Chapter 2

Gradient-Based Learning

In this chapter, we describe how the perceptron learning algorithm works, which we then build upon in Chapter 3, "Sigmoid Neurons and Backpropagation," by extending it to multilevel networks. These two chapters contain more mathematical content than other chapters in this book, but we also describe the concepts in an intuitive manner for readers who do not like reading mathematical formulas.

Intuitive Explanation of the Perceptron Learning Algorithm

In Chapter 1, "The Rosenblatt Perceptron," we presented and used the perceptron learning algorithm, but we did not explain why it works. Let us now look at what the learning algorithm does. To refresh our memory, the weight adjustment step in the perceptron learning algorithm is first restated in Code Snippet 2-1, where

Code Snippet 2-1 Weight Update Step of Perceptron Learning Algorithm

```
for i in range(len(w)):
   w[i] += (y * LEARNING_RATE * x[i])
```

w is an array representing the weight vector, x is an array representing the input vector, and y is the desired output.

If an example is presented to the perceptron and the perceptron correctly predicts the output, we do not adjust any weights at all (the code that ensures this is not shown in the snippet). This makes sense because if the current weights already result in the correct output, there is no good reason to adjust them.

In the cases where the perceptron predicts the outputs incorrectly, we need to adjust the weights as shown in Code Snippet 2-1, and we see that the weight adjustment is computed by combining the desired y value, the input value, and a parameter known as LEARNING_RATE. We now show why the weights are adjusted the way they are. Let us consider three different training examples where x_0 represents the bias input that is always 1:

Training example 1: $x_0 = 1, x_1 = 0, x_2 = 0, y = 1$

Training example 2: $x_0 = 1, x_1 = 0, x_2 = 1.5, y = -1$

Training example 3: $x_0 = 1, x_1 = -1.5, x_2 = 0, y = 1$

We further know that the z-value (the input to the signum function) for our perceptron is computed as

$$z = w_0x_0 + w_1x_1 + w_2x_2$$

For training example 1, the result is

$$z = w_0 1 + w_1 0 + w_2 0 = w_0$$

Clearly, w_1 and w_2 do not affect the result, so the only weight that makes sense to adjust is w_0. Further, if the desired output value is positive ($y = 1$), then we would want to increase the value of w_0. On the other hand, if the desired output value is negative ($y = -1$), then we want to decrease the value of w_0. Assuming that the LEARNING_RATE parameter is positive, Code Snippet 2-1 does exactly this when it adjusts w_i by adding a value that is computed as `y * LEARNING_RATE * x[i]`, where x_1 and x_2 are zero for training example 1 and thus only w_0 will be adjusted.

Doing the same kind of analysis for training example 2, we see that only w_0 and w_2 will be adjusted, both in a negative direction because y is -1 and x_0 and x_2 are positive. Further, the magnitude of the adjustment for w_2 is greater than for w_0 since x_2 is greater than x_1.

Similarly, for training example 3, only w_0 and w_1 will be adjusted, where w_0 will be adjusted in a positive direction and w_1 will be adjusted in a negative direction because y is positive and x_1 is negative.

To make this even more concrete, we compute the adjustment value for each weight for the three training examples, with an assumed learning rate of 0.1. They are summarized in Table 2-1.

We make a couple of observations:

- The adjustment of the bias weight depends only on the desired output value and will thus be determined by whether the majority of the training examples have positive or negative desired outputs.[1]
- For a given training example, only the weights that can significantly affect the output will see a significant adjustment, because the adjustments are proportional to the input values. In the extreme, where an input value is 0 for a training example, its corresponding weight will see zero adjustment.

This makes much sense. In a case where more than 50% of the training examples have the same output value, adjusting the bias weight toward that output value will make the perceptron be right more than 50% of the time if all other weights are 0. It also makes sense to not adjust weights that do not have a big impact on a given training example, which will likely do more harm than good because the weight could have a big impact on other training examples.

