20 STATISTICAL LEARNING METHODS

In which we view learning as a form of uncertain reasoning from observations.

Part V pointed out the prevalence of uncertainty in real environments. Agents can handle uncertainty by using the methods of probability and decision theory, but first they must learn their probabilistic theories of the world from experience. This chapter explains how they can do that. We will see how to formulate the learning task itself as a process of probabilistic inference (Section 20.1). We will see that a Bayesian view of learning is extremely powerful, providing general solutions to the problems of noise, overfitting, and optimal prediction. It also takes into account the fact that a less-than-omniscient agent can never be certain about which theory of the world is correct, yet must still make decisions by using some theory of the world.

We describe methods for learning probability models—primarily Bayesian networks—in Sections 20.2 and 20.3. Section 20.4 looks at learning methods that store and recall specific instances. Section 20.5 covers **neural network** learning and Section 20.6 introduces **kernel machines**. Some of the material in this chapter is fairly mathematical (requiring a basic understanding of multivariate calculus), although the general lessons can be understood without plunging into the details. It may benefit the reader at this point to review the material in Chapters 13 and 14 and to peek at the mathematical background in Appendix A.

20.1 STATISTICAL LEARNING

The key concepts in this chapter, just as in Chapter 18, are **data** and **hypotheses**. Here, the data are **evidence**—that is, instantiations of some or all of the random variables describing the domain. The hypotheses are probabilistic theories of how the domain works, including logical theories as a special case.

Let us consider a *very* simple example. Our favorite Surprise candy comes in two flavors: cherry (yum) and lime (ugh). The candy manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor. The candy is sold in very large bags, of which there are known to be five kinds—again, indistinguishable from the outside:

h_1: 100% cherry
h_2: 75% cherry + 25% lime
h_3: 50% cherry + 50% lime
h_4: 25% cherry + 75% lime
h_5: 100% lime

Given a new bag of candy, the random variable H (for *hypothesis*) denotes the type of the bag, with possible values h_1 through h_5. H is not directly observable, of course. As the pieces of candy are opened and inspected, data are revealed—D_1, D_2, ..., D_N, where each D_i is a random variable with possible values *cherry* and *lime*. The basic task faced by the agent is to predict the flavor of the next piece of candy.[1] Despite its apparent triviality, this scenario serves to introduce many of the major issues. The agent really does need to infer a theory of its world, albeit a very simple one.

BAYESIAN LEARNING

Bayesian learning simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis. That is, the predictions are made by using *all* the hypotheses, weighted by their probabilities, rather than by using just a single "best" hypothesis. In this way, learning is reduced to probabilistic inference. Let **D** represent all the data, with observed value **d**; then the probability of each hypothesis is obtained by Bayes' rule:

$$P(h_i|\mathbf{d}) = \alpha P(\mathbf{d}|h_i)P(h_i)\ . \tag{20.1}$$

Now, suppose we want to make a prediction about an unknown quantity X. Then we have

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|\mathbf{d}, h_i)\mathbf{P}(h_i|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)P(h_i|\mathbf{d})\ , \tag{20.2}$$

where we have assumed that each hypothesis determines a probability distribution over X. This equation shows that predictions are weighted averages over the predictions of the individual hypotheses. The hypotheses themselves are essentially "intermediaries" between the raw data and the predictions. The key quantities in the Bayesian approach are the **hypothesis prior**, $P(h_i)$, and the **likelihood** of the data under each hypothesis, $P(\mathbf{d}|h_i)$.

HYPOTHESIS PRIOR

LIKELIHOOD

For our candy example, we will assume for the time being that the prior distribution over $h_1, \ldots, h_5$ is given by $\langle 0.1, 0.2, 0.4, 0.2, 0.1\rangle$, as advertised by the manufacturer. The likelihood of the data is calculated under the assumption that the observations are **i.i.d.**—that is, independently and identically distributed—so that

I.I.D.

$$P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i)\ . \tag{20.3}$$

For example, suppose the bag is really an all-lime bag (h_5) and the first 10 candies are all lime; then $P(\mathbf{d}|h_3)$ is 0.5^{10}, because half the candies in an h_3 bag are lime.[2] Figure 20.1(a) shows how the posterior probabilities of the five hypotheses change as the sequence of 10 lime candies is observed. Notice that the probabilities start out at their prior values, so h_3 is initially the most likely choice and remains so after 1 lime candy is unwrapped. After 2

[1] Statistically sophisticated readers will recognize this scenario as a variant of the **urn-and-ball** setup. We find urns and balls less compelling than candy; furthermore, candy lends itself to other tasks, such as deciding whether to trade the bag with a friend—see Exercise 20.3.

[2] We stated earlier that the bags of candy are very large; otherwise, the i.i.d. assumption fails to hold. Technically, it is more correct (but less hygienic) to rewrap each candy after inspection and return it to the bag.

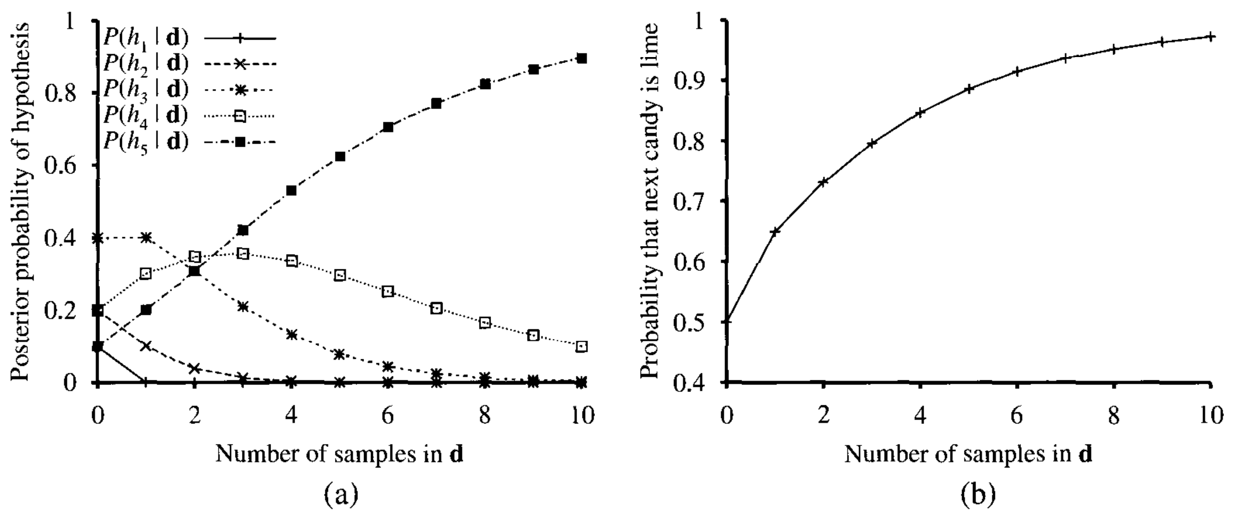


Figure 20.1 (a) Posterior probabilities $P(h_i|d_1,\ldots,d_N)$ from Equation (20.1). The number of observations N ranges from 1 to 10, and each observation is of a lime candy. (b) Bayesian prediction $P(d_{N+1} = lime|d_1,\ldots,d_N)$ from Equation (20.2).

lime candies are unwrapped, h_4 is most likely; after 3 or more, h_5 (the dreaded all-lime bag) is the most likely. After 10 in a row, we are fairly certain of our fate. Figure 20.1(b) shows the predicted probability that the next candy is lime, based on Equation (20.2). As we would expect, it increases monotonically toward 1.

The example shows that *the true hypothesis eventually dominates the Bayesian prediction.* This is characteristic of Bayesian learning. For any fixed prior that does not rule out the true hypothesis, the posterior probability of any false hypothesis will eventually vanish, simply because the probability of generating "uncharacteristic" data indefinitely is vanishingly small. (This point is analogous to one made in the discussion of PAC learning in Chapter 18.) More importantly, the Bayesian prediction is *optimal*, whether the data set be small or large. Given the hypothesis prior, any other prediction will be correct less often.

The optimality of Bayesian learning comes at a price, of course. For real learning problems, the hypothesis space is usually very large or infinite, as we saw in Chapter 18. In some cases, the summation in Equation (20.2) (or integration, in the continuous case) can be carried out tractably, but in most cases we must resort to approximate or simplified methods.

MAXIMUM A POSTERIORI

A very common approximation—one that is usually adopted in science—is to make predictions based on a single *most probable* hypothesis—that is, an h_i that maximizes $P(h_i|\mathbf{d})$. This is often called a **maximum a posteriori** or MAP (pronounced "em-ay-pee") hypothesis. Predictions made according to an MAP hypothesis h_{MAP} are approximately Bayesian to the extent that $\mathbf{P}(X|\mathbf{d}) \approx \mathbf{P}(X|h_{\mathrm{MAP}})$. In our candy example, $h_{\mathrm{MAP}} = h_5$ after three lime candies in a row, so the MAP learner then predicts that the fourth candy is lime with probability 1.0—a much more dangerous prediction than the Bayesian prediction of 0.8 shown in Figure 20.1. As more data arrive, the MAP and Bayesian predictions become closer, because the competitors to the MAP hypothesis become less and less probable. Although our example doesn't show it, finding MAP hypotheses is often much easier than Bayesian learning, because it requires solving an optimization problem instead of a large summation (or integration) problem. We will see examples of this later in the chapter.

In both Bayesian learning and MAP learning, the hypothesis prior $P(h_i)$ plays an important role. We saw in Chapter 18 that **overfitting** can occur when the hypothesis space is too expressive, so that it contains many hypotheses that fit the data set well. Rather than placing an arbitrary limit on the hypotheses to be considered, Bayesian and MAP learning methods use the prior to *penalize complexity*. Typically, more complex hypotheses have a lower prior probability—in part because there are usually many more complex hypotheses than simple hypotheses. On the other hand, more complex hypotheses have a greater capacity to fit the data. (In the extreme case, a lookup table can reproduce the data exactly with probability 1.) Hence, the hypothesis prior embodies a trade-off between the complexity of a hypothesis and its degree of fit to the data.

We can see the effect of this trade-off most clearly in the logical case, where H contains only *deterministic* hypotheses. In that case, $P(\mathbf{d}|h_i)$ is 1 if h_i is consistent and 0 otherwise. Looking at Equation (20.1), we see that h_{MAP} will then be the *simplest logical theory that is consistent with the data.* Therefore, maximum *a posteriori* learning provides a natural embodiment of Ockham's razor.

Another insight into the trade-off between complexity and degree of fit is obtained by taking the logarithm of Equation (20.1). Choosing h_{MAP} to maximize $P(\mathbf{d}|h_i)P(h_i)$ is equivalent to minimizing

$$-\log_2 P(\mathbf{d}|h_i) - \log_2 P(h_i)\ .$$

MINIMUM DESCRIPTION LENGTH

Using the connection between information encoding and probability that we introduced in Chapter 18, we see that the $-\log_2 P(h_i)$ term equals the number of bits required to specify the hypothesis h_i. Furthermore, $-\log_2 P(\mathbf{d}|h_i)$ is the additional number of bits required to specify the data, given the hypothesis. (To see this, consider that no bits are required if the hypothesis predicts the data exactly—as with h_5 and the string of lime candies—and $\log_2 1 = 0$.) Hence, MAP learning is choosing the hypothesis that provides maximum *compression* of the data. The same task is addressed more directly by the **minimum description length**, or MDL, learning method, which attempts to minimize the size of hypothesis and data encodings rather than work with probabilities.

MAXIMUM-LIKELIHOOD

A final simplification is provided by assuming a **uniform** prior over the space of hypotheses. In that case, MAP learning reduces to choosing an h_i that maximizes $P(\mathbf{d}|H_i)$. This is called a **maximum-likelihood** (ML) hypothesis, h_{ML}. Maximum-likelihood learning is very common in statistics, a discipline in which many researchers distrust the subjective nature of hypothesis priors. It is a reasonable approach when there is no reason to prefer one hypothesis over another *a priori*—for example, when all hypotheses are equally complex. It provides a good approximation to Bayesian and MAP learning when the data set is large, because the data swamps the prior distribution over hypotheses, but it has problems (as we shall see) with small data sets.

20.2 LEARNING WITH COMPLETE DATA

PARAMETER LEARNING

COMPLETE DATA

Our development of statistical learning methods begins with the simplest task: **parameter learning** with **complete data**. A parameter learning task involves finding the numerical parameters for a probability model whose structure is fixed. For example, we might be interested in learning the conditional probabilities in a Bayesian network with a given structure. Data are complete when each data point contains values for every variable in the probability model being learned. Complete data greatly simplify the problem of learning the parameters of a complex model. We will also look briefly at the problem of learning structure.

Maximum-likelihood parameter learning: Discrete models

Suppose we buy a bag of lime and cherry candy from a new manufacturer whose lime–cherry proportions are completely unknown—that is, the fraction could be anywhere between 0 and 1. In that case, we have a continuum of hypotheses. The **parameter** in this case, which we call θ, is the proportion of cherry candies, and the hypothesis is h_θ. (The proportion of limes is just $1 - \theta$.) If we assume that all proportions are equally likely *a priori*, then a maximum-likelihood approach is reasonable. If we model the situation with a Bayesian network, we need just one random variable, *Flavor* (the flavor of a randomly chosen candy from the bag). It has values *cherry* and *lime*, where the probability of *cherry* is θ (see Figure 20.2(a)). Now suppose we unwrap N candies, of which c are cherries and $\ell = N - c$ are limes. According to Equation (20.3), the likelihood of this particular data set is

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^{N} P(d_j|h_\theta) = \theta^c \cdot (1-\theta)^\ell .$$

LOG LIKELIHOOD

The maximum-likelihood hypothesis is given by the value of θ that maximizes this expression. The same value is obtained by maximizing the **log likelihood**,

$$L(\mathbf{d}|h_\theta) = \log P(\mathbf{d}|h_\theta) = \sum_{j=1}^{N} \log P(d_j|h_\theta) = c\log\theta + \ell\log(1-\theta) .$$

(By taking logarithms, we reduce the product to a sum over the data, which is usually easier to maximize.) To find the maximum-likelihood value of θ, we differentiate L with respect to θ and set the resulting expression to zero:

$$\frac{dL(\mathbf{d}|h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \qquad \Rightarrow \qquad \theta = \frac{c}{c+\ell} = \frac{c}{N} .$$

In English, then, the maximum-likelihood hypothesis h_{ML} asserts that the actual proportion of cherries in the bag is equal to the observed proportion in the candies unwrapped so far!

It appears that we have done a lot of work to discover the obvious. In fact, though, we have laid out one standard method for maximum-likelihood parameter learning:

1. Write down an expression for the likelihood of the data as a function of the parameter(s).
2. Write down the derivative of the log likelihood with respect to each parameter.
3. Find the parameter values such that the derivatives are zero.

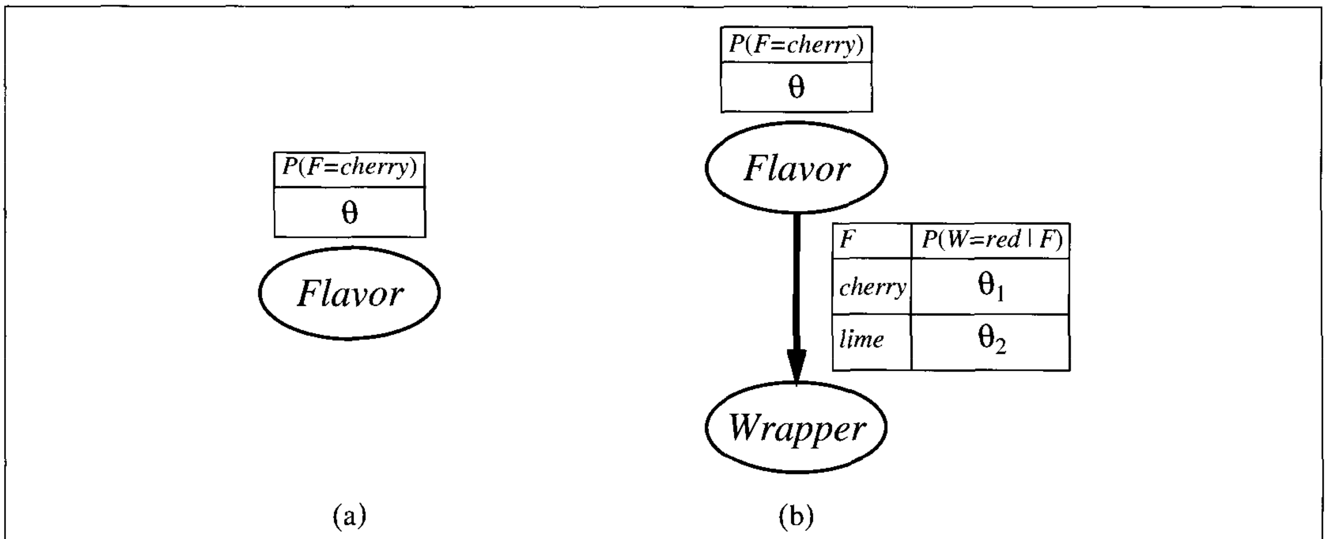


Figure 20.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherries and limes. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

The trickiest step is usually the last. In our example, it was trivial, but we will see that in many cases we need to resort to iterative solution algorithms or other numerical optimization techniques, as described in Chapter 4. The example also illustrates a significant problem with maximum-likelihood learning in general: *when the data set is small enough that some events have not yet been observed—for instance, no cherry candies—the maximum likelihood hypothesis assigns zero probability to those events.* Various tricks are used to avoid this problem, such as initializing the counts for each event to 1 instead of zero.

Let us look at another example. Suppose this new candy manufacturer wants to give a little hint to the consumer and uses candy wrappers colored red and green. The *Wrapper* for each candy is selected *probabilistically*, according to some unknown conditional distribution, depending on the flavor. The corresponding probability model is shown in Figure 20.2(b). Notice that it has three parameters: θ, θ_1, and θ_2. With these parameters, the likelihood of seeing, say, a cherry candy in a green wrapper can be obtained from the standard semantics for Bayesian networks (page 495):

$$\begin{aligned}&P(Flavor = cherry,\ Wrapper = green | h_{\theta,\theta_1,\theta_2})\\ &= P(Flavor = cherry | h_{\theta,\theta_1,\theta_2}) P(Wrapper = green | Flavor = cherry, h_{\theta,\theta_1,\theta_2})\\ &= \theta \cdot (1-\theta_1)\,.\end{aligned}$$

Now, we unwrap N candies, of which c are cherries and ℓ are limes. The wrapper counts are as follows: r_c of the cherries have red wrappers and g_c have green, while r_ℓ of the limes have red and g_ℓ have green. The likelihood of the data is given by

$$P(\mathbf{d}|h_{\theta,\theta_1,\theta_2}) = \theta^c(1-\theta)^\ell \cdot \theta_1^{r_c}(1-\theta_1)^{g_c} \cdot \theta_2^{r_\ell}(1-\theta_2)^{g_\ell}\,.$$

This looks pretty horrible, but taking logarithms helps:

$$L = [c\log\theta + \ell\log(1-\theta)] + [r_c\log\theta_1 + g_c\log(1-\theta_1)] + [r_\ell\log\theta_2 + g_\ell\log(1-\theta_2)]\,.$$

The benefit of taking logs is clear: the log likelihood is the sum of three terms, each of which contains a single parameter. When we take derivatives with respect to each parameter and set them to zero, we get three independent equations, each containing just one parameter:

$$\begin{array}{lll}
\frac{\partial L}{\partial \theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 & \Rightarrow & \theta = \frac{c}{c+\ell} \\
\frac{\partial L}{\partial \theta_1} = \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 & \Rightarrow & \theta_1 = \frac{r_c}{r_c+g_c} \\
\frac{\partial L}{\partial \theta_2} = \frac{r_\ell}{\theta_2} - \frac{g_\ell}{1-\theta_2} = 0 & \Rightarrow & \theta_2 = \frac{r_\ell}{r_\ell+g_\ell} \,.
\end{array}$$

The solution for θ is the same as before. The solution for θ_1, the probability that a cherry candy has a red wrapper, is the observed fraction of cherry candies with red wrappers, and similarly for θ_2.

These results are very comforting, and it is easy to see that they can be extended to any Bayesian network whose conditional probabilities are represented as tables. The most important point is that, *with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parameter.*[3] The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.

Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the **naive Bayes** model. In this model, the "class" variable C (which is to be predicted) is the root and the "attribute" variables X_i are the leaves. The model is "naive" because it assumes that the attributes are conditionally independent of each other, given the class. (The model in Figure 20.2(b) is a naive Bayes model with just one attribute.) Assuming Boolean variables, the parameters are

$$\theta = P(C = true), \theta_{i1} = P(X_i = true|C = true), \theta_{i2} = P(X_i = true|C = false).$$

The maximum-likelihood parameter values are found in exactly the same way as for Figure 20.2(b). Once the model has been trained in this way, it can be used to classify new examples for which the class variable C is unobserved. With observed attribute values $x_1, \ldots, x_n$, the probability of each class is given by

$$\mathbf{P}(C|x_1, \ldots, x_n) = \alpha\, \mathbf{P}(C) \prod_i \mathbf{P}(x_i|C)\,.$$

A deterministic prediction can be obtained by choosing the most likely class. Figure 20.3 shows the learning curve for this method when it is applied to the restaurant problem from Chapter 18. The method learns fairly well but not as well as decision-tree learning; this is presumably because the true hypothesis—which is a decision tree—is not representable exactly using a naive Bayes model. Naive Bayes learning turns out to do surprisingly well in a wide range of applications; the boosted version (Exercise 20.5) is one of the most effective general-purpose learning algorithms. Naive Bayes learning scales well to very large problems: with n Boolean attributes, there are just $2n + 1$ parameters, and *no search is required to find h_{ML}, the maximum-likelihood naive Bayes hypothesis.* Finally, naive Bayes learning has no difficulty with noisy data and can give probabilistic predictions when appropriate.

[3] See Exercise 20.7 for the nontabulated case, where each parameter affects several conditional probabilities.

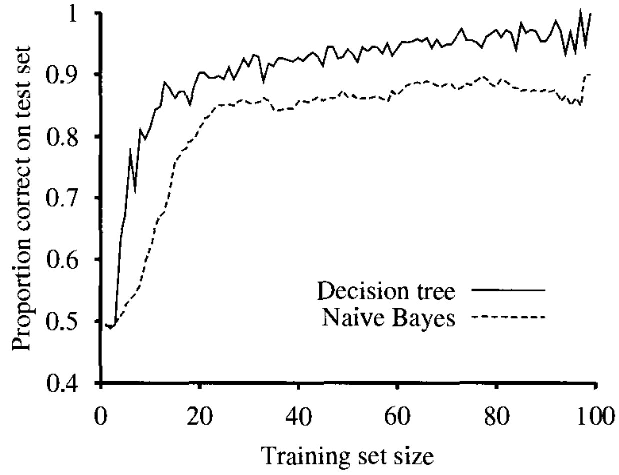


Figure 20.3 The learning curve for naive Bayes learning applied to the restaurant problem from Chapter 18; the learning curve for decision-tree learning is shown for comparison.

Maximum-likelihood parameter learning: Continuous models

Continuous probability models such as the **linear-Gaussian** model were introduced in Section 14.3. Because continuous variables are ubiquitous in real-world applications, it is important to know how to learn continuous models from data. The principles for maximum-likelihood learning are identical to those of the discrete case.

Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, the data are generated as follows:

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} .$$

The parameters of this model are the mean μ and the standard deviation σ. (Notice that the normalizing "constant" depends on σ, so we cannot ignore it.) Let the observed values be $x_1, \ldots, x_N$. Then the log likelihood is

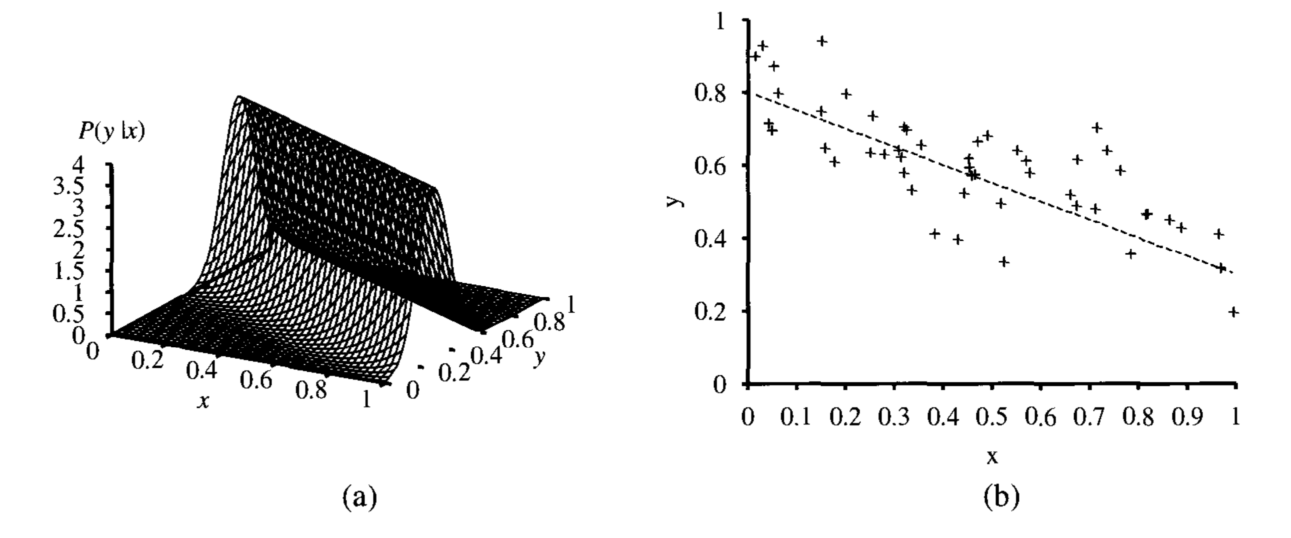
$$L = \sum_{j=1}^{N} \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} = N(-\log\sqrt{2\pi} - \log\sigma) - \sum_{j=1}^{N} \frac{(x_j-\mu)^2}{2\sigma^2} .$$

Setting the derivatives to zero as usual, we obtain

$$\begin{aligned} \frac{\partial L}{\partial \mu} &= -\frac{1}{\sigma^2}\sum_{j=1}^{N}(x_j-\mu) = 0 & \Rightarrow \quad & \mu = \frac{\sum_j x_j}{N} \\ \frac{\partial L}{\partial \sigma} &= -\frac{N}{\sigma} + \frac{1}{\sigma^3}\sum_{j=1}^{N}(x_j-\mu)^2 = 0 & \Rightarrow \quad & \sigma = \sqrt{\frac{\sum_j (x_j-\mu)^2}{N}} . \end{aligned} \tag{20.4}$$

That is, the maximum-likelihood value of the mean is the sample average and the maximum-likelihood value of the standard deviation is the square root of the sample variance. Again, these are comforting results that confirm "commonsense" practice.

Now consider a linear Gaussian model with one continuous parent X and a continuous child Y. As explained on page 502, Y has a Gaussian distribution whose mean depends linearly on the value of X and whose standard deviation is fixed. To learn the conditional



(a)

(b)

Figure 20.4 (a) A linear Gaussian model described as $y = \theta_1 x + \theta_2$ plus Gaussian noise with fixed variance. (b) A set of 50 data points generated from this model.

distribution $P(Y|X)$, we can maximize the conditional likelihood

$$P(y|x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-(\theta_1 x+\theta_2))^2}{2\sigma^2}} . \tag{20.5}$$

Here, the parameters are θ_1, θ_2, and σ. The data are a collection of (x_j, y_j) pairs, as illustrated in Figure 20.4. Using the usual methods (Exercise 20.6), we can find the maximum-likelihood values of the parameters. Here, we want to make a different point. If we consider just the parameters θ_1 and θ_2 that define the linear relationship between x and y, it becomes clear that maximizing the log likelihood with respect to these parameters is the same as *minimizing* the numerator in the exponent of Equation (20.5):

$$E = \sum_{j=1}^{N} (y_j - (\theta_1 x_j + \theta_2))^2 .$$

ERROR

SUM OF SQUARED ERRORS

LINEAR REGRESSION

The quantity $(y_j - (\theta_1 x_j + \theta_2))$ is the **error** for (x_j, y_j)—that is, the difference between the actual value y_j and the predicted value $(\theta_1 x_j + \theta_2)$—so E is the well-known **sum of squared errors**. This is the quantity that is minimized by the standard **linear regression** procedure. Now we can understand why: minimizing the sum of squared errors gives the maximum-likelihood straight-line model, *provided that the data are generated with Gaussian noise of fixed variance.*

Bayesian parameter learning

Maximum-likelihood learning gives rise to some very simple procedures, but it has some serious deficiencies with small data sets. For example, after seeing one cherry candy, the maximum-likelihood hypothesis is that the bag is 100% cherry (i.e., $\theta = 1.0$). Unless one's hypothesis prior is that bags must be either all cherry or all lime, this is not a reasonable conclusion. The Bayesian approach to parameter learning places a hypothesis prior over the possible values of the parameters and updates this distribution as data arrive.

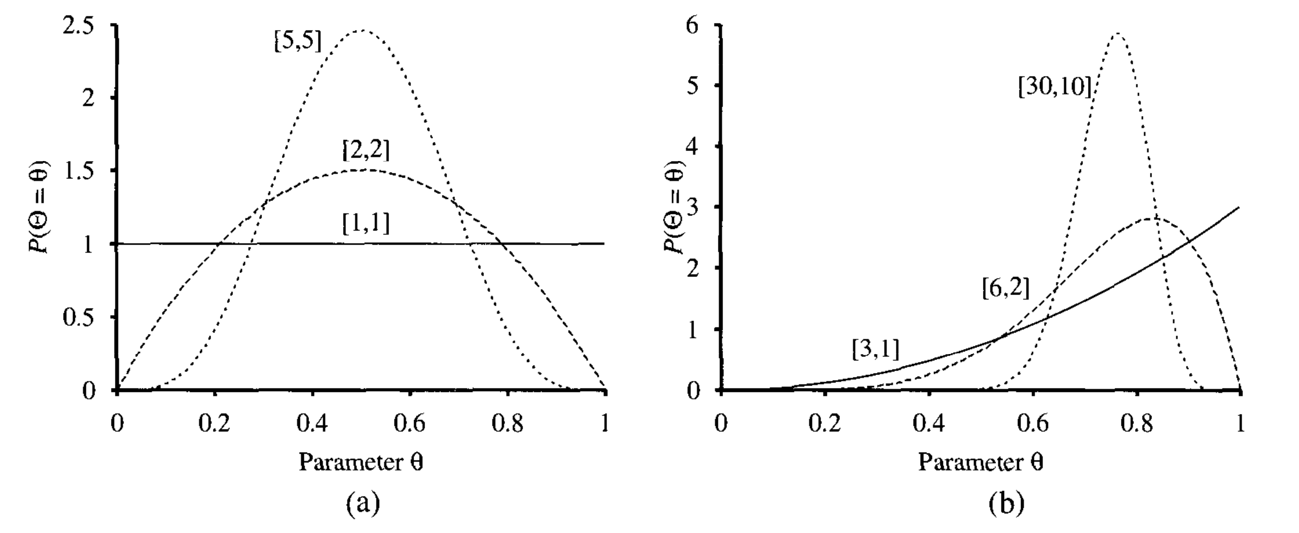


Figure 20.5 Examples of the beta$[a, b]$ distribution for different values of $[a, b]$.

The candy example in Figure 20.2(a) has one parameter, θ: the probability that a randomly selected piece of candy is cherry flavored. In the Bayesian view, θ is the (unknown) value of a random variable Θ; the hypothesis prior is just the prior distribution $\mathbf{P}(\Theta)$. Thus, $P(\Theta = \theta)$ is the prior probability that the bag has a fraction θ of cherry candies.

If the parameter θ can be any value between 0 and 1, then $\mathbf{P}(\Theta)$ must be a continuous distribution that is nonzero only between 0 and 1 and that integrates to 1. The uniform density $P(\theta) = U[0, 1](\theta)$ is one candidate. (See Chapter 13.) It turns out that the uniform density is a member of the family of **beta distributions**. Each beta distribution is defined by two **hyperparameters**[4] a and b such that

BETA DISTRIBUTIONS

HYPERPARAMETER

$$\text{beta}[a, b](\theta) = \alpha\, \theta^{a-1}(1-\theta)^{b-1}\,, \tag{20.6}$$

for θ in the range $[0, 1]$. The normalization constant α depends on a and b. (See Exercise 20.8.) Figure 20.5 shows what the distribution looks like for various values of a and b. The mean value of the distribution is $a/(a + b)$, so larger values of a suggest a belief that Θ is closer to 1 than to 0. Larger values of $a + b$ make the distribution more peaked, suggesting greater certainty about the value of Θ. Thus, the beta family provides a useful range of possibilities for the hypothesis prior.

Besides its flexibility, the beta family has another wonderful property: if Θ has a prior beta$[a, b]$, then, after a data point is observed, the posterior distribution for Θ is also a beta distribution. The beta family is called the **conjugate prior** for the family of distributions for a Boolean variable.[5] Let's see how this works. Suppose we observe a cherry candy; then

CONJUGATE PRIOR

$$\begin{aligned} P(\theta|D_1 = cherry) &= \alpha\, P(D_1 = cherry|\theta)P(\theta) \\ &= \alpha'\, \theta \cdot \text{beta}[a, b](\theta) = \alpha'\, \theta \cdot \theta^{a-1}(1-\theta)^{b-1} \\ &= \alpha'\, \theta^{a}(1-\theta)^{b-1} = \text{beta}[a+1, b](\theta)\,. \end{aligned}$$

[4] They are called hyperparameters because they parameterize a distribution over θ, which is itself a parameter.

[5] Other conjugate priors include the **Dirichlet** family for the parameters of a discrete multivalued distribution and the **Normal–Wishart** family for the parameters of a Gaussian distribution. See Bernardo and Smith (1994).

VIRTUAL COUNTS

Thus, after seeing a cherry candy, we simply increment the a parameter to get the posterior; similarly, after seeing a lime candy, we increment the b parameter. Thus, we can view the a and b hyperparameters as **virtual counts**, in the sense that a prior $\mathrm{beta}[a, b]$ behaves exactly as if we had started out with a uniform prior $\mathrm{beta}[1, 1]$ and seen $a - 1$ actual cherry candies and $b - 1$ actual lime candies.

By examining a sequence of beta distributions for increasing values of a and b, keeping the proportions fixed, we can see vividly how the posterior distribution over the parameter Θ changes as data arrive. For example, suppose the actual bag of candy is 75% cherry. Figure 20.5(b) shows the sequence $\mathrm{beta}[3, 1]$, $\mathrm{beta}[6, 2]$, $\mathrm{beta}[30, 10]$. Clearly, the distribution is converging to a narrow peak around the true value of Θ. For large data sets, then, Bayesian learning (at least in this case) converges to give the same results as maximum-likelihood learning.

PARAMETER INDEPENDENCE

The network in Figure 20.2(b) has three parameters, θ, θ_1, and θ_2, where θ_1 is the probability of a red wrapper on a cherry candy and θ_2 is the probability of a red wrapper on a lime candy. The Bayesian hypothesis prior must cover all three parameters—that is, we need to specify $\mathbf{P}(\Theta, \Theta_1, \Theta_2)$. Usually, we assume **parameter independence**:

$$\mathbf{P}(\Theta, \Theta_1, \Theta_2) = \mathbf{P}(\Theta)\mathbf{P}(\Theta_1)\mathbf{P}(\Theta_2) .$$

With this assumption, each parameter can have its own beta distribution that is updated separately as data arrive.

Once we have the idea that unknown parameters can be represented by random variables such as Θ, it is natural to incorporate them into the Bayesian network itself. To do this, we also need to make copies of the variables describing each instance. For example, if we have observed three candies then we need $Flavor_1$, $Flavor_2$, $Flavor_3$ and $Wrapper_1$, $Wrapper_2$, $Wrapper_3$. The parameter variable Θ determines the probability of each $Flavor_i$ variable:

$$P(Flavor_i = cherry | \Theta = \theta) = \theta .$$

Similarly, the wrapper probabilities depend on Θ_1 and Θ_2, For example,

$$P(Wrapper_i = red | Flavor_i = cherry, \Theta_1 = \theta_1) = \theta_1 .$$

Now, the entire Bayesian learning process can be formulated as an *inference* problem in a suitably constructed Bayes net, as shown in Figure 20.6. Prediction for a new instance is done simply by adding new instance variables to the network, some of which are queried. This formulation of learning and prediction makes it clear that Bayesian learning requires no extra "principles of learning." Furthermore, *there is, in essence, just one learning algorithm,* i.e., the inference algorithm for Bayesian networks.

Learning Bayes net structures

So far, we have assumed that the structure of the Bayes net is given and we are just trying to learn the parameters. The structure of the network represents basic causal knowledge about the domain that is often easy for an expert, or even a naive user, to supply. In some cases, however, the causal model may be unavailable or subject to dispute—for example, certain corporations have long claimed that smoking does not cause cancer—so it is important to

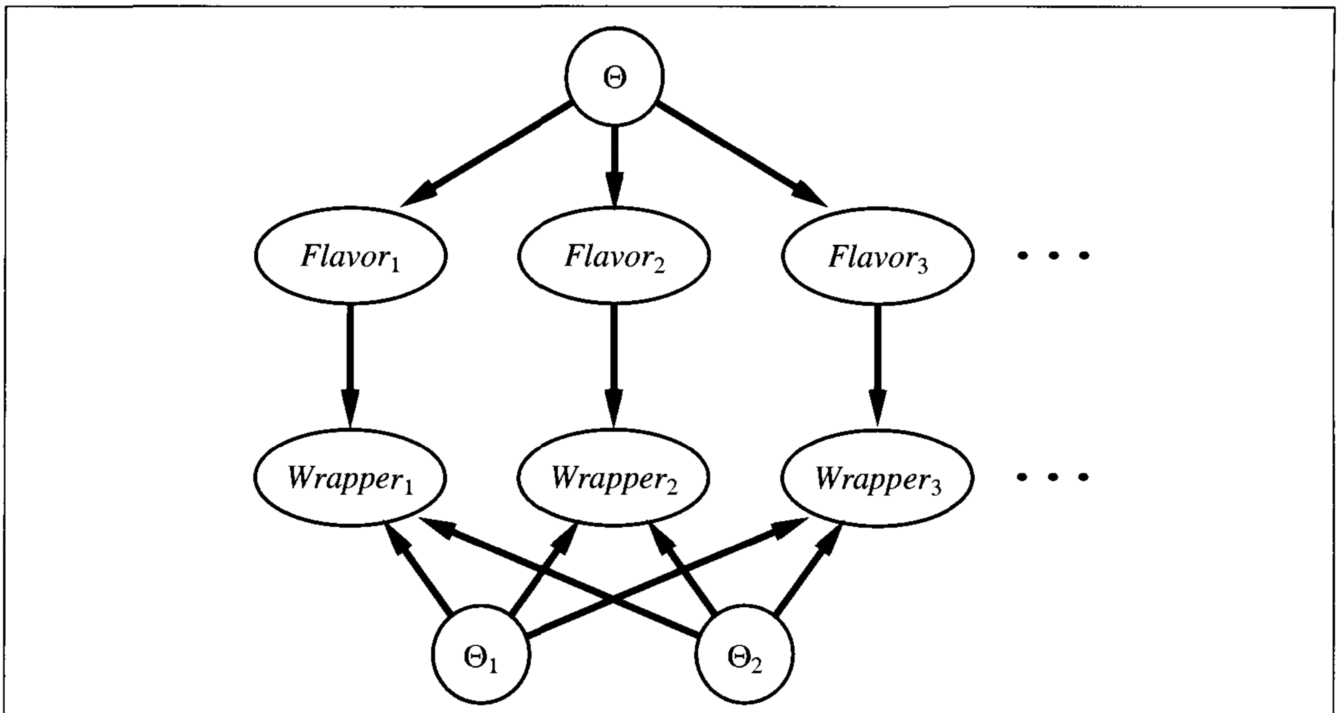


Figure 20.6 A Bayesian network that corresponds to a Bayesian learning process. Posterior distributions for the parameter variables Θ, Θ_1, and Θ_2 can be inferred from their prior distributions and the evidence in the $Flavor_i$ and $Wrapper_i$ variables.

understand how the structure of a Bayes net can be learned from data. At present, structural learning algorithms are in their infancy, so we will give only a brief sketch of the main ideas.

