FROM "LINEAR ALGEBRA ... Data"

4 344 G. STRANG (2019)

VI.4 Gradient Descent Toward the Minimum

This section of the book is about a fundamental problem: **Minimize a function** $f(x_1, \ldots, x_n)$. Calculus teaches us that all the first derivatives $\partial f/\partial x_i$ are zero at the minimum (when f is smooth). If we have $n = 20$ unknowns (a small number in deep learning) then minimizing one function f produces 20 equations $\partial f/\partial x_i = 0$. *"Gradient descent" uses the derivatives $\partial f/\partial x_i$ to find a direction that reduces $f(x)$.* The steepest direction, in which $f(x)$ decreases fastest, is given by the gradient $-\nabla f$:

$$\boxed{\textbf{Gradient descent} \qquad x_{k+1} = x_k - s_k \nabla f(x_k)} \tag{1}$$

The symbol ∇f represents the vector of n partial derivatives of f: its **gradient**. So (1) is a vector equation for each step $k = 1, 2, 3, \ldots$ and s_k is the *stepsize* or the *learning rate*. We hope to move toward the point x^* where the graph of $f(x)$ hits bottom.

We are willing to assume for now that 20 first derivatives exist and can be computed. We are not willing to assume that those 20 functions also have 20 convenient derivatives $\partial/\partial x_j(\partial f/\partial x_i)$. Those are the 210 **second derivatives** of f—which go into a 20 by 20 symmetric matrix H. (Symmetry reduces $n^2 = 400$ to $\frac{1}{2}n^2 + \frac{1}{2}n = 210$ computations.) The second derivatives would be very useful extra information, but in many problems we have to go without.

You should know that 20 first derivatives and 210 second derivatives don't multiply the computing cost by 20 and 210. The neat idea of **automatic differentiation**—rediscovered and extended as **backpropagation** in machine learning—makes those cost factors much smaller in practice. This idea is described in Section VII.2.

Return for a moment to equation (1). The step $-s_k \nabla f(x_k)$ includes a minus sign (to descend) and a factor s_k (to control the the stepsize) and the gradient vector ∇f (containing the first derivatives of f computed at the current point x_k). A lot of thought and computational experience has gone into the choice of stepsize and search direction.

We start with the main facts from calculus about derivatives and gradient vectors ∇f.

The Derivative of $f(x)$: $n = 1$

The derivative of $f(x)$ involves a *limit*—this is the key difference between calculus and algebra. We are comparing the values of f at two nearby points x and $x + \Delta x$, as Δx approaches zero. More accurately, we are watching the slope $\Delta f/\Delta x$ between two points on the graph of $f(x)$:

$$\textbf{Derivative of } f \textbf{ at } x \qquad \frac{df}{dx} = \textbf{limit of } \frac{\Delta f}{\Delta x} = \textbf{limit of } \left[\frac{f(x + \Delta x) - f(x)}{\Delta x}\right]. \tag{2}$$

This is a forward difference when Δx is positive and a backward difference when $\Delta x < 0$. When we have the same limit from both sides, that number is the slope of the graph at x.

The ramp function $\text{ReLU}(x) = f(x) = \max(0, x)$ is heavily involved in deep learning (see VII.1). It has unequal slopes **1** to the right and **0** to the left of $x = 0$. So the derivative df/dx does not exist at that corner point in the graph. For $n = 1$, df/dx is the gradient ∇f.

$$\text{ReLU} = \begin{matrix} x \text{ for } x \geq 0 \\ 0 \text{ for } x \leq 0 \end{matrix} \qquad \textbf{slope } \frac{\Delta f}{\Delta x} = \frac{f(0+\Delta x) - f(0)}{\Delta x} = \begin{matrix} \Delta x/\Delta x = 1 \text{ if } \Delta x > 0 \\ 0/\Delta x = 0 \text{ if } \Delta x < 0 \end{matrix}$$

For the smooth function $f(x) = x^2$, the ratio $\Delta f/\Delta x$ will safely approach the derivative df/dx from both sides. But the approach could be slow (just first order). Look again at the point $x = 0$, where the true derivative $df/dx = 2x$ is now zero:

$$\text{The ratio } \frac{\Delta f}{\Delta x} \text{ at } x = 0 \text{ is } \frac{f(\Delta x) - f(0)}{\Delta x} = \frac{(\Delta x)^2 - 0}{\Delta x} = \Delta x \text{ Then limit = slope = 0.}$$

In this case and in almost all cases, we get a better ratio (closer to the limiting slope df/dx) by averaging the **forward difference** (where $\Delta x > 0$) with the **backward difference** (where $\Delta x < 0$). The average is the more accurate **centered difference**.

$$\textbf{Centered at } x \quad \frac{1}{2}\left[\frac{f(x+\Delta x) - f(x)}{\Delta x} + \frac{f(x-\Delta x) - f(x)}{-\Delta x}\right] = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\,\Delta x}$$

For the example $f(x) = x^2$ this centering will produce the exact derivative $df/dx = 2x$. In the picture we are averaging plus and minus slopes to get the correct slope 0 at $x = 0$. For all smooth functions, the centered differences reduce the error to size $(\Delta x)^2$. This is a big improvement over the error of size Δx for uncentered differences $f(x+\Delta x) - f(x)$.

Figure VI.6: ReLU function = ramp from deep learning. Centered slope of $f = x^2$ is exact.

Most finite difference approximations are centered for extra accuracy. But we are still dividing by a small number $2\,\Delta x$. And for a multivariable function $F(x_1, x_2, \ldots, x_n)$ we will need ratios $\Delta F/\Delta x_i$ in n different directions—possibly for large n. Those ratios approximate the n partial derivatives that go into the **gradient vector grad F = ∇F**.

The gradient of $F(x_1, \ldots, x_n)$ is the column vector $\nabla F = \left(\dfrac{\partial F}{\partial x_1}, \ldots, \dfrac{\partial F}{\partial x_n}\right)$.

Its components are the n partial derivatives of F. ∇F points in the steepest direction.

Examples 1-3 will show the value of vector notation (∇F is always a column vector).

Example 1 For a constant vector $a = (a_1, \ldots, a_n)$, $F(x) = a^{\mathrm{T}}x$ has gradient $\nabla F = a$.

The partial derivatives of $F = a_1x_1 + \cdots + a_nx_n$ are the numbers $\partial F/\partial x_k = a_k$.

Example 2 For a symmetric matrix S, the gradient of $F(x) = x^{\mathrm{T}}Sx$ is $\nabla F = 2\,Sx$. To see this, write out the function $F(x_1, x_2)$ when $n = 2$. The matrix S is 2 by 2:

$$F = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{matrix} ax_1^2 + cx_2^2 \\ +2bx_1x_2 \end{matrix} \quad \begin{bmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \end{bmatrix} = 2\begin{bmatrix} ax_1 + bx_2 \\ bx_1 + cx_2 \end{bmatrix} = 2S\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

Example 3 For a positive definite symmetric S, the minimum of a quadratic $F(x) = \frac{1}{2}x^{\mathrm{T}}Sx - a^{\mathrm{T}}x$ is the negative number $F_{\min} = -\frac{1}{2}a^{\mathrm{T}}Sa$ at $x^* = S^{-1}a$.

This is an important example! The minimum occurs where first derivatives of F are zero:

$$\boxed{\nabla F = \begin{bmatrix} \partial F/\partial x_1 \\ \vdots \\ \partial F/\partial x_n \end{bmatrix} = Sx - a = 0 \text{ at } x^* = S^{-1}a = \arg\min F.} \tag{3}$$

As always, **that notation $\arg\min F$ stands for the point x^* where the minimum of $F(x) = \frac{1}{2}x^{\mathrm{T}}Sx - a^{\mathrm{T}}x$ is reached.** Often we are more interested in this minimizing x^* than in the actual minimum value $F_{\min} = F(x^*)$ at that point:

$$F_{\min} \text{ is } \frac{1}{2}(S^{-1}a)^{\mathrm{T}}S(S^{-1}a) - a^{\mathrm{T}}(S^{-1}a) = \frac{1}{2}a^{\mathrm{T}}S^{-1}a - a^{\mathrm{T}}S^{-1}a = -\frac{1}{2}a^{\mathrm{T}}S^{-1}a.$$

The graph of F is a bowl passing through zero at $x = 0$ and dipping to its minimum at x^*.

Example 4 The **determinant** $F(x) = \det X$ is a function of all n^2 variables x_{ij}. In the formula for $\det X$, each x_{ij} along a row is multiplied by its "cofactor" C_{ij}. This cofactor is a determinant of size $n-1$, using all rows of X except row i and all columns except column j—and multiplied by $(-1)^{i+j}$:

The partial derivatives $\dfrac{\partial(\det X)}{\partial x_{ij}} = C_{ij}$ in the matrix of cofactors of X give ∇F.

Example 5 The **logarithm of the determinant** is a most remarkable function:

$$L(X) = \log(\det X) \text{ has partial derivatives } \frac{\partial L}{\partial x_{ij}} = \frac{C_{ij}}{\det X} = j, i \text{ entry of } X^{-1}.$$

The chain rule for $L = \log F$ is $(\partial L/\partial F)(\partial F/\partial x_{ij}) = (1/F)(\partial F/\partial x_{ij}) = (1/\det X)\,C_{ij}$. Then this ratio of cofactors to determinant gives the j, i entries of the inverse matrix X^{-1}.

It is neat that X^{-1} contains the n^2 first derivatives of $L = \log\det X$. The second derivatives of L are remarkable too. We have n^2 variables x_{ij} and n^2 first derivatives in $\nabla L = (X^{-1})^{\mathrm{T}}$. This means n^4 second derivatives! What is amazing is that the matrix of second derivatives is **negative definite** when $X = S$ is symmetric positive definite. So we reverse the sign of L: **positive definite second derivatives $\Rightarrow$ convex function.**

$-\log(\det S)$ is a convex function of the entries of the positive definite matrix S.

