# FUTURE VISION BIE

## One Stop for All Study Materials

## & Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

**Visit : https://hemanthrajhemu.github.io**

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

**Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM**

**Contact: MAIL: futurevisionbie@gmail.com**

**INSTAGRAM: www.instagram.com/hemanthraj_hemu/**

**INSTAGRAM: www.instagram.com/futurevisionbie/**

**WHATSAPP SHARE: https://bit.ly/FVBIESHARE**

# MACHINE LEARNING

TOM M. MITCHELL

# CHAPTER 5

# EVALUATING HYPOTHESES

Empirically evaluating the accuracy of hypotheses is fundamental to machine learning. This chapter presents an introduction to statistical methods for estimating hypothesis accuracy, focusing on three questions. First, given the observed accuracy of a hypothesis over a limited sample of data, how well does this estimate its accuracy over additional examples? Second, given that one hypothesis outperforms another over some sample of data, how probable is it that this hypothesis is more accurate in general? Third, when data is limited what is the best way to use this data to both learn a hypothesis and estimate its accuracy? Because limited samples of data might misrepresent the general distribution of data, estimating true accuracy from such samples can be misleading. Statistical methods, together with assumptions about the underlying distributions of data, allow one to bound the difference between observed accuracy over the sample of available data and the true accuracy over the entire distribution of data.

## 5.1 MOTIVATION

In many cases it is important to evaluate the performance of learned hypotheses as precisely as possible. One reason is simply to understand whether to use the hypothesis. For instance, when learning from a limited-size database indicating the effectiveness of different medical treatments, it is important to understand as precisely as possible the accuracy of the learned hypotheses. A second reason is that evaluating hypotheses is an integral component of many learning methods. For example, in post-pruning decision trees to avoid overfitting, we must evaluate

the impact of possible pruning steps on the accuracy of the resulting decision tree. Therefore it is important to understand the likely errors inherent in estimating the accuracy of the pruned and unpruned tree.

Estimating the accuracy of a hypothesis is relatively straightforward when data is plentiful. However, when we must learn a hypothesis and estimate its future accuracy given only a limited set of data, two key difficulties arise:

- *Bias in the estimate.* First, the observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples. This is especially likely when the learner considers a very rich hypothesis space, enabling it to overfit the training examples. To obtain an unbiased estimate of future accuracy, we typically test the hypothesis on some set of test examples chosen independently of the training examples and the hypothesis.
- *Variance in the estimate.* Second, even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, depending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

This chapter discusses methods for evaluating learned hypotheses, methods for comparing the accuracy of two hypotheses, and methods for comparing the accuracy of two learning algorithms when only limited data is available. Much of the discussion centers on basic principles from statistics and sampling theory, though the chapter assumes no special background in statistics on the part of the reader. The literature on statistical tests for hypotheses is very large. This chapter provides an introductory overview that focuses only on the issues most directly relevant to learning, evaluating, and comparing hypotheses.

## 5.2 ESTIMATING HYPOTHESIS ACCURACY

When evaluating a learned hypothesis we are most often interested in estimating the accuracy with which it will classify future instances. At the same time, we would like to know the probable error in this accuracy estimate (i.e., what error bars to associate with this estimate).

Throughout this chapter we consider the following setting for the learning problem. There is some space of possible instances $X$ (e.g., the set of all people) over which various target functions may be defined (e.g., people who plan to purchase new skis this year). We assume that different instances in $X$ may be encountered with different frequencies. A convenient way to model this is to assume there is some unknown probability distribution $\mathcal{D}$ that defines the probability of encountering each instance in $X$ (e.g., $\mathcal{D}$ might assign a higher probability to encountering 19-year-old people than 109-year-old people). Notice $\mathcal{D}$ says nothing

about whether $x$ is a positive or negative example; it only determines the probability that $x$ will be encountered. The learning task is to learn the target concept or target function $f$ by considering a space $H$ of possible hypotheses. Training examples of the target function $f$ are provided to the learner by a trainer who draws each instance independently, according to the distribution $\mathcal{D}$, and who then forwards the instance $x$ along with its correct target value $f(x)$ to the learner.

To illustrate, consider learning the target function "people who plan to purchase new skis this year," given a sample of training data collected by surveying people as they arrive at a ski resort. In this case the instance space $X$ is the space of all people, who might be described by attributes such as their age, occupation, how many times they skied last year, etc. The distribution $\mathcal{D}$ specifies for each person $x$ the probability that $x$ will be encountered as the next person arriving at the ski resort. The target function $f : X \rightarrow \{0, 1\}$ classifies each person according to whether or not they plan to purchase skis this year.

Within this general setting we are interested in the following two questions:

1. Given a hypothesis $h$ and a data sample containing $n$ examples drawn at random according to the distribution $\mathcal{D}$, what is the best estimate of the accuracy of $h$ over future instances drawn from the same distribution?
2. What is the probable error in this accuracy estimate?

### 5.2.1 Sample Error and True Error

To answer these questions, we need to distinguish carefully between two notions of accuracy or, equivalently, error. One is the error rate of the hypothesis over the sample of data that is available. The other is the error rate of the hypothesis over the entire unknown distribution $\mathcal{D}$ of examples. We will call these the *sample error* and the *true error* respectively.

The *sample error* of a hypothesis with respect to some sample $S$ of instances drawn from $X$ is the fraction of $S$ that it misclassifies:

> ***Definition:*** The **sample error** (denoted $error_S(h)$) of hypothesis $h$ with respect to target function $f$ and data sample $S$ is
>
> $$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$
>
> Where $n$ is the number of examples in $S$, and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise.

The *true error* of a hypothesis is the probability that it will misclassify a single randomly drawn instance from the distribution $\mathcal{D}$.

> ***Definition:*** The **true error** (denoted $error_{\mathcal{D}}(h)$) of hypothesis $h$ with respect to target function $f$ and distribution $\mathcal{D}$, is the probability that $h$ will misclassify an instance drawn at random according to $\mathcal{D}$.
>
> $$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

Here the notation $\Pr_{x \in \mathcal{D}}$ denotes that the probability is taken over the instance distribution $\mathcal{D}$.

What we usually wish to know is the true error $error_{\mathcal{D}}(h)$ of the hypothesis, because this is the error we can expect when applying the hypothesis to future examples. All we can measure, however, is the sample error $error_S(h)$ of the hypothesis for the data sample $S$ that we happen to have in hand. The main question considered in this section is "How good an estimate of $error_{\mathcal{D}}(h)$ is provided by $error_S(h)$?"

### 5.2.2 Confidence Intervals for Discrete-Valued Hypotheses

Here we give an answer to the question "How good an estimate of $error_{\mathcal{D}}(h)$ is provided by $error_S(h)$?" for the case in which $h$ is a discrete-valued hypothesis. More specifically, suppose we wish to estimate the true error for some discrete-valued hypothesis $h$, based on its observed sample error over a sample $S$, where

- the sample $S$ contains $n$ examples drawn independent of one another, and independent of $h$, according to the probability distribution $\mathcal{D}$
- $n \geq 30$
- hypothesis $h$ commits $r$ errors over these $n$ examples (i.e., $error_S(h) = r/n$).

Under these conditions, statistical theory allows us to make the following assertions:

1. Given no other information, the most probable value of $error_{\mathcal{D}}(h)$ is $error_S(h)$
2. With approximately 95% probability, the true error $error_{\mathcal{D}}(h)$ lies in the interval

$$error_S(h) \pm 1.96\sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

To illustrate, suppose the data sample $S$ contains $n = 40$ examples and that hypothesis $h$ commits $r = 12$ errors over this data. In this case, the sample error $error_S(h) = 12/40 = .30$. Given no other information, the best estimate of the true error $error_{\mathcal{D}}(h)$ is the observed sample error .30. However, we do not expect this to be a perfect estimate of the true error. If we were to collect a second sample $S'$ containing 40 new randomly drawn examples, we might expect the sample error $error_{S'}(h)$ to vary slightly from the sample error $error_S(h)$. We expect a difference due to the random differences in the makeup of $S$ and $S'$. In fact, if we repeated this experiment over and over, each time drawing a new sample $S_i$ containing 40 new examples, we would find that for approximately 95% of these experiments, the calculated interval would contain the true error. For this reason, we call this interval the 95% confidence interval estimate for $error_{\mathcal{D}}(h)$. In the current example, where $r = 12$ and $n = 40$, the 95% confidence interval is, according to the above expression, $0.30 \pm (1.96 \cdot .07) = 0.30 \pm .14$.

| Confidence level $N\%$: | 50% | 68% | 80% | 90% | 95% | 98% | 99% |
|---|---|---|---|---|---|---|---|
| Constant $z_N$: | 0.67 | 1.00 | 1.28 | 1.64 | 1.96 | 2.33 | 2.58 |

**TABLE 5.1**
Values of $z_N$ for two-sided $N\%$ confidence intervals.

The above expression for the 95% confidence interval can be generalized to any desired confidence level. The constant 1.96 is used in case we desire a 95% confidence interval. A different constant, $z_N$, is used to calculate the $N\%$ confidence interval. The general expression for approximate $N\%$ confidence intervals for $error_{\mathcal{D}}(h)$ is

$$error_S(h) \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}} \tag{5.1}$$

where the constant $z_N$ is chosen depending on the desired confidence level, using the values of $z_N$ given in Table 5.1.

Thus, just as we could calculate the 95% confidence interval for $error_{\mathcal{D}}(h)$ to be $0.30 \pm (1.96 \cdot .07)$ (when $r = 12$, $n = 40$), we can calculate the 68% confidence interval in this case to be $0.30 \pm (1.0 \cdot .07)$. Note it makes intuitive sense that the 68% confidence interval is smaller than the 95% confidence interval, because we have reduced the probability with which we demand that $error_{\mathcal{D}}(h)$ fall into the interval.

Equation (5.1) describes how to calculate the confidence intervals, or error bars, for estimates of $error_{\mathcal{D}}(h)$ that are based on $error_S(h)$. In using this expression, it is important to keep in mind that this applies only to discrete-valued hypotheses, that it assumes the sample $S$ is drawn at random using the same distribution from which future data will be drawn, and that it assumes the data is independent of the hypothesis being tested. We should also keep in mind that the expression provides only an approximate confidence interval, though the approximation is quite good when the sample contains at least 30 examples, and $error_S(h)$ is not too close to 0 or 1. A more accurate rule of thumb is that the above approximation works well when

$$n\, error_S(h)(1 - error_S(h)) \geq 5$$

Above we summarized the procedure for calculating confidence intervals for discrete-valued hypotheses. The following section presents the underlying statistical justification for this procedure.

## 5.3 BASICS OF SAMPLING THEORY

This section introduces basic notions from statistics and sampling theory, including probability distributions, expected value, variance, Binomial and Normal distributions, and two-sided and one-sided intervals. A basic familiarity with these

- A *random variable* can be viewed as the name of an experiment with a probabilistic outcome. Its value is the outcome of the experiment.
- A *probability distribution* for a random variable $Y$ specifies the probability $\Pr(Y = y_i)$ that $Y$ will take on the value $y_i$, for each possible value $y_i$.
- The *expected value*, or *mean*, of a random variable $Y$ is $E[Y] = \sum_i y_i \Pr(Y = y_i)$. The symbol $\mu_Y$ is commonly used to represent E[Y].
- The *variance* of a random variable is $Var(Y) = E[(Y - \mu_Y)^2]$. The variance characterizes the width or dispersion of the distribution about its mean.
- The *standard deviation* of $Y$ is $\sqrt{Var(Y)}$. The symbol $\sigma_Y$ is often used used to represent the standard deviation of $Y$.
- The *Binomial distribution* gives the probability of observing $r$ heads in a series of $n$ independent coin tosses, if the probability of heads in a single toss is $p$.
- The *Normal distribution* is a bell-shaped probability distribution that covers many natural phenomena.
- The *Central Limit Theorem* is a theorem stating that the sum of a large number of independent, identically distributed random variables approximately follows a Normal distribution.
- An *estimator* is a random variable $Y$ used to estimate some parameter $p$ of an underlying population.
- The *estimation bias* of $Y$ as an estimator for $p$ is the quantity $(E[Y] - p)$. An unbiased estimator is one for which the bias is zero.
- A *N% confidence interval* estimate for parameter $p$ is an interval that includes $p$ with probability $N\%$.

**TABLE 5.2**
Basic definitions and facts from statistics.

concepts is important to understanding how to evaluate hypotheses and learning algorithms. Even more important, these same notions provide an important conceptual framework for understanding machine learning issues such as overfitting and the relationship between successful generalization and the number of training examples considered. The reader who is already familiar with these notions may skip or skim this section without loss of continuity. The key concepts introduced in this section are summarized in Table 5.2.

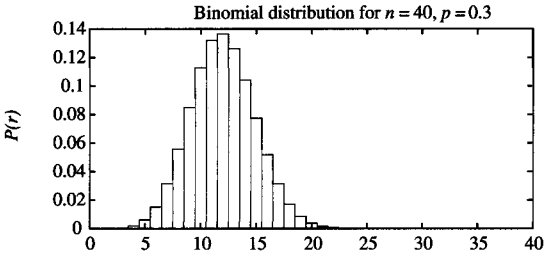
### 5.3.1 Error Estimation and Estimating Binomial Proportions

Precisely how does the deviation between sample error and true error depend on the size of the data sample? This question is an instance of a well-studied problem in statistics: the problem of estimating the proportion of a population that exhibits some property, given the observed proportion over some random sample of the population. In our case, the property of interest is that $h$ misclassifies the example.

The key to answering this question is to note that when we measure the sample error we are performing an experiment with a random outcome. We first collect a random sample $S$ of $n$ independently drawn instances from the distribution $\mathcal{D}$, and then measure the sample error $error_S(h)$. As noted in the previous

section, if we were to repeat this experiment many times, each time drawing a different random sample $S_i$ of size $n$, we would expect to observe different values for the various $error_{S_i}(h)$, depending on random differences in the makeup of the various $S_i$. We say in such cases that $error_{S_i}(h)$, the outcome of the $i$th such experiment, is a *random variable*. In general, one can think of a random variable as the name of an experiment with a random outcome. The value of the random variable is the observed outcome of the random experiment.

Imagine that we were to run $k$ such random experiments, measuring the random variables $error_{S_1}(h), error_{S_2}(h) \ldots error_{S_k}(h)$. Imagine further that we then plotted a histogram displaying the frequency with which we observed each possible error value. As we allowed $k$ to grow, the histogram would approach the form of the distribution shown in Table 5.3. This table describes a particular probability distribution called the *Binomial distribution*.




A *Binomial distribution* gives the probability of observing $r$ heads in a sample of $n$ independent coin tosses, when the probability of heads on a single coin toss is $p$. It is defined by the probability function

$$P(r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r}$$

If the random variable $X$ follows a Binomial distribution, then:

- The probability $\Pr(X = r)$ that $X$ will take on the value $r$ is given by $P(r)$
- The expected, or mean value of $X$, $E[X]$, is

$$E[X] = np$$

- The variance of $X$, $Var(X)$, is

$$Var(X) = np(1-p)$$

- The standard deviation of $X$, $\sigma_X$, is

$$\sigma_X = \sqrt{np(1-p)}$$

For sufficiently large values of $n$ the Binomial distribution is closely approximated by a Normal distribution (see Table 5.4) with the same mean and variance. Most statisticians recommend using the Normal approximation only when $np(1-p) \geq 5$.

**TABLE 5.3**
The Binomial distribution.

### 5.3.2 The Binomial Distribution

A good way to understand the Binomial distribution is to consider the following problem. You are given a worn and bent coin and asked to estimate the probability that the coin will turn up heads when tossed. Let us call this unknown probability of heads $p$. You toss the coin $n$ times and record the number of times $r$ that it turns up heads. A reasonable estimate of $p$ is $r/n$. Note that if the experiment were rerun, generating a new set of $n$ coin tosses, we might expect the number of heads $r$ to vary somewhat from the value measured in the first experiment, yielding a somewhat different estimate for $p$. The Binomial distribution describes for each possible value of $r$ (i.e., from 0 to $n$), the probability of observing exactly $r$ heads given a sample of $n$ independent tosses of a coin whose true probability of heads is $p$.

Interestingly, estimating $p$ from a random sample of coin tosses is equivalent to estimating $error_{\mathcal{D}}(h)$ from testing $h$ on a random sample of instances. A single toss of the coin corresponds to drawing a single random instance from $\mathcal{D}$ and determining whether it is misclassified by $h$. The probability $p$ that a single random coin toss will turn up heads corresponds to the probability that a single instance drawn at random will be misclassified (i.e., $p$ corresponds to $error_{\mathcal{D}}(h)$). The number $r$ of heads observed over a sample of $n$ coin tosses corresponds to the number of misclassifications observed over $n$ randomly drawn instances. Thus $r/n$ corresponds to $error_S(h)$. The problem of estimating $p$ for coins is identical to the problem of estimating $error_{\mathcal{D}}(h)$ for hypotheses. The Binomial distribution gives the general form of the probability distribution for the random variable $r$, whether it represents the number of heads in $n$ coin tosses or the number of hypothesis errors in a sample of $n$ examples. The detailed form of the Binomial distribution depends on the specific sample size $n$ and the specific probability $p$ or $error_{\mathcal{D}}(h)$.

The general setting to which the Binomial distribution applies is:

1. There is a base, or underlying, experiment (e.g., toss of the coin) whose outcome can be described by a random variable, say $Y$. The random variable $Y$ can take on two possible values (e.g., $Y = 1$ if heads, $Y = 0$ if tails).
2. The probability that $Y = 1$ on any single trial of the underlying experiment is given by some constant $p$, independent of the outcome of any other experiment. The probability that $Y = 0$ is therefore $(1 - p)$. Typically, $p$ is not known in advance, and the problem is to estimate it.
3. A series of $n$ independent trials of the underlying experiment is performed (e.g., $n$ independent coin tosses), producing the sequence of independent, identically distributed random variables $Y_1, Y_2, \ldots, Y_n$. Let $R$ denote the number of trials for which $Y_i = 1$ in this series of $n$ experiments

$$R \equiv \sum_{i=1}^{n} Y_i$$

4. The probability that the random variable $R$ will take on a specific value $r$ (e.g., the probability of observing exactly $r$ heads) is given by the Binomial distribution

$$\Pr(R = r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r} \tag{5.2}$$

A plot of this probability distribution is shown in Table 5.3.

The Binomial distribution characterizes the probability of observing $r$ heads from $n$ coin flip experiments, as well as the probability of observing $r$ errors in a data sample containing $n$ randomly drawn instances.

### 5.3.3 Mean and Variance

Two properties of a random variable that are often of interest are its expected value (also called its mean value) and its variance. The expected value is the average of the values taken on by repeatedly sampling the random variable. More precisely

***Definition:*** Consider a random variable $Y$ that takes on the possible values $y_1, \ldots y_n$. The **expected value** of $Y$, $E[Y]$, is

$$E[Y] \equiv \sum_{i=1}^{n} y_i \Pr(Y = y_i) \tag{5.3}$$

For example, if $Y$ takes on the value 1 with probability .7 and the value 2 with probability .3, then its expected value is $(1 \cdot 0.7 + 2 \cdot 0.3 = 1.3)$. In case the random variable $Y$ is governed by a Binomial distribution, then it can be shown that

$$E[Y] = np \tag{5.4}$$

where $n$ and $p$ are the parameters of the Binomial distribution defined in Equation (5.2).

A second property, the variance, captures the "width" or "spread" of the probability distribution; that is, it captures how far the random variable is expected to vary from its mean value.

***Definition:*** The **variance** of a random variable $Y$, $Var[Y]$, is

$$Var[Y] \equiv E[(Y - E[Y])^2] \tag{5.5}$$

The variance describes the expected squared error in using a single observation of $Y$ to estimate its mean $E[Y]$. The square root of the variance is called the *standard deviation* of $Y$, denoted $\sigma_Y$.

***Definition:*** The **standard deviation** of a random variable $Y$, $\sigma_Y$, is

$$\sigma_Y \equiv \sqrt{E[(Y - E[Y])^2]} \tag{5.6}$$

In case the random variable $Y$ is governed by a Binomial distribution, then the variance and standard deviation are given by

$$Var[Y] = np(1-p)$$
$$\sigma_Y = \sqrt{np(1-p)} \tag{5.7}$$

### 5.3.4 Estimators, Bias, and Variance

Now that we have shown that the random variable $error_S(h)$ obeys a Binomial distribution, we return to our primary question: What is the likely difference between $error_S(h)$ and the true error $error_{\mathcal{D}}(h)$?

