CHAPTER

LEAST SQUARES REGRESSION

22

CHAPTER CONTENTS

Let us consider a regression problem of learning a real-valued function $y = f(\boldsymbol{x})$ defined on d-dimensional input space from input-output paired training samples $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$. In practice, the training output values $\{y_i\}_{i=1}^n$ may be noisy observations of the true values $\{f(\boldsymbol{x}_i)\}_{i=1}^n$. In this chapter, the most fundamental regression technique called LS *regression* is introduced.

22.1 METHOD OF LS

Let $\boldsymbol{\theta}$ be the parameter of model $f_{\boldsymbol{\theta}}(\boldsymbol{x})$. The method of LS, learns the parameter $\boldsymbol{\theta}$ so that the squared difference between training output $\{y_i\}_{i=1}^n$ and model output $f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ is minimized:

$$\widehat{\boldsymbol{\theta}}_{\mathrm{LS}} = \operatorname*{argmin}_{\boldsymbol{\theta}} J_{\mathrm{LS}}(\boldsymbol{\theta}),$$

where

$$J_{\mathrm{LS}}(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n}\Big(y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\Big)^2. \tag{22.1}$$

Since the squared error $(y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i))^2$ is the ℓ_2-norm of the *residual*

$$y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i),$$

the LS method is also referred to as ℓ_2*-loss minimization*.

An Introduction to Statistical Machine Learning. DOI: 10.1016/B978-0-12-802121-7.00033-9

22.2 SOLUTION FOR LINEAR-IN-PARAMETER MODEL

For a linear-in-parameter model (see Section 21.1),

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{j=1}^{b} \theta_i \phi_i(\boldsymbol{x}) = \boldsymbol{\theta}^\top \boldsymbol{\phi}(\boldsymbol{x}),$$

the training squared error J_{LS} is expressed as

$$J_{\mathrm{LS}}(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\theta}\|^2,$$

where

$$\boldsymbol{y} = (y_1, \ldots, y_n)^\top$$

is the n-dimensional vector consisting of training output values and $\boldsymbol{\Phi}$ is the $n \times b$ matrix called *design matrix*:

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi_1(\boldsymbol{x}_1) & \cdots & \phi_b(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ \phi_1(\boldsymbol{x}_n) & \cdots & \phi_b(\boldsymbol{x}_n) \end{pmatrix}.$$

The partial derivative of the training squared error J_{LS} with respect to parameter $\boldsymbol{\theta}$ is given by

$$\nabla_{\boldsymbol{\theta}} J_{\mathrm{LS}} = \left(\frac{\partial J_{\mathrm{LS}}}{\partial \theta_1}, \ldots, \frac{\partial J_{\mathrm{LS}}}{\partial \theta_b}\right)^\top = \boldsymbol{\Phi}^\top \boldsymbol{y} - \boldsymbol{\Phi}^\top \boldsymbol{\Phi}\boldsymbol{\theta}.$$

Setting this to zero shows that the solution should satisfy

$$\boldsymbol{\Phi}^\top \boldsymbol{\Phi}\boldsymbol{\theta} = \boldsymbol{\Phi}^\top \boldsymbol{y}.$$

Then the LS solution $\widehat{\boldsymbol{\theta}}_{\mathrm{LS}}$ is given by

$$\widehat{\boldsymbol{\theta}}_{\mathrm{LS}} = \boldsymbol{\Phi}^\dagger \boldsymbol{y},$$

where $\boldsymbol{\Phi}^\dagger$ is the *generalized inverse* (see Fig. 22.1) of the design matrix $\boldsymbol{\Phi}$. When $\boldsymbol{\Phi}^\top \boldsymbol{\Phi}$ is invertible, the generalized inverse $\boldsymbol{\Phi}^\dagger$ is expressed as

$$\boldsymbol{\Phi}^\dagger = (\boldsymbol{\Phi}^\top \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^\top.$$

