Chapter 4

Backpropagation

- Understanding Gradients
- Calculating Gradients
- Understanding Backpropagation
- Momentum and Learning Rate

So far we have only looked at how to calculate the output from a neural network. The output from the neural network is a result of applying the input to the neural network across the weights of several layers. In this chapter we will see how these weights are adjusted to produce outputs that are closer to the desired output.

This process is called training. Training is an iterative process. To make use of training, you perform multiple training iterations. It is hoped that the training iterations will lower the global error of the neural network. Global and local error were discussed in Chapter 2.

4.1 Understanding Gradients

The first step is to calculate the gradients of the neural network. The gradients are used to calculate the slope, or gradient, of the error function for a particular weight. This allows the training method to know to either increase or decrease the weight. There are number of different training methods that make use of gradients. These training methods are called propagation training. This book will discuss the following propagation training methods.

- Backpropagation
- Resilient Propagation
- Quick Propagation

This chapter will focus on using the gradients to train the neural network using back propagation. The next few chapters will focus on the other propagation methods.

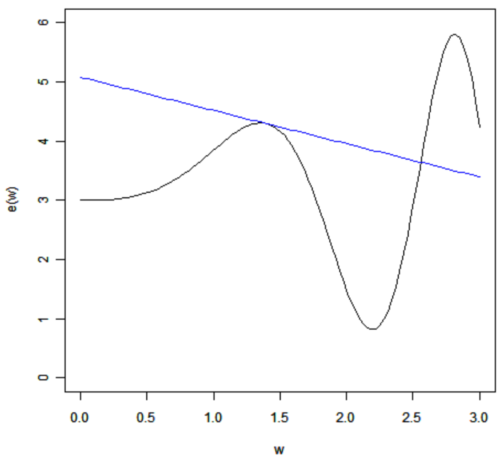
4.1.1 What is a Gradient

We will begin by looking at what a gradient is. Basically training is a search. You are searching for the set of weights that will cause the neural network to have the lowest global error for a training set. If we had an infinite amount of computation resources we would simply try every possible combination of weights and see which provided the absolute best global error.

Because we do not have unlimited computing resources we will have to take some sort of short cut. A shortcut is essentially all that neural network training methods really are. Each training method is a cleaver way of finding an optimal set of weights without doing an imposable exhaustive search.

Consider a chart that shows the global error of a neural network for each possible weight. This graph might look something like Figure 4.1.

Figure 4.1: Gradient



Looking at this chart you can easily see the optimal weight. The optimal weight is the location where the line has the lowest y value. The problem is, we do not get to see the entire graph. This would be the exhaustive search previously mentioned. We only see the error for the current value of the weight. However, we can determine the slope of the error curve at a particular weight. In the above chart we see the slope of the error curve at 1.5. The slope is given by the straight line that just barely touches the error curve at 1.5. This is this slope is the gradient. In this case the slope is -0.5622.

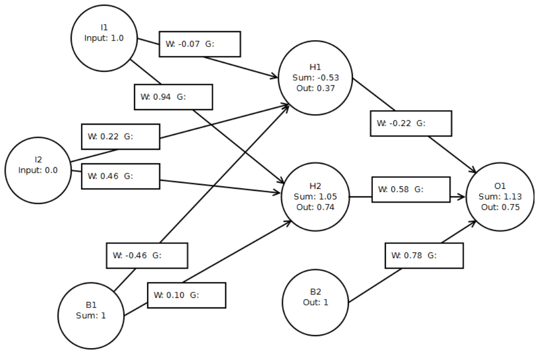
The gradient is the instantaneous slope of the error function at the specified weight. This is the same definition for the derivative that we learned in Chapter 3. The gradient is given by the derivative of the error curve at that point. This line tells us something about how steep the error function is at the given weight.

Used with a training technique, this can provide some insight into how the weight should be adjusted for a lower error. Now that we have seen what a gradient is, we will see how to actually calculate a gradient in the next section.

