A Brief Look at Optimization

CSC 412/2506 Tutorial

Presented by Jonathan Lorraine

Slides adapted from last (and prior) year's versions by David Madras

Overview

- Introduction
- Classes of optimization problems
- Linear programming
- Steepest (gradient) descent
- Newton's method
- Quasi-Newton methods
- Conjugate gradients
- Stochastic gradient descent

What is optimization?

- Typical setup (in machine learning, life):
  - Formulate a problem
  - Design a solution (usually a model)
  - Use some quantitative measure to determine how good the solution is.
- E.g., classification:
  - Create a system to classify images
  - Model is some simple classifier, like logistic regression
  - Quantitative measure is classification error (lower is better in this case)
- The natural question to ask is: can we find a solution with a better score?
- Question: what could we change in the classification setup to lower the classification error (what are the free variables)?

Formal definition

$$\begin{aligned} \text{minimize} \quad & f(\theta) \\ \text{subject to} \quad & c(\theta) \end{aligned}$$

- f(θ): some arbitrary function
- c(θ): some arbitrary constraints
- Minimizing f(θ) is equivalent to maximizing -f(θ), so we can just talk about minimization and be OK.
- Games are a generalization of this where each player has a separate objective and parameter set.

Types of optimization problems

- Depending on f, c, and the domain of θ we get many problems with many different characteristics.
- General optimization of arbitrary functions with arbitrary constraints is extremely hard.
- Most techniques exploit structure in the problem to find a solution more efficiently.

Types of optimization

- Simple enough problems have a closed form solution:
  - $f(x) = x^2$
  - Linear regression
- If f and c are linear functions then we can use linear programming (solvable in polynomial time).
- If f and c are convex then we can use convex optimization technique (most of machine learning uses these).
- If f and c are non-convex we usually pretend it's convex and find a sub-optimal, but hopefully good enough solution (e.g., deep learning).
- There are also specialized techniques for non-convex opt, which not may be helpful for convex objectives.
- In the worst case there are global optimization techniques (operations research is very good at these).
- There are yet more techniques when the domain of θ is discrete.
- This list is far from exhaustive.

Types of optimization

- Takeaway:

  Think hard about your problem, find the simplest category that it fits into, use the tools from that branch of optimization. There is "No Free Lunch"

- Sometimes you can solve a hard problem with a special-purpose algorithm, but most times we favor a black-box approach because it's simple and usually works.

Really naïve optimization algorithm

- Suppose $$\theta \in [a_i, b_i]^D$$
  - D-dimensional vector of parameters where each dimension is bounded above and below. "Box constraints" are easy.
- For each dimension I pick some set of values to try:

$$\mathbf{x}_i = \{x_{i1}, x_{i2}, \ldots, x_{iN}\}$$

- Try all combinations of values for each dimension, record f for each one.
- Pick the combination that minimizes f.

Really naïve optimization algorithm

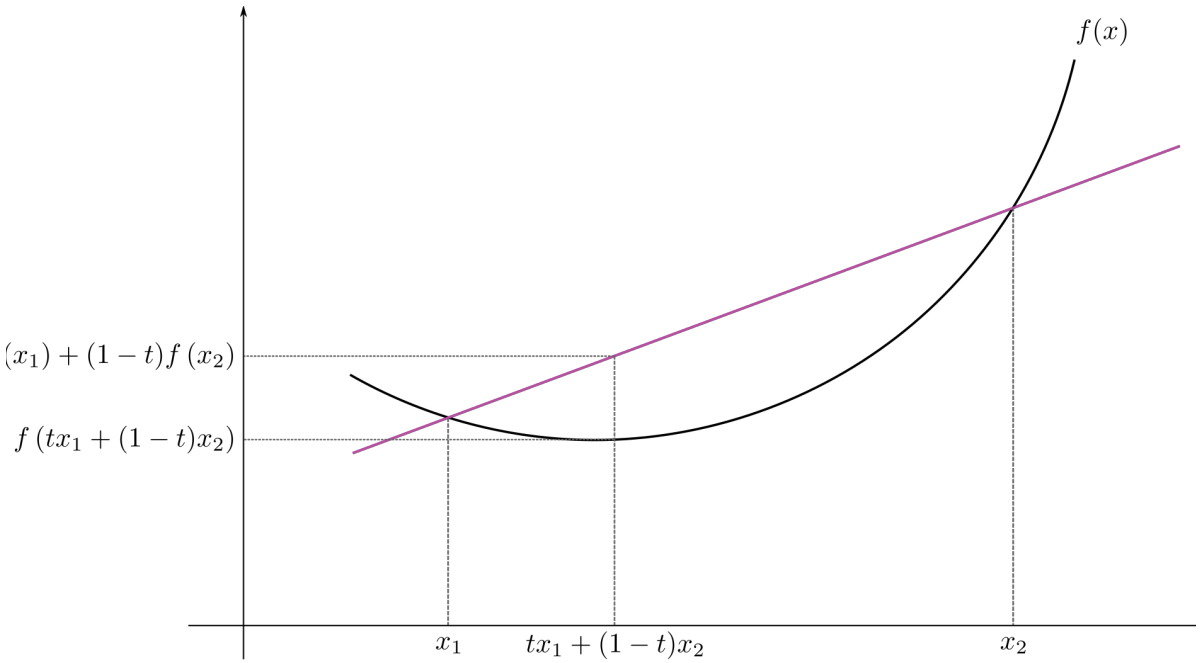
- This is called grid search. It works really well in low dimensions when you can afford to evaluate f many times.
- Less appealing when f is expensive, in high dimensions, or non-smooth.
- You may have already done this when searching for a good L2 penalty value.

