Randomized Algorithms for $\mathbf{Ax} = \mathbf{b}$

Benjamin Chu

January 2, 2020

Problem statement: Given matrix $\mathbf{A}$ and vector $\mathbf{b}$, find vector $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$.

1 Review

Some motivating examples:

- Least squares seek the solution to $\mathbf{X}^t\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^t\mathbf{y}$.
- PDE solvers and finite element methods
- Image reconstruction: a series of angular projections (i.e. sinogram) can be used to reconstruct an image.

Notable computational challenges

- $\mathbf{A}$ may be large: a dense $10^6 \times 10^6$ float64 matrix is 6 terabytes
- $\mathbf{A}$ may be fat&short or tall&skinny, so $\mathbf{x}$ may not be unique or even exist (i.e. need approximations).
- Numerical stability

Messages:

- Never do $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ for any real problem.
- Take advantage of $\mathbf{A}$'s special structure. (e.g. sparse, sparse + rank 1 update, triangular...etc). Learn everything about numerical linear algebra in Dr. Hua Zhou's class - biostats M280. Biomath 205 and Math 270c may be good to learn theories.
- Large, dense problems requires *randomized* iterative methods.

1.1 Direct methods for dense, small problems

Solving $\mathbf{Ax} = \mathbf{b}$ (review):

(1) Factorize:

- $\mathbf{LU}$: $\mathbf{A} = \mathbf{LU}, \mathbf{L}$ = lower triangular matrix and $\mathbf{U}$ = upper triangular
- $\mathbf{QR}$: $\mathbf{A} = \mathbf{QR}, \mathbf{QQ}^t = \mathbf{I}$ and $\mathbf{R}$ = upper triangular.
- Cholesky: $\mathbf{A} = \mathbf{LL}^t, \mathbf{L}$ = lower triangular matrix.

(2) Solving:

- $\mathbf{LU}$: $\mathbf{Ax} = \mathbf{b} \iff \mathbf{L}(\mathbf{Ux}) = \mathbf{b} \iff \mathbf{Ly} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{y}$.
- $\mathbf{QR}$: $\mathbf{Ax} = \mathbf{b} \iff \mathbf{QRx} = \mathbf{b} \iff \mathbf{Rx} = \mathbf{Q}^t\mathbf{b}$
- Cholesky: $\mathbf{Ax} = \mathbf{b} \iff \mathbf{LL}^t\mathbf{x} = \mathbf{b} \iff \mathbf{L}^t\mathbf{x} = \mathbf{y}$ and $\mathbf{Ly} = \mathbf{b}$.

(3) Remarks:

- Last step in (2) uses backward substitution due to triangular structure.
- For non-square matrices, step (1) uses QR (in general).
- For square matrices,
  - If $\mathbf{A}$ is positive definite, step (1) use Cholesky (in general).
  - Otherwise, step (1) uses LU (in general).
- Other methods (e.g. householder, modified Gram Schmidt, SVD, sweep) may be useful in certain cases.
- When are QR/LU/Cholesky preferred over iterative/stochastic-iterative algorithms?

Example 1.1 Direct methods require "small" matrices

My 2019 macbook pro with 8 Intel Core i9-9880H CPUs have 423.2 GFLOPS computing power total. Thus, it runs 423,200,000,000 (single precision) floating point operations per second. Algorithms like QR/LU scale as $O(n^3)$. How long does it take to compute a factorization for a $10,000 \times 10,000$ matrix? Ans: $10000^3/423200000000 \approx 2.4$ seconds. If the matrix is 10 times bigger (homework: is it possible to make this matrix on a laptop?), it would take roughly 2362 seconds, or 39 minutes. Another 10 times bigger would take 655 hours. Obviously, factorization methods break down in this regime.

1.2 Iterative methods for large, sparse problems

Large, sparse problems are usually solved by iterative methods. An iterative algorithm means an algorithm that computes a sequence of points $x_0, x_1, ..., x_k \to x_{\text{true}}$ as $k \to \infty$. This algorithm converges when $x_k \approx x_{k+1}$, which hopefully implies $x_k \approx x_{\text{true}}$ with some reasonably small k. Now, one must first define what is meant by a solution x_{true} since $\mathbf{A}$ may be over/under-determined. The following is typically used:

Definition 1.2

$\mathbf{x}^*$ is the solution to $\mathbf{Ax} = \mathbf{b}$ if $\mathbf{x}^* = \text{argmin}_{\mathbf{x}\in\mathbb{R}^n} ||\mathbf{Ax} - \mathbf{b}||$ with respect to some norm.

