Chapter 12.
Dynamic Programming

Neural Networks and Learning Machines (Haykin)

Lecture Notes on
Self-learning Neural Algorithms

Byoung-Tak Zhang
School of Computer Science and Engineering
Seoul National University

Version: 20171011/20191017
12.1 Introduction (1/2)

Two paradigms of learning
1. Learning with a teacher: Supervised learning
2. Learning without a teacher: Unsupervised learning / reinforcement learning / semisupervised learning

Reinforcement learning
1. Behavioral learning (action, sequential decision-making)
2. Interaction between an agent and its environment
3. Achieving a specific goal under uncertainty

Two approaches to reinforcement learning
1. Classical approach: punishment and reward (classical conditioning), highly skilled behavior
2. Modern approach: dynamic programming, planning
Dynamic programming (DP)

- 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.
- Decisions cannot be made in isolation, but the desire for a low cost at the present must be balanced against the undesirability of high cost in the future.
- Credit or blame must be assigned to each one of a set of interacting decisions (credit assignment problem)
- Decision making by an agent that operates in a stochastic environment
- How can an agent or decision maker improve its long-term performance in a stochastic environment when the attainment of this improvement may require having to sacrifice short-term performance?
- Markov decision process

Right balance between

- Realistic description of a given problem (practical)
- Power of analytic and computational methods to apply to the problem (theoretical)
12.2 Markov Decision Process (1/3)

**Markov decision process (MDP):**
- a finite set of discrete states
- a finite set of possible actions
- cost (or reward)
- discrete time

The **state** of the environment is a summary of the entire past experience of an agent gained from its interaction with the environment, such that the information necessary for the agent to predict the future behavior of the environment is contained in that summary.

\[
\text{MDP} = \text{The sequence of states } \{X_n, \ n=0,1,2,\ldots\}
\]

i.e., a Markov chain with transition probabilities \(p_{ij}(\mu(i))\) for actions \(\mu(i)\).

Figure 12.1 An agent interacting with its environment.
12.2 Markov Decision Process (2/3)

\[ i, j \in X: \text{states} \]
\[ A_i = \{a_{ik}\}: \text{actions} \]
\[ \pi = \{\mu_0, \mu_1, \ldots\}: \text{policy (states } X \text{ to actions } A) \]
\[ \mu_n(i) \in A_i \quad \text{for all states } i \]

Nonstationary policy: \( \pi = \{\mu_0, \mu_1, \ldots\} \)
Stationary policy: \( \pi = \{\mu, \mu, \ldots\} \)
\[ p_{ij}(a): \text{transition probability} \]
\[ p_{ij}(a) = P(X_{n+1} = j | X_n = i, A_n = a) \]
1. \( p_{ij}(a) \geq 0 \quad \text{for all } i \text{ and } j \)
2. \( \sum_j p_{ij}(a) = 1 \quad \text{for all } i \)
\[ g(i, a_{ik}, j): \text{cost function} \]
\[ \gamma: \text{discount factor} \]
\[ \gamma^n g(i, a_{ik}, j): \text{discounted cost} \]

Figure 12.2 Illustration of two possible transitions: The transition from state to state is probabilistic, but the transition from state to is deterministic.
12.2 Markov Decision Process (3/3)

Dynamic programming (DP) problem
- Finite-horizon problem
- Infinite-horizon problem

Cost-to-go function (total expected cost)

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

\[ g(X_n, \mu_n(X_n), X_{n+1}) \]: observed cost

Optimal value

\[ J^*(i) = \min_{\pi} J^\pi(i) \quad (\text{For stationary policy: } J^\mu(i) = J^*(i)) \]

Basic problem in DP

Given a stationary MDP, find a stationary policy \( \pi \) that minimizes the cost-to-go function \( J^\mu(i) \) for all initial states \( i \).

Notation:

Cost function \( J(i) \)
\( \Leftrightarrow \) Value function \( V(s) \)

Cost \( g(.) \)
\( \Leftrightarrow \) Reward \( r(.) \)
12.3 Bellman’s Optimality Criterion (1/3)

**Principle of optimality**

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

Consider a finite-horizon problem for which the cost-to-go function is:

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

Suppose we wish to minimize the cost-to-go function

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

Then, the truncated policy \( \{\mu_n^*, \mu_{n+1}^*, \ldots, \mu_{K-1}^*\} \) is optimal for the subproblem.
12.3 Bellman’s Optimality Criterion (2/3)