The most obvious approach is to *search* for a good model. We can start with a model containing no links and begin adding parents for each node, fitting the parameters with the methods we have just covered and measuring the accuracy of the resulting model. Alternatively, we can start with an initial guess at the structure and use hill-climbing or simulated annealing search to make modifications, retuning the parameters after each change in the structure. Modifications can include reversing, adding, or deleting arcs. We must not introduce cycles in the process, so many algorithms assume that an ordering is given for the variables, and that a node can have parents only among those nodes that come earlier in the ordering (just as in the construction process Chapter 14). For full generality, we also need to search over possible orderings.

There are two alternative methods for deciding when a good structure has been found. The first is to test whether the conditional independence assertions implicit in the structure are actually satisfied in the data. For example, the use of a naive Bayes model for the restaurant problem assumes that

$$\mathbf{P}(Fri/Sat, Bar|\, WillWait) = \mathbf{P}(Fri/Sat|\, WillWait)\mathbf{P}(Bar|\, WillWait)$$

and we can check in the data that the same equation holds between the corresponding conditional frequencies. Now, even if the structure describes the true causal nature of the domain, statistical fluctuations in the data set mean that the equation will never be satisfied *exactly*, so we need to perform a suitable statistical test to see if there is sufficient evidence that the independence hypothesis is violated. The complexity of the resulting network will depend on the threshold used for this test—the stricter the independence test, the more links will be added and the greater the danger of overfitting.

An approach more consistent with the ideas in this chapter is to the degree to which the proposed model explains the data (in a probabilistic sense). We must be careful how we measure this, however. If we just try to find the maximum-likelihood hypothesis, we will end up with a fully connected network, because adding more parents to a node cannot decrease the likelihood (Exercise 20.9). We are forced to penalize model complexity in some way. The MAP (or MDL) approach simply subtracts a penalty from the likelihood of each structure (after parameter tuning) before comparing different structures. The Bayesian approach places a joint prior over structures and parameters. There are usually far too many structures to sum over (superexponential in the number of variables), so most practitioners use MCMC to sample over structures.

Penalizing complexity (whether by MAP or Bayesian methods) introduces an important connection between the optimal structure and the nature of the representation for the conditional distributions in the network. With tabular distributions, the complexity penalty for a node's distribution grows exponentially with the number of parents, but with, say, noisy-OR distributions, it grows only linearly. This means that learning with noisy-OR (or other compactly parameterized) models tends to produce learned structures with more parents than does learning with tabular distributions.

20.3 Learning with Hidden Variables: The EM Algorithm

LATENT VARIABLES

The preceding section dealt with the fully observable case. Many real-world problems have **hidden variables** (sometimes called **latent variables**) which are not observable in the data that are available for learning. For example, medical records often include the observed symptoms, the treatment applied, and perhaps the outcome of the treatment, but they seldom contain a direct observation of the disease itself![6] One might ask, "If the disease is not observed, why not construct a model without it?" The answer appears in Figure 20.7, which shows a small, fictitious diagnostic model for heart disease. There are three observable predisposing factors and three observable symptoms (which are too depressing to name). Assume that each variable has three possible values (e.g., *none*, *moderate*, and *severe*). Removing the hidden variable from the network in (a) yields the network in (b); the total number of parameters increases from 78 to 708. Thus, *latent variables can dramatically reduce the number of parameters required to specify a Bayesian network.* This, in turn, can dramatically reduce the amount of data needed to learn the parameters.

Hidden variables are important, but they do complicate the learning problem. In Figure 20.7(a), for example, it is not obvious how to learn the conditional distribution for *HeartDisease*, given its parents, because we do not know the value of *HeartDisease* in each case; the same problem arises in learning the distributions for the symptoms. This section

[6] Some records contain the diagnosis suggested by the physician, but this is a causal consequence of the symptoms, which are in turn caused by the disease.

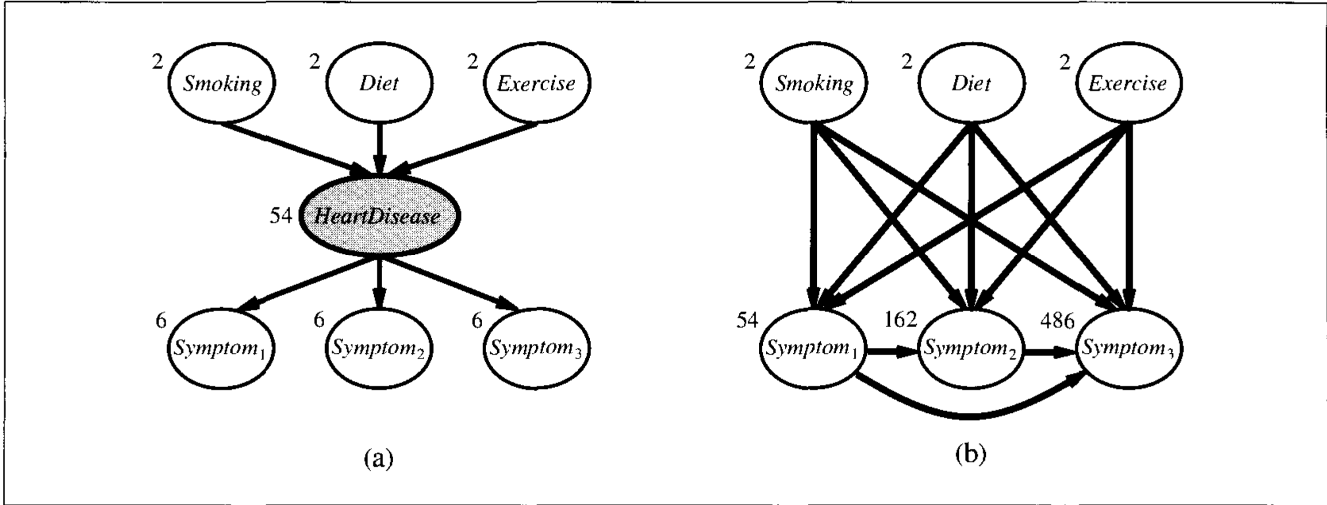


Figure 20.7 (a) A simple diagnostic network for heart disease, which is assumed to be a hidden variable. Each variable has three possible values and is labeled with the number of independent parameters in its conditional distribution; the total number is 78. (b) The equivalent network with *HeartDisease* removed. Note that the symptom variables are no longer conditionally independent given their parents. This network requires 708 parameters.

EXPECTATION-MAXIMIZATION

describes an algorithm called **expectation–maximization**, or EM, that solves this problem in a very general way. We will show three examples and then provide a general description. The algorithm seems like magic at first, but once the intuition has been developed, one can find applications for EM in a huge range of learning problems.

Unsupervised clustering: Learning mixtures of Gaussians

UNSUPERVISED CLUSTERING

Unsupervised clustering is the problem of discerning multiple categories in a collection of objects. The problem is unsupervised because the category labels are not given. For example, suppose we record the spectra of a hundred thousand stars; are there different *types* of stars revealed by the spectra, and, if so, how many and what are their characteristics? We are all familiar with terms such as "red giant" and "white dwarf," but the stars do not carry these labels on their hats—astronomers had to perform unsupervised clustering to identify these categories. Other examples include the identification of species, genera, orders, and so on in the Linnæan taxonomy of organisms and the creation of natural kinds to categorize ordinary objects (see Chapter 10).

MIXTURE DISTRIBUTION

COMPONENT

Unsupervised clustering begins with data. Figure 20.8(a) shows 500 data points, each of which specifies the values of two continuous attributes. The data points might correspond to stars, and the attributes might correspond to spectral intensities at two particular frequencies. Next, we need to understand what kind of probability distribution might have generated the data. Clustering presumes that the data are generated from a **mixture distribution**, P. Such a distribution has k **components**, each of which is a distribution in its own right. A data point is generated by first choosing a component and then generating a sample from that component. Let the random variable C denote the component, with values $1, \ldots, k$; then the mixture

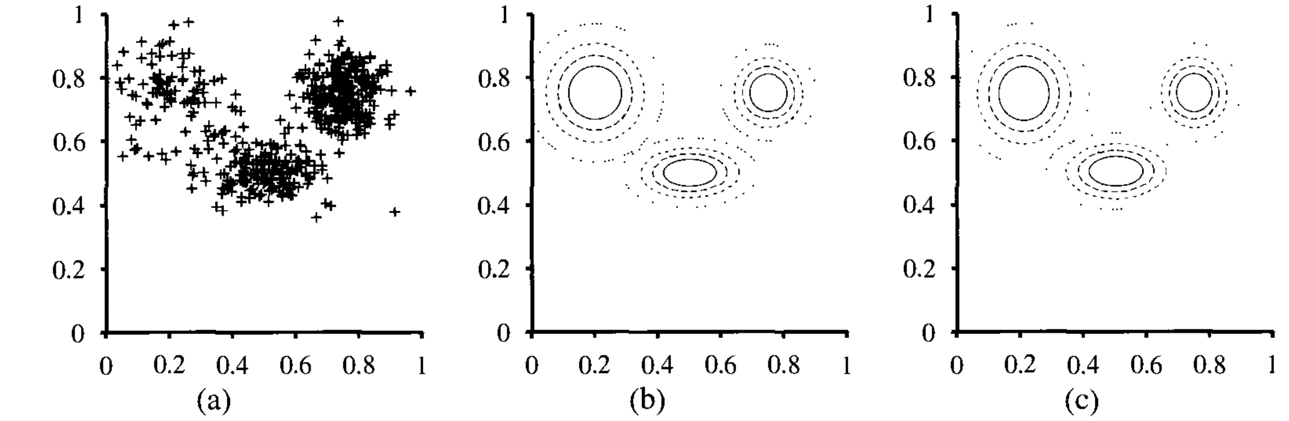


Figure 20.8 (a) 500 data points in two dimensions, suggesting the presence of three clusters. (b) A Gaussian mixture model with three components; the weights (left-to-right) are 0.2, 0.3, and 0.5. The data in (a) were generated from this model. (c) The model reconstructed by EM from the data in (b).

distribution is given by

$$P(\mathbf{x}) = \sum_{i=1}^{k} P(C=i)\, P(\mathbf{x}|C=i) ,$$

MIXTURE OF GAUSSIANS

where **x** refers to the values of the attributes for a data point. For continuous data, a natural choice for the component distributions is the multivariate Gaussian, which gives the so-called **mixture of Gaussians** family of distributions. The parameters of a mixture of Gaussians are $w_i = P(C=i)$ (the weight of each component), $\boldsymbol{\mu}_i$ (the mean of each component), and Σ_i (the covariance of each component). Figure 20.8(b) shows a mixture of three Gaussians; this mixture is in fact the source of the data in (a).

The unsupervised clustering problem, then, is to recover a mixture model like the one in Figure 20.8(b) from raw data like that in Figure 20.8(a). Clearly, if we *knew* which component generated each data point, then it would be easy to recover the component Gaussians: we could just select all the data points from a given component and then apply (a multivariate version of) Equation (20.4) for fitting the parameters of a Gaussian to a set of data. On the other hand, if we *knew* the parameters of each component, then we could, at least in a probabilistic sense, assign each data point to a component. The problem is that we know neither the assignments nor the parameters.

The basic idea of EM in this context is to *pretend* that we know the the parameters of the model and then to infer the probability that each data point belongs to each component. After that, we refit the components to the data, where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component. The process iterates until convergence. Essentially, we are "completing" the data by inferring probability distributions over the hidden variables—which component each data point belongs to—based on the current model. For the mixture of Gaussians, we initialize the mixture model parameters arbitrarily and then iterate the following two steps:

1. E-step: Compute the probabilities $p_{ij} = P(C = i|\mathbf{x}_j)$, the probability that datum $\mathbf{x}_j$ was generated by component i. By Bayes' rule, we have $p_{ij} = \alpha P(\mathbf{x}_j|C = i)P(C = i)$. The term $P(\mathbf{x}_j|C = i)$ is just the probability at $\mathbf{x}_j$ of the ith Gaussian, and the term $P(C = i)$ is just the weight parameter for the ith Gaussian. Define $p_i = \sum_j p_{ij}$.
2. M-step: Compute the new mean, covariance, and component weights as follows:

$$\begin{aligned}
\boldsymbol{\mu}_i &\leftarrow \sum_j p_{ij}\mathbf{x}_j / p_i \\
\boldsymbol{\Sigma}_i &\leftarrow \sum_j p_{ij}\mathbf{x}_j\mathbf{x}_j^{\top} / p_i \\
w_i &\leftarrow p_i \,.
\end{aligned}$$

INDICATOR VARIABLE

The E-step, or *expectation* step, can be viewed as computing the expected values p_{ij} of the hidden **indicator variables** Z_{ij}, where Z_{ij} is 1 if datum $\mathbf{x}_j$ was generated by the ith component and 0 otherwise. The M-step, or *maximization* step, finds the new values of the parameters that maximize the log likelihood of the data, given the expected values of the hidden indicator variables.

The final model that EM learns when it is applied to the data in Figure 20.8(a) is shown in Figure 20.8(c); it is virtually indistinguishable from the original model from which the data were generated. Figure 20.9(a) plots the log likelihood of the data according to the current model as EM progresses. There are two points to notice. First, the log likelihood for the final learned model slightly *exceeds* that of the original model, from which the data were generated. This might seem surprising, but it simply reflects the fact that the data were generated randomly and might not provide an exact reflection of the underlying model. The second point is that *EM increases the log likelihood of the data at every iteration.* This fact can be proved in general. Furthermore, under certain conditions, EM can be proven to reach a local maximum in likelihood. (In rare cases, it could reach a saddle point or even a local minimum.) In this sense, EM resembles a gradient-based hill-climbing algorithm, but notice that it has no "step size" parameter!

Things do not always go as well as Figure 20.9(a) might suggest. It can happen, for example, that one Gaussian component shrinks so that it covers just a single data point. Then its variance will go to zero and its likelihood will go to infinity! Another problem is that two components can "merge," acquiring identical means and variances and sharing their data points. These kinds of degenerate local maxima are serious problems, especially in high dimensions. One solution is to place priors on the model parameters and to apply the MAP version of EM. Another is to restart a component with new random parameters if it gets too small or too close to another component. It also helps to initialize the parameters with reasonable values.

Learning Bayesian networks with hidden variables

To learn a Bayesian network with hidden variables, we apply the same insights that worked for mixtures of Gaussians. Figure 20.10 represents a situation in which there are two bags of candies that have been mixed together. Candies are described by three features: in addition to the *Flavor* and the *Wrapper*, some candies have a *Hole* in the middle and some do not.

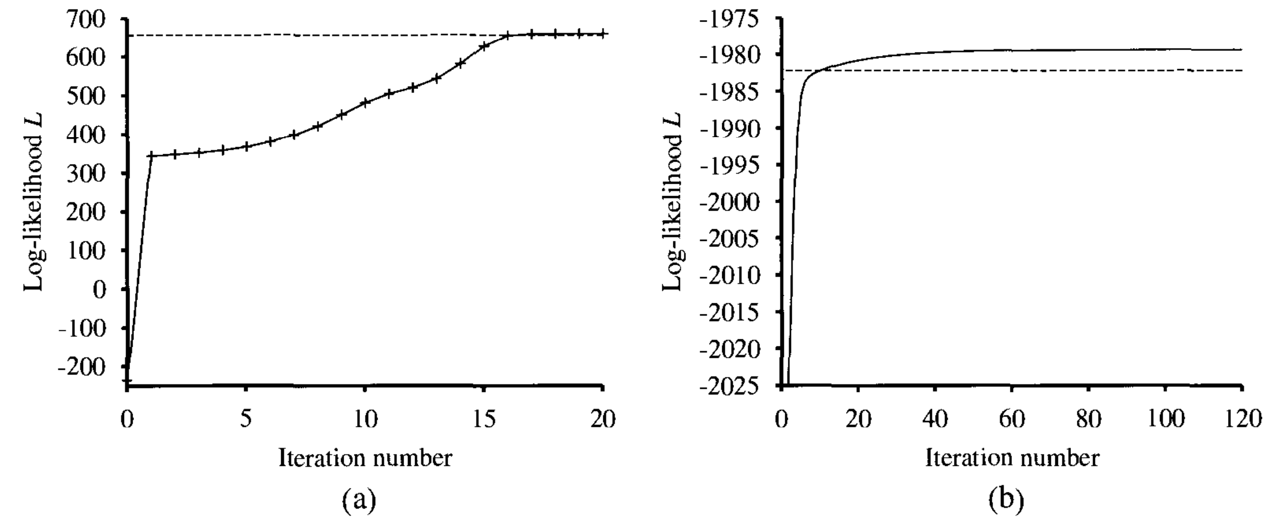


Figure 20.9 Graphs showing the log-likelihood of the data, L, as a function of the EM iteration. The horizontal line shows the log-likelihood according to the true model. (a) Graph for the Gaussian mixture model in Figure 20.8. (b) Graph for the Bayesian network in Figure 20.10(a).

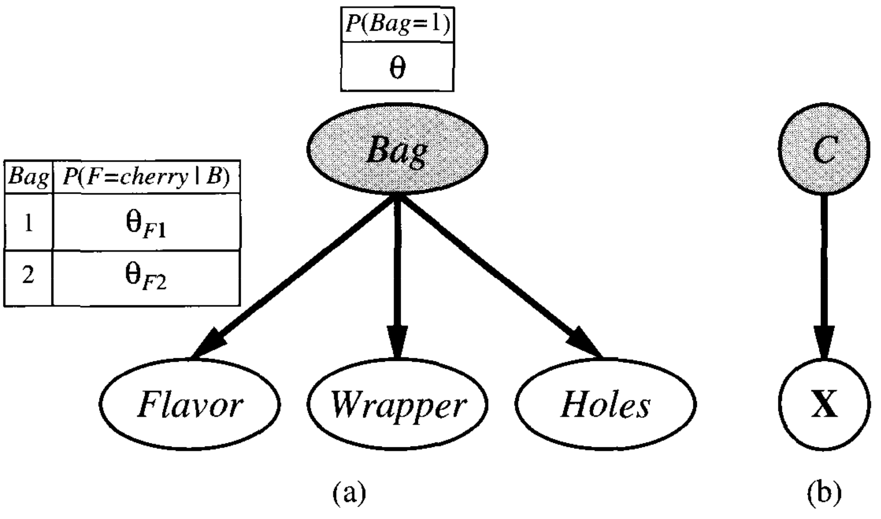


Figure 20.10 (a) A mixture model for candy. The proportions of different flavors, wrappers, and numbers of holes depend on the bag, which is not observed. (b) Bayesian network for a Gaussian mixture. The mean and covariance of the observable variables **X** depend on the component C.

The distribution of candies in each bag is described by a **naive Bayes** model: the features are independent, given the bag, but the conditional probability distribution for each feature depends on the bag. The parameters are as follows: θ is the prior probability that a candy comes from Bag 1; θ_{F1} and θ_{F2} are the probabilities that the flavor is cherry, given that the candy comes from Bag 1 and Bag 2 respectively; θ_{W1} and θ_{W2} give the probabilities that the wrapper is red; and θ_{H1} and θ_{H2} give the probabilities that the candy has a hole. Notice that the overall model is a mixture model. (In fact, we can also model the mixture of Gaussians as a Bayesian network, as shown in Figure 20.10(b).) In the figure, the bag is is a hidden variable because, once the candies have been mixed together, we no longer know which bag each candy came from. In such a case, can we recover the descriptions of the two bags by observing candies from the mixture?

