*. For example, we might know that a given input vector (a, b, c, d) should generate the output vector (x, y, z). We *train* the neural network by feeding the input vector (a, b, c, d) into the neurons in the input layer, seeing what output vector (p, q, r) is generated by the network in the *feed forward* phase, comparing the desired output vector (x, y, z) with the generated output vector (p, q, r), and then adjusting the weights in the network in an attempt to move the generated output vector closer to the desired output vector. The hope is that making adjustments in this way over a large number of training examples will result in a neural network whose function resembles the one implied by all of the input-output pairs.

The back-propagation algorithm tells us how to incrementally adjust the weights in response to the difference between the generated and desired output vectors for each training example. The back-propagation algorithm uses a technique called *gradient descent*.

1.1 Gradient Descent

Consider the function, $f(x, y) = w_1x^2 + w_2y$, where w_1 and w_2 are parameters that we can choose. We want to find values of w_1 and w_2 that enable this function to simulate our training examples. For simplicity, we start with $w_1 = w_2 = 0.5$. Our first training example says that the input vector $(2, 3)$ should go with the output 9. In the feed-forward phase, we plug-and-chug:

$$f(2, 3) = 0.5 * 2^2 + 0.5 * 3 = 2 + 1.5 = 3.5$$

Well, 3.5 is not very close to our desired value of 9. How should we adjust the weights of our function to get closer to the desired output value?

In general, we want to minimize the difference between the desired output value D and the generated output value G. Mathematically, it turns out to be easier to minimize the square of this difference, which leads to the same result. So, we want to minimize the squared error, $E = (D - G)^2$. For our one training example, we have $(D - G)^2 = (9 - 3.5)^2 = 30.25$. That's a big number. The question is: how should we incrementally adjust the weights, w_1 and w_2, to nudge our function f to generate an output closer to the desired output?

Well, calculus to the rescue! We begin by computing the *gradient* of the squared error, which requires computing the partial derivatives of E with respect to the parameters w_1 and w_2. (Computing a partial derivative of E with respect to w_1 is done by treating E as a function of w_1, where all other "variables" are treated as constants.) First, $\frac{\partial E}{\partial w_1} = \frac{\partial}{\partial w_1}(D - f(x, y))^2 = \frac{\partial}{\partial w_1}(D - (w_1x^2 + w_2y))^2 = 2(D - (w_1x^2 + w_2y))(-x^2)$. Evaluating this partial derivative at $x = 2$ and $y = 3$, and $w_1 = w_2 = 0.5$ yields $\frac{\partial E}{\partial w_1} = 2(9 - (0.5(4) + 0.5(3)))(-4) = 2(9 - 2 - 1.5)(-4) = 2(5.5)(-4) = -44$. Similarly, $\frac{\partial E}{\partial w_2} = \frac{\partial}{\partial w_2}(D - f(x, y))^2 = \frac{\partial}{\partial w_2}(D - (w_1x^2 + w_2y))^2 = 2(D - (w_1x^2 + w_2y))(-y)$. Evaluating this partial derivative at $x = 2$ and $y = 3$ yields $\frac{\partial E}{\partial w_2} = 2(9 - (0.5(4) + 0.5(3)))(-3) = 2(9 - 2 - 1.5)(-3) = 2(5.5)(-3) = -33$. Putting these two partial derivatives together yields the vector $(-44, -33)$—which is the gradient of E. This vector points in the direction of maximum instantaneous *increase* of E. The negative of this vector (i.e., $(44, 33)$) points in the direction of maximum instantaneous *decrease* of E. Since we want to *minimize* the squared error, E, we want to move the values of w_1 and w_2 a tiny bit in the direction of $(44, 33)$. This can be done by adding 0.044 to w_1 and 0.033 to w_2. (Notice that we have multiplied each component of the negative gradient by the same fraction, $\frac{1}{100}$. The choice of fraction is arbitrary; but we use a small

fraction because we don't want to let this one example influence the weights overly much.) Okay, so we get our new weights: $w_1 = 0.5 + 0.044 = 0.544$ and $w_2 = 0.5 + 0.033 = 0.533$.

If you want, you can verify that the output generated using these new weights is slightly closer to the desired output that is the output generated using the original weights:

Original: $f(2,3) = 0.5(2^2) + 0.5(3) = 2 + 1.5 = 3.5$

New: $f(2,3) = 0.544 * (2^2) + 0.533(3) = 3.775$

Notice that the new value, 3.775, is slightly closer to the desired value, 9. Although it doesn't seem like we've made much progress; keep in mind that this is only the result of *one* training example. Also, given that we only moved the values of w_1 and w_2 a little bit, we got the biggest bang for our buck by moving in the direction of the negative gradient. (For fun, you might see what happens if you move in the direction of the gradient. In that case, the new value of f will be further from the desired value.)

