Using Backpropagation to Compute the Gradient

It is now time to explore how to apply gradient descent to a multilevel network. To keep it simple, we begin with one of the simplest multilevel networks that we can think of, with only a single neuron in each layer. We assume two inputs to the first neuron. The network is shown in Figure 3-5.

We have named the two neurons G and F, where G has three adjustable weights, and F has two adjustable weights. In total, we have a network with five adjustable weights. We want a learning algorithm to automatically find values for these weights that result in the network implementing the desired functionality. The figure also contains something that looks like a neuron to the very right, but this is not a part of the network. Instead, it represents a function needed to determine how right or wrong the network is, which is needed for learning. (This is described in more detail shortly).

We use a weight-naming convention whereby the first letter in the subscript represents the source layer and the second letter represents the destination layer (the layer that the weight is a part of). The digit represents the input number, where 0 is the bias input. That is, w_{xg2} means input number 2 to the *g*-neuron, which receives its input from layer *x*. For consistency, we use the same naming

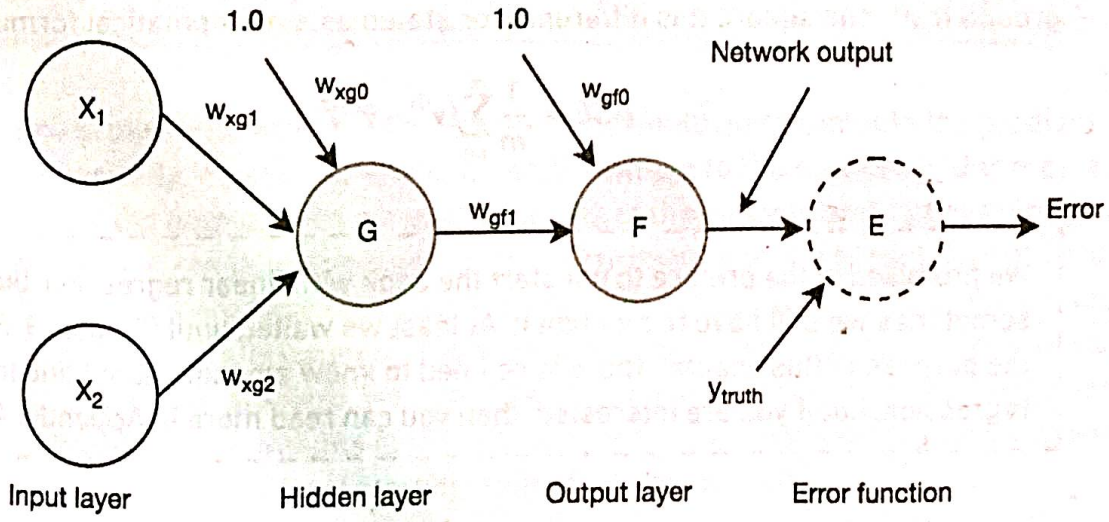


Figure 3-5 Simple two-layer network used to explain backpropagation. The last unit (dashed) is not a part of the network but represents the error function that compares the output to ground truth.

convention for the bias term as for the other terms, although technically it does not have a source layer.

The perceptron studied in Chapter 2 used the sign function as its activation function. As already noted, the sign function is not differentiable in all points. In this chapter, we use activation functions that are differentiable in all points. Neuron G uses tanh as an activation function, and neuron F uses S (the logistic sigmoid function). This means that a small change to any of the five weights will result in only a small change to the output. When we used a sign function, a small change in a weight would not change anything until the change was big enough to make one perceptron flip, in which case all bets were off because it could easily flip dependent neurons as well.