In Chapter 1, we described how the z-value of a two-input (plus bias term) perceptron creates a plane in a 3D space (where x_1 is one dimension, x_2 the second, and the resulting value z is the third). One way to visualize the perceptron learning algorithm is to consider how it adjusts the orientation of this plane. Every

Table 2-1 Adjustment Values for Each Weight for the Three Training Examples

	W_0 CHANGE	W_1 CHANGE	W_2 CHANGE
Example 1	1*1*0.1 = 0.1	1*0*0.1 = 0	1*0*0.1 = 0
Example 2	(–1)*1*0.1 = –0.1	(–1)*0*0.1 = 0	(–1)*1.5*0.1 = –0.15
Example 3	1*1*0.1 = 0.1	1*(–1.5)*0.1 = –0.15	1*0*0.1 = 0

1. Only training examples that are incorrectly predicted will cause an adjustment. Thus, a case with many training examples with positive outputs can still result in a negative bias weight if many of the positive training examples already are correctly predicted and thus do not cause any weight adjustments.

update will adjust the bias weight. This will push the overall plane upward for positive training examples and downward for negative training examples.

For example, close to the z-axis (x_1 and x_2 are small), the bias weight is all that counts. For cases that are further away from the z-axis, the angle of the plane becomes a more significant lever. Thus, for mispredicted learning examples where the x_1 value is big, we make a big change to the weight that determines the tilt angle in the x_1 direction, and the same applies for cases with big x_2 values but in the orthogonal direction. A point on the plane that is located directly on the x_2 axis will not move as we rotate the plane around the x_2 axis, which is what we do when we adjust the weight corresponding to the x_1 value.

An attempt at illustrating this is shown in Figure 2-1, with $w_0 = 1.0$, $w_1 = -1.0$, and $w_2 = -1.0$, which are weights that we could imagine would result from repeatedly applying the weight adjustments from Table 2-1.

Looking at the plane, we can now reason about how it satisfies the three training examples. Because $w_0 = 1.0$, the output will be positive when x_1 and x_2 are close to zero ($z = 1.0$ when x_1 and x_2 are 0), which will ensure that training example 1 is correctly handled. We further see that w_1 is chosen so that the plane is slanted in a direction that z increases as x_1 decreases. This ensures that training example 3 is taken care of because it has a negative x_1 value and wants a positive output. Finally, w_2 is chosen so that the plane is slanted in a direction (around its other axis) that z increases as x_2 decreases. This satisfies training example 2 with its positive x_2 input and desired negative output value.

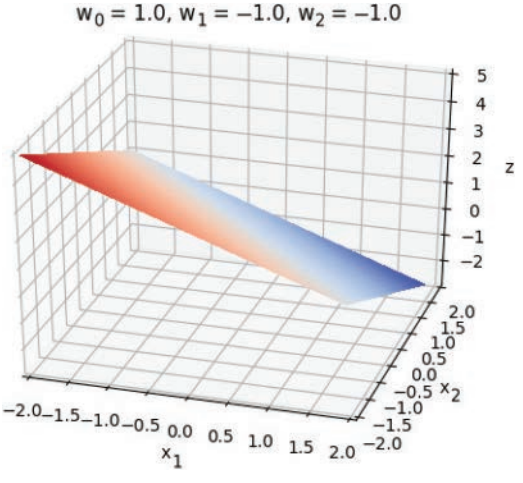

Figure 2-1 Example of weights that correctly predicts all three training examples

We believe that the reasoning is sufficient to give most people an intuitive idea of why the learning algorithm works the way it does. It also turns out that for cases that are linearly separable (i.e., cases where a perceptron has the ability to distinguish between the two classes), this learning algorithm is guaranteed to converge to a solution. This is true regardless of the magnitude of the learning rate parameter. In other words, the value of this parameter will only affect how quickly the algorithm converges.

To prepare ourselves for the learning algorithm for multilevel networks, we would now like to arrive at an analytical explanation of why we adjust the weights the way that we do in the perceptron learning algorithm, but we will first go through some concepts from calculus and numerical optimization that we will build upon.

