Math 6880/7875: Advanced Optimization
Descent algorithms, Part II

Akil Narayan[1]

[1]Department of Mathematics, and Scientific Computing and Imaging (SCI) Institute
University of Utah

Feburary 15, 2022

Descent algorithms

A foundational computational algorithmic idea for unconstrained + smooth optimization is a descent method.

Our focus for the second half of our discussion about descent methods is least squares problems.

- Least squares problems
- Large-scale gradient computations
- Acceleration strategies

Basics of least squares problems

(Nonlinear) least squares problems

Consider the unconstrained optimization,

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 ,$$

so that the objective is a sum-of-squares loss.

The residual vector $\boldsymbol{r}$ is typically a discrepancy between a model $y(c; x)$ and some available labeled data $\{(c_m, y_m)\}_{m=1}^M$:

$$\boldsymbol{r}(x) := (r_1(x), \ \ldots, \ r_M(x))^T , \qquad r_m(x) = y_m - y(c_m; x).$$

The prototypical example is linear least squares:

$$y(c; x) := \sum_{j=1}^n x_j \phi_j(c).$$

The general case occurs in pretty much any model fitting scenario, such as deep learning.

The residual vector

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 ,$$

First note the unremarkable expression for the gradient:

$$\nabla \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 = \nabla \boldsymbol{r}(x) \boldsymbol{r}(x), \quad \nabla \boldsymbol{r}(x) \in \mathbb{R}^{n \times M}$$

The residual vector

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 ,$$

First note the unremarkable expression for the gradient:

$$\nabla \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 = \nabla \boldsymbol{r}(x) \boldsymbol{r}(x), \quad \nabla \boldsymbol{r}(x) \in \mathbb{R}^{n \times M}$$

where $\nabla \boldsymbol{r}$ is the Jacobian of $\boldsymbol{r}$:

$$(\nabla \boldsymbol{r}(x))_{j,m} = \frac{\partial}{\partial x_j} r_m(x), \qquad \nabla \boldsymbol{r} = (\nabla r_1, \ \ldots, \ \nabla r_m) .$$

If we are in a data fitting scenario, with $r_m = y_m - y(c_m; x)$, then $\nabla r_j = -\nabla y(c_m; x)$, i.e., the required gradient is just the gradient of the model being fit.

The residual vector

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 ,$$

First note the unremarkable expression for the gradient:

$$\nabla \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 = \nabla \boldsymbol{r}(x) \boldsymbol{r}(x), \quad \nabla \boldsymbol{r}(x) \in \mathbb{R}^{n \times M}$$

The rather remarkable property is the form of the Hessian,

$$\nabla^2 \boldsymbol{r}(x) = \nabla \boldsymbol{r}(x) \nabla \boldsymbol{r}(x)^T + \sum_{m=1}^{M} r_m(x) \nabla^2 \boldsymbol{r}(x).$$

I.e., the first "part" of the Hessian is available simply from the gradient.

This begs the approximation $\nabla^2 \boldsymbol{r} \approx \nabla \boldsymbol{r} \nabla \boldsymbol{r}^T$ (plus perhaps some small corrections), which is the basis for several least squares algorithms.

Linear least squares

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2,$$

When $\boldsymbol{r}$ is affine in x, i.e., $\boldsymbol{r}(x) = \nabla \boldsymbol{r}^T x - a$ where $\nabla \boldsymbol{r}$ is independent of x, then the Hessian,

$$\nabla^2 \boldsymbol{r} = \nabla \boldsymbol{r} \nabla \boldsymbol{r}^T,$$

is positive semi-definite. The stationary points x are defined by,

$$\nabla \boldsymbol{r} \nabla \boldsymbol{r}^T x = \nabla \boldsymbol{r} a.$$

There is a unique stationary point if and only if $\nabla^2 \boldsymbol{r}(x) \succ 0$.

Otherwise there are infinitely many stationary points, and each minimizes the sum-of-squares residual.

In any case, the least squares solution above is equivalent to solving the linear equation

$$\nabla \boldsymbol{r}^T x = a,$$

in the least squares sense.

Solution methods for linear least squares

$$\nabla \boldsymbol{r} \nabla \boldsymbol{r}^T x = \nabla \boldsymbol{r} a.$$

$$\nabla \boldsymbol{r}^T x = a \text{ in least squares sense,}$$

There are (at least) three common ways to solve linear least squares problems. Assume initally that $\nabla^2 \boldsymbol{r} > 0$, implying that $\nabla \boldsymbol{r}^T$ is full-rank.

- The QR decomposition: with $\nabla \boldsymbol{r}^T = QR$ the reduced QR decomposition of $\nabla \boldsymbol{r}^T$, then the staionary point is

  $$x = R^{-1} Q^T a$$

  This method is typically the easiest and most stable.
- The SVD: with $\nabla \boldsymbol{r}^T = U \Sigma V^T$ the economy SVD of $\nabla \boldsymbol{r}^T$, then the stationary point is,

  $$x = V \Sigma^{-1} U^T a.$$

  This method is slightly more expensive than QR, but is more useful if more than the solution is desired.
- The Cholesky factorization: with $\nabla^2 \boldsymbol{r} = LL^T$ the Cholesky factorization of $\nabla^2 \boldsymbol{r}$, then the stationary point is,

  $$x = L^{-T} L^{-1} \nabla \boldsymbol{r} a$$

Solution methods for linear least squares

$$\nabla \boldsymbol{r} \nabla \boldsymbol{r}^T x = \nabla \boldsymbol{r} a.$$

$$\nabla \boldsymbol{r}^T x = a \text{ in least squares sense,}$$

There are (at least) three common ways to solve linear least squares problems. Assume initally that $\nabla^2 \boldsymbol{r} > 0$, implying that $\nabla \boldsymbol{r}^T$ is full-rank.

