

11.24 In this problem we explore the use of deterministic annealing for pattern classification using a neural network (Miller et al., 1996). The output of neuron j in the output layer is denoted by $F_j(\mathbf{x})$ where $\mathbf{x}$ is the input vector. The classification decision is based on the maximum discriminant $F_j(\mathbf{x})$.

(a) For a probabilistic objective function, consider the following:

$$F = \frac{1}{N} \sum_{(\mathbf{x}, \ell) \in \mathcal{T}} \sum_j P(\mathbf{x} \in \mathcal{R}_j) F_j(\mathbf{x})$$

where $\mathcal{T}$ is a training set of labeled vectors with $\mathbf{x}$ denoting an input vector and ℓ its class label, and $P(\mathbf{x} \in \mathcal{R}_j)$ is the probability of association between input vector $\mathbf{x}$ and class region $\mathcal{R}_j$. Using the maximum entropy principle that is discussed in Chapter 10, formulate the Gibbs distribution for $P(\mathbf{x} \in \mathcal{R}_j)$.

(b) Let $\langle P_e \rangle$ denote the average misclassification cost. Formulate the Lagrangian for minimization of $\langle P_e \rangle$ subject to the constraint that the entropy corresponding to the association probabilities $P(\mathbf{x} \in \mathcal{R}_j)$ is equal to some constant value H .

CHAPTER 12

Neurodynamic Programming

12.1 INTRODUCTION

In Chapter 2 we identify two main paradigms of learning: learning with a teacher, and learning without a teacher. The paradigm of learning without a teacher is subdivided into self-organized (unsupervised) learning and reinforcement learning. Different forms of learning with a teacher or supervised learning are covered in Chapters 4 through 7, and different forms of unsupervised learning are discussed in Chapters 8 through 11. In this chapter we discuss *reinforcement learning*.

Supervised learning is a “cognitive” learning problem performed under the tutelage of a teacher: It relies on the availability of an adequate set of input–output examples that are representative of the operating environment. In contrast, reinforcement learning is a “behavioral” learning problem: It is performed through *interaction* between the learning system and its environment, in which the system seeks to achieve a specific goal despite the presence of uncertainties (Barto et al., 1983; Sutton and Barto, 1998). The fact that this interaction is performed without a teacher makes reinforcement learning particularly attractive for dynamic situations where it is costly or difficult (if not impossible) to gather a satisfactory set of input–output examples.

There are two approaches to the study of reinforcement learning,¹ summarized as follows:

1. *Classical approach*, in which learning takes place through a process of punishment and reward with the goal of achieving a *highly skilled behavior*.
2. *Modern approach*, which builds on a mathematical technique known as dynamic programming to decide on a course of action by considering possible future stages without actually experiencing them; the emphasis here is on *planning*.

Our discussion focuses on modern reinforcement learning.

*Dynamic programming*² is a technique that deals with situations where decisions are made in stages, with the outcome of each decision being predictable to some extent before the next decision is made. A key aspect of such situations is that decisions cannot be made in isolation. Rather, the desire for a low cost at the present must be balanced against the undesirability of high costs in the future. This is a *credit assignment problem* because credit or blame must be assigned to each one of a set of interacting

decisions. For optimal planning it is necessary to have an efficient tradeoff between immediate and future costs. Such a tradeoff is indeed captured by the formalism of dynamic programming. In particular, dynamic programming addresses the question: How can a system learn to improve long-term performance when this may require sacrificing short-term performance?

Following Bertsekas and Tsitsiklis (1996), we refer to the modern approach to reinforcement learning as *neurodynamic programming*. We do so primarily for two reasons:

- The theoretical foundation is provided by dynamic programming.
- The learning capability is provided by neural networks.

A succinct definition of neurodynamic programming is (Bertsekas and Tsitsiklis, 1996):

Neurodynamic programming enables a system to learn how to make good decisions by observing its own behavior, and to improve its actions by using a built-in mechanism through reinforcement.

The observation of behavior is attained through the use of Monte Carlo simulation in an off-line manner. The improvement of actions through reinforcement is attained through the use of an iterative optimization scheme.

Organization of the Chapter

Dynamic programming has two main features: an underlying discrete-time dynamic system, and a cost function that is additive over time. These two features are discussed in Section 12.2. This is followed by a formulation of Bellman's optimality equation in Section 12.3, which plays an important role in dynamic programming. In Sections 12.4 and 12.5 we discuss two different methods for computing an optimal policy for dynamic programming, namely policy iteration and value iteration.

In Section 12.6 we present an overview of the issues involved in neurodynamic programming. This overview leads to the discussion of approximate policy iteration and Q-learning, which lend themselves to the use of neural networks for function approximation. These two algorithms are discussed in Sections 12.7 and 12.8, respectively. Section 12.9 presents a computer experiment on the use of Q-learning.

The chapter concludes with some final remarks in Section 12.10.

12.2 MARKOVIAN DECISION PROCESS

Consider a *learning system* or *agent* that interacts with its environment in the manner illustrated in Fig. 12.1. The system operates in accordance with a *finite, discrete-time Markovian decision process* that is characterized as follows:

- The environment evolves probabilistically occupying a finite set of discrete states. Note, however, that the state does *not* contain past statistics, even though these statistics could be useful to the learning system.
- For each environmental state there is a finite set of possible actions that may be taken by the learning system.

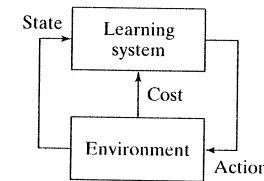


FIGURE 12.1 Block diagram of a learning system interacting with its environment.

- Every time the learning system takes an action, a certain cost is incurred.
- States are observed, actions are taken, and costs are incurred at discrete times.

In the context of our present discussion, the *state* of the environment is defined as: *a summary of the entire past experience of the learning system gained from its interaction with the environment, such that the information necessary for the learning system to predict future behavior of the environment is contained in that summary*. The random variable denoting the state at time step n is X_n , and the actual state at time step n is denoted by $\mathbf{x}(n)$. The finite set of states is denoted by $\mathcal{X}$. A surprising aspect of dynamic programming is that its applicability depends very little on the nature of the state. We may therefore proceed without any assumption on the structure of the state space.

For state i , for example, the available set of *actions* (i.e., inputs applied to the environment by the learning system) is denoted by $\mathcal{A}_i = \{a_{ik}\}$, where the second subscript k in action a_{ik} taken by the learning system merely indicates the availability of more than one possible action when the environment is in state i . The transition of the environment from the state i to the new state j , for example, due to action a_{ik} is probabilistic in nature. Most importantly, however, *the transition probability from state i to state j depends entirely on the current state i and the corresponding action a_{ik}* . This is the *Markov property*, which is discussed in Chapter 11. This property is crucial because it means that the current state of the environment provides the necessary information for the learning system to decide what action to take.

The random variable denoting the action taken by the learning system at time step n is denoted by A_n . Let $p_{ij}(a)$ denote the transition probability from state i to state j due to action taken at time step n , where $A_n = a$. By virtue of the Markov property we have

$$p_{ij}(a) = P(X_{n+1} = j \mid X_n = i, A_n = a) \quad (12.1)$$

The transition probability $p_{ij}(a)$ satisfies two conditions that are imposed on it by probability theory:

$$1. \quad p_{ij}(a) \geq 0 \quad \text{for all } i \text{ and } j \quad (12.2)$$

$$2. \quad \sum_j p_{ij}(a) = 1 \quad \text{for all } i \quad (12.3)$$

For a given number of states and given transition probabilities the sequence of environmental states resulting from the actions taken by the learning system over time forms a *Markov chain*. Markov chains are discussed in Chapter 11.

At each transition from one state to another, a *cost* is incurred by the learning system. Thus, at the n th transition from state i to state j under action a_{ik} , the learning

system incurs a cost denoted by $\gamma^n g(i, a_{ik}, j)$, where $g(\cdot, \cdot, \cdot)$ is a prescribed function, and γ is a scalar with $0 \leq \gamma < 1$ called the *discount factor*. By adjusting γ we are able to control the extent to which the learning system is concerned with long-term versus short-term consequences of its own actions. In the limit, when $\gamma = 0$ the system is myopic in the sense that it is only concerned with immediate consequences of its actions. In what follows, we will ignore this limiting value, that is, confine the discussion to $0 < \gamma < 1$. As γ approaches 1, future costs become more important in determining optimal actions.

Our interest is in the formulation of a *policy*, defined as a *mapping of states into actions*. In other words, a policy is a rule used by the learning system to decide what to do, given knowledge of the current state of the environment. The policy is denoted by

$$\pi = \{\mu_0, \mu_1, \mu_2, \dots\} \quad (12.4)$$

where μ_n is a function that maps the state $X_n = i$ into an action $A_n = a$ at time step $n = 0, 1, 2, \dots$. This mapping is such that

$$\mu_n(i) \in \mathcal{A}_i \quad \text{for all states } i \in \mathcal{X}$$

where $\mathcal{A}_i$ denotes the set of all possible actions taken by the learning system in state i . Such policies are said to be *admissible*.

A policy can be nonstationary or stationary. A *nonstationary* policy is time varying, as indicated in Eq. (12.4). When, however, the policy is independent of time, that is,

$$\pi = \{\mu, \mu, \mu, \dots\}$$

the policy is said to be *stationary*. In other words, a stationary policy specifies exactly the same action each time a particular state is visited. For a stationary policy, the underlying Markov chain may be stationary or nonstationary; it is possible to use a stationary policy on a nonstationary Markov chain, but this is *not* a wise thing to do. If a stationary policy μ is employed, then the sequence of states $\{X_n, n = 0, 1, 2, \dots\}$ forms a Markov chain with transition probabilities $p_{ij}(\mu(i))$, where $\mu(i)$ signifies an action. It is for this reason that the process is referred to as a *Markov decision process*.

Basic Problem

A dynamic programming problem can be of a finite-horizon or infinite-horizon kind. In a *finite-horizon* problem the cost accumulates over a finite number of stages. In an *infinite-horizon* problem the cost accumulates over an infinite number of stages. Infinite-horizon problems provide a reasonable approximation to problems involving a finite but very large number of stages. They are also of particular interest because discounting ensures that the costs for all states are finite for any policy.

The total expected cost in an infinite-horizon problem, starting from an initial state $X_0 = i$ and using a policy $\pi = \{\mu_n\}$, is defined by

$$J^\pi(i) = E \left[\sum_{n=0}^{\infty} \gamma^n g(X_n, \mu_n(X_n), X_{n+1}) \mid X_0 = i \right] \quad (12.5)$$

where the expected value is taken with respect to the Markov chain $\{X_1, X_2, \dots\}$. The function $J^\pi(i)$ is called the *cost-to-go function* for policy π starting from state i . Its *optimal* value, denoted by $J^*(i)$, is defined by

$$J^*(i) = \min_{\pi} J^\pi(i) \quad (12.6)$$

When the policy π is stationary, that is, $\pi = \{\mu, \mu, \dots\}$, we use the notation $J^\mu(i)$ in place of $J^\pi(i)$, and say that μ is optimal if

$$J^\mu(i) = J^*(i) \quad \text{for all initial states } i \quad (12.7)$$

We may now sum up the basic problem in dynamic programming as follows:

Given a stationary Markovian decision process describing the interaction between a learning system and its environment, find a stationary policy $\pi = \{\mu, \mu, \mu, \dots\}$ that minimizes the cost-to-go function $J^\mu(i)$ for all initial states i .

Note that during learning the behavior of a learning system may change with time. However, the optimal policy that the learning system seeks will be stationary (Watkins, 1989).

