Chapter 6

Backpropagation Training

- Gradient Calculation
- Backpropagation
- Learning Rate & Momentum
- Stochastic Gradient Descent

Backpropagation is one of the most common methods for training a neural network. Rumelhart, Hinton, & Williams (1986) introduced backpropagation, and it remains popular today. Programmers frequently train deep neural networks with backpropagation because it scales really well when run on graphical processing units (GPUs). To understand this algorithm for neural networks, we must examine how to train it as well as how it processes a pattern.

Classic backpropagation has been extended and modified to give rise to many different training algorithms. In this chapter, we will discuss the most commonly used training algorithms for neural networks. We begin with classic backpropagation and then end the chapter with stochastic gradient descent (SGD).

6.1 Understanding Gradients

Backpropagation is a type of gradient descent, and many texts will use these two terms interchangeably. Gradient descent refers to the calculation of a gradient on each weight in the neural network for each training element. Because the neural network will not output the expected value for a training element, the gradient of each weight will give you an indication about how to modify each weight to achieve the expected output. If the neural network did output exactly what was expected, the gradient for each weight would be 0, indicating that no change to the weight is necessary.

The gradient is the derivative of the error function at the weight's current value. The error function measures the distance of the neural network's output from the expected output. In fact, we can use gradient descent, a process in which each weight's gradient value can reach even lower values of the error function.

With respect to the error function, the gradient is essentially the partial derivative of each weight in the neural network. Each weight has a gradient that is the slope of the error function. A weight is a connection between two neurons. Calculating the gradient of the error function allows the training method to determine whether it should increase or decrease the weight. In turn, this determination will decrease the error of the neural network. The error is the difference between the expected output and actual output of the neural network. Many different training methods called propagation-training algorithms utilize gradients. In all of them, the sign of the gradient tells the neural network the following information:

- **Zero gradient** - The weight is not contributing to the error of the neural network.
- **Negative gradient** - The weight should be increased to achieve a lower error.
- **Positive gradient** - The weight should be decreased to achieve a lower error.

Because many algorithms depend on gradient calculation, we will begin with an analysis of this process.

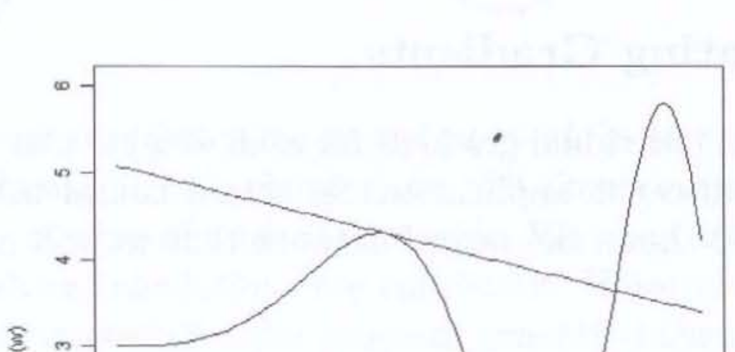
6.1.1 What is a Gradient

First of all, let's examine the gradient. Essentially, training is a search for the set of weights that will cause the neural network to have the lowest error for a training set. If we had an infinite amount of computation resources, we would simply try every possible combination of weights to determine the one that provided the lowest error during the training.

Because we do not have unlimited computing resources, we have to use some sort of shortcut to prevent the need to examine every possible weight combination. These training methods utilize clever techniques to avoid performing a brute-force search of all weight values. This type of exhaustive search would be impossible because even small networks have an infinite number of weight combinations.

Consider a chart that shows the error of a neural network for each possible weight. Figure 6.1 is a graph that demonstrates the error for a single weight:

Figure 6.1: Gradient of a Single Weight



Looking at this chart, you can easily see that the optimal weight is the location where the line has the lowest y-value. The problem is that we see only the error for the current value of the weight; we do not see the entire graph because that process would require an exhaustive search. 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 straight line that barely touches the error curve at 1.5 gives the slope. In this case, the slope, or gradient, is -0.5622. The negative slope indicates that an increase in the weight will lower the error.