Derivatives and Optimization Problems

In this section, we briefly introduce the mathematical concepts that we use in this chapter. It is mostly meant as a refresher for readers who have not used calculus lately, so feel free to skip to the next section if that does not apply to you. We start by briefly revisiting what a derivative is. Given a function

$$y = f(x)$$

the derivative of y with respect to x tells us how much the value of y changes given a small change in x. A few common notations are

$$y', \quad f'(x), \quad \frac{dy}{dx}$$

The first notation (y') can be somewhat ambiguous if y is a function of multiple variables, but in this case, where y is only a function of x, the notation is unambiguous. Because our neural networks typically are functions of many variables, we will prefer the two other notations.

Figure 2-2 plots the value of an arbitrary function $y = f(x)$. The plot also illustrates the derivative $f'(x)$ by plotting the tangent line in three different points. The tangent to a curve is a straight line with the same slope (derivative) as the curve at the location where the line touches the curve.

We can make a couple of observations. First, the derivative at the point that minimizes the value of y is 0 (the tangent is horizontal). Second, as we move further away from the minimum, the derivative increases (it decreases if we move in the other direction). We can make use of these observations when solving an

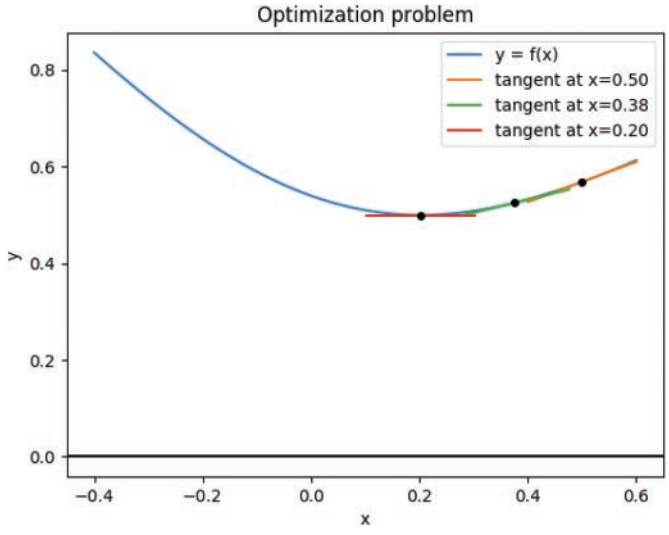

Figure 2-2 Plot showing a curve $y = f(x)$ and its derivative at the minimum value and in two other points

optimization problem in which we want to find what value of the variable x will minimize[2] the value of the function y. Given an initial value x and its corresponding y, the sign of the derivative indicates in what direction to adjust x to reduce the value of y. Similarly, if we know how to solve x for 0, we will find an extreme point (minimum, maximum, or saddle point)[3] of y.

As we saw in Chapter 1, we typically work with many variables. Therefore, before moving on to how to apply these concepts to neural networks, we need to extend them to two or more dimensions. Let us assume that we have a function of two variables, that is, $y = f(x_0, x_1)$, or alternatively, $y = f(\boldsymbol{x})$,where $\boldsymbol{x}$ is a 2D vector. This function can be thought of as a landscape that can contain hills and valleys,[4] as in Figure 2-3.

We can now compute two partial derivatives:

$$\frac{\partial y}{\partial x_0} \text{ and } \frac{\partial y}{\partial x_1}$$

2. We assume that the optimization problem is a minimization problem. There are also maximization problems, but we can convert a maximization problem into a minimization problem by negating the function we want to maximize.
3. We worry only about minima in this book. Further, it is worth noting that these extreme points may well be local extremes. That is, there is no guarantee that a global minimum is found.
4. It can also contain discontinuities, asymptotes, and so on.

