Αλγοριθμικές Μέθοδοι Βελτιστοποίησης με Έμφαση σε Κατανεμημένα Προβλήματα

Εισαγωγή στην Βελτιστοποίηση & Παραδείγματα σε Julia
Optimization Lecture 1: Key Concepts

• The central importance of convexity, which replaces linearity
  • Convexity involves second derivatives-the graph of $F(\mathbf{x})$ will bend upwards. A big dilemma – First order vs Second order Methods

• The meaning of Lagrange multipliers, which build the constraints into the equation derivative = zero.

• Gradient Descent vs Newton Vs Accelerated Descent vs Levenberg-Marquardt

• Stochastic gradient descent.
  • One step accounts for a part of the data but not all. We hope and expect that the part is reasonably typical of the whole.
• Much of machine learning can be written as an optimization problem

\[
\min_x \sum_{i=1}^{N} f(x; y_i)
\]

- Example loss functions: logistic regression, linear regression, principle component analysis, neural network loss
Types of Optimization

• Convex optimization
  • The easy case
  • Includes logistic regression, linear regression, SVM

• Non-convex optimization
  • NP-hard in general
  • Includes deep learning
$\forall \alpha \in [0, 1], \, f(\alpha x + (1 - \alpha)y) \leq \alpha f(x) + (1 - \alpha)f(y)$
Example: Quadratic

\[ f(x) = x^2 \]

\[
(\alpha x + (1 - \alpha)y)^2 = \alpha^2 x^2 + 2\alpha(1 - \alpha)xy + (1 - \alpha)^2 y^2 \\
= \alpha x^2 + (1 - \alpha)y^2 - \alpha(1 - \alpha)(x^2 + 2xy + y^2) \\
\leq \alpha x^2 + (1 - \alpha)y^2
\]
Example: Absolute Value

Example: Abs

\[ f(x) = |x| \]

\[
|\alpha x + (1 - \alpha)y| \leq |\alpha x| + |(1 - \alpha)y| \\
= \alpha|x| + (1 - \alpha)|y|
\]
Example: Exponential

\[ f(x) = e^x \]

\[ e^{\alpha x + (1-\alpha)y} = e^y e^{\alpha(x-y)} = e^y \sum_{n=0}^{\infty} \frac{1}{n!} \alpha^n (x-y)^n \]

\[ \leq e^y \left( 1 + \alpha \sum_{n=1}^{\infty} \frac{1}{n!} (x-y)^n \right) \]  
(if \( x > y \))

\[ = e^y \left( (1 - \alpha) + \alpha e^{x-y} \right) \]

\[ = (1 - \alpha) e^y + \alpha e^x \]
Properties of Convex Functions

- Any line segment we draw between two points lies above the curve.

- Corollary: every local minimum is a global minimum
  - Why?

- This is what makes convex optimization easy
  - It suffices to find a local minimum, because we know it will be global.
Properties of Convex Functions

• Non-negative combinations of convex functions are convex
  \[ h(x) = af(x) + bg(x) \]

• Affine scalings of convex functions are convex
  \[ h(x) = f(Ax + b) \]

• Compositions of convex functions are \textbf{NOT} generally convex
  • Neural nets are like this
    \[ h(x) = f(g(x)) \]
Convex Functions: Alternative Definitions

• First-order condition

\[ \langle x - y, \nabla f(x) - \nabla f(y) \rangle \geq 0 \]

• Second-order condition

\[ \nabla^2 f(x) \succeq 0 \]

• This means that the matrix of second derivatives is positive semidefinite

\[ A \succeq 0 \iff \forall x, \langle x, Ax \rangle \geq 0 \]
Concave Functions

• A function is concave if its negation is convex

\[ f \text{ is convex } \iff h(x) = -f(x) \text{ is concave} \]

• Example: \( f(x) = \log(x) \)

\[ f''(x) = -\frac{1}{x^2} \leq 0 \]
The Expression "argmin"

• The minimizing x for $F(x) = (x - 1)^2$ is
  • $x^* = \text{argmin} F(x) = 1$.

