EVALUATING HYPOTHESIS

Chapter 5

Empirically evaluating the accuracy of hypotheses is fundamental to machine learning. Statistical methods for estimating hypothesis accuracy, focuses 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?

MOTIVATION

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. 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.

ESTIMATING HYPOTHESIS ACCURACY

Consider the following setting for the learning problem. There is some space of possible instances X, over which various target functions may be defined. 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 ***D*** that defines the probability of encountering each instance in X.

Sample Error and True Error

Sample Error and True Error are the two notions of accuracy or, equivalently, error. One is the error rate of the hypothesis over the sample of data that is available.

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 δ(f ***(x), h(x))*** is ***1*** if f ***(x)*** ≠ ***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 ***D.***

Definition: The **true error** (denoted **$error_D(h)$**) of hypothesis **h** with respect to target function f and distribution **D,** is the probability that **h** will misclassify **an** instance drawn at random according to **D.**

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

What we usually wish to know is the true error ***$error_D(h)$*** 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. How good an estimate of ***errorD(h)*** is provided by ***errors (h)?***

Confidence Intervals for Discrete-Valued Hypotheses

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 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 ***$error_D(h)$*** is ***$error_s(h)$***

2. With approximately ***95%*** probability, the true error ***$error_D(h)$*** 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 $error_D(h)$ is the observed

sample error .30. For the given confidence interval of 95% the true error will be ***0.30*** ± ***(1.96 - .07)*** = ***0.30*** ± .14.

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 $error_v(h)$ is

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

Where the constant Z_N is chosen depending on the desired confidence level, using the values of Z_N given in Table *5.1*.

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.

INSTANCE BASED LEARNING

Chapter 8

Instance-based learning methods such as nearest neighbour and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions.

k - NEAREST NEIGHBOUR LEARNING

The most basic instance-based method is the k-NEARESNT NEIGHBOUR algorithm. The nearest neighbours 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-neighbour learning the target function may be either discrete-valued or real-valued. Let us first consider learning discrete-valued target functions. If we choose k = 1, then the 1-NEAREST NEIGHBOUR 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. The k-NEARESNT NEIGHBOR algorithm for approximating a discrete-valued target function is given in Table 8.1.

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 $f : \Re^n \rightarrow V$.

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 we replace the final line of the above algorithm by the line

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

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

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

where

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

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

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

where w_i is as defined in Equation (8.3).

Note all of the above variants of the k-NEAREST NEIGHBOR algorithm consider only the k nearest neighbours to classify the query point.

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

Remarks on k-NEAREST NEIGHBOR algorithm

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

An even more drastic alternative is to completely eliminate the least relevant attributes from the instance space.

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 neighbours can

be identified more efficiently at some additional cost in memory. One such indexing method is the kd-tree.

Note: For example problem on k nearest neighbour algorithm refer class notes.

A Note on Terminology

Much of the literature on nearest-neighbour 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 $w_i = K(d(x_i, x_q))$.

LOCALLY WEIGHTED REGRESSION

- The nearest-neighbour approaches 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.
- 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.

Locally Weighted Linear Regression

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

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

Procedure to derive local approximation.

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 neighbours:

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

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:

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

Remarks on Locally Weighted Regression

We considered using a linear function to approximate f in the neighbourhood of the query instance x_q. 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,

(2) These simple approximations model the target function quite well over a sufficiently small sub region of the instance space.

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. 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 centered at the point x_u with some variance σ^2_u.

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

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 σ^2_u 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.

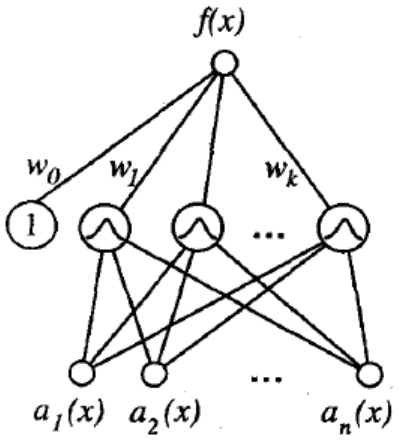


FIGURE 8.2
A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at some instance x_u. Therefore, its activation will be close to zero unless the input x is near x_u. The output unit produces a linear combination of the hidden unit activations.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

Methods 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 $(x_i, f(x_i))$, centering this Gaussian at the point x_i.

