CHAPTER

NONLINEAR DIMENSIONALITY REDUCTION

36

CHAPTER CONTENTS

In this chapter, supervised and unsupervised methods of *nonlinear dimensionality reduction* are introduced, including approaches based on kernels and neural networks.

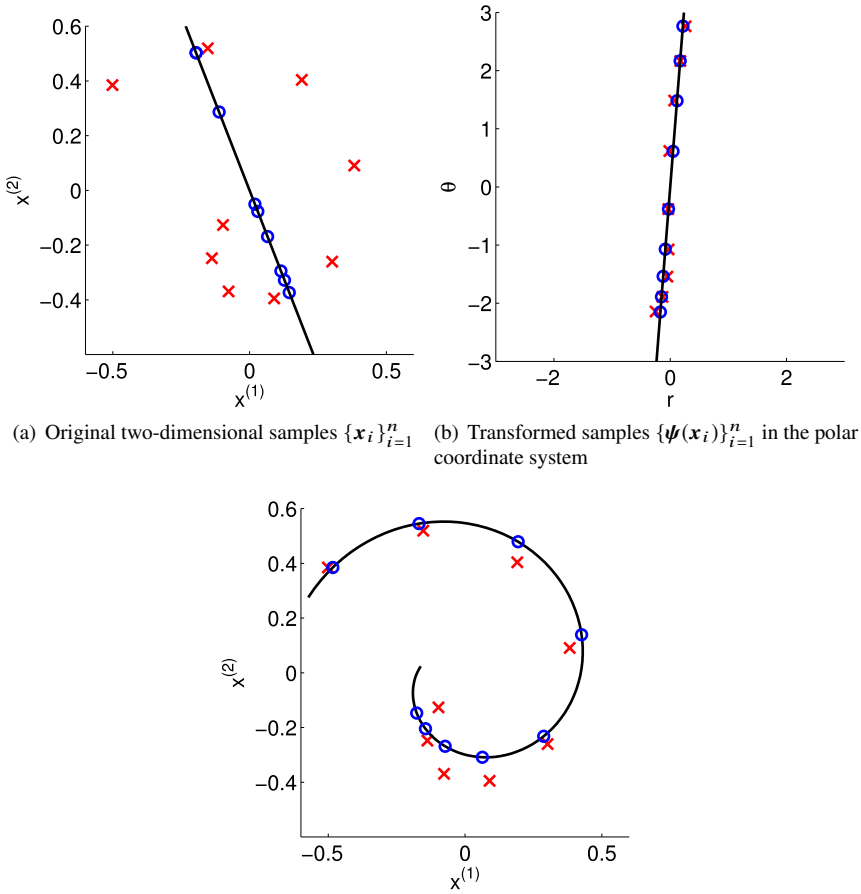
36.1 DIMENSIONALITY REDUCTION WITH KERNEL TRICK

The linear dimensionality reduction methods explained in the previous chapter handle training samples only in terms of their inner products. This means that the *kernel trick* (see Section 27.4) is applicable to obtain nonlinear methods. More specifically, training input samples $\{\boldsymbol{x}_i\}_{i=1}^n$ are first transformed by a nonlinear mapping $\boldsymbol{\psi}$ and then ordinary linear dimensionality reduction algorithms are applied in the transformed space, which corresponds to nonlinear dimensionality reduction in the original input space. In this section, kernel-based nonlinear dimensionality reduction methods are introduced.

36.1.1 KERNEL PCA

Let us illustrate the behavior of PCA in a feature space. The original two-dimensional samples $\boldsymbol{x}_i = (x_i^{(1)}, x_i^{(2)})^\top$ are plotted in Fig. 36.1(a). Due to the curved spiral shape,

An Introduction to Statistical Machine Learning. DOI: 10.1016/B978-0-12-802121-7.00047-9



(a) Original two-dimensional samples $\{\boldsymbol{x}_i\}_{i=1}^n$

(b) Transformed samples $\{\boldsymbol{\psi}(\boldsymbol{x}_i)\}_{i=1}^n$ in the polar coordinate system

(c) Result of PCA transformed back to the original input space

FIGURE 36.1

Nonlinear PCA in a feature space. "×" denotes a sample, the solid line denotes the one-dimensional embedding subspace found by PCA, and "○" denotes a projected sample.

linear PCA does not work properly for this data set. If the samples are transformed into the *polar system* (i.e. radius r and angle θ), the spiral shape can be nicely unfolded, as plotted in Fig. 36.1(b). Then linear PCA in the transformed space can well capture the global structure of the unfolded data. Finally, transforming back the projected samples to the original input space gives nice nonlinear projection of the original spiral-shape samples, as plotted in Fig. 36.1(c).

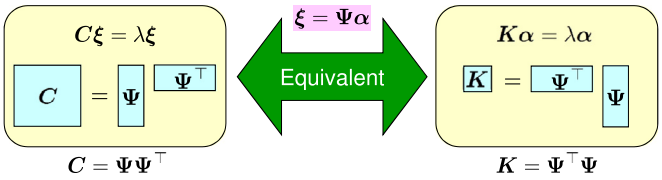


FIGURE 36.2

Eigenvalue problems for PCA. Appropriately choosing the expression of eigenvalue problem depending on whether matrix $\boldsymbol{\Psi}$ is fat or skinny allows us to reduce the computational costs.

If the dimensionality of the feature space is not high, directly performing PCA as explained above is fine. That is, the eigenvalue problem for the total scatter matrix $\boldsymbol{C}$ in the feature space is solved:

$$\boldsymbol{C}\boldsymbol{\xi} = \lambda\boldsymbol{\xi},$$

where

$$\boldsymbol{C} = \sum_{i=1}^{n} \boldsymbol{\psi}(\boldsymbol{x}_i)\boldsymbol{\psi}(\boldsymbol{x}_i)^\top.$$

However, if the dimensionality of the feature space is high, the computational complexity grows significantly. More extremely, if the Gaussian kernel,

$$K(\boldsymbol{x},\boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|^2}{2h^2}\right), \tag{36.1}$$

is used as a nonlinear mapping, the dimensionality of the feature space is actually infinite, and thus explicitly performing PCA in the feature space is not possible.

