9 Unsupervised learning

In the previous learning problems we had training examples with feature vectors $\mathbf{x}$ and labels $\mathbf{y}$. In this chapter we discuss unsupervised learning problems in which no labels are given. The luck of training labeled examples restricts the type of learning that can be done, but unsupervised has important applications and even can be an important part in aiding supervised learning. Unsupervised does not mean that the learning is not guided at all; the learning follows specific principles that are used to organize the system based on the characteristics provided by the data. We will discuss several examples in this chapter.

9.1 K-means clustering

The first example is **data clustering**. In this problem domain we are given unlabelled data described by feature and asked to put them into k categories. 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 shown in Figure 9.1.

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

Fig. 9.1 k-means clustering algorithm

An example is shown in Figure **??**. The corresponding program is shown is

```
%% Demo of k-mean clustering on Gaussian data
```

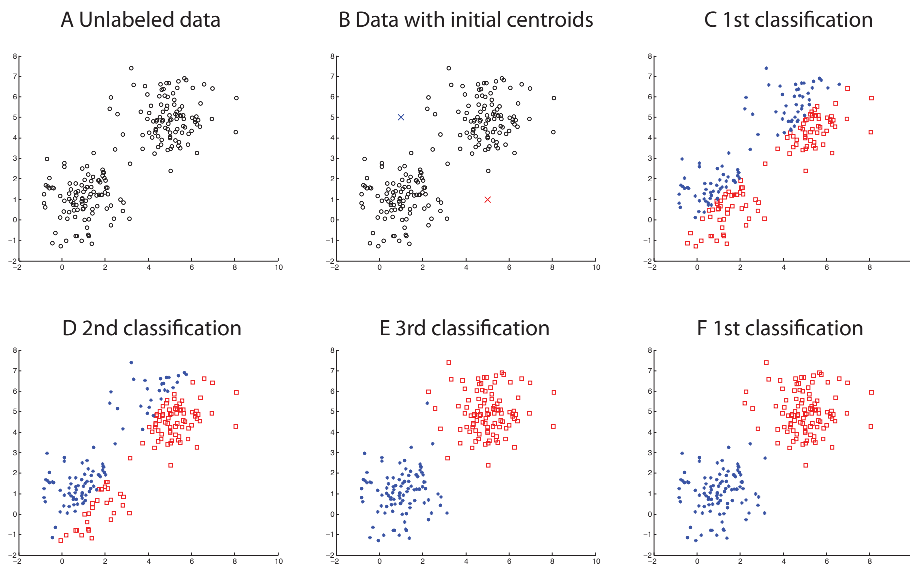


Fig. 9.2 Example of k-means clustering with two clusters.

```
%  Thomas Trappenberg, March 09
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)];

% plotting points
plot(x(:,1),x(:,2),'ko');

%two centers
mu1=[5 1]; mu2=[1 5];

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
```

9.2 Mixture of Gaussian and the EM algorithm

We have previously discussed generative models where we assumed specific models for the in-cass 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,

$$z^{(i)} \propto \text{multinomial}(\Phi_j) \tag{9.1}$$

$$x^{(i)}|z^{(i)} \propto N(\mu_j, \Sigma_j) \tag{9.2}$$

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

$$l(\Phi, \mu, \sigma) = \sum_{i=1}^{m} \log \sum_{z^{(i)}=1}^{k} p(x^{(i)}|z^{(i)}; \mu, \Sigma) p(z^{(i)}; \Phi). \tag{9.3}$$

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

$$l(\Phi, \mu, \sigma) = \sum_{i=1}^{m} \log p(x^{(i)}; z^{(i)}, \mu, \Sigma), \tag{9.4}$$

which we could use to calculate the maximum likelihood estimates of the parameter (see equations 6.27-6.29),

$$\phi_k = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j) \tag{9.5}$$

$$\mu_k = \frac{\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j)\mathbf{x}^{(i)}}{\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j)} \tag{9.6}$$

$$\Sigma_k = \frac{\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j)(x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} \mathbb{1}(y^{(i)} = k)}. \tag{9.7}$$

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.9.3. 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 ϕ, μ, Σ 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. 9.3 EM algorithm

An example is shown in Fig. 9.4. 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. 9.4A 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.

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

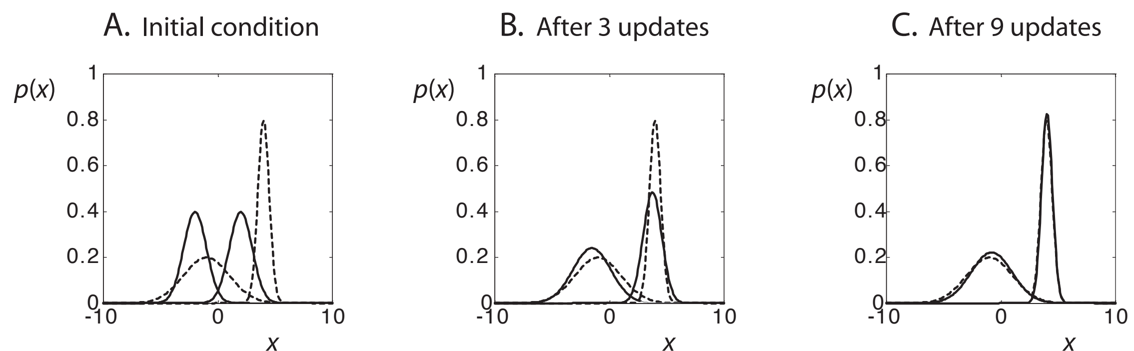


Fig. 9.4 Example of the expectation maximization (EM) algorithm for a world model with two Gaussian distributions. The Gaussian distributions of the world data (input data) are shown with dashed lines. (A) The generative model, shown with solid lines, is initialized with arbitrary parameters. In the 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 .

self-supervised strategy, which repeats the following two steps until convergence:

E-step: We make assumptions of training labels (or the probability that the data were produced by a specific cause) from the current model (expectation step); and

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

Since we do not know appropriate parameters yet, we just chose some arbitrary values as the starting point. In the example shown in Fig. 9.4A 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 9.4B and C.

Table 9.1 Program ExpectationMaximization.m

```
1   %% 1d example EM algorithm
2    clear; hold on; x0=-10:0.1:10;
3    var1=1; var2=1; mu1=-2; mu2=2;
4    normal= @(x,mu,var) exp(-(x-mu).^2/(2*var))/sqrt(2*pi*var);
5    while 1
6   %%plot distribution
7       clf; hold on;
8       plot(x0, normal(x0,-1,4),'k:');
9       plot(x0, normal(x0,4,.25),'k:');
10      plot(x0, normal(x0,mu1,var1),'r');
11      plot(x0, normal(x0,mu2,var2),'b');
12      waitforbuttonpress;
13  %% data
14      x=[2*randn(50,1)-1;0.5*randn(50,1)+4;];
15  %% recogintion
16      c=normal(x,mu1,var1)>normal(x,mu2,var2);
17  %% maximization
18      mu1=sum(x(c>0.5))/sum(c);
19      var1=sum((x(c>0.5)-mu1).^2)/sum(c);
20      mu2=sum(x(c<0.5))/(100-sum(c));
21      var2=sum((x(c<0.5)-mu2).^2)/(100-sum(c));
22   end
```

Simulation

The program used to produce Fig. 9.4 is shown in Table 9.1. 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,

$$P(c|\mathbf{x};G) = \frac{P(\mathbf{x}|c;G)P(c;G)}{P(\mathbf{x};G)}. \tag{9.8}$$

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.