Convex functions

A function f is convex iff...

$$\forall \theta_1, \theta_2, \forall \alpha \in [0,1]:$$

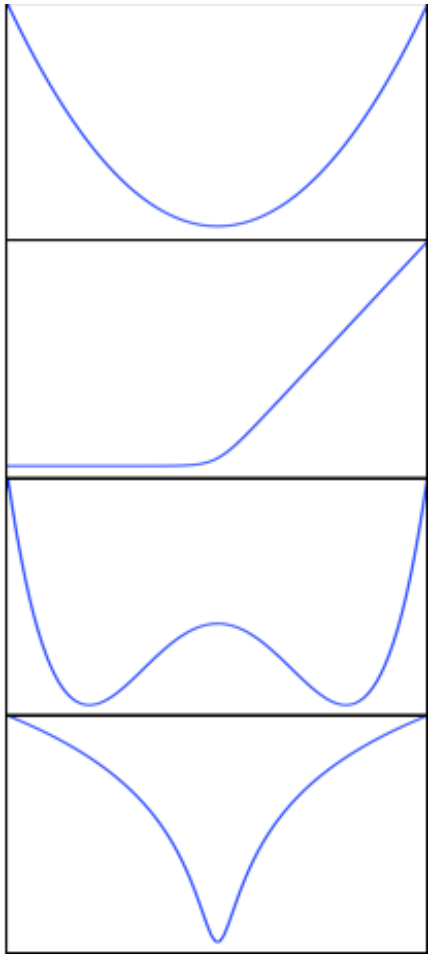
$$f(\alpha\theta_1 + (1-\alpha)\theta_2) \leq \alpha f(\theta_1) + (1-\alpha) f(\theta_2)$$



Use the line test.

Convex functions

Which functions are convex?