To cope with this problem, let us consider the eigenvalue problem for the *kernel matrix* $\boldsymbol{K}$ [88]:

$$\boldsymbol{K}\boldsymbol{\alpha} = \lambda\boldsymbol{\alpha},$$

where the (i,i')th element of $\boldsymbol{K}$ is defined as

$$K_{i,i'} = K(\boldsymbol{x}_i,\boldsymbol{x}_{i'}) = \langle\boldsymbol{\psi}(\boldsymbol{x}_i),\boldsymbol{\psi}(\boldsymbol{x}_{i'})\rangle.$$

The matrices $\boldsymbol{C}$ and $\boldsymbol{K}$ can be expressed by using the *design matrix*

$$\boldsymbol{\Psi} = (\boldsymbol{\psi}(\boldsymbol{x}_1),\ldots,\boldsymbol{\psi}(\boldsymbol{x}_n))$$

as

$$\boldsymbol{C} = \boldsymbol{\Psi}\boldsymbol{\Psi}^\top \quad \text{and} \quad \boldsymbol{K} = \boldsymbol{\Psi}^\top\boldsymbol{\Psi}.$$

This implies that the eigenvalues of $\boldsymbol{C}$ and $\boldsymbol{K}$ are actually common. Furthermore, eigenvector $\boldsymbol{\alpha}$ of $\boldsymbol{K}$ and eigenvector $\boldsymbol{\xi}$ of $\boldsymbol{C}$ are related to each other as follows (see Fig. 36.2):

$$\boldsymbol{\xi} = \boldsymbol{\Psi}\boldsymbol{\alpha} \quad \text{and} \quad \boldsymbol{\alpha} = \boldsymbol{\Psi}^\top \boldsymbol{\xi}.$$

While the size of covariance matrix $\boldsymbol{C}$ depends on the dimensionality of the feature space, the size of kernel matrix $\boldsymbol{K}$ depends only on the number of samples and is independent of the dimensionality of the feature space. Thus, if the dimensionality of the feature space is larger than the number of samples, solving the eigenvalue problem with $\boldsymbol{K}$ is computationally more efficient.

As explained in Section 35.2.1, PCA requires centering of samples. However, when the eigenvalue problem with $\boldsymbol{K}$ is solved, feature vectors $\{\boldsymbol{\psi}(\boldsymbol{x}_i)\}_{i=1}^n$ are not explicitly handled and thus feature vectors cannot be centralized directly. In kernel PCA, the kernel matrix $\boldsymbol{K}$ is *implicitly* centralized as

$$\boldsymbol{K} \longleftarrow \boldsymbol{H}\boldsymbol{K}\boldsymbol{H},$$

where

$$\boldsymbol{H} = \boldsymbol{I}_n - \frac{1}{n}\mathbf{1}_{n\times n}$$

is the *centering matrix*, $\boldsymbol{I}_n$ denotes the n-dimensional identity matrix, and $\mathbf{1}_{n\times n}$ is the $n \times n$ matrix with all ones.

Another issue to be considered is that kernel PCA requires eigenvectors to be normalized as $\|\boldsymbol{\xi}_j\| = 1$, but solving the eigenvalue problem with $\boldsymbol{K}$ usually gives eigenvectors such that $\|\boldsymbol{\alpha}_j\| = 1$. Therefore, normalization of eigenvectors should be carried out explicitly as

$$\boldsymbol{\alpha}_j \longleftarrow \frac{1}{\sqrt{\lambda_j}}\boldsymbol{\alpha}_j \quad \text{for } j = 1,\ldots,m,$$

which comes from

$$\|\boldsymbol{\xi}_j\| = \sqrt{\|\boldsymbol{\xi}_j\|^2} = \sqrt{\|\boldsymbol{\Psi}\boldsymbol{\alpha}_j\|^2} = \sqrt{\langle \boldsymbol{\Psi}^\top\boldsymbol{\Psi}\boldsymbol{\alpha}_j, \boldsymbol{\alpha}_j\rangle} = \sqrt{\langle \boldsymbol{K}\boldsymbol{\alpha}_j, \boldsymbol{\alpha}_j\rangle} = \sqrt{\lambda_j}.$$

Summarizing the above discussions, final embedding solutions of samples $\{\boldsymbol{x}_i\}_{i=1}^n$ by kernel PCA are given by

$$(z_1,\ldots,z_n) = \left(\frac{1}{\sqrt{\lambda_1}}\boldsymbol{\alpha}_1,\ldots,\frac{1}{\sqrt{\lambda_m}}\boldsymbol{\alpha}_m\right)^\top \boldsymbol{H}\boldsymbol{K}\boldsymbol{H},$$

where $\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_m$ are eigenvectors of $\boldsymbol{H}\boldsymbol{K}\boldsymbol{H}$ corresponding to the m largest eigenvalues. Similarly, an embedding solution of a new sample $\boldsymbol{x}$ by kernel PCA is given by

$$z = \left(\frac{1}{\sqrt{\lambda_1}}\boldsymbol{\alpha}_1,\ldots,\frac{1}{\sqrt{\lambda_m}}\boldsymbol{\alpha}_m\right)^\top \boldsymbol{H}\left(\boldsymbol{k} - \frac{1}{n}\boldsymbol{K}\mathbf{1}_n\right),$$

```
n=200; a=linspace(0,pi,n/2);
u=[a.*cos(a) (a+pi).*cos(a)]';
v=[a.*sin(a) (a+pi).*sin(a)]';
x=[u v]; y=[ones(1,n/2) 2*ones(1,n/2)]';

x2=sum(x.^2,2); hh=2*2^2; H=eye(n)-ones(n,n)/n;
K=exp(-(repmat(x2,1,n)+repmat(x2',n,1)-2*x*x')/hh);
G=H*K*H; [A,L]=eigs(G,2); z=(diag(diag(L).^(-1/2))*A'*G)';

figure(1); clf; hold on;
plot(z(y==1,1),z(y==1,2),'bo');
plot(z(y==2,1),z(y==2,2),'rx');
```

FIGURE 36.3

MATLAB code for kernel PCA with Gaussian kernels.

where $\mathbf{1}_n$ denotes the n-dimensional vectors with all ones and

$$\boldsymbol{k} = (K(\boldsymbol{x},\boldsymbol{x}_1),\ldots,K(\boldsymbol{x},\boldsymbol{x}_n))^\top.$$

A MATLAB code of PCA for Gaussian kernel (36.1) is provided in Fig. 36.3, and its behavior is illustrated in Fig. 36.4. This shows that, while the original two-dimensional samples are not linearly separable, PCA for Gaussian kernel with width $h = 2$ gives linearly separable embedding in the feature space.