4.1.2 Calculating Gradients

We will now look at how to actually calculate the gradient. We will calculate an individual gradient for each weight. A weight is a connection between two neurons. I will show you the equations, as well as how to apply them to an actual neural network with real numbers. The neural network that we will use is shown in Figure 4.2.

Figure 4.2: An XOR Network



The neural network above is a typical three layer feedforward network like we have seen before. The circles indicate neurons. The lines connecting the circles are the weights. The rectangles in the middle of the connections give the weight for each connection.

The problem that we must now face is how to calculate the partial derivative for the output of each neuron. This can be accomplished using a method based on the chain rule of Calculus. The chain rule was discussed in Chapter 3. We will begin with one training set element. For Figure 4.2 we are providing an input of [**1**,**0**] and expecting an output of [**1**]. You can see the input being applied on the above figure as the first input neuron has an input value of 1.0 and the second input neuron has an input value of 0.0.

This input feeds through the network and eventually produces an output. The exact process by which the output and sums are calculated was covered in Chapter 1. Backpropagation has both a forward and backward pass. When forward pass occurred when the output of the neural network was calculated. We will calculate the gradients for only this item in the training set. Other items in the training set will have different gradients. How the gradients for each individual training set element are combined will be discussed later in this chapter when we discuss ”Batch and Online Training”.

We are now ready to calculate the gradients. There are several steps involved in calculating the gradients for each weight. These steps are summarized here.

- Calculate the error, based on the ideal of the training set
- Calculate the node delta for the output neurons
- Calculate the node delta for the interior neurons
- Calculate individual gradients

These steps will be covered in the following sections.

4.1.3 Calculating the Node Deltas

The first step is to calculate a constant value for every nod, or neuron, in the neural network. We will start with the output nodes and work our way backwards through the neural network. This is where the term backpropagation comes from. We initially calculate the errors for the output neurons and propagate these errors backwards through the neural network.

The value that we will calculate for each node is called the node delta. The term layer delta is also sometimes used to describe this value as well. Layer delta describes the fact that these deltas are calculated one layer at a time. The method for calculating the node deltas differs depending on if you are calculating for an output or interior node. The output neurons are obviously, all output nodes. The hidden and input neurons are the interior nodes. The equation to calculate the node delta is provided in Equation 4.1.

$$\delta_i = \begin{cases} -Ef'_i & \text{, output nodes} \\ f'_i \sum_k w_{ki}\delta_k & \text{, interier nodes} \end{cases} \tag{4.1}$$

We will calculate the node delta for all hidden and non-bias neurons. There is no need to calculate the node delta for the input and bias neurons. The node delta can easily be calculated for input and bias neurons using the above equation. However, these values are not needed for the gradient calculation. As you will soon see, gradient calculation for a weight only looks at the neuron that the weight is connected to. Bias and input neurons are only the beginning point for a connection. They are never the end point.

We will begin by using the formula for the output neurons. You will notice that the formula uses a value **E**. This is the error for this output neuron. You can see how to calculate **E** from Equation 4.2.

$$E = (a - i) \tag{4.2}$$

You may recall a similar equation as Equation 4.2 from Chapter 3. This is the error function. Here we subtract the ideal from the actual. For the example neural network provided earlier this becomes the following.

```
E = 0.75 − 1.00 = −0.25
```

Now that we have **E**, we can calculate the node delta for the first (and only) output node. Filling in Equation 4.1 we get the following.

```
−( −0.25) ∗ dA(1.1254) = 0.185 ∗ 0.25 = 0.05
```

The value of 0.05 will be used for the node delta of output neuron. In the above equation, **dA** represents the derivative of the activation function. For this example, we are using a sigmoid activation function. The sigmoid activation function is shown in Equation 4.3.

$$s(x) = \frac{1}{1 + e^{-x}} \tag{4.3}$$

The derivative of the sigmoid function is shown in Equation 4.4.