Convex optimization

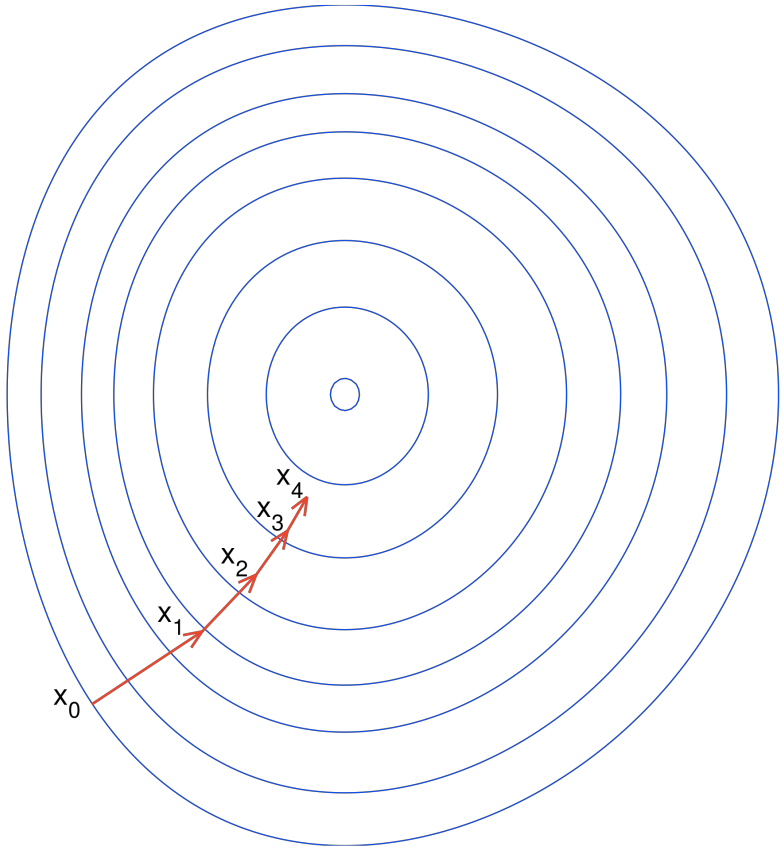
- We've talked about 1D functions, but the definition still applies to higher dimensions.
- Why do we care about convex functions?
- In a convex function, any local minimum is automatically a global minimum. This means a greedy update always takes us "closer" to the optimum!
- Thus we can apply fairly naïve techniques to find the nearest local minimum and still guarantee that we've found the best solution!
- **This is not true for non-convex functions.**

Steepest (gradient) descent

- Cauchy (1847)

Begin with some initial θ_0
$t \leftarrow 0$
while not converged:

- Pick a step size η_t
- $\theta_{t+1} \leftarrow \theta_t - \eta_t \nabla f(\theta_t)$
- $t \leftarrow t + 1$

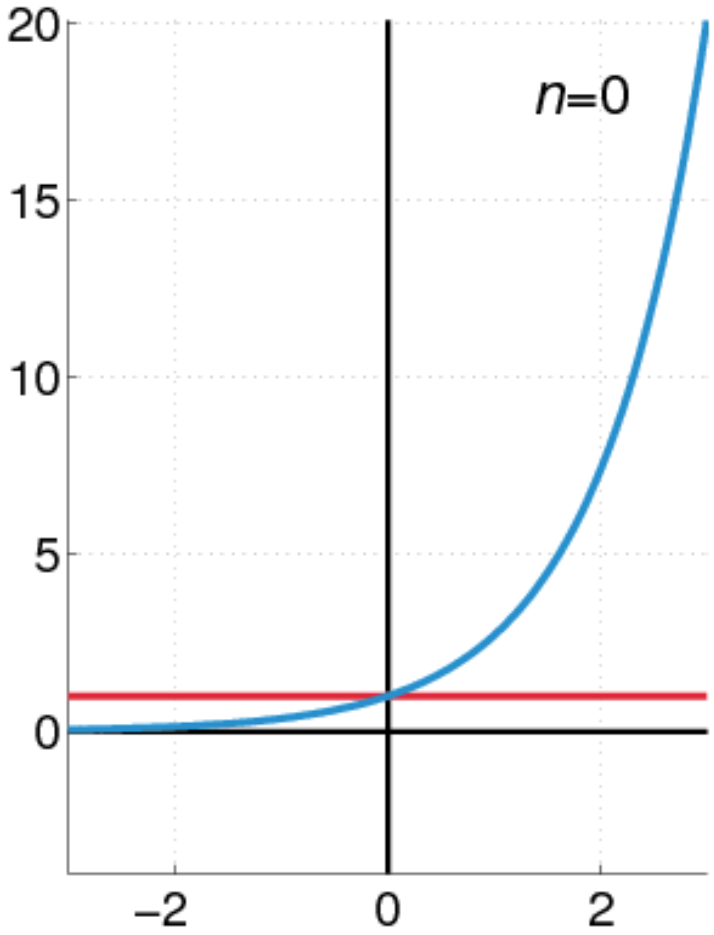

x_0
x_1
x_2
x_3
x_4

Aside: Taylor series

- A Taylor series is a polynomial series that converges to a function f.
- We say that the Taylor series expansion of at x around a point a, f(x + a) is:

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \cdots.$$

- Truncating this series gives a polynomial approximation to a function.