9.3 Dimensionality reduction

Eigenspace and PCA, ICA, factor analysis, nonlinear dimensionality reduction (?), spectral clustering.

on-board object recognition

9.4 The Boltzmann machine

9.4.1 General one-layer module

Our last model that uses unsupervised learning is again a general learning machine invented by Geoffrey Hinton and Terrance Sejnowski in the mid 1980 called **Boltzmann machine**. This machine is a general form of a recurrent neural network with **visible nodes** that receive input or provide output, and **hidden notes** that are not connected to the outside world directly. Such a stochastic dynamic network, a recurrent system with hidden nodes, together with the adjustable connections, provide the system with enough degrees of freedom to approximate any dynamical system. While this has been recognized for a long time, finding practical training rules for such systems have been a major challenge for which there was only recently major progress. These machines use unsupervised learning to learn hierarchical representations based on the statistics of the world. Such representations are key to more advanced applications of machine learning and to human abilities.

The basic building block is a one-layer network with one visible layer and one hidden layer. An example of such a network is shown in Fig. 9.5. The nodes represent

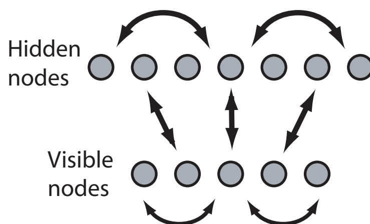


Fig. 9.5 A Boltzmann machine with one visible and one hidden layer.

random variable similar to the Bayesian networks discussed before. We will specifically consider binary nodes that mimic neuronal states which are either firing or not. The connections between the have weights w_{ij} which specify how much they influence the on-state of connected nodes. Such systems can be described by an energy function. The energy between two nodes that are symmetrically connected with strength w_{ij} is

$$H^{nm} = -\frac{1}{2}\sum_{ij} w_{ij} s_i^n s_j^m. \tag{9.9}$$

The state variables, s, have superscripts n or m which can have values (v) or (h) to indicate visible and hidden nodes. We consider again the probabilistic update rule,

$$p(s_i^n = +1) = \frac{1}{1 + \exp(-\beta \sum_j w_{ij} s_j^n)}, \tag{9.10}$$

with inverse temperature, β, which is called the Glauber dynamics in physics and describes the competitive interaction between minimizing the energy and the randomizing thermal force. The probability distribution for such a stochastic system is called the Boltzmann–Gibbs distribution. Following this distribution, the distribution of visible states, in thermal equilibrium, is given by

$$p(\mathbf{s}^v; \mathbf{w}) = \frac{1}{Z}\sum_{m \in h} \exp(-\beta H^{vm}), \tag{9.11}$$

where we summed over all hidden states. In other words, this function describes the distribution of visible states of a Boltzmann machine with specific parameters, $\mathbf{w}$, representing the weights of the recurrent network. The normalization term, $Z = \sum_{n,m} \exp(-\beta H^{nm})$, is called the **partition function**, which provides the correct normalization so that the sum of the probabilities of all states sums to one. These stochastic networks with symmetrical connections have been termed **Boltzmann machines** by **Ackley**, **Hinton** and **Sejnowski**.

Let us consider the case where we have chosen enough hidden nodes so that the system can, given the right weight values, implement a generative model of a given world. Thus, by choosing the right weight values, we want this dynamical system to approximate the probability function, $p(\mathbf{s}^v)$, of the sensory states (states of visible nodes) caused by the environment. To derive a learning rule, we need to define an objective function. In this case, we want to minimize the difference between two density functions. A common measure for the difference between two probabilistic distributions is the Kulbach–Leibler divergence (see Appendix 2.1.6),

$$\mathrm{KL}(p(\mathbf{s}^v), p(\mathbf{s}^v; \mathbf{w})) = \sum_{\mathbf{s}}^{v} p(\mathbf{s}^v) \log \frac{p(\mathbf{s}^v)}{p(\mathbf{s}^v; \mathbf{w})} \tag{9.12}$$

$$= \sum_{\mathbf{s}}^{v} p(\mathbf{s}^v) \log p(\mathbf{s}^v) - \sum_{\mathbf{s}}^{v} p(\mathbf{s}^v) \log p(\mathbf{s}^v; \mathbf{w}). \tag{9.13}$$

To minimize this divergence with a gradient method, we need to calculate the derivative of this 'distance measure' with respect to the weights. The first term in the difference in

eqn 9.13 is the entropy (see Appendix **??**) of sensory states, which does not depend on the weights of the Boltzmann machine. Minimizing the Kulbach–Leibler divergence is therefore equivalent to maximizing the average log-likelihood function,

$$l(\mathbf{w}) = \sum_{\mathbf{s}}^{v} p(\mathbf{s}^v) \log p(\mathbf{s}^v; \mathbf{w}) = \langle \log p(\mathbf{s}^v; \mathbf{w}) \rangle. \tag{9.14}$$

In other words, we treat the probability distribution produced by the Boltzmann machine as a function of the parameters, w_ij, and choose the parameters which maximize the likelihood of the training data (the actual world states). Therefore, the averages of the model are evaluated over actual visible states generated by the environment. The log-likelihood of the model increases the better the model approximates the world. A standard method of maximizing this function is gradient ascent, for which we need to calculate the derivative of $l(\mathbf{w})$ with respect to the weights. We omit the detailed derivation here, but we note that the resulting learning rule can be written in the form

$$\Delta w_{ij} = \eta \frac{\partial l}{\partial w_{ij}} = \eta \frac{\beta}{2} \left(\langle s_i s_j \rangle_{\text{clamped}} - \langle s_i s_j \rangle_{\text{free}} \right). \tag{9.15}$$

The meaning of the terms on the right-hand side is as follows. The term labelled 'clamped' is the thermal average of the correlation between two nodes when the states of the visible nodes are fixed. The termed labelled 'free' is the thermal average when the recurrent system is running freely. The Boltzmann machine can thus be trained, in principle, to represent any arbitrary density functions, given that the network has a sufficient number of hidden nodes.

This result is encouraging as it gives as an exact algorithm to train general recurrent networks to approximate arbitrary density functions. The learning rule looks interesting since the clamped phase could be associated with a sensory driven agent during an awake state, whereas the freely running state could be associated with a sleep phase. Unfortunately, it turns out that this learning rule is too demanding in practice. The reason for this is that the averages, indicated by the angular brackets in eqn 9.15, have to be evaluated at thermal equilibrium. Thus, after applying each sensory state, the system has to run for a long time to minimize the initial transient response of the system. The same has to be done for the freely running phase. Even when the system reaches equilibrium, it has to be sampled for a long time to allow sufficient accuracy of the averages so that the difference of the two terms is meaningful. Further, the applicability of the gradient method can be questioned since such methods are even problematic in recurrent systems without hidden states since small changes of system parameters (weights) can trigger large changes in the dynamics of the dynamical systems. These problems prevented, until recently, more practical progress in this area. Recently, Hinton and colleagues developed more practical, and biologically more plausible, systems which are described next.

9.4.2 The restricted Boltzmann machine and contrastive Hebbian learning

Training of the Boltzmann machine with the above rule is challenging because the states of the nodes are always changing. Even with the visible states clamped, the

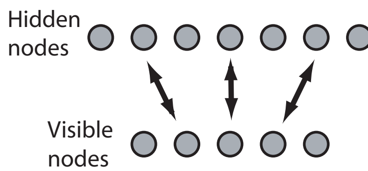


Fig. 9.6 Restricted Boltzmann machine in which recurrences within each later are removed.

states of the hidden nodes are continuously changing for two reasons. First, the update rule is probabilistic, which means that even with constant activity of the visible nodes, hidden nodes receive variable input. Second, the recurrent connections between hidden nodes can change the states of the hidden nodes rapidly and generate rich dynamics in the system. We certainly want to keep the probabilistic update rule since we need to generate different responses of the system in response to sensory data. However, we can simplify the system by eliminating recurrent connections within each layer, although connections between the layers are still bidirectional. While the simplification of omitting collateral connections is potentially severe, much of the abilities of general recurrent networks with hidden nodes can be recovered through the use of many layers which bring back indirect recurrencies. A **restricted Boltzmann machine** (RBM) is shown in Fig. 9.6.