Dynamic programming algorithm

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 algorithm

$$J_n(X_n) = \min_{\mu_n} \mathbb{E} \left[ g_n(X_n, \mu_n(X_n), X_{n+1}) + J_{n+1}(X_{n+1}) \right]$$  \hspace{1cm} (12.13)

which runs backward in time, with

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

Furthermore, if $\mu_n^*$ minimizes the right-hand side of Eq. (12.13) for each $X_n$ and $n$, then the policy $\pi^* = \{\mu_0^*, \mu_1^*, ..., \mu_{K-1}^*\}$ is optimal.
Bellman's optimality equation

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

Immediate expected cost

\[ c(i, \mu(i)) = \mathbb{E}_{X_1} [g(i, \mu(i), X_1)] = \sum_{j=1}^{N} p_{ij} g(i, \mu(i), j) \]

Two methods for computing an optimal policy

- Policy iteration
- Value iteration
12.4 Policy Iteration (1/2)

\[ Q^\mu(i,a) = c(i,a) + \gamma \sum_{j=1}^{n} p_{ij}(a) J^\mu(j) \]  \hspace{1cm} (Q-factor)

1. **Policy evaluation step**: 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**: the current policy is updated in order to be greedy with respect to the cost-to-go function computed in step 1.

Figure 12.3 Policy iteration algorithm.
12.4 Policy Iteration (2/2)

TABLE 12.1 Summary of the Policy Iteration Algorithm

1. Start with an arbitrary initial policy $\mu_0$.
2. For $n = 0, 1, 2, \ldots$, compute $J^{\mu_n}(i)$ and $Q^{\mu_n}(i, a)$ for all states $i \in \mathcal{S}$ and actions $a \in A_i$.
3. For each state $i$, compute

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

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

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

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

(c) 2017 Biointelligence Lab, SNU
12.5 Value Iteration (1/4)

**TABLE 12.2  Summary of the Value Iteration Algorithm**

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

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

   Continue this computation until

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

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

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

3. Compute the $Q$-factor

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

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

   $$\mu^*(i) = \arg \min_{a \in \mathcal{A}_i} Q^*(i, a)$$
Figure 12.4  Illustrative backup diagrams for (a) policy iteration and (b) value iteration.

Take the maximum over all possible state-action pairs
12.5 Value Iteration (3/4)

Figure 12.5  Flow graph for stagecoach problem.
12.5 Value Iteration (4/4)

Figure 12.6  Steps involved in calculating the $Q$-factors for the stagecoach problem. The routes (printed in blue) from $A$ to $J$ are the optimal ones.
12.6 Approximate Dynamic Programming: Direct Methods

• Dynamic programming (DP) requires an explicit model, i.e. transition probabilities.

• **Approximate DP**: We may use Monte Carlo simulation to explicitly estimate (i.e. approximate) the transition probabilities.
  1. Direct methods
  2. Indirect methods: Approximate policy evaluation, approximate cost-to-go

• **Direct methods for approximate DP**
  1. Value iteration: **Temporal-difference (TD) learning**
  2. Policy iteration: **Q-learning**

• **Reinforcement learning as the direct approximation of DP**

(c) 2017 Biointelligence Lab, SNU
12.7 Temporal-Difference Learning (1/3)

Sequence of states: \( \{i_n\}_{n=0}^{N} \)

Cost-to-go function for Bellman equation

\[
J^\mu(i_n) = E\left[ g(i_n, i_{n+1}) + J^\mu(i_{n+1}) \right], \quad n = 0, 1, \ldots, N-1
\]

Applying the Robbins - Monro stochastic approximation

\[
r^+ = (1 - \eta)r + \eta g(r, \bar{v})
\]

we have

\[
J^+(i_n) = (1 - \eta)J(i_n) + \eta \left[ g(i_n, i_{n+1}) + J(i_{n+1}) \right]
\]

\[
= J(i_n) + \eta \left[ g(i_n, i_{n+1}) + J(i_{n+1}) - J(i_n) \right]
\]

**Temporal difference (TD)**

\[
d_n = g(i_n, i_{n+1}) + J(i_{n+1}) - J(i_n), \quad n = 0, 1, \ldots, N-1
\]

**TD learning**

\[
J^+(i_n) = J(i_n) + \eta d_n
\]
12.7 Temporal-Difference Learning (2/3)

Monte Carlo simulation algorithm

\[
J^\mu(i_n) = \mathbb{E}\left[ \sum_{k=0}^{N-n-1} g(i_{n+k}, i_{n+k+1}) \right], \quad n = 0, 1, \ldots, N-1
\]