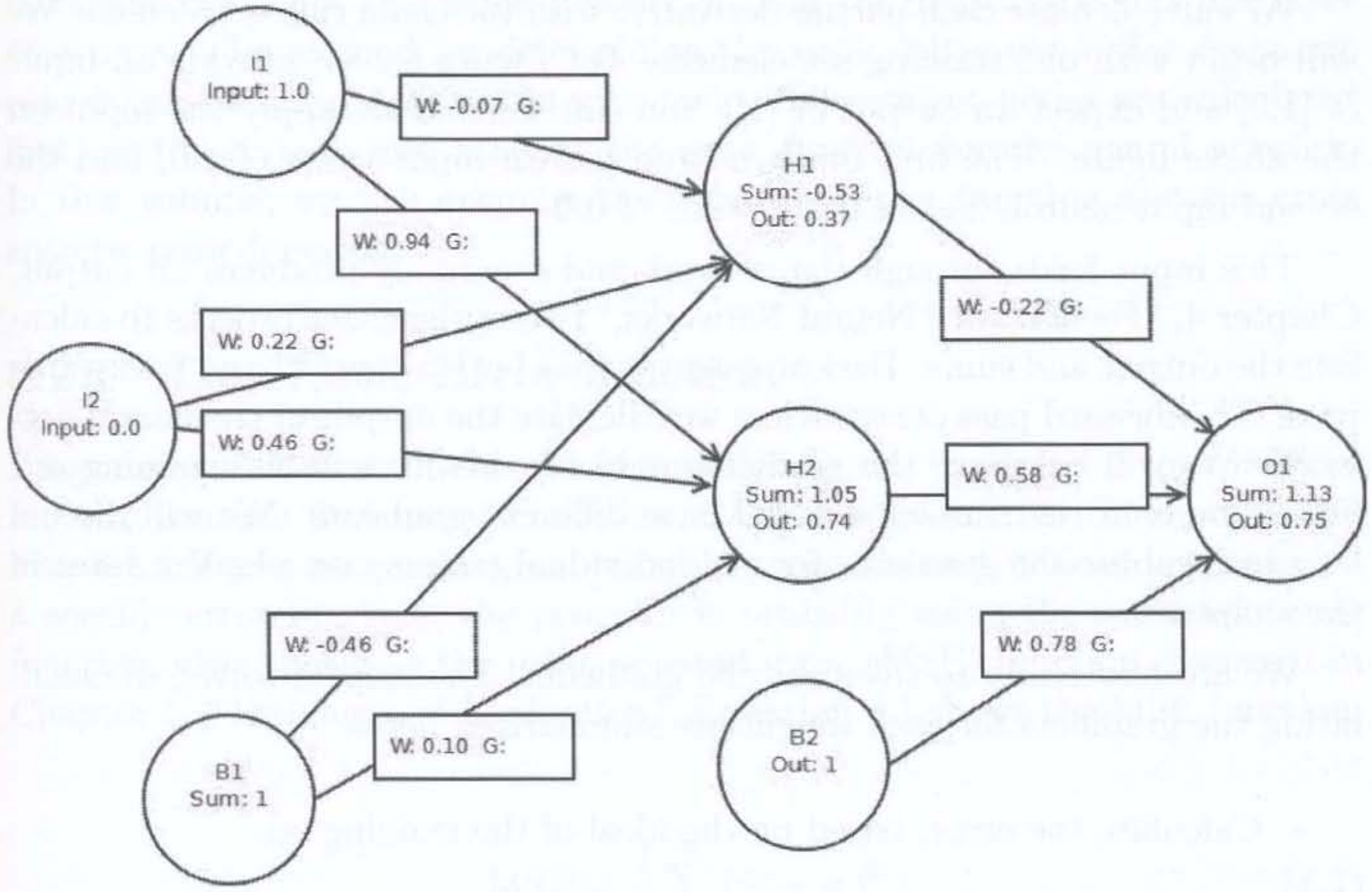
The gradient is the instantaneous slope of the error function at the specified weight. The derivative of the error curve at that point gives the gradient. This line tells us the steepness of the error function at the given weight.

Derivatives are one of the most fundamental concepts in calculus. For the purposes of this book, you just need to understand that a derivative provides the slope of a function at a specific point. A training technique and this slope can give you the information to adjust the weight for a lower error. Using our working definition of the gradient, we will now show how to calculate it.

