CS281 Section 2: Practical Optimization

Dougal Maclaurin

Since many parameter estimation problems in machine learning cannot be solved in closed form we often have to resort to numerical optimization. In this section we will describe some of the common optimization techniques used in machine learning. Even if you use canned routines it's helpful to understand what's going on under the hood. Reference: Numerical Recipes (Press, Teukolsky, Vetterling and Flannery) chapter 10.

1. Gradient Descent

   How would we minimize some function $f(\mathbf{x})$ given access to queries of both $f(\mathbf{x})$ itself and its gradient $\nabla f(\mathbf{x})$? If we start at some $\mathbf{x} = \mathbf{x}_0$, an obvious strategy is just to go downhill:

   $$\mathbf{x}_{i+1} = \mathbf{x}_i - \eta \nabla f(\mathbf{x_i}) \tag{1}$$

   This can actually be quite effective but it has some obvious problems. First, the parameter η, known as the "learning rate" is fairly arbitrary. People often actually use an η that changes with iteration number, so that the initial steps are large and they become smaller over time. An alternative is not to use a fixed η at all, but to perform an explicit line search in the direction of the gradient $\nabla f(\mathbf{x})$. In high dimensions, it's really the gradient that's doing most of the work whereas one-dimensional optimization is usually very easy.

   In high dimensions there is another deep pathology that is particularly bad when the problem is *ill-conditioned* (the condition number of a matrix is the ratio of its highest to its lowest eigenvalues. Matrices with large condition number are known as 'ill-conditioned').

   If we look at the example given, we see that stepping in the direction of the gradient can lead to a zig-zag pattern, known as 'Hamming stitches'. So even when the function is a pure quadratic, convergence can be very slow.

2. Newton's method

   If you don't want to use an arbitrary η and you don't want to perform a line search, you can use Newtonian methods. The idea is to approximate $f(\mathbf{x})$ as a quadratic form and to jump straight to the minimum. For this, we need second derivative information. In one dimension, this is easy:

   $$f(x) \approx f(x_0) + (x - x_0) f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0) \tag{2}$$

   So the Newtonian update rule just becomes:

   $$x_{i+1} = x_i - f'(x_i)/f''(x_i) \tag{3}$$

Be cautioned that, naively implemented, the method will fail catastrophically if the function is concave at some x_i.

Note that this is equivalent to the Newton-Raphson rule for finding the roots of a function. Finding a local minimum of a function is (almost) the same as finding the roots of its first derivative.

What about higher dimensions, where $x \in \mathcal{R}^D$? Now the second derivative is known as the Hessian, A, which is a matrix of size D by D:

$$A_{ij} = \frac{\partial}{\partial x_i}\frac{\partial}{\partial x_j} f(\mathbf{x}) \tag{4}$$

The objective function's Taylor expansion is:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\nabla f(\mathbf{x}_0))^T(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T A(\mathbf{x} - \mathbf{x}_0) \tag{5}$$

So the Newton update rule is:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - A^{-1} \cdot \nabla f(\mathbf{x}_0) \tag{6}$$

This is fine, and it solves the problem of Hamming stitches. The problem is that if D is very large, it can become unfeasible to even compute and store A ($O(D^2)$ time and space) let alone invert it ($O(D^3)$ time). Additionally, if $f(\mathbf{x})$ is not convex everywhere, then A may not be positive definite, and the inversion may be either impossible, or give bad results. These problems are circumvented using techniques such as nonlinear conjugate gradient and the L-BFGS method.

3. Conjugate gradients

   Imagine minimizing a poorly conditioned quadratic function like that shown in the Hamming stitches example:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x} \tag{7}$$

   Note that this is equivalent to solving the matrix equation

$$A\mathbf{x} = \mathbf{b} \tag{8}$$

   Imagine what would happen if you rescaled things. If you squished the space so that the elliptical contours become circles, the Hessian is the identity, and things become very easy. You wouldn't even need to use gradient information - just minimize the function along each axis separately. Unfortunately the operation to do the squishing is to multiply by $A^{1/2}$ which has the same asymptotic complexity as just doing the inversion.