The Geometry of the Gradient Vector ∇f

Start with a function $f(x, y)$. It has $n = 2$ variables. Its gradient is $\nabla f = (\partial f/\partial x, \partial f/\partial y)$. This vector changes length as we move the point x, y where the derivatives are computed:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \quad \textbf{Length} = ||\nabla f|| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} = \text{steepest slope of } f$$

That length $||\nabla f||$ tells us the steepness of the graph of $z = f(x, y)$. The graph is normally a curved surface—like a mountain or a valley in xyz space. At each point there is a slope $\partial f/\partial x$ in the x-direction and a slope $\partial f/\partial y$ in the y-direction. **The steepest slope is in the direction of ∇f = grad f. The magnitude of that steepest slope is $||\nabla f||$.**

Example 6 The graph of a linear function $f(x, y) = ax + by$ is the plane $z = ax + by$. The gradient is the vector $\nabla f = \begin{bmatrix} a \\ b \end{bmatrix}$ of partial derivatives. The length of that vector is $||\nabla f|| = \sqrt{a^2 + b^2}$ = slope of the roof. The slope is steepest in the direction of ∇f.

That steepest direction is perpendicular to the level direction. The level direction z = constant has $ax + by$ = constant. It is the safe direction to walk, perpendicular to ∇f. The component of ∇f in that flat direction is zero. Figure VI.7 shows the two perpendicular directions (level and steepest) on the plane $z = x + 2y = f(x, y)$.

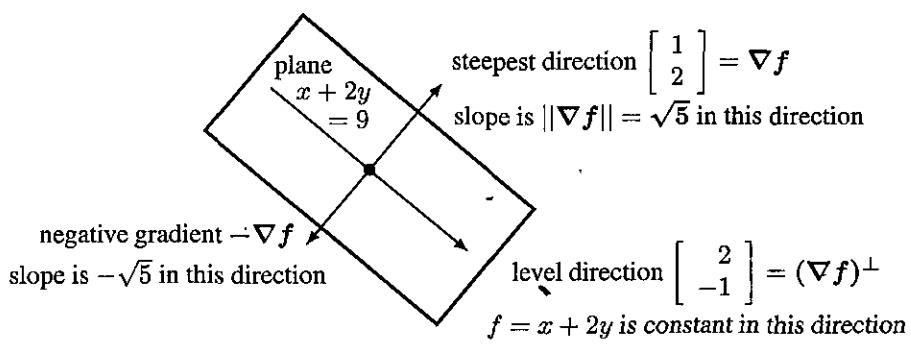
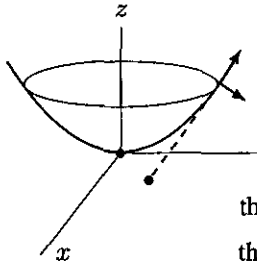


Figure VI.7: The negative gradient $-\nabla f$ gives the direction of steepest descent.

For the nonlinear function $f(x, y) = ax^2 + by^2$, the gradient is $\nabla f = \begin{bmatrix} 2ax \\ 2by \end{bmatrix}$. That tells us the steepest direction, changing from point to point. We are on a curved surface (a bowl opening upward). The bottom of the bowl is at $x = y = 0$ where the gradient vector is zero. The slope in the steepest direction is $||\nabla f||$. At the minimum, $\nabla f = (2ax, 2by) = (0, 0)$ and *slope = zero.*

The level direction has $z = ax^2 + by^2 =$ constant height. That plane $z =$ constant cuts through the bowl in a level curve. In this example the level curve $ax^2 + by^2 = c$ is an ellipse. The direction of the ellipse (level direction) is perpendicular to the gradient vector (steepest direction). But there is a serious difficulty for steepest descent:

The steepest direction changes as you go down! The gradient doesn't point to the bottom!



steepest direction ∇f up and down the bowl $ax^2 + by^2 = z$

flat direction $(\nabla f)^\perp$ along the ellipse $ax^2 + by^2 =$ constant

the steepest direction is perpendicular to the flat direction but

the steepest direction is not aimed at the minimum point

Figure VI.8: **Steepest descent moves down the bowl in the gradient direction** $\begin{bmatrix} -2ax \\ -2by \end{bmatrix}$.

Let me repeat. At the point x_0, y_0 the gradient direction for $f = ax^2 + by^2$ is along $\nabla f = (2ax_0, 2by_0)$. The steepest line through x_0, y_0 is $2ax_0(y - y_0) = 2by_0(x - x_0)$. But then the lowest point $(x, y) = (0, 0)$ does not lie on the line! **We will not find that minimum point in one step of "gradient descent". The steepest direction does not lead to the bottom of the bowl**—except when $b = a$ and the bowl is circular.

Water changes direction as it goes down a mountain. Sooner or later, we must change direction too. In practice we keep going in the gradient direction and stop when our cost function f is not decreasing quickly. At that point Step 1 ends and we recompute the gradient ∇f. This gives a new descent direction for Step 2.

An Important Example with Zig-Zag

The example $f(x, y) = \frac{1}{2}(x^2 + by^2)$ is extremely useful for $0 < b \leq 1$. Its gradient ∇f has two components $\partial f/\partial x = x$ and $\partial f/\partial y = by$. The minimum value of f is zero. That minimum is reached at the point $(x^*, y^*) = (0, 0)$. Best of all, steepest descent with exact line search produces a simple formula for each point (x_k, y_k) in the slow progress down the bowl toward $(0, 0)$. Starting from $(x_0, y_0) = (b, 1)$ we find these points:

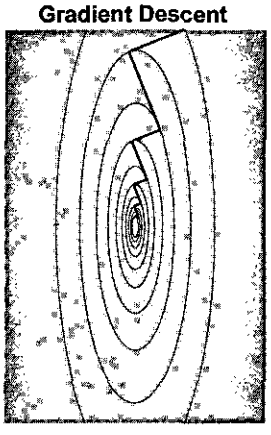
$$x_k = b\left(\frac{b-1}{b+1}\right)^k \quad y_k = \left(\frac{1-b}{1+b}\right)^k \quad f(x_k, y_k) = \left(\frac{1-b}{1+b}\right)^{2k} f(x_0, y_0) \tag{4}$$

If $b = 1$, you see immediate success in one step. The point (x_1, y_1) is $(0, 0)$. The bowl is perfectly circular with $f = \frac{1}{2}(x^2 + y^2)$. The negative gradient direction goes exactly through $(0, 0)$. Then the first step of gradient descent finds that correct minimizing point where $f = 0$.

The real purpose of this example is seen when b is small. The crucial ratio in equation (4) is $r = (b - 1)/(b + 1)$. For $b = \frac{1}{10}$ this ratio is $r = -9/11$. For $b = \frac{1}{100}$ the ratio is $-99/101$. The ratio is approaching -1 and the progress toward $(0, 0)$ has virtually stopped when b is very small.

Figure VI.9 shows the frustrating zig-zag pattern of the steps toward $(0, 0)$. Every step is short and progress is very slow. This is a case where the stepsize s_k in $x_{k+1} = x_k - s_k \nabla f(x_k)$ was exactly chosen to minimize f (an exact line search). But the direction of $-\nabla f$, even if steepest, is pointing far from the final answer $(x^*, y^*) = (0, 0)$.

The bowl has become a narrow valley when b is small, and we are uselessly crossing the valley instead of moving down the valley to the bottom.



The first descent step starts out perpendicular to the level set. As it crosses through lower level sets, the function $f(x, y)$ is decreasing. **Eventually its path is tangent to a level set L.** Descent has stopped. Going further will increase f. The first step ends. The next step is perpendicular to L. So the zig-zag path took a 90° turn.

Figure VI.9: Slow convergence on a zig-zag path to the minimum of $f = x^2 + by^2$.

For b close to 1, this gradient descent is faster. First-order convergence means that the distance to $(x^*, y^*) = (0, 0)$ is reduced by the constant factor $(1 - b)/(1 + b)$ at every step. The following analysis will show that linear convergence extends to all strongly convex functions f—first when each line search is exact, and then (more realistically) when the search at each step is close to exact.

Machine learning often uses **stochastic gradient descent**. The next section will describe this variation (especially useful when n is large and x has many components). And we recall that **Newton's method** uses second derivatives to produce quadratic convergence—the reduction factor $(1 - b)/(1 + b)$ drops to zero and the error is squared at every step. (Our model problem of minimizing a quadratic $\frac{1}{2}x^{\mathrm{T}}Sx$ is solved in one step.) This is a gold standard that approximation algorithms don't often reach in practice.

Convergence Analysis for Steepest Descent

On this page we are following the presentation by Boyd and Vandenberghe in *Convex Optimization* (published by Cambridge University Press). From the specific choice of $f(x,y) = \frac{1}{2}(x^2 + by^2)$, we move to any strongly convex $f(\boldsymbol{x})$ in n dimensions. Convexity is tested by the positive definiteness of the symmetric matrix $H = \boldsymbol{\nabla^2 f}$ of second derivatives (the Hessian matrix). In one dimension this is the number d^2f/dx^2:

Strongly convex $\boldsymbol{m > 0}$ $\quad H_{ij} = \dfrac{\partial^2 f}{\partial x_i \partial x_j}$ has eigenvalues between $m \le \lambda \le M$ at all $\boldsymbol{x}$

The quadratic $f = \frac{1}{2}\left(x^2 + by^2\right)$ has second derivatives 1 and b. The mixed derivative $\partial^2 f/\partial x \partial y$ is zero. So the matrix is diagonal and its two eigenvalues are $m = b$ and $M = 1$. We will now see that **the ratio m/M controls the speed of steepest descent.**

The gradient descent step is $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s\boldsymbol{\nabla f}_k$. We estimate f by its Taylor series:

$$f(\boldsymbol{x}_{k+1}) \le f(\boldsymbol{x}_k) + \boldsymbol{\nabla f}^{\mathrm{T}}(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) + \frac{M}{2}||\boldsymbol{x}_{k+1} - \boldsymbol{x}_k||^2 \tag{5}$$

$$= f(\boldsymbol{x}_k) - s||\boldsymbol{\nabla f}||^2 + \frac{Ms^2}{2}||\boldsymbol{\nabla f}||^2 \tag{6}$$

The best s minimizes the left side (exact line search). The minimum of the right side is at $s = 1/M$. Substituting that number for s, the next point $\boldsymbol{x}_{k+1}$ has

$$f(\boldsymbol{x}_{k+1}) \le f(\boldsymbol{x}_k) - \frac{1}{2M}||\boldsymbol{\nabla f}(\boldsymbol{x}_k)||^2. \tag{7}$$