6.1.2 Calculating Gradients

We will calculate an individual gradient for each weight. Our focus is not only the equations but also the applications in actual neural networks with real numbers. Figure 6.2 shows the neural network that we will use:

Figure 6.2: An XOR Network



Additionally, we use this same neural network in several examples on the website for this book. In this chapter, we will show several calculations that demonstrate the training of a neural network. We must use the same starting weights so that these calculations are consistent. However, the above weights have no special characteristic; the program generated them randomly.

The aforementioned neural network is a typical three-layer feedforward network like the ones we have previously studied. 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 now face is calculating the partial derivative for each of the weights in the neural network. We use a partial derivative when an equation has more than one variable. Each of the weights is considered a variable because these weight values will change independently as the neural network changes. The partial derivatives of each weight simply show each weight's independent effect on the error function. This partial derivative is the gradient.

We can calculate each partial derivative with the chain rule of calculus. We will begin with one training set element. For Figure 6.2 we provide an input of [1,0] and expect an output of [1]. You can see that we apply the input on the above figure. 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. Chapter 4, "Feedforward Neural Networks," covers the exact process to calculate the output and sums. Backpropagation has both a forward and backwards pass. The forward pass occurs when we calculate the output of the neural network. We will calculate the gradients only for this item in the training set. Other items in the training set will have different gradients. We will discuss how to combine the gradients for the individual training set element later in the chapter.

We are now ready to calculate the gradients. The steps involved in calculating the gradients for each weight are summarized here:

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

We will discuss these steps in the subsequent sections.

6.2 Calculating Output Node Deltas

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

The node delta is the value that we will calculate for each node. Layer delta also describes this value because we can calculate the deltas one layer at a time. The method for determining the node deltas can differ if you are calculating for an output or interior node. The output nodes are calculated first, and they take into account the error function for the neural network. In this volume, we will examine the quadratic error function and the cross entropy error function.

6.2.1 Quadratic Error function

Programmers of neural networks frequently use the quadratic error function. In fact, you can find many examples of the quadratic error function on the Internet. If you are reading an example program, and it does not mention a specific error function, the program is probably using the quadratic error function, also known as the mean squared error (MSE) function discussed in Chapter 5, "Training and Evaluation." Equation 6.1 shows the MSE function:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{6.1}$$

The above equation compares the neural network's actual output (y) with the expected output (y-hat). The variable n contains the number of training elements times the number of output neurons. MSE handles multiple output neurons as individual cases. Equation 6.2 shows the node delta used in conjunction with the quadratic error function:

$$\delta_i = (\hat{y}_i - y_i)\phi'_i \tag{6.2}$$

The quadratic error function is very simple because it takes the difference between the expected and actual output for the neural network. The Greek letter ϕ (phi-prime) represents the derivative of the activation function.

6.2.2 Cross Entropy Error Function

The quadratic error function can sometimes take a long time to properly adjust the weight. Equation 6.3 shows the cross entropy error function:

$$CE = -\frac{1}{n}\sum_{x}[y \ln a + (1-y)\ln(1-a)] \tag{6.3}$$

The node delta calculation for the cross entropy error turns out to be much less complex than the MSE, as seen in Equation 6.4.

$$\delta_i = \hat{y}_i - y_i \tag{6.4}$$

The cross entropy error function will typically better results than the quadratic it will create a much steeper gradient for errors. You should always use the cross entropy error function.

6.3 Calculating Remaining Node Deltas

Now that the output node delta has been calculated according to the appropriate error function, we can calculate the node deltas for the interior nodes, as demonstrated by Equation 6.5:

$$\delta_i = \phi_i' \sum_{k} w_{ki}\delta_k \tag{6.5}$$

We will calculate the node delta for all hidden and non-bias neurons, but we do not need to calculate the node delta for the input and bias neurons. Even though we can easily calculate the node delta for input and bias neurons with Equation 6.5, gradient calculation does not require these values. As you will soon see, gradient calculation for a weight only considers the neuron to which the weight is connected. Bias and input neurons are only the beginning point for a connection; they are never the end point.

If you would like to see the gradient calculation process, several JavaScript examples will show the individual calculations. These examples can be found at the following URL:

http://www.heatonresearch.com/aifh/vol3/

6.4 Derivatives of the Activation Functions

The backpropagation process requires the derivatives of the activation functions, and they often determine how the backpropagation process will perform. Most modern deep neural networks use the linear, softmax, and ReLU activation functions. We will also examine the derivatives of the sigmoid and hyperbolic tangent activation functions so that we can see why the ReLU activation function performs so well.

