Optimization 2

CS5240 Theoretical Foundations in Multimedia

Leow Wee Kheng

Department of Computer Science
School of Computing
National University of Singapore
Gradient descent updates variables by

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

(1)

Gradient descent can be slow. To increase convergence rate, find good step length \( \alpha \) using line search.
Gradient descent minimizes $f(x)$ by updating $x_k$ along $-\nabla f(x_k)$:

$$x_{k+1} = x_k - \alpha \nabla f(x_k).$$  \hspace{2cm} (2)

In general, other descent directions $p_k$ are possible if $\nabla f(X)^\top p_k < 0$. Then, update $x_k$ according to

$$x_{k+1} = x_k + \alpha p_k.$$  \hspace{2cm} (3)

Ideally, want to find $\alpha > 0$ that minimizes $f(x_k + \alpha p_k)$. Too expensive.

In practice, find $\alpha > 0$ that reduces $f(x)$ sufficiently (not too small). The simple condition $f(x_k + \alpha p_k) < f(x_k)$ may not be sufficient.
A popular sufficient line search condition is Armijo condition:

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f(x_k)^{\top} p_k,$$  \hspace{1cm} (4)

where $c_1 > 0$, usually quite small, say, $c_1 = 10^{-4}$.

Armijo condition may accepts very small $\alpha$.

So, add another condition to ensure sufficient reduction of $f(x)$:

$$\nabla f(x_k + \alpha_k p_k)^{\top} p_k \geq c_2 \nabla f(x)^{\top} p_k,$$  \hspace{1cm} (5)

with $0 < c_1 < c_2 \leq 1$, typically $c_2 = 0.9$.

These two conditions together are called **Wolfe conditions**.

Known result:
Gradient descent will find local minimum if $\alpha$ satisfies Wolfe conditions.
Another way is to bound $\alpha$ from below, giving \textbf{Goldstein conditions}:

$$f(x_k) + (1-c)\alpha \nabla f(x_k)^\top p_k \leq f(x_k + \alpha p_k) \leq f(x_k) + c \alpha \nabla f(x_k)^\top p_k. \quad (6)$$

Another simple way is to start with large $\alpha$ and reduce it iteratively:

\textbf{Backtracking Line Search}

1. Choose large $\alpha > 0$, $0 < \rho < 1$.
2. Repeat until Armijo condition is met:
   2.1 $\alpha \leftarrow \rho \alpha$.

In this case, Armijo condition alone is enough.
Gauss-Newton Method

Carl Friedrich Gauss

Sir Isaac Newton
Gauss-Newton method estimates Hessian $\mathbf{H}$ by 1st-order derivatives.

Express $E(\mathbf{a})$ as the sum of square of $r_i(\mathbf{a})$,

$$E(\mathbf{a}) = \sum_{i=1}^{n} r_i(\mathbf{a})^2,$$

(7)

where $r_i$ is the error or residual of each data point $\mathbf{x}_i$:

$$r_i(\mathbf{a}) = v_i - f(\mathbf{x}_i; \mathbf{a}),$$

(8)

and

$$\mathbf{r} = [r_1 \cdots r_n]^\top.$$

(9)

Then,

$$\frac{\partial E}{\partial a_j} = 2 \sum_{i=1}^{n} r_i \frac{\partial r_i}{\partial a_j},$$

(10)

$$\frac{\partial^2 E}{\partial a_j \partial a_l} = 2 \sum_{i=1}^{n} \left[ \frac{\partial r_i}{\partial a_j} \frac{\partial r_i}{\partial a_l} + r_i \frac{\partial^2 r_i}{\partial a_j \partial a_l} \right] \approx 2 \sum_{i=1}^{n} \frac{\partial r_i}{\partial a_j} \frac{\partial r_i}{\partial a_l}.$$  

(11)
The derivative \( \frac{\partial r_i}{\partial a_j} \) is the \((i, j)\)-th element of the Jacobian \( J_r \) of \( r \).