Let us work through an iteration of EM for this problem. First, let's look at the data. We generated 1000 samples from a model whose true parameters are

$$\theta = 0.5,\ \theta_{F1} = \theta_{W1} = \theta_{H1} = 0.8, \theta_{F2} = \theta_{W2} = \theta_{H2} = 0.3\ . \tag{20.7}$$

That is, the candies are equally likely to come from either bag; the first is mostly cherries with red wrappers and holes; the second is mostly limes with green wrappers and no holes. The counts for the eight possible kinds of candy are as follows:

	$W = red$		$W = green$	
	$H=1$	$H=0$	$H=1$	$H=0$
$F = cherry$	273	93	104	90
$F = lime$	79	100	94	167

We start by initializing the parameters. For numerical simplicity, we will choose[7]

$$\theta^{(0)} = 0.6,\ \theta^{(0)}_{F1} = \theta^{(0)}_{W1} = \theta^{(0)}_{H1} = 0.6, \theta^{(0)}_{F2} = \theta^{(0)}_{W2} = \theta^{(0)}_{H2} = 0.4\ . \tag{20.8}$$

First, let us work on the θ parameter. In the fully observable case, we would estimate this directly from the *observed* counts of candies from bags 1 and 2. Because the bag is a hidden variable, we calculate the *expected* counts instead. The expected count $\hat{N}(Bag = 1)$ is the sum, over all candies, of the probability that the candy came from bag 1:

$$\theta^{(1)} = \hat{N}(Bag = 1)/N = \sum_{j=1}^{N} P(Bag = 1|flavor_j, wrapper_j, holes_j)/N\ .$$

These probabilities can be computed by any inference algorithm for Bayesian networks. For a naive Bayes model such as the one in our example, we can do the inference "by hand," using Bayes' rule and applying conditional independence:

$$\theta^{(1)} = \frac{1}{N}\sum_{j=1}^{N} \frac{P(flavor_j|Bag = 1)P(wrapper_j|Bag = 1)P(holes_j|Bag = 1)P(Bag = 1)}{\sum_i P(flavor_j|Bag = i)P(wrapper_j|Bag = i)P(holes_j|Bag = i)P(Bag = i)}\ .$$

(Notice that the normalizing constant also depends on the parameters.) Applying this formula to, say, the 273 red-wrapped cherry candies with holes, we get a contribution of

$$\frac{273}{1000} \cdot \frac{\theta^{(0)}_{F1}\theta^{(0)}_{W1}\theta^{(0)}_{H1}\theta^{(0)}}{\theta^{(0)}_{F1}\theta^{(0)}_{W1}\theta^{(0)}_{H1}\theta^{(0)} + \theta^{(0)}_{F2}\theta^{(0)}_{W2}\theta^{(0)}_{H2}(1-\theta^{(0)})} \approx 0.22797\ .$$

Continuing with the other seven kinds of candy in the table of counts, we obtain $\theta^{(1)} = 0.6124$.

[7] It is better in practice to choose them randomly, to avoid local maxima due to symmetry.

Now let us consider the other parameters, such as θ_{F1}. In the fully observable case, we would estimate this directly from the *observed* counts of cherry and lime candies from bag 1. The *expected* count of cherry candies from bag 1 is given by

$$\sum_{j:Flavor_j \,=\, cherry} P(Bag = 1 | Flavor_j = cherry, wrapper_j, holes_j) \ .$$

Again, these probabilities can be calculated by any Bayes net algorithm. Completing this process, we obtain the new values of all the parameters:

$$\begin{array}{l} \theta^{(1)} = 0.6124,\ \theta^{(1)}_{F1} = 0.6684,\ \theta^{(1)}_{W1} = 0.6483,\ \theta^{(1)}_{H1} = 0.6558, \\ \theta^{(1)}_{F2} = 0.3887, \theta^{(1)}_{W2} = 0.3817,\ \theta^{(1)}_{H2} = 0.3827 \ . \end{array} \tag{20.9}$$

The log likelihood of the data increases from about -2044 initially to about -2021 after the first iteration, as shown in Figure 20.9(b). That is, the update improves the likelihood itself by a factor of about $e^{23} \approx 10^{10}$. By the tenth iteration, the learned model is a better fit than the original model ($L = \ -\ 1982.214$). Thereafter, progress becomes very slow. This is not uncommon with EM, and many practical systems combine EM with a gradient-based algorithm such as Newton–Raphson (see Chapter 4) for the last phase of learning.

The general lesson from this example is that *the parameter updates for Bayesian network learning with hidden variables are directly available from the results of inference on each example. Moreover, only* local *posterior probabilities are needed for each parameter.* For the general case in which we are learning the conditional probability parameters for each variable X_i, given its parents —that is, $\theta_{ijk} = P(X_i = x_{ij} | Pa_i = pa_{ik})$—the update is given by the normalized expected counts as follows:

$$\theta_{ijk} \leftarrow \hat{N}(X_i = x_{ij}, Pa_i = pa_{ik}) / \hat{N}(Pa_i = pa_{ik}) \ .$$

The expected counts are obtained by summing over the examples, computing the probabilities $P(X_i = x_{ij}, Pa_i = pa_{ik})$ for each by using any Bayes net inference algorithm. For the exact algorithms—including variable elimination—all these probabilities are obtainable directly as a by-product of standard inference, with no need for extra computations specific to learning. Moreover, the information needed for learning is available *locally* for each parameter.

Learning hidden Markov models

Our final application of EM involves learning the transition probabilities in hidden Markov models (HMMs). Recall from Chapter 15 that a hidden Markov model can be represented by a dynamic Bayes net with a single discrete state variable, as illustrated in Figure 20.11. Each data point consists of an observation *sequence* of finite length, so the problem is to learn the transition probabilities from a set of observation sequences (or possibly from just one long sequence).

We have already worked out how to learn Bayes nets, but there is one complication: in Bayes nets, each parameter is distinct; in a hidden Markov model, on the other hand, the individual transition probabilities from state i to state j at time t, $\theta_{ijt} = P(X_{t+1} = j | X_t = i)$, are *repeated* across time—that is, $\theta_{ijt} = \theta_{ij}$ for all t. To estimate the transition probability

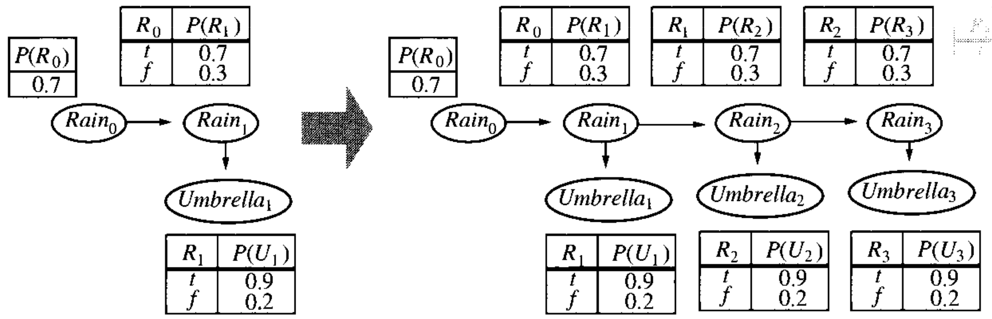


Figure 20.11 An unrolled dynamic Bayesian network that represents a hidden Markov model (repeat of Figure 15.14).

from state i to state j, we simply calculate the expected proportion of times that the system undergoes a transition to state j when in state i:

$$\theta_{ij} \leftarrow \sum_t \hat{N}(X_{t+1}=j, X_{t=i}) / \sum_t \hat{N}(X_{t=i}) .$$

Again, the expected counts are computed by any HMM inference algorithm. The **forward–backward** algorithm shown in Figure 15.4 can be modified very easily to compute the necessary probabilities. One important point is that the probabilities required are those obtained by **smoothing** rather than **filtering**; that is, we need to pay attention to subsequent evidence in estimating the probability that a particular transition occurred. As we said in Chapter 15, the evidence in a murder case is usually obtained *after* the crime (i.e., the transition from state i to state j) occurs.

The general form of the EM algorithm

We have seen several instances of the EM algorithm. Each involves computing expected values of hidden variables for each example and then recomputing the parameters, using the expected values as if they were observed values. Let $\mathbf{x}$ be all the observed values in all the examples, let $\mathbf{Z}$ denote all the hidden variables for all the examples, and let $\boldsymbol{\theta}$ be all the parameters for the probability model. Then the EM algorithm is

$$\boldsymbol{\theta}^{(i+1)} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{\mathbf{z}} P(\mathbf{Z}=\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}^{(i)}) L(\mathbf{x}, \mathbf{Z}=\mathbf{z}|\boldsymbol{\theta}) .$$

This equation is the EM algorithm in a nutshell. The E-step is the computation of the summation, which is the expectation of the log likelihood of the "completed" data with respect to the distribution $P(\mathbf{Z}=\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}^{(i)})$, which is the posterior over the hidden variables, given the data. The M-step is the maximization of this expected log likelihood with respect to the parameters. For mixtures of Gaussians, the hidden variables are the Z_{ij}s, where Z_{ij} is 1 if example j was generated by component i. For Bayes nets, the hidden variables are the values of the unobserved variables for each example. For HMMs, the hidden variables are the $i \rightarrow j$ transitions. Starting from the general form, it is possible to derive an EM algorithm for a specific application once the appropriate hidden variables have been identified.

As soon as we understand the general idea of EM, it becomes easy to derive all sorts of variants and improvements. For example, in many cases the E-step—the computation of posteriors over the hidden variables—is intractable, as in large Bayes nets. It turns out that one can use an *approximate* E-step and still obtain an effective learning algorithm. With a sampling algorithm such as MCMC (see Section 14.5), the learning process is very intuitive: each state (configuration of hidden and observed variables) visited by MCMC is treated exactly as if it were a complete observation. Thus, the parameters can be updated directly after each MCMC transition. Other forms of approximate inference, such as variational and loopy methods, have also proven effective for learning very large networks.

Learning Bayes net structures with hidden variables

In Section 20.2, we discussed the problem of learning Bayes net structures with complete data. When hidden variables are taken into consideration, things get more difficult. In the simplest case, the hidden variables are listed along with the observed variables; although their values are not observed, the learning algorithm is told that they exist and must find a place for them in the network structure. For example, an algorithm might try to learn the structure shown in Figure 20.7(a), given the information that *HeartDisease* (a three-valued variable) should be included in the model. If the learning algorithm is not told this information, then there are two choices: either pretend that the data is really complete—which forces the algorithm to learn the parameter-intensive model in Figure 20.7(b)—or *invent* new hidden variables in order to simplify the model. The latter approach can be implemented by including new modification choices in the structure search: in addition to modifying links, the algorithm can add or delete a hidden variable or change its arity. Of course, the algorithm will not know that the new variable it has invented is called *HeartDisease*; nor will it have meaningful names for the values. Fortunately, newly invented hidden variables will usually be connected to pre-existing variables, so a human expert can often inspect the local conditional distributions involving the new variable and ascertain its meaning.

As in the complete-data case, pure maximum-likelihood structure learning will result in a completely connected network (moreover, one with no hidden variables), so some form of complexity penalty is required. We can also apply MCMC to approximate Bayesian learning. For example, we can learn mixtures of Gaussians with an unknown number of components by sampling over the number; the approximate posterior distribution for the number of Gaussians is given by the sampling frequencies of the MCMC process.

So far, the process we have discussed has an outer loop that is a structural search process and an inner loop that is a parametric optimization process. For the complete-data case, the inner loop is very fast—just a matter of extracting conditional frequencies from the data set. When there are hidden variables, the inner loop may involve many iterations of EM or a gradient-based algorithm, and each iteration involves the calculation of posteriors in a Bayes net, which is itself an NP-hard problem. To date, this approach has proved impractical for learning complex models. One possible improvement is the so-called **structural EM** algorithm, which operates in much the same way as ordinary (parametric) EM except that the algorithm can update the structure as well as the parameters. Just as ordinary EM uses the current parameters to compute the expected counts in the E-step and then applies those counts in the M-step to choose new parameters, structural EM uses the current structure to compute

STRUCTURAL EM

expected counts and then applies those counts in the M-step to evaluate the likelihood for potential new structures. (This contrasts with the outer-loop/inner-loop method, which computes new expected counts for each potential structure.) In this way, structural EM may make several structural alterations to the network without once recomputing the expected counts, and is capable of learning nontrivial Bayes net structures. Nonetheless, much work remains to be done before we can say that the structure learning problem is solved.

20.4 INSTANCE-BASED LEARNING

PARAMETRIC LEARNING

So far, our discussion of statistical learning has focused primarily on fitting the parameters of a *restricted* family of probability models to an *unrestricted* data set. For example, unsupervised clustering using mixtures of Gaussians assumes that the data are explained by the *sum* a *fixed* number of *Gaussian* distributions. We call such methods **parametric learning**. Parametric learning methods are often simple and effective, but assuming a particular restricted family of models often oversimplifies what's happening in the real world, from where the data come. Now, it is true when we have very little data, we cannot hope to learn a complex and detailed model, but it seems silly to keep the hypothesis complexity fixed even when the data set grows very large!

NONPARAMETRIC LEARNING

INSTANCE-BASED LEARNING

In contrast to parametric learning, **nonparametric learning** methods allow the hypothesis complexity to grow with the data. The more data we have, the wigglier the hypothesis can be. We will look at two very simple families of nonparametric **instance-based learning** (or **memory-based learning**) methods, so called because they construct hypotheses directly from the training instances themselves.

Nearest-neighbor models

NEAREST-NEIGHBOR

The key idea of **nearest-neighbor** models is that the properties of any particular input point $\mathbf{x}$ are likely to be similar to those of points in the neighborhood of $\mathbf{x}$. For example, if we want to do **density estimation**—that is, estimate the value of an unknown probability density at $\mathbf{x}$—then we can simply measure the density with which points are scattered in the neighborhood of $\mathbf{x}$. This sounds very simple, until we realize that we need to specify exactly what we mean by "neighborhood." If the neighborhood is too small, it won't contain any data points; too large, and it may include *all* the data points, resulting in a density estimate that is the same everywhere. One solution is to define the neighborhood to be just big enough to include k points, where k is large enough to ensure a meaningful estimate. For fixed k, the size of the neighborhood varies—where data are sparse, the neighborhood is large, but where data are dense, the neighborhood is small. Figure 20.12(a) shows an example for data scattered in two dimensions. Figure 20.13 shows the results of k-nearest-neighbor density estimation from these data with $k = 3$, 10, and 40 respectively. For $k = 3$, the density estimate at any point is based on only 3 neighboring points and is highly variable. For $k = 10$, the estimate provides a good reconstruction of the true density shown in Figure 20.12(b). For $k = 40$, the neighborhood becomes too large and structure of the data is altogether lost. In practice, using

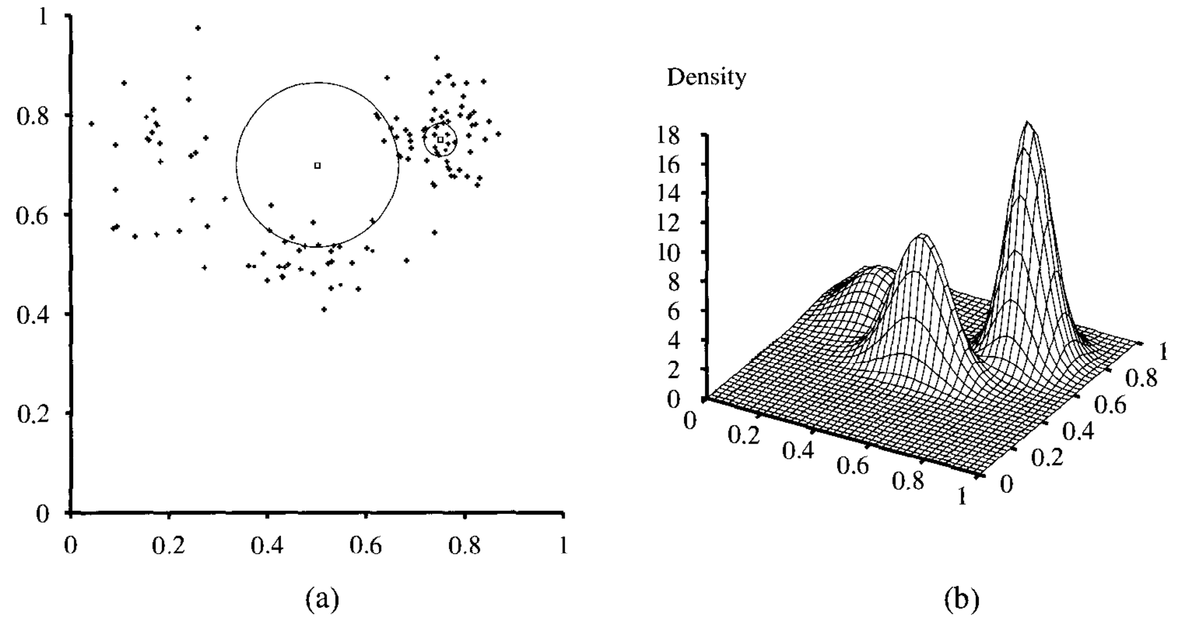


Figure 20.12 (a) A 128-point subsample of the data shown in Figure 20.8(a), together with two query points and their 10-nearest-neighborhoods. (b) A 3-D plot of the mixture of Gaussians from which the data were generated.

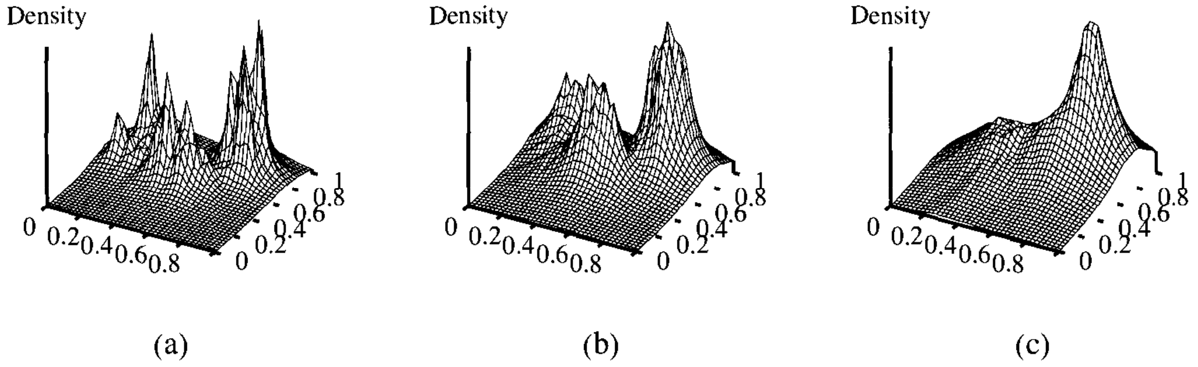


Figure 20.13 Density estimation using k-nearest-neighbors, applied to the data in Figure 20.12(a), for $k = 3$, 10, and 40 respectively.

a value of k somewhere between 5 and 10 gives good results for most low-dimensional data sets. A good value of k can also be chosen by using cross-validation.

To identify the nearest neighbors of a query point, we need a distance metric, $D(\mathbf{x}_1, \mathbf{x}_2)$. The two-dimensional example in Figure 20.12 uses Euclidean distance. This is inappropriate when each dimension of the space is measuring something different—for example, height and weight—because changing the scale of one dimension would change the set of nearest neighbors. One solution is to standardize the scale for each dimension. To do this, we measure the standard deviation of each feature over the whole data set and express feature values as multiples of the standard deviation for that feature. (This is a special case of the **Mahalanobis distance**, which takes into account the covariance of the features as well.) Finally, for discrete features we can use the **Hamming distance**, which defines $D(\mathbf{x}_1, \mathbf{x}_2)$ to be the number of features on which $\mathbf{x}_1$ and $\mathbf{x}_2$ differ.

MAHALANOBIS DISTANCE

HAMMING DISTANCE

Density estimates like those shown in Figure 20.13 define joint distributions over the input space. Unlike a Bayesian network, however, an instance-based representation cannot contain hidden variables, which means that we cannot perform unsupervised clustering as we did with the mixture-of-Gaussians model. We can still use the density estimate to predict a target value y given input feature values $\mathbf{x}$ by calculating $P(y|\mathbf{x}) = P(y, \mathbf{x})/P(\mathbf{x})$, provided that the training data include values for the target feature.

It is also possible to use the nearest-neighbor idea for direct supervised learning. Given a test example with input $\mathbf{x}$, the output $y = h(\mathbf{x})$ is obtained from the y-values of the k nearest neighbors of $\mathbf{x}$. In the discrete case, we can obtain a single prediction by majority vote. In the continuous case, we can average the k values or do local linear regression, fitting a hyperplane to the k points and predicting the value at $\mathbf{x}$ according to the hyperplane.

The k-nearest-neighbor learning algorithm is very simple to implement, requires little in the way of tuning, and often performs quite well. It is a good thing to try first on a new learning problem. For large data sets, however, we require an efficient mechanism for finding the nearest neighbors of a query point $\mathbf{x}$—simply calculating the distance to every point would take far too long. A variety of ingenious methods have been proposed to make this step efficient by preprocessing the training data. Unfortunately, most of these methods do not scale well with the dimension of the space (i.e., the number of features).