36.1.2 LAPLACIAN EIGENMAP

A kernelized version of locality preserving projection is called the *Laplacian eigenmap* [13]. In the generalized eigenvalue problem of locality preserving projection,

$$\boldsymbol{X}\boldsymbol{L}\boldsymbol{X}^\top\boldsymbol{\xi} = \lambda\boldsymbol{X}\boldsymbol{D}\boldsymbol{X}^\top\boldsymbol{\xi},$$

multiplying $\boldsymbol{X}^\top$ from the left-hand side and letting $\boldsymbol{\xi} = \boldsymbol{X}\boldsymbol{\beta}$ yield

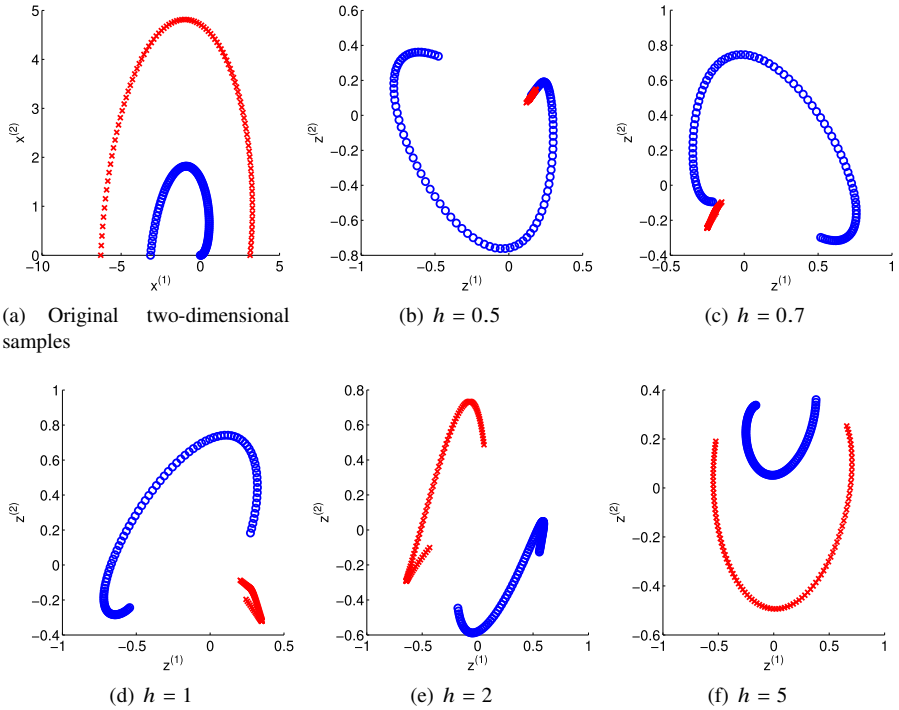
$$\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{L}\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{\beta} = \lambda\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{D}\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{\beta}.$$

If $\boldsymbol{X}^\top\boldsymbol{X}$ is replaced with kernel matrix $\boldsymbol{K}$, a kernelized version of locality preserving projection is obtained as

$$\boldsymbol{K}\boldsymbol{L}\boldsymbol{K}\boldsymbol{\beta} = \lambda\boldsymbol{K}\boldsymbol{D}\boldsymbol{K}\boldsymbol{\beta}.$$

If $\boldsymbol{K}$ is invertible, multiplying $\boldsymbol{K}^{-1}$ from the left-hand side and letting $\boldsymbol{\alpha} = \boldsymbol{K}\boldsymbol{\beta}$ yield

$$\boldsymbol{L}\boldsymbol{\alpha} = \lambda\boldsymbol{D}\boldsymbol{\alpha}, \tag{36.2}$$



(a) Original two-dimensional samples (b) $h = 0.5$ (c) $h = 0.7$

(d) $h = 1$ (e) $h = 2$ (f) $h = 5$

FIGURE 36.4

Examples of kernel PCA with Gaussian kernels. Original two-dimensional samples are transformed to infinite-dimensional feature space by Gaussian kernels with width h, and then PCA is applied to reduce the dimensionality to two.

which is the eigenvalue problem that the Laplacian eigenmap solves. Let $\lambda_1 \geq \cdots \geq \lambda_n \geq 0$ and $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$ be the generalized eigenvalues and generalized eigenvectors of Eq. (36.2). Then the definition of graph Laplacian matrix $\boldsymbol{L}$, Eq. (35.2), yields $\boldsymbol{L}\mathbf{1}_n = \mathbf{0}_n$. This means that $\lambda_n = 0$ and $\boldsymbol{\alpha}_n = \mathbf{1}_n$ hold, which is trivial. In the Laplacian eigenmap, this trivial eigenvector is removed and the final embedding solution is obtained as

$$(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = (\boldsymbol{\alpha}_{n-1}, \boldsymbol{\alpha}_{n-2}, \ldots, \boldsymbol{\alpha}_{n-m})^\top.$$

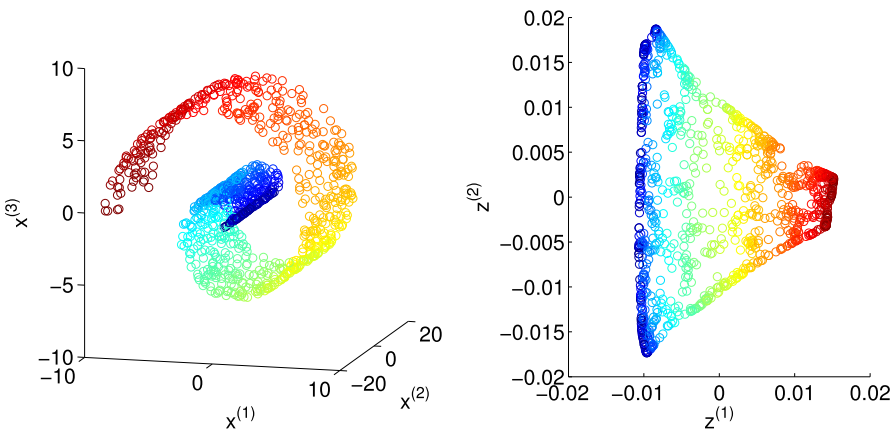
If the similarity matrix $\boldsymbol{W}$ is *sparse* (see Fig. 35.8), $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$ is also sparse and thus eigenvalue problem (36.2) can be solved efficiently even if n is large.