Robbins-Monro stochastic approximation

\[
J^*(i_n) = J(i_n) + \mu_k \left( \sum_{k=0}^{N-n-1} g(i_{n+k}, i_{n+k+1}) - J(i_n) \right)
\]

\[
= J(i_n) + \mu_k [g(i_{n+1}, i_{n+1}) + J(i_{n+1}) - J(i_n)]
\]

\[
+ g(i_{n+1}, i_{n+2}) + J(i_{n+2}) - J(i_{n+1})
\]

\[
+ g(i_{n+2}, i_{n+2}) + J(i_{n+2}) - J(i_{n+1})
\]

Using the temporal difference

\[
J^*(i_n) = J(i_n) + \sum_{k=0}^{N-n-1} d_{n+k}
\]

To justify this is an iterative implementation of Monte Carlo simulation:

Total sum of the cost for traj \(\{i_n, i_{n+1}, \ldots, i_N\}\)

\[
c(i_n) = \sum_{k=0}^{N-n-1} g(i_{n+k}, i_{n+k+1}), \quad n = 0, \ldots, N-1
\]

Cost-to-go after visiting \(i_n\) for \(T\) simulations

\[
J(i_n) = \frac{1}{T} \sum_{n=1}^{T} c(i_n)
\]

Ensemble averaged cost-to-go

\[
J^\mu(i_n) = \mathbb{E}[c(i_n)] \quad \text{for all } n
\]

Iterative formula

\[
J^+(i_n) = J(i_n) + \eta_n (c(i_n) - J(i_n))
\]
12.7 Temporal-Difference Learning (3/3)

**TD(λ)**

\[
J^+(i_n) = J(i_n) + \eta \sum_{k=n}^{\infty} \lambda^{k-n} d_k
\]

\[
d_k = g(i_k, i_{k+1}) + J(i_{k+1}) - J(i_k)
\]

\[
\lambda = 0: \\
J^+(i_n) = J(i_n) + \eta d_n
\]

\[
\lambda = 1: \\
J^+(i_n) = J(i_n) + \eta \sum_{k=0}^{N-n-1} d_{n+k}
\]

**TD methods**

- Online prediction methods that learn how to compute their estimates, partly, on the basis of other estimates.
- Boostrapping methods
- Do not require a model of the environment
12.8 Q-Learning (1/3)

Two-step version of Bellman's optimality equation

\[ Q^*(i,a) = \sum_{j=1}^{N} p_{ij}(a) \left( g(i,a,j) + \gamma \min_{b \in A_j} Q^*(j,b) \right) \text{ for all } (i,a) \]

The value iteration algorithm

\[ Q^*(i,a) = \sum_{j=1}^{N} p_{ij}(a) \left( g(i,a,j) + \gamma \min_{b \in A_j} Q(j,b) \right) \text{ for all } (i,a) \]

Small-step size version

\[ Q^*(i,a) = (1 - \eta)Q(i,a) + \eta \sum_{j=1}^{N} p_{ij}(a) \left( g(i,a,j) + \gamma \min_{b \in A_j} Q(j,b) \right) \text{ for all } (i,a) \]

Stochastic version based on a single sample

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

\[ J_n(j) = \min_{b \in A_j} Q_n(j,b) \]

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

**Q-learning algorithm:**

\[ Q_{n+1}(i,a) = Q_n(i,a) + \eta_n(i,a) \left[ g(i,a,j) + \gamma \min_{b \in A_j} Q_n(j,b) - Q_n(i,a) \right] \]

Is there any on-line procedure for learning an optimal control policy through experience that is gained solely on the basis of observing samples?

\[ s_n = (i_n, a_n, j_n, g_n) \]

\[ g_n = g(i_n, a_n, j_n) \]
Convergence Theorem

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

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

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.

Time-varying learning parameter: \( \eta_n = \frac{\alpha}{\beta + n}, \quad n = 1, 2, \ldots \)

Q-learning algorithm may be viewed in one of two equivalent ways

- Robins-Monro stochastic approximation algorithm, or
- combination of value iteration and Monte Carlo simulation.

Compromise between two conflicting objectives in reinforcement learning

- Exploration: ensures that all admissible state–action pairs are explored often enough to satisfy the Q-learning convergence theorem
- Exploitation: seeks to minimize the cost-to-go function by following a greedy policy.
Mixed nonstationary policy
switches between an auxiliary Markov process and the original Markov process controlled by a stationary greedy policy determined by Q-learning

\[ n_k = m_{k-1} + L, \ k = 1, 2, ..., \text{and} \ m_0 = 1 \]

\[ m_k = n_k + kL, \ k = 1, 2, ... \]
12.9 Approximate DP: Indirect Methods (1/2)

Indirect approach to approximate DP:

- Rather than explicitly estimate the transition probabilities and associated transition costs, use Monte Carlo simulation to generate one or more system trajectories, so as to approximate the cost-to-go function of a given policy, or even the optimal cost-to-go function, and then optimize the approximation in some statistical sense.

