Approximate Newton Methods

1 Model fitting and the Gauss-Newton algorithm

A frequently arising optimization problem is that of fitting a parametric model to empirical data. To keep consistent notation, we denote the model parameters (optimization variables) as $\mathbf{x} \in \mathbb{R}^n$, and assume that for every i-th datapoint we can measure some error $e_i(\mathbf{x})$ (the function $e : \mathbb{R}^n \to \mathbb{R}$ incorporates the actual model being used). Accepting a quadratic error criterion, we end up with the following nonlinear *least squares* problem

$$\min_{\mathbf{x}} \frac{1}{2} \sum_{i=1}^{m} e_i^2(\mathbf{x})$$

where m is the number of measurements.

In order to use Newton's method, we need the gradient and the Hessian of the objective, which we will henceforth denote by $f(\mathbf{x})$. Simple chain rule suggests

$$\begin{aligned} \nabla f(\mathbf{x}) &= \sum_i e_i(\mathbf{x}) \nabla e_i(\mathbf{x}) \\ \nabla^2 f(\mathbf{x}) &= \sum_i \nabla e_i(\mathbf{x}) \nabla^{\mathrm{T}} e_i(\mathbf{x}) + e_i(\mathbf{x}) \nabla^2 e_i(\mathbf{x}). \end{aligned}$$

Denoting by $\mathbf{e}(\mathbf{x}) = (e_1(\mathbf{x}), \dots, e_m(\mathbf{x}))$ the vector of errors, and by

$$\mathbf{G}(\mathbf{x}) = \nabla \mathbf{e}(\mathbf{x}) = (\nabla e_1(\mathbf{x}), \dots, \nabla e_m(\mathbf{x}))$$

its Jacobian matrix, we can write the gradient in matrix form as

$$\nabla f(\mathbf{x}) = \mathbf{G}(\mathbf{x}) \mathbf{e}(\mathbf{x}).$$

Similarly, we can express the Hessian as

$$\nabla^2 f(\mathbf{x}) = \mathbf{G}(\mathbf{x}) \mathbf{G}^{\mathrm{T}}(\mathbf{x}) + \sum_i e_i(\mathbf{x}) \nabla^2 e_i(\mathbf{x}).$$

Note that close to the solution, the error terms are (presumably) small, therefore, the second term dominated by $e_i(\mathbf{x})$ is close to zero. This allows to approximate the Hessian as the outer product of the gradients,

$$\nabla^2 f(\mathbf{x}) \approx \tilde{\mathbf{H}} = \mathbf{G}(\mathbf{x}) \mathbf{G}^{\mathrm{T}}(\mathbf{x}).$$

Newton's method in which $\tilde{\mathbf{H}}$ is used instead of the exact Hessian is know as the *Gauss-Newton* algorithm. The approximation of the Hessian by outer products ensures its positive-semidefiniteness. To guarantee $\tilde{\mathbf{H}} \succ 0$, the Hessian is modified by adding a diagonal matrix **Delta**, which can be computed using modified Cholesky factorization.

The particular choice of $\mathbf{\Delta} = \epsilon\mathbf{I}$ leads to the so-called *Levenberg* algorithm. Another popular choice is $\mathbf{\Delta} = \epsilon \, \mathrm{diag}(\tilde{\mathbf{H}})$, which is called the *Levenberg-Marquardt* algorithm.

2 Quasi-Newton algorithms

The inversion of the Hessian requiring $O(n^3)$ operations becomes the computational bottleneck of Newton methods (including the variants of Gauss-Newton algorithm) even for moderately-sized problems. The family of *quasi-Newton* algorithms avoids this complexity by gradually building an approximation not the Hessian itself, but to its inverse.

A general quasi-Newton algorithm starts with some initial guess of the inverse of the Hessian $\mathbf{B}_0$ (if nothing is known about the Hessian, $\mathbf{B}_0 = \mathbf{I}$ is the standard choice), and proceeds like the Newton method, replacing $\mathbf{H}^{-1}$ in the computation of the direction by the current estimate $\mathbf{B}_k$. Then, a step size is found using exact or inexact linesearch and the current point is updated, exactly as in the Newton algorithm. Once a new point and a new gradient are computed, the change in the point and the gradient are used to update the approximation of the inverse of the Hessian using an update rule.

