Unit - III

Reinforcement Learning

Introduction

Reinforcement learning might be considered to encompass all of AI: an agent is placed in an environment and must learn to behave successfully therein. To keep the chapter manageable, we will concentrate on simple environments and simple agent designs. For the most part, we will assume a fully observable environment, so that the current state is supplied by each percept. On the other hand, we will assume that the agent does not know how the environment works or what its actions do, and we will allow for probabilistic action outcomes. Thus, the agent faces an unknown Markov decision process. We will consider three of the agent designs first introduced in Chapter 2:

• A utility-based agent learns a utility function on states and uses it to select actions that maximize the expected outcome utility.

Q-LEARNING • A Q-learning agent learns an action-utility function, or Q-function, giving the ex Q-FUNCTION pected utility of taking a given action in a given state.

• A reflex agent learns a policy that maps directly from states to actions.

A utility-based agent must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead. For example, in order to make use of a backgammon evaluation function, a backgammon program must know what its legal moves are and how they affect the board position. Only in this way can it apply the utility function to the outcome states. A Q-learning agent, on the other hand, can compare the expected utilities for its available choices without needing to know their outcomes, so it does not need a model of the environment. On the other hand, because they do not know where their actions lead, Q-learning agents cannot look ahead.

passive learning, where the agent's policy is fixed and the task is to learn the utilities of states (or state–action pairs); this could also involve learning ACTIVE LEARNING a model of the environment. The principal issue is exploration: an agent must experience as much as possible of its environment in order to learn how to behave in it.

Passive Reinforcement learning

- we start with the case of a passive learning agent using a state-based representation in a fully observable environment. In passive learning, the agent's policy π is fixed: in state s, it always executes the action $\pi(s)$. Its goal is simply to learn how good the policy is—that is, to learn the utility function $U\pi(s)$. We will use as our example the 4×3 world.
- shows a policy for that world and the corresponding utilities. Clearly, the passive learning task is similar to the policy evaluation task, part of the policy iteration algorithm described in Section 17.3. The main difference is that the passive learning agent does not know the transition model P(s | s, a), which specifies the probability of reaching state s from state s after doing action a; nor does it know the reward function R(s), which specifies the reward for each state

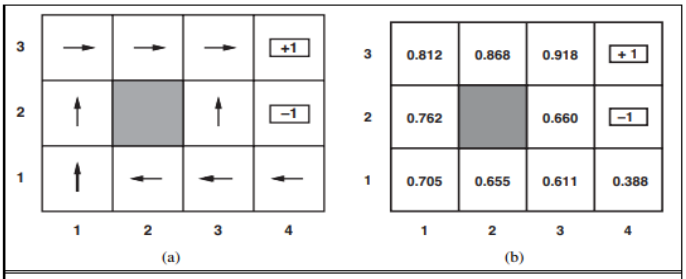


Figure 21.1 (a) A policy π for the 4×3 world; this policy happens to be optimal with rewards of $R(s) = -0.04$ in the nonterminal states and no discounting. (b) The utilities of the states in the 4×3 world, given policy π.

The agent executes a set of trials in the environment using its policy π. In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received in that state.

Typical trials might look like this:

$$(1,1)_{\textbf{-.04}} \rightsquigarrow (1,2)_{\textbf{-.04}} \rightsquigarrow (1,3)_{\textbf{-.04}} \rightsquigarrow (1,2)_{\textbf{-.04}} \rightsquigarrow (1,3)_{\textbf{-.04}} \rightsquigarrow (2,3)_{\textbf{-.04}} \rightsquigarrow (3,3)_{\textbf{-.04}} \rightsquigarrow (4,3)_{\textbf{+1}}$$
$$(1,1)_{\textbf{-.04}} \rightsquigarrow (1,2)_{\textbf{-.04}} \rightsquigarrow (1,3)_{\textbf{-.04}} \rightsquigarrow (2,3)_{\textbf{-.04}} \rightsquigarrow (3,3)_{\textbf{-.04}} \rightsquigarrow (3,2)_{\textbf{-.04}} \rightsquigarrow (3,3)_{\textbf{-.04}} \rightsquigarrow (4,3)_{\textbf{+1}}$$
$$(1,1)_{\textbf{-.04}} \rightsquigarrow (2,1)_{\textbf{-.04}} \rightsquigarrow (3,1)_{\textbf{-.04}} \rightsquigarrow (3,2)_{\textbf{-.04}} \rightsquigarrow (4,2)_{\textbf{-1}} \ .$$

Note that each state percept is subscripted with the reward received. The object is to use the information about rewards to learn the expected utility Uπ(s) associated with each nonterminal state s. The utility is defined to be the expected sum of (discounted) rewards obtained if policy π is followed.
we write

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

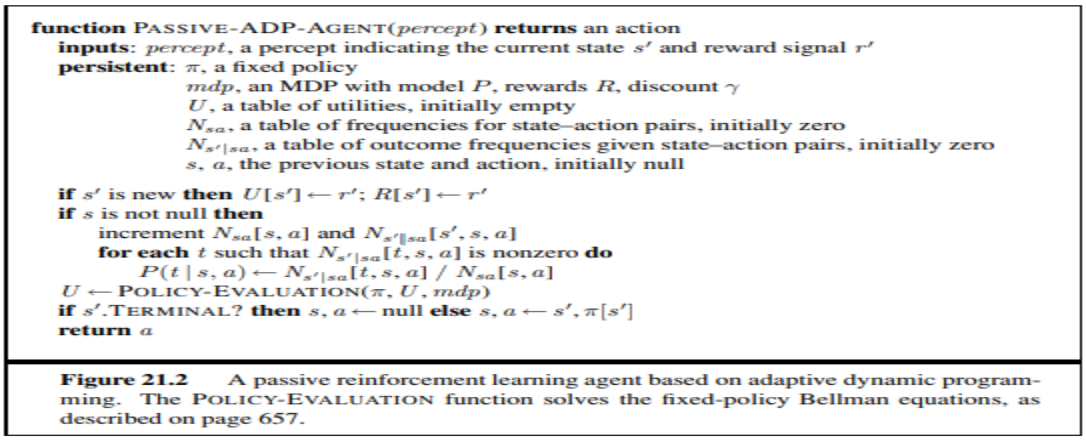
where R(s) is the reward for a state, St (a random variable) is the state reached at time t when executing policy π, and S0 = s. We will include a discount factor γ in all of our equations, but for the 4 × 3 world we will set γ = 1.

Direct utility estimation

- A simple method for direct utility estimation was invented in the late 1950s in the area of adaptive control theory by Widrow and Hoff (1960). The idea is that the utility of a state is the expected total reward from that state onward (called the expected reward-to-go), and each trial provides a sample of this quantity for each state visited.
- For example, the first trial in the set of three given earlier provides a sample total reward of 0.72 for state (1,1), two samples of 0.76 and 0.84 for (1,2), two samples of 0.80 and 0.88 for (1,3), and so on. Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state in a table. In the limit of infinitely many trials
- Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem, about which much is known. Unfortunately, it misses a very important source of information, namely, the fact that the utilities of states are not independent! The utility of each state equals its own reward plus the expected utility of its successor states.

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) U^{\pi}(s')$$

- By ignoring the connections between states, direct utility estimation misses opportunities for learning. For example, the second of the three trials given earlier reaches the state (3,2), which has not previously been visited.
- The next transition reaches (3,3), which is known from the first trial to have a high utility. The Bellman equation suggests immediately that (3,2) is also likely to have a high utility, because it leads to (3,3), but direct utility estimation learns nothing until the end of the trial. More broadly, we can view direct utility estimation as searching for U in a hypothesis space that is much larger than it needs to be, in that it includes many functions that violate the Bellman equations. For this reason, the algorithm often converges very slowly



function PASSIVE-ADP-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    mdp, an MDP with model P, rewards R, discount γ
    U, a table of utilities, initially empty
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    $N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s' is new **then** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$
    for each t such that $N_{s'|sa}[t, s, a]$ is nonzero **do**
      $P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \,/\, N_{sa}[s, a]$
  $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
  if s'.TERMINAL? **then** $s, a \leftarrow$ null **else** $s, a \leftarrow s', \pi[s']$
  return a

Figure 21.2 A passive reinforcement learning agent based on adaptive dynamic programming. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page 657.

Adaptive dynamic programming

- An adaptive dynamic programming (or ADP) agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using a dynamic programming method. For a passive learning agent, this means plugging the learned transition model P(s | s, π(s)) and the observed rewards R(s) into the Bellman equations to calculate the utilities of the states.
- As we remarked in our discussion of policy iteration , these equations are linear (no maximization involved) so they can be solved using any linear algebra package.

- Alternatively, we can adopt the approach of modified policy iteration using a simplified value iteration process to update the utility estimates after each change to the learned model. Because the model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and should converge quite quickly.

- The process of learning the model itself is easy, because the environment is fully observable. This means that we have a supervised learning task where the input is a state–action pair and the output is the resulting state.
- In the simplest case, we can represent the transition model as a table of probabilities. We keep track of how often each action outcome occurs and estimate the transition probability P(s | s, a) from the frequency with which s is reached when executing a in s. For example, in the three trials Right is

executed three times in (1,3) and two out of three times the resulting state is (2,3), so P((2, 3)| (1, 3), Right) is estimated to be 2/3.

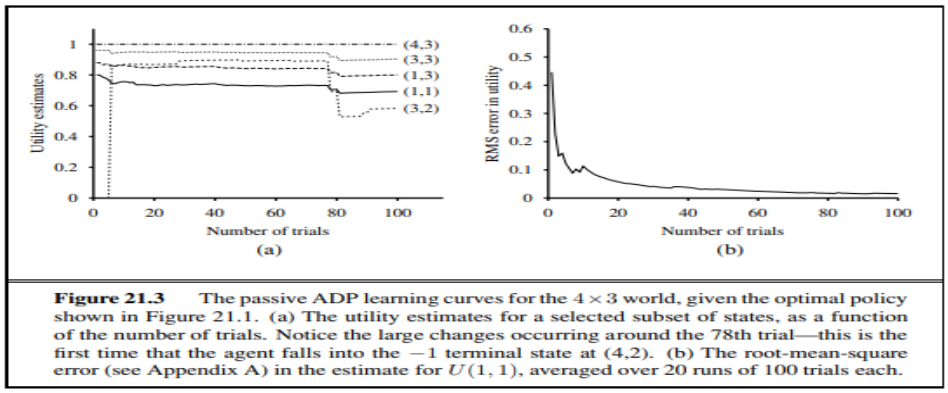


Figure 21.3 The passive ADP learning curves for the 4×3 world, given the optimal policy shown in Figure 21.1. (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice the large changes occurring around the 78th trial—this is the first time that the agent falls into the -1 terminal state at (4,2). (b) The root-mean-square error (see Appendix A) in the estimate for $U(1, 1)$, averaged over 20 runs of 100 trials each.

The full agent program for a passive ADP agent

- Its performance on the 4 × 3 world is shown in Figure 21.3. In terms of how quickly its value estimates improve, the ADP agent is limited only by its ability to learn the transition model.
- In this sense, it provides a standard against which to measure other reinforcement learning algorithms. It is, however, intractable for large state spaces. In backgammon, for example, it would involve solving roughly 1050 equations in 1050 unknowns.
- A reader familiar with the Bayesian learning ideas of Chapter 20 will have noticed that the algorithm is using maximum-likelihood estimation to learn the transition model; moreover, by choosing a policy based solely on the estimated model it is acting as if the model were correct. This is not necessarily a good idea! For example, a taxi agent that didn't know about how traffic lights might ignore a red light once or twice without no ill effects and then formulate a policy to ignore red lights from then on. Instead, it might be a good idea to choose a policy that, while not optimal for the model estimated by maximum likelihood, works reasonably well for the whole range of models that have a reasonable chance of being the true model.
- There are two mathematical approaches that have this flavor. The first approach, Bayesian reinforcement learning, assumes a prior probability P(h) for each hypothesis h about what the true model is; the posterior probability P(h | e) is obtained in the usual way by Bayes' rule given the observations to date.
- Then, if the agent has decided to stop learning, the optimal policy is the one that gives the highest expected utility. Let uπ h be the expected utility, averaged over all possible start states, obtained by executing policy π in model h. Then we have

$$\pi^* = \operatorname*{argmax}_{\pi} \sum_{h} P(h \mid \mathbf{e}) u_h^{\pi} .$$

Often, the set H will be the set of models that exceed some likelihood threshold on P(h | e), so the robust and Bayesian approaches are related. Sometimes, the robust solution can be computed efficiently. There are, moreover, reinforcement learning algorithms that tend to produce robust solutions, although we do not cover them here.