Taking a step back, our neural network implements the following function:

$$\hat{y} = S\left(w_{gf0} + w_{gf1} \tanh\left(w_{xg0} + w_{xg1} x_1 + w_{xg2} x_2\right)\right)$$

We know of an algorithm (gradient descent) that can be used to minimize a function. To make use of this algorithm we want to define an *error function*, also known as a *loss function*, which has the property that if it is minimized, then the overall network produces the results that we desire. Defining and minimizing error functions is not unique to neural networks but is used in many other contexts as well. One popular error function that comes to mind is the mean squared error (MSE), which we introduced in Chapter 2, and you might already have been familiar with because it is used for linear regression. The way to compute it is to, for each learning example, subtract the predicted value from the ground truth and square this difference, or stated as a mathematical formula:

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2$$

We promised in the preface to not start the book with linear regression, but sometimes we still have to mention it. At least we waited until Chapter 3. For the purpose of this chapter, there is no need to know anything about linear regression, but if you are interested, then you can read more in Appendix A.

Remember that **error function** and **loss function** are two names for the same thing.

There are multiple loss functions to choose among. We use MSE for historical reasons in this section, but in reality, it is not a good choice in combination with a sigmoid activation function.

In other words, MSE is the *mean* (sum divided by *m*) of the squared error $(y-\hat{y})^2$ for all *m* training examples. In Chapter 6, we will learn that using MSE as an error function with this type of neural network is not optimal, but we will use it for now just to keep things familiar and simple. That is, assuming a single training example, we want to minimize our loss function $(y-\hat{y})^2$, where $\hat{y}$ was defined above and *y* is a part of the training example. The following equation combines the formula for MSE with the network function to arrive at the expression of the error function that we want to minimize for a single learning example:

$$Error = \left(y - S\left(w_{gf0} + w_{gf1}\tanh\left(w_{xg0} + w_{xg1}x_1 + w_{xg2}x_2\right)\right)\right)^2$$

We know that we can minimize this function using gradient descent by computing the gradient of the loss function (∇Error) with respect to our weights ***w***, then multiplying this gradient by the learning rate (η), and then subtract this result from the initial guess of our weights. That seems straightforward enough, except that computing the gradient of our loss function seems a little bit scary.

A key thing to remember here is that because we want to adjust the weights, we view the **weights *w* as variables**, and we view the **inputs *x* as constants;** that is, the gradient is computed with respect to ***w*** and not with respect to ***x***.

One brute-force way to solve this problem would be to compute the gradient numerically. We could present an input example to the network and compute and record the output. Then we add Δw to one of the weights and compute the new output and can now compute Δy. An approximation of the partial derivative is now $\Delta y/\Delta w$. Once we have repeated this procedure for all weights, we have computed the gradient. Unfortunately, this is an extremely computationally intensive way of computing our gradient. We need to run through the network n+1 times, where n is the number of weights in the network (the +1 is necessary because we need the baseline output without any adjustments to the weights).

The backpropagation algorithm solves this problem in an elegant way by computing the gradient analytically in a computationally efficient manner. The

starting point is to decompose our equation into smaller expressions. We start with a function that computes the input to the activation function of neuron G:

$$z_g(w_{xg0}, w_{xg1}, w_{xg2}) = w_{xg0} + w_{xg1}x_1 + w_{xg2}x_2$$

Next is the activation function for neuron G:

$$g(z_g) = tanh(z_g)$$

It is followed by the input to the activation function of neuron F:

$$z_f(w_{gf0}, w_{gf1}, g) = w_{gf0} + w_{gf1}g$$

This in turn is followed by the activation function of neuron F:

$$f(z_f) = S(z_f)$$

Finally, we conclude with the error function:

$$e(f) = \frac{(y-f)^2}{2}$$

Looking closely at the formulas, you might wonder where the 2 in the denominator of the error (*e*) came from. We added that to the formula because it will simplify the solution further down. This is legal to do because the values of variables that will minimize an expression do not change if we divide the expression by a constant.

Overall, the error function that we want to minimize can now be written as a composite function:

$$Error(w_{gf0}, w_{gf1}, w_{xg0}, w_{xg1}, w_{xg2}) = e \circ f \circ z_f \circ g \circ z_g$$

That is, *e* is a function of *f*, which is a function of z_f, which is a function of *g*, which is a function of z_g. Function z_f is not only a function of *g* but also of the two variables w_{gf0} and w_{gf1}. That was already shown further up in the definition of z_f. Similarly, z_g is a function of the three variables w_{xg0}, w_{xg1}, and w_{xg2}.