- The QR decomposition: with $\nabla \boldsymbol{r}^T = QR$ the reduced QR decomposition of $\nabla \boldsymbol{r}^T$, then the staionary point is

  $$x = R^{-1} Q^T a$$

  This method is typically the easiest and most stable.
- The SVD: with $\nabla \boldsymbol{r}^T = U \Sigma V^T$ the economy SVD of $\nabla \boldsymbol{r}^T$, then the stationary point is,

  $$x = V \Sigma^{-1} U^T a.$$

  This method is slightly more expensive than QR, but is more useful if more than the solution is desired.
- The Cholesky factorization: with $\nabla^2 \boldsymbol{r} = LL^T$ the Cholesky factorization of $\nabla^2 \boldsymbol{r}$, then the stationary point is,

  $$x = L^{-T} L^{-1} \nabla \boldsymbol{r} a$$

Solution methods for linear least squares

$$\nabla \boldsymbol{r} \nabla \boldsymbol{r}^T x = \nabla \boldsymbol{r} a. \qquad \text{"normal eqns"}$$

$$\nabla \boldsymbol{r}^T x = a \text{ in least squares sense,}$$

There are (at least) three common ways to solve linear least squares problems. Assume initally that $\nabla^2 \boldsymbol{r} > 0$, implying that $\nabla \boldsymbol{r}^T$ is full-rank.

- The QR decomposition: with $\nabla \boldsymbol{r}^T = QR$ the reduced QR decomposition of $\nabla \boldsymbol{r}^T$, then the staionary point is

  $$x = R^{-1} Q^T a$$

  This method is typically the easiest and most stable.
- The SVD: with $\nabla \boldsymbol{r}^T = U \Sigma V^T$ the economy SVD of $\nabla \boldsymbol{r}^T$, then the stationary point is,

  $$x = V \Sigma^{-1} U^T a.$$

  This method is slightly more expensive than QR, but is more useful if more than the solution is desired.
- The Cholesky factorization: with $\nabla^2 \boldsymbol{r} = LL^T$ the Cholesky factorization of $\nabla^2 \boldsymbol{r}$, then the stationary point is,

  $$x = L^{-T} L^{-1} \nabla \boldsymbol{r} a$$

Rank-deficient linear least squares, I

Before: $\nabla^2 r \succ 0$

$\nabla^2 r$

$$\nabla \boldsymbol{r} \nabla \boldsymbol{r}^T x = \nabla \boldsymbol{r} a.$$

$$\nabla \boldsymbol{r}^T x = a \text{ in least squares sense,}$$

Now assume $\nabla^2 \boldsymbol{r}$ is rank deficient (and hence $\nabla \boldsymbol{r}^T$ does not have full column rank).

There are infinitely many stationary points/solutions. Which to select?

One option is to select the "simplest" solution:

$$\min_x \|x\|_2^2 \quad \text{such that} \quad \frac{1}{2}\|\boldsymbol{r}(x)\|_2^2 \text{ is minimized.}$$

Such a solution is a *minimum norm* solution to the least squares problem.

Rank-deficient linear least squares, II

$(\nabla r^T x = a$ in LS sense$)$

$$\min_x \|x\|_2^2 \quad \text{such that} \quad \frac{1}{2}\|\boldsymbol{r}(x)\|_2^2 \text{ is minimized.}$$

We assume $\nabla^2 \boldsymbol{r}$ is rank deficient, and hence $\operatorname{rank} \nabla \boldsymbol{r}^T = s < n$.

Again, linear algebra comes to the rescue. There are two ways to compute this solution:

- The QR decomposition: with the reduced QR decomposition,

  $$\nabla \boldsymbol{r}^T = QR, \qquad Q \in \mathbb{R}^{m\times s},\ R \in \mathbb{R}^{s\times s}$$

  then $x = R^{-1}Q^T a$ is the minimum norm solution.
- The SVD: with the reduced SVD,

  $$\nabla \boldsymbol{r}^T = U\Sigma V^T, \qquad U \in m \times s,\ \Sigma \in \mathbb{R}^{s\times s},\ V \in \mathbb{R}^{n\times s},$$

  then

  $$x = V\Sigma^{-1}U^T a =: (\nabla \boldsymbol{r}^T)^+ a,$$

  is the minimum norm solution. The matrix $(\nabla \boldsymbol{r}^T)^+$ is the *Moore-Penrose* pseudoinverse of $\nabla \boldsymbol{r}^T$.

Rank-deficient linear least squares, II

$$\min_x \|x\|_2^2 \quad \text{such that} \quad \frac{1}{2}\|\boldsymbol{r}(x)\|_2^2 \text{ is minimized.}$$

We assume $\nabla^2 \boldsymbol{r}$ is rank deficient, and hence $\operatorname{rank} \nabla \boldsymbol{r}^T = s < n$.

