5 Unsupervised Learning

As already stated, the aim of learning is to find a mapping function $y = f(\mathbf{x})$ or probability density function $p(y|\mathbf{x})$. An important insight that we explore in this sections is that finding such relations is much easier if the representation of the feature vector is chosen carefully. For example, it is actually very challenging to use raw pixel values to infer the content of a digital photo such as the recognition of a face. In contrast, if we have given a useful descriptions of faces, such as the distance between eyes and other landmark features, the colour of hair, and the length of the nose etc, it is much easier to classify photographs to specific target faces. Finding a useful representation of a problem is often key for successful applications. When we use learning techniques for this task we talk about **representational learning**. Representational learning is usually exploiting statistical characteristics of the environment without the need of labeled training examples. This is therefore an important area of **unsupervised learning**.

We will start our discussion of unsupervised learning with the example of **data clustering**, specifically **k-means clustering** and the more general **Expectation Maximization**. While these models are generally applied to very specific data model, we are discussing again more general learning machines in the second half of this chapter, which includes some thoughts on basic **Signal Processing** and finally **Restricted Boltzmann Machines**.

5.1 k-means clustering

We have discussed supervised classification in which we are given traing examples of feature values and class labels and our task was to classify new data that have no labels. We discussed SVMs as discriminative method to achieve this. Alternatively, we could have use an generative model where we model each class distribution $p(\mathbf{y}|\mathbf{x})$ separately and then apply Bayes to do the discrimination.

In the following we discuss the situation where we are given unlabelled data described by a set of feature values and asked to put them into k categorize. In the first example of such clustering we categories the data by proximity to a mean value. That is, we assume a model that specifies a mean feature value of the data and classifies the data based on the proximity to the mean value. Of course, we do not know this mean value for each class. The idea of the following algorithm is that we start with a guess for this mean value and label the data accordingly. We then use the labeled data from this hypothesis to improve the model by calculating a new mean value, and repeat these steps until convergence is reached. Such an algorithm usually converges quickly to a stable solution. More formally, given a training set of data points $\{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$ and a hypothesis of the number of clusters, k, the k-means clustering algorithm is

Table 5.1 k-means clustering algorithm

```
1. Initialize the means $\mu_1, ...\mu_k$ randomly.
2. Repeat until convergence: {
        **Model prediction:**
            For each data point $i$, classify data to class with closest mean
                $c^{(i)} = arg\min_j ||x^{(i)} - \mu_j||$
        **Model refinement:**
            Calculate new means for each class
                $\mu_j = \frac{1\ \ 1(c^{(i)}=j)x^{(i)}}{1\ \ 1(c^{(i)}=j)}$
} convergence
```

shown in Table 5.1. An example is shown in Figure **??** and the corresponding program is shown is Table **??**.