12.3 BELLMAN'S OPTIMALITY CRITERION

The dynamic programming technique rests on a very simple idea known as the *principle of optimality* due to Bellman (1957). Simply stated this principle says (Bellman and Dreyfus, 1962):

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

As used here, a "decision" is a choice of control at a particular time, and a "policy" is the entire control sequence or control function.

To formulate the principle of optimality in mathematical terms, consider a finite-horizon problem for which the cost-to-go function is defined by

$$J_0(X_0) = E \left[g_K(X_K) + \sum_{n=0}^{K-1} g_n(X_n, \mu_n(X_n), X_{n+1}) \right] \quad (12.8)$$

where K is the *horizon* (i.e., number of stages) and $g_K(X_K)$ is the *terminal cost*. Given X_0 , the expectation in Eq. (12.8) is with respect to the remaining states $X_1, \dots, X_{K-1}$. With this terminology we may now formally state the principle of optimality as (Bertsekas, 1995b):

Let $\pi^* = \{\mu_0^*, \mu_1^*, \dots, \mu_{K-1}^*\}$ be an optimal policy for the basic finite-horizon problem. Assume that when using the optimal policy π^* a given state X_n occurs with positive probability. Consider the subproblem where the environment is in state X_n at time n , and suppose we wish to minimize the corresponding cost-to-go function

$$J_n(X_n) = E \left[g_K(X_K) + \sum_{k=n}^{K-1} g_k(X_k, \mu_k(X_k), X_{k+1}) \right] \quad (12.9)$$

for $n = 0, 1, \dots, K-1$. Then the truncated policy $\{\mu_n^*, \mu_{n+1}^*, \dots, \mu_{K-1}^*\}$ is optimal for the subproblem.

We may intuitively justify the principle of optimality by the following argument: If the truncated policy $\{\mu_n^*, \mu_{n+1}^*, \dots, \mu_{K-1}^*\}$ was not optimal as stated, then once the state X_n is reached at time n we could reduce the cost-to-go function $J_n(X_n)$ simply by switching to a policy that is optimal for the subproblem.

The principle of optimality builds on the engineering notion of "divide and conquer." Basically, an optimal policy for a complex multistage planning or control problem can be constructed by proceeding as follows:

- Construct an optimal policy for the "tail subproblem" involving only the last stage of the system.
- Extend the optimal policy to the "tail subproblem" involving the last two stages of the system.
- Continue the procedure in this fashion until the entire problem has been dealt with.

Dynamic Programming Algorithm

On the basis of the procedure just described, we may now formulate the dynamic programming algorithm, which proceeds backward in time from period $N-1$ to period 0. Let $\pi = \{\mu_0, \mu_1, \dots, \mu_{K-1}\}$ denote an admissible policy. For each $n = 0, 1, \dots, K-1$, let $\pi^n = \{\mu_n, \mu_{n+1}, \dots, \mu_{K-1}\}$ and let $J_n^*(X_n)$ be the optimal cost for the $(K-n)$ -stage problem that starts at state X_n and time n and ends at time K ; that is,

$$J_n^*(X_n) = \min_{\pi^n} E_{(X_n, \mu_n, \dots, \mu_{K-1})} \left[g_K(X_K) + \sum_{k=n}^{K-1} g_k(X_k, \mu_k(X_k), X_{k+1}) \right] \quad (12.10)$$

which represents the optimal form of Eq. (12.9). Recognizing that $\pi^n = (\mu_n, \pi^{n+1})$ and partially expanding the summation on the right-hand side of Eq. (12.10), we may write

$$\begin{aligned} J_n^*(X_n) &= \min_{(\mu_n, \pi^{n+1})} E_{(X_n, \mu_n, \dots, \mu_{K-1})} \left[g_n(X_n, \mu_n(X_n), X_{n+1}) \right. \\ &\quad \left. + g_K(X_K) + \sum_{k=n+1}^{K-1} g_k(X_k, \mu_k(X_k), X_{k+1}) \right] \\ &= \min_{\mu_n} E_{X_{n+1}} \left\{ g_n(X_n, \mu_n(X_n), X_{n+1}) \right. \\ &\quad \left. + \min_{\pi^{n+1}} E_{(X_{n+1}, \mu_{n+1}, \dots, \mu_{K-1})} \left[g_K(X_K) + \sum_{k=n+1}^{K-1} g_k(X_k, \mu_k(X_k), X_{k+1}) \right] \right\} \\ &= \min_{\mu_n} E_{X_{n+1}} \left[g_n(X_n, \mu_n(X_n), X_{n+1}) + J_{n+1}^*(X_{n+1}) \right] \end{aligned} \quad (12.11)$$

where in the last line we have made use of the defining equation (12.10) with $n+1$ used in place of n . Now assume that for some n and all X_{n+1} , we have

$$J_{n+1}^*(X_{n+1}) = J_{n+1}(X_{n+1}) \quad (12.12)$$

Then we may rewrite Eq. (12.11) in the form

$$J_n^*(X_n) = \min_{\mu_n} E_{X_{n+1}} [g_n(X_n, \mu_n(X_n), X_{n+1}) + J_{n+1}(X_{n+1})] \quad (12.13)$$

If Eq. (12.12) holds for all X_{n+1} , then clearly the equation

$$J_n^*(X_n) = J_n(X_n)$$

also holds for all X_n . Accordingly, we deduce from Eq. (12.13) that

$$J_n(X_n) = \min_{\mu_n} E_{X_{n+1}} [g_n(X_n, \mu_n(X_n), X_{n+1}) + J_{n+1}(X_{n+1})]$$

We may thus formally state the *dynamic programming algorithm* as follows (Bertsekas, 1995b):

For every initial state X_0 , the optimal cost $J^*(X_0)$ of the basic finite-horizon problem is equal to $J_0(X_0)$, where the function J_0 is obtained from the last step of the following algorithm:

$$J_n(X_n) = \min_{\mu_n} E_{X_{n+1}} [g_n(X_n, \mu_n(X_n), X_{n+1}) + J_{n+1}(X_{n+1})] \quad (12.14)$$

which runs backward in time, with

$$J_K(X_K) = g_K(X_K)$$

Furthermore, if μ_n^* minimizes the right-hand side of Eq. (12.14) for each X_n and n , then the policy $\pi^* = \{\mu_0^*, \mu_1^*, \dots, \mu_{K-1}^*\}$ is optimal.

Bellman's Optimality Equation

In its basic form, the dynamic programming algorithm deals with a finite-horizon problem. We are interested in extending the use of this algorithm to deal with the infinite-horizon discounted problem described by the cost-to-go function of Eq. (12.5) under a stationary policy $\pi = \{\mu, \mu, \mu, \dots\}$. With this objective in mind, we do two things:

- Reverse the time index of the algorithm so that it corresponds to the discounted problem.
- Define the cost $g_n(X_n, \mu(X_n), X_{n+1})$ as

$$g_n(X_n, \mu(X_n), X_{n+1}) = \gamma^n g(X_n, \mu(X_n), X_{n+1}) \quad (12.15)$$

We may now reformulate the dynamic programming algorithm as follows (see Problem 12.4):

$$J_{n+1}(X_0) = \min_{\mu} E_{X_1} [g(X_0, \mu(X_0), X_1) + \gamma J_n(X_1)] \quad (12.16)$$

which starts from the initial conditions

$$J_0(X) = 0 \quad \text{for all } X$$

The state X_0 is the initial state, X_1 is the new state that results from the action of policy μ , and γ is the discount factor.

Let $J^*(i)$ denote the optimal infinite-horizon cost for the initial state $X_0 = i$. We may then view $J^*(i)$ as the limit of the corresponding K -stage optimal cost $J_K(i)$ as the horizon K approaches infinity; that is,

$$J^*(i) = \lim_{K \rightarrow \infty} J_K(i) \quad \text{for all } i \quad (12.17)$$

This relation is the connecting link between the finite-horizon and infinite-horizon discounted problems. Putting $n + 1 = K$ and $X_0 = i$ in Eq. (12.16), and then applying Eq. (12.17), we obtain

$$J^*(i) = \min_{\mu} E_{X_1} [g(i, \mu(i), X_1) + \gamma J^*(X_1)] \quad (12.18)$$

To evaluate the optimal infinite-horizon cost $J^*(i)$, we proceed in two stages:

1. We evaluate the expectation of the cost $g(i, \mu(i), X_1)$ with respect to X_1 by writing

$$E[g(i, \mu(i), X_1)] = \sum_{j=1}^N p_{ij} g(i, \mu(i), j) \quad (12.19)$$

where N is the number of states of the environment, and p_{ij} is the transition probability from the initial state $X_0 = i$ to the new state $X_1 = j$. The quantity defined in Eq. (12.19) is the *immediate expected cost* incurred at state i by following the action recommended by the policy μ . Denoting this cost by $c(i, \mu(i))$, we may write

$$c(i, \mu(i)) = \sum_{j=1}^N p_{ij} g(i, \mu(i), j) \quad (12.20)$$

2. We evaluate the expectation of $J^*(X_1)$ with respect to X_1 . Here we note that if we know the cost $J^*(X_1)$ for each state X_1 of a finite-state system, we may readily determine the expectation of $J^*(X_1)$ in terms of the transition probabilities of the underlying Markov chain by writing

$$E[J^*(X_1)] = \sum_{j=1}^N p_{ij} J^*(j) \quad (12.21)$$

Thus, using Eqs. (12.19) to (12.21) in Eq. (12.16), we obtain the desired result

$$J^*(i) = \min_{\mu} \left(c(i, \mu(i)) + \gamma \sum_{j=1}^N p_{ij}(\mu) J^*(j) \right) \quad \text{for } i = 1, 2, \dots, N \quad (12.22)$$

Equation (12.22) is called *Bellman's optimality equation*. It should *not* be viewed as an algorithm. Rather, it represents a system of N equations with one equation per state. The solution of this system of equations defines the optimal cost-to-go functions for the N states of the environment.

There are two basic methods for computing an optimal policy. They are called policy iteration and value iteration. These two methods are described in Sections 12.4 and 12.5, respectively.

12.4 POLICY ITERATION

To set the stage for a description of the policy iteration algorithm, we begin by introducing a concept called the Q-factor due to Watkins (1989). Consider an existing policy μ for which the cost-to-go function $J^{\mu}(i)$ is known for all states i . The *Q-factor* for each

state $i \in \mathcal{X}$ and action $a \in \mathcal{A}_i$ is defined as the *immediate cost plus the sum of the discounted costs of all successor states following policy μ* , as shown by

$$Q^{\mu}(i, a) = c(i, a) + \gamma \sum_{j=1}^N p_{ij}(a) J^{\mu}(j) \quad (12.23)$$

where the action $a = \mu(i)$. Note that the Q-factors $Q^{\mu}(i, a)$ contain more information than the cost-to-go function $J^{\mu}(i)$. For example, actions may be ranked on the basis of Q-factors alone, whereas ranking on the basis of cost-to-go function also requires knowledge of the state-transition probabilities and costs.

We may develop insight into the meaning of the Q-factor by visualizing a new system whose states are made up of the original states $1, 2, \dots, N$ and all the possible state-action pairs (i, a) , as portrayed in Fig. 12.2. There are two distinct possibilities that can occur:

- The system is in state (i, a) , in which case no action is taken. Transition is made automatically to state j , say, with probability $p_{ij}(a)$; and a cost $g(i, a, j)$ is incurred.
- The system is in state i , say, in which case action $a \in \mathcal{A}_i$ is taken. The next state is (i, a) , deterministically.