Again, linear algebra comes to the rescue. There are two ways to compute this solution:

- The QR decomposition: with the reduced QR decomposition,

  $$\nabla \boldsymbol{r}^T = QR, \qquad Q \in \mathbb{R}^{m\times s},\ R \in \mathbb{R}^{s\times s}$$

  $Q^TQ = I_{s\times s}$

  $QQ^T \neq I_{m\times m}$

  then $x = R^{-1}Q^T a$ is the minimum norm solution.
- The SVD: with the reduced SVD,

  $$\nabla \boldsymbol{r}^T = U\Sigma V^T, \qquad U \in \mathbb{R}^{m\times s},\ \Sigma \in \mathbb{R}^{s\times s},\ V \in \mathbb{R}^{n\times s},$$

  $\mathbb{R}^{m\times s}$

  then

  $$x = V\Sigma^{-1}U^T a =: (\nabla \boldsymbol{r}^T)^+ a,$$

  is the minimum norm solution. The matrix $(\nabla \boldsymbol{r}^T)^+$ is the *Moore-Penrose* pseudoinverse of $\nabla \boldsymbol{r}^T$.

Back to nonlinear problems: Gauss-Newton

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 ,$$

with

$$\boldsymbol{r}(x) := (r_1(x), \ \ldots, \ r_M(x))^T , \qquad r_m(x) = y_m - y(c_m; x),$$

where now we assume f is not affine in x.

The Gauss-Newton method uses an approximate Hessian to perform a Newton-type update. The approximation employed is,

$$\nabla^2 \boldsymbol{r}(x) = \nabla \boldsymbol{r}(x) \nabla \boldsymbol{r}(x)^T + \sum_{m=1}^{M} r_m(x) \nabla^2 \boldsymbol{r}(x) \approx \nabla \boldsymbol{r}(x) \nabla \boldsymbol{r}(x)^T,$$

The corresponding update uses a stepsize α_k, acknowledging that the Newton direction is approximate:

$$x_{k+1} = x_k - \alpha_k d_k, \qquad d_k = -\left(\nabla \boldsymbol{r}(x_k) \nabla \boldsymbol{r}(x_k)^T \right)^{-1} \nabla \boldsymbol{r}(x_k) \boldsymbol{r}(x_k)$$

where α_k is typically chosen via the Armijo/Wolfe/Goldstein conditions.

Back to nonlinear problems: Gauss-Newton

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 ,$$

with

$$\boldsymbol{r}(x) := (r_1(x), \ \ldots, \ r_M(x))^T , \qquad r_m(x) = y_m - y(c_m; x),$$

where now we assume f is not affine in x.

The Gauss-Newton method uses an approximate Hessian to perform a Newton-type update. The approximation employed is,

$$\nabla^2 \boldsymbol{r}(x) = \nabla \boldsymbol{r}(x) \nabla \boldsymbol{r}(x)^T + \sum_{m=1}^{M} r_m(x) \nabla^2 \boldsymbol{r}_m(x) \approx \nabla \boldsymbol{r}(x) \nabla \boldsymbol{r}(x)^T,$$

The corresponding update uses a stepsize α_k, acknowledging that the Newton direction is approximate:

$$x_{k+1} = x_k + \alpha_k d_k, \qquad d_k = -\left(\nabla \boldsymbol{r}(x_k) \nabla \boldsymbol{r}(x_k)^T \right)^{-1} \nabla \boldsymbol{r}(x_k) \boldsymbol{r}(x_k)$$

where α_k is typically chosen via the Armijo/Wolfe/Goldstein conditions.

Gauss-Newton as "local" linear least squares

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} \|\boldsymbol{r}(x)\|_2^2 ,$$

with

$$\boldsymbol{r}(x) := (r_1(x), \ \ldots, \ r_M(x))^T , \qquad r_m(x) = y_m - y(c_m; x),$$

where now we assume y is not affine in x.

Gauss-Newton is equivalently a linear least squares problem, where $\boldsymbol{r}$ is approximated by its linear approximation at step x_k:

$$\boldsymbol{r}(x) \approx r_k(x) := \boldsymbol{r}(x_k) + (\nabla \boldsymbol{r}(x_k))^T (x - x_k).$$

Using this approximation, one could exactly solve, obtaining

$$x_{k+1} = \arg\min_x \|r_k(x)\|_2^2 = x_k - \left(\nabla \boldsymbol{r}(x_k) \nabla \boldsymbol{r}(x_k)^T \right)^{-1} \nabla \boldsymbol{r}(x_k) \boldsymbol{r}(x_k).$$

But recognizing that this is an approximation, we introduce a stepsize:

$$x_{k+1} = x_k - \alpha_k \left(\nabla \boldsymbol{r}(x_k) \nabla \boldsymbol{r}(x_k)^T \right)^{-1} \nabla \boldsymbol{r}(x_k) \boldsymbol{r}(x_k).$$

Advantages of Gauss-Newton

The Gauss-Newton algorithm is an attractive one:

- Only gradients/Jacobians are required to compute an approximate Hessian, and this is often a fairly good approximation.
- The Jacobian can be sparse in large-scale applications
- The direction for update is actually a descent direction:

$$d_k^T \nabla \left(\frac{1}{2} \left\| \boldsymbol{r} \right\|_2^2 \right) = d_k^T (\nabla \boldsymbol{r}(x_k)) \boldsymbol{r}(x_k) = -d_k^T \left(\nabla \boldsymbol{r}(x_k) \nabla \boldsymbol{r}(x_k)^T \right) d_k \leqslant 0$$

Gauss-Newton has weaker convergence guarantees than some of the previous methods we have seen.

One weakness is that if the Jacobian $\nabla \boldsymbol{r}$ is near-singular, the step lengths α_k become very small.