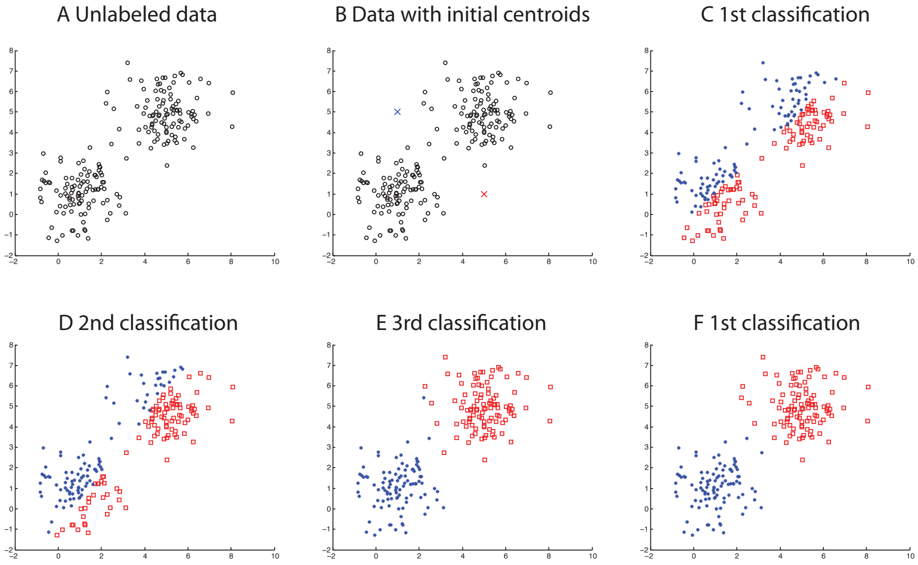

Fig. 5.1 Example of k-means clustering with two clusters.

5.2 Mixture of Gaussian and the EM algorithm

We have previously discussed generative models where we assumed specific models for the in-class distributions. In particular, we have discussed linear discriminant analysis where we had labelled data and assumed that each class is Gaussian distributed. Here we assume that we have k Gaussian classes, where each class is chosen randomly from a multinominal distribution,

$$p(z^{(i)} = j) \propto \text{multinomial}(\Phi_j) \tag{5.1}$$

Table 5.2 Program to demonstrate k-mean clustering on Gaussian Data.

```
clear; clf; hold on;

%% training data generation; 2 classes, each gaussian with mean (1,1) and (2,2) and diagonal
n0=100; %number of points in class 0
n1=100; %number of points in class 1

x=[1+randn(n0,1), 1+randn(n0,1); ...
   5+randn(n1,1), 5+randn(n1,1)];

plot(x(:,1),x(:,2),'ko'); % plotting points
mu1=[5 1]; mu2=[1 5]; % initial two centers
while(true) waitforbuttonpress;

plot(mu1(1),mu1(2),'rx','MarkerSize',12)
plot(mu2(1),mu2(2),'bx','MarkerSize',12)

for i=1:n0+n1;
    d1=(x(i,1)-mu1(1))^2+(x(i,2)-mu1(2))^2;
    d2=(x(i,1)-mu2(1))^2+(x(i,2)-mu2(2))^2;
    y(i)=(d1<d2)*1;
end

waitforbuttonpress;
x1=x(y>0.5,:);
x2=x(y<0.5,:);

clf; hold on;
plot(x1(:,1),x1(:,2),'rs');
plot(x2(:,1),x2(:,2),'b*');
mu1=mean(x1);
mu2=mean(x2);

end
```

$$p(x^{(i)}|z^{(i)}=j) \propto N(\mu_j,\Sigma_j) \tag{5.2}$$

This is called a **Gaussian Mixture Model**. The corresponding log-likelihood function is

$$l(\Phi,\mu,\sigma)=\sum_{i=1}^{m}\log\sum_{z^{(i)}=1}^{k}p(x^{(i)}|z^{(i)};\mu,\Sigma)p(z^{(i)};\Phi). \tag{5.3}$$

Since we consider here unsupervised learning in which we are given data without labels, the random variables $z^{(i)}$ are latent variables. This makes the problem hard. If we would be give the class membership, than the log-likelihood would be

$$l(\Phi, \mu, \sigma) = \sum_{i=1}^{m} \log p(x^{(i)}; z^{(i)}, \mu, \Sigma), \tag{5.4}$$

which we could use to calculate the maximum likelihood estimates of the parameter (see equations **??-??**),

$$\phi_k = \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j) \tag{5.5}$$

$$\mu_k = \frac{\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j)\mathbf{x}^{(i)}}{\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j)} \tag{5.6}$$

$$\Sigma_k = \frac{\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j)(x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} \mathbb{1}(y^{(i)} = k)}. \tag{5.7}$$