Let us describe $error_S(h)$ and $error_{\mathcal{D}}(h)$ using the terms in Equation (5.2) defining the Binomial distribution. We then have

$$error_S(h) = \frac{r}{n}$$
$$error_{\mathcal{D}}(h) = p$$

where $n$ is the number of instances in the sample $S$, $r$ is the number of instances from $S$ misclassified by $h$, and $p$ is the probability of misclassifying a single instance drawn from $\mathcal{D}$.

Statisticians call $error_S(h)$ an *estimator* for the true error $error_{\mathcal{D}}(h)$. In general, an estimator is any random variable used to estimate some parameter of the underlying population from which the sample is drawn. An obvious question to ask about any estimator is whether on average it gives the right estimate. We define the *estimation bias* to be the difference between the expected value of the estimator and the true value of the parameter.

***Definition:*** The **estimation bias** of an estimator $Y$ for an arbitrary parameter $p$ is

$$E[Y] - p$$

If the estimation bias is zero, we say that $Y$ is an *unbiased estimator* for $p$. Notice this will be the case if the average of many random values of $Y$ generated by repeated random experiments (i.e., $E[Y]$) converges toward $p$.

Is $error_S(h)$ an unbiased estimator for $error_{\mathcal{D}}(h)$? Yes, because for a Binomial distribution the expected value of $r$ is equal to $np$ (Equation [5.4]). It follows, given that $n$ is a constant, that the expected value of $r/n$ is $p$.

Two quick remarks are in order regarding the estimation bias. First, when we mentioned at the beginning of this chapter that testing the hypothesis on the training examples provides an optimistically biased estimate of hypothesis error, it is exactly this notion of estimation bias to which we were referring. In order for $error_S(h)$ to give an unbiased estimate of $error_{\mathcal{D}}(h)$, the hypothesis $h$ and sample $S$ must be chosen independently. Second, this notion of *estimation bias* should not be confused with the *inductive bias* of a learner introduced in Chapter 2. The

estimation bias is a numerical quantity, whereas the inductive bias is a set of assertions.

A second important property of any estimator is its variance. Given a choice among alternative unbiased estimators, it makes sense to choose the one with least variance. By our definition of variance, this choice will yield the smallest expected squared error between the estimate and the true value of the parameter.

To illustrate these concepts, suppose we test a hypothesis and find that it commits $r = 12$ errors on a sample of $n = 40$ randomly drawn test examples. Then an unbiased estimate for $error_{\mathcal{D}}(h)$ is given by $error_S(h) = r/n = 0.3$. The variance in this estimate arises completely from the variance in $r$, because $n$ is a constant. Because $r$ is Binomially distributed, its variance is given by Equation (5.7) as $np(1-p)$. Unfortunately $p$ is unknown, but we can substitute our estimate $r/n$ for $p$. This yields an estimated variance in $r$ of $40 \cdot 0.3(1 - 0.3) = 8.4$, or a corresponding standard deviation of $\sqrt{8.4} \approx 2.9$. This implies that the standard deviation in $error_S(h) = r/n$ is approximately $2.9/40 = .07$. To summarize, $error_S(h)$ in this case is observed to be 0.30, with a standard deviation of approximately 0.07. (See Exercise 5.1.)

In general, given $r$ errors in a sample of $n$ independently drawn test examples, the standard deviation for $error_S(h)$ is given by

$$\sigma_{error_S(h)} = \frac{\sigma_r}{n} = \sqrt{\frac{p(1-p)}{n}} \tag{5.8}$$

which can be approximated by substituting $r/n = error_S(h)$ for $p$

$$\sigma_{error_S(h)} \approx \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}} \tag{5.9}$$

### 5.3.5 Confidence Intervals

One common way to describe the uncertainty associated with an estimate is to give an interval within which the true value is expected to fall, along with the probability with which it is expected to fall into this interval. Such estimates are called *confidence interval* estimates.

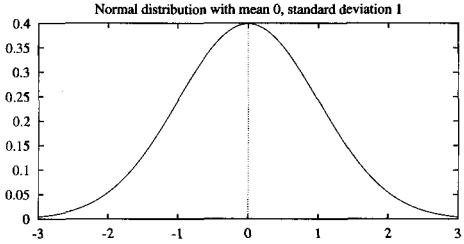
> ***Definition:*** An $N\%$ **confidence interval** for some parameter $p$ is an interval that is expected with probability $N\%$ to contain $p$.

For example, if we observe $r = 12$ errors in a sample of $n = 40$ independently drawn examples, we can say with approximately 95% probability that the interval $0.30 \pm 0.14$ contains the true error $error_{\mathcal{D}}(h)$.

How can we derive confidence intervals for $error_{\mathcal{D}}(h)$? The answer lies in the fact that we know the Binomial probability distribution governing the estimator $error_S(h)$. The mean value of this distribution is $error_{\mathcal{D}}(h)$, and the standard deviation is given by Equation (5.9). Therefore, to derive a 95% confidence interval, we need only find the interval centered around the mean value $error_{\mathcal{D}}(h)$,

which is wide enough to contain 95% of the total probability under this distribution. This provides an interval surrounding $error_{\mathcal{D}}(h)$ into which $error_S(h)$ must fall 95% of the time. Equivalently, it provides the size of the interval surrounding $error_S(h)$ into which $error_{\mathcal{D}}(h)$ must fall 95% of the time.

For a given value of $N$ how can we find the size of the interval that contains $N\%$ of the probability mass? Unfortunately, for the Binomial distribution this calculation can be quite tedious. Fortunately, however, an easily calculated and very good approximation can be found in most cases, based on the fact that for sufficiently large sample sizes the Binomial distribution can be closely approximated by the Normal distribution. The Normal distribution, summarized in Table 5.4, is perhaps the most well-studied probability distribution in statistics. As illustrated in Table 5.4, it is a bell-shaped distribution fully specified by its




A Normal distribution (also called a Gaussian distribution) is a bell-shaped distribution defined by the probability density function

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

A Normal distribution is fully determined by two parameters in the above formula: $\mu$ and $\sigma$.

If the random variable $X$ follows a normal distribution, then:

- The probability that $X$ will fall into the interval $(a, b)$ is given by

$$\int_a^b p(x)dx$$

- The expected, or mean value of $X$, $E[X]$, is

$$E[X] = \mu$$

- The variance of $X$, $Var(X)$, is

$$Var(X) = \sigma^2$$

- The standard deviation of $X$, $\sigma_X$, is

$$\sigma_X = \sigma$$

The Central Limit Theorem (Section 5.4.1) states that the sum of a large number of independent, identically distributed random variables follows a distribution that is approximately Normal.

**TABLE 5.4**
The Normal or Gaussian distribution.

mean $\mu$ and standard deviation $\sigma$. For large $n$, any Binomial distribution is very closely approximated by a Normal distribution with the same mean and variance.

One reason that we prefer to work with the Normal distribution is that most statistics references give tables specifying the size of the interval about the mean that contains $N\%$ of the probability mass under the Normal distribution. This is precisely the information needed to calculate our $N\%$ confidence interval. In fact, Table 5.1 is such a table. The constant $z_N$ given in Table 5.1 defines the width of the smallest interval about the mean that includes $N\%$ of the total probability mass under the bell-shaped Normal distribution. More precisely, $z_N$ gives half the width of the interval (i.e., the distance from the mean in either direction) measured in standard deviations. Figure 5.1($a$) illustrates such an interval for $z_{.80}$.

To summarize, if a random variable $Y$ obeys a Normal distribution with mean $\mu$ and standard deviation $\sigma$, then the measured random value $y$ of $Y$ will fall into the following interval $N\%$ of the time

$$\mu \pm z_N \sigma \tag{5.10}$$

Equivalently, the mean $\mu$ will fall into the following interval $N\%$ of the time

$$y \pm z_N \sigma \tag{5.11}$$

We can easily combine this fact with earlier facts to derive the general expression for $N\%$ confidence intervals for discrete-valued hypotheses given in Equation (5.1). First, we know that $error_S(h)$ follows a Binomial distribution with mean value $error_{\mathcal{D}}(h)$ and standard deviation as given in Equation (5.9). Second, we know that for sufficiently large sample size $n$, this Binomial distribution is well approximated by a Normal distribution. Third, Equation (5.11) tells us how to find the $N\%$ confidence interval for estimating the mean value of a Normal distribution. Therefore, substituting the mean and standard deviation of $error_S(h)$ into Equation (5.11) yields the expression from Equation (5.1) for $N\%$ confidence

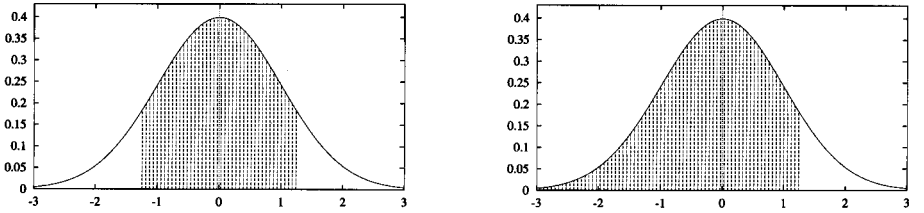



**FIGURE 5.1**
A Normal distribution with mean 0, standard deviation 1. ($a$) With 80% confidence, the value of the random variable will lie in the two-sided interval $[-1.28, 1.28]$. Note $z_{.80} = 1.28$. With 10% confidence it will lie to the right of this interval, and with 10% confidence it will lie to the left. ($b$) With 90% confidence, it will lie in the one-sided interval $[-\infty, 1.28]$.

intervals for discrete-valued hypotheses

$$error_S(h) \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

Recall that two approximations were involved in deriving this expression, namely:

1. in estimating the standard deviation $\sigma$ of $error_S(h)$, we have approximated $error_{\mathcal{D}}(h)$ by $error_S(h)$ [i.e., in going from Equation (5.8) to (5.9)], and
2. the Binomial distribution has been approximated by the Normal distribution.

The common rule of thumb in statistics is that these two approximations are very good as long as $n \geq 30$, or when $np(1-p) \geq 5$. For smaller values of $n$ it is wise to use a table giving exact values for the Binomial distribution.

### 5.3.6 Two-Sided and One-Sided Bounds

Notice that the above confidence interval is a *two-sided* bound; that is, it bounds the estimated quantity from above and from below. In some cases, we will be interested only in a *one-sided* bound. For example, we might be interested in the question "What is the probability that $error_{\mathcal{D}}(h)$ is at most $U$?" This kind of one-sided question is natural when we are only interested in bounding the maximum error of $h$ and do not mind if the true error is much smaller than estimated.

There is an easy modification to the above procedure for finding such one-sided error bounds. It follows from the fact that the Normal distribution is symmetric about its mean. Because of this fact, any two-sided confidence interval based on a Normal distribution can be converted to a corresponding one-sided interval with twice the confidence (see Figure 5.1(*b*)). That is, a $100(1-\alpha)\%$ confidence interval with lower bound $L$ and upper bound $U$ implies a $100(1-\alpha/2)\%$ confidence interval with lower bound $L$ and no upper bound. It also implies a $100(1-\alpha/2)\%$ confidence interval with upper bound $U$ and no lower bound. Here $\alpha$ corresponds to the probability that the correct value lies outside the stated interval. In other words, $\alpha$ is the probability that the value will fall into the *unshaded* region in Figure 5.1(*a*), and $\alpha/2$ is the probability that it will fall into the unshaded region in Figure 5.1(*b*).

To illustrate, consider again the example in which $h$ commits $r = 12$ errors over a sample of $n = 40$ independently drawn examples. As discussed above, this leads to a (two-sided) 95% confidence interval of $0.30 \pm 0.14$. In this case, $100(1-\alpha) = 95\%$, so $\alpha = 0.05$. Thus, we can apply the above rule to say with $100(1-\alpha/2) = 97.5\%$ confidence that $error_{\mathcal{D}}(h)$ is at most $0.30 + 0.14 = 0.44$, making no assertion about the lower bound on $error_{\mathcal{D}}(h)$. Thus, we have a one-sided error bound on $error_{\mathcal{D}}(h)$ with double the confidence that we had in the corresponding two-sided bound (see Exercise 5.3).

## 5.4 A GENERAL APPROACH FOR DERIVING CONFIDENCE INTERVALS

The previous section described in detail how to derive confidence interval estimates for one particular case: estimating $error_{\mathcal{D}}(h)$ for a discrete-valued hypothesis $h$, based on a sample of $n$ independently drawn instances. The approach described there illustrates a general approach followed in many estimation problems. In particular, we can see this as a problem of estimating the mean (expected value) of a population based on the mean of a randomly drawn sample of size $n$. The general process includes the following steps:

1. Identify the underlying population parameter $p$ to be estimated, for example, $error_{\mathcal{D}}(h)$.
2. Define the estimator $Y$ (e.g., $error_S(h)$). It is desirable to choose a minimum-variance, unbiased estimator.
3. Determine the probability distribution $\mathcal{D}_Y$ that governs the estimator $Y$, including its mean and variance.
4. Determine the $N\%$ confidence interval by finding thresholds $L$ and $U$ such that $N\%$ of the mass in the probability distribution $\mathcal{D}_Y$ falls between $L$ and $U$.

In later sections of this chapter we apply this general approach to several other estimation problems common in machine learning. First, however, let us discuss a fundamental result from estimation theory called the *Central Limit Theorem.*

### 5.4.1 Central Limit Theorem

One essential fact that simplifies attempts to derive confidence intervals is the Central Limit Theorem. Consider again our general setting, in which we observe the values of $n$ independently drawn random variables $Y_1 \ldots Y_n$ that obey the same unknown underlying probability distribution (e.g., $n$ tosses of the same coin). Let $\mu$ denote the mean of the unknown distribution governing each of the $Y_i$ and let $\sigma$ denote the standard deviation. We say that these variables $Y_i$ are *independent, identically distributed* random variables, because they describe independent experiments, each obeying the same underlying probability distribution. In an attempt to estimate the mean $\mu$ of the distribution governing the $Y_i$, we calculate the sample mean $\bar{Y}_n \equiv \frac{1}{n}\sum_{i=1}^{n} Y_i$ (e.g., the fraction of heads among the $n$ coin tosses). The Central Limit Theorem states that the probability distribution governing $\bar{Y}_n$ approaches a Normal distribution as $n \to \infty$, *regardless of the distribution that governs the underlying random variables* $Y_i$. Furthermore, the mean of the distribution governing $\bar{Y}_n$ approaches $\mu$ and the standard deviation approaches $\frac{\sigma}{\sqrt{n}}$. More precisely,

**Theorem 5.1. Central Limit Theorem.** Consider a set of independent, identically distributed random variables $Y_1 \ldots Y_n$ governed by an arbitrary probability distribution with mean $\mu$ and finite variance $\sigma^2$. Define the sample mean, $\bar{Y}_n \equiv \frac{1}{n}\sum_{i=1}^{n} Y_i$.

Then as $n \to \infty$, the distribution governing

$$\frac{\bar{Y}_n - \mu}{\frac{\sigma}{\sqrt{n}}}$$

approaches a Normal distribution, with zero mean and standard deviation equal to 1.

This is a quite surprising fact, because it states that we know the form of the distribution that governs the sample mean $\bar{Y}$ even when we do not know the form of the underlying distribution that governs the individual $Y_i$ that are being observed! Furthermore, the Central Limit Theorem describes how the mean and variance of $\bar{Y}$ can be used to determine the mean and variance of the individual $Y_i$.

The Central Limit Theorem is a very useful fact, because it implies that whenever we define an estimator that is the mean of some sample (e.g., $error_S(h)$ is the mean error), the distribution governing this estimator can be approximated by a Normal distribution for sufficiently large $n$. If we also know the variance for this (approximately) Normal distribution, then we can use Equation (5.11) to compute confidence intervals. A common rule of thumb is that we can use the Normal approximation when $n \geq 30$. Recall that in the preceding section we used such a Normal distribution to approximate the Binomial distribution that more precisely describes $error_S(h)$.

## 5.5 DIFFERENCE IN ERROR OF TWO HYPOTHESES

Consider the case where we have two hypotheses $h_1$ and $h_2$ for some discrete-valued target function. Hypothesis $h_1$ has been tested on a sample $S_1$ containing $n_1$ randomly drawn examples, and $h_2$ has been tested on an independent sample $S_2$ containing $n_2$ examples drawn from the same distribution. Suppose we wish to estimate the difference $d$ between the true errors of these two hypotheses.

$$d \equiv error_{\mathcal{D}}(h_1) - error_{\mathcal{D}}(h_2)$$

We will use the generic four-step procedure described at the beginning of Section 5.4 to derive a confidence interval estimate for $d$. Having identified $d$ as the parameter to be estimated, we next define an estimator. The obvious choice for an estimator in this case is the difference between the sample errors, which we denote by $\hat{d}$

$$\hat{d} \equiv error_{S_1}(h_1) - error_{S_2}(h_2)$$

Although we will not prove it here, it can be shown that $\hat{d}$ gives an unbiased estimate of $d$; that is $E[\hat{d}\,] = d$.

What is the probability distribution governing the random variable $\hat{d}$? From earlier sections, we know that for large $n_1$ and $n_2$ (e.g., both $\geq 30$), both $error_{S_1}(h_1)$ and $error_{S_2}(h_2)$ follow distributions that are approximately Normal. Because the difference of two Normal distributions is also a Normal distribution, $\hat{d}$ will also

follow a distribution that is approximately Normal, with mean $d$. It can also be shown that the variance of this distribution is the sum of the variances of $error_{S_1}(h_1)$ and $error_{S_2}(h_2)$. Using Equation (5.9) to obtain the approximate variance of each of these distributions, we have

$$\sigma_{\hat{d}}^2 \approx \frac{error_{S_1}(h_1)(1 - error_{S_1}(h_1))}{n_1} + \frac{error_{S_2}(h_2)(1 - error_{S_2}(h_2))}{n_2} \quad (5.12)$$

Now that we have determined the probability distribution that governs the estimator $\hat{d}$, it is straightforward to derive confidence intervals that characterize the likely error in employing $\hat{d}$ to estimate $d$. For a random variable $\hat{d}$ obeying a Normal distribution with mean $d$ and variance $\sigma^2$, the N% confidence interval estimate for $d$ is $\hat{d} \pm z_N\sigma$. Using the approximate variance $\sigma_{\hat{d}}^2$ given above, this approximate $N\%$ confidence interval estimate for $d$ is

$$\hat{d} \pm z_N \sqrt{\frac{error_{S_1}(h_1)(1 - error_{S_1}(h_1))}{n_1} + \frac{error_{S_2}(h_2)(1 - error_{S_2}(h_2))}{n_2}} \quad (5.13)$$

where $z_N$ is the same constant described in Table 5.1. The above expression gives the general two-sided confidence interval for estimating the difference between errors of two hypotheses. In some situations we might be interested in one-sided bounds—either bounding the largest possible difference in errors or the smallest, with some confidence level. One-sided confidence intervals can be obtained by modifying the above expression as described in Section 5.3.6.

Although the above analysis considers the case in which $h_1$ and $h_2$ are tested on independent data samples, it is often acceptable to use the confidence interval seen in Equation (5.13) in the setting where $h_1$ and $h_2$ are tested on a single sample $S$ (where $S$ is still independent of $h_1$ and $h_2$). In this later case, we redefine $\hat{d}$ as

$$\hat{d} \equiv error_S(h_1) - error_S(h_2)$$

The variance in this new $\hat{d}$ will usually be smaller than the variance given by Equation (5.12), when we set $S_1$ and $S_2$ to $S$. This is because using a single sample $S$ eliminates the variance due to random differences in the compositions of $S_1$ and $S_2$. In this case, the confidence interval given by Equation (5.13) will generally be an overly conservative, but still correct, interval.

### 5.5.1 Hypothesis Testing

In some cases we are interested in the probability that some specific conjecture is true, rather than in confidence intervals for some parameter. Suppose, for example, that we are interested in the question "what is the probability that $error_{\mathcal{D}}(h_1) > error_{\mathcal{D}}(h_2)$?" Following the setting in the previous section, suppose we measure the sample errors for $h_1$ and $h_2$ using two independent samples $S_1$ and $S_2$ of size 100 and find that $error_{S_1}(h_1) = .30$ and $error_{S_2}(h_2) = .20$, hence the observed difference is $\hat{d} = .10$. Of course, due to random variation in the data sample,

we might observe this difference in the sample errors even when $error_{\mathcal{D}}(h_1) \leq error_{\mathcal{D}}(h_2)$. What is the probability that $error_{\mathcal{D}}(h_1) > error_{\mathcal{D}}(h_2)$, given the observed difference in sample errors $\hat{d} = .10$ in this case? Equivalently, what is the probability that $d > 0$, given that we observed $\hat{d} = .10$?

