A Brief Look at Optimization

- Introduction
- Classes of optimization problems
- Grid search
- Gradient descent
- Newton's method
- Quasi-Newton methods
- Stochastic gradient descent

Thanks to David Duvenaud, Jimmy Ba, and others

What is optimization?

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

Formal definition

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

- $f(\theta)$: some arbitrary function
- $c(\theta)$: some arbitrary constraints

Minimizing $f(\theta)$ is equivalent to maximizing $-f(\theta)$, so we'll just talk about minimization

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
- 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)
- 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

- Sometimes you can solve a hard problem with a special-purpose algorithm, but often we favour 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
- 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

curse of dimensionality,
amount of search is exponential to dimension

- Less appealing when *f* is expensive or in high dimensions
- You may have already done something like this when searching for a good L_2 penalty value

say pick several lambdas

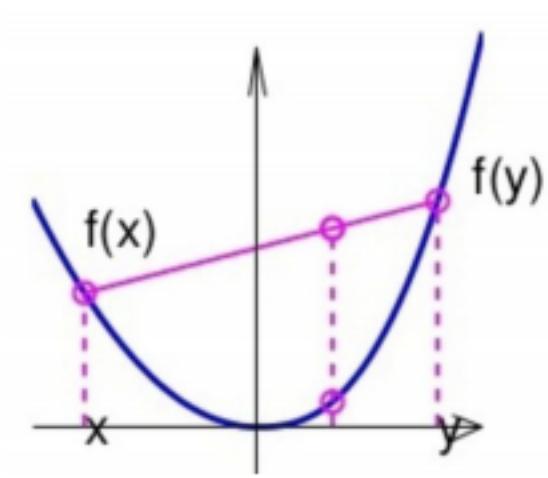
Convex functions

A function f is convex iff...

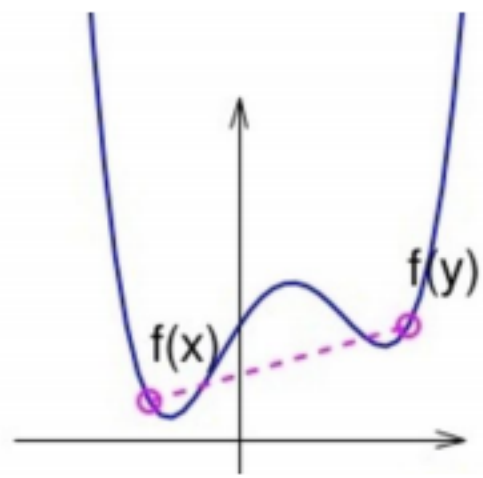
$$\forall \theta_1, \theta_2, \forall \alpha \in [0, 1] :$$

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

convex



non-convex

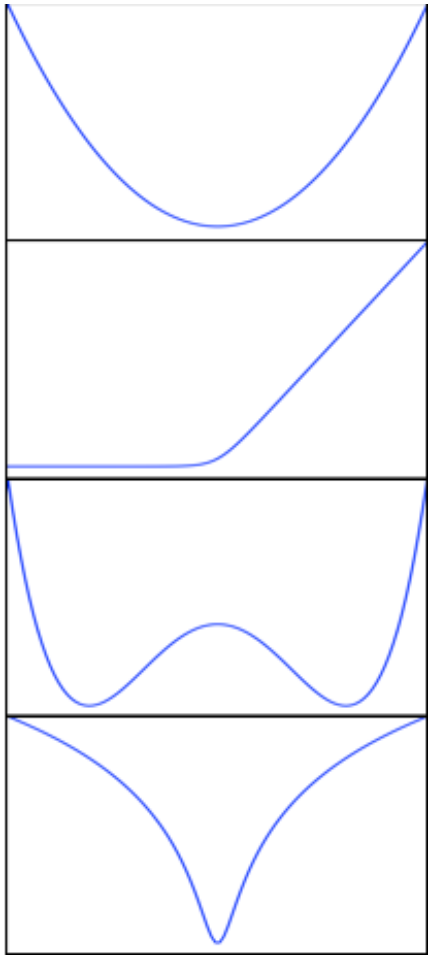


Use the line test.

pick 2 points on function and connect them to form a line,
function always below the line

Convex functions

Which functions are convex?