When applying the learning rule of eqn 9.15 to one layer of an RBM, we can expect faster convergence of the rule due to the restricted dynamics in the hidden layer. We can also write the learning rule in a slightly different form by using the following procedure. A sensory input state is applied to the input layer, which triggers some probabilistic recognition in the hidden layer. The states of the visible and hidden nodes can then be used to update the expectation value of the correlation between these nodes, $\langle s_i^v s_j^h \rangle^0$, at the initial time step. The pattern in the hidden layer can then be used to approximately reconstruct the pattern of visible nodes. This **alternating Gibbs sampling** is illustrated in Fig. 9.7 for a connection between one visible node and one hidden node, although this learning can be done in parallel for all connections. The learning rule can then be written in form,

$$\Delta w_{ij} \propto \langle s_i^v s_j^h \rangle^0 - \langle s_i^v s_j^h \rangle^\infty. \tag{9.16}$$

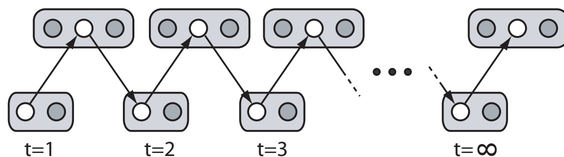


Fig. 9.7 Alternating Gibbs sampling.

Alternating Gibbs sampling becomes equivalent to the Boltzmann machine learning rule (eqn 9.15) when repeating this procedure for an infinite number of time steps, at which point it produces pure fantasies. However, this procedure still requires averaging over long sequences of simulated network activities, and sufficient evaluations of thermal averages can still take a long time. Also, the learning rule of eqn 9.16 does

not seem to correspond to biological learning. While developmental learning also takes some time, it does not seems reasonable that the brain produces and evaluates long sequences of responses to individual sensory stimulations. Instead, it seems more reasonable to allow some finite number of alternations between hidden responses and the reconstruction of sensory states. While this does not formally correspond to the mathematically derived gradient leaning rule, it is an important step in solving the learning problem for practical problems, which is a form of **contrastive divergence** introduced by Geoffrey Hinton. It is heuristically clear that such a restricted training procedure can work. In each step we create only a rough approximation of ideal average fantasies, but the system learns the environment from many examples, so that it continuously improves its expectations. While it might be reasonable to use initially longer sequences, as infants might do, Hinton and colleagues showed that learning with only a few reconstructions is able to self-organize the system. The self-organization, which is based on input from the environment, is able to form internal representations that can be used to generate reasonable sensory expectations and which can also be used to recognize learned and novel sensory patterns.

The basic Bolzmann machine with a visible and hidden layer can easily be combined into hierarchical networks by using the activities of hidden nodes in one layer as inputs to the next layer. Hinton and colleagues have demonstrated the power of restricted Boltzmann machines for a number of examples. For example, they applied layered RBMs as auto-encoders where restricted alternating Gibbs sampling was used as pre-training to find appropriate initial internal representations that could be fine-tuned with backpropagation techniques to yield results surpassing support vector machines. However, for our discussions of brain functions it is not even necessary to yield perfect solutions in a machine learning sense, and machines can indeed outperform humans in some classification tasks solved by machine learning methods. For us, it is more important to understand how the brain works.

Simulation 1: Hinton

To illustrate the function of an anticipating brain model, we briefly outline a demonstration by the Hinton group. The online demonstration can be run in a browser from `http://www.cs.toronto.edu/~hinton/adi`, and a stand alone version of this demonstration is available at this book's resource page. MATLAB source code for restricted Boltzmann machines are available at Hinton's home page. An image of the demonstration program is shown in Fig. 9.8. The model consists of a combination of restricted Boltzmann machines and a Helmholz machine. The input layer is called the **model retina** in the figure, and the system also contains a **recognition-readout-and-stimulation** layer. The model retina is used to apply images of handwritten characters to the system. The recognition-readout-and-stimulation layer is a brain imaging and stimulation device from and to the uppermost RBM layer. This device is trained by providing labels as inputs to the RBM for the purpose of 'reading the mind' of the system and to give it high-level instructions. This device learns to recognize patterns in the uppermost layer and map them to their meaning, as supplied during supervised learning of this device. This is somewhat analogous to **brain–computer interfaces** developed with different brain-imaging devices such as EEG, fMRI, or implanted electrodes. The advantage of the simulated device is that it can read the activity of

every neuron in the upper RBM layer. The device can also be used with the learned connections in the opposite direction to stimulate the upper RBM layer with typical patterns for certain image categories.

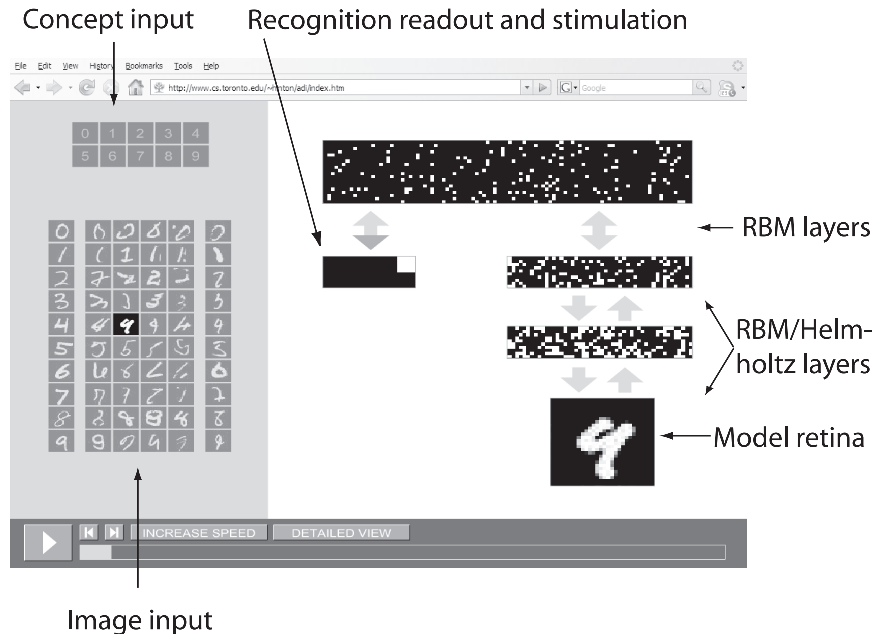


Fig. 9.8 Simulation of restricted Boltzmann machine by Geoffrey Hinton and colleagues, available at `www.cs.toronto.edu/∼hinton/adi`.

The model for this demonstration was trained on images of handwritten numbers from a large database. Some example images can be seen on the left-hand side. All layers of this model were first treated as RBMs with symmetrical weights. Specifically, these were trained by applying images of handwritten characters to the model retina and using three steps of alternating Gibbs sampling for training the different layers. The evolving representations in each layer are thus purely unsupervised. After this basic training, the model was allowed, for fine-tuning purposes, to develop different weight values for the recognition and generative model as in Helmholtz machines with a wake–sleep training algorithm as mentioned above.