While we do not know the class labels, we can follow a similar strategy to the k-means clustering algorithm and just propose some labels and use them to estimate the parameters. We can then use the new estimate of the distributions to find better labels for the data, and repeat this procedure until a stable configuration is reached. In general, this strategy is called the **EM algorithm** for expectation-maximization. The algorithm is outlined in Fig.5.2. In this version we do not hard classify the data into one or another class, but we take a more soft classification approach that considers the probability estimate of a data point belonging to each class.

```
1. Initialize parameters $\phi, \mu, \Sigma$ randomly.
2. Repeat until convergence: {
        E step:
            For each data point $i$ and class $j$ (soft-)classify data as
                $w_j^{(i)} = p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma)$
        M step:
            Update the parameters according to
                $\phi_j = \frac{1}{m}\sum_{i=1}^{m} w_j^{(i)}$
                $\mu_j = \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}}$
                $\Sigma_k = \frac{\sum_{i=1}^{m} w_j^{(i)}(x^{(i)}-\mu_j)(x^{(i)}-\mu_j)^T}{\sum_{i=1}^{m} \mathbb{1}w_j^{(i)}}.$
} convergence
```

Fig. 5.2 EM algorithm

An example is shown in Fig. 5.3. In this simple world, data are generated with equal likelihood from two Gaussian distributions, one with mean $\mu_1 = -1$ and standard deviation $\sigma_1 = 2$, the other with mean $\mu_2 = 4$ and standard deviation $\sigma_2 = 0.5$. These two distributions are illustrated in Fig. 5.3A with dashed lines. Let us assume that we know that the world consists only of data from two Gaussian distributions with equal likelihood, but that we do not know the specific realizations (parameters) of these distributions. The pre-knowledge of two Gaussian distributions encodes a specific **hypothesis** which makes up this **heuristic model**. In this simple example, we have chosen the heuristics to match the actual data-generating system (world), that is, we have explicitly used some knowledge of the world.

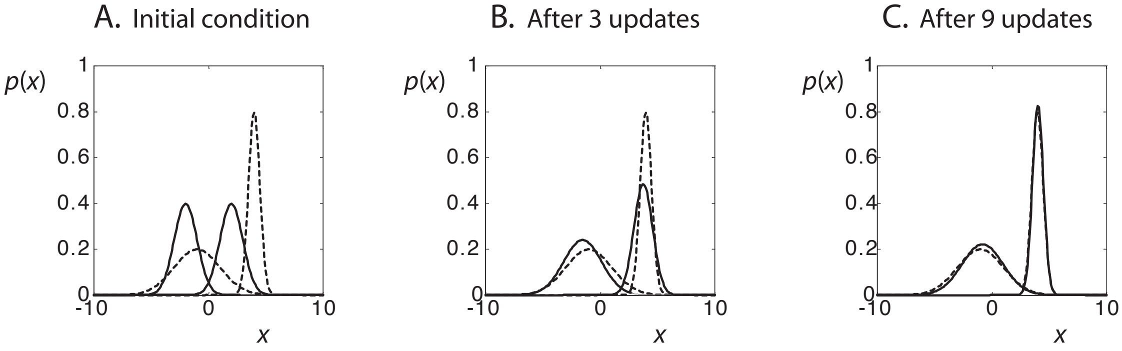

Fig. 5.3 Example of the expectation maximization (EM) algorithm for a world model with two Gaussian distributions. The Gaussian distributions of the world data (input data) are shown with dashed lines. (A) The generative model, shown with solid lines, is initialized with arbitrary parameters. In the EM algorithm, the unlabelled input data are labelled with a recognition model, which is, in this example, the inverse of the generative model. These labelled data are then used for parameter estimation of the generative model. The results of learning are shown in (B) after three iterations, and in (C) after nine iterations .

Learning the parameters of the two Gaussians would be easy if we had access to the information about which data point was produced by which Gaussian, that is, which cause produced the specific examples. Unfortunately, we can only observe the data without a teacher label that could supervise the learning. We choose therefore a self-supervised strategy, which repeats the following two steps until convergence:

E-step: We make assumptions of training labels from the current model (expectation step)

M-step: use this hypothesis to update the parameters of the model to maximize the probability of the observations (maximization step).