$\text{argmin} F(x) = \text{value(s) of x where F reaches its minimum.}$

• For strictly convex functions, $\text{argmin} F(x)$ is one point $x^*$: an isolated minimum.

$F$ is Convex $F(px + (1-p)y) \leq pF(x) + (1-p)F(y)$ for $0 < p < 1$,
For a strictly convex function, this holds with strict inequality.
• Machine learning involves functions $F(x_1, \ldots, x_n)$ of many variables.

\[
\text{One function } F \\
\text{One variable } x \\
F(x + \Delta x) \approx F(x) + \Delta x \frac{dF}{dx}(x) + \frac{1}{2} (\Delta x)^2 \frac{d^2F}{dx^2}(x)
\]

\[
\text{One function } F \\
\text{Variables } x_1 \text{ to } x_n \\
F(x + \Delta x) \approx F(x) + (\Delta x)^T \nabla F + \frac{1}{2} (\Delta x)^T H(\Delta x)
\]

\[
\text{m functions } f = (f_1, \ldots, f_m) \\
\text{n variables } x = (x_1, \ldots, x_n) \\
f(x + \Delta x) \approx f(x) + J(x) \Delta x
\]
Minimum Problems: Convexity and Newton's Method

- We focus on problems of minimization, for functions $F(x)$ with many variables

<table>
<thead>
<tr>
<th>Linear constraints</th>
<th>$Ax = b$</th>
<th>(the set of these $x$ is convex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inequality constraints</td>
<td>$x \geq 0$</td>
<td>(the set of these $x$ is convex)</td>
</tr>
<tr>
<td>Integer constraints</td>
<td>Each $x_i$ is 0 or 1</td>
<td>(the set of these $x$ is not convex)</td>
</tr>
</tbody>
</table>

- $K$ is a convex set
  - If $x$ and $y$ are in $K$, so is the line from $x$ to $y$

- $F$ is a convex function
  - The set of points on and above the graph of $F$ is convex

- $F$ is smooth and convex
  - $F(x) \geq F(y) + (\nabla F(y), x - y)$
Convexity - Properties

• The convexity of a function is an important fact!

• This is where linearity fails but convexity succeeds!

• The maximum of two or more linear functions is rarely linear. But the maximum \( F(x) \) of two or more convex functions \( F_i(x) \) is always convex.

• Similarly the maximum of any family of convex functions will be convex.

• But the minimum of two convex functions is generally not convex-it can have a "double well".

• Convexity avoids the truly dangerous situation when \( F \) has its minimum value at an unknown number of separate points in \( K \).
The $l_1$ and $l_2$ and $l_\infty$ Norms of $x$

Norms $F(x) = ||x||$ are convex functions of $x$. The unit ball where $||x|| \leq 1$ is a convex set $K$ of vectors $x$. That first sentence is exactly the triangle inequality:

**Convexity of $||x||$**

$$||p\cdot x + (1 - p)\cdot y|| \leq p||x|| + (1 - p)||y||$$

There are three favorite vector norms $\ell^1$, $\ell^2$, $\ell^\infty$. We draw the unit balls $||x|| \leq 1$ in $\mathbb{R}^2$:

- $l_1$ norm
  - $||x||_1 = |x_1| + |x_2|$

- $l_2$ norm
  - $||x||_2 = \sqrt{x_1^2 + x_2^2}$

- $l_\infty$ norm
  - $||x||_\infty = \max(|x_1|, |x_2|)$
Newton's Method (1/2)

• We are looking for the point $x^*$ where $F(x)$ has a minimum and its gradient is the zero vector. We have reached a nearby point after $k$ iterations.

• Near our current point $X_k$, the gradient is often well estimated by using its first derivative

$$\nabla F(x_{k+1}) \approx \nabla F(x_k) + H(x_k)(x_{k+1} - x_k).$$

• We want that left hand side to be zero. So the natural choice for $X_{k+1}$ comes when the right side is zero: we have $n$ linear equations for the step

$$x_{k+1} \text{ minimizes } F(x_k) + \nabla F(x_k)^T(x - x_k) + \frac{1}{2}(x - x_k)^TH(x_k)(x - x_k).$$
Newton's Method (2/2)

• Newton's method is second order. It uses second derivatives (in H).