The simulations provided by Hinton demonstrate the ability of the system after training. The system can be tested in two ways, either by supplying a handwritten image and asking for recognition, or by asking the system to produce images of a certain letter. These two modes can be initiated by selecting either an image or by selecting a letter category on the left-hand side. In the example shown in Fig. 9.8, we selected an example of an image of the number 4. When running the simulation, this image triggers response patterns in the layers. These patterns change in every time step, due to the probabilistic nature of the updating rule. The recognition read-out of the uppermost layer does, therefore, also fluctuate. In the shown example, the response of the system is 4, but the letter 9 is also frequently reported. This makes sense, as

this image does also look somewhat like the letter 9. A histogram of responses can be constructed when counting the responses over time, which, when properly normalized, corresponds to an estimate of the probability over high-level concepts generated by this sensory state. Thus, this mode tests the recognition ability of the model.

The stimulation device connected to the upper RBM layer allows us to instruct the system to 'visualize' specific letters, which corresponds to testing the generative ability of the model. For example, if we ask the system to visualize a letter 4 by evoking corresponding patterns in the upper layer, the system responds with varying images on the model retina. There is not a single right answer, and the answers of the system change with time. In this way, the system produces examples of possible images of letter 4, proportional to some likelihood that these images are encountered in the sensory world on which the system was trained. The probabilistic nature of the system much better resembles human abilities to produce a variety of responses, in contrast to the neural networks that have been popular in the 1980s, so called multilayer perceptrons, which were only able to produce single answers for each input.

Simulation 2: Simplified one layer model

A simplified version of the RBM trained on some letters are included in folder `RBM example` on the web resource page. The overage reconstruction error and some examples of reconstructions after training are shown in Fig.9.9.

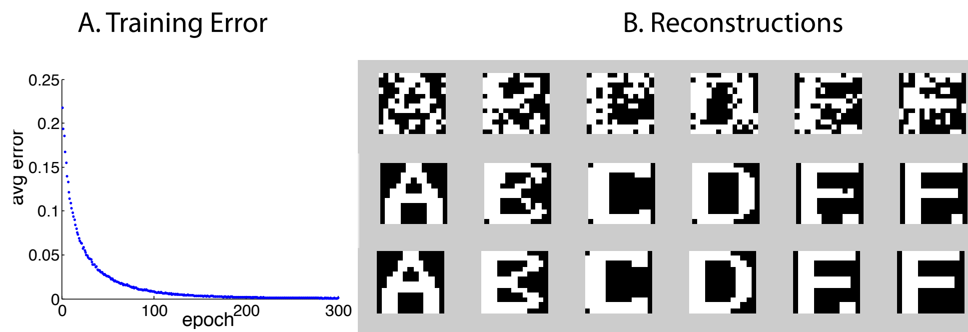


Fig. 9.9 (A) Reconstruction error during training of alphabet letters the letters, and (B) reconstructions after learning.

9.5 Sparse representations

Part V

Reinforcement learning

10 Markov Decision Process

This chapter is an introduction to reinforcement learning. We introduce here the general idea and formulation of reinforcement learning, and we will then then concentrate in this chapter on the most basic case of a Markov Decision Process (MDP). These processes are characterized by completely observable states of the system and by transition processes that only depend on the last states. In the next chapters this will be extended to temporal difference (TD) learning and partially observable situations.

10.1 Learning from reward and the credit assignment problem

We discussed in previous chapters supervised learning in which a teacher showed an agent the desired response $\mathbf{y}$ to a given input state $\mathbf{x}$. We are now moving to the problem when the agent must discover the right action to choose and only receives some qualitative feedback from the environment such as reward or punishment. The reward feedback does not tell the agent directly which action to take. Rather, it indicates which states-action pairs are desirable (rewarded) or not (punished). The agent has to discover the right sequence of actions to take to optimize the reward over time. Choosing the right action of an actuator is traditionally the subject of control theory, and this subject is thus also called optimal control.

Reward learning introduces several challenges. For example, in typical circumstances reward is only received after a long sequence of actions. The problem is then how to assign the credit for the reward to specific actions. This is the **temporal credit assignment problem**. To illustrate this, let us think about a car that crashed into a wall. It is likely that the driver used the breaks before the car crashed into the wall, though the breaks could not prevent the accident. However, from this we should not conclude that breaking is not good and lead to crashes. In some distributed systems there is in addition a **spatial credit assignment problem** which is the problem of how to assign the appropriate credit when different parts of a system contributed to a specific outcome.

Another challenge in reinforcement learning is the balance between **exploitation** and **exploration**. That is, we might find a way to receive some small food reward if we repeat certain actions, but if we only repeat these specific actions, we might never discover a bigger reward following different actions. Some escape from self-reinforcement is important.

The idea of reinforcement learning is to use the reward feedback to build up a **value function** that reflect the expected future payoff of visiting certain states and taking certain actions. We can use such a value function to make decisions of which action to take and thus which states to visit. This is called a **policy**. To formalize

these ideas we start with simple processes where the transitions to new states depend only on the current state. A process which such a characteristics is called a **Markov process**. In addition to the Markov property, we also assume at first that the agent has full knowledge its environment. Finally, it is again important that we acknowledge uncertainties and possible errors. For example, we can take error in motor commands into account by considering probabilistic state transitions.

10.2 The Markov Decision Process

Before formalizing the decision processes in this chapter, let us begin with an example to illustrate a common setting. In this example we consider an agent that should learn to navigate through the maze shown in Figure 10.1. The states of the maze are the possible discrete positions that are simply numbered consecutively in this example, that is, $S = \{1, 2, ..., 18\}$. In The possible actions of the agent is to move one step forward, either to the north, east, south and west, that is, $A = \{N, E, S, W\}$. However, even though the agents gives these commands to its actuators, stochastic circumstances – such as faulty hardware or environmental conditions (e.g. someone 'kicking' the agent) – make the agent end up in different states with certain probabilities specified by probabilistic transition matrix $T(s'|s, a)$. For example, the probability of following actions $a = N$ might just be 80%, and the agent could then take actions $a = W$ and $a = E$ in 10% of the cases each, but never go south. We assume for now that the transition probability is given explicitly, although in many practical circumstances we might need to estimate this from examples (e.g. supervised learning). Of course, some directions are not possible from all states. Such situations can be handled in different ways such as by a reduced set of states for these positions, or by the action that the agent returns to the sending state when taking these actions. We will use the later case in the example here. Finally, the agent is given reward or punishment when the agent is moving into a new state s. For example, we can consider a deterministic reward function in which the agent is given a large reward when finding the exit to the maze ($r(18) = 1$ in the example of Figure 10.1). In practice it is also common and useful to give some small negative reward to the other states. This could, for example, represent the battery resource that the Lego robot consumes when moving to a cell in the grid, whereas it gets recharged at the exit of the maze.

A common approach to solve a maze navigation problem is path planing based on some search algorithms such as the A^* search algorithm. However, the task here is different in that the agent must discover itself the task of completing the maze. Indeed, the agent might not even be aware of this as the main task for the agent is simply to optimize future reward. Also, the probabilistic nature of the state transition is challenging for traditional search algorithms, although this can be accomplished with some dynamic extensions of the standard search algorithms. So what is the benefit of this approach? The great thing about reinforcement learning is that it is very general and can readily be applied to many task. Also, being a learning system, we can even change the task at any point by changing the reward feedback, and there should be no need to change anything in the program of the agent. Indeed, when training animals, this is usually the main way that we can communicate with the animals in learning situations since we can not verbally communicate the task goal.