The Levenberg-Marquardt method

The Levenberg-Marquardt method is, in one interpretation, a trust region analogue of Gauss-Newton:

We solve the linearized problem in a trust region:

$$\min_d \frac{1}{2}\|\nabla \boldsymbol{r}(x_k)^T d + \boldsymbol{r}(x_k)\|_2^2, \qquad \text{subject to } \|d\| \leqslant \Delta_k.$$

By KKT, the solution to this problem is a (d, λ) satisfying,

$$\left(\nabla \boldsymbol{r}(x_k)\nabla \boldsymbol{r}(x_k)^T + \lambda I\right) d = -\nabla \boldsymbol{r}(x_k)\boldsymbol{r}(x_k), \leftarrow \text{stationarity of augmented Lagrangian}$$

$$\lambda(\|d\| - \Delta_k) = 0, \longrightarrow \text{complementary slackness}$$

I.e., unless the Gauss-Newton algorithm descent point satisfies $\|d\| \leqslant \Delta$, then we solve the *regularized* Gauss-Newton problem,

$$\left(\nabla \boldsymbol{r}(x_k)\nabla \boldsymbol{r}(x_k)^T + \lambda I\right) d = -\nabla \boldsymbol{r}(x_k)\boldsymbol{r}(x_k),$$

where λ is computed approximately.

The Levenberg-Marquardt method

The Levenberg-Marquardt method is, in one interpretation, a trust region analogue of Gauss-Newton:

We solve the linearized problem in a trust region:

$$\min_d \frac{1}{2}\|\nabla \boldsymbol{r}(x_k)^T d + \boldsymbol{r}(x_k)\|_2^2, \qquad \text{subject to } \|d\| \leqslant \Delta_k.$$

By KKT, the solution to this problem is a (d, λ) satisfying,

$$\left(\nabla \boldsymbol{r}(x_k)\nabla \boldsymbol{r}(x_k)^T + \lambda I\right) d = -\nabla \boldsymbol{r}(x_k)\boldsymbol{r}(x_k),$$
$$\lambda(\|d\| - \Delta_k) = 0,$$

I.e., unless the Gauss-Newton algorithm descent point satisfies $\|d\| \leqslant \Delta$, then we solve the *regularized* Gauss-Newton problem,

$$\left(\nabla \boldsymbol{r}(x_k)\nabla \boldsymbol{r}(x_k)^T + \lambda I\right) d = -\nabla \boldsymbol{r}(x_k)\boldsymbol{r}(x_k),$$

where λ is computed approximately.

GN and LM

The Levenberg-Marquardt method ameliorates some of the small step sizes taken by the Gauss-Newton method when the gradient/Jacobian is ill-conditioned.

When the iterates get "close" to a local minimum, the Levenberg-Marquardt method reduces to Gauss-Newton. (The trust region constraint is inactive.)

If the local minimum x_* satisfies $\nabla^2 \boldsymbol{r}(x_*) \approx \nabla \boldsymbol{r}(x_*) \nabla \boldsymbol{r}(x_*)^T$, then both methods have quadratic convergence behavior near the minimum.

Large least squares problems

Large least squares

E.g., in machine learning, large least squares problems arise given data $(c_m, y_m)_{m\in[M]}$,

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}\|\boldsymbol{r}(x)\|_2^2, \qquad r_m(x) = y_m - y(c_m; x),$$

which we'll rewrite in a more natural form for this discussion:

$$\min_{x\in\mathbb{R}^n} f(x), \qquad f(x) = \sum_{m\in[M]} f_m(x)$$

$f_m(x) = \frac{1}{2}(y_m - y(c_m; x))^2$

We are interested in the case when $M \gg 1$.

We've seen that computing the gradient,

$$\nabla f(x) = \sum_{m\in[M]} \nabla f_m(x),$$

is a necessary component of several algorithms. When $M \gg 1$, there are algorithmic strategies to make this more efficient.

Stochastic gradient descent

$$\min_{x \in \mathbb{R}^n} \sum_{m \in [M]} f_m(x)$$

Computing the full ("batch") gradient can be approximated by a single component,

$$\sum_{m \in [M]} \nabla f_m(x) \approx \nabla f_i(x), \qquad i \in [M],$$

and then $\nabla f_i(x)$ is used as a gradient in optimization.

This is stochastic/incremental gradient descent. Two strategies to compute i:

- *Cyclic*: On iteration k, choose $i = 1 + (k - 1 \mod M)$.
  Typically also randomly shuffle f_i after each cycle.
- *Randomized*: Choose $i \in [M]$ uniformly at random. This strategy is appealing since,

$$\mathbb{E} \nabla f_i(x) = \frac{1}{M} \nabla f.$$

  I.e., this produces an *unbiased* statistical estimator for ∇f.

Stochastic gradient descent

$$\min_{x \in \mathbb{R}^n} \sum_{m \in [M]} f_m(x)$$

Computing the full ("batch") gradient can be approximated by a single component,

$$\sum_{m \in [M]} \nabla f_m(x) \approx \nabla f_i(x), \qquad i \in [M],$$

and then $\nabla f_i(x)$ is used as a gradient in optimization.

This is stochastic/incremental gradient descent. Two strategies to compute i:

- *Cyclic*: On iteration k, choose $i = 1 + (k - 1 \mod M)$.
  Typically also randomly shuffle f_i after each cycle.
- *Randomized*: Choose $i \in [M]$ uniformly at random. This strategy is appealing since,

  over the random draw of i

  $$\mathbb{E} \nabla f_i(x) = \frac{1}{M} \nabla f.$$

  I.e., this produces an *unbiased* statistical estimator for ∇f.