High-dimensional spaces pose an additional problem, namely that nearest neighbors in such spaces are usually a long way away! Consider a data set of size N in the d-dimensional unit hypercube, and assume hypercubic neighborhoods of side b and volume b^d. (The same argument works with hyperspheres, but the formula for the volume of a hypersphere is more complicated.) To contain k points, the average neighborhood must occupy a fraction k/N of the entire volume, which is 1. Hence, $b^d = k/N$, or $b = (k/N)^{1/d}$. So far, so good. Now let the number of features d be 100 and let k be 10 and N be 1,000,000. Then we have $b \approx 0.89$—that is, the neighborhood has to span almost the entire input space! This suggests that nearest-neighbor methods cannot be trusted for high-dimensional data. In low dimensions there is no problem; with $d = 2$ we have $b = 0.003$.

Kernel models

KERNEL MODEL

KERNEL FUNCTION

In a **kernel model**, we view each training instance as generating a little density function—a **kernel function**—of its own. The density estimate as a whole is just the normalized sum of all the little kernel functions. A training instance at $\mathbf{x}_i$ will generate a kernel function $K(\mathbf{x}, \mathbf{x}_i)$ that assigns a probability to each point $\mathbf{x}$ in the space. Thus, the density estimate is

$$P(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} K(\mathbf{x}, \mathbf{x}_i) .$$

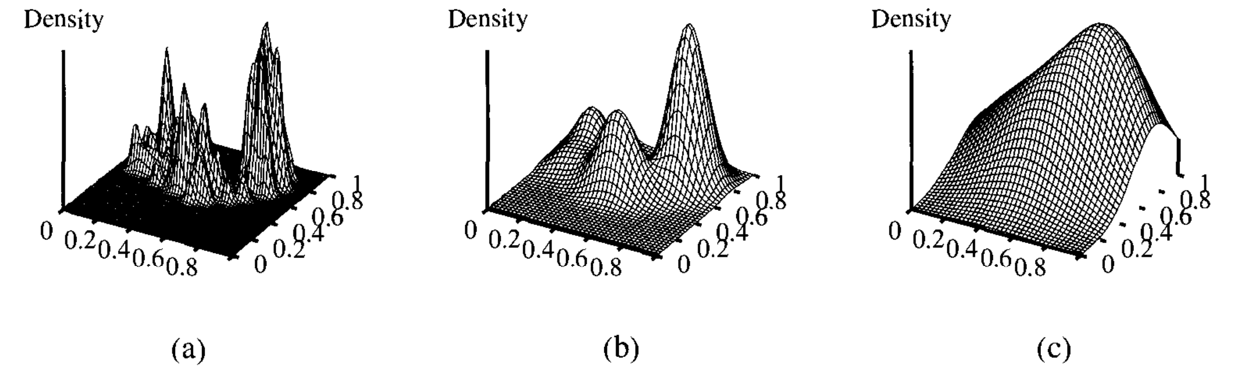


Figure 20.14 Kernel density estimation for the data in Figure 20.12(a), using Gaussian kernels with $w = 0.02$, 0.07, and 0.20 respectively.

The kernel function normally depends only on the *distance* $D(\mathbf{x}, \mathbf{x}_i)$ from $\mathbf{x}$ to the instance $\mathbf{x}_i$. The most popular kernel function is (of course) the Gaussian. For simplicity, we will assume spherical Gaussians with standard deviation w along each axis, i.e.,

$$K(\mathbf{x}, \mathbf{x}_i) = \frac{1}{(w^2\sqrt{2\pi})^d} e^{-\frac{D(\mathbf{x},\mathbf{x}_i)^2}{2w^2}} ,$$

where d is the number of dimensions in $\mathbf{x}$. We still have the problem of choosing a suitable value for w; as before, making the neighborhood too small gives a very spiky estimate—see Figure 20.14(a). In (b), a medium value of w gives a very good reconstruction. In (c), too large a neighborhood results in losing the structure altogether. A good value of w can be chosen by using cross-validation.

Supervised learning with kernels is done by taking a *weighted* combination of *all* the predictions from the training instances. (Compare this with k-nearest-neighbor prediction, which takes an unweighted combination of the nearest k instances.) The weight of the ith instance for a query point $\mathbf{x}$ is given by the value of the kernel $K(\mathbf{x}, \mathbf{x}_i)$. For a discrete prediction, we can take a weighted vote; for a continuous prediction, we can take weighted average or a weighted linear regression. Notice that making predictions with kernels requires looking at *every* training instance. It is possible to combine kernels with nearest-neighbor indexing schemes to make weighted predictions from just the nearby instances.

20.5 Neural Networks

A **neuron** is a cell in the brain whose principal function is the collection, processing, and dissemination of electrical signals. Figure 1.2 on page 11 showed a schematic diagram of a typical neuron. The brain's information-processing capacity is thought to emerge primarily from *networks* of such neurons. For this reason, some of the earliest AI work aimed to create
NEURAL NETWORKS artificial **neural networks**. (Other names for the field include **connectionism**, **parallel dis-**

tributed processing, and **neural computation**.) Figure 20.15 shows a simple mathematical model of the neuron devised by McCulloch and Pitts (1943). Roughly speaking, it "fires" when a linear combination of its inputs exceeds some threshold. Since 1943, much more detailed and realistic models have been developed, both for neurons and for larger systems in the brain, leading to the modern field of **computational neuroscience**. On the other hand, researchers in AI and statistics became interested in the more abstract properties of neural networks, such as their ability to perform distributed computation, to tolerate noisy inputs, and to learn. Although we understand now that other kinds of systems—including Bayesian networks—have these properties, neural networks remain one of the most popular and effective forms of learning system and are worthy of study in their own right.

COMPUTATIONAL NEUROSCIENCE

Units in neural networks

UNITS

LINKS

ACTIVATION

WEIGHT

Neural networks are composed of nodes or **units** (see Figure 20.15) connected by directed **links**. A link from unit j to unit i serves to propagate the **activation** a_j from j to i. Each link also has a numeric **weight** $W_{j,i}$ associated with it, which determines the strength and sign of the connection. Each unit i first computes a weighted sum of its inputs:

$$in_i = \sum_{j=0}^{n} W_{j,i}a_j \, .$$

ACTIVATION FUNCTION

Then it applies an **activation function** g to this sum to derive the output:

$$a_i = g(in_i) = g\left(\sum_{j=0}^{n} W_{j,i}a_j\right) \, . \qquad (20.10)$$

BIAS WEIGHT

Notice that we have included a **bias weight** $W_{0,i}$ connected to a fixed input $a_0 = -1$. We will explain its role in a moment.

The activation function g is designed to meet two desiderata. First, we want the unit to be "active" (near +1) when the "right" inputs are given, and "inactive" (near 0) when the "wrong" inputs are given. Second, the activation needs to be *nonlinear*, otherwise the entire neural network collapses into a simple linear function (see Exercise 20.17). Two choices for g

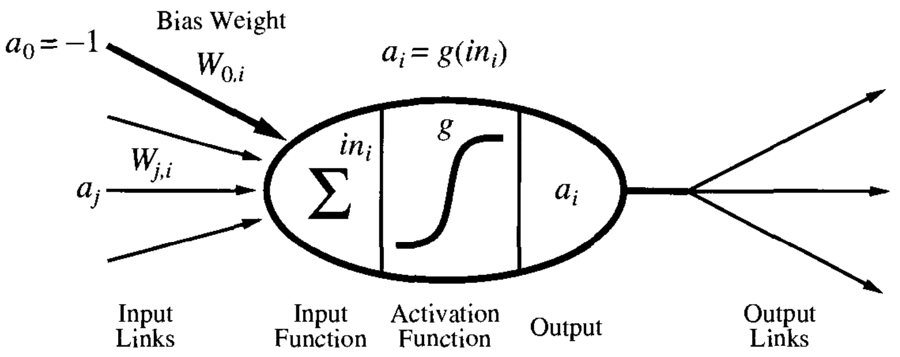


Figure 20.15 A simple mathematical model for a neuron. The unit's output activation is $a_i = g(\sum_{j=0}^{n} W_{j,i}a_j)$, where a_j is the output activation of unit j and $W_{j,i}$ is the weight on the link from unit j to this unit.

THRESHOLD
SIGMOID FUNCTION
LOGISTIC FUNCTION

are shown in Figure 20.16: the **threshold** function and the **sigmoid function** (also known as the **logistic function**). The sigmoid function has the advantage of being differentiable, which we will see later is important for the weight-learning algorithm. Notice that both functions have a threshold (either hard or soft) at zero; the bias weight $W_{0,i}$ sets the *actual* threshold for the unit, in the sense that the unit is activated when the weighted sum of "real" inputs $\sum_{j=1}^{n} W_{j,i} a_j$ (i.e., excluding the bias input) exceeds $W_{0,i}$.

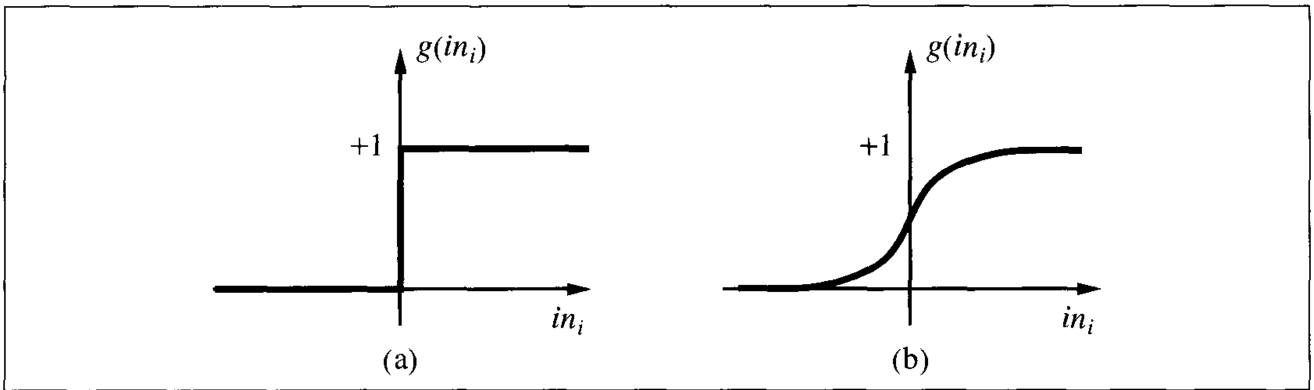


Figure 20.16 (a) The **threshold** activation function, which outputs 1 when the input is positive and 0 otherwise. (Sometimes the **sign** function is used instead, which outputs ±1 depending on the sign of the input.) (b) The **sigmoid** function $1/(1+e^{-x})$.

We can get a feel for the operation of individual units by comparing them with logic gates. One of the original motivations for the design of individual units (McCulloch and Pitts, 1943) was their ability to represent basic Boolean functions. Figure 20.17 shows how the Boolean functions AND, OR, and NOT can be represented by threshold units with suitable weights. This is important because it means we can use these units to build a network to compute any Boolean function of the inputs.

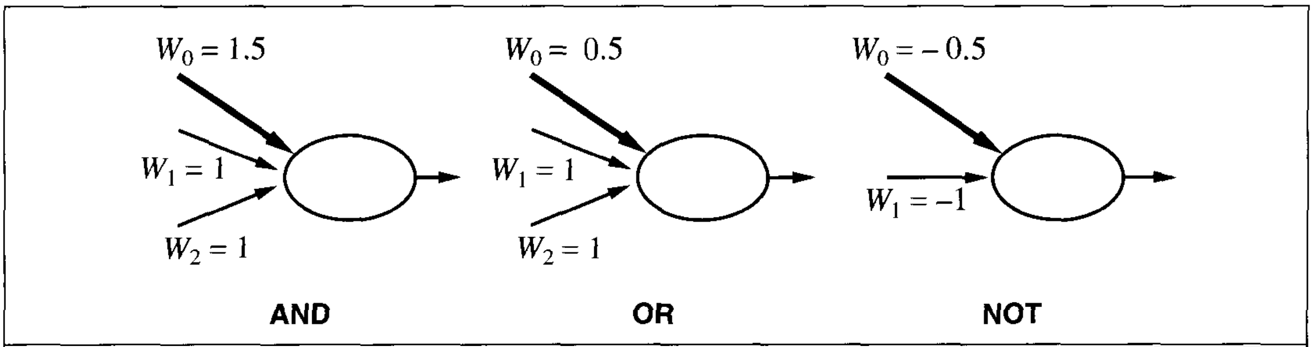


Figure 20.17 Units with a threshold activation function can act as logic gates, given appropriate input and bias weights.

Network structures

FEED-FORWARD NETWORKS
RECURRENT NETWORKS

There are two main categories of neural network structures: acyclic or **feed-forward networks** and cyclic or **recurrent networks**. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A recurrent network, on the other hand, feeds its outputs back into its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behavior. Moreover, the response of the network to a given input depends on its initial state, which may depend on previous inputs. Hence, recurrent networks (unlike feed-forward networks) can support short-term memory. This makes them more interesting as models of the brain, but also more difficult to understand. This section will concentrate on feed-forward networks; some pointers for further reading on recurrent networks are given at the end of the chapter.

HIDDEN UNITS

Let us look more closely into the assertion that a feed-forward network represents a function of its inputs. Consider the simple network shown in Figure 20.18, which has two input units, two **hidden units**, and an output unit. (To keep things simple, we have omitted the bias units in this example.) Given an input vector $\mathbf{x} = (x_1, x_2)$, the activations of the input units are set to $(a_1, a_2) = (x_1, x_2)$ and the network computes

$$\begin{aligned} a_5 &= g(W_{3,5}a_3 + W_{4,5}a_4) \\ &= g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2)) . \end{aligned} \tag{20.11}$$

That is, by expressing the output of each hidden unit as a function of *its* inputs, we have shown that output of the network as a whole, a_5, is a function of the network's inputs. Furthermore, we see that the weights in the network act as *parameters* of this function; writing $\mathbf{W}$ for the parameters, the network computes a function $h_{\mathbf{W}}(\mathbf{x})$. By adjusting the weights, we change the function that the network represents. This is how learning occurs in neural networks.

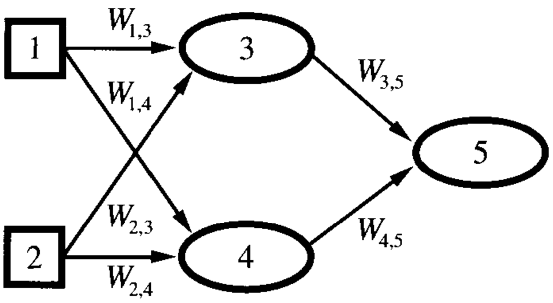
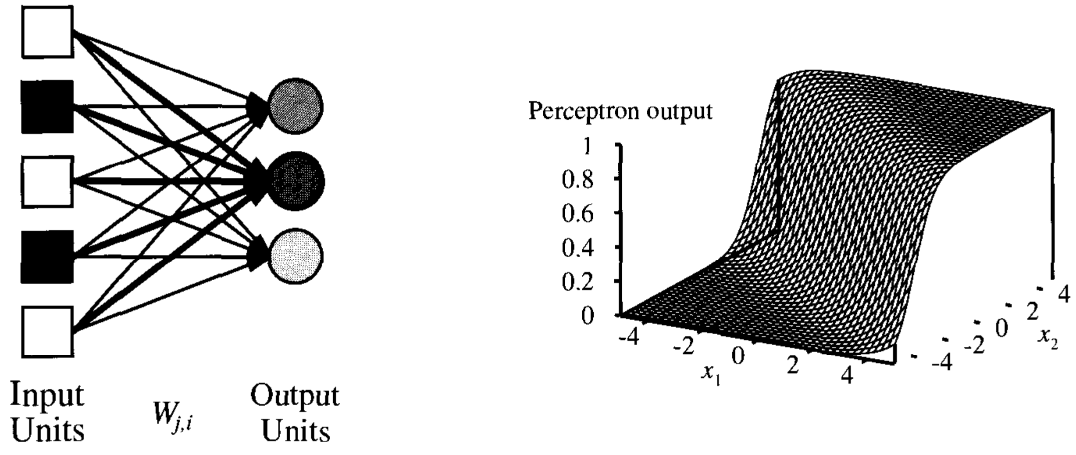


Figure 20.18 A very simple neural network with two inputs, one hidden layer of two units, and one output.

A neural network can be used for classification or regression. For Boolean classification with continuous outputs (e.g., with sigmoid units), it is traditional to have a single output unit, with a value over 0.5 interpreted as one class and a value below 0.5 as the other. For k-way classification, one could divide the single output unit's range into k portions, but it is more common to have k separate output units, with the value of each one representing the relative likelihood of that class given the current input.

LAYERS

Feed-forward networks are usually arranged in **layers**, such that each unit receives input only from units in the immediately preceding layer. In the next two subsections, we will look at single layer networks, which have no hidden units, and multilayer networks, which have one or more layers of hidden units.



(a) (b)

Figure 20.19 (a) A perceptron network consisting of three perceptron output units that share five inputs. Looking at a particular output unit (say the second one, outlined in bold), we see that the weights on its incoming links have no effect on the other output units. (b) A graph of the output of a two-input perceptron unit with a sigmoid activation function.

Single layer feed-forward neural networks (perceptrons)

SINGLE-LAYER NEURAL NETWORK

PERCEPTRON

A network with all the inputs connected directly to the outputs is called a **single-layer neural network**, or a **perceptron** network. Since each output unit is independent of the others—each weight affects only one of the outputs—we can limit our study to perceptrons with a single output unit, as explained in Figure 20.19(a).

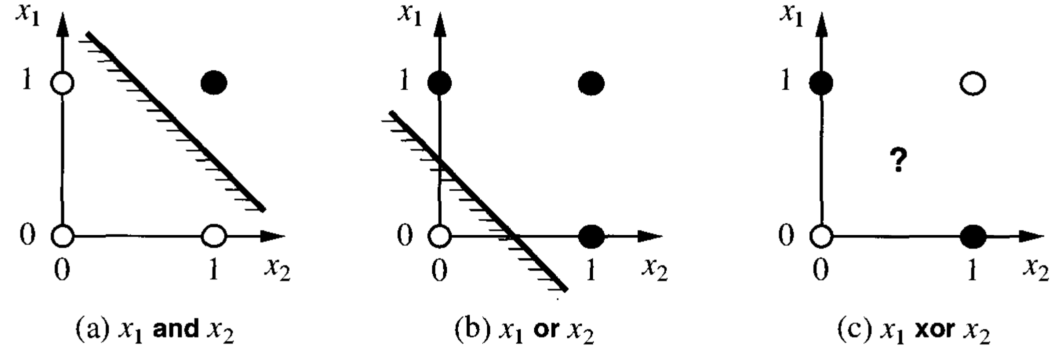
Let us begin by examining the hypothesis space that a perceptron can represent. With a threshold activation function, we can view the perceptron as representing a Boolean function. In addition to the elementary Boolean functions AND, OR, and NOT (Figure 20.17), a perceptron can represent some quite "complex" Boolean functions very compactly. For example, the **majority function**, which outputs a 1 only if more than half of its n inputs are 1, can be represented by a perceptron with each $W_j = 1$ and threshold $W_0 = n/2$. A decision tree would need $O(2^n)$ nodes to represent this function.

Unfortunately, there are many Boolean functions that the threshold perceptron cannot represent. Looking at Equation (20.10), we see that the threshold perceptron returns 1 if and only if the weighted sum of its inputs (including the bias) is positive:

$$\sum_{j=0}^{n} W_j x_j > 0 \qquad \text{or} \qquad \mathbf{W} \cdot \mathbf{x} > 0 \, .$$

LINEAR SEPARATOR

Now, the equation $\mathbf{W} \cdot \mathbf{x} = 0$ defines a *hyperplane* in the input space, so the perceptron returns 1 if and only if the input is on one side of that hyperplane. For this reason, the threshold perceptron is called a **linear separator**. Figure 20.20(a) and (b) show this hyperplane (a line, in two dimensions) for the perceptron representations of the AND and OR functions of two inputs. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron can represent these functions because there is some line that separates all the white dots from all the black



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Figure 20.20 Linear separability in threshold perceptrons. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron returns 1 on the region on the non-shaded side of the line. In (c), no such line exists that correctly classifies the inputs.

LINEARLY SEPARABLE

dots. Such functions are called **linearly separable**. Figure 20.20(c) shows an example of a function that is *not* linearly separable—the XOR function. Clearly, there is no way for a threshold perceptron to learn this function. In general, *threshold perceptrons can represent only linearly separable functions.* These constitute just a small fraction of all functions; Exercise 20.14 asks you to quantify this fraction. Sigmoid perceptrons are similarly limited, in the sense that they represent only "soft" linear separators. (See Figure 20.19(b).)

