数值优化（Numerical Optimization）学习系列-大规模无约束最优化（Large-Scale Unconstrained Optimization）

概述

当最优化问题参数个数增加，求解问题所需要的时间和 空间复杂度 会增加。计算时间和空间是一个权衡，只需要存储一阶梯度时，时间复杂度可能为超线性；如果利用Hessian矩阵可以达到二次收敛，但是需要$o(n^2)$的空间复杂度。
另外对于拟牛顿算法所得到的Hessian矩阵式稠密的，不能利用到 稀疏矩阵 的一些性质。针对以上问题本小节给出求解大规模无约束最优化问题的一些思路，主要内容包括

1. 非精确牛顿方法
2. 基于有限内存的拟牛顿方法
3. 稀疏拟牛顿方法
4. 其他

非精确牛顿方法(Inexact Newton Methods)

在牛顿算法中，根据牛顿方程 $\nabla^2 f_k p_k^N = -\nabla f_k$，在Hessian矩阵存在并且可逆的情况下，可以得到搜索方向，并且收敛速度为二次收敛。但是一般情况下Hessian逆矩阵求解复杂度较高，对于稀疏的Hessian可以通过稀疏消元法或者矩阵分解得到。
非精确牛顿方法，可以快速得到搜索方向，一般通过共轭梯度（CG）或者Lanczos方法。通过这些方法可以不用显式存储Hessian矩阵，但是效率没有牛顿方法快。

局部收敛性

非精确牛顿方法需要保证残差满足一定规则保证收敛，对于非精确的搜索方向p_k，满足$r_k = \nabla^2 f_k p_k + \nabla f_k$，并且$||r_k|| \leq \eta_k ||\nabla f_k||$，其中$\eta_k$的序列称之为forcing sequence，决定了收敛速度。

定理：如果$\nabla^2 f_k$存在并且连续，存在最优解x*。对于非精确牛顿方法，如果初始点x_0在最优解附近，则有$x_k \to x^*$，并且
$||\nabla^2 f(x^*)(x_{k+1}-x^*)|| \le \hat{\eta}||\nabla^2 f(x^*)(x_{k+1}-x^*)||$
定理：在上述定理基础上如果每一步的$\eta_k \to 0$则收敛速度为超线性；进一步，如果$\nabla^2 f_k$Lipschitz连续并且$\eta_k = o(|\nabla f_k|)$则收敛速度为二次收敛。

注：当$\eta_k = min(0.5, \sqrt{||\nabla f_k||})$时，方法能得到超线性收敛速度；当$\eta_k = min(0.5, ||\nabla f_k||)$时，达到二次收敛速度。

Line Search Newton-CG方法

通过CG方法计算得到搜索方向，也称之为Truncated Newton 方法。由于CG算法需要保证Hessian正定，如果Hessian存在非正的特征值，则需要在找到某个下降方向则立即返回，算法如下：

Algorithm 7.1 (Line Search Newton–CG).
Given initial point x_0;
for $k = 0, 1, 2, \ldots$
  Define tolerance $\epsilon_k = \min(0.5, \sqrt{\|\nabla f_k\|})\|\nabla f_k\|$;
  Set $z_0 = 0, r_0 = \nabla f_k, d_0 = -r_0 = -\nabla f_k$;
  for $j = 0, 1, 2, \ldots$
    if $d_j^T B_k d_j \le 0$
      if $j = 0$
        return $p_k = -\nabla f_k$;
      else
        return $p_k = z_j$;
    Set $\alpha_j = r_j^T r_j / d_j^T B_k d_j$;
    Set $z_{j+1} = z_j + \alpha_j d_j$;
    Set $r_{j+1} = r_j + \alpha_j B_k d_j$;
    if $\|r_{j+1}\| < \epsilon_k$
      return $p_k = z_{j+1}$;
    Set $\beta_{j+1} = r_{j+1}^T r_{j+1} / r_j^T r_j$;
    Set $d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$;
  end (for)
  Set $x_{k+1} = x_k + \alpha_k p_k$, where α_k satisfies the Wolfe, Goldstein, or
    Armijo backtracking conditions (using $\alpha_k = 1$ if possible);
end

