Optimization Methods — Date: *03-04-2018*

Instructor: *Dr. Sujit Gujar*

Scribes: *Pritish Moulik*, *Alefiah Mubeen*, *Ayush Deva*

# Lecture 21: Conjugate Gradient and Quasi Newton Methods

## 1 Recap

**Conjugate Gradient Method**

Conjugate gradient method is an algorithm to find solution for unconstrained optimization problems of the form:

$$\min_{x \in R^n} f(x) = \frac{1}{2}x^T Q x - x^T b,$$

where $Q$ is positive-definite and symmetric.
Let $d_1 ... d_n$ be conjugate directions for $Q$.
In Gram-Schmidt Method,

$$d_{k+1} = x_{k+1} - \sum_{i<k} \frac{x_{k+1}^T Q d_i}{d_k^T Q d_k} d_i$$

This method is space inefficient as it stores $d_1 ... d_k$ to compute $d_{k+1}$. So, instead of Gram-Schmidt approach, we use the following algorithm (Algorithm 1) to generate the conjugate directions $d_1, d_2, ... d_n$.

**Algorithm 1** Conjugate Gradient Method

```
1: $k \leftarrow 0$
2: $x_0 \leftarrow 0$
3: $r_0 \leftarrow -\nabla f(x_o) = b - Qx_0 = b$
4: if $r_0 = 0$ then STOP
5: $d_0 \leftarrow r_0$
6: $\alpha_k \leftarrow \frac{r_k^T d_k}{d_k^T Q d_k}$
7: $x_{k+1} \leftarrow x_k + \alpha_k d_k$.
8: if $r_{k+1} = b - Qx_{k+1} = 0$ then STOP.
9: $\beta_k \leftarrow -\frac{r_{k+1}^T Q d_k}{d_k^T Q d_k}$.
10: $d_{k+1} \leftarrow r_{k+1} + \beta_k d_k$
11: $k \leftarrow k + 1$.
12: goto 6.
```

## 2 Concepts

The following topics were covered in this lecture:

1. Lemma corresponding to Q-Conjugate directions.
2. Theorem and proof for Q-Conjugate.
3. BFGS Algorithm and its properties along with comparison with L-BFGS and SVM.
4. Overview of Adam Optimizer (Stochastic Gradient Descent)

## 3 Proofs

**Lemma 3.1.** In Conjugate Gradient Method, if $d_0$, $d_1$, ... $d_n$ are Q-conjugate directions, then

$$r_{k+1}^T d_i = 0 \qquad \forall i = 0, 1, 2...k$$

*Proof.* We shall prove this using induction.

1. Lets start with $r_1 (i = 0)$.

$$\begin{aligned} r_1 &= b - Qx_1 \\ &= r_0 - Qx_1 \\ &= r_0 - Q(x_0 + \alpha_0 d_0) \\ &= r_0 - Q(\alpha_0 d_0) \qquad\qquad \because x_0 = 0 \end{aligned}$$

$$\begin{aligned} r_1^T d_0 &= r_0^T d_0 - \alpha_0 d_0^T Q d_0, \\ \alpha_0 &= \frac{r_0^T d_0}{d_0^T Q d_0}. \\ \implies r_1^T d_0 &= 0 \end{aligned}$$

2. Let's apply induction on $i = k$.

$$r_{k+1}^T d_i = r_k^T d_i - \alpha_k d_k^T Q d_i$$

Two cases to consider - (a) $i < k$, (b) $i = k$.

(a) if $i < k$,

$$\begin{aligned} r_k^T d_i &= 0 && (induction\ hypothesis) \\ d_k^T Q d_i &= 0 && (given) \\ \therefore r_{k+1}^T d_i &= 0 && (thus,\ true\ for\ i < k) \end{aligned}$$

(b) if $i = k$,

$$\begin{aligned} r_{k+1}^T d_k &= r_k^T d_k - \alpha_k d_k^T Q d_k \\ \alpha_k &= \frac{r_k^T d_k}{d_k^T Q d_k} \\ \therefore r_{k+1}^T d_k &= 0 \qquad\qquad (thus,\ true\ for\ i = k) \end{aligned}$$

Hence, proved. □

**Lemma 3.2.** If $r_k$ denotes the residue in the $k^{th}$ iteration of Conjugate Gradient Descent Method, then we have, $\forall k$,

$$r_{k+1}^T r_j = 0 \qquad \forall j = 0, 1, 2, ...k$$

*Proof.* We know that,

$$\begin{aligned} d_j &= r_j + \beta_k d_{j-1} \\ r_j &= d_j - \beta_k d_{j-1} \end{aligned}$$

Pre-multiplying with $r_{k+1}^T$ on both sides, we get,

$$\begin{aligned} r_{k+1}^T r_j &= r_{k+1}^T d_j - r_{k+1}^T \beta_k d_{j-1} \\ &= 0 + 0 \qquad \because \text{Lemma 3.1} \\ \therefore r_{k+1}^T r_j &= 0 \end{aligned}$$

Hence, proved. □

**Theorem 3.3.** Let $d_0, d_1, d_2, ...d_{n-1}$ be the directions generated in conjugate gradient method. These directions are Q-conjugate, i.e.,

$$d_i^T Q d_j = 0 \qquad \forall i, j \text{ and } i \neq j$$

*Proof.* We prove this using induction.