> Again, note that this formula contains neither **x** nor *y* because they are not treated as variables but as constants for a given training example.

Now that we have stated our error function as a composition of multiple functions, we can make use of the chain rule. We use that to compute the partial derivative

of the error function e with respect to the input variables w_{gf0}, w_{gf1}, w_{xg0}, w_{xg1}, and w_{xg2}. Let us start with the first one: We compute the partial derivative of e with respect to the variable w_{gf0}. We do this by simply regarding the other variables as constants, which also implies that the function g is a constant, and we then have a function

$$Error = e \circ f \circ z_f\left(w_{gf0}\right)$$

Applying the chain rule now yields:

$$\frac{\partial e}{\partial w_{gf0}} = \frac{\partial e}{\partial f} \cdot \frac{\partial f}{\partial z_f} \cdot \frac{\partial z_f}{\partial w_{gf0}} \qquad (1)$$

Doing the same exercise, but with respect to w_{gf1}, yields

$$\frac{\partial e}{\partial w_{gf1}} = \frac{\partial e}{\partial f} \cdot \frac{\partial f}{\partial z_f} \cdot \frac{\partial z_f}{\partial w_{gf1}} \qquad (2)$$

Moving on to w_{xg0}, w_{xg1}, and w_{xg2} results in expressions with two more functions in the composite function because the functions g and z_g are no longer treated as constants:

$$Error = e \circ f \circ z_f \circ g \circ z_g$$

The resulting partial derivatives are

$$\frac{\partial e}{\partial w_{xg0}} = \frac{\partial e}{\partial f} \cdot \frac{\partial f}{\partial z_f} \cdot \frac{\partial z_f}{\partial g} \cdot \frac{\partial g}{\partial z_g} \cdot \frac{\partial z_g}{\partial w_{xg0}} \qquad (3)$$

$$\frac{\partial e}{\partial w_{xg1}} = \frac{\partial e}{\partial f} \cdot \frac{\partial f}{\partial z_f} \cdot \frac{\partial z_f}{\partial g} \cdot \frac{\partial g}{\partial z_g} \cdot \frac{\partial z_g}{\partial w_{xg1}} \qquad (4)$$

$$\frac{\partial e}{\partial w_{xg2}} = \frac{\partial e}{\partial f} \cdot \frac{\partial f}{\partial z_f} \cdot \frac{\partial z_f}{\partial g} \cdot \frac{\partial g}{\partial z_g} \cdot \frac{\partial z_g}{\partial w_{xg2}} \qquad (5)$$

One thing that sticks out when looking at the five partial derivatives is that there are a whole lot of common subexpressions. For example, the first two factors are the same in each of the five formulas, and three of the formulas share yet another two factors. This provides some intuition for why the backpropagation algorithm is an efficient way of computing the gradient. Instead of being recomputed over and over, these subexpressions are computed once and then reused for each partial derivative where they are needed.

Now let us attempt to compute one of the partial derivatives in practice. We start with number (1) above.

$$\frac{\partial e}{\partial f} = \frac{\partial \frac{(y-f)^2}{2}}{\partial f} = \frac{2(y-f)}{2} \cdot (-1) = -(y-f)$$

$$\frac{\partial f}{\partial z_f} = \frac{\partial (S(z_f))}{\partial z_f} = S'(z_f)$$

$$\frac{\partial z_f}{\partial w_{gf0}} = \frac{\partial (w_{gf0} + w_{gf1} g)}{\partial w_{gf0}} = 1$$

Note the **negative sign** for $-(y - f)$. Some texts simply flip the position of the two terms to get rid of it. In addition, some code implementations of the algorithm omit it and later compensate by using + instead of − when adjusting the weights further down.

Combining the three together, we get

$$\frac{\partial e}{\partial w_{gf0}} = -(y-f) \cdot S'(z_f)$$

There are three key observations here. First, we have all the values y, f, and z_f because y comes from the training example and the others were computed when doing the forward pass through the network. Second, the derivative of S is possible to compute because we consciously chose S as an activation function. This would not have been the case if we had stuck with the sign function. Third, not only can we compute the derivative of S but, as we saw earlier in this chapter, the derivative is a function of S itself. Therefore, we can compute the derivative from the value f that was computed during the forward pass. We revisit these three observations later in the chapter in a numerical example.