$$f'(x) = s(x) * (1.0 - s(x)) \tag{4.4}$$

Now that the node delta has been calculated for the output neuron, we should calculate it for the interior neurons as well. The equation to calculate the node delta for interior neurons was provided in Equation 4.1. Appling this for the first hidden neuron we have the following.

```
dA(sum of H1) ∗ (O1 Node Delta ∗ Weight of H1 −> O1)
```

You will notice that Equation 4.1 called for the summing up a number of items based on the number of inbound connections to output neuron one. Because there is only one inbound connection to output neuron one there is only one value to sum. This value is the product of the output neuron one node delta and the weight between hidden neuron one and output neuron one.

Filling in actual values for the above expression, we are left with the following.

```
dA(−0.53) ∗ (0.05 ∗ −0.22) = −0.0025
```

The value -0.0025 is the node delta for the first hidden neuron. Calculating the second hidden neuron follows exactly the same form as above. The second neuron would be computed as follows.

```
dA(sum of H2) ∗ (O1 Node Delta ∗ Weight of H2 −> O1)
```

Plugging in actual numbers, we have the following.

```
dA(1.05) ∗ (0.05 ∗ 0.58) = 0.0055
```

The value of 0.0055 is the node delta for the second hidden neuron.

As previously explained, there is no reason to calculate the node delta for either the bias neurons or the input neurons. We now have every layer delta needed to calculate a gradient for each weight in the neural network. Calculation of the individual gradients will be discussed in the next section.

4.1.4 Calculating the Individual Gradients

We can now calculate the individual gradients. Unlike the layer deltas there is only one equation used to calculate the actual gradient. A gradient is calculated using Equation 4.5.

$$\frac{\partial E}{\partial w_{(ik)}} = \delta_k \cdot o_i \tag{4.5}$$

The above equation calculates the partial derivative of the error (**E**) with respect to each individual weight. The partial derivatives are the gradients. Partial derivatives were discussed in Chapter 3. To determine an individual gradient multiply the node delta for the target neuron by the weight from the source neuron. In the above equation **k** represents the target neuron and **i** represents the source neuron.

To calculate the gradient for the weight from **H1** to **O1** the following values would be used.

```
output(h1) ∗ nodeDelta(o1)
(0.37 ∗ 0.05) = 0.01677
```

It is important to note that in the above equation we are multiplying by the output of hidden 1, not the sum. When dealing directly with a derivative you should supply the sum. Otherwise, you would be indirectly applying the activation function twice. In the above equation we are not dealing directly with the derivative, so we use the regular node output. The node output has already had the activation function applied.

Once the gradients are calculated the individual positions of the weights no longer matter. We can simply think of the weights and gradients as single dimensional arrays. The individual training methods that we will look at will treat all weights and gradients equal. It does not matter if a weight is from an input neuron or an output neuron. It is only important the correct weight be used with the correct gradient. The ordering of this weight and gradient array is arbitrary. However, Encog uses the following order for the above neural network.

```
Weight/Gradient 0: Hidden 1 -> Output 1
Weight/Gradient 1: Hidden 2 -> Output 1
Weight/Gradient 2: Bias 2 -> Output 1
Weight/Gradient 3: Input 1 -> Hidden 1
Weight/Gradient 4: Input 2 -> Hidden 1
Weight/Gradient 5: Bias 1 -> Hidden 1
Weight/Gradient 6: Input 1 -> Hidden 2
Weight/Gradient 7: Input 2 -> Hidden 2
Weight/Gradient 8: Bias 1 -> Hidden 2
```

The various learning algorithms will make use of the weight and gradient arrays. It is also important to note that these are two separate arrays. There is a weight array, as well as a gradient array. Both arrays will be exactly the same length. Training a neural network is nothing more than adjusting the weights to provide desirable output. In this chapter we will see how backpropagation uses the gradient array to modify the weight array.