1. **Base Case** : $\qquad d_1^T Q d_0 = 0 \qquad$ (Trivially)

2. Assume $d_0, d_1, d_2, ...d_k$ are Q-conjugate, then we need to prove that $d_0, d_1, d_2, ...d_{k+1}$ are also Q-conjugate, i.e.,

$$d_{k+1}^T Q d_j$$

**Case A** : j = k

$$\begin{aligned} d_{k+1}^T Q d_k &= (r_{k+1} + \beta_k d_k)^T Q d_k \\ &= r_{k+1}^T Q d_k + \beta_k d_k^T Q d_k \\ &= 0 \qquad \because \beta_k = \frac{-r_{k+1}^T Q d_k}{d_k^T Q d_k} \end{aligned}$$

**Case B** : j < k

$$\begin{aligned} r_{j+1} &= b - Q x_{j+1} \\ r_j &= b - Q x_j \\ r_{j+1} - r_j &= Q(x_j - x_{j+1}) \\ &= -\alpha_j Q d_j \\ \therefore \qquad Q d_j &= \frac{r_{j+1} - r_j}{-\alpha_j} \end{aligned}$$

$$\begin{aligned} d_{k+1}^T Q d_j &= d_{k+1}^T \frac{r_{j+1} - r_j}{-\alpha_j} \\ &= (r_{k+1} + \beta_k d_k)^T \frac{r_{j+1} - r_j}{-\alpha_j} \\ &= \frac{r_{k+1}^T (r_{j+1} - r_j)}{-\alpha_j} + \frac{\beta_k d_k (r_{j+1} - r_j)}{-\alpha_j} \\ &= \frac{r_{k+1}^T (r_{j+1} - r_j)}{-\alpha_j} + \beta_k d_k^T Q d_j \end{aligned}$$

Now,

1. $r_{k+1}^T r_{j+1} = r_{k+1}^T r_j = 0 \qquad$ (by Lemma 2)

2. $d_k^T Q d_j = 0 \qquad$ (by induction hypothesis)

$$\therefore d_{k+1}^T Q d_j = 0$$

Hence, proved. □

## 4 Quasi-Newton Methods

According to Newton's method, we have,

1. $x_{k+1} = x_k + H_k^{-1}\nabla f(x_k)$ where $H_k$ is the Hessian matrix.
2. $f(x+\Delta) = f(x_k) + \Delta^T \nabla f(x_k) + \frac{1}{2}\Delta^T H_k \Delta$ (Taylor Series Expansion)

But, calculating $H_k$ at every iteration is computationally challenging. Instead, we use another matrix $B_k$ as follows :

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

where $\alpha_k$ must satisfy the Armijo-Wolfe conditions.

Since we are now using a matrix different from the Hessian, we must be able to define it for each iteration. After suitable initialization, the matrix $B_k$ is updated using one of the following two rules:

1. **Rank One Update Rule** :
$$B_{k+1} = B_k + U_k$$
where $U_k$ is a rank one matrix. Since every rank one matrix can be written as a product of a column and a row vector, thus, we have,
$$U_k = y_k y_k^T \qquad \text{for some } y_k$$
2. **Rank Two Update Rule** : Rank One Update Rule worked fine for quadratic programming for suitable chosen $y_k$ but not for higher degree polynomials. Hence, a rank two update rule was proposed as follows :
$$B_{k+1} = B_k + U_k + V_k$$
where both $U_k$ and $V_k$ are rank one matrices.

Such methods are known as **Quasi-Newton Methods** due to their similarity to Newton's methods.

## 5 Broyden-Fletcher-Goldfarb-Shanno (BFGS) Algorithm

In numerical optimization, the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm is an iterative method for solving unconstrained nonlinear optimization problems.[1]

The BFGS method belongs to quasi-Newton methods, a class of hill-climbing optimization techniques that seek a stationary point of a (preferably twice continuously differentiable) function. For such problems, a necessary condition for optimality is that the gradient be zero. Newton's method and the BFGS methods are not guaranteed to converge unless the function has a quadratic Taylor expansion near an optimum. However, BFGS has proven to have good performance even for non-smooth optimizations.[2] It is most widely used in machine learning algorithms.