A MATLAB code of LS regression for the linear-in-parameter model with sinusoidal basis functions,

$$\boldsymbol{\phi}(x) = \left(1, \sin\frac{x}{2}, \cos\frac{x}{2}, \sin\frac{2x}{2}, \cos\frac{2x}{2}, \ldots, \sin\frac{15x}{2}, \cos\frac{15x}{2}\right)^\top,$$

is provided in Fig. 22.3. In the program, instead of explicitly obtaining the generalized inverse $\boldsymbol{\Phi}^\dagger$ by the `pinv` function, the equation $\boldsymbol{\Phi}\boldsymbol{\theta} = \boldsymbol{y}$ is directly solved by

A matrix $\boldsymbol{X}$ is called the *generalized inverse* of real matrix $\boldsymbol{A}$, if it satisfies the following four conditions:

$$\begin{aligned}\boldsymbol{A}\boldsymbol{X}\boldsymbol{A} &= \boldsymbol{A},\\ \boldsymbol{X}\boldsymbol{A}\boldsymbol{X} &= \boldsymbol{X},\\ (\boldsymbol{X}\boldsymbol{A})^\top &= \boldsymbol{X}\boldsymbol{A},\\ (\boldsymbol{A}\boldsymbol{X})^\top &= \boldsymbol{A}\boldsymbol{X}.\end{aligned}$$

The generalized inverse of $\boldsymbol{A}$ is often denoted by $\boldsymbol{A}^\dagger$ and is also referred to as the *Moore-Penrose pseudoinverse*. The ordinary inverse is defined only for full-rank square matrices, while the generalized inverse is defined for singular or even nonsquare matrices. In MATLAB, the generalized inverse can be obtained by the `pinv` function. A generalized inverse of a scalar a is given by

$$a^\dagger = \begin{cases} 1/a & (a \neq 0),\\ 0 & (a = 0),\end{cases}$$

and a generalized inverse of a $d \times m$ matrix $\boldsymbol{A}$ is given by

$$\boldsymbol{A}^\dagger = \sum_{k=1}^{\min(d,m)} \kappa_k^\dagger \boldsymbol{\psi}_k \boldsymbol{\phi}_k^\top,$$

where $\boldsymbol{\psi}_k$, $\boldsymbol{\phi}_k$, and κ_k are a *left singular vector*, a *right singular vector*, and a *singular value* of $\boldsymbol{A}$, respectively (see Fig. 22.2). When $\boldsymbol{A}$ is square and full rank, its generalized inverse $\boldsymbol{A}^\dagger$ is reduced to the ordinary inverse $\boldsymbol{A}^{-1}$.

FIGURE 22.1

Generalized inverse.

`t=p\y`, which is computationally more efficient. The behavior of LS regression is illustrated in Fig. 22.4, showing that a complicated nonlinear function can be nicely approximated.

A LS method where the loss function for the ith training sample is weighted according to $w_i \geq 0$ is referred to as *weighted LS*:

$$\min_{\boldsymbol{\theta}} \frac{1}{2} \sum_{i=1}^{n} w_i \left(y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\right)^2.$$

Singular value decomposition is an extension *eigenvalue decomposition* (see Fig. 6.2) to nonsquare matrices. For $d \times m$ matrix $\boldsymbol{A}$, a d-dimensional nonzero vector $\boldsymbol{\psi}$, an m-dimensional nonzero vector $\boldsymbol{\varphi}$, and a non-negative scalar κ such that

$$\boldsymbol{A}\boldsymbol{\varphi} = \kappa\boldsymbol{\psi}$$

are called a *left singular vector*, a *right singular vector*, and a *singular value* of $\boldsymbol{A}$, respectively. Generally, there exist c singular values $\kappa_1, \ldots, \kappa_c$, where

$$c = \min(d, m).$$

Singular vectors $\boldsymbol{\varphi}_1, \ldots, \boldsymbol{\varphi}_c$ and $\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_c$ corresponding to singular values $\kappa_1, \ldots, \kappa_c$ are mutually orthogonal and are usually normalized, i.e., they are *orthonormal* as

$$\boldsymbol{\varphi}_k^\top \boldsymbol{\varphi}_{k'} = \begin{cases} 1 & (k = k') \\ 0 & (k \neq k') \end{cases} \quad \text{and} \quad \boldsymbol{\psi}_k^\top \boldsymbol{\psi}_{k'} = \begin{cases} 1 & (k = k'), \\ 0 & (k \neq k'). \end{cases}$$

A matrix $\boldsymbol{A}$ can be expressed by using its singular vectors and singular values as

$$\boldsymbol{A} = \sum_{k=1}^{c} \kappa_k \boldsymbol{\psi}_k \boldsymbol{\varphi}_k^\top.$$

In MATLAB, singular value decomposition can be performed by the `svd` function.