- Thus, having abandoned the notion of optimality, we may capture the goal of the indirect approach to approximate dynamic programming in the following simple statement: *Do as well as possible, and not more.*

- Performance optimality is traded off for computational optimality. This strategy is precisely what the human brain does on a daily basis: *Given a difficult decision-making problem, the brain provides a suboptimal solution that is the “best” in terms of reliability and available resource allocation.*

Goal of approximate DP:

Find a function $\tilde{J}(i, w)$ that approximates the optimal cost-to-go function $J^*(i)$ for state $i$, such that the cost difference $J^*(i) - \tilde{J}(i, w)$ is minimized according to some statistical criterion.

Two basic questions

**Question 1:** How do we choose the approximation function $\tilde{J}(i, w)$ in the first place?

**Question 2:** Having chosen an appropriate approximation function $\tilde{J}(i, w)$, how do we adapt the weight vector $w$ so as to provide the “best fit” to Bellman's equation of optimality
12.9 Approximate DP: Indirect Methods (2/2)

Approximate DP
1. Linear approach
   \[ \tilde{J}(i, w) = \sum_j \phi_{ij} w_j = \phi_i^T w \] for all \( i \)
2. Nonlinear approach
   - Recurrent multilayer perceptrons (deep architectures)
   - Supervised training of a recurrent multilayer perceptron by nonlinear sequential-state estimation algorithm that is derivative free.

Figure 12.8 Architectural layout of the linear approach to approximate dynamic programming.
12.10 Least-Squares Policy Evaluation (1/4)

Perform value iteration within a lower dimensional subspace spanned by a set of basis functions.

\[
\phi = \begin{bmatrix}
\phi_1^T \\
\phi_2^T \\
\vdots \\
\phi_n^T
\end{bmatrix} \quad \phi w_{n+1} = \Pi \phi \phi w_n + \text{(simulation noise)}
\]

\[
J(i) = E \left[ \sum_{n=0}^{\infty} \gamma^n g(i_n, i_{n+1}) \mid i_0 = i \right] \quad J(i) \approx \tilde{J}(i, w) = \phi^T(i)w \quad S = \left\{ \phi w \mid w \in \mathbb{R}^S \right\}
\]

1. **The Markov chain has positive steady-state probabilities;** that is,

\[
\lim_{n \to \infty} \frac{1}{n} \sum_{k=1}^{n} P(i_k = j \mid i_0 = i) = \pi_j > 0 \quad \text{for all } i
\]

The implication of this assumption is that the Markov chain has a single recurrent class with no transient states.

2. **The rank of matrix } \phi is } s.

The implication of this second assumption is that the columns of the feature matrix } \phi and therefore the basis functions represented by } \phi w, are linearly independent.
12.10 Least-Squares Policy Evaluation (2/4)

\[ TJ(i) = \sum_{j=1}^{N} p_{ij}(g(i, j) + \gamma J(i)), \quad i = 1, 2, \ldots, N \]

\[ g = \begin{bmatrix} \sum_j p_{1j}g(1, j) \\ \sum_j p_{2j}g(2, j) \\ \vdots \\ \sum_j p_{Nj}g(N, j) \end{bmatrix} \]

\[ P = \begin{pmatrix} p_{11} & \cdots & p_{1N} \\ \vdots & \ddots & \vdots \\ p_{N1} & \cdots & p_{NN} \end{pmatrix} \quad J = \begin{bmatrix} J(1) \\ J(2) \\ \vdots \\ J(N) \end{bmatrix} \approx \Phi W \]

\[ T J_n = g + \gamma P J \]

\[ J_{n+1} = T J_n \]

\[ \Phi W_{n+1} = \Pi T (\Phi W_n), \quad n = 0, 1, 2, \ldots \]

\[ \Pi: \text{projection onto subspace } S \]

Figure 12.9 Projected value iteration (PVI) method.