The policy μ is said to be *greedy* with respect to the cost-to-go function $J^{\mu}(i)$ if, for all states, $\mu(i)$ is an action that satisfies the condition

$$Q^{\mu}(i, \mu(i)) = \min_{a \in \mathcal{A}_i} Q^{\mu}(i, a) \quad \text{for all } i \quad (12.24)$$

The following two observations on Eq. (12.24) are noteworthy:

- It is possible for more than one action to minimize the set of Q-factors for some state, in which case there can be more than one greedy policy with respect to the pertinent cost-to-go function.
- A policy can be greedy with respect to many different cost-to-go functions.

Moreover, the following fact is basic to all dynamic programming methods:

$$Q^{\mu^*}(i, \mu^*(i)) = \min_{a \in \mathcal{A}_i} Q^{\mu^*}(i, a) \quad (12.25)$$

where μ^* is an optimal policy and J^* is the corresponding optimal cost-to-go function.

With the notions of Q-factor and greedy policy at our disposal, we are ready to describe the *policy iteration algorithm*. Specifically, the algorithm operates by alternating between two steps (Bertsekas, 1995b):

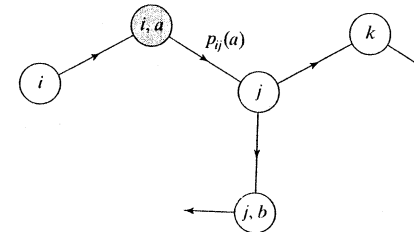
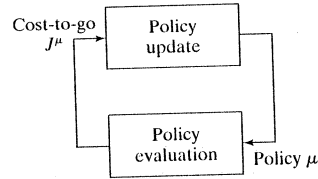


FIGURE 12.2 Illustration of two possible transitions: the transition from state (i, a) to state j is probabilistic, but the transition from state i to (i, a) is deterministic.

FIGURE 12.3 Block diagram for policy iteration algorithm.



1. *Policy evaluation step*, in which the cost-to-go function for some current policy and the corresponding Q-factor are computed for all states and actions.
2. *Policy improvement step*, in which the current policy is updated in order to be greedy with respect to the cost-to-go function computed in step 1.

These two steps are illustrated in Fig. 12.3. To be specific, we start with some initial policy μ_0 , and then generate a sequence of new policies $\mu_1, \mu_2, \dots$. Given the current policy μ_n , we perform the policy evaluation step by computing the cost-to-go function $J^{\mu_n}(i)$ as the solution of the linear system of equations (see Eq. (12.22))

$$J^{\mu_n}(i) = c(i, \mu_n(i)) + \gamma \sum_{j=1}^N p_{ij}(\mu_n(i)) J^{\mu_n}(j), \quad i = 1, 2, \dots, N \quad (12.26)$$

in the unknowns $J^{\mu_n}(1), J^{\mu_n}(2), \dots, J^{\mu_n}(N)$. Using these results, we then compute the Q-factor for state-action pair (i, a) (see Eq. (12.23))

$$Q^{\mu_n}(i, a) = c(i, a) + \gamma \sum_{j=1}^N p_{ij}(a) J^{\mu_n}(j), \quad a \in \mathcal{A}_i \text{ and } i = 1, 2, \dots, N \quad (12.27)$$

Next we perform the policy improvement step by computing a new policy μ_{n+1} defined by (see Eq. (12.24))

$$\mu_{n+1}(i) = \arg \min_{a \in \mathcal{A}_i} Q^{\mu_n}(i, a), \quad i = 1, 2, \dots, N \quad (12.28)$$

The two-step process just described is repeated with policy μ_{n+1} used in place of μ_n , unless we have

$$J^{\mu_{n+1}}(i) = J^{\mu_n}(i) \quad \text{for all } i$$

in which case the algorithm is terminated with policy μ_n . With $J^{\mu_{n+1}} \leq J^{\mu_n}$ (see Problem 12.5) we may then say that the policy iteration algorithm will terminate after a finite number of iterations because the underlying Markovian decision process has a finite number of states. Table 12.1 presents a summary of the policy iteration algorithm based on Eqs. (12.26) to (12.28).

5 VALUE ITERATION

In the policy iteration algorithm, the cost-to-go function has to be recomputed entirely at each iteration of the algorithm, which is expensive. Even though the cost-to-go function for the new policy may be similar to that for the old policy, unfortunately there is

TABLE 12.1 Summary of the Policy Iteration Algorithm

1. Start with an arbitrary initial policy μ_0 .
2. For $n = 0, 1, 2, \dots$, compute $J^{\mu_n}(i)$ and $Q^{\mu_n}(i, a)$ for all states $i \in \mathcal{X}$ and actions $a \in \mathcal{A}_i$.
3. For each state i , compute

$$\mu_{n+1}(i) = \arg \min_{a \in \mathcal{A}_i} Q^{\mu_n}(i, a)$$

4. Repeat steps 2 and 3 until μ_{n+1} is not an improvement on μ_n , at which point the algorithm terminates with μ_n as the desired policy.

no dramatic shortcut for this computation. There is, however, another method for finding the optimal policy that avoids the burdensome task of repeatedly computing the cost-to-go function. This alternative method, based on successive approximations, is known as the value iteration algorithm.

The *value iteration algorithm* involves solving Bellman's optimality equation (12.22) for each of a sequence of finite-horizon problems. In the limit, the cost-to-go function of the finite-horizon problem converges uniformly over all states to the corresponding cost-to-go function of the infinite-horizon problem as the number of iterations of the algorithm approaches infinity (Ross, 1983; Bertsekas, 1995b).

Let $J_n(i)$ denote the cost-to-go function for state i at iteration n of the value iteration algorithm. The algorithm begins with an arbitrary guess $J_0(i)$ for $i = 1, 2, \dots, N$. The only restriction on $J_0(i)$ is that it should be bounded; this is automatically true for finite-state problems. If some estimate of the optimal cost-to-go function $J^*(i)$ is available, it should be used as the initial value $J_0(i)$. Once $J_0(i)$ has been chosen, we may compute the sequence of cost-to-go functions $J_1(i), J_2(i), \dots$, using the value iteration algorithm

$$J_{n+1}(i) = \min_{a \in \mathcal{A}_i} \left\{ c(i, a) + \gamma \sum_{j=1}^N p_{ij}(a) J_n(j) \right\}, \quad i = 1, 2, \dots, N \quad (12.29)$$

Application of the update to the cost-to-go function, described in Eq. (12.29) for state i , is referred to as *backing up of i 's cost*. This backup is a direct implementation of Bellman's optimality equation (12.22). Note that the values of the cost-to-go functions in Eq. (12.29) for states $i = 1, 2, \dots, N$ are backed up simultaneously on each iteration of the algorithm. This method of implementation represents the traditional *synchronous* form of the value iteration algorithm.³ Thus, starting from arbitrary initial values $J_0(1), J_0(2), \dots, J_0(N)$, the algorithm described by Eq. (12.29) converges to the corresponding optimal values $J^*(1), J^*(2), \dots, J^*(N)$ as the number of iterations n approaches infinity (Ross, 1983; Bertsekas, 1995b).

Unlike the policy iteration algorithm, an optimal policy is not computed directly in the value iteration algorithm. Rather, the optimal values $J^*(1), J^*(2), \dots, J^*(N)$ are first computed using Eq. (12.29). Then a greedy policy with respect to that optimal set is obtained as an optimal policy. That is,

$$\mu^*(i) = \arg \min_{a \in \mathcal{A}_i} Q^*(i, a), \quad i = 1, 2, \dots, N \quad (12.30)$$

TABLE 12.2 Summary of the Value Iteration Algorithm

1. Start with arbitrary initial value $J_0(i)$ for state $i = 1, 2, \dots, N$.
2. For $n = 0, 1, 2, \dots$, compute

$$J_{n+1}(i) = \min_{a \in \mathcal{A}_i} \left\{ c(i, a) + \gamma \sum_{j=1}^N p_{ij}(a) J_n(j) \right\}, \quad \begin{matrix} a \in \mathcal{A}_i \\ i = 1, 2, \dots, N \end{matrix}$$

Continue this computation until

$$|J_{n+1}(i) - J_n(i)| < \epsilon \quad \text{for each state } i$$

where ϵ is a prescribed tolerance parameter. It is presumed that ϵ is sufficiently small for $J_n(i)$ to be close enough to the optimal cost-to-go function $J^*(i)$. We may thus set

$$J_n(i) = J^*(i) \quad \text{for all states } i$$

3. Compute the Q-factor

$$Q^*(i, a) = c(i, a) + \gamma \sum_{j=1}^N p_{ij}(a) J^*(j) \quad \begin{matrix} \text{for } a \in \mathcal{A}_i \text{ and} \\ i = 1, 2, \dots, N \end{matrix}$$

Hence, determine the optimal policy as a greedy policy for $J^*(i)$:

$$\mu^*(i) = \arg \min_{a \in \mathcal{A}_i} Q^*(i, a)$$

where

$$Q^*(i, a) = c(i, a) + \gamma \sum_{j=1}^N p_{ij}(a) J^*(j), \quad i = 1, 2, \dots, N \quad (12.31)$$

A summary of the value iteration algorithm, based on Eqs. (12.29) to (12.31), is presented in Table 12.2. This summary includes a stopping criterion for Eq. (12.29).

Example 12.1 Stagecoach Problem

To illustrate the usefulness of the Q -factor in dynamic programming, we consider the *stagecoach problem*. A fortune seeker in Missouri decided to go west to join the gold rush in California in the mid-nineteenth century (Hiller and Lieberman, 1995). The journey required traveling by stagecoach through unsettled country, which posed a serious danger of attack by marauders along the way. The starting point of the journey (Missouri) and the destination (California) were fixed, but there was considerable choice as to which other eight states to travel through en route, as shown in Fig. 12.4. In this figure we have the following:

- A total of 10 states, with each state represented by a letter.
- The direction of travel is from left to right.
- There are four stages (i.e., stagecoach runs) from the point of embarkation in state A (Missouri) to the destination in state J (California).
- In moving from one state to the next, the action taken by the fortune seeker is to move up, straight, or down.
- There are a total of 18 possible routes from state A to state J.

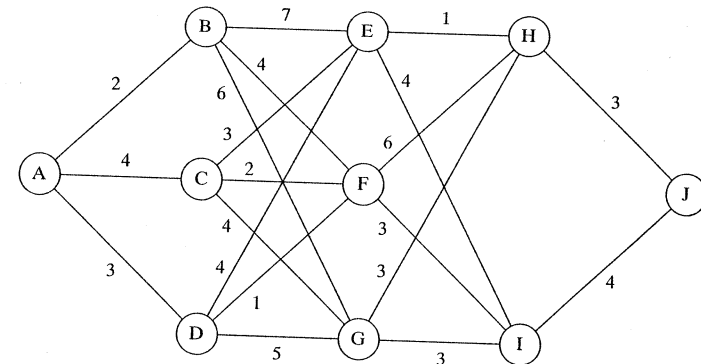
**FIGURE 12.4** Flow-graph for stagecoach problem.

Figure 12.4 also includes the cost of a life insurance policy for taking any stagecoach run based on a careful evaluation of the safety of that run. The problem is to find the route from state A to state J with the cheapest insurance policy.

To find the optimum route, we consider a sequence of finite horizon problems, starting from the destination in state J and working backward. This is in accordance with Bellman's principle of optimality described in Section 12.3.

Calculating the Q -factors for the last stage before the destination, we readily find in Fig. 12.5a that the terminal Q -values are as follows:

$$\begin{aligned} Q(H, \text{down}) &= 3 \\ Q(I, \text{up}) &= 4 \end{aligned}$$

These numbers are indicated on states H and I, respectively, in Fig. 12.5a.