• There will still be an error in the new point. But that error is proportional to the square of the error in the previous point:

\[
\|x_{k+1} - x^*\| \leq C \|x_k - x^*\|^2.
\]

• Newton's method is eventually fast, because it uses the second derivatives of F(x). But those can be too expensive to compute—especially in high dimensions.

• Next two slides describe a compromise that gets better near x*.
Newton's Method Examples

\[ f(x, y) = (x - y)^4 + 2x^2 + y^2 - x + 2y \]
Levenberg-Marquardt: Nonlinear Least Squares (1/5)

• Least squares begins with a set of m data points \((t_i, Y_i)\)

• It aims to fit those m points as well as possible by choosing the parameters \(p = (p_1, \ldots, p_n)\) in a fitting function \(y_i = C + Dt_i\)

• Then the sum of squared errors depends on \(C\) and \(D\) (i.e., \(p=(C,D))\):

\[
E(C, D) = (y_1 - C - Dt_1)^2 + \cdots + (y_m - C - Dt_m)^2.
\]

• The minimum error \(E\) is at the values \(C\) and \(D\) where the derivatives with respect to \(C\) and \(D\) are zero
This is linear least squares. The fitting function $f_j$ is linear in $C$ and $D$. $J$ would normally be called $A$.

But for nonlinear least squares the fitting function $f_j(p)$ depends in a nonlinear way on the parameters $p = (p_1, \ldots, p_n)$. 

\[ m \text{ equations} \quad J \begin{bmatrix} C \\ D \end{bmatrix} = \begin{bmatrix} 1 & t_1 \\ \vdots & \vdots \\ 1 & t_m \end{bmatrix} \begin{bmatrix} C \\ D \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = y \] 

\[ 2 \text{ equations} \quad J^T J \begin{bmatrix} \hat{C} \\ \hat{D} \end{bmatrix} = J^T y \quad \text{for the best parameters} \quad \hat{p} = \begin{bmatrix} \hat{C} \\ \hat{D} \end{bmatrix}. \]
• When we minimize the total error $E = \text{sum of squares}$, we expect $n$ nonlinear equations to determine the best parameters.

\[
E(p) = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 = (y - \hat{y}(p))^T (y - \hat{y}(p))
\]

\[
= y^Ty - 2y^T\hat{y}(p) + \hat{y}(p)^T\hat{y}(p).
\]

• This is the "square loss" error function to minimize by choosing the best parameters $p$.

• Our problem is to minimize $E(p)$. The next slide describes an algorithm to minimize $E$ approximating Newton but avoiding second derivatives of $E$. 
Levenberg-Marquardt: Nonlinear Least Squares (4/5)

\[ \nabla E = 2J^T(y - \hat{y}(p_n)) = 0 \] with \( m \) by \( n \) Jacobian matrix \( J = \frac{\partial y}{\partial p} \) at \( \hat{p} \).

\( J \) was a constant \( m \) by 2 matrix when the fitting function \( y = C + Dt \) was linear in the parameters \( p = (C, D) \). The least squares equation for minimum error is by setting to zero the gradient \( J^TJ\hat{\rho} = J^T\hat{y} \).

**Gradient descent**

\[ p_{n+1} - p_n = -sJ^T(y - \hat{y}(p_n)) \]

**Newton (approximate)**

\[ J^TJ(p_{n+1} - p_n) = J^T(y - \hat{y}(p_n)) \]

That symmetric matrix is an approximation to the second derivative matrix 1/2H (the Hessian of the function \( E \)).
The key idea of Levenberg and Marquardt was to combine the gradient descent and Newton update rules into one rule.

\[
\text{Levenberg-Marquardt} \quad (J^T J + \lambda I)(p_{n+1} - p_n) = J^T(y - \hat{y}(p_n)).
\]

Small values of \( \lambda \) will lean toward Newton, large values of \( \lambda \) will lean more toward gradient descent.