**Projected value iteration (PVI) for policy evaluation:**

At iteration \( n \), the current iterate \( \Phi W_n \) is operated on by the mapping \( T \) and the new vector \( T(\Phi W_n) \) is projected onto the subspace \( S \), thereby yielding the updated iterate \( \Phi W_{n+1} \).
12.10 Least-Squares Policy Evaluation (3/4)

From projected value iteration to least-square policy evaluation (LSPE)

Least-squares minimization for the projection \( \Pi \), i.e. for \( \phi w_{n+1} = \Pi T(\phi w_n) \)

\[
  w_{n+1} = \arg \min_w \left\| \phi w - T(\phi w_n) \right\|_\pi^2
\]

Least-squares version of PVI algorithm

\[
  w_{n+1} = \arg \min_w \sum_{i=1}^{N} \pi_i \left( \phi^T(i)w - \left( \sum_{j=1}^{N} p_{ij} g(i, j) + \gamma \phi^T(j)w_n \right) \right)
\]

Use Monte-Carlo by generating a trajectory \( (i_0, i_1, i_2, ...) \) for state \( i \) and updating \( w_n \) after each iteration \( (i_n, i_{n+1}) \)

\[
  w_{n+1} = \arg \min_w \sum_{k=1}^{n} \left( \phi^T(i_k)w - g(i_k, i_{k+1}) - \gamma \phi^T(i_{k+1})w_n \right)^2
\]

(Least-squares policy evaluation or LSPE)

LSPE converges to PVI

\( \phi w^* = \Pi T(\phi w^*) \)

Figure 12.10 Illustration of the least-squares policy evaluation (LSPE) as a stochastic version of the projected value iteration (PVI).
At iteration $n+1$ of the LSPE($\lambda$) algorithm, the updated weight vector $w$ is computed as the particular value of the weight vector $w$ that minimizes the least-squares difference between the following two quantities:

- the inner product $\Phi^T(i_k)w$ approximating the cost function $J(i_k)$
- its temporal difference counterpart

$$
\Phi^T(i_k)w_n + \sum_{m=k}^{n} (\gamma \lambda)^{m-k} d_n(i_m,i_{m+1})
$$

which is extracted from a single simulated trajectory for $k = 0, 1, ..., n$. 

**LSPE($\lambda$)**

$$
d_n(i_k,i_{k+1}) = g(i_k,i_{k+1}) + \gamma \Phi^T(i_{k+1})w_n - \Phi^T(i_k)w_n
$$

$$
w_{n+1} = \arg\min_w \sum_{k=1}^{n} \left( \phi^T(i_k)w_n - \phi^T(i_k)w_n - \sum_{m=k}^{n} (\gamma \lambda)^{m-k} d_n(i_m,i_{m+1}) \right)^2
$$
1. **Approximate policy evaluation step.** Given the current policy, we compute a cost-to-go function $\tilde{J}^\mu(i, w)$ approximating the actual cost-to-go function $J^\mu$ for all states $i$. The vector $w$ is the weight vector of the neural network used to perform the approximation.

2. **Policy improvement step.** Using the approximate cost-to-go function $\tilde{J}^\mu(i, w)$, we generate an improved policy $\mu$. This new policy is designed to be greedy with respect to $\tilde{J}^\mu(i, w)$.
12.11 Approximate Policy Iteration (2/3)

\[ \varepsilon(w) = \sum_{i \in X} \sum_{m=1}^{M(i)} (k(i,m) - \tilde{J}^\mu(i,w))^2 \]

\[ Q(i,a,w) = \sum_{j \in X} p_{ij}(a)(g(i,a,j) + \gamma \tilde{J}^\mu(j,w)) \]

\[ \mu(i) = \arg\min_{a \in A_i} Q(i,a,w) \]

Figure 12.12 Block diagram of the approximate policy iteration algorithm.
### 12.11 Approximate Policy Iteration (3/3)

**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 $\mu$ 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 $w$ of the neural network employed as the least-squares solver:

   $$w^* = \min_w E(w)$$

   $$= \min_w \sum_{i \in \mathcal{X}} \sum_{m=1}^{M(i)} (k(i, m) - \tilde{J}^\mu(i, w))^2$$

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

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

4. Determine the improved policy

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

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.
Summary and Discussion

- Approximate Dynamic Programming: Direct Methods
  - Temporal difference (TD) learning
  - $Q$-learning

- Approximate Dynamic Programming: Indirect Methods
  - Linear structural approach, which involves
    - Feature extraction of the state $i$
    - Least-squares minimization (e.g., LSPE)
  - Nonlinear structural approach, which relies on universal approximators (e.g., neural networks)

- Partial Observability (POMDP)

- Relationship between Dynamic Programming and Viterbi Algorithm