Chapter 2 Basic Concepts and Optimization Algorithms

We recall the instances in Chapter 1 where mathematical optimization was encountered in several ML applications:

- PCA-based feature extraction:

$$\begin{aligned} \underset{\boldsymbol{U}_q,\,\boldsymbol{H}_q}{\text{minimize}} \quad & \| \boldsymbol{X} - \boldsymbol{U}_q \cdot \boldsymbol{H}_q \|_F \\ \text{subject to:} \quad & \boldsymbol{U}_q^T \boldsymbol{U}_q = \boldsymbol{I}_q \end{aligned}$$

- K-means clustering:

$$\begin{aligned} \underset{\{r_{n,k}\},\,\{\boldsymbol{\mu}_k\}}{\text{minimize}} \quad & \sum_{n=1}^{N}\sum_{k=1}^{K} r_{n,k} \| \boldsymbol{x}_n - \boldsymbol{\mu}_k \|_2^2 \\ \text{subject to:} \quad & r_{n,k} \in \{0,1\} \;\text{ for }\; n = 1,2,...,N,\;\; k = 1,2,..,K \\ & \textstyle\sum_{k=1}^{K} r_{n,k} = 1 \;\text{ for }\; k = 1,2,..,K \end{aligned}$$

- Constructing a linear model for prediction:

$$\underset{\boldsymbol{W},\boldsymbol{b}}{\text{minimize}} \quad \sum_{n=1}^{N} \left\| \left(\boldsymbol{W}^T \boldsymbol{x}_n + \boldsymbol{b} \right) - \boldsymbol{y}_n \right\|_2^2$$

Obviously the problems involved in PCA and K-means algorithms fit into the general constrained formulation

$$\begin{aligned} \text{minimize} \quad & f(\boldsymbol{x}) && (1.1a) \\ \text{subject to:} \quad & a_i(\boldsymbol{x}) = 0 \quad \text{for } i = 1,2,\ldots,p && (1.1b) \\ & c_j(\boldsymbol{x}) \le 0 \quad \text{for } j = 1,2,\ldots,q && (1.1c) \end{aligned}$$

while constructing a linear model for prediction is an unconstrained optimization problem of the form

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad f(\boldsymbol{x}) \tag{2.1}$$

Below is another example in machine learning that involves unconstrained optimization.

Example 2.1 *Logistic regression* is a probabilistic classification model used for predicting class labels based on input data or features. From a statistical perspective, supervised learning is essentially to learn conditional probability $P(y \mid \boldsymbol{x})$, i.e. the probability of having label y for an *observed* sample $\boldsymbol{x}$ from a labelled data set. Specifically, given train data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, N\}$ where label y_n takes value 1 for class $\mathcal{P}$ or -1 for class $\mathcal{N}$, logistic regression models the conditional probability as

$$P(y \mid \boldsymbol{x}) = \begin{cases} h(\boldsymbol{x}) & \text{for } y = 1 \\ 1 - h(\boldsymbol{x}) & \text{for } y = -1 \end{cases} \tag{2.2a}$$

where $h(\boldsymbol{x})$ is a *logistic* (also known as *sigmoid*) *function* that assumes the form

$$h(\boldsymbol{x}) = \frac{1}{1+e^{-(\boldsymbol{w}^T\boldsymbol{x}+b)}} \tag{2.2b}$$

and weight vector $\boldsymbol{w}$ and bias b are parameters that can be trained to fit data set $\mathcal{D}$.

To justify Eq. (2.2b), note that the value of $h(\boldsymbol{x})$ (as well as $1-h(\boldsymbol{x})$) falls within the range $(0, 1)$ and hence may be interpreted as *probability*. If we let

$$\hat{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \quad \text{and} \quad \hat{\boldsymbol{w}} = \begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$$

the logistic function can be expressed more compactly as

$$h(\boldsymbol{x}) = \frac{1}{1+e^{-\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}}}$$

Based on (2.2a) and (2.2b), the conditional probability can be expressed as

$$P(y \mid \boldsymbol{x}) = \frac{1}{1+e^{-y(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}})}} \tag{2.3}$$

Now suppose the training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, ..., N\}$ are independently and identically distributed (i.i.d.), then the probability of having observed data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n),\ n = 1, 2, \ldots, N\}$ is given by

$$\prod_{n=1}^{N} P(y_n \mid \boldsymbol{x}_n) \tag{2.4}$$

Therefore, in order for (2.3) to be a good model for the conditional probability $P(y \mid \boldsymbol{x})$, it intuitively makes sense to tune parameters $\boldsymbol{w}$ and b such that the probability in (2.4) becomes largest. In the literature, such a tuning process is called *maximum likelihood*. Evidently, maximizing the probability in (2.4) is equivalent to minimizing negative logarithm of the probability in (2.4). With the help of (2.3), the negative logarithm of (2.4) can be expressed as

$$-\log\left(\prod_{n=1}^{N} P(y_n \mid \boldsymbol{x}_n)\right) = \sum_{n=1}^{N} \log\left(\frac{1}{P(y_n \mid \boldsymbol{x}_n)}\right) = \sum_{n=1}^{N} \ln\left(1+e^{-y_n\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_n}\right)$$

Summarizing the above analysis, logistic regression employs model (2.3) for conditional probability with optimal parameters $\boldsymbol{w}$ and b that are obtained by solving the unconstrained problem

$$\underset{\hat{\boldsymbol{w}}}{\text{minimize}} \ \ f(\hat{\boldsymbol{w}}) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(1+e^{-y_n\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_n}\right) \tag{2.5}$$

Note that, without effecting its solution, we have included a scaling factor $1/N$ in (2.5) to prevent the objective function from being overwhelmingly large, especially for problems with large-scale data sets.

Once the (global) minimizer $\hat{\boldsymbol{w}}^*$ of problem (2.5) is obtained, the conditional probability for a new sample $\boldsymbol{x}$ outside train data $\mathcal{D}$ can be computed by evaluating conditional probability

$$P(y \mid \boldsymbol{x}) = \frac{1}{1+e^{-y(\hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}})}} \tag{2.6}$$

Obviously, if $P(1 \mid \boldsymbol{x}) > 0.5$, then we claim $\boldsymbol{x} \in \mathcal{P}$; and if $P(-1 \mid \boldsymbol{x}) > 0.5$ we claim $\boldsymbol{x} \in \mathcal{N}$. From (2.6), we note that

$$P(1 \mid \boldsymbol{x}) = \frac{1}{1+e^{-(\hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}})}} > 0.5 \text{ if and only if } \hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}} > 0$$

and

$$P(-1 \mid \boldsymbol{x}) = \frac{1}{1+e^{\hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}}}} > 0.5 \text{ if and only if } \hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}} < 0$$

In consequence, the classification of a new sample $\boldsymbol{x}$ can be performed as follows:

$$\boldsymbol{x} \text{ belongs to class } \mathcal{P} \text{ if } \hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}} > 0 \text{ and } \boldsymbol{x} \text{ belongs to class } \mathcal{N} \text{ if } \hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}} < 0 \tag{2.7}$$

■

This chapter is devoted to several algorithms for solving problem (2.1). We start our study with gradient and Hessian – two concepts of fundamental importance in the theory and practice of mathematical optimization.

2.1 *Gradient and Hessian*

Respectively, gradient and Hessian are the most natural extensions of first-order and second-order derivatives of single-variable smooth functions to multi-variable functions.

Given a smooth function $f(\boldsymbol{x})$ with $\boldsymbol{x} \in R^{n\times 1}$, the *gradient* of $f(\boldsymbol{x})$ is a column vector of dimension n that collects all first-order partial derivatives of the function, namely,

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \dfrac{\partial f}{\partial x_2} \\ \vdots \\ \dfrac{\partial f}{\partial x_n} \end{bmatrix} \tag{2.8}$$

Sometimes we use $\boldsymbol{g}(\boldsymbol{x})$ to denote $\nabla f(\boldsymbol{x})$.

The *Hessian* of $f(\boldsymbol{x})$ is an n by n *symmetric* matrix that collects all second-order derivatives of $f(\boldsymbol{x})$ and assumes the form

$$\nabla^2 f(\boldsymbol{x}) = \nabla\left(\nabla^T f(x)\right) = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \cdots & \dfrac{\partial^2 f}{\partial x_n \partial x_1} \\ \dfrac{\partial^2 f}{\partial x_1 \partial x_2} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_n \partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_1 \partial x_n} & \dfrac{\partial^2 f}{\partial x_2 \partial x_n} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{2.9}$$

Sometimes we use $\boldsymbol{H}(\boldsymbol{x})$ to denote $\nabla^2 f(\boldsymbol{x})$.

Example 2.2 Let us consider the *Rosenbrock function* defined by

$$f(x_1, x_2) = (x_1 - 1)^2 + 100(x_1^2 - x_2)^2$$

By definition, the gradient and Hessian of the function are found to be

$$\nabla f(x_1, x_2) = \begin{bmatrix} 2(x_1 - 1) + 400x_1(x_1^2 - x_2) \\ -200(x_1^2 - x_2) \end{bmatrix}$$

and

$$\nabla^2 f(x_1, x_2) = \begin{bmatrix} 1200x_1^2 - 400x_2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}$$

■

2.2 Optimality Conditions

2.2.1. Taylor expansion of multivariable functions

We learned from calculus the *Taylor expansion* of smooth one-variable function $f(x)$. It is about a function's local behaviour in a small vicinity of at a point, say x. The Taylor expansion states that

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2 + \frac{1}{6}f'''(x)\delta^3 + \cdots \tag{2.10}$$

As long as perturbation δ in (2.10) remains sufficiently small, the "perturbed" point, $x+\delta$, will vary in a small neighborhood of x, and the Taylor expansion leads to a *linear approximation* of $f(x+\delta)$ as

$$f(x+\delta) \approx f(x) + f'(x)\delta$$

and a *quadratic approximation* of $f(x+\delta)$ as

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2$$

An essential message conveyed by the Taylor expansion is that *locally any smooth function* acts like a *polynomial*, and if one examines the function closely in a small neighborhood of a given

point, it resembles a quadratic polynomial and, it tends to be like a short segment of a straight line as the neighborhood shrinks. Fig. 2.1 illustrates how local linear and quadratic models mimic an original (and possibly complicated) objective.

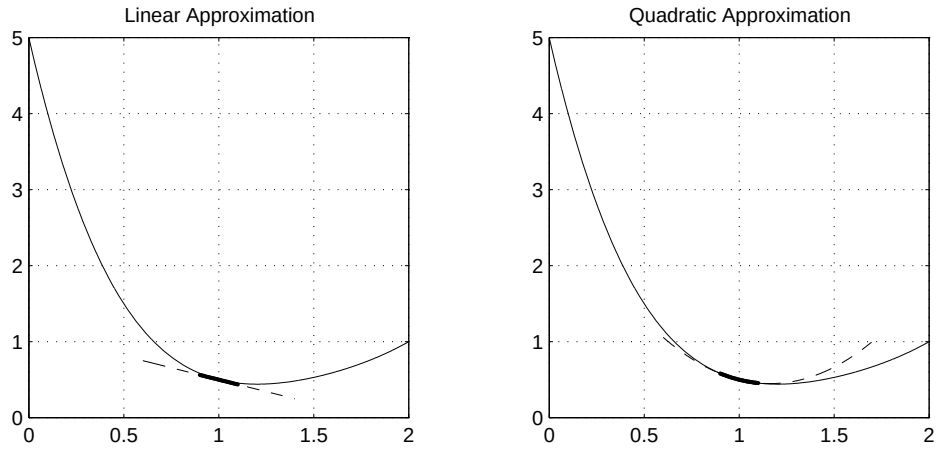

Figure 2.1 Both linear and quadratic approximations work well in a small vicinity of the chosen point (in this case $x = 1$). Also notice that while the linear approximation quickly deviates from the original function $f(x)$ as the size of the vicinity get larger (e.g. over the interval from 0.6 to 1.4), the quadratic approximation remains fairly close to $f(x)$ over the same interval.