Levenberg-Marquardt is an enhanced first order method, extremely useful for nonlinear least squares. It is one way to train neural networks of moderate size.
Lagrange Multipliers = Derivatives of the Cost

Minimize $F(x) = x_1^2 + x_2^2$ on the line $K: a_1 x_1 + a_2 x_2 = b$

The constraint line is tangent to the minimum cost circle at the solution $x^*$. 

slope $x_2^*/x_1^* = a_2/a_1$

constraint line $a_1 x_1 + a_2 x_2 = b$
slope $-a_1/a_2$
Lagrange Multipliers = Derivatives of the Cost

Multiply $a_1 x_1 + a_2 x_2 - b$ by an unknown multiplier $\lambda$ and add it to $F(x)$

Lagrangian $L(x, \lambda) = F(x) + \lambda(a_1 x_1 + a_2 x_2 - b)$

$= x_1^2 + x_2^2 + \lambda(a_1 x_1 + a_2 x_2 - b)$

Set the derivatives $\partial L/\partial x_1$ and $\partial L/\partial x_2$ and $\partial L/\partial \lambda$ to zero.

Solve those three equations for $x_1, x_2, \lambda$.

\[
\begin{align*}
\frac{\partial L}{\partial x_1} &= 2x_1 + \lambda a_1 = 0 \\
\frac{\partial L}{\partial x_2} &= 2x_2 + \lambda a_2 = 0 \\
\frac{\partial L}{\partial \lambda} &= a_1 x_1 + a_2 x_2 - b = 0 \quad \text{(the constraint!)}
\end{align*}
\]

\[
\begin{align*}
x_1^* &= -\frac{1}{2} \lambda a_1 = \frac{a_1 b}{a_1^2 + a_2^2} \\
x_2^* &= -\frac{1}{2} \lambda a_2 = \frac{a_2 b}{a_1^2 + a_2^2} \\
(x_1^*)^2 + (x_2^*)^2 &= \frac{b^2}{a_1^2 + a_2^2}
\end{align*}
\]

\[
\frac{d}{db} \left( \frac{b^2}{a_1^2 + a_2^2} \right) = \frac{2b}{a_1^2 + a_2^2} = -\lambda.
\]
Minimize a Quadratic with Linear Constraints

- Instead of one constraint on x we have m constraints. There will be m Lagrange multipliers, one for each constraint.

Problem: Minimize $F = \frac{1}{2}x^T S x$ subject to $A^T x = b$.

$L(x, \lambda) = \frac{1}{2}x^T S x + \lambda^T (A^T x - b)$

- $x$-derivatives of $L$: $S x + A \lambda = 0$
- $\lambda$-derivatives of $L$: $A^T x = b$
Minimize a Quadratic with Linear Constraints

Solution $\lambda^*, x^*$

$$\lambda^* = -(A^T S^{-1} A)^{-1} b \quad x^* = S^{-1} A (A^T S^{-1} A)^{-1} b.$$ 

Minimum cost $F^*$

$$F^* = \frac{1}{2} (x^*)^T S x^* = \frac{1}{2} b^T (A^T S^{-1} A)^{-1} A^T S^{-1} S S^{-1} A (A^T S^{-1} A)^{-1} b. \quad \text{This simplifies a lot!}$$ 

Minimum cost $F^*$

$$F^* = \frac{1}{2} b^T (A^T S^{-1} A)^{-1} b$$

Gradient of cost $\frac{\partial F^*}{\partial b} = (A^T S^{-1} A)^{-1} b = -\lambda^*$
Gradient Descent Toward the Minimum

- Calculus teaches us that all the first derivatives are zero at the minimum (when f is smooth).
- The steepest direction, in which f(x) decreases fastest, is given by the gradient \(-\nabla f\):

\[
x_{k+1} = x_k - s_k \nabla f(x_k)
\]

- \(s_k\) is the stepsize or the learning rate. We hope to move toward the point \(x^*\) where the graph of f(x) hits bottom.

The gradient of \(F(x_1, \ldots, x_n)\) is the column vector \(\nabla F = \left( \frac{\partial F}{\partial x_1}, \ldots, \frac{\partial F}{\partial x_n} \right)\).
Gradient Descent Example

\[ f(x, y) = (x - y)^4 + 2x^2 + y^2 - x + 2y \]

- sk = s = 0.09
- sk = s = 0.2
Gradient Examples

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

The partial derivatives of $F = a_1 x_1 + \cdots + a_n x_n$ are the numbers $\partial F/\partial x_k = a_k$.

