Newton Method

http://freemind.pluskid.org/machine-learning/newton-method

posted on Free Mind on June 9, 2014
generated with pandoc on December 3, 2015
category: Machine Learning

tags: Optimization, Julia, Algorithm

上一次我们讨论了具有 Q-线性收敛性的普通的 gradient descent 方法，今天我们要介绍一种收敛速度更快的算法：Newton Method（或者叫 Newton's Method）。

可能大家知道有两个算法同时叫做牛顿法，一个是用迭代法来求方程的根的方法，另一个是 optimization 里的牛顿法，实际上这两者是同一个方法，或者同一个思想。今天我们要讲的是后者，但是由于前者比较简单，并且有有趣的故事，所以我们还是从前者讲起。故事开始于雷神之锤 III 竞技场里的一段神秘的快速求平方根倒数的代码：

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;
    i  = 0x5f3759df - ( i >> 1 );
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );
//  y  = y * ( threehalfs - ( x2 * y * y ) );
    return y;
}
```

据称比直接计算要快了 4 倍。其中的两次迭代（第二步迭代被注释掉了）就是用的牛顿法来求解方程 $y = 1/\sqrt{x}$，也就是 $1/y^2 - x = 0$ 的根。牛顿法的思想其实很简单，给定一个初始点 y_0，使用在该点处的切线来近似函数，然后寻找切线的根作为一次迭代。比如对于这个例子，令 $f(y) = 1/y^2 - x$，给定初始点 y_0，在该点处的导数是 $-2/y_0^3$，由此可以得到该处的切线为 $h(y) = -2(y - y_0)/y_0^3 + f(y_0)$，求解 $h(y) = 0$ 得到

$$y = y_0 \left(\frac{3}{2} - \frac{1}{2} x y_0^2 \right)$$

正是代码中的迭代。当然代码的重点其实不在这里，而在 `0x5f3759df` 这个奇怪的 magic number，用于得到一个好的初始点。这个神奇的数字到底是谁发现的，根据 wikipedia 上的说法似乎至今还没有定论。xkcd 还为此画了一条漫画，讽刺说每次我们惊奇地发现工业界里不知道哪个无名人士写出了 `0x5f3759df` 之类的神奇数字，背后都有成千上万的其他无名人士我们无从知晓，说不定他们中的某一个人已经解决了 P=NP 的问题，但是那人却还在调某个自动打蛋器的代码所以我们至今仍无知晓。`:D`

回到我们今天的话题，从这段代码中我们可以看到两点：

1. 牛顿法收敛非常快，对于精度要求不是特别高的情况，比如上面的图形学相关的计算中，甚至只用了一次计算迭代。
2. 另一方面，初始值的选取非常重要，我们接下去将会看到，初始值选得不好有可能会直接导致算法不收敛。

轶事讲完之后，正式回到我们的 optimization 的话题。回忆一下上一次我们提到迭代优化算法许多都可以理解为通过不动点法寻找 $\nabla f(x) = 0$ 的根，如果我们套用刚才提到的牛顿法，可以在每一次迭代的时候对 $\nabla f(x)$ 进行一阶近似来寻找其零点。注意到对 gradient $\nabla f(x)$ 进行一阶近似其实就是对原函数 $f(x)$ 进行二阶近似，由此我们得到了 optimization 里的牛顿法。

具体来说，从初始点 x_0 开始迭代，每一次迭代，在当前位置对 f 进行二阶近似，在 x_k 点处进行泰勒展开并丢掉二次以上的项，得到：

$$f(x) \approx \tilde{f}(x) = f(x_k) + \langle \nabla f(x_k), x - x_k \rangle + \frac{1}{2}\langle x - x_k, H(x_k)(x - x_k)\rangle$$