We now formalize such an environment as a **Markov Decision Process (MDP)**. A MDP is characterized by a set of 5 quantities, expressed as $(S, A, T(s'|s,a), R(r|s,a), \theta)$. The meaning of these quantities are as follows.

- S is a set of states.
- A is a set of actions.
- $T(s'|s,a)$ is a **transition probability**, for reaching state s' when taking action a from state s. This transition probability only depends on the previous state, which is called the Markov condition; hence the name of the process.
- $R(r|s)$ is the probability of receiving **reward** when getting to state s. This quantity provides feedback from the environment. r is a numeric value with positive values indicating reward and negative values indicating punishment.
- θ are specific parameters for some of the different kinds of RL settings. This will be the **discount factor** γ in our first examples.

1 Start R=-0.1	6 R=-0.1			15 R=-0.1
2 R=-0.1	7 R=-0.1	10 R=-0.1	11 R=-0.1	16 R=-0.1
3 R=-0.1			12 R=-0.1	
4 R=-0.1	8 R=-0.1		13 R=-0.1	17 R=-0.1
5 R=-0.1	9 R=-0.1		14 R=-0.1	18 Goal R=1

Fig. 10.1 A maze where each state is rewarded with a value r.

An MDP is fully determined by these 5 quantities that characterize the environment completely. However, to make decisions we define two quantities that will guide the behaviour of an agent. The first quantities is the **value function** $Q^\pi(s,a)$ that specifies how valuable state s is under the policy π for different actions a. This quantity is defined as the **expected reward** as formalized below. The second quantity is the **policy** $\pi(a|s)$ which is the probability of choosing action a from state s. Note that we have kept the formulation here very general by considering probabilistic rewards and probabilistic policies, although some applications can be formulated with deterministic functions for these quantities. Since the action is uniquely determined for deterministic policies, one can then use the **state value function** $V^\pi(s)$, although it is important to realize that it is still specific to the actions taken under the policy. The function $Q^\pi(s,a)$ is often called the **state-action value function** to distinguish it from $V^\pi(s)$. Also, we consider here rewards that only depend on the state. In same rare cases reward might depend on the way a state is reached, in which case the reward probability can be easily extended to R(rls,a).

Reinforcement learning algorithms are aimed at calculating or estimating value functions to determine useful actions. However, most of the time we are mostly interested in finding the best, or **optimal policy**. Since choosing the right actions from states is the aim of control theory, this is sometimes called **optimal control**. The optimal policy is the policy which maximizes the value (expected reward) for each state. Thus, if we denote the maximal value as

$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \tag{10.1}$$

the optimal policy is the policy that maximizes the expected reward,

$$\pi^*(a|s) = \arg\max_\pi Q^\pi(s,a). \tag{10.2}$$

While direct search in the space of all possible policies is possible in examples with small sets of states and actions, a major problem of reinforcement learning is the exploding number of policies and states with increasing dimension, and solving this 'course of dimensionality' will be part of the advanced discussions later.

Finally we need define the value function more precisely. As already stated above, the value function is defined as the expected value of all future rewards, which is called the **total payoff**. The total payoff is the sum of all future reward, that is, the immediate reward of reaching state s as well as the rewards of subsequent states by taking the specific actions under the policy. Let us consider the specific episode of consecutive states $s_1, s_2, s_3, ...$ following s. Note that the states s_n are functions of the starting state s and the actual policy. The cumulative reward for this specific episode when visiting the consecutive states $s_1, s_2, s_3, ...$ from the starting state s under policy π is thus

$$r_\infty(s) = r(s) + r(s_1) + r(s_2) + r(s_3) + ... \tag{10.3}$$

One problem with this definition that this value could be unbounded as it runs over infinitely many states into the future. A possible solution of this problem is to restrict the sum by considering only a **finite reward horizon**, for example by only consider rewards given within a certain finite number of steps such as

$$r_4(s) = r(s) + r(s_1) + r(s_2) + r(s_3). \tag{10.4}$$

Another way to solve the infinite payoff problem is to consider reward that is discounted when it is given at later times. In particular, if we consider the discount factor $0 < \gamma < 1$ for each step, we have a total payoff

$$r_\gamma(s) = r(s) + \gamma r(s_1) + \gamma^2 r(s_2) + \gamma^3 r(s_3) + ... \tag{10.5}$$

Since we consider probabilistic state transitions, policies and rewards, we can only estimate the expected value of the total payoff when starting at state s and taking actions according to a policy $\pi(a|s)$. We denote this expected value with the function $E\{R_\gamma(s)\}_\pi$. The expected total discounted payoff from state s when following policy π is thus

$$Q^\pi(s,a) = E\{r(s) + \gamma r(s_1) + \gamma^2 r(s_2) + \gamma^3 r(s_3) + ...\}_\pi. \tag{10.6}$$

This is called the **value-function** for policy pi. Note that this value function not only depends on a specific state but also on the action taken from state s since it is specific for a policy. We will now derive some methods to estimate the value-function for a specific policy before discussing methods of finding the optimal policy.

10.3 The Bellman equation

10.3.1 Bellman equation for a specific policy

With a complete knowledge of the system, that includes a perfect knowledge of the state the agent is in as well as the transition probability and reward function, it is possible to estimate the value function for each policy π from a self-consistent equation. This was already noted by Richard Bellman in the mid 1950s, and this method is known as **dynamic programming**. To derive the Bellman equation we consider the value function, equation 10.6 and separate the expected value of the immediate reward from the expected value of the reward fro visiting subsequent states,

$$Q^\pi(s,a) = E\{r(s)\}_\pi + \gamma E\{r(s_1) + \gamma r(s_2) + \gamma^2 r(s_3) + ...\}_\pi. \tag{10.7}$$

The second expected value on the right hand side is that of the value function for state s_1, but state s_1 is related to state s since state s_1 is the state that can be reached with a certain probability from s when taking action a_1 according to policy π, for example like $s_1 = s + a_1$ and $s_n = s_{n-1} + a_n$. We can incorporate this into the equation by writing

$$Q^\pi(s,a) = r(s) + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') E\{r(s') + \gamma R(s'_1) + \gamma^2 R(s'_2) + ...\}_\pi, \tag{10.8}$$

where s'_1 is the next state after state s', etc. Thus, the expression on the right is the state-value-function of state s'. If we substitute the corresponding expression of equation 10.6 into the above formula, we get the **Bellman equation** for a specific policy, namely

$$Q^\pi(s,a) = r(s) + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') Q^\pi(s',a'). \tag{10.9}$$

In the case of deterministic policies, the action a is given by the policy and the value function $Q^\pi(s,a)$ reduces to $V^\pi(s)$. In this case the equation simplifies to

$$V^\pi(s) = r(s) + \gamma \sum_{s'} T(s'|s,a) V^\pi(s'). \tag{10.10}$$

Such a linear equation system can be solved with our complete knowledge of the environment. In an environment with N states, the Bellman equation is a set of N linear equations, one for each state, with N unknowns which are the expected value for each state. We can thus use well known methods from linear algebra to solve for $V^\pi(s)$. This can be formulated compactly with Matrix notation,

$$\mathbf{r} = (\mathbb{1} - \gamma \mathbf{T}) \mathbf{V}^\pi, \tag{10.11}$$

where $\mathbf{r}$ is the reward vector, $\mathbb{1}$ is the unit diagonal matrix, and $\mathbf{T}$ is the transition matrix. To solve this equation we have to invert a matrix and multiply this with the reward values,

$$\mathbf{V}^\pi = (\mathbb{1} - \gamma \mathbf{T})^{-1} \mathbf{r}^t, \tag{10.12}$$

where $\mathbf{r}^t$ is the transpose of $\mathbf{r}$

Note that the analytical solution of the Bellman equation is only possible because we have complete knowledge of the system, including the reward function r, which itself requires a perfect knowledge of the state in which the agent is in. Also, while we used this solution technique from linear algebra, it is much more common to use the Bellman equation directly and calculate a state-value-function iteratively for each policy. We can start with a guess $\mathbf{V}$ for the value of each state, and calculating from this a better estimate

$$\mathbf{V} \leftarrow \mathbf{r} + \gamma \mathbf{T}\mathbf{V} \tag{10.13}$$

until this process converges. While we mainly use this iterative approach, we will also give an example below of using the analytical example.