Despite their limited expressive power, threshold perceptrons have some advantages. In particular, *there is a simple learning algorithm that will fit a threshold perceptron to any linearly separable training set.* Rather than present this algorithm, we will *derive* a closely related algorithm for learning in sigmoid perceptrons.

WEIGHT SPACE

The idea behind this algorithm, and indeed behind most algorithms for neural network learning, is to adjust the weights of the network to minimize some measure of the error on the training set. Thus, learning is formulated as an optimization search in **weight space.**[8] The "classical" measure of error is the **sum of squared errors**, which we used for linear regression on page 720. The squared error for a single training example with input $\mathbf{x}$ and true output y is written as

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 ,$$

where $h_{\mathbf{W}}(\mathbf{x})$ is the output of the perceptron on the example.

We can use gradient descent to reduce the squared error by calculating the partial derivative of E with respect to each weight. We have

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j}\\ &= Err \times \frac{\partial}{\partial W_j}\left(y - g\left(\sum_{j=0}^{n} W_j x_j\right)\right)\\ &= -Err \times g'(in) \times x_j ,\end{aligned}$$

[8] See Section 4.4 for general optimization techniques applicable to continuous spaces.

where g' is the derivative of the activation function.[9] In the gradient descent algorithm, where we want to *reduce* E, we update the weight as follows:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j \,, \qquad (20.12)$$

where α is the **learning rate**. Intuitively, this makes a lot of sense. If the error $Err = y - h_{\mathbf{W}}(\mathbf{x})$ is positive, then the network output is too small and so the weights are *increased* for the positive inputs and *decreased* for the negative inputs. The opposite happens when the error is negative.[10]

EPOCH

STOCHASTIC GRADIENT

The complete algorithm is shown in Figure 20.21. It runs the training examples through the net one at a time, adjusting the weights slightly after each example to reduce the error. Each cycle through the examples is called an **epoch**. Epochs are repeated until some stopping criterion is reached—typically, that the weight changes have become very small. Other methods calculate the gradient for the whole training set by adding up all the gradient contributions in Equation (20.12) before updating the weights. The **stochastic gradient** method selects examples randomly from the training set rather than cycling through them.

```
function PERCEPTRON-LEARNING(examples, network) returns a perceptron hypothesis
  inputs: examples, a set of examples, each with input $\mathbf{x} = x_1, \ldots, x_n$ and output $y$
          network, a perceptron with weights $W_j$, $j = 0 \ldots n$, and activation function $g$

  repeat
     for each $e$ in examples do
        $in \leftarrow \sum_{j=0}^{n} W_j\, x_j[e]$
        $Err \leftarrow y[e] - g(in)$
        $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)
```

Figure 20.21 The gradient descent learning algorithm for perceptrons, assuming a differentiable activation function g. For threshold perceptrons, the factor $g'(in)$ is omitted from the weight update. NEURAL-NET-HYPOTHESIS returns a hypothesis that computes the network output for any given example.

Figure 20.22 shows the learning curve for a perceptron on two different problems. On the left, we show the curve for learning the majority function with 11 Boolean inputs (i.e., the function outputs a 1 if 6 or more inputs are 1). As we would expect, the perceptron learns the function quite quickly, because the majority function is linearly separable. On the other hand, the decision-tree learner makes no progress, because the majority function is very hard (although not impossible) to represent as a decision tree. On the right, we have the restaurant

[9] For the sigmoid, this derivative is given by $g' = g(1 - g)$.

[10] For threshold perceptrons, where $g'(in)$ is undefined, the original **perceptron learning rule** developed by Rosenblatt (1957) is identical to Equation (20.12) except that $g'(in)$ is omitted. Since $g'(in)$ is the same for all weights, its omission changes only the magnitude and not the direction of the overall weight update for each example.

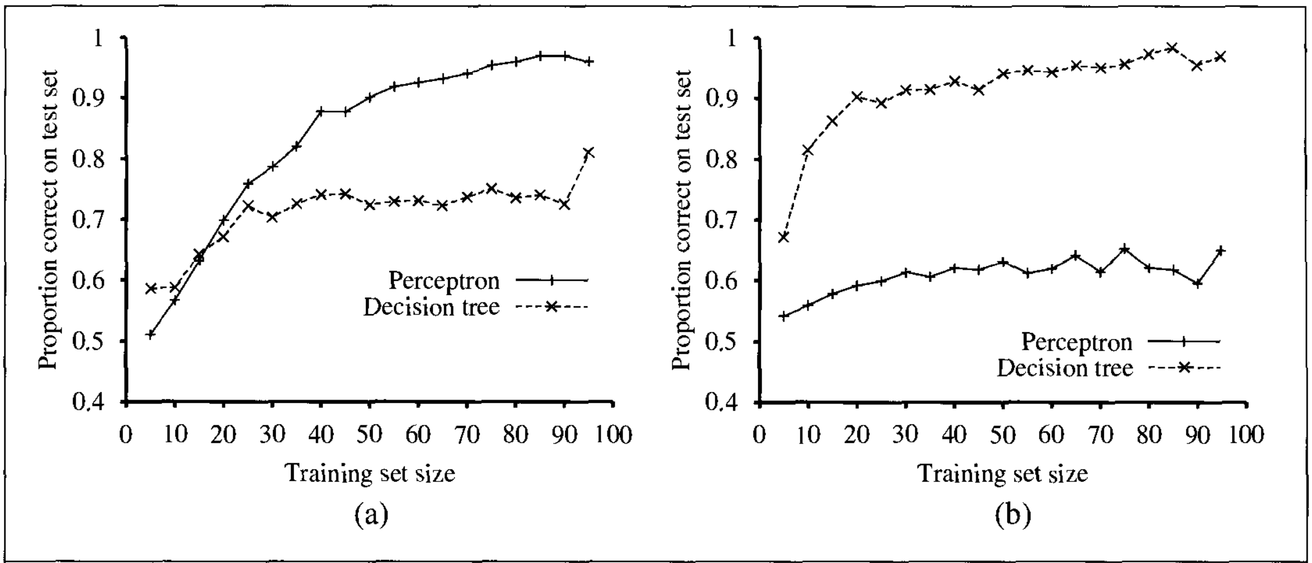


Figure 20.22 Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WillWait* predicate in the restaurant example.

example. The solution problem is easily represented as a decision tree, but is not linearly separable. The best plane through the data correctly classifies only 65%.

So far, we have treated perceptrons as deterministic functions with possibly erroneous outputs. It is also possible to interpret the output of a sigmoid perceptron as a *probability*—specifically, the probability that the true output is 1 given the inputs. With this interpretation, one can use the sigmoid as a canonical representation for conditional distributions in Bayesian networks (see Section 14.3). One can also derive a learning rule using the standard method of maximizing the (conditional) log likelihood of the data, as described earlier in this chapter. Let's see how this works.

INDICATOR VARIABLE

Consider a single training example with true output value T, and let p be the probability returned by the perceptron for this example. If $T = 1$, the conditional probability of the datum is p, and if $T = 0$, the conditional probability of the datum is $(1 - p)$. Now we can use a simple trick to write the log likelihood in a form that is differentiable. The trick is that a 0/1 variable in the *exponent* of an expression acts as an **indicator variable**: p^T is p if $T = 1$ and 1 otherwise; similarly $(1 - p)^{(1-T)}$ is $(1 - p)$ if $T = 0$ and 1 otherwise. Hence, we can write the log likelihood of the datum as

$$L = \log p^T (1 - p)^{(1-T)} = T \log p + (1 - T) \log(1 - p) . \qquad (20.13)$$

Thanks to the properties of the sigmoid function, the gradient reduces to a very simple formula (Exercise 20.16):

$$\frac{\partial L}{\partial W_j} = Err \times a_j .$$

Notice that *the weight-update vector for maximum likelihood learning in sigmoid perceptrons is essentially identical to the update vector for squared error minimization.* Thus, we could say that perceptrons have a probabilistic interpretation even when the learning rule is derived from a deterministic viewpoint.

Multilayer feed-forward neural networks

Now we will consider networks with hidden units. The most common case involves a single hidden layer,[11] as in Figure 20.24. The advantage of adding hidden layers is that it enlarges the space of hypotheses that the network can represent. Think of each hidden unit as a perceptron that represents a soft threshold function in the input space, as shown in Figure 20.19(b). Then, think of an output unit as as a soft-thresholded linear combination of several such functions. For example, by adding two opposite-facing soft threshold functions and thresholding the result, we can obtain a "ridge" function as shown in Figure 20.23(a). Combining two such ridges at right angles to each other (i.e., combining the outputs from four hidden units), we obtain a "bump" as shown in Figure 20.23(b).

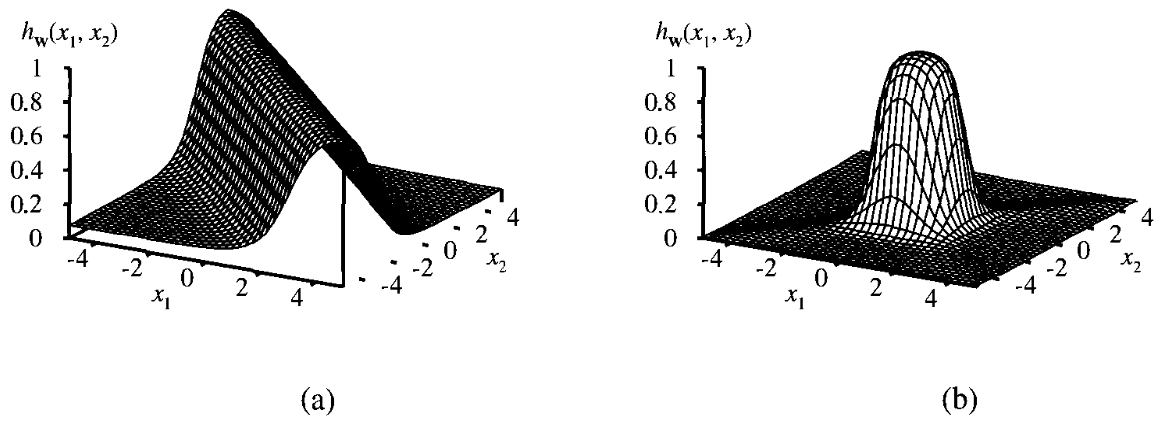


Figure 20.23 (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

With more hidden units, we can produce more bumps of different sizes in more places. In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented.[12] Unfortunately, for any *particular* network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

Suppose we want to construct a hidden layer network for the restaurant problem. We have 10 attributes describing each example, so we will need 10 input units. How many hidden units are needed? In Figure 20.24, we show a network with four hidden units. This turns out to be about right for this problem. The problem of choosing the right number of hidden units in advance is still not well understood. (See page 748.)

Learning algorithms for multilayer networks are similar to the perceptron learning algorithm show in Figure 20.21. One minor difference is that we may have several outputs, so

[11] Some people call this a three-layer network, and some call it a two-layer network (because the inputs aren't "real" units). We will avoid confusion and call it a "single-hidden-layer network."

[12] The proof is complex, but the main point is that the required number of hidden units grows exponentially with the number of inputs. For example, $2^n/n$ hidden units are needed to encode all Boolean functions of n inputs.

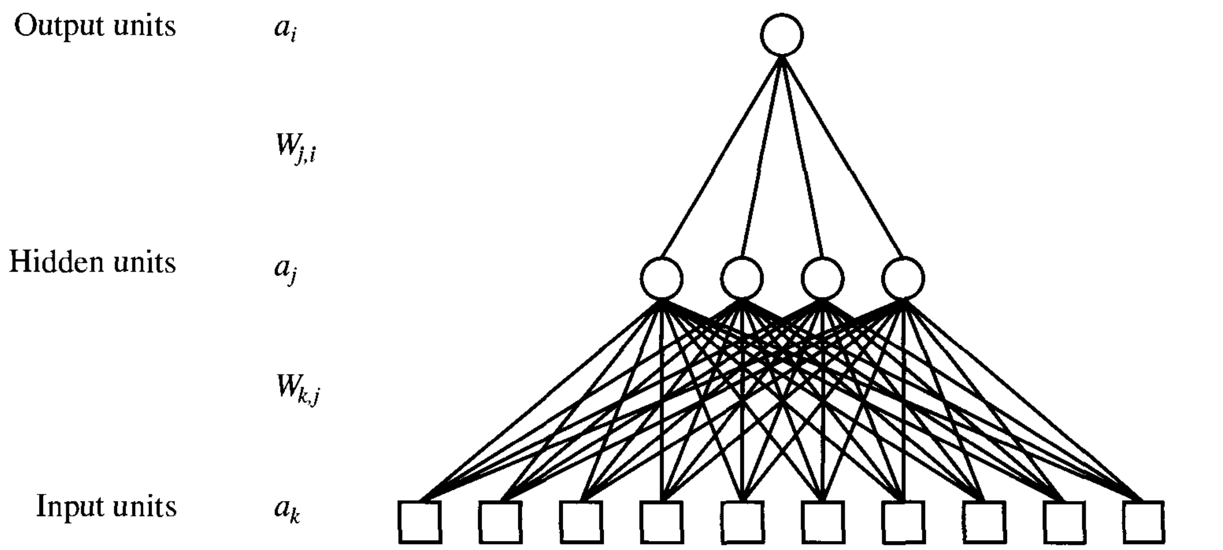


Figure 20.24 A multilayer neural network with one hidden layer and 10 inputs, suitable for the restaurant problem.

we have an output vector $\mathbf{h_W}(\mathbf{x})$ rather than a single value, and each example has an output vector $\mathbf{y}$. The major difference is that, whereas the error $\mathbf{y} - \mathbf{h_W}$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data does not say what value the hidden nodes should have. It turns out that we can **back-propagate** the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient. First, we will describe the process with an intuitive justification; then, we will show the derivation.

BACK-PROPAGATION

At the output layer, the weight-update rule is identical to Equation (20.12). We have multiple output units, so let Err_i be ith component of the error vector $\mathbf{y} - \mathbf{h_W}$. We will also find it convenient to define a modified error $\Delta_i = Err_i \times g'(in_i)$, so that the weight-update rule becomes

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \; . \qquad (20.14)$$

To update the connections between the input units and the hidden units, we need to define a quantity analogous to the error term for output nodes. Here is where we do the error back-propagation. The idea is that hidden node j is "responsible" for some fraction of the error Δ_i in each of the output nodes to which it connects. Thus, the Δ_i values are divided according to the strength of the connection between the hidden node and the output node and are propagated back to provide the Δ_j values for the hidden layer. The propagation rule for the Δ values is the following:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \; . \qquad (20.15)$$

Now the weight-update rule for the weights between the inputs and the hidden layer is almost identical to the update rule for the output layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \; .$$

The back-propagation process can be summarized as follows:

```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
          network, a multilayer network with $L$ layers, weights $W_{j,i}$, activation function $g$

  repeat
     for each $e$ in examples do
        for each node $j$ in the input layer do $a_j \leftarrow x_j[e]$
        for $\ell = 2$ to $L$ do
           $in_i \leftarrow \sum_j W_{j,i} \, a_j$
           $a_i \leftarrow g(in_i)$
        for each node $i$ in the output layer do
           $\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$
        for $\ell = L - 1$ to $1$ do
           for each node $j$ in layer $\ell$ do
              $\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \, \Delta_i$
              for each node $i$ in layer $\ell + 1$ do
                 $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)
```

Figure 20.25 The back-propagation algorithm for learning in multilayer networks.

- Compute the Δ values for the output units, using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
  - Propagate the Δ values back to the previous layer.
  - Update the weights between the two layers.

The detailed algorithm is shown in Figure 20.25.

For the mathematically inclined, we will now derive the back-propagation equations from first principles. The squared error on a single example is defined as

$$E = \frac{1}{2} \sum_i (y_i - a_i)^2 \,,$$

where the sum is over the nodes in the output layer. To obtain the gradient with respect to a specific weight $W_{j,i}$ in the output layer, we need only expand out the activation a_i as all other terms in the summation are unaffected by $W_{j,i}$:

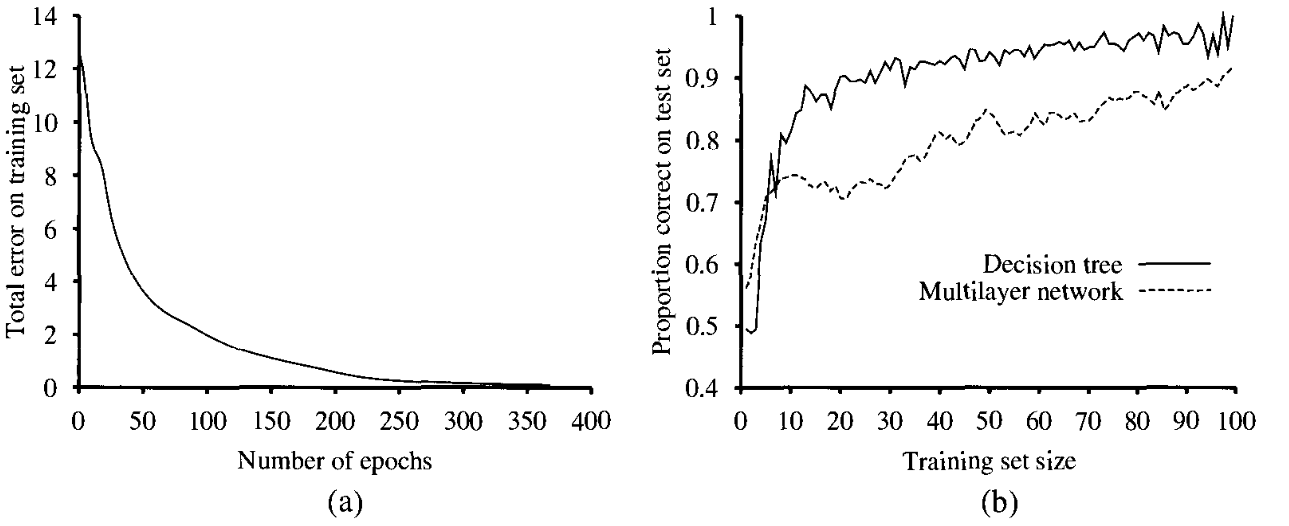
$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i) g'(in_i) \frac{\partial}{\partial W_{j,i}} \left(\sum_j W_{j,i} a_j \right) \\
&= -(y_i - a_i) g'(in_i) a_j = -a_j \Delta_i \,,
\end{aligned}$$

with Δ_i defined as before. To obtain the gradient with respect to the $W_{k,j}$ weights connecting the input layer to the hidden layer, we have to keep the entire summation over i because each output value a_i may be affected by changes in $W_{k,j}$. We also have to expand out the activations a_j. We will show the derivation in gory detail because it is interesting to see how the derivative operator propagates back through the network:

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j \,,
\end{aligned}$$

where Δ_j is defined as before. Thus, we obtain the update rules obtained earlier from intuitive considerations. It is also clear that the process can be continued for networks with more than one hidden layer, which justifies the general algorithm given in Figure 20.25.

Having made it through (or skipped over) all the mathematics, let's see how a single-hidden-layer network performs on the restaurant problem. In Figure 20.26, we show two



(a)

(b)

Figure 20.26 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better than back-propagation in a multilayer network.

TRAINING CURVE

curves. The first is a **training curve**, which shows the mean squared error on a given training set of 100 restaurant examples during the weight-updating process. This demonstrates that the network does indeed converge to a perfect fit to the training data. The second curve is the standard learning curve for the restaurant data. The neural network does learn well, although not quite as fast as decision-tree learning; this is perhaps not surprising, because the data were generated from a simple decision tree in the first place.

Neural networks are capable of far more complex learning tasks of course, although it must be said that a certain amount of twiddling is needed to get the network structure right and to achieve convergence to something close to the global optimum in weight space. There are literally tens of thousands of published applications of neural networks. Section 20.7 looks at one such application in more depth.

Learning neural network structures

So far, we have considered the problem of learning weights, given a fixed network structure; just as with Bayesian networks, we also need to understand how to find the best network structure. If we choose a network that is too big, it will be able to memorize all the examples by forming a large lookup table, but will not necessarily generalize well to inputs that have not been seen before.[13] In other words, like all statistical models, neural networks are subject to **overfitting** when there are too many parameters in the model. We saw this in Figure 18.1 (page 652), where the high-parameter models in (b) and (c) fit all the data, but might not generalize as well as the low-parameter models in (a) and (d).