$\tilde{f}(x)$ 是一个二次函数，直接令 $\nabla \tilde{f}(x) = 0$ 可以解得：

$$x = x_k - H(x_k)^{-1}\nabla f(x_k) \tag{1}$$

这就是牛顿法的基本迭代步骤。这实际上是上一次介绍的 general 迭代框架的一个特殊情况：当 $A = H(x_k)^{-1}$ 的时候。但是这个特殊的 A 的取法使得算法的收敛性有了质的飞越。具体来说，现在我们有了 Q-二次收敛性。一个数列 $\{s_i\}$ 满足 Q-二次收敛是指 $\lim_{i\to\infty} s_i = s^*$ 且

$$\lim_{i\to\infty} \frac{|s_{i+1} - s^*|}{|s_i - s^*|^2} = \delta < \infty$$

虽然听起来只有二次，但是实际上是非常非常快的，比如如下的数列

$$s_i = \left(\frac{1}{10}\right)^{2^i}, \quad i = 1, 2, \ldots$$

就是二次收敛的，$s_1 = 0.01, s_2 = 0.0001, s_3 = 0.00000001$，粗略地讲，如果一个数列按 Q-二次收敛，那么没一次迭代我们可以使得小数点后的精确位数翻一倍。当然牛顿法并不是万金油，首先需要注意的一点是，牛顿法并不能总是保证收敛，当然，在上一次我们分析 gradient descent 的时候也是做了许多假设，不过牛顿法在这方面更加娇贵：首先函数的性质要好，并且初始解必须在离最优解比较近的地方才能保证收敛，而这个邻域可以是多大则依赖于一堆通常无法计算或估计的常数。

实际上，如果 Hessian 在当前的 x_k 处是不可逆的，那么 (1) 本身就是没法计算的。如果 Hessian 是可逆的，并且是 (semi)definite 的话，则

$$\langle -H(x)^{-1}\nabla f(x), -\nabla f(x)\rangle \geq 0$$

也就是说牛顿方向和 gradient 的负方向呈锐角，是一个下降（或者非增）方向。但是如果 Hessian 不是 (semi)definite 的话，就无法保证这一点了。另外，所谓下降方向的意思只是说沿着该方向移动足够小的步长可以保证函数是下降（非增）的，但是这里使用固定步长 1 却不能保证这一点（但是步长 1 又是证明二次收敛性所需要的）。

因此有一种牛顿法的变种，叫做 Damped Newton Method，就是在计算出牛顿方向之后，再用 backtracking 的方式做一次 linesearch。Backtracking linesearch 比之前介绍的基于 Wolfe's Condition 的 linesearch 要简单，它有两个参数 $0 < \alpha < 0.5$ 和 $0 < \beta < 1$。简而言之 backtrack linesearch 从初始步长 1 开始，每一次迭代将步长乘以 β，直到满足如下条件为止：

$$f(x + t\Delta x) \leq f(x) + \alpha t\langle\nabla f(x), \Delta x\rangle$$

这里的 Δx 是 linesearch 的方向。我们从 [Boyd and Vandenberghe, 2004] 盗一张图来说明一下。

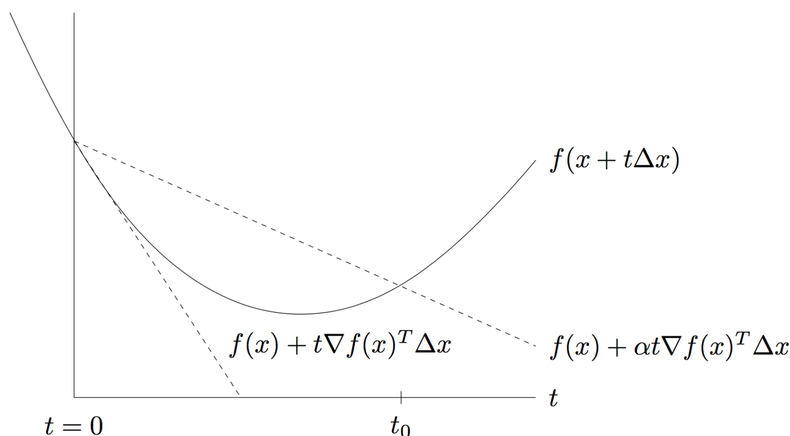

Figure 1: ....

由于 Δx 是下降方向，所以对于足够小的 t，我们有

$$f(x+t\Delta x) \approx f(x) + t\langle \nabla f(x), \Delta x\rangle < f(x) + \alpha t\langle \nabla f(x), \Delta x\rangle$$

所以 linesearch 可以保证停止，从图中来看，实际上就是寻找 $(\beta t_0, t_0]$ 中的一个步长，α 的大小决定了我们期望的下降量，比如极端情况 $\alpha = 0$ 则是任何非增的步长都会接受。以下是 julia 代码：

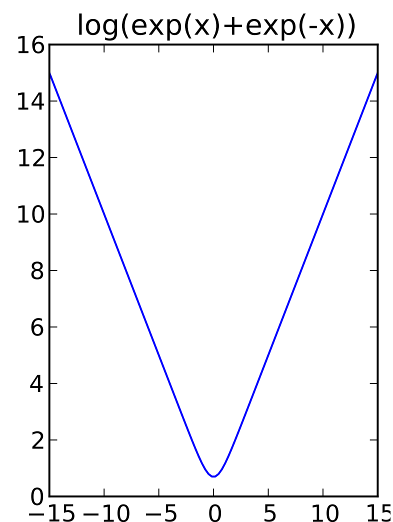

Figure 2: . . . .