A MATLAB code of the Laplacian eigenmap for 10-nearest neighbor similarity (see Fig. 35.8) is provided in Fig. 36.5, and its behavior is illustrated in Fig. 36.6. This shows that the "swiss roll" structure can be nicely unfolded.

```
n=1000; k=10; a=3*pi*rand(n,1);
x=[a.*cos(a) 30*rand(n,1) a.*sin(a)];
x=x-repmat(mean(x),[n,1]); x2=sum(x.^2,2);
d=repmat(x2,1,n)+repmat(x2',n,1)-2*x*x'; [p,i]=sort(d);
W=sparse(d<=ones(n,1)*p(k+1,:)); W=(W+W'~=0);
D=diag(sum(W,2)); L=D-W; [z,v]=eigs(L,D,3,'sm');
figure(1); clf; hold on; view([15 10]);
scatter3(x(:,1),x(:,2),x(:,3),40,a,'o');
figure(2); clf; hold on; scatter(z(:,2),z(:,1),40,a,'o');
```

FIGURE 36.5

MATLAB code of Laplacian eigenmap for 10-nearest neighbor similarity.



(a) Original three-dimensional samples $\{\boldsymbol{x}_i\}_{i=1}^n$ (b) Embedded two-dimensional samples $\{\boldsymbol{z}_i\}_{i=1}^n$

FIGURE 36.6

Example of Laplacian eigenmap for 10-nearest neighbor similarity.

36.2 SUPERVISED DIMENSIONALITY REDUCTION WITH NEURAL NETWORKS

A *neural network* is a nonlinear model having hierarchical structure. If the number of layers is three (i.e. input, hidden, and output layers), its function is expressed as follows (see Section 21.3):

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

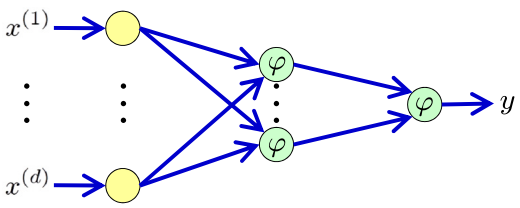


FIGURE 36.7

Dimensionality reduction by neural network. The number of hidden nodes is smaller than the number of input (and output) nodes.

where $\phi(\boldsymbol{x};\boldsymbol{\beta})$ is a basis function parameterized by $\boldsymbol{\beta}$. As explained in Section 22.5, LS training of neural networks based on input-output paired training samples $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ can be carried out by the gradient method called the *error back-propagation* algorithm.

If the number of hidden units, b, is set to be less than the dimensionality of input $\boldsymbol{x}$, such a neural network can be used for supervised dimensionality reduction. More specifically, the output of the hidden layer,

$$\boldsymbol{z} = (\phi(\boldsymbol{x};\boldsymbol{\beta}_1),\ldots,\phi(\boldsymbol{x};\boldsymbol{\beta}_b))^\top,$$

can be regarded as a dimensionality reduction solution (Fig. 36.7).

36.3 UNSUPERVISED DIMENSIONALITY REDUCTION WITH AUTOENCODER

In this section, the back-propagation approach is applied to unsupervised dimensionality reduction.

36.3.1 AUTOENCODER

The *autoencoder* [14, 116] is a three-layer neural network given by

$$\boldsymbol{x} = \boldsymbol{\varphi}\left(\boldsymbol{W}^\top \boldsymbol{\varphi}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}) + \boldsymbol{b}\right),$$

where $\boldsymbol{W}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are parameters. As illustrated in Fig. 36.8, the autoencoder has a bottleneck layer, and the connection weights between the first and second layers and the connection weights between the second and third layers are shared, i.e. $\boldsymbol{W}$ and $\boldsymbol{W}^\top$ are used. The connection weights are learned so that output is as close to input as possible, by which a compressed expression of input can be obtained in the bottleneck layer. Let us use the *sigmoidal activation function* (see Fig. 21.6) for the

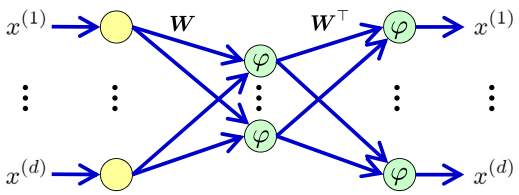


FIGURE 36.8

Autoencoder. Input and output are the same and the number of hidden nodes is smaller than the number of input nodes.

second and third layers:

$$\varphi(\boldsymbol{x}) = \frac{1}{1 + \exp\left(-\boldsymbol{x}^\top \boldsymbol{w} - a\right)}.$$

36.3.2 TRAINING BY GRADIENT DESCENT

For d-dimensional input $\boldsymbol{x} = (x^{(1)}, \ldots, x^{(d)})^\top$ and d-dimensional output

$$\boldsymbol{y} = (y^{(1)}, \ldots, y^{(d)})^\top = \boldsymbol{\varphi}\left(\boldsymbol{W}^\top \boldsymbol{\varphi}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}) + \boldsymbol{b}\right),$$

the parameters $\boldsymbol{W}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are learned by the *stochastic gradient* algorithm introduced in Section 15.3:

$$\begin{aligned}
\boldsymbol{W} &\longleftarrow \boldsymbol{W} - \varepsilon \nabla_{\boldsymbol{W}} J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}), \\
\boldsymbol{b} &\longleftarrow \boldsymbol{b} - \varepsilon \nabla_{\boldsymbol{b}} J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}), \\
\boldsymbol{c} &\longleftarrow \boldsymbol{c} - \varepsilon \nabla_{\boldsymbol{c}} J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}),
\end{aligned}$$

where the *squared loss*,

$$J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) = \frac{1}{2} \sum_{k=1}^{d} \left(x^{(k)} - y^{(k)}\right)^2, \tag{36.3}$$

may be used for regression (this corresponds to the log-likelihood of Gaussian distributions), and the *cross entropy loss*,

$$J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) = -\sum_{k=1}^{d} \left(x^{(k)} \log y^{(k)} + (1 - x^{(k)}) \log(1 - y^{(k)})\right), \tag{36.4}$$

may be used for binary classification (this corresponds to the log-likelihood of binomial distributions).