算法中关键点是CG算法，该方法能够适用于大规模问题，但是存在一个缺点是当实际中Hessian接近奇异时，该方向不是一个好的选择。

通过上述算法我们可以在不知道Hessian矩阵的前提下计算得到Hessian矩阵和向量的乘积即$\nabla^2 f_k d$，d可以为任何向量。另外通过自动微分的方法也可以得到类似解，思路为$\nabla^2 f_k d = \frac{\nabla f(x_k + hd) - \nabla f(x_k)}{h}$，其中需要核实的选择向量h。

Trust Region Newton-CG方法

在基础Cauchy Point算法的基础上，优化搜索方向。TR算法的建模方法为

$$m_k(p) = f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p,\ st.\ ||p|| \leq \Delta$$

，可以通过CG算法得到B.算法为

Algorithm 7.2 (CG–Steihaug).
Given tolerance $\epsilon_k > 0$;
Set $z_0 = 0, r_0 = \nabla f_k, d_0 = -r_0 = -\nabla f_k$;
if $\|r_0\| < \epsilon_k$
    return $p_k = z_0 = 0$;
for $j = 0, 1, 2, \ldots$
    if $d_j^T B_k d_j \leq 0$
        Find τ such that $p_k = z_j + \tau d_j$ minimizes $m_k(p_k)$ in (4.5)
            and satisfies $\|p_k\| = \Delta_k$;
        return p_k;
    Set $\alpha_j = r_j^T r_j / d_j^T B_k d_j$;
    Set $z_{j+1} = z_j + \alpha_j d_j$;
    if $\|z_{j+1}\| \geq \Delta_k$
        Find $\tau \geq 0$ such that $p_k = z_j + \tau d_j$ satisfies $\|p_k\| = \Delta_k$;
        return p_k;
    Set $r_{j+1} = r_j + \alpha_j B_k d_j$;
    if $\|r_{j+1}\| < \epsilon_k$
        return $p_k = z_{j+1}$;
    Set $\beta_{j+1} = r_{j+1}^T r_{j+1} / r_j^T r_j$;
    Set $d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$;
end (for).

该算法和Line Search方法的主要区别在于搜索方向需要满足一定约束。

Trust-Region Newton–LANCZOS方法

上述算法的缺点是会接受任何下降方向作为搜索方向，即使下降方向非常小。Lanczos就是一类避免该问题的方法。

有限内存的拟牛顿方法

基于有限内存的方法主要解决的问题是，1）Hessian矩阵存储较大或者计算困难。2）二是近似的Hessian是稠密的。
有限内存方法通过维护有限个向量来隐式表示Hessian矩阵，通过该方法至少线性收敛。
该类方法比较类似，本节主要介绍LBFGS算法，即在BFGS上进行有限内存限制。
L-BFGS通过最近m次的曲度信息来重建Hessian矩阵。

LBFGS推导

回顾一下BFGS算法思路

$$
\begin{aligned}
x_{k+1} &= x_k - \alpha_k H_k \nabla f_k \\
H_{k+1} &= v_k^T H_k v_k + \rho_k s_k s_k^T \\
\rho_k &= 1/y_k^T s_k, v_k = \mathbf{I} - \rho_k y_k s_k^T \\
s_k &= x_{k+1} - x_k, y_k = \nabla f_{k+1} - \nabla f_k
\end{aligned}
$$

L-BFGS 的思路是保留有限个m个 (s_i, y_i)对，重建Hessian矩阵而不是保存整个Hessian矩阵。
根据Hessian重构规则，计算第k步的Hessian矩阵只能通过前m个 (s_i, y_i)计算得到，即

$$
\begin{aligned}
H_k &= v_{k-1}^T H_{k-1} v_{k-1} + \rho_{k-1} s_{k-1} s_{k-1}^T \\
&= v_{k-1}^T (v_{k-2}^T H_{k-2} v_{k-2} + \rho_{k-2} s_{k-2} s_{k-2}^T) v_{k-1} + \rho_{k-1} s_{k-1} s_{k-1}^T \\
&= v_{k-1}^T v_{k-2}^T H_{k-2} v_{k-2} v_{k-1} + v_{k-1}^T \rho_{k-2} s_{k-2} s_{k-2}^T v_{k-1} + \rho_{k-1} s_{k-1} s_{k-1}^T \\
&= \dots \\
&= v_{k-1}^T v_{k-2} \dots v_{k-m}^T H_k^0 v_{k-m} \dots v_{k-2} v_{k-1} \\
&+ \rho_{k-m} (v_{k-1}^T \dots v_{k-m+1}{}^T) s_{k-m} s_{k-m}^T (v_{k-m+1} \dots v_{k-1}) \\
&+ \dots \\
&+ \rho_{k-1} s_{k-1} s_{k-1}^T
\end{aligned}
$$

通过该分解可以快速计算得到 $H_k \nabla f_k$，算法为：

Algorithm 7.4 (L-BFGS two-loop recursion).