A parallel argument uses m instead of M to reverse the inequality sign in (5).

$$f(\boldsymbol{x}^*) \ge f(\boldsymbol{x}_k) - \frac{1}{2m}||\boldsymbol{\nabla f}(\boldsymbol{x}_k)||^2. \tag{8}$$

Multiply (7) by M and (8) by m and subtract to remove $||\boldsymbol{\nabla f}(\boldsymbol{x}_k)||^2$. Rewrite the result as

Steady drop in f

$$\boxed{f(\boldsymbol{x}_{k+1}) - f(\boldsymbol{x}^*) \le \left(1 - \frac{m}{M}\right)\left(f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*)\right)} \tag{9}$$

This says that every step reduces the height above the bottom of the valley by at least $c = 1 - \frac{m}{M}$. That is **linear convergence: very slow when $b = m/M$ is small.**

Our zig-zag example had $m = b$ and $M = 1$. The estimate (9) guarantees that the height $f(\boldsymbol{x}_k)$ above $f(\boldsymbol{x}^*) = 0$ is reduced by at least $1 - b$. The exact formula in that totally computable problem produced the reduction factor $(1-b)^2/(1+b)^2$. When b is small this is about $1 - 4b$. So the actual improvement was only 4 times better than the rough estimate $1 - b$ in (9). **This gives us considerable faith that (9) is realistic.**

Inexact Line Search and Backtracking

That ratio m/M appears throughout approximation theory and numerical linear algebra. This is the point of mathematical analysis—to find numbers like m/M that control the rate of descent to the minimum value $f(x^*)$.

Up to now all line searches were exact: x_{k+1} exactly minimized $f(x)$ along the line $x = x_k - s\nabla f_k$. Choosing s is a one-variable minimization. The line moves from x_k in the direction of steepest descent. But we can't expect an exact formula for minimizing a general function $f(x)$, even just along a line. So we need a fast sensible way to find an approximate minimum (and the analysis needs a bound on this additional error).

One sensible way is **backtracking**. Start with the full step $s = 1$ to $X = x_k - \nabla f_k$.

Test If $f(X) \le f(x_k) - \frac{s}{3}\|\nabla f_k\|^2$, with $s = 1$, stop and accept X as x_{k+1}.

Otherwise backtrack: Reduce s to $\frac{1}{2}$ and try the test on $X = x_k - \frac{1}{2}\nabla f_k$.

If the test fails again, try the stepsize $s = \frac{1}{4}$. Since ∇f is a descent direction, the test is eventually passed. The factors $\frac{1}{3}$ and $\frac{1}{2}$ could be any numbers $\alpha < \frac{1}{2}$ and $\beta < 1$.

Boyd and Vandenberghe show that the convergence analysis for exact line search extends also to this backtracking search. Of course the guaranteed reduction factor $1 - (m/M)$ for each step toward the minimum is now not so large. But the new factor $1 - \min(2m\alpha, 2m\alpha\beta/M)$ is still below 1. Steepest descent with backtracking search still has linear convergence—a constant factor (or better) at every step.

Momentum and the Path of a Heavy Ball

The slow zig-zag path of steepest descent is a real problem. We have to improve it. Our model example $f = \frac{1}{2}(x^2 + by^2)$ has only two variables x, y and its second derivative matrix H is diagonal—constant entries $f_{xx} = 1$ and $f_{yy} = b$. But it shows the zig-zag problem very clearly when $b = \lambda_{\min}/\lambda_{\max} = m/M$ **is small.**

Key idea: Zig-zag would not happen for a heavy ball rolling downhill. Its momentum carries it through the narrow valley—bumping the sides but moving mostly forward. So we **add momentum with coefficient β to the gradient** (Polyak's important idea). This gives one of the most convenient and powerful ideas in deep learning.

The direction z_k of the new step remembers the previous direction z_{k-1}.

Descent with momentum
$$\boxed{x_{k+1} = x_k - sz_k \text{ with } z_k = \nabla f(x_k) + \beta z_{k-1}} \quad (10)$$

Now we have two coefficients to choose—the stepsize s and also β. Most important, **the step to x_{k+1} in equation (10) involves z_{k-1}.** Momentum has turned a one-step method (gradient descent) into a two-step method. To get back to one step, we have to rewrite equation (10) as **two coupled equations** (one vector equation) for the state at time $k+1$:

Descent with momentum
$$\boxed{\begin{array}{rl} x_{k+1} & = x_k - sz_k \\ z_{k+1} - \nabla f(x_{k+1}) & = \beta z_k \end{array}} \quad (11)$$

With those two equations, we have recovered a one-step method. This is exactly like reducing a single second order differential equation to a system of two first order equations. Second order reduces to first order when dy/dt becomes a second unknown along with y.

2nd order equation $\frac{d^2y}{dt^2} + b\frac{dy}{dt} + ky = 0$ becomes $\frac{d}{dt}\begin{bmatrix} y \\ dy/dt \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -k & -b \end{bmatrix}\begin{bmatrix} y \\ dy/dt \end{bmatrix}$.
1st order system

Interesting that this b is damping the motion while β adds momentum to encourage it.

The Quadratic Model

When $f(x) = \frac{1}{2}x^{\mathrm{T}}Sx$ is quadratic, its gradient $\nabla f = Sx$ is linear. This is the model problem to understand: S is symmetric positive definite and $\nabla f(x_{k+1})$ becomes Sx_{k+1} in equation (11). Our 2 by 2 supermodel is included, when the matrix S is diagonal with entries 1 and b. For a bigger matrix S, you will see that its largest and smallest eigenvalues determine the best choices for β and the stepsize s—so the 2 by 2 case actually contains the essence of the whole problem.

To follow the steps of accelerated descent, we track each eigenvector of S. Suppose $Sq = \lambda q$ and $x_k = c_k\, q$ and $z_k = d_k\, q$ and $\nabla f_k = Sx_k = \lambda\, c_k\, q$. Then our equation (11) connects the numbers c_k and d_k at step k to c_{k+1} and d_{k+1} at step $k+1$.

Following the eigenvector q

$$\begin{array}{ll} c_{k+1} & = c_k - s\,d_k \\ -\lambda\, c_{k+1} + d_{k+1} & = \beta\, d_k \end{array} \qquad \begin{bmatrix} 1 & 0 \\ -\lambda & 1 \end{bmatrix}\begin{bmatrix} c_{k+1} \\ d_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & -s \\ 0 & \beta \end{bmatrix}\begin{bmatrix} c_k \\ d_k \end{bmatrix} \tag{12}$$

Finally we invert the first matrix ($-\lambda$ becomes $+\lambda$) to see each descent step clearly:

Descent step multiplies by R

$$\boxed{\begin{bmatrix} c_{k+1} \\ d_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \lambda & 1 \end{bmatrix}\begin{bmatrix} 1 & -s \\ 0 & \beta \end{bmatrix}\begin{bmatrix} c_k \\ d_k \end{bmatrix} = \begin{bmatrix} 1 & -s \\ \lambda & \beta - \lambda s \end{bmatrix}\begin{bmatrix} c_k \\ d_k \end{bmatrix} = R\begin{bmatrix} c_k \\ d_k \end{bmatrix}} \tag{13}$$

After k steps the starting vector is multiplied by R^k. For fast convergence to zero (which is the minimum of $f = \frac{1}{2}x^{\mathrm{T}}Sx$) we want both eigenvalues e_1 and e_2 of R to be as small as possible. Clearly those eigenvalues of R depend on the eigenvalue λ of S. That eigenvalue λ could be anywhere between $\lambda_{\min}(S)$ and $\lambda_{\max}(S)$. Our problem is:

Choose s and β to minimize $\max\Big[|e_1(\lambda)|,\, |e_2(\lambda)|\Big]$ for $\lambda_{\min}(S) \le \lambda \le \lambda_{\max}(S)$. (14)

It seems a miracle that this problem has a beautiful solution. The optimal s and β are

$$\boxed{s = \left(\frac{2}{\sqrt{\lambda_{\max}} + \sqrt{\lambda_{\min}}}\right)^2 \quad \text{and} \quad \beta = \left(\frac{\sqrt{\lambda_{\max}} - \sqrt{\lambda_{\min}}}{\sqrt{\lambda_{\max}} + \sqrt{\lambda_{\min}}}\right)^2.} \tag{15}$$

Think of the 2 by 2 supermodel, when S has eigenvalues $\lambda_{\max} = 1$ and $\lambda_{\min} = b$:

$$s = \left(\frac{2}{1+\sqrt{b}}\right)^2 \qquad \text{and} \qquad \beta = \left(\frac{1-\sqrt{b}}{1+\sqrt{b}}\right)^2 \tag{16}$$

These choices of stepsize and momentum give a convergence rate that looks like the rate in equation (4) for ordinary steepest descent (no momentum). But there is a crucial difference: **b is replaced by $\sqrt{b}$.**

$$\boxed{\begin{array}{llll} \textbf{Ordinary} & \left(\dfrac{1-b}{1+b}\right)^2 & \textbf{Accelerated} & \left(\dfrac{1-\sqrt{b}}{1+\sqrt{b}}\right)^2 \\ \textbf{descent factor} & & \textbf{descent factor} & \end{array}} \tag{17}$$

So similar but so different. The real test comes when b is very small. Then the ordinary descent factor is essentially $1 - 4b$, very close to 1. The accelerated descent factor is essentially $1 - 4\sqrt{b}$, *much further from* 1.

To emphasize the improvement that momentum brings, suppose $b = 1/100$. Then $\sqrt{b} = 1/10$ (ten times larger than b). The convergence factors in equation (17) are

$$\textbf{Steepest descent} \quad \left(\frac{.99}{1.01}\right)^2 = .96 \qquad \textbf{Accelerated descent} \quad \left(\frac{.9}{1.1}\right)^2 = .67$$

Ten steps of ordinary descent multiply the starting error by 0.67. This is matched by a single momentum step. Ten steps with the momentum term multiply the error by 0.018.