For

$$\boldsymbol{y} = \boldsymbol{\varphi}(\boldsymbol{h}), \quad \boldsymbol{h} = \boldsymbol{W}^\top \boldsymbol{z} + \boldsymbol{b}, \quad \boldsymbol{z} = \boldsymbol{\varphi}(\boldsymbol{g}), \quad \text{and} \quad \boldsymbol{g} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}, \tag{36.5}$$

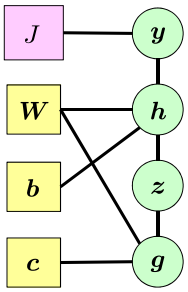


FIGURE 36.9

Chain rule for autoencoder.

the *chain rule* of the derivative (Fig. 36.9) yields

$$\frac{\partial J}{\partial W_{j,k}} = \frac{\partial J}{\partial y^{(k)}}\frac{\partial y^{(k)}}{\partial h_k}\frac{\partial h_k}{\partial W_{j,k}} + \frac{\partial J}{\partial y^{(k)}}\frac{\partial y^{(k)}}{\partial h_k}\frac{\partial h_k}{\partial z^{(j)}}\frac{\partial z^{(j)}}{\partial g_j}\frac{\partial g_j}{\partial W_{j,k}},$$
$$\frac{\partial J}{\partial b_k} = \frac{\partial J}{\partial y^{(k)}}\frac{\partial y^{(k)}}{\partial h_k}\frac{\partial h_k}{\partial b_k},$$
$$\frac{\partial J}{\partial c_j} = \sum_{k=1}^{d}\frac{\partial J}{\partial y^{(k)}}\frac{\partial y^{(k)}}{\partial h_k}\frac{\partial h_k}{\partial z^{(j)}}\frac{\partial z^{(j)}}{\partial g_j}\frac{\partial g_j}{\partial c_j}.$$

Then the gradients are given by

$$\nabla_{\boldsymbol{W}} J = \boldsymbol{z}(\nabla_{\boldsymbol{b}} J)^\top + (\nabla_{\boldsymbol{c}} J)\boldsymbol{x}^\top,$$
$$\nabla_{\boldsymbol{b}} J = \begin{cases} (\boldsymbol{y}-\boldsymbol{x}) * \boldsymbol{y} * (\boldsymbol{1}-\boldsymbol{y}) & \text{(squared loss)}, \\ \boldsymbol{y}-\boldsymbol{x} & \text{(cross entropy loss)}, \end{cases}$$
$$\nabla_{\boldsymbol{c}} J = \boldsymbol{W}\nabla_{\boldsymbol{b}} J * \boldsymbol{z} * (1-\boldsymbol{z}),$$

where "$*$" denotes the elementwise product.

A MATLAB code for dimensionality reduction by the autoencoder is provided in Fig. 36.10, and its behavior is illustrated in Fig. 36.11. Here, the autoencoder is trained with 100 noiseless images of hand-written digit "2" (see Section 12.5 for the details of the hand-written digit data set). Then a test image illustrated in Fig. 36.11(b), which is not included in the training data set, is input to the trained autoencoder. Since the autoencoder is trained only with noiseless images, noise components may be eliminated in the second layer. Then a denoised image illustrated in Fig. 36.11(c) is obtained as output, which looks much better than the output of linear PCA illustrated in Fig. 36.11(d).

```
load digit.mat; x=X(:,1:100,2); [d,n]=size(x); g=min(x,[],2);
x=(x-repmat(g,1,n))./repmat(max(x,[],2)-g,1,n); x=(x>0.5);
m=10; e=0.01/n; W0=randn(m,d); b0=randn(d,1); c0=rand(m,1);
for o=1:100000
  Z=1./(1+exp(-W0*x-repmat(c0,[1 n])));
  Y=1./(1+exp(-W0'*Z-repmat(b0,[1 n])));
  nb=Y-x; % Cross entropy loss
%  nb=(Y-x).*Y.*(1-Y); % Squared loss
  nc=(W0*nb).*Z.*(1-Z); W=W0-e*(Z*nb'+nc*x');
  b=b0-e*sum(nb,2); c=c0-e*sum(nc,2);
  if norm(W-W0)+norm(b-b0)+norm(c-c0)<0.003, break, end
  W0=W; b0=b; c0=c;
end
t=T(:,1,2)>0.5; u=t; u(rand(d,1)>0.9)=1; u(rand(d,1)>0.9)=0;
z=1./(1+exp(-W*t-c)); y=1./(1+exp(-W'*z-b));

figure(1); clf; hold on; colormap gray
subplot(1,3,1); imagesc(reshape(t,[16 16])')
subplot(1,3,2); imagesc(reshape(u,[16 16])')
subplot(1,3,3); imagesc(reshape(y>0.5,[16 16])')
```

FIGURE 36.10

MATLAB code for denoising autoencoder. See Section 12.5 for details of hand-written digit data set "`digit.mat`."