10.3.2 Policy iteration

The equations above depends on a specific policy. As mentioned above, in many cases we are mainly interested in finding the policy that gives us the **optimal payoff** and we could simply search for this by considering all possible policies. But this is usually not practical in most but a small number of examples since the number of possible policies is equal to the number of actions to the power of the number of states. This explosion of the problem size with the number of states is one of the main challenges in reinforcement learning and was termed **curse of dimensionality** by Richard Bellman. Most of the problems discussed here have small number of states and small number of possible actions, so that this is not a major concern here.

However, a much more efficient method is incrementally find the value function for a specific policy and then use the policy which maximizes this value function for the next round. The **policy iteration** algorithm is outlined in Figure 10.2. In addition to an initial guess of the value function, we have now also to initialize the policy, which could be randomly chosen from the set of possible actions at each state. For this value functions we could then calculate the corresponding state-value-function according to equation 10.9. This step is the evaluating the specific policy. The next step is to take this value functions and calculate the corresponding best set of actions to take accordingly, which corresponds to the next candidate policy. This policy is again simply to take the action from each state that would maximize the corresponding future payoff. These two steps, the policy evaluation and the policy improvement are repeated the policy won't change any more.

To demonstrate this scheme for solving MDPs, we will follow a simple example, that of a chain of N states. The states of the chain are labeled consecutively from left to right, $s = 1, 2, ..., N$. An agent has two possible actions, go to the left (lower state numbers; $a = -1$), or go to the right (higher state numbers; $a = +1$). However, in P cases the system responds with the opposite move from the intended. The last state in the chain, state number N, is rewarded with $r(N) = 1$, whereas going to the first state in the chain is punished with $r(1) = -1$. The reward of the intermediate states is set to a small negative value, such as $r(i) = -0.1, 1 < i < N$. We consider a discount factor γ.

The transition probabilities $T(s'|s, a)$ for the chain example are zero expect for the following elements,

$$T(1|1, -1) = 1 \tag{10.14}$$

Choose initial policy and value function
Repeat until policy is stable {
    1. Policy evaluation
    Repeat until change in values is sufficiently small {
        For each state {
            Calculate the value of neighbouring states when taking action according to current policy.
            Update estimate of optimal value function. } V^π equation 10.9
        } each state
    } convergence
    2. Policy improvement
    new policy according to equation 10.21, assuming $V^* \approx$ current V^π
} policy

Fig. 10.2 Policy iteration with asynchronous update.

$$T(N|N,+1) = 1 \tag{10.15}$$
$$T(s-a|s,a) = 1-P \tag{10.16}$$
$$T(s+a|s,a) = P \tag{10.17}$$

The first two entries specify the ends of the chain as **absorbing boundaries** as the agent would stay in this state one it reaches these states. We can also write this as two **transfer matrices**, one for each possible actions. For $a = 1$ this is,

$$\begin{pmatrix} 1 \cdots & \cdots 0 \\ \vdots & \vdots \\ 0 \cdots 0 \; 1-P \; 0 \; P \; 0 \cdots & 0 \\ \vdots & \vdots \\ 0 \cdots & \cdots 1 \end{pmatrix} \tag{10.18}$$

and for $a = -1$ this is

$$\begin{pmatrix} 1 \cdots & \cdots 0 \\ \vdots & \vdots \\ 0 \cdots 0 \; P \; 0 \; 1-P \; 0 \cdots & 0 \\ \vdots & \vdots \\ 0 \cdots & \cdots 1 \end{pmatrix} \tag{10.19}$$

The corresponding Matlab code for setting up the chain example is

```
% Chain example:
% Policy iteration with analytical solution of Bellman equation
clear;
N=10; P=0.8; gamma=0.9; % parameters
U=diag(ones(1,N)); % unit diaogonal matrix
```

```
T=zeros(N,N,2); % transfer matrix
r=zeros(1,N)-0.1; r(1)=-1; r(N)=1; % reward function

T(1,1,:)=1; T(N,N,:)=1;
for i=2:N-1;
    T(i,i-1,1)=P;
    T(i,i+1,1)=1-P;
    T(i,i-1,2)=1-P;
    T(i,i+1,2)=P;
end
```

The policy iteration part of the program is then given as follows:

```
% random start policy
policy=floor(2*rand(1,N))+1; %random vector of 1 (going left) and 2 (going right)
Vpi=zeros(N,1); % initial arbitrary value function
iter = 0; % counting iteration
converge=0;
% Loop until convergence
    while ~converge
        % Updating the number of iterations
        iter = iter + 1;
        % Backing up the current V
        old_V = Vpi;
        %Transfer matrix of choosen action
        Tpi=zeros(N); Tpi(1,1)=1; T(N,N)=1;
        for s=2:N-1;
            Tpi(s,s-1)=T(s,s-1,policy(s));
            Tpi(s,s+1)=T(s,s+1,policy(s));
        end
        % Calculate V for this policy
        Vpi=inv(U-gamma*Tpi)*r';
        % Updating policy
        policy(1)=0; policy(N)=0; %absorbing states
        for s=2:N-1
            [tmp,policy(s)] = max([Vpi(s-1),Vpi(s+1)])
        end
        % Check for convergence
        if abs(sum(old_V - Vpi)) < 0.01
            converge = 1;
        end
    end
iter, policy
```

The whole procedure should be run until the policy does not change any more. This stable policy is then the policy we should execute in the agent.

10.3.3 Bellman equation for optimal policy and value iteration

Instead of using the above Bellman equation for the value function and then calculating the optimal value functions, we can also derive a version of **Bellman's equation for the optimal value function** itself. This second kind of a Bellman equation is given by

$$V^*(s) = r(s) + \max_a \gamma \sum_{s'} T(s'|s,a)V^*(s'). \tag{10.20}$$

The max function is a bit more difficult to implement in the analytic solution, but we can again easily use and iterative method to solve for this optimal value function. This is called **value iteration**. Note that this version includes a max function over all possible actions in contrast to the Bellman equation for a given policy, equation 10.9. As outlined in figure 10.3, we start again with a random guess for the value of each state and then iterate over all possible states using the Bellman equation for the optimal value function, equation 10.20. More specifically, This algorithm takes an initial guess of the optimal value function, typically random or all zeros. We then iterate over the main loop until the change of the value function is sufficiently small. For example, we could calculate the sum of value functions in each iteration (t) and then terminate the procedure if the absolute difference of consecutive iterations is sufficiently small, that is if $|\sum_s V_t^*(s) - \sum_s V_{t-1}^*(s)| <$threshold. In each of those iterations, we iterate over all states and update the estimated optimal value functions according to equation 10.20.

Choose initial estimate of optimal value function
Repeat until change in values is sufficiently small {
  For each state {
    Calculate the maximum expected value of neighbouring states for each possible action.
    Use maximal value of this list to update estimate of optimal value function. } V^* equation 10.20
  } each state
} convergence
Calculate optimal value function from equation 10.21

Fig. 10.3 Value Iteration with asynchronous update.