If we stick to fully connected networks, the only choices to be made concern the number of hidden layers and their sizes. The usual approach is to try several and keep the best. The **cross-validation** techniques of Chapter 18 are needed if we are to avoid **peeking** at the test set. That is, we choose the network architecture that gives the highest prediction accuracy on the validation sets.

If we want to consider networks that are not fully connected, then we need to find some effective search method through the very large space of possible connection topologies.

OPTIMAL BRAIN DAMAGE

The **optimal brain damage** algorithm begins with a fully connected network and removes connections from it. After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped. The network is then retrained, and if its performance has not decreased then the process is repeated. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.

Several algorithms have been proposed for growing a larger network from a smaller one.

TILING

One, the **tiling** algorithm, resembles decision-list learning. The idea is to start with a single unit that does its best to produce the correct output on as many of the training examples as possible. Subsequent units are added to take care of the examples that the first unit got wrong. The algorithm adds only as many units as are needed to cover all the examples.

[13] It has been observed that very large networks *do* generalize well *as long as the weights are kept small.* This restriction keeps the activation values in the *linear* region of the sigmoid function $g(x)$ where x is close to zero. This, in turn, means that the network behaves like a linear function (Exercise 20.17) with far fewer parameters.

20.6 Kernel Machines

Our discussion of neural networks left us with a dilemma. Single-layer networks have a simple and efficient learning algorithm, but have very limited expressive power—they can learn only linear decision boundaries in the input space. Multilayer networks, on the other hand, are much more expressive—they can represent general nonlinear functions—but are very hard to train because of the abundance of local minima and the high dimensionality of the weight space. In this section, we will explore a relatively new family of learning methods called **support vector machines** (SVMs) or, more generally, **kernel machines**. To some extent, kernel machines give us the best of both worlds. That is, these methods use an efficient training algorithm *and* can represent complex, nonlinear functions.

SUPPORT VECTOR MACHINE

KERNEL MACHINE

The full treatment of kernel machines is beyond the scope of the book, but we can illustrate the main idea through an example. Figure 20.27(a) shows a two-dimensional input space defined by attributes $\mathbf{x} = (x_1, x_2)$, with positive examples ($y = +1$) inside a circular region and negative examples ($y = -1$) outside. Clearly, there is no linear separator for this problem. Now, suppose we re-express the input data using some computed features—i.e., we map each input vector $\mathbf{x}$ to a new vector of feature values, $F(\mathbf{x})$. In particular, let us use the three features

$$f_1 = x_1^2 , \qquad f_2 = x_2^2 , \qquad f_3 = \sqrt{2} x_1 x_2 . \tag{20.16}$$

We will see shortly where these came from, but, for now, just look at what happens. Figure 20.27(b) shows the data in the new, three-dimensional space defined by the three features; the data are *linearly separable* in this space! This phenomenon is actually fairly general: if data are mapped into a space of sufficiently high dimension, then they will always be linearly separable. Here, we used only three dimensions,[14] but if we have N data points then, except in special cases, they will always be separable in a space of $N-1$ dimensions or more (Exercise 20.21).

So, is that it? Do we just produce loads of computed features and then find a linear separator in the corresponding high-dimensional space? Unfortunately, it's not that easy. Remember that a linear separator in a space of d dimensions is defined by an equation with d parameters, so we are in serious danger of overfitting the data if $d \approx N$, the number of data points. (This is like overfitting data with a high-degree polynomial, which we discussed in Chapter 18.) For this reason, kernel machines usually find the *optimal* linear separator—the one that has the largest **margin** between it and the positive examples on one side and the negative examples on the other. (See Figure 20.28.) It can be shown, using arguments from computational learning theory (Section 18.5), that this separator has desirable properties in terms of robust generalization to new examples.

MARGIN

QUADRATIC PROGRAMMING

Now, how do we find this separator? It turns out that this is a **quadratic programming** optimization problem. Suppose we have examples $\mathbf{x}_i$ with classifications $y_i = \pm 1$ and we want to find an optimal separator in the input space; then the quadratic programming problem

[14] The reader may notice that we could have used just f_1 and f_2, but the 3D mapping illustrates the idea better.

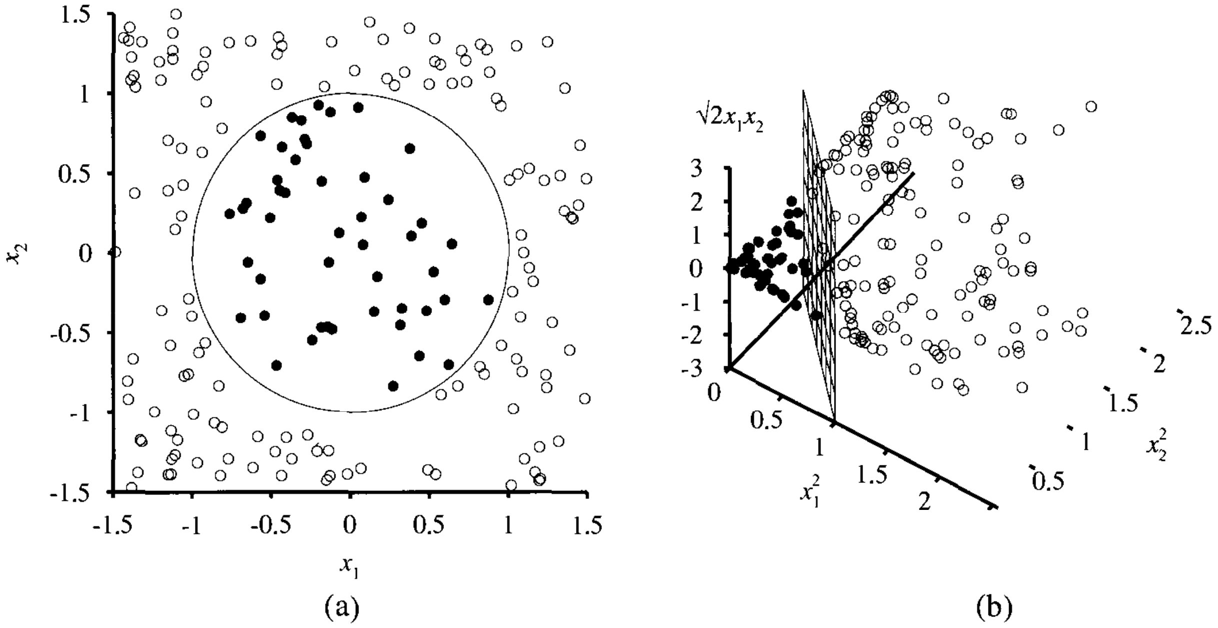


Figure 20.27 (a) A two-dimensional training with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions.

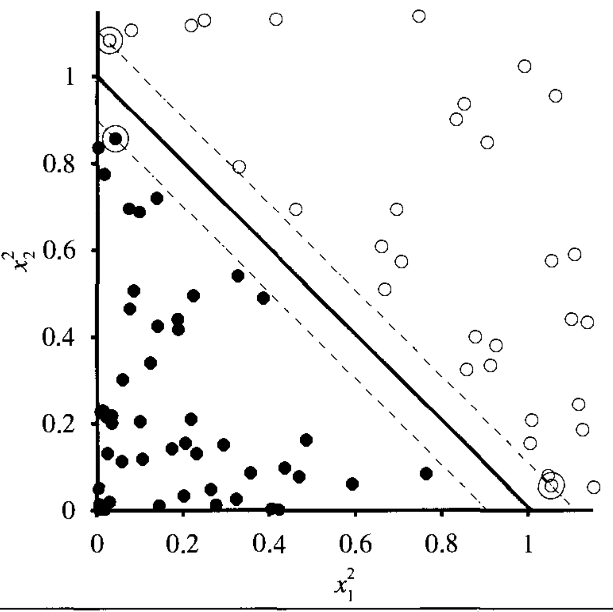


Figure 20.28 A close-up, projected onto the first two dimensions, of the optimal separator shown in Figure 20.27(b). The separator is shown as a heavy line, with the closest points—the **support vectors**—marked with circles. The **margin** is the separation between the positive and negative examples.

is to find values of the parameters α_i that maximize the expression

$$\sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \tag{20.17}$$

subject to the constraints $\alpha_i \geq 0$ and $\sum_i \alpha_i y_i = 0$. Although the derivation of this expression is not crucial to the story, it does have two important properties. First, the expression has a single global maximum that can be found efficiently. Second, *the data enter the expression only in the form of dot products of pairs of points.* This second property is also true of the equation for the separator itself; once the optimal α_is have been calculated, it is

$$h(\mathbf{x}) = \text{sign}\left(\sum_i \alpha_i y_i (\mathbf{x} \cdot \mathbf{x}_i)\right) . \tag{20.18}$$

A final important property of the optimal separator defined by this equation is that the weights α_i associated with each data point are *zero* except for those points closest to the separator—

SUPPORT VECTOR

the so-called **support vectors**. (They are called this because they "hold up" the separating plane.) Because there are usually many fewer support vectors than data points, the effective number of parameters defining the optimal separator is usually much less than N.

Now, we would not usually expect to find a linear separator in the input space $\mathbf{x}$, but it is easy to see that we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_i \cdot \mathbf{x}_j$ in Equation (20.17) with $F(\mathbf{x}_i) \cdot F(\mathbf{x}_j)$. This by itself is not remarkable—replacing $\mathbf{x}$ by $F(\mathbf{x})$ in *any* learning algorithm has the required effect—but the dot product has some special properties. It turns out that $F(\mathbf{x}_i) \cdot F(\mathbf{x}_j)$ can often be computed without first computing F for each point. In our three-dimensional feature space defined by Equation (20.16), a little bit of algebra shows that

$$F(\mathbf{x}_i) \cdot F(\mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2 .$$

The expression $(\mathbf{x}_i \cdot \mathbf{x}_j)^2$ is called a **kernel function**, usually written as $K(\mathbf{x}_i, \mathbf{x}_j)$. In the kernel machine context, this means a function that can be applied to pairs of input data to evaluate dot products in some corresponding feature space. So, we can restate the claim at the beginning of this paragraph as follows: we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_i \cdot \mathbf{x}_j$ in Equation (20.17) with a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$. Thus, we can learn in the high-dimensional space but we compute only kernel functions rather than the full list of features for each data point.

The next step, which should by now be obvious, is to see that there's nothing special about the kernel $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$. It corresponds to a particular higher-dimensional feature space, but other kernel functions correspond to other feature spaces. A venerable

MERCER'S THEOREM

result in mathematics, **Mercer's theorem** (1909), tells us that any "reasonable" [15] kernel function corresponds to *some* feature space. These feature spaces can be very large, even for

POLYNOMIAL KERNEL

innocuous-looking kernels. For example, the **polynomial kernel**, $K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i \cdot \mathbf{x}_j)^d$, corresponds to a feature space whose dimension is exponential in d. Using such kernels in Equation (20.17), then, *optimal linear separators can be found efficiently in feature spaces with billions (or, in some cases, infinitely many) dimensions.* The resulting linear separators,

[15] Here, "reasonable" means that the matrix $\mathbf{K}_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ is positive definite; see Appendix A.

when mapped back to the original input space, can correspond to arbitrarily wiggly, nonlinear boundaries between the positive and negative examples.

We mentioned in the preceding section that kernel machines excel at handwritten digit recognition; they are rapidly being adopted for other applications—especially those with many input features. As part of this process, many new kernels have been designed that work with strings, trees, and other non-numerical data types. It has also been observed that the kernel method can be applied not only with learning algorithms that find optimal linear separators, but also with any other algorithm that can be reformulated to work only with dot products of pairs of data points, as in Equations 20.17 and 20.18. Once this is done, the dot product is replaced by a kernel function and we have a **kernelized** version of the algorithm. This can be done easily for k-nearest-neighbor and perceptron learning, among others.

KERNELIZATION

20.7 CASE STUDY: HANDWRITTEN DIGIT RECOGNITION

Recognizing handwritten digits is an important problem with many applications, including automated sorting of mail by postal code, automated reading of checks and tax returns, and data entry for hand-held computers. It is an area where rapid progress has been made, in part because of better learning algorithms and in part because of the availability of better training sets. The United States National Institute of Science and Technology (**NIST**) has archived a database of 60,000 labeled digits, each $20 \times 20 = 400$ pixels with 8-bit grayscale values. It has become one of the standard benchmark problems for comparing new learning algorithms. Some example digits are shown in Figure 20.29.

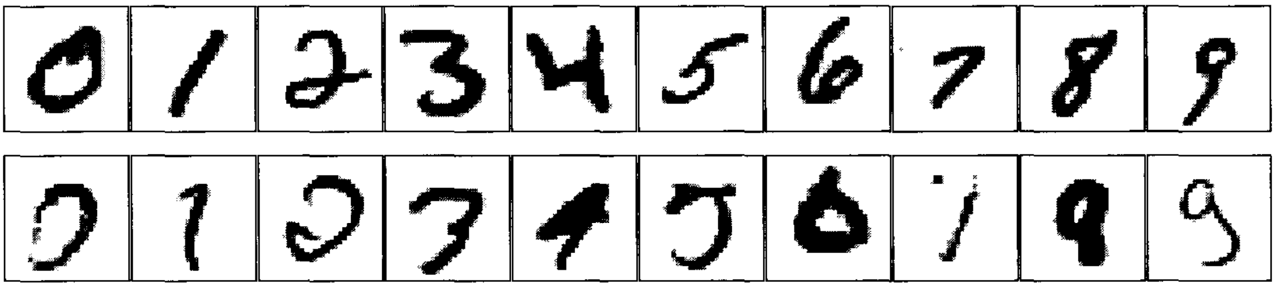


Figure 20.29 Examples from the NIST database of handwritten digits. Top row: examples of digits 0–9 that are easy to identify. Bottom row: more difficult examples of the same digits.

Many different learning approaches have been tried. One of the first, and probably the simplest, is the **3-nearest-neighbor** classifier, which also has the advantage of requiring no training time. As a memory-based algorithm, however, it must store all 60,000 images, and its runtime performance is slow. It achieved a test error rate of 2.4%.

A **single-hidden-layer neural network** was designed for this problem with 400 input units (one per pixel) and 10 output units (one per class). Using cross-validation, it was found that roughly 300 hidden units gave the best performance. With full interconnections between layers, there were a total of 123,300 weights. This network achieved a 1.6% error rate.

A series of **specialized neural networks** called LeNet were devised to take advantage of the structure of the problem—that the input consists of pixels in a two–dimensional array, and that small changes in the position or slant of an image are unimportant. Each network had an input layer of 32×32 units, onto which the 20×20 pixels were centered so that each input unit is presented with a local neighborhood of pixels. This was followed by three layers of hidden units. Each layer consisted of several planes of $n \times n$ arrays, where n is smaller than the previous layer so that the network is down-sampling the input, and where the weights of every unit in a plane are constrained to be identical, so that the plane is acting as a feature detector: it can pick out a feature such as a long vertical line or a short semi-circular arc. The output layer had 10 units. Many versions of this architecture were tried; a representative one had hidden layers with 768, 192, and 30 units, respectively. The training set was augmented by applying affine transformations to the actual inputs: shifting, slightly rotating, and scaling the images. (Of course, the transformations have to be small, or else a 6 will be transformed into a 9!) The best error rate achieved by LeNet was 0.9%.

A **boosted neural network** combined three copies of the LeNet architecture, with the second one trained on a mix of patterns that the first one got 50% wrong, and the third one trained on patterns for which the first two disagreed. During testing, the three nets voted with their weights for each of the ten digits, and the scores are added to determine the winner. The test error rate was 0.7%.

A **support vector machine** (see Section 20.6) with 25,000 support vectors achieved an error rate of 1.1%. This is remarkable because the SVM technique, like the simple nearest-neighbor approach, required almost no thought or iterated experimentation on the part of the developer, yet it still came close to the performance of LeNet, which had had years of development. Indeed, the support vector machine makes no use of the structure of the problem, and would perform just as well if the pixels were presented in a permuted order.

VIRTUAL SUPPORT VECTOR MACHINE

A **virtual support vector machine** starts with a regular SVM and then improves it with a technique that is designed to take advantage of the structure of the problem. Instead of allowing products of all pixel pairs, this approach concentrates on kernels formed from pairs of nearby pixels. It also augments the training set with transformations of the examples, just as LeNet did. A virtual SVM achieved the best error rate recorded to date, 0.56%.

Shape matching is a technique from computer vision used to align corresponding parts of two different images of objects. (See Chapter 24.) The idea is to pick out a set of points in each of the two images, and then compute, for each point in the first image, which point in the second image it corresponds to. From this alignment, we then compute a transformation between the images. The transformation gives us a measure of the distance between the images. This distance measure is better motivated than just counting the number of differing pixels, and it turns out that a 3–nearest neighbor algorithm using this distance measure performs very well. Training on only 20,000 of the 60,000 digits, and using 100 sample points per image extracted from a Canny edge detector, a shape matching classifier achieved 0.63% test error.

Humans are estimated to have an error rate of about 0.2% on this problem. This figure is somewhat suspect because humans have not been tested as extensively as have machine learning algorithms. On a similar data set of digits from the United States Postal Service, human errors were at 2.5%.

The following figure summarizes the error rates, runtime performance, memory requirements, and amount of training time for the seven algorithms we have discussed. It also adds another measure, the percentage of digits that must be rejected to achieve 0.5% error. For example, if the SVM is allowed to reject 1.8% of the inputs—that is, pass them on for someone else to make the final judgment—then its error rate on the remaining 98.2% of the inputs is reduced from 1.1% to 0.5%.

The following table summarizes the error rate and some of the other characteristics of the seven techniques we have discussed.

	3	300		Boosted		Virtual	Shape
	NN	Hidden	LeNet	LeNet	SVM	SVM	Match
Error rate (pct.)	2.4	1.6	0.9	0.7	1.1	0.56	0.63
Runtime (millisec/digit)	1000	10	30	50	2000	200	
Memory requirements (Mbyte)	12	.49	.012	.21	11		
Training time (days)	0	7	14	30	10		
% rejected to reach 0.5% error	8.1	3.2	1.8	0.5	1.8		

20.8 Summary

Statistical learning methods range from simple calculation of averages to the construction of complex models such as Bayesian networks and neural networks. They have applications throughout computer science, engineering, neurobiology, psychology, and physics. This chapter has presented some of the basic ideas and given a flavor of the mathematical underpinnings. The main points are as follows:

- **Bayesian learning** methods formulate learning as a form of probabilistic inference, using the observations to update a prior distribution over hypotheses. This approach provides a good way to implement Ockham's razor, but quickly becomes intractable for complex hypothesis spaces.
- **Maximum a posteriori** (MAP) learning selects a single most likely hypothesis given the data. The hypothesis prior is still used and the method is often more tractable than full Bayesian learning.
- **Maximum likelihood** learning simply selects the hypothesis that maximizes the likelihood of the data; it is equivalent to MAP learning with a uniform prior. In simple cases such as linear regression and fully observable Bayesian networks, maximum likelihood solutions can be found easily in closed form. **Naive Bayes** learning is a particularly effective technique that scales well.
- When some variables are hidden, local maximum likelihood solutions can be found using the EM algorithm. Applications include clustering using mixtures of Gaussians, learning Bayesian networks, and learning hidden Markov models.

- Learning the structure of Bayesian networks is an example of **model selection**. This usually involves a discrete search in the space of structures. Some method is required for trading off model complexity against degree of fit.
- **Instance-based models** represent a distribution using the collection of training instances. Thus, the number of parameters grows with the training set. **Nearest-neighbor** methods look at the instances nearest to the point in question, whereas **kernel** methods form a distance-weighted combination of all the instances.
- **Neural networks** are complex nonlinear functions with many parameters. Their parameters can be learned from noisy data and they have been used for thousands of applications.
- A **perceptron** is a feed-forward neural network with no hidden units that can represent only **linearly separable** functions. If the data are linearly separable, a simple weight-update rule can be used to fit the data exactly.
- **Multilayer feed-forward** neural networks can represent any function, given enough units. The **back-propagation** algorithm implements a gradient descent in parameter space to minimize the output error.