Now, if you're not familiar with derivatives, let alone partial derivatives, this may have seemed pretty complicated. Take a deep breath. Although a neural network typically involves *way more* variables; the same basic approach is used to train the network (i.e., to incrementally adjust the weights to move the generated output closer to the desired output.

2 Neural Network Notation

A neuron, N_j, is as described in Russell & Norvig's Fig. 18.19.

- $N_1, \ldots, N_n$ are the neurons that provide inputs to the neuron N_j.
- $a_1, \ldots, a_n$ are the outputs of neurons $N_1, \ldots, N_n$, which are then fed as inputs to N_j. In addition, $a_0 = 1$ is an input fixed at the value 1.
- $w_{0,j}, \ldots, w_{n,j}$ are the *weights* corresponding to the inputs, $a_0, a_1, \ldots, a_n$. Thus, $w_{i,j}$ is the weight on the link (or directed edge) from neuron N_i to neuron N_j.
- $z_j = \Sigma_{i=0}^{n} w_{i,j} a_i$ is the weighted sum of the inputs to neuron N_j. (Russell & Norvig call this in_j.)
- $g(x) = \frac{1}{1-e^{-x}}$ is the sigmoid *activation* function used by each neuron. In particular, the output of neuron N_j, denoted by a_j, is given by: $a_j = g(z_j) = g(\Sigma_{i=0}^{n} w_{i,j} a_i)$.

Okay, so we are now thinking of our neural network as implementing a vector function: each vector of input values generates a corresponding vector of output values. We have a bunch of input-output training examples that we want to use to incrementally adjust the weights in our network so that the function implemented by the network generates output vectors that are "not that far" from the desired output vectors in our training examples. In particular, we want to minimize the squared error. And, of course, we'll use gradient descent.

Okay, let $\mathbf{x}$ be a vector of input values, and $\mathbf{y}$ be the corresponding vector of output values in our training example. Let $\mathbf{h_w}$ be the function implemented by our neural network. Notice that $\mathbf{h_w}$ is parameterized by the weights. (We can start with randomly selected values for our weights.) Thus, $\mathbf{h_w}(\mathbf{x})$ is the output vector generated by our neural network when given the input vector $\mathbf{x}$. (Note that the values in the output vector correspond to the a_j values for the neurons, N_j, in the output layer.) The squared error, E, is therefore given by: $E = (y - \mathbf{h_w}(\mathbf{x}))^2$. The gradient of E is computed by computing the partial derivative of E with respect to *each* weight. (Remember, there is a weight on every single edge in the neural network!) But that's okay... we'll just do it.

3 Deriving the Back Propagation Algorithm

3.1 Base Case: The weights on edges feeding into an output neuron

Suppose N_j is an output neuron (i.e., a neuron in the output layer), and that N_i is some neuron that feeds N_j (i.e., the output of N_i provides one of the inputs to N_j). Consider the weight $w_{i,j}$ on the edge from N_i to N_j.

$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial w_{i,j}}$, by the Chain Rule

Now, taking each part separately:

$\frac{\partial E}{\partial z_j} = \frac{\partial}{\partial z_j}(y_j - a_j)^2$, since the j^{th} component of the squared error, E, is $(y_j - a_j)$.[1]

$= \frac{\partial}{\partial z_j}(y - g(z_j))^2$, since $a_j = g(z_j)$

[1] Only the j^{th} component of the squared error is affected by the weight $w_{i,j}$; thus only the j^{th} component appears in the partial derivative, $\frac{\partial E}{\partial z_j}$.

$= -2(y - g(z_j))g'(z_j)$, by the Chain Rule

$= -2(y - g(z_j))g(z_j)(1 - g(z_j))$ since $g'(z_j) = g(z_j)(1 - g(z_j)) \ldots$ [2]

$= -2(y - a_j)a_j(1 - a_j)$, since $a_j = g(z_j)$

and:

$\frac{\partial z_j}{\partial w_{i,j}} = \frac{\partial}{\partial w_{i,j}}(\Sigma_q a_q w_{q,j}) = a_i$, since the only term that involves $w_{i,j}$ is $a_i w_{i,j}$.