Since we do not know appropriate parameters yet, we just choose some arbitrary values as the starting point. In the example shown in Fig. 5.3A we used $\mu_1 = 2$, $\mu_2 = -2$, $\sigma_1 = \sigma_2 = 1$. These distributions are shown with solid lines. Comparing the generated data with the environmental data corresponds to hypothesis testing.

The results are not yet very satisfactory, but we can use the generative model to express our **expectation** of the data. Specifically, we can assign each data point to the class which produces the larger probability within the current world model. Thus, we are using our specific hypothesis here as a **recognition model**. In the example we can use Bayes' rule to invert the generative model into a recognition model as detailed in the simulation section below. If this inversion is not possible, then we can introduce a separate recognition model, Q, to approximate the inverse of the generative model. Such a recognition model can be learned with similar methods and interleaved with the generative model.

Of course, the recognition with the recognition model early in learning is not expected to be exact, but estimation of new parameters from the recognized data in the M-step to maximize the expectation can be expected to be better than the model with the initial arbitrary values. The new model can then be compared to the data again and, when necessary, be used to generate new expectations from which the model is refined. This procedure is known as the **expectation maximization** (EM) algorithm. The distributions after three and nine such iterations, where we have chosen new data points in each iteration, are shown in Figs 5.3B and C.

Simulation

The program used to produce Fig. 5.3 is shown in Table 5.3. The vector x_0, defined in Line 2, is used to plot the distributions later in the program. The arbitrary random initial conditions of the distribution parameters are set in Line 3. Line 4 defines an **inline function** of a properly normalized Gaussian since this function is used several times in the program. An inline function is an alternative to writing a separate function file. It defines the name of the functions, followed by a list of parameters and an expression, as shown in Line 4. The rest of the program consist of an infinite loop produced with the statement `while 1`, which is always true. The program has thus to be interrupted by closing the figure window or with the interruption command `Ctrl C`. In Lines 7–12, we produce plots of the real-world models (dotted lines) and the model distributions (plotted with a red and a blue curve when running the program). The command `waitforbuttonpress` is used in Line 12 so that we can see the results after each iteration.

In Line 14 we produce new random data in each iteration. Recognition of this data is done in Line 16 by inverting the generative model using Bayes' formula,

$$P(c|\mathbf{x};G) = \frac{P(\mathbf{x}|c;G)P(c;G)}{P(\mathbf{x};G)}. \tag{5.8}$$

In this specific example, we know that the data are equally distributed from each Gaussian so that the **prior distribution over causes**, $P(c;G)$ is $1/2$ for each cause. Also, the **marginal distribution of data** is equally distributed, so that we can ignore this normalizing factor. The recognition model in Line 16 uses the Bayesian decision criterion, in which the data point is assigned to the cause with a larger **recognition distribution**, $P(c|\mathbf{x};G)$. Using the labels of the data generated by the recognition model, we can then use the data to obtain new estimates of the parameters for each Gaussian in Lines 17–21.

Note that when testing the system for a long time, it can happen that one of the distributions is dominating the recognition model so that only data from one distribution are generated. The model of one Gaussian would then be **explaining away** data from the other cause. More practical solutions must take such factors into account.

5.3 Dimensionality reduction

A large amount of data are collected these days. Much data is unlabelled and high dimensional in the many feature dimension are recorded for specific application areas. It is hence not surprising that a large number of methods were developed to simplify