Let us now compute the partial derivative with respect to w_{gf1}, that is, number (2) presented earlier. The only difference compared to (1) is the third factor, which becomes

$$\frac{\partial z_f}{\partial w_{gf1}} = \frac{\partial (w_{gf0} + w_{gf1} g)}{\partial w_{gf1}} = g$$

Combining this with the first two factors yields

$$\frac{\partial e}{\partial w_{gf1}} = -(y-f) \cdot S'(z_f) \cdot g$$

This is the same as we had for w_{gf0} but multiplied by g, which is the output of neuron G that we already computed during the forward pass.

We can do similar exercises for the remaining three partial derivatives and arrive at all five derivatives that the gradient consists of (Equation 3-1).

$$\frac{\partial e}{\partial w_{gf0}} = -(y-f)\cdot S'(z_f) \quad (1)$$

$$\frac{\partial e}{\partial w_{gf1}} = -(y-f)\cdot S'(z_f)\cdot g \quad (2)$$

$$\frac{\partial e}{\partial w_{xg0}} = -(y-f)\cdot S'(z_f)\cdot w_{gf1}\cdot tanh'(z_g) \quad (3)$$

$$\frac{\partial e}{\partial w_{xg1}} = -(y-f)\cdot S'(z_f)\cdot w_{gf1}\cdot tanh'(z_g)\cdot x_1 \quad (4)$$

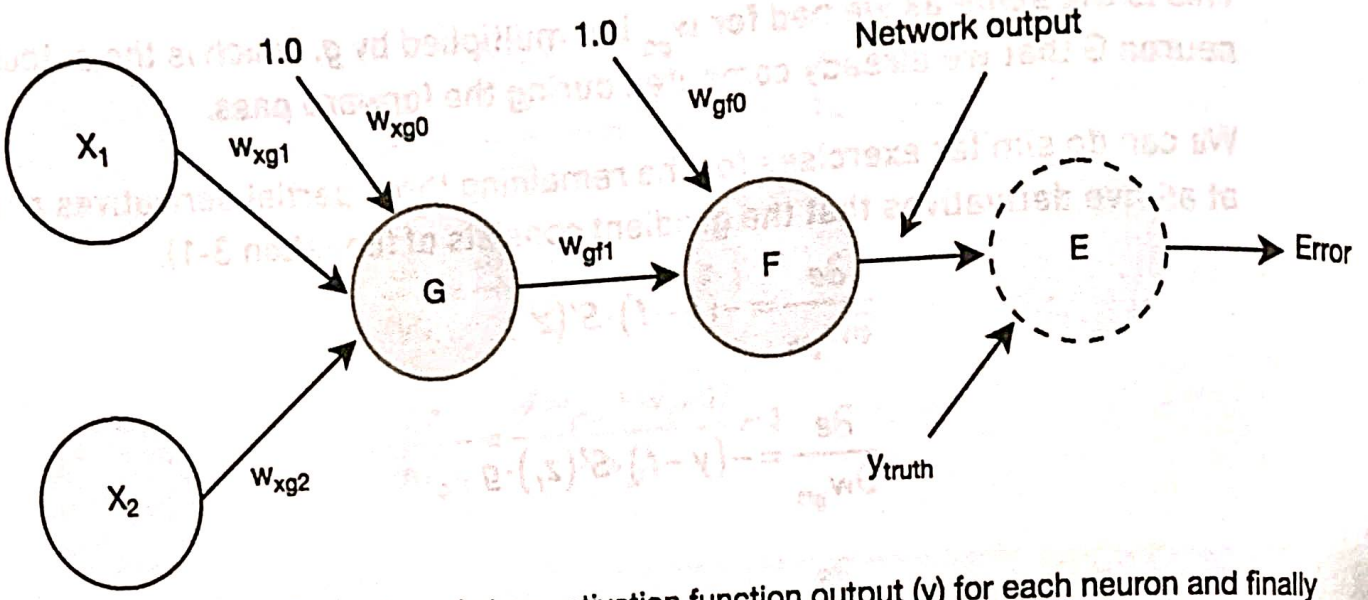
$$\frac{\partial e}{\partial w_{xg2}} = -(y-f)\cdot S'(z_f)\cdot w_{gf1}\cdot tanh'(z_g)\cdot x_2 \quad (5)$$