6.4.1 Derivative of the Linear Activation Function

The linear activation function is barely an activation function at all because it simply returns whatever value it is given. For this reason, the linear activation function is sometimes called the identity activation function. The derivative of this function is 1, as demonstrated by Equation 6.6:

$$\phi\prime(x) = 1 \tag{6.6}$$

The Greek letter ϕ (phi) represents the activation function, as in previous chapters. However, the apostrophe just above and to the right of ϕ (phi) means that we are using the derivative of the activation function. This is one of several ways that a derivative is expressed in a mathematical form.

6.4.2 Derivative of the Softmax Activation Function

In this volume, the softmax activation function, along with the linear activation function, is used only on the output layer of the neural networks. As mentioned in Chapter 1, "Neural Network Basics," the softmax activation function is different from the other activation functions in that its value is dependent on the other output neurons, not just on the output neuron currently being calculated. For convenience, the softmax activation function is repeated in Equation 6.7:

$$\phi_i = \frac{e^{z_i}}{\sum\limits_{j \in group} e^{z_j}} \tag{6.7}$$

The z vector represents the output from all output neurons. Equation 6.8 shows the derivative of this activation function:

$$\frac{\partial \phi_i}{\partial z_i} = \phi_i(1 - \phi_i) \tag{6.8}$$

We used slightly different notation for the above derivative. The ratio, with the cursive-stylized "d" symbol means a partial derivative, which occurs when you differentiate an equation with multiple variables. To take a partial derivative, you differentiate the equation relative to one variable, holding all others constant. The top "d" tells you what function you are differentiating. In this case, it is the activation function ϕ (phi). The bottom "d" denotes the respective differentiation of the partial derivative. In this case, we are calculating the output of the neuron. All other variables are treated as constant. A derivative is the instantaneous rate of change–only one thing can change at once.

You will not use the derivative of the linear or softmax activation functions to calculate the gradients of the neural network if you use the cross entropy error function. You should use the linear and softmax activation functions only at the output layer of a neural network. Therefore, we do not need to worry about their derivatives for the interior nodes. For the output nodes with cross entropy, the derivative of both linear and softmax is always 1. As a result, you will never use the linear or softmax derivatives for interior nodes.

6.4.3 Derivative of the Sigmoid Activation Function

Equation 6.9 shows the derivative of the sigmoid activation function:

$$\phi\prime(x) = \phi(x)(1 - \phi(x)) \tag{6.9}$$

Machine learning frequently utilizes the sigmoid function represented in the above equation. We derived the formula through algebraic manipulation of the sigmoid derivative in order to use the sigmoid activation function in its own derivative. For computational efficiency, the Greek letter ϕ (phi) in the above activation function represents the sigmoid function. During the feed-forward pass, we calculated the value of the sigmoid function. Retaining the sigmoid function makes the sigmoid derivative a simple calculation. If you are interested in how to obtain Equation 6.9, you can refer to the following URL:

http://www.heatonresearch.com/aifh/vol3/deriv_sigmoid.html

6.4.4 Derivative of the Hyperbolic Tangent Activation Function

Equation 6.10 shows the derivative of the hyperbolic tangent activation function:

$$\phi\prime(x) = 1.0 - \phi^2(x) \tag{6.10}$$

We recommend that you always use the hyperbolic tangent activation function instead of the sigmoid activation function.

6.4.5 Derivative of the ReLU Activation Function

Equation 6.11 shows the derivative of the ReLU function:

$$\frac{dy}{dx}\phi(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{6.11}$$

Strictly speaking, the ReLU function does not have a derivative at 0. However, because of convention, the gradient of 0 is substituted when x is 0. Deep neural networks with sigmoid and hyperbolic tangent activation functions can be difficult to train using backpropagation. Several factors cause this difficulty. The vanishing gradient problem is one the most common causes. Figure 6.3 shows the hyperbolic tangent function, along with its gradient/derivative:

Figure 6.3: Tanh Activation Function & Derivative

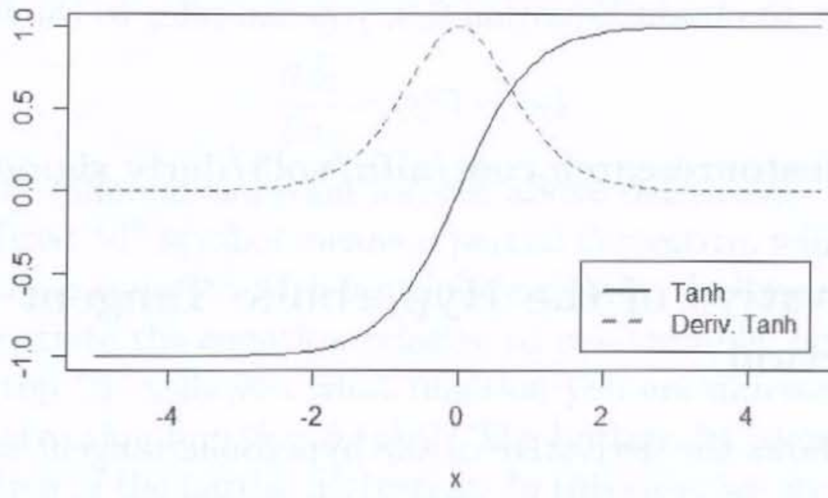
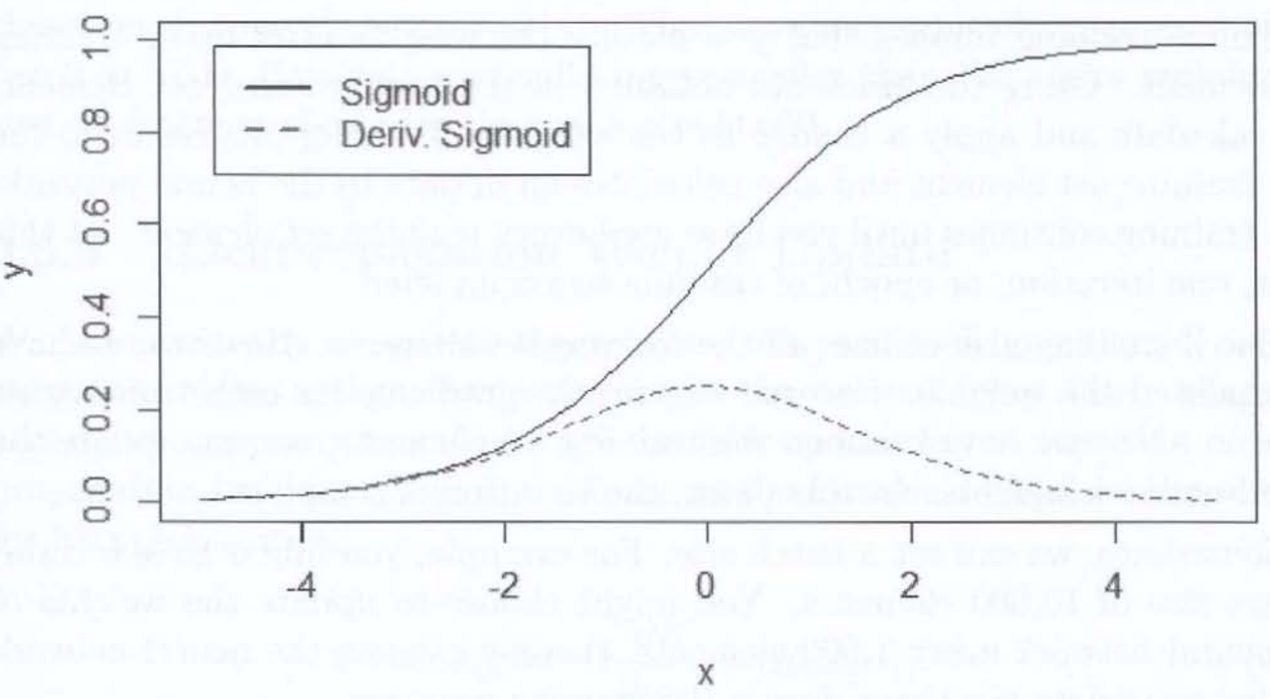


Figure 6.3 shows that as the hyperbolic tangent (blue line) saturates to -1 and 1, the derivative of the hyperbolic tangent (red line) vanishes to 0. The sigmoid and hyperbolic tangent activation functions both have this problem, but ReLU doesn't. Figure 6.4 shows the same graph for the sigmoid activation function and its vanishing derivative:

Figure 6.4: Tanh Activation Function & Derivative



6.5 Applying Backpropagation

Backpropagation is a simple training method that adjusts the weights of the neural network with its calculated gradients. This method is a form of gradient descent since we are descending the gradients to lower values. As the program adjusts these weights, the neural network should produce more desirable output. The global error of the neural network should fall as it trains. Before we can examine the backpropagation weight update process, we must examine two different ways to update the weights.