Temporal-difference learning

Solving the underlying MDP as in the preceding section is not the only way to bring the Bellman equations

to bear on the learning problem. Another way is to use the observed transitions to adjust the utilities of the observed states so that they agree with the constraint equations. Consider, for example, the transition from (1,3) to (2,3) in the second trial on page 832. Suppose that, as a result of the first trial, the utility estimates are $U^\pi(1, 3) = 0.84$ and $U^\pi(2, 3) = 0.92$. Now, if this transition occurred all the time, we would expect the utilities to obey the equation

$$U^\pi(1,3) = -0.04 + U^\pi(2,3) \ ,$$

so $U^\pi(1, 3)$ would be 0.88. Thus, its current estimate of 0.84 might be a little low and should be increased. More generally, when a transition occurs from state s to state s , we apply the following update to $U^\pi(s)$:

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s)) \ .$$

Here, α is the learning rate parameter. Because this update rule uses the difference in utilities between successive states, it is often called the temporal-difference, or TD, equation. TEMPORALDIFFERENCE

All temporal-difference methods work by adjusting the utility estimates towards the ideal equilibrium that holds locally when the utility estimates are correct. In the case of passive learning, the equilibrium does in fact cause the agent to reach the equilibrium but there is some subtlety involved. First, notice that the update involves only the observed successor s , whereas the actual equilibrium conditions involve all possible next states.

One might think that this causes an improperly large change in $U^\pi(s)$ when a very rare transition occurs; but, in fact, because rare transitions occur only rarely, the average value of $U^\pi(s)$ will converge to the correct value. Furthermore, if we change α from a fixed parameter to a function that decreases as the number of times a state has been visited increases, then $U^\pi(s)$ itself will converge to the correct value.

function PASSIVE-TD-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    U, a table of utilities, initially empty
    N_s, a table of frequencies for states, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if s' is new **then** $U[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$
  if s'.TERMINAL? **then** $s, a, r \leftarrow$ null **else** $s, a, r \leftarrow s', \pi[s'], r'$
  return a

Figure 21.4 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function $\alpha(n)$ is chosen to ensure convergence, as described in the text.

This gives us the agent program shown in Figure 21.4. Figure 21.5 illustrates the performance of the passive TD agent on the 4 × 3 world. It does not learn quite as fast as the ADP agent and shows much higher variability, but it is much simpler and requires much less computation per observation. Notice that TD does not need a transition model to perform its updates. The environment supplies the connection

between neighboring states in the form of observed transitions.

The ADP approach and the TD approach are actually closely related. Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. One difference is that TD adjusts a state to agree with its observed successor , whereas ADP adjusts the state to agree with all of the successors that might occur, weighted by their probabilities.

This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each successor in the set of transitions is approximately proportional to its probability.
A more important difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimates U and the environment model P. Although the observed transition makes only a local change in P, its effects might need to be propagated throughout U .
Thus, TD can be viewed as a crude but efficient first approximation to ADP. Each adjustment made by ADP could be seen, from the TD point of view, as a result of a "pseudoexperience" generated by simulating the current environment model.
It is possible to extend the TD approach to use an environment model to generate several pseudo experiences—transitions that the TD agent can imagine might happen, given its current model. For each observed transition, the TD agent can generate a large number of imaginary transitions. In this way, the resulting utility estimates will approximate more and more closely those of ADP—of course, at the expense of increased computation time.

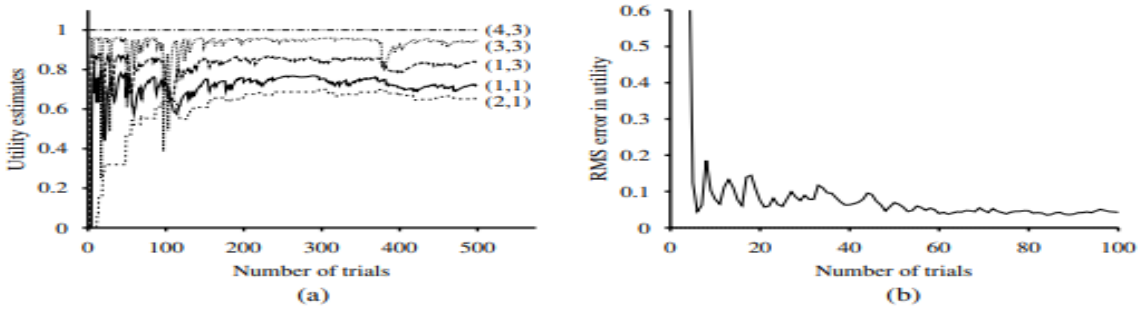


Figure 21.5 The TD learning curves for the 4×3 world. (a) The utility estimates for a selected subset of states, as a function of the number of trials. (b) The root-mean-square error in the estimate for $U(1, 1)$, averaged over 20 runs of 500 trials each. Only the first 100 trials are shown to enable comparison with Figure 21.3.

In a similar vein, we can generate more efficient versions of ADP by directly approximating the algorithms for value iteration or policy iteration. Even though the value iteration algorithm is efficient, it is intractable if we have, say, 10100 states. However, many of the necessary adjustments to the state values on each iteration will be extremely tiny.
One possible approach to generating reasonably good answers quickly is to bound the number of adjustments made after each observed transition. One can also use a heuristic to rank the possible adjustments so as to carry out only the most significant ones. The prioritized sweeping heuristic prefers to make adjustments to states whose likely successors have just undergone a large adjustment in their own utility estimates.
Using heuristics like this, approximate ADP algorithms usually can learn roughly as fast as full ADP, in terms of the number of training sequences, but can be several orders of magnitude more efficient in terms of computation. (See Exercise 21.3.) This enables them to handle state spaces that are far too large for full

ADP. Approximate ADP algorithms have an additional advantage: in the early stages of learning a new environment, the environment model P often will be far from correct, so there is little point in calculating an exact utility function to match it. An approximation algorithm can use a minimum adjustment size that decreases as the environment model becomes more accurate. This eliminates the very long value iterations that can occur early in learning due to large changes in the model.

Active Reinforcement Learning

A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take. Let us begin with the adaptive dynamic programming agent and consider how it must be modified to handle this new freedom.
First, the agent will need to learn a complete model with outcome probabilities for all actions, rather than just the model for the fixed policy. The simple learning mechanism used by PASSIVE-ADP-AGENT will do just fine for this. Next, we need to take into account the fact that the agent has a choice of actions. The utilities it needs to learn are those defined by the optimal policy; they obey the Bellman equations given on page 652, which we repeat here for convenience:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s' \mid s, a) U(s') \ .$$

These equations can be solved to obtain the utility function U using the value iteration or policy iteration algorithms from Chapter 17. The final issue is what to do at each step. Having obtained a utility function U that is optimal for the learned model, the agent can extract an optimal action by one-step look-ahead to maximize the expected utility; alternatively, if it uses policy iteration, the optimal policy is already available, so it should simply execute the action the optimal policy recommends. Or should it?

Exploration

the results of one sequence of trials for an ADP agent that follows the recommendation of the optimal policy for the learned model at each step. The agent does not learn the true utilities or the true optimal policy! What happens instead is that, in the 39th trial, it finds a policy that reaches the +1 reward along the lower route via (2,1), (3,1), (3,2), and (3,3).

After experimenting with minor variations, from the 276th trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3). We call this agent the greedy agent. Repeated experiments show that the greedy agent very seldom converges to the optimal policy for this environment and sometimes converges to really horrendous policies. How can it be that choosing the optimal action leads to suboptimal results? The answer is that the learned model is not the same as the true environment; what is optimal in the learned model can therefore be suboptimal in the true environment.

Unfortunately, the agent does not know what the true environment is, so it cannot compute the optimal action for the true environment. What, then, is to be done? What the greedy agent has overlooked is that actions do more than provide rewards according to the current learned model; they also contribute to learning the true model by affecting the percepts that are received. By improving the model, the agent will receive greater rewards in the future. An agent therefore must make a tradeoff between exploitation to maximize its reward—as reflected in its current utility estimates—and exploration to maxi mize its long-term well-being. Pure exploitation risks getting stuck in a rut. Pure exploration to improve one’s

knowledge is of no use if one never puts that knowledge into practice. In the real world, one constantly has to decide between continuing in a comfortable existence and striking out into the unknown in the hopes of discovering a new and better life. With greater understanding, less exploration is necessary. Can we be a little more precise than this? Is there an optimal exploration policy? This question has been studied in depth in the subfield of statistical decision theory that deals with so-called bandit problems.

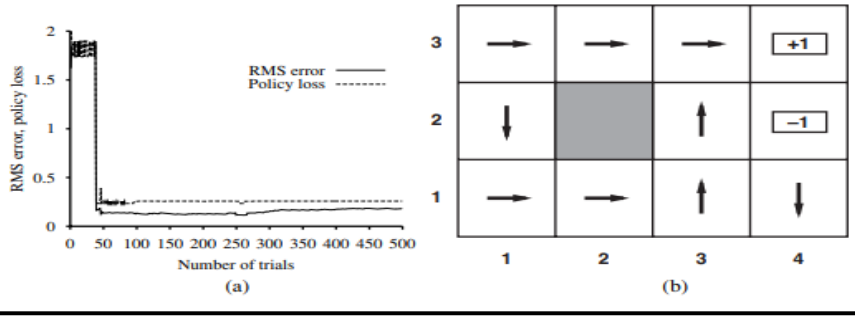


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials.

Although bandit problems are extremely difficult to solve exactly to obtain an optimal exploration method, it is nonetheless possible to come up with a reasonable scheme that will eventually lead to optimal behavior by the agent. Technically, any such scheme needs to be greedy in the limit of infinite exploration, or GLIE. A GLIE scheme must try each action in each state an unbounded number of times to avoid having a finite probability that an optimal action is missed because of an unusually bad series of outcomes. An ADP agent using such a scheme will eventually learn the true environment model. A GLIE scheme must also eventually become greedy, so that the agent's actions become optimal with respect to the learned (and hence the true) model.

There are several GLIE schemes; one of the simplest is to have the agent choose a random action a fraction 1/t of the time and to follow the greedy policy otherwise. While this does eventually converge to an optimal policy, it can be extremely slow. A more sensible approach would give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility. This can be implemented by altering the constraint equation (21.4) so that it assigns a higher utility estimate to relatively unexplored state–action pairs. Essentially, this amounts to an optimistic prior over the possible environments and causes the agent to behave initially as if there were wonderful rewards scattered all over the place. Let us use U +(s) to denote the optimistic estimate of the utility (i.e., the expected reward-to-go) of the state s, and let N(s, a) be the number of times action a has been tried in state s. Suppose we are using value iteration in an ADP learning agent; then we need to rewrite the update equation (Equation (17.6) on page 652) to incorporate the optimistic estimate.

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s' \mid s,a) U^+(s'),\ N(s,a)\right) .$$

Here, f(u, n) is called the exploration function. It determines how greed (preference for high values of u) is traded off against curiosity (preference for actions that have not been tried often and have low n). The function f(u, n) should be increasing in u and decreasing in n. Obviously, there are many possible functions that fit these conditions. One particularly simple definition is

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

where R+ is an optimistic estimate of the best possible reward obtainable in any state and Ne is a fixed parameter. This will have the effect of making the agent try each action–state pair at least Ne times. The fact that U + rather than U appears on the right-hand side of Equation (21.5) is very important. As exploration proceeds, the states and actions near the start state might well be tried a large number of times. If we used U, the more pessimistic utility estimate, then the agent would soon become disinclined to explore further afield. The use of U + means that the benefits of exploration are propagated back from the edges of unexplored regions, so that actions that lead toward unexplored regions are weighted more highly, rather than just actions that are themselves unfamiliar. The effect of this exploration policy can be seen clearly in Figure 21.7, which shows a rapid convergence toward optimal performance, unlike that of the greedy approach. A very nearly optimal policy is found after just 18 trials. Notice that the utility estimates themselves do not converge as quickly. This is because the agent stops exploring the unrewarding parts of the state space fairly soon, visiting them only "by accident" thereafter. However, it makes perfect sense for the agent not to care about the exact utilities of states that it knows are undesirable and can be avoided.