4.2 Applying Back Propagation

Backpropagation is a simple training method that uses the calculated gradients of a neural network to adjust the weights of the neural network. As these weights are adjusted the neural network should produce more desirable output. The global error of the neural network should fall as it is trained. Before we can examine the backpropagation weight update process me must look at two different ways that the gradients can be calculated.

4.2.1 Batch and Online Training

We have already seen how to calculate the gradients for an individual training set element. Earlier in this chapter we saw how we could calculate the gradients for a case where the neural network was given an input of [1,0] and an output of [1] was expected. This works fine for a single training set element. However, most training sets have many elements. There are two different ways to handle multiple training set elements. These two approaches are called online and batch training.

Online training implies that you modify the weights after every training set element. Using the gradients that you obtained for the first training set element you calculate and apply a change to the weights. Training progresses to the next training set element and also calculates an update to the neural network. This training progresses until every training set element has been used. At this point one iteration, or epoch, of training has completed.

Batch training also makes use of every training set element. However, the weights are not updated for every training set element. Rather the gradients for each training set element are summed. Once every training set element has been used, the neural network weights can be updated. At this point the iteration is considered complete.

Sometimes a batch size will be set. For example, you might have a training set size of 10,000 elements. You might choose to update the weights of the neural network every 1,000 elements. This would cause the neural network weights to be updated 10 times during the training iteration.

Online training was the original method used for backpropagation. However, online training is inefficient, as the neural network must be constantly updated. Additionally, online training is very difficult to implement in a multi-threaded manor that will take advantage of multi-core processors. Because of these reasons batch training is generally preferable to online training.

4.2.2 Backpropagation Weight Update

We are now ready to update the weights. As previously mentioned, we will treat the weights and gradients as a single dimensional array. Give these two arrays we are ready to calculate the weight update for an iteration of backpropagation training. The formula to update the weights for backpropagation is shown in Equation 4.6.

$$\Delta w_{(t)} = \epsilon \frac{\partial E}{\partial w_{(t)}} + \alpha \Delta w_{(t-1)} \tag{4.6}$$

The above equation calculates the change in weight for each element in the weight array. You will also notice that the above equation calls for the weight change from the previous iteration. These values must be kept in another array.

The above equation calculates the weight delta to be the product of the gradient and the learning rate (represented by epsilon). Additionally the product of the previous weight change and the moment value (represented by alpha) is added. The learning rate and momentum are two parameters that must be provided to the backpropagation algorithm. Choosing values for learning rate and momentum is very important to the performance of the training. Unfortunately, the process for determining learning rate and momentum is mostly trial and error.

The learning rate scales the gradient and can slow down or speed up learning. A learning rate below zero will slow down learning. For example, a learning rate of 0.5 would decrease every gradient by 50%. A learning rate above 1.0 would speed up training. In reality the learning rate is almost always below zero.

Choosing two high of a learning rate will cause your neural network to fail to converge. A neural network that is failing to converge will generally have a high global error that simply bounces around, rather than converging to a low value. Choosing too low of a learning rate will cause the neural network to take a great deal of time to converge.

Like the learning rate, the momentum is also a scaling factor. Momentum determines what percent of the previous iteration's weight change should be applied to this iteration. Momentum is optional. If you do not want to use momentum, just specify a value of zero.

Momentum is a technique that was added to backpropagation to help the training find its way out of local minima. Local minima are low points on the error graph that are not the true global minimum. Backpropagation has a tendency to find its way into a local minimum and not find its way back out again. This will cause the training to converge to a higher undesirable error. Momentum gives the neural network some force in its current direction and may allow it to force through a local minimum.