6.5.1 Batch and Online Training

We have already shown how to calculate the gradients for an individual training set element. Earlier in this chapter, we calculated the gradients for a case in which we gave the neural network an input of [1,0] and expected an output of

[1]. This result is acceptable for a single training set element. However, most training sets have many elements. Therefore, we can handle multiple training set elements through two approaches called online and batch training.

Online training implies that you modify the weights after every training set element. Using the gradients obtained in 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 continues until you have used every training set element. At this point, one iteration, or epoch, of training has completed.

Batch training also utilizes all the training set elements. However, we have not updated the weights. Instead, we sum the gradients for each training set element. Once we have summed the training set elements, we can update the neural network weights. At this point, the iteration is complete.

Sometimes, we can set a batch size. 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, thereby causing the neural network weights to update ten times during the training iteration.

Online training was the original method for backpropagation. If you would like to see the calculations for the batch version of this program, refer to the following online example:

http://www.heatonresearch.com/aifh/vol3/xor_batch.html

6.5.2 Stochastic Gradient Descent

Batch and online training are not the only choices for backpropagation. Stochastic gradient descent (SGD) is the most popular of the backpropagation algorithms. SGD can work in either batch or online mode. Online stochastic gradient descent simply selects a training set element at random and then calculates the gradient and performs a weight update. This process continues until the error reaches an acceptable level. Choosing random training set elements will usually converge to an acceptable weight faster than looping through the entire training set for each iteration.

Batch stochastic gradient descent works by choosing a batch size. For each iteration, a mini-batch is chosen by randomly selecting a number of training set elements up to the chosen batch size. The gradients from the mini-batch are summed just as regular backpropagation batch updating. This update is very similar to regular batch updating except that the mini-batches are randomly chosen each time they are needed. The iterations typically process a single batch in SGD. Batches are usually much smaller than the entire training set size. A common choice for the batch size is 600.

6.5.3 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. Given these two arrays, we are ready to calculate the weight update for an iteration of backpropagation training. Equation 6.6 shows the formula to update the weights for backpropagation:

$$\Delta w_{(t)} = -\epsilon \frac{\partial E}{\partial w_{(t)}} + \alpha \Delta w_{(t-1)} \tag{6.12}$$

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. You must keep these values in another array. As previously mentioned, the direction of the weight update is inversely related to the sign of the gradient–a positive gradient should cause a weight decrease, and vice versa. Because of this inverse relationship Equation 6.12 begins with a negative.

The above equation calculates the weight delta as the product of the gradient and the learning rate (represented by ϵ, epsilon). Furthermore, we add the product of the previous weight change and the momentum value (represented by α, alpha). The learning rate and momentum are two parameters that we must provide 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 0 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 accelerate training. In reality, the learning rate is almost always below 1.

Choosing a learning rate that is too high will cause your neural network to fail to converge and have a high global error that simply bounces around instead of converging to a low value. Choosing a learning rate that is too low 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. Although it is optional, momentum determines the percent of the previous iteration's weight change that should be applied to the iteration. If you do not want to use momentum, just specify a value of 0.

Momentum is a technique added to backpropagation that helps the training escape local minima, which 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 process causes 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 break through a local minimum.

6.5.4 Choosing Learning Rate and Momentum

Momentum and learning rate contribute to the success of the training, but they are not actually part of the neural network. Once training is complete, the trained weights remain and no longer utilize momentum or the learning rate. They are essentially part of the temporary scaffolding that creates a trained neural network. Choosing the correct momentum and learning rate can impact the effectiveness of your training.

The learning rate affects the speed at which your neural network trains. Decreasing the learning rate makes the training more meticulous. Higher learning rates might skip past optimal weight settings. A lower training rate will always produce better results. However, lowering the training rate can greatly increase runtime. Lowering the learning rate as the network trains can be an effective technique.

You can use the momentum to combat local minima. If you find the neural network stagnating, a higher momentum value might push the training past the local minimum that it encountered. Ultimately, choosing good values for momentum and learning rate is a process of trial and error. You can vary both as training progresses. Momentum is often set to 0.9 and the learning rate to 0.1 or lower.