Learning an action-utility function

Now that we have an active ADP agent, let us consider how to construct an active temporaldifference learning agent. The most obvious change from the passive case is that the agent is no longer equipped with a fixed policy, so, if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on U via one-step look-ahead. The model acquisition problem for the TD agent is identical to that for the ADP agent. What of the TD update rule itself? Perhaps surprisingly, the update rule (21.3) remains unchanged. This might seem odd, for the following reason: Suppose the agent takes a step that normally leads to a good destination, but because of nondeterminism in the environment the agent ends up in a catastrophic state. The TD update rule will take this as seriously as if the outcome had been the normal result of the action, whereas one might suppose that, because the outcome was a fluke, the agent should not worry about it too much. In fact, of course, the unlikely outcome will occur only infrequently in a large set of training sequences; hence in the long run its effects will be weighted proportionally to its probability, as we would hope. Once again, it can be shown that the TD algorithm will converge to the same values as ADP as the number of training sequences tends to infinity.

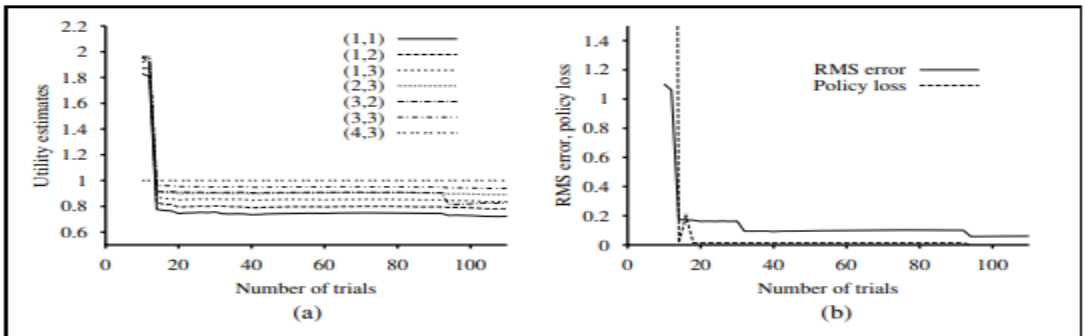


Figure 21.7 Performance of the exploratory ADP agent. using $R^+ = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. (b) The RMS error in utility values and the associated policy loss.

There is an alternative TD method, called Q-learning, which learns an action-utility representation instead of learning utilities. We will use the notation Q(s, a) to denote the value of doing action a in state s. Q-values are directly related to utility values as follows:

$$U(s) = \max_a Q(s, a) \, .$$

Q-functions may seem like just another way of storing utility information, but they have a very important

property: a TD agent that learns a Q-function does not need a model of the form P(s | s, a), either for learning or for action selection. For this reason, Q-learning is called a model-free method. As with utilities, we can write a constraint equation that must MODEL-FREE hold at equilibrium when the Q-values are correct:

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a'} Q(s',a') \ .$$

As in the ADP learning agent, we can use this equation directly as an update equation for an iteration process that calculates exact Q-values, given an estimated model. This does, however, require that a model also be learned, because the equation uses P(s | s, a). The temporal-difference approach, on the other hand, requires no model of state transitions—all it needs are the Q values. The update equation for TD Q-learning is

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)) \ ,$$

which is calculated whenever action a is executed in state s leading to state s . The complete agent design for an exploratory Q-learning agent using TD is shown in Figure 21.8. Notice that it uses exactly the same exploration function f as that used by the exploratory ADP agent—hence the need to keep statistics on actions taken (the table N). If a simpler exploration policy is used—say, acting randomly on some fraction of steps, where the fraction decreases over time—then we can dispense with the statistics. Q-learning has a close relative called SARSA (for State-Action-Reward-State-Action). The update rule for SARSA is very similar to Equation (21.8):

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma\, Q(s\ , a\ ) - Q(s, a)) \ ,$$

where a is the action actually taken in state s . The rule is applied at the end of each s, a, r, s , a quintuplet—hence the name. The difference from Q-learning is quite subtle: whereas Q-learning backs up the best Q-value from the state reached in the observed transition, SARSA waits until an action is actually taken and backs up the Q-value for that action. Now, for a greedy agent that always takes the action with best Q-value, the two algorithms are identical. When exploration is happening, however, they differ significantly. Because Q-learning uses the best Q-value, it pays no attention to the actual policy being followed—it is an off-policy learning algorithm, whereas SARSA is an on-policy algorithm. Q-learning is more flexible than SARSA, in the sense that a Q-learning agent can learn how to behave well even when guided by a random or adversarial exploration policy. On the other hand, SARSA is more realistic: for example, if the overall policy is even partly controlled by other agents, it is better to learn a Q-function for what will actually happen rather than what the agent would like to happen.

function Q-LEARNING-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if TERMINAL?(s) **then** $Q[s, None] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$
    $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
  $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$
  return a

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s, a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

Both Q-learning and SARSA learn the optimal policy for the 4 × 3 world, but do so at a much slower rate than the ADP agent. This is because the local updates do not enforce consistency among all the Q-values via the model. The comparison raises a general question: is it better to learn a model and a utility function or to learn an action-utility function with no model?

In other words, what is the best way to represent the agent function? This is an issue at the foundations of artificial intelligence. As we stated in Chapter 1, one of the key historical characteristics of much of AI research is its (often unstated) adherence to the knowledge-based approach. This amounts to an assumption that the best way to represent the agent function is to build a representation of some aspects of the environment in which the agent is situated.

Some researchers, both inside and outside AI, have claimed that the availability of model-free methods such as Q-learning means that the knowledge-based approach is unnecessary. There is, however, little to go on but intuition. Our intuition, for what it's worth, is that as the environment becomes more complex, the advantages of a knowledge-based approach become more apparent.

This is borne out even in games such as chess, checkers (draughts), and backgammon (see next section), where efforts to learn an evaluation function by means of a model have met with more success than Q-learning methods.

Generalization in Reinforcement Learning

we have assumed that the utility functions and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple. Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time per iteration increase rapidly as the space gets larger. With carefully controlled, approximate ADP methods, it might be possible to handle 10,000 states or more.

This suffices for two-dimensional maze-like environments, but more realistic worlds are out of the question. Backgammon and chess are tiny subsets of the real world, yet their state spaces contain on the order of 1020 and 1040 states, respectively. It would be absurd to suppose that one must visit all these states many times in order to learn how to play the game! One way to handle such problems is to use function approximation, which simply means using any sort of representation for the Q-function other than a lookup table.

The representation is viewed as approximate because it might not be the case that the true utility function

or Q-function can be represented in the chosen form. For example, in Chapter 5 we described an evaluation function for chess that is represented as a weighted linear function of a set of features (or basis functions) f1,...,fn: $\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$.
A reinforcement learning algorithm can learn values for the parameters
$\theta = \theta_1,...,\theta_n$
such that the evaluation function $\hat{U}_\theta$ approximates the true utility function.
Instead of, say, 1040 values in a table, this function approximator is characterized by,
say, n = 20 parameters— an enormous compression.
Although no one knows the true utility function for chess, no one believes that it can be represented exactly in 20 numbers. If the approximation is good enough, however, the agent might still play excellent chess.
3 Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit. The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited. That is, the most important aspect of function approximation is not that it requires less space, but that it allows for inductive generalization over input states. To give you some idea of the power of this effect: byexamining only one in every 1012 of the possible backgammon states, it is possible to learn a utility function that allows a program to play as well as any human.

On the flip side, of course, there is the problem that there could fail to be any function in the chosen hypothesis space that approximates the true utility function sufficiently well.
As in all inductive learning, there is a tradeoff between the size of the hypothesis space and the time it takes to learn the function. A larger hypothesis space increases the likelihood that a good approximation can be found, but also means that convergence is likely to be delayed.

Let us begin with the simplest case, which is direct utility estimation.

With function approximation, this is an instance of supervised learning. For example, suppose we represent the utilities for the 4 × 3 world using a simple linear function. The features of the squares are just their x and y coordinates, so we have
$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$. (21.10)

Thus, if $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}_\theta(1, 1) = 0.8$. Given a collection of trials, we obtain a set of sample values of $\hat{U}_\theta(x, y)$, and we can find the best fit, in the sense of minimizing the squared error, using standard linear regression.
For reinforcement learning, it makes more sense to use an online learning algorithm that updates the parameters after each trial. Suppose we run a trial and the total reward obtained starting at (1,1) is 0.4.

This suggests that $\hat{U}_\theta(1, 1)$, currently 0.8, is too large and must be reduced. How should the parameters be adjusted to achieve this? As with neural network learning, we write an error function and compute its gradient with respect to the parameters.
If uj (s) is the observed total reward from state s onward in the jth trial, then the error is defined as (half) the squared difference of the predicted total and the actual total:
$E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2/2$.
The rate of change of the error with respect to each parameter θ_i is $\partial E_j/\partial\theta_i$, so to move the parameter in the direction of decreasing the error, we want

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha\,(u_j(s) - \hat{U}_\theta(s)) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} .$$

This is called the Widrow–Hoff rule, or the delta rule, for online least-squares. For the linear function

approximator Uˆθ(s) in Equation (21.10), we get three simple update rules:

θ0 ← θ0 + α (uj (s) − Uˆθ(s)) ,
θ1 ← θ1 + α (uj (s) − Uˆθ(s))x ,
θ2 ← θ2 + α (uj (s) − Uˆθ(s))y

We can apply these rules to the example where Uˆθ(1, 1) is 0.8 and uj (1, 1) is 0.4. θ0, θ1, and θ2 are all decreased by 0.4α, which reduces the error for (1,1). Notice that changing the parameters θ in response to an observed transition between two states also changes the values of Uˆθ for every other state

If we put the +1 reward at (5,5), the true utility is more like a pyramid and the function will fail miserably. All is not lost, however! Remember that what matters for linear function approximation is that the function be linear in the parameters—the features themselves can be arbitrary nonlinear functions of the state variables. Hence, we can include a term such as θ3f3(x, y) = θ3(x − xg)2 + (y − yg)2 that measures the distance to the goal.

We can apply these ideas equally well to temporal-difference learners. All we need do is adjust the parameters to try to reduce the temporal difference between successive states. The new versions of the TD and Q-learning equations are given by

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma\,\hat{U}_\theta(s') - \hat{U}_\theta(s)]\frac{\partial U_\theta(s)}{\partial \theta_i}$$

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma\,\max_{a'}\hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)]\frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i}$$

for Q-values. For passive TD learning, the update rule can be shown to converge to the closest possible approximation to the true function when the function approximator is linear in the parameters. With active learning and nonlinear functions such as neural networks, all bets are off:

There are some very simple cases in which the parameters can go off to infinity even though there are good solutions in the hypothesis space. There are more sophisticated algorithms that can avoid these problems, but at present reinforcement learning with general function approximators remains a delicate art.

Function approximation can also be very helpful for learning a model of the environment. Remember that learning a model for an observable environment is a supervised learning problem, because the next percept gives the outcome state. Any of the supervised learning methods can be used, with suitable adjustments for the fact that we need to predict a complete state description rather than just a Boolean classification or a single real value.

For a partially observable environment, the learning problem is much more difficult. If we know what the hidden variables are and how they are causally related to each other and to the observable variables, then we can fix the structure of a dynamic Bayesian network and use the EM algorithm to learn the parameters. Inventing the hidden variables and learning the model structure are still open problems.

Policy search

The final approach we will consider for reinforcement learning problems is called policy search. In some ways, policy search is the simplest of all the methods in this chapter: the idea is to keep twiddling the

policy as long as its performance improves, then stop.

Let us begin with the policies themselves. Remember that a policy π is a function that maps states to actions. We are interested primarily in parameterized representations of π that have far fewer parameters than there are states in the state space (just as in the preceding section). For example, we could represent π by a collection of parameterized Q-functions, one for each action, and take the action with the highest predicted value:

$$\pi(s) = \max_a \hat{Q}_\theta(s, a)$$

Each Q-function could be a linear function of the parameters θ, or it could be a nonlinear function such as a neural network. Policy search will then adjust the parameters θ to improve the policy. Notice that if the policy is represented by Q-functions, then policy search results in a process that learns Q-functions. This process is not the same as Q-learning! In Q-learning with function approximation, the algorithm finds a value of θ such that $\hat{Q}_\theta$ is "close" to Q_*, the optimal Q-function. Policy search, on the other hand, finds a value of θ that results in good performance; the values found by the two methods may differ very substantially. (For example, the approximate Q-function defined

by $\hat{Q}_\theta(s, a) = Q_*(s, a)/10$ gives optimal performance, even though it is not at all close to Q_*.) Another clear instance of the difference is the case where $\pi(s)$ is calculated using, say, depth-10 look-ahead search with an approximate utility function $\hat{U}_\theta$. A value of θ that gives good results may be a long way from making $\hat{U}_\theta$ resemble the true utility function. One problem with policy representations of the kind given in is that

the policy is a discontinuous function of the parameters when the actions are discrete. (For a continuous action space, the policy can be a smooth function of the parameters.) That is, there will be values of θ such that an infinitesimal change in θ causes the policy to switch from one action to another. This means that the value of the policy may also change discontinuously, which makes gradient-based search difficult. For this reason, policy search methods often use a stochastic policy representation $\pi_\theta(s, a)$, which specifies the probability of selecting action in state s. One popular representation is the softmax function:

$$\pi_\theta(s, a) = e^{\hat{Q}_\theta(s,a)} / \sum_{a'} e^{\hat{Q}_\theta(s,a')} .$$

Softmax becomes nearly deterministic if one action is much better than the others, but it always gives a differentiable function of θ; hence, the value of the policy (which depends in a continuous fashion on the action selection probabilities) is a differentiable function of θ. Softmax is a generalization of the logistic function (page 725) to multiple variables.