Table 5.3 Program ExpectationMaximization.m

```
 %% 1d example EM algorithm
clear; hold on; x0=?10:0.1:10;
var1=1; var2=1; mu1=?2; mu2=2;
normal= @(x,mu,var) exp(?(x?mu).^2/(2?var))/sqrt(2?pi?var);
while 1
    %%plot distribution
    clf; hold on; ylim([0 1]);
    plot(x0, normal(x0,?1,4),k: );
    plot(x0,normal(x0,4 ,.25) , k: );
    plot(x0, normal(x0,mu1,var1),r);
    plot(x0, normal(x0,mu2,var2),b);
    waitforbuttonpress ;
    %% data
    x=[2?randn(50,1)?1;0.5?randn(50 ,1)+4;];
    %% recogintion
    p1=normal(x,mu1,var1);
    p2=normal(x,mu2,var2);
    nrm=p1+p2 ; p1=p1./nrm; p1=p1./sum(p1); p2=p2./nrm; p2=p2./sum(p2);
    %% maximization
    mu1 =x?p1; var1=(x?mu1).^2 ? p1;
    mu2 =x?p2; var2=(x?mu2).^2 ? p2;
    %equivalently :
    %p1=p1./nrm; p2=p2./nrm;
    %mu1=sum(x.?p1)/sum(p1); var1=sum(p1.?(x?mu1).^2)./sum(p1);
    %mu2=sum(x.?p2)/sum(p2); var2=sum(p2.?(x?mu2).^2) ./sum(p2);
end
```

or summarize the data. This is often done under the label **dimensionality reduction**. Typically methods include Eigenspace and PCA, ICA, factor analysis, nonlinear dimensionality reduction, spectral clustering, etc. There are some good collections of methods available, both in Matlab and Python. We will not discuss these above mentioned methods in detail, but the more general discussion below applies directly to this problem domain.

5.4 Representations and the restricted Boltzmann machine

Let us now return to the more general discussion of representational learning. To show how different representations can influence information processing, consider how to represent numbers. An easy method, likely used in ancient times, is to denote a certain number with as many lines as shown in Fig.5.4A. Adding two numbers is relatively easy, but representing large numbers is becoming cumbersome. The Romans used a different representations shown in Fig.5.4B in which larger numbers can be represented. But adding two large numbers is not easy. The Arabic in Fig.5.4C representations was a major breakthrough to calculate with larger numbers. Adding number in a binary

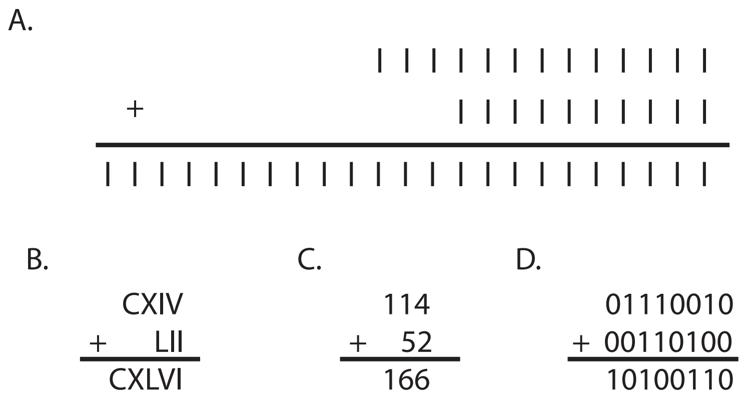

Fig. 5.4 Different representation of numbers and their sum.

representation, Fig.5.4D, is good to add numbers with logical gates that enabled the automation of adding numbers in a electronic calculator. A major power of quantum computing lies in the fact that complicated probabilistic operations could be performed easily from quantum effects.

Representational learning itself can be viewed as a mapping problem, such as the mapping from raw pixel values to more direct features of a face or the mapping from Roman number to digital numbers. This is illustrated in Fig.5.5A where the raw input feature vector, **x**, is represented by a layer of nodes at the bottom. Let's call this layer the **input layer**. The feature vector for higher order representations, **h**, is represented as nodes in the upper layer of this network. Let's call this the **representational layer** or **hidden layer**. The connections between the nodes represent the desired transformation between input layer and hidden layer. In line with our probabilistic framework, each node represents a random variable. The main idea of the principle that we will employ to find useful representations is that these representations should be useful in reconstructing the input.