Next, moving back by one more stage and using the Q -values in Fig. 12.5a, we have the following Q values:

$$\begin{aligned} Q(E, \text{straight}) &= 1 + 3 = 4 \\ Q(E, \text{down}) &= 4 + 4 = 8 \\ Q(F, \text{up}) &= 6 + 3 = 9 \\ Q(F, \text{down}) &= 3 + 4 = 7 \\ Q(G, \text{up}) &= 3 + 3 = 6 \\ Q(G, \text{straight}) &= 3 + 4 = 7 \end{aligned}$$

Since the requirement is to find the route with the smallest insurance policy, the Q -values indicate that only the stage runs $E \rightarrow H$, $F \rightarrow I$, and $G \rightarrow H$ should be retained and the remaining ones pruned, as indicated in Fig. 12.5b.

Moving back one further stage, repeating the calculations of the Q -factors for states B, C, and D in the manner described and retaining only those stage runs from the states B, C, and D that are covered by the lowest insurance costs, we obtain the picture depicted in Fig. 12.5c.

Finally, moving back one last stage and proceeding in the same way as before, we obtain the picture depicted in Fig. 12.5d. From this figure we see that there are indeed three optimal routes, as described here:

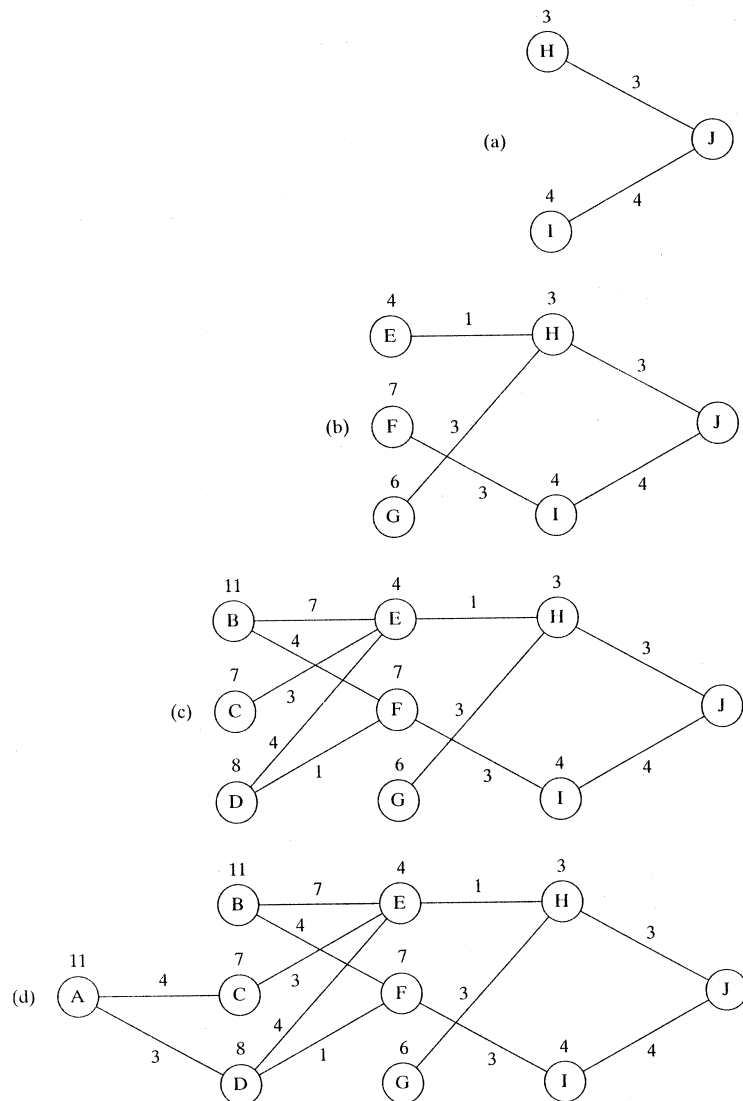


FIGURE 12.5 Steps involved in calculating the Q-factors for the stagecoach problem.

$$A \rightarrow C \rightarrow E \rightarrow H \rightarrow J$$

$$A \rightarrow D \rightarrow E \rightarrow H \rightarrow J$$

$$A \rightarrow D \rightarrow F \rightarrow I \rightarrow J$$

They all yield a total cost of 11.

12.6 NEURODYNAMIC PROGRAMMING

The primary objective of dynamic programming is to find an optimal policy, that is, an optimal choice of the action that should be taken by the learning system for each possible state of the environment. In this context there are two practical issues that must be remembered when considering use of the policy iteration or value iteration algorithm to solve a dynamic programming problem:

- *Curse of dimensionality.* For many difficult real-world problems the numbers of possible states and admissible actions are so large that the computational requirements of dynamic programming are overwhelming. For a dynamic programming problem involving a total of N possible states and M admissible actions for each state, each iteration of the value iteration algorithm, for example, requires about N^2M operations for a stationary policy. This frequently makes it impossible to complete even one iteration of the algorithm when N is very large. For example, backgammon has 10^{20} states, which means that a single iteration of the algorithm would take more than 1000 years using a 1000 MIPS processor (Barto, et al., 1995).
- *Incomplete information.* The policy iteration or value iteration algorithm requires prior knowledge of the underlying Markov decision process. That is, for the computation of an optimal policy to be feasible we require that the state-transition probabilities p_{ij} and the observed costs $g(i, a, j)$ be known. Unfortunately, this prior knowledge is not always available.

In light of one or the other or both of these difficulties, we often have to abandon the quest for an optimal policy and settle for a *suboptimal policy*.

Our interest here is in suboptimal procedures that involve the use of neural networks and/or simulation for the purpose of approximating the optimal cost-to-go function $J^*(i)$ for all $i \in \mathcal{X}$. Specifically, for a specified state i , $J^*(i)$ is replaced with a suitable approximation $\hat{J}(i, \mathbf{w})$, where $\mathbf{w}$ is a parameter vector. The function $\hat{J}(\cdot, \mathbf{w})$ is called the *scoring function* or *approximate cost-to-go function*, and the value $\hat{J}(i, \mathbf{w})$ is called the *score* or *approximate-to-go cost* for state i . Thus, as illustrated in Fig. 12.6, the score $\hat{J}(i, \mathbf{w})$ is the output of the neural network in response to the state i as input. The property that is exploited here is that of *universal approximation*, which, as discussed in previous chapters, is an inherent characteristic of multilayer perceptrons and radial-basis function networks.

Dynamic programming problems of particular interest are those with a large number of states, where the requirement is to find a scoring function $\hat{J}(\cdot, \mathbf{w})$ for which the parameter vector $\mathbf{w}$ has a small dimension. In this form of approximation, referred

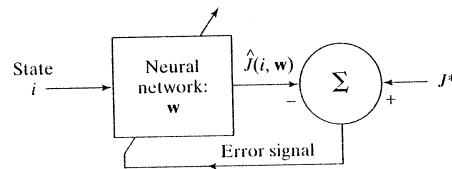


FIGURE 12.6 Neural network for approximating an optimal cost-to-go function J^* .

to as a *compact representation*, only the parameter vector $\mathbf{w}$ and general structure of the scoring function $\hat{J}(\cdot, \mathbf{w})$ are stored. The scores $\hat{J}(i, \mathbf{w})$ for all states $i \in \mathcal{X}$ are generated only when they are needed. The challenge is to find the parameter vector $\mathbf{w}$ algorithmically, such that for a given neural network structure (e.g., multilayer perceptron), the score $\hat{J}(i, \mathbf{w})$ provides a satisfactory approximation to the optimal value $J^*(i)$ for all $i \in \mathcal{X}$.

From the material presented in Chapters 4 through 7 on learning with a teacher, we know that a neural network, irrespective of its type, requires a set of labeled data that are representative of this task. However, in the context of dynamic programming problems, no such training data (i.e., input-output examples $\{(i, J^*(i))\}$) are available to train the neural network in Fig. 12.6, so as to optimize its design in some statistical sense. The only possibility is to use *Monte Carlo simulation*, where a surrogate model is used for the actual system underlying the Markovian decision process. The result is a novel off-line dynamic programming mode of operation that offers the following potential benefits (Bertsekas and Tsitsiklis, 1996):

1. The use of simulation to evaluate approximately the optimal cost-to-go function is the key idea that distinguishes the methodology of neurodynamic programming from traditional approximation methods in dynamic programming.
2. Simulation permits the use of neurodynamic programming methods to design systems for which no explicit models are available. For such systems, the traditional dynamic programming techniques are inapplicable, as it is cumbersome if not impossible to provide estimates of the state-transition probabilities.
3. Through simulation, it is possible to implicitly identify the most important or most representative states of the system as those states that are most often visited during the simulation. Consequently, the scoring function discovered by the neural network may provide a good approximation to the optimal cost-to-go function for those particular states. The end result may be a good suboptimal policy for a difficult dynamic programming problem.

However, it is important to recognize that once approximations are introduced, convergence of the scoring function $\hat{J}(\cdot, \mathbf{w})$ to the optimal cost-to-go function $J^*(\cdot)$ cannot be expected. That is for the simple reason that $J^*(\cdot)$ may not be within the set of functions represented exactly by the chosen neural network structure.

In the next two sections we discuss two approximate dynamic programming procedures with cost-to-go function approximations. The first procedure, described in Section 12.7, deals with approximate policy iteration, assuming that a Markovian model

of the system is available. The second procedure, described in Section 12.8, deals with a procedure called Q-learning, which makes no such assumption.

12.7 APPROXIMATE POLICY ITERATION

Suppose we have a dynamic programming problem for which the numbers of possible states and admissible actions are too large, making the use of a traditional approach impractical. It is assumed that we do have a model of the system; that is, the transition probabilities $p_{ij}(a)$ and the observed costs $g(i, a, j)$ are all known. To deal with this situation, we propose to use an approximation to policy iteration, based on Monte Carlo simulation and the method of least-squares, as described next (Bertsekas and Tsitsiklis, 1996).

Figure 12.7 shows a simplified block diagram of the *approximate policy iteration algorithm*. It is similar to the block diagram of Fig. 12.3 for the traditional policy iteration algorithm, but with an important difference: The policy evaluation step in Fig. 12.3 has been replaced with an *approximate* one. Thus, the approximate policy iteration algorithm proceeds by alternating between an approximate policy evaluation step and a policy improvement step as follows:

1. *Approximate policy evaluation step.* Given the current policy μ , a cost-to-go function $\hat{J}^\mu(i, \mathbf{w})$ approximating the actual cost-to-go function $J^\mu(i)$ is computed for all states i . The vector $\mathbf{w}$ is the parameter vector of the neural network used to perform the approximation.
2. *Policy improvement step.* Using the approximate cost-to-go function $\hat{J}^\mu(i, \mathbf{w})$, an improved policy μ is generated. This new policy is designed to be greedy with respect to $\hat{J}^\mu(i, \mathbf{w})$ for all i .

For the approximate policy iteration algorithm to yield satisfactory results, it is important to carefully choose the policy used to initialize the algorithm. This could be done through the use of heuristics. Alternatively, we may start with some weight vector $\mathbf{w}$, and use it to derive a greedy policy, which is in turn used as the initial policy.

Suppose then, in addition to the known transition probabilities and observed costs, we have the following items:

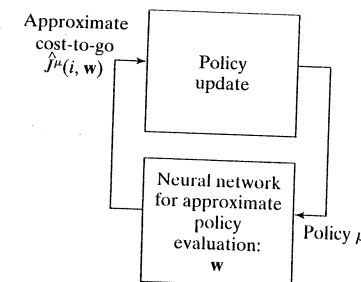


FIGURE 12.7 Simplified block diagram of the approximate policy iteration algorithm.

