CS 4: Lecture 22
Wednesday, April 12, 2006

OPTIMIZATION

Let f(T, p) be a function that maps the temperature T of your car's engine and the percentage p of air in the air/gasoline mixture to the amount of carbon dioxide your car emits per mile of driving. Being a good citizen, you want to tune your engine to minimize your CO2 emission. This is called an _optimization_problem_.

Science and engineering have literally millions of optimization problems.

- Find the antenna length that maximizes signal reception.
- Find the driving speed that maximizes fuel efficiency.
- Find the doping (impurity) levels in silicon that maximize transistor speed.
- Choose the shape of an airplane wing that maximizes lift.
- Find the diet that meets a student's nutritional requirements for the fewest dollars.

Formally, many optimization problems look like this.

    Find the values x_1, x_2, ..., x_d that minimize f(x_1, x_2, ..., x_d).

Often you want to maximize a quantity, rather than minimize it, but maximizing f is the same as minimizing -f, so most optimization problems can be cast as minimization problems.

The variables x_1, x_2, ..., x_d are the _independent_variables_ or _parameters_ of the optimization problem. To simplify notation, we'll wrap them up into one vector x = (x_1, x_2, ..., x_d). This seems natural when x is the position of a radio tower in 3D space that optimizes the cell phone reception in Berkeley. It seems less natural when x_1 is temperature, x_2 is air/gasoline mixture, and x_3 is engine RPMs. Nevertheless, even when the variables are in entirely different units, we're still going to wrap them up into one vector, and treat it as a point in a d-dimensional _parameter_space_. That's why the number of parameters is denoted by d--it stands for "dimension."

The dependent variable f(x) is called the _objective_function_. Memorize this term. Virtually every optimization problem revolves around optimizing some objective function.

Local vs. Global Minima

Ideally, we would like to find the _global_minimum_ of f--that is, the value x such that

    f(x) <= f(y)   for _every_ vector y.

Unfortunately, objective functions often have local minima. A _local_minimum_ of f is a value x such that

    f(x) <= f(x + y)   for every y such that |y| < r for some positive r.

In other words, there is no better solution than x within some radius r of x.

Consider the function f(a) = 3 a^4 - 4 a^3 - 12 a^2.

```
                        *           ^ f(a)            *
                        *           |                 *
                         *          |                 *
                         *          |                 *
                          *         |                *
                           *        |                *
                       -+---*-+---**O**---+-----+----*+-> a
                             *****     **    global  *
                                         *  minimum *
                              ^           *     |   *
                              |            *    v  *
                              |             **    *
                        local minimum         ****
```

This function has two local minima, at a = -1 and a = 2. However, only a = 2 is a global minimum. The other local minimum isn't nearly as good.

As a general rule, finding local minima is easy. Finding global minima is sometimes easy, and sometimes very, very hard. Many practical problems require the minimum of a quadratic function of several variables. A quadratic function has at most one local minimum, which therefore must be the global minimum!

The reason it's sometimes very hard to find a global minimum is because it might be very far from the local minima you find. If your goal is to maximize your altitude, you might feel pretty good on top of Mount McKinley, but Mount Everest is on the other side of the planet.

Consider the function f(theta) = sin theta.

```
              **            ** f(theta)^  **            **            **
  *          *  *          *  *        | *  *          *  *          *
   *        *    *        *    *       |*    *        *    *        *
    *      *      *      *      *      *      *      *      *      *
  --*------*------*------*------*------O------*------*------*------*----> theta
    *      *      *      *      *      *      *      *      *      *
     *    *        *    *        *    *        *    *        *    *
      *  *          *  *          *  *          *  *          *  *
       **            **            **            **            **
```

This function has infinitely many local minima, all of which are global minima too. But suppose we add another sinusoidal curve with a different frequency, like f(theta) = sin theta + 0.1 cos (theta / pi).

```
                             * f(theta)^  **
               *            * *        | *  *           **             *
              * *          *   *       |*    *         *  *           *
  *          *   *        *     *      *      *       *    *         *
   *        *     *      *      *      *      *      *      *       *
  --*------*------*------*------*------O------*------*------*------*----> theta
    *      *      *      *       *    *        *     *      *      *
    *      *       *    *         *  *          *   *        *     *
     *    *         *  *           **            * *          *   *
      *  *           **                           *            * *
       **                                                       *
```

This curve has infinitely many local minima, but none of them is a global minimum! No matter which local minimum you pick, you can always find one that's even lower. (The local minima get arbitrarily close to -1.1, but none reaches it.)

In an optimization problem, what kind of minimum do we need? For a few problems, any local minimum will do. For a few problems, the global minimum is absolutely necessary. But for most problems, we just need a very good local minimum--preferably one that's almost as good as the global minimum. Your car might not be getting the absolute minimum carbon dioxide emissions possible, but if you're only off by a few percent, you won't get all stressed about it.