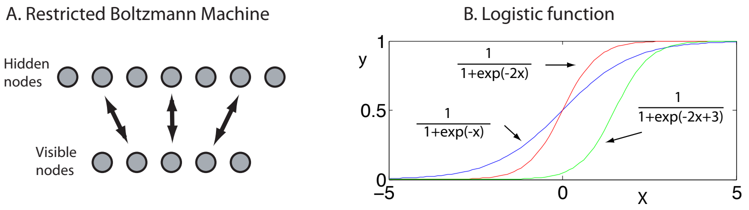

Fig. 5.5 (A) Restricted Boltzmann machine which is a probabilistic two layer network with bidirectional symmetric connections between the input layer and the representational (hidden) layer. (B) Logistic function with different slopes and offsets.

Before we discuss different variations of hidden representations, let us make the functions of the model more concrete. Specifically, let us consider mainly binary random variables for illustration purposes. Given the values of the inputs, we choose to calculate the value of the hidden nodes with index i, or more precisely the probability of having a certain value, with a logistic function shown in Fig.5.5B,

$$p(h_i = 1|\mathbf{x}) = \frac{1}{1 + e^{-\frac{1}{T}(\mathbf{w}_i\mathbf{x}+b_i)}}, \tag{5.9}$$

where T is a temperature parameter controlling the steepness of the curve, $\mathbf{w}$ are the weight values of the connections between input and hidden layer, and b_i^{h} is the offset of the logistic function, also called bias of the hidden node. In this model, which is called a restricted Boltzmann machine (Smolensky 1986), there are no connections between hidden nodes so that the hidden nodes represent random variables that are conditionally independent when the inputs are observed. In other words, the joint density function with fixed inputs factorizes,

$$p(\mathbf{h}|\mathbf{x}) = \prod_i \frac{1}{1 + e^{-\frac{1}{T}\sum_j w_{ij}x_j + b_i^{\mathrm{h}}}}. \tag{5.10}$$

The connections in this model are bidirectional, and such a model represents therefore a symmetric Bayesian network which is a special case of the Bayesian networks discussed later. The state of the input nodes can be generated by hidden activations according to

$$p(\mathbf{x}|\mathbf{h}) = \prod_i \frac{1}{1 + e^{-\frac{1}{T}\sum_j w_{ij}h_j + b_i^{\mathrm{v}}}}, \tag{5.11}$$

where b_i^{v} are the biases for each visible (input) node.

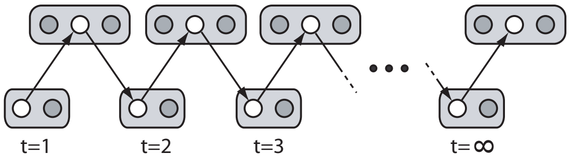

Fig. 5.6 Alternating Gibbs sampling.

The remaining question is how to choose parameters, specifically the weights and biases of the model? Since our aim is to reconstruct the world, we can formulate this in a probabilistic framework by minimizing the distance between the world distribution (the density function of visible nodes when set by unlabelled examples from the environment) and the generated model of the world when sampled from hidden activities. The difference between distributions is often measured with the Kullbach-Leibler divergence, and minimizing this objective function with a gradient method leads to a Hebbian-type learning rule

$$\Delta w_{ij} = \eta \frac{\partial l}{\partial w_{ij}} = \eta \frac{1}{2T}\left(\langle s_i s_j \rangle_{\text{clamped}} - \langle s_i s_j \rangle_{\text{free}}\right). \tag{5.12}$$

The angular brackets $\langle . \rangle$ denote sample averages, either in the clamped mode where the inputs are fixed or in the free running mode where the input nodes activity is determined

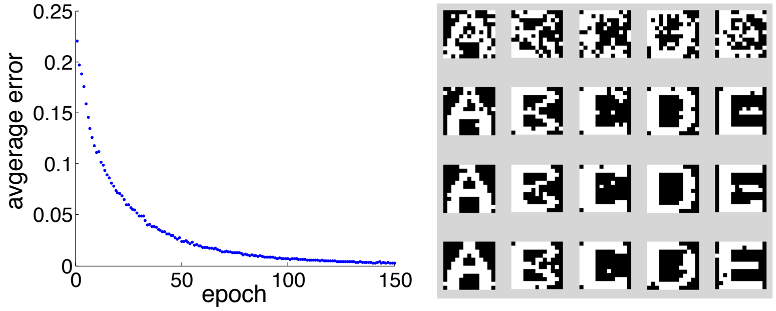

Fig. 5.7 Output of the example program for a restricted Boltzmann machine. The learning curve on the left shows the development of the average reconstruction error, and the reconstructions of noisy patterns after training are shown on the right.