Example 2.3 Suppose we want to calculate $\sqrt{4.01}$. Obviously the calculation is related to function $f(x) = \sqrt{x}$. We compute its first-order derivative (i.e. gradient)

$$\frac{d}{dx}\left(\sqrt{x}\right) = \frac{1}{2\sqrt{x}}$$

hence a linear approximation of $f(x)$ is given by

$$\sqrt{x+\delta} \approx \sqrt{x} + \frac{1}{2\sqrt{x}}\delta$$

To calculate $\sqrt{4.01}$, let $x = 4$ and $\delta = 0.01$. Since δ is small, the above linear approximate is applicable, which yields

$$\sqrt{4.01} \approx 2 + 0.01/4 = 2.0025$$

For comparison, the true value of $\sqrt{4.01}$ is found to be $\sqrt{4.01} \approx 2.00249839\cdots$, and the error introduced by the linear approximation is about 1.6×10^{-6}. ■

The Taylor expansion of a multivariable smooth function $f(\boldsymbol{x})$ is a straightforward but

important generalization of its single-variable counterpart. At a *fixed* $\boldsymbol{x}$ and a change δ (or perturbation or variation) in $\boldsymbol{x}$, which is small in magnitude, the Taylor expansion of $f(\boldsymbol{x})$ at point $\boldsymbol{x}$ is given by

$$f(\boldsymbol{x}+\boldsymbol{\delta}) \approx f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{\delta} + \frac{1}{2}\boldsymbol{\delta}^T \nabla^2 f(\boldsymbol{x})\boldsymbol{\delta} + O\left(\|\boldsymbol{\delta}\|^3\right) \tag{2.11}$$

We stress that in (2.11) $\boldsymbol{x}$ is *given* and *fixed* while $\boldsymbol{\delta}$ is considered as a *variable*. Immediately, Eq. (2.11) yields a linear approximations of $f(\boldsymbol{x}+\boldsymbol{\delta})$ as

$$f(\boldsymbol{x}+\boldsymbol{\delta}) \approx f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{\delta} \tag{2.12a}$$

and a quadratic approximation of $f(\boldsymbol{x}+\boldsymbol{\delta})$ as

$$f(\boldsymbol{x}+\boldsymbol{\delta}) \approx f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{\delta} + \frac{1}{2}\boldsymbol{\delta}^T \nabla^2 f(\boldsymbol{x})\boldsymbol{\delta} \tag{2.12b}$$

Example 2.4 Let us simplify the 6th-order polynomial of variables x_1 and x_2

$$\begin{aligned} f(x_1, x_2) = 1.2 - 80x_1 - 40x_2 + 40x_1^2 + 20x_2^2 + 0.5x_1^3 - x_1^2 x_2 \\ + 0.3x_2^3 + 0.5x_1^4 - 0.2x_1^4 x_2^2 + 0.1x_1^2 x_2^4 + 0.2x_2^6 \end{aligned}$$

in a small neighborhood of $\boldsymbol{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$.

Our procedure of simplification starts with computing the partial derivatives of the function:

$$\frac{\partial f}{\partial x_1} = -80 + 80x_1 + 1.5x_1^2 - 2x_1 x_2 + 2x_1^3 - 0.8x_1^3 x_2^2 + 0.2x_1 x_2^4$$

$$\frac{\partial f}{\partial x_2} = -40 + 40x_2 - x_1^2 + 0.9x_2^2 - 0.4x_1^4 x_2 + 0.4x_1^2 x_2^3 + 1.2x_2^5$$

$$\frac{\partial^2 f}{\partial x_1^2} = 80 + 3x_1 - 2x_2 + 6x_1^2 - 2.4x_1^2 x_2^2 + 0.2x_2^4$$

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = -2x_1 - 1.6x_1^3 x_2 + 0.8x_1 x_2^3$$

$$\frac{\partial^2 f}{\partial x_2^2} = 40 + 1.8x_2 - 0.4x_1^4 + 1.2x_1^2 x_2^2 + 6x_2^4$$

Based on these, the function and its gradient and Hessian at $\boldsymbol{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$ are evaluated and found to be

$$f(\boldsymbol{x}) = -58.4,\ \nabla f(\boldsymbol{x}) = \begin{bmatrix} 0.9 \\ 1.1 \end{bmatrix},\ \text{and}\ \ \nabla^2 f(\boldsymbol{x}) = \begin{bmatrix} 84.8 & -2.8 \\ -2.8 & 48.6 \end{bmatrix}$$

hence a quadratic approximation of the function, which is valid in small neighborhood of $\boldsymbol{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$ is obtained as

$$f(\boldsymbol{x}+\boldsymbol{\delta}) \approx -58.4 + \begin{bmatrix}\delta_1 & \delta_2\end{bmatrix}\begin{bmatrix}0.9\\ 1.1\end{bmatrix} + \frac{1}{2}\begin{bmatrix}\delta_1 & \delta_2\end{bmatrix}\begin{bmatrix}84.8 & -2.8\\ -2.8 & 48.6\end{bmatrix}\begin{bmatrix}\delta_1\\ \delta_2\end{bmatrix}$$

where δ_1, δ_2 are limited to be small scalars. For comparison, the original function and its quadratic simplification in a small region centered at $\boldsymbol{x} = \begin{bmatrix}1 & 1\end{bmatrix}^T$ are shown in Fig. 2.2. ■

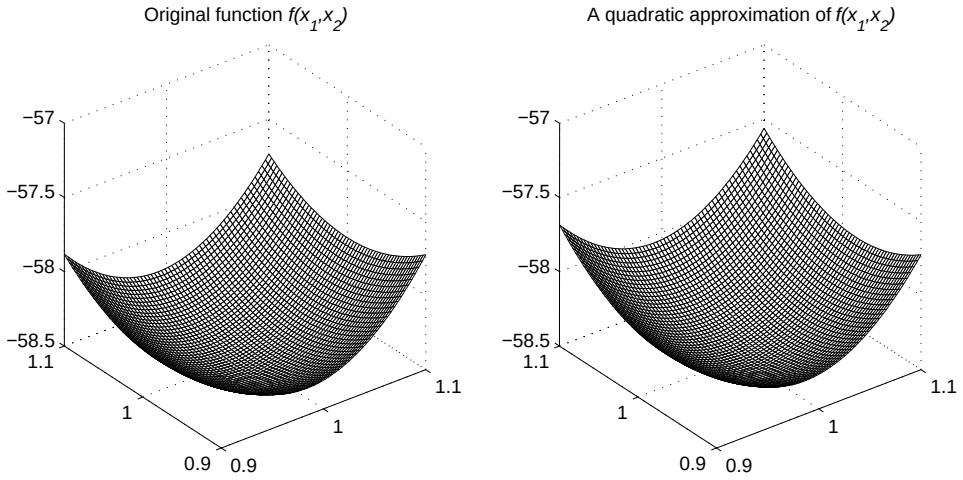

Figure 2.2 A 6th-order polynomial at point $[1\ 1]^T$ versus its quadratic approximation for Example 2.4.

2.2.2. Properties of an unconstrained minimizer

We now examine several important properties of an unconstrained minimizer which solves the problem (2.1).

- ***First-order necessary condition***

Theorem 2.1 ***First-order necessary condition for an unconstrained minimum***

If $\boldsymbol{x}^*$ is local minimizer of $f(\boldsymbol{x})$, then

$$\nabla f(\boldsymbol{x}^*) = \boldsymbol{0} \tag{2.13}$$

Proof: From Taylor expansion of $f(\boldsymbol{x})$ in a neighborhood of $\boldsymbol{x}^*$, namely,

$$f(\boldsymbol{x}^* + \boldsymbol{\delta}) = f(\boldsymbol{x}^*) + \nabla f(\boldsymbol{x}^*)^T \boldsymbol{\delta} + O(\| \boldsymbol{\delta} \|^2)$$

and $\boldsymbol{x}^*$ being a local minimizer, we have

$$f(\boldsymbol{x}^* + \boldsymbol{\delta}) - f(\boldsymbol{x}^*) = \nabla f(\boldsymbol{x}^*)^T \boldsymbol{\delta} + O(\| \boldsymbol{\delta} \|^2) \geq 0$$

which implies that, for *any* $\boldsymbol{\delta}$ of small magnitude, $\nabla f(\boldsymbol{x}^*)^T \boldsymbol{\delta} \geq 0$. By choosing $\boldsymbol{\delta} = -\nabla f(\boldsymbol{x}^*)$ (up to an appropriate scaling factor), this means that $-\| \nabla f(\boldsymbol{x}^*) \|^2 \geq 0$ which immediately leads to (2.13). ■

- ***Second-order necessary conditions and second-order sufficient conditions***

To describe the second-order conditions for an unconstrained minimizer, we need a concept

concerning the definiteness of a square symmetric matrix and its quadratic form.

Definition

(*a*) Let $\boldsymbol{d}$ be an arbitrary direction vector at point $\boldsymbol{x}$. The quadratic form $\boldsymbol{d}^T\nabla^2 f(\boldsymbol{x})\boldsymbol{d}$ is said to be *positive definite, positive semidefinite, negative definite,* and *negative semidefinite* if $\boldsymbol{d}^T\nabla^2 f(\boldsymbol{x})\boldsymbol{d} > 0, \geq 0, < 0$, and ≤ 0, respectively, for all $\boldsymbol{d} \neq \boldsymbol{0}$ at $\boldsymbol{x}$. The quadratic form $\boldsymbol{d}^T\nabla^2 f(\boldsymbol{x})\boldsymbol{d}$ is said to be *indefinite* if it yields positive values for some $\boldsymbol{d}$'s and negative values for some other $\boldsymbol{d}$'s.

(*b*) Symmetric matrix $\nabla^2 f(\boldsymbol{x})$ is said to be *positive definite* (denoted by $\nabla^2 f(\boldsymbol{x}) \succ \boldsymbol{0}$), *positive semidefinite* (denoted by $\nabla^2 f(\boldsymbol{x}) \succeq \boldsymbol{0}$), *negative definite* (denoted by $\nabla^2 f(\boldsymbol{x}) \prec \boldsymbol{0}$), *negative semidefinite* (denoted by $\nabla^2 f(\boldsymbol{x}) \preceq \boldsymbol{0}$) and *indefinite*, if quadratic form $\boldsymbol{d}^T\nabla^2 f(\boldsymbol{x})\boldsymbol{d}$ is positive definite. positive semidefinite, negative definite, and negative semidefinite, and indefinite, respectively. The next theorem provides an effective way to check the definiteness of a symmetric matrix.

Theorem 2.2 A symmetric matrix is positive definite, positive semidefinite, negative semidefinite, negative definite, or negative semidefinite if and only if its engenvalues are positive, non-negative, negative, or non-positive, respectively. A symmetric matrix is indefinite if and only if it possesses both positive and negative eigenvalues.

Theorem 2.3 ***Second-order necessary conditions for an unconstrained minimum***

If $f(\boldsymbol{x}) \in C^2$ and $\boldsymbol{x}^*$ is a local minimizer, then

(a) $\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$

(b) $\nabla^2 f(\boldsymbol{x}^*)$ is positive semidefinite.

Proof: From Taylor expansion of $f(\boldsymbol{x})$ in a neighborhood of $\boldsymbol{x}^*$, namely

$$f(\boldsymbol{x}^* + \boldsymbol{\delta}) = f(\boldsymbol{x}^*) + \nabla f(\boldsymbol{x}^*)^T\boldsymbol{\delta} + \tfrac{1}{2}\boldsymbol{\delta}^T\nabla^2 f(\boldsymbol{x}^*)^T\boldsymbol{\delta} + O(\|\boldsymbol{\delta}\|^3)$$

in conjunction with (2.8) and $\boldsymbol{x}^*$ being a local minimizer, we have

$$f(\boldsymbol{x}^* + \boldsymbol{\delta}) - f(\boldsymbol{x}^*) = \tfrac{1}{2}\boldsymbol{\delta}^T\nabla^2 f(\boldsymbol{x}^*)^T\boldsymbol{\delta} + O(\|\boldsymbol{\delta}\|^3) \geq 0$$

which implies that, for *any* $\boldsymbol{\delta}$ of small magnitude, $\boldsymbol{\delta}^T\nabla^2 f(\boldsymbol{x}^*)^T\boldsymbol{\delta} \geq 0$. Therefore $\nabla^2 f(\boldsymbol{x}^*)$ is positive semidefinite. ■

To better understand why the conditions given in Theorem 2.3 are only "necessary", the next example shows that conditions $\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$ and $\nabla^2 f(\boldsymbol{x}^*) \succeq \boldsymbol{0}$ are *not sufficient* for $\boldsymbol{x}^*$ to be a minimizer.

Example 2.5 Examine the objective function

$$f(x_1, x_2) = x_1^2 - 2x_1 + x_2^3$$

By setting the gradient $\nabla f(\boldsymbol{x})$ to zero, we have

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} 2x_1 - 2 \\ 3x_2^2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

which yields the only candidate point as

$$\boldsymbol{x}^* = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

At $\boldsymbol{x} = \boldsymbol{x}^*$, the Hessian is found to be

$$\nabla^2 f(\boldsymbol{x}^*) = \begin{bmatrix} 2 & 0 \\ 0 & 6x_2 \end{bmatrix}\Bigg|_{\boldsymbol{x}=\boldsymbol{x}^*} = \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}$$

which is obviously positive semidefinite (but *not* positive definite). Hence point $\boldsymbol{x}^*$ satisfies the second-order necessary conditions.