Finally, after convergence of the procedure to get a good approximation of the optimal value function, we can calculate the **optimal policy** by considering all possible actions from each state,

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s'|s,a)V^*(s'), \tag{10.21}$$

which should be used by an agent to achieve good performance.

As a minor side note, the state iteration can be done in various ways. For example, in the **sequential asynchronous updating schema** we update each state in sequence and repeat this procedure over several iterations. Small variations of this schema are concerned with how the algorithm iterates over states. For example, instead of

iterating sequentially over the states, we could also use a random oder. We could also first calculate the maximum value of neighbours for all states before updating the value function for all states with an **synchronous updating schema**. Since it can be shown that theses procedure will converge to the optimal solution, all these schemas should work similarly well though might differ slightly for particular examples. Important is however that that the agent goes repeatedly to every possible state in the system. While this works only because we have complete knowledge of the system, it also only works well in the examples with small state spaces. In general, such iterations over the state space are problematic for large state space.

The previously discussed policy iteration has some advantages over value iterations. In value iteration we have to try out all possible actions when evaluating the value function, and this can be time consuming when there are many possible actions. In policy iteration, we choose a specific policy, although we have then to iterate over consecutive policies. In practice it turns out that policy iteration often converges fairly rapidly so that it becomes a practical method. However, value iteration is a little bit easier and has more similarities to the algorithms discussed below that are also applicable to situations where we do not know the environment a priori.

Exercise:

Implement the value iteration for the chain problem and plot the learning curve (how the error changes over time), the optimal value function, and the optimal policy. Change parameters such as N, γ, and the number of iterations and discuss the results.

Exercise:

Solve the Russel grid with the policy iteration using the basic Bellman functions iteratively, and compare this method to the value iteration.

11 Temporal Difference learning and POMDP

11.1 Temporal Difference learning

Dynamic programming can solve the basic reinforcement learning problem since we assumed a complete knowledge of the system, which includes the knowledge about the precise state of the agent, transition probabilities, the reward functions, etc. In reality, and commonly in robotics, we might not know the state and other quantities directly but most usually estimate them from interacting with the environment. The algorithms in this chapter are all focused of solving the reinforcement problem on-line by interacting with the environment. We will first assume that we still know the exact state of the agent at each step and will then discuss partially observable situations below.

Likely the most direct methods of estimating the value of states is to just act in the environment and thereby sampling and memorizing reward from which the expected value can be calculated by averaging. Such methods are generally called **Monte Carlo methods**. While general monte Carlo methods are very universal and might work well in some applications, we will concentrate here right away on algorithms which combine ideas from Monte Carlo methods with that of dynamic programming. Such influential methods in reinforcement learning have been developed by Rick Satton and Andrew Barto, and also by Chris Watkins, although some of those methods have even been applied to learning to play checkers by Arthur Samuel in the late 1950s. Common to these methods is that they use the difference between expected reward and actual reward. Such algorithms are therefore generally called **temporal difference (TD) learning**. We start again by estimating the value function for a specific policy before moving to schemas for the estimating the optimal policy.

Let us recall Bellman's equation for value function of a policy π (eq.10.9),

$$V^{\pi}(s) = r(s) + \gamma \sum_{s''} T(s''|s,a) V^{\pi}(s''). \tag{11.1}$$

The sum on the right-hand side is over all the states that can be reached from state s. A difficulty in practice is often that we don't know the transition probability and have to estimate this somehow. The strategy we are taking now is that we approximate the sum on the right hand side by a specific episode taken by the agent. This interaction of the interaction with the environment that makes this an on-line learning tasks as in Monte Carlo methods. But in contrast to Monte Carlo methods we do not take and memorize the following steps and associated reward but estimate the expected reward of the following step with the current estimate of the value function which is an estimate of the reward of the whole episode. Such strategies are sometime called a **bootstrap**

method as if pulling oneself out of the boots by one owns strap. We will label the actual state reached by the agent as s'. Thus, the approximation can be written as

$$\sum_{s''} T(s''|s,a)V^{\pi}(s'') \approx V^{\pi}(s'). \tag{11.2}$$

While this term makes certainly an error, the idea is that this will still result in an improvement of the estimation of the value function, and that other trials have the possibility to evaluate other states that have not been reached in this trial. The value function should then be updated carefully, by considering the new estimate only incrementally,

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\{r(s) + \gamma V^{\pi}(s') - V^{\pi}(s)\}. \tag{11.3}$$

This is called **TD learning**. The constant α is called a learning rate and should be fairly small. This policy evaluation can then be combined with policy iteration as discussed already in the section on dynamic programming.

11.2 Temporal difference methods for optimal control

Most of the time we are mainly interested in optimal control that maximizes the reward receiver over time. We will now turn to this topic. In this section we will explicitly consider stochastic policies and will thus return to the notation of the state-action value function. Also, since we are always talking about the optimal value function in the following, we will drop the star in the formulas and juts use $Q(s,a) = Q^*(s,a)$ for the optimal value.

A major challenge in on-line learning of optimal control when the agent is interacting with the environment is the trade-off between maximizing the reward in each step and exploring the environment for larger future reward while excepting some smaller immediate reward. This was not a problem in dynamic programming since we would iterate over all states. However, in large state space and in situation where exploring takes time and resources, typical for robotics applications, we can not expect to iterate extensively over all states and we must thrive for a good balance between **exploration** and **exploitation**. Without exploration it can easily happen that the agent get stuck in a suboptimal solution. Indeed, we could only solve the chain problem above because it included some probabilistic transition matrices that helped us exploring the space.

Optimal control demands to maximize reward and therefore to always go to the state with the maximal expected reward at each time. But this could prevent finding even higher payoffs. A essential ingredient of the following algorithms is thus the inclusion of randomness in the policy. For example, we could follow most of the time the **greedy policy**, which chooses another possible actions in a small number ϵ of times. This probabilistic policy is called the **ϵ-greedy policy**,

$$\pi(a = \arg\max_a Q(s,a)) = \epsilon. \tag{11.4}$$

This policy is choosing the policy with the highest expected payoff most of the time while treating all other actions the same. A more graded approach is using the **softmax policy** that choses each action proportional to a Boltzmann distribution

$$\pi(a|s) = \frac{e^{Q(s,a)}}{\sum_{a'} e^{Q(s,a')}}. \tag{11.5}$$

While there are other possible choices of a probabilistic policy, the general idea of the following algorithms do not depend on this details, and we therefore use the ϵ-greedy policy for illustration purposes.

To derive the following on-line algorithms for optimal control, we now consider the Bellman equation for the optimal value function (eq 10.20) generalized for stochastic policies,

$$Q(s,a) = r(s) + \max_a \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') Q(s',a'). \tag{11.6}$$

We again use an online procedure in which the agent takes specific actions. Indeed, we now always consider policies that choses actions most of the time that lead to the largest expected payoff. Thus, by taking the action according to the policy we can write a temporal difference learning rule for the optimal stochastic policy as

$$Q(s,a) \leftarrow Q(s,a) + \alpha\{r(s) + \gamma Q(s',a') - Q(s,a)\}, \tag{11.7}$$

where the actions a' is the action chosen according to the policy. This **on-policy TD algorithm** is called **Sarsa** for state-action-reward-state-action. Note that the action a' will not always be the action that maximizes the expected reward since we are using stochastic policies. Thus, slightly different approach is using only the action to the maximal expected reward for the value function update while still exploring the state space through the policy.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\{r(s) + \max_{a'} \gamma Q(s',a') - Q(s,a)\}. \tag{11.8}$$