by the hidden nodes. Unfortunately, in practice this learning rule suffers from the long time it takes to produce unbiased average from sequentially sampled time series. However, it turns out that learning still works for a few steps in Gibbs sampling as illustrated in Fig.5.12. This learning rule, which has finally made Boltzmann machines applicable, is called contrastive divergence (Hinton 2002).

An example of a basic restricted Boltzmann machine is given in Table 5.4. This network is used to learn digitized letters of the alphabet that are provided in file `pattern1.txt` at `www.cs.dal.ca/ tt/repository/MLintro2012` together with the other programs of this article. This RBM has $nh = 100$ hidden nodes and is trained for $nepochs = 150$ epoch, where one epoch consists of presenting all images once. The network is trained with contrastive divergence in the next block of code. The training curve, which shows the average error of recall of patterns, is shown on the left in Fig.5.7. After training, 20% of the bits of the training patterns are flipped and presented as input to the network, and the program then plots the patterns after repeated reconstructions as shown on the right in Fig.5.7. Only the first 5 letters are shown here, but this number can be increased to inspect more letters.

5.5 Sparse representations

In the previous section we reviewed a basic probabilistic network that implements representational learning based on reconstructions of inputs. There are many other unsupervised algorithms that can do representational learning and we will encounter some additional ones later. Also, many other representational learning algorithm are known from signal processing such as Fourier transformation, wavelet analysis, or independent component analysis (ICA). Indeed, most advanced signal processing include steps to re-represent or decompose a signal into basis functions. For example, the Fourier transformation decomposes a signal into sine waves with different amplitudes and phases. The individual sine waves can then be reconstructed from the coefficient for each of these basis functions. An example is shown in Fig.5.8. The signal in the upper left is made out of three sine waves as revealed by the power spectrum on the right that plots basically the square of the corresponding coefficients.

Table 5.4 Basic restricted Boltzmann machine to learn letter patterns

```
clear; nh=100;  nepochs=150;  lrate=0.01;
%load data from text file and rearrange into matrix
load pattern1.txt;
letters = permute( reshape( pattern1, [12 26 13]), [1 3 2]);

%%train rbm for nepochs presentations of the 26 letters
input = reshape(letters,[12*13 26])
vb =zeros(12*13,1);  hb =zeros(nh,1);  w =.1*randn(nh,12*13);

figure; hold on;
xlabel ’epoch’; ylabel ’error’; xlim([0 nepochs]);
for epoch=1:nepochs;
  err=0;
  for i=1:26
    %Sample hidden units given input, then reconstruct.
    v = input(:,i);
    h = 1./(1 + exp(-(w *v + hb))); %sigmoidal activation
    hs= h > rand(nh,1);             %probabilistic sampling
    vr= 1./(1 + exp(-(w’*hs+ vb))); %input reconstruction
    hr= 1./(1 + exp(-(w *vr+ hb))); %hidden reconstruction

    %Contrastive Divergence rule: dw ˜ h*v - hr*vr
    dw  = lrate*(h*v’-hr*vr’);  w = w +dw;
    dvb = lrate*( v  -  vr  );  vb= vb+dvb;
    dhb = lrate*( h  -  hr  );  hb= hb+dhb;
    err = err   + sum( (v-vr).^2 );   %reconstruction error
  end
  plot( epoch, err/(12*13*26), ’.’);  drawnow;%figure output
end

%%plot reconstructions of noisy letters
r = randomFlipMatrix(round(.2*12*13)); %(20% of bits flipped)
noisy_letters = abs(letters - reshape(r,[12 13 26]));
recon = reshape(noisy_letters, 12*13, 26); %put data in matrix
recon=recon(:,1:5);                        %only plot first 10
figure; set(gcf,’Position’,get(0,’screensize’));

for i=0:3
  for j=1:5
    subplot(3+1, 5, i*5 + j);
    imagesc( reshape(recon(:,j),[12 13]) );  %plot
    colormap gray; axis off; axis image;

    h = 1./(1 + exp(-(w *recon(:,j) + hb))); %compute hidden
    hs= h > rand(nh,1);                      %sample hidden
    recon(:,j) = 1./(1 + exp(-(w’*hs + vb)));%compute visible
    recon(:,j) = recon(:,j) > rand(12*13,1); %sample visible
  end
end

function r=randomFlipMatrix(n);
% returns matrix with components 1 at n random positions
r=zeros(156,26);
for i=1:26
    x=randperm(156);
    r(x(1:n),i)=1;
end
```