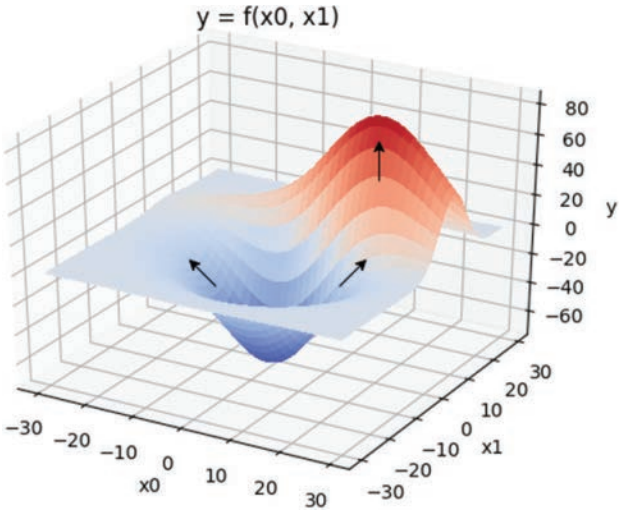

Figure 2-3 Plot of function of two variables and the direction and slope of steepest ascent in three different points

A partial derivative is just like a normal derivative, but we pretend that all but one of the variables are constants. The one variable that we want to compute the derivative with respect to is the only variable that is not treated as a constant. A simple example is if we have the function $y = ax_0 + bx_1$, in which case our two partial derivatives become

$$\frac{\partial y}{\partial x_0} = a$$

$$\frac{\partial y}{\partial x_1} = b$$

If we arrange these partial derivatives in a vector, we get

$$\nabla y = \begin{pmatrix} \frac{\partial y}{\partial x_0} \\ \frac{\partial y}{\partial x_1} \end{pmatrix}$$

which is called the *gradient* of the function—that is, the gradient is a derivative but generalized to a function with multiple variables. The symbol ∇ (upside-down Greek letter delta) is pronounced "nabla."

The gradient has a geometric interpretation. Being a vector, the gradient consists of a direction and a magnitude.[5] The direction is defined in the same dimensional space as the inputs to the function. That is, in our example, this is the 2D space represented by the horizontal plane in Figure 2-3. For our example gradient, the direction is (*a*, *b*). Geometrically, this *direction* indicates where to move from a given point (x_0, x_1) in order for the resulting function value (*y*) to increase the most. That is, it is the direction of the steepest ascent. The *magnitude* of the gradient indicates the slope of the hill in that direction.

The three arrows in Figure 2-3 illustrate both the direction and the slope of the steepest ascent in three point. Each arrow is defined by the gradient in its point, but the arrow does not represent the gradient vector itself. Remember that the direction of the gradient falls in the horizontal plane, whereas the arrows in the figure also have a vertical component that illustrates the slope of the hill in that point.

There is nothing magic about two input dimensions, but we can compute partial derivatives of a function of any number of dimensions and create the gradient by arranging them into a vector. However, this is not possible to visualize in a chart.

Solving a Learning Problem with Gradient Descent

One way to state our learning problem is to identify the weights that, given the input values for a training example, result in the network output matching the desired output for that training example. Mathematically, this is the same as solving the following equation:

$$y - \hat{y} = 0$$

where *y* is the desired output value and $\hat{y}$ (pronounced "y hat") is the value predicted by the network. In reality, we do not have just a single training example (data point), but we have a set of training examples that we want our function

5. This explanation assumes that you are familiar with the direction and magnitude of vectors. The magnitude can be computed using the distance formula that is derived from the Pythagorean theorem. Details of this theorem can be found in texts about linear algebra.

to satisfy. We can combine these multiple training examples into a single error metric by computing their mean squared error (MSE):[6]

$$\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}-\hat{y}^{(i)}\right)^2 \qquad (\textit{mean squared error})$$

The notation with a superscript number inside parentheses is used to distinguish between different training examples. It is not an indication to raise y to the power of i. Looking closer, it seems like using MSE presents a problem. For most problems, the MSE is strictly greater than 0, so trying to solve it for 0 is impossible. Instead, we will treat our problem as an optimization problem in which we try to find weights that *minimize the value* of the error function.