Such a **off-policy TD algorithm** is called **Q-leaning**

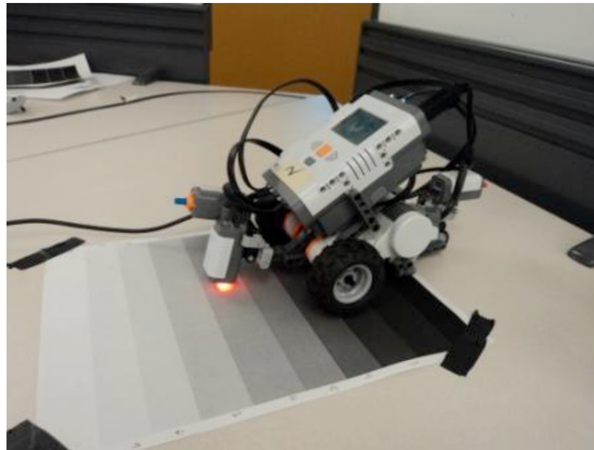
11.3 Robot exercise with reinforcement learning

11.3.1 Chain example

The first example follows closely the chain example discussed in the text. We consider thereby an environment with 8 states. An important requirement for the algorithms is that the robot must know in which state it is in. As discussed in Chapter **??**, this localization problem is a mayor challenge in robotics. We use here the example where we use a state indicator sheet as used in section **??**. You should thereby use the implemented of the calibration from the earlier exercise.

Choose initial policy and value function
Repeat until policy is stable {
    1. Policy evaluation
    Repeat until change in values is sufficiently small {
    Remembering the value function and reward of current state (eligibility trace)
    If rand>ϵ
        Go to next state according to policy of equation **??**
    else
        go to different state
    Update value function of previous state according to (equation 11.3)
    $V^\pi(s-1) \leftarrow V^\pi(s-1) + \alpha(R(s-1) + \gamma V^\pi(s) - V^\pi(s-1))$
        } convergence
    2. Policy improvement
    new policy according to equation 10.21, assuming $V^* \approx$ current V^π
} policy

Fig. 11.1 On-policy Temporal Difference (TD) learning



Our aim is for the robot to learn to always travel to state 8 on the state sheet from any initial position. It is easy to write a script with explicit instruction for the robot, but the main point here is that the robot must learn the appropriate action sequence from only given reward feedback. Here you should implement three RL algorithms. The first two are the basic dynamic programming algorithms of value iteration and policy iteration. Note that you can assume at this point that the robot has full knowledge of the environment so that the robot can find the solution by 'contemplating' about the problem. However, the robot must be able to execute the final policy.

The third algorithm that you should implement for this specific example is the temporal difference (TD) learning algorithm. This should be a full online implementation in which the robot actively explores the space.

11.3.2 Wall Avoider Robot Using Reinforcement Learning

The goal of this experiment is to teach the NXT robot to avoid walls. Use the Tribot similar with an ultrasonic sensor and a touch sensor mounted at the front. The ultrasonic sensor should be mounted to the third motor so that the robot can look around. An example is shown in Fig.11.2. Write a program so that the robot learns to avoid bumping

Fig. 11.2 Tribot configuration for the wall avoidance experiment.

by giving negative feedback when it hits an obstacle.

11.4 POMDP

With the introduction of a probability map, the POMDP can be mapped on a MDP

11.5 Model-based RL

In all of the above discussions we have assumed a discrete state space such as a chain or a grid. Of course, in practice, we might have a continuous state space, such as the position of a robot arm or a mobile robot in the environment. While discretizing the state space is a common and sometimes sufficient approach, it can also be part of the reason behind the curse of dimensionality since a increasing the resolution of the discretization will increase the number of states exponentially. We are now discussing model-based methods to overcome these problems and to make reinforcement learning applicable to a wider application area.

The idea behind this section is similar to the distinction between the histogram-based and model-based methods for approximating a pdf. The histogram method makes discrete bins and estimates the probability of each bin by averaging over examples. In contrast, a model-based approach makes a hypothesis in form of a parameterized function and estimates the parameters from examples. Thus, the later approach can be applied by making an hypothesis of the functional form of the predicted value at a specific time, $V_t(\mathbf{x}_t)$, from input $\mathbf{x}_t$ at time t,

$$V_t(\mathbf{x}_t) = f_t(\mathbf{x}_t; \theta). \tag{11.9}$$

This is equivalent to supervised learning, and we can use the same methods for learning the parameters from data such as maximum likelihood estimation. Also, similar to the different approaches in supervised learning, we could build very specific hypothesis for a specific problem or use hypothesis that are very general. While the later approach might suffer from a large number of parameters compared to the first method, we will follow this line here as it is more universally applicable.

A basic method of adjusting the weights is using a gradient-descent methods on an objective function. We will here consider the popular MSE[10], for which the gradient-descent rule is given by

$$\Delta\theta = \sum_t \alpha(r - V_t)\frac{\partial f}{\partial \mathbf{x}_t}. \tag{11.10}$$

We considered here the total change of the weights for a whole episode by summing the errors for each time step. However, one specific difference of the situation here compared to the supervised-learning examples before is that the reward is typically only received after several time steps in the future at the end of an episode. One approach is to keep a history of our predictions and make the changes for the whole episode only after the reward is received at the end of the episode. However, we can make incremental (online) updates by following the approach of temporal difference learning and replacing the supervision signal for a particular time step by the prediction of the value of the next time step,

$$\Delta_t\theta = \alpha(V_{t+1} - V_t)\sum_{k=1}^{t}\frac{\partial f_k}{\partial x}. \tag{11.11}$$

The summed change for the parameters over the whole episode is the same as from equation 11.10. We still have to keep a memory of all the gradients from the previous time steps, or at least a running sum of these gradients.

While the rules 11.10 and 11.11 are equivalent, we also introduce here some modified rules suggested by Richard Sutton. In particular, we can weight recent gradients more than gradients in the more remote past by introducing a decay factor $0 \leq \lambda \leq 1$. The rule above correspond to $\lambda = 1$ and is thus called the **TD(1) rule**. Correspondingly, the general **TD(λ) rule** is given by

$$\Delta_t\theta = \alpha(V_{t+1} - V_t)\sum_{k=1}^{t}\lambda^{t-k}\frac{\partial f_k}{\partial x}. \tag{11.12}$$

If we take the extreme of $\lambda = 0$, then the **TD(0) rule** is given by

$$\Delta_t\theta = \alpha(V_{t+1} - V_t)\frac{\partial f_t}{\partial x}. \tag{11.13}$$

Note that this rule gives different results to the supervised learning rule TD(1), but this rule is local in time and does not require any memory.

[10] As discussed in section **??**, this is appropriate for Gaussian data

11.6 Free-energy-based reinforcement learning

$$E \propto \left(y - \theta^T \mathbf{x}\right)^2 \tag{11.14}$$

$$E \propto \left(y - h(\mathbf{x}; \theta)\right)^2 \tag{11.15}$$

$$E \propto \left(y - \theta^T \phi(\mathbf{x})\right)^2 \tag{11.16}$$

$$E \propto \alpha_i \alpha_j y_i y_j \mathbf{x}^T \mathbf{x} + \text{constraints} \tag{11.17}$$

$$E \propto \alpha_i \alpha_j y_i y_j \phi(\mathbf{x})^T \phi(\mathbf{x}) + \text{constraints} \tag{11.18}$$

$$K(x, x) \tag{11.19}$$

$$P(B, E, A, J, M) = P(B)P(E)P(A|B, E)P(J|A)P(M|A) \tag{11.20}$$

$$= P(B)P(E|B)P(A|B, E)P(J|B, E, A)P(M|B, E, A, J) \tag{11.21}$$