Chapter 7

Other Propagation Training

- Resilient Propagation
- Levenberg-Marquardt
- Hessian and Jacobean Matrices

The backpropagation algorithm has influenced many training algorithms, such as the stochastic gradient descent (SGD), introduced in the previous chapter. For most purposes, the SGD algorithm, along with Nesterov momentum, is a good choice for a training algorithm. However, other options exist. In this chapter, we examine two popular algorithms inspired by elements from backpropagation.

To make use of these two algorithms, you do not need to understand every detail of their implementation. Essentially, both algorithms accomplish the same objective as backpropagation. Thus, you can substitute them for backpropagation or stochastic gradient descent (SGD) in most neural network frameworks. If you find SGD is not converging, you can switch to resilient propagation (RPROP) or Levenberg-Marquardt algorithm in order to experiment. However, you can skip this chapter if you are not interested in the actual implementation details of either algorithm.

7.1 Resilient Propagation

RPROP functions very much like backpropagation. Both backpropagation and RPROP must first calculate the gradients for the weights of the neural network. However, backpropagation and RPROP differ in the way they use the gradients. Reidmiller & Braun (1993) introduced the RPROP algorithm.

One important feature of the RPROP algorithm is that it has no necessary training parameters. When you utilize backpropagation, you must specify the learning rate and momentum. These two parameters can greatly impact the effectiveness of your training. Although RPROP does include a few training parameters, you can almost always leave them at their default.

The RPROP protocol has several variants. Some of the variants are listed below:

- RPROP+
- RPROP-
- iRPROP+
- iRPROP-

We will focus on classic RPROP, as described by Reidmiller & Braun (1994). The other four variants described above are relatively minor adaptations of classic RPROP. In the next sections, we will describe how to implement the classic RPROP algorithm.

7.2 RPROP Arguments

As previously mentioned, one advantage RPROP has over backpropagation is that you don't need to provide any training arguments in order to use RPROP. However, this doesn't mean that RPROP lacks configuration settings. It simply means that you usually do not need to change the configuration settings for RPROP from their defaults. However, if you really want to change them, you can choose among the following configuration settings:

- Initial Update Values
- Maximum Step

As you will see in the next section, RPROP keeps an array of update values for the weights, which determines how much you will alter each weight. This change is similar to the learning rate in backpropagation, but it is much better because the algorithm adjusts the update value of every weight in the neural network as training progresses. Although some backpropagation algorithms will vary the learning rate and momentum as learning progresses, most will use a single learning rate for the entire neural network. Therefore, the RPROP approach has an advantage over backpropagation algorithms.

We start these update values at the default of 0.1, according to the initial update values argument. As a general rule, we should never change this default. However, we can make an exception to this rule if we have already trained the neural network. In the case of a previously trained neural network, some of the initial update values are going to be too strong, and the neural network will regress for many iterations before it can improve. As a result, a trained neural network may benefit from a much smaller initial update.

Another approach for an already trained neural network is to save the update values once training stops and use them for the new training. This method will allow you to resume training without the initial spike in errors that you would normally see when resuming resilient propagation training. This approach will only work if you are continuing resilient propagation on an already trained network. If you were previously training the neural network with a different training algorithm, then you will be able to restore from an array of update values.

As training progresses, you will use the gradients to adjust the updates up and down. The *maximum step* argument defines the maximum upward step size that the gradient can take over the update values. The default value for the maximum step argument is 50. It is unlikely that you will need to change the value of this argument.

In addition to these arguments, RPROP keeps constants during processing. These are values that you can never change. The constants are listed as follows:

- Delta Minimum (1e-6)
- Negative η (Eta) (0.5)
- Positive -η (Eta) (1.2)
- Zero Tolerance (1e-16)

Delta minimum specifies the minimum value that any of the update values can reach. If an update value were at 0, it would never be able to increase beyond 0. We will describe negative and positive η (eta) in the next sections.