- A stationary policy μ as the initial policy
- A set of states $\mathcal{X}$ representative of the operating environment
- A set of $M(i)$ samples of the cost-to-go function $J^\mu(i)$ for each state $i \in \mathcal{X}$; one such sample is denoted by $k(i, m)$, where $m = 1, 2, \dots, M(i)$

Let $\hat{J}^\mu(i, \mathbf{w})$ denote an approximate representation of the cost-to-go function $J^\mu(i)$. The approximation is performed by a neural network (e.g., multilayer perceptron trained with the back-propagation algorithm). The parameter vector $\mathbf{w}$ of the neural network is determined by using the method of least-squares, that is, minimizing the cost function:

$$\mathcal{E}(\mathbf{w}) = \sum_{i \in \mathcal{X}} \sum_{m=1}^{M(i)} (k(i, m) - \hat{J}^\mu(i, \mathbf{w}))^2 \quad (12.32)$$

Having determined the optimum weight vector $\mathbf{w}$ and therefore the approximate cost-to-go function $\hat{J}^\mu(i, \mathbf{w})$, we next determine the approximate Q-factors using the formula (see Eqs. (12.20) and (12.23))

$$Q(i, a, \mathbf{w}) = \sum_{j \in \mathcal{X}} p_{ij}(a)(g(i, a, j) + \gamma \hat{J}^\mu(j, \mathbf{w})) \quad (12.33)$$

where the $p_{ij}(a)$ is the transition probability from state i to state j under action a (known), $g(i, a, j)$ is the observed cost (also known), and γ is a prescribed discount factor. The iteration is completed by using these approximate Q-factors to determine an improved policy based on the formula (see Eq. (12.28))

$$\mu(i) = \arg \min_{a \in \mathcal{A}_i} Q(i, a, \mathbf{w}) \quad (12.34)$$

It is important to note that Eqs. (12.33) and (12.34) are only used by the simulator to generate actions at the states that are *actually visited* by the simulation, rather than for all states. As such, these two equations need not suffer from the curse of dimensionality.

The block diagram of Fig. 12.8 presents a more detailed picture of the approximate policy iteration algorithm. This diagram consists of four interconnected modules

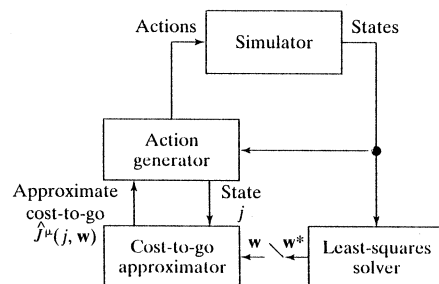


FIGURE 12.8 Detailed layout of the approximate policy iteration algorithm.

1. *Simulator*, which uses the given state-transition probabilities and observed one step costs to construct a surrogate model of the environment. The simulator generates two things: (a) states in response to actions to mimic the environment, and (b) samples of the cost-to-go function for a given policy.
2. *Action generator*, which generates an improved policy (i.e., sequence of actions in accordance with Eq. (12.34).
3. *Cost-to-go approximator*, which generates the approximate cost-to-go function $\hat{J}^\mu(i, \mathbf{w})$ for state i and parameter vector $\mathbf{w}$, for use in Eqs. (12.33) and (12.34).
4. *Least-squares solver*, which takes samples of the cost-to-go function $J^\mu(i)$ supplied by the simulator for policy μ and state i , and computes the optimum parameter vector $\mathbf{w}$ that minimizes the cost function of Eq. (12.32). The link from the least-squares solver to the cost-to-go approximator is switched on only after a policy has been fully evaluated and an optimal parameter vector $\mathbf{w}^*$ has been determined. At that point, the cost-to-go approximation $\hat{J}^\mu(i, \mathbf{w})$ is replaced by $\hat{J}^\mu(i, \mathbf{w}^*)$.

Table 12.3 presents a summary of the approximate policy iteration algorithm.

Naturally, the operation of this algorithm is subject to errors due to unavoidable imperfections in the design of the simulator and least-squares solver. The neural network used to perform the least-squares approximation of the desired cost-to-go function may lack adequate computing power, hence the first source of error. Optimization

TABLE 12.3 Summary of the Approximate Policy Iteration Algorithm

Known parameters: transition probabilities $p_{ij}(a)$ and costs $g(i, a, j)$.

Computation:

1. Choose a stationary policy μ as the initial policy.
2. Using a set of samples $\{k(i, m)\}_{m=1}^{M(i)}$ of the cost-to-go function $J^\mu(i)$ generated by the simulator, determine the parameter vector $\mathbf{w}$ of the neural network employed as the least-squares solver:

$$\begin{aligned} \mathbf{w}^* &= \min_{\mathbf{w}} \mathcal{E}(\mathbf{w}) \\ &= \min_{\mathbf{w}} \sum_{i \in \mathcal{X}} \sum_{m=1}^{M(i)} (k(i, m) - \hat{J}^\mu(i, \mathbf{w}))^2 \end{aligned}$$

3. For the parameter vector $\mathbf{w}$ determined in step 2, compute the approximate cost-to-go function $\hat{J}^\mu(i, \mathbf{w})$ for the states visited. Determine the approximate Q-factors:

$$Q(i, a, \mathbf{w}) = \sum_{j \in \mathcal{X}} p_{ij}(a)(g(i, a, j) + \gamma \hat{J}^\mu(j, \mathbf{w}))$$

4. Determine the improved policy

$$\mu(i) = \arg \min_{a \in \mathcal{A}_i} Q(i, a, \mathbf{w})$$

5. Repeat steps 2 through 4.

Note: Steps 3 and 4 apply only to actions at the states that are actually visited, rather than all states.

of the neural network approximator and therefore tuning of the parameter vector $\mathbf{w}$ is based on a desired response provided by the simulator, hence the second source of error. Assuming that all policy evaluations and all policy improvements are performed within certain error tolerances of ϵ and δ , respectively, it is shown in Bertsekas and Tsitsiklis (1996) that the approximate policy iteration algorithm will produce policies whose performances differ from the optimal policies by a factor that decreases to zero as ϵ and δ are reduced. In other words, the approximate policy iteration algorithm is sound with minimal performance guarantees. According to Bertsekas and Tsitsiklis (1996), the approximate policy iteration algorithm tends to initially make rapid and fairly monotonic progress, but a sustained policy oscillation of a random nature may result as a limiting condition. This oscillatory behavior occurs after the approximating cost-to-go function $\hat{J}$ gets within a zone of $O(\delta + 2\gamma\epsilon)/(1 - \gamma)^2$ of the optimal value J^* , where γ is the discount parameter. Apparently there is a fundamental structure that is common to all variants of approximate policy iteration, which causes an oscillatory behavior.

2.8 Q-LEARNING

The behavioral task of the reinforcement learning system in Fig. 12.1 is how to find an optimal (i.e., minimal cost) policy after trying out various possible sequences of actions, and observing the costs incurred and the state transitions that occur. In this context we may raise the following question: Is there an on-line procedure for learning the optimal policy through experience gained solely on the basis of samples of the form

$$s_n = (i_n, a_n, j_n, g_n) \quad (12.35)$$

where n denotes discrete time, and each sample s_n consists of a four-tuple, described by a trial action a_n on state i_n that results in a transition to state $j_n = i_{n+1}$ at a cost $g_n = g(i_n, a_n, j_n)$. The answer to this fundamental question is an emphatic yes, and it is to be found in a stochastic method called *Q-learning*⁴ due to Watkins (1989). Q-learning is an incremental dynamic programming procedure that determines the optimal policy in a step-by-step manner. It is highly suited for solving Markovian decision problems without explicit knowledge of the transition probabilities. However, successful use of Q-learning hinges on the assumption that the state of the environment is *fully observable*, which in turn means that the environment is a fully observable Markov chain.

We recall from Section 12.4 that the Q-factor $Q(i, a)$ for state-action pair (i, a) is defined by Eq. (12.23), and Bellman's optimality equation is defined by Eq. (12.22). By combining these two equations and using the definition of the immediate expected cost $c(i, a)$ given in Eq. (12.20), we obtain

$$Q^*(i, a) = \sum_{j=1}^N p_{ij}(a) \left(g(i, a, j) + \gamma \min_{b \in \mathcal{A}_i} Q^*(j, b) \right) \quad \text{for all } (i, a) \quad (12.36)$$

which can be viewed as a two-step version of Bellman's optimality equation. The solutions to the linear system of equations (12.36) define the optimal Q-factors $Q^*(i, a)$ uniquely for all state-action pairs (i, a) .

We may use the value iteration algorithm formulated in terms of the Q-factors to solve this linear system of equations. Thus, for one iteration of the algorithm we have

$$Q(i, a) := \sum_{j=1}^N p_{ij}(a) \left(g(i, a, j) + \gamma \min_{b \in \mathcal{A}_i} Q(j, b) \right) \quad \text{for all } (i, a)$$

The small step-size version of this iteration is described by

$$Q(i, a) := (1 - \eta)Q(i, a) + \eta \sum_{j=1}^N p_{ij}(a) \left(g(i, a, j) + \gamma \min_{b \in \mathcal{A}_i} Q(j, b) \right) \quad \text{for all } (i, a) \quad (12.37)$$

where η is a small *learning-rate parameter* that lies in the range $0 < \eta < 1$.

As it stands, an iteration of the value iteration algorithm described in Eq. (12.37) requires knowledge of the transition probabilities. We may eliminate the need for this prior knowledge by formulating a *stochastic* version of Eq. (12.37). Specifically, the averaging performed in an iteration of Eq. (12.37) over all possible states is replaced by a single sample, thereby resulting in the following update for the Q-factor:

$$Q_{n+1}(i, a) = (1 - \eta_n(i, a))Q_n(i, a) + \eta_n(i, a)[g(i, a, j) + \gamma J_n(j)] \quad \text{for } (i, a) = (i_n, a_n) \quad (12.38)$$

where

$$J_n(j) = \min_{b \in \mathcal{A}_i} Q_n(j, b) \quad (12.39)$$

and j is the successor state, and $\eta_n(i, a)$ is the learning-rate parameter at time step n for the state-action pair (i, a) . The update equation (12.38) applies to the current state-action pair (i_n, a_n) , for which $j = j_n$ in accordance with Eq. (12.35). For all other admissible state-action pairs, the Q-factors remain unchanged as shown by

$$Q_{n+1}(i, a) = Q_n(i, a) \quad \text{for all } (i, a) \neq (i_n, a_n) \quad (12.40)$$

Equations (12.38) to (12.40) constitute one iteration of the *Q-learning algorithm*.

Convergence Theorem⁵

Suppose that the learning-rate parameter $\eta_n(i, a)$ satisfies the conditions

$$\sum_{n=0}^{\infty} \eta_n(i, a) = \infty \quad \text{and} \quad \sum_{n=0}^{\infty} \eta_n^2(i, a) < \infty \quad \text{for all } (i, a) \quad (12.41)$$

Then, the sequence of Q-factors $\{Q_n(i, a)\}$ generated by the Q-learning algorithm converges with probability 1 to the optimal value $Q^*(i, a)$ for all state-action pairs (i, a) as the number of iterations n approaches infinity, provided that all state-action pairs are visited infinitely often.

An example of a time-varying learning parameter that guarantees convergence of the algorithm is

$$\eta_n = \frac{\alpha}{\beta + n}, \quad n = 1, 2, \dots \quad (12.42)$$

where α and β are positive numbers.

To sum up, the Q-learning algorithm is a stochastic approximation form of the value iteration policy. It backs up the Q-factor for a *single* state-action pair at each iteration of the algorithm, namely the observed current state and the action actually executed. Most importantly, in the limit, the algorithm converges to the optimal Q-values without forming an explicit model of the underlying Markovian decision processes. Once the optimal Q-values are available, an optimal policy can be determined with relatively little computation using Eq. (12.30).