We are now ready to plug values into Equation 4.6 and calculate a weight delta. We will calculate a weight change for the first iteration using the neural network we previously used in this chapter. So far we have only calculated gradients for one training set element. There are still four other training set elements to calculate for. We will sum all four gradients together to apply batch training. The batch gradient is calculated by summing the individual gradients, which are listed here.

```
0.01677795762852397
−0.05554301180824532
0.02194053316555356
−0.05861906411780882
```

Earlier in the chapter we calculated the first gradient, listed above. If you would like to see the calculation for the others, this information can be found at the following URL.

http://www.heatonresearch.com/wiki/Back_Propagation

Summing all of these gradients produces the following batch update gradient.

```
−0.07544358513197481
```

For this neural network we will use a learning rate of 0.7 and a momentum of 0.3. These are just arbitrary values. However, they do work well for training an XOR neural network. Plugging these values into Equation 4.6 results in the following.

```
delta = (0.7 * −0.0754) + (0.3 * 0.0) = −0.052810509592382364
```

This is the first training iteration, so the previous delta value is 0.0. The momentum has no effect on the first iteration. This delta value will be added to the weight to alter the neural network for the first training iteration. All of the other weights in this neural network will be updated in the same way, according to their calculated gradient.

This first training iteration will lower the neural network's global error slightly. Additional training iterations will further lower the error. The following program output shows the convergence of this neural network.

```
Epoch #1 Error:0.3100155809627523
Epoch #2 Error:0.2909988918032235
Epoch #3 Error:0.2712902750837602
Epoch #4 Error:0.2583119003843881
Epoch #5 Error:0.2523050561276289
Epoch #6 Error:0.2502986971902545
Epoch #7 Error:0.2498182295192154
Epoch #8 Error:0.24974245650541688
Epoch #9 Error:0.24973458893806627
Epoch #10 Error:0.24972923906975902
...
Epoch #578 Error:0.010002702374503777
Epoch #579 Error:0.009947830890527089
Neural Network Results:
1.0,0.0, actual=0.9040102333814147,ideal=1.0
0.0,0.0, actual=0.09892634022671229,ideal=0.0
0.0,1.0, actual=0.904020682439766,ideal=1.0
1.0,1.0, actual=0.10659032105865764,ideal=0.0
```

Each iteration, or epoch, decreases the error. Once the error drops below one percent, the training stops. You can also see the output from the neural network for the XOR data. The answers are not exactly correct, however, it is very clear that that the two training cases that should be 1.0 are much closer to 1.0 than the others.

4.3 Chapter Summary

In this chapter you were introduced to backpropagation. Backpropagation is one of the oldest, and most commonly used, training algorithms available for neural networks. Backpropagation works by calculating a gradient value for each weight in the network. Many other training methods also make use of these gradient values.

There is one gradient for each weight in the neural network. Calculation of the gradients is a step by step process. The first step in calculating the gradients is to calculate the error for each of the outputs of the neural network. This error is for one training set element. The gradients for all training set elements may be batched together later in the process.

Once the error for the output layer has been calculated values can be calculated for each of the output neurons. These values are called the node deltas for each of the output neurons. We must calculate the node deltas for the output layer first. We calculate the node deltas for each layer of the neural network working out way backwards to the input layer. This is why this technique is called backpropagation.

Once the node deltas have been calculated it is very easy to calculate the gradients. At the end you will have all of the gradients for one training set element. If you are using online training you will now use these gradients to apply a change to the weights of the neural network. If you are using batch training, you will sum the gradients from each of the training set elements into a single set of gradients for the entire training set.

Backpropagation must be provided a learning rate and momentum. Both of these are configuration items that will have an important effect on the training speed of your neural network. Learning rate specifies how fast the weights should be updated. Too high of a learning rate will cause a network to become unstable. Too low of a learning rate will cause the neural network to take too long to train. Momentum allows the neural network to escape local minima. Local minima are low points in the error graph that are not the true global minimum.

Choosing values for the learning rate and momentum can be tricky. Often it is just a matter of trial and error. The Resilient Propgation training method (RPROP) requires no paramaters to be set. Further, RPROP often trains much faster than backpropagation. RPROP will be covered in the next chapter.