Equation 3-1 All five partial derivatives of the gradient

The derivative of tanh, like the derivative of S, is also simple to compute. Looking at the preceding equations, we can see a pattern. We start with the derivative of the error function and then multiply that by the derivative of the activation function for the output neuron. Let us call this product the error for the output neuron (neuron F). Now, the partial derivative with respect to an input weight for that neuron is obtained by multiplying the neuron error by the input value to that weight. In the case of the bias weight, the input value is 1, so the partial derivative is simply the neuron error. For the other weight, we multiply the neuron error by the output of the preceding neuron, which is the input to the weight.

Moving to the next (preceding) layer, we take the error for the output neuron, multiply it by the weight connecting to the preceding neuron, and multiply the result by the derivative of the activation function for the preceding neuron. We call this the error of the preceding neuron (neuron G). These computations *propagate the error backward* from the output of the network toward the beginning of the network, hence the name *backpropagation algorithm*. The full learning algorithm is shown in Figure 3-6.

So, as described, we start by applying the input example to the network to compute the current error. This is known as the *forward pass*. During this pass,



1. Forward pass: Compute and store activation function output (y) for each neuron and finally the error

Resulting stored variables: y_g y_f *Error*

2. Backward pass: Compute the derivative $e'(y_f)$ of the error function. Compute (*back propagate*) the error for each neuron by multiplying the error from the subsequent neuron (that it feeds) by the weight to that neuron and then multiply by the derivative of its own activation function. (e.g. the error for neuron *G* is $error_g = error_f * w_{gf1} * g'(z_g)$ *where* $g'(z_g)$ is the derivative of the activation function for neuron *G*). This derivative can be computed from the stored output of the activation function.

Resulting stored variables: $error_g$ $error_f$ $e'(y_f)$

3. Update weights: For each weight, subtract (*learning_rate* * *input* * *error*) where input is the input value to that weight (from network input or output from preceding neuron) and error is the error term for the neuron the weight belongs to (e.g., for weight w_{gf1} the adjustment will be $-(learning\_rate * y_g * error_f)$ where y_g is the output of neuron G).

Figure 3-6 Network learning algorithm based on gradient descent using backpropagation to compute gradient

we also store the outputs (*y*) for all neurons because we will use them during the backward pass. We then start the backward pass during which we propagate the error backward and compute and store an error term for each neuron. We need the derivative to compute this error term, and the derivative for each neuron can be computed from the stored output (*y*) for the neuron. Finally, we can use this error term together with the input values to the layer to compute the partial derivatives that are used to adjust the weights. The input values to a hidden layer are the output values from the preceding layer. The input values to the first layer are simply the *x*-values from the training example.

Backpropagation consists of the following steps:

Compute the derivative of the error function with respect to network output, and call this the **output error**. Multiply this output error by the **derivative of the activation function** of the output neuron, and call this the **error term** for that neuron. The partial derivative with respect to any weight of that neuron is the **error term** times the **input value to the weight.** The error term for the preceding neuron is the **error term for the current neuron** times the **weight between the two neurons** times the **derivative of the activation function** of the preceding neuron.

Looking at the formula for a single component of the gradient, we can see that there are a number of things that determine how much to adjust the weight when the gradient is later used for gradient descent:

- The overall error—this makes sense in that a big error should lead to a big adjustment.
- All the weights and derivatives on the path from the weight in question to the error in the end of the network—this makes sense because if one or more weights or derivatives on this path will suppress the effect of this weight change, then it is not helpful to change it.
- The input to the weight in question—this makes sense because if the input to the weight is small, then adjusting the weight will not have much of an effect.