Each of these kernels may be assigned the same width σ^2. Given this approach, the RBF network learns a global approximation to the target function in which each training example $(x_i, f(x_i))$ can influence the value of f only in the neighbourhood of x_i. One advantage of this choice of kernel functions is that it allows the RBF network to fit the training data exactly.

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 centres spaced uniformly throughout the instance space X.

CASE-BASED REASONING

Instance-based methods such as k-NEARESTN NEIGHBOUR and locally weighted regression share three key properties.

1. 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.

2. They classify new query instances by analyzing similar instances while ignoring instances that are very different from the query.

3. 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.

Let us consider a prototypical example of a case-based reasoning system "CADET system".

- 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 water flow 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 water flow Q_3 increases with increasing input water flow Q_1. Similarly 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 water flow.
- 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 sub graph of the water faucet function graph.

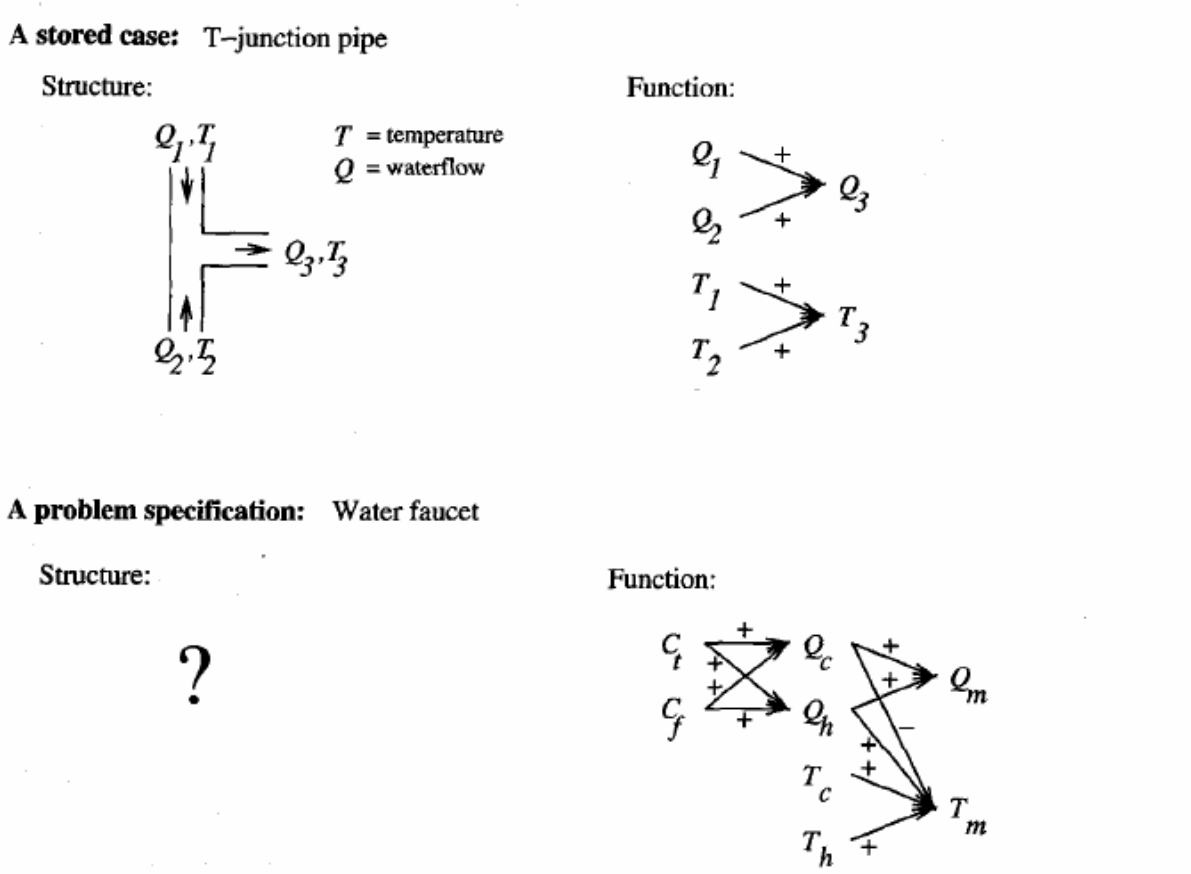


FIGURE 8.3
A stored case and a new problem. The top half of the figure describes a typical design fragment in the case library of CADET. The function is represented by the graph of qualitative dependencies among the T-junction variables (described in the text). The bottom half of the figure shows a typical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

Generic properties of case-based reasoning systems that distinguish them from approaches such as k-NEAREST NEIGHBOUR:

- 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 sub graph between two function graphs.

- Multiple retrieved cases may be combined to form the solution to the newproblem. This is similar to the k-NEAREST NEIGHBOUR approach, in that multiple similar cases are used to construct a response for the new query.
- There may be a tight coupling between case retrieval, knowledge-based reasoning, and problem solving.

REINFORCEMENT LEARNING
Chapter 13

INTRODUCTION

