Limited Memory Options for quasi-Newton Methods

The major drawback for quasi-Newton methods is that they can be impractical when the number of decision variables is large. This is not so much because we need matrix-vector products such as Bs, but because storing $n \times n$ matrix B is prohibitive.

We can make progress when realizing that while B may be dense (few or no zeros) it is of low rank for iterations $k \ll n$. So, effectively we can "store B" by saving r vectors (where r is the rank of B).

Example:

$$M = \begin{pmatrix} 9 & -3 & 6 & 0 & 3 \\ -3 & 1 & -2 & 0 & -1 \\ 6 & -2 & 4 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 3 & -1 & 2 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 3 \\ -1 \\ 2 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 3 & -1 & 2 & 0 & 1 \end{pmatrix} = u u^T$$

We could store $n \times n$ matrix M or r $n \times 1$ vectors u.

Storage fraction $\frac{r \cdot n}{n \cdot n} = \frac{r}{n}$.

Consider SR1 Updating. Each iteration increases the rank of B by at most 1. So, $r \leq \min\{k, n\}$. It appears strategic to store r vectors rather than the entire $n \times n$ hessian matrix. Even so, we never wish to actually construct the hessian. What we need is a method of computing the Newton step directly from the stored vectors.

This is fairly simple if we have a representation $B_k = \sum_{i=1}^{k} u_i u_i^T$.

Then $B_k x = \sum_{i=1}^{k} u_i u_i^T x = \sum_{i=1}^{k} (u_i^T x) u_i$.

But quasi-Newton updates are recursive!

Consider the BFGS Update. let $V_k = I - \rho_k s_k s_k^T$

$$H_0 = H_0$$

$$H_1 = V_0^T H_0 V_0 + \rho_0 s_0 s_0^T$$

$$H_2 = V_1^T H_1 V_1 + \rho_1 s_1 s_1^T$$

$$= V_1^T \left[V_0^T H_0 V_0 + \rho_0 s_0 s_0^T \right] V_1 + \rho_1 s_1 s_1^T$$

$$= V_1^T V_0^T H_0 V_0 V_1^T + \rho_0 V_1^T s_0 s_0^T V_1 + \rho_1 s_1 s_1^T$$

$$H_3 = V_2^T H_2 V_2 + \rho_2 s_2 s_2^T$$

$$= V_2^T \left[V_1^T V_0^T H_0 V_0 V_1 + \rho_0 V_1^T s_0 s_0^T V_1 + \rho_1 s_1 s_1^T \right] V_2 + \rho_2 s_2 s_2^T$$

$$= V_2^T V_1^T V_0^T H_0 V_0 V_1 V_2 + \rho_0 V_2^T V_1^T s_0 s_0^T V_1 V_2 + \rho_1 V_2^T s_1 s_2 V_2 + \rho_2 s_2 s_2^T$$

$$\Rightarrow \boxed{\begin{aligned} H_k &= (V_{k-1}^T \cdots V_{k-m}) H_{k-m} (V_{k-m} \cdots V_{k-1}) \\ &+ \sum_{j=k-m}^{k-2} (V_{k-1}^T \cdots V_{j+1}^T) s_j s_j^T (V_{j+1} \cdots V_{k-1}) \\ &+ \rho_{k-1} s_{k-1} s_{k-1}^T \end{aligned}}$$

Storing H_{k-m} and the vectors V_j is sufficient to compute H_k.

And, more importantly: the following pseudocode demonstrates how to compute $r = H_k x$ using only V, ρ, s, y.

```
for i = k-1 to k-m
    α_i ← ρ_i s_i^T x
    x ← x − α_i y_i
end
r ← H_{k-m} x
for i = k-m to k-1
    β = ρ_i y_i^T r
    r ← r + s_i (α_i − β)
end
```

In particular, we need $p_k = -H_k \nabla f$ as the Newton step.

Notice in the above algorithm, we compute $H_k x$ very efficiently without use of large matrices or need of much memory ... <u>If</u>

(1) H_{k-m} is sparse ← H_0 is sparse, for example

(2) $m \ll n$ ← otherwise, storing H_k is more efficient!

This leads to the L-BFGS idea (Limited Memory BFGS)

At each iteration k, update H using only secant information from the previous m iterations and an initial estimate of H.

$$H_k = (V_{k-1}^T \cdots V_{k-m}) H_k^0 (V_{k-m} \cdots V_{k-1})$$
$$+ \sum_{j=k-m}^{k-2} (V_{k-1}^T \cdots V_{j+1}^T) s_j s_j^T (V_{j+1} \cdots V_{k-1})$$
$$+ \rho_{k-1} s_{k-1} s_{k-1}^T$$

$$H_k^0 = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} I$$

```
for i = k-1 to k-m
    α_i ← ρ_i s_i^T x
    x ← x − α_i y_i
end
r ← H_k^0 x
for i = k-m to k-1
    β = ρ_i y_i^T r
    r ← r + s_i (α_i − β)
end
```

L-BFGS update for $r = H_k x$

When $x = -\nabla f_k$ we have $r = -H_k \nabla f_k$ the Newton Step

Notice that the initial hessian H_k^0 is a diagonal approximation of the current hessian. Then it is modified by the m previous iteration data.

(Similar strategies exist for SR1)

L - BFGS Algorithm

Given : x_0, BFGS parameters, $m > 0$ integer

Set $k = 0$.

Until some stopping criterion is met

If $k = 1$

$H_k^0 = |f(x_0)| I$

Else

$$H_k^0 = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}$$

End

Compute $p_k = -H_k \nabla f_k$ by $\square$ algorithm.

Perform Strong Wolfe Line Search

If $k > m$

discard s_{k-m} and y_{k-m}

Save s_k and y_k

End

$k \leftarrow k+1$

End

Small values of m tend to work very well.

Say $m \approx 10$.