Example 2  For a symmetric matrix $S$, the gradient of $F(x) = x^T S x$ is $\nabla F = 2 S x$.

To see this, write out the function $F(x_1, x_2)$ when $n = 2$. The matrix $S$ is $2$ by $2$:

$$F = \begin{bmatrix} x_1 & x_2 \\ \end{bmatrix} \begin{bmatrix} a & b \\ b & c \\ \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \end{bmatrix} = ax_1^2 + cx_2^2 + 2bx_1x_2$$

$$\begin{bmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \\ \end{bmatrix} = 2 \begin{bmatrix} ax_1 + bx_2 \\ bx_1 + cx_2 \\ \end{bmatrix} = 2S \begin{bmatrix} x_1 \\ x_2 \\ \end{bmatrix}.$$ 

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

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

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

26/2/2021 ΑΛΓΟΡΙΘΜΙΚΕΣ ΜΕΘΟΔΟΙ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗΣ ΜΕ ΕΜΦΑΣΗ ΣΕ ΚΑΤΑΝΕΜΗΜΕΝΑ ΠΡΟΒΛΗΜΑΤΑ 32
Gradient Examples

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 $(-1)^{i+j}$:

$$\frac{\partial(\det X)}{\partial x_{ij}} = C_{ij}$$

in the matrix of cofactors of $X$ give $\nabla F$.

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

$$L(X) = \log(\det X)$$

has partial derivatives

$$\frac{\partial L}{\partial x_{ij}} = \frac{C_{ij}}{\det X} = j, \text{ i entry of } X^{-1}.$$
The Geometry of the Gradient Vector

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

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

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 $||\nabla f|| = \sqrt{a^2 + b^2} = \text{slope of the roof}$. The slope is steepest in the direction of $\nabla f$.

$$\text{plane} \quad x + 2y = 9$$

$$\text{negative gradient } -\nabla f \quad \text{slope is } -\sqrt{5} \text{ in this direction}$$

$$\text{steepest direction} \quad \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \nabla f$$

$$\text{slope is } ||\nabla f|| = \sqrt{5} \text{ in this direction}$$

$$\text{level direction} \quad \begin{bmatrix} 2 \\ -1 \end{bmatrix} = (\nabla f)_{\perp}$$

$f = x + 2y$ is constant in this direction
The Geometry of the Gradient Vector

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

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

Flat direction $(\nabla f) \perp$ along the ellipse $ax^2 + by^2 = \text{constant}$

the steepest direction is perpendicular to the flat direction but the steepest direction is not aimed at the minimum point.
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 $\nabla f$ has two components $\frac{\partial f}{\partial x} = x$ and $\frac{\partial f}{\partial y} = by$. 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:

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

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.
An Important Example with Zig-Zag

- \( b = 0.1, \ s = 0.1 \)
- \( b = 0.8, \ s = 0.1 \)
An Important Example with Zig-Zag

b=0.1, optimal line search

b=0.99, optimal line search
Convergence Analysis for Steepest Descent

Convergence Analysis for Steepest Descent

The gradient descent step is $x_{k+1} = x_k - s\nabla f_k$. We estimate $f$ by its Taylor series:

$$f(x_{k+1}) \leq f(x_k) + \nabla f^T(x_{k+1} - x_k) + \frac{M}{2}||x_{k+1} - x_k||^2$$

$$= f(x_k) - s||\nabla f||^2 + \frac{Ms^2}{2}||\nabla f||^2$$

Steady drop in $f$

$$f(x_{k+1}) - f(x^*) \leq \left(1 - \frac{m}{M}\right)\left(f(x_k) - f(x^*)\right)$$
Inexact Line Search and Backtracking

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 $f(X) \leq f(x_k) - \frac{s}{3} \| \nabla f_k \|^2$, 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 $\nabla f$ 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 $\beta < 1$. 
Momentum and the Path of a Heavy Ball

**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 $\beta$ to the gradient (Polyak's important idea).
- The direction $z_k$ of the new step remembers the previous direction $z_{k-1}$.