Why do we need iterative algorithms instead of QR/LU/Cholesky for sparse systems? Even if $\mathbf{A}$ is sparse, $\mathbf{L}, \mathbf{U}, \mathbf{Q}, \mathbf{R}$ is generally not sparse (e.g. consider $\mathbf{A}$ with 1's along first row and column, and along the diagonal). Banded matrices is an exception. A number of non-stochastic iterative algorithms algorithms is popular:

- Conjugate gradient (sparse, symmetric, positive definite)
- GMRES (requires good preconditioner)
- (L-) BFGS (requires approximate hessian)
- https://juliamath.github.io/IterativeSolvers.jl/dev/

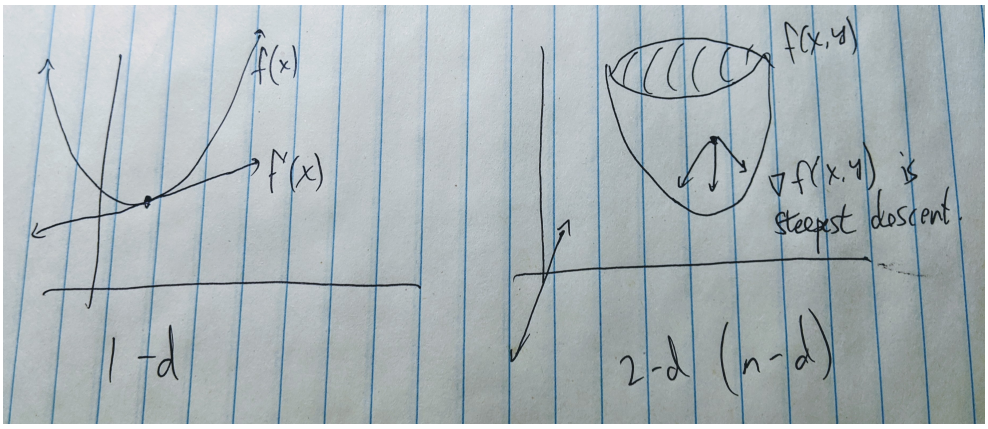
If $\mathbf{A}$ is not sparse enough, then one cannot load $\mathbf{A}$ into RAM and the methods above break down. For large and dense problems, recent literature focuses on *randomized* algorithms. We cover 2:

- Stochastic gradient descent
- Randomized Kaczmarz (used for overdetermined systems, wiki says sparsity is needed to be superior to conjugate gradient)
- Randomized Gauss-Siedel, Randomized MM algorithm?

2 Stochastic gradient descent for large, dense problems

When $\mathbf{A}$ is dense and large, we cannot load the matrix into memory. Therefore the idea is to approximate the gradient with a subset of $\mathbf{A}$. For SGD, this subset is a single row of $\mathbf{A}$, denoted by $\mathbf{A}_i^t$. We first begin with intuition behind gradient descent.

2.1 Geometric intuition of gradient descent



2.2 Why does gradient methods work, mathematically?

We want to solve $\mathbf{Ax} = \mathbf{b} \iff \mathbf{x}$ minimizes $f(\mathbf{x}) = \frac{1}{2}||\mathbf{Ax}-\mathbf{b}||_2^2 + \lambda||\mathbf{x}||_2$. Gradient methods achieve this via iterations:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - s_k \nabla f(\mathbf{x}_k), \tag{2.1}$$

where s_k is a positive step size. $\lambda||\mathbf{x}||_2$ term is typically added to prefer solutions with small coefficients.

Theorem 2.1

The iteration defined by $\mathbf{x}_{k+1} = \mathbf{x}_k - s_k \nabla f(\mathbf{x}_k)$ satisfies $f(x_{k+1}) \leq f(x_k)$.

Proof.

$$\begin{aligned} f(\mathbf{x}_{k+1}) &= f(\mathbf{x}_k - s_k \nabla f(\mathbf{x}_k)) \\ &= f(\mathbf{x}_k) - s_k \nabla f(\mathbf{x}_k)^t \nabla f(\mathbf{x}_k) + O(s_k^2) \\ &= f(\mathbf{x}_k) - s_k ||\nabla f(\mathbf{x}_k)||_2^2 + O(s_k^2) \\ &\leq f(\mathbf{x}_k). \end{aligned}$$

Here the second equality is by Taylor series and the last line is because $s_k > 0$. □

- Geometric intuition of gradient descent. Note that $\nabla \frac{1}{2}||\mathbf{Ax}-\mathbf{b}||_2^2 = \mathbf{A}^t(\mathbf{A}^t\mathbf{x}-\mathbf{b})$ (see homework).
- Why does gradient methods work? (theorem showing each iteration decreases error)
- Explain savings in memory
- Idea behind SGD is how it agrees with GD *in expectation*. (last part of proof requires some notation: see homework) Variance does not decrease but we don't care about that. Show example of SGD with

missing data Needell's paper

- Convergence? Rate of convergence?
- Discuss parallel computing -> recall Hogwild paper claims that lockings not required if matrix is sparse (i.e. race condition is rare).