The algorithm is as follows:

From an initial guess $\mathbf{x}_0$ and an approximate Hessian matrix $B_0$ the following steps are repeated as $\mathbf{x}_k$ converges to the solution.

**Algorithm 2** BFGS

```
1: $B_0 = H(x_0)$ where $d_0 = -\beta_0^{-1}\nabla f(x_0)$ and k=0
2: $\alpha_k$ satisfyies the Armijo-Wolfe condition
3: $S_k = \alpha_k d_k$ obtain a direction by solving $d_k$
4: $x_{k+1} = x_k - \alpha_k B_k^{-1}\nabla f(x_k)$
5: $y_{k+1} = \nabla f(x_{k+1}) - \nabla f(x_k)$
6: $B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T S_k} - \frac{(B_k S_k)(S_k^T B_k)}{S_k^T B_k S_k}$
```

$f(x)$ denotes the objective function to be minimized. Convergence can be checked by observing the norm of the gradient,$||\nabla f(\mathbf{x}_k)||$. Practically, $B_0$ can be initialized with $B_0 = I$, so that the first step will be equivalent to a gradient descent, but further steps are more and more refined by $B_k$, the approximation to the Hessian.

The first step of the algorithm is carried out using the inverse of the matrix $B_k$, which can be obtained efficiently by applying the Sherman-Morrison[3] formula to the step 6 of the algorithm, giving

$$B_{k+1}^{-1} = \left(I - \frac{s_k y_k^T}{y_k^T s_k}\right) B_k^{-1} \left(I - \frac{y_k s_k^T}{y_k^T s_k}\right) + \frac{s_k s_k^T}{y_k^T s_k}.$$

This can be computed efficiently without temporary matrices, recognizing that $B_k^{-1}$ is symmetric, and that $\mathbf{y}_k^{\mathrm{T}} B_k^{-1}\mathbf{y}_k$ and $\mathbf{s}_k^{\mathrm{T}}\mathbf{y}_k$ are scalar, using an expansion such as

$$B_{k+1}^{-1} = B_k^{-1} + \frac{(\mathbf{s}_k^{\mathrm{T}}\mathbf{y}_k + \mathbf{y}_k^{\mathrm{T}} B_k^{-1}\mathbf{y}_k)(\mathbf{s}_k\mathbf{s}_k^{\mathrm{T}})}{(\mathbf{s}_k^{\mathrm{T}}\mathbf{y}_k)^2} - \frac{B_k^{-1}\mathbf{y}_k\mathbf{s}_k^{\mathrm{T}} + \mathbf{s}_k\mathbf{y}_k^{\mathrm{T}} B_k^{-1}}{\mathbf{s}_k^{\mathrm{T}}\mathbf{y}_k}..$$

In statistical estimation problems (such as maximum likelihood or Bayesian inference), credible intervals or confidence intervals for the solution can be estimated from the inverse of the final Hessian matrix. However, these quantities are technically defined by the true Hessian matrix, and the BFGS approximation may not converge to the true Hessian matrix.

## 5.1 Comparisons

L-BFGS is an approximation to BFGS, one which requires a lot less memory. BFGS computes and stores the full Hessian H at each step; this requires $\Theta(n^2)$ space, where n counts the number of variables (dimensions) that you're optimizing over. L-BFGS computes and stores an approximation to the Hessian, chosen so that the approximation can be stored in $\Theta$(n) space. Effectively, L-BFGS uses the approximation H$\approx$ $M^T$M for some k by n matrix M.

Each step of L-BFGS is an attempt at approximating/guessing what the corresponding step of BFGS would do. However, a single step of L-BFGS takes a lot less space and time than a single step of BFGS. Consequently, you can do many more steps of L-BFGS within a particular time bound than BFGS. Therefore, you might find that L-BFGS converges faster, because it can do so many more iterations within a given amount of time than BFGS can. Support Vector Machine (SVM) will have constraints on datapoints so it will be constrained non linear optimization In BFGS rank-2 update is done, inverse is close form solution as only for the first time we find inverse hence for higher degree terms it performs better.

## 6 Adam Optimizer (Stochastic Gradient Descent)

Adam is an optimization algorithm that can used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. a method for efficient stochastic optimization that only requires first-order gra- dients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name Adam is derived from adaptive moment estimation.[4]

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

where $\nabla f_i(x)$ takes any error function for any point.

$$\frac{1}{2}\sum_{i=1}^{n} g_i(x)^2$$

$$\nabla f(x) = \sum g_i(x) \nabla g_i(x)$$

For this error function

$$x_{k+1} = x_k + \alpha_k g_i(x) \nabla g_i(x) \qquad \text{for some randomly chosen } i$$

Here only calculate gradient for that error point/function. Sometimes instead of a single point, we use a batch of points.