   The insight of the conjugate gradient method is to note that it's possible to construct a set of directions, $\mathbf{x}_i$, that *would become* orthogonal to each other if the space were squished. Such directions are known as conjugate directions, and they satisfy $\langle \mathbf{x}_i, \mathbf{x}_j \rangle_A = 0$ where $\langle \cdot, \cdot \rangle_A$ is an inner product, defined as:

$$\langle \mathbf{a}, \mathbf{b} \rangle_A = \mathbf{a}^T A\mathbf{b} = (A^{1/2}\mathbf{a})^T(A^{1/2}\mathbf{b}) \tag{9}$$

So if we can find a set of conjugate directions, then minimizing the function is easy - just minimize along each conjugate direction in turn and you're guaranteed to get there after D steps. Now it's easy to see how you would make a minimization algorithm based on this insight. At each iteration, instead of searching in the direction of the gradient itself, search in the direction of the *conjugated* gradient. That is, the projection of the gradient onto the subspace *orthogonal* to the space spanned by the previous search directions. This can be done with a procedure like Gram-Schmidt orthogonalization. As it turns out, however, you only need to orthogonalize with respect to the most recent search direction. The other relevant inner products are magically zero. To see this, consider first the conjugate gradient method naively implemented. $\mathbf{p}_i$ are the search directions and $\mathbf{r}_i$ are the gradients.

$$\mathbf{p}_0 = \mathbf{r}_0 = -\nabla f(\mathbf{x}_0) \tag{10}$$

$$\mathbf{r}_i = \nabla f(\mathbf{x}_i) = \mathbf{b} - A\mathbf{x}_i = \mathbf{r}_{i-1} - \alpha_{i-1} A\mathbf{p}_{i-1} \tag{11}$$

$$\mathbf{p}_i = \mathbf{r}_i - \sum_{j<i} \frac{\mathbf{r}_i^T A\mathbf{p}_j}{\mathbf{p}_j^T A\mathbf{p}_j}\mathbf{p}_j \quad \text{(Gram-Schmidt orthogonalization)} \tag{12}$$

$$\alpha_i = \text{argmin}_\alpha \left[f(\mathbf{x}_i + \alpha\mathbf{p}_i)\right] = \frac{\mathbf{r}_i^T\mathbf{p}_i}{\mathbf{p}_i^T A\mathbf{p}_i} \tag{13}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i\mathbf{p}_i \tag{14}$$

$$\tag{15}$$

There are some convenient orthogonality conditions due to this construction. First, we have the following conjugacy due to the Gram-Schmidt procedure itself:

$$\langle \mathbf{p}_i, \mathbf{p}_j \rangle_A = 0, \quad i \neq j \tag{16}$$

Next, at each step, $\mathbf{x}_{i+1}$ is the minimizer of the quadratic form within the subspace $\mathcal{D}_i$, defined as:

$$\mathcal{D}_i = \text{span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, ..., A^i\mathbf{r}_0\} \tag{17}$$

$$= \text{span}\{\mathbf{r}_0, \mathbf{r}_1, ..., \mathbf{r}_i\} \tag{18}$$

$$= \text{span}\{\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_i\} \tag{19}$$

This means that $\mathbf{r}_i$, the gradient at $\mathbf{x}_i$, is perpendicular to $\mathcal{D}_i$. Since $\mathbf{r}_i \in \mathcal{D}_i$ and $\mathbf{p}_i \in \mathcal{D}_i$ by construction, we have the following orthogonality relations too:

$$\mathbf{r}_i^T\mathbf{r}_j = 0, \quad \mathbf{r}_i^T\mathbf{p}_j = 0, \quad \mathbf{r}_i^T A\mathbf{p}_{j-1} = 0, \quad j < i \tag{20}$$

This means that all the terms in the Gram-Schmidt procedure except for the last one vanish, and the conjugate gradient algorithm is modified as follows:

$$\beta = \frac{\mathbf{r}_i^T A\mathbf{p}_{i-1}}{\mathbf{p}_{i-1}^T A\mathbf{p}_{i-1}} = \frac{\mathbf{r}_i^T\mathbf{r}_i}{\mathbf{r}_{i-1}^T\mathbf{r}_{i-1}} \tag{21}$$

$$\mathbf{p}_i = \mathbf{r}_i - \beta\mathbf{p}_{i-1} \tag{22}$$

$$\tag{23}$$

4. Nonlinear conjugate gradient

   The above procedure is a nice way to approximately solve $A^{-1}\mathbf{b}$ for large matrices. But it can also be extended to general optimization, even when you don't have access to the Hessian. The key observation is that α is the only quantity that requires the Hessian. But you would find the same α if you just minimized the objective function along the search direction $\mathbf{p}_i$. The only change that needs to be made is a pragmatic one. Instead of β as given above, we like to use the following.

$$\beta = \frac{(\mathbf{r}_i - \mathbf{r}_{i-1})^T \mathbf{r}_i}{\mathbf{r}_{i-1}^T \mathbf{r}_{i-1}} \tag{24}$$

$$\tag{25}$$

   This is exactly the same as the previous β for the case of an exact quadratic form, but for general functions this allows the algorithm to recover as orthogonality is lost, resetting the search direction in line with the current gradient.