13 Optimization and Search

Once you've gathered your data, selected a representation to work with, chosen an architecture for functional approximation, specified an error metric, and expressed your prior beliefs about the model, then comes the essential step of choosing the best parameters. If they enter linearly, the best global values can be found in one step with a singular value decomposition, but as we saw in the last chapter coping with the curse of dimensionality usually requires parameters to be inside nonlinearities. This entails an iterative search starting from an initial guess. Such exploration is similar to the challenge faced by a mountaineer in picking a route up a demanding peak, but with two essential complications: the search might be in a 200-dimensional space instead of just 2D, and because of the cost of function evaluation you must do the equivalent of climbing while looking down at your feet, using only information available in a local neighborhood.

The need to search for parameters to make a function extremal occurs in many kinds of optimization. We already saw one nice way to do nonlinear search, the Levenberg–Marquardt method (Section 10.4.1). But this is far from the end of the story. Levenberg–Marquardt assumes that it is possible to calculate both the first and second derivatives of the function to be minimized, that the starting parameter values are near the desired extremum of the cost function, and that the function is reasonably smooth. In practice these assumptions often do not apply, and so in this chapter we will look at ways to relax them.

Throughout, we will assume that the goal is to find an acceptable solution rather than a provably optimal one. The latter is of course much more work, extra effort that is pointless if there are many solutions that are equally good, particularly if other sources of uncertainty such as measurement noise are larger than the differences among the solutions. If an optimal answer really is needed then all possible ones much be checked, although in many problems as the search proceeds it's possible to prune parts of the space that can be shown to be worse than the current best solution (the *Branch-and-Bound* algorithm, [Lawler & Wood, 1966]).

Perhaps because nature must solve these kinds of optimization problems so frequently, the algorithms in this chapter share a kind of natural familiarity missing in most numerical methods. We will start with blobs (for the downhill simplex search), add mass (momentum for avoiding local minima), then temperature (via simulated annealing), and finally reproduction (with genetic algorithms). The cost of this kind of intuition is rigor. While there is some supporting theory, the real justification for this collection of algorithms is their empirical performance. Consequently it's important to view them not as fixed received wisdom, but rather as a framework to guide further exploration.

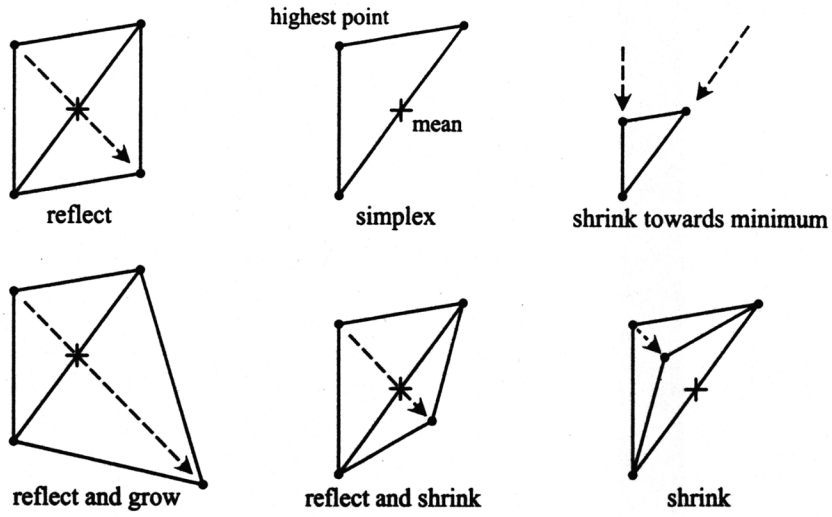

Figure 13.1. Downhill simplex update moves.