The following generic iterative procedure describes the family of quasi-Newton algorithms; different instances differ in the specific update rule.

```
input  : function $f$; initial point $\mathbf{x}_0$
output: (approximate) local minimizer $\mathbf{x}^*$ of $f$
Start with an initial guess of the inverse of the Hessian $\mathbf{B}_0$
for $k = 1, 2, \ldots,$ until convergence do
    Compute descent direction $\mathbf{d}_k = -\mathbf{B}_{k-1}\nabla f(\mathbf{x}_{k-1})$
    Find step size $\alpha_k$ ensuring that $f'_{\mathbf{d}_k}(\mathbf{x}_{k-1}) < f'_{\mathbf{d}_k}(\mathbf{x}_{k-1} + \alpha_k \mathbf{d}_k)$.
    Update current point $\mathbf{x}_k \leftarrow \mathbf{x}_{k-1} + \alpha_k \mathbf{d}_k$
    Update inverse Hessian approximation
    $\mathbf{B}_k = \mathrm{Update}(\mathbf{B}_{k-1}, \mathbf{x}_k - \mathbf{x}_{k-1}, \nabla f(\mathbf{x}_k) - \nabla f(\mathbf{x}_{k-1}))$
end
Return $\mathbf{x}^* = \mathbf{x}^k$
```

Algorithm 1: Generic quasi-Newton method

Let us now have a closer look at the construction of the inverse Hessian approximation. To simplify notation, we denote $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$ and $\mathbf{H}_k = \nabla^2 f(\mathbf{x}_k)$.. Intuition suggests that the way the gradient changes as the result of a change in $\mathbf{x}$ tells information about the Hessian. This is captured by the definition of the differential of the gradient,

$$\mathbf{dg} = \mathbf{Hdx}.$$

In our case, we deal with finite differences, which we denote for simplicity as $\mathbf{p}_k = \mathbf{x}_k - \mathbf{x}_{k-1}$, and $\mathbf{q}_k = \mathbf{g}_k - \mathbf{g}_{k-1}$. Requiring the latter equality to hold for $\mathbf{p}$ replacing $\mathbf{dx}$ and $\mathbf{q}$ replacing

dg, we have

$$\mathbf{q}_k = \mathbf{H}_k \mathbf{p}_k,$$

which is usually called the *secant equation* (geometrically, a secant is a finite-difference approximation to the gradient).

Since we are interested in the inverse of the Hessian, we can write

$$\mathbf{p}_k = \mathbf{B}_k \mathbf{q}_k.$$

Our goal is to update the previous approximation of the inverse of the Hessian, $\mathbf{B}_{k-1}$ in such a way that the new approximation $\mathbf{B}_k$ satisfies the secant equation. One of the simplest forms of update is a rank-1 update of the form

$$\mathbf{B}_k = \mathbf{B}_{k-1} + \mathbf{u}\mathbf{v}^{\mathrm{T}},$$

where $\mathbf{u}$ and $\mathbf{v}$ are some vectors. The outer product $\mathbf{u}\mathbf{v}^{\mathrm{T}}$ is guaranteed to have rank 1 unless one of the vectors is zero.

Substituting the latter update to the secant equation, we have

$$(\mathbf{B}_{k-1} + \mathbf{u}\mathbf{v}^{\mathrm{T}})\mathbf{q}_k = \mathbf{p}_k,$$

from where

$$\mathbf{u} = \frac{\mathbf{p}_k - \mathbf{B}_{k-1}\mathbf{q}_k}{\mathbf{v}^{\mathrm{T}}\mathbf{q}_k}.$$

The second vector $\mathbf{v}$ can be any vector non-orthogonal to $\mathbf{q}_k$. In order to preserve the symmetry of the inverse Hessian approximation, we will further require $\mathbf{v} \propto \mathbf{u}$. For example, we can select $\mathbf{v} = \mathbf{p}_k - \mathbf{B}_{k-1}\mathbf{q}_k$, leading to the update scheme

$$\mathbf{B}_k = \mathbf{B}_{k-1} + \frac{(\mathbf{p}_k - \mathbf{B}_{k-1}\mathbf{q}_k)(\mathbf{p}_k - \mathbf{B}_{k-1}\mathbf{q}_k)^{\mathrm{T}}}{(\mathbf{p}_k - \mathbf{B}_{k-1}\mathbf{q}_k)^{\mathrm{T}}\mathbf{q}_k}.$$