## References

[1] *Fletcher, Roger (1987). Practical methods of optimization (2nd ed.), New York:.* John Wiley and Sons, ISBN 978-0-471-91547-8

[2] *Lewis, Adrian S.; Overton, Michael (2009),. Nonsmooth optimization via BFGS.*

[3] WIKI Source, https://en.wikipedia.org/wiki/Sherman_Morrison_formula

[4] *Diederik P. Kingma, Jimmy Ba Adam: A Method for Stochastic Optimization (2015) 3rd International Conference for Learning Representations, San Diego*

[5] WIKI Source, https://en.wikipedia.org/wiki/Limited-memory_BFGS

[6] The lib for L-BFGS implementation, https://en.wikipedia.org/wiki/SciPy

[7] Numerical Optimization, http://www.bioinfo.org.cn/~wangchao/maa/Numerical_Optimization.pdf

# Appendix

**Sherman-Morrison Formula**

## A Statement

Suppose $A \in \mathbb{R}^{n\times n}$ is an invertible square matrix and $u$,$v \in \mathbb{R}^n$ are column vectors. Then $A + uv^T$is invertible iff $1 + v^T A^{-1} u \neq 0$.
If $A + uv^T$is invertible, then its inverse is given by

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1}u}$$

Here, $uv^T$ is the outer product of two vectors $u$ and $v$.

## B Proof

To prove that the backward direction( $1 + v^T A^{-1}u \neq 0 \Rightarrow A + uv^T$ is invertible with inverse given as above) is true, we verify the properties of the inverse. A matrix $Y$ (in this case the right-hand side of the Sherman-Morrison formula) is the inverse of a matrix $X$ (in this case $A + uv^T$) if and only if $XY = YX = I$.
We first verify that the right hand side ( $Y$ ) satisfies $XY = I$

$$\begin{aligned}
XY &= (A + uv^T)\left(A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1}u}\right) \\
&= AA^{-1} + uv^T A^{-1} - \frac{AA^{-1}uv^T A^{-1} + uv^T A^{-1}uv^T A^{-1}}{1 + v^T A^{-1}u} \\
&= I + uv^T A^{-1} - \frac{uv^T A^{-1} + uv^T A^{-1}uv^T A^{-1}}{1 + v^T A^{-1}u} \\
&= I + uv^T A^{-1} - \frac{u(1 + v^T A^{-1}u)v^T A^{-1}}{1 + v^T A^{-1}u} \\
&= I + uv^T A^{-1} - uv^T A^{-1} \qquad 1 + v^T A^{-1}u \text{ is a scalar} \\
&= I
\end{aligned}$$

In the same way, one can verify that: $YX = \left(A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1}u}\right)(A + uv^T) = I$.
($\Rightarrow$) To prove the reverse direction, we suppose that u âĽă 0 $u \neq 0$, otherwise the result is trivial. Then,
$(A + uv^T)A^{-1}u = u + uv^T A^{-1}u = (1 + v^T A^{-1}u)u$.

Since $A + uv^T$is assumed invertible, $(A + uv^T)A^{-1}$ is invertible as the product of invertible matrices. Thus, by our assumption that $u \neq 0$, we have that $(A + uv^T)A^{-1}u \neq 0$. By the identity above, this means that$(1 + v^T A^{-1}u)u \neq 0$ , and hence $1 + v^T A^{-1}u \neq 0$, as was to be shown.

## C Application

If the inverse of $A$ is already known, the formula provides a numerically cheap way to compute the inverse of $A$ corrected by the matrix $uv^T$ (depending on the point of view, the correction may be seen as a perturbation or as a rank-1 update). The computation is relatively cheap because the inverse of $A + uv^T$ does not have to be computed from scratch (which in general is expensive), but can be computed by correcting (or perturbing)$A^{-1}$.
Using unit columns (columns from the identity matrix) for $u$ or $v$, individual columns or rows of $A$ may be manipulated and a correspondingly updated inverse computed relatively cheaply in this way.In the general case, where $A^{-1}$ is a $n$ - by - $n$ matrix and $u$ and $v$ are arbitrary vectors of dimension $n$, the whole matrix is updated and the computation takes $3n^2$ scalar multiplications.If $u$ is a unit column, the computation takes only $2n^2$ scalar multiplications. The same goes if $v$ is a unit column. If both u $u$and $v$ are unit columns, the computation takes only $n^2$ scalar multiplications.