The question now is whether $\boldsymbol{x}^*$ is a local minimizer of $f(\boldsymbol{x})$. To address the question, we examine the difference $f(\boldsymbol{x}^* + \boldsymbol{\delta}) - f(\boldsymbol{x}^*)$ as follows:

$$f(\boldsymbol{x}^* + \boldsymbol{\delta}) - f(\boldsymbol{x}^*) = \left(\delta_1^2 + \delta_2^3 - 1\right) - (-1) = \delta_1^2 + \delta_2^3$$

If we draw a line that vertically passes though point $\boldsymbol{x}^* = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$ and move a point along that line crossing point $\boldsymbol{x}^*$ while evaluating how the value of function $f(\boldsymbol{x})$ changes relative to the value of $f(\boldsymbol{x}^*)$ = –1, we are actually evaluating the difference $f(\boldsymbol{x}^* + \boldsymbol{\delta}) - f(\boldsymbol{x}^*)$: when the point on the vertical line moves from $\boldsymbol{x}^*$ to the right, we have $\delta_1 = 0$ and $\delta_2 > 0$, hence $f(\boldsymbol{x}^* + \boldsymbol{\delta})$ grows up (thus $\boldsymbol{x}^*$ can't be a maximize); and when the point on the vertical line moves from $\boldsymbol{x}^*$ to the left, we have $\delta_1 = 0$ and $\delta_2 < 0$, hence $f(\boldsymbol{x}^* + \boldsymbol{\delta})$ goes down (thus $\boldsymbol{x}^*$ can't be a minimizer). Therefore we conclude that $\boldsymbol{x}^*$ is a saddle point. The function in a small neighborhood of $\boldsymbol{x}^*$ is depicted in Fig. 2.3. ■

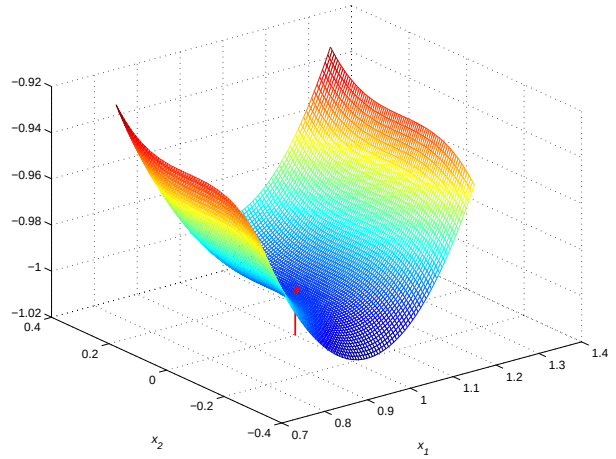

Figure 2.3 Function $f(\boldsymbol{x})$ in a small neighborhood of $\boldsymbol{x}^*$ for Example 2.5.

Theorem 2.4 ***Second-order sufficient conditions for an unconstrained minimum***

Suppose $f(\boldsymbol{x}) \in C^2$. Then the conditions

(a) $\nabla f(\boldsymbol{x}^*) = \mathbf{0}$

(b) $\nabla^2 f(\boldsymbol{x}^*)$ is positive definite

are sufficient for $\boldsymbol{x}^*$ to be a strong local minimizer.

Proof: Under conditions (a) and (b), we have for sufficiently small $\boldsymbol{\delta}$

$$f(\boldsymbol{x}^* + \boldsymbol{\delta}) - f(\boldsymbol{x}^*) = \tfrac{1}{2}\boldsymbol{\delta}^T \nabla^2 f(\boldsymbol{x}^*)^T \boldsymbol{\delta} + O(\| \boldsymbol{\delta} \|^3) > 0$$

Hence $f(\boldsymbol{x}^* + \boldsymbol{\delta}) > f(\boldsymbol{x}^*)$. ■

Example 2.6 Find the minimizer for the Rosenbrock function

$$f(x_1, x_2) = (x_1 - 1)^2 + 100(x_1^2 - x_2)^2$$

Solution The gradient of the function was found in Example 2.2 as

$$\nabla f(x_1, x_2) = \begin{bmatrix} 2(x_1 - 1) + 400x_1(x_1^2 - x_2) \\ -200(x_1^2 - x_2) \end{bmatrix}$$

By setting $\nabla f(x_1, x_2) = \mathbf{0}$, we obtain two equations

$$2(x_1 - 1) + 400x_1(x_1^2 - x_2) = 0$$

$$-200(x_1^2 - x_2) = 0$$

whose (unique) solution is found to be

$$\boldsymbol{x}^* = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The Hessian of the function was found in Example 2.2 to be

$$\nabla^2 f(x_1, x_2) = \begin{bmatrix} 1200x_1^2 - 400x_2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}$$

At $\boldsymbol{x}^*$,

$$\nabla^2 f(\boldsymbol{x}^*) = \begin{bmatrix} 802 & -400 \\ -400 & 200 \end{bmatrix}$$

which is positive definite. Hence $\boldsymbol{x}^*$ satisfies the second-order sufficient conditions and it is a strong local minimizer of the objective function. Furthermore, note that at $\boldsymbol{x}^*$ we have $f(\boldsymbol{x}^*) = 0$. Since the Rosenbrock function is sum of two squared terms, it is always nonnegative and hence $\boldsymbol{x}^*$ is a global minimizer of $f(\boldsymbol{x})$. ■

Example 2.7 Verify that $\boldsymbol{x}^* = \begin{bmatrix} 3 & 2 \end{bmatrix}^T$ is a global minimizer of the Himmelblau function

$$f(\boldsymbol{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

Solution We start with symbolically computing the gradient and Hessian of the function as

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} 4x_1(x_1^2 + x_2 - 11) + 2(x_1 + x_2^2 - 7) \\ 2(x_1^2 + x_2 - 11) + 4x_2(x_1 + x_2^2 - 7) \end{bmatrix}$$

and

$$\nabla^2 f(\boldsymbol{x}) = \begin{bmatrix} 12x_1^2 + 4x_2 - 42 & 4(x_1 + x_2) \\ 4(x_1 + x_2) & 4x_1 + 12x_2^2 - 26 \end{bmatrix}$$

At $\boldsymbol{x}^* = \begin{bmatrix} 3 & 2 \end{bmatrix}^T$, we compute

$$\nabla f(\boldsymbol{x}^*) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ and } \nabla^2 f(\boldsymbol{x}^*) = \begin{bmatrix} 74 & 20 \\ 20 & 34 \end{bmatrix}$$

Since the two eigenvalues of $\nabla^2 f(\boldsymbol{x}^*)$, 82.2843 and 25.7157, are strictly positive, the Hessian $\nabla^2 f(\boldsymbol{x}^*)$ is positive definite. Hence $\boldsymbol{x}^*$ satisfies the second-order sufficient conditions and hence is a strong local minimizer. Moreover, since the objective function is a sum of two squared term, it is always nonnegative while $f(\boldsymbol{x}^*) = 0$, therefore $\boldsymbol{x}^*$ is a global minimizer of the function. ■

2.3 Gradient Descent Algorithm

Originated from the work of A. Cauchy in 1847, the *gradient descent* (GD) algorithm has been popular for unconstrained optimization primarily because of its low computational complexity. We begin by presenting a general structure that most optimization algorithms obey.

2.3.1. General structure of optimization algorithms

Most of algorithms entails a series of steps which are executed sequentially. A typical pattern is as follows.

Step 1 Input an initial point $\boldsymbol{x}_0$ and a convergence tolerance ε, set $k = 0$.

Step 2 Given $\boldsymbol{x}_k$, compute a search direction $\boldsymbol{d}_k$ by an appropriate procedure.

Step 3 Compute a positive scalar α_k that minimizes $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$. Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$.

Step 4 If $\|\alpha_k \boldsymbol{d}_k\| < \varepsilon$, output $\boldsymbol{x}_{k+1}$ as the solution and stop; otherwise, set $k := k + 1$ and go to Step 2.

2.3.2. The GD algorithm

The GD algorithm approaches a local minimizer by *iteratively* computing search directions based

solely on the gradient of the objective function. The algorithm can be understood by observing that the value of a smooth $f(\boldsymbol{x})$ decreases most rapidly along the direction of negative gradient $-\nabla f(\boldsymbol{x})$. This is because

$$f(\boldsymbol{x}+\boldsymbol{\delta})-f(\boldsymbol{x}) \approx \nabla f(\boldsymbol{x})^T \boldsymbol{\delta} = \| \nabla f(\boldsymbol{x}) \| \cdot \| \boldsymbol{\delta} \| \cdot \cos\left\langle \nabla f(\boldsymbol{x}), \boldsymbol{\delta} \right\rangle$$

where $\cos\left\langle \nabla f(\boldsymbol{x}), \boldsymbol{\delta} \right\rangle$ reaches its most negative value -1 when $\boldsymbol{\delta}$ goes along with $-\nabla f(\boldsymbol{x})$.

In other words, $-\nabla f(\boldsymbol{x})$ is the *steepest descent direction* to reduce the objective, and this suggests a formula to update current iterate $\boldsymbol{x}_k$ to the next iterate $\boldsymbol{x}_{k+1}$ as

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k) \tag{2.14}$$

where $\alpha_k > 0$ is a positive real number that controls the step size of the iteration.

The GD algorithm can be summarized as follows.

Gradient descent algorithm

Step 1 Input an initial point $\boldsymbol{x}_0$ and a convergence tolerance ε, set $k = 0$.

Step 2 Compute search direction $\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k)$.

Step 3 Compute a positive scalar α_k that minimizes $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$. Using (2.14) to update $\boldsymbol{x}_k$ to $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$.

Step 4 If $\|\alpha_k \boldsymbol{d}_k\| < \varepsilon$, output $\boldsymbol{x}_{k+1}$ as the solution and stop; otherwise, set $k := k+1$ and go to Step 2.

• MATLAB code of the GD algorithm, **`grad_desc.m`**, is available from the course website.

Line search techniques

The process of determining a suitable α_k, which is required in Step 3 of the GD algorithm, is called *line search*.

For the objective function is convex and quadratic, namely

$$f(\boldsymbol{x}) = \frac{1}{2} \boldsymbol{x}^T \boldsymbol{H} \boldsymbol{x} + \boldsymbol{x}^T \boldsymbol{p} + \kappa$$

where $\boldsymbol{H}$ is positive definite, the α_k that minimizes $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ is given by

$$\alpha_k = \frac{-\boldsymbol{d}_k^T \boldsymbol{g}_k}{\boldsymbol{d}_k^T \boldsymbol{H} \boldsymbol{d}_k} \quad \text{where} \quad \boldsymbol{g}_k = \nabla f(\boldsymbol{x})\big|_{\boldsymbol{x}=\boldsymbol{x}_k} = \boldsymbol{H}\boldsymbol{x}_k + \boldsymbol{p} \tag{2.15}$$

To use (2.15) in Step 3 of the GD algorithm, we follow Step 2 to substitute $\boldsymbol{d}_k = -\boldsymbol{g}_k$ into (2.15), which gives

$$\alpha_k = \frac{\boldsymbol{g}_k^T \boldsymbol{g}_k}{\boldsymbol{g}_k^T \boldsymbol{H} \boldsymbol{g}_k} \tag{2.16}$$

For a general objective function, there is a simple yet effective technique, known as *back-tracking* line search (BTLS):

Backtracking line search (BTLS) ***to compute step size*** α_k

Input A point $\boldsymbol{x}_k$ and a descent direction $\boldsymbol{d}_k$ at $\boldsymbol{x}_k$.

Step 1 Select constants $\rho \in (0, 0.5)$ and $\gamma \in (0,1)$. Set $\alpha = 1$.

Step 2 While $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) > f(\boldsymbol{x}_k) + \rho\alpha \boldsymbol{g}_k^T \boldsymbol{d}_k$, set $\alpha := \gamma\alpha$.

Step 3 Output $\alpha_k = \alpha$.

• MATLAB code of the BTLS algorithm, `bt_lsearch.m`, is available from the course website.

To implement Step 1, values $\rho = 0.1$ and $\gamma = 0.5$ are recommended as they often work well. To see how BTLS works, note that the expression on the right-hand side of the inequality in Step 2 represents a line in the coordinate system for function $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ versus α that passes through the point $(0, f(\boldsymbol{x}_k))$ with a negative slop because

$$\rho \boldsymbol{g}_k^T \boldsymbol{d}_k = \rho \frac{df(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)}{d\alpha}\bigg|_{\alpha=0} < 0$$

Also note that with $\rho \in (0, 0.5)$ the above slop is less steeper than $df(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)/d\alpha\,|_{\alpha=0}$. Fig. 2.4 visualizes the linear function on the right-hand side of the inequality in Step 2 as a dashed line which in conjunction with objective function $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ defines a value α_0. From the figure we observe that a value of α less than α_0 may be deemed acceptable because such an α insures that $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ is no greater than $f(\boldsymbol{x}_k) + \rho\alpha \boldsymbol{g}_k^T \boldsymbol{d}_k$. This explains Step 2 which says one shall keep reducing the value of α by a factor of γ until the inequality in Step 2 is no longer valid.