Notice that $\lambda_{\max}/\lambda_{\min} = 1/b = \kappa$ is the **condition number of S.** This controls everything. For the non-quadratic problems studied next, the condition number is still the key. That number κ becomes L/μ as you will see.

A short editorial *This is not part of the expository textbook.* It concerns the rate of convergence for gradient descent. We know that one step methods (computing $\boldsymbol{x}_{k+1}$ from $\boldsymbol{x}_k$) can multiply the error by $1 - O(1/\kappa)$. Two step methods that use $\boldsymbol{x}_{k-1}$ in the momentum term can achieve $1 - O(\sqrt{1/\kappa})$. The condition number is $\kappa = \lambda_{\max}/\lambda_{\min}$ for the convex quadratic model $\boldsymbol{f} = \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{S}\boldsymbol{x}$. Our example had $\boldsymbol{1/\kappa = b}$.

It is natural to hope that $\boldsymbol{1 - c\kappa^{-1/n}}$ can be achieved by using n known values $\boldsymbol{x}_k, \boldsymbol{x}_{k-1}, \ldots, \boldsymbol{x}_{k-n+1}$. *This might be impossible*—even if it is exactly parallel to finite difference formulas for $\boldsymbol{dx/dt = f(x)}$. Stability there requires a stepsize bound on Δt. Stability in descent requires a bound on the stepsize s. MATLAB's low order code ODE15S is often chosen for stiff equations and ODE45 is the workhorse for smoother solutions. Those are predictor-corrector combinations, not prominent so far in optimization.

The speedup from momentum is like "**overrelaxation**" in the 1950's. David Young's thesis brought the same improvement for iterative methods applied to a class of linear equations $\boldsymbol{Ax = b}$. In those early days of numerical analysis, A was separated into $S - T$ and the iterations were $\boldsymbol{Sx_{k+1} = Tx_k + b}$. They took forever. Now overrelaxation is virtually forgotten, replaced by faster methods (multigrid). Will accelerated steepest descent give way to completely different ideas for minimizing $f(\boldsymbol{x})$?

This is possible but hard to imagine.

Nesterov Acceleration

Another way to bring x_{k-1} into the formula for x_{k+1} is due to Yuri Nesterov. Instead of evaluating the gradient ∇f at x_k, he shifted that evaluation point to $x_k + \gamma_k(x_k - x_{k-1})$. And choosing $\gamma = \beta$ (the momentum coefficient) combines both ideas.

Gradient Descent	Stepsize s	$\beta = 0$	$\gamma = 0$
Heavy Ball	Stepsize s	Momentum β	$\gamma = 0$
Nesterov Acceleration	Stepsize s	Momentum β	shift ∇f by $\gamma \Delta x$

Accelerated descent involves all three parameters s, β, γ:

$$x_{k+1} = x_k + \beta\,(x_k - x_{k-1}) - s\,\nabla f\,(x_k + \gamma\,(x_k - x_{k-1})) \tag{18}$$

To analyze the convergence rate for Nesterov with $\gamma = \beta$, reduce (18) to first order:

$$\textbf{Nesterov} \qquad x_{k+1} = y_k - s\nabla f(y_k) \text{ and } y_{k+1} = x_{k+1} + \beta(x_{k+1} - x_k). \tag{19}$$

Suppose $f(x) = \frac{1}{2}x^{\mathrm{T}}Sx$ and $\nabla f = Sx$ and $Sq = \lambda q$ as before. To track this eigenvector set $x_k = c_k q$ and $y_k = d_k q$ and $\nabla f(y_k) = \lambda d_k q$ in (19):

$c_{k+1} = (1 - s\lambda)d_k$ and $d_{k+1} = (1+\beta)c_{k+1} - \beta c_k = (1+\beta)\,(1 - s\lambda)d_k - \beta c_k$ becomes

$$\begin{bmatrix} c_{k+1} \\ d_{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 - s\lambda \\ -\beta & (1+\beta)(1 - s\lambda) \end{bmatrix} \begin{bmatrix} c_k \\ d_k \end{bmatrix} = R \begin{bmatrix} c_k \\ d_k \end{bmatrix} \tag{20}$$

Every Nesterov step is a multiplication by R. Suppose R has eigenvalues e_1 and e_2, depending on s and β and λ. We want the larger of $|e_1|$ and $|e_2|$ to be as small as possible for all λ between $\lambda_{\min}(S)$ and $\lambda_{\max}(S)$. These choices for s and β give small e's:

$$s = \frac{1}{\lambda_{\max}} \text{ and } \beta = \frac{\sqrt{\lambda_{\max}} - \sqrt{\lambda_{min}}}{\sqrt{\lambda_{\max}} + \sqrt{\lambda_{\min}}} \text{ give } \max(|e_1|, |e_2|) = \frac{\sqrt{\lambda_{\max}} - \sqrt{\lambda_{min}}}{\sqrt{\lambda_{\max}}} \tag{21}$$

When S is the 2 by 2 matrix with eigenvalues $\lambda_{\max} = 1$ and $\lambda_{\min} = b$, that convergence factor (the largest eigenvalue of R) is $1 - \sqrt{b}$.

This shows the same highly important improvement (*from b to $\sqrt{b}$*) as the momentum (heavy ball) formula. The complete analysis by Lessard, Recht, and Packard discovered that Nesterov's choices for s and β can be slightly improved. Su, Boyd, and Candès developed a deeper link between a particular Nesterov optimization and this equation:

$$\textbf{Model for descent} \qquad \frac{d^2y}{dt^2} + \frac{3}{t}\frac{dy}{dt} + \nabla f(t) = 0.$$

Functions to Minimize: The Big Picture

The function $f(x)$ can be strictly convex or barely convex or non-convex. Its gradient can be linear or nonlinear. Here is a list of function types in increasing order of difficulty.

1. $f(x,y) = \frac{1}{2}(x^2 + by^2)$. This has only 2 variables. Its gradient $\nabla f = (x, by)$ is linear. Its Hessian H is a diagonal 2 by 2 matrix with entries 1 and b. Those are the eigenvalues of H. The condition number is $\kappa = 1/b$ when $0 < b < 1$. *Strictly convex.*

2. $f(x_1, \ldots, x_n) = \frac{1}{2}x^{\mathrm{T}}Sx - c^{\mathrm{T}}x$. Here S is a symmetric positive definite matrix. The gradient $\nabla f = Sx - c$ is linear. The Hessian is $H = S$. Its eigenvalues are λ_1 to λ_n. Its condition number is $\kappa = \lambda_{\max}/\lambda_{\min}$. *Strictly convex.*

3. $f(x_1, \ldots, x_n)$ = **smooth strictly convex function.** Its Hessian $H(x)$ is positive definite at all x (H varies with x). The eigenvalues of H are $\lambda_1(x)$ to $\lambda_n(x)$, always positive. The condition number is the maximum over all x of $\lambda_{\max}/\lambda_{\min}$.

   An essentially equivalent condition number is the ratio $L/\lambda_{\min}(x)$:

   $$L = \text{"Lipschitz constant"} \text{ in } ||\nabla f(x) - \nabla f(y)|| \leq L\,||x - y||. \qquad (22)$$

   This allows corners in the gradient ∇f and jumps in the second derivative matrix H.

4. $f(x_1, \ldots, x_n)$ = **convex but not strictly convex.** The Hessian can be only semi-definite, with $\lambda_{\min} = 0$. A small example is the ramp function $f = \mathrm{ReLU}(x) = \max(0, x)$. The gradient ∇f becomes a "**subgradient**" that has multiple values at a corner point. The subgradient of ReLU at $x = 0$ has all values from 0 to 1. *The lines through* $(0,0)$ *with those slopes stay below the ramp function ReLU* (x).

Positive definite H is allowed but so is $\lambda_{\min} = 0$. The condition number can be infinite. The simplest example with $\lambda_{\min} = 0$ has its minimum along the whole line $x + y = 0$:

$$f(x,y) = (x+y)^2 = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \text{ with a } \textbf{semidefinite matrix } S$$

This degeneracy is very typical of deep learning. The number of weights used by the network often far exceeds the number of training samples that determine those weights. (The "MNIST" data set can have 60,000 training samples and 300,000 weights.) Those weights are underdetermined but still gradient descent succeeds. *Why do its weights generalize well—to give good answers for unseen test data?*

When strict convexity is lost (Case 4), a convergence proof is still possible. But the condition number is infinite. And the rate of convergence can become sublinear.

Note. We speak about linear convergence when the error $x_k - x^*$ (the distance to the minimizing point) is reduced by an approximately constant factor $C < 1$ at each step:

$$\textbf{Linear convergence} \qquad ||x_{k+1} - x^*|| \approx C\,||x_k - x^*||. \tag{23}$$

This means that the error decreases **exponentially** (like C^k or $e^{k \log C}$ with $\log C < 0$). Exponential sounds fast, but it can be very slow when C is near 1.

In minimizing quadratics, **non-adaptive methods generally converge to minimum norm solutions.** Those solutions (like $x^+ = A^+ b$ from the pseudoinverse A^+) have zero component in the row space of A. They have the *largest margin.*

Here are good textbook references for gradient descent, including the stochastic version that is coming in VI.5:

1. D. Bertsekas, *Convex Analysis and Optimization*, Athena Scientific (2003).
2. S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge Univ. Press (2004).
3. Yu. Nesterov, *Introductory Lectures on Convex Optimization*, Springer (2004).
4. J. Nocedal and S. Wright, *Numerical Optimization*, Springer (1999).

Four articles coauthored by Ben Recht have brought essential new ideas to the analysis of gradient methods. Papers 1 and 2 study accelerated descent (this section). Papers 3 and 4 study stochastic descent and adaptive descent. Video 5 is excellent.

1. L. Lessard, B. Recht, and A. Packard. *Analysis and design of optimization algorithms via integral quadratic constraints*, arXiv: 1408.3595v7, 28 Oct 2015.
2. A. C. Wilson, B. Recht, and M. Jordan, *A Lyapunov analysis of momentum methods in optimization*, arXiv: 1611.02635v3, 31 Dec 2016.
3. A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, *The marginal value of adaptive gradient methods in machine learning*, arXiv:1705.08292v1, 23 May 2017.
4. C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, *Understanding deep learning requires rethinking generalization*, arXiv:1611.03530v2, 26 Feb 2017, International Conference on Learning Representations (2017).
5. **https://simons.berkeley.edu/talks/ben-recht-2013-09-04**

Section VI.5 on stochastic gradient descent returns to these papers for this message: Adaptive methods can possibly converge to undesirable weights in deep learning. Gradient descent from $x_0 = 0$ (and SGD) finds the minimum norm solution to least squares.