\[
J_r = \begin{bmatrix}
\frac{\partial r_1}{\partial a_1} & \ldots & \frac{\partial r_1}{\partial a_m} \\
\vdots & \ddots & \vdots \\
\frac{\partial r_n}{\partial a_1} & \ldots & \frac{\partial r_n}{\partial a_m}
\end{bmatrix}.
\] (12)

The gradient and Hessian are related to the Jacobian by

\[
\nabla E = 2 J_r^\top r.
\] (13)

\[
H \approx 2 J_r^\top J_r,
\] (14)

So, the update equation becomes

\[
a_{k+1} = a_k - (J_r^\top J_r)^{-1} J_r^\top r.
\] (15)
We can also use the Jacobian $\mathbf{J}_f$ of $f(\mathbf{x}_i)$:

$$
\mathbf{J}_f = \begin{bmatrix}
\frac{\partial f(\mathbf{x}_1)}{\partial a_1} & \cdots & \frac{\partial f(\mathbf{x}_1)}{\partial a_m} \\
\vdots & \ddots & \vdots \\
\frac{\partial f(\mathbf{x}_n)}{\partial a_1} & \cdots & \frac{\partial f(\mathbf{x}_n)}{\partial a_m}
\end{bmatrix}.
$$

Then, $\mathbf{J}_f = -\mathbf{J}_r$, and the update equation becomes

$$
a_{k+1} = a_k + (\mathbf{J}_f^\top \mathbf{J}_f)^{-1} \mathbf{J}_f^\top \mathbf{r}.
$$
Gauss-Newton Method

Compare to linear fitting:

\[
\text{linear fitting} \quad a = (D^\top D)^{-1}D^\top v
\]

\[
\text{Gauss-Newton} \quad \Delta a = (J_f^\top J_f)^{-1}J_f^\top r
\]

\((J_f^\top J_f)^{-1}J_f^\top\) is analogous to pseudo-inverse.

Compare to gradient descent:

\[
\text{gradient descent} \quad \Delta a = -\alpha \nabla E
\]

\[
\text{Gauss-Newton} \quad \Delta a = -\frac{1}{2}(J_r^\top J_r)^{-1}\nabla E
\]

Gauss-Newton updates \(a\) along negative gradient at variable rate.

Gauss-Newton method may converge slowly or diverge if

- initial guess \(a(0)\) is far from minimum, or
- matrix \(J^\top J\) is ill-conditioned.
The condition number of (real) matrix $\mathbf{D}$ is given by the ratio of its singular values:

$$\kappa(\mathbf{D}) = \frac{\sigma_{\text{max}}(\mathbf{D})}{\sigma_{\text{min}}(\mathbf{D})}. \quad (18)$$

If $\mathbf{D}$ is normal, i.e., $\mathbf{D}^\top \mathbf{D} = \mathbf{D} \mathbf{D}^\top$, then its condition number is given by the ratio of its eigenvalues:

$$\kappa(\mathbf{D}) = \frac{\lambda_{\text{max}}(\mathbf{D})}{\lambda_{\text{min}}(\mathbf{D})}. \quad (19)$$

Large condition number means the elements of $\mathbf{D}$ are less balanced.
For the linear equation

$$Da = v,$$

the **condition number** indicates change of solution $a$ vs. change of $v$.

- **Small condition number:**
  - Small change in $v$ gives small change in $a$: **stable**.
  - Small error in $v$ gives small error in $a$.
  - $D$ is **well-conditioned**.

- **Large condition number:**
  - Small change in $v$ gives large change in $a$: **unstable**.
  - Small error in $v$ gives large error in $a$.
  - $D$ is **ill-conditioned**.
Practical Issues

The update equation, with $J = J_f$,

$$a_{k+1} = a_k + (J^\top J)^{-1}J^\top r.$$  \hfill(20)

requires computation of matrix inverse, which can be expensive. Can use singular value decomposition (SVD) to help.