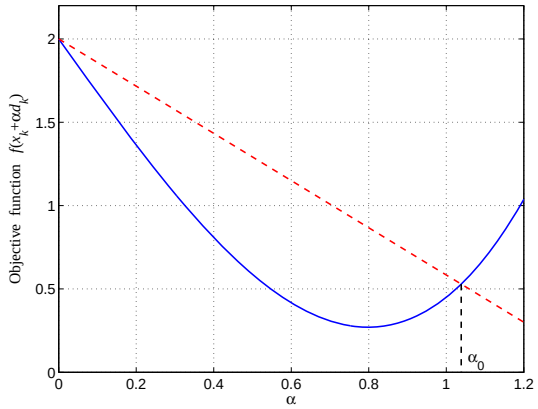

Fig. 2.4 Both $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ (blue curve) and $f(\boldsymbol{x}_k) + \rho\alpha \boldsymbol{g}_k^T \boldsymbol{d}_k$ (red dashed line) are shown as functions of α. An acceptable α is found by gradually reducing α to a value less than α_0, which is signified by $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) > f(\boldsymbol{x}_k) + \rho\alpha \boldsymbol{g}_k^T \boldsymbol{d}_k$.

Estimating average CPU time required by an algorithm

A popular measure for the *computational complexity* of an algorithm is the CPU time the algorithm consumes. The CPU time may not be measured accurately on a PC especially when it is as small as a tiny fraction of a second. Below is an MATLAB pseudocode that is of use for estimating average CPU time consumed by an algorithm with reasonable accuracy.

1. Select a sufficiently large integer *K* (e.g. *K* = 1000).
Set initial_cpu_time = cputime;

2. for *i* = 1:*K*,

run the algorithm;

end

3. Set final_cpu_time = cputime;

4. Compute the average CPU time as

Average CPU time = (final_cpu_time – initial_cpu_time)/*K*;

Example 2.8 Apply the GD algorithm to the unconstrained problem

$$\text{minimize } f(\boldsymbol{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

where the objective function is the Himmelblau function studied in Example 7. Try four initial points

$$\boldsymbol{x}_0^{(1)} = \begin{bmatrix} 6 \\ 6 \end{bmatrix}, \quad \boldsymbol{x}_0^{(2)} = \begin{bmatrix} -6 \\ 6 \end{bmatrix}, \quad \boldsymbol{x}_0^{(3)} = \begin{bmatrix} -6 \\ -6 \end{bmatrix}, \quad \boldsymbol{x}_0^{(4)} = \begin{bmatrix} 6 \\ -6 \end{bmatrix}$$

and convergence tolerance is set to $\varepsilon = 10^{-9}$.

Solution

(1) With initial point $\boldsymbol{x}_0 = \boldsymbol{x}_0^{(1)}$, it took 17 iterations for the GD algorithm to converge to a point

$$\boldsymbol{x}_1^* = \begin{bmatrix} -3.779310253478946 & -3.283185991258242 \end{bmatrix}^T$$

at which $f(\boldsymbol{x}_1^*) = 7.0961 \times 10^{-19}$. The gradient and Hessian of $f(\boldsymbol{x})$ at $\boldsymbol{x}_1^*$ are found to be

$$\nabla f(\boldsymbol{x}_1^*) = \begin{bmatrix} -0.1255 \\ 0.0532 \end{bmatrix} \times 10^{-7} \text{ and } \nabla^2 f(\boldsymbol{x}_1^*) = \begin{bmatrix} 116.2655 & -28.25 \\ -28.25 & 88.2345 \end{bmatrix} \succ \mathbf{0}$$

which confirms that $\boldsymbol{x}_1^*$ is a minimizer (actually a globol minimizer (why?)).

(2) With initial point $\boldsymbol{x}_0 = \boldsymbol{x}_0^{(2)}$, it took 20 iterations for the GD algorithm to converge to a point

$$\boldsymbol{x}_2^* = \begin{bmatrix} -2.805118086943204 & 3.131312518364652 \end{bmatrix}^T$$

at which $f(\boldsymbol{x}_2^*) = 5.2780 \times 10^{-19}$. The gradient and Hessian of $f(\boldsymbol{x})$ at $\boldsymbol{x}_2^*$ are obtained as

$$\nabla f(\boldsymbol{x}_2^*) = \begin{bmatrix} 0.0769 \\ 0.9189 \end{bmatrix} \times 10^{-8} \text{ and } \nabla^2 f(\boldsymbol{x}_2^*) = \begin{bmatrix} 64.9495 & 1.3048 \\ 1.3048 & 80.4409 \end{bmatrix} \succ \mathbf{0}$$

which confirms that $\boldsymbol{x}_2^*$ is a global minimizer.

(3) With initial point $\boldsymbol{x}_0 = \boldsymbol{x}_0^{(3)}$, it took 50 iterations for the GD algorithm to converge to a point $\boldsymbol{x}_3^* = \begin{bmatrix} 3.584428340593605 & -1.848126526940458 \end{bmatrix}^T$ at which $f(\boldsymbol{x}_3^*) = 3.6792 \times 10^{-18}$. The gradient and Hessian of $f(\boldsymbol{x})$ at $\boldsymbol{x}_3^*$ are obtained as

$$\nabla f(\boldsymbol{x}_3^*) = \begin{bmatrix} 0.2774 \\ 0.0253 \end{bmatrix} \times 10^{-7} \text{ and } \nabla^2 f(\boldsymbol{x}_3^*) = \begin{bmatrix} 104.7850 & 6.9452 \\ 6.9452 & 29.3246 \end{bmatrix} \succ \mathbf{0}$$

which confirms that $\boldsymbol{x}_3^*$ is a global minimizer.

(4) With initial point $\boldsymbol{x}_0 = \boldsymbol{x}_0^{(4)}$, it took 32 iterations for the GD algorithm to converge to a point

$$\boldsymbol{x}_4^* = \begin{bmatrix} 3.000000000116121 & 1.999999999907941 \end{bmatrix}^T$$

at which $f(\boldsymbol{x}_4^*) = 4.2918 \times 10^{-19}$. The gradient and Hessian of $f(\boldsymbol{x})$ at $\boldsymbol{x}_4^*$ are obtained as

$$\nabla f(\boldsymbol{x}_4^*) = \begin{bmatrix} 0.6752 \\ -0.0808 \end{bmatrix} \times 10^{-8} \text{ and } \nabla^2 f(\boldsymbol{x}_4^*) = \begin{bmatrix} 74 & 20 \\ 20 & 34 \end{bmatrix} \succ \mathbf{0}$$

which confirms that $\boldsymbol{x}_4^*$ is a global minimizer.

As an instance, the average CPU time of the GD algorithm for case (4) on a Windows 7 PC with a 3.4 GHz i7-3770 CPU was about 19.48 ms. ■

Note: Unless stated otherwise, numerical simulations presented in the course notes were all conducted using the same computer.

2.3.3. Convergence issues and variable scaling

The GD algorithm is a *first-order* algorithm since it is based on the linear approximation of the Taylor series. It can be shown that if the line search in each GD iteration is performed exactly, the iterates generated by GD always form a zigzag path as depicted in Fig. 2.5. Consequently, the GD algorithm can be slow, especially when the Hessian of the objective is *ill-conditioned* (i.e. has a large *condition number*) and in that case the contours of the objective function tends to be less circular as seen from Fig. 2.5.

If $\boldsymbol{x}^*$ is a local minimizer of a smooth objective function $f(\boldsymbol{x})$ whose Hessian at kth iterate $\boldsymbol{x}_k$, $\boldsymbol{H}_k$, is positive definite, then a theoretical analysis shows that as $\boldsymbol{x}_k$ gets closer to $\boldsymbol{x}^*$, we have

$$|f(\boldsymbol{x}_{k+1}) - f(\boldsymbol{x}^*)| \le \left(\frac{\kappa - 1}{\kappa + 1}\right)^2 |f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*)| \tag{2.17}$$

where κ denotes the condition number of $\boldsymbol{H}_k$, namely

$$\kappa = \frac{\text{largest eigenvalue of } \boldsymbol{H}_k}{\text{smallest eigenvalue of } \boldsymbol{H}_k}$$

From (2.17), it follows that the reduction in the objective function from $\boldsymbol{x}_k$ to $\boldsymbol{x}_{k+1}$ can be insignificant if the condition number of $\boldsymbol{H}_k$ is too large.

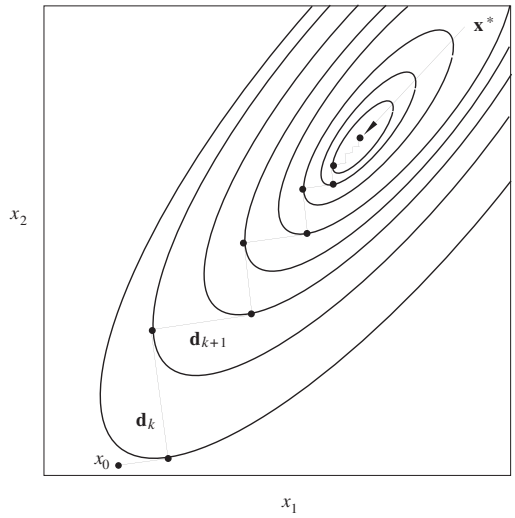

Fig. 2.5 Typical solution path in a gradient descent algorithm [9].

Scaling the variables

There are several ways to improve the rate of convergence of the basic GD algorithm. One simple technique is to scale the variables through a variable transformation of the form $\boldsymbol{x} = \boldsymbol{T}\boldsymbol{y}$ where $\boldsymbol{T}$ is a diagonal matrix. This variable change converts the original problem of minimizing $f(\boldsymbol{x})$ with respect to $\boldsymbol{x}$ to minimizing $h(\boldsymbol{y}) = f(\boldsymbol{T}\boldsymbol{y})$ with respect to $\boldsymbol{y}$. The gradient and Hessian of $h(\boldsymbol{y})$ are found to be

$$\nabla h(\boldsymbol{y}) = \boldsymbol{T}^T \nabla f(\boldsymbol{x})\big|_{\boldsymbol{x}=\boldsymbol{T}\boldsymbol{y}} \quad \text{and} \quad \nabla^2 h(\boldsymbol{y}) = \boldsymbol{T}^T \nabla^2 f(\boldsymbol{x})\big|_{\boldsymbol{x}=\boldsymbol{T}\boldsymbol{y}} \boldsymbol{T} \tag{2.18}$$

respectively. We see that both gradient and Hessian have been changed, hence the GD algorithm is expect to perform better for objective $h(\boldsymbol{y})$ with an appropriate scaling matrix $\boldsymbol{T}$. Unfortunately, the choice of $\boldsymbol{T}$ tends to depend heavily on the problem at hand and, as a result, no general rules can be stated.

2.4 Accelerated Gradient Algorithm

Nesterov's accelerated gradient (NAG) algorithm [11], [12] is a good representative of the class of efficient gradient descent algorithms that work for convex functions with Lipschitz continuous gradient.

A smooth function $f(\boldsymbol{x})$ is said to have *Lipschitz continuous gradient* if there exists constant L such that

$$\| \nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y}) \|_2 \leq L \| \boldsymbol{x} - \boldsymbol{y} \|_2 \tag{2.19}$$

holds for any $\boldsymbol{x}$ and $\boldsymbol{y}$. The NAG algorithm applies to the unconstrained problem (2.1) where $f(\boldsymbol{x})$ is convex with Lipschitz continuous gradient.

Nesterov's accelerated gradient (NAG) algorithm

Step 1 Input initial point $\boldsymbol{x}_0$, Lipschitz constant L, and tolerance ε. Set $\boldsymbol{y}_1 = \boldsymbol{x}_0$, $\alpha = 1/L$, $t_1 = 1$, and $k = 1$.

Step 2 Compute $\boldsymbol{x}_k = \boldsymbol{y}_k - \alpha \nabla f(\boldsymbol{y}_k)$.

Step 3 Compute $t_{k+1} = \dfrac{1+\sqrt{1+4t_k^2}}{2}$.

Step 4 Update $\boldsymbol{y}_{k+1} = \boldsymbol{x}_k + \left(\dfrac{t_k - 1}{t_{k+1}}\right)(\boldsymbol{x}_k - \boldsymbol{x}_{k-1})$.

Step 5 If $f(\boldsymbol{x}_{k-1}) - f(\boldsymbol{x}_k) < \varepsilon$, output $\boldsymbol{x}_k$ as the solution and stop; otherwise set $k = k + 1$ and repeat from Step 2.

The NAG algorithm can be shown to converge to the minimizer $\boldsymbol{x}^*$ with rate $O(1/k^2)$ in the sense that

$$f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*) \le \frac{2 \| \boldsymbol{x}_0 - \boldsymbol{x}^* \|_2^2}{\alpha k^2}$$

The algorithmic steps described above show that the NAG algorithm is essentially a variant of the conventional GD algorithm, where the gradient descent step (Step 2) is taken at a point obtained as weighted sum of the last two iterates (Step 4). It turns out that with appropriate weights calculated in Step 3, the weighted sum of the last two iterates yields a smoothed direction that helps avoid zigzag profile of typical steepest descent iterates and hence improves the convergence rate. Also note that the algorithm uses a *constant* α. As a result, an NAG iteration can be even more economical relative to a conventional GD iteration.