Stepsizes

$$\min_{x \in \mathbb{R}^n} \sum_{m \in [M]} f_m(x),$$

Assuming a vanilla stochastic gradient descent update,

$$x_{k+1} = x_k - \alpha_k \nabla f_i(x_k),$$

$i = i(k)$

it remains to choose α_k.

Even for very nice problems, SGD may not converge without a controlled decay of α_k.

E.g.,

$$M = 2, \qquad f_1(x) = (x-1)^2, \qquad f_2(x) = (x+1)^2,$$

with α_k chosen via exact linesearch. With $x_0 = 0.5$ and no termination, the SGD iterates satisfy,

$$-2 = \liminf_k (x_{k+1} - x_k) < \limsup_k (x_{k+1} - x_k) = +2, \quad \text{with probability 1,}$$

$$\lim_{k \uparrow \infty} f(x_k) = 4 > \min_x f(x) = 2$$

Stepsizes

$$\min_{x \in \mathbb{R}^n} \sum_{m \in [M]} f_m(x),$$

Assuming a vanilla stochastic gradient descent update,

$$x_{k+1} = x_k - \alpha_k \nabla f_i(x_k),$$

it remains to choose α_k.

Even for very nice problems, SGD may not converge without a controlled decay of α_k.

E.g., $f'(x) = 2x^2 + 2$

$n = 1$ $\quad M = 2, \qquad f_1(x) = (x-1)^2, \qquad f_2(x) = (x+1)^2,$

with α_k chosen via exact linesearch. With $x_0 = 0.5$ and no termination, the SGD iterates satisfy,

$$-2 = \liminf_k (x_{k+1} - x_k) < \limsup_k (x_{k+1} - x_k) = +2, \quad \text{with probability 1,}$$

$$\lim_{k \uparrow \infty} f(x_k) = 4 > \min_x f(x) = 2$$

Stepsizes

The stepsizes α_k are therefore not necessarily chosen via classical stepsize criteria (e.g., Wolfe, Goldstein), but are instead chosen with some decaying behavior, e.g.,

$$\alpha_k = \frac{1}{k}.$$

In order to prove convergence, the stepsizes α_k must be chosen so that

- they do decay: $\alpha_k \xrightarrow{k\to\infty} 0$
- they do not decay too quickly: $\sum_k \alpha_k = \infty$

Convergence

Typically, if

- f has bounded gradient and is convex
- stepsizes α_k are chosen to decay appropriately

Then if x_* is a local minimum, for SGD one obtains

$$\mathbb{E} f(x_k) - f(x_*) = \mathcal{O}(1/\sqrt{k}).$$

Note that for such fixed stepsizes with *standard* gradient descent, we have

$$f(x_k) - f(x_*) = \mathcal{O}(1/k).$$

Hence, SGD converges slower than vanilla gradient descent.

Mini-batches

GD and SGD live on opposite ends of the spectrum in terms of data usage:

- GD uses all the data in one iteration
- SGD uses a single piece of data in one iteration

Mini-batch SGD uses $P \ll M$ pieces of data:

$$\min_{x \in \mathbb{R}^n} \sum_{m \in [M]} f_m(x)$$

$$\sum_{m \in [M]} \nabla f_m(x) \approx \sum_{m \in S} \nabla f_m(x), \quad S \subset [M],\ |S| = P \ll M$$

This attempts to mitigate randomized oscillations of SGD, but requires more computational effort and memory management.

Mini-batch SGD generally does not improve convergence characteristics of SGD.

Stochastic gradient descent

SGD is attractive because,

- it is very efficient (in memory and computation)
- approximate solutions are often good enough in practice
- theory exists and guarantees convergence, although there are still many open questions.

Using SGD comes with some drawbacks:

- convergence is slower, more fickle, than GD
- SGD because well away from minimizers, but performs relatively poorly close to them
- in much (not all) practice, one typically uses fixed stepsizes, contrary to theory

SGD is often paired with other approaches to improve convergence.

- Acceleration
- Adaptive stepsizes

Stochastic gradient descent

SGD is attractive because,

- it is very efficient (in memory and computation)
- approximate solutions are often good enough in practice
- theory exists and guarantees convergence, although there are still many open questions.

Using SGD comes with some drawbacks:

- convergence is slower, more fickle, than GD
- SGD ~~because~~ performs well away from minimizers, but performs relatively poorly close to them
- in much (not all) practice, one ~~typically~~ frequently uses fixed stepsizes, contrary to theory

SGD is often paired with other approaches to improve convergence.

- Acceleration
- Adaptive stepsizes

Stochastic gradient descent

SGD is attractive because,

- it is very efficient (in memory and computation)
- approximate solutions are often good enough in practice
- theory exists and guarantees convergence, although there are still many open questions.

Using SGD comes with some drawbacks:

- convergence is slower, more fickle, than GD
- SGD because well away from minimizers, but performs relatively poorly close to them
- in much (not all) practice, one typically uses fixed stepsizes, contrary to theory

SGD is often paired with other approaches to improve convergence.

- Acceleration
- Adaptive stepsizes

Acceleration techniques

Acceleration techniques

Acceleration techniques attempt to improve convergence of gradient descent methods. They frequently attempt to address noisy gradients and poorly scaled problems.