Blue: exponential function; Red: Taylor series approximation

Multivariate Taylor Series

- The first-order Taylor series expansion of a function f(θ) around a point d is:

$$f(\theta + d) \approx f(\theta) + \nabla f(\theta)^\top d$$

Steepest descent derivation

- Suppose we are at θ and we want to pick a direction d (with norm 1) such that f(θ + ηd) is as small as possible for some step size η. This is equivalent to maximizing f(θ) - f(θ + ηd).
- Using a linear approximation:

$$f(\theta) - f(\theta + \eta d) \approx f(\theta) - f(\theta) - \eta \nabla f(\theta)^\top d$$
$$= -\eta \nabla f(\theta)^\top d$$

- This approximation gets better as η gets smaller since as we zoom in on a differentiable function it will look more and more linear. "Approximately locally linear"

Steepest descent derivation

- We need to find the value for d that maximizes $-\eta \nabla f(\theta)^\top d$ subject to $||d||_2 = 1$
- We could use a Lagrange multiplier to deal with the constraint, or…
- Using the definition of cosine as the angle between two vectors:

$$\cos(\theta) = \frac{\nabla f(\theta)^T d}{||\nabla f(\theta)||_2 ||d||_2}$$

Rearranging terms (and noting that $||d||_2 = 1$),

$$\cos(\theta)||\nabla f(\theta)||_2 = \nabla f(\theta)^\top d$$

This means that $\nabla f^\top d$ is maximized when $\cos(\theta) = 1$, or $d \propto \nabla f(\theta)$. In this case, because of the norm constraint we have that $d = \frac{\nabla f(\theta)}{||\nabla f(\theta)||_2}$. We can roll the denominator into η.

Therefore, to maximize $-\eta \nabla f(\theta)^\top d$, we should set d to be $-\eta \nabla f(\theta)$. This recovers steepest descent.

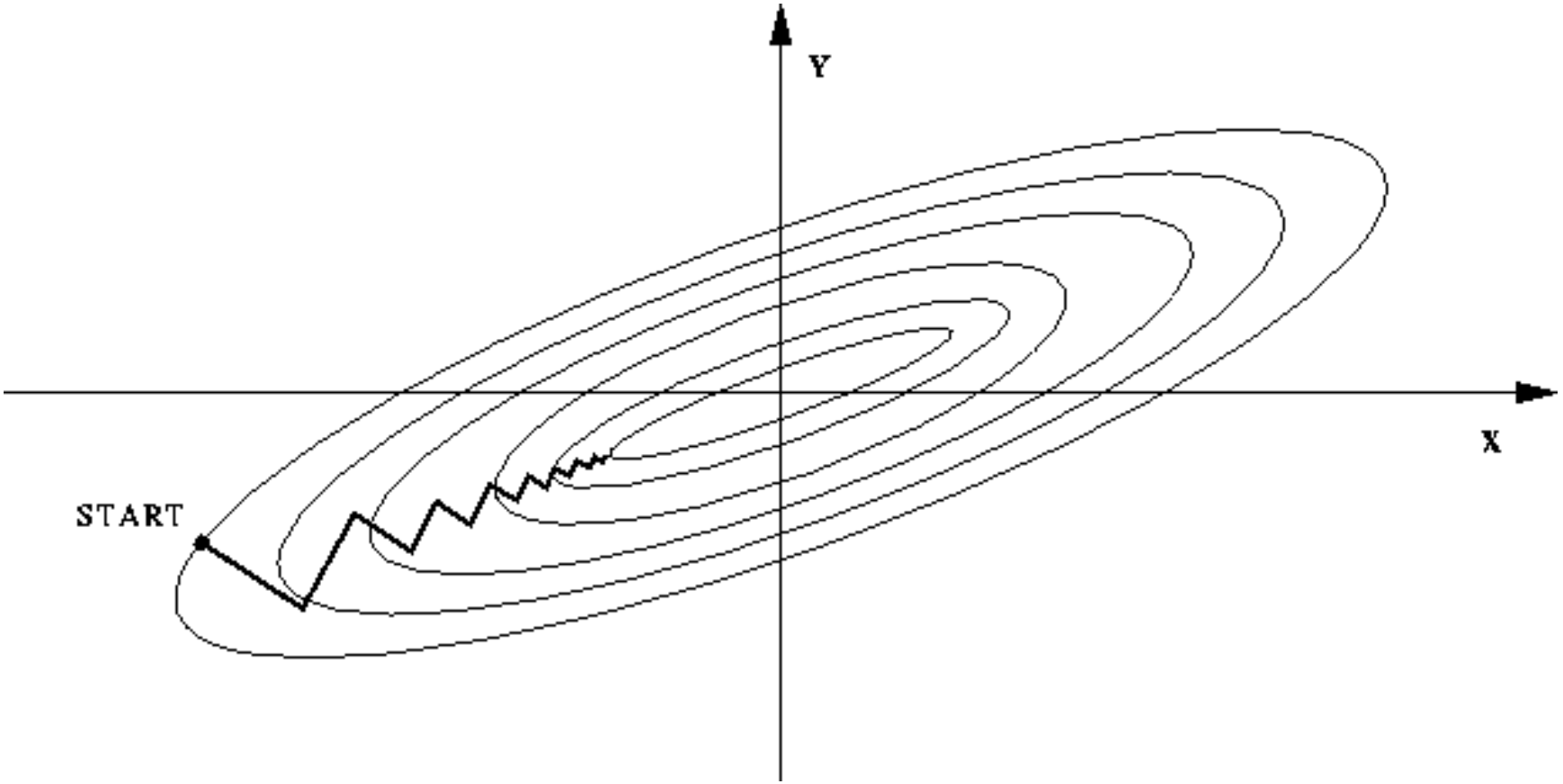
How to choose the step size?