Consider building a learning robot. The robot, or *agent*, has a set of sensors (such as a camera and sonars) to observe the *state* of its environment, and a set of *actions* (such as "move forward" and "turn") it can perform to alter this state. 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.

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. 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.

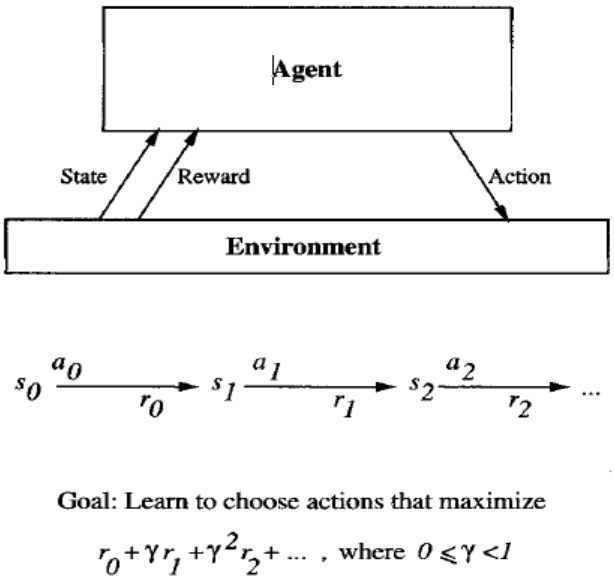


FIGURE 13.1
An agent interacting with its environment. The agent exists in an environment described by some set of possible states S. It can perform any of a set of possible actions A. Each time it performs an action a_t in some state s_t the agent receives a real-valued reward r_t that indicates the 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.

Reinforcement learning problem differs from other function approximation tasks in several important respects.

1. Delayed reward. The task of the agent is to learn a target function π that maps from the current state ***s*** to the optimal action **a** = **π(s). *E***arlier we have always assumed that when learning some target function such as **π,** each training example would be a pair of the form **(s, π(s)).** 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.

2. 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).

3. 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.

4. 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.

THE LEARNING TASK

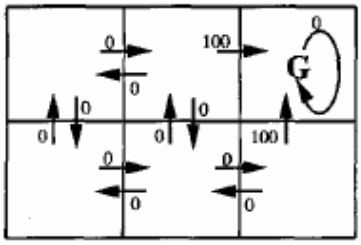
Consider a simple grid-world environment is depicted in the topmost diagram of Figure 13.2. The six grid squares in this diagram represent six possible states, or locations, for the agent. Each arrow in the diagram represents a possible action the agent can take to move from one state to another. The number associated with each arrow represents the immediate reward r(s, a) the agent receives if it executes the corresponding state-action transition. Note 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 γ, we can determine the optimal policy π* and its value function V*(s). In

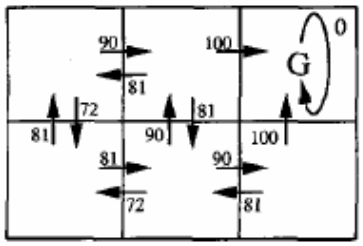
this case, let us choose $\gamma = 0.9$. The diagram at the bottom of the figure shows one optimal policy for this setting (there are others as well). The optimal policy directs the agent along the shortest path toward the state G.