Note the probability $\Pr(d > 0)$ is equal to the probability that $\hat{d}$ has not overestimated $d$ by more than .10. Put another way, this is the probability that $\hat{d}$ falls into the one-sided interval $\hat{d} < d + .10$. Since $d$ is the mean of the distribution governing $\hat{d}$, we can equivalently express this one-sided interval as $\hat{d} < \mu_{\hat{d}} + .10$.

To summarize, the probability $\Pr(d > 0)$ equals the probability that $\hat{d}$ falls into the one-sided interval $\hat{d} < \mu_{\hat{d}} + .10$. Since we already calculated the approximate distribution governing $\hat{d}$ in the previous section, we can determine the probability that $\hat{d}$ falls into this one-sided interval by calculating the probability mass of the $\hat{d}$ distribution within this interval.

Let us begin this calculation by re-expressing the interval $\hat{d} < \mu_{\hat{d}} + .10$ in terms of the number of standard deviations it allows deviating from the mean. Using Equation (5.12) we find that $\sigma_{\hat{d}} \approx .061$, so we can re-express the interval as approximately

$$\hat{d} < \mu_{\hat{d}} + 1.64\sigma_{\hat{d}}$$

What is the confidence level associated with this one-sided interval for a Normal distribution? Consulting Table 5.1, we find that 1.64 standard deviations about the mean corresponds to a two-sided interval with confidence level 90%. Therefore, the one-sided interval will have an associated confidence level of 95%.

Therefore, given the observed $\hat{d} = .10$, the probability that $error_{\mathcal{D}}(h_1) > error_{\mathcal{D}}(h_2)$ is approximately .95. In the terminology of the statistics literature, we say that we accept the hypothesis that "$error_{\mathcal{D}}(h_1) > error_{\mathcal{D}}(h_2)$" with confidence 0.95. Alternatively, we may state that we reject the opposite hypothesis (often called the null hypothesis) at a $(1 - 0.95) = .05$ level of significance.

## 5.6 COMPARING LEARNING ALGORITHMS

Often we are interested in comparing the performance of two learning algorithms $L_A$ and $L_B$, rather than two specific hypotheses. What is an appropriate test for comparing learning algorithms, and how can we determine whether an observed difference between the algorithms is statistically significant? Although there is active debate within the machine-learning research community regarding the best method for comparison, we present here one reasonable approach. A discussion of alternative methods is given by Dietterich (1996).

As usual, we begin by specifying the parameter we wish to estimate. Suppose we wish to determine which of $L_A$ and $L_B$ is the better learning method on average for learning some particular target function $f$. A reasonable way to define "on average" is to consider the relative performance of these two algorithms averaged over all the training sets of size $n$ that might be drawn from the underlying instance distribution $\mathcal{D}$. In other words, we wish to estimate the expected value

of the difference in their errors

$$\underset{S \subset \mathcal{D}}{E}[error_{\mathcal{D}}(L_A(S)) - error_{\mathcal{D}}(L_B(S))] \tag{5.14}$$

where $L(S)$ denotes the hypothesis output by learning method $L$ when given the sample $S$ of training data and where the subscript $S \subset \mathcal{D}$ indicates that the expected value is taken over samples $S$ drawn according to the underlying instance distribution $\mathcal{D}$. The above expression describes the expected value of the difference in errors between learning methods $L_A$ and $L_B$.

Of course in practice we have only a limited sample $D_0$ of data when comparing learning methods. In such cases, one obvious approach to estimating the above quantity is to divide $D_0$ into a training set $S_0$ and a disjoint test set $T_0$. The training data can be used to train both $L_A$ and $L_B$, and the test data can be used to compare the accuracy of the two learned hypotheses. In other words, we measure the quantity

$$error_{T_0}(L_A(S_0)) - error_{T_0}(L_B(S_0)) \tag{5.15}$$

Notice two key differences between this estimator and the quantity in Equation (5.14). First, we are using $error_{T_0}(h)$ to approximate $error_{\mathcal{D}}(h)$. Second, we are only measuring the difference in errors for one training set $S_0$ rather than taking the expected value of this difference over all samples $S$ that might be drawn from the distribution $\mathcal{D}$.

One way to improve on the estimator given by Equation (5.15) is to repeatedly partition the data $D_0$ into disjoint training and test sets and to take the mean of the test set errors for these different experiments. This leads to the procedure shown in Table 5.5 for estimating the difference between errors of two learning methods, based on a fixed sample $D_0$ of available data. This procedure first partitions the data into $k$ disjoint subsets of equal size, where this size is at least 30. It then trains and tests the learning algorithms $k$ times, using each of the $k$ subsets in turn as the test set, and using all remaining data as the training set. In this way, the learning algorithms are tested on $k$ independent test sets, and the mean difference in errors $\bar{\delta}$ is returned as an estimate of the difference between the two learning algorithms.

The quantity $\bar{\delta}$ returned by the procedure of Table 5.5 can be taken as an estimate of the desired quantity from Equation 5.14. More appropriately, we can view $\bar{\delta}$ as an estimate of the quantity

$$\underset{S \subset D_0}{E}[error_{\mathcal{D}}(L_A(S)) - error_{\mathcal{D}}(L_B(S))] \tag{5.16}$$

where $S$ represents a random sample of size $\frac{k-1}{k}|D_0|$ drawn uniformly from $D_0$. The only difference between this expression and our original expression in Equation (5.14) is that this new expression takes the expected value over subsets of the available data $D_0$, rather than over subsets drawn from the full instance distribution $\mathcal{D}$.

1. Partition the available data $D_0$ into $k$ disjoint subsets $T_1, T_2, \ldots, T_k$ of equal size, where this size is at least 30.
2. For $i$ from 1 to $k$, do
   *use $T_i$ for the test set, and the remaining data for training set $S_i$*
   - $S_i \leftarrow \{D_0 - T_i\}$
   - $h_A \leftarrow L_A(S_i)$
   - $h_B \leftarrow L_B(S_i)$
   - $\delta_i \leftarrow error_{T_i}(h_A) - error_{T_i}(h_B)$
3. Return the value $\bar{\delta}$, where

$$\bar{\delta} \equiv \frac{1}{k} \sum_{i=1}^{k} \delta_i \tag{T5.1}$$

**TABLE 5.5**
A procedure to estimate the difference in error between two learning methods $L_A$ and $L_B$. Approximate confidence intervals for this estimate are given in the text.

The approximate $N\%$ confidence interval for estimating the quantity in Equation (5.16) using $\bar{\delta}$ is given by

$$\bar{\delta} \pm t_{N,k-1}\, s_{\bar{\delta}} \tag{5.17}$$

where $t_{N,k-1}$ is a constant that plays a role analogous to that of $z_N$ in our earlier confidence interval expressions, and where $s_{\bar{\delta}}$ is an estimate of the standard deviation of the distribution governing $\bar{\delta}$. In particular, $s_{\bar{\delta}}$ is defined as

$$s_{\bar{\delta}} \equiv \sqrt{\frac{1}{k(k-1)} \sum_{i=1}^{k} (\delta_i - \bar{\delta})^2} \tag{5.18}$$

Notice the constant $t_{N,k-1}$ in Equation (5.17) has two subscripts. The first specifies the desired confidence level, as it did for our earlier constant $z_N$. The second parameter, called the number of *degrees of freedom* and usually denoted by $\nu$, is related to the number of independent random events that go into producing the value for the random variable $\bar{\delta}$. In the current setting, the number of degrees of freedom is $k - 1$. Selected values for the parameter $t$ are given in Table 5.6. Notice that as $k \rightarrow \infty$, the value of $t_{N,k-1}$ approaches the constant $z_N$.

Note the procedure described here for comparing two learning methods involves testing the two learned hypotheses on identical test sets. This contrasts with the method described in Section 5.5 for comparing hypotheses that have been evaluated using two independent test sets. Tests where the hypotheses are evaluated over identical samples are called *paired tests*. Paired tests typically produce tighter confidence intervals because any differences in observed errors in a paired test are due to differences between the hypotheses. In contrast, when the hypotheses are tested on separate data samples, differences in the two sample errors might be partially attributable to differences in the makeup of the two samples.

| | Confidence level $N$ | | | |
|---|---|---|---|---|
| | 90% | 95% | 98% | 99% |
| $\nu = 2$ | 2.92 | 4.30 | 6.96 | 9.92 |
| $\nu = 5$ | 2.02 | 2.57 | 3.36 | 4.03 |
| $\nu = 10$ | 1.81 | 2.23 | 2.76 | 3.17 |
| $\nu = 20$ | 1.72 | 2.09 | 2.53 | 2.84 |
| $\nu = 30$ | 1.70 | 2.04 | 2.46 | 2.75 |
| $\nu = 120$ | 1.66 | 1.98 | 2.36 | 2.62 |
| $\nu = \infty$ | 1.64 | 1.96 | 2.33 | 2.58 |

**TABLE 5.6**
Values of $t_{N,\nu}$ for two-sided confidence intervals. As $\nu \to \infty$, $t_{N,\nu}$ approaches $z_N$.

### 5.6.1 Paired $t$ Tests

Above we described one procedure for comparing two learning methods given a fixed set of data. This section discusses the statistical justification for this procedure, and for the confidence interval defined by Equations (5.17) and (5.18). It can be skipped or skimmed on a first reading without loss of continuity.

The best way to understand the justification for the confidence interval estimate given by Equation (5.17) is to consider the following estimation problem:

- We are given the observed values of a set of independent, identically distributed random variables $Y_1, Y_2, \ldots, Y_k$.
- We wish to estimate the mean $\mu$ of the probability distribution governing these $Y_i$.
- The estimator we will use is the sample mean $\bar{Y}$

$$\bar{Y} \equiv \frac{1}{k} \sum_{i=1}^{k} Y_i$$

This problem of estimating the distribution mean $\mu$ based on the sample mean $\bar{Y}$ is quite general. For example, it covers the problem discussed earlier of using $error_S(h)$ to estimate $error_{\mathcal{D}}(h)$. (In that problem, the $Y_i$ are 1 or 0 to indicate whether $h$ commits an error on an individual example from $S$, and $error_{\mathcal{D}}(h)$ is the mean $\mu$ of the underlying distribution.) The $t$ test, described by Equations (5.17) and (5.18), applies to a special case of this problem—the case in which the individual $Y_i$ follow a Normal distribution.

Now consider the following idealization of the method in Table 5.5 for comparing learning methods. Assume that instead of having a fixed sample of data $D_0$, we can request new training examples drawn according to the underlying instance distribution. In particular, in this idealized method we modify the procedure of Table 5.5 so that on each iteration through the loop it generates a new random training set $S_i$ and new random test set $T_i$ by drawing from this underlying instance distribution instead of drawing from the fixed sample $D_0$. This idealized method

perfectly fits the form of the above estimation problem. In particular, the $\delta_i$ measured by the procedure now correspond to the independent, identically distributed random variables $Y_i$. The mean $\mu$ of their distribution corresponds to the expected difference in error between the two learning methods [i.e., Equation (5.14)]. The sample mean $\bar{Y}$ is the quantity $\bar{\delta}$ computed by this idealized version of the method. We wish to answer the question "how good an estimate of $\mu$ is provided by $\bar{\delta}$?"

First, note that the size of the test sets $T_i$ has been chosen to contain at least 30 examples. Because of this, the individual $\delta_i$ will each follow an approximately Normal distribution (due to the Central Limit Theorem). Hence, we have a special case in which the $Y_i$ are governed by an approximately Normal distribution. It can be shown in general that when the individual $Y_i$ each follow a Normal distribution, then the sample mean $\bar{Y}$ follows a Normal distribution as well. Given that $\bar{Y}$ is Normally distributed, we might consider using the earlier expression for confidence intervals (Equation [5.11]) that applies to estimators governed by Normal distributions. Unfortunately, that equation requires that we know the standard deviation of this distribution, which we do not.

The $t$ test applies to precisely these situations, in which the task is to estimate the sample mean of a collection of independent, identically and Normally distributed random variables. In this case, we can use the confidence interval given by Equations (5.17) and (5.18), which can be restated using our current notation as

$$\mu = \bar{Y} \pm t_{N,k-1}\, s_{\bar{Y}}$$

where $s_{\bar{Y}}$ is the estimated standard deviation of the sample mean

$$s_{\bar{Y}} \equiv \sqrt{\frac{1}{k(k-1)} \sum_{i=1}^{k} (Y_i - \bar{Y})^2}$$

and where $t_{N,k-1}$ is a constant analogous to our earlier $z_N$. In fact, the constant $t_{N,k-1}$ characterizes the area under a probability distribution known as the $t$ distribution, just as the constant $z_N$ characterizes the area under a Normal distribution. The $t$ distribution is a bell-shaped distribution similar to the Normal distribution, but wider and shorter to reflect the greater variance introduced by using $s_{\bar{Y}}$ to approximate the true standard deviation $\sigma_{\bar{Y}}$. The $t$ distribution approaches the Normal distribution (and therefore $t_{N,k-1}$ approaches $z_N$) as $k$ approaches infinity. This is intuitively satisfying because we expect $s_{\bar{Y}}$ to converge toward the true standard deviation $\sigma_{\bar{Y}}$ as the sample size $k$ grows, and because we can use $z_N$ when the standard deviation is known exactly.

### 5.6.2 Practical Considerations

Note the above discussion justifies the use of the confidence interval estimate given by Equation (5.17) in the case where we wish to use the sample mean $\bar{Y}$ to estimate the mean of a sample containing $k$ independent, identically and Normally distributed random variables. This fits the idealized method described

above, in which we assume unlimited access to examples of the target function. In practice, given a limited set of data $D_0$ and the more practical method described by Table 5.5, this justification does not strictly apply. In practice, the problem is that the only way to generate new $\delta_i$ is to resample $D_0$, dividing it into training and test sets in different ways. The $\delta_i$ are not independent of one another in this case, because they are based on overlapping sets of training examples drawn from the limited subset $D_0$ of data, rather than from the full distribution $\mathcal{D}$.

When only a limited sample of data $D_0$ is available, several methods can be used to resample $D_0$. Table 5.5 describes a $k$-fold method in which $D_0$ is partitioned into $k$ disjoint, equal-sized subsets. In this $k$-fold approach, each example from $D_0$ is used exactly once in a test set, and $k - 1$ times in a training set. A second popular approach is to randomly choose a test set of at least 30 examples from $D_0$, use the remaining examples for training, then repeat this process as many times as desired. This randomized method has the advantage that it can be repeated an indefinite number of times, to shrink the confidence interval to the desired width. In contrast, the $k$-fold method is limited by the total number of examples, by the use of each example only once in a test set, and by our desire to use samples of size at least 30. However, the randomized method has the disadvantage that the test sets no longer qualify as being independently drawn with respect to the underlying instance distribution $\mathcal{D}$. In contrast, the test sets generated by $k$-fold cross validation are independent because each instance is included in only one test set.

To summarize, no single procedure for comparing learning methods based on limited data satisfies all the constraints we would like. It is wise to keep in mind that statistical models rarely fit perfectly the practical constraints in testing learning algorithms when available data is limited. Nevertheless, they do provide approximate confidence intervals that can be of great help in interpreting experimental comparisons of learning methods.

## 5.7 SUMMARY AND FURTHER READING

The main points of this chapter include:

- Statistical theory provides a basis for estimating the true error ($error_{\mathcal{D}}(h)$) of a hypothesis $h$, based on its observed error ($error_S(h)$) over a sample $S$ of data. For example, if $h$ is a discrete-valued hypothesis and the data sample $S$ contains $n \geq 30$ examples drawn independently of $h$ and of one another, then the $N\%$ confidence interval for $error_{\mathcal{D}}(h)$ is approximately

$$error_S(h) \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

  where values for $z_N$ are given in Table 5.1.
- In general, the problem of estimating confidence intervals is approached by identifying the parameter to be estimated (e.g., $error_{\mathcal{D}}(h)$) and an estimator

(e.g., $error_S(h)$) for this quantity. Because the estimator is a random variable (e.g., $error_S(h)$ depends on the random sample $S$), it can be characterized by the probability distribution that governs its value. Confidence intervals can then be calculated by determining the interval that contains the desired probability mass under this distribution.

- One possible cause of errors in estimating hypothesis accuracy is *estimation bias*. If $Y$ is an estimator for some parameter $p$, the estimation bias of $Y$ is the difference between $p$ and the expected value of $Y$. For example, if $S$ is the training data used to formulate hypothesis $h$, then $error_S(h)$ gives an optimistically biased estimate of the true error $error_{\mathcal{D}}(h)$.
- A second cause of estimation error is *variance* in the estimate. Even with an unbiased estimator, the observed value of the estimator is likely to vary from one experiment to another. The variance $\sigma^2$ of the distribution governing the estimator characterizes how widely this estimate is likely to vary from the correct value. This variance decreases as the size of the data sample is increased.
- Comparing the relative effectiveness of two learning algorithms is an estimation problem that is relatively easy when data and time are unlimited, but more difficult when these resources are limited. One possible approach described in this chapter is to run the learning algorithms on different subsets of the available data, testing the learned hypotheses on the remaining data, then averaging the results of these experiments.
- In most cases considered here, deriving confidence intervals involves making a number of assumptions and approximations. For example, the above confidence interval for $error_{\mathcal{D}}(h)$ involved approximating a Binomial distribution by a Normal distribution, approximating the variance of this distribution, and assuming instances are generated by a fixed, unchanging probability distribution. While intervals based on such approximations are only approximate confidence intervals, they nevertheless provide useful guidance for designing and interpreting experimental results in machine learning.

The key statistical definitions presented in this chapter are summarized in Table 5.2.

An ocean of literature exists on the topic of statistical methods for estimating means and testing significance of hypotheses. While this chapter introduces the basic concepts, more detailed treatments of these issues can be found in many books and articles. Billingsley et al. (1986) provide a very readable introduction to statistics that elaborates on the issues discussed here. Other texts on statistics include DeGroot (1986); Casella and Berger (1990). Duda and Hart (1973) provide a treatment of these issues in the context of numerical pattern recognition.

Segre et al. (1991, 1996), Etzioni and Etzioni (1994), and Gordon and Segre (1996) discuss statistical significance tests for evaluating learning algorithms whose performance is measured by their ability to improve computational efficiency.

Geman et al. (1992) discuss the tradeoff involved in attempting to minimize bias and variance simultaneously. There is ongoing debate regarding the best way to learn and compare hypotheses from limited data. For example, Dietterich (1996) discusses the risks of applying the paired-difference $t$ test repeatedly to different train-test splits of the data.

## EXERCISES

**5.1.** Suppose you test a hypothesis $h$ and find that it commits $r = 300$ errors on a sample $S$ of $n = 1000$ randomly drawn test examples. What is the standard deviation in $error_S(h)$? How does this compare to the standard deviation in the example at the end of Section 5.3.4?

**5.2.** Consider a learned hypothesis, $h$, for some boolean concept. When $h$ is tested on a set of 100 examples, it classifies 83 correctly. What is the standard deviation and the 95% confidence interval for the true error rate for $Error_{\mathcal{D}}(h)$?

**5.3.** Suppose hypothesis $h$ commits $r = 10$ errors over a sample of $n = 65$ independently drawn examples. What is the 90% confidence interval (two-sided) for the true error rate? What is the 95% one-sided interval (i.e., what is the upper bound $U$ such that $error_{\mathcal{D}}(h) \leq U$ with 95% confidence)? What is the 90% one-sided interval?

**5.4.** You are about to test a hypothesis $h$ whose $error_{\mathcal{D}}(h)$ is known to be in the range between 0.2 and 0.6. What is the minimum number of examples you must collect to assure that the width of the two-sided 95% confidence interval will be smaller than 0.1?

**5.5.** Give general expressions for the upper and lower one-sided $N\%$ confidence intervals for the difference in errors between two hypotheses tested on different samples of data. Hint: Modify the expression given in Section 5.5.

**5.6.** Explain why the confidence interval estimate given in Equation (5.17) applies to estimating the quantity in Equation (5.16), and not the quantity in Equation (5.14).

## REFERENCES

Billingsley, P., Croft, D. J., Huntsberger, D. V., & Watson, C. J. (1986). *Statistical inference for management and economics*. Boston: Allyn and Bacon, Inc.

Casella, G., & Berger, R. L. (1990). *Statistical inference*. Pacific Grove, CA: Wadsworth and Brooks/Cole.

DeGroot, M. H. (1986). *Probability and statistics*. (2d ed.) Reading, MA: Addison Wesley.

Dietterich, T. G. (1996). *Proper statistical tests for comparing supervised classification learning algorithms* (Technical Report). Department of Computer Science, Oregon State University, Corvallis, OR.