The *zero tolerance* defines how closely a number should reach 0 before that number is equal to 0. In computer programming, it is typically bad practice to compare a floating-point number to 0 because the number would have to equal 0 exactly. Rather, you typically see if the absolute value of a number is below an arbitrarily small number. A sufficiently small number is considered 0.

7.3 Data Structures

You must keep several data structures in memory while you perform RPROP training. These structures are all arrays of floating-point numbers. They are summarized here:

- Current Update Values
- Last Weight Change Values
- Current Weight Change Values
- Current Gradient Values
- Previous Gradient Values

You keep the current update values for the training. If you want to resume training at some point, you must store this update value array. Each weight has

one update value that cannot go below the minimum delta constant. Likewise, these update values cannot exceed the maximum step argument.

RPROP must keep several values between iterations. You must also track the last weight delta value. Backpropagation keeps the previous weight delta for momentum. RPROP uses this delta value in a different way that we will examine in the next section. You also need the current and previous gradients. RPROP needs to know when the sign changes from the current gradient to the previous gradient. This change indicates that you must act on the update values. We will discuss these actions in the next section.

7.4 Understanding RPROP

In the previous sections, we examined the arguments, constants, and data structures necessary for RPROP. In this section, we will show you an iteration of RPROP. When we discussed backpropagation in earlier sections, we mentioned the online and batch weight update methods. However, RPROP does not support online training so all weight updates for RPROP will be performed in batch mode. As a result, each iteration of RPROP will receive gradients that are the sum of the individual gradients of each training set. This aspect is consistent with backpropagation in batch mode.

7.4.1 Determine Sign Change of Gradient

At this point, we have the gradients that are the same as the gradients calculated by the backpropagation algorithm. Because we use the same process to obtain gradients in both RPROP and backpropagation, we will not repeat it here. For the first step, we compare the gradient of the current iteration to the gradient of the previous iteration. If there is no previous iteration, then we can assume that the previous gradient was 0.

To determine whether the gradient sign has changed, we will use the sign (sgn) function. Equation 7.1 defines the sgn function:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases} \tag{7.1}$$

The sgn function returns the sign of the number provided. If x is less than 0, the result is -1. If x is greater than 0, then the result is 1. If x is equal to 0, then the result is 0. We usually implement the sgn function to use a tolerance for 0, since it is nearly impossible for floating-point operations to hit 0 precisely on a computer.

To determine whether the gradient has changed sign, we use Equation 7.2:

$$c = \frac{\partial E}{\partial w_{ij}}^{(t)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t-1)} \tag{7.2}$$

Equation 7.2 will result in a constant c. We evaluate this value as negative or positive or close to 0. A negative value for c indicates that the sign has changed. A positive value indicates that there is no change in sign for the gradient. A value near 0 indicates a very small change in sign or almost no change in sign.

Consider the following situations for these three outcomes:

```
-1 * 1 = -1  (negative, changed from negative to positive)
1 * 1 = 1 (positive, no change in sign)
1.0 * 0.000001 = 0.000001 (near zero, almost changed signs, but
    not quite)
```

Now that we have calculated the constant c, which gives some indication of sign change, we can calculate the weight change. The next section includes a discussion of this calculation.

7.4.2 Calculate Weight Change

Now that we have the change in sign of the gradient, we can observe what happens in each of the three cases mentioned in the previous section. Equation 7.3 summarizes these three cases:

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)} & \text{, if } c > 0 \\ +\Delta_{ij}^{(t)} & \text{, if } c < 0 \\ 0 & \text{, otherwise} \end{cases} \tag{7.3}$$

This equation calculates the actual weight change for each iteration. If the value of c is positive, then the weight change will be equal to the negative of the weight update value. Similarly, if the value of c is negative, the weight change will be equal to the positive of the weight update value. Finally, if the value of c is near 0, there will be no weight change.

we create a hybrid method. To understand how this hybrid works, we will first examine Newton's method. Equation 7.5 shows Newton's method:

$$W_{min} = W_0 - H^{-1}g \tag{7.5}$$

You will notice several variables in the above equation. The result of the equation is that you can apply deltas to the weights of the neural network. The variable H represents the Hessian, which we will discuss in the next section. The variable g represents the gradients of the neural network. You will also notice the -1 "exponent" on the variable H, which specifies that we are doing a matrix decomposition of the variables H and g.