yes

yes

no

no

Convex optimization

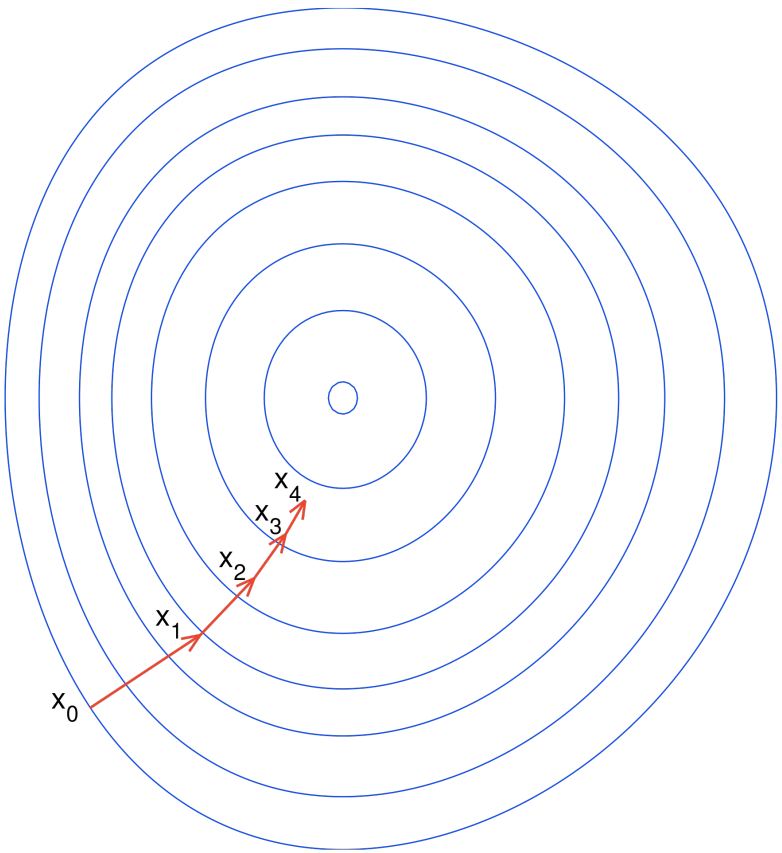
- We've talked about 1D functions, but the definition still applies to higher dimensions
- Why do we care about convex functions?
- In a convex function, any local minimum is automatically a global minimum
- This means we can apply fairly naïve techniques to find the nearest local minimum and still guarantee that we've found the best solution!

Steepest (gradient) descent

- Cauchy (1847)

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

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

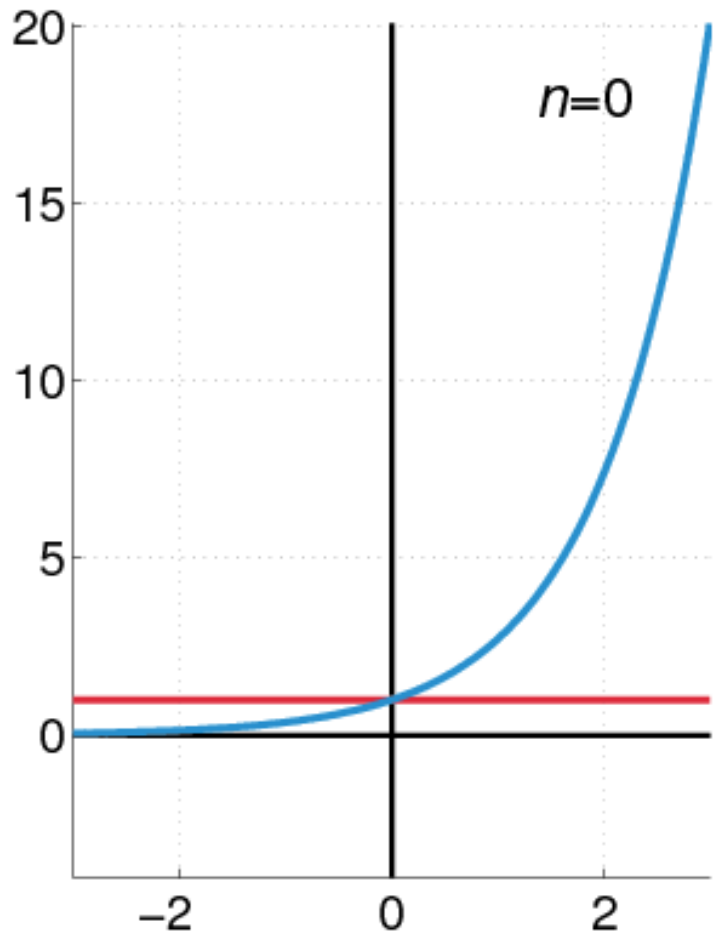

x_0
x_1
x_2
x_3
x_4

Aside: Taylor series

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

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

- Truncating this series gives a polynomial approximation to a function



Blue: exponential function; Red: Taylor series approximation

Multivariate Taylor Series

- For vector θ instead of scalar *a,* the first-order Taylor series expansion of a function *f(θ)* around a point is:

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

How to choose the step size?