Those adaptive methods (variants of *Adam*) are popular. The formula for x_{k+1} that stopped at x_{k-1} will go much further back—to include *all earlier points* starting at x_0. In many problems that leads to faster training. As with momentum, *memory can help.*

When there are more weights to determine than samples to use (underdetermined problems), we can have multiple minimum points for $f(x)$ and multiple solutions to $\nabla f = 0$. A crucial question in VI.5 is whether improved adaptive methods find good solutions.

Constraints and Proximal Points

How does steepest descent deal with a constraint restricting x to a convex set K? The **projected gradient** and **proximal gradient** methods use four fundamental ideas.

1 Projection onto K *The projection Πx of x onto K is the point in K nearest to x.*

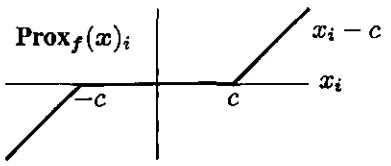
If K is curved then Π is not linear. Think of projection onto the unit ball $||x|| \le 1$. In this case $\Pi x = \text{proj}_K(x) = x/||x||$ for points outside the ball. A key property is that **Π is a contraction**. Projecting two points onto K reduces the distance between them:

$$\textbf{Projection } \Pi = \textbf{proj}_K \qquad ||\Pi x - \Pi z|| \le ||x - z|| \quad \text{for all } x \text{ and } z \text{ in } \mathbf{R}^n. \tag{24}$$

2 Proximal mapping $\textbf{Prox}_f(x)$ *is the vector z that minimizes $\frac{1}{2}||x - z||^2 + f(z)$.* In case $f = 0$ inside K and $f = \infty$ outside K, $\text{Prox}_f(x)$ is exactly the projection Πx.

Important example If $f(x) = c||x||_1$ then Prox_f is the **shrinkage function** in statistics.

The ith component of x leads to
the ith component of $\text{Prox}_f(x)$
This is **soft thresholding** $S(x)$
$S(x_i) = \text{sign}\,(x_i) \cdot \max\,(x_i - c,\, 0)$



We are denoising and regularizing, as in the ℓ^1 LASSO construction of Section III.4. The graph shows how small components are set to zero—producing zeros is the effect that the ℓ^1 norm already achieves compared to ℓ^2.

3 Projected gradient descent takes a normal descent step (which may go outside the constraint set K). Then it projects the result back onto K: a basic idea.

$$\boxed{\textbf{Projected descent} \qquad x_{k+1} = \text{proj}_K(x_k - s_k \nabla f(x_k))} \tag{25}$$

4 Proximal gradient descent also starts with a normal step. Now the projection onto K is replaced by the **proximal map** to determine $s = s_k$: a subtle idea.

$$\boxed{\textbf{Proximal descent} \qquad x_{k+1} = \text{prox}_s(x_k - s\nabla f(x_k))} \tag{26}$$

The LASSO function to minimize is $f(x) = \frac{1}{2}||b - Ax||_2^2 + \lambda||x||_1$. The proximal mapping decides the soft thresholding and the stepsize s at x by

$$\begin{aligned} \text{prox}_s(x) &= \underset{z}{\text{argmin}}\ \frac{1}{2s}||x - z||^2 + \lambda||x||_1 \\ &= \underset{z}{\text{argmin}}\ \frac{1}{2}||x - z||^2 + \lambda s||x||_1 = S_{s\lambda}(x) \end{aligned} \tag{27}$$

That produces the soft-thresholding function S in the graph above as the update x_{k+1}.

$$\boxed{\textbf{Proximal gradients for LASSO (fast descent)} \quad x_{k+1} = S_{\lambda s}(x_k + sA^{\mathrm{T}}(b - Ax_k))}$$

Problem Set VI.4

1 For a 1 by 1 matrix in Example 3, the determinant is just $\det X = x_{11}$. Find the first and second derivatives of $F(X) = -\log(\det X) = -\log x_{11}$ for $x_{11} > 0$. Sketch the graph of $F = -\log x$ to see that this function F is convex.

2 The determinant of a 2 by 2 matrix is $\det(X) = ad - bc$. Its first derivatives are $d, -c, -b, a$ in ∇F. After dividing by $\det X$, those fill the inverse matrix X^{-1}. That division by $\det X$ makes them the four derivatives of $\log(\det X)$:

$$\begin{matrix}\textbf{Derivatives}\\ \textbf{of } F = \det X\end{matrix} \quad \nabla F = \begin{bmatrix} d & -c \\ -b & a \end{bmatrix} \quad \begin{matrix}\textbf{Inverse}\\ \textbf{of } X\end{matrix} = \frac{1}{\det X}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{(\nabla F)^{\mathrm{T}}}{\det X}$$

Symmetry gives $b = c$. Then $F = -\log(ad - b^2)$ is a convex function of a, b, d. *Show that the 3 by 3 second derivative matrix of this function F is positive definite.*

3 Show how equations (7) and (8) lead to the basic estimate (9) for linear convergence of steepest descent. (This extends to backtracking for another choice of c.)

4 A non-quadratic example with its minimum at $x = 0$ and $y = +\infty$ is

$$f(x,y) = \frac{1}{2}x^2 + e^{-y} \qquad \nabla f = \begin{bmatrix} x \\ -e^{-y} \end{bmatrix} \quad H = \begin{bmatrix} 1 & 0 \\ 0 & e^{-y} \end{bmatrix} \quad \kappa = \frac{1}{e^{-y}}$$

5 Explain why projection onto a convex set K is a *contraction* in equation (24). Why is the distance $||x - y||$ never increased when x and y are projected onto K?

6 What is the gradient descent equation $x_{k+1} = x_k - s_k \nabla f(x_k)$ for the least squares problem of minimizing $f(x) = \frac{1}{2}||Ax - b||^2$?

VI.5 Stochastic Gradient Descent and ADAM

Gradient descent is fundamental in training a deep neural network. It is based on a step of the form $x_{k+1} = x_k - s_k \nabla L(x_k)$. That step should lead us downhill toward the point x^* where the loss function $L(x)$ is minimized for the test data v. But for large networks with many samples in the training set, this algorithm (as it stands) is not successful!

It is important to recognize two different problems with classical steepest descent:

1. Computing ∇L at every descent step—the derivatives of the total loss L with respect to all the weights x in the network—is too expensive. That total loss adds the individual losses $\ell(x, v_i)$ *for every sample* v_i *in the training set*—potentially millions of separate losses are computed and added in every computation of L.

2. The number of weights is even larger. So $\nabla_x L = 0$ for many different choices x^* of the weights. **Some of those choices can give poor results on unseen test data.** The learning function F can fail to "generalize". But **stochastic gradient descent (SGD)** does find weights x^* that generalize—weights that will succeed on unseen vectors v from a similar population.

Stochastic gradient descent uses only a "minibatch" of the training data at each step. B samples will be chosen randomly. Replacing the full batch of all the training data by a minibatch changes $L(x) = \frac{1}{n}\sum \ell_i(x)$ to a sum of only B losses. This resolves both difficulties at once. *The success of deep learning rests on these two facts:*

1. Computing $\nabla \ell_i$ by backpropagation on B samples is much faster. Often $B = 1$.

2. The stochastic algorithm produces weights x^* that also succeed on unseen data.

The first point is clear. The calculation per step is greatly reduced. The second point is a miracle. Generalizing well to new data is a gift that researchers work hard to explain.

We can describe the big picture of weight optimization in a large neural network. The weights are determined by the training data. That data often consists of thousands of samples. We know the "*features*" of each sample—maybe its height and weight, or its shape and color, or the number of nouns and verbs and commas (for a sample of text). Those features go into a vector v for each sample. We use a minibatch of samples.

And for each sample in the training set, we know if it is "a cat or a dog"—or if the text is "poetry or prose". We look for a **learning function** F that assigns good weights. Then for v in a similar population, F outputs the correct classification "cat" or "poetry".

We use this function F for unidentified **test data**. The features of the test data are the new inputs v. The output from F will be the correct (?) classification—provided the function has learned the training data in a way that generalizes.

Here is a remarkable observation from experience. *We don't want to fit the training data too perfectly.* That would often be **overfitting**. The function F becomes oversensitive. It memorized everything but it hasn't learned anything. **Generalization by SGD** is the ability to give the correct classification for unseen test data v, based on the weights x that were learned from the training data.

I compare overfitting with choosing a polynomial of degree 60 that fits exactly to 61 data points. Its 61 coefficients a_0 to a_{60} will perfectly learn the data. But that high degree polynomial will oscillate wildly between the data points. For test data at a nearby point, the perfect-fit polynomial gives a *completely wrong answer.*

So a fundamental strategy in training a neural network (which means finding a function that learns from the training data and generalizes well to test data) is **early stopping.** Machine learning needs to know when to quit ! Possibly this is true of human learning too.

The Loss Function and the Learning Function

Now we establish the optimization problem that the network will solve. We need to define the "loss" $L(x)$ that our function will (approximately) minimize. This is the sum of the errors in classifying each of the training data vectors v. And we need to describe the form of the learning function F that classifies each data vector v.

At the beginning of machine learning the function F was *linear*—a severe limitation. Now F is certainly nonlinear. Just the inclusion of one particular nonlinear function at each neuron in each layer has made a dramatic difference. It has turned out that with thousands of samples, the function F can be correctly trained.

It is the processing power of the computer that makes for fast operations on the data. In particular, we depend on the speed of GPU's (the Graphical Processing Units that were originally developed for computer games). They make deep learning possible.

We first choose a loss function to minimize. Then we describe stochastic gradient descent. The gradient is determined by the network architecture—the "feedforward" steps whose weights we will optimize. Our goal in this section is to find *optimization algorithms that apply to very large problems.* Then Chapter VII will describe how the architecture and the algorithms have made the learning functions successful.