We could easily spend an entire chapter on matrix decomposition. However, we will simply treat matrix decomposition as a black box atomic operator for the purposes of this book. Because we will not explain how to calculate matrix decomposition, we have included a common piece of code taken from the JAMA package. Many mathematical computer applications have used this public domain code, adapted from a FORTRAN program. To perform matrix decomposition, you can use JAMA or another source.

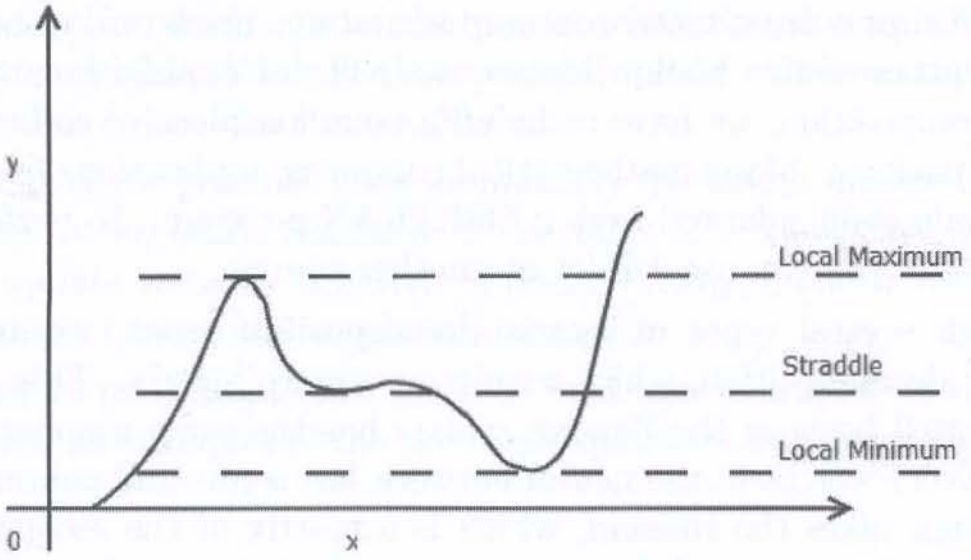
Although several types of matrix decomposition exist, we are going to use the LU decomposition, which requires a square matrix. This decomposition works well because the Hessian matrix has the same number of rows as columns. Every weight in the neural network has a row and column. The LU decomposition takes the Hessian, which is a matrix of the second derivative of the partial derivatives of the output of each of the weights. The LU decomposition solves the Hessian by the gradients, which are the square of the error of each weight. These gradients are the same as those that we calculated in Chapter 6, "Backpropagation Training," except they are squared. Because the errors are squared, we must use the sum of square error when dealing with LMA.

Second derivative is an important term to know. It is the derivative of the first derivative. Recall from Chapter 6, "Backpropagation Training," that the derivative of a function is the slope at any point. This slope shows the direction that the curve is approaching for a local minimum. The second derivative is also a slope, and it points in a direction to minimize the first derivative. The goal of Newton's method, as well as of the LMA, is to reduce all of the gradients to 0.

It's interesting to note that the goal does not include the error. Newton's method and LMA can be oblivious to the error because they try to reduce all the gradients to 0. In reality, they are not completely oblivious to the error because they use it to calculate the gradients.