In most deep learning (DL) problems, it is not feasible to find a closed form solution[7] to this minimization problem. Instead, a numerical method known as *gradient descent* is used. It is an iterative method in which we start with an initial guess of the solution and then gradually refine it. Gradient descent is illustrated in Figure 2-4, where we start with an initial guess x_0. We can insert this value into

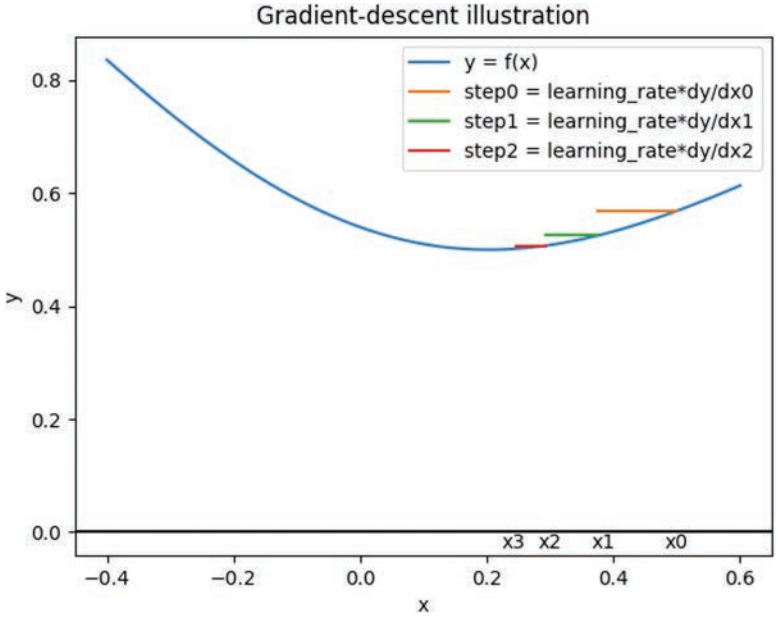

Figure 2-4 Gradient descent in one dimension

6. We will later see that MSE is not necessarily a great error function for some neural networks, but we use it for now because many readers are likely familiar with it.
7. A closed form solution is a solution found by analytically solving an equation to find an exact solution. An alternative is to use a numerical method. A numerical method often results in an approximate solution.

$f(x)$ and compute the corresponding y as well as its derivative. Assuming that we are not already at the minimum value of y, we can now come up with an improved guess x_1 by either increasing or decreasing x_0 slightly. The sign of the derivative indicates whether we should increase or decrease x_0. A positive slope (as in the figure), indicates that y will decrease if we decrease x. We can then iteratively refine the solution by repeatedly doing small adjustments to x.

Gradient descent is a commonly used learning algorithm in DL.

In addition to indicating in what direction to adjust x, the derivative provides an indication of whether the current value of x is close to or far away from the value that will minimize y. Gradient descent makes use of this property by using the value of the derivative to decide how much to adjust x. This is shown in the update formula used by gradient descent:

$$x_{n+1} = x_n - \eta f'(x_n)$$

where η (Greek letter eta) is a parameter known as the *learning rate*. We see that the step size depends on both the learning rate and the derivative, so the step size will decrease as the derivative decreases. The preceding figure illustrates the behavior of gradient descent using a learning rate (η) of 0.3. We see how the step size decreases as the derivative gets closer to 0. As the algorithm converges at the minimum point, the fact that the derivative approaches 0 implies that the step size also approaches 0.

If the learning rate is set to too large a value, gradient descent can also overshoot the solution and fail to converge. Further, even with a small step size, the algorithm is not guaranteed to find the global minimum because it can get stuck in a local minimum. However, in practice, it has been shown to work well for neural networks.

If you have encountered numerical optimization problems before, chances are that you have used a different iterative algorithm known as the *Newton-Raphson* or *Newton's method*. If you are curious about how it relates to gradient descent, you can find a description in Appendix E.