FIGURE 22.2

Singular value decomposition.

The solution of weighted LS is given as

$$(\boldsymbol{\Phi}^\top \boldsymbol{W} \boldsymbol{\Phi})^\dagger \boldsymbol{\Phi}^\top \boldsymbol{W} \boldsymbol{y},$$

where $\boldsymbol{W}$ is the diagonal matrix consisting of weights:

$$\boldsymbol{W} = \operatorname{diag}(w_1, \ldots, w_n).$$

The kernel model introduced in Section 21.2 is also linear in terms of parameters:

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{j=1}^{n} \theta_j K(\boldsymbol{x}, \boldsymbol{x}_j). \tag{22.2}$$

```
n=50; N=1000; x=linspace(-3,3,n)'; X=linspace(-3,3,N)';
pix=pi*x; y=sin(pix)./(pix)+0.1*x+0.05*randn(n,1);

p(:,1)=ones(n,1); P(:,1)=ones(N,1);
for j=1:15
  p(:,2*j)=sin(j/2*x); p(:,2*j+1)=cos(j/2*x);
  P(:,2*j)=sin(j/2*X); P(:,2*j+1)=cos(j/2*X);
end
t=p\y; F=P*t;

figure(1); clf; hold on; axis([-2.8 2.8 -0.5 1.2]);
plot(X,F,'g-'); plot(x,y,'bo');
```

FIGURE 22.3

MATLAB code for LS regression.

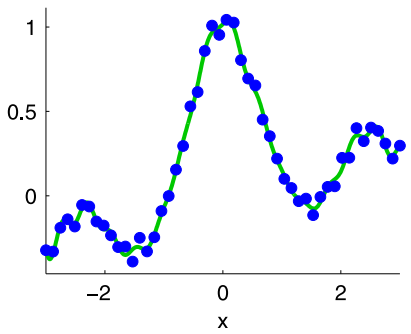


FIGURE 22.4

Example of LS regression with sinusoidal basis functions $\boldsymbol{\phi}(x) = (1, \sin\frac{x}{2}, \cos\frac{x}{2}, \sin\frac{2x}{2}, \cos\frac{2x}{2}, \ldots, \sin\frac{15x}{2}, \cos\frac{15x}{2})^\top$.

Thus, the LS solution for the kernel model can also be obtained in the same way, by replacing the design matrix $\boldsymbol{\Phi}$ with the *kernel matrix*:

$$\boldsymbol{K} = \begin{pmatrix} K(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_1, \boldsymbol{x}_n) \\ \vdots & \ddots & \vdots \\ K(\boldsymbol{x}_n, \boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_n, \boldsymbol{x}_n) \end{pmatrix}.$$

22.3 PROPERTIES OF LS SOLUTION

Let us consider *singular value decomposition* (see Fig. 22.2) of design matrix $\boldsymbol{\Phi}$:

$$\boldsymbol{\Phi} = \sum_{k=1}^{\min(n,b)} \kappa_k \boldsymbol{\psi}_k \boldsymbol{\varphi}_k^\top,$$

where $\boldsymbol{\psi}_k$, $\boldsymbol{\varphi}_k$, and κ_k are a *left singular vector*, a *right singular vector*, and a *singular value* of $\boldsymbol{\Phi}$, respectively.

