More on optimization

A surprisingly large space of machine learning methods can be fitted to data using fairly simple variants of stochastic gradient descent. SGD doesn't usually find the optimal parameters of a cost function to many significant figures. Although convergence to an optimum often isn't the goal in machine learning settings; we often use early stopping.

However, it's possible that in applications you face in future, you'll want something more sophisticated. This note sketches some of the options as a starting point for further reading. As motivation, here are some things that simple stochastic gradient descent doesn't do:

1) Fit constrained problems (unless reparameterized by the user to be unconstrained).

2) Jump straight to the optimum if the cost functions has an analytic solutions. For example, if the cost function is a quadratic form. We'd also like to get close to the optimum if the cost function is close to one with a neat solution.

3) Accurately find a local optimum to many significant figures.

4) Fit some of the parameters to be exactly zero, as is optimal for sparse regularizers such as L1.

There are alternative optimizers that do better on some or all of these points.

1 EM

When fitting mixtures of Gaussians, we saw an instance of the EM algorithm, which is a bound-based algorithm for models with latent variables. If applied to a "mixture" of one Gaussian, EM finds the maximum likelihood parameters in one update. If a mixture of $K = 2$ Gaussians is fitted to two widely-separated groups of points, then once the parameters are close to optimal, the responsibilities are close to zero and one. In one more step, the algorithm (approximately) sets the Gaussian components to the maximum likelihood fits of each group. This rapid convergence in easy cases is one of the reasons EM is popular. Another reason is that EM is an easy way to fit a constrained problem; the updates automatically give valid parameters.

2 Newton's method

Newton's method uses both the current gradient of the cost function E,

$$\mathbf{g} = \nabla_{\mathbf{w}} E(\mathbf{w})\big|_{\mathbf{w}=\mathbf{w}^{(t)}}, \tag{1}$$

and the Hessian[1],

$$H = \left.\frac{\partial^2 E(\mathbf{w})}{\partial w_i w_j}\right|_{\mathbf{w}=\mathbf{w}^{(t)}}, \tag{2}$$

evaluated at the current parameters $\mathbf{w}^{(t)}$. The Newton update is:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - H^{-1}\mathbf{g}. \tag{3}$$

[1] Cross-reference: The Hessian, the matrix of second derivatives of a cost function E, is also used in the Laplace approximation, where the cost function is the negative log-posterior.

If the cost function is a quadratic with optimum $\mathbf{w}^*$ and Hessian H,

$$E(\mathbf{w}) = \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top H(\mathbf{w} - \mathbf{w}^*) + \text{constant}, \tag{4}$$

then one update takes the weights to the optimum:

$$\mathbf{g} = H(\mathbf{w}^{(t)} - \mathbf{w}^*) \tag{5}$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - H^{-1}H(\mathbf{w}^{(t)} - \mathbf{w}^*) = \mathbf{w}^*. \tag{6}$$

Informally, when the cost function is *nearly* quadratic, which Taylor series suggest it usually will be close enough to an optimum, convergence is fast.

Unlike gradient descent, we don't need to tell Newton's method whether we are maximizing or minimizing a function. If we apply Newton's method to a quadratic function we will hop straight to its minimum (if H is positive definite) or its maximum (if H is negative definite) automatically. A downside is that saddle-points are also attractors in Newton's method, and the optimizer could get stuck at one, just as it could converge to a local optimum.

Newton's method seems appealing, because the basic version has no free parameters. However, if the matrix H is poorly-conditioned, Newton's method can take large steps and diverge. The method can be made more robust, but is then also more complicated. The Hessian can be made better conditioned by adding a constant to the diagonal. Also a step-size parameter is often introduced, $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta H^{-1}\mathbf{g}$, where the step-size η must be set or adapted somehow.

Aside: I've outlined Newton's method for optimizing multiple parameters, because most machine learning models have many parameters. However, the simple one-dimensional version of Newton's method is also commonly-used in numerical computing. Sometimes when you evaluate a standard function in a numerical library, internally it finds the answer by solving an equation using Newton's method.

3 Lower-memory algorithms

Storing the entire Hessian matrix, H, is infeasible for moderately large neural networks.[2] There are *Hessian free* implementations of Newton's method and related algorithms, which can numerically estimate $H^{-1}\mathbf{g}$ or quantities like it, without ever computing the Hessian itself. There is also a large collection of other low-memory optimizers that find sensible update directions using only gradient evaluations. You don't need to know how they work for this course. Personally I like to know how things work, and have worked through the details of these algorithms at some point, but I can't honestly say I keep all the details in my head. As a user, I tend to treat them as "black-boxes".

In assignment 2, Matlab/Octave users used a variant of non-linear conjugate gradients, while Python users used a low-memory quasi-Newton method called L-BFGS. These methods are often good on small to medium problems, as they often converge quickly, and don't have tweak parameters to set. They're not the most popular methods for large neural networks, and large-scale problems. However, anecdotally I know people in industry who use them on large problems, and there is some literature supporting their wider use for neural networks[3].