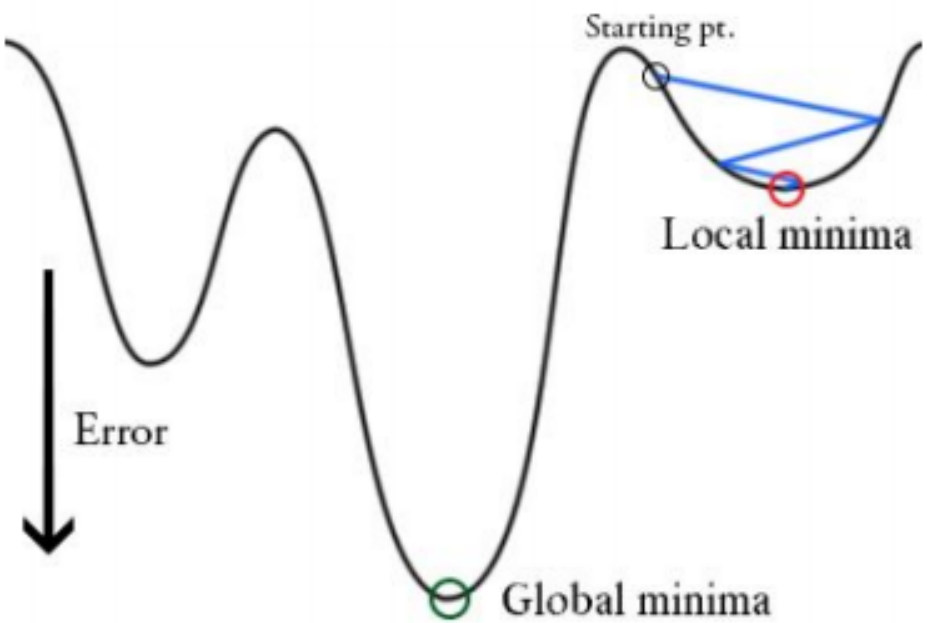
- At iteration t

Idea is that the update is instead a

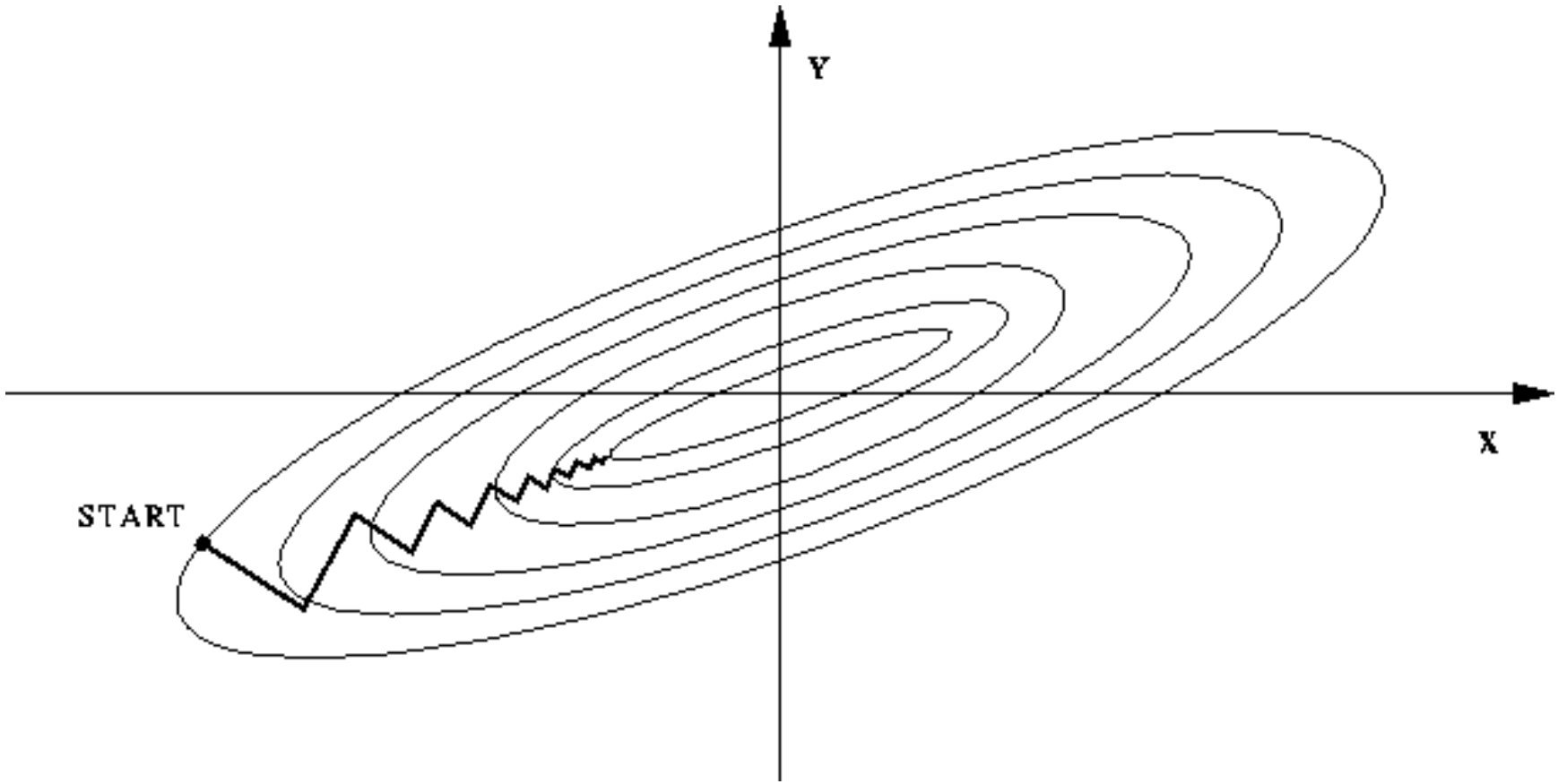
- General idea: vary η_t until we find the minimum along $f(\theta - \eta_t \nabla f(\theta))$
- This is a 1D optimization problem
- In the worst case we can just make η_t very small, but then we need to take a lot more steps
- General strategy: start with a big η_t and progressively make it smaller,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

