STA414/2104: Week 2

Introduction to Optimization

Alex Stringer

Jan 8th - 12th, 2018

Adapted from Prof. David Duvenaud's slides on optimization from last year's offering of STA414.

Today

Today, we will start by introducing the concept of optimization, and the gradient-based methods that are extremely prevalent in modern machine-learning research (and under the hood in most software packages).

We will include a brief note on *automatic differentiation*, which is a new tool that gives us gradients of complicated functions for free.

We will then plow through some more theory: geometry of the multivariate Gaussian, more maximum likelihood, and pull it all together with exponential families.

Today

Today will be a disjoint combination of some practice, then a bunch more theory.

Remember, the problem sets are the best idea of what may be asked on the tests.

Optimization

Optimization refers to finding minimums of functions.

Minimize $f(\theta)$ subject to $c(\theta) = 0$.

f is called the *objective function*, and c is the *constraint*.

Maximizing f is the same as minimizing $-f$.

Example: Least Squares

Minimize

$$E(\mathbf{w}) = \frac{1}{2} (\mathbf{X}\mathbf{w} - \mathbf{t})^T (\mathbf{X}\mathbf{w} - \mathbf{t})$$

with respect to $\mathbf{w}$.

No constraints.

We saw that this one has a unique solution.

Optimization

Some simple cases have closed form solutions, e.g. linear regression.

General solutions to this problem are very hard. Most algorithms exploit structure in the objective function and the constraints to simplify things.

No general solution for finding *global* minimums exist. All optimization finds *local* minima.

Gradient and Hessian

The *gradient* of a multivariable function $f : \mathbb{R}^m \to \mathbb{R}$ is the vector of partial derivatives,

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_m} \right)$$

The *Hessian* is the $m \times m$ matrix of second-order partials,

$$H_f(\mathbf{x})_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

If f is twice continuously-differentiable, then the Hessian is symmetric.

Gradient and Hessian

If $\nabla f(\mathbf{x}^*) = 0$ and $H_f(\mathbf{x}^*)$ is *positive definite*, then $\mathbf{x}^*$ is a local minimum of f.

So to optimize a function, set its gradient equal to zero, solve the resulting equations, and check that the Hessian is positive definite.

This gets tricky when we have constraints.

Jacobian

The *Jacobian* of a vector-to-vector function $\mathbf{g} : \mathbb{R}^m \to \mathbb{R}^p$ is the $m \times p$ matrix J_g which satisfies

$$J_g(\mathbf{x})_{ij} = \frac{\partial g_i}{\partial x_j}$$

The i-th row is the gradient of the i-th element of $\mathbf{g}(\mathbf{x})$.

The Hessian is the Jacobian of the gradient (why?)

Automatic Differentiation

Differentiating functions analytically is difficult and time-consuming, and very prone to human error.

Numerical differentiation -approximating derivatives using *finite differences*- is computationally expensive, generally inaccurate in practice, and also *provably* inaccurate beyond a certain point due to the trade-off between the accuracy of the finite-difference approximation and rounding error.

Automatic Differentiation

Every function programmed into a computer can be written as a composition of some elementary operations- addition, multiplication, exponentiation, logarithms, trig, . . . etc.

Calculus gives us a chain rule for differentiating compositions of functions $f(g(h(\mathbf{x})))$.

Using the chain rule analytically results in a lot of tedious work. But computers love tedious work.

Automatic Differentiation exploits this idea to give us *exact derivatives of essentially arbitrary functions, as long as they can be programmed as a composition of operations whose derivatives are known*.

Automatic Differentiation

In Python: `autograd`, https://github.com/HIPS/autograd

In R: `madness` ("Multivariate Automatic Differentiation... ness")

I recommend Python + Autograd. Focus your energy on optimization, not differentiation.

Autograd

```
import autograd.numpy as np
from autograd import grad, jacobian

x1 = np.array([1.,2.,3.])

def vecinscalout(x):
    return np.power(np.linalg.norm(x),2)

vecinscalout(x1) # 14.0
grad(vecinscalout)(x1) # (2,4,6)
jacobian(grad(vecinscalout))(x1)
# array([[ 2.,  0.,  0.],
#        [ 0.,  2.,  0.],
#        [ 0.,  0.,  2.]])
```

Autograd

Homework: verify *analytically* that the examples on the previous slide are correct.

Optimization

We can't go in to any further detail; in math they have entire courses on this, and the sub-field of *operations research* is an entire field of study based around optimization.

We will introduce a very general optimization technique that is used often in machine learning: gradient descent.