Now let us look at methods for improving the policy. We start with the simplest case: a deterministic policy and a deterministic environment. Let $\rho(\theta)$ be the policy value, i.e., the expected reward-to-go when π_θ is executed. If we can derive an expression for $\rho(\theta)$ in closed form, then we have a standard optimization problem, as described in.

the policy gradient vector $\nabla_\theta \rho(\theta)$ provided $\rho(\theta)$ is differentiable. Alternatively, if $\rho(\theta)$ is not available in closed form, we can evaluate π_θ simply by executing it and observing the accumulated reward. We can follow the empirical gradient by hill climbing—i.e., evaluating the change in policy value for small increments in each parameter. With the usual caveats, this process will converge to a local optimum in policy space. When the environment (or the policy) is stochastic, things get more difficult. Suppose we are trying to do hill climbing, which requires comparing $\rho(\theta)$ and $\rho(\theta + \Delta\theta)$ for some small $\Delta\theta$.

The problem is that the total reward on each trial may vary widely, so estimates of the policy value from a small number of trials will be quite unreliable; trying to compare
two such estimates will be even more unreliable. One solution is simply to run lots of trials, measuring the sample variance and using it to determine that enough trials have been run to get a reliable indication of the direction of improvement for ρ(θ). Unfortunately, this is impractical for many real problems where each trial may be expensive, time-consuming, and perhaps even dangerous.
For the case of a stochastic policy πθ(s, a), it is possible to obtain an unbiased estimate of the gradient at θ, ∇θρ(θ), directly from the results of trials executed at θ. For simplicity, we will derive this estimate for the simple case of a non sequential environment in which the reward R(a) is obtained immediately after doing action a in the start state s0. In this case, the policy value is just the expected value of the reward, and we

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0, a) R(a) = \sum_a (\nabla_\theta \pi_\theta(s_0, a)) R(a) \ .$$

Now we perform a simple trick so that this summation can be approximated by samples generated from the probability distribution defined by $\pi_\theta(s_0, a)$. Suppose that we have N trials in all and the action taken on the jth trial is a_j. Then

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a) \cdot \frac{(\nabla_\theta \pi_\theta(s_0, a)) R(a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s_0, a_j)) R(a_j)}{\pi_\theta(s_0, a_j)} \ .$$

Thus, the true gradient of the policy value is approximated by a sum of terms involving the gradient of the action-selection probability in each trial. For the sequential case, this generalizes to

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s, a_j)) R_j(s)}{\pi_\theta(s, a_j)}$$

have

for each state s visited, where aj is executed in s on the jth trial and Rj (s) is the total reward received from state s onwards in the jth trial. The resulting algorithm is calledREINFORCE (Williams, 1992); it is usually much more effective than hill climbing using lots of trials at each value of θ. It is still much slower than necessary, however.

Applications of Reinforcement Learning

We consider applications in game playing, where the transition model is known and the goal is to learn the utility function, and in robotics, where the model is usually unknown.
Applications to game playing

The first significant application of reinforcement learning was also the first significant learning program of any kind—the checkers program written by Arthur Samuel (1959, 1967). Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied a version of Equation (21.12) to update the weights. There were some significant differences, however, between his program and current methods. First, he updated the weights using the difference between the current state and the backed-up value generated by full look-ahead in the search tree. This works fine, because it amounts to viewing the state space at a different granularity. A second difference was that the program did not use any observed rewards! That is, the values of terminal states reached in self-play were ignored. This means that it is theoretically possible for Samuel's program not to converge, or to converge on a strategy designed to lose rather than to win. He managed to avoid this fate by insisting that the weight for material advantage should always be positive. Remarkably, this was sufficient to direct the program into areas of weight space corresponding to good checkers play. Gerry Tesauro's backgammon program TD-GAMMON

(1992) forcefully illustrates the potential of reinforcement learning techniques. In earlier work (Tesauro and Sejn owski, 1989), Tesauro tried learning a neural network representation of Q(s, a) directly from ex

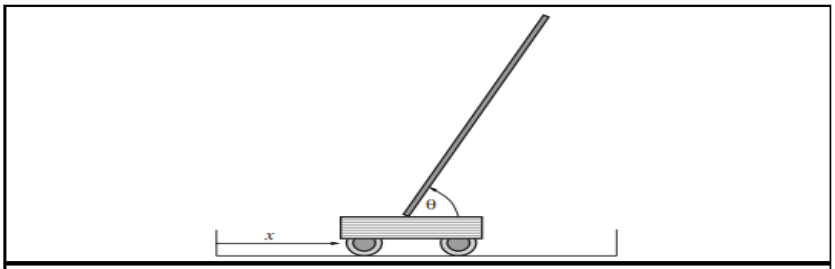


Figure 21.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

amples of moves labeled with relative values by a human expert. This approach proved extremely tedious for the expert. It resulted in a program, called NEUROGAMMON, that was strong by computer standards, but not competitive with human experts. The TD-GAMMON project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game. The evaluation function was represented by a fully connected neural network with a single hidden layer containing 40 nodes. Simply by repeated application of Equation (21.12), TD-GAMMON learned to play considerably better than NEUROGAMMON, even though the input representation contained just the raw board position with no computed features. This took about 200,000 training games and two weeks of computer time. Although that may seem like a lot of games, it is only a vanishingly small fraction of the state space. When pre computed features were added to the input representation, a network with 80 hidden nodes was able, after 300,000 training games, to reach a standard of play comparable to that of the top three human players worldwide. Kit Woolsey, a top player and analyst, said that "There is no question in my mind that its positional judgment is far better than mine."

Application to robot control

The setup for the famous cart–pole balancing problem, also known as the inverted pendulum, is shown in Figure 21.9. The problem is to control the position x of the cart so that the pole stays roughly upright ($\theta \approx \pi/2$), while staying within the limits of the cart track as shown. Several thousand papers in reinforcement learning and control theory have been published on this seemingly simple problem. The cart–pole problem differs from the problems described earlier in that the state variables x, θ, $\dot{x}$, and $\dot{\theta}$ are continuous. The actions are usually discrete: jerk left or jerk right, the so-called bang-bang control regime. BANG-BANG CONTROL The earliest work on learning for this problem was carried out by Michie and Chambers (1968). Their BOXES algorithm was able to balance the pole for over an hour after only about 30 trials. Moreover, unlike many subsequent systems, BOXES was implemented with a real cart and pole, not a simulation. The algorithm first discretized the four-dimensional state space into boxes—hence the name. It then ran trials until the pole fell over or the cart hit the end of the track. Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence. It was found that the discretization caused some problems when the apparatus was initialized in a position different from those used in training, suggesting that generalization was not perfect. Improved generalization and faster learning can be obtained using an algorithm that adaptively partitions the state space according to the observed variation in the reward, or by using a continuous-state, nonlinear function approximator such as a neural network. Nowadays, balancing a triple inverted pendulum is a common exercise—a feat far beyond the capabilities of most humans. Still more impressive is the application of

reinforcement learning to helicopter flight (Figure 21.10). This work has generally used policy search (Bagnell and Schneider, 2001) as well as the PEGASUS algorithm with simulation based on a learned transition model

Natural Language Processing

There are over a trillion pages of information on the Web, almost all of it in natural language. An agent that wants to do knowledge acquisition needs to understand the ambiguous, messy languages that humans use. We examine the problem from the point of view of specific information-seeking tasks: text classification, information retrieval, and information extraction. One common factor in addressing these tasks is the use of language models: models that predict the probability distribution of language expressions

Language Models

Formal languages, such as the programming languages Java or Python, have precisely defined language models. A language can be defined as a set of strings; "print(2 + 2)" is a legal program in the language Python, whereas "2)+(2 print" is not. Since there are an infinite number of legal programs, they cannot be enumerated; instead they are specified by a set of rules called a grammar. Formal languages also have rules that define the meaning or semantics of a program; for example, the rules say that the "meaning" of "2+2" is 4, and the meaning of "1/0" is that an error is signaled.

Natural languages, such as English or Spanish, cannot be characterized as a definitive set of sentences. Everyone agrees that "Not to be invited is sad" is a sentence of English, but people disagree on the grammaticality of "To be not invited is sad." Therefore, it is more fruitful to define a natural language model as a probability distribution over sentences rather than a definitive set. That is, rather than asking if a string of words is or is not a member of the set defining the language, we instead ask for P(S = words)—what is the probability that a random sentence would be words. Natural languages are also ambiguous. "He saw her duck" can mean either that he saw a waterfowl belonging to her, or that he saw her move to evade something. Thus, again, we cannot speak of a single meaning for a sentence, but rather of a probability distribution over possible meanings. Finally, natural languages are difficult to deal with because they are very large, and constantly changing. Thus, our language models are, at best, an approximation. We start with the simplest possible approximations and move up from there.

N-gram character models

Ultimately, a written text is composed of characters—letters, digits, punctuation, and spaces in English (and more exotic characters in some other languages). Thus, one of the simplest language models is a probability distribution over sequences of characters. As in Chapter 15, we write P(c1:N) for the probability of a sequence of N characters, c1 through cN . In one Web collection, P("the")=0.027 and P("zgq")=0.000000002. A sequence of written symbols of length n is called an n-gram (from the Greek root for writing or letters), with special case "unigram" for 1-gram, "bigram" for 2-gram, and "trigram" for 3-gram. A model of the probability distribution of n-letter sequences is thus called an n-gram model. (But be careful: we can have n-gram models over sequences of words, syllables, or other units; not just over characters.) An n-gram model is defined as a Markov chain of order n − 1. Recall from page 568 that in a Markov chain the probability of character ci depends only on the immediately preceding characters, not on any other characters. So in a trigram model (Markov chain of order 2) we have

$$P(c_i \mid c_{1:i-1}) = P(c_i \mid c_{i-2:i-1}) .$$

We can define the probability of a sequence of characters $P(c_{1:N})$ under the trigram model by first factoring with the chain rule and then using the Markov assumption:

$$P(c_{1:N}) = \prod_{i=1}^{N} P(c_i \mid c_{1:i-1}) = \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1}) .$$

For a trigram character model in a language with 100 characters, P(Ci|Ci−2:i−1) has a million entries, and can be accurately estimated by counting character sequences in a body of text of CORPUS 10 million characters or more. We call a body of text a corpus (plural corpora), from the Latin word for body.
What can we do with n-gram character models? One task for which they are well suited is language identification: given a text, determine what natural language it is written in. This is a relatively easy task; even with short texts such as “Hello, world” or “Wie geht es dir,” it is easy to identify the first as English and the second as German. Computer systems identify languages with greater than 99% accuracy; occasionally, closely related languages, such as Swedish and Norwegian, are confused. One approach to language identification is to first build a trigram character model of each candidate language, P(ci | ci−2:i−1,), where the variable ranges over languages. For each the model is built by counting trigrams in a corpus of that language. (About 100,000 characters of each language are needed.) That gives us a model of P(Text | Language), but we want to select the most probable language given the text, so we apply Bayes’ rule followed by the Markov assumption to get the most probable language:

$$\begin{aligned} \ell^* &= \operatorname*{argmax}_{\ell} P(\ell \mid c_{1:N}) \\ &= \operatorname*{argmax}_{\ell} P(\ell) P(c_{1:N} \mid \ell) \\ &= \operatorname*{argmax}_{\ell} P(\ell) \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1}, \ell) \end{aligned}$$

The trigram model can be learned from a corpus, but what about the prior probability P()? We may have some estimate of these values; for example, if we are selecting a random Web page we know that English is the most likely language and that the probability of Macedonian will be less than 1%. The exact number we select for these priors is not critical because the trigram model usually selects one language that is several orders of magnitude more probable than any other. Other tasks for character models include spelling correction, genre classification, and named-entity recognition. Genre classification means deciding if a text is a news story, a legal document, a scientific article, etc. While many features help make this classification, counts of punctuation and other character n-gram features go a long way (Kessler et al., 1997). Named-entity recognition is the task of finding names of things in a document and deciding what class they belong to. For example, in the text “Mr. Sopersteen was prescribed aciphex,” we should recognize that “Mr. Sopersteen” is the name of a person and “aciphex” is the name of a drug. Character-level models are good for this task because they can associate the character sequence “ex ” (“ex” followed by a space) with a drug name and “steen ” with a person name, and thereby identify words that they have never seen before.