Dietterich, T. G., & Kong, E. B. (1995). *Machine learning bias, statistical bias, and statistical variance of decision tree algorithms* (Technical Report). Department of Computer Science, Oregon State University, Corvallis, OR.

Duda, R., & Hart, P. (1973). *Pattern classification and scene analysis*. New York: John Wiley & Sons.

Efron, B., & Tibshirani, R. (1991). Statistical data analysis in the computer age. *Science*, 253, 390–395.

Etzioni, O., & Etzioni, R. (1994). Statistical methods for analyzing speedup learning experiments. *Machine Learning*, 14, 333–347.

Geman, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, 4, 1–58.

Gordon, G., & Segre, A.M. (1996). Nonparametric statistical methods for experimental evaluations of speedup learning. *Proceedings of the Thirteenth International Conference on Machine Learning*, Bari, Italy.

Maisel, L. (1971). *Probability, statistics, and random processes*. Simon and Schuster Tech Outlines. New York: Simon and Schuster.

Segre, A., Elkan, C., & Russell, A. (1991). A critical look at experimental evaluations of EBL. *Machine Learning*, 6(2).

Segre, A.M, Gordon G., & Elkan, C. P. (1996). Exploratory analysis of speedup learning data using expectation maximization. *Artificial Intelligence*, 85, 301–319.

Speigel, M. R. (1991). *Theory and problems of probability and statistics*. Schaum's Outline Series. New York: McGraw Hill.

Thompson, M.L., & Zucchini, W. (1989). On the statistical analysis of ROC curves. *Statistics in Medicine*, 8, 1277–1290.

White, A. P., & Liu, W. Z. (1994). Bias in information-based measures in decision tree induction. *Machine Learning*, 15, 321–329.

# CHAPTER 8

# INSTANCE-BASED LEARNING

In contrast to learning methods that construct a general, explicit description of the target function when training examples are provided, instance-based learning methods simply store the training examples. Generalizing beyond these examples is postponed until a new instance must be classified. Each time a new query instance is encountered, its relationship to the previously stored examples is examined in order to assign a target function value for the new instance. Instance-based learning includes nearest neighbor and locally weighted regression methods that assume instances can be represented as points in a Euclidean space. It also includes case-based reasoning methods that use more complex, symbolic representations for instances. Instance-based methods are sometimes referred to as "lazy" learning methods because they delay processing until a new instance must be classified. A key advantage of this kind of delayed, or lazy, learning is that instead of estimating the target function once for the entire instance space, these methods can estimate it locally and differently for each new instance to be classified.

## 8.1 INTRODUCTION

Instance-based learning methods such as nearest neighbor and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions. Learning in these algorithms consists of simply storing the presented training data. When a new query instance is encountered, a set of similar related instances is retrieved from memory and used to classify the

new query instance. One key difference between these approaches and the methods discussed in other chapters is that instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified. In fact, many techniques construct only a local approximation to the target function that applies in the neighborhood of the new query instance, and never construct an approximation designed to perform well over the entire instance space. This has significant advantages when the target function is very complex, but can still be described by a collection of less complex local approximations.

Instance-based methods can also use more complex, symbolic representations for instances. In case-based learning, instances are represented in this fashion and the process for identifying "neighboring" instances is elaborated accordingly. Case-based reasoning has been applied to tasks such as storing and reusing past experience at a help desk, reasoning about legal cases by referring to previous cases, and solving complex scheduling problems by reusing relevant portions of previously solved problems.

One disadvantage of instance-based approaches is that the cost of classifying new instances can be high. This is due to the fact that nearly all computation takes place at classification time rather than when the training examples are first encountered. Therefore, techniques for efficiently indexing training examples are a significant practical issue in reducing the computation required at query time. A second disadvantage to many instance-based approaches, especially nearest-neighbor approaches, is that they typically consider *all* attributes of the instances when attempting to retrieve similar training examples from memory. If the target concept depends on only a few of the many available attributes, then the instances that are truly most "similar" may well be a large distance apart.

In the next section we introduce the *k*-NEAREST NEIGHBOR learning algorithm, including several variants of this widely-used approach. The subsequent section discusses locally weighted regression, a learning method that constructs local approximations to the target function and that can be viewed as a generalization of *k*-NEAREST NEIGHBOR algorithms. We then describe radial basis function networks, which provide an interesting bridge between instance-based and neural network learning algorithms. The next section discusses case-based reasoning, an instance-based approach that employs symbolic representations and knowledge-based inference. This section includes an example application of case-based reasoning to a problem in engineering design. Finally, we discuss the fundamental differences in capabilities that distinguish lazy learning methods discussed in this chapter from eager learning methods discussed in the other chapters of this book.

## 8.2 *k*-NEAREST NEIGHBOR LEARNING

The most basic instance-based method is the *k*-NEAREST NEIGHBOR algorithm. This algorithm assumes all instances correspond to points in the $n$-dimensional space $\Re^n$. The nearest neighbors of an instance are defined in terms of the standard

Euclidean distance. More precisely, let an arbitrary instance $x$ be described by the feature vector

$$\langle a_1(x), a_2(x), \ldots a_n(x) \rangle$$

where $a_r(x)$ denotes the value of the $r$th attribute of instance $x$. Then the distance between two instances $x_i$ and $x_j$ is defined to be $d(x_i, x_j)$, where

$$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

In nearest-neighbor learning the target function may be either discrete-valued or real-valued. Let us first consider learning discrete-valued target functions of the form $f : \Re^n \rightarrow V$, where $V$ is the finite set $\{v_1, \ldots v_s\}$. The $k$-NEAREST NEIGHBOR algorithm for approximating a discrete-valued target function is given in Table 8.1. As shown there, the value $\hat{f}(x_q)$ returned by this algorithm as its estimate of $f(x_q)$ is just the most common value of $f$ among the $k$ training examples nearest to $x_q$. If we choose $k = 1$, then the 1-NEAREST NEIGHBOR algorithm assigns to $\hat{f}(x_q)$ the value $f(x_i)$ where $x_i$ is the training instance nearest to $x_q$. For larger values of $k$, the algorithm assigns the most common value among the $k$ nearest training examples.

Figure 8.1 illustrates the operation of the $k$-NEAREST NEIGHBOR algorithm for the case where the instances are points in a two-dimensional space and where the target function is boolean valued. The positive and negative training examples are shown by "+" and "−" respectively. A query point $x_q$ is shown as well. Note the 1-NEAREST NEIGHBOR algorithm classifies $x_q$ as a positive example in this figure, whereas the 5-NEAREST NEIGHBOR algorithm classifies it as a negative example.

What is the nature of the hypothesis space $H$ implicitly considered by the $k$-NEAREST NEIGHBOR algorithm? Note the $k$-NEAREST NEIGHBOR algorithm never forms an explicit general hypothesis $\hat{f}$ regarding the target function $f$. It simply computes the classification of each new query instance as needed. Nevertheless,

---

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance $x_q$ to be classified,
  - Let $x_1 \ldots x_k$ denote the $k$ instances from *training_examples* that are nearest to $x_q$
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

---

**TABLE 8.1**
The $k$-NEAREST NEIGHBOR algorithm for approximating a discrete-valued function $f : \Re^n \rightarrow V$.

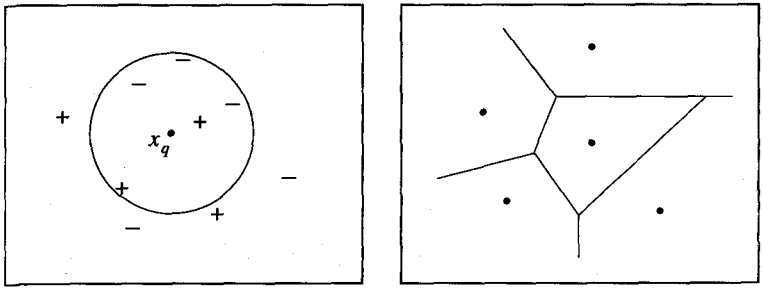



**FIGURE 8.1**
*k*-NEAREST NEIGHBOR. A set of positive and negative training examples is shown on the left, along with a query instance $x_q$ to be classified. The 1-NEAREST NEIGHBOR algorithm classifies $x_q$ positive, whereas 5-NEAREST NEIGHBOR classifies it as negative. On the right is the decision surface induced by the 1-NEAREST NEIGHBOR algorithm for a typical set of training examples. The convex polygon surrounding each training example indicates the region of instance space closest to that point (i.e., the instances for which the 1-NEAREST NEIGHBOR algorithm will assign the classification belonging to that training example).

we can still ask what the implicit general function is, or what classifications would be assigned if we were to hold the training examples constant and query the algorithm with every possible instance in $X$. The diagram on the right side of Figure 8.1 shows the shape of this decision surface induced by 1-NEAREST NEIGHBOR over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples. For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the *Voronoi diagram* of the set of training examples.

The *k*-NEAREST NEIGHBOR algorithm is easily adapted to approximating continuous-valued target functions. To accomplish this, we have the algorithm calculate the mean value of the $k$ nearest training examples rather than calculate their most common value. More precisely, to approximate a real-valued target function $f : \Re^n \rightarrow \Re$ we replace the final line of the above algorithm by the line

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k} \tag{8.1}$$

### 8.2.1 Distance-Weighted NEAREST NEIGHBOR Algorithm

One obvious refinement to the *k*-NEAREST NEIGHBOR algorithm is to weight the contribution of each of the $k$ neighbors according to their distance to the query point $x_q$, giving greater weight to closer neighbors. For example, in the algorithm of Table 8.1, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from $x_q$.

This can be accomplished by replacing the final line of the algorithm by

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i)) \tag{8.2}$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2} \tag{8.3}$$

To accommodate the case where the query point $x_q$ exactly matches one of the training instances $x_i$ and the denominator $d(x_q, x_i)^2$ is therefore zero, we assign $\hat{f}(x_q)$ to be $f(x_i)$ in this case. If there are several such training examples, we assign the majority classification among them.

We can distance-weight the instances for real-valued target functions in a similar fashion, replacing the final line of the algorithm in this case by

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \tag{8.4}$$

where $w_i$ is as defined in Equation (8.3). Note the denominator in Equation (8.4) is a constant that normalizes the contributions of the various weights (e.g., it assures that if $f(x_i) = c$ for all training examples, then $\hat{f}(x_q) \leftarrow c$ as well).

Note all of the above variants of the $k$-NEAREST NEIGHBOR algorithm consider only the $k$ nearest neighbors to classify the query point. Once we add distance weighting, there is really no harm in allowing all training examples to have an influence on the classification of the $x_q$, because very distant examples will have very little effect on $\hat{f}(x_q)$. The only disadvantage of considering all examples is that our classifier will run more slowly. If all training examples are considered when classifying a new query instance, we call the algorithm a *global* method. If only the nearest training examples are considered, we call it a *local* method. When the rule in Equation (8.4) is applied as a global method, using all training examples, it is known as Shepard's method (Shepard 1968).

### 8.2.2 Remarks on $k$-NEAREST NEIGHBOR Algorithm

The distance-weighted $k$-NEAREST NEIGHBOR algorithm is a highly effective inductive inference method for many practical problems. It is robust to noisy training data and quite effective when it is provided a sufficiently large set of training data. Note that by taking the weighted average of the $k$ neighbors nearest to the query point, it can smooth out the impact of isolated noisy training examples.

What is the inductive bias of $k$-NEAREST NEIGHBOR? The basis for classifying new query points is easily understood based on the diagrams in Figure 8.1. The inductive bias corresponds to an assumption that the classification of an instance $x_q$ will be most similar to the classification of other instances that are nearby in Euclidean distance.

One practical issue in applying $k$-NEAREST NEIGHBOR algorithms is that the distance between instances is calculated based on *all* attributes of the instance

(i.e., on all axes in the Euclidean space containing the instances). This lies in contrast to methods such as rule and decision tree learning systems that select only a subset of the instance attributes when forming the hypothesis. To see the effect of this policy, consider applying $k$-NEAREST NEIGHBOR to a problem in which each instance is described by 20 attributes, but where only 2 of these attributes are relevant to determining the classification for the particular target function. In this case, instances that have identical values for the 2 relevant attributes may nevertheless be distant from one another in the 20-dimensional instance space. As a result, the similarity metric used by $k$-NEAREST NEIGHBOR—depending on all 20 attributes—will be misleading. The distance between neighbors will be dominated by the large number of irrelevant attributes. This difficulty, which arises when many irrelevant attributes are present, is sometimes referred to as the *curse of dimensionality*. Nearest-neighbor approaches are especially sensitive to this problem.

One interesting approach to overcoming this problem is to weight each attribute differently when calculating the distance between two instances. This corresponds to stretching the axes in the Euclidean space, shortening the axes that correspond to less relevant attributes, and lengthening the axes that correspond to more relevant attributes. The amount by which each axis should be stretched can be determined automatically using a cross-validation approach. To see how, first note that we wish to stretch (multiply) the $j$th axis by some factor $z_j$, where the values $z_1 \ldots z_n$ are chosen to minimize the true classification error of the learning algorithm. Second, note that this true error can be estimated using cross-validation. Hence, one algorithm is to select a random subset of the available data to use as training examples, then determine the values of $z_1 \ldots z_n$ that lead to the minimum error in classifying the remaining examples. By repeating this process multiple times the estimate for these weighting factors can be made more accurate. This process of stretching the axes in order to optimize the performance of $k$-NEAREST NEIGHBOR provides a mechanism for suppressing the impact of irrelevant attributes.

An even more drastic alternative is to completely eliminate the least relevant attributes from the instance space. This is equivalent to setting some of the $z_i$ scaling factors to zero. Moore and Lee (1994) discuss efficient cross-validation methods for selecting relevant subsets of the attributes for $k$-NEAREST NEIGHBOR algorithms. In particular, they explore methods based on leave-one-out cross-validation, in which the set of $m$ training instances is repeatedly divided into a training set of size $m-1$ and test set of size 1, in all possible ways. This leave-one-out approach is easily implemented in $k$-NEAREST NEIGHBOR algorithms because no additional training effort is required each time the training set is redefined. Note both of the above approaches can be seen as stretching each axis by some constant factor. Alternatively, we could stretch each axis by a value that varies over the instance space. However, as we increase the number of degrees of freedom available to the algorithm for redefining its distance metric in such a fashion, we also increase the risk of overfitting. Therefore, the approach of locally stretching the axes is much less common.

One additional practical issue in applying $k$-NEAREST NEIGHBOR is efficient memory indexing. Because this algorithm delays all processing until a new query is received, significant computation can be required to process each new query. Various methods have been developed for indexing the stored training examples so that the nearest neighbors can be identified more efficiently at some additional cost in memory. One such indexing method is the *kd*-tree (Bentley 1975; Friedman et al. 1977), in which instances are stored at the leaves of a tree, with nearby instances stored at the same or nearby nodes. The internal nodes of the tree sort the new query $x_q$ to the relevant leaf by testing selected attributes of $x_q$.

### 8.2.3 A Note on Terminology

Much of the literature on nearest-neighbor methods and weighted local regression uses a terminology that has arisen from the field of statistical pattern recognition. In reading that literature, it is useful to know the following terms:

- *Regression* means approximating a real-valued target function.
- *Residual* is the error $\hat{f}(x) - f(x)$ in approximating the target function.
- *Kernel function* is the function of distance that is used to determine the weight of each training example. In other words, the kernel function is the function $K$ such that $w_i = K(d(x_i, x_q))$.

## 8.3 LOCALLY WEIGHTED REGRESSION

The nearest-neighbor approaches described in the previous section can be thought of as approximating the target function $f(x)$ at the single query point $x = x_q$. Locally weighted regression is a generalization of this approach. It constructs an explicit approximation to $f$ over a local region surrounding $x_q$. Locally weighted regression uses nearby or distance-weighted training examples to form this local approximation to $f$. For example, we might approximate the target function in the neighborhood surrounding $x_q$ using a linear function, a quadratic function, a multilayer neural network, or some other functional form. The phrase "locally weighted regression" is called *local* because the function is approximated based only on data near the query point, *weighted* because the contribution of each training example is weighted by its distance from the query point, and *regression* because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.

Given a new query instance $x_q$, the general approach in locally weighted regression is to construct an approximation $\hat{f}$ that fits the training examples in the neighborhood surrounding $x_q$. This approximation is then used to calculate the value $\hat{f}(x_q)$, which is output as the estimated target value for the query instance. The description of $\hat{f}$ may then be deleted, because a different local approximation will be calculated for each distinct query instance.

### 8.3.1 Locally Weighted Linear Regression

Let us consider the case of locally weighted regression in which the target function $f$ is approximated near $x_q$ using a linear function of the form

$$\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$$

As before, $a_i(x)$ denotes the value of the $i$th attribute of the instance $x$.

Recall that in Chapter 4 we discussed methods such as gradient descent to find the coefficients $w_0 \ldots w_n$ to minimize the error in fitting such linear functions to a given set of training examples. In that chapter we were interested in a global approximation to the target function. Therefore, we derived methods to choose weights that minimize the squared error summed over the set $D$ of training examples

$$E \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \tag{8.5}$$

which led us to the gradient descent training rule

$$\Delta w_j = \eta \sum_{x \in D} (f(x) - \hat{f}(x)) a_j(x) \tag{8.6}$$

where $\eta$ is a constant learning rate, and where the training rule has been re-expressed from the notation of Chapter 4 to fit our current notation (i.e., $t \rightarrow f(x)$, $o \rightarrow \hat{f}(x)$, and $x_j \rightarrow a_j(x)$).

How shall we modify this procedure to derive a local approximation rather than a global one? The simple way is to redefine the error criterion $E$ to emphasize fitting the local training examples. Three possible criteria are given below. Note we write the error $E(x_q)$ to emphasize the fact that now the error is being defined as a function of the query point $x_q$.

1. Minimize the squared error over just the $k$ nearest neighbors:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2$$

2. Minimize the squared error over the entire set $D$ of training examples, while weighting the error of each training example by some decreasing function $K$ of its distance from $x_q$:

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

Criterion two is perhaps the most esthetically pleasing because it allows every training example to have an impact on the classification of $x_q$. However,

this approach requires computation that grows linearly with the number of training examples. Criterion three is a good approximation to criterion two and has the advantage that computational cost is independent of the total number of training examples; its cost depends only on the number $k$ of neighbors considered.

If we choose criterion three above and rederive the gradient descent rule using the same style of argument as in Chapter 4, we obtain the following training rule (see Exercise 8.1):

$$\Delta w_j = \eta \sum_{x \in k \text{ nearest nbrs of } x_q} K(d(x_q, x))\ (f(x) - \hat{f}(x))\ a_j(x) \qquad (8.7)$$

Notice the only differences between this new rule and the rule given by Equation (8.6) are that the contribution of instance $x$ to the weight update is now multiplied by the distance penalty $K(d(x_q, x))$, and that the error is summed over only the $k$ nearest training examples. In fact, if we are fitting a linear function to a fixed set of training examples, then methods much more efficient than gradient descent are available to directly solve for the desired coefficients $w_0 \ldots w_n$. Atkeson et al. (1997a) and Bishop (1995) survey several such methods.

### 8.3.2 Remarks on Locally Weighted Regression

Above we considered using a linear function to approximate $f$ in the neighborhood of the query instance $x_q$. The literature on locally weighted regression contains a broad range of alternative methods for distance weighting the training examples, and a range of methods for locally approximating the target function. In most cases, the target function is approximated by a constant, linear, or quadratic function. More complex functional forms are not often found because (1) the cost of fitting more complex functions for each query instance is prohibitively high, and (2) these simple approximations model the target function quite well over a sufficiently small subregion of the instance space.

## 8.4 RADIAL BASIS FUNCTIONS

One approach to function approximation that is closely related to distance-weighted regression and also to artificial neural networks is learning with radial basis functions (Powell 1987; Broomhead and Lowe 1988; Moody and Darken 1989). In this approach, the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \qquad (8.8)$$

where each $x_u$ is an instance from $X$ and where the kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance $d(x_u, x)$ increases. Here $k$ is a user-provided constant that specifies the number of kernel functions to be included. Even though $\hat{f}(x)$ is a global approximation to $f(x)$, the contribution from each of the $K_u(d(x_u, x))$ terms is localized to a region nearby the point $x_u$. It is common

to choose each function $K_u(d(x_u, x))$ to be a Gaussian function (see Table 5.4) centered at the point $x_u$ with some variance $\sigma_u^2$.

$$K_u(d(x_u, x)) = e^{\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

We will restrict our discussion here to this common Gaussian kernel function. As shown by Hartman et al. (1990), the functional form of Equation (8.8) can approximate any function with arbitrarily small error, provided a sufficiently large number $k$ of such Gaussian kernels and provided the width $\sigma^2$ of each kernel can be separately specified.