Here are three loss functions—cross-entropy is a favorite for neural nets. Section VII.4 will describe the advantages of cross-entropy loss over square loss (as in least squares).

1 **Square loss** $L(x) = \frac{1}{N}\sum_{1}^{N} ||F(x, v_i) - \text{true}||^2$: sum over the training samples v_i

2 **Hinge loss** $L(x) = \frac{1}{N}\sum_{1}^{N} \max(0, 1 - t\,F(x))$ for **classification** $t = 1$ or -1

3 **Cross-entropy loss** $L(x) = -\frac{1}{N}\sum_{1}^{N} [y_i \log \widehat{y_i} + (1-y_i)\log(1-\widehat{y_i})]$ for $y_i = 0$ or 1

Cross-entropy loss or "logistic loss" is preferred for *logistic regression* (with two choices only). The true label $y_i = 0$ or 1 could be -1 or 1 ($\widehat{y_i}$ is a computed label).

For a minibatch of size B, replace N by B. And choose the B samples randomly.

This section was enormously improved by Suvrit Sra's lecture in 18.065 on 20 April 2018.

Stochastic Descent Using One Sample Per Step

To simplify, suppose each minibatch contains only one sample v_k (so $B = 1$). That sample is chosen randomly. The theory of stochastic descent usually assumes that the sample is replaced after use—in principle the sample could be chosen again at step $k+1$. *But replacement is expensive compared to starting with a random ordering of the samples.* In practice, we often omit replacement and work through samples in a random order.

Each pass through the training data is **one epoch** of the descent algorithm. Ordinary gradient descent computes one epoch per step (batch mode). Stochastic gradient descent needs many steps (for minibatches). The online advice is to choose $B \leq 32$.

Stochastic descent began with a seminal paper of Herbert Robbins and Sutton Monro in the *Annals of Mathematical Statistics* **22** (1951) 400-407: A Stochastic Approximation Method. Their goal was a fast method that converges to x^* in probability:

$$\text{To prove} \qquad \text{Prob}\,(\|x_k - x^*\| > \epsilon) \text{ approaches zero as } k \to \infty.$$

Stochastic descent is more sensitive to the stepsizes s_k than full gradient descent. If we randomly choose sample v_i at step k, then the kth descent step is familiar:

$$\boxed{x_{k+1} = x_k - s_k \nabla_x \ell(x_k, v_i)} \qquad \boxed{\nabla_x \ell = \text{derivative of the loss term from sample } v_i}$$

We are doing much less work per step (B inputs instead of all inputs from the training set). But we do not necessarily converge more slowly. A typical feature of stochastic gradient descent is **"semi-convergence"**: fast convergence at the start.

Early steps of SGD often converge more quickly than GD toward the solution x^*.

This is highly desirable for deep learning. Section VI.4 showed zig-zag for full batch mode. This was improved by adding momentum from the previous step (which we may also do for SGD). Another improvement frequently comes by using **adaptive methods** like some variation of ADAM. Adaptive methods look further back than momentum—now all previous descent directions are remembered and used. Those come later in this section.

Here we pause to look at semi-convergence: Fast start by stochastic gradient descent. We admit immediately that later iterations of SGD are frequently erratic. **Convergence at the start changes to large oscillations near the solution.** Figure VI.10 will show this. *One response is to stop early. And thereby we avoid overfitting the data.*

In the following example, the solution x^* is in a specific interval I. If the current approximation x_k is outside I, the next approximation x_{k+1} is closer to I (or inside I). That gives semiconvergence—a good start. *But eventually the x_k bounce around inside I.*

Fast Convergence at the Start: Least Squares with $n = 1$

We learned from Suvrit Sra that the simplest example is the best. The vector x has only one component x. The ith loss is $\ell_i = \frac{1}{2}(a_i x - b_i)^2$ with $a_i > 0$. The gradient of ℓ_i is its derivative $a_i(a_i x - b_i)$. It is zero and ℓ_i is minimized at $x = b_i/a_i$. The total loss over all N samples is $L(x) = \frac{1}{2N} \sum (a_i x - b_i)^2$: Least squares with N equations, 1 unknown.

$$\text{The equation to solve is } \nabla L = \frac{1}{N}\sum_{1}^{N} a_i(a_i x - b_i) = 0. \text{ The solution is } x^* = \frac{\sum a_i b_i}{\sum a_i^2} \quad (1)$$

Important If B/A is the largest ratio b_i/a_i, then the true solution x^* **is below** B/A. This follows from a row of four inequalities:

$$\text{all } \frac{b_i}{a_i} \le \frac{B}{A} \qquad A\,a_i\,b_i \le B\,a_i^2 \qquad A\left(\sum a_i b_i\right) \le B\left(\sum a_i^2\right) \qquad x^* = \frac{\sum a_i b_i}{\sum a_i^2} \le \frac{B}{A}. \quad (2)$$

Similarly x^* is above the smallest ratio β/α. Conclusion: If x_k is outside the interval I from β/α to B/A, then the kth gradient descent step will move *toward that interval I* containing x^*. Here is what we can expect from stochastic gradient descent:

> **If x_k is outside I, then x_{k+1} moves toward the interval $\beta/\alpha \le x \le B/A$.**
> **If x_k is inside I, then so is x_{k+1}. The iterations can bounce around inside I.**

A typical sequence $x_0, x_1, x_2, \ldots$ from minimizing $||Ax - b||^2$ by stochastic gradient descent is graphed in Figure VI.10. *You see the fast start and the oscillating finish.* This behavior is a perfect signal to think about early stopping or averaging (page 365) when the oscillations start.

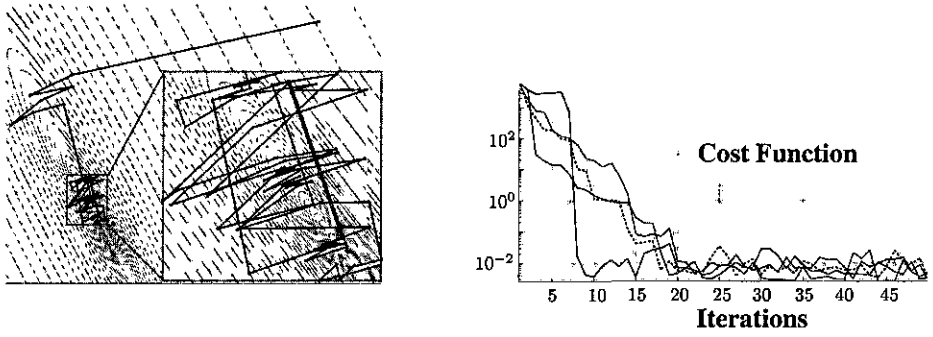


Figure VI.10: The left figure shows a trajectory of stochastic gradient descent with two unknowns. The early iterations succeed but later iterations oscillate (as shown in the inset). On the right, the quadratic cost function decreases quickly at first and then fluctuates instead of converging. The four paths start from the same x_0 with random choices of i in equation (3). The condition number of the 40 by 2 matrix A is only 8.6.

Randomized Kaczmarz is Stochastic Gradient Descent for $Ax = b$

$$\textbf{Kaczmarz for } Ax = b \textbf{ with random } i(k) \qquad x_{k+1} = x_k + \frac{b_i - a_i^{\mathrm{T}} x_k}{||a_i||^2}\, a_i \tag{3}$$

We are randomly selecting row i of A at step k. We are adjusting x_{k+1} to solve equation i in $Ax = b$. (Multiply equation (3) by a_i^{T} to verify that $a_i^{\mathrm{T}} x_{k+1} = b_i$. This is equation i in $Ax = b$.) Geometrically, x_{k+1} is the projection of x_k onto one of the hyperplanes $a_i^{\mathrm{T}} x = b_i$ that meet at $x^* = A^{-1}b$.

This algorithm resisted a close analysis for many years. The equations $a_1^{\mathrm{T}} x = b_1$, $a_2^{\mathrm{T}} x = b_2 \ldots$ were taken in cyclic order with step $s = 1$. Then Strohmer and Vershynin proved fast convergence for random Kaczmarz. They used SGD with *norm-squared sampling (importance sampling)* as in Section II.4: Choose row i of A with probability p_i proportional to $||a_i^{\mathrm{T}}||^2$.

The previous page described the Kaczmarz iterations for $Ax = b$ when A was N by 1. The sequence $x_0, x_1, x_2, \ldots$ moved toward the interval I. The least squares solution x^* was in that interval. For an N by K matrix A, we expect the K by 1 vectors x_i to move into a K-dimensional box around x^*. Figure VI.10 showed this for $K = 2$.

The next page will present numerical experiments for stochastic gradient descent:

A variant of random Kaczmarz was developed by Gower and Richtarik, with no less than six equivalent randomized interpretations. Here are references that connect the many variants from the original by Kaczmarz in the 1937 Bulletin de l'Académie Polonaise.

1 T. Strohmer and R. Vershynin , A randomized Kaczmarz algorithm with exponential convergence, *Journal of Fourier Analysis and Applications* **15** (2009) 262-278.

2 A. Ma, D. Needell, and A. Ramdas, Convergence properties of the randomized extended Gauss-Seidel and Kaczmarz methods, arXiv : 1503.08235$v3$ 1 Feb 2018.

3 D. Needell, N. Srebro, and R. Ward, Stochastic gradient descent, weighted sampling, and the randomized Kaczmarz algorithm, *Math. Progr.* **155** (2015) 549-573.

4 R. M. Gower and P. Richtarik, Randomized iterative methods for linear systems, *SIAM J. Matrix Analysis.* **36** (2015) 1660-1690; arXiv : 1506.03296v5 6 Jan 2016.

5 L. Bottou et al, in *Advances in Neural Information Processing Systems*, NIPS **16** (2004) and NIPS **20** (2008), MIT Press.

6 S. Ma, R. Bassily, and M. Belkin, *The power of interpolation: Understanding the effectiveness of SGD in modern over-parametrized learning*, arXiv : 1712.06559.

7 S. Reddi, S. Sra, B. Poczos, and A. Smola, *Fast stochastic methods for nonsmooth nonconvex optimization*, arXiv: 1605.06900, 23 May 2016.