The convergence of Q-learning to an optimal policy assumes the use of a *look-up table* representation for the Q-factors $Q_n(i, a)$. This method of representation is straightforward and computationally efficient. However, when the input space consisting of state-action pairs is large or the input variables are continuous, the use of a look-up table can be prohibitively expensive due to the requirement for a huge memory. In such a situation, we may resort to the use of a neural network for the purpose of function approximation.

Approximate Q-Learning

Equations (12.38) and (12.39) define the update formulas for the Q-factor for the current state-action pair (i_n, a_n) . This pair of equations may be rewritten in the equivalent form

$$Q_{n+1}(i_n, a_n) = Q_n(i_n, a_n) + \eta_n(i_n, a_n) \left[g(i_n, a_n, j_n) + \gamma \min_{b \in \mathcal{A}_{j_n}} Q_n(j_n, b) - Q_n(i_n, a_n) \right] \quad (12.43)$$

Treating the expression inside the square brackets on the right-hand side of Eq. (12.43) as the error signal involved in updating the current Q-factor $Q_n(i_n, a_n)$, we may identify the target (desired) Q-factor at time step n as:

$$Q_n^{\text{target}}(i_n, a_n) = g(i_n, a_n, j_n) + \gamma \min_{b \in \mathcal{A}_{j_n}} Q_n(j_n, b) \quad (12.44)$$

where $j_n = i_{n+1}$ is the successor state. Equation (12.44) shows that the successor state j_n plays a critical role in determining the target Q-factor. Using this definition of the target Q-factor, we may reformulate the Q-learning algorithm as:

$$Q_{n+1}(i, a) = Q_n(i, a) + \Delta Q_n(i, a) \quad (12.45)$$

where the incremental change in the current Q-factor is defined by

$$\Delta Q_n(i, a) = \begin{cases} \eta_n (Q_n^{\text{target}}(i, a) - Q_n(i, a)) & \text{for } (i, a) = (i_n, a_n) \\ 0, & \text{otherwise} \end{cases} \quad (12.46)$$

By definition, the “optimal” action a_n at the current state i_n is the particular action at that state for which the Q-factor at time step n is minimum. Hence, given the Q-factors $Q_n(i_n, a)$ for admissible actions $a \in \mathcal{A}_{i_n}$ at state i_n , the optimal action a_n for use in Eq. (12.44) is given by

$$Q_n = \min_{a \in \mathcal{A}_{i_n}} Q_n(i_n, a) \quad (12.47)$$

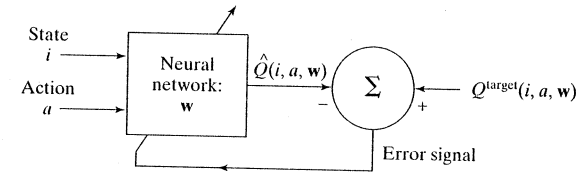


FIGURE 12.9 Neural network layout for approximating the target Q-factor $Q^{\text{target}}(i, a, \mathbf{w})$.

Let $\hat{Q}_n(i_n, a_n, \mathbf{w})$ denote the approximation to the Q-factor $Q_n(i_n, a_n)$, computed by means of a neural network (e.g., multilayer perceptron trained with the back-propagation algorithm). The current state-action pair (i_n, a_n) is input to the neural network with parameter vector $\mathbf{w}$, producing the output $\hat{Q}_n(i_n, a_n, \mathbf{w})$, as illustrated in Fig. 12.9. At each iteration of the algorithm, the weight vector $\mathbf{w}$ of the neural network is changed slightly in a way that brings the output $\hat{Q}_n(i_n, a_n, \mathbf{w})$ closer to the target value $Q_n^{\text{target}}(i_n, a_n)$. However, once the weight vector $\mathbf{w}$ is changed, the target value is itself implicitly affected by the change, assuming the modified value $Q_n^{\text{target}}(i_n, a_n, \mathbf{w})$. There is therefore no guarantee that the distance between these two Q-values is reduced on every iteration. This is also the reason why the approximate Q-learning algorithm has the potential of divergence. If the algorithm does not diverge, the weight vector $\mathbf{w}$ provides a means of storing the approximated Q-factor in the trained neural network, because it outputs $\hat{Q}_n(i_n, a_n, \mathbf{w})$ in response to the input (i_n, a_n) .

Table 12.4 presents a summary of the approximate Q-learning algorithm.

Exploration

In policy iteration, all potentially important parts of the state space should be explored. In Q-learning we have an additional requirement: all potentially profitable actions should be tried as well. In particular, all admissible state-action pairs should be explored often enough to satisfy the convergence theorem. For a greedy policy denoted by μ , only the state-action pairs $(i, \mu(i))$ are explored. Unfortunately, there is no guarantee that all profitable actions would be tried, even if the entire state-space is explored.

What we need is a strategy that expands on Q-learning by providing a compromise between two conflicting objectives (Thrun, 1992):

- *Exploration*, which ensures that all admissible state-action pairs are explored often enough to satisfy the Q-learning convergence theorem.
- *Exploitation*, which seeks to minimize the cost-to-go function by following a greedy policy.

One way to achieve this compromise is to follow a *mixed nonstationary policy* that switches between an auxiliary Markov process and the original Markov process controlled by a stationary greedy policy determined by Q-learning (Cybenko, 1995). The auxiliary process has the following interpretation: The transition probabilities between

TABLE 12.4 Summary of the Approximate Q-Learning Algorithm

1. Start with an initial weight vector $\mathbf{w}_0$, resulting in the Q-factor $Q(i_0, a_0, \mathbf{w}_0)$; the weight vector $\mathbf{w}_0$ refers to a neural network used to perform the approximation.
2. For iteration $n = 1, 2, \dots$, do the following:
 - (a) For the setting $\mathbf{w}$ of the neural network, determine the optimal action

$$a_n = \min_{a \in \mathcal{A}_i} Q_n(i_n, a, \mathbf{w})$$

- (b) Determine the target Q-factor

$$Q_n^{\text{target}}(i_n, a_n, \mathbf{w}) = g(i_n, a_n, j_n) + \gamma \min_{b \in \mathcal{A}_{j_n}} Q_n(j_n, b, \mathbf{w})$$

- (c) Update the Q-factor

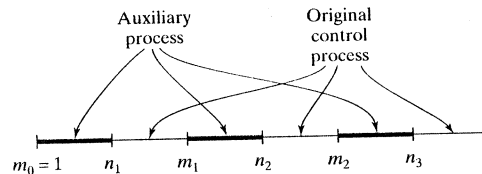
$$Q_{n+1}(i_n, a_n, \mathbf{w}) = Q_n(i_n, a_n, \mathbf{w}) + \Delta Q_n(i_n, a_n, \mathbf{w})$$

where

$$\Delta Q_n(i_n, a_n, \mathbf{w}) = \begin{cases} \eta_n(i_n, a_n) (Q_n^{\text{target}}(i_n, a_n, \mathbf{w}) - Q_n(i_n, a_n, \mathbf{w})), & (i, a) = (i_n, a_n) \\ 0, & \text{otherwise} \end{cases}$$

- (d) Apply (i_n, a_n) as input to the neural network producing the output $\hat{Q}_n(i_n, a_n, \mathbf{w})$ as an approximation to the target Q-factor $Q_n^{\text{target}}(i_n, a_n, \mathbf{w})$. Change the weight vector $\mathbf{w}$ slightly in a way that brings $\hat{Q}_n(i_n, a_n, \mathbf{w})$ closer to the target value $Q_n^{\text{target}}(i_n, a_n, \mathbf{w})$.
 - (e) Go back to step (a) and repeat the computation.

FIGURE 12.10 The time slots pertaining to the auxiliary and original control processes.



possible states are determined by the transition probabilities of the original controlled process with the added ingredient that the corresponding actions are uniformly randomized. The mixture policy starts in any state of the auxiliary process and chooses actions by following it, then switches to the original controlled process, and it goes back and forth in the manner illustrated in Fig. 12.10. The time spent operating on the auxiliary process occupies a fixed number of steps L , say, defined as twice the longest expected time to visit all states of the auxiliary process. The time spent operating on the original controlled process increases progressively with every switch. Let n_k denote the times at which we switch from the auxiliary process to the original controlled process, and m_k denote the times at which we switch back to the auxiliary process, with n_k and m_k defined as follows, respectively:

$$n_k = m_{k-1} + L, \quad k = 1, 2, \dots, \text{ and } m_0 = 1$$

and

$$m_k = n_k + kL, \quad k = 1, 2, \dots$$

The auxiliary process is constructed in such a way that as $k \rightarrow \infty$, there is an infinite number of visits to all states with probability 1, thereby guaranteeing convergence to the optimal Q-factors. Moreover, as $k \rightarrow \infty$, the time spent by the mixed policy operating in the auxiliary process becomes an asymptotically small fraction of the time spent operating in the original controlled process, which in turn means that the mixed policy asymptotically converges to a greedy policy. Hence, given the convergence of the Q-factors to their optimal values, the greedy policy must indeed be optimal provided that the policy becomes greedy slowly enough.

12.9 COMPUTER EXPERIMENT

In this computer experiment we revisit the stagecoach problem considered in Example 12.1. This time we use approximate Q-learning to solve the problem. Two approaches were used to implement the algorithm: One approach used a table to represent the Q-values, and the other approach used a neural network.

Figure 12.11 presents the learning histories for the following Q-factors: $Q(A, \text{up})$, $Q(C, \text{straight})$, $Q(E, \text{straight})$, and $Q(I, \text{up})$ using the table method. The dotted lines in Fig. 12.11 represent the desired Q-values. Each trial was a complete route from state i to the destination in state J . The starting state for each trial was chosen at random. The learning-rate parameter $\eta_n(i, a)$ was defined by

$$\eta_n(i, a) = \frac{\alpha v_n(i, a)}{K + v_n(i, a)}$$

where $v_n(i, a)$ is the number of times the state-action pairs (i, a) was visited up to the current time n , $\alpha = 1.6$, and $K = 600$. After a total of 1000 trials were performed, the optimal route was found to be

$$A \rightarrow D \rightarrow F \rightarrow I \rightarrow J$$

which is recognized to be one of the optimal routes with an overall cost of 11.

Figure 12.12 presents the corresponding results obtained using a multilayer perceptron with two input nodes, 10 hidden neurons, and one output neuron. One of the input nodes represented the state and the other nodes represented the action taken in moving from one state to the next. The output of the multilayer perceptron represented the Q-value computed by the network. The network was trained using the standard back-propagation algorithm. The target Q-value used at time n was calculated using Eq. (12.44). The learning-rate parameter was set at 0.012, and no momentum was used. The network was trained for 10,000 trials for each state-action pair. Figure 12.12 presents the learning histories for the Q-values: $Q(A, \text{up})$, $Q(C, \text{straight})$, $Q(E, \text{straight})$, and $Q(I, \text{up})$. The optimal route found by the network was

$$A \rightarrow D \rightarrow E \rightarrow H \rightarrow J$$

which is also recognized to be one of the optimal routes with a total cost of 11.