13.1 MULTIDIMENSIONAL SEARCH

As features are added to a model it quickly becomes inconvenient, if not impossible, to analytically calculate the partial derivatives with respect to the parameters (impossible because the function being searched might itself be the result of a numerical simulation). A simple-minded solution is to evaluate the function at a constellation of points around the current location and use a finite difference approximation to find the partials, but the number of function evaluations required at each step rapidly becomes prohibitive in a high-dimensional space. After each function evaluation there should be some kind of update of the search to make sure that the next function evaluation is as useful as possible. Each function evaluation requires choosing not only the most promising direction but also the length scale to be checked. In Levenberg–Marquardt we used the Hessian to set the scale, but doing that numerically would require still more function evaluations. The *Downhill Simplex* method, also called the *Nelder–Mead* method after its original authors [Nelder & Mead, 1965], is a delightful (although far from optimal) solution to these problems. Of all algorithms it arguably provides the most functionality for the least amount of code, along with the most entertainment.

A *simplex* is a geometrical figure that has one more vertex than dimension: a triangle in 2D, a tetrahedtron in 3D, and so forth. Nelder–Mead starts by choosing an initial simplex, for example by picking a random location and taking unit vectors for the edges, and evaluating the function being searched at the $D + 1$ vertices of the simplex. Then an iterative procedure attempts to improve the vertex with the highest value of the function at each step (assuming that the goal is minimization; for maximization the vertex with the smallest value is updated).

The moves are shown in Figure 13.1. Let $\vec{x}_i$ by the location of the ith vertex, ordered so that $f(\vec{x}_1) > f(\vec{x}_2) > \ldots > f(\vec{x}_{D+1})$. The first step is to calculate the center of the face

of the simplex defined by all of the vertices other than the one we're trying to improve:

$$\vec{x}_{mean} = \frac{1}{D}\sum_{i=2}^{D+1}\vec{x}_i \quad . \tag{13.1}$$

Since all of the other vertices have a better function value it's a reasonable guess that they give a good direction to move in. Therefore the next step is to reflect the point across the face:

$$\begin{aligned}\vec{x}_1 \rightarrow \vec{x}_1^{new} &= \vec{x}_{mean} + (\vec{x}_{mean} - \vec{x}_1) \\ &= 2\vec{x}_{mean} - \vec{x}_1 \quad .\end{aligned} \tag{13.2}$$

If $f(\vec{x}_1^{new}) < f(x_{D+1})$ the new point is now the best vertex in the simplex and the move is clearly a good one. Therefore it's worth checking to see if it's even better to double the size of the step:

$$\begin{aligned}\vec{x}_1 \rightarrow \vec{x}_1^{new} &= \vec{x}_{mean} + 2(\vec{x}_{mean} - \vec{x}_1) \\ &= 3\vec{x}_{mean} - 2\vec{x}_1 \quad .\end{aligned} \tag{13.3}$$

If growing like this gives a better function value than just reflecting we keep the move, otherwise we go back to the point found by reflecting alone. If growing does succeed it's possible to try moving the point further still in the same direction, but this isn't done because it would result in a long skinny simplex. For the simplex to be most effective its size in each direction should be appropriate for the scale of variation of the function, therefore after growing it's better to go back and improve the new worse point (the one that had been $\vec{x}_2$).

If after reflecting $\vec{x}_1^{new}$ is still the worst point it means that we overshot the minimum of the function. Therefore instead of reflecting and growing we can try reflecting and shrinking:

$$\begin{aligned}\vec{x}_1 \rightarrow \vec{x}_1^{new} &= \vec{x}_{mean} + \frac{1}{2}(\vec{x}_{mean} - \vec{x}_1) \\ &= \frac{3}{2}\vec{x}_{mean} - \frac{1}{2}\vec{x}_1 \quad .\end{aligned} \tag{13.4}$$