- At iteration t
- General idea: vary η_t until we find the minimum along $f(\theta - \eta_t \nabla f(\theta))$
- This is a 1D optimization problem.
- In the worst case we can just make η_t very small, but then we need to take a lot more steps.
- General strategy: start with a big η_t and progressively make it smaller by e.g., halving it until the function decreases.

When have we converged?

- When $||\nabla f(\theta)|| = 0$ (for no constraints)
- If the function is convex then we have reached a global minimum.
- If we have equality constraints, this generalizes to the Lagrange conditions.
- For inequality & equality constraints, we get the KKT conditions.
- Constraints can be extremely difficult to deal with!

The problem with gradient descent



source: http://trond.hjorteland.com/thesis/img208.gif

Newton's method

- To speed up convergence, we can use a more accurate approximation.
- Second order Taylor expansion:

$$f(\theta + \eta d) \approx f(\theta) + \eta \nabla f(\theta)^\top d + \frac{\eta}{2} d^\top H(\theta) d$$

- H is the *Hessian* matrix containing second derivatives.

$$H_{ij}(\theta) = \frac{\partial^2 f(\theta)}{\partial \theta_i \partial \theta_j}$$

- H has non-negative eigenvalues if f convex! For a single optimization objective, H symmetric so real eigenvalues.

Newton's method

Once again we seek to minimize $f(\theta) - f(\theta + \eta d)$ with respect to d. Differentiating with respect to d gives:

$$-\nabla f(\theta + \eta d) = -\eta f(\theta) - \eta H(\theta) d$$

Setting this to 0:

$$d = -H(\theta)^{-1} \nabla f(\theta)$$

What is it doing?

- At each step, Newton's method approximates the function with a quadratic bowl, then goes to the minimum of this bowl.
- For twice or more differentiable (C^2) convex functions, this is usually much faster than steepest descent.
- Con: computing Hessian requires $O(D^2)$ time and storage. Inverting the Hessian is even more expensive (up to $O(D^3)$). This is problematic in high dimensions.
- For non-convex objectives, we don't know if this is a descent direction… Here a better choice may be generalized Gauss-Newton matrix (GGN), which is always PSD!

Quasi-Newton methods

- Computation involving the Hessian is expensive.
- Modern approaches use computationally cheaper *approximations* to the Hessian or it's inverse.
- Deriving these is beyond the scope of this tutorial, but we'll outline some of the key ideas.
- These are implemented in many good software packages in many languages and can be treated as black box solvers, but it's good to know where they come from so that you know when you use them.

K-FAC

- Estimate the Fisher as the Kronecker product of matrices

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix} \qquad (\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}$$

- Efficiently compute inverse, by inverting the factors!

BFGS

- Maintain a running estimate of the Hessian B_t.
- At each iteration, set $B_{t+1} = B_t + U_t + V_t$ where U and V are rank 1 matrices (these are derived specifically for the algorithm).
- Using a low-rank update to improve the Hessian estimate allows cheap inversion at each iteration.
- Sherman-Morrison formula: $(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}$
- Memory cost could blow up – use L-BFGS.

