Introduction to Gradient Descent

In a previous post, I presented (generalized) linear regression from a linear algebra point of view (namely, the “normal equations”) and in this post, I discuss yet another take on the problem, this time using gradient descent.

Gradient descent isn’t particularly fascinating for this particular task (as we know closed, analytical expressions for obtaining the parameters), but linear regression is the simplest example of gradient descent I could come up with without being completely trivial.

So we have a bunch of \( n \) points \( \{(x_i, y_i)\}_{i=1}^{n} \), and we want to fit a line of equation \( a + bx \) through them, minimizing some error function. Usually, we use the least mean square error (which corresponds to “vertical offsets”) because it usually yields easier analytical solutions. Let us define

\[
\hat{y}_i = \hat{y}(x_i) = a + bx_i,
\]

and the error function

\[
E(a, b) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2.
\]

Since \( E(a, b) \) is convex in both \( a \) and \( b \), we can solve
After quite a bit of algebra, we find

\[
\frac{\partial E(a, b)}{\partial a} = 0 \quad \text{and} \quad \frac{\partial E(a, b)}{\partial b} = 0
\]

for \( a \) and \( b \). Noticing that terms repeat a number of times, and possibly using special cases if the \( \{x_i\}_{i=1}^n = \{0, 1, 2, \ldots, n-1\} \), we can arrive at an efficient, linear-time, implementation.

While the relatively simple case of linear regression affords an analytical solution, we may (in real life) end up with another function 1) for which we can compute partial derivatives but 2) we cannot solve analytically for all its parameters, because, maybe, it is non-elementary (http://en.wikipedia.org/wiki/Elementary_function), or it poses some other difficulty. The technique to use then is gradient descent.

Gradient descent is a relatively simple procedure conceptually—while in practice it does have its share of gotchas. Let’s say we have some function \( f(x; \theta) \) with parameter(s) \( \theta \) which we want to minimize over the \( x \)s. The variable(s) to adjust is \( \theta \). If we want to minimize \( f(x; \theta) \), we must find \( \theta^* \) that minimizes given the \( x \)s which are fixed (or “observed” depending on how you look at it). If we can compute the partial derivative of \( f(x; \theta) \) with respect to \( \theta \), we know in which direction change \( \theta \) so that \( f(x; \theta) \) decreases. We then have the update rule

\[
\theta' = \theta - \eta \frac{\partial f(x; \theta)}{\partial \theta},
\]

which changes \( \theta \) in the direction in which \( f(x; \theta) \) decreases.

However, if we know in which direction we must change it, we do not necessarily know by how much exactly, and so a function-dependent scaling factor \( \eta \) is introduced. This scaling, the gradient step, will prevent us from updating \( \theta \) too far and miss the minimum. If \( \eta \) is too small it will take too long to find the minimum; if it is too large the procedure may diverge—essentially crash.

(The underlying assumption for gradient descent to work properly is that the function one wants to minimize is somewhat approximately convex, so that we follow the slope and find a (good local) minimum. If the function would highly non-convex, it would probably not work very well.)

If we have more than one parameter, then \( \theta \) is a vector of parameters, and the derivative is replaced by a gradient (http://en.wikipedia.org/wiki/Gradient), which is the vector formed by the partial derivatives in all the parameters. That is,
\[ \nabla f = \left( \frac{\partial f(x; \theta)}{\partial \theta_1}, \frac{\partial f(x; \theta)}{\partial \theta_2}, \ldots, \frac{\partial f(x; \theta)}{\partial \theta_k} \right). \]

if we have \( k \) parameters. It is therefore the general case where \( \theta \) is a vector that gives the method its name, gradient descent.

* * *

Back to the linear regression problem. Using \( E(a, b) \), we find that the partial derivative w/ respect to the two parameters \( a \) and \( b \) are

\[
\frac{\partial E(a, b)}{\partial a} = -2 \sum_{i=1}^{n} (y_i - a - bx_i)
\]

and

\[
\frac{\partial E(a, b)}{\partial b} = -2 \sum_{i=1}^{n} (y_i - a - bx_i)x_i
\]

and therefore

\[
\nabla E(a, b) = \left( \frac{\partial E(a, b)}{\partial a}, \frac{\partial E(a, b)}{\partial b} \right)
\]

and the update rule is

\[
(a, b)' = (a, b) - \eta \nabla E(a, b)
\]

However, for faster convergence, it is sometimes useful to use a different \( \eta \) for each parameter. We then have

\[
(a, b)' = (a, b) - \left( \eta_a \frac{\partial E(a, b)}{\partial a}, \eta_b \frac{\partial E(a, b)}{\partial b} \right)
\]

* * *

Let us have a look at a C++ implementation. First, as with the general regression, we see that the expressions of the gradient have quite a few invariants we can factor out. Using the identities

\[
\sum_{i=1}^{n} i = \frac{1}{2}n(n + 1)
\]

and

\[
\sum_{i=1}^{n} i^2 = \frac{1}{6}n(n + 1)(2n + 1),
\]

we arrive at a constant-time computation of gradient in the iteration step (but it does take linear time, once, to initialize the invariants). Here’s how the actual code looks like:
Where are the factors $2$ in front of the partial derivatives? Why are they gone? Think about it for a while, I’ll come back to it in a moment.