Descent with momentum:

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

- Now we have two coefficients to choose-the stepsize $s$ and also $\beta$.
- Momentum has turned a one-step method (gradient descent) into a two-step method.

Descent with momentum:

$$x_{k+1} = x_k - sqz_k$$

$$z_{k+1} - \nabla f(x_{k+1}) = \beta z_k$$
The Quadratic Model

- Let us assume that \( f(x) = x'Sx \) is a quadratic function. Then the gradient is a linear function.

- To follow the steps of accelerated descent, we track each eigenvector of \( S \).

\[
\text{Following the eigenvector } q: \quad c_{k+1} = c_k - s d_k \\
\quad \beta d_k \left[ \begin{array}{c} 1 \\ -\lambda \\ 1 \end{array} \right] \left[ \begin{array}{c} c_{k+1} \\ d_{k+1} \end{array} \right] = \left[ \begin{array}{c} 1 \\ 0 \\ -s \end{array} \right] \left[ \begin{array}{c} c_k \\ \beta d_k \end{array} \right]
\]

Descent step multiplies by \( R \):

\[
\left[ \begin{array}{c} c_{k+1} \\ d_{k+1} \end{array} \right] = \left[ \begin{array}{cc} 1 & 0 \\ \lambda & 1 \end{array} \right] \left[ \begin{array}{c} 1 \\ -s \\ \beta \end{array} \right] \left[ \begin{array}{c} c_k \\ \beta d_k \end{array} \right] = R \left[ \begin{array}{c} c_k \\ d_k \end{array} \right]
\]

Choose \( s \) and \( \beta \) to minimize \( \max \left| e_1(\lambda), |e_2(\lambda)| \right| \) for \( \lambda_{\min}(S) \leq \lambda \leq \lambda_{\max}(S) \).

\[
s = \left( \frac{2}{\sqrt{\lambda_{\max}} + \sqrt{\lambda_{\min}}} \right)^2 \quad \text{and} \quad \beta = \left( \frac{\sqrt{\lambda_{\max}} - \sqrt{\lambda_{\min}}}{\sqrt{\lambda_{\max}} + \sqrt{\lambda_{\min}}} \right)^2.
\]

Ordinary descent factor: \( \left( \frac{1 - b}{1 + b} \right)^2 \)

Accelerated descent factor: \( \left( \frac{1 - \sqrt{b}}{1 + \sqrt{b}} \right)^2 \)
Another way to bring $x_{k-1}$ into the formula for $x_{k+1}$ is due to Yuri Nesterov. Instead of evaluating the gradient $\nabla f$ 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 $\beta$
- $\gamma = 0$

**Nesterov Acceleration**
- Stepsize $s$
- Momentum $\beta$
- Shift $\nabla f$ by $\gamma \Delta x$

Accelerated descent involves all three parameters $s$, $\beta$, $\gamma$:

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

To analyze the convergence rate for Nesterov with $\gamma = \beta$, we get

**Nesterov**

$$x_{k+1} = y_k - s \nabla f(y_k) \quad \text{and} \quad y_{k+1} = x_{k+1} + \beta (x_{k+1} - x_k).$$
Nesterov Acceleration

Suppose $f(x) = \frac{1}{2} x^T S x$ and $\nabla f = S x$ and $S q = \lambda q$ 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):

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

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

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

$$
\begin{align*}
    s &= \frac{1}{\lambda_{\max}} \quad \text{and} \quad \beta = \frac{\sqrt{\lambda_{\max}} - \sqrt{\lambda_{\min}}}{\sqrt{\lambda_{\max}} + \sqrt{\lambda_{\min}}} \\
    \text{give} \quad \max(|e_1|, |e_2|) &= \frac{\sqrt{\lambda_{\max}} - \sqrt{\lambda_{\min}}}{\sqrt{\lambda_{\max}}}
\end{align*}
$$

26/2/2021
Stochastic Gradient Descent

Consider minimizing an average of functions

\[
\min_x \frac{1}{m} \sum_{i=1}^{m} f_i(x)
\]