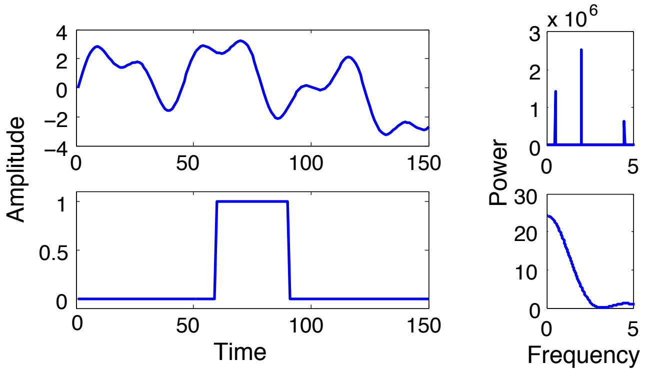

Fig. 5.8 Decomposition of signals into sine waves. The example signals are shown on the left side, and the corresponding description of the power spectrum on the right. The power spectrum shows the square of the amplitude for each contributing sine wave with specified frequency.

The Fourier transformation has been very useful in describing periodic signals, but one problem with this representation is that an infinite number of basis functions are needed to represent a signal that is localized in time. An example of a square signal localized in time is shown in the lower left panel of Fig.5.8 together with its power spectrum on the right. In the case of the time-localized signal, the power spectrum shows that a continuous number of frequencies are necessary to accurately represent the original signal. Thus, a better choice for applications with localized features would be basis functions that are localized in time. An example of such transformations are wavelet transforms (Graps 2012) or the Huang-Hilbert transform (Huang et al. 1998). The usefulness of a specific transformation depends of course on the nature of the signals. Periodic signals with few frequency components, such as the rhythm of the heart or yearly fluctuations of natural events, are well represented by Fourier transforms, while signals with localized features, such as objects in a visual scene, are often well represented with wavelets. The main reason for calling a representation useful is that the original signal can be represented with only a small number of basis functions, or with other words, when only a small number of coefficients have significant large values. Thus, even if the dictionary might be large, each example of a signal of the specific environment can be represented with a small number of components. Such representations are called **sparse**.

The major question is then how to find good (sparse) representations for specific environments. One solution within the learning domain is to learn representations by unsupervised learning as demonstrated above with the example of a Boltzmann machine. To learn sparse representations we now add additional constrains that force the learning of specific basis functions. In order to do this we can keep track of the mean activation of the hidden nodes,

$$q_j(t) = (1 - \lambda)q_j(t - 1) + \lambda h_j(t), \tag{5.13}$$

where the parameter λ determines the averaging window. We then add the constraint of minimizing the difference between the **desired sparseness** ρ and the **actual sparseness**

q to the learning rule,

$$\Delta w_{ij} \propto v_i(h_j + \rho - q_j) - v_i^r h_j^r. \tag{5.14}$$

This works well in practice and has the added advantage of preventing **dead nodes** (Hinton 2010).