6.5.5 Nesterov Momentum

The stochastic gradient descent (SGD) algorithm can sometimes produce erratic results because of the randomness introduced by the mini-batches. The weights might get a very beneficial update in one iteration, but a poor choice of training elements can undo it in the next mini-batch. Therefore, momentum is a resourceful tool that can mitigate this sort of erratic training result.

Nesterov momentum is a relatively new application of a technique invented by Yu Nesterov in 1983 and updated in his book, *Introductory Lectures on Convex Optimization: A Basic Course* (Nesterov, 2003). This technique is occasionally referred to as Nesterov's accelerated gradient descent. Although a full mathematical explanation of Nesterov momentum is beyond the scope of this book, we will present it for the weights in sufficient detail so you can implement it. This book's examples, including those for the online JavaScript, contain an implementation of Nesterov momentum. Additionally, the book's website contains Javascript that output example calculations for the weight updates of Nesterov momentum.

Equation 6.13 calculates a partial weight update based on both the learning rate (ϵ, epsilon) and momentum (α, alpha):

$$n_0 = 0 \ , \ n_t = \alpha n_{t-1} + \epsilon \frac{\partial E}{\partial w_t} \tag{6.13}$$

The current iteration is signified by *t*, and the previous iteration by *t*-1. This partial weight update is called *n* and initially starts out at 0. Subsequent calculations of the partial weight update are based on the previous value of the partial weight update. The partial derivative in the above equation is the gradient of the error function at the current weight. Equation 6.14 shows the Nesterov momentum update that replaces the standard backpropagation weight update shown earlier in Equation 6.12:

$$\Delta w_t = \alpha n_{t-1} - (1 + \alpha) n_t \tag{6.14}$$

The above weight change is calculated as an amplification of the partial weight change. The delta weight shown in the above equation is added to the current weight. Stochastic gradient descent (SGD) with Nesterov momentum is one of the most effective training algorithms for deep learning.

6.6 Chapter Summary

This chapter introduced classic backpropagation as well as stochastic gradient descent (SGD). These methods are all based on gradient descent. In other words, they optimized individual weights with derivatives. For a given weight value, the derivative gave the program the slope of the error function. The slope allowed the program to determine how to change the weight value. Each training algorithm interprets this slope, or gradient, differently.

Despite the fact that backpropagation is one of the oldest training algorithms, it remains one of the most popular ones. Backpropagation simply adds the gradient to the weight. A negative gradient will increase the weight, and a positive gradient will decrease the weight. We scale the weight by the learning rate in order to prevent the weights from changing too rapidly. A learning rate of 0.5 would mean to add half of the gradient to the weight, whereas a learning rate of 2.0 would mean to add twice the gradient.

There are a number of variants to the backpropagation algorithm. Some of these, such as resilient propagation, are somewhat popular. The next chapter will introduce some backpropagation variants. Though these variants are useful to know, stochastic gradient descent (SGD) remains the most common deep learning training algorithm.

$$[illegible] \quad (6.1[illegible])$$

The delta weight, [illegible] is calculated as [illegible] of the partial weight change. The delta weight shown in the above equation is added to the current weight. Stochastic gradient descent (SGD) with [illegible] is one of the most [illegible] training algorithms for deep learning.

6.6 Chapter Summary

This chapter has covered classic backpropagation as well as stochastic gradient descent (SGD). These methods are all based on gradient descent. In other words, they optimize individual weights with derivatives. For a given weight value, the derivative gives the program the slope of the error function. The slope allows the program to determine how to change the weight value. Each training algorithm interprets this slope, or gradient, differently.

Despite the fact that backpropagation is one of the oldest training algorithms, it remains one of the most popular ones. Backpropagation simply adds the gradient to the weight. A negative gradient will increase the weight, and a positive gradient will decrease the weight. We scale the weight by the learning rate in order to prevent the weights from changing too rapidly. A learning rate of 0.5 would mean to add half of the gradient to the weight, whereas a learning rate of 2.0 would mean to add twice the gradient.

There are a number of variants to the backpropagation algorithm. Some of these, such as resilient propagation, are somewhat popular. The next chapter will introduce some backpropagation variants. Though these variants are useful to know, stochastic gradient descent (SGD) remains the most common deep learning training algorithm.