As \( \nabla \sum_{i=1}^{m} f_i(x) = \sum_{i=1}^{m} \nabla f_i(x) \), gradient descent would repeat:

\[
x^{(k)} = x^{(k-1)} - t_k \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla f_i(x^{(k-1)}), \quad k = 1, 2, 3, \ldots
\]

In comparison, stochastic gradient descent or SGD (or incremental gradient method) repeats:

\[
x^{(k)} = x^{(k-1)} - t_k \cdot \nabla f_{i_k}(x^{(k-1)}), \quad k = 1, 2, 3, \ldots
\]

where \( i_k \in \{1, \ldots, m\} \) is some chosen index at iteration \( k \).

(Robbins and Monro, 1951, Annals of Mathematical Statistics)
Stochastic Gradient Descent

Two rules for choosing index $i_k$ at iteration $k$:

- **Randomized rule:** choose $i_k \in \{1, \ldots, m\}$ uniformly at random
- **Cyclic rule:** choose $i_k = 1, 2, \ldots, m, 1, 2, \ldots, m, \ldots$

Randomized rule is more common in practice. For randomized rule, note that

$$\mathbb{E}[\nabla f_{i_k}(x)] = \nabla f(x)$$

so we can view SGD as using an unbiased estimate of the gradient at each step.

Main appeal of SGD:

- Iteration cost is independent of $m$ (number of functions)
- Can also be a big savings in terms of memory usage
Stochastic Gradient Descent – Step Size

Standard in SGD is to use diminishing step sizes, e.g., \( t_k = 1/k \), for \( k = 1, 2, 3, \ldots \).

Why not fixed step sizes? Here’s some intuition. Suppose we take cyclic rule for simplicity. Set \( t_k = t \) for \( m \) updates in a row, we get:

\[
x^{(k+m)} = x^{(k)} - t \sum_{i=1}^{m} \nabla f_i(x^{(k+i-1)})
\]

Meanwhile, full gradient with step size \( t \) would give:

\[
x^{(k+1)} = x^{(k)} - t \sum_{i=1}^{m} \nabla f_i(x^{(k)})
\]

The difference here: \( t \sum_{i=1}^{m} [\nabla f_i(x^{(k+i-1)}) - \nabla f_i(x^{(k)})] \), and if we hold \( t \) constant, this difference will not generally be going to zero.
Stochastic Gradient Descent – Mini Batches

Also common is mini-batch stochastic gradient descent, where we choose a random subset \( I_k \subseteq \{1, \ldots, m\} \), of size \(|I_k| = b \ll m\), and repeat:

\[
x^{(k)} = x^{(k-1)} - t_k \cdot \frac{1}{b} \sum_{i \in I_k} \nabla f_i(x^{(k-1)}), \quad k = 1, 2, 3, \ldots
\]

Again, we are approximating full gradient by an unbiased estimate:

\[
\mathbb{E} \left[ \frac{1}{b} \sum_{i \in I_k} \nabla f_i(x) \right] = \nabla f(x)
\]

Using mini-batches reduces the variance of our gradient estimate by a factor \(1/b\), but is also \(b\) times more expensive.
Some Julia Examples: Newton

```julia
### Newton Method Version 1

```macro
function Newton(grad, dgrad, start, n_iter = 100, tolerance = 1e-10)
    dim_size = length(start)
    vector = Array{Float64,2}(undef,n_iter,dim_size)
    diff = Array{Float64,1}(undef,dim_size)
    vector[1,:] = start
    iterations = 0;
    for i in range(2, step=1, stop = n_iter)
        diff = - dgrad(vector[i-1,:])\grad(vector[i-1,:])
        ## Uses the convenient NumPy functions numpy.all() and numpy.abs() to compare
        ## the absolute values of diff and tolerance in a single statement
        #if np.all(np.abs(diff).<= tolerance)
        # break
        #end
        eps=sum(abs.(grad(vector[i-1,:])));
        if eps<= tolerance
            break
        end
        vector[i,:] = vector[i-1,:]+ diff[:]
        iterations +=1
    end
    return vector[1:iterations,:], sum(abs.(grad(vector[iterations,:]))), iterations
end
```
Some Julia Examples: GD with fixed sk