SVD of $J$ is

$$J = U \Sigma V^\top.$$  \hfill(21)

Substituting it into the update equation yields  \hfill(Homework)

$$a_{k+1} = a_k + V \Sigma^{-1} U^\top r.$$  \hfill(22)
Convergence criteria: same as gradient descent

\[ \frac{|E_{k+1} - E_k|}{E_k} < \epsilon, \text{ e.g., } \epsilon = 0.0001. \]

\[ \frac{|a_{j,k+1} - a_{j,k}|}{|a_{j,k}|} < \epsilon, \text{ for each component } j. \]

If divergence occurs, can introduce step length \( \alpha \):

\[ a_{k+1} = a_k + \alpha (J^\top J)^{-1} J^\top r, \quad (23) \]

where \( 0 < \alpha < 1 \).

Is there an alternative way to control convergence? Yes: Levenberg-Marquardt Method.
Levenberg-Marquardt Method

Gauss-Newton method updates $\mathbf{a}$ according to

$$(\mathbf{J}^\top \mathbf{J})\Delta \mathbf{a} = \mathbf{J}^\top \mathbf{r}. \quad (24)$$

Levenberg modifies the update equation to

$$(\mathbf{J}^\top \mathbf{J} + \lambda \mathbf{I})\Delta \mathbf{a} = \mathbf{J}^\top \mathbf{r}. \quad (25)$$

- $\lambda$ is adjusted at each iteration.
- If reduction of error $E$ is large, can use small $\lambda$: more similar to Gauss-Newton.
- If reduction of error $E$ is small, can use large $\lambda$: more similar to gradient descent.
If $\lambda$ is very large, $(\mathbf{J}^\top \mathbf{J} + \lambda \mathbf{I}) \approx \lambda \mathbf{I}$, i.e., just doing gradient descent.

Marquardt scales each component of gradient:

$$(\mathbf{J}^\top \mathbf{J} + \lambda \text{diag}(\mathbf{J}^\top \mathbf{J})) \mathbf{\Delta a} = \mathbf{J}^\top \mathbf{r}, \quad (26)$$

where $\text{diag}(\mathbf{J}^\top \mathbf{J})$ is a diagonal matrix of the diagonal elements of $\mathbf{J}^\top \mathbf{J}$.

The $(i, i)$-th component of $\text{diag}(\mathbf{J}^\top \mathbf{J})$ is

$$\sum_{l=1}^{n} \left( \frac{\partial f(x_l)}{\partial a_i} \right)^2. \quad (27)$$

- If $(i, i)$-th component of gradient is small, $a_i$ is updated more: faster convergence along $i$-th direction.
- Then, $\lambda$ doesn’t have to be too large.
Quasi-Newton

Quasi-Newton also minimizes $f(x)$ by iteratively updating $x$:

$$x_{k+1} = x_k + \alpha p_k$$  \hspace{1cm} (28)

where the descent direction $p_k$ is

$$p_k = -H_k^{-1} \nabla f(x_k).$$  \hspace{1cm} (29)

There are many methods of estimating $H^{-1}$.

The most effective is the method by Broyden, Fletcher, Goldfarb and Shanno (BFGS).
**BFGS Method**

Let \( \Delta x_k = x_{k+1} - x_k \) and \( y_k = \nabla f(x_{k+1}) - \nabla f(x_k) \).

Then, estimate \( H_k \) or \( H_k^{-1} \) by

\[
H_{k+1} = H_k - \frac{H_k \Delta x_k \Delta x_k^\top H_k}{\Delta x_k^\top H_k \Delta x_k} + \frac{y_k y_k^\top}{y_k^\top \Delta x_k},
\]

\[
H_{k+1}^{-1} = \left( I - \frac{\Delta x_k y_k^\top}{y_k^\top \Delta x_k} \right) H_k^{-1} \left( I - \frac{y_k \Delta x_k^\top}{y_k^\top \Delta x_k} \right) + \frac{\Delta x_k \Delta x_k^\top}{y_k^\top \Delta x_k}.
\]

(For derivation, please refer to [1].)
BFGS Algorithm

1. Set $\alpha$ and initialize $x_0$ and $H_0^{-1}$.

2. For $k = 0, 1, \ldots$ until convergence:

   2.1 Compute search direction

   $$p_k = -H_k^{-1}\nabla f(x_k).$$

   2.2 Perform line search to find $\alpha$ that satisfies Wolfe conditions.

   2.3 Compute $\Delta x_k = \alpha p_k$, and set $x_{k+1} = x_k + \Delta x_k$.

   2.4 Compute $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$.

   2.5 Compute $H_{k+1}^{-1}$ by Eq. 31.

- No standard way to initialize $H$. Can try $H = I$. 
Approximation of Gradient

All previous methods require the gradient $\nabla f$ (or $\nabla E$). In many complex problems, it is very difficult to derive $\nabla f$.