An important consideration in choosing an optimization algorithm is, how long does a function evaluation take? Sometimes they're fast, but in some applications, function evaluations take a long time. In an extreme case, a function evaluation might involve performing a scientific experiment, like actually measuring the carbon dioxide output of a car under certain conditions.

Random Search

Our first optimization algorithm is very simple.

```
- Pick an integer n signifying how long you're willing to wait for an answer.
- Repeat n times:
  o Choose a random vector x in the domain.
  o If f(x) is the smallest solution so far, remember x.
- Output the best x you found.
```

Random search is a lousy algorithm, but it's better than nothing. Its main fault is that it doesn't exploit the fact that a good value of x probably has an even better value of x nearby. When you find yourself halfway up Mount Everest, you should look around for the peak. Instead, the random search algorithm immediately jumps to another random location on earth.

A lesser fault is that random search may neglect some parts of the search space, because it just randomly neglects to check them.

Grid Search

The grid search algorithm addresses the lesser fault of the random search algorithm. The algorithm evaluates f at the vertices of a rectangular grid, and chooses the vertex with the least value of f. For example, in the domain at right, we evaluate f at the points marked 'o'.

```
+-------------------+
|                   |
| o o o o o o o o o |
|                   |
| o o o o o o o o o |
|                   |
| o o o o o o o o o |
|                   |
| o o o o o o o o o |
|                   |
+-------------------+
       domain
```

Grid search seems clever in a two-dimensional domain, but its weakness is "the curse of dimensionality": if you use, say, a thousand grid points along each axis, then you need 1,000^d grid points overall, where d is the dimension of the parameter space. That gets big if d is more than two! If you have six parameters and can afford a million function evaluations, that only gives you ten grid points in each direction. That's not usually enough to find a good minimum. (Random search suffers from the curse of dimensionality just as badly, though it's not as obvious as with a grid.)

As a general rule, grid search is most appropriate when function evaluations are cheap or the number of parameters is small. But it's only a good start; you should refine the result using an algorithm like grid walk or Nelder-Mead.

Grid Walk

The grid walk algorithm is like the grid search algorithm, except that you don't search the whole grid. Instead, you choose a starting point and "walk downhill" from there, evaluating f only at nearby locations.

For grid walk, you need to choose a starting point x (perhaps the center of the domain) and a step size s (perhaps 10% of the domain width). Then you evaluate the objective function f at the following locations.

```
    f(x_1, x_2, ..., x_n)
    f(x_1 + s, x_2, ..., x_n)
    f(x_1 - s, x_2, ..., x_n)
    f(x_1, x_2 + s, ..., x_n)
    f(x_1, x_2 - s, ..., x_n)
    ...
    f(x_1, x_2, ..., x_n + s)
    f(x_1, x_2, ..., x_n - s)
```

In other words, you consider taking a step along each of the parameter axes, in either the positive or negative directions. Then you walk to the best point (i.e. set x to be the point with the best objective function) and repeat.

Observe that grid walk breaks the curse of dimensionality. Instead of testing a number of points exponential in d, now you just check 2d points other than x. Thus grid walk can be a lot faster than the previous two algorithms.

What if x is better than those 2d points? You reduce the step size, perhaps by cutting s in half, and try again. If x is still best, reduce the step size again. This gives grid walk another big advantage over the previous algorithms: when it finds a good point, it tries to _refine_ the result by searching locally, in successively smaller steps, until it homes in on a local minimum.

In the following example, grid walk takes a step to the right, then reduces the step size, then takes a step down, then terminates.

```
    +---------+    +---------+    +---------+    +---------+
    |         |    |         |    |         |    |         |
    |   o     |    |     o   |    |         |    |         |
    |         |    |         |    |    o    |    |         |
    | o x o   | -> |   o x o | -> |   oxo   | -> |     o   | -> done
    |         |    |         |    |    o    |    |    oxo  |
    |   o     |    |     o   |    |         |    |     o   |
    |         |    |         |    |         |    |         |
    +---------+    +---------+    +---------+    +---------+
```

An important decision you need to make when you implement grid walk is when to stop. The usual rule is to stop when the step size is so small that you don't expect to make much further progress. In other words, choose an s* > 0 and stop when s < s*.

Note that you don't have to use the step size for each parameter. In fact, if the parameters are measured in different units--say, temperature, percentage air/gas mixture, and engine RPMs, then you need a different step size for each parameter. When you get stuck, you cut all the step sizes in half.

Grid search represents a huge advance in speed over the previous algorithms, especially in high-dimensional parameter spaces. But it has several faults of its own.

- It's happy to stop at any local minimum, even a "bad" one that's not close to the global minimum.
- It can prematurely shrink the step size, then take many, many, many steps to walk to a local minimum. You'll also have this problem if you choose too small a step size from the start.

The first problem can be lessened by combining two algorithms. The second problem can be lessened by a more complicated optimization algorithm called the Nelder-Mead simplex algorithm.