As explained in Fig. 22.1, the generalized inverse $\boldsymbol{\Phi}^\dagger$ of the matrix $\boldsymbol{\Phi}$ can be expressed as

$$\boldsymbol{\Phi}^\dagger = \sum_{k=1}^{\min(n,b)} \kappa_k^\dagger \boldsymbol{\varphi}_k \boldsymbol{\psi}_k^\top,$$

where $\kappa^\dagger$ is the generalized inverse of scalar κ:

$$\kappa^\dagger = \begin{cases} 1/\kappa & (\kappa \neq 0), \\ 0 & (\kappa = 0). \end{cases}$$

Then the LS solution $\widehat{\boldsymbol{\theta}}_{\mathrm{LS}}$ can be expressed as

$$\widehat{\boldsymbol{\theta}}_{\mathrm{LS}} = \sum_{k=1}^{\min(n,b)} \kappa_k^\dagger (\boldsymbol{\psi}_k^\top \boldsymbol{y}) \boldsymbol{\varphi}_k.$$

The n-dimensional vector consisting of the output values of the LS solution $f_{\widehat{\boldsymbol{\theta}}_{\mathrm{LS}}}$ at training input samples $\{\boldsymbol{x}_i\}_{i=1}^n$ is given by

$$\left(f_{\widehat{\boldsymbol{\theta}}_{\mathrm{LS}}}(\boldsymbol{x}_1), \ldots, f_{\widehat{\boldsymbol{\theta}}_{\mathrm{LS}}}(\boldsymbol{x}_n)\right)^\top = \boldsymbol{\Phi}\widehat{\boldsymbol{\theta}}_{\mathrm{LS}} = \boldsymbol{\Phi}\boldsymbol{\Phi}^\dagger \boldsymbol{y}.$$

Since $(\boldsymbol{\Phi}\boldsymbol{\Phi}^\dagger)^2 = \boldsymbol{\Phi}\boldsymbol{\Phi}^\dagger$ and $(\boldsymbol{\Phi}\boldsymbol{\Phi}^\dagger)^\top = \boldsymbol{\Phi}\boldsymbol{\Phi}^\dagger$, $\boldsymbol{\Phi}\boldsymbol{\Phi}^\dagger$ is the *projection* matrix onto the range of $\boldsymbol{\Phi}$ and thus the LS solution actually projects the training output vector $\boldsymbol{y}$ onto the range of $\boldsymbol{\Phi}$.

When there exists a parameter $\boldsymbol{\theta}^*$ such that the true learning target function f is given by $f_{\boldsymbol{\theta}^*}$, the vector consisting of the output values of the true function f at training input samples $\{\boldsymbol{x}_i\}_{i=1}^n$ is given by

$$\left(f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_n)\right)^\top = \boldsymbol{\Phi}\boldsymbol{\theta}^*.$$

This means that the true output vector belongs to the range of $\boldsymbol{\Phi}$, and thus the LS method tries to remove noise included in $\boldsymbol{y}$ by projecting it onto the range of $\boldsymbol{\Phi}$ (Fig. 22.5).

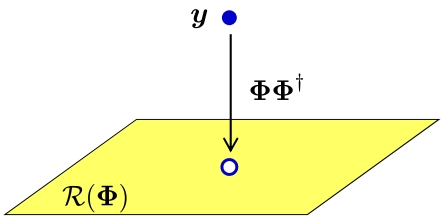


FIGURE 22.5

Geometric interpretation of LS method for linear-in-parameter model. Training output vector $\boldsymbol{y}$ is projected onto the range of $\boldsymbol{\Phi}$, denoted by $\mathcal{R}(\boldsymbol{\Phi})$, for denoising purposes.

When the expectation of noise is zero, the LS solution $\widehat{\boldsymbol{\theta}}_{\mathrm{LS}}$ is an *unbiased estimator* of the true parameter $\boldsymbol{\theta}^*$:

$$\mathbb{E}[\widehat{\boldsymbol{\theta}}_{\mathrm{LS}}] = \boldsymbol{\theta}^*,$$

where $\mathbb{E}$ denotes the expectation over the noise included in training output $\{y_i\}_{i=1}^n$. Even if the model is *misspecified* (see Section 17.1.2), i.e., $f \neq f_{\boldsymbol{\theta}}$ for any $\boldsymbol{\theta}$, $\mathbb{E}[\widehat{\boldsymbol{\theta}}_{\mathrm{LS}}]$ converges to the optimal parameter in the model, as the number of training samples, n, tends to infinity. This property is called *asymptotic unbiasedness* (see Section 13.2).