Gradient Descent

- Begin with $\mathbf{x}_0$
- Set $t = 0$
- While $\|\nabla f(\mathbf{x})\| > 0$:
  - Pick a step size η_t
  - Set $\mathbf{x}_{t+1} = \mathbf{x}_t - \eta_t \nabla f(\mathbf{x}_t)$
  - Set $t = t + 1$

Gradient Descent Derivation

At each step, we move through the vector space $\mathbb{R}^m$ by a small amount in the direction of the gradient. Why?

Consider a Taylor expansion of f about $\mathbf{x}_t$:

$$f(\mathbf{x}_t) - f(\mathbf{x}_t + \eta_t d) \approx -\eta \nabla f(\mathbf{x}_t)^T d$$

Approximation gets better as η_t gets smaller, because when you zoom in on any differentiable function, it starts getting more and more linear.

We want to pick the direction d that makes $f(\mathbf{x}_t + \eta_t d)$ as small as possible- i.e. we want to move towards the minimum of the function in the quickest way we can for a fixed step size.

So we want to maximize $f(\mathbf{x}_t) - f(\mathbf{x}_t + \eta_t d) \approx -\eta \nabla f(\mathbf{x}_t)^T d$.

Gradient Descent Derivation

For any vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ with $\|\mathbf{y}\| = 1$, we have

$$cos(\theta_{x,y}) = \frac{\mathbf{x}^T\mathbf{y}}{\|\mathbf{x}\|}$$

which is maximized when $cos(\theta) = 1$, or $\mathbf{y} \propto \mathbf{x}$.

So we choose the direction of the gradient, $d = \nabla f(\mathbf{x}_t)$. The proportionality constant just gets rolled into η_t.

Choosing the step size

Choosing η_t is a subjective decision. Technically, it is also an optimization problem.

Usually it is a simple problem, and we use *grid search*, which is a fancy term for "try a bunch of values and pick the one that causes f to decrease the most".

So in practice: start with a big value ("big" might be $\eta_t = 1$ or 0.1), and decrease it by some constant factor like 0.5 until the function decreases.

η_t is sometimes fixed at a constant value for all steps, and called the *learning rate*.

Newton's Method

This can be viewed as a "relaxed" Newton's method.

Newton's method: at each iteration, approximate f using a *quadratic* function, then minimize this approximation.

Newton's Method

- Begin with $\mathbf{x}_0$
- Set $t = 0$
- While $\left\|\nabla f(\mathbf{x})\right\| > 0$:
  - Set $\mathbf{x}_{t+1} = \mathbf{x}_t - H_f(\mathbf{x}_t)^{-1}\nabla f(\mathbf{x}_t)$
  - Set $t = t + 1$

Just replaces the step size with a matrix-multiplication by the inverse Hessian.

Newton's Method

In practice, converges much faster if the starting value $\mathbf{x}_0$ is good, but is very sensitive to starting values.

Computing the Hessian takes $O(m^2)$ operations, and computing its inverse takes $O(m^3)$ operations, in general. So this doesn't scale to high dimensions.

Methods exist that keep a running estimate of the Hessian and update it cheaply at each iteration- "quasi-Newton" methods.

Stochastic Gradient Descent

"Stochastic" means "Random".

Often, the objective function is a sum over the entire training set- for example, the log-likelihood

$$\ell(\theta) = \sum_{i=1}^{n} \ell_i(\theta|\mathbf{x}_i)$$

Derivatives are linear, so the gradient $\nabla\ell(\theta)$ is also a sum over the whole training set.

You can view the log-likelihood (and its gradient), in finite samples, as being *estimates* of the population log-likelihood and gradient as functions of θ.

Do we need to use the whole dataset to compute estimates of population quantities?

Stochastic Gradient Descent

Sub-sample the training data at each iteration, and you get *stochastic gradient descent*.

Less computational burden, but convergence of the algorithm is more sensitive to step size.

When sub-sampling is done without replacement, a single iteration of gradient descent is called an *epoch*, and the actual subset of datapoints chosen is called a *mini-batch*.

"Big Data"

When I first learned of this, I thought it was hilarious: all this talk about "Big Data", but internally all these algorithms are only using a subsample. Dumb, right?

Suppose N, the number of data points, is huge. It turns out that subsampling $M << N$ points at each iteration of gradient descent can provide much more stable, lower-variance estimates and predictions than what would have been obtained if we only had M datapoints.

This is because there is still a non-zero probability of using each one of the N points at some point during the learning procedure.