When have we converged?

- When $||\nabla f(\theta)|| = 0$
- If the function is convex then we have reached a global minimum. Otherwise, not necessarily:



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

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

Quick review from Calculus – the Hessian is:

2nd order partial

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

- The Hessian is the Jacobian of the gradient:

$$\mathbf{H}(f(\mathbf{x})) = \nabla\nabla f(\mathbf{x}) = \frac{\partial(\nabla f(\mathbf{x}))}{\partial \mathbf{x}} = \mathbf{J}(\nabla f(\mathbf{x}))$$

Jacobians

$$\mathbf{J}_{\mathbf{x}\to\mathbf{y}} = \mathbf{J}(\mathbf{y}(\mathbf{x})) = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial(y_1, y_2, \ldots y_m)}{\partial(x_1, x_2, \ldots x_n)}$$

$$= \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

= (partial f / partial x_1 , …. , partial f / partial x_n)

J_{ij} = partial f_i / partial x_j

The determinant, $\left|\mathbf{J}_{\mathbf{x}\to\mathbf{y}}\right|$, measures how much a unit cube changes in volume when we transform the data from $\mathbf{x}$ to $\mathbf{y}$.

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 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

Quasi-Newton methods

- Computation involving the Hessian is expensive
- Modern approaches use computationally cheaper *approximations* to the Hessian or its inverse
- We won't derive these but will 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

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)

  sums to rank-two update matrix.
- The advantage of using a low-rank update to improve the Hessian estimate is that B can be cheaply inverted at each iteration

Limited memory BFGS

- BFGS progressively updates B and so one can think of B_t as a sum of rank-1 matrices from steps 1 to t. We could instead store these updates and recompute B_t at each iteration (although this would involve a lot of redundant work)
- L-BFGS only stores the most recent updates, therefore the approximation itself is always low rank and only a limited amount of memory needs to be used (linear in D)
- L-BFGS works extremely well in practice
- L-BFGS-B extends L-BFGS to handle bound constraints on the variables

Stochastic Gradient Descent

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

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

maximize log likelihood
minimize negative log likelihood

and so:

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

Stochastic gradient descent

- Any iteration of a gradient-descent method (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 its gradient!
- One reason SGD works is because similar data yields similar gradients, so if there is enough redundancy in the data, the noise from subsampling won't be so bad

Stochastic gradient descent

- In the stochastic setting, line searches break down and so do estimates of the Hessian, so stochastic quasi-Newton methods are very difficult to get right
- So how do we choose an appropriate step size?
- Robbins and Monro (1951): pick a sequence of η_t such that:

$$\sum_{t=0}^{\infty} \eta_t = \infty, \ \sum_{t=0}^{\infty} \eta_t^2 < \infty$$

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

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)

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/