22.4 LEARNING ALGORITHM FOR LARGE-SCALE DATA

The size of design matrix $\boldsymbol{\Phi}$ is $n \times b$ and thus it cannot be stored in the memory space if the number of training samples, n, and the number of parameters, b, are large. In such a situation, the *stochastic gradient* algorithm introduced in Section 15.3 is useful, which updates parameter $\boldsymbol{\theta}$ to go down the gradient of the training squared error J_{LS}. For linear-in-parameter models, the training squared error J_{LS} is a *convex function* (see Fig. 8.3). Thus, the solution obtained by the stochastic gradient algorithm gives the global optimal solution. The algorithm of the stochastic gradient algorithm for linear-in-parameter models is summarized in Fig. 22.6.

In Fig. 22.7, a MATLAB code of stochastic gradient descent for LS regression with the Gaussian kernel model is provided:

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{j=1}^{n} \theta_j K(\boldsymbol{x}, \boldsymbol{x}_j),$$

where

$$K(\boldsymbol{x}, \boldsymbol{c}) = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{c}\|^2}{2h^2}\right).$$

1. Initialize the parameter $\boldsymbol{\theta}$.
2. Choose the ith training sample $(\boldsymbol{x}_i, y_i)$ randomly.
3. Update the parameter $\boldsymbol{\theta}$ to go down the gradient as

$$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} - \varepsilon \nabla J_{\mathrm{LS}}^{(i)}(\boldsymbol{\theta}),$$

   where ε is a small positive scalar called the *step size*, and $\nabla J_{\mathrm{LS}}^{(i)}$ is the gradient of the training squared error for the ith training sample:

$$\nabla J_{\mathrm{LS}}^{(i)}(\boldsymbol{\theta}) = -\boldsymbol{\phi}(\boldsymbol{x}_i)\Big(y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\Big).$$

4. Iterate 2–3 until convergence.

FIGURE 22.6

Algorithm of stochastic gradient descent for LS regression with a linear-in-parameter model.

For $n = 50$ training samples, the Gaussian bandwidth is set at $h = 0.3$. The behavior of the algorithm is illustrated in Fig. 22.8, where starting from a random initial value, a function close to the final solution is obtained after around 200 iterations. However, 11,556 iterations are required until convergence.

The convergence speed of the stochastic gradient algorithm depends on the step size (`e=0.1` in Fig. 22.7) and the convergence condition (`norm(t-t0)<0.000001` in Fig. 22.7). The algorithm may converge faster if these parameters are tuned. In particular, as for the step size, starting from a large value and gradually decreasing it would be appropriate, as discussed in Section 15.3. However, appropriately implementing this idea is not straightforward in practice.

22.5 LEARNING ALGORITHM FOR HIERARCHICAL MODEL

Stochastic gradient descent is a popular choice also for training hierarchical models such as neural networks introduced in Section 21.3. However, since the training squared error J_{LS} is not a convex function in hierarchical models, there exist multiple local optimal solutions in general and the gradient method may only find one of the local optimal solutions (Fig. 22.9). In practice, starting from different initial values, gradient descent is performed multiple times and the solution that gives the smallest training squared error is chosen as the most promising one.

For the sigmoidal neural network model introduced in Section 21.3,

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{j=1}^{b} \alpha_j \phi(\boldsymbol{x}; \boldsymbol{\beta}_j, \gamma_j),$$

```
n=50; x=linspace(-3,3,n)'; pix=pi*x;
y=sin(pix)./(pix)+0.1*x+0.05*randn(n,1);

hh=2*0.3^2; t=randn(n,1); e=0.1;
for o=1:n*1000
  i=ceil(rand*n);
  ki=exp(-(x-x(i)).^2/hh); t0=t+e*ki*(y(i)-ki'*t);
  if norm(t-t0)<0.000001, break, end
  t=t0;
end

N=1000; X=linspace(-3,3,N)';
K=exp(-(repmat(X.^2,1,n)+repmat(x.^2',N,1)-2*X*x')/hh);
F=K*t;
figure(1); clf; hold on; axis([-2.8 2.8 -0.5 1.2]);
plot(X,F,'g-'); plot(x,y,'bo');
```