Statistical learning continues to be a very active area of research. Enormous strides have been made in both theory and practice, to the point where it is possible to learn almost any model for which exact or approximate inference is feasible.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The application of statistical learning techniques in AI was an active area of research in the early years (see Duda and Hart, 1973) but became separated from mainstream AI as the latter field concentrated on symbolic methods. It continued in various forms—some explicitly probabilistic, others not—in areas such as **pattern recognition** (Devroye *et al.*, 1996) and **information retrieval** (Salton and McGill, 1983). A resurgence of interest occurred shortly after the introduction of Bayesian network models in the late 1980s; at roughly the same time, a statistical view of neural network learning began to emerge. In the late 1990s, there was a noticeable convergence of interests in machine learning, statistics, and neural networks, centered on methods for creating large probabilistic models from data.

The naive Bayes model is one of the oldest and simplest forms of Bayesian network, dating back to the 1950s. Its origins were mentioned in the notes at the end of Chapter 13. is partially explained by Domingos and Pazzani (1997). A boosted form of naive Bayes learning won the first KDD Cup data mining competition (Elkan, 1997). Heckerman (1998) gives an excellent introduction to the general problem of Bayes net learning. Bayesian parameter learning with Dirichlet priors for Bayesian networks was discussed by Spiegelhalter *et al.* (1993). The BUGS software package (Gilks *et al.*, 1994) incorporates many of these ideas and provides a very powerful tool for formulating and learning complex probability models. The first algorithms for learning Bayes net structures used conditional independence tests (Pearl, 1988; Pearl and Verma, 1991). Spirtes *et al.* (1993) developed a comprehensive approach and the TETRAD package for Bayes net learning using similar ideas. Algorithmic improvements since then led to a clear victory in the 2001 KDD Cup data mining competition for a Bayes net learning method (Cheng *et al.*, 2002). (The specific task here was a bioinformatics problem with 139,351 features!) A structure-learning approach based on maximizing likelihood was developed by Cooper and Herskovits (1992) and improved by Heckerman *et al.* (1994). Friedman and Goldszmidt (1996) pointed out the influence of the representation of local conditional distributions on the learned structure.

The general problem of learning probability models with hidden variables and missing data was addressed by the EM algorithm (Dempster *et al.*, 1977), which was abstracted from several existing methods including the Baum–Welch algorithm for HMM learning (Baum and Petrie, 1966). (Dempster himself views EM as a schema rather than an algorithm, since a good deal of mathematical work may be required before it can be applied to a new family of distributions.) EM is now one of the most widely used algorithms in science, and McLachlan and Krishnan (1997) devote an entire book to the algorithm and its properties. The specific problem of learning mixture models, including mixtures of Gaussians, is covered by Titterington *et al.* (1985). Within AI, the first successful system that used EM for mixture modeling was AUTOCLASS (Cheeseman *et al.*, 1988; Cheeseman and Stutz, 1996). AUTOCLASS has been applied to a number of real-world scientific classification tasks, including the discovery of new types of stars from spectral data (Goebel *et al.*, 1989) and new classes of proteins and introns in DNA/protein sequence databases (Hunter and States, 1992).

An EM algorithm for learning Bayes nets with hidden variables was developed by Lauritzen (1995). Gradient-based techniques have also proved effective for Bayes nets as well as dynamic Bayes nets (Russell *et al.*, 1995; Binder *et al.*, 1997a). The structural EM algorithm was developed by (Friedman, 1998). The ability to learn the structure of Bayesian networks is closely connected to the issue of recovering *causal* information from data. That is, is it possible to learn Bayes nets in such a way that the recovered network structure indicates real causal influences? For many years, statisticians avoided this question, believing that observational data (as opposed to data generated from experimental trials) could yield only correlational information—after all, any two variables that appear related might in fact be influenced by third, unknown causal factor rather than influencing each other directly. Pearl (2000) has presented convincing arguments to the contrary, showing that there are in fact many cases where causality can be ascertained and developing the **causal network** formalism to express causes and the effects of intervention as well as ordinary conditional probabilities.

CAUSAL NETWORK

Nearest-neighbor models date back at least to (Fix and Hodges, 1951) and have been a standard tool in statistics and pattern recognition ever since. Within AI, they were popularized by (Stanfill and Waltz, 1986), who investigated methods for adapting the distance metric to the data. Hastie and Tibshirani (1996) developed a way to localize the metric to each point in the space, depending on the distribution of data around that point. Efficient indexing schemes for finding nearest neighbors are studied within the algorithms community (see, e.g., Indyk, 2000). Kernel density estimation, also called **Parzen window** density estimation, was investigated initially by Rosenblatt (1956) and Parzen (1962). Since that time, a huge literature has developed investigating the properties of various estimators. Devroye (1987) gives a thorough introduction.

The literature on neural networks is rather too large (approximately 100,000 papers to date) to cover in detail. Cowan and Sharp (1988b, 1988a) survey the early history, beginning with the work of McCulloch and Pitts (1943). Norbert Wiener, a pioneer of cybernetics and control theory (Wiener, 1948), worked with McCulloch and Pitts and influenced a number of young researchers including Marvin Minsky, who may have been the first to develop a working neural network in hardware in 1951 (see Minsky and Papert, 1988, pp. ix–x). Meanwhile, in Britain, W. Ross Ashby (also a pioneer of cybernetics; see Ashby, 1940), Alan Turing, Grey Walter, and others formed the Ratio Club for "those who had Wiener's ideas before Wiener's book appeared." Ashby's *Design for a Brain* (1948, 1952) put forth the idea that intelligence could be created by the use of **homeostatic** devices containing appropriate feedback loops to achieve stable adaptive behavior. Turing (1948) wrote a research report titled *Intelligent Machinery* that begins with the sentence "I propose to investigate the question as to whether it is possible for machinery to show intelligent behaviour" and goes on to describe a recurrent neural network architecture he called "B-type unorganized machines" and an approach to training them. Unfortunately, the report went unpublished until 1969, and was all but ignored until recently.

Frank Rosenblatt (1957) invented the modern "perceptron" and proved the perceptron convergence theorem (1960), although it had been foreshadowed by purely mathematical work outside the context of neural networks (Agmon, 1954; Motzkin and Schoenberg, 1954). Some early work was also done on multilayer networks, including **Gamba perceptrons** (Gamba *et al.*, 1961) and **madalines** (Widrow, 1962). *Learning Machines* (Nilsson, 1965) covers much of this early work and more. The subsequent demise of early perceptron research efforts was hastened (or, the authors later claimed, merely explained) by the book *Perceptrons* (Minsky and Papert, 1969), which lamented the field's lack of mathematical rigor. The book pointed out that single-layer perceptrons could represent only linearly separable concepts and noted the lack of effective learning algorithms for multilayer networks.

The papers in (Hinton and Anderson, 1981), based on a conference in San Diego in 1979, can be regarded as marking the renaissance of connectionism. The two-volume "PDP" (Parallel Distributed Processing) anthology (Rumelhart *et al.*, 1986a) and a short article in *Nature* (Rumelhart *et al.*, 1986b) attracted a great deal of attention—indeed, the number of papers on "neural networks" multiplied by a factor of 200 between 1980–84 and 1990–94. The analysis of neural networks using the physical theory of magnetic spin glasses (Amit *et al.*, 1985) tightened the links between statistical mechanics and neural network theory—providing not only useful mathematical insights but also *respectability*. The back-propagation technique had been invented quite early (Bryson and Ho, 1969) but it was rediscovered several times (Werbos, 1974; Parker, 1985).

Support vector machines were originated in the 1990s (Cortes and Vapnik, 1995) and are now the subject of a fast-growing literature, including textbooks such as Cristianini and Shawe-Taylor (2000). They have proven to be very popular and effective for tasks such as text categorization (Joachims, 2001), bioinformatics research (Brown *et al.*, 2000), and natural language processing, such as the handwritten digit recognition of DeCoste and Scholkopf (2002). A related technique that also uses the "kernel trick" to implicitly represent an exponential feature space is the voted perceptron (Collins and Duffy, 2002).

The probabilistic interpretation of neural networks has several sources, including Baum and Wilczek (1988) and Bridle (1990). The role of the sigmoid function is discussed by Jordan (1995). Bayesian parameter learning for neural networks was proposed by MacKay (1992) and is explored further by Neal (1996). The capacity of neural networks to represent functions was investigated by Cybenko (1988, 1989), who showed that two hidden layers are enough to represent any function and a single layer is enough to represent any *continuous* function. The "optimal brain damage" method for removing useless connections is by LeCun et al. (1989), and Sietsma and Dow (1988) show how to remove useless units. The tiling algorithm for growing larger structures is due to Mézard and Nadal (1989). LeCun *et al.* (1995) survey a number of algorithms for handwritten digit recognition. Improved error rates since then were reported by Belongie *et al.* (2002) for shape matching and DeCoste and Schölkopf (2002) for virtual support vectors.

The complexity of neural network learning has been investigated by researchers in computational learning theory. Early computational results were obtained by Judd (1990), who showed that the general problem of finding a set of weights consistent with a set of examples is NP-complete, even under very restrictive assumptions. Some of the first sample complexity results were obtained by Baum and Haussler (1989), who showed that the number of examples required for effective learning grows as roughly $W \log W$, where W is the number of weights.[16] Since then, a much more sophisticated theory has been developed (Anthony and Bartlett, 1999), including the important result that the representational capacity of a network depends on the *size* of the weights as well as on their number.

RADIAL BASIS FUNCTION

The most popular kind of neural network that we did not cover is the **radial basis function**, or RBF, network. A radial basis function combines a weighted collection of kernels (usually Gaussians, of course) to do function approximation. RBF networks can be trained in two phases: first, an unsupervised clustering approach is used to train the parameters of the Gaussians—the means and variances—are trained, as in Section 20.3. In the second phase, the relative weights of the Gaussians are determined. This is a system of linear equations, which we know how to solve directly. Thus, both phases of RBF training have a nice benefit: the first phase is unsupervised, and thus does not require labelled training data, and the second phase, although supervised, is efficient. See Bishop (1995) for more details.

HOPFIELD NETWORKS

ASSOCIATIVE MEMORY

Recurrent networks, in which units are linked in cycles, were mentioned in the chapter but not explored in depth. **Hopfield networks** (Hopfield, 1982) are probably the best-understood class of recurrent networks. They use *bidirectional* connections with *symmetric* weights (i.e., $W_{i,j} = W_{j,i}$), all of the units are both input and output units, the activation function g is the sign function, and the activation levels can only be ± 1. A Hopfield network functions as an **associative memory**: after the network trains on a set of examples, a new stimulus will cause it to settle into an activation pattern corresponding to the example in the training set that *most closely resembles* the new stimulus. For example, if the training set consists of a set of photographs, and the new stimulus is a small piece of one of the photographs, then the network activation levels will reproduce the photograph from which the piece was

[16] This approximately confirmed "Uncle Bernie's rule." The rule was named after Bernie Widrow, who recommended using roughly ten times as many examples as weights.

taken. Notice that the original photographs are not stored separately in the network; each weight is a partial encoding of all the photographs. One of the most interesting theoretical results is that Hopfield networks can reliably store up to $0.138N$ training examples, where N is the number of units in the network.

BOLTZMANN MACHINES

Boltzmann machines (Hinton and Sejnowski, 1983, 1986) also use symmetric weights, but include hidden units. In addition, they use a *stochastic* activation function, such that the probability of the output being 1 is some function of the total weighted input. Boltzmann machines therefore undergo state transitions that resemble a simulated annealing search (see Chapter 4) for the configuration that best approximates the training set. It turns out that Boltzmann machines are very closely related to a special case of Bayesian networks evaluated with a stochastic simulation algorithm. (See Section 14.5.)

The first application of the ideas underlying kernel machines was by Aizerman *et al.* (1964), but the full development of the theory, under the heading of support vector machines, is due to Vladimir Vapnik and colleagues (Boser *et al.*, 1992; Vapnik, 1998). Cristianini and Shawe-Taylor (2000) and Schölkopf and Smola (2002) provide rigorous introductions; a friendlier exposition appears in the *AI Magazine* article by Cristianini and Schölkopf (2002).

The material in this chapter brings together work from the fields of statistics, pattern recognition, and neural networks, so the story has been told many times in many ways. Good texts on Bayesian statistics include those by DeGroot (1970), Berger (1985), and Gelman *et al.* (1995). Hastie *et al.* (2001) provide an excellent introduction to statistical learning methods. For pattern classification, the classic text for many years has been Duda and Hart (1973), now updated (Duda *et al.*, 2001). For neural nets, Bishop (1995) and Ripley (1996) are the leading texts. The field of computational neuroscience is covered by Dayan and Abbott (2001). The most important conference on neural networks and related topics is the annual NIPS (Neural Information Processing Conference) conference, whose proceedings are published as the series *Advances in Neural Information Processing Systems*. Papers on learning Bayesian networks also appear in the *Uncertainty in AI* and *Machine Learning* conferences and in several statistics conferences. Journals specific to neural networks include *Neural Computation*, *Neural Networks*, and the *IEEE Transactions on Neural Networks*.

EXERCISES

20.1 The data used for Figure 20.1 can be viewed as being generated by h_5. For each of the other four hypotheses, generate a data set of length 100 and plot the corresponding graphs for $P(h_i|d_1, \ldots, d_m)$ and $P(D_{m+1} = lime|d_1, \ldots, d_m)$. Comment on your results.

20.2 Repeat Exercise 20.1, this time plotting the values of $P(D_{m+1} = lime|h_{\mathrm{MAP}})$ and $P(D_{m+1} = lime|h_{\mathrm{ML}})$.

20.3 Suppose that Ann's utilities for cherry and lime candies are c_A and ℓ_A, whereas Bob's utilities are c_B and ℓ_B. (But once Ann has unwrapped a piece of candy, Bob won't buy it.) Presumably, if Bob likes lime candies much more than Ann, it would be wise to sell for Ann to sell her bag of candies once she is sufficiently sure of its lime content. On the other hand, if Ann unwraps too many candies in the process, the bag will be worth less. Discuss the problem of determining the optimal point at which to sell the bag. Determine the expected utility of the optimal procedure, given the prior distribution from Section 20.1.

20.4 Two statisticians go to the doctor and are both given the same prognosis: A 40% chance that the problem is the deadly disease A, and a 60% chance of the fatal disease B. Fortunately, there are anti-A and anti-B drugs that are inexpensive, 100% effective, and free of side-effects. The statisticians have the choice of taking one drug, both, or neither. What will the first statistician (an avid Bayesian) do? How about the second statistician, who always uses the maximum likelihood hypothesis?

The doctor does some research and discovers that disease B actually comes in two versions, dextro-B and levo-B, which are equally likely and equally treatable by the anti-B drug. Now that there are three hypotheses, what will the two statisticians do?

20.5 Explain how to apply the boosting method of Chapter 18 to naive Bayes learning. Test the performance of the resulting algorithm on the restaurant learning problem.

20.6 Consider m data points (x_j, y_j), where the y_js are generated from the x_js according to the linear Gaussian model in Equation (20.5). Find the values of θ_1, θ_2, and σ that maximize the conditional log likelihood of the data.

20.7 Consider the noisy-OR model for fever described in Section 14.3. Explain how to apply maximum-likelihood learning to fit the parameters of such a model to a set of complete data. (*Hint*: use the chain rule for partial derivatives.)

20.8 This exercise investigates properties of the Beta distribution defined in Equation (20.6).

a. By integrating over the range $[0, 1]$, show that the normalization constant for the distribution beta$[a, b]$ is given by $\alpha = \Gamma(a+b)/\Gamma(a)\Gamma(b)$ where $\Gamma(x)$ is the **Gamma function**, defined by $\Gamma(x+1) = x \cdot \Gamma(x)$ and $\Gamma(1) = 1$. (For integer x, $\Gamma(x+1) = x!$.) GAMMA FUNCTION

b. Show that the mean is $a/(a+b)$.

c. Find the mode(s) (the most likely value(s) of θ).

d. Describe the distribution beta$[\epsilon, \epsilon]$ for very small ϵ. What happens as such a distribution is updated?

20.9 Consider an arbitrary Bayesian network, a complete data set for that network, and the likelihood for the data set according to the network. Give a simple proof that the likelihood of the data cannot decrease if we add a new link to the network and recompute the maximum-likelihood parameter values.

20.10 Consider the application of EM to learn the parameters for the network in Figure 20.10(a), given the true parameters in Equation (20.7).

a. Explain why the EM algorithm would not work if there were just two attributes in the model rather than three.

b. Show the calculations for the first iteration of EM starting from Equation (20.8).

c. What happens if we start with all the parameters set to the same value p? (*Hint*: you may find it helpful to investigate this empirically before deriving the general result.)

d. Write out an expression for the log likelihood of the tabulated candy data on page 729 in terms of the parameters, calculate the partial derivatives with respect to each parameter, and investigate the nature of the fixed point reached in part (c).

20.11 Construct by hand a neural network that computes the XOR function of two inputs. Make sure to specify what sort of units you are using.

20.12 Construct a support vector machine that computes the XOR function. It will be convenient to use values of 1 and –1 instead of 1 and 0 for the inputs and for the outputs. So an example looks like $([-1, 1], 1)$ or $([-1, -1], -1)$. It is typical to map an input **x** into a space consisting of five dimensions, the two original dimensions x_1 and x_2, and the three combination x_1^2, x_2^2 and $x_1\, x_2$. But for this exercise we will consider only the two dimensions x_1 and $x_1\, x_2$. Draw the four input points in this space, and the maximal margin separator. What is the margin? Now draw the separating line back in the original Euclidean input space.

20.13 A simple perceptron cannot represent XOR (or, generally, the parity function of its inputs). Describe what happens to the weights of a four-input, step-function perceptron, beginning with all weights set to 0.1, as examples of the parity function arrive.

20.14 Recall from Chapter 18 that there are 2^{2^n} distinct Boolean functions of n inputs. How many of these are representable by a threshold perceptron?

20.15 Consider the following set of examples, each with six inputs and one target output:

I_1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
I_2	0	0	0	1	1	0	0	1	1	0	1	0	1	1
I_3	1	1	1	0	1	0	0	1	1	0	0	0	1	1
I_4	0	1	0	0	1	0	0	1	0	1	1	1	0	1
I_5	0	0	1	1	0	1	1	0	1	1	0	0	1	0
I_6	0	0	0	1	0	1	0	1	1	0	1	1	1	0
T	1	1	1	1	1	1	0	1	0	0	0	0	0	0

a. Run the perceptron learning rule on these data and show the final weights.

b. Run the decision tree learning rule, and show the resulting decision tree.

c. Comment on your results.

20.16 Starting from Equation (20.13), show that $\partial L/\partial W_j = Err \times a_j$.

20.17 Suppose you had a neural network with linear activation functions. That is, for each unit the output is some constant c times the weighted sum of the inputs.

a. Assume that the network has one hidden layer. For a given assignment to the weights **W**, write down equations for the value of the units in the output layer as a function of **W** and the input layer **I**, without any explicit mention to the output of the hidden layer. Show that there is a network with no hidden units that computes the same function.

b. Repeat the calculation in part (a), this time for a network with any number of hidden layers. What can you conclude about linear activation functions?

20.18 Implement a data structure for layered, feed-forward neural networks, remembering to provide the information needed for both forward evaluation and backward propagation. Using this data structure, write a function NEURAL-NETWORK-OUTPUT that takes an example and a network and computes the appropriate output values.

20.19 Suppose that a training set contains only a single example, repeated 100 times. In 80 of the 100 cases, the single output value is 1; in the other 20, it is 0. What will a back-propagation network predict for this example, assuming that it has been trained and reaches a global optimum? (*Hint:* to find the global optimum, differentiate the error function and set to zero.)

20.20 The network in Figure 20.24 has four hidden nodes. This number was chosen somewhat arbitrarily. Run systematic experiments to measure the learning curves for networks with different numbers of hidden nodes. What is the optimal number? Would it be possible to use a cross-validation method to find the best network before the fact?

20.21 Consider the problem of separating N data points into positive and negative examples using a linear separator. Clearly, this can always be done for $N = 2$ points on a line of dimension $d = 1$, regardless of how the points are labelled or where they are located (unless the points are in the same place).

a. Show that it can always be done for $N = 3$ points on a plane of dimension $d = 2$, unless they are collinear.

b. Show that it cannot always be done for $N = 4$ points on a plane of dimension $d = 2$.

c. Show that it can always be done for $N = 4$ points in a space of dimension $d = 3$, unless they are coplanar.

d. Show that it cannot always be done for $N = 5$ points in a space of dimension $d = 3$.

e. The ambitious student may wish to prove that N points in general position (but not $N + 1$ are linearly separable in a space of dimension $N - 1$. From this it follows that the **VC dimension** (see Chapter 18) of linear halfspaces in dimension $N - 1$ is N.