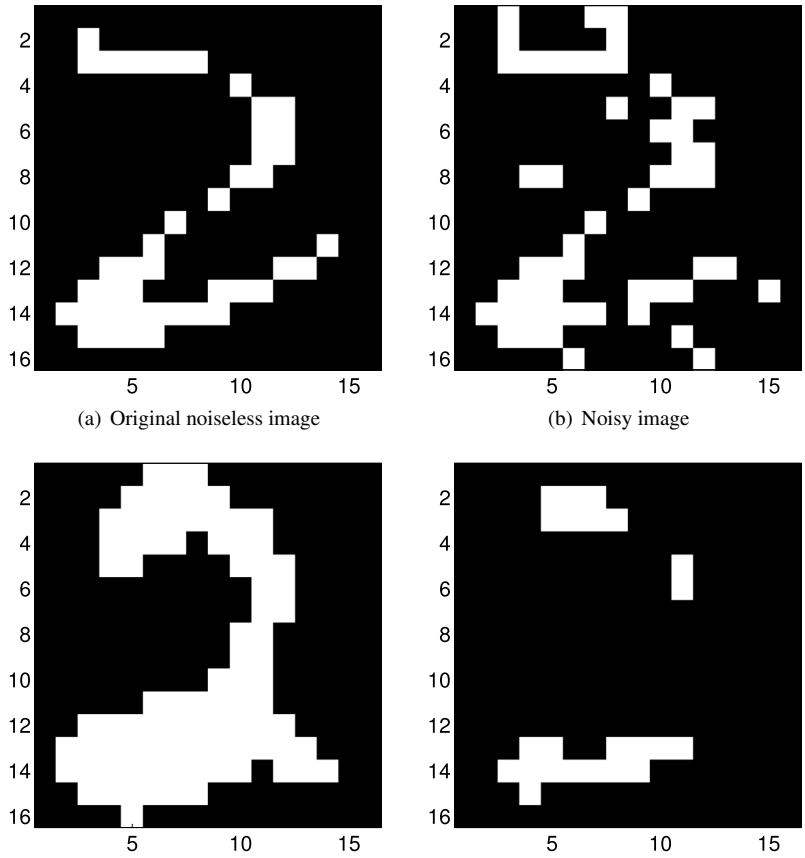
36.3.3 SPARSE AUTOENCODER

Let us consider a nonbottleneck neural network, i.e. the number of hidden nodes, m, is larger than the number of input (and output) nodes, d. If this architecture is used as an autoencoder, just the identity mapping may be learned as a trivial solution. However, if the activations in the hidden layer, $\boldsymbol{z} = (z^{(1)}, \ldots, z^{(m)})$ (see Eq. (36.5)), are enforced to be *sparse*, meaningful features may be extracted.

More specifically, in addition to the loss function $J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c})$ (see Eqs (36.3) and (36.4)), a sparsity-inducing regularization term is minimized at the same time:

$$\min_{\boldsymbol{W},\boldsymbol{b},\boldsymbol{c}} J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) + \lambda \sum_{j=1}^{m} R(z^{(j)}),$$

where $\lambda > 0$ is the regularization parameter and $R(z)$ is the regularization functional, e.g. the ℓ_1-norm $R(z) = |z|$ (see Chapter 24), or the KL divergence (see Section 14.2)



(a) Original noiseless image

(b) Noisy image

(c) Denoised image by autoencoder (10 hidden nodes)

(d) Denoised image by linear PCA (reduced to 10-dimensional subspace)

FIGURE 36.11

Example of denoising autoencoder.

from a constant $\rho \in (0, 1)$:

$$R(z) = \rho \log \frac{\rho}{z/m} + (1 - \rho) \log \frac{(1 - \rho)}{1 - z/m}.$$

36.4 UNSUPERVISED DIMENSIONALITY REDUCTION WITH RESTRICTED BOLTZMANN MACHINE

Although the autoencoder is an unsupervised neural network, it is trained in a supervised way by regarding output as input. In this section, another unsupervised

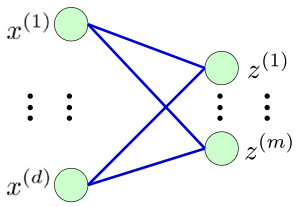


FIGURE 36.12

Restricted Boltzmann machine.

neural network called the *restricted Boltzmann machine* [55, 96] is introduced, which is trained in an unsupervised way.

36.4.1 MODEL

The restricted Boltzmann machine is a two-layered neural network illustrated in Fig. 36.12, where the left layer for d-dimensional input $\boldsymbol{x} = (x^{(1)}, \ldots, x^{(d)})^\top$ is called the *visible layer* and the right layer for m-dimensional hidden variable

$$\boldsymbol{z} = (z^{(1)}, \ldots, z^{(m)})^\top \in \{0,1\}^m$$

is called the *hidden layer*. Fig. 36.12 shows that any unit in the input layer and any unit in the hidden layer are completely connected. On the other hand, units in the input layer are not connected to each other and units in the hidden layer are not connected to each other. For the moment, let us assume that input is binary: $\boldsymbol{x} \in \{0,1\}^d$.

The restricted Boltzmann machine is a model of joint probability $q(\boldsymbol{x}, \boldsymbol{z})$ given by

$$q(\boldsymbol{x}, \boldsymbol{z}) = \frac{e^{-E(\boldsymbol{x}, \boldsymbol{z})}}{\displaystyle\sum_{\boldsymbol{x}' \in \{0,1\}^d} \sum_{\boldsymbol{z}' \in \{0,1\}^m} e^{-E(\boldsymbol{x}', \boldsymbol{z}')}}.$$

Here, $E(\boldsymbol{x}, \boldsymbol{z})$ is called the *energy function* defined as

$$E(\boldsymbol{x}, \boldsymbol{z}) = -\boldsymbol{b}^\top \boldsymbol{x} - \boldsymbol{c}^\top \boldsymbol{z} - \boldsymbol{z}^\top \boldsymbol{W} \boldsymbol{x},$$

where $\boldsymbol{W}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are parameters. Because there is no connection between the input and hidden units, the conditional probabilities can be factorized as

$$q(\boldsymbol{z}|\boldsymbol{x}) = \prod_{j=1}^{m} q(z^{(j)}|\boldsymbol{x}) \quad \text{and} \quad q(\boldsymbol{x}|\boldsymbol{z}) = \prod_{k=1}^{d} q(x^{(k)}|\boldsymbol{z}), \tag{36.6}$$

where each conditional probability is given by the *sigmoidal function*:

$$q(z^{(j)} = 1|\boldsymbol{x}) = \frac{1}{1 + \exp(-\sum_{k=1}^{d} W_{j,k} x^{(k)} - c_j)},$$
$$q(x^{(k)} = 1|z) = \frac{1}{1 + \exp(-\sum_{j=1}^{m} W_{j,k} z^{(j)} - b_k)}.$$

36.4.2 TRAINING BY GRADIENT ASCENT

For training the restricted Boltzmann machine, let us employ MLE (see Chapter 12). However, since hidden variable z is not observable, MLE cannot be performed directly. Here, let us consider the *marginal* model,

$$q(\boldsymbol{x}) = \sum_{z \in \{0,1\}^m} q(\boldsymbol{x}, z),$$

and perform MLE for this model:

$$\max_{\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}} L(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}),$$

where

$$L(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) = \frac{1}{n} \sum_{i=1}^{n} \log \left(\sum_{z \in \{0,1\}^m} q(\boldsymbol{x}_i, z) \right).$$