```
function linesearch{T}(p::OptimizationProblem, x::Vector{T},
                       d::Vector{T})
    t = convert(T, 1)

    fx = objval(p, x)
    dfx = gradient(p, x)

    while objval(p, x + t*d) > fx + alpha * t * dot(dfx, d)
        t = beta * t
    end
    return t
end
```

增加了 linesearch 之后的 Damped Newton Method 有一个比较好的性质就是当迭代进入离最优解足够近的一个收敛区域之后，linesearch 一定会返回默认步长 1，从而进入 pure Newton Method 的 Q-二次收敛的阶段。具体细节和证明可以参考 [Boyd and Vandenberghe, 2004] 第 9 章的 Newton Method 一节。

这里我们给一个简单的一维情况下的例子，最小化 $f(x) = \log(e^x + e^{-x})$，这是一个 convex 函数，其最小值在 $x = 0$ 处达到。这里我们给出 Newton Method、Damped Newton Method 和普通的 Gradient Descent 的结果。

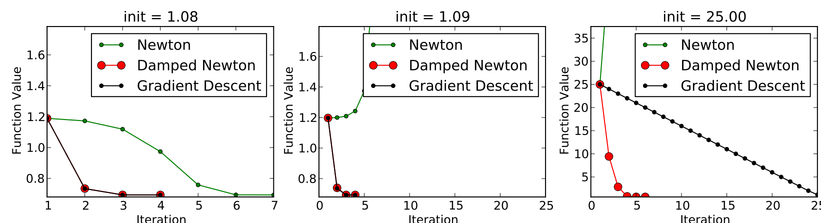

Figure 3: . . . .

大约在 $[-1.08, 1.08]$ 这个区间内是该函数的 Newton 收敛区间[1]，可以看到当我们选取初始值为 1.09 之后 Newton Method 直接不收敛了。但是 Damped Newton Method 则表现良好，并且在第三个例子中初始点离最优解比较远的情况，二次收敛的速度优势就体现出来了。

接下来我们可以再看一个更加实际一点的例子，也就是我们上一次介绍过的 Logistic Regression 的情况。为了将牛顿法用在 Logistic Regression

[1] 这里 Gradient Descent 比 Newton Method 要收敛快，并不是违反了收敛性分析，因为收敛性分析是一个粗略的 “阶” 的分析，一般需要在迭代步数很多的时候才能看出区别来，这里我们的初始点本身和最优值点已经非常接近了，各种算法都很容易就收敛到了最优点。

上，我们需要求目标函数的 Hessian，为了方便起见我再把目标函数搬过来：

$$L(w) = \sum_{i=1}^{N} \log\left(1 + \exp(-y_i w^T x_i)\right) + \frac{1}{2}\|w\|^2$$

上一次我们已经求过其 gradient 为

$$\nabla L(w) = \sum_{i=1}^{N} -\frac{\exp(-y_i w^T x_i)}{1 + \exp(-y_i w^T x_i)} y_i x_i + \lambda w$$

接下来很容易算出[2]其 Hessian 矩阵为

[2] 好吧，其实我很不好意思说我自己算了好久……对于像我这样向量二阶导搞得不熟悉的同学可以在求导的时候先针对向量的每一个元素单独求，也就是求出 Hessian 矩阵的每个元素 H_{ij}，将其形式写出来再找矩阵形式的式子。

$$H(w) = \sum_{i=1}^{N} \frac{x_i x_i^T}{\exp(y_i w^T x_i) + \exp(-y_i w^T x_i) + 2} + \lambda I$$

然后直接套用我们刚才说的 Damped Newton Method，目标函数在 USPS 数据集上的收敛结果如图所示，可以和上一次的 Gradient Descent + Wolfe Linesearch 进行对比。

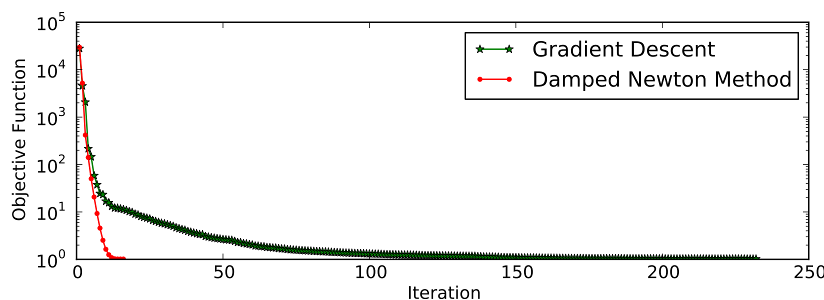

Figure 4: ……………………