Momentum (and variants thereof) is perhaps the most widely used acceleration technique.

The basic idea is as follows: while we take discrete steps with gradient descent,

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k),$$

as $\alpha_k \downarrow 0$, this can be interpreted as gradient flow, a differential equation limit:

$$\frac{\mathrm{d}}{\mathrm{d}t} x(t) = -\nabla f(x),$$

where f is now intepreted as a potential function.

When f is non-smooth, the trajectory oscillates, much like a poorly scaled gradient descent.

The idea to "fix" this is to impart physical momentum, or a type of viscous drag, to the problem.

Acceleration techniques

Acceleration techniques attempt to improve convergence of gradient descent methods. They frequently attempt to address noisy gradients and poorly scaled problems.

Momentum (and variants thereof) is perhaps the most widely used acceleration technique.

The basic idea is as follows: while we take discrete steps with gradient descent,

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k),$$

as $\alpha_k \downarrow 0$, this can be interpreted as gradient flow, a differential equation limit:

$$\frac{\mathrm{d}}{\mathrm{d}t} x(t) = -\nabla f(x),$$

where f is now intepreted as a potential function.

When f is non-smooth, the trajectory oscillates, much like a poorly scaled gradient descent.

The idea to "fix" this is to impart physical momentum, or a type of viscous drag, to the problem.

Momentum gradient descent, I

We therefore rewrite our system as a second-order system:

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}x + a\frac{\mathrm{d}}{\mathrm{d}t}x = -\nabla f,$$

which follows the force produced by the potential f, subject to a viscous drag $a > 0$.

Rewriting this in a system of first-order equations, we introduce a velocity $v = \frac{\mathrm{d}}{\mathrm{d}t}x$, obtaining:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}x &= v\\ \frac{\mathrm{d}}{\mathrm{d}t}v &= -\alpha v - \nabla f.\end{aligned}$$

This system is triangular, so we will use a *semi-implicit* scheme to perform updates.

Momentum gradient descent, II

$$\frac{\mathrm{d}}{\mathrm{d}t}x = v \tag{1a}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}v = -\alpha v - \nabla f. \tag{1b}$$

If we are able to update v_{k+1}, we update x_k using the new value:

$$x_{k+1} = x_k + (\Delta t) v_{k+1}.$$

We update v_{k+1} using an *exponential time integrator*:

$$v_{k+1} = e^{-\alpha \Delta t} v_k - \Delta t \nabla f(x_k).$$

Why use semi-implicit and exponential time integrators? The system (1) can be *stiff*, and these techniques are standard numerical procedures to mitigate stiffness.

With appropriate rescaling, the momentum update is given by,

$$v_{k+1} = \alpha v_k - \eta \nabla f(x_k),$$

$$x_{k+1} = x_k + v_{k+1}.$$

Momentum gradient descent, II

$$\frac{\mathrm{d}}{\mathrm{d}t}x = v \tag{1a}$$
$$\frac{\mathrm{d}}{\mathrm{d}t}v = -\alpha v - \nabla f. \tag{1b}$$

If we are able to update v_{k+1}, we update x_k using the new value:

$$x_{k+1} = x_k + (\Delta t) v_{k+1}.$$

We update v_{k+1} using an *exponential time integrator*:

$$v_{k+1} = e^{-\alpha \Delta t} v_k - \Delta t \nabla f(x_k).$$

Why use semi-implicit and exponential time integrators? The system (1) can be *stiff*, and these techniques are standard numerical procedures to mitigate stiffness.

With appropriate rescaling, the momentum update is given by,

$$v_{k+1} = \alpha v_k - \eta \nabla f(x_k),$$
$$x_{k+1} = x_k + v_{k+1}.$$

Momentum gradient descent, III

$$v_{k+1} = \alpha v_k - \eta \nabla f(x_k),$$
$$x_{k+1} = x_k + v_{k+1}.$$

Note that since $\alpha = \exp(-a\Delta t)$, we choose $\alpha \in (0, 1)$.
The "Δt" that was in front of v_{k+1} can be omitted by rescaling η and α.

Empirically, choosing η appropriately has a bigger impact on convergence than the choice of α.

In line with previously stated convergence guarantees of SGD, one typically also imposes a decrease on η in time.

A common interpretation of momentum GD is as a particle subject to drag due to momentum.

The decay parameter α

$$v_{k+1} = \alpha v_k - \eta \nabla f(x_k),$$
$$x_{k+1} = x_k + v_{k+1}.$$

The effect of the parameter α can be understood via a simple scenario:

Assume zero initial velocity $\nabla f(x_k)$ is constant in k, say $y = \nabla f(x_k)$. Then a direct calculation shows:

$$v_k = -\eta y \sum_{j=0}^{k-1} \alpha^j \quad \Longrightarrow \quad \lim_{k \uparrow \infty} \|v_k\|_2 = \frac{\eta}{1-\alpha} \|y\|_2,$$

i.e., the terminal velocity of x is proportional to $1/(1-\alpha)$, giving guidance on how to set α.

Nesterov's accelerated GD

A somewhat different scheme for accelerating GD is due to Nesterov. The update scheme is as follows:

$$y_{k+1} = x_k - \eta \nabla f(x_k),$$
$$x_{k+1} = (1-\gamma_k) x_k + \gamma_k y_{k+1},$$

where η is tunable, and γ_k is a sequence defined as,

$$\gamma_k = \frac{1-\lambda_k}{\lambda_{k+1}} \leqslant 0, \qquad \lambda_{k+1} = \frac{1+\sqrt{1+4\lambda_k^2}}{2},$$

with $\lambda_0 = 0$.