FIGURE 22.7

MATLAB code of stochastic gradient descent for LS regression with the Gaussian kernel model.

where

$$\phi(\boldsymbol{x};\boldsymbol{\beta},\gamma) = \frac{1}{1+\exp\left(-\boldsymbol{x}^\top\boldsymbol{\beta}-\gamma\right)},$$

the gradient $\nabla_{\boldsymbol{\theta}} J_{\mathrm{LS}}^{(i)}$ can be computed efficiently as

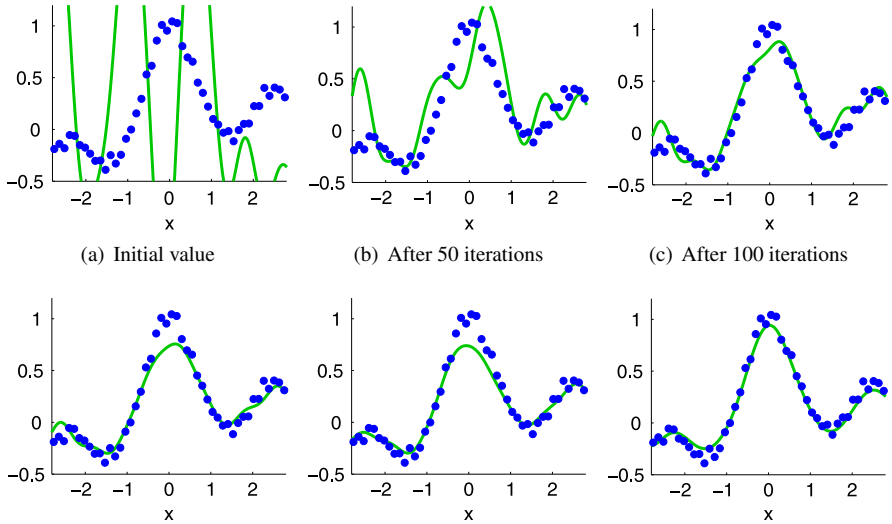
$$\begin{aligned}
\frac{\partial J_{\mathrm{LS}}^{(i)}}{\partial \alpha_j} &= -z_{i,j} r_i, \\
\frac{\partial J_{\mathrm{LS}}^{(i)}}{\partial \beta_j^{(k)}} &= -\alpha_j z_{i,j}\left(1-z_{i,j}\right) x_i^{(k)} r_i, \\
\frac{\partial J_{\mathrm{LS}}^{(i)}}{\partial \gamma_j} &= -\alpha_j z_{i,j}\left(1-z_{i,j}\right) r_i,
\end{aligned}$$

where $\beta_j^{(k)}$ denotes the kth element of vector $\boldsymbol{\beta}_j$, $x_i^{(k)}$ denotes the kth element of vector $\boldsymbol{x}_i$, r_i denotes the residual for the ith training sample,

$$r_i = y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i),$$

and $z_{i,j}$ denotes the output of the jth basis function for the ith training sample:

$$z_{i,j} = \phi(\boldsymbol{x}_i;\boldsymbol{\beta}_j,\gamma_j).$$



(a) Initial value (b) After 50 iterations (c) After 100 iterations

(d) After 150 iterations (e) After 200 iterations (f) Final solution (after 11,556 iterations)

FIGURE 22.8

Example of stochastic gradient descent for LS regression with the Gaussian kernel model. For $n = 50$ training samples, the Gaussian bandwidth is set at $h = 0.3$.

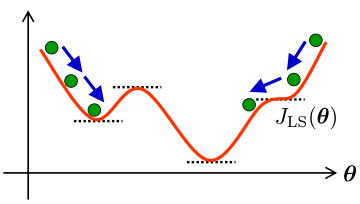


FIGURE 22.9

Gradient descent for nonlinear models. The training squared error J_{LS} is nonconvex and there exist multiple local optimal solutions in general.

Since the residual r_i is propagated backward when computing the gradient, the gradient method for neural networks is often referred to as *error back-propagation* [85].