Newton's method will converge the weights of a neural network to a local minimum, a local maximum, or a straddle position. We achieve this convergence by minimizing all the gradients (first derivatives) to 0. The derivatives will be 0 at local minima, maxima, or straddle position. Figure 7.1 shows these three points:

Figure 7.1: Local Minimum, Straddle and Local Maximum



The algorithm implementation must ensure that local maxima and straddle points are filtered out. The above algorithm works by taking the matrix decomposition of the Hessian matrix and the gradients. The Hessian matrix is typically estimated. Several methods exist to estimate the Hessian matrix. However, if it is inaccurate, it can harm Newton's method.

LMA enhances Newton's algorithm to the following formula in Equation 7.6:

$$W_{min} = W_0 - (H + \lambda I)^{-1} g \tag{7.6}$$

In this equation, we add a damping factor multiplied by an identity matrix. The damping factor is represented by λ (lambda), and I represents the iden-

tity matrix, which is a square matrix with all the values at 0 except for a northwest (NW) line of values at 1. As lambda increases, the Hessian will be factored out of the above equation. As lambda decreases, the Hessian becomes more significant than gradient descent, allowing the training algorithm to interpolate between gradient descent and Newton's method. Higher lambda favors gradient descent; lower lambda favors Newton. A training iteration of LMA begins with a low lambda and increases it until a desirable outcome is produced.

7.6 Calculation of the Hessian

The Hessian matrix is a square matrix with rows and columns equal to the number of weights in the neural network. Each cell in this matrix represents the second order derivative of the output of the neural network with respect to a given weight combination. Equation 7.7 shows the Hessian:

$$H(e) = \begin{bmatrix} \frac{\partial^2 e}{\partial w_1^2} & \frac{\partial^2 e}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_1 \partial w_n} \\ \frac{\partial^2 e}{\partial w_2 \partial w_1} & \frac{\partial^2 e}{\partial w_2^2} & \cdots & \frac{\partial^2 e}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 e}{\partial w_n \partial w_1} & \frac{\partial^2 e}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_n^2} \end{bmatrix} \tag{7.7}$$

It is important to note that the Hessian is symmetrical about the diagonal, which you can use to enhance performance of the calculation. Equation 7.8 calculates the Hessian by calculating the gradients:

$$\frac{\partial E}{\partial w_{(i)}} = 2(y - t)\frac{\partial y}{\partial w_{(i)}} \tag{7.8}$$

The second derivative of the above equation becomes an element of the Hessian matrix. You can use Equation 7.9 to calculate it:

$$\frac{\partial^2 E}{\partial w_i w_j} = 2\left(\frac{\partial y}{\partial w_i}\frac{\partial y}{\partial w_j} + (y-t)\frac{\partial^2 y}{\partial w_j \partial w_j}\right) \tag{7.9}$$

If not for the second component, you could easily calculate the above formula. However, this second component involves the second partial derivative and that is difficult to calculate. Because the component is not important, you can actually drop it because its value does not significantly contribute to the outcome. While the second partial derivative might be important for an individual training case, its overall contribution is not significant. The second component of Equation 7.9 is multiplied by the error of that training case. We assume that the errors in a training set are independent and evenly distributed about 0. On an entire training set, they should essentially cancel each other out. Because we are not using all components of the second derivative, we have only an approximation of the Hessian, which is sufficient to get a good training result.

Equation 7.10 uses the approximation, resulting in the following:

$$\frac{\partial^2 E}{\partial w_i w_j} = 2\left(\frac{\partial y}{\partial w_i}\frac{\partial y}{\partial w_j}\right) \tag{7.10}$$

While the above equation is only an approximation of the true Hessian, the simplification of the algorithm to calculate the second derivative is well worth the loss in accuracy. In fact, an increase in λ (lambda) will account for the loss of accuracy.

To calculate the Hessian and gradients, we must determine the partial first derivatives of the output of the neural network. Once we have these partial first derivatives, the above equations allow us to easily calculate the Hessian and gradients.

Calculation of the first derivatives of the output of the neural network is very similar to the process that we used to calculate the gradients for backpropagation. The main difference is that we take the derivative of the output. In standard backpropagation, we take the derivative of the error function. We will not review the entire backpropagation process here. Chapter 6, "Backpropagation Training," covers backpropagation and gradient calculation.