If $f(\vec{x}_1^{new}) < f(\vec{x}_2)$ we accept the move and try to improve $f(\vec{x}_2)$; if after reflecting and shrinking $f(\vec{x}_1^{new})$ is still worse we can try just shrinking:

$$\begin{aligned}\vec{x}_1 \rightarrow \vec{x}_1^{new} &= \vec{x}_{mean} - \frac{1}{2}(\vec{x}_{mean} - \vec{x}_1) \\ &= \frac{1}{2}(\vec{x}_{mean} + \vec{x}_1) \quad .\end{aligned} \tag{13.5}$$

If after shrinking $f(\vec{x}_1^{new})$ is still worse the conclusion is that the moves we're making are too big to find the minimum, so we give up and shrink all of the vertices towards the best one:

$$\begin{aligned}\vec{x}_i \rightarrow \vec{x}_i^{new} &= \vec{x}_i - \frac{1}{2}(\vec{x}_i - \vec{x}_{D+1}) \quad (i = 1, \ldots, D) \\ &= \frac{1}{2}(\vec{x}_i + \vec{x}_{D+1}) \quad .\end{aligned} \tag{13.6}$$

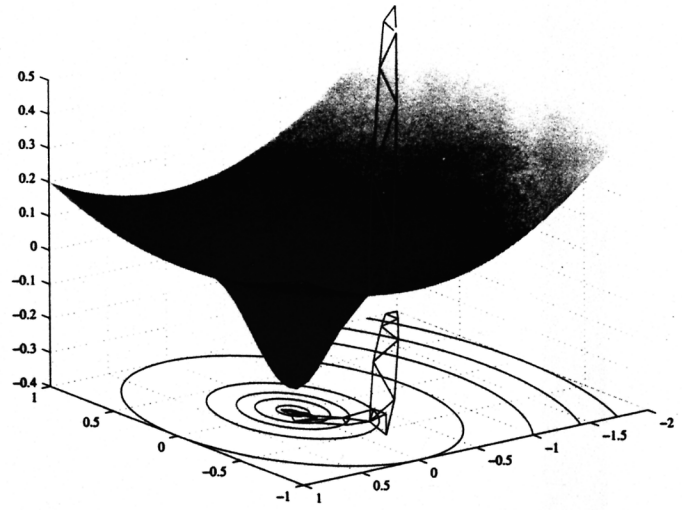

Figure 13.2. Downhill simplex function minimization. Above is the evolution of the simplex shown on the surface; below the 2D projection on a contour map.

Taken together these moves result in a surprisingly adept simplex. As it tumbles downhill it grows and shrinks to find the most advantageous length scale for searching in each direction, squeezing through narrow valleys where needed and racing down smooth regions where possible. When it reaches a minimum it will give up and shrink down around it, triggering a stopping decision when the values are no longer improving. The beauty of this algorithm is that other than the stopping criterion there are no adjustable algorithm parameters to set. Figure 13.2 shows what a simplex looks like searching for the minimum of a function.

A simplex search takes on a particularly simple form in 1D, where the simplex is just a pair of points. Each step takes the higher point and moves it around the lower point (in 1D this can be optimized by using triples of points with scale factors based on the golden mean [Press *et al.*, 1992]). A 1D search is useful in more than 1D, because it can be used to find the minimum of a function in a given direction, $\min_\alpha f(\vec{x}_0 + \alpha\hat{x})$. This is called *line minimization*.

Performing a series of line minimizations provides another way to do multi-dimensional search. A naive way to do this is to cycle through the axes of the space, minimizing along each direction in turn. This does not work well, because each succeeding minimization can influence the optimizations that came before it, leading to a large number of zigs and zags to follow the function along directions that might not be lined up with the axes of the space. *Powell's method* improves on this idea by updating the directions that are searched to try to find a set of directions that don't interfere with each other [Acton, 1990]. Starting from an initial guess $\vec{x}_0$, D line minimizations are performed, initially along each of the D axes of the space. The resulting point $\vec{x}_1$ defines a change vector, $\Delta\vec{x} = \vec{x}_1 - \vec{x}_0$. This is a good direction to keep for future minimizations, assuming that moving in that direction again is advantageous, which is easily checked.