If input $\boldsymbol{x}_i$ is augmented as

$$\widetilde{\boldsymbol{x}}_i = (\boldsymbol{x}_i^\top, 1)^\top \in \mathbb{R}^{d+1},$$

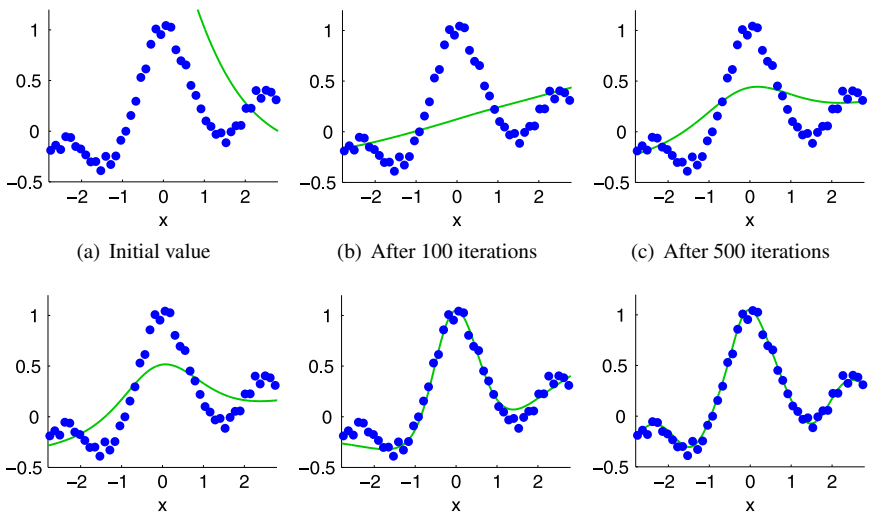
```
n=50; N=1000; x=linspace(-3,3,n)'; pix=pi*x;
y=sin(pix)./(pix)+0.1*x+0.05*randn(n,1);

x(:,2)=1; d=1; m=20; e=0.1; a=ones(m,1); b=randn(m,d+1);
for o=1:n*100000
  i=ceil(rand*n); z=1./(1+exp(-b*x(i,:)')); r=y(i)-a'*z;
  a0=a+e*z*r; b0=b+e*(a.*z.*(1-z)*x(i,:))*r;
  if norm(a-a0)+norm(b-b0)<0.00000001, break, end
  a=a0; b=b0;
end

X=linspace(-3,3,N)'; X(:,2)=1; Y=a'*(1./(1+exp(-b*X')));
figure(1); clf; hold on; axis([-2.8 2.8 -0.5 1.2]);
plot(X(:,1),Y,'g-'); plot(x(:,1),y,'bo');
```

FIGURE 22.10

MATLAB code for error back-propagation algorithm.



(a) Initial value (b) After 100 iterations (c) After 500 iterations

(d) After 1000 iterations (e) After 10,000 iterations (f) Final solution (after 1229,742 iterations)

FIGURE 22.11

Example of regression by error back-propagation algorithm.

the basis parameter,

$$\widetilde{\boldsymbol{\beta}}_j = (\boldsymbol{\beta}_j^\top, \gamma_j)^\top \in \mathbb{R}^{d+1},$$

can be updated together as

$$\frac{\partial J_{\mathrm{LS}}^{(i)}}{\partial \widetilde{\beta}_j^{(k)}} = -\alpha_j z_{i,j} \left(1 - z_{i,j}\right) \widetilde{x}_i^{(k)} r_i \quad \text{for } k = 1, \ldots, d+1.$$

A MATLAB code of the error back-propagation algorithm for a three-layered neural network is provided in Fig. 22.10, and its behavior is illustrated in Fig. 22.11. This shows that the solution fits the training samples well after many iterations.

Although the error back-propagation is applicable to any neural networks models, learning a neural network having many layers is often difficult in practice. Indeed, the error is propagated only to parameters close to the output layer, and parameters near the input layer are rarely updated. To efficiently learn a *deep neural network*, iteratively initializing each layer one by one based on unsupervised learning is shown to be useful [55]. Such a method will be explained in Chapter 36.