Let us try it out!

Let $a = 5$ and $b = 3$ (why not) and let us add some uniform random noise $\varepsilon \sim \mathcal{U}(-1,1)$. With $n = 100$, and not particularly efficient initial values for $a$ and $b$, we have the following convergence:
So we see the parameters adjusting smoothly (which will not always be the case; more complex functions may yield surprises) and that we arrive to very good estimates after \( \approx 250 \) iterations (which, in this case, are inexpensive as being constant-time). After \( \approx 475 \) iterations, the algorithm bails out as it has detected convergence.

We said earlier that \( \eta \) might be function-specific. In the case of linear regression by means of gradient descent, the following \( \eta \) seems to be working fine:

\[
\begin{array}{|c|c|c|}
\hline
n & \eta_a & \eta_b \\
\hline
100 & 1e-3 & 1e-6 \\
1000 & 1e-4 & 1e-9 \\
10000 & 1e-5 & 1e-12 \\
100000 & 1e-6 & 1e-15 \\
\hline
\end{array}
\]

So \( \eta_a \) and \( \eta_b \) are the reason why we can drop the \( \frac{1}{2} \) in front of the derivatives: the constant is absorbed into the \( \eta_a \) and \( \eta_b \) and so can be ignored.

While this seems to be quite laborious for linear regression, gradient descent proves most useful in all kind of optimization problems, including machine learning. In machine learning gradient descent (and back-propagation (http://en.wikipedia.org/wiki/Backpropagation)) is used to train neural networks of all kinds. Newton’s method for finding square roots (last week’s post (https://hbfs.wordpress.com/2012/04/17/new-way-of-computing-square-roots/)) can be thought of as gradient-descent type algorithm with an expression for \( \eta \) that depends explicitly on \( x \).

We will certainly discuss gradient-descent methods again future posts.

This entry was posted on Tuesday, April 24th, 2012 at 9:37 am and is filed under algorithms, machine learning, Mathematics. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.
5 Responses to *Introduction to Gradient Descent*

**Manish says:**
February 15, 2013 at 0:28 am
Can you please explain the same thing with a small example?

**Reply**

**Steven Pigeon** says:
February 15, 2013 at 12:39 pm
Ok. Let . If you want to find the minimum of , you can either solve analytically the equation (and find trivially that ) or use gradient descend, start at some reasonnable guess and use gradient descent to find . You will find

So you would iterate

starting at with reasonnable guess . Of course you can conlapulate into , and get .

Normally, you don’t get functions that are trivially analytically solvable. Usually, you get monsters with lots of internal parameters and non-separable expression, and you have to use gradient descent.

Let re-do the example with a “more realistic” example: —that’s still a basic quadratic, but now the minimum is not zero, it’s , found at . Applying gradient descent:

We start at some (possibly if you have *a priori* assumptions that should be small-ish), and we iterate:

Note that formally, we do not observe , , and . We can however estimate

for a small-ish . The choice of is also very important and related to the (local) curvature of the function. If you pick it too large, you oscillate around the solution (and if the function is not very well-behaved, it can even bring you further away from the solution), if you pick it too small, it takes forever to get to the solution.

With the preceeding example, with , , and $c=3$, with initial guess , the gradient descent produces the following path (in magenta):

(right-click and “view image” to view image to full size).

I hope that helped a bit.

**Reply**

有 趣的 机器学习: 最 简 明入 门 指 南 | zengine says:
[...] 这种找出最佳权重的办法被称为批量梯度下降，上面是对它的高度概括。如果想搞懂细节，不要害怕，继续深入下去吧。[...]

Reply
有趣的机器学习：最简明入门指南 | 我爱互联网 says:
May 11, 2014 at 22:36 pm
[...] 这种找出最佳权重的办法被称为批量梯度下降，上面是对它的高度概括。如果想搞懂细节，不要害怕，继续深入下去吧。[...]

Reply
Machine learning is fun! | Actuaires – Big Data says:
January 6, 2016 at 12:12 pm
[...] way to find the best weights for your function called batch gradient descent. Don’t be afraid to dig deeper if you are interested on learning the [...]

Reply

The Contempt Theme.