Smoothing n-gram models

The major complication of n-gram models is that the training corpus provides only an estimate of the true probability distribution. For common character sequences such as “ th” any English corpus will give a good estimate: about 1.5% of all trigrams. On the other hand, “ ht” is very uncommon—no dictionary words start

with ht. It is likely that the sequence would have a count of zero in a training corpus of standard English. Does that mean we should assign P(“ th”)=0? If we did, then the text “The program issues an http request” would have an English probability of zero, which seems wrong. We have a problem in generalization: we want our language models to generalize well to texts they haven’t seen yet. Just because we have never seen “ http” before does not mean that our model should claim that it is impossible. Thus, we will adjust our language model so that sequences that have a count of zero in the training corpus will be assigned a small nonzero probability (and the other counts will be adjusted downward slightly so that the probability still sums to 1). The process od adjusting the probability of low-frequency counts is called smoothing. The simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th century: he said that, in the lack of further information, if a random Boolean variable X has been false in all n observations so far then the estimate for P(X = true) should be 1/(n+ 2). That is, he assumes that with two more trials, one might be true and one false. Laplace smoothing (also called add-one smoothing) is a step in the right direction, but performs relatively poorly. A better approach is a backoff model, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to (n − 1)-grams. Linear interpolation smoothing is a backoff model that combines trigram, bigram, and unigram models by linear interpolation. It defines the probability estimate as
$P^*(c_i|c_{i-2:i-1}) = \lambda_3 P(c_i|c_{i-2:i-1}) + \lambda_2 P(c_i|c_{i-1}) + \lambda_1 P(c_i)$, where $\lambda_3 + \lambda_2 + \lambda_1 = 1$.
The parameter values λ_i can be fixed, or they can be trained with an expectation–maximization algorithm. It is also possible to have the values of λ_i depend on the counts: if we have a high count of trigrams, then we weigh them relatively more; if only a low count, then we put more weight on the bigram and unigram models. One camp of researchers has developed ever more sophisticated smoothing models, while the other camp suggests gathering a larger corpus so that even simple smoothing models work well. Both are getting at the same goal: reducing the variance in the language model. One complication: note that the expression $P(c_i \mid c_{i-2:i-1})$ asks for $P(c_1 \mid c_{-1:0})$ when i = 1, but there are no characters before c1.
We can introduce artificial characters, for example, defining c_0 to be a space character or a special “begin text” character. Or we can fall back on lower-order Markov models, in effect defining $c_{-1:0}$ to be the empty sequence and thus $P(c_1 \mid c_{-1:0}) = P(c_1)$.

Model evaluation

With so many possible n-gram models—unigram, bigram, trigram, interpolated smoothing with different values of λ, etc.—how do we know what model to choose? We can evaluate a model with cross-validation. Split the corpus into a training corpus and a validation corpus. Determine the parameters of the model from the training data. Then evaluate the model on the validation corpus. The evaluation can be a task-specific metric, such as measuring accuracy on language identification. Alternatively we can have a task-independent model of language quality: calculate the probability assigned to the validation corpus by the model; the higher the probability the better. This metric is inconvenient because the probability of a large corpus will be a very small number, and floating-point underflow becomes an issue. A different way of describing the probability of a sequence is with a measure called perplexity, defined as $\text{Perplexity}(c_{1:N}) = P(c_{1:N})^{-1/N}$. Perplexity can be thought of as the reciprocal of probability, normalized by sequence length. It can also be thought of as the weighted average branching factor of a model. Suppose there are 100 characters in our language, and our model says they are all equally likely. Then for a sequence of any length, the perplexity will be 100. If some characters are more likely than others, and the model reflects that, then the model will have a perplexity less than 100.

N-gram word models

Now we turn to n-gram models over words rather than characters. All the same mechanism applies equally to word and character models. The main difference is that the vocabulary— the set of symbols that make

up the corpus and the model—is larger. There are only about 100 characters in most languages, and sometimes we build character models that are even more restrictive, for example by treating "A" and "a" as the same symbol or by treating all punctuation as the same symbol. But with word models we have at least tens of thousands of symbols, and sometimes millions. The wide range is because it is not clear what constitutes a word. In English a sequence of letters surrounded by spaces is a word, but in some languages, like Chinese, words are not separated by spaces, and even in English many decisions must be made to have a clear policy on word boundaries: how many words are in "ne'er-do-well"? Or in "(Tel:1-800-960-5660x123)"? Word n-gram models need to deal with out of vocabulary words. With character models, we didn't have to worry about someone inventing a new letter of the alphabet.1 But with word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model. This can be done by adding just one new word to the vocabulary: , standing for the unknown word. We can estimate n-gram counts for by this trick: go through the training corpus, and the first time any individual word appears it is previously unknown, so replace it with the symbol . All subsequent appearances of the word remain unchanged. Then compute n-gram counts for the corpus as usual, treating just like any other word. Then when an unknown word appears in a test set, we look up its probability under . Sometimes multiple unknown-word symbols are used, for different classes. For example, any string of digits might be replaced with , or any email address with . To get a feeling for what word models can do, we built unigram, bigram, and trigram models over the words in this book and then randomly sampled sequences of words from the models. The results are
Unigram: logical are as are confusion a may right tries agent goal the was ...
Bigram: systems are very similar computational approach would be represented ...
Trigram: planning and scheduling are integrated the success of naive bayes model is ...
Even with this small sample, it should be clear that the unigram model is a poor approximation of either English or the content of an AI textbook, and that the bigram and trigram models are much better. The models agree with this assessment: the perplexity was 891 for the unigram model, 142 for the bigram model and 91 for the trigram model. With the basics of n-gram models—both character- and word-based—established, we can turn now to some language tasks

TEXT CLASSIFICATION

We now consider in depth the task of text classification, also known as categorization: given a text of some kind, decide which of a predefined set of classes it belongs to. Language identification and genre classification are examples of text classification, as is sentiment analysis (classifying a movie or product review as positive or negative) and spam detection (classifying an email message as spam or not-spam). Since "not-spam" is awkward, researchers have coined the term ham for not-spam. We can treat spam detection as a problem in supervised learning. A training set is readily available: the positive (spam) examples are in my spam folder, the negative (ham) examples are in my inbox. Here is an excerpt:

Spam: Wholesale Fashion Watches -57% today. Designer watches for cheap ...
Spam: You can buy ViagraFr$1.85 All Medications at unbeatable prices! ...
Spam: WE CAN TREAT ANYTHING YOU SUFFER FROM JUST TRUST US ...
Spam: Sta.rt earn*ing the salary yo,u d-eserve by o'btaining the prope,r crede'ntials!
Ham: The practical significance of hypertree width in identifying more ...
Ham: Abstract: We will motivate the problem of social identity clustering: ...
Ham: Good to see you my friend. Hey Peter, It was good to hear from you. ...
Ham: PDS implies convexity of the resulting optimization problem (Kernel Ridge ...

From this excerpt we can start to get an idea of what might be good features to include in the supervised learning model. Word n-grams such as "for cheap" and "You can buy" seem to be indicators of

spam (although they would have a nonzero probability in ham as well). Character-level features also seem important: spam is more likely to be all uppercase and to have punctuation embedded in words. Apparently the spammers thought that the word bigram "you deserve" would be too indicative of spam, and thus wrote "yo,u d-eserve" instead. A character model should detect this. We could either create a full character n-gram model of spam and ham, or we could handcraft features such as "number of punctuation marks embedded in words." Note that we have two complementary ways of talking about classification. In the language-modeling approach, we define one n-gram language model for P(Message | spam) by training on the spam folder, and one model for P(Message | ham) by training on the inbox. Then we can classify a new message with an application of Bayes' rule:

$$\underset{c \in \{spam, ham\}}{\operatorname{argmax}} P(c \mid message) = \underset{c \in \{spam, ham\}}{\operatorname{argmax}} P(message \mid c)\, P(c)\ .$$

where P(c) is estimated just by counting the total number of spam and ham messages. This approach works well for spam detection, just as it did for language identification.

In the machine-learning approach we represent the message as a set of feature/value pairs and apply a classification algorithm h to the feature vector X. We can make the language-modeling and machine-learning approaches compatible by thinking of the n-grams as features. This is easiest to see with a unigram model. The features are the words in the vocabulary: "a," "aardvark," ..., and the values are the number of times each word appears in the message. That makes the feature vector large and sparse. If there are 100,000 words in the language model, then the feature vector has length 100,000, but for a short email message almost all the features will have count zero. This unigram representation has been called the bag of words model. You can think of the model as putting the words of the training corpus in a bag and then selecting words one at a time. The notion of order of the words is lost; a unigram model gives the same probability to any permutation of a text. Higher-order n-gram models maintain some local notion of word order.

With bigrams and trigrams the number of features is squared or cubed, and we can add in other, non-n-gram features: the time the message was sent, whether a URL or an image is part of the message, an ID number for the sender of the message, the sender's number of previous spam and ham messages, and so on. The choice of features is the most important part of creating a good spam detector—more important than the choice of algorithm for processing the features. In part this is because there is a lot of training data, so if we can propose a feature, the data can accurately determine if it is good or not. It is necessary to constantly update features, because spam detection is an adversarial task; the spammers modify their spam in response to the spam detector's changes.

It can be expensive to run algorithms on a very large feature vector, so often a process of feature selection is used to keep only the features that best discriminate between spam and ham. For example, the bigram "of the" is frequent in English, and may be equally frequent in spam and ham, so there is no sense in counting it. Often the top hundred or so features do a good job of discriminating between classes.

Once we have chosen a set of features, we can apply any of the supervised learning techniques we have seen; popular ones for text categorization include k-nearest-neighbors, support vector machines, decision trees, naive Bayes, and logistic regression. All of these have been applied to spam detection, usually with accuracy in the 98%–99% range. With a carefully designed feature set, accuracy can exceed 99.9%.

Classification by data compression

Another way to think about classification is as a problem in data compression. A lossless compression algorithm takes a sequence of symbols, detects repeated patterns in it, and writes a description of the sequence that is more compact than the original. For example, the text "0.142857142857142857" might be compressed to "0.[142857]*3." Compression algorithms work by building dictionaries of subsequences of

the text, and then referring to entries in the dictionary. The example here had only one dictionary entry, "142857."
In effect, compression algorithms are creating a language model. The LZW algorithm in particular directly models a maximum-entropy probability distribution. To do classification by compression, we first lump together all the spam training messages and compress them as a unit. We do the same for the ham. Then when given a new message to classify, we append it to the spam messages and compress the result. We also append it to the ham and compress that. Whichever class compresses better—adds the fewer number of additional bytes for the new message—is the predicted class. The idea is that a spam message will tend to share dictionary entries with other spam messages and thus will compress better when appended to a collection that already contains the spam dictionary.
Experiments with compression-based classification on some of the standard corpora for text classification—the 20-Newsgroups data set, the Reuters-10 Corpora, the Industry Sector corpora—indicate that whereas running off-the-shelf compression algorithms like gzip, RAR, and LZW can be quite slow, their accuracy is comparable to traditional classification algorithms. This is interesting in its own right, and also serves to point out that there is promise for algorithms that use character n-grams directly with no preprocessing of the text or feature selection: they seem to be captiring some real patterns.

INFORMATION RETRIEVAL

Information retrieval is the task of finding documents that are relevant to a user's need for information. The best-known examples of information retrieval systems are search engines on the World Wide Web. A Web user can type a query such as [AI book]2 into a search engine and see a list of relevant pages. In this section, we will see how such systems are built. An IR information retrieval (henceforth IR) system can be characterized by
1. A corpus of documents. Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.
2. Queries posed in a query language. A query specifies what the user wants to know. The query language can be just a list of words, or it can specify a phrase of words that must be adjacent, it can contain Boolean operators as in it can include non-Boolean operators
3. A result set. This is the subset of documents that the IR system judges to be relevant to RELEVANT the query. By relevant, we mean likely to be of use to the person who posed the query, for the particular information need expressed in the query.
4. A presentation of the result set. This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three dimensional space, rendered as a two-dimensional display.