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

上述算法有三个要点，可证明通过该算法可以得到 $H_k\nabla f_k$，分别说明如下

1) 参数q推导如下： $q_k = \nabla f_k$，则

$$\begin{aligned} q_i &= q_{i+1} - \alpha_i y_i \\ &= q_i - (\rho_i s_i^T q_{k+1}) y_i \\ &= q_{i+1} - (\rho_i y_i s_{i+1}^T q_{k+1}) \\ &= (\mathbf{I} - \rho_i y_i s_i^T) q_{i+1} \\ &= v_i q_{i+1} \\ &= \dots \\ &= v_i v_{i+1} v_{i+2} \dots v_{k-1} q_k \end{aligned}$$

2) 由于

$$\begin{aligned} \alpha_i &= \rho_i s_i^T q_i \\ &= \rho_i s_i^T v_i v_{i+1} v_{i+2} \dots v_{k-1} q_k \end{aligned}$$

3）初始化r时，此时 $r_{k-m-1} = H_k^0 q_{k-m} = v_{k-m} v_{k-m+1} v_{k-m+2} \dots v_{k-1} q_k$

4)推导r,

$$\begin{aligned} r_i &= r_{i-1} + s_i(\alpha_i - \beta) \\ &= r_{i-1} + s_i(\alpha_i - \rho_i y_i^T r_{i-1}) \\ &= s_i\alpha_i + r_{i-1} - s_i\rho_i y_i^T r_{i-1} \\ &= s_i\alpha_i + v_i^T r_{i-1} \\ &= s_i\alpha_i + v_i^T(s_{i-1}\alpha_{i-1} + v_{i-1}^T r_{i-2}) \\ &= s_i\alpha_i + v_i^T s_{i-1}\alpha_{i-1} + v_i^T v_{i-1}^T s_{i-2}\alpha_{i-2} \\ &+ \dots + v_i^T v_{i-1}^T \dots v_{k-m}^T s_{k-m}\alpha_{k-m} \\ &+ v_i^T v_{i_1}^T \dots v_{k-m}^T r_{k-m-1} \end{aligned}$$

5)计算 $r_{k-1} = H_k\nabla_k$并且代入相关参数。

6）该算法时间复杂度大约为4mn。

7）变量 H_k^0和计算没有关系，并且每一步的选择都不同，一般选择为 $H_k^0 = \gamma_k\mathbf{I}$， $\gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}$，该选择在实践中比较有效。

LBFGS算法

综上可以得到LBFGS算法

Algorithm 7.5 (L-BFGS).
Choose starting point x_0, integer $m > 0$;
$k \leftarrow 0$;
repeat
  Choose H_k^0 (for example, by using (7.20));
  Compute $p_k \leftarrow -H_k \nabla f_k$ from Algorithm 7.4;
  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.

在实际中LBFGS表现较好，主要缺点是当实际的Hessian矩阵的特征向量分布较宽时，即条件数比较大时，收敛速度变慢。
另外LBFGS中应用到的思路可以应用到其他拟牛顿方法中。LBFGS主要用到的原理是矩阵可以近似为低秩空间中表示。

稀疏拟牛顿方法

实际需求要保持近似的Hessian矩阵和实际的保持相同的稀疏性，期望用较少的存储得到更好的解和收敛性。
前提是通过知道Hessian中不为0的点，即为$\Omega = (i,j)|H(i,j) \neq 0$
建模思路为

$$min||B - B_k||^2 = \sum_{(i,j)\in\Omega} (B(i,j) - B_k(i,j))^2$$
$$s.t.\ Bs_k = y_k, B = B^T, B(i,j) = 0, (i,j) \neq \Omega$$

实际效果不太好

总结

通过该小结的学习，需要了解
1）大规模无约束最优化需要解决那些问题
2）非精确牛顿方法求解思路
3）LBFGS求解和推导