Random Kaczmarz and Iterated Projections

Suppose $A\boldsymbol{x}^* = \boldsymbol{b}$. A typical step of random Kaczmarz projects the current error $\boldsymbol{x}_k - \boldsymbol{x}^*$ onto the hyperplane $\boldsymbol{a}_i^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}_i$. Here i is chosen randomly at step k (often with importance sampling using probabilities proportional to $||a_i||^2$). To see that projection matrix $\boldsymbol{a}_i\boldsymbol{a}_i^{\mathrm{T}}/\boldsymbol{a}_i^{\mathrm{T}}\boldsymbol{a}_i$, substitute $b_i = \boldsymbol{a}_i^{\mathrm{T}}\boldsymbol{x}^*$ into the update step (3):

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}^* = \boldsymbol{x}_k - \boldsymbol{x}^* + \frac{b_i - \boldsymbol{a}_i^{\mathrm{T}}\boldsymbol{x}_k}{||\boldsymbol{a}_i||^2}\,\boldsymbol{a}_i = (\boldsymbol{x}_k - \boldsymbol{x}^*) - \frac{\boldsymbol{a}_i\boldsymbol{a}_i^{\mathrm{T}}}{\boldsymbol{a}_i^{\mathrm{T}}\boldsymbol{a}_i}(\boldsymbol{x}_k - \boldsymbol{x}^*) \qquad (4)$$

Orthogonal projection never increases length. The error can only decrease. The error norm $||\boldsymbol{x}_k - \boldsymbol{x}^*||$ decreases steadily, even if the cost function $||A\boldsymbol{x}_k - \boldsymbol{b}||$ does not. *But convergence is usually slow* ! Strohmer-Vershynin estimate the expected error:

$$\mathrm{E}\left[||\boldsymbol{x}_k - \boldsymbol{x}^*||^2\right] \le \left(1 - \frac{1}{c^2}\right)^k ||\boldsymbol{x}_0 - \boldsymbol{x}^*||^2,\ c = \text{ condition number of } A. \qquad (5)$$

This is slow compared to gradient descent (there c^2 is replaced by c, and then $\sqrt{c}$ with momentum in VI.4). But (5) is independent of the size of A : attractive for large problems.

The theory of alternating projections was initiated by von Neumann (in Hilbert space). See books and papers by Bauschke-Borwein, Escalante-Raydan, Diaconis, Xu,...

Our experiments converge slowly ! The 100 by 10 matrix A is random with $c \approx 400$. The figures show random Kaczmarz for 600,000 steps. We measure convergence by the angle θ_k between $\boldsymbol{x}_k - \boldsymbol{x}^*$ and the row $\boldsymbol{a}_i$ chosen at step k. The error equation (4) is

$$||\boldsymbol{x}_{k+1} - \boldsymbol{x}^*||^2 = (1 - \cos^2\theta_k)\,||\boldsymbol{x}_k - \boldsymbol{x}^*||^2 \qquad (6)$$

The graph shows that those numbers $1 - \cos^2\theta_k$ *are very close to* 1 : **slow convergence.** But the second graph confirms that convergence does occur. The Strohmer-Vershynin bound (5) becomes $\mathrm{E}[\cos^2\theta_k] \ge 1/c^2$. Our example matrix has $1/c^2 \approx 10^{-5}$ and often $\cos^2\theta_k \approx 2 \cdot 10^{-5}$, confirming that bound.

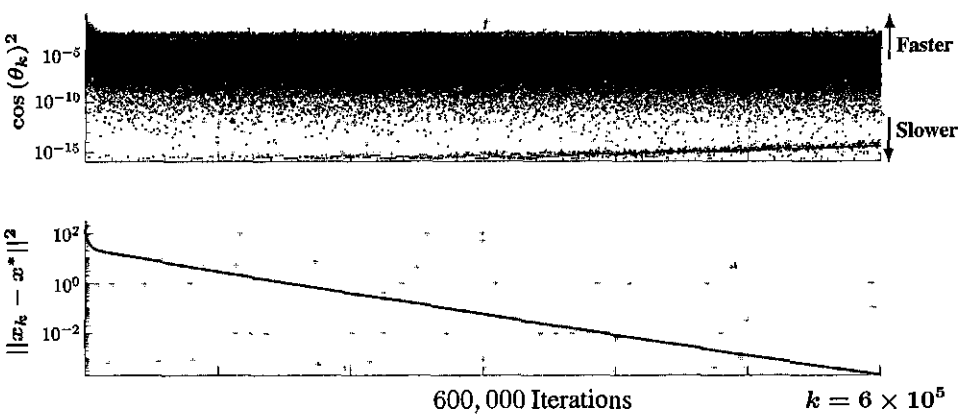


Figure VI.11: Convergence of the squared error for random Kaczmarz. Equation (6) with $1 - \cos^2\theta_k$ close to $1 - 10^{-5}$ produces the slow convergence in the lower graph.

Convergence in Expectation

For a stochastic algorithm like SGD, we need a convergence proof that accounts for randomness—in the assumptions and also in the conclusions. Suvrit Sra provided us with such a proof, and we reproduce it here. The function $f(\boldsymbol{x})$ is a sum $\frac{1}{n}\Sigma f_i(\boldsymbol{x})$ of n terms. The sampling chooses $i(k)$ at step k uniformly from the numbers 1 to n (with replacement!) and the stepsize is $s = \text{constant}/\sqrt{T}$. First come assumptions on $f(\boldsymbol{x})$ and $\nabla f(\boldsymbol{x})$, followed by a standard requirement (no bias) for the random sampling.

1 **Lipschitz smoothness of $\nabla f(\boldsymbol{x})$** $\quad \|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\| \le L\,\|\boldsymbol{x} - \boldsymbol{y}\|$

2 **Bounded gradients** $\quad \|\nabla f_{i(k)}(\boldsymbol{x})\| \le G$

3 **Unbiased stochastic gradients** $\quad \mathbf{E}\left[\nabla f_{i(k)}(\boldsymbol{x}) - \nabla f(\boldsymbol{x})\right] = 0$

From Assumption 1 it follows that

$$f(\boldsymbol{x}_{k+1}) \le f(\boldsymbol{x}_k) + (\nabla f(\boldsymbol{x}_k), \boldsymbol{x}_{k+1} - \boldsymbol{x}_k) + \frac{1}{2}\,Ls^2\|\nabla f_{i(k)}(\boldsymbol{x}_k)\|^2$$

$$f(\boldsymbol{x}_{k+1}) \le f(\boldsymbol{x}_k) + (\nabla f(\boldsymbol{x}_k), -s\,\nabla f_{i(k)}(\boldsymbol{x}_k)) + \frac{1}{2}\,Ls^2\|\nabla f_{i(k)}(\boldsymbol{x}_k)\|^2$$

Now take expectations of both sides and use Assumptions 2-5:

$$\mathbf{E}\,[f(\boldsymbol{x}_{k+1})] \le \mathbf{E}\,[f(\boldsymbol{x}_k)] - s\,\mathbf{E}\,[\,\|\nabla f(\boldsymbol{x}_k)\|^2\,] + \frac{1}{2}\,Ls^2\,G^2$$

$$\Rightarrow \mathbf{E}\,[\,\|\nabla f(\boldsymbol{x}_k)\|^2\,] \le \frac{1}{s}\mathbf{E}\,[f(\boldsymbol{x}_k) - f(\boldsymbol{x}_{k+1})] + \frac{1}{2}\,Ls^2\,G^2. \tag{7}$$

Choose the stepsize $s = c/\sqrt{T}$, and add up (7) from $k = 1$ to T. The sum telescopes:

$$\frac{1}{T}\sum_{k=1}^{T}\mathbf{E}\,[\,\|\nabla f(\boldsymbol{x}_k)\|^2\,] \le \frac{1}{\sqrt{T}}\left(\frac{f(\boldsymbol{x}_1) - f(\boldsymbol{x}^*)}{c} + \frac{Lc}{2}\,G^2\right) = \frac{C}{\sqrt{T}}. \tag{8}$$

Here $f(\boldsymbol{x}^*)$ is the global minimum. The smallest term in (8) is below the average:

$$\min_{1 \le k \le T}\ \mathbf{E}[\,\|\nabla f(\boldsymbol{x}_k)\|^2\,] \le C/\sqrt{T}. \tag{9}$$

The conclusion of Sra's theorem is convergence in expectation at a sublinear rate.

Weight Averaging Inside SGD

The idea of **averaging the outputs** from several steps of stochastic gradient descent looks promising. The learning rate (stepsize) can be constant or cyclical over each group of outputs. Gordon Wilson et al have named this method *Stochastic Weight Averaging* (SWA). They emphasize that this gives promising results for training deep networks, with better generalization and almost no overhead. It seems natural and effective.

P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, A. Gordon Wilson, *Averaging weights leads to wider optima and better generalization*, arXiv: 1803.05407.

Adaptive Methods Using Earlier Gradients

For faster convergence of gradient descent and stochastic gradient descent, adaptive methods have been a major direction. The idea is *to use gradients from earlier steps.* That "memory" guides the choice of search direction $\boldsymbol{D}$ and the all-important stepsize s. We are searching for the vector $\boldsymbol{x}^*$ that minimizes a specified loss function $L(\boldsymbol{x})$. In the step from $\boldsymbol{x}_k$ to $\boldsymbol{x}_{k+1}$, we are free to choose $\boldsymbol{D}_k$ and s_k:

$$\boldsymbol{D}_k = \boldsymbol{D}(\nabla \boldsymbol{L}_k, \nabla \boldsymbol{L}_{k-1}, \ldots, \nabla \boldsymbol{L}_0) \quad \text{and} \quad s_k = s(\nabla \boldsymbol{L}_k, \nabla \boldsymbol{L}_{k-1}, \ldots, \nabla \boldsymbol{L}_0). \tag{10}$$

For a standard SGD iteration, $\boldsymbol{D}_k$ depends only on the current gradient $\nabla \boldsymbol{L}_k$ (and s_k might be $s/\sqrt{k}$). That gradient $\nabla \boldsymbol{L}_k(\boldsymbol{x}_k, B)$ is evaluated only on a random minibatch B of the test data. Now, deep networks often have the option of using some or all of the earlier gradients (computed on earlier random minibatches):

$$\boxed{\textbf{Adaptive Stochastic Gradient Descent} \qquad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s_k \boldsymbol{D}_k} \tag{11}$$

Success or failure will depend on $\boldsymbol{D}_k$ and s_k. The first adaptive method (called ADAGRAD) chose the usual search direction $\boldsymbol{D}_k = \nabla \boldsymbol{L}(\boldsymbol{x}_k)$ but computed the stepsize from all previous gradients [Duchi-Hazan-Singer]:

$$\textbf{ADAGRAD stepsize} \qquad s_k = \left(\frac{\alpha}{\sqrt{k}}\right) \left[\frac{1}{k} \operatorname{diag} \left(\sum_1^k ||\nabla \boldsymbol{L}_i||^2\right)\right]^{1/2} \tag{12}$$

$\alpha/\sqrt{k}$ is a typical decreasing stepsize in proving convergence of stochastic descent. It is often omitted when it slows down convergence in practice. The "memory factor" in (12) led to real gains in convergence speed. Those gains made adaptive methods a focus for more development.