While this scheme is a valid quasi-Newton update, it is not ideal. One of the disadvantages is that it does not guarantee positive definiteness of the inverse Hessian approximation. A more elaborate family of schemes, known as the *Broyden family* uses a rank 2 update of the form

$$\mathbf{B}_k = \mathbf{B}_{k-1} + \frac{\mathbf{p}_k\mathbf{p}_k^{\mathrm{T}}}{\mu_k} - \frac{\mathbf{s}_k\mathbf{s}_k^{\mathrm{T}}}{\tau_k} + \xi_k\tau_k\mathbf{v}_k\mathbf{v}_k^{\mathrm{T}},$$

where $\mathbf{s}_k = \mathbf{B}_{k-1}\mathbf{q}_k$, $\tau_k = \mathbf{s}_k^{\mathrm{T}}\mathbf{q}_k$, $\mu_k = \mathbf{p}_k^{\mathrm{T}}\mathbf{q}_k$, and

$$\mathbf{v}_k = \frac{\mathbf{p}_k}{\mu_k} - \frac{\mathbf{s}_k}{\tau_k}.$$

The choice of $\xi_k = 1$ leads to the very popular *BFGS* (Broyden-Fletcher-Goldfarb-Shanno) quasi-Newton algorithm; the choice of $\xi_k = 0$ is called *DFP* (Davidon-Fletcher-Powell). Both formulas lead to the smallest update ($\mathbf{B}_k$ closest to $\mathbf{B}_{k-1}$) guaranteeing symmetric positive-definite approximation of the inverse of the Hessian. The latter requires a small technical condition: the step size α_k has to be selected to increase the directional derivative, $f'_{\mathbf{d}_k}(\mathbf{x}_{k-1}) < f'_{\mathbf{d}_k}(\mathbf{x}_{k-1} + \alpha_k \mathbf{d}_k)$.

BFGS is a natural choice for medium-scale problems, as it requires only $O(n^2)$ computations at each iteration as opposed to $O(n^3)$ required to solve the exact Newton system. For convex quadratic functions, the algorithm converges to the minimizer in n iterations, and $\mathbf{B}_n$ coincides with the inverse of the Hessian at the solution point. With the initialization $\mathbf{B}_0 = \mathbf{I}$, the trajectory $\mathbf{x}_k$ produced by BFGS coincides exactly with the one produced by another popular steepest descent algorithm called *conjugate gradients* (but, unlike the latter, BFGS works well with inexact line search). However, for general problems, BFGS is known to converge in fewer iterations. BFGS has super-linear asymptotic convergence rate.

While quasi-Newton schemes (and BFGS in particular) alleviate the requirement to solve the Newton system, they still require to store the $n \times n$ inverse Hessian approximation and multiply by it ($O(n^2)$ computation and storage complexity). However, note that when $\mathbf{B}$ is rank 1, it can be stored in the form of $\mathbf{u}\mathbf{u}^{\mathrm{T}}$ requiring only $O(n)$ storage. Furthermore, the multiplication of $\mathbf{B}$ by a vector $\mathbf{g}$ requires only $2n$ arithmetic operations. The same is roughly true at the first L iterations, when $\mathbf{B}_L$ can be written as

$$\mathbf{B}_L = \mathbf{I} + \mathbf{u}_1\mathbf{u}_1^{\mathrm{T}} + \cdots + \mathbf{u}_L\mathbf{u}_L^{\mathrm{T}}$$

assuming rank 1 updates (or a conceptually similar, yet more elaborate form for the rank 2 updates of BFGS). Keeping the last L updates in the form of the vectors $\mathbf{u}_k$ without actually constructing the matrix is the core idea of the family of *limited-memory BFGS* schemes that requires only $O(Ln)$ storage and computation complexity. Typically, the size of the history kept by the algorithm is small ($L \approx 10$), which allows it to scale linearly to hundreds of thousands or millions of variables.

3 Truncated Newton

Another (though less common) way of obtaining a scalable approximate version of Newton's method consists of solving the Newton system approximately. Remember that at each Newton iteration, we have to solve

$$\mathbf{H}\mathbf{d} = -\mathbf{g}$$

(we drop the iteration indices for simplicity). The system can be written alternatively as the minimization of the following quadratic convex function

$$\min_{\mathbf{d}} \|\mathbf{H}\mathbf{d} + \mathbf{g}\|^2$$

over all directions $\mathbf{d}$. The optimization can be performed by any large-scale iterative algorithm (typically, conjugate gradients). However, since the solution usually takes an unaffordable amount of time, the iterative algorithm is stopped after a certain precision has been reached. Such approximate Newton algorithms consisting of outer iterations resembling the exact Newton's method, with inner iterations used to approximately solve the Newton system are typically called *truncated Newton algorithms*.