Example 2.9 We illustrate the NAG algorithm by applying it to minimize the convex quadratic function

$$f(\boldsymbol{x}) = 5x_1^2 - 9x_1x_2 + 4.075x_2^2 - x_1$$

and comparing its performance with that of the conventional GD algorithm. Both algorithms will use the same initial point $\boldsymbol{x}_0 = [1 \;\; 1]^T$.

Solution

The gradient and Hessian of $f(\boldsymbol{x})$ are given by

$$g(\boldsymbol{x}) = \begin{bmatrix} 10x_1 - 9x_2 - 1 \\ -9x_1 + 8.15x_2 \end{bmatrix} \text{ and } \boldsymbol{H} = \begin{bmatrix} 10 & -9 \\ -9 & 8.15 \end{bmatrix}$$

respectively. To implement the NAG algorithm, we choose $\alpha = 0.044$ and run 109 iterations that yields a solution point

$$\boldsymbol{x}_{\text{nag}}^* = \begin{bmatrix} 16.3033 \\ 18.0036 \end{bmatrix}$$

at which $f(\boldsymbol{x}^*_{\text{nag}}) = -8.14999967$. For comparison purposes, we are interested in finding the perfect global minimizer of $f(\boldsymbol{x})$. Since the Hessian $\boldsymbol{H}$ is positive definite, $f(\boldsymbol{x})$ is strictly convex and the unique global solution can be obtained by solving the linear equation

$$\nabla f(\boldsymbol{x}) = \boldsymbol{H}\boldsymbol{x} + \boldsymbol{b} = \begin{bmatrix} 10 & -9 \\ -9 & 8.15 \end{bmatrix} \boldsymbol{x} - \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The solution is given by

$$\boldsymbol{x}^* = \begin{bmatrix} 10 & -9 \\ -9 & 8.15 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 16.3 \\ 18 \end{bmatrix}$$

at which $f(\boldsymbol{x}^*) = -8.15$. It is observed that $\boldsymbol{x}^*_{\text{nag}}$ is a good approximation of the global minimizer $\boldsymbol{x}^*$. For comparison, the conventional GD algorithm was also applied to minimize $f(\boldsymbol{x})$. Since the objective function is convex and quadratic, exact line search can be done using (2.11). With the same initial point, it took the GD algorithm 2618 iterations to reach a solution with comparable performance as that of $\boldsymbol{x}^*_{\text{nag}}$. Trajectories of the values of $f(\boldsymbol{x})$ at the first 109 NAG (in blue) and GD (in red) iterates are shown in Fig. 2.6. ■

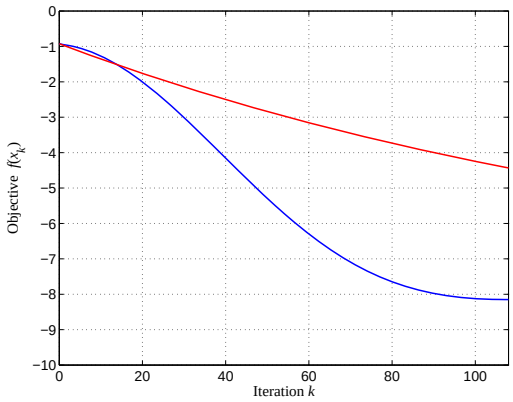

Figure 2.6. Objective function at the first 109 NAG (in blue) and GD (in red) iterates for Example 2.9.

2.5 Stochastic Gradient Descent Algorithms

In machine learning problems, one encounters objective functions of the form

$$f(\boldsymbol{x}) = \sum_{i=1}^{N} f_i(\boldsymbol{x}) \tag{2.20}$$

with a large N. By definition the gradient of $f(\boldsymbol{x})$ is evaluated term by term as $\nabla f(\boldsymbol{x}) = \sum_{i=1}^{N} \nabla f_i(\boldsymbol{x})$. This can however be problematic when N is very large. In its kth iteration, the method of *stochastic gradient descent* (SGD) selects a very small number of indices, say $\{i_1^{(k)}, i_2^{(k)}, \ldots, i_m^{(k)}\}$, uniformly *at random* from index set $\{1, 2, \ldots, N\}$ and computes the

average of the m gradients $\{\nabla f_{i_t^{(k)}}(\boldsymbol{x}_k)\}$ to generate search direction $\boldsymbol{d}_k = -\frac{1}{m}\sum_{t=1}^{m}\nabla f_{i_t^{(k)}}(\boldsymbol{x}_k)$. In machine learning problems, typically each component function $f_i(\boldsymbol{x})$ is associated to an individual data point, hence selecting an index is the same as selecting a data point. The success of the SGD algorithm has to do with independently randomly picking an individual data point (in each iteration). There is a great deal of research for convergence analysis, performance evaluation, and development of improved SGD algorithms. The standard SGD algorithm is summarized as follows.

Stochastic gradient descent algorithm

Step 1 Input initial point $\boldsymbol{x}_0$, learning rate α, integer m, and iteration number K. Set $k = 0$.

Step 2 If $k < K$, select m indices $\{i_1^{(k)}, i_2^{(k)}, \ldots, i_m^{(k)}\}$ uniformly at random from set $\{1, 2, \ldots, N\}$, compute search direction

$$\boldsymbol{d}_k = -\frac{1}{m}\sum_{t=1}^{m}\nabla f_{i_t^{(k)}}(\boldsymbol{x}_k)$$

and go to Step 3; otherwise, output $\boldsymbol{x}_K$ as the solution and stop.

Step 3 Update $\boldsymbol{x}_k$ to $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha\boldsymbol{d}_k$.

Step 4 Set $k := k+1$ and go to Step 2.

The selection of learning rate α is of importance for the success of SGD. To guarantee convergence, it is necessary to gradually reduce α and hence we need to denote the learning rate by α_k. A sufficient condition to assure convergence of the SGD is that

$$\sum_{k=0}^{\infty}\alpha_k = \infty \text{ and } \sum_{k=0}^{\infty}\alpha_k^2 < \infty$$

In practice, Step 3 of the SGD is revised to

Step 3' Update $\boldsymbol{x}_k$ to $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k\boldsymbol{d}_k$ where the learning rate is taken to be

$$\alpha_k = \left(1 - \frac{k}{K_1}\right)\alpha_0 + \frac{k}{K_1}\alpha_{K_1} \tag{2.21}$$

for the first $K1$ iterations where α_k is linearly reducing from α_0 down to α_{K_1}, and assumes a constant rate α_{K_1} thereafter.

Example 2.10 In this example, we consider Fisher's data set of Iris plants (see Example 1.2). The dataset to be considered in this example is a subset of Fisher's, only including two classes of the data, namely Iris Setosa and Iris Versicolour, where for each class we select 40 examples at random for training leaving the remaining 10 examples for testing purposes. The objective function involved in training is a logistic loss function of the form (see Example 2.1)

$$f(\hat{\boldsymbol{w}}) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(1 + e^{-y_n \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_n}\right) \tag{2.22}$$

where $N = 80$; $\{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, N\}$ is a dataset of 80 examples where each $\boldsymbol{x}_n$ is a vector of 5 components representing sepal length, sepal width, petal length, and petal width in cm, respectively, and unity (1). The labels are set to $y_n = 1$ for iris setosa and $y_n = -1$ for iris versicolour. Once the (global) minimizer $\hat{\boldsymbol{w}}^*$ of $f(\hat{\boldsymbol{w}})$ in (2.22) is obtained, it follows from Example 2.1 that

$\boldsymbol{x}$ **represents iris setosa if** $\hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}} > 0$ **and** $\boldsymbol{x}$ **represents iris versicolour if** $\hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}} < 0$. (2.23)

Applying GD and SGD algorithms to find the minimizer $\hat{\boldsymbol{w}}^*$ of problem (2.22).

Solution Function $f(\hat{\boldsymbol{w}})$ in (2.22) fits the form of (2.20), in effect we can write

$$f(\hat{\boldsymbol{w}}) = \sum_{i=1}^{N} f_i(\hat{\boldsymbol{w}}) \text{ with } f_i(\hat{\boldsymbol{w}}) = \frac{1}{N}\ln\left(1 + e^{-y_i \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_i}\right)$$

The gradient of $f(\hat{\boldsymbol{w}})$ is given by $\nabla f(\hat{\boldsymbol{w}}) = \sum_{i=1}^{N} \nabla f_i(\hat{\boldsymbol{w}})$ where

$$\nabla f_i(\hat{\boldsymbol{w}}) = -\frac{y_i e^{-y_i \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_i}}{N\left(1 + e^{-y_i \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_i}\right)} \hat{\boldsymbol{x}}_i$$

With a random initial point $\hat{\boldsymbol{w}}_0 = [-1.6656 \quad 0.1253 \quad 0.2877 \quad -1.1465 \quad -0.4326]^T$ and $\varepsilon = 5 \times 10^{-5}$, it took the GD algorithm $22{,}234$ iterations to a solution

$$\hat{\boldsymbol{w}}^* = [0.9378 \quad 4.8284 \quad -6.3854 \quad -4.3325 \quad 0.7087]^T$$

The objective function at $\boldsymbol{w}_0$ was 4.1013 while at $\hat{\boldsymbol{w}}^*$ it was reduced to 6.0511×10^{-5}. Using this $\hat{\boldsymbol{w}}^*$ in (2.23), the 20 iris in the test data were all classified correctly, see Fig. 2.7 which depicts the inner products $\hat{\boldsymbol{w}}^{*T}\hat{\boldsymbol{x}}$ for the 10 test examples from iris setosa and the 10 test examples from iris versicolour. The average CPU time consumed by the GD algorithm was about 34.5620 seconds.

Next, the SGD algorithm was applied with the same initial point $\hat{\boldsymbol{w}}_0$, $m = 2$, learning rate using (2.21) with $K_1 = 500$, $\alpha_0 = 1.2$, and $\alpha_{K_1} = 0.6$, and a total of $K = 900$ iterations, the solution produced by the algorithm is given by

$$\hat{\boldsymbol{w}}^* = [0.4753 \quad 6.5026 \quad -8.4365 \quad -5.1566 \quad 1.0121]^T$$

at which the objective function was reduced to 5.9670×10^{-5}. The values of of the objective function versus the number of iterations is shown in Fig. 2.8. As expected, the figure clearly shows the non-descent nature of SGD algorithm. On the other hand, we note the rapid reduction in the objective function from 4.1013 to 2.0939×10^{-4} in the first 30 iterations. Using the solution weight vector $\boldsymbol{w}^*$ in (2.23), the 20 samples in the test set were all classified

correctly. The average CPU time required by the SGD algorithm was about 0.1730 seconds, a very small fraction of that required by the GD algorithm. ■

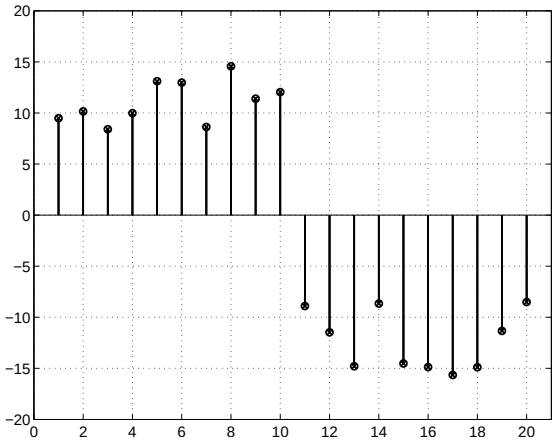

Figure 2.7 Values of $\boldsymbol{w}^{*T}\boldsymbol{x}$ for the test data, the first 10 of which are from iris setosa and the rest from iris versicolor. Clearly the test data are all classified correctly.

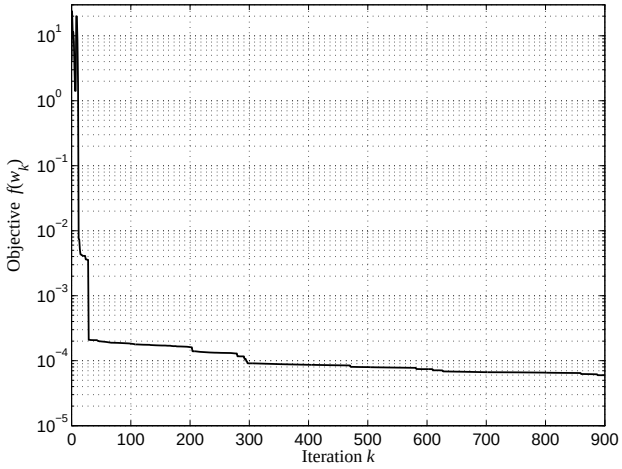

Figure 2.8 Objective function $f(\boldsymbol{w}_k)$ versus number of iterations k. Notice the rapid reduction in the objective in the first 30 iterations and the non-descent nature of the SGD algorithm.