Exponential moving averages in ADAM [Kingma-Ba] have become the favorites. Unlike (12), recent gradients $\nabla \boldsymbol{L}$ have greater weight than earlier gradients in both s_k and the step direction $\boldsymbol{D}_k$. The exponential weights in $\boldsymbol{D}$ and s come from $\delta < 1$ and $\beta < 1$:

$$\boldsymbol{D}_k = (1-\delta) \sum_{i=1}^k \delta^{k-i} \nabla \boldsymbol{L}(\boldsymbol{x}_i) \quad s_k = \left(\frac{\alpha}{\sqrt{k}}\right) \left[(1-\beta) \operatorname{diag} \sum_{i=1}^k \beta^{k-i} ||\nabla \boldsymbol{L}(\boldsymbol{x}_i)||^2\right]^{1/2} \tag{13}$$

Typical values are $\delta = 0.9$ and $\beta = 0.999$. Small values of δ and β will effectively kill off the moving memory and lose the advantages of adaptive methods for convergence speed. That speed is important in the total cost of gradient descent! The look-back formula for the direction $\boldsymbol{D}_k$ is like including momentum in the heavy ball method of Section VI.4.

The actual computation of $\boldsymbol{D}_k$ and s_k will be a recursive combination of old and new:

$$\boxed{\boldsymbol{D}_k = \delta \boldsymbol{D}_{k-1} + (1-\delta) \nabla \boldsymbol{L}(\boldsymbol{x}_k) \qquad s_k^2 = \beta s_{k-1}^2 + (1-\beta) ||\nabla \boldsymbol{L}(\boldsymbol{x}_k)||^2} \tag{14}$$

For several class projects, this adaptive method clearly produced faster convergence.

ADAM is highly popular in practice (this is written in 2018). But several authors have pointed out its defects. Hardt, Recht, and Singer constructed examples to show that its limit x_∞ for the weights in deep learning could be problematic: Convergence may fail or (worse) the limiting weights may *generalize poorly* in applications to unseen test data.

Equations (13)–(14) follow the recent conference paper of Reddi, Kale, and Kumar. Those authors prove non-convergence of ADAM, with simple examples in which *the stepsize s_k increases in time*—an undesired outcome. In their example, ADAM takes the wrong direction twice and the right direction once in every three steps. The exponential decay scales down that good step and overall the stepsizes s_k do not decrease. A large β (near 1) is needed and used, but there are always convex optimization problems on which ADAM will fail. *The idea is still good.*

One approach is to use an increasing minibatch size B. The NIPS 2018 paper proposes a new adaptive algorithm YOGI, which better controls the learning rate (the stepsize). Compared with ADAM, a key change is to an additive update; other steps are unchanged. At this moment, experiments are showing improved results with YOGI.

And after fast convergence to weights that nearly solve $\nabla L(x) = 0$ there is still the crucial issue: **Why do those weights generalize well to unseen test data?**

References

1. S. Ruder, *An overview of gradient descent optimization algorithms*, arXiv:1609.04747.
2. J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. of Machine Learning Research* **12** (2011) 2121–2159.
3. P. Kingma and J. Ba, ADAM: A method for stochastic optimization, ICLR, **2015**.
4. M. Hardt, B. Recht, and Y. Singer, *Train faster, generalize better: Stability of stochastic gradient descent*, arXiv:1509.01240v2, 7 Feb 2017, Proc. ICML (2016).
5. A. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, *The marginal value of adaptive gradient methods in machine learning*, arXiv: 1705.08292, 23 May 2017.
6. S. Reddi, S. Kale, and S. Kumar, On the convergence of ADAM and beyond, ICLR **2018**: *Proc. Intl. Conference on Learning Representations.*
7. S. Reddi, M. Zaheer, D. Sachan, S. Kale, and S. Kumar, *Adaptive methods for nonconvex optimization*, NIPS (2018).

We end this chapter by emphasizing: Stochastic gradient descent is now the leading method to find weights x that minimize the loss $L(x)$ and solve $\nabla L(x^*) = 0$. Those weights from SGD normally succeed on unseen test data.

Generalization : Why is Deep Learning So Effective ?

We end Chapter VI—and connect to Chapter VII—with a short discussion of a central question for deep learning. The issue here is **generalization**. This refers to the behavior of a neural network on test data that it has not seen. If we construct a function $F(x, v)$ that successfully classifies the known training data v, will F continue to give correct results when v is outside the training set ?

The answer must lie in the stochastic gradient descent algorithm that chooses weights. Those weights x minimize a loss function $L(x, v)$ over the training data. The question is : **Why do the computed weights do so well on the test data ?**

Often we have more free parameters in x than data in v. In that case we can expect many sets of weights (many vectors x) to be equally accurate on the training set. Those weights could be good or bad. They could generalize well or poorly. Our algorithm chooses a particular x and applies those weights to new data v_{test}.

An unusual experiment produced unexpectedly positive results. The components of each input vector v were randomly shuffled. So the individual features represented by v suddenly had no meaning. Nevertheless the deep neural net learned those randomized samples. The learning function $F(x, v)$ still classified the test data correctly. Of course F could not succeed with unseen data, when the components of v are reordered.

It is a common feature of optimization that smooth functions are easier to approximate than irregular functions. But here, for completely randomized input vectors, stochastic gradient descent needed only three times as many epochs (triple the number of iterations) to learn the training data. This random labeling of the training samples (the experiment has become famous) is described in arXiv : 1611.03530.

The Kaczmarz Method in Tomographic Imaging (CT)

A key property of Kaczmarz is its quick success in early iterations. This is called *semi-convergence* in tomography (where solving $Ax = b$ constructs a CT image, and the method produces a regularized solution when the data is noisy). Quick semi-convergence for noisy data is an excellent property for such a simple method. The first steps all approach the correct interval from α/β to A/B (for one scalar unknown). But inside that interval, Kaczmarz jumps around unimpressively.

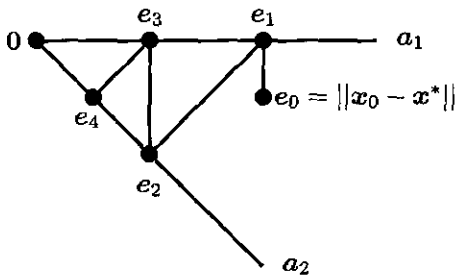
We are entering here the enormous topic of ill-conditioned inverse problems (see the books of P. C. Hansen). In this book we can do no more than open the door.

Problem Set VI.5

1 The rank-one matrix $P = \boldsymbol{a}\boldsymbol{a}^{\mathrm{T}}/\boldsymbol{a}^{\mathrm{T}}\boldsymbol{a}$ is an orthogonal projection onto the line through $\boldsymbol{a}$. Verify that $P^2 = P$ (projection) and that $P\boldsymbol{x}$ is on that line and that $\boldsymbol{x} - P\boldsymbol{x}$ is always perpendicular to $\boldsymbol{a}$ (*why is* $\boldsymbol{a}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{a}^{\mathrm{T}}P\boldsymbol{x}$?)

2 Verify that equation (4) which shows that $\boldsymbol{x}_{k+1} - \boldsymbol{x}^*$ is exactly $P(\boldsymbol{x}_k - \boldsymbol{x}^*)$.

3 If A has only two rows $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$, then Kaczmarz will produce the *alternating projections* in this figure. Starting from any error vector $\boldsymbol{e}_0 = \boldsymbol{x}_0 - \boldsymbol{x}^*$, why does $\boldsymbol{e}_k$ approach zero ? How fast ?



4 Suppose we want to minimize $F(x, y) = y^2 + (y - x)^2$. The actual minimum is $F = 0$ at $(x^*, y^*) = (0, 0)$. Find the gradient vector ∇F at the starting point $(x_0, y_0) = (1, 1)$. For full gradient descent (*not stochastic*) with step $s = \frac{1}{2}$, where is (x_1, y_1) ?

5 In minimizing $F(\boldsymbol{x}) = ||A\boldsymbol{x} - \boldsymbol{b}||^2$, stochastic gradient descent with minibatch size $B = 1$ will solve one equation $\boldsymbol{a}_i^{\mathrm{T}}\boldsymbol{x} = b_i$ at each step. Explain the typical step for minibatch size $B = 2$.

6 (Experiment) For a random A and $\boldsymbol{b}$ (20 by 4 and 20 by 1), try stochastic gradient descent with minibatch sizes $B = 1$ and $B = 2$. Compare the convergence rates—the ratios $r_k = ||\boldsymbol{x}_{k+1} - \boldsymbol{x}^*|| \,/\, ||\boldsymbol{x}_k - \boldsymbol{x}^*||$.

7 (Experiment) Try the weight averaging on page 365 proposed in arXiv : 1803.05407. Apply it to the minimization of $||A\boldsymbol{x} - \boldsymbol{b}||^2$ with randomly chosen A (20 by 10) and $\boldsymbol{b}$ (20 by 1), and minibatch $B = 1$.

Do averages in stochastic descent converge faster than the usual iterates $\boldsymbol{x}_k$?