What is surprising about this scheme is that (a) the x_{k+1} update is a bit unexpected since $\gamma_k \leqslant 0$, and (b) the method converges like $1/k^2$. (Unaccelerated convergence is $\sim 1/k$.)

This method as written is not tooo transparent. But it turns out this is equivalent to a type of momentum approach.

Nesterov's accelerated GD

A somewhat different scheme for accelerating GD is due to Nesterov. The update scheme is as follows:

$$y_{k+1} = x_k - \eta \nabla f(x_k),$$
$$x_{k+1} = (1-\gamma_k) x_k + \gamma_k y_{k+1},$$

where η is tunable, and γ_k is a sequence defined as,

$$\gamma_k = \frac{1-\lambda_k}{\lambda_{k+1}} \leqslant 0, \qquad \lambda_{k+1} = \frac{1+\sqrt{1+4\lambda_k^2}}{2},$$

with $\lambda_0 = 0$.

What is surprising about this scheme is that (a) the x_{k+1} update is a bit unexpected since $\gamma_k \leqslant 0$, and (b) the method converges like $1/k^2$. (Unaccelerated convergence is $\sim 1/k$.)

This method as written is not too transparent. But it turns out this is equivalent to a type of momentum approach.

Nesterov's accelerated GD as momentum

The Nesterov accelerated GD algorithm can actually be written as a type of momentum:

$$v_{k+1} = \alpha v_k - \eta \nabla f(x_k + \alpha v_k),$$
$$x_{k+1} = x_k + v_{k+1}.$$

The only difference between this and the "classical" momentum technique is where the gradient is evaluated: Nesterov's approach is somewhat of a predictor-corrector scheme, using gradients computed from taking a particular step.

This seemingly small change to momentum can make a big difference in practice.

Adaptive learning rate techniques

Adaptive learning rates

Acceleration strategies for optimization work, but only so much improvement can be achieved since empirically one observes that the learning rate is often the issue.

Even more problematic is that the update,

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k),$$

imposes an identical learning rate α_k on all components of x_k.

In practice, different components of α_k have different scaling and sensitivity to learning rate.

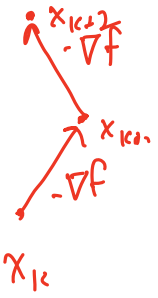
Thus, it makes sense to do something more like,

$$x_{k+1} = x_k - \boldsymbol{\alpha}_k \odot \nabla f(x_k), \qquad \boldsymbol{\alpha}_k \in \mathbb{R}^n,$$

where $\odot$ is the elementwise product between vectors.

This is the basis of *adaptive* methods.

A simple adaptation



Let us use the adaptive update,

$$x_{k+1} = x_k - \boldsymbol{\alpha}_k \odot \nabla f(x_k).$$

Here is one idea that works reasonably well:
We update the jth component of $\boldsymbol{\alpha}_k$ as follows:

- If $\operatorname{sign}(\nabla f(x_k))_j = \operatorname{sign}(\nabla f(x_{k+1}))_j$, component j is updating in a consistent way across k: make $(\alpha_k)_j$ larger.
- If $\operatorname{sign}(\nabla f(x_k))_j \neq \operatorname{sign}(\nabla f(x_{k+1}))_j$, component j is updating in an oscillatory fashion: make $(\alpha_k)_j$ smaller.

The above is called the *delta-bar-delta* algorithm, and works reasonably well.

The problem is that it typically works well only for full (batch) GD approaches.

Adagrad

A modern slew of adaptive learning techniques starts with an Adaptive Gradient, or Adagrad method.

Idea: use history of gradients

- Components j with large gradient history should have small learning rates
- Components j with small gradient history should have large learning rates

I.e., this attempts to mitigate poorly scaled convex problems. In details, we initialize a gradient norm history $h_0 = 0$, and iterate:

$$\begin{aligned} g_k &= \nabla f(x_k) \\ h_k &= h_{k-1} + g_k \odot g_k, \\ \boldsymbol{\alpha}_k &= \frac{\eta}{\delta + \sqrt{h_k}}, \\ x_{k+1} &= x_k - \boldsymbol{\alpha}_k g_k, \end{aligned}$$

history → h_k ← gradient norm

where η is a fixed (initial) learning rate, and $\delta > 0$ is a small factor to prevent divide-by-zero.

Adagrad is very successful for convex (or locally convex) problems where the gradient behavior does not change wildly.

It is less successful for non-convex problems.

Adagrad

A modern slew of adaptive learning techniques starts with an Adaptive Gradient, or Adagrad method.

Idea: use history of gradients

- Components j with large gradient history should have small learning rates
- Components j with small gradient history should have large learning rates

I.e., this attempts to mitigate poorly scaled convex problems. In details, we initialize a gradient norm history $h_0 = 0$, and iterate:

$$\begin{aligned} g_k &= \nabla f(x_k) \\ h_k &= h_{k-1} + g_k \odot g_k, \\ \boldsymbol{\alpha}_k &= \frac{\eta}{\delta + \sqrt{h_k}}, \\ x_{k+1} &= x_k - \boldsymbol{\alpha}_k g_k, \end{aligned}$$

where η is a fixed (initial) learning rate, and $\delta > 0$ is a small factor to prevent divide-by-zero.