The function given by Equation (8.8) can be viewed as describing a two-layer network where the first layer of units computes the values of the various $K_u(d(x_u, x))$ and where the second layer computes a linear combination of these first-layer unit values. An example radial basis function (RBF) network is illustrated in Figure 8.2.

Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process. First, the number $k$ of hidden units is determined and each hidden unit $u$ is defined by choosing the values of $x_u$ and $\sigma_u^2$ that define its kernel function $K_u(d(x_u, x))$. Second, the weights $w_u$ are trained to maximize the fit of the network to the training data, using the global error criterion given by Equation (8.5). Because the kernel functions are held fixed during this second stage, the linear weight values $w_u$ can be trained very efficiently.

Several alternative methods have been proposed for choosing an appropriate number of hidden units or, equivalently, kernel functions. One approach is to allocate a Gaussian kernel function for each training example $\langle x_i, f(x_i)\rangle$, centering this Gaussian at the point $x_i$. Each of these kernels may be assigned the same width $\sigma^2$. Given this approach, the RBF network learns a global approximation to the target function in which each training example $\langle x_i, f(x_i)\rangle$ can influence the value of $\hat{f}$ only in the neighborhood of $x_i$. One advantage of this choice of kernel functions is that it allows the RBF network to fit the training data exactly. That is, for any set of $m$ training examples the weights $w_0 \ldots w_m$ for combining the $m$ Gaussian kernel functions can be set so that $\hat{f}(x_i) = f(x_i)$ for each training example $\langle x_i, f(x_i)\rangle$.

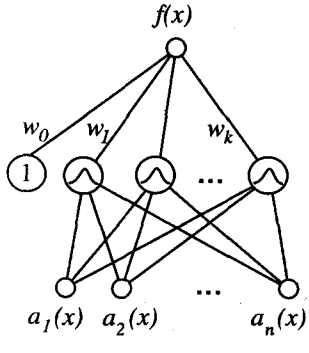



**FIGURE 8.2**
A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at some instance $x_u$. Therefore, its activation will be close to zero unless the input $x$ is near $x_u$. The output unit produces a linear combination of the hidden unit activations. Although the network shown here has just one output, multiple output units can also be included.

A second approach is to choose a set of kernel functions that is smaller than the number of training examples. This approach can be much more efficient than the first approach, especially when the number of training examples is large. The set of kernel functions may be distributed with centers spaced uniformly throughout the instance space $X$. Alternatively, we may wish to distribute the centers nonuniformly, especially if the instances themselves are found to be distributed nonuniformly over $X$. In this later case, we can pick kernel function centers by randomly selecting a subset of the training instances, thereby sampling the underlying distribution of instances. Alternatively, we may identify prototypical clusters of instances, then add a kernel function centered at each cluster. The placement of the kernel functions in this fashion can be accomplished using unsupervised clustering algorithms that fit the training instances (but not their target values) to a mixture of Gaussians. The EM algorithm discussed in Section 6.12.1 provides one algorithm for choosing the means of a mixture of $k$ Gaussians to best fit the observed instances. In the case of the EM algorithm, the means are chosen to maximize the probability of observing the instances $x_i$, given the $k$ estimated means. Note the target function value $f(x_i)$ of the instance does not enter into the calculation of kernel centers by unsupervised clustering methods. The only role of the target values $f(x_i)$ in this case is to determine the output layer weights $w_u$.

To summarize, radial basis function networks provide a global approximation to the target function, represented by a linear combination of many local kernel functions. The value for any given kernel function is non-negligible only when the input $x$ falls into the region defined by its particular center and width. Thus, the network can be viewed as a smooth linear combination of many local approximations to the target function. One key advantage to RBF networks is that they can be trained much more efficiently than feedforward networks trained with BACKPROPAGATION. This follows from the fact that the input layer and the output layer of an RBF are trained separately.

## 8.5 CASE-BASED REASONING

Instance-based methods such as $k$-NEAREST NEIGHBOR and locally weighted regression share three key properties. First, they are *lazy* learning methods in that they defer the decision of how to generalize beyond the training data until a new query instance is observed. Second, they classify new query instances by analyzing similar instances while ignoring instances that are very different from the query. Third, they represent instances as real-valued points in an $n$-dimensional Euclidean space. Case-based reasoning (CBR) is a learning paradigm based on the first two of these principles, but not the third. In CBR, instances are typically represented using more rich symbolic descriptions, and the methods used to retrieve similar instances are correspondingly more elaborate. CBR has been applied to problems such as conceptual design of mechanical devices based on a stored library of previous designs (Sycara et al. 1992), reasoning about new legal cases based on previous rulings (Ashley 1990), and solving planning and

scheduling problems by reusing and combining portions of previous solutions to similar problems (Veloso 1992).

Let us consider a prototypical example of a case-based reasoning system to ground our discussion. The CADET system (Sycara et al. 1992) employs case-based reasoning to assist in the conceptual design of simple mechanical devices such as water faucets. It uses a library containing approximately 75 previous designs and design fragments to suggest conceptual designs to meet the specifications of new design problems. Each instance stored in memory (e.g., a water pipe) is represented by describing both its structure and its qualitative function. New design problems are then presented by specifying the desired function and requesting the corresponding structure. This problem setting is illustrated in Figure 8.3. The top half of the figure shows the description of a typical stored case called a T-junction pipe. Its function is represented in terms of the qualitative relationships among the waterflow levels and temperatures at its inputs and outputs. In the functional description at its right, an arrow with a "+" label indicates that the variable at the arrowhead increases with the variable at its tail. For example, the output waterflow $Q_3$ increases with increasing input waterflow $Q_1$. Similarly,

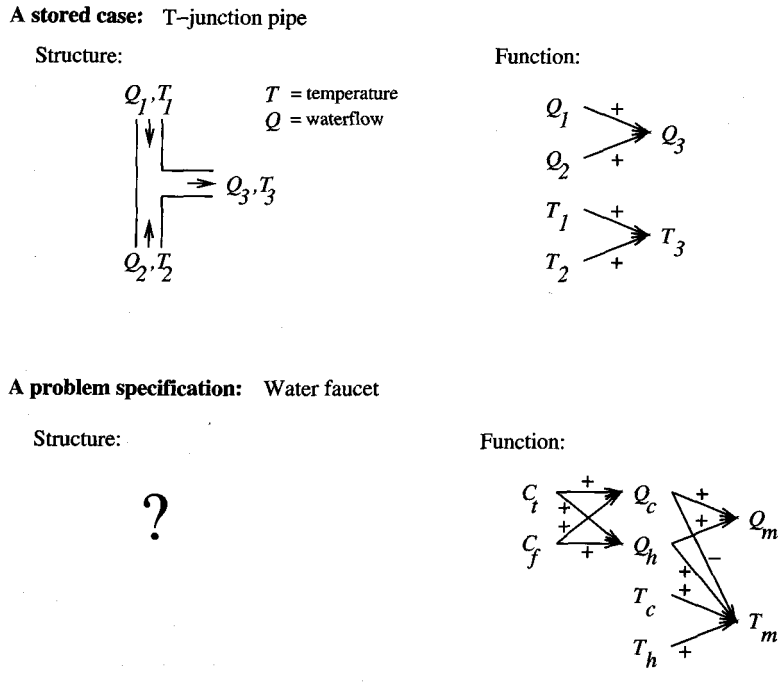



**FIGURE 8.3**
A stored case and a new problem. The top half of the figure describes a typical design fragment in the case library of CADET. The function is represented by the graph of qualitative dependencies among the T-junction variables (described in the text). The bottom half of the figure shows a typical design problem.

a "–" label indicates that the variable at the head decreases with the variable at the tail. The bottom half of this figure depicts a new design problem described by its desired function. This particular function describes the required behavior of one type of water faucet. Here $Q_c$ refers to the flow of cold water into the faucet, $Q_h$ to the input flow of hot water, and $Q_m$ to the single mixed flow out of the faucet. Similarly, $T_c$, $T_h$, and $T_m$ refer to the temperatures of the cold water, hot water, and mixed water respectively. The variable $C_t$ denotes the control signal for temperature that is input to the faucet, and $C_f$ denotes the control signal for waterflow. Note the description of the desired function specifies that these controls $C_t$ and $C_f$ are to influence the water flows $Q_c$ and $Q_h$, thereby indirectly influencing the faucet output flow $Q_m$ and temperature $T_m$.

Given this functional specification for the new design problem, CADET searches its library for stored cases whose functional descriptions match the design problem. If an exact match is found, indicating that some stored case implements exactly the desired function, then this case can be returned as a suggested solution to the design problem. If no exact match occurs, CADET may find cases that match various subgraphs of the desired functional specification. In Figure 8.3, for example, the T-junction function matches a subgraph of the water faucet function graph. More generally, CADET searches for subgraph isomorphisms between the two function graphs, so that parts of a case can be found to match parts of the design specification. Furthermore, the system may elaborate the original function specification graph in order to create functionally equivalent graphs that may match still more cases. It uses general knowledge about physical influences to create these elaborated function graphs. For example, it uses a rewrite rule that allows it to rewrite the influence

$$A \xrightarrow{+} B$$

as

$$A \xrightarrow{+} x \xrightarrow{+} B$$

This rewrite rule can be interpreted as stating that if $B$ must increase with $A$, then it is sufficient to find some other quantity $x$ such that $B$ increases with $x$, and $x$ increases with $A$. Here $x$ is a universally quantified variable whose value is bound when matching the function graph against the case library. In fact, the function graph for the faucet shown in Figure 8.3 is an elaboration of the original functional specification produced by applying such rewrite rules.

By retrieving multiple cases that match different subgraphs, the entire design can sometimes be pieced together. In general, the process of producing a final solution from multiple retrieved cases can be very complex. It may require designing portions of the system from first principles, in addition to merging retrieved portions from stored cases. It may also require backtracking on earlier choices of design subgoals and, therefore, rejecting cases that were previously retrieved. CADET has very limited capabilities for combining and adapting multiple retrieved cases to form the final design and relies heavily on the user for this adaptation stage of the process. As described by Sycara et al. (1992), CADET is

a research prototype system intended to explore the potential role of case-based reasoning in conceptual design. It does not have the range of analysis algorithms needed to refine these abstract conceptual designs into final designs.

It is instructive to examine the correspondence between the problem setting of CADET and the general setting for instance-based methods such as $k$-NEAREST NEIGHBOR. In CADET each stored training example describes a function graph along with the structure that implements it. New queries correspond to new function graphs. Thus, we can map the CADET problem into our standard notation by defining the space of instances $X$ to be the space of all function graphs. The target function $f$ maps function graphs to the structures that implement them. Each stored training example $\langle x, f(x)\rangle$ is a pair that describes some function graph $x$ and the structure $f(x)$ that implements $x$. The system must learn from the training example cases to output the structure $f(x_q)$ that successfully implements the input function graph query $x_q$.

The above sketch of the CADET system illustrates several generic properties of case-based reasoning systems that distinguish them from approaches such as $k$-NEAREST NEIGHBOR.

- Instances or cases may be represented by rich symbolic descriptions, such as the function graphs used in CADET. This may require a similarity metric different from Euclidean distance, such as the size of the largest shared subgraph between two function graphs.
- Multiple retrieved cases may be combined to form the solution to the new problem. This is similar to the $k$-NEAREST NEIGHBOR approach, in that multiple similar cases are used to construct a response for the new query. However, the process for combining these multiple retrieved cases can be very different, relying on knowledge-based reasoning rather than statistical methods.
- There may be a tight coupling between case retrieval, knowledge-based reasoning, and problem solving. One simple example of this is found in CADET, which uses generic knowledge about influences to rewrite function graphs during its attempt to find matching cases. Other systems have been developed that more fully integrate case-based reasoning into general search-based problem-solving systems. Two examples are ANAPRON (Golding and Rosenbloom 1991) and PRODIGY/ANALOGY (Veloso 1992).

To summarize, case-based reasoning is an instance-based learning method in which instances (cases) may be rich relational descriptions and in which the retrieval and combination of cases to solve the current query may rely on knowledge-based reasoning and search-intensive problem-solving methods. One current research issue in case-based reasoning is to develop improved methods for indexing cases. The central issue here is that syntactic similarity measures (e.g., subgraph isomorphism between function graphs) provide only an approximate indication of the relevance of a particular case to a particular problem. When the CBR system attempts to reuse the retrieved cases it may uncover difficulties that were not

captured by this syntactic similarity measure. For example, in CADET the multiple retrieved design fragments may turn out to be incompatible with one another, making it impossible to combine them into a consistent final design. When this occurs in general, the CBR system may backtrack and search for additional cases, adapt the existing cases, or resort to other problem-solving methods. Importantly, when such difficulties are detected they also provide training data for improving the similarity metric or, equivalently, the indexing structure for the case library. In particular, if a case is retrieved based on the similarity metric, but found to be irrelevant based on further analysis, then the similarity metric should be refined to reject this case for similar subsequent queries.

## 8.6 REMARKS ON LAZY AND EAGER LEARNING

In this chapter we considered three *lazy* learning methods: the $k$-NEAREST NEIGHBOR algorithm, locally weighted regression, and case-based reasoning. We call these methods lazy because they defer the decision of how to generalize beyond the training data until each new query instance is encountered. We also discussed one *eager* learning method: the method for learning radial basis function networks. We call this method eager because it generalizes beyond the training data before observing the new query, committing at training time to the network structure and weights that define its approximation to the target function. In this same sense, every other algorithm discussed elsewhere in this book (e.g., BACKPROPAGATION, C4.5) is an eager learning algorithm.

Are there important differences in what can be achieved by lazy versus eager learning? Let us distinguish between two kinds of differences: differences in computation time and differences in the classifications produced for new queries. There are obviously differences in computation time between eager and lazy methods. For example, lazy methods will generally require less computation during training, but more computation when they must predict the target value for a new query.

The more fundamental question is whether there are essential differences in the inductive bias that can be achieved by lazy versus eager methods. The key difference between lazy and eager methods in this regard is

- Lazy methods may consider the query instance $x_q$ when deciding how to generalize beyond the training data $D$.
- Eager methods cannot. By the time they observe the query instance $x_q$ they have already chosen their (global) approximation to the target function.

Does this distinction affect the generalization accuracy of the learner? It does if we require that the lazy and eager learner employ the same hypothesis space $H$. To illustrate, consider the hypothesis space consisting of linear functions. The locally weighted linear regression algorithm discussed earlier is a lazy learning method based on this hypothesis space. For each new query $x_q$ it generalizes from the training data by choosing a new hypothesis based on the training examples near $x_q$. In contrast, an eager learner that uses the same hypothesis space of linear functions

must choose its approximation before the queries are observed. The eager learner must therefore commit to a single linear function hypothesis that covers the entire instance space and all future queries. The lazy method effectively uses a richer hypothesis space because it uses many different local linear functions to form its implicit global approximation to the target function. Note this same situation holds for other learners and hypothesis spaces as well. A lazy version of BACKPROPAGATION, for example, could learn a different neural network for each distinct query point, compared to the eager version of BACKPROPAGATION discussed in Chapter 4.

The key point in the above paragraph is that a lazy learner has the option of (implicitly) representing the target function by a combination of many local approximations, whereas an eager learner must commit at training time to a single global approximation. The distinction between eager and lazy learning is thus related to the distinction between global and local approximations to the target function.

Can we create eager methods that use multiple local approximations to achieve the same effects as lazy local methods? Radial basis function networks can be seen as one attempt to achieve this. The RBF learning methods we discussed are eager methods that commit to a global approximation to the target function at training time. However, an RBF network represents this global function as a linear combination of multiple local kernel functions. Nevertheless, because RBF learning methods must commit to the hypothesis before the query point is known, the local approximations they create are not specifically targeted to the query point to the same degree as in a lazy learning method. Instead, RBF networks are built eagerly from local approximations centered around the training examples, or around clusters of training examples, but not around the unknown future query points.

To summarize, lazy methods have the option of selecting a different hypothesis or local approximation to the target function for each query instance. Eager methods using the same hypothesis space are more restricted because they must commit to a single hypothesis that covers the entire instance space. Eager methods can, of course, employ hypothesis spaces that combine multiple local approximations, as in RBF networks. However, even these combined local approximations do not give eager methods the full ability of lazy methods to customize to unknown future query instances.

## 8.7 SUMMARY AND FURTHER READING

The main points of this chapter include:

- Instance-based learning methods differ from other approaches to function approximation because they delay processing of training examples until they must label a new query instance. As a result, they need not form an explicit hypothesis of the entire target function over the entire instance space, independent of the query instance. Instead, they may form a different local approximation to the target function for each query instance.

- Advantages of instance-based methods include the ability to model complex target functions by a collection of less complex local approximations and the fact that information present in the training examples is never lost (because the examples themselves are stored explicitly). The main practical difficulties include efficiency of labeling new instances (all processing is done at query time rather than in advance), difficulties in determining an appropriate distance metric for retrieving "related" instances (especially when examples are represented by complex symbolic descriptions), and the negative impact of irrelevant features on the distance metric.
- $k$-NEAREST NEIGHBOR is an instance-based algorithm for approximating real-valued or discrete-valued target functions, assuming instances correspond to points in an $n$-dimensional Euclidean space. The target function value for a new query is estimated from the known values of the $k$ nearest training examples.
- Locally weighted regression methods are a generalization of $k$-NEAREST NEIGHBOR in which an explicit local approximation to the target function is constructed for each query instance. The local approximation to the target function may be based on a variety of functional forms such as constant, linear, or quadratic functions or on spatially localized kernel functions.
- Radial basis function (RBF) networks are a type of artificial neural network constructed from spatially localized kernel functions. These can be seen as a blend of instance-based approaches (spatially localized influence of each kernel function) and neural network approaches (a global approximation to the target function is formed at training time rather than a local approximation at query time). Radial basis function networks have been used successfully in applications such as interpreting visual scenes, in which the assumption of spatially local influences is well-justified.
- Case-based reasoning is an instance-based approach in which instances are represented by complex logical descriptions rather than points in a Euclidean space. Given these complex symbolic descriptions of instances, a rich variety of methods have been proposed for mapping from the training examples to target function values for new instances. Case-based reasoning methods have been used in applications such as modeling legal reasoning and for guiding searches in complex manufacturing and transportation planning problems.

The $k$-NEAREST NEIGHBOR algorithm is one of the most thoroughly analyzed algorithms in machine learning, due in part to its age and in part to its simplicity. Cover and Hart (1967) present early theoretical results, and Duda and Hart (1973) provide a good overview. Bishop (1995) provides a discussion of $k$-NEAREST NEIGHBOR and its relation to estimating probability densities. An excellent current survey of methods for locally weighted regression is given by Atkeson et al. (1997). The application of these methods to robot control is surveyed by Atkeson et al. (1997b).

A thorough discussion of radial basis functions is provided by Bishop (1995). Other treatments are given by Powell (1987) and Poggio and Girosi (1990). See Section 6.12 of this book for a discussion of the EM algorithm and its application to selecting the means of a mixture of Gaussians.

Kolodner (1993) provides a general introduction to case-based reasoning. Other general surveys and collections describing recent research are given by Aamodt et al. (1994), Aha et al. (1991), Haton et al. (1995), Riesbeck and Schank (1989), Schank et al. (1994), Veloso and Aamodt (1995), Watson (1995), and Wess et al. (1994).

## EXERCISES

**8.1.** Derive the gradient descent rule for a distance-weighted local linear approximation to the target function, given by Equation (8.1).

**8.2.** Consider the following alternative method for accounting for distance in weighted local regression. Create a virtual set of training examples $D'$ as follows: For each training example $\langle x, f(x)\rangle$ in the original data set $D$, create some (possibly fractional) number of copies of $\langle x, f(x)\rangle$ in $D'$, where the number of copies is $K(d(x_q, x))$. Now train a linear approximation to minimize the error criterion

$$E_4 \equiv \frac{1}{2}\sum_{x \in D'}(f(x) - \hat{f}(x))^2$$

The idea here is to make more copies of training examples that are near the query instance, and fewer of those that are distant. Derive the gradient descent rule for this criterion. Express the rule in the form of a sum over members of $D$ rather than $D'$, and compare it with the rules given by Equations (8.6) and (8.7).

**8.3.** Suggest a lazy version of the eager decision tree learning algorithm ID3 (see Chapter 3). What are the advantages and disadvantages of your lazy algorithm compared to the original eager algorithm?

## REFERENCES

Aamodt, A., & Plazas, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1), 39–52.

Aha, D., & Kibler, D. (1989). Noise-tolerant instance-based learning algorithms. *Proceedings of the IJCAI-89* (794–799).