The gradient of the log-likelihood is given by

$$\frac{\partial L}{\partial \theta} = -\frac{1}{n} \sum_{i=1}^{n} \sum_{z \in \{0,1\}^m} \frac{\partial E}{\partial \theta} q(z|\boldsymbol{x} = \boldsymbol{x}_i) + \sum_{\boldsymbol{x} \in \{0,1\}^d} \sum_{z \in \{0,1\}^m} \frac{\partial E}{\partial \theta} q(\boldsymbol{x}, z), \tag{36.7}$$

where θ represents either $\boldsymbol{W}$, $\boldsymbol{b}$, or $\boldsymbol{c}$, and $\frac{\partial E}{\partial \theta}$ is given by

$$\frac{\partial E}{\partial \boldsymbol{W}} = -z^\top \boldsymbol{x}, \quad \frac{\partial E}{\partial \boldsymbol{b}} = -\boldsymbol{x}, \quad \text{and} \quad \frac{\partial E}{\partial \boldsymbol{c}} = -z.$$

Thanks to factorization (36.6), the first term in Eq. (36.7) can be computed naively as

$$-\frac{1}{n} \sum_{i=1}^{n} \sum_{z \in \{0,1\}^m} \frac{\partial E}{\partial \theta} q(z|\boldsymbol{x} = \boldsymbol{x}_i) = -\frac{1}{n} \sum_{i=1}^{n} \frac{\partial E}{\partial \theta} \prod_{j=1}^{m} \sum_{z^{(j)} \in \{0,1\}} q(z^{(j)}|\boldsymbol{x} = \boldsymbol{x}_i).$$

On the other hand, the second term in Eq. (36.7) can be approximated by the empirical distribution of samples $\{\boldsymbol{x}_i\}_{i=1}^n$ as

$$\begin{aligned} \sum_{\boldsymbol{x} \in \{0,1\}^d} \sum_{z \in \{0,1\}^m} \frac{\partial E}{\partial \theta} q(\boldsymbol{x}, z) &= \sum_{\boldsymbol{x} \in \{0,1\}^d} \sum_{z \in \{0,1\}^m} \frac{\partial E}{\partial \theta} q(z|\boldsymbol{x}) p(\boldsymbol{x}) \\ &\approx \frac{1}{n} \sum_{i=1}^{n} \sum_{z \in \{0,1\}^m} \frac{\partial E}{\partial \theta} q(z|\boldsymbol{x}_i). \end{aligned}$$

1. Let $\widehat{\boldsymbol{x}}_i = \boldsymbol{x}_i$ for $i = 1, \ldots, n$.
2. Generate $\{\widehat{z}_i\}_{i=1}^n$ from $\{\widehat{\boldsymbol{x}}_i\}_{i=1}^n$ following $q(z|\boldsymbol{x} = \widehat{\boldsymbol{x}}_i)$.
3. Generate $\{\widehat{\boldsymbol{x}}_i\}_{i=1}^n$ from $\{\widehat{z}_i\}_{i=1}^n$ following $q(\boldsymbol{x}|z = \widehat{z}_i)$.
4. Iterate 2–3 until convergence.

FIGURE 36.13

Contrastive divergence algorithm for restricted Boltzmann machine. Note that $q(z|\boldsymbol{x} = \widehat{\boldsymbol{x}}_i)$ and $q(\boldsymbol{x}|z = \widehat{z}_i)$ can be factorized as Eq. (36.6), which allows efficient computation.

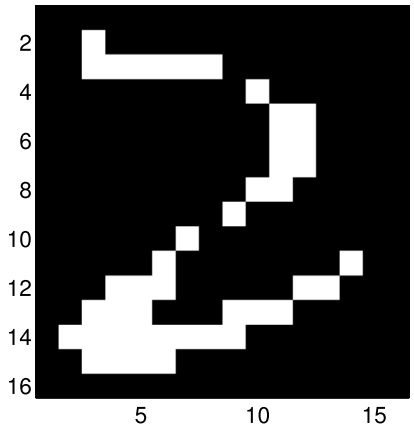
```
load digit.mat; x=X(:,1:100,2); [d,n]=size(x); g=min(x,[],2);
x=(x-repmat(g,1,n))./repmat(max(x,[],2)-g,1,n); x=(x>0.5);
m=10; e=0.01/n; W0=randn(m,d); b0=randn(d,1); c0=rand(m,1);
for o=1:100000
  pZ=1./(1+exp(-W0*x-repmat(c0,1,n))); Z=1*(rand(m,n)<pZ);
  pY=1./(1+exp(-W0'*Z-repmat(b0,1,n))); Y=1*(rand(d,n)<pY);
  pX=1./(1+exp(-W0*Y-repmat(c0,1,n))); W=W0+e*(pZ*x'-pX*Y');
  b=b0+e*(x*prod(pZ)'-Y*prod(pX)');
  c=c0+e*(sum(pZ,2)-sum(pX,2));
  if norm(W-W0)+norm(b-b0)+norm(c-c0)<0.007, break, end
  W0=W; b0=b; c0=c;
end
t=T(:,1,2)>0.5; u=t; u(rand(d,1)>0.9)=1; u(rand(d,1)>0.9)=0;
z=1./(1+exp(-W*t-c)); y=1./(1+exp(-W'*z-b));

figure(1); clf; hold on; colormap gray
subplot(1,3,1); imagesc(reshape(t,[16 16])')
subplot(1,3,2); imagesc(reshape(u,[16 16])')
subplot(1,3,3); imagesc(reshape(y>0.5,[16 16])')
```

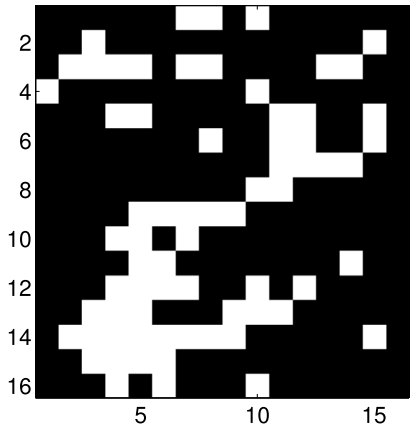
FIGURE 36.14

MATLAB code for denoising restricted Boltzmann machine. See Section 12.5 for details of handwritten digit data set "`digit.mat`."