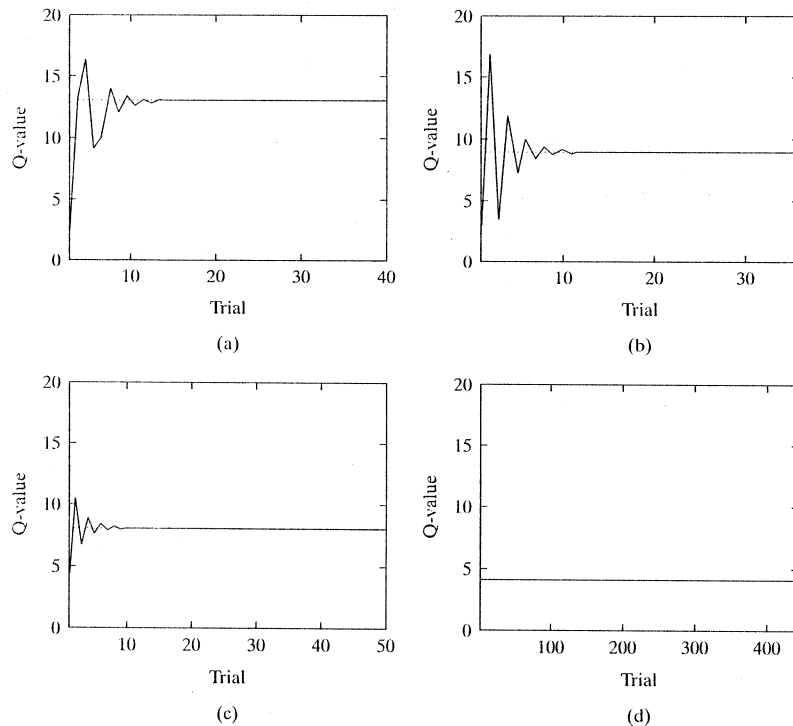


FIGURE 12.11 Learning curves for the stagecoach problem using a look up table. (a) Learning curve for $Q(A, \text{up})$. (b) Learning curve for $Q(C, \text{straight})$. (c) Learning curve for $Q(E, \text{straight})$. (d) Learning curve for $Q(I, \text{up})$.

The computational requirements for the two methods of implementation are summarized as follows:

- (a) Neural network:
 - Number of inputs = 2
 - Number of hidden neurons = 10
 - Number of output neurons = 1
 - Total number of synaptic weights and biases = $2 \times 10 + 10 + 10 \times 1 + 1 = 41$
- (b) Look-up table:
 - Number of states = 10
 - Number of actions = 2 or 3
 - Size of table = 21

In this experiment the number of possible states is small, with the result that the look-up table requires less storage than the neural network. When, however, the number of states is large as in large-scale problems, the neural network usually gains an advantage over the table method in storage requirement.

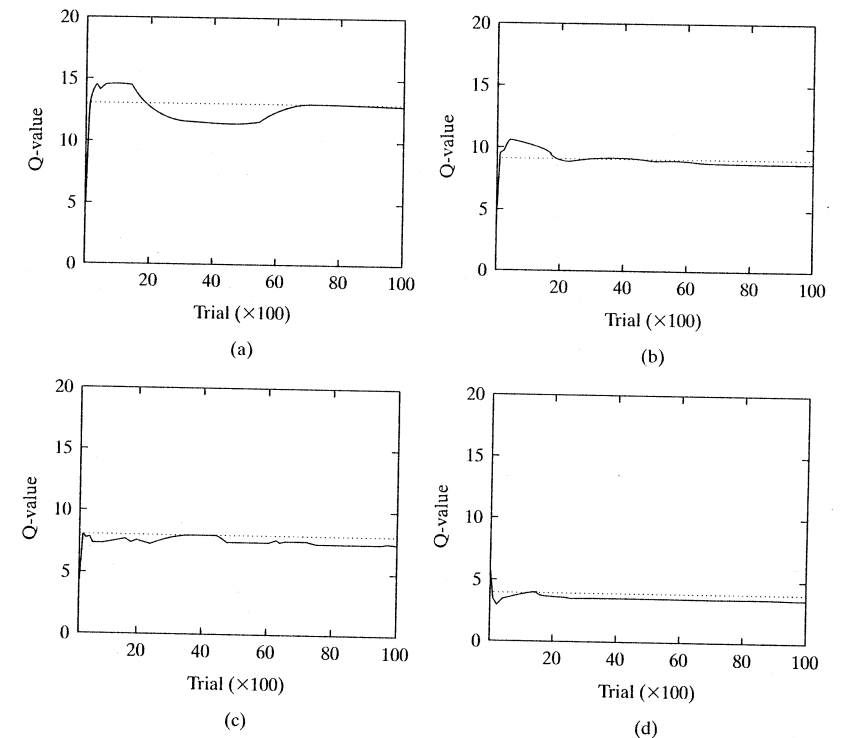


FIGURE 12.12 Learning curves for the stagecoach problem using a neural network. (a) Learning curve for $Q(A, \text{up})$. (b) Learning curve for $Q(C, \text{straight})$. (c) Learning curve for $Q(E, \text{straight})$. (d) Learning curve for $Q(I, \text{up})$.

12.10 SUMMARY AND DISCUSSION

Neurodynamic programming, combining the mathematical formalism of classical dynamic programming and the learning capability of neural networks, offers a powerful approach to the solution of behavioral tasks that require planning. In this modern approach to reinforcement learning, a system learns to do two things: make good decisions by observing its own behavior, and improve its actions through a reinforcement mechanism. The underlying decision-making process follows a Markovian model.

In this chapter we have described two neurodynamic programming procedures:

1. *Approximate policy iteration.* Policy iteration alternates between two basic steps:
 - Policy evaluation, in which the cost-to-go function for the current policy is determined.
 - Policy improvement, in which the current policy is updated to be greedy with respect to the current cost-to-go function.

In approximate policy iteration, simulation and function approximation are combined for the purpose of policy evaluation. To simulate the Markovian model of

the system, knowledge of the state-transition probabilities is required. To perform the function approximation, we may use a neural network (e.g., multilayer perceptron, radial-basis function network, or support vector machine), for which it is well suited because of its universal approximation property.

2. *Approximate Q-learning.* In value iteration, the alternative to policy iteration, a Markovian decision problem is solved by using a successive approximation procedure that converges to the optimal policy. Q-learning is an asynchronous form of value iteration formulated to avoid the need for explicit knowledge of the state-transition probabilities. It offers the following attractive properties:
 - Q-learning converges to the optimal Q-factors with probability 1, provided that all state-action pairs are visited infinitely often and the learning-rate parameters satisfy the conditions specified in Eq. (12.41).
 - Q-learning directly updates estimates of the Q-factors associated with an optimal policy, and thereby avoids the multiple policy evaluation steps involved in policy iteration.

In approximate Q-learning, a neural network is used to approximate estimates of the Q-factors in order to avoid the need for excessive memory requirement when the number of possible states is large. In short, approximate Q-learning is a simulation-based algorithm for solving a Markovian decision problem when a model of the system is unavailable and memory requirement is at a premium. Of course, it can be applied even if a model of the system is available, in which case it provides an alternative to approximate policy iteration.

Neurodynamic programming techniques are particularly effective in solving large-scale problems where planning is of major concern. Traditional approaches to dynamic programming are hardly applicable to problems of this kind because of the enormous size of the state space that would have to be explored. Indeed, neurodynamic programming has been successfully applied to solve difficult real-world problems in many diverse fields, which include backgammon (Tesauro, 1989, 1994), combinatorial optimization (Bertsekas and Tsitsiklis, 1996), elevator dispatching (Crites and Barto, 1996), and dynamic channel allocation (Singh and Bertsekas, 1997; Nic and Haykin, 1996, 1998). In what follows, the application to backgammon is described in some detail.

The development of a neural network-based computer program player to play *backgammon*, first reported in Tesauro (1989) and subsequently improved in Tesauro (1994), is a particularly impressive success story that has been a source of motivation for research in neurodynamic programming. Backgammon is an ancient two-player board game. It is played along an effectively one-dimensional track. The players take turns rolling a pair of dice, and accordingly move their checkers in opposite directions along the track. The legal moves made by each player depend on the outcome of the dice roll and the board configuration. The first player to move all of his or her checkers all the way forward to the ultimate end of the board is the winner. The game can be modeled as a Markov decision process, with a state being defined by a description of the board configuration, the outcome of the dice roll, and the identity of the player making a move. The first version of the neuro-backgammon built by Tesauro (1989) used supervised learning. It was able to learn at a strong intermediate level, given only a "raw" description of the state. Perhaps the most interesting finding reported was a

good scaling behavior, in the sense that as the size of the neural network and amount of training experience were increased, substantial improvements in performance were observed. The neural network used in the study was a multilayer perceptron (MLP) trained with the back-propagation algorithm. The best performance was obtained using an MLP with 40 hidden neurons, and the training was performed over a total of 200,000 games. In a subsequent study reported in Tesauro (1994), a form of policy iteration called *optimistic TD*(λ) was used to train the neural network; TD stands for *temporal difference learning* due to Sutton (1988). Optimistic TD(λ) is a simulation-based method for approximating the cost-to-go function J^μ , in which the policy μ is replaced with a new policy μ that is greedy with respect to the approximation to J^μ at every state transition (Bertsekas and Tsitsiklis, 1996). The computer program based on this neurodynamic programming method is commonly referred to as *TD-gammon*. Handcrafted functions of the state (i.e., features) were added by Tesauro to the neural network's input representation, enabling TD-gammon to play at a strong master level extremely close to the world's best human player. Among the indications contributing to this assessment are numerous tests of TD-gammon playing against several world class human grandmasters (Tesauro, 1995).

NOTES AND REFERENCES

1. The classical approach to reinforcement learning is rooted in psychology, going back to the early work of Thorndike (1911) on animal learning and that of Pavlov (1927) on conditioning. Contributions to classical reinforcement learning also include the work of Widrow et al. (1973); in that paper, the notion of a *critic* was introduced. Classical reinforcement learning is discussed in book form in Hampson (1990).
Major contributions to modern reinforcement learning include the works of Samuel (1959) on his celebrated checkers playing program, Barto et al. (1983) on adaptive critic systems, Sutton (1988) on temporal difference methods, and Watkins (1989) on Q-learning. The handbook of intelligent control by White and Sofge (1992) presents material on optimal control by White and Jordan, reinforcement learning and adaptive critic methods by Barto, and heuristic dynamic programming by Werbos.
Bertsekas and Tsitsiklis (1996) present the first treatment of modern reinforcement learning in book form. For a historical account of reinforcement learning, see Sutton and Barto (1998).
2. Dynamic programming was developed by R.E. Bellman in the late 1950s; see Bellman (1957), Bellman and Dreyfus (1962). For a detailed exposition of the subject, see the two-volume book by Bertsekas (1995b).
3. Policy iteration and value iteration are two principal methods of dynamic programming. There are two other dynamic programming methods that deserve to be mentioned: the *Gauss-Seidel method* and *asynchronous dynamic programming* (Barto et al., 1995; Bertsekas, 1995b). In the Gauss-Seidel method, the cost-to-go function is updated at one state at a time in a sequential sweep of all the states, with the competition for each state based on the most recent costs of the other states. Asynchronous dynamic programming differs from the Gauss-Seidel method in that it is not organized in terms of systematic successive sweeps of the set of states.
4. On page 96 of his Ph.D. thesis, Watkins (1989) makes the following remarks on Q-learning:
"Appendix 1 presents a proof that this learning method does work for finite Markov decision processes. The proof also shows that the learning method will

converge rapidly to the optimal action-value function. Although this is a very simple idea, it has not, as far as I know, been suggested previously. However, it must be said that finite Markov decision processes and stochastic dynamic programming have been extensively studied for use in several different fields for over thirty years, and it is unlikely that nobody has considered the Monte-Carlo method before."

In a footnote commentary on these remarks, Barto et al. (1995) point out that although the idea of assigning values to state-action pairs formed the basis of the approach to dynamic programming taken in Denardo (1967), they have not seen algorithms like Q-learning for estimating these values that predate Watkins' 1989 thesis.