Aha, D., Kibler, D., & Albert, M. (1991). Instance-based learning algorithms. *Machine Learning*, 6, 37–66.

Ashley, K. D. (1990). *Modeling legal argument: Reasoning with cases and hypotheticals*. Cambridge, MA: MIT Press.

Atkeson, C. G., Schaal, S. A., & Moore, A. W. (1997a). Locally weighted learning. *AI Review*, (to appear).

Atkeson, C. G., Moore, A. W., & Schaal, S. A. (1997b). Locally weighted learning for control. *AI Review*, (to appear).

Bareiss, E. R., Porter, B., & Weir, C. C. (1988). PROTOS: An exemplar-based learning apprentice. *International Journal of Man-Machine Studies*, 29, 549–561.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 509–517.

Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford, England: Oxford University Press.

Bisio, R., & Malabocchia, F. (1995). Cost estimation of software projects through case-based reasoning. In M. Veloso and A. Aamodt (Eds.), *Lecture Notes in Artificial Intelligence* (pp. 11–22). Berlin: Springer-Verlag.

Broomhead, D. S., & Lowe, D. (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2, 321–355.

Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13, 21–27.

Duda, R., & Hart, P. (1973). *Pattern classification and scene analysis*. New York: John Wiley & Sons.

Franke, R. (1982). Scattered data interpolation: Tests of some methods. *Mathematics of Computation*, 38, 181–200.

Friedman, J., Bentley, J., & Finkel, R. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3), 209–226.

Golding, A., & Rosenbloom, P. (1991). Improving rule-based systems through case-based reasoning. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 22–27). Cambridge: AAAI Press/The MIT Press.

Hartman, E. J., Keller, J. D., & Kowalski, J. M. (1990). Layered neural networks with Gaussian hidden units as universal approximations. *Neural Computation*, 2(2), 210–215.

Haton, J.-P., Keane, M., & Manago, M. (Eds.). (1995). *Advances in case-based reasoning: Second European workshop*. Berlin: Springer-Verlag.

Kolodner, J. L. (1993). *Case-Based Reasoning*. San Francisco: Morgan Kaufmann.

Moody, J. E., & Darken, C. J. (1989). Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1(2), 281–294.

Moore, A. W., & Lee, M. S. (1994). Efficient algorithms for minimizing cross validation error. *Proceedings of the 11th International Conference on Machine Learning*. San Francisco: Morgan Kaufmann.

Poggio, T., & Girosi, F. (1990). Networks for approximation and learning. *Proceedings of the IEEE*, 78(9), 1481–1497.

Powell, M. J. D. (1987). Radial basis functions for multivariable interpolation: A review. In Mason, J., & Cox, M. (Eds.). *Algorithms for approximation* (pp. 143–167). Oxford: Clarendon Press.

Riesbeck, C., & Schank, R. (1989). *Inside case-based reasoning*. Hillsdale, NJ: Lawrence Erlbaum.

Schank, R. (1982). *Dynamic Memory*. Cambridge, England: Cambridge University Press.

Schank, R., Riesbeck, C., & Kass, A. (1994). *Inside case-based explanation*. Hillsdale, NJ: Lawrence Erlbaum.

Shepard, D. (1968). A two-dimensional interpolation function for irregularly spaced data. *Proceedings of the 23rd National Conference of the ACM* (pp. 517–523).

Stanfill, C., & Waltz, D. (1986). Toward memory-based reasoning. *Communications of the ACM*, 29(12), 1213–1228.

Sycara, K., Guttal, R., Koning, J., Narasimhan, S., & Navinchandra, D. (1992). CADET: A case-based synthesis tool for engineering design. *International Journal of Expert Systems*, 4(2), 157–188.

Veloso, M. M. (1992). *Planning and learning by analogical reasoning*. Berlin: Springer-Verlag.

Veloso, M. M., & Aamodt, A. (Eds.). (1995). *Case-based reasoning research and development*. Lecture Notes in Artificial Intelligence. Berlin: Springer-Verlag.

Watson, I. (Ed.). (1995). *Progress in case-based reasoning: First United Kingdom workshop*. Berlin: Springer-Verlag.

Wess, S., Althoff, K., & Richter, M. (Eds.). (1994). *Topics in case-based reasoning*. Berlin: Springer-Verlag.

# CHAPTER 13

# REINFORCEMENT LEARNING

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals. This very generic problem covers tasks such as learning to control a mobile robot, learning to optimize operations in factories, and learning to play board games. Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state. For example, when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states. The task of the agent is to learn from this indirect, delayed reward, to choose sequences of actions that produce the greatest cumulative reward. This chapter focuses on an algorithm called $Q$ learning that can acquire optimal control strategies from delayed rewards, even when the agent has no prior knowledge of the effects of its actions on the environment. Reinforcement learning algorithms are related to dynamic programming algorithms frequently used to solve optimization problems.

## 13.1 INTRODUCTION

Consider building a learning robot. The robot, or *agent*, has a set of sensors to observe the *state* of its environment, and a set of *actions* it can perform to alter this state. For example, a mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn." Its task is to learn a control strategy, or *policy*, for choosing actions that achieve its goals. For example, the robot may have a goal of docking onto its battery charger whenever its battery level is low.

This chapter is concerned with how such agents can learn successful control policies by experimenting in their environment. We assume that the goals of the agent can be defined by a *reward* function that assigns a numerical value—an immediate payoff—to each distinct action the agent may take from each distinct state. For example, the goal of docking to the battery charger can be captured by assigning a positive reward (e.g., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition. This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot. The task of the robot is to perform sequences of actions, observe their consequences, and learn a control policy. The control policy we desire is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent. This general setting for robot learning is summarized in Figure 13.1.

As is apparent from Figure 13.1, the problem of learning a control policy to maximize cumulative reward is very general and covers many problems beyond robot learning tasks. In general the problem is one of learning to control sequential processes. This includes, for example, manufacturing optimization problems in which a sequence of manufacturing actions must be chosen, and the reward to be maximized is the value of the goods produced minus the costs involved. It includes sequential scheduling problems such as choosing which taxis to send for passengers in a large city, where the reward to be maximized is a function of the wait time of the passengers and the total fuel costs of the taxi fleet. In general, we are interested in any type of agent that must learn to choose actions that alter the state of its environment and where a cumulative reward function is used to define the quality of any given action sequence. Within this class of problems we will consider specific settings, including settings in which the actions have deterministic or nondeterministic outcomes, and settings in which the agent

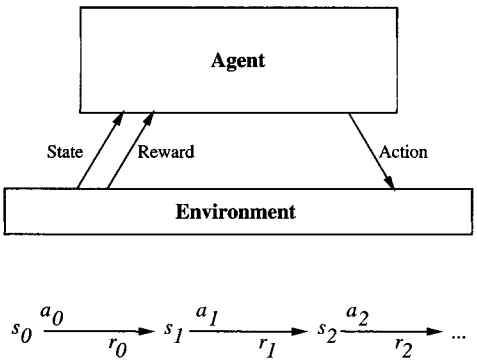



**FIGURE 13.1**
An agent interacting with its environment. The agent exists in an environment described by some set of possible states $S$. It can perform any of a set of possible actions $A$. Each time it performs an action $a_t$ in some state $s_t$ the agent receives a real-valued reward $r_t$ that indicates the immediate value of this state-action transition. This produces a sequence of states $s_i$, actions $a_i$, and immediate rewards $r_i$ as shown in the figure. The agent's task is to learn a control policy, $\pi : S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

has or does not have prior knowledge about the effects of its actions on the environment.

Note we have touched on the problem of learning to control sequential processes earlier in this book. In Section 11.4 we discussed explanation-based learning of rules to control search during problem solving. There the problem is for the agent to choose among alternative actions at each step in its search for some goal state. The techniques discussed here differ from those of Section 11.4, in that here we consider problems where the actions may have nondeterministic outcomes and where the learner lacks a domain theory that describes the outcomes of its actions. In Chapter 1 we discussed the problem of learning to choose actions while playing the game of checkers. There we sketched the design of a learning method very similar to those discussed in this chapter. In fact, one highly successful application of the reinforcement learning algorithms of this chapter is to a similar game-playing problem. Tesauro (1995) describes the TD-GAMMON program, which has used reinforcement learning to become a world-class backgammon player. This program, after training on 1.5 million self-generated games, is now considered nearly equal to the best human players in the world and has played competitively against top-ranked players in international backgammon tournaments.

The problem of learning a control policy to choose actions is similar in some respects to the function approximation problems discussed in other chapters. The target function to be learned in this case is a control policy, $\pi : S \rightarrow A$, that outputs an appropriate action $a$ from the set $A$, given the current state $s$ from the set $S$. However, this reinforcement learning problem differs from other function approximation tasks in several important respects.

- *Delayed reward.* The task of the agent is to learn a target function $\pi$ that maps from the current state $s$ to the optimal action $a = \pi(s)$. In earlier chapters we have always assumed that when learning some target function such as $\pi$, each training example would be a pair of the form $\langle s, \pi(s)\rangle$. In reinforcement learning, however, training information is not available in this form. Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of *temporal credit assignment*: determining which of the actions in its sequence are to be credited with producing the eventual rewards.
- *Exploration.* In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective learning. The learner faces a tradeoff in choosing whether to favor *exploration* of unknown states and actions (to gather new information), or *exploitation* of states and actions that it has already learned will yield high reward (to maximize its cumulative reward).
- *Partially observable states.* Although it is convenient to assume that the agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information. For example, a robot with a forward-pointing camera cannot see what is

behind it. In such cases, it may be necessary for the agent to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment.

- *Life-long learning.* Unlike isolated function approximation tasks, robot learning often requires that the robot learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

## 13.2 THE LEARNING TASK

In this section we formulate the problem of learning sequential control strategies more precisely. Note there are many ways to do so. For example, we might assume the agent's actions are deterministic or that they are nondeterministic. We might assume that the agent can predict the next state that will result from each action, or that it cannot. We might assume that the agent is trained by an expert who shows it examples of optimal action sequences, or that it must train itself by performing actions of its own choice. Here we define one quite general formulation of the problem, based on Markov decision processes. This formulation of the problem follows the problem illustrated in Figure 13.1.

In a Markov decision process (MDP) the agent can perceive a set $S$ of distinct states of its environment and has a set $A$ of actions that it can perform. At each discrete time step $t$, the agent senses the current state $s_t$, chooses a current action $a_t$, and performs it. The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$. Here the functions $\delta$ and $r$ are part of the environment and are not necessarily known to the agent. In an MDP, the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions. In this chapter we consider only the case in which $S$ and $A$ are finite. In general, $\delta$ and $r$ may be nondeterministic functions, but we begin by considering only the deterministic case.

The task of the agent is to learn a *policy*, $\pi : S \rightarrow A$, for selecting its next action $a_t$ based on the current observed state $s_t$; that is, $\pi(s_t) = a_t$. How shall we specify precisely which policy $\pi$ we would like the agent to learn? One obvious approach is to require the policy that produces the greatest possible cumulative reward for the robot over time. To state this requirement more precisely, we define the cumulative value $V^\pi(s_t)$ achieved by following an arbitrary policy $\pi$ from an arbitrary initial state $s_t$ as follows:

$$V^\pi(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$
$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{13.1}$$

where the sequence of rewards $r_{t+i}$ is generated by beginning at state $s_t$ and by repeatedly using the policy $\pi$ to select actions as described above (i.e., $a_t = \pi(s_t)$, $a_{t+1} = \pi(s_{t+1})$, etc.). Here $0 \leq \gamma < 1$ is a constant that determines the relative value of delayed versus immediate rewards. In particular, rewards received $i$ time steps into the future are discounted exponentially by a factor of $\gamma^i$. Note if we set $\gamma = 0$, only the immediate reward is considered. As we set $\gamma$ closer to 1, future rewards are given greater emphasis relative to the immediate reward.

The quantity $V^\pi(s)$ defined by Equation (13.1) is often called the *discounted cumulative reward* achieved by policy $\pi$ from initial state $s$. It is reasonable to discount future rewards relative to immediate rewards because, in many cases, we prefer to obtain the reward sooner rather than later. However, other definitions of total reward have also been explored. For example, *finite horizon* reward, $\sum_{i=0}^{h} r_{t+i}$, considers the undiscounted sum of rewards over a finite number $h$ of steps. Another possibility is *average reward*, $\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$, which considers the average reward per time step over the entire lifetime of the agent. In this chapter we restrict ourselves to considering discounted reward as defined by Equation (13.1). Mahadevan (1996) provides a discussion of reinforcement learning when the criterion to be optimized is average reward.

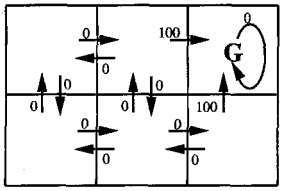
We are now in a position to state precisely the agent's learning task. We require that the agent learn a policy $\pi$ that maximizes $V^\pi(s)$ for all states $s$. We will call such a policy an *optimal policy* and denote it by $\pi^*$.

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^\pi(s), (\forall s) \tag{13.2}$$

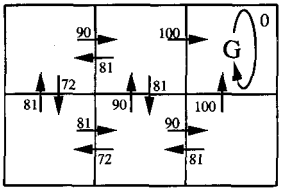
To simplify notation, we will refer to the value function $V^{\pi^*}(s)$ of such an optimal policy as $V^*(s)$. $V^*(s)$ gives the maximum discounted cumulative reward that the agent can obtain starting from state $s$; that is, the discounted cumulative reward obtained by following the optimal policy beginning at state $s$.

To illustrate these concepts, a simple grid-world environment is depicted in the topmost diagram of Figure 13.2. The six grid squares in this diagram represent six possible states, or locations, for the agent. Each arrow in the diagram represents a possible action the agent can take to move from one state to another. The number associated with each arrow represents the immediate reward $r(s, a)$ the agent receives if it executes the corresponding state-action transition. Note the immediate reward in this particular environment is defined to be zero for all state-action transitions except for those leading into the state labeled **G**. It is convenient to think of the state **G** as the goal state, because the only way the agent can receive reward, in this case, is by entering this state. Note in this particular environment, the only action available to the agent once it enters the state **G** is to remain in this state. For this reason, we call **G** an *absorbing* state.

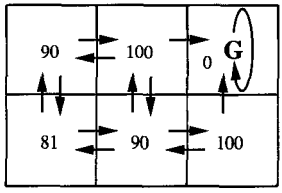
Once the states, actions, and immediate rewards are defined, and once we choose a value for the discount factor $\gamma$, we can determine the optimal policy $\pi^*$ and its value function $V^*(s)$. In this case, let us choose $\gamma = 0.9$. The diagram at the bottom of the figure shows one optimal policy for this setting (there are others as well). Like any policy, this policy specifies exactly one action that the



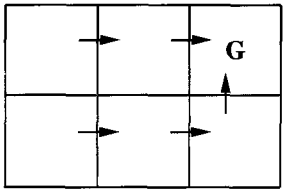

$r(s, a)$ (immediate reward) values




$Q(s, a)$ values




$V^*(s)$ values




One optimal policy

**FIGURE 13.2**
A simple deterministic world to illustrate the basic concepts of $Q$-learning. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, $r(s, a)$ gives reward 100 for actions entering the goal state **G**, and zero otherwise. Values of $V^*(s)$ and $Q(s, a)$ follow from $r(s, a)$, and the discount factor $\gamma = 0.9$. An optimal policy, corresponding to actions with maximal $Q$ values, is also shown.

agent will select in any given state. Not surprisingly, the optimal policy directs the agent along the shortest path toward the state **G**.

The diagram at the right of Figure 13.2 shows the values of $V^*$ for each state. For example, consider the bottom right state in this diagram. The value of $V^*$ for this state is 100 because the optimal policy in this state selects the "move up" action that receives immediate reward 100. Thereafter, the agent will remain in the absorbing state and receive no further rewards. Similarly, the value of $V^*$ for the bottom center state is 90. This is because the optimal policy will move the agent from this state to the right (generating an immediate reward of zero), then upward (generating an immediate reward of 100). Thus, the discounted future reward from the bottom center state is

$$0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + \cdots = 90$$

Recall that $V^*$ is defined to be the sum of discounted future rewards over the infinite future. In this particular environment, once the agent reaches the absorbing state **G** its infinite future will consist of remaining in this state and receiving rewards of zero.

## 13.3 $Q$ LEARNING

How can an agent learn an optimal policy $\pi^*$ for an arbitrary environment? It is difficult to learn the function $\pi^* : S \rightarrow A$ directly, because the available training data does not provide training examples of the form $\langle s, a \rangle$. Instead, the only training information available to the learner is the sequence of immediate rewards $r(s_i, a_i)$ for $i = 0, 1, 2, \ldots$. As we shall see, given this kind of training information it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

What evaluation function should the agent attempt to learn? One obvious choice is $V^*$. The agent should prefer state $s_1$ over state $s_2$ whenever $V^*(s_1) > V^*(s_2)$, because the cumulative future reward will be greater from $s_1$. Of course the agent's policy must choose among actions, not among states. However, it can use $V^*$ in certain settings to choose among actions as well. The optimal action in state $s$ is the action $a$ that maximizes the sum of the immediate reward $r(s, a)$ plus the value $V^*$ of the immediate successor state, discounted by $\gamma$.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s, a) + \gamma V^*(\delta(s, a))] \tag{13.3}$$

(recall that $\delta(s, a)$ denotes the state resulting from applying action $a$ to state $s$.) Thus, the agent can acquire the optimal policy by learning $V^*$, *provided it has perfect knowledge of the immediate reward function $r$ and the state transition function $\delta$*. When the agent knows the functions $r$ and $\delta$ used by the environment to respond to its actions, it can then use Equation (13.3) to calculate the optimal action for any state $s$.

Unfortunately, learning $V^*$ is a useful way to learn the optimal policy *only* when the agent has perfect knowledge of $\delta$ and $r$. This requires that it be able to perfectly predict the immediate result (i.e., the immediate reward and immediate successor) for every possible state-action transition. This assumption is comparable to the assumption of a perfect domain theory in explanation-based learning, discussed in Chapter 11. In many practical problems, such as robot control, it is impossible for the agent or its human programmer to predict in advance the exact outcome of applying an arbitrary action to an arbitrary state. Imagine, for example, the difficulty in describing $\delta$ for a robot arm shoveling dirt when the resulting state includes the positions of the dirt particles. In cases where either $\delta$ or $r$ is unknown, learning $V^*$ is unfortunately of no use for selecting optimal actions because the agent cannot evaluate Equation (13.3). What evaluation function should the agent use in this more general setting? The evaluation function $Q$, defined in the following section, provides one answer.

### 13.3.1 The $Q$ Function

Let us define the evaluation function $Q(s, a)$ so that its value is the maximum discounted cumulative reward that can be achieved starting from state $s$ and applying action $a$ as the first action. In other words, the value of $Q$ is the reward received immediately upon executing action $a$ from state $s$, plus the value (discounted by $\gamma$) of following the optimal policy thereafter.

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a)) \tag{13.4}$$

Note that $Q(s, a)$ is exactly the quantity that is maximized in Equation (13.3) in order to choose the optimal action $a$ in state $s$. Therefore, we can rewrite Equation (13.3) in terms of $Q(s, a)$ as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s, a) \tag{13.5}$$

Why is this rewrite important? Because it shows that if the agent learns the $Q$ function instead of the $V^*$ function, it will be able to select optimal actions *even when it has no knowledge of the functions r and δ*. As Equation (13.5) makes clear, it need only consider each available action $a$ in its current state $s$ and choose the action that maximizes $Q(s, a)$.

It may at first seem surprising that one can choose globally optimal action sequences by reacting repeatedly to the local values of $Q$ for the current state. This means the agent can choose the optimal action without ever conducting a lookahead search to explicitly consider what state results from the action. Part of the beauty of $Q$ learning is that the evaluation function is defined to have precisely this property—the value of $Q$ for the current state and action summarizes in a single number all the information needed to determine the discounted cumulative reward that will be gained in the future if action $a$ is selected in state $s$.

To illustrate, Figure 13.2 shows the $Q$ values for every state and action in the simple grid world. Notice that the $Q$ value for each state-action transition equals the $r$ value for this transition plus the $V^*$ value for the resulting state discounted by $\gamma$. Note also that the optimal policy shown in the figure corresponds to selecting actions with maximal $Q$ values.

### 13.3.2 An Algorithm for Learning $Q$

Learning the $Q$ function corresponds to learning the optimal policy. How can $Q$ be learned?