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 centre 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 centre state is

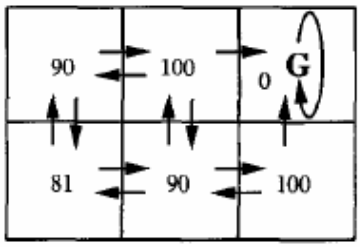
$$0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + \ldots = 90$$



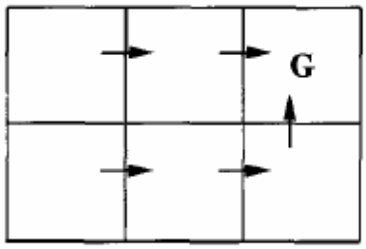
$r(s, a)$ (immediate reward) values



$Q(s, a)$ 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.

Q LEARNING

How can an agent learn an optimal policy n* 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 *(s, a).* Instead, the only training information available to the learner is the sequence of immediate rewards $r(s_i, a_i)$ for $i = 0, 1, 2, \ldots$ .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.

The Q Function

Let us define the evaluation function *Q(s, a)* so that the value of *Q* is the reward received immediately upon executing action *a* from state *s*, plus the value (discounted by y) of following the optimal policy thereafter.

$$Q(s,a) - r(s,a) + \gamma V^*(\delta(s,a)) \qquad 13.4$$

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 y. Note also that the optimal policy shown in the figure corresponds to selecting actions with maximal *Q* values.

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

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

This recursive definition of Q provides the basis for algorithms that iteratively approximate Q. Q^ to refer to the learner's estimate, or hypothesis, of the actual ***Q*** function. In this algorithm the learner represents its hypothesis Q^ by a large table with a separate entry for each state-action pair. The table entry for the pair (s, a) stores the value for 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. 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' = δ(s, a). It then updates the table entry for Q^(s, ***a)*** following each such transition, according to the rule:

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

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 Q^ converges in the limit to the actual Q function, provided the system can be modelled as a deterministic Markov decision process, the reward function r is bounded, and actions are chosen so that every state-action pair is visited infinitely often.

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:

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

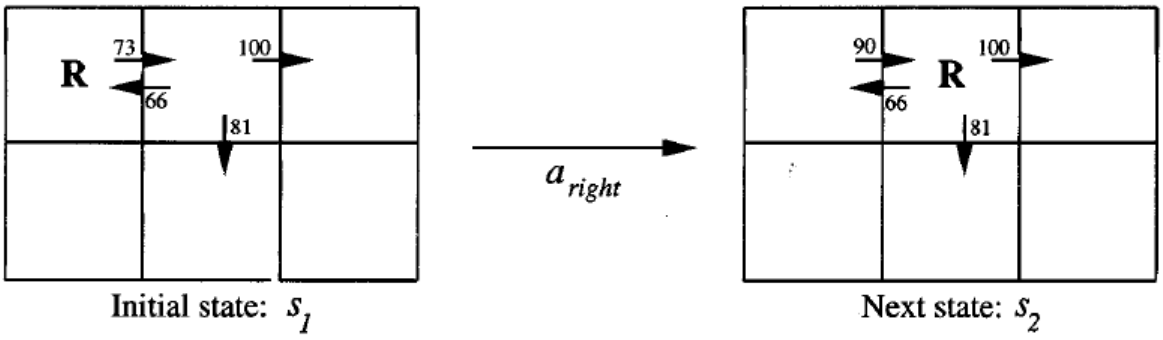
- $s \leftarrow s'$

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

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 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 Q^ for the state-action transition it just executed. According to the training rule, the new Q^ estimate for this transition is the sum of the received reward (zero) and the highest Q^ value associated with the resulting state (l00), discounted by γ (.9).

Each time the agent moves forward from an old state to a new one, Q learning propagates 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 Q^.



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

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 $\gamma = 0.9$.