可以看到 Newton Method 十来次迭代就收敛了。不过这里有一个细节需要注意，就是我们在上一次的例子中给出用 gradient descent 来优化 Logistic Regression 的目标函数的时候，直接丢掉了 regularizer ($\lambda = 0$)，而这里则必须要加上 regularizer，因为否者计算出来的 Hessian 会接近 singular，在求逆（或者说解线性方程组）计算 (1) 的时候直接爆炸掉了，没法收敛到一个好的结果。图中的结果对应取 $\lambda = 0.1$ 的情况。

实际上这里暴露出来的一个问题也确实是牛顿法在实际中经常碰到的一个问题：Hessian 矩阵 singular 或者是接近 singular 的情况。通常的做法是在对角线上加正数，在这里其实就对应于目标函数里的 regularizer $\lambda\|w\|^2$。上一次的 Gradient Descent 算法中我们也讨论过，加 regularizer 可以减小 condition number，从而改善收敛速度。但是需要注意的是并不是 regularizer 加得越厉害越好。因为加 regularizer 的原因不仅是出于优化来考虑，而且还有 learning theory 方面的 generalization performance 方面的考虑，从后者的角度来说 regularizer （系数）的大小是有一个合适

的值范围的，太大太小都不好（实际中一般通过 cross validation 来选取 regularizer 系数）。因为我们的最终目的是做 learning 和 prediction，如果一切只是为了优化方便，那么只要把目标函数变成一个 constant function 之类的，那么任何优化算法都可以一步收敛，但是却没有什么用处。

除了加 regularizer 之外，解决 Newton Method 的 near-singular Hessian 问题的方法还有其他很多，比如并不精确计算 Hessian，而是用一个性质良好的正定矩阵去近似估计当前的 Hessian，这一类方法通常被称为 Quasi-Newton Method，比如比较有名的 L-BFGS 算法，今天就不在这里细讲了。

不过 Quasi-Newton Method 的更重要的 motivation 其实是 Newton Method 的另一个严重的弱点：计算量。虽然 Newton Method 收敛快，只需要很少次的迭代，但是每一步迭代的计算量却比普通的 Gradient Descent 要大很多：Gradient Descent 只要计算一个 d 维 Gradient 向量即可，Newton Method 不仅要计算 Hessian 矩阵（d^2 维），而且要对该矩阵求逆（或者是解线性方程组）——对于通常的 dense 矩阵，这是一个复杂度为 $O(d^3)$ 的操作。

对于所谓的 big data 问题，Newton Method 通常难以应用。比如刚才的 Logistic Regression 的例子，如果数据点的个数非常多，由于计算 Hessian 每次都要遍历一遍所有数据并计算外积 $x_i x_i^T$，这会在每一次迭代都花掉很多时间。更严重的是，如果数据的维度非常高，且不说后面的求逆操作，即使 Hessian 矩阵本身的求解甚至存储都会出现问题。有时候碰到的情况是连 Newton Method 迭代一次的计算量都承担不起，所以在大规模机器学习或者 general 的大规模优化问题中，简单的一阶（只需要用到 gradient 信息）算法得到了更多的关注。以后我们还会看到，不仅是 Hessian，连 Gradient 向量的计算都非常 costly 的情况，以及对应提出的 Stochastic Gradient Descent 系的算法。

References

[Boyd and Vandenberghe, 2004] Boyd, S. P. and Vandenberghe, L. (2004). *Convex optimization.* Cambridge university press.