The current value of the weight to adjust is not a part of the formula. Overall, these observations make intuitive sense for how to identify which weights should get significant adjustments.

To make this more concrete, we now walk through a numerical example for the forward pass, backward pass, and weight adjustment for a single training example:

Initial weights: $w_{xg0} = 0.3$; $w_{xg1} = 0.6$; $w_{xg2} = -0.1$; $w_{gf0} = -0.2$; $w_{gf1} = 0.5$

Training example: $x_1 = -0.9$; $x_2 = 0.1$; $y_{truth} = 1.0$

Learning rate $= lr = 0.1$

FORWARD PASS

We compute the output of neuron G by applying the *tanh* activation function to the weighted sum of the inputs, namely the bias term and the two *x*-values:

$$y_g = \tanh\left(w_{xg0} + w_{xg1}x_1 + w_{xg2}x_2\right)$$
$$= \tanh\left(0.3 + 0.6\cdot(-0.9) + (-0.1)\cdot 0.1\right) \cong -0.25$$

We then compute the output of neuron F by applying the logistic activation function to the weighted sum of the inputs to this neuron, which is simply the bias term and the output from neuron G:

$$y_f = S\left(w_{gf0} + w_{gf1}y_g\right) = S\left(-0.2 + 0.5\cdot(-0.25)\right) = 0.42$$

We conclude the forward pass with computing the MSE between the desired output and the actual output to see how well the current weights work, but we will not use this computation for the backward pass.

$$MSE = \frac{(y - y_f)^2}{2} = \frac{(1.0 - 0.42)^2}{2} = 0.17$$

BACKWARD PASS

We start the backward pass with computing the derivative of the error function:

$$MSE' = -(y - y_f) = -(1.0 - 0.42) = -0.58$$

We then compute the error term for neuron F. The general way of doing this is to multiply the just-computed error term (for the layer that follows the current neuron) by the weight that connects this error to the current neuron and then multiply by the derivative of the activation function for the current neuron. This last layer is a little bit special in that there is no weight that connects the output to the error function (i.e., the weight is 1). Thus, the error term for neuron F is computed as

$$\text{Error term } f = MSE' \cdot y_f' = -0.58\cdot 0.42\cdot(1 - 0.42) = -0.14$$

In this formula, we computed the derivative of the logistic sigmoid function as $S\cdot(1-S)$.

We then move on to do the same computation for neuron G, where we now multiply the just-computed error term for neuron F by the weight that connects

neuron F to neuron G and then multiply by the derivative of the activation function for neuron G:

$$Error\ term\ g = Error\ term\ f \cdot w_{gf1} \cdot y'_g = -0.14 \cdot 0.5 \cdot \left(1-(-0.24)^2\right) = -0.066$$

In this formula, we computed the derivative of the tanh function as $(1 - tanh^2)$.

WEIGHT ADJUSTMENT

We are now ready to adjust the weights. We compute the adjustment value for a weight by multiplying the learning rate by the input value to the weight and then multiply by the error term for the neuron that follows the weight. The input values to the bias weights are 1. Note that for the weight connecting G to F, the input value is the output of neuron G (–0.25):

$$\Delta w_{xg0} = -lr \cdot 1 \cdot Error\ term\ g = -0.1 \cdot 1 \cdot (-0.066) = 0.0066$$

$$\Delta w_{xg1} = -lr \cdot x_1 \cdot Error\ term\ g = -0.1 \cdot (-0.9) \cdot (-0.066) = -0.0060$$

$$\Delta w_{xg2} = -lr \cdot x_2 \cdot Error\ term\ g = -0.1 \cdot 0.1 \cdot (-0.066) = 0.00066$$

$$\Delta w_{gf0} = -lr \cdot 1 \cdot Error\ term\ f = -0.1 \cdot 1 \cdot (-0.14) = 0.014$$

$$\Delta w_{gf1} = -lr \cdot y_g \cdot Error\ term\ f = -0.1 \cdot (-0.25) \cdot (-0.14) = -0.0035$$

We included a negative sign in the deltas above, so the updated weights can now be computed by simply adding the deltas to the existing weights:

$$w_{xg0} = 0.3 + 0.0066 = 0.3066$$

$$w_{xg1} = 0.6 - 0.0060 = 0.5940$$

$$w_{xg2} = -0.1 + 0.00066 = -0.0993$$

$$w_{gf0} = -0.2 + 0.014 = -0.1859$$

$$w_{gf1} = 0.5 - 0.0035 = 0.4965$$

Figure 3-7 shows the network annotated with key values computed during the forward and backward passes. The green and red arrows indicate the direction (green = positive, red = negative) and magnitude (wider is greater) of the resulting weight adjustments.

