methods usually converge rapidly and are robust, but they require detailed information about the objective function, which can be difficult to obtain in some applications.

9.1 LIMITED-MEMORY BFGS

Limited-memory quasi-Newton methods are useful for solving large problems whose Hessian matrices cannot be computed at a reasonable cost or are too dense to be manipulated easily. These methods maintain simple and compact approximations of Hessian matrices: Instead of storing fully dense $n \times n$ approximations, they save just a few vectors of length n that represent the approximations implicitly. Despite these modest storage requirements, they often yield an acceptable (albeit linear) rate of convergence. Various limited-memory methods have been proposed; we will focus mainly on an algorithm known as L-BFGS, which as its name suggests, is based on the BFGS updating formula. The main idea of this method is to use curvature information from only the most recent iterations to construct the Hessian approximation. Curvature information from earlier iterations, which is less likely to be relevant to the actual behavior of the Hessian at the current iteration, is discarded in the interests of saving storage.

We begin our description of the L-BFGS method by recalling its parent, the BFGS method, which was described in Algorithm 8.1. Each step of the BFGS method has the form

$$x_{k+1} = x_k - \alpha_k H_k \nabla f_k, \qquad k = 0, 1, 2, \ldots, \tag{9.1}$$

where α_k is the step length, and H_k is updated at every iteration by means of the formula

$$H_{k+1} = V_k^T H_k V_k + \rho_k s_k s_k^T \tag{9.2}$$

(see (8.16)), where

$$\rho_k = \frac{1}{y_k^T s_k}, \qquad V_k = I - \rho_k y_k s_k^T, \tag{9.3}$$

and

$$s_k = x_{k+1} - x_k, \qquad y_k = \nabla f_{k+1} - \nabla f_k. \tag{9.4}$$

We say that the matrix H_{k+1} is obtained by updating H_k using the pair $\{s_k, y_k\}$.

The inverse Hessian approximation H_k will generally be dense, so that the cost of storing and manipulating it is prohibitive when the number of variables is large. To circumvent this problem, we store a *modified* version of H_k *implicitly*, by storing a certain number (say m) of the vector pairs $\{s_i, y_i\}$ that are used in the formulae (9.2)–(9.4). The product $H_k \nabla f_k$ can be obtained by performing a sequence of inner products and vector summations involving

∇f_k and the pairs $\{s_i, y_i\}$. After the new iterate is computed, the oldest vector pair in the set of pairs $\{s_i, y_i\}$ is deleted and replaced by the new pair $\{s_k, y_k\}$ obtained from the current step (9.4). In this way, the set of vector pairs includes curvature information from the m most recent iterations. Practical experience has shown that modest values of m (between 3 and 20, say) often produce satisfactory results. Apart from the modified matrix updating strategy and a modified technique for handling the initial Hessian approximation (described below), the implementation of L-BFGS is identical to that of the standard BFGS method given in Algorithm 8.1. In particular, the same line search strategy can be used.

We now describe the updating process in a little more detail. At iteration k, the current iterate is x_k and the set of vector pairs contains $\{s_i, y_i\}$ for $i = k - m, \ldots, k - 1$. We first choose some initial Hessian approximation H_k^0 (in contrast to the standard BFGS iteration, this initial approximation is allowed to vary from iteration to iteration), and find by repeated application of the formula (9.2) that the L-BFGS approximation H_k satisfies the following formula:

$$\begin{aligned} H_k = & \left(V_{k-1}^T \cdots V_{k-m}^T\right) H_k^0 \left(V_{k-m} \cdots V_{k-1}\right) \\ & + \rho_{k-m} \left(V_{k-1}^T \cdots V_{k-m+1}^T\right) s_{k-m} s_{k-m}^T \left(V_{k-m+1} \cdots V_{k-1}\right) \\ & + \rho_{k-m+1} \left(V_{k-1}^T \cdots V_{k-m+2}^T\right) s_{k-m+1} s_{k-m+1}^T \left(V_{k-m+2} \cdots V_{k-1}\right) \\ & + \cdots \\ & + \rho_{k-1} s_{k-1} s_{k-1}^T. \end{aligned} \tag{9.5}$$

From this expression we can derive a recursive procedure to compute the product $H_k \nabla f_k$ efficiently.

Algorithm 9.1 (L-BFGS two-loop recursion).

$q \leftarrow \nabla f_k$;
for $i = k - 1, k - 2, \ldots, k - m$
  $\alpha_i \leftarrow \rho_i s_i^T q$;
  $q \leftarrow q - \alpha_i y_i$;
end (**for**)
$r \leftarrow H_k^0 q$;
for $i = k - m, k - m + 1, \ldots, k - 1$
  $\beta \leftarrow \rho_i y_i^T r$;
  $r \leftarrow r + s_i(\alpha_i - \beta)$
end (**for**)
stop with result $H_k \nabla f_k = r$.

Without considering the multiplication $H_k^0 q$, the two-loop recursion scheme requires $4mn$ multiplications; if H_k^0 is diagonal, then n additional multiplications are needed. Apart from being inexpensive, this recursion has the advantage that the multiplication by the initial matrix H_k^0 is isolated from the rest of the computations, allowing this matrix to be chosen

freely and to vary between iterations. We may even use an implicit choice of H_k^0 by defining some initial approximation B_k^0 to the Hessian (not its inverse), and obtaining r by solving the system $B_k^0 r = q$.

A method for choosing H_k^0 that has proved to be effective in practice is to set $H_k^0 = \gamma_k I$, where

$$\gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}. \tag{9.6}$$

As discussed in Chapter 8, γ_k is the scaling factor that attempts to estimate the size of the true Hessian matrix along the most recent search direction (see (8.21)). This choice helps to ensure that the search direction p_k is well-scaled, and as a result the step length $\alpha_k = 1$ is accepted in most iterations. As discussed in Chapter 8, it is important that the line search be based on the Wolfe conditions (3.6) or strong Wolfe conditions (3.7), so that BFGS updating is stable.

The limited-memory BFGS algorithm can be stated formally as follows.

Algorithm 9.2 (L-BFGS).

Choose starting point x_0, integer $m > 0$;
$k \leftarrow 0$;
repeat
    Choose H_k^0 (for example, by using (9.6));
    Compute $p_k \leftarrow -H_k \nabla f_k$ from Algorithm 9.1;
    Compute $x_{k+1} \leftarrow x_k + \alpha_k p_k$, where α_k is chosen to
        satisfy the Wolfe conditions;
    if $k > m$
        Discard the vector pair $\{s_{k-m}, y_{k-m}\}$ from storage;
    Compute and save $s_k \leftarrow x_{k+1} - x_k$, $y_k = \nabla f_{k+1} - \nabla f_k$;
    $k \leftarrow k + 1$;
until convergence.

During its first $m - 1$ iterations, Algorithm 9.2 is equivalent to the BFGS algorithm of Chapter 8 if the initial matrix H_0 is the same in both methods, and if L-BFGS chooses $H_k^0 = H_0$ at each iteration. In fact, we could reimplement the standard BFGS method by setting m to some large value in Algorithm 9.2 (larger than the number of iterations required to find the solution). However, as m approaches n (specifically, $m > n/2$), this approach would be more costly in terms of computer time and storage than the approach of Algorithm BFGS.

The strategy of keeping the m most recent correction pairs $\{s_i, y_i\}$ works well in practice, and no other strategy has yet proved to be consistently better. Other criteria may be considered, however—for example, one in which we maintain the set of correction pairs for which the matrix formed by the s_i components has the best conditioning, thus tending to avoid sets of vector pairs in which some of the s_i's are nearly collinear.