The key problem is finding a reliable way to estimate training values for $Q$, given only a sequence of immediate rewards $r$ spread out over time. This can be accomplished through iterative approximation. To see how, notice the close relationship between $Q$ and $V^*$,

$$V^*(s) = \max_{a'} Q(s, a')$$

which allows rewriting Equation (13.4) as

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') \tag{13.6}$$

This recursive definition of $Q$ provides the basis for algorithms that iteratively approximate $Q$ (Watkins 1989). To describe the algorithm, we will use the symbol $\hat{Q}$ to refer to the learner's estimate, or hypothesis, of the actual $Q$ function. In this algorithm the learner represents its hypothesis $\hat{Q}$ by a large table with a separate entry for each state-action pair. The table entry for the pair $\langle s, a \rangle$ stores the value for $\hat{Q}(s, a)$—the learner's current hypothesis about the actual but unknown value $Q(s, a)$. The table can be initially filled with random values (though it is easier to understand the algorithm if one assumes initial values of zero). The agent repeatedly observes its current state $s$, chooses some action $a$, executes this action, then observes the resulting reward $r = r(s, a)$ and the new state $s' = \delta(s, a)$. It then updates the table entry for $\hat{Q}(s, a)$ following each such transition, according to the rule:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a') \tag{13.7}$$

Note this training rule uses the agent's current $\hat{Q}$ values for the new state $s'$ to refine its estimate of $\hat{Q}(s, a)$ for the previous state $s$. This training rule is motivated by Equation (13.6), although the training rule concerns the agent's approximation $\hat{Q}$, whereas Equation (13.6) applies to the actual $Q$ function. Note although Equation (13.6) describes $Q$ in terms of the functions $\delta(s, a)$ and $r(s, a)$, the agent does not need to know these general functions to apply the training rule of Equation (13.7). Instead it executes the action in its environment and then observes the resulting new state $s'$ and reward $r$. Thus, it can be viewed as sampling these functions at the current values of $s$ and $a$.

The above $Q$ learning algorithm for deterministic Markov decision processes is described more precisely in Table 13.1. Using this algorithm the agent's estimate $\hat{Q}$ converges in the limit to the actual $Q$ function, provided the system can be modeled as a deterministic Markov decision process, the reward function $r$ is

---

$Q$ learning algorithm

For each $s, a$ initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state $s$
Do forever:

- Select an action $a$ and execute it
- Receive immediate reward $r$
- Observe the new state $s'$
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

---

**TABLE 13.1**
$Q$ learning algorithm, assuming deterministic rewards and actions. The discount factor $\gamma$ may be any constant such that $0 \leq \gamma < 1$.

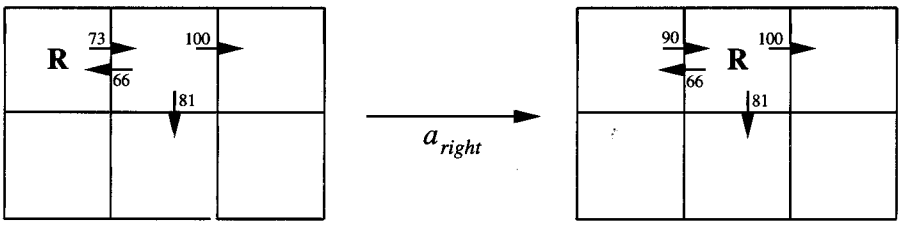
bounded, and actions are chosen so that every state-action pair is visited infinitely often.

### 13.3.3 An Illustrative Example

To illustrate the operation of the $Q$ learning algorithm, consider a single action taken by an agent, and the corresponding refinement to $\hat{Q}$ shown in Figure 13.3. In this example, the agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition. It then applies the training rule of Equation (13.7) to refine its estimate $\hat{Q}$ for the state-action transition it just executed. According to the training rule, the new $\hat{Q}$ estimate for this transition is the sum of the received reward (zero) and the highest $\hat{Q}$ value associated with the resulting state (100), discounted by $\gamma$ (.9).

Each time the agent moves forward from an old state to a new one, $Q$ learning propagates $\hat{Q}$ estimates *backward* from the new state to the old. At the same time, the immediate reward received by the agent for the transition is used to augment these propagated values of $\hat{Q}$.

Consider applying this algorithm to the grid world and reward function shown in Figure 13.2, for which the reward is zero everywhere, except when entering the goal state. Since this world contains an absorbing goal state, we will assume that training consists of a series of *episodes*. During each episode, the agent begins at some randomly chosen state and is allowed to execute actions until it reaches the absorbing goal state. When it does, the episode ends and




$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$
$$\leftarrow 0 + 0.9 \max\{66, 81, 100\}$$
$$\leftarrow 90$$

**FIGURE 13.3**
The update to $\hat{Q}$ after executing a single action. The diagram on the left shows the initial state $s_1$ of the robot (**R**) and several relevant $\hat{Q}$ values in its initial hypothesis. For example, the value $\hat{Q}(s_1, a_{right}) = 72.9$, where $a_{right}$ refers to the action that moves **R** to its right. When the robot executes the action $a_{right}$, it receives immediate reward $r = 0$ and transitions to state $s_2$. It then updates its estimate $\hat{Q}(s_1, a_{right})$ based on its $\hat{Q}$ estimates for the new state $s_2$. Here $\gamma = 0.9$.

the agent is transported to a new, randomly chosen, initial state for the next episode.

How will the values of $\hat{Q}$ evolve as the $Q$ learning algorithm is applied in this case? With all the $\hat{Q}$ values initialized to zero, the agent will make no changes to any $\hat{Q}$ table entry until it happens to reach the goal state and receive a nonzero reward. This will result in refining the $\hat{Q}$ value for the single transition leading into the goal state. On the next episode, if the agent passes through this state adjacent to the goal state, its nonzero $\hat{Q}$ value will allow refining the value for some transition two steps from the goal, and so on. Given a sufficient number of training episodes, the information will propagate from the transitions with nonzero reward back through the entire state-action space available to the agent, resulting eventually in a $\hat{Q}$ table containing the $Q$ values shown in Figure 13.2.

In the next section we prove that under certain assumptions the $Q$ learning algorithm of Table 13.1 will converge to the correct $Q$ function. First consider two general properties of this $Q$ learning algorithm that hold for any deterministic MDP in which the rewards are non-negative, assuming we initialize all $\hat{Q}$ values to zero. The first property is that under these conditions the $\hat{Q}$ values never decrease during training. More formally, let $\hat{Q}_n(s, a)$ denote the learned $\hat{Q}(s, a)$ value after the $n$th iteration of the training procedure (i.e., after the $n$th state-action transition taken by the agent). Then

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a)$$

A second general property that holds under these same conditions is that throughout the training process every $\hat{Q}$ value will remain in the interval between zero and its true $Q$ value.

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s, a) \leq Q(s, a)$$

### 13.3.4 Convergence

Will the algorithm of Table 13.1 converge toward a $\hat{Q}$ equal to the true $Q$ function? The answer is yes, under certain conditions. First, we must assume the system is a deterministic MDP. Second, we must assume the immediate reward values are bounded; that is, there exists some positive constant $c$ such that for all states $s$ and actions $a$, $|r(s, a)| < c$. Third, we assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often. By this third condition we mean that if action $a$ is a legal action from state $s$, then over time the agent must execute action $a$ from state $s$ repeatedly and with nonzero frequency as the length of its action sequence approaches infinity. Note these conditions are in some ways quite general and in others fairly restrictive. They describe a more general setting than illustrated by the example in the previous section, because they allow for environments with arbitrary positive or negative rewards, and for environments where any number of state-action transitions may produce nonzero rewards. The conditions are also restrictive in that they require the agent to visit every distinct state-action transition infinitely often. This is a very strong assumption in large (or continuous!) domains. We will discuss stronger

convergence results later. However, the result described in this section provides the basic intuition for understanding why $Q$ learning works.

The key idea underlying the proof of convergence is that the table entry $\hat{Q}(s, a)$ with the largest error must have its error reduced by a factor of $\gamma$ whenever it is updated. The reason is that its new value depends only in part on error-prone $\hat{Q}$ estimates, with the remainder depending on the error-free observed immediate reward $r$.

**Theorem 13.1. Convergence of $Q$ learning for deterministic Markov decision processes.** Consider a $Q$ learning agent in a deterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$. The $Q$ learning agent uses the training rule of Equation (13.7), initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor $\gamma$ such that $0 \leq \gamma < 1$. Let $\hat{Q}_n(s, a)$ denote the agent's hypothesis $\hat{Q}(s, a)$ following the $n$th update. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ as $n \to \infty$, for all $s, a$.

***Proof.*** Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists of showing that the maximum error over all entries in the $\hat{Q}$ table is reduced by at least a factor of $\gamma$ during each such interval. $\hat{Q}_n$ is the agent's table of estimated $Q$ values after $n$ updates. Let $\Delta_n$ be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n \equiv \max_{s,a} |\hat{Q}_n(s, a) - Q(s, a)|$$

Below we use $s'$ to denote $\delta(s, a)$. Now for any table entry $\hat{Q}_n(s, a)$ that is updated on iteration $n + 1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is

$$\begin{aligned}
|\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\
&= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\
&\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\
&\leq \gamma \max_{s'', a'} |\hat{Q}_n(s'', a') - Q(s'', a')| \\
|\hat{Q}_{n+1}(s, a) - Q(s, a)| &\leq \gamma \Delta_n
\end{aligned}$$

The third line above follows from the second line because for any two functions $f_1$ and $f_2$ the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|$$

In going from the third line to the fourth line above, note we introduce a new variable $s''$ over which the maximization is performed. This is legitimate because the maximum value will be at least as great when we allow this additional variable to vary. Note that by introducing this variable we obtain an expression that matches the definition of $\Delta_n$.

Thus, the updated $\hat{Q}_{n+1}(s, a)$ for any $s, a$ is at most $\gamma$ times the maximum error in the $\hat{Q}_n$ table, $\Delta_n$. The largest error in the initial table, $\Delta_0$, is bounded because values of $\hat{Q}_0(s, a)$ and $Q(s, a)$ are bounded for all $s, a$. Now after the first interval

during which each $s, a$ is visited, the largest error in the table will be at most $\gamma\Delta_0$. After $k$ such intervals, the error will be at most $\gamma^k\Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite, and $\Delta_n \to 0$ as $n \to \infty$. This proves the theorem. ■

### 13.3.5 Experimentation Strategies

Notice the algorithm of Table 13.1 does not specify how actions are chosen by the agent. One obvious strategy would be for the agent in state $s$ to select the action $a$ that maximizes $\hat{Q}(s, a)$, thereby exploiting its current approximation $\hat{Q}$. However, with this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high $\hat{Q}$ values, while failing to explore other actions that have even higher values. In fact, the convergence theorem above requires that each state-action transition occur infinitely often. This will clearly not occur if the agent always selects actions that maximize its current $\hat{Q}(s, a)$. For this reason, it is common in $Q$ learning to use a probabilistic approach to selecting actions. Actions with higher $\hat{Q}$ values are assigned higher probabilities, but every action is assigned a nonzero probability. One way to assign such probabilities is

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

where $P(a_i|s)$ is the probability of selecting action $a_i$, given that the agent is in state $s$, and where $k > 0$ is a constant that determines how strongly the selection favors actions with high $\hat{Q}$ values. Larger values of $k$ will assign higher probabilities to actions with above average $\hat{Q}$, causing the agent to *exploit* what it has learned and seek actions it believes will maximize its reward. In contrast, small values of $k$ will allow higher probabilities for other actions, leading the agent to *explore* actions that do not currently have high $\hat{Q}$ values. In some cases, $k$ is varied with the number of iterations so that the agent favors exploration during early stages of learning, then gradually shifts toward a strategy of exploitation.

### 13.3.6 Updating Sequence

One important implication of the above convergence theorem is that $Q$ learning need not train on optimal action sequences in order to converge to the optimal policy. In fact, it can learn the $Q$ function (and hence the optimal policy) while training from actions chosen completely at random at each step, as long as the resulting training sequence visits every state-action transition infinitely often. This fact suggests changing the sequence of training example transitions in order to improve training efficiency without endangering final convergence. To illustrate, consider again learning in an MDP with a single absorbing goal state, such as the one in Figure 13.1. Assume as before that we train the agent with a sequence of episodes. For each episode, the agent is placed in a random initial state and is allowed to perform actions and to update its $\hat{Q}$ table until it reaches the absorbing goal state. A new training episode is then begun by removing the agent from the

goal state and placing it at a new random initial state. As noted earlier, if we begin with all $\hat{Q}$ values initialized to zero, then after the first full episode only one entry in the agent's $\hat{Q}$ table will have been changed: the entry corresponding to the final transition into the goal state. Note that if the agent happens to follow the same sequence of actions from the same random initial state in its second full episode, then a second table entry would be made nonzero, and so on. If we run repeated identical episodes in this fashion, the frontier of nonzero $\hat{Q}$ values will creep backward from the goal state at the rate of one new state-action transition per episode. Now consider training on these same state-action transitions, but in reverse chronological order for each episode. That is, we apply the same update rule from Equation (13.7) for each transition considered, but perform these updates in reverse order. In this case, after the first full episode the agent will have updated its $\hat{Q}$ estimate for every transition along the path it took to the goal. This training process will clearly converge in fewer iterations, although it requires that the agent use more memory to store the entire episode before beginning the training for that episode.

A second strategy for improving the rate of convergence is to store past state-action transitions, along with the immediate reward that was received, and retrain on them periodically. Although at first it might seem a waste of effort to retrain on the same transition, recall that the updated $\hat{Q}(s, a)$ value is determined by the values $\hat{Q}(s', a)$ of the successor state $s' = \delta(s, a)$. Therefore, if subsequent training changes one of the $\hat{Q}(s', a)$ values, then retraining on the transition $\langle s, a \rangle$ may result in an altered value for $\hat{Q}(s, a)$. In general, the degree to which we wish to replay old transitions versus obtain new ones from the environment depends on the relative costs of these two operations in the specific problem domain. For example, in a robot domain with navigation actions that might take several seconds to perform, the delay in collecting a new state-action transition from the external world might be several orders of magnitude more costly than internally replaying a previously observed transition. This difference can be very significant given that $Q$ learning can often require thousands of training iterations to converge.

Note throughout the above discussion we have kept our assumption that the agent does not know the state-transition function $\delta(s, a)$ used by the environment to create the successor state $s' = \delta(s, a)$, or the function $r(s, a)$ used to generate rewards. If it does know these two functions, then many more efficient methods are possible. For example, if performing external actions is expensive the agent may simply ignore the environment and instead simulate it internally, efficiently generating simulated actions and assigning the appropriate simulated rewards. Sutton (1991) describes the DYNA architecture that performs a number of simulated actions after each step executed in the external world. Moore and Atkeson (1993) describe an approach called *prioritized sweeping* that selects promising states to update next, focusing on predecessor states when the current state is found to have a large update. Peng and Williams (1994) describe a similar approach. A large number of efficient algorithms from the field of dynamic programming can be applied when the functions $\delta$ and $r$ are known. Kaelbling et al. (1996) survey a number of these.

## 13.4 NONDETERMINISTIC REWARDS AND ACTIONS

Above we considered $Q$ learning in deterministic environments. Here we consider the nondeterministic case, in which the reward function $r(s, a)$ and action transition function $\delta(s, a)$ may have probabilistic outcomes. For example, in Tesauro's (1995) backgammon playing program, action outcomes are inherently probabilistic because each move involves a roll of the dice. Similarly, in robot problems with noisy sensors and effectors it is often appropriate to model actions and rewards as nondeterministic. In such cases, the functions $\delta(s, a)$ and $r(s, a)$ can be viewed as first producing a probability distribution over outcomes based on $s$ and $a$, and then drawing an outcome at random according to this distribution. When these probability distributions depend solely on $s$ and $a$ (e.g., they do not depend on previous states or actions), then we call the system a nondeterministic Markov decision process.

In this section we extend the $Q$ learning algorithm for the deterministic case to handle nondeterministic MDPs. To accomplish this, we retrace the line of argument that led to the algorithm for the deterministic case, revising it where needed.

In the nondeterministic case we must first restate the objective of the learner to take into account the fact that outcomes of actions are no longer deterministic. The obvious generalization is to redefine the value $V^{\pi}$ of a policy $\pi$ to be the *expected value* (over these nondeterministic outcomes) of the discounted cumulative reward received by applying this policy

$$V^{\pi}(s_t) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

where, as before, the sequence of rewards $r_{t+i}$ is generated by following policy $\pi$ beginning at state $s$. Note this is a generalization of Equation (13.1), which covered the deterministic case.

As before, we define the optimal policy $\pi^*$ to be the policy $\pi$ that maximizes $V^{\pi}(s)$ for all states $s$. Next we generalize our earlier definition of $Q$ from Equation (13.4), again by taking its expected value.

$$\begin{aligned} Q(s, a) &\equiv E[r(s, a) + \gamma V^*(\delta(s, a))] \\ &= E[r(s, a)] + \gamma E[V^*(\delta(s, a))] \\ &= E[r(s, a)] + \gamma \sum_{s'} P(s'|s, a) V^*(s') \end{aligned} \tag{13.8}$$

where $P(s'|s, a)$ is the probability that taking action $a$ in state $s$ will produce the next state $s'$. Note we have used $P(s'|s, a)$ here to rewrite the expected value of $V^*(\delta(s, a))$ in terms of the probabilities associated with the possible outcomes of the probabilistic $\delta$.

As before we can re-express $Q$ recursively

$$Q(s, a) = E[r(s, a)] + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a') \tag{13.9}$$

which is the generalization of the earlier Equation (13.6). To summarize, we have simply redefined $Q(s, a)$ in the nondeterministic case to be the expected value of its previously defined quantity for the deterministic case.

Now that we have generalized the definition of $Q$ to accommodate the nondeterministic environment functions $r$ and $\delta$, a new training rule is needed. Our earlier training rule derived for the deterministic case (Equation 13.7) fails to converge in this nondeterministic setting. Consider, for example, a nondeterministic reward function $r(s, a)$ that produces different rewards each time the transition $\langle s, a\rangle$ is repeated. In this case, the training rule will repeatedly alter the values of $\hat{Q}(s, a)$, even if we initialize the $\hat{Q}$ table values to the correct $Q$ function. In brief, this training rule does not converge. This difficulty can be overcome by modifying the training rule so that it takes a decaying weighted average of the current $\hat{Q}$ value and the revised estimate. Writing $\hat{Q}_n$ to denote the agent's estimate on the $n$th iteration of the algorithm, the following revised training rule is sufficient to assure convergence of $\hat{Q}$ to $Q$:

$$\hat{Q}_n(s, a) \leftarrow (1 - \alpha_n)\hat{Q}_{n-1}(s, a) + \alpha_n[r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a')] \tag{13.10}$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s, a)} \tag{13.11}$$

where $s$ and $a$ here are the state and action updated during the $n$th iteration, and where $visits_n(s, a)$ is the total number of times this state-action pair has been visited up to and including the $n$th iteration.

The key idea in this revised rule is that revisions to $\hat{Q}$ are made more gradually than in the deterministic case. Notice if we were to set $\alpha_n$ to 1 in Equation (13.10) we would have exactly the training rule for the deterministic case. With smaller values of $\alpha$, this term is now averaged in with the current $\hat{Q}(s, a)$ to produce the new updated value. Notice that the value of $\alpha_n$ in Equation (13.11) decreases as $n$ increases, so that updates become smaller as training progresses. By reducing $\alpha$ at an appropriate rate during training, we can achieve convergence to the correct $Q$ function. The choice of $\alpha_n$ given above is one of many that satisfy the conditions for convergence, according to the following theorem due to Watkins and Dayan (1992).

**Theorem 13.2. Convergence of $Q$ learning for nondeterministic Markov decision processes.** Consider a $Q$ learning agent in a nondeterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$. The $Q$ learning agent uses the training rule of Equation (13.10), initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor $\gamma$ such that $0 \leq \gamma < 1$. Let $n(i, s, a)$ be the iteration corresponding to the $i$th time that action $a$ is applied to state $s$. If each state-action pair is visited infinitely often, $0 \leq \alpha_n < 1$, and

$$\sum_{i=1}^{\infty} \alpha_{n(i,s,a)} = \infty, \quad \sum_{i=1}^{\infty} [\alpha_{n(i,s,a)}]^2 < \infty$$

then for all $s$ and $a$, $\hat{Q}_n(s, a) \rightarrow Q(s, a)$ as $n \rightarrow \infty$, with probability 1.

While $Q$ learning and related reinforcement learning algorithms can be proven to converge under certain conditions, in practice systems that use $Q$ learning often require many thousands of training iterations to converge. For example, Tesauro's TD-GAMMON discussed earlier trained for 1.5 million backgammon games, each of which contained tens of state-action transitions.