Several methods to overcome this problem:

- Use **symbolic differentiation** to derive math expression of $\nabla f$. Provided in Matlab, Maple, etc.
- Approximate gradient numerically.
- Optimize without gradient.
Approximation of Gradient

**Forward Difference**

From Taylor series expansion,

\[
f(x + \Delta x) = f(x) + \nabla f(x)^\top \Delta x + \cdots
\]  \hspace{1cm} (32)

Denote the \(j\)-th component of \(\Delta x\) as \(\epsilon e_j\), where \(e_j\) is the unit vector of the \(j\)-th dimension. Then, the \(j\)-th component of \(\nabla f(x)\) is

\[
\frac{\partial f(x)}{\partial x_j} \approx \frac{f(x + \epsilon e_j) - f(x)}{\epsilon}.
\]  \hspace{1cm} (33)

Note that \(x + \epsilon e_j\) updates only the \(j\)-th component:

\[
x + \epsilon e_j = \begin{bmatrix} x_1 & \cdots & x_{j-1} & x_j & x_{j+1} & \cdots & x_m \end{bmatrix}^\top + \begin{bmatrix} 0 & \cdots & 0 & \epsilon & 0 & \cdots & 0 \end{bmatrix}^\top
\]  \hspace{1cm} (34)

Then,

\[
\nabla f(x) \approx \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} & \cdots & \frac{\partial f(x)}{\partial x_m} \end{bmatrix}^\top.
\]  \hspace{1cm} (35)
Central Difference

Forward difference method has forward bias: \( x + \epsilon e_j \).
Error in forward bias can accumulate quickly.

To be less biased, use central difference.

\[
f(x + \epsilon e_j) \approx f(x) + \epsilon \frac{\partial f(x)}{\partial x_j}
\]

\[
f(x - \epsilon e_j) \approx f(x) - \epsilon \frac{\partial f(x)}{\partial x_j}.
\]

So,

\[
\frac{\partial f(x)}{\partial x_j} \approx \frac{f(x + \epsilon e_j) - f(x - \epsilon e_j)}{2\epsilon}.
\] (36)
A simple idea is to descend along each coordinate direction iteratively. Let $e_1, \ldots, e_m$ denote the unit vectors along coordinates $1, \ldots, m$.

**Coordinate Descent**

1. Choose initial $x_0$.
2. Repeat until convergence:
   2.1 For each $j = 1, \ldots, m$:
      2.1.1 Find $x_{k+1}$ along $e_j$ that minimizes $f(x_k)$. 

![Diagram showing the coordinate descent process](image-url)
Alternative for Step 2.1.1:
Find $\mathbf{x}_{k+1}$ along $\mathbf{e}_j$ using line search to reduce $f(\mathbf{x}_k)$ sufficiently.

- **Strength:** Simplicity.
- **Weakness:** Can be slow.
  Can iterate around the minimum, never approaching it.

**Powell’s Method**

Overcomes the problem of coordinate descent.

**Basic Idea:**
Choose directions that do not interfere with or undo previous effort.
Alternating Optimization

Let $\mathcal{S} \subseteq \mathbb{R}^m$ denote the set of $\mathbf{x}$ that satisfy the constraints. Then, constrained optimization problem can be written as:

$$\min_{\mathbf{x} \in \mathcal{S}} f(\mathbf{x}).$$  \hfill (37)

Alternating optimization (AO) is similar to coordinate descent except that the variables $x_j, j = 1, \ldots, m$, are split into groups.

Partition $\mathbf{x} \in \mathcal{S}$ into $p$ non-overlapping parts $\mathbf{y}_l$ such that

$$\mathbf{x} = (\mathbf{y}_1, \ldots, \mathbf{y}_l, \ldots, \mathbf{y}_p).$$  \hfill (38)

Example: $\mathbf{x} = (x_1, x_2, x_4, x_3, x_7, x_5, x_6, x_8, x_9)$.