7.7 LMA with Multiple Outputs

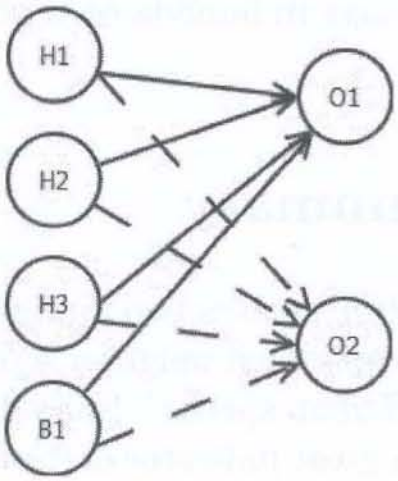
Some implementations of LMA support only a single-output neuron because LMA has roots in mathematical function approximation. In mathematics, functions typically return only a single value. As a result, many books and papers do not contain discussions of multiple-output LMA. However, you can use LMA with multiple outputs.

Support for multiple-output neurons involves summing each cell of the Hessian as you calculate the additional output neurons. The process works as if you calculated a separate Hessian matrix for each output neuron and then summed the Hessian matrices together. Encog (Heaton, 2015) uses this approach, and it leads to fast convergence times.

You need to realize that you will not use every connection with multiple outputs. You will need to calculate independently an update for the weight of each output neuron. Depending on the output neuron you are currently calculating, there will be unused connections for the other output neurons. Therefore, you must set the partial derivative for each of these unused connections to 0 when you are calculating the other output neurons.

For example, consider a neural network that has two output neurons and three hidden neurons. Each of these two output neurons would have a total of four connections from the hidden layer. Three connections result from the three hidden neurons, and a fourth comes from the bias neuron. This segment of the neural network would resemble Figure 7.2:

Figure 7.2: Calculating Output Neuron 1



Here we are calculating output neuron 1. Notice that output neuron 2 has four connections that must have their partial derivatives treated as 0. Because we are calculating output 1 as the current neuron, it only uses its normal partial derivatives. You can repeat this process for each output neuron.

7.8 Overview of the LMA Process

So far, we have examined only the math behind LMA. To be effective, LMA must be part of an algorithm. The following steps summarize the LMA process:

1. Calculate the first derivative of output of the neural network with respect to every weight.
2. Calculate the Hessian.
3. Calculate the gradients of the error (ESS) with respect to every weight.
4. Either set lambda to a low value (first iteration) or the lambda of the previous iteration.
5. Save the weights of the neural network.
6. Calculate delta weight based on the lambda, gradients, and Hessian.
7. Apply the deltas to the weights and evaluate error.
8. If error has improved, end the iteration.
9. If error has not improved, increase lambda (up to a max lambda), restore the weights, and go back to step 6.

As you can see, the process for LMA revolves around setting the lambda value low and then slowly increasing it if the error rate does not improve. You must save the weights at each change in lambda so that you can restore them if the error does not improve.

7.9 Chapter Summary

Resilient propagation (RPROP) solves two limitations of simple backpropagation. First, the program assigns each weight a separate learning rate, allowing the weights to learn at different speeds. Secondly, RPROP recognizes that while the gradient's sign is a great indicator of the direction to move the weight,

the size of the gradient does not indicate how far to move. Additionally, while the programmer must determine an appropriate learning rate and momentum for backpropagation, RPROP automatically sets similar arguments.

Genetic algorithms (GAs) are another means of training neural networks. There is an entire family of neural networks that use GAs to evolve every aspect of the neural network, from weights to the overall structure. This family includes the NEAT, CPPN and HyperNEAT neural networks that we will discuss in the next chapter. The GA used by NEAT, CPPN and HyperNEAT is not just another training algorithm because these neural networks introduce a new architecture based on the feedforward neural networks examined so far in this book.