COMPSCI 689
Lecture 6: Numerical Optimization

Benjamin M. Marlin

College of Information and Computer Sciences
University of Massachusetts Amherst

Slides by Benjamin M. Marlin (marlin@cs.umass.edu).

Finding Solutions

- Some optimization problems can be solved analytically by forming the system of gradient equations $\nabla f(\mathbf{x}) = 0$ to identify one or more stationary points $\mathbf{x}_*$, and then checking that $\nabla^2 f(\mathbf{x}_*)$ is positive definite (or appealing to convexity).
- When an analytic approach fails, an iterative numerical approach can be used to approximately locate a stationary point.
- The key idea is to produce a sequence of iterates $\mathbf{x}_0, \mathbf{x}_1, \ldots$ starting from and initial guess $\mathbf{x}_0$.
- On each iteration k, we identify a direction $\mathbf{d}_k$ in which f decreases and then take a "step" of length $\alpha > 0$ in that direction to form the new iterate $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha \mathbf{d}_k$.

Finding Solutions

- A key result is that $-\nabla f(\mathbf{x}_k)$ is the direction of steepest decrease in f at $\mathbf{x}_k$.
- Further, for any direction $\mathbf{d}_k$ that has a positive inner product with $-\nabla f(\mathbf{x}_k)$, there exists an α that will decrease f.
- The key problems are how to choose $\mathbf{d}_k$ and how to choose α.

Steepest Descent with Fixed Step Sizes

This algorithm assumes we take $\mathbf{d}_k$ to be the steepest descent direction and we fix α to a constant value.

1. Set $\mathbf{x}_0$, $\alpha > 0$,
2. On each iteration k:
    1. Compute $\mathbf{d}_k$
    2. $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha \mathbf{d}_k$

The problem with this simple method is that it may not converge!

Backtracking Line Search

Backtracking Line Search is a simple method to choose α while monotonically decreasing f.

```
1 Set $\mathbf{x}_0$, $\alpha_0 > 0$, $0 < \rho < 1$
2 On each iteration $k$:
    1 $f_k \leftarrow f(\mathbf{x}_k)$
    2 Compute $\mathbf{d}_k$
    3 $\alpha \leftarrow \alpha_0$
    4 While $f(\mathbf{x}_k + \alpha \mathbf{d}_k) \geq f_k$ : $\alpha \leftarrow \rho\alpha$
    5 $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha \mathbf{d}_k$
```

To ensure that the algorithm makes sufficient progress on each step, the Armijo rule is used: $f_k + c\alpha d_k^T \nabla f(\mathbf{x}_k) \leq f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ for $0 < c \leq 1$.

Common choices are $\mathbf{d}_k = -\nabla f(\mathbf{x}_k)$, $\alpha_0 = 1$, $\rho = 0.5$.

The Newton Direction

- The Newton direction $\mathbf{d}_k = -(\nabla^2 f(\mathbf{x}_k))^{-1} \nabla f(\mathbf{x}_k)$ is the optimal descent direction for a quadratic function. Such a method is often called a "Newton" or "second-order" method since it uses the Hessian and not just the gradient (gradient methods are referred to as "first-order" methods).
- Using the Newton direction to optimize convex functions is typically more efficient than using first-order methods in terms of the number of iterations required.
- However, the overall run-time time for a Newton method can be higher due to the need to compute and invert the hessian matrix on every iteration. Newton methods also need quadratic storage, which can be prohibitive for large-scale problems.

Quasi-Newton Methods

- Quasi-Newton methods attempt to preserve the convergence benefits of Newton methods while reducing the run time and storage requirements.
- The general form of the descent direction is $\mathbf{d}_k = -(\mathbf{B}_k)^{-1}\nabla f(\mathbf{x})$ for a positive definite matrix $\mathbf{B}_k$.
- One of the more robust methods of this type is the Limited Memory BFGS (Broyden-Fletcher-Goldfarb-Shanno) method, which iteratively constructs a low-rank, positive definite approximation to the hessian matrix. It can be used regardless of whether a function is (locally) convex.
- For most problems, L-BFGS will out-perform the use of the steepest descent direction. It's generally a good choice when computationally feasible.

Convergence Assessment

- Convergence of numerical optimization methods is typically assessed either by looking at convergence of the gradient, the function values, or the iterates.
- A gradient condition will look like $\|\nabla f(\mathbf{x})\|_p \leq \tau_1$ for some p norm.
- A function value condition will look like $\frac{|f(\mathbf{x}_k)-f(\mathbf{x}_{k-1})|}{|f(\mathbf{x}_{k-1})|} \leq \tau_2$.
- An iterate condition will look like $\frac{\|\mathbf{x}_k-\mathbf{x}_{k-1}\|_p}{\|\mathbf{x}_{k-1}\|_p} \leq \tau_3$.
- A given method may use a mix of conditions. Incorrectly setting convergence parameters can result in the optimizer stopping too early, or running longer than needed.

Numerical Optimization APIs

- A numerical optimization API will typically request a function that computes the objective function, a function that computes the gradient vector, and an initial guess for $\mathbf{x}_0$.
- Other parameters of an API might include parameters of a convergence rule.
- An optimizer will return the location of the minimizer, the value at the minimum, and information about convergence.
- You always need to pay attention to convergence outputs as any problems with the objective function or gradient vector computation could lead to a failure of the optimizer.
- Common problem error messages are “Invalid descent direction," “Failed to converge," “Invalid encountered in gradient vector," etc.

Debugging Gradients

- Most of the time in machine learning, the form of objective function is known or easily derived (e.g., via maximum likelihood) and is relatively easy to code correctly.
- The gradient vector usually first needs to be derived and then needs to be coded, so there are at least two stages where errors can be made.
- A good debugging strategy is to first implement the objective function and then attempt to verify correctness (e.g., by calculating it by hand or using a separate simpler, but less computationally efficient implementation).
- Next, implement the gradient vector computation and evaluate it using finite differences based on the (hopefully correct) objective function (e.g., scipy.optimize.approx_fprime).

Finite Difference Approximations

- Finite differences can be computed in a few different ways including:

$$\frac{\partial}{\partial x_i} f(\mathbf{x}) \approx \frac{f(\mathbf{x}+\mathbf{e}_i) - f(\mathbf{x}-\mathbf{e}_i)}{2\epsilon}, \qquad e_j = \begin{cases} \epsilon & \text{... if } i = j \\ 0 & \text{... otherwise} \end{cases}$$

- This is typically computed with a value around $\epsilon = 10^{-8}$ for floating point computations.
- The total run time is the the length of the parameter vector times the computational complexity of the objective function.
- You may need to test on a smaller version of the problem you are trying to solve to make the computation feasible.
- For very, very small problems you can sometime run the optimizer using the approximate gradient as a method for checking the objective function.

Gradcheck

- If finite differences indicate that your objective and gradient function do not “agree" for all parameters, you almost certainty have an error somewhere.
- When looking for agreement, you should look for small differences in both the absolute **and** relative errors between analytic and approximate gradients per dimension.
- There will always be some error since you are comparing to an approximation, but with $\epsilon = 10^{-8}$ you should expect at least 4 correct significant figures.
- If your objective and gradient agree for multiple values of $\mathbf{x}$ both in a relative and absolute sense, and you are sure your objective is correct, you are good to go.
- If the optimizer still fails, you have a problem in the objective function!