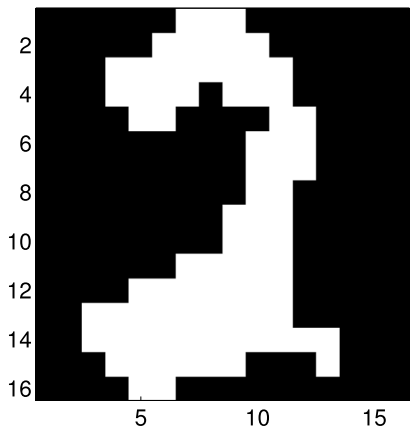
However, since this is exactly the same as the negative of the first term, the gradient approximately computed in this way always vanishes and thus is useless in practice. To cope with this problem, let us consider another set of samples $\{\widehat{\boldsymbol{x}}_i\}_{i=1}^n$ generated by the *contrastive divergence algorithm* [54], described in Fig. 36.13, which is based on *Gibbs sampling* (see Section 19.3.3).



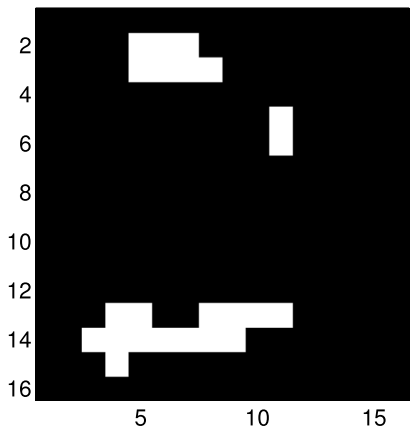
(a) Original noiseless image



(b) Noisy image



(c) Denoised image by restricted Boltzmann machine (10 hidden nodes)



(d) Denoised image by linear PCA (reduced to 10-dimensional subspace)

FIGURE 36.15

Example of denoising restricted Boltzmann machine.

Finally, the gradient of the log-likelihood is approximated as

$$\frac{\partial L}{\partial \theta} \approx \frac{1}{n} \sum_{i=1}^{n} \frac{\partial E}{\partial \theta} \prod_{j=1}^{m} \sum_{z^{(j)} \in \{0,1\}} \Big(- q(z^{(j)} | \boldsymbol{x} = \boldsymbol{x}_i) + q(z^{(j)} | \boldsymbol{x} = \widehat{\boldsymbol{x}}_i) \Big).$$

Based on this approximated gradient, the parameters $\boldsymbol{W}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ can be learned by *gradient ascent* (see Section 15.3).

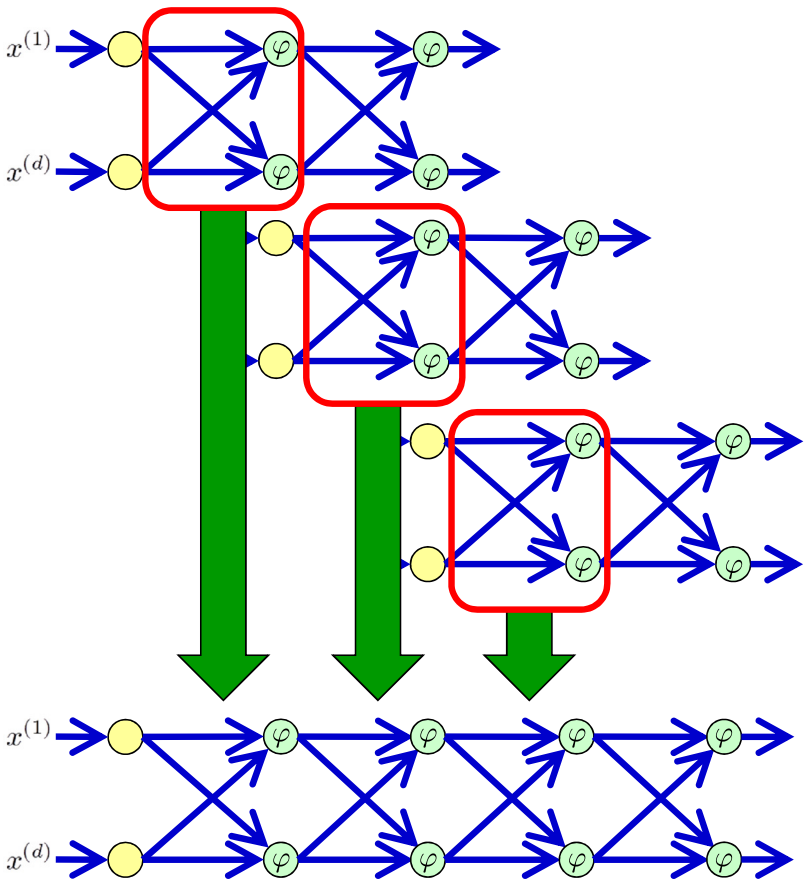


FIGURE 36.16

Construction of deep neural network by stacking.

When input $\boldsymbol{x}$ is continuous, the energy function may be replaced with

$$E(\boldsymbol{x},\boldsymbol{z}) = \frac{1}{2}\boldsymbol{x}^\top\boldsymbol{x} - \boldsymbol{b}^\top\boldsymbol{x} - \boldsymbol{c}^\top\boldsymbol{z} - \boldsymbol{z}^\top\boldsymbol{W}\boldsymbol{x},$$

which results in the Gaussian conditional probability:

$$q(x^{(k)} = 1|\boldsymbol{z}) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(x^{(k)} - b_k - \sum_{j=1}^{m} z^{(j)}W_{j,k}\right)^2\right).$$

A MATLAB code for dimensionality reduction by the restricted Boltzmann machine is provided in Fig. 36.14, and its behavior is illustrated in Fig. 36.15. This is

the same denoising experiment as Fig. 36.11, and the restricted Boltzmann machine also works much better than linear PCA.

36.5 DEEP LEARNING

In Section 22.5, the *error back-propagation* algorithm for supervised training of neural networks was introduced. As pointed out there, due to the hierarchical structure of neural networks, gradient-based training is prone to be trapped by a local optimal solution.

It was experimentally demonstrated that training autoencoders or restricted Boltzmann machines layer by layer by unsupervised learning and stacking them (see Fig. 36.16) can produce good initialization of connection weights [55, 86]. In other words, such deep autoencoders and deep Boltzmann machines can be regarded as good feature extractors for succeeding supervised learning tasks.

Extensive research on deep learning is on going and latest information can be found, e.g. from "`http://deeplearning.net/`."