5. The outline of a proof of the convergence theorem for Q-learning was presented in Watkins (1989), which was refined later in Watkins and Dayan (1992). More general results on the convergence of Q-learning were presented in Tsitsiklis (1994); see also Bertsekas and Tsitsiklis (1996).

PROBLEMS

Bellman's optimality criterion

- 12.1 When the discount factor γ approaches 1, computation of the cost-to-go function in Eq. (12.22) becomes longer. Why? Justify your answer.
- 12.2 In this problem we present another proof of Bellman's optimality equation (12.22) due to Ross (1983).
- (a) Let π be any arbitrary policy, and suppose that π chooses action a at time step 0 with probability p_a and $a \in \mathcal{A}_i$. Then,

$$J^\pi(i) = \sum_{a \in \mathcal{A}_i} p_a \left(c(i, a) + \sum_{j=1}^N p_{ij}(a) W^\pi(j) \right)$$

where $W^\pi(j)$ represents the expected cost-to-go function from time step 1 onward, given that policy π is being used and that j is the state at time step 1. Hence, show that

$$J^\pi(i) \leq \max_{a \in \mathcal{A}_i} \left(c(i, a) + \gamma \sum_{j=1}^N p_{ij}(a) J(j) \right)$$

where

$$W^\pi(j) \leq \gamma J(j)$$

- (b) Let π be the policy that chooses action a_0 at time step 0 and, if the next state is j , it views the process as originating in state j , following a policy π_j such that

$$J^{\pi_j}(j) \geq J(j) - \epsilon$$

where ϵ is a small positive number. Hence, show that

$$J(i) \geq \max_{a \in \mathcal{A}_i} \left(c(i, a) + \gamma \sum_{j=1}^N p_{ij}(a) J(j) \right) - \gamma \epsilon$$

- (c) Using the results derived in parts (a) and (b), prove Eq. (12.22).

- 12.3 Equation (12.22) represents a linear system of N equations, with one equation per state. Let

$$\begin{aligned} \mathbf{J}^\pi &= [J^\pi(1), J^\pi(2), \dots, J^\pi(N)]^T \\ \mathbf{c}(\mu) &= [c(1, \mu), c(2, \mu), \dots, c(N, \mu)]^T \\ \mathbf{P}(\mu) &= \begin{bmatrix} p_{11}(\mu) & p_{12}(\mu) & \dots & p_{1N}(\mu) \\ p_{21}(\mu) & p_{22}(\mu) & \dots & p_{2N}(\mu) \\ \vdots & \vdots & & \vdots \\ p_{N1}(\mu) & p_{N2}(\mu) & \dots & p_{NN}(\mu) \end{bmatrix} \end{aligned}$$

Show that Eq. (12.22) may be reformulated in the equivalent matrix form:

$$(\mathbf{I} - \gamma \mathbf{P}(\mu)) \mathbf{J}^\mu = \mathbf{c}(\mu)$$

where $\mathbf{I}$ is the identity matrix. Comment on the uniqueness of the vector $\mathbf{J}^\mu$ representing the cost-to-go functions for the N states.

- 12.4 In Section 12.3 we derive the dynamic programming algorithm for a finite-horizon problem. In this problem we rederive this algorithm for a discounted problem for which the cost-to-go function is defined by

$$J^\mu(X_0) = \lim_{K \rightarrow \infty} \left[\sum_{n=0}^{K-1} \gamma^n g(X_n, \mu(X_n), X_{n+1}) \right]$$

In particular, show that

$$J_K(X_0) = \min_{\mu} E_{X_1} [g(X_0, \mu(X_0), X_1) + \gamma J_{K-1}(X_1)]$$

Policy iteration

- 12.5 In Section 12.4 we say that the cost-to-go function satisfies the statement

$$J^{\mu_{n+1}} \leq J^{\mu_n}$$

Justify this statement.

- 12.6 Discuss the significance of the statement described in Eq. (12.25).
- 12.7 Using a two-dimensional picture, illustrate the interaction between the policy update and policy evaluation in the policy iteration algorithm.

Value iteration

- 12.8 A dynamic programming problem involves a total of N possible states and M admissible actions. Assuming the use of a stationary policy, show that a single iteration of the value iteration algorithm requires on the order of $N^2 M$ operations.
- 12.9 Table 12.2 presents a summary of the value iteration algorithm formulated in terms of the cost-to-go function $J^\mu(i)$ for states $i \in \mathcal{X}$. Reformulate this algorithm in terms of the Q-factors $Q(i, a)$.
- 12.10 Policy iteration always terminates finitely, whereas value iteration may require an infinite number of iterations. Discuss other differences between these two methods of dynamic programming.

Q-learning

- 12.11 Show that

$$J^*(i) = \min_{a \in \mathcal{A}_i} Q(i, a)$$

- 12.12 The Q -learning algorithm is sometimes referred to as an adaptive form of the value iteration policy. Justify the validity of this description.
- 12.13 Construct a signal-flow graph for the approximate Q -learning algorithm summarized in Table 12.4.
- 12.14 The approximate Q -learning algorithm summarized in Table 12.4 assumes lack of knowledge of the state-transition probabilities. Reformulate this algorithm assuming the availability of these probabilities.

CHAPTER 13

Temporal Processing Using Feedforward Networks

13.1 INTRODUCTION

Time constitutes an essential ingredient of the learning process. It can be continuous or discrete. Whatever its form, time is an ordered entity that is basic to many of the cognitive tasks encountered in practice such as vision, speech, signal processing, and motor control. It is through the incorporation of time into the operation of a neural network that it is enabled to follow statistical variations in nonstationary processes such as speech signals, radar signals, signals picked up from the engine of an automobile, and fluctuations in stock market prices, just to mention a few. The question is: How do we build time into the operation of a neural network? The answer to this fundamental question lies in one of two possible ways:

- *Implicit representation.* Time is represented by the effect it has on signal processing in an implicit manner.¹ For example, the input signal is *uniformly sampled*, and the sequence of synaptic weights of each neuron connected to the input layer of the network is *convolved* with a different sequence of input samples. In so doing, the temporal structure of the input signal is embedded in the spatial structure of the network.
- *Explicit representation.* Time is given its own particular representation.² For example, the echo-location system of a bat operates by emitting a short frequency modulated (FM) signal, so that the same intensity level is maintained for each frequency channel restricted to a very short period within the FM sweep. Multiple comparisons between several different frequencies encoded by an array of auditory receptors are made for the purpose of extracting accurate distance (range) information about a target (Suga and Kanwal, 1995). When an echo is received from the target with an unknown delay, a neuron (in the auditory system) with a matching delay line responds, thereby providing an estimate of the range to the target.

In this chapter we are concerned with the implicit representation of time, whereby a "static" neural network (e.g., multilayer perceptron) is provided with *dynamic* properties. This, in turn, makes the network responsive to the temporal structure of information-bearing signals.

For a neural network to be dynamic, it must be given *memory*. As pointed out in Chapter 2, memory may be divided into "short-term" and "long-term" memory, depending on the retention time. Long-term memory is built into a neural network through supervised learning, whereby the information content of the training data set is stored (in part or in full) in the synaptic weights of the network. However, if the task at hand has a temporal dimension, we need some form of short-term memory to make the network dynamic. One simple way of building short-term memory into the structure of a neural network is through the use of *time delays*, which can be implemented at the synaptic level inside the network or at the input layer of the network. The use of time delays in neural networks is neurobiologically motivated, since it is well known that signal delays are omnipresent in the brain and play an important role in neurobiological information processing (Braitenberg, 1967, 1977, 1986; Miller, 1987).

Organization of the Chapter

The material in this chapter is organized in three parts. The first part, consisting of Sections 13.2 and 13.3, deals with network structures and models. In Section 13.2 we present a discussion of memory structures, followed by Section 13.3 on a description of two different network architectures for temporal processing of signals.

The second part of the chapter, consisting of Sections 13.4 to 13.6, deals with a class of neural networks known as focused time lagged feedforward networks; the term "focused" refers to the fact that the short-term memory is located entirely at the front end of the network. A computer experiment on this structure is described in Section 13.6.

The third part of the chapter, consisting of Sections 13.7 to 13.9, deals with distributed time lagged feedforward networks, where delay lines are distributed throughout the network. Section 13.7 describes spatio-temporal models of a neuron, followed by a discussion in Section 13.8 on the second class of neural networks just mentioned. In Section 13.9 we describe the "temporal" back-propagation algorithm for the supervised training of distributed time lagged feedforward networks.

The chapter concludes with some final remarks in Section 13.10.

SHORT-TERM MEMORY STRUCTURES

The primary role of memory is to *transform a static network into a dynamic one*. In particular, by embedding memory into the structure of a static network such as an ordinary multilayer perceptron, the output of the network becomes a function of time. This approach for building a nonlinear dynamical system is straightforward because it provides for a clear separation of responsibilities: The static network accounts for nonlinearity, and the memory accounts for time.

Short-term memory³ can be implemented in continuous time or in discrete time. Continuous time is denoted by t , and discrete time is denoted by n . The resistance-

capacitance circuit in Fig. 13.1 is an example of continuous-time memory, which is characterized by an impulse response (i.e., memory trace) $h(t)$ that decays exponentially with time t . This circuit is responsible for memory, at the synaptic level, in an analog implementation of the additive model of a neuron to be described later in the chapter. In this section we are concerned mainly with discrete-time memory.

A useful tool for dealing with discrete-time systems is the *z-transform*. Let $\{x(n)\}$ denote a discrete-time sequence, which may extend into the infinite past. The *z-transform* of this sequence, denoted by $X(z)$, is defined by

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n} \quad (13.1)$$

where z^{-1} is the *unit delay operator*; that is, z^{-1} operating on $x(n)$ yields its delayed version $x(n-1)$. Suppose $x(n)$ is applied to a discrete-time system of impulse response $h(n)$. The output of the system, $y(n)$, is defined by the *convolution sum*

$$y(n) = \sum_{k=-\infty}^{\infty} h(k)x(n-k) \quad (13.2)$$

For $x(n)$ equal to the unit impulse, $y(n)$ reduces to the *impulse response* $h(n)$ of the system. An important property of the *z-transform* is that *convolution in the time domain is transformed into multiplication in the z-domain* (Oppenheim and Schaffer, 1989; Haykin and Van Veen, 1998). If we denote the *z-transform* of the sequences $\{h(n)\}$ and $\{y(n)\}$ by $H(z)$ and $Y(z)$, respectively, application of the *z-transform* to Eq. (13.2) yields

$$Y(z) = H(z)X(z) \quad (13.3)$$

or equivalently

$$H(z) = \frac{Y(z)}{X(z)} \quad (13.4)$$

The function $H(z)$ is called the *transfer function* of the system.

Figure 13.2 shows a block diagram of a discrete-time memory consisting of p identical sections connected in cascade; hereafter p is referred to as the *order* of the

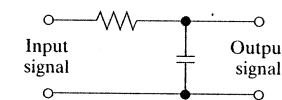


FIGURE 13.1 Resistance-capacitance circuit.

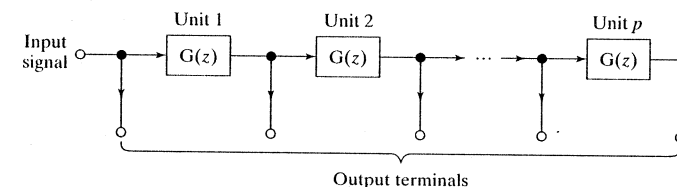


FIGURE 13.2 Generalized tapped delay line memory of order p .