4 Proximal methods, example method for L1 regularization

There is a large literature on methods for fitting models with L1 regularization (Murphy 13.3–13.4 discusses some of these). This section describes the basic version of one of them

[2] For example, with a million parameters, a $10^6 \times 10^6$ matrix of 64 bit floating point numbers needs 8 TB of RAM.

[3] "On optimization methods for deep learning", Le et al., ICML 2011.

because it's fairly simple[4], although before using a method in practice you would want to read about its refinements, or other methods, or use existing software.

Given the L1 cost function,

$$c(\mathbf{w}) = E(\mathbf{w}) + \lambda \sum_d |w_d| = E(\mathbf{w}) + \lambda \|\mathbf{w}\|_1. \tag{7}$$

we could try to upper-bound the complicated data-dependent term $E(\mathbf{w})$ with a simple quadratic function at the current parameters $\mathbf{w}^{(t)}$:

$$\tilde{E}(\mathbf{w}; \mathbf{w}^{(t)}) = E(\mathbf{w}^{(t)}) + \left[\nabla_{\mathbf{w}} E(\mathbf{w})|_{\mathbf{w}=\mathbf{w}^{(t)}}\right]^\top (\mathbf{w} - \mathbf{w}^{(t)}) + \frac{L}{2}(\mathbf{w} - \mathbf{w}^{(t)})^\top (\mathbf{w} - \mathbf{w}^{(t)}). \tag{8}$$

The first two terms are a local linear approximation to the training error. For the third term we have assumed there is a 'Lipschitz' constant L that we can find for our error function such that we'll get an upper bound: $\tilde{E}(\mathbf{w}) \geq E(\mathbf{w})$.

Now, like in EM, if we have a bound that is tight at the current parameters, we can optimize it to obtain new parameters $\mathbf{w}^{(t+1)}$ that have lower cost. We then form a new bound $E(\mathbf{w}; \mathbf{w}^{(t+1)})$ that's tight at the new parameters $\mathbf{w}^{(t+1)}$ and repeat. We could ensure that each update improves the cost, by performing a line-search to find an L that works. In practice we may get faster convergence by setting L larger, even though individual updates could make the cost worse (end of Murphy 13.4.3.2).

To implement the method, we need to be able to find the minimum of

$$\tilde{c}(\mathbf{w}; \mathbf{w}^{(t)}) = \tilde{E}(\mathbf{w}; \mathbf{w}^{(t)}) + \lambda \|\mathbf{w}\|_1. \tag{9}$$

Because the first term is a simple axis-aligned quadratic function, after a few lines of linear algebra, we can find a closed-form solution to

$$\mathbf{w}^{(t+1)} = \arg\min_{\mathbf{w}} \tilde{c}(\mathbf{w}; \mathbf{w}^{(t)}). \tag{10}$$

To compute this solution we first find the locally optimal weights, ignoring the regularizer, which corresponds to a steepest descent update:

$$\mathbf{w}' = \mathbf{w}^{(t)} - \frac{1}{L} \nabla_{\mathbf{w}} E(\mathbf{w})|_{\mathbf{w}=\mathbf{w}^{(t)}} \tag{11}$$

These weights are then individually shrunk and thresholded:

$$w_d^{(t+1)} = w'_d \max\left(0,\, 1 - \frac{\lambda}{|w'_d|}\right). \tag{12}$$

The thresholding means that some of the weights might be set to zero.

There is a similarity with Newton's method: we make a local approximation to the function that we can optimize in closed form, and repeat. The methods can be combined: proximal-Newton methods use the Hessian to form a quadratic approximation, but minimizing a general quadratic added to a L1 regularizer is harder.

5 Final thoughts

There are standard classes of constrained but convex optimization problems such as "linear programs" and "quadratic programs", that have practical algorithms to find the global optimum. For example, a popular classifier called the Support Vector Machine (SVM) can be fitted with a quadratic program solver (although faster more specialized solvers are usually used). There's a school of thought that it's worth working hard to cast a problem as a standard convex optimization as found in a textbook (e.g., Boyd and Vandenberghe, 2004), and then the algorithms have been worked out. I think it's fair to say that much of machine learning has shifted towards using simple optimizers on non-convex problems. However, there will always be applications where being able to reproducibly find a global optimum will be valued.

[4] I learned about it in Convex Optimization with Sparsity-Inducing Norms, Bach et al., 2010.

6 What it’s useful to know

We only covered some superficial details very quickly. However, you might have identified the idea of iteratively making a local approximation and optimizing it, which is common to several methods. Also, the bound-based optimization idea is useful, because it ensures convergence to a local optimum.

It’s useful to be aware that there is a rich variety of optimization algorithms. Given a description, hopefully you’ll think about their possible pros and cons, compared to the more generic gradient-based methods, that are currently the fashion.