The earliest IR systems worked on a Boolean keyword model. Each word in the document collection is treated as a Boolean feature that is true of a document if the word occurs in the document and false if it does not. So the feature "retrieval" is true for the current chapter but false. The query language is the language of Boolean expressions over features. A document is relevant only if the expression evaluates to true. For example, the query [information AND retrieval] is true for the current chapter and false.
This model has the advantage of being simple to explain and implement. However, it has some disadvantages. First, the degree of relevance of a document is a single bit, so there is no guidance as to how to order the relevant documents for presentation. Second, Boolean expressions are unfamiliar to users who are not programmers or logicians. Users find it unintuitive that when they want to know about farming in the states of Kansas and Nebraska they need to issue the query [farming (Kansas OR Nebraska)]. Third, it can be hard to formulate an appropriate query, even for a skilled user. Suppose we try [information AND retrieval AND models AND optimization] and get an empty result set. We could try [information OR retrieval OR models OR optimization], but if that returns too many results, it is difficult to know what to

try next.

IR scoring functions

Most IR systems have abandoned the Boolean model and use models based on the statistics of word counts. We describe the BM25 scoring function, which comes from the Okapi project of Stephen Robertson and Karen Sparck Jones at London's City College, and has been used in search engines such as the open-source Lucene project.

A scoring function takes a document and a query and returns a numeric score; the most relevant documents have the highest scores. In the BM25 function, the score is a linear weighted combination of scores for each of the words that make up the query. Three factors affect the weight of a query term: First, the frequency with which a query term appears in a document (also known as TF for term frequency). For the query [farming in Kansas], documents that mention "farming" frequently will have higher scores. Second, the inverse document frequency of the term, or IDF. The word "in" appears in almost every document, so it has a high document frequency, and thus a low inverse document frequency, and thus it is not as important to the query as "farming" or "Kansas." Third, the length of the document. A million-word document will probably mention all the query words, but may not actually be about the query. A short document that mentions all the words is a much better candidate.

The BM25 function takes all three of these into account. We assume we have created an index of the N documents in the corpus so that we can look up TF(qi, dj), the count of the number of times word qi appears in document dj . We also assume a table of document frequency counts, DF(qi), that gives the number of documents that contain the word qi. Then, given a document dj and a query consisting of the words q1:N , we have

$$BM25(d_j, q_{1:N}) = \sum_{i=1}^{N} IDF(q_i) \cdot \frac{TF(q_i, d_j) \cdot (k+1)}{TF(q_i, d_j) + k \cdot (1 - b + b \cdot \frac{|d_j|}{L})} ,$$

where |dj | is the length of document dj in words, and L is the average document length in the corpus: L = i |di|/N. We have two parameters, k and b, that can be tuned by cross-validation; typical values are k = 2.0 and b = 0.75. IDF(qi) is the inverse document frequency of word qi, given by

$$IDF(q_i) = \log \frac{N - DF(q_i) + 0.5}{DF(q_i) + 0.5} .$$

Of course, it would be impractical to apply the BM25 scoring function to every document in the corpus. Instead, systems create an index ahead of time that lists, for each vocabulary word, the documents that contain the word. This is called the hit list for the word. Then when given a query, we intersect the hit lists of the query words and only score the documents in the intersection.

IR system evaluation

How do we know whether an IR system is performing well? We undertake an experiment in which the system is given a set of queries and the result sets are scored with respect to human relevance judgments. Traditionally, there have been two measures used in the scoring: recall and precision. We explain them with the help of an example. Imagine that an IR system has returned a result set for a single query, for which we know which documents are and are not relevant, out of a corpus of 100 documents. The document counts in each category are given in the following table:

	In result set	Not in result set
Relevant	30	20
Not relevant	10	40

Precision measures the proportion of documents in the result set that are actually relevant. In our example, the precision is 30/(30 + 10) = .75. The false positive rate is 1 − .75 = .25. RECALL Recall measures the proportion of all the relevant documents in the collection that are in the result set. In our example, recall is 30/(30 + 20) = .60. The false negative rate is 1 − .60 = .40. In a very large document collection, such as the World Wide Web, recall is difficult to compute, because there is no easy way to examine every page on the Web for relevance. All we can do is either estimate recall by sampling or ignore recall completely and just judge precision. In the case of a Web search engine, there may be thousands of documents in the result set, so it makes more sense to measure precision for several different sizes, such as "P@10" (precision in the top 10 results) or "P@50," rather than to estimate precision in the entire result set. It is possible to trade off precision against recall by varying the size of the result set returned. In the extreme, a system that returns every document in the document collection is guaranteed a recall of 100%, but will have low precision. Alternately, a system could return a single document and have low recall, but a decent chance at 100% precision. A summary of both measures is the F1 score, a single number that is the harmonic mean of precision and recall, 2P R/(P + R).

IR refinements

There are many possible refinements to the system described here, and indeed Web search engines are continually updating their algorithms as they discover new approaches and as the Web grows and changes. One common refinement is a better model of the effect of document length on relevance. Singhal et al. (1996) observed that simple document length normalization schemes tend to favor short documents too much and long documents not enough. They propose a pivoted document length normalization scheme; the idea is that the pivot is the document length at which the old-style normalization is correct; documents shorter than that get a boost and longer ones get a penalty.

The BM25 scoring function uses a word model that treats all words as completely independent, but we know that some words are correlated: "couch" is closely related to both "couches" and "sofa." Many IR systems attempt to account for these correlations. For example, if the query is [couch], it would be a shame to exclude from the result set those documents that mention "COUCH" or "couches" but not "couch." Most IR systems CASE FOLDING do case folding of "COUCH" to "couch," and some use a stemming algorithm to reduce STEMMING "couches" to the stem form "couch," both in the query and the documents. This typically yields a small increase in recall (on the order of 2% for English). However, it can harm precision.

For example, stemming "stocking" to "stock" will tend to decrease precision for queries about either foot coverings or financial instruments, although it could improve recall for queries about warehousing. Stemming algorithms based on rules (e.g., remove "-ing") cannot avoid this problem, but algorithms based on dictionaries (don't remove "-ing" if the word is already listed in the dictionary) can. While stemming has a small effect in English, it is more important in other languages. In German, for example, it is not uncommon to see words like "Lebensversicherungsgesellschaftsangestellter" (life insurance company employee). Languages such as Finnish, Turkish, Inuit, and Yupik have recursive morphological rules that in principle generate words of unbounded length.

The next step is to recognize synonyms, such as "sofa" for "couch." As with stemming, this has the potential for small gains in recall, but can hurt precision. A user who gives the query [Tim Couch] wants to see results about the football player, not sofas. The problem is that "languages abhor absolute synonyms just as nature abhors a vacuum" (Cruse, 1986). That is, anytime there are two words that mean the same thing, speakers of the language conspire to evolve the meanings to remove the confusion. Related words

that are not synonyms also play an important role in ranking—terms like “leather”, “wooden,” or “modern” can serve to confirm that the document really is about “couch.” Synonyms and related words can be found in dictionaries or by looking for correlations in documents or in queries—if we find that many users who ask the query [new sofa] follow it up with the query [new couch], we can in the future alter [new sofa] to be [new sofa OR new couch].

As a final refinement, IR can be improved by considering metadata—data outside of the text of the document. Examples include human-supplied keywords and publication data. LINKS On the Web, hypertext links between documents are a crucial source of information.

The PageRank algorithm

PageRank was one of the two original ideas that set Google’s search apart from other Web search engines when it was introduced in 1997. (The other innovation was the use of anchor

```
function HITS(query) returns pages with hub and authority numbers
  pages ← EXPAND-PAGES(RELEVANT-PAGES(query))
  for each p in pages do
    p.AUTHORITY ← 1
    p.HUB ← 1
  repeat until convergence do
    for each p in pages do
      p.AUTHORITY ← ∑_i INLINK_i(p).HUB
      p.HUB ← ∑_i OUTLINK_i(p).AUTHORITY
    NORMALIZE(pages)
  return pages
```

Figure 22.1 The HITS algorithm for computing hubs and authorities with respect to a query. RELEVANT-PAGES fetches the pages that match the query, and EXPAND-PAGES adds in every page that links to or is linked from one of the relevant pages. NORMALIZE divides each page’s score by the sum of the squares of all pages’ scores (separately for both the authority and hubs scores).

text—the underlined text in a hyperlink—to index a page, even though the anchor text was on a different page than the one being indexed.) PageRank was invented to solve the problem of the tyranny of TF scores: how do we make sure that IBM’s home page, ibm.com, is the first result, even if another page mentions the term “IBM” more frequently? The idea is that ibm.com has many in-links (links to the page), so it should be ranked higher: each in-link is a vote for the quality of the linked-to page. But if we only counted in-links, then it would be possible for a Web spammer to create a network of pages and have them all point to a page of his choosing, increasing the score of that page. Therefore, the PageRank algorithm is designed to weight links from high-quality sites more heavily. What is a highquality site? One that is linked to by other high-quality sites. The definition is recursive, but we will see that the recursion bottoms out properly. The PageRank for a page p is defined as:

$$PR(p) = \frac{1-d}{N} + d\sum_i \frac{PR(in_i)}{C(in_i)},$$

where P R(p) is the PageRank of page p, N is the total number of pages in the corpus, ini are the pages that link in to p, and C(ini) is the count of the total number of out-links on page ini. The constant d is a damping factor. It can be understood through the random surfer model: imagine a Web surfer who starts at some random page and begins exploring. RANDOM SURFER MODEL With probability d (we’ll assume d = 0.85) the surfer clicks on one of the links on the page (choosing uniformly among them), and with probability 1 − d she gets bored with the page and restarts on a random page anywhere on the Web. The PageRank of page p is then the probability that the random surfer will be at page p at any point in time. PageRank can be computed by an iterative procedure: start with all pages having P R(p)=1, and iterate the algorithm, updating ranks until they converge.

The HITS algorithm

The Hyperlink-Induced Topic Search algorithm, also known as "Hubs and Authorities" or HITS, is another influential link-analysis algorithm (see Figure 22.1). HITS differs from PageRank in several ways. First, it is a query-dependent measure: it rates pages with respect to a query. That means that it must be computed anew for each query—a computational burden that most search engines have elected not to take on. Given a query, HITS first finds a set of pages that are relevant to the query. It does that by intersecting hit lists of query words, and then adding pages in the link neighborhood of these pages—pages that link to or are linked from one of the pages in the original relevant set.

Each page in this set is considered an authority on the query to the degree that other HUB pages in the relevant set point to it. A page is considered a hub to the degree that it points to other authoritative pages in the relevant set. Just as with PageRank, we don't want to merely count the number of links; we want to give more value to the high-quality hubs and authorities. Thus, as with PageRank, we iterate a process that updates the authority score of a page to be the sum of the hub scores of the pages that point to it, and the hub score to be the sum of the authority scores of the pages it points to. If we then normalize the scores and repeat k times, the process will converge.

Both PageRank and HITS played important roles in developing our understanding of Web information retrieval. These algorithms and their extensions are used in ranking billions of queries daily as search engines steadily develop better ways of extracting yet finer signals of search relevance.

Question answering

Information retrieval is the task of finding documents that are relevant to a query, where the query may be a question, or just a topic area or concept. Question answering is a somewhat different task, in which the query really is a question, and the answer is not a ranked list of documents but rather a short response—a sentence, or even just a phrase. There have been question-answering NLP (natural language processing) systems since the 1960s, but only since 2001 have such systems used Web information retrieval to radically increase their breadth of coverage.

ASKMSR does not attempt this kind of sophistication—it knows nothing about pronoun reference, or about killing, or any other verb. It does know 15 different kinds of questions, and how they can be rewritten as queries to a search engine. It knows that [Who killed Abraham Lincoln] can be rewritten as the query [* killed Abraham Lincoln] and as [Abraham Lincoln was killed by *]. It issues these rewritten queries and examines the results that come back— not the full Web pages, just the short summaries of text that appear near the query terms. The results are broken into 1-, 2-, and 3-grams and tallied for frequency in the result sets and for weight: an n-gram that came back from a very specific query rewrite (such as the exact phrase match query ["Abraham Lincoln was killed by *"]) would get more weight than one from a general query rewrite, such as [Abraham OR Lincoln OR killed]. We would expect that "John Wilkes Booth" would be among the highly ranked n-grams retrieved, but so would "Abraham Lincoln" and "the assassination of" and "Ford's Theatre."

Once the n-grams are scored, they are filtered by expected type. If the original query starts with "who," then we filter on names of people; for "how many" we filter on numbers, for "when," on a date or time. There is also a filter that says the answer should not be part of the question; together these should allow us to return "John Wilkes Booth" (and not "Abraham Lincoln") as the highest-scoring response.

In some cases the answer will be longer than three words; since the components responses only go up to 3-grams, a longer response would have to be pieced together from shorter pieces. For example, in a system that used only bigrams, the answer "John Wilkes Booth" could be pieced together from high-scoring pieces "John Wilkes" and "Wilkes Booth."