Denote $\mathcal{S}_l \subset \mathcal{S}$ as the set that contains $\mathbf{y}_l$. 
Alternating Optimization

1. Choose initial $x^{(0)} = (y_1^{(0)}, \ldots, y_p^{(0)})$.

2. For $k = 0, 1, 2, \ldots$, until convergence:
    
    2.1 For $l = 1, \ldots, p$:
        
        2.1.1 Find $y_l^{(k+1)}$ such that

        $$y_l^{(k+1)} = \arg \min_{y_l \in S_l} f(y_1^{(k+1)}, \ldots, y_{l-1}^{(k+1)}, y_l, y_{l+1}^{(k)}, \ldots, y_p^{(k)}).$$

        That is, find $y_l^{(k+1)}$ while fixing other variables.

        (Alternative) Find $y_l^{(k+1)}$ that decreases $f(x)$ sufficiently:

        $$f(y_1^{(k+1)}, \ldots, y_{l-1}^{(k+1)}, y_l^{(k)}, y_{l+1}^{(k)}, \ldots, y_p^{(k)}) - f(y_1^{(k+1)}, \ldots, y_{l-1}^{(k+1)}, y_l^{(k+1)}, y_{l+1}^{(k)}, \ldots, y_p^{(k)}) > c > 0.$$
Example:

Initially \( (y_1^{(0)}, y_2^{(0)}, y_3^{(0)}, \ldots, y_l^{(0)}, \ldots, y_p^{(0)}) \)

\[ \downarrow \]

\( k = 0, l = 1 \)
\( (y_1^{(1)}, y_2^{(0)}, y_3^{(0)}, \ldots, y_l^{(0)}, \ldots, y_p^{(0)}) \)

\[ \downarrow \]

\( k = 0, l = 2 \)
\( (y_1^{(1)}, y_2^{(1)}, y_3^{(0)}, \ldots, y_l^{(0)}, \ldots, y_p^{(0)}) \)

\[ \downarrow \]

\( k = 0, l = p \)
\( (y_1^{(1)}, y_2^{(1)}, y_3^{(1)}, \ldots, y_l^{(1)}, \ldots, y_p^{(1)}) \)

\[ \downarrow \]

\( k = 1, l = 1 \)
\( (y_1^{(2)}, y_2^{(1)}, y_3^{(1)}, \ldots, y_l^{(1)}, \ldots, y_p^{(1)}) \)

\[ \vdots \]

Obviously, we want to partition the variables so that

- Each sub-problem of finding \( y_l \) is relatively easy to solve.
- Solving \( y_l \) does not undo previous partial solutions.
Example: $k$-means clustering

$k$-means clustering tries to group data points $\mathbf{x}$ into $k$ clusters.

Let $\mathbf{c}_j$ denote the prototype of cluster $C_j$.
Then, $k$-means clustering finds $C_j$ and $\mathbf{c}_j$, $j = 1, \ldots, k$, that minimize within-cluster difference:

$$D = \sum_{j=1}^{k} \sum_{\mathbf{x} \in C_j} (\mathbf{x} - \mathbf{c}_j)^2.$$ (39)
Suppose $C_j$ are already known. Then, the $c_j$ that minimize $D$ is given by

$$\frac{\partial D}{\partial c_j} = 2 \sum_{x \in C_j} (x - c_j) = 0. \quad (40)$$

That is, $c_j$ is the centroid of $C_j$:

$$c_j = \frac{1}{|C_j|} \sum_{x \in C_j} x. \quad (41)$$

How to find $C_j$?
Use alternating optimization.
**k-means clustering**

1. Initialize $c_j, j = 1, \ldots, k$.
2. Repeat until convergence:
   1. For each $x$, assign it to the nearest cluster $C_j$:
      (fix $c_j$, update $C_j$)
      $$\|x - c_j\| \leq \|x - c_l\|, \text{ for } l \neq j.$$  
   2. Update cluster centroids:
      (fix $C_j$, update $c_j$)
      $$c_j = \frac{1}{|C_j|} \sum_{x \in C_j} x.$$  

Step 2.2 is (locally) optimal.
Step 2.1 does not undo step 2.2. Why?
So, $k$-means clustering can converge to a local minimum.
Convergence Rates

Suppose the sequence \( \{x_k\} \) converges to \( x^* \).