2.6 Newton Algorithm

A second-order method known as the *Newton* (also known as the *Newton-Raphson*) *method* can be deduced by using the quadratic approximation of the Taylor series:

$$f(\boldsymbol{x}+\boldsymbol{\delta}) \approx f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{\delta} + \tfrac{1}{2}\boldsymbol{\delta}^T \nabla^2 f(\boldsymbol{x})\boldsymbol{\delta} \tag{2.24}$$

Let us first consider the case where $\nabla^2 f(\boldsymbol{x})$ is positive definite. The right-hand side of (2.24) in this case is a *convex* quadratic approximation of the objective function in a vicinity of point $\boldsymbol{x}$, and this quadratic function has a unique minimizer which can be found by computing its gradient with respect to $\boldsymbol{\delta}$ and then setting it to zero. This yields a system of linear equations for the

optimum change $\boldsymbol{\delta}$ in $\boldsymbol{x}$ to satisfy:

$$\nabla^2 f(\boldsymbol{x})\boldsymbol{\delta} = -\nabla f(\boldsymbol{x}) \tag{2.25}$$

Unlike the GD direction $-\nabla f(\boldsymbol{x})$ which is always descent, the Newton direction determined by solving (2.25) is not always a descent one. Now suppose a general *non-quadratic* function $f(\boldsymbol{x})$ is to be minimized and an arbitrary point $\boldsymbol{x}$ is assumed. Let us consider two scenarios:

(a) If Hessian $\boldsymbol{H}(\boldsymbol{x})$ is positive definite, the right-hand side of (2.12) in this case is a convex approximation of $f(\boldsymbol{x})$, hence the $\boldsymbol{\delta}$ obtained from (2.13) is a reasonable change for updating point $\boldsymbol{x}$. As long as point $\boldsymbol{x}$ is updated to $\boldsymbol{x}+\boldsymbol{\delta}$, the updating procedure may be repeated iteratively to yield a sequence of iterates which converge to a local minimizer. More formally, Newton method selects $\boldsymbol{x}_{k+1}$ as

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k \tag{2.26a}$$

where $\boldsymbol{d}_k$ solves the system of linear equations

$$\nabla^2 f(\boldsymbol{x}_k)\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k) \tag{2.26b}$$

and α_k is the value of α that minimizes $f(\boldsymbol{x}+\alpha\boldsymbol{d}_k)$. The vector $\boldsymbol{d}_k$ in (2.26b) is referred to as *Newton direction* at point $\boldsymbol{x}_k$.

(b) If Hessian $\boldsymbol{H}(\boldsymbol{x})$ is not positive definite, the $\boldsymbol{d}_k$ obtained from (2.26b) will fail to work because it does not yield a reduction in the objective function (why?). The problem can be overcome by forcing $\boldsymbol{H}(\boldsymbol{x})$ to become positive definite by means of a simple modification (see details below). The modified Hessian then replaces $\nabla^2 f(\boldsymbol{x}_k)$ in (2.26b) to calculate $\boldsymbol{d}_k$, which allows the iterative procedure to continue.

Figure 2.9 [10] illustrates the first two Newton steps for the convex function

$$f(x) = x^2 + 0.05e^{-x} + 8$$

With $x_0 = -5$, the first two Newton iterations yield $x_1 = -3.15$ and $x_2 = -0.7929$ while the global minimizer is at $\boldsymbol{x}^* = 0.0242$.

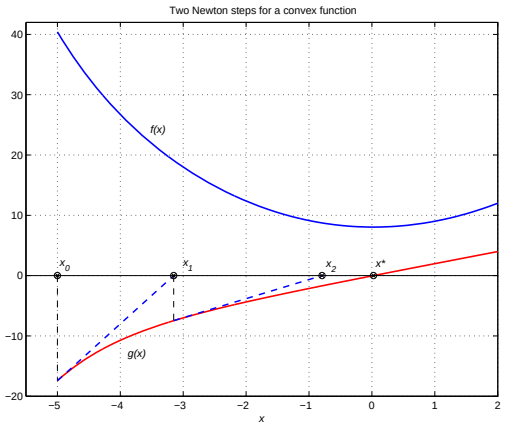

Figure 2.9 Two Newton iterations for a convex objective [10].

The Newton algorithm can now be summarized as follows.

Newton algorithm

Step 1 Input $\boldsymbol{x}_0$ and initialize the tolerance ε. Set $k = 0$.

Step 2 Compute $\nabla f(\boldsymbol{x}_k)$ and $\nabla^2 f(\boldsymbol{x}_k)$. If $\nabla^2 f(\boldsymbol{x}_k)$ is not positive definite, force it to become positive definite.

Step 3 Compute $\boldsymbol{d}_k$ by solving the linear system of equations in (2.26b).

Step 4 Using BTLS to find α_k, the value of α that minimizes $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$.

Step 5 Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$.

Step 6 If $\| \alpha_k \boldsymbol{d}_k \| < \varepsilon$, then output $\boldsymbol{x}^* = \boldsymbol{x}_{k+1}$ and stop; otherwise, set $k = k + 1$ and repeat from Step 2.

- MATLAB code of Newton algorithm, **`newton.m`**, is available from the course website.

Modification of the Hessian

If the Hessian is not positive definite in an iteration of Newton algorithm, it is forced to become positive definite in Step 2. One approach to modifying $\nabla^2 f(\boldsymbol{x}_k)$ is to replace it by

$$\hat{\boldsymbol{H}}_k = \frac{\nabla^2 f(\boldsymbol{x}_k) + \beta \boldsymbol{I}_n}{1 + \beta} \tag{2.27}$$

where $\boldsymbol{I}_n$ is the $n \times n$ identity matrix, β is set to be slightly larger than the magnitude of the smallest eigenvalue of $\nabla^2 f(\boldsymbol{x}_k)$. Note that if β happens to be very large, then (2.27) implies that

$$\hat{\boldsymbol{H}}_k \approx \boldsymbol{I}_n$$

which leads (2.26b) to

$$\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k)$$

thus the modification in (2.27) converts a Newton direction to a GD direction. We remark that a $\nabla^2 f(\boldsymbol{x}_k)$ that is not positive definite is likely to arise at points far from the solution where the GD algorithm is most effective in reducing the value of $f(\boldsymbol{x})$. Therefore, the modification in (2.27) leads to an algorithm that combines the complementary convergence properties of Newton and GD algorithms.

Example 2.11 Apply GD and Newton algorithms to minimize the Rosenbrock function (see Example 2.2)

$$f(x_1, x_2) = (x_1 - 1)^2 + 100(x_1^2 - x_2)^2$$

Each algorithm starts at initial point $\boldsymbol{x}_0 = [-1 \ \ -1]^T$ and use tolerance $\varepsilon = 10^{-9}$ for GD and

$\varepsilon = 10^{-6}$ for Newton algorithm.

Solution With $\boldsymbol{x}_0 = [-1 \ \ -1]^T$ and $\varepsilon = 10^{-9}$, it took the GD algorithm $15{,}555$ iterations to converge to a point $\boldsymbol{x}^*$ = $[0.999999586435335 \quad 0.999999171970399]^T$ at which $f(\boldsymbol{x}^*) = 1.7112 \times 10^{-13}$. The average CPU time of the algorithm was about 13.0417 seconds.

With $\boldsymbol{x}_0 = [-1 \ \ -1]^T$ and $\varepsilon = 10^{-6}$, it took Newton algorithm 21 iterations to converge to a point $\boldsymbol{x}^*$ = $[0.999999999999998 \quad 0.999999999999997]^T$ at which $f(\boldsymbol{x}^*) = 1.3509 \times 10^{-29}$. The average CPU time of the algorithm was about 5.1 ms. ■

From the above example, we see that Newton algorithm wins big in terms of both computational efficiency and solution accuracy. We have to point out however that the efficiency of Newton algorithm is closely related the *size* of the optimization problem, which is determined by the dimension n of variable $\boldsymbol{x}$. Newton algorithm becomes less efficient as n gets larger because solving the system of linear equations in (2.26b) requires $O(n^3)$ multiplications. In addition, computing the Hessian matrix $\nabla^2 f(\boldsymbol{x}_k)$ requires $O(n^2)$ function evaluations that adds additional burden to every Newton iteration. One of the solution techniques to this problem is to use a quasi-Newton algorithm which is addressed in Sec. 2.7.

2.7 Quasi-Newton Algorithms

2.7.1. Broyden-Fletcher-Goldfarb-Shanno (BFGS) *Algorithm*

One may take a unifying look at the GD and Newton algorithms by writing their search directions as

$$\boldsymbol{d}_k = -\boldsymbol{S}_k \nabla f(\boldsymbol{x}_k) \tag{2.28}$$

where

$$\boldsymbol{S}_k = \begin{cases} \boldsymbol{I} & \text{for GD} \\ \left(\nabla^2 f(\boldsymbol{x}_k)\right)^{-1} & \text{for Newton} \end{cases}$$

Quasi-Newton algorithms are developed to provide convergent rates comparable with that of Newton algorithm with reduced complexity, especially for problems of median and large scales. The distinction between Newton and quasi-Newton algorithms is that essentially Newton algorithm requires evaluating inverse of the Hessian while quasi-Newton algorithms are based only on gradient information and do not require explicit evaluation of the Hessian and its inverse.

Quasi-Newton algorithms follow the general algorithmic structure described earlier, where search direction $\boldsymbol{d}_k$ is evaluated using (2.28) where matrix $\boldsymbol{S}_k$ is updated in each iteration using gradient information. The most well-known quasi-Newton algorithm is the one known as BFGS algorithm, developed by Broyden, Fletcher, Goldfarb, and Shanno in late 1960s and early 1970s.

Matrix $\boldsymbol{S}_k$ is usually initiated as $\boldsymbol{S}_0 = \boldsymbol{I}$. In its kth iteration, the BFGS algorithm updates matrix $\boldsymbol{S}_k$ in a couple of simple steps:

- Compute $$\boldsymbol{\delta}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k, \quad \boldsymbol{\gamma}_k = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k) \tag{2.29a}$$
- Update $\boldsymbol{S}_k$ to

$$\boldsymbol{S}_{k+1} = \boldsymbol{S}_k + \left(1 + \frac{\boldsymbol{\gamma}_k^T \boldsymbol{S}_k \boldsymbol{\gamma}_k}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}\right) \frac{\boldsymbol{\delta}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} - \frac{\boldsymbol{\delta}_k \boldsymbol{\gamma}_k^T \boldsymbol{S}_k + \boldsymbol{S}_k \boldsymbol{\gamma}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \tag{2.29b}$$

and the algorithm may be summarized as follows.

BFGS Algorithm

Step 1 Input $\boldsymbol{x}_0$ and initialize the tolerance ε. Set $k = 0$ and $\boldsymbol{S}_0 = \boldsymbol{I}_n$. Compute $\nabla f(\boldsymbol{x}_0)$.

Step 2 Set $\boldsymbol{d}_k = -\boldsymbol{S}_k \nabla f(\boldsymbol{x}_k)$. Find α_k, the value of α that minimizes $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ using BTLS.

Step 3 Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$ and compute $\boldsymbol{\delta}_k$ using (2.29a).

Step 4 If $\| \boldsymbol{\delta}_k \|_2 < \varepsilon$, output $\boldsymbol{x}^* = \boldsymbol{x}_{k+1}$ and stop; otherwise proceed.

Step 5 Compute $\nabla f(\boldsymbol{x}_{k+1})$ and $\boldsymbol{\gamma}_k$ using (2.29a). Compute $\boldsymbol{S}_{k+1}$ using (2.29b). Set $k = k + 1$ and repeat from Step 2.

- MATLAB code of the BFGS algorithm, **`bfgs.m`**, is available from the course website.
- **Remark** For a convex quadratic objective function

$$f(\boldsymbol{x}) = \tfrac{1}{2} \boldsymbol{x}^T \boldsymbol{H} \boldsymbol{x} + \boldsymbol{x}^T \boldsymbol{b} + \kappa$$

it can readily be verified that the value of α_k in the line search step (Step 2) can be calculated by using the closed-form formula

$$\alpha_k = \frac{\boldsymbol{g}_k^T \boldsymbol{S}_k \boldsymbol{g}_k}{\boldsymbol{g}_k^T \boldsymbol{S}_k \boldsymbol{H} \boldsymbol{S}_k \boldsymbol{g}_k} \quad \text{where} \quad \boldsymbol{g}_k = \boldsymbol{H} \boldsymbol{x}_k + \boldsymbol{b} \tag{2.30}$$

see Prob. 2.5b.

2.7.2. Properties of the BFGS algorithm

The BFGS algorithm possesses several good properties and we describe two of them below.