We can gain some intuition by looking at the magnitude and direction of the weight adjustments. Considering neuron G, we see that the weights for the bias term and the x_1 input are adjusted by an order of magnitude more than the weight for x_2 (0.0066 and –0.0060 for the bias and x_1 weights vs. 0.00066 for

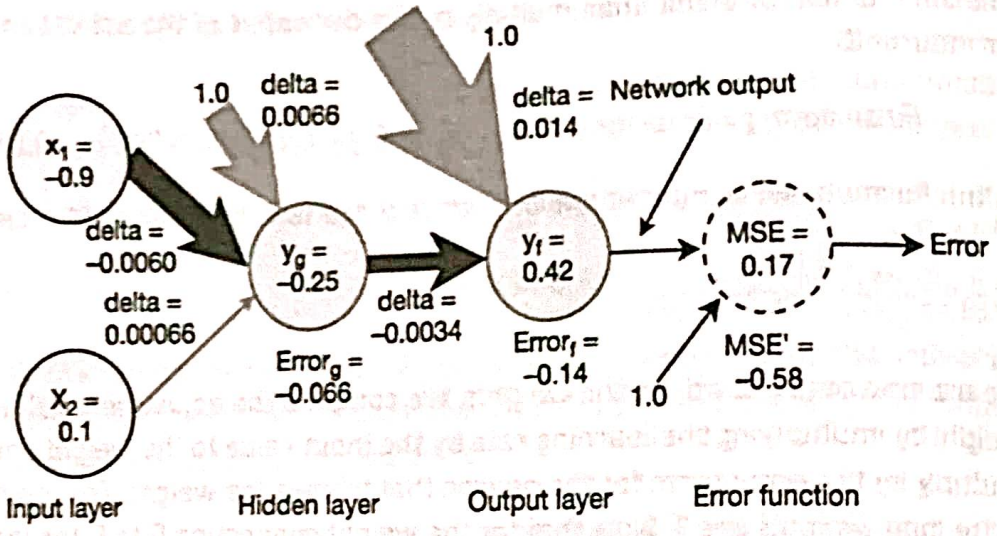


Figure 3-7 Network annotated with numbers computed during forward and backward passes. A green arrow represents a positive weight adjustment, and red represents a negative adjustment. The width of the arrow indicates the magnitude of the adjustment. Note that the actual weights are not shown in the figure; only the adjustment value (delta) is shown.

the x_2 weight). This makes sense because the magnitude of the bias input and x_1 is greater than the magnitude for x_2, and thus these two weights are more significant levers. Another observation is that the output is less than the desired output, so we want to increase the output of the network. This property, together with the sign of the input values that feed each weight, will determine the direction that the weight is adjusted. For example, the bias weight is increased, whereas the weight corresponding to input x_1 is decreased because the x_1 input is a negative value.

Just as for the perceptron learning algorithm, we provide a spreadsheet that replicates the computations. In addition, this spreadsheet contains multiple iterations of the algorithm. We recommend playing with that spreadsheet to get a better understanding of the computations and gain some intuition. The location of the spreadsheet can be found under "Programming Examples" in Appendix J.

The number of computations needed to compute the entire gradient is about the same as the number of computations that are needed for one forward pass. There is one derivative per neuron in the network and one multiplication per weight. This can be compared with the N+1 times the forward pass that would have been needed if we computed the gradient numerically using the brute-force method that we envisioned before describing the backpropagation algorithm. This makes it clear that the backpropagation algorithm is an efficient way of computing the gradient.