Newton's Method (and Friends)

CS 111

Winter 2023

RECALL: LOOKING FOR x^* WITH $\nabla f(x^*) = 0$.

Simple example: Quadratic in 1 dimension:

$$f(x) = \frac{a}{2}x^2 + bx + c \quad [\text{CONVEX: } a > 0!]$$

$$\nabla f = ax + b = 0 \quad \text{at} \quad x^* = \frac{-b}{a}$$

That was easy, but in general we have to ITERATE:

$$x_0 \rightarrow x_0 + \Delta x = x_1 \rightarrow x_2 \rightarrow \cdots$$

To avoid confusion, now we'll write elements of vectors like this:

$$x_0 = \begin{bmatrix} x_0(0) \\ x_0(1) \\ \vdots \\ x_0(n-1) \end{bmatrix} \quad \text{etc.}$$

TAYLOR'S THEOREM in one dimension:

If f is smooth enough,

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

strictly convex implies >0

This is min over choices of Δx when

$$0 = \frac{d}{d(\Delta x)} = f'(x_0) + f''(x_0)\cdot(\Delta x)$$

that is, when

$$\Delta x = -f'(x_0)/f''(x_0).$$

How about in n dimensions?

TAYLOR'S THEOREM IN N DIMENSIONS

If $f:\mathbb{R}^n \to \mathbb{R}$ is smooth enough,

$$f(x_0+\Delta x) \approx$$

$$f(x_0) + \underset{\substack{\uparrow \\ \Delta x \text{ is a vector!}}}{(\Delta x)^T} \underbrace{\nabla f(x_0)}_{\substack{\text{gradient} \\ \text{vector}}} + \frac{1}{2}(\Delta x)^T \underset{\substack{\uparrow \\ \text{Hessian matrix}}}{H} (\Delta x)$$

H is the "Hessian" matrix of second derivatives,

$$H(i,j) = \partial^2 f / \partial x(i) \, \partial x(j) \quad \left[\begin{array}{l}\text{evaluated} \\ \text{at } x=x_0\end{array}\right]$$

STRICTLY CONVEX $\Rightarrow$ H is SPD, $y^T H y > 0$ $\left\{\begin{array}{l}\text{for all} \\ \text{vectors} \\ y\end{array}\right.$

This is min. over choices of Δx when:

$$\nabla f + H(\Delta x) = 0$$

or $\quad \Delta x = -H^{-1}\nabla f$

(actually, solve $H(\Delta x) = -\nabla f$ for Δx)

NEWTON'S METHOD to find $x^* = \text{argmin}(f(x))$

START WITH A GUESS x_0

FOR $k = 0, 1, \ldots$:

$$\left[\begin{array}{l} \text{SOLVE } H(x_k)\cdot(\Delta x) = -\nabla f(x_k) \text{ for } \Delta x \\ x_{k+1} = x_k + \Delta x \end{array}\right.$$

sometimes put in a "stepsize" $s < 1$.

THEOREM

For x_0 close enough to x^*, convergence is quadratic:

$$\|x_{k+1} - x^*\| \le c \cdot \|x_k - x^*\|^2$$

Number of correct digits doubles each iteration

(But if you're not close enough, you might not converge at all.)

Examples: Rosenbrock function
Strang function
(see Jupyter notebook)

QUASI-NEWTON METHODS

Maybe you don't know the Hessian H, or maybe it's too slow to compute all those second derivatives.

Quasi-Newton methods (like BFGS) just compute the gradient ∇f (the first derivatives) at each step, and use the sequence of gradients to build up an approximation to H. Lots of details that we won't get into here!

Advantage: Less work per step than Newton

Disadvantage: Steps are not as good, so you need more steps to converge.

See Jupyter notebook for examples on Rosenbrock and Strang functions.

DIGRESSION: HOW DO YOU GET THE DERIVATIVES?

- **Analytic:**

  Strang: $f(x_0, x_1) = \frac{1}{2}(x_0^2 + \alpha x_1^2)$ we took $\alpha = \frac{1}{10}$

  $$\nabla f = \begin{bmatrix} x_0 \\ \alpha x_1 \end{bmatrix} \qquad H = \begin{bmatrix} 1 & 0 \\ 0 & \alpha \end{bmatrix}$$

  H is constant only because f is quadratic.

  But usually you don't know a formula for f ☹

- **Finite difference approximation:**

  $$f'(x) \sim \frac{f(x+h) - f(x-h)}{2h} \quad \text{as } h \to 0$$

  But floating-point subtraction loses accuracy. ☹

- **Automatic differentiation (AKA "Backpropagation"):**

  Transforms a <u>program</u> for $f(x)$ into a <u>program</u> for $f'(x)$, $\nabla f(x)$, $H(x)$.

  Amazing technology, no details here. ☺

  Many implementations, including AUTOGRAD in numpy.