GRADIENT DESCENT FOR MULTIDIMENSIONAL FUNCTIONS

The preceding example worked with a function of a single variable, but our neural networks are functions of many variables, so we need the ability to

minimize multidimensional functions. Extending it to more dimensions is straightforward. As described in the section "Derivatives and Optimization Problems," a gradient is a vector consisting of partial derivatives and indicates the direction in the input space that results in the steepest ascent for the function value. Conversely, the negative gradient is the direction of steepest descent, or the direction of the quickest path to reducing the function value. Therefore, if we are at the point $\boldsymbol{x} = (x_0, x_1)$ and want to minimize y, then we choose our next point as

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} - \eta \nabla y$$

where ∇y is the gradient. This generalizes to functions of any number of dimensions. In other words, if we have a function of n variables, then our gradient will consist of n partial derivatives, and we can compute the next step as

$$\boldsymbol{x} - \eta \nabla y$$

where both $\boldsymbol{x}$ and ∇y are vectors consisting of n elements. Figure 2-5 shows gradient descent for a function of two input variables. The function value y gradually decreases as we move from point 1 to point 2 and 3.

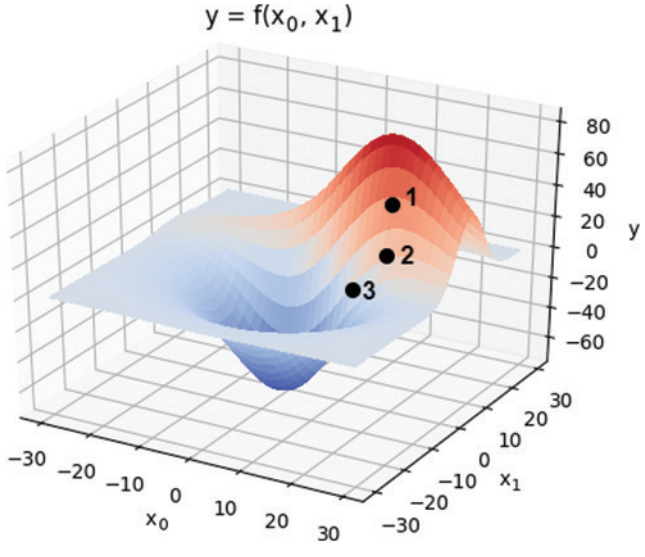

Figure 2-5 Gradient descent for a function of two variables

It is worth noting again that the algorithm can get stuck in a local minimum. There are various ways of trying to avoid this, some of which are mentioned in later chapters but are not discussed in depth in this book.

We are now almost ready to apply gradient descent to our neural networks. First, we need to point out some pitfalls related to working with the multidimensional functions implemented by neural networks.

Constants and Variables in a Network

A key idea when applying gradient descent to our neural network is that we consider input values (***x***) to be constants, with our goal being to adjust the weights (***w***), including the bias input weight (w_0). This might seem odd given our description of gradient descent, where we try to find input values that minimize a function. At first sight, for the two-input perceptron, it seems like x_1 and x_2 would be considered input values. That would be true if we had a perceptron with fixed weights and a desired output value and the task at hand was to find the x-values that result in this output value given the fixed weights. However, this is not what we are trying to do with our learning algorithm. The purpose of our learning algorithm is to, given a fixed input (x_1, x_2), adjust the weights (w_0, w_1, w_2) so that the output value takes on the value we want to see. That is, we treat x_1 and x_2 as constants (x_0 as well, but that is always the constant 1, as stated earlier), while we treat w_0, w_1, and w_2 as variables that we can adjust.

During **learning,** not the inputs (***x***) but the **weights (*w*) are considered to be the variables** in our function.

To make it more concrete, if we are training a network to distinguish between a dog and a cat, the pixel values would be the inputs (***x***) to the network. If it turned out that the network incorrectly classified a picture of a dog as being a cat, we would not go ahead and adjust the picture to look more like a cat. Instead, we would adjust the weights of the network to try to make it correctly classify the dog as being a dog.