1. If matrix $\boldsymbol{S}_0$ is positive definite, then matrices $\boldsymbol{S}_k$ for $k > 0$ produced by (2.29b) are positive definite provided that $\boldsymbol{\delta}_k^T \boldsymbol{\gamma}_k > 0$. Consequently, BFGS directions $\boldsymbol{d}_k = -\boldsymbol{S}_k \nabla f(\boldsymbol{x}_k)$ are descent directions.

2. If $f(\boldsymbol{x})$ is a strictly convex and quadratic with $\boldsymbol{x} \in R^n$, then the BFGS algorithm converges to the global minimizer of $f(\boldsymbol{x})$ with matrix $\boldsymbol{S}_k$ converging to the inverse of the Hessian of $f(\boldsymbol{x})$ in no more than n iterations.

Example 2.12 Apply the BFGS algorithm to minimize the objective function

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \boldsymbol{x} + \boldsymbol{x}^T \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

with initial point $\boldsymbol{x}_0 = [0 \;\; 0]^T$ and $\varepsilon = 10^{-6}$.

Solution To prepare BFGS iterations, we denote the Hessian of $f(\boldsymbol{x})$ by

$$\boldsymbol{H} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

and evaluate the gradient

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

1st iteration:

Step 1 $$\boldsymbol{x}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{g}_0 = \nabla f(\boldsymbol{x}_0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \; \boldsymbol{S}_0 = \boldsymbol{I}, \text{ and } \varepsilon = 10^{-6}.$$

Step 2 $$\boldsymbol{d}_0 = -\boldsymbol{S}_0 \nabla f(\boldsymbol{x}_0) = -\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\alpha_0 = \frac{\boldsymbol{g}_0^T \boldsymbol{S}_0 \boldsymbol{g}_0}{\boldsymbol{g}_0^T \boldsymbol{S}_0 \boldsymbol{H} \boldsymbol{S}_0 \boldsymbol{g}_0} = 0.5$$

Step 3

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + \alpha_0 \boldsymbol{d}_0 = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix}$$

$$\boldsymbol{\delta}_0 = \boldsymbol{x}_1 - \boldsymbol{x}_0 = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix}$$

Step 4 Since $\|\boldsymbol{\delta}_0\|_2 > \varepsilon$, go to Step 5.

Step 5 $$\boldsymbol{g}_1 = \nabla f(\boldsymbol{x}_1) = \begin{bmatrix} 0 \\ -0.5 \end{bmatrix}, \quad \boldsymbol{\gamma}_0 = \nabla f(\boldsymbol{x}_1) - \nabla f(\boldsymbol{x}_0) = \begin{bmatrix} -1 \\ -0.5 \end{bmatrix}$$

$$\boldsymbol{S}_1 = \boldsymbol{S}_0 + \left(1 + \frac{\boldsymbol{\gamma}_0^T \boldsymbol{S}_0 \boldsymbol{\gamma}_0}{\boldsymbol{\gamma}_0^T \boldsymbol{\delta}_0}\right) \frac{\boldsymbol{\delta}_0 \boldsymbol{\delta}_0^T}{\boldsymbol{\gamma}_0^T \boldsymbol{\delta}_0} - \frac{(\boldsymbol{\delta}_0 \boldsymbol{\gamma}_0^T \boldsymbol{S}_0 + \boldsymbol{S}_0 \boldsymbol{\gamma}_0 \boldsymbol{\delta}_0^T)}{\boldsymbol{\gamma}_0^T \boldsymbol{\delta}_0}$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + (1 + \frac{1.25}{0.5}) \begin{bmatrix} 0.5 & 0 \\ 0 & 0 \end{bmatrix} - \frac{1}{0.5} \begin{bmatrix} 1 & 0.25 \\ 0.25 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0.75 & -0.5 \\ -0.5 & 1 \end{bmatrix}$$

2nd iteration:

Step 2

$$\boldsymbol{d}_1 = -\boldsymbol{S}_1 \boldsymbol{g}_1 = \begin{bmatrix} -0.25 \\ 0.5 \end{bmatrix}$$

$$\alpha_1 = \frac{\boldsymbol{g}_1^T \boldsymbol{S}_1 \boldsymbol{g}_1}{\boldsymbol{g}_1^T \boldsymbol{S}_1 \boldsymbol{H} \boldsymbol{S}_1 \boldsymbol{g}_1} = \frac{0.25}{\begin{bmatrix} 0.25 & -0.5 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0.25 \\ -0.5 \end{bmatrix}} = 2$$

Step 3

$$\boldsymbol{x}_2 = \boldsymbol{x}_1 + \alpha_1 \boldsymbol{d}_1 = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix} + 2 \cdot \begin{bmatrix} -0.25 \\ 0.5 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

$$\boldsymbol{\delta}_1 = \boldsymbol{x}_2 - \boldsymbol{x}_1 = \begin{bmatrix} -0.5 \\ 1 \end{bmatrix}$$

Step 4 Since $\|\boldsymbol{\delta}_1\|_2 > \varepsilon$, go to Step 5.

Step 5 At $\boldsymbol{x}_2$ we obtain

$$\boldsymbol{g}_2 = \nabla f(\boldsymbol{x}_2) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

which means that $\boldsymbol{x}_2 = [-1 \quad 1]^T$ is a stationary point. Since $\boldsymbol{H}$ is P.D., $\boldsymbol{x}_2$ is a strong global minimizer of the problem.

In case we want to check formula (2.29b) to see if it indeed recovers the inverse of the Hessian $\boldsymbol{H}$ in at most n (here $n = 2$) iterations, we compute $\boldsymbol{\gamma}_1 = \boldsymbol{g}_2 - \boldsymbol{g}_1 = \begin{bmatrix} 0 \\ 0.5 \end{bmatrix}$ and

$$\boldsymbol{S}_2 = \boldsymbol{S}_1 + \left(1 + \frac{\boldsymbol{\gamma}_1^T \boldsymbol{S}_1 \boldsymbol{\gamma}_1}{\boldsymbol{\gamma}_1^T \boldsymbol{\delta}_1}\right) \frac{\boldsymbol{\delta}_1 \boldsymbol{\delta}_1^T}{\boldsymbol{\gamma}_1^T \boldsymbol{\delta}_1} - \frac{(\boldsymbol{\delta}_1 \boldsymbol{\gamma}_1^T \boldsymbol{S}_1 + \boldsymbol{S}_1 \boldsymbol{\gamma}_1 \boldsymbol{\delta}_1^T)}{\boldsymbol{\gamma}_1^T \boldsymbol{\delta}_1}$$

$$= \begin{bmatrix} 0.75 & -0.5 \\ -0.5 & 1 \end{bmatrix} + (1 + \frac{0.25}{0.5}) \begin{bmatrix} 0.5 & -1 \\ -1 & 2 \end{bmatrix} - \frac{1}{0.5} \begin{bmatrix} 0.25 & -0.5 \\ -0.5 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} (= \boldsymbol{H}^{-1}) \quad \blacksquare$$

2.7.3. Memoryless BFGS Algorithm

For large-scale problems, producing, storing, and manipulation of matrix $\boldsymbol{S}_k$ (see (2.29b)), which is usually not sparse, can be problematic. The *memoryless BFGS algorithm* deals with this difficulty by replacing (2.29b) with a simplified updating formula that reduces the computation of the

search direction $-\boldsymbol{S}_{k+1}\nabla f(\boldsymbol{x}_{k+1})$ to a few inner products. We start by pointing out a fact that the BFGS formula in (2.29b) can be written as

$$\boldsymbol{S}_{k+1} = \left(\boldsymbol{I} - \frac{\boldsymbol{\delta}_k \boldsymbol{\gamma}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}\right) \boldsymbol{S}_k \left(\boldsymbol{I} - \frac{\boldsymbol{\gamma}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}\right) + \frac{\boldsymbol{\delta}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \tag{2.31}$$

where $\boldsymbol{\delta}_k$ and $\boldsymbol{\gamma}_k$ are defined by (2.29a). The memoryless BFGS computes matrix $\boldsymbol{S}_{k+1}$ by replacing matrix $\boldsymbol{S}_k$ on the right-hand side of (2.31) with the identity matrix so that (2.31) becomes

$$\boldsymbol{S}_{k+1} = \left(\boldsymbol{I} - \frac{\boldsymbol{\delta}_k \boldsymbol{\gamma}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}\right)\left(\boldsymbol{I} - \frac{\boldsymbol{\gamma}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}\right) + \frac{\boldsymbol{\delta}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \tag{2.32}$$

As a result, calculating $\boldsymbol{S}_{k+1}$ no longer relies on $\boldsymbol{S}_k$, hence the name "memoryless". Moreover, matrix $\boldsymbol{S}_{k+1}$ needs not to be computed explicitly. In effect, the search direction $\boldsymbol{d}_{k+1} = -\boldsymbol{S}_{k+1}\nabla f(\boldsymbol{x}_{k+1})$ can be obtained by computing a few inner products as follows:

Step 1: Compute

$$\rho_k = \frac{1}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k},\ t_k = \boldsymbol{\delta}_k^T \nabla f(\boldsymbol{x}_{k+1}),\ \text{and}\ \boldsymbol{q}_k = \nabla f(\boldsymbol{x}_{k+1}) - \rho_k t_k \boldsymbol{\gamma}_k$$

Step 2: Compute search direction

$$\boldsymbol{d}_{k+1} = \rho_k(\boldsymbol{\gamma}_k^T \boldsymbol{q}_k - t_k)\boldsymbol{\delta}_k - \boldsymbol{q}_k \tag{2.33}$$

• MATLAB code of the memoryless BFGS algorithm, **`bfgs_ML.m`**, is available from the course website.

Example 2.13 Apply Newton, BFGS, and memoryless BFGS (ML-BFGS) algorithms to the unconstrained minimization problem in Example 2.10.

Solution

The objective function and its gradient are given by

$$f(\hat{\boldsymbol{w}}) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(1 + e^{-y_n \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_n}\right)$$

and

$$\nabla f(\hat{\boldsymbol{w}}) = \sum_{i=1}^{N} \nabla f_i(\hat{\boldsymbol{w}}) \quad \text{where} \quad \nabla f_i(\hat{\boldsymbol{w}}) = -\frac{y_i e^{-y_i \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_i}}{N\left(1 + e^{-y_i \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_i}\right)} \hat{\boldsymbol{x}}_i$$

To implement Newton's algorithm, we evaluate the Hessian of $f(\hat{\boldsymbol{w}})$ as

$$\nabla^2 f(\hat{\boldsymbol{w}}) = \sum_{i=1}^{N} \nabla^2 f_i(\hat{\boldsymbol{w}}) \ \text{where}\ \nabla^2 f_i(\hat{\boldsymbol{w}}) = \frac{e^{-y_i \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_i}}{N\left(1 + e^{-y_i \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_i}\right)^2} \hat{\boldsymbol{x}}_i \hat{\boldsymbol{x}}_i^T$$

Clearly, $\nabla^2 f_i(\hat{\boldsymbol{w}})$ is always positive semidefinite and hence the Hessian $\nabla^2 f(\hat{\boldsymbol{w}})$ is very likely positive definite when the number data points, N, is at least greater than five. With $\beta = 0.1$ (see (2.37)) and the same initial $\hat{\boldsymbol{w}}_0$ as in Example 2.10, it took Newton algorithm 11 iterations to converge to the solution

$$\hat{\boldsymbol{w}}^* = \begin{bmatrix} -0.0345 & 3.8682 & -4.0096 & -9.5477 & 6.3745 \end{bmatrix}^T$$

at which the objective function assumes the value $f(\hat{\boldsymbol{w}}^*) = 5.6931 \times 10^{-5}$. As expected, the classifier based on the above solution recognizes all 20 test iris samples correctly. The average CPU time required by Newton algorithm was about 0.0084 seconds.

Next, the BFGS algorithm was applied to the same data set used in Example 2.10. With the same initial $\hat{\boldsymbol{w}}_0$ as in Example 2.10, it took BFGS algorithm 16 iterations to converge to the solution

$$\hat{\boldsymbol{w}}^* = \begin{bmatrix} 0.9525 & 5.6573 & -6.8623 & -4.3244 & 0.7033 \end{bmatrix}^T$$

at which the objective function assumes the value $f(\hat{\boldsymbol{w}}^*) = 6.1151 \times 10^{-5}$. The classifier based on the above solution recognizes all 20 test iris samples correctly. The average CPU time required by BFGS algorithm was about 0.0085 seconds.

Finally, the ML-BFGS algorithm was applied to the same data set with the same initial point. It took ML-BFGS 15 iterations to converge to the solution

$$\hat{\boldsymbol{w}}^* = \begin{bmatrix} 1.1924 & 6.3638 & -7.7392 & -4.6986 & 0.8289 \end{bmatrix}^T$$

at which the objective function assumes the value $f(\hat{\boldsymbol{w}}^*) = 5.6859 \times 10^{-5}$. The classifier based on the above solution recognizes all 20 test iris samples correctly. The average CPU time required by BFGS-ML algorithm was about 0.0066 seconds.