3 Kaczmarz and Randomized Kaczmarz method for overdetermined systems

- Krylov subspace methods
- Iteration scheme and geometric intuition
- Exponential convergence rate
- Does not converge to OLS solution for inconsistent systems.
- RK is special case of SGD with reweighted importance sampling

4 Problems

In honor of Ken Lange, you are required to do 2 problems. If you do more I will grade your top 2 problems. Every problem is worth the same number of points. If a problem has subproblems, each subproblem is worth the same number of points.

Problem 4.1 BLAS is better than you

Consider the problem of dense matrix-vector multiplication.

(1) Write your own matrix-vector multiplication routine in Julia.

(2) Compare your code's speed (consider using `BenchmarkTools.jl`) to BLAS routines on $\mathbf{X} \in \mathbb{R}^{n\times n}$ matrices where $n = 100, 1000$, and 10000. Which is faster?

(3) *Parallelize* your code using Julia's built-in multithreading (suitable for single machine parallel code) or `Distributed.jl` package (suitable for multi-core distributed computing), then perform the comparison. What is the speedup in your code? Did you have to use additional memory? How close are you to beating BLAS?

Problem 4.2 Sparse matrices is somtimes better than BLAS

Consider the problem of dense matrix-vector and matrix-matrix multiplication when the matrices involved is potentially sparse.

(1) Suppose $\mathbf{X} \in \mathbb{R}^{n\times n}$ where $n = 10^6$. How much RAM memory is required (in gigabytes) to store

a dense matrix in double precision (i.e. Float64)? In single precision (Float32)? Half precision (Float16)? You will need at least this much RAM in order to create a matrix of this size.

(2) If $n = 10^6$ and only 1% of the entries are non-zero, how much memory do you need to store a sparse double precision matrix? Create such a matrix with Julia's `SparseArrays.jl` and verify your guess with Julia's `sizeof()` command.

(3) Using functions associated with Julia's `SparseArrays.jl`, generate *sparse* matrices with sizes $n = 10^4, 10^5, 10^6$ and with sparsity level $0.001, 0.01, 0.1$. Convert these matrices to *dense* matrices (hint: is this possible?), and compare the speed of a (sparse) matrix-vector and matrix-matrix multiplication to a (dense) matrix-vector and matrix-matrix multiplication. Why is BLAS slower in some cases but not others? Use dense vectors for the matrix-vector multiplication.

Problem 4.3 Exact 2nd order Taylor's expansion

Suppose f is continuous and twice differentiable. Show that there exists $y \in (x_0, x)$ such that:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(y)(x - x_0)^2.$$

This fact is used (in chapter 9.2, without proof) to motivate the quadratic upper bound principle used ubiquitously in MM algorithms.

Problem 4.4 Notation problem

Let $\mathbf{X} \in \mathbb{R}^{n \times p}$, $\lambda_i \in \mathbb{R}$, and $\mathbf{x}_i^T \in \mathbb{R}^{1 \times p}$ be row i of $\mathbf{X}$. Show that

$$\sum_{i=1}^{n} \lambda_i \mathbf{x}_i \mathbf{x}_i^T = \mathbf{X}^T \begin{bmatrix} \lambda_1 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \lambda_n \end{bmatrix} \mathbf{X}$$

This is standard notation (e.g. used in section 3.4 of Dobson and Barnett (2008)), and in my proof that SGD agrees with GD in expectation.

Problem 4.5 L2 norm enjoys nice quadratic behavior

Let $\mathbf{A} \in \mathbb{R}^{n \times p}, \mathbf{x} \in \mathbb{R}^p$ and $\mathbf{b} \in \mathbb{R}^n$.

(1) Compute the gradient of $F(\mathbf{x}) = \frac{1}{2}||\mathbf{A}\mathbf{x} - \mathbf{b}||_2^2$.

(2) Compute the gradient of $f_i(\mathbf{x}) = \frac{1}{2}(\mathbf{A}_i^t \mathbf{x} - \mathbf{b}_i)^2$ where $\mathbf{A}_i^t \in \mathbb{R}^{1 \times p}$ is row i of $\mathbf{A}$.

References

Dobson, A. J. and Barnett, A. G. (2008). *An introduction to generalized linear models*. Chapman and Hall/CRC.

Lange, K. (2016). *MM optimization algorithms*, volume 147. SIAM.