Then, the sequence converges to \( x^* \) **linearly** if \( \exists \mu \in (0, 1) \) such that

\[
\lim_{k \to \infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|} = \mu.
\]  

(42)

- The sequence converges **sublinearly** if \( \mu = 1 \).
- The sequence converges **superlinearly** if \( \mu = 0 \).

If the sequence converges sublinearly and

\[
\lim_{k \to \infty} \frac{|x_{k+2} - x_{k+1}|}{|x_{k+1} - x_k|} = 1,
\]  

(43)

then, the sequence converges **logarithmically**.
We can distinguish between different rates of superlinear convergence.

The sequence converges with order of $q$, for $q > 1$, if $\exists \mu > 0$ such that

$$
\lim_{k \to \infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^q} = \mu > 0.
$$

(44)

► If $q = 1$, the sequence converges \textit{q-linearly}.
► If $q = 2$, the sequence converges \textit{quadratically} or \textit{q-quadratically}.
► If $q = 3$, the sequence converges \textit{cubically} or \textit{q-cubically}.
► etc.
Known Convergence Rates

- Gradient descent converges linearly, often slowly linearly.
- Gauss-Newton converges quadratically, if it converges.
- Levenberg-Marquardt converges quadratically.
- Quasi-Newton converges superlinearly.
- Alternating optimization converges q-linearly if $f$ is convex around a local minimizer $\mathbf{x}^*$. 
<table>
<thead>
<tr>
<th>Method</th>
<th>For</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient descent</td>
<td>optimization</td>
<td>$\nabla f$</td>
</tr>
<tr>
<td>Gauss-Newton</td>
<td>nonlinear least-square</td>
<td>$\nabla E$, estimate of $H$</td>
</tr>
<tr>
<td>Levenberg-Marquardt</td>
<td>nonlinear least-square</td>
<td>$\nabla E$, estimate of $H$</td>
</tr>
<tr>
<td>quasi-Newton</td>
<td>optimization</td>
<td>$\nabla f$, estimate of $H$</td>
</tr>
<tr>
<td>Newton</td>
<td>optimization</td>
<td>$\nabla f$, $H$</td>
</tr>
<tr>
<td>coordinate descent</td>
<td>optimization</td>
<td>$f$</td>
</tr>
<tr>
<td>alternating optimization</td>
<td>optimization</td>
<td>$f$</td>
</tr>
</tbody>
</table>
SciPy supports many optimization methods including

- linear least-square
- Levenberg-Marquardt method
- quasi-Newton BFGS method
- line search that satisfies Wolfe conditions
- etc.

For more details and other methods, refer to [1].
Probing Questions

- Can you apply an algorithm specialized for nonlinear least square, e.g., Gauss-Newton, to generic optimization?

- Gauss-Newton may converge slowly or diverge if the matrix $J^TJ$ is ill-conditioned. How to make the matrix well-conditioned?

- How can your alternating optimization program detect interference of previous update by current update?

- If your alternating optimization program detects frequent interference between updates, what can you do about it?

- So far, we have look at minimizing a scalar function $f(x)$. How to minimize a vector function $f(x)$ or multiple scalar functions $f_i(x)$ simultaneously?

- How to ensure that your optimization program will always converge regardless of the test condition?
1. Describe the essence of Gauss-Newton, Levenberg-Marquardt, quasi-Newton, coordinate descent and alternating optimization each in one sentence.

2. With SVD of $J$ given by $J = U \Sigma V^\top$, show that

\[(J^\top J)^{-1} J^\top = V \Sigma^{-1} U^\top.\]

Hint: For matrices $A$ and $B$, $(AB)^{-1} = B^{-1} A^{-1}$.

3. Q3(c) of AY2015/16 Final Evaluation.

References


   www.nr.com