The performance of these algorithms, plus GD and SGD algorithms (see Example 2.10), in terms of computational efficiency is summarized below. For a fair comparison, the number of iterations for each algorithm was adjusted so that the solution accuracies achieved by different algorithms were comparable with each other.

Algorithm	# of Iterations	$f(\hat{\boldsymbol{w}}^*)$	Average CPU Time (Sec.)
GD	22,234	6.0511×10^{-5}	34.6494
SGD	900	5.9670×10^{-5}	0.1730
Newton	11	5.6931×10^{-5}	0.0084
BFGS	16	6.1151×10^{-5}	0.0085
ML-BFGS	15	5.6859×10^{-5}	0.0066

From the above table it is observed that GD algorithm is rather slow relative to the other algorithms. SGD was considerably faster than GD, but Newton as well as BFGS and BFGS-ML

algorithms were the best performers in this case. The high efficiency of Newton algorithm has to do with a very small data size ($N = 80$) and Hessian (5×5). In effect, the complexity of Newton algorithm can be severe even for problems of moderate sizes. For large-scale problems, SGD is expected to become much competitive and often times the choice of the algorithm. ■

Problems

2.1 As illustrated in Figure P2.1, we are given a smooth function $h(x)$ over interval $[0,\ 1]$ where any point x in the interval is associated with a shaded region. The problem we consider here is to find the "sweet spot" $\boldsymbol{x}^*$ in interval $[0,\ 1]$ at which the area of the shaded region reaches maximum.

(a) Formulate the problem as a standard constrained minimization problem (i.e. in the form of Eq. (1.1)).

(b) Suppose function $h(x)$ represents the straight line connecting point $[0 \quad 1]^T$ and $[1 \quad 0]^T$. Solve the problem obtained from part (a). A *numerical* solution is required.

(c) Solve the problem obtained from part (a) where $h(x) = x^2 - 2x + 1$. A *numerical* solution is required.

(d) For an arbitrary $h(x)$, derive an equation that the sweet spot $\boldsymbol{x}^*$ must satisfy.

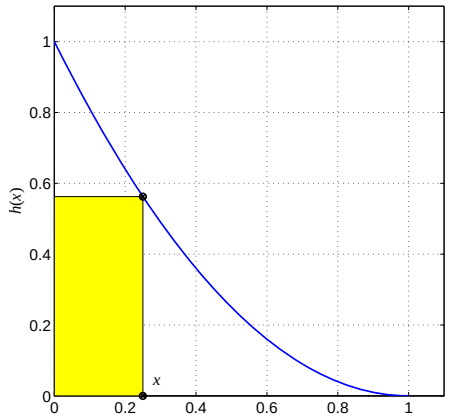

Figure P2.1 for Problem 2.1.

2.2 Formulate the problem of finding the shortest distance between two separate ellipses R and S shown in Fig. P2.2 as a standard constrained minimization problem. Note that both

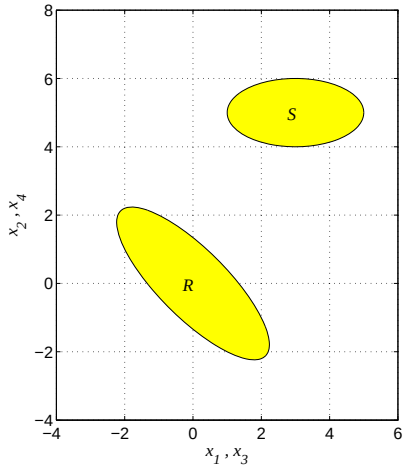

Figure P2.2 for Problem 2.2.

regions R and S have elliptical boundaries which are defined by

$$\text{boundary of region } R: \quad \frac{(x_1 - x_2)^2}{18} + \frac{(x_1 + x_2)^2}{2} = 1$$

$$\text{boundary of region } S: \quad \frac{(x_3 - 3)^2}{4} + (x_4 - 5)^2 = 1$$

respectively.

2.3 Compute gradient and Hessian of the functions given below.

(a) $f(x_1, x_2) = x_1^2 - 2x_2 \sin x_1 + 100$

(b) $f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T \boldsymbol{H}\boldsymbol{x} + \boldsymbol{x}^T \boldsymbol{b} + \kappa$ where $\boldsymbol{H} \in R^{n\times n}$, $\boldsymbol{b} \in R^{n\times 1}$, and $\kappa \in R^{1\times 1}$ are known data.

(c) $f(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \ln\left(1 + e^{-\boldsymbol{\theta}^T \boldsymbol{x}_i}\right)$ where variable $\boldsymbol{\theta} \in R^{n\times 1}$, $\ln(\cdot)$ denotes natural logarithm, and $\{\boldsymbol{x}_i \text{ for } i = 1, 2, \ldots, N\}$ with $\boldsymbol{x}_i \in R^{n\times 1}$ is a known data set.

2.4 Point $\boldsymbol{x}$ is called a *stationary point* of function $f(\boldsymbol{x})$ if $\nabla f(\boldsymbol{x}) = \boldsymbol{0}$. Find and classify the stationary points of the following functions as minimizer, maximizer, or none of the above (in that case the stationary point is called a *saddle point*):

(a) $f(\boldsymbol{x}) = 2x_1^2 + x_2^2 - 2x_1x_2 + 2x_1^3 + x_1^4$

(b) $f(\boldsymbol{x}) = x_1^2x_2^2 - 4x_1^2x_2 + 4x_1^2 + 2x_1x_2^2 + x_2^2 - 8x_1x_2 + 8x_1 - 4x_2$

(c) $f(\boldsymbol{x}) = (x_1^2 - x_2)^2 + x_1^5$

(Hint: Use the second-order sufficient conditions).

2.5 Function $f(\boldsymbol{x})$ is convex in a region if its Hessian is positive semidefinite in that region. Function $f(\boldsymbol{x})$ is concave if $-f(\boldsymbol{x})$ is convex. Determine whether the following functions are convex or concave or else:

(a) $f(\boldsymbol{x}) = x_1^2 + \cosh(x_2)$

(b) $f(\boldsymbol{x}) = x_1^2 + 2x_2^2 + 2x_3^2 + x_4^2 - x_1x_2 + x_1x_3 - 2x_2x_4 + x_1x_4$

(c) $f(\boldsymbol{x}) = x_1^2 - 2x_2^2 - 2x_3^2 + x_4^2 - x_1x_2 + x_1x_3 - 2x_2x_4 + x_1x_4$

(d) $f(\hat{\boldsymbol{w}}) = \frac{1}{N}\sum_{i=1}^{N} \ln\left(1 + e^{-y_i \hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}_i}\right)$ where variable $\hat{\boldsymbol{w}} \in R^{n\times 1}$, $\ln(\cdot)$ denotes natural logarithm, $\hat{\boldsymbol{x}}_i \in R^{n\times 1}$ and $y_i \in R^{1\times 1}$ are known data.

2.6

(a) Find all stationary points of the objective function

$$f(\boldsymbol{x}) = (x_1 + x_2)^2 + \left[2(x_1^2 + x_2^2 - 1) - \tfrac{1}{3}\right]^2$$

(b) Classify each stationary point found from part (a) as a minimizer, maximizer, or saddle point.

2.7 This exercise problem is concerned with performing a line search for a convex quadratic function

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{H}\boldsymbol{x} + \boldsymbol{x}^T\boldsymbol{b} + \kappa$$

where $\boldsymbol{H} \in R^{n\times n}$ is positive definite.

(a) Given the kth iterate $\boldsymbol{x}_k$ and search direction $\boldsymbol{d}_k = -\boldsymbol{g}_k$ where $\boldsymbol{g}_k = \nabla f(\boldsymbol{x}_k)$, show that the minimizer of the one-dimensional optimization problem (called *line search*)

$$\underset{\alpha}{\text{minimize}}\ f(\boldsymbol{x}_k + \alpha\boldsymbol{d}_k)$$

is given by

$$\alpha_k = \frac{\boldsymbol{g}_k^T\boldsymbol{g}_k}{\boldsymbol{g}_k^T\boldsymbol{H}\boldsymbol{g}_k}$$

(b) If the search direction is specified as $\boldsymbol{d}_k = -\boldsymbol{S}_k\boldsymbol{g}_k$, show that the minimizer of the one-dimensional optimization problem

$$\underset{\alpha}{\text{minimize}}\ f(\boldsymbol{x}_k + \alpha\boldsymbol{d}_k)$$

is given by

$$\alpha_k = \frac{\boldsymbol{g}_k^T\boldsymbol{S}_k\boldsymbol{g}_k}{\boldsymbol{g}_k^T\boldsymbol{S}_k\boldsymbol{H}\boldsymbol{S}_k\boldsymbol{g}_k}$$

where $\boldsymbol{g}_k = \nabla f(\boldsymbol{x})$.

2.8

(a) With initial point $\boldsymbol{x}_0 = [0.5 \quad 0.1]^T$, perform two GD iterations *by hand* for the Rosenbrock function $f(x_1, x_2) = (x_1 - 1)^2 + 100(x_1^2 - x_2)^2$. Refer to Example 2.2 for its gradient and Hessian. Use optimized step size $\alpha_0 = 0.0026$ for the first iteration and $\alpha_1 = 0.0592$ for the second iteration.

(b) With the same initial point, perform two Newton iterations *by hand* for the Rosenbrock function. Use optimized step size $\alpha_0 = 1$ for the first iteration and $\alpha_1 = 0.2640$ for the second iteration.

(c) It is known that the Rosenbrock function possesses a global minimizer $\boldsymbol{x}^* = [1 \quad 1]^T$ at which $f(\boldsymbol{x}^*) = 0$. Use this information to evaluate the performance of GD and Newton algorithms in terms of closeness of the iterates to the global solution and how much reduction in the objective function is achieved by each iteration of the GD and Newton algorithms.

2.9 Optimization techniques can be of use for solving systems of linear or nonlinear equations. This exercise problem examines a simple case of this matter where one tries to solve a system of two nonlinear equations, namely, to find point(s) $\boldsymbol{x}$ that satisfy both of the equations

$$f_1(\boldsymbol{x}) = \mathbf{0} \text{ and } f_2(\boldsymbol{x}) = \mathbf{0}$$

(a) Show that point $\boldsymbol{x}$ is a solution of the above equations if and only if $\boldsymbol{x}$ is a global minimizer of the unconstrained problem

$$\text{minimize } F(\boldsymbol{x}) = f_1^2(\boldsymbol{x}) + f_2^2(\boldsymbol{x})$$

(b) By applying the idea from part (a), find an approximate solution of the system of equations

$$f_1(\boldsymbol{x}) = x_1^3 - x_2 - 1 = 0$$
$$f_2(\boldsymbol{x}) = x_1^2 - x_2 = 0$$

Specifically, Apply GD, Newton, and BFGS algorithms to minimize function $F(\boldsymbol{x})$ with initial point $\boldsymbol{x}_0 = \begin{bmatrix} 0.25 & 0.15 \end{bmatrix}^T$. Report and comment on the numerical results obtained.

2.10 Apply GD, Newton, and BFGS algorithms to minimize the objective function (known as the Beale function) given by

$$f(\boldsymbol{x}) = (x_1 x_2 - x_1 + 1.5)^2 + (x_1 x_2^2 - x_1 + 2.25)^2 + (x_1 x_2^3 - x_1 + 2.625)^2$$

(a) Derive the gradient and Hessian of the Beale function.

(b) With each of the four initial points given below

$$\boldsymbol{x}_0^{(1)} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \boldsymbol{x}_0^{(2)} = \begin{bmatrix} -3 \\ 3 \end{bmatrix}, \quad \boldsymbol{x}_0^{(3)} = \begin{bmatrix} 2 \\ -3 \end{bmatrix}, \quad \boldsymbol{x}_0^{(4)} = \begin{bmatrix} -3 \\ -3 \end{bmatrix}$$

and convergence tolerance ε = 10^{-6}, apply GD algorithm to minimize the Beale function. Report results in terms of the solution point found and the value of the objective function at the solution point with at least 8 decimal places and verify if the solution obtained is a local or global minimizer (explain why).

(c) With the same initial points and convergence tolerance as in part (b), apply Newton algorithm to minimize the Beale function. Report the solution point found and the value of the objective function at the solution point with at least 8 decimal places and verify if the solution obtained is a local or global minimizer (explain why).

(d) With the same initial points and convergence tolerance as in part (b), apply BFGS algorithm to minimize the Beale function. Report the solution point found and the value of the objective function at the solution point with at least 8 decimal places and verify if the solution obtained is a local or global minimizer (explain why).

(e) Compare and comment the results obtained in parts (b)-(d).

2.11 Apply the memoryless BFGS algorithm to

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \boldsymbol{x} + \boldsymbol{x}^T \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

with initial point $\boldsymbol{x}_0 = [0 \quad 0]^T$. Perform iterations *by hand* until the exact solution is found and compare the result with that obtained in Example 2.12. This problem is intended to be carried out without using software.