## 13.5 TEMPORAL DIFFERENCE LEARNING

The $Q$ learning algorithm learns by iteratively reducing the discrepancy between $Q$ value estimates for adjacent states. In this sense, $Q$ learning is a special case of a general class of *temporal difference* algorithms that learn by reducing discrepancies between estimates made by the agent at different times. Whereas the training rule of Equation (13.10) reduces the difference between the estimated $\hat{Q}$ values of a state and its immediate successor, we could just as well design an algorithm that reduces discrepancies between this state and more distant descendants or ancestors.

To explore this issue further, recall that our $Q$ learning training rule calculates a training value for $\hat{Q}(s_t, a_t)$ in terms of the values for $\hat{Q}(s_{t+1}, a_{t+1})$ where $s_{t+1}$ is the result of applying action $a_t$ to the state $s_t$. Let $Q^{(1)}(s_t, a_t)$ denote the training value calculated by this one-step lookahead

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

One alternative way to compute a training value for $Q(s_t, a_t)$ is to base it on the observed rewards for two steps

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

or, in general, for $n$ steps

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

Sutton (1988) introduces a general method for blending these alternative training estimates, called TD($\lambda$). The idea is to use a constant $0 \leq \lambda \leq 1$ to combine the estimates obtained from various lookahead distances in the following fashion

$$Q^{\lambda}(s_t, a_t) \equiv (1 - \lambda) \left[ Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \cdots \right]$$

An equivalent recursive definition for $Q^{\lambda}$ is

$$Q^{\lambda}(s_t, a_t) = r_t + \gamma [ (1 - \lambda) \max_a \hat{Q}(s_t, a_t) + \lambda Q^{\lambda}(s_{t+1}, a_{t+1})]$$

Note if we choose $\lambda = 0$ we have our original training estimate $Q^{(1)}$, which considers only one-step discrepancies in the $\hat{Q}$ estimates. As $\lambda$ is increased, the algorithm places increasing emphasis on discrepancies based on more distant lookaheads. At the extreme value $\lambda = 1$, only the observed $r_{t+i}$ values are considered,

with no contribution from the current $\hat{Q}$ estimate. Note when $\hat{Q} = Q$, the training values given by $Q^\lambda$ will be identical for all values of $\lambda$ such that $0 \leq \lambda \leq 1$.

The motivation for the TD($\lambda$) method is that in some settings training will be more efficient if more distant lookaheads are considered. For example, when the agent follows an optimal policy for choosing actions, then $Q^\lambda$ with $\lambda = 1$ will provide a perfect estimate for the true $Q$ value, regardless of any inaccuracies in $\hat{Q}$. On the other hand, if action sequences are chosen suboptimally, then the $r_{t+i}$ observed far into the future can be misleading.

Peng and Williams (1994) provide a further discussion and experimental results showing the superior performance of $Q^\lambda$ in one problem domain. Dayan (1992) shows that under certain assumptions a similar TD($\lambda$) approach applied to learning the $V^*$ function converges correctly for any $\lambda$ such that $0 \leq \lambda \leq 1$. Tesauro (1995) uses a TD($\lambda$) approach in his TD-GAMMON program for playing backgammon.

## 13.6 GENERALIZING FROM EXAMPLES

Perhaps the most constraining assumption in our treatment of $Q$ learning up to this point is that the target function is represented as an explicit lookup table, with a distinct table entry for every distinct input value (i.e., state-action pair). Thus, the algorithms we discussed perform a kind of rote learning and make no attempt to estimate the $Q$ value for unseen state-action pairs by generalizing from those that have been seen. This rote learning assumption is reflected in the convergence proof, which proves convergence only if every possible state-action pair is visited (infinitely often!). This is clearly an unrealistic assumption in large or infinite spaces, or when the cost of executing actions is high. As a result, more practical systems often combine function approximation methods discussed in other chapters with the $Q$ learning training rules described here.

It is easy to incorporate function approximation algorithms such as BACKPROPAGATION into the $Q$ learning algorithm, by substituting a neural network for the lookup table and using each $\hat{Q}(s, a)$ update as a training example. For example, we could encode the state $s$ and action $a$ as network inputs and train the network to output the target values of $\hat{Q}$ given by the training rules of Equations (13.7) and (13.10). An alternative that has sometimes been found to be more successful in practice is to train a separate network for each action, using the state as input and $\hat{Q}$ as output. Another common alternative is to train one network with the state as input, but with one $\hat{Q}$ output for each action. Recall that in Chapter 1, we discussed approximating an evaluation function over checkerboard states using a linear function and the LMS algorithm.

In practice, a number of successful reinforcement learning systems have been developed by incorporating such function approximation algorithms in place of the lookup table. Tesauro's successful TD-GAMMON program for playing backgammon used a neural network and the BACKPROPAGATION algorithm together with a TD($\lambda$) training rule. Zhang and Dietterich (1996) use a similar combination of BACKPROPAGATION and TD($\lambda$) for job-shop scheduling tasks. Crites and Barto (1996) describe

a neural network reinforcement learning approach for an elevator scheduling task. Thrun (1996) reports a neural network based approach to $Q$ learning to learn basic control procedures for a mobile robot with sonar and camera sensors. Mahadevan and Connell (1991) describe a $Q$ learning approach based on clustering states, applied to a simple mobile robot control problem.

Despite the success of these systems, for other tasks reinforcement learning fails to converge once a generalizing function approximator is introduced. Examples of such problematic tasks are given by Boyan and Moore (1995), Baird (1995), and Gordon (1995). Note the convergence theorems discussed earlier in this chapter apply only when $\hat{Q}$ is represented by an explicit table. To see the difficulty, consider using a neural network rather than an explicit table to represent $\hat{Q}$. Note if the learner updates the network to better fit the training $Q$ value for a particular transition $\langle s_i, a_i \rangle$, the altered network weights may also change the $\hat{Q}$ estimates for arbitrary other transitions. Because these weight changes may increase the error in $\hat{Q}$ estimates for these other transitions, the argument proving the original theorem no longer holds. Theoretical analyses of reinforcement learning with generalizing function approximators are given by Gordon (1995) and Tsitsiklis (1994). Baird (1995) proposes gradient-based methods that circumvent this difficulty by directly minimizing the sum of squared discrepancies in estimates between adjacent states (also called Bellman residual errors).

## 13.7 RELATIONSHIP TO DYNAMIC PROGRAMMING

Reinforcement learning methods such as $Q$ learning are closely related to a long line of research on dynamic programming approaches to solving Markov decision processes. This earlier work has typically assumed that the agent possesses perfect knowledge of the functions $\delta(s, a)$ and $r(s, a)$ that define the agent's environment. Therefore, it has primarily addressed the question of how to compute the optimal policy using the least computational effort, assuming the environment could be perfectly simulated and no direct interaction was required. The novel aspect of $Q$ learning is that it assumes the agent does *not* have knowledge of $\delta(s, a)$ and $r(s, a)$, and that instead of moving about in an internal mental model of the state space, it must move about the real world and observe the consequences. In this latter case our primary concern is usually the number of real-world actions that the agent must perform to converge to an acceptable policy, rather than the number of computational cycles it must expend. The reason is that in many practical domains such as manufacturing problems, the costs in time and in dollars of performing actions in the external world dominate the computational costs. Systems that learn by moving about the real environment and observing the results are typically called *online* systems, whereas those that learn solely by simulating actions within an internal model are called *offline* systems.

The close correspondence between these earlier approaches and the reinforcement learning problems discussed here is apparent by considering Bellman's equation, which forms the foundation for many dynamic programming approaches

to solving MDPs. Bellman's equation is

$$(\forall s \in S) V^*(s) = E[r(s, \pi(s)) + \gamma V^*(\delta(s, \pi(s)))]$$

Note the very close relationship between Bellman's equation and our earlier definition of an optimal policy in Equation (13.2). Bellman (1957) showed that the optimal policy $\pi^*$ satisfies the above equation and that any policy $\pi$ satisfying this equation is an optimal policy. Early work on dynamic programming includes the Bellman-Ford shortest path algorithm (Bellman 1958; Ford and Fulkerson 1962), which learns paths through a graph by repeatedly updating the estimated distance to the goal for each graph node, based on the distances for its neighbors. In this algorithm the assumption that graph edges and the goal node are known is equivalent to our assumption that $\delta(s, a)$ and $r(s, a)$ are known. Barto et al. (1995) discuss the close relationship between reinforcement learning and dynamic programming.

## 13.8 SUMMARY AND FURTHER READING

The key points discussed in this chapter include:

- Reinforcement learning addresses the problem of learning control strategies for autonomous agents. It assumes that training information is available in the form of a real-valued reward signal given for each state-action transition. The goal of the agent is to learn an action policy that maximizes the total reward it will receive from any starting state.
- The reinforcement learning algorithms addressed in this chapter fit a problem setting known as a Markov decision process. In Markov decision processes, the outcome of applying any action to any state depends only on this action and state (and not on preceding actions or states). Markov decision processes cover a wide range of problems including many robot control, factory automation, and scheduling problems.
- $Q$ learning is one form of reinforcement learning in which the agent learns an evaluation function over states and actions. In particular, the evaluation function $Q(s, a)$ is defined as the maximum expected, discounted, cumulative reward the agent can achieve by applying action $a$ to state $s$. The $Q$ learning algorithm has the advantage that it can be employed even when the learner has no prior knowledge of how its actions affect its environment.
- $Q$ learning can be proven to converge to the correct $Q$ function under certain assumptions, when the learner's hypothesis $\hat{Q}(s, a)$ is represented by a lookup table with a distinct entry for each $\langle s, a \rangle$ pair. It can be shown to converge in both deterministic and nondeterministic MDPs. In practice, $Q$ learning can require many thousands of training iterations to converge in even modest-sized problems.
- $Q$ learning is a member of a more general class of algorithms, called temporal difference algorithms. In general, temporal difference algorithms learn

by iteratively reducing the discrepancies between the estimates produced by the agent at different times.

- Reinforcement learning is closely related to dynamic programming approaches to Markov decision processes. The key difference is that historically these dynamic programming approaches have assumed that the agent possesses knowledge of the state transition function $\delta(s, a)$ and reward function $r(s, a)$. In contrast, reinforcement learning algorithms such as $Q$ learning typically assume the learner lacks such knowledge.

The common theme that underlies much of the work on reinforcement learning is to iteratively reduce the discrepancy between evaluations of successive states. Some of the earliest work on such methods is due to Samuel (1959). His checkers learning program attempted to learn an evaluation function for checkers by using evaluations of later states to generate training values for earlier states. Around the same time, the Bellman-Ford, single-destination, shortest-path algorithm was developed (Bellman 1958; Ford and Fulkerson 1962), which propagated distance-to-goal values from nodes to their neighbors. Research on optimal control led to the solution of Markov decision processes using similar methods (Bellman 1961; Blackwell 1965). Holland's (1986) bucket brigade method for learning classifier systems used a similar method for propagating credit in the face of delayed rewards. Barto et al. (1983) discussed an approach to temporal credit assignment that led to Sutton's paper (1988) defining the TD($\lambda$) method and proving its convergence for $\lambda = 0$. Dayan (1992) extended this result to arbitrary values of $\lambda$. Watkins (1989) introduced $Q$ learning to acquire optimal policies when the reward and action transition functions are unknown. Convergence proofs are known for several variations on these methods. In addition to the convergence proofs presented in this chapter see, for example, (Baird 1995; Bertsekas 1987; Tsitsiklis 1994, Singh and Sutton 1996).

Reinforcement learning remains an active research area. McCallum (1995) and Littman (1996), for example, discuss the extension of reinforcement learning to settings with hidden state variables that violate the Markov assumption. Much current research seeks to scale up these methods to larger, more practical problems. For example, Maclin and Shavlik (1996) describe an approach in which a reinforcement learning agent can accept imperfect advice from a trainer, based on an extension to the KBANN algorithm (Chapter 12). Lin (1992) examines the role of teaching by providing suggested action sequences. Methods for scaling up by employing a hierarchy of actions are suggested by Singh (1993) and Lin (1993). Dietterich and Flann (1995) explore the integration of explanation-based methods with reinforcement learning, and Mitchell and Thrun (1993) describe the application of the EBNN algorithm (Chapter 12) to $Q$ learning. Ring (1994) explores continual learning by the agent over multiple tasks.

Recent surveys of reinforcement learning are given by Kaelbling et al. (1996); Barto (1992); Barto et al. (1995); Dean et al. (1993).

## EXERCISES

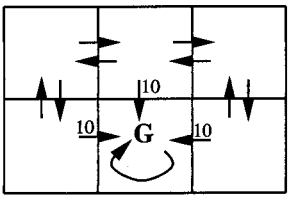
**13.1.** Give a second optimal policy for the problem illustrated in Figure 13.2.

**13.2.** Consider the deterministic grid world shown below with the absorbing goal-state G. Here the immediate rewards are 10 for the labeled transitions and 0 for all unlabeled transitions.

($a$) Give the $V^*$ value for every state in this grid world. Give the $Q(s, a)$ value for every transition. Finally, show an optimal policy. Use $\gamma = 0.8$.

($b$) Suggest a change to the reward function $r(s, a)$ that alters the $Q(s, a)$ values, but does not alter the optimal policy. Suggest a change to $r(s, a)$ that alters $Q(s, a)$ but does not alter $V^*(s, a)$.

($c$) Now consider applying the $Q$ learning algorithm to this grid world, assuming the table of $\hat{Q}$ values is initialized to zero. Assume the agent begins in the bottom left grid square and then travels clockwise around the perimeter of the grid until it reaches the absorbing goal state, completing the first training episode. Describe which $\hat{Q}$ values are modified as a result of this episode, and give their revised values. Answer the question again assuming the agent now performs a second identical episode. Answer it again for a third episode.




**13.3.** Consider playing Tic-Tac-Toe against an opponent who plays randomly. In particular, assume the opponent chooses with uniform probability any open space, unless there is a forced move (in which case it makes the obvious correct move).

($a$) Formulate the problem of learning an optimal Tic-Tac-Toe strategy in this case as a $Q$-learning task. What are the states, transitions, and rewards in this nondeterministic Markov decision process?

($b$) Will your program succeed if the opponent plays optimally rather than randomly?

**13.4.** Note in many MDPs it is possible to find two policies $\pi_1$ and $\pi_2$ such that $\pi_1$ outperforms $\pi_2$ if the agent begins in some state $s_1$, but $\pi_2$ outperforms $\pi_1$ if it begins in some other state $s_2$. Put another way, $V^{\pi_1}(s_1) > V^{\pi_2}(s_1)$, but $V^{\pi_2}(s_2) > V^{\pi_1}(s_2)$. Explain why there will always exist a single policy that maximizes $V^{\pi}(s)$ for *every* initial state $s$ (i.e., an optimal policy $\pi^*$). In other words, explain why an MDP always allows a policy $\pi^*$ such that $(\forall \pi, s)\ V^{\pi^*}(s) \geq V^{\pi}(s)$.

## REFERENCES

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 30–37). San Francisco: Morgan Kaufmann.

Barto, A. (1992). Reinforcement learning and adaptive critic methods. In D. White & S. Sofge (Eds.), *Handbook of intelligent control: Neural, fuzzy, and adaptive approaches* (pp. 469–491). New York: Van Nostrand Reinhold.

Barto, A., Bradtke, S., & Singh, S. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, Special volume: Computational research on interaction and agency, 72(1), 81–138.

Barto, A., Sutton, R., & Anderson, C. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5), 834–846.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, 16(1), 87–90.

Bellman, R. (1961). *Adaptive control processes*. Princeton, NJ: Princeton University Press.

Berenji, R. (1992). Learning and tuning fuzzy controllers through reinforcements. *IEEE Transactions on Neural Networks*, 3(5), 724–740.

Bertsekas, D. (1987). *Dynamic programming: Deterministic and stochastic models*. Englewood Cliffs, NJ: Prentice Hall.

Blackwell, D. (1965). Discounted dynamic programming. *Annals of Mathematical Statistics*, 36, 226–235.

Boyan, J., & Moore, A. (1995). Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. Touretzky, & T. Leen (Eds.), *Advances in Neural Information Processing Systems 7*. Cambridge, MA: MIT Press.

Crites, R., & Barto, A. (1996). Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, & M. C. Hasselmo (Eds.), *Advances in Neural Information Processing Systems*, 8.

Dayan, P. (1992). The convergence of TD($\lambda$) for general $\lambda$. *Machine Learning*, 8, 341–362.

Dean, T., Basye, K., & Shewchuk, J. (1993). Reinforcement learning for planning and control. In S. Minton (Ed.), *Machine Learning Methods for Planning* (pp. 67–92). San Francisco: Morgan Kaufmann.

Dietterich, T. G., & Flann, N. S. (1995). Explanation-based learning and reinforcement learning: A unified view. *Proceedings of the 12th International Conference on Machine Learning* (pp. 176–184). San Francisco: Morgan Kaufmann.

Ford, L., & Fulkerson, D. (1962). *Flows in networks*. Princeton, NJ: Princeton University Press.

Gordon, G. (1995). Stable function approximation in dynamic programming. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 261–268). San Francisco: Morgan Kaufmann.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of AI Research*, 4, 237–285. Online journal at http://www.cs.washington.edu/research/jair/-home.html.

Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, Carbonell, & Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2, pp. 593–623). San Francisco: Morgan Kaufmann.

Laird, J. E., & Rosenbloom, P. S. (1990). Integrating execution, planning, and learning in SOAR for external environments. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 1022–1029). Menlo Park, CA: AAAI Press.

Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, 8, 293–321.

Lin, L. J. (1993). Hierarchical learning of robot skills by reinforcement. *Proceedings of the International Conference on Neural Networks*.

Littman, M. (1996). *Algorithms for sequential decision making* (Ph.D. dissertation and Technical Report CS-96-09). Brown University, Department of Computer Science, Providence, RI.

Maclin, R., & Shavlik, J. W. (1996). Creating advice-taking reinforcement learners. *Machine Learning*, 22, 251–281.

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1), 159–195.

Mahadevan, S., & Connell, J. (1991). Automatic programming of behavior-based robots using reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*. San Francisco: Morgan Kaufmann.

McCallum, A. (1995). *Reinforcement learning with selective perception and hidden state* (Ph.D. dissertation). Department of Computer Science, University of Rochester, Rochester, NY.

Mitchell, T. M., & Thrun, S. B. (1993). Explanation-based neural network learning for robot control. In C. Giles, S. Hanson, & J. Cowan (Eds.), *Advances in Neural Information Processing Systems 5* (pp. 287–294). San Francisco: Morgan-Kaufmann.

Moore, A., & Atkeson C. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13, 103.

Peng, J., & Williams, R. (1994). Incremental multi-step $Q$-learning. *Proceedings of the Eleventh International Conference on Machine Learning* (pp. 226–232). San Francisco: Morgan Kaufmann.

Ring, M. (1994). *Continual learning in reinforcement environments* (Ph.D. dissertation). Computer Science Department, University of Texas at Austin, Austin, TX.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3, 211–229.

Singh, S. (1992). Reinforcement learning with a hierarchy of abstract models. *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 202–207). San Jose, CA: AAAI Press.

Singh, S. (1993). *Learning to solve markovian decision processes* (Ph.D. dissertation). Also CMPSCI Technical Report 93-77, Department of Computer Science, University of Massachusetts at Amherst.

Singh, S., & Sutton, R. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22, 123.

Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3, 9–44.

Sutton R. (1991). Planning by incremental dynamic programming. *Proceedings of the Eighth International Conference on Machine Learning* (pp. 353–357). San Francisco: Morgan Kaufmann.

Tesauro, G. (1995). Temporal difference learning and TD-GAMMON. *Communications of the ACM*, 38(3), 58–68.

Thrun, S. (1992). The role of exploration in learning control. In D. White & D. Sofge (Eds.), *Handbook of intelligent control: Neural, fuzzy, and adaptive approaches* (pp. 527–559). New York: Van Nostrand Reinhold.

Thrun, S. (1996). Explanation-based neural network learning: A lifelong learning approach. Boston: Kluwer Academic Publishers.

Tsitsiklis, J. (1994). Asynchronous stochastic approximation and $Q$-learning. *Machine Learning*, 16(3), 185–202.

Watkins, C. (1989). *Learning from delayed rewards* (Ph.D. dissertation). King's College, Cambridge, England.

Watkins, C., & Dayan, P. (1992). $Q$-learning. *Machine Learning*, 8, 279–292.

Zhang, W., & Dietterich, T. G. (1996). High-performance job-shop scheduling with a time-delay TD($\lambda$) network. In D. S. Touretzky, M. C. Mozer, & M. E. Hasselmo (Eds.), *Advances in neural information processing systems*, 8, 1024–1030.