```julia
function gradient_descent_v3(f, start, learn_rate, n_iter = 100, tolerance = 1e-10)
    dim_size = length(start)
    vector = Array{Float64,2}(undef,n_iter,dim_size)
    vector[1,:] = start
    iter=1;
    for i in range(2, step=1, stop = n_iter)
        #diff2 = - learn_rate.*ForwardDiff.gradient(f,vector[i-1,:])
        x = vector[i-1,:];
        #gradf = [4*(x[1]-x[2])^3+4*x[1]-1; -4*(x[1]-x[2])^3+2*x[2]+2]
        gradf = ForwardDiff.gradient(f,x)
        diff = - learn_rate.*gradf
        ## Uses the convenient NumPy functions numpy.all() and numpy.abs() to compare
        ## the absolute values of diff and tolerance in a single statement
        eps=sum(abs.(gradf[:]));
        if eps<= tolerance
            break
        end
        vector[i,:] = vector[i-1,:]+diff[:]
        iter=iter+1;
    end
    return vector[1:iter,:]
end
```
Some Julia Examples: GD Optimal Line Search

```julia
function gradient_descentv3(f, start, learn_rate, n_iter = 100, tolerance = 1e-10)
    dim_size = length(start)
    vector = Array{Float64,2}(undef,n_iter,dim_size)
    vector[1,:] = start
    iter=1;
    for i in range(2, step=1, stop = n_iter)
        #diff2 = - learn_rate.*ForwardDiff.gradient(f,vector[i-1,:])
        x = vector[i-1,:];
        gradf = [4*(x[1]-x[2])^3+4*x[1]-1;-4*(x[1]-x[2])^3+2*x[2]+2]
        gradf = ForwardDiff.gradient(f,x)
        diff = - learn_rate.*gradf
        ## Uses the convenient NumPy functions numpy.all() and numpy.abs() to compare
        ## the absolute values of diff and tolerance in a single statement
        eps=sum(abs.(gradf[:]));
        if eps<= tolerance
            break
        end
        vector[i,:] = vector[i-1,:] + diff[:]
        iter=iter+1;
    end
    return vector[1:iter,:]
end
```
Some Julia Examples: Animated Plots

```julia
function test_zig_zag(b)
    g(x) = 0.5*(x[1]^2+b*x[2]^2)
    x₀=[1 0]
    y₁, eps, iter_num = Newton(v→ForwardDiff.gradient(g,v), v→ForwardDiff.hessian(g,v), x₀);
    y₂ = gradient_descent2(v→ForwardDiff.gradient(g,v), x₀, 0.09);
    y, iter_num = gradient_descentv3(g, x₀, 0.5);
    println(iter_num)
    y₃, iter = gradient_descentv2(v→ForwardDiff.gradient(g,v), v→ForwardDiff.hessian(g,v), x₀);

    xs = LinRange(-1,1,100)
    ys = LinRange(-1,1,100)
    zs = [0.5*x^2+0.5*b*z^2 for x in xs, z in ys]
    figure = surface(xs,ys,zs)
    save("$b zig zag.png", figure)

    points = Node(Point2f0([(y[1], y[2])])
    xvector = Float64[]
    yvector = Float64[]

    fig, ax, sc = scatter(points)
    contour!(xs,ys,zs;levels = 10)
    limits!(ax, -1, 1, -1, 1)
    frames = 1:size(y,1)
    record(fig, "$b zigzag_animation_GD_2D.mp4", frames; framerate = 1) do frame
        new_point = Point2f0(y[frame,1], y[frame,2])
        points[] = push!(points[], new_point)
        xvector = push!(xvector,Float64(y[frame,1]))
        yvector = push!(yvector,Float64(y[frame,2]))
        lines!(xvector, yvector, color:=green, linewidth = 3)
    end
```
For the Next Course

• Prepare animated plots for the quadratic example using:
  • Momentum and Nesterov acceleration for various $s,b$

• Prepare animated plots for SGD and various schemes
  • Randomized, Cyclic rule and mini batch
Ερωτήσεις