Adagrad is very successful for convex (or locally convex) problems where the gradient behavior does not change wildly.

It is less successful for non-convex problems.

RMSProp

RMSProp was developed to mitigate problems with keeping the gradient norm history. The main innovation is essentially to use momentum decay to decrease long-time gradient norms:

$$h_k = \rho h_{k-1} + (1-\rho)(g_k \odot g_k),$$

where $\rho \in (0,1)$ is a momenutm decay parameter. The first term essentially imposes the exponential decay,

$$\frac{\mathrm{d}}{\mathrm{d}t} h = (\log \rho) h,$$

which forces decay of h.

This imposes and exponential decay on the history of gradient norms, mitigating issues with non-convex optimization when the gradient behaves differently at the start of optimization compared to the end.

Adadelta, I

The Adadelta method also aims to improve the long history weakness of Adagrad for non-convex problems.

Adadelta makes two innovations:

- Imposes momentum-based exponential decay on the gradient norms
- Uses "correct" units in the update

The first innovation is essentially RMSProp: with $\rho \in (0, 1)$ a momentum decay parameter, update the gradient norm history as,

$$h_k = \rho h_{k-1} + (1 - \rho)(g_k \odot g_k).$$

Adadelta, II

The second innovation Adadelta employs is the following: if we simply used our new gradient norm history, we would do,

$$\boldsymbol{\alpha}_k = \frac{\eta}{\delta + \sqrt{h_k}},$$
$$x_{k+1} = x_k - \boldsymbol{\alpha}_k g_k.$$

The Adadelta philosophy is that the highlighted term has the wrong units: it's dimensionless.

To fix this, change the units of η to be the units of x in a "smart" way:

$$\boldsymbol{X}_k = \rho \boldsymbol{X}_{k-1} + \eta^2 \left(\underbrace{(\boldsymbol{\alpha}_k \odot g_k)}_{\text{“}\Delta x_k''} \odot (\boldsymbol{\alpha}_k \odot g_k) \right),$$
$$\eta \leftarrow \sqrt{\delta + \boldsymbol{X}_k \odot \boldsymbol{X}_k}.$$

Thus, Adagrad simultaneously imposes strong decay on gradient norms, and changes the learning to be unit-consistent.

Adam

The last adaptation strategy we'll discuss is one with *adaptive moments*, or Adam.

Adam keeps a decaying history of both the second (uncentered moment) h, and the first moment m:

$$h_k = \rho h_{k-1} + (1-\rho)(g_k \odot g_k), \qquad m_k = \tilde{\rho} m_{k-1} + (1-\tilde{\rho}) g_k,$$

where m_k is interpretable as a momentum.

Using initializations $h_0 = m_0 = 0$, one can show that these are 0-biased estimates of these moments, which can be corrected via,

$$\tilde{h}_k = \frac{h_k}{1-\rho^k}, \qquad \tilde{m}_k = \frac{m_k}{1-\tilde{\rho}^k}.$$

The full update is then,

$$x_{k+1} = x_k - \frac{\eta}{\delta + \sqrt{\tilde{h}_k}} \tilde{m}_k.$$

Adam empirically is a bit robust in selection of η, δ.

Acceleration and adaptation

It is common to use a combination of acceleration and adaptation strategies, e.g., RMSProp and Nesterov momentum are often used together.

Most of these algorithms are written with the full batch gradient, but are really designed to work with mini-batch or stochastic GD versions to exploit efficiency.

Which algorithm(s) to use? There isn't really a consensus, and this typically boils down to specifics of a problem. E.g., Adam is used frequently at first due to its hyperparameter robustness.

References I


Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng, *Large Scale Distributed Deep Networks*, Advances in Neural Information Processing Systems, vol. 25, Curran Associates, Inc., 2012.


John Duchi, Elad Hazan, and Yoram Singer, *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, Journal of Machine Learning Research **12** (2011), no. 61, 2121–2159.


Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press, 2016.


Robert A. Jacobs, *Increased rates of convergence through learning rate adaptation*, Neural Networks **1** (1988), no. 4, 295–307.


Diederik P. Kingma and Jimmy Ba, *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980 [cs] (2017), arXiv: 1412.6980.


A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro, *Robust Stochastic Approximation Approach to Stochastic Programming*, SIAM Journal on Optimization **19** (2009), no. 4, 1574–1609.

References II

- Y. Nesterov, *Introductory Lectures on Convex Optimization: A Basic Course*, Springer Science & Business Media, 2013.
- Yurii E Nesterov, *A method for solving the convex programming problem with convergence rate O (1/k^ 2)*, Dokl. akad. nauk Sssr, vol. 269, 1983, pp. 543–547.
- Jorge Nocedal and S. Wright, *Numerical Optimization*, 2 ed., Springer Series in Operations Research and Financial Engineering, Springer-Verlag, New York, 2006.
- B. T. Polyak, *Some methods of speeding up the convergence of iteration methods*, USSR Computational Mathematics and Mathematical Physics **4** (1964), no. 5, 1–17.
- James C. Spall, *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*, John Wiley & Sons, March 2005.
- Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton, *On the importance of initialization and momentum in deep learning*, Proceedings of the 30th International Conference on Machine Learning, PMLR, 2013, ISSN: 1938-7228, pp. 1139–1147.

References III


Matthew D. Zeiler, *ADADELTA: An Adaptive Learning Rate Method*, arXiv:1212.5701 [cs] (2012), arXiv: 1212.5701.