Thus, putting things back together:

$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{i,j}} = -2(y - a_j)a_j(1 - a_j)a_i$

Now, for convenience, we define Δ_j as follows:

$\Delta_j = -\frac{\partial E}{\partial z_j} = 2(y - a_j)a_j(1 - a_j)$.

Thus, we have:

$\frac{\partial E}{\partial w_{i,j}} = -\Delta_j a_i$

Recall that the *negative* of the partial derivative tells us the relative amount by which we want to incrementally adjust the weight $w_{i,j}$. Thus, we use the following weight update rule for $w_{i,j}$:

$w_{i,j}$ `+=` $\alpha\Delta_j a_i$, where α is some small value (e.g., 0.01) that regulates the sensitivity of our adjustments to the weights.

3.2 Recursive Case: The weights on edges feeding into a neuron in a hidden layer

Now suppose N_j is a neuron in a hidden layer. As before, let N_i be one of the neurons that provides an input for N_j; and let $w_{i,j}$ be the weight on the edge from N_i to N_j. We still want to compute the partial derivative of the squared error, E, with respect to the weight $w_{i,j}$. But to do this, we now need to consider the edges *leaving* N_j on the output side; and consider all of the neurons N_k for which N_j provides inputs. The weight $w_{i,j}$ only affects the squared error, E, through its effect on the neurons N_k. Thus, when computing the partial derivative of E with respect to $w_{i,j}$ we can view E as a function of the input values, z_k, of the neurons, N_k. The z_k values in turn depend on the output, a_j, of N_j. The a_j value in turn depends on its input z_j, which in turn depends on $w_{i,j}$. Thus, according to a very "chainy" use of the Chain Rule, we have:

$\frac{\partial E}{\partial w_{i,j}} = \Sigma_k(\frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial a_j}\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{i,j}})$

Now, addressing each of these separately:

$\frac{\partial E}{\partial z_k} = -\Delta_k$, by definition of Δ_k

$\frac{\partial z_k}{\partial a_j} = \frac{\partial}{\partial a_j}(\Sigma_s a_s w_{s,k}) = w_{j,k}$, since the only term in the sum that involves a_j is $a_j w_{j,k}$

$\frac{\partial a_j}{\partial z_j} = \frac{\partial}{\partial z_j}g(z_j) = g'(z_j) = g(z_j)(1 - g(z_j)) = a_j(1 - a_j)$

$\frac{\partial z_j}{\partial w_{i,j}} = a_i$, since the only term in the sum, z_j, that involves $w_{i,j}$ is $a_i w_{i,j}$.

Thus, we get:

$\frac{\partial E}{\partial w_{i,j}} = \Sigma_k(-\Delta_k w_{j,k}a_j(1 - a_j)a_i)$

$= -a_j(1 - a_j)\Sigma_k(\Delta_k w_{j,k})a_i$

$= -\Delta_j a_i$, where: $\Delta_j = a_j(1 - a_j)\Sigma_k(\Delta_k w_{j,k})$

And, we get the following weight-update rule (following the gradient descent technique):

$w_{i,j}$ `+=` $\alpha\Delta_j a_i$, where α is a small numerical value representing the *learning rate*.

Notice that, given the values Δ_k computed in the base case, we can generate the Δ_j values needed for the partial derivatives in the next layer. Similarly, given the Δ_j values computed for the first hidden layer, we can generate the Δ_i values needed for the partial derivatives in the second hidden layer (counting right to left). And so on.

[2] One of the reasons for using the logistic function, g, is that its derivative satisfies the following nice equation: $g'(z) = g(z)(1 - g(z))$, which can be verified by direct computation.

4 The Back Propagation Algorithm

Given an input-output pair, $(\mathbf{x}, \mathbf{y})$. Use the current set of weights in the network to generate the output vector $\mathbf{h}(\mathbf{x})$, whose individual values comprise the a_k values for the neurons, N_k, in the output layer. Next, compute the Δ_k values for each neuron, N_k, in the output layer and use those values to update the weights, $w_{j,k}$, on the incoming edges for those neurons, using the update rule seen in the base case. Next, for the neurons in the first hidden layer (just behind the output layer), compute the Δ_j values and similarly use those values to update the incoming edges for those neurons. And so on, for each neuron in the next layer, compute the corresponding Δ value and use it update the weights on the incoming edges. Eventually, you will reach the input layer, which has no incoming edges; and hence no weights to update.

Do this for each input-output pair in the training examples.