At the Text Retrieval Evaluation Conference (TREC), ASKMSR was rated as one of the top systems,

beating out competitors with the ability to do far more complex language understanding. ASKMSR relies upon the breadth of the content on the Web rather than on its own depth of understanding. It won't be able to handle complex inference patterns like associating "who killed" with "ended the life of." But it knows that the Web is so vast that it can afford to ignore passages like that and wait for a simple passage it can handle.

INFORMATION EXTRACTION

Information extraction is the process of acquiring knowledge by skimming a text and look ing for occurrences of a particular class of object and for relationships among objects. A typical task is to extract instances of addresses from Web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation. In a limited domain, this can be done with high accuracy. As the domain gets more general, more complex linguistic models and more complex learning techniques are necessary. We will see in Chapter 23 how to define complex language models of the phrase structure (noun phrases and verb phrases) of English. But so far there are no complete models of this kind, so for the limited needs of information extraction, we define limited models that approximate the full English model, and concentrate on just the parts that are needed for the task at hand. The models we describe in this section are approximations in the same way that the simple 1-CNF logical model

Finite-state automata for information extraction

The simplest type of information extraction system is an attribute-based extraction system that assumes that the entire text refers to a single object and the task is to extract attributes of that object. For example, we mentioned in Section 12.7 the problem of extracting from the text "IBM ThinkBook 970. Our price: $399.00" the set of attributes {Manufacturer=IBM, Model=ThinkBook970, Price=$399.00}. We can address this problem by defining a temTEMPLATE plate (also known as a pattern) for each attribute we would like to extract. The template is defined by a finite state automaton, the simplest example of which is the regular expression, REGULAR EXPRESSION or regex. Regular expressions are used in Unix commands such as grep, in programming languages such as Perl, and in word processors such as Microsoft Word. The details vary slightly from one tool to another and so are best learned from the appropriate manual, but here we show how to build up a regular expression template for prices in dollars:

[0-9] matches any digit from 0 to 9
[0-9]+ matches one or more digits
[.][0-9][0-9] matches a period followed by two digits
([.][0-9][0-9])? matches a period followed by two digits,
or nothing [$][0-9]+([.][0-9][0-9])? matches $249.99 or $1.23 or $1000000 or . . .
Templates are often defined with three parts: a prefix regex, a target regex, and a postfix regex. For prices, the target regex is as above, the prefix would look for strings such as "price:" and the postfix could be empty. The idea is that some clues about an attribute come from the attribute value itself and some come from the surrounding text.

If a regular expression for an attribute matches the text exactly once, then we can pull out the portion of the text that is the value of the attribute. If there is no match, all we can do is give a default value or leave the attribute missing; but if there are several matches, we need a process to choose among them. One strategy is to have several templates for each attribute, ordered by priority. So, for example, the top-priority template for price might look for the prefix "our price:"; if that is not found, we look for the prefix "price:" and if that is not found, the empty prefix. Another strategy is to take all the matches and find some way to choose among them. For example, we could take the lowest price that is within 50% of the highest price.

That will select $78.00 as the target from the text "List price $99.00, special sale price $78.00, shipping $3.00."

One step up from attribute-based extraction systems are relational extraction systems, which deal with multiple objects and the relations among them. Thus, when these systems see the text "$249.99," they need to determine not just that it is a price, but also which object has that price. A typical relational-based extraction system is FASTUS, which handles news stories about corporate mergers and acquisitions. It can read the story Bridgestone Sports Co. said Friday it has set up a joint venture in Taiwan with a local concern and a Japanese trading house to produce golf clubs to be shipped to Japan. and extract the relations:

e∈ JointVentures ∧ Product(e, "golf clubs") ∧ Date(e, "F riday")
∧ Member (e, "Bridgestone Sports Co") ∧ Member (e, "a local concern")
∧ Member (e, "a Japanese trading house") .

A relational extraction system can be built as a series of cascaded finite-state transducers.
That is, the system consists of a series of small, efficient finite-state automata (FSAs), where each automaton receives text as input, transduces the text into a different format, and passes it along to the next automaton. FASTUS consists of five stages:

1. Tokenization
2. Complex-word handling
3. Basic-group handling
4. Complex-phrase handling
5. Structure merging

FASTUS's first stage is tokenization, which segments the stream of characters into tokens (words, numbers, and punctuation). For English, tokenization can be fairly simple; just separating characters at white space or punctuation does a fairly good job. Some tokenizers also deal with markup languages such as HTML, SGML, and XML.

The second stage handles complex words, including collocations such as "set up" and "joint venture," as well as proper names such as "Bridgestone Sports Co." These are recognized by a combination of lexical entries and finite-state grammar rules. For example, a company name might be recognized by the rule CapitalizedWord+ ("Company" | "Co" | "Inc" | "Ltd")

The third stage handles basic groups, meaning noun groups and verb groups. The idea is to chunk these into units that will be managed by the later stages. We will see how to write a complex description of noun and verb phrases in Chapter 23, but here we have simple rules that only approximate the complexity of English, but have the advantage of being representable by finite state automata. The example sentence would emerge from this stage as the following sequence of tagged groups:

1 NG: Bridgestone Sports Co.	10 NG: a local concern
2 VG: said	11 CJ:
3 NG: Friday	12 NG: a Japanese trading house
4 NG: it	13 VG: to produce
5 VG: had set up	14 NG: golf clubs
6 NG: a joint venture	15 VG: to be shipped
7 PR: in	16 PR: to
8 NG: Taiwan	17 NG: Japan
9 PR: with	

Here NG means noun group, VG is verb group, PR is preposition, and CJ is conjunction.

The fourth stage combines the basic groups into complex phrases. Again, the aim is to have rules that are finite-state and thus can be processed quickly, and that result in unambiguous (or nearly unambiguous)

output phrases. One type of combination rule deals with domain-specific events.
Company+ SetUp JointVenture (“with” Company+)?

For example, the rule captures one way to describe the formation of a joint venture. This stage is the first one in the cascade where the output is placed into a database template as well as being placed in the output stream. The final stage merges structures that were built up in the previous step. If the next sentence says “The joint venture will start production in January,” then this step will notice that there are two references to a joint venture, and that they should be merged into one. This is an instance of the identity uncertainty problem.
In general, finite-state template-based information extraction works well for a restricted domain in which it is possible to predetermine what subjects will be discussed, and how they will be mentioned. The cascaded transducer model helps modularize the necessary knowledge, easing construction of the system. These systems work especially well when they are reverse-engineering text that has been generated by a program. For example, a shopping site on the Web is generated by a program that takes database entries and formats them into Web pages; a template-based extractor then recovers the original database. Finite-state information extraction is less successful at recovering information in highly variable format, such as text written by humans on a variety of subjects.

Probabilistic models for information extraction

When information extraction must be attempted from noisy or varied input, simple finite-state approaches fare poorly. It is too hard to get all the rules and their priorities right; it is better to use a probabilistic model rather than a rule-based model. The simplest probabilistic model for sequences with hidden state is the hidden Markov model, or HMM.
HMM models a progression through a sequence of hidden states, xt, with an observation et at each step. To apply HMMs to information extraction, we can either build one big HMM for all the attributes or build a separate HMM for each attribute. We’ll do the second. The observations are the words of the text, and the hidden states are whether we are in the target, prefix, or postfix part of the attribute template, or in the background (not part of a template). For example, here is a brief text and the most probable (Viterbi) path for that text for two HMMs, one trained to recognize the speaker in a talk announcement, and one trained to recognize dates. The “-” indicates a background state:
Text: There will be a seminar by Dr. Andrew McCallum on Friday
Speaker: - - - - PRE PRE TARGET TARGET TARGET POST –
Date: - - - - - - - - - PRE TARGET
HMMs have two big advantages over FSAs for extraction. First, HMMs are probabilistic, and thus tolerant to noise. In a regular expression, if a single expected character is missing, the regex fails to match; with HMMs there is graceful degradation with missing characters/words, and we get a probability indicating the degree of match, not just a Boolean match/fail.

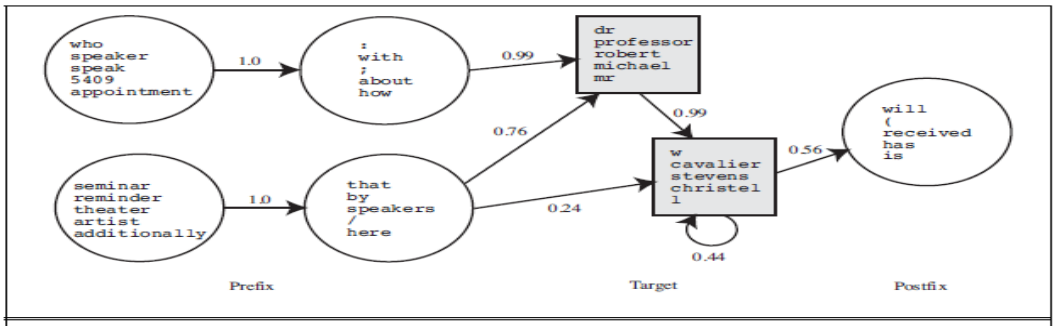


Figure 22.2 Hidden Markov model for the **speaker** of a talk announcement. The two square states are the target (note the second target state has a self-loop, so the target can match a string of any length), the four circles to the left are the prefix, and the one on the right is the postfix. For each state, only a few of the high-probability words are shown. From Freitag and McCallum (2000).

Second, HMMs can be trained from data; they don't require laborious engineering of templates, and thus they can more easily be kept up to date as text changes over time.
Note that we have assumed a certain level of structure in our HMM templates: they all consist of one or more target states, and any prefix states must precede the targets, postfix states most follow the targets, and other states must be background. This structure makes it easier to learn HMMs from examples. With a partially specified structure, the forward backward algorithm can be used to learn both the transition probabilities **P**(**X**t |**X**t−1) between states and the observation model, **P**(**E**t |**X**t), which says how likely each word is in each state. For example, the word "Friday" would have high probability in one or more of the target states of the date HMM, and lower probability elsewhere.

With sufficient training data, the HMM automatically learns a structure of dates that we find intuitive: the date HMM might have one target state in which the high-probability words are "Monday," "Tuesday," etc., and which has a high-probability transition to a target state with words "Jan", "January," "Feb," etc. Figure 22.2 shows the HMM for the speaker of a talk announcement, as learned from data. The prefix covers expressions such as "Speaker:" and "seminar by," and the target has one state that covers titles and first names and another state that covers initials and last names.
Once the HMMs have been learned, we can apply them to a text, using the Viterbi algorithm to find the most likely path through the HMM states. One approach is to apply each attribute HMM separately; in this case you would expect most of the HMMs to spend most of their time in background states. This is appropriate when the extraction is sparse when the number of extracted words is small compared to the length of the text.

The other approach is to combine all the individual attributes into one big HMM, which would then find a path that wanders through different target attributes, first finding a speaker target, then a date target, etc. Separate HMMs are better when we expect just one of each attribute in a text and one big HMM is better when the texts are more free-form and dense with attributes. With either approach, in the end we have a collection of target attribute
observations, and have to decide what to do with them. If every expected attribute has one target filler then the decision is easy: we have an instance of the desired relation. If there are multiple fillers, we need to decide which to choose, as we discussed with template-based systems. HMMs have the advantage of

supplying probability numbers that can help make the choice. If some targets are missing, we need to decide if this is an instance of the desired
relation at all, or if the targets found are false positives. A machine learning algorithm can be trained to make this choice.

Conditional random fields for information extraction

One issue with HMMs for the information extraction task is that they model a lot of probabilities that we don't really need. An HMM is a generative model; it models the full joint probability of observations and hidden states, and thus can be used to generate samples. That is, we can use the HMM model not only to parse a text and recover the speaker and date, but also to generate a random instance of a text containing a speaker and a date. Since we're not interested in that task, it is natural to ask whether we might be better off with a model that doesn't bother modeling that possibility. All we need in order to understand a text is a **discriminative model**, one that models the conditional probability of the hidden attributes given the observations (the text). Given a text **e**1:N, the conditional model finds the hidden state sequence **X**1:N that maximizes P(**X**1:N | **e**1:N).
Modeling this directly gives us some freedom. We don't need the independence assumptions of the Markov model—we can have an **x**t that is dependent on **x**1. A framework for this type of model is the **conditional random** CONDITIONAL **field**, or CRF, which models a conditional probability distribution of a set of target variables given a set of observed variables. Like Bayesian networks, CRFs can represent many different structures of dependencies among the
variables. One common structure is the **linear-chain conditional random field** for representing Markov dependencies among variables in a temporal sequence. Thus, HMMs are the temporal version of naive Bayes models, and linear-chain CRFs are the temporal version of logistic regression, where the predicted target is an entire state sequence rather than a single binary variable.