Conjugate gradients

- Steepest descent often picks a direction it's travelled in before (this results in the wiggly behavior).
- Conjugate gradients make sure we don't travel in the same direction again (orthogonal directions). Move exactly as far as we need in each direction one time!
- Orthogonal := $x^T_i x_j = 0$, A-orthogonal := $x^T_i A x_j = 0$
- Will exactly minimize a d-dimensional (PSD) quadratic function f in d-steps. 2^{nd} order methods finish in 1 step, while 1^{st} order methods may never exactly reach the optimum (despite getting arbitrarily close).
- Use for solving linear systems with large, symmetric (or Hermitian) matrices.
- Takeaway: conjugate gradient sometimes works better than 1^{st} order methods, and can be cheaper than 2^{nd} order methods. It also has a linear per-iteration cost.

Stochastic Gradient Descent

- Recall that we can write the log-likelihood of a distribution as:

$$\mathcal{L}(x|\theta) = \sum_{i=1}^{N} \log P(x_i|\theta)$$

$$\nabla\mathcal{L}(x|\theta) = \sum_{i=1}^{N} \frac{\nabla P(x_i|\theta)}{P(x_i|\theta)}$$

Stochastic gradient descent

- Any iteration of a gradient descent (or quasi-Newton) method requires that we sum over the entire dataset to compute the gradient.
- SGD idea: at each iteration, sub-sample a small amount of data (even just 1 point can work) and use that to estimate the gradient.
- Each update is noisy, but very fast!
- This is the basis of optimizing ML algorithms with huge datasets (e.g., recent deep learning).
- Computing gradients using the full dataset is called batch learning, using subsets of data is called mini-batch learning.

Stochastic gradient descent

- Suppose we made a copy of each point, y=x so that we now have twice as much data. The log-likelihood is now:

$$\mathcal{L}(x|\theta) = \sum_{i=1}^{N} \log P(x_i|\theta) + \sum_{i=1}^{N} \log P(y_i|\theta) = 2\sum_{i=1}^{N} \log P(x_i|\theta)$$

- In other words, the optimal parameters don't change, but we have to do twice as much work to compute the log-likelihood and it's gradient!
- The reason SGD works is because similar data yields similar gradients, so if there is enough redundancy in the data, the noisy from subsampling won't be so bad.

Stochastic gradient descent

- In the stochastic setting, line searches are less useful because they depend on the batch.
- So how do we choose an appropriate step size?
- Robbins and Monro (1951): pick a sequence of η_t such that:

Lim_{t-> inf} \eta_t = 0, $\sum_{t=0}^{\infty} \eta_t = \infty, \ \sum_{t=0}^{\infty} \eta_t^2 < \infty$

- Satisfied by $\eta_t \propto \frac{1}{t}$ (as one example).
- Balances “making progress” with averaging out noise.

- In the non-convex setting there is no-free lunch with step sizes.

Final words on SGD

- SGD is very easy to implement compared to other methods, but the step sizes need to be tuned to different problems, whereas batch learning typically "just works".
- Tip 1: divide the log-likelihood estimate by the size of your mini-batches. This makes the learning rate invariant to mini-batch size.
- Tip 2: subsample without replacement so that you visit each point on each pass through the dataset (this is known as an epoch).

- **SGD works better than full-batch for non-convex optimization. Why do you think this is true?**

Useful References

- Linear programming:
  - Linear Programming: Foundations and Extensions (http://www.princeton.edu/~rvdb/LPbook/
- Convex optimization:
  - http://web.stanford.edu/class/ee364a/index.html
  - http://stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf

- LP solver:
  - Gurobi: http://www.gurobi.com/
- Stats (python):
  - Scipy stats:http://docs.scipy.org/doc/scipy-0.14.0/reference/stats.html
- Optimization (python):
  - Scipy optimize: http://docs.scipy.org/doc/scipy/reference/optimize.html
- Optimization (Matlab):
  - minFunc: http://www.cs.ubc.ca/~schmidtm/Software/minFunc.html
- General ML:
  - Scikit-Learn: http://scikit-learn.org/stable/

- Martens, James, and Roger Grosse. "Optimizing neural networks with kronecker-factored approximate curvature." *International conference on machine learning*. 2015.