Let **e**1:N be the observations (e.g., words in a document), and **x**1:N be the sequence of hidden states (e.g., the prefix, target, and postfix states). A linear-chain conditional random field defines a conditional probability distribution:

$$\mathbf{P}(\mathbf{x}_{1:N}|\mathbf{e}_{1:N}) = \alpha\, e^{\left[\sum_{i=1}^{N} F(\mathbf{x}_{i-1},\mathbf{x}_i,\mathbf{e},i)\right]} ,$$

where α is a normalization factor (to make sure the probabilities sum to 1), and F is a feature function defined as the weighted sum of a collection of k component feature functions:

$$F(\mathbf{x}_{i-1},\mathbf{x}_i,\mathbf{e},i) = \sum_k \lambda_k\, f_k(\mathbf{x}_{i-1},\mathbf{x}_i,\mathbf{e},i) .$$

The λk parameter values are learned with a MAP (maximum a posteriori) estimation procedure that maximizes the conditional likelihood of the training data. The feature functions are the key components of a CRF. The function fk has access to a pair of adjacent states, **x**i−1 and **x**i, but also the entire observation (word) sequence **e**, and the current position in the temporal sequence, i. This gives us a lot of flexibility in defining features. We can define a simple
feature function, for example one that produces a value of 1 if the current word is and the current state is SPEAKER:

$$f_1(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i) = \begin{cases} 1 & \text{if } \mathbf{x}_i = \text{SPEAKER and } \mathbf{e}_i = \text{ANDREW} \\ 0 & \text{otherwise} \end{cases}$$

How are features like these used? It depends on their corresponding weights. If $\lambda 1 > 0$, then whenever f1 is true, it increases the probability of the hidden state sequence **x**1:N . This is another way of saying "the CRF model should prefer the target state SPEAKER for the word ANDREW." If on the other hand $\lambda 1 < 0$, the CRF model will try to avoid this association, and if $\lambda 1 = 0$, this feature is ignored. Parameter values can be set manually or can be learned
from data. Now consider a second feature function:

$$f_2(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i) = \begin{cases} 1 & \text{if } \mathbf{x}_i = \text{SPEAKER and } \mathbf{e}_{i+1} = \text{SAID} \\ 0 & \text{otherwise} \end{cases}$$

This feature is true if the current state is SPEAKER and the next word is "said." One would therefore expect a positive $\lambda 2$ value to go with the feature. More interestingly, note that both f1 and f2 can hold at the same time for a sentence like "Andrew said . . . ." In this case, the two features overlap each other and both boost the belief in **x**1 = SPEAKER. Because of the independence assumption, HMMs cannot use overlapping features; CRFs can. Furthermore,
a feature in a CRF can use any part of the sequence **e**1:N. Features can also be defined over transitions between states. The features we defined here were binary, but in general, a feature function can be any real-valued function. For domains where we have some knowledge about the types of features we would like to include, the CRF formalism gives us a great deal of flexibility in defining them. This flexibility can lead to accuracies that are higher than with
less flexible models such as HMMs.

Ontology extraction from large corpora

So far we have thought of information extraction as finding a specific set of relations (e.g., speaker, time, location) in a specific text (e.g., a talk announcement). A different application of extraction technology is building a large knowledge base or ontology of facts from a corpus. This is different in three ways: First it is open-ended—we want to acquire facts about all types of domains, not just one specific domain. Second, with a large corpus, this
task is dominated by precision, not recall—just as with question answering on the Web. Third, the results can be statistical aggregates gathered from multiple sources, rather than being extracted from one specific text.
For example, Hearst (1992) looked at the problem of learning an ontology of conceptcategories and subcategories from a large corpus. (In 1992, a large corpus was a 1000-page encyclopedia; today it would be a 100-million-page Web corpus.) The work concentrated on templates that are very general (not tied to a specific domain) and have high precision (are almost always correct when they match) but low recall (do not always match). Here is one of
the most productive templates:

NP **such as** NP (**,** NP)* (**,**)? ((**and** | **or**) NP)? .

Here the bold words and commas must appear literally in the text, but the parentheses are for grouping, the asterisk means *repetition of zero or more*, and the question mark means *optional.* NP is a variable standing for a noun phrase; Chapter 23 describes how to identify noun phrases; for now just assume that we know

some words are nouns and other words (such as verbs) that we can reliably assume are not part of a simple noun phrase. This template
matches the texts “diseases such as rabies affect your dog” and “supports network protocols such as DNS,” concluding that rabies is a disease and DNS is a network protocol. Similar templates can be constructed with the key words “including,” “especially,” and “or other.” Of course these templates will fail to match many relevant passages, like “Rabies is a disease.” That is intentional. The “NP is a NP” template does indeed sometimes denote a subcategory relation, but it often means something else, as in “There is a God” or “She is a little tired.”
With a large corpus we can afford to be picky; to use only the high-precision templates. We’ll miss many statements of a subcategory relationship, but most likely we’ll find a paraphrase of the statement somewhere else in the corpus in a form we can use.

Automated template construction

The *subcategory* relation is so fundamental that is worthwhile to handcraft a few templates to help identify instances of it occurring in natural language text. But what about the thousands of other relations in the world? There aren’t enough AI grad students in the world to create and debug templates for all of them. Fortunately, it is possible to *learn* templates from a few examples, then use the templates to learn more examples, from which more templates can be
learned, and so on. In one of the first experiments of this kind, Brin (1999) started with a data set of just five examples:
(“Isaac Asimov”, “The Robots of Dawn”)
(“David Brin”, “Startide Rising”)
(“James Gleick”, “Chaos—Making a New Science”)
(“Charles Dickens”, “Great Expectations”)
(“William Shakespeare”, “The Comedy of Errors”)
Clearly these are examples of the author–title relation, but the learning system had no knowledge of authors or titles. The words in these examples were used in a search over a Web corpus, resulting in 199 matches. Each match is defined as a tuple of seven strings,
(*Author, Title, Order, Prefix, Middle, Postfix, URL*) ,
where *Order* is true if the author came first and false if the title came first, *Middle* is the characters between the author and title, *Prefix* is the 10 characters before the match, *Suffix* is the 10 characters after the match, and *URL* is the Web address where the match was made.
Given a set of matches, a simple template-generation scheme can find templates to explain the matches. The language of templates was designed to have a close mapping to the matches themselves, to be amenable to automated learning, and to emphasize high precision
Each template has the same seven components as a
match. The Author and Title are regexes consisting of any characters (but beginning and ending in letters) and constrained to have a length from half the minimum length of the examples to twice the maximum length. The prefix, middle, and postfix are restricted to literal strings, not regexes. The middle is the easiest to learn: each distinct middle string in the set of matches is a distinct candidate template. For each such candidate, the template’s
Prefix is then defined as the longest common suffix of all the prefixes in the matches, and the *Postfix* is defined as the longest common prefix of all the postfixes in the matches. If either of these is of length zero, then the template is rejected. The *URL* of the template is defined as the longest prefix of the URLs in the matches.

In the experiment run by Brin, the first 199 matches generated three templates. The most productive template was
<LI><B> *Title* </B> by *Author* (
URL: www.sff.net/locus/c
The three templates were then used to retrieve 4047 more (author, title) examples. The examples were then used to generate more templates, and so on, eventually yielding over 15,000 titles. Given a good set of templates, the system can collect a good set of examples. Given a good set of examples, the system can build a good set of templates.
The biggest weakness in this approach is the sensitivity to noise. If one of the first few templates is incorrect, errors can propagate quickly. One way to limit this problem is to not accept a new example unless it is verified by multiple templates, and not accept a new template unless it discovers multiple examples that are also found by other templates.

Machine reading

Automated template construction is a big step up from handcrafted template construction, but it still requires a handful of labeled examples of each relation to get started. To build a large ontology with many thousands of relations, even that amount of work would be onerous; we would like to have an extraction system with *no* human input of any kind—a system that could read on its own and build up its own database. Such a system would be relation-independent; would work for any relation. In practice, these systems work on *all* relations in parallel,
because of the I/O demands of large corpora. They behave less like a traditional information extraction system that is targeted at a few relations and more like a human reader who learns from the text itself; because of this the field has been called **machine reading**.
A representative machine-reading system is TEXTRUNNER (Banko and Etzioni, 2008) uses cotraining to boost its performance, but it needs something to bootstrap from. In the case of Hearst (1992), specific patterns (e.g., *such as*) provided the bootstrap, and for Brin (1998), it was a set of five author–title pairs. For TEXTRUNNER, the original inspiration was a taxonomy of eight very general syntactic templates. It was felt that a small number of templates like this could cover most of the ways that relationships are expressed in English. The actual bootsrapping starts from a set of labelled examples that are extracted from the Penn Treebank, a corpus of parsed sentences. For example, from the parse of the sentence “Einstein received the Nobel Prize in 1921” TEXTRUNNER is able is able to extract the relation (“Einstein,” “received,” “Nobel Prize”).
Given a set of labeled examples of this type, TEXTRUNNER trains a linear-chain CRF to extract further examples from unlabeled text. The features in the CRF include function words like “to” and “of” and “the,” but not nouns and verbs (and not noun phrases or verb phrases). Because TEXTRUNNER is domain-independent, it cannot rely on predefined lists of nouns and verbs.

Type	Template	Example	Frequency
Verb	NP_1 *Verb* NP_2	X established Y	38%
Noun–Prep	NP_1 *NP Prep* NP_2	X settlement with Y	23%
Verb–Prep	NP_1 *Verb Prep* NP_2	X moved to Y	16%
Infinitive	NP_1 **to** *Verb* NP_2	X plans to acquire Y	9%
Modifier	NP_1 *Verb* NP_2 *Noun*	X is Y winner	5%
Noun-Coordinate	NP_1 (, | **and** | **-** | :) NP_2 *NP*	X-Y deal	2%
Verb-Coordinate	NP_1 (,| **and**) NP_2 *Verb*	X, Y merge	1%
Appositive	NP_1 *NP* (:| ,)? NP_2	X hometown : Y	1%

Figure 22.3 Eight general templates that cover about 95% of the ways that relations are expressed in English.

TEXTRUNNER achieves a precision of 88% and recall of 45% (F1 of 60%) on a large Web corpus. TEXTRUNNER has extracted hundreds of millions of facts from a corpus of a half-billion Web pages. For example, even though it has no predefined medical knowledge, it has extracted over 2000 answers to the query [what kills bacteria]; correct answers include antibiotics, ozone, chlorine, Cipro, and broccoli sprouts. Questionable answers include "water,"
which came from the sentence "Boiling water for at least 10 minutes will kill bacteria." It would be better to attribute this to "boiling water" rather than just "water."
With the techniques outlined in this chapter and continual new inventions, we are starting to get closer to the goal of machine reading.

PART- A

1. What is Reinforcement Learning ?

   A utility-based agent learns a utility function on states and uses it to select actions that maximize the expected outcome utility.

   A Q-learning agent learns an action-utility function, or Q-function, giving the expected utility of taking a given action in a given state.

   A reflex agent learns a policy that maps directly from states to actions.

2. Define Active Reinforcement Learning

   Active reinforcement learning (ARL) is **a variant on reinforcement learning where the agent does not observe the reward unless it chooses to pay a query** cost c > 0. The central question of ARL is how to quantify the long-term value of reward information.

3. Define Passive Reinforcement Learning

In case of passive RL, **the agent's policy is fixed which means that it is told what to do**. ... Therefore, the goal of a passive RL agent is to execute a fixed policy (sequence of actions) and evaluate it while that of an active RL agent is to act and learn an optimal policy.

4. What is Natural Language Processing ?

Natural Language Processing (NLP) is a branch of Artificial Intelligence (AI) that **enables machines to understand the human language**. Its goal is to build systems that can make sense of text and automatically perform tasks like translation, spell check, or topic classification

5. Define Information Retrieval.

the techniques of storing and recovering and often disseminating recorded data especially through the use of a computerized system.

6. Define Information Extraction.

Information extraction (IE) is the task of **automatically extracting structured information from unstructured and/**or semi-structured machine-readable documents, while information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information.

PART –B

1. Explain about Active Reinforcement Learning and Passive Reinforcement Learning .
2. Write about Generalization in Reinforcement Learning.
3. Write about Applications of RL.
4. Explain the types of Language Models in Natural Language Processing.
5. Describe the Text Classifications.
6. Write about Information Retrieval and Information Extraction.