```
function PASSIVE-RL-AGENT(e) returns an action
  static: U, a table of utility estimates
          N, a table of frequencies for states
          M, a table of transition probabilities from state to state
          percepts, a percept sequence (initially empty)

  add e to percepts
  increment N[STATE[e]]
  U ← UPDATE(U, e, percepts, M, N)
  if TERMINAL?[e] then percepts ← the empty sequence
  return the action Observe
```

Figure 20.2 Skeleton for a passive reinforcement learning agent that just observes the world and tries to learn the utilities, U, of each state. The agent also keeps track of transition frequencies and probabilities. The rest of this section is largely devoted to defining the UPDATE function.

```
function LMS-UPDATE(U, e, percepts, M, N) returns an updated U

  if TERMINAL?[e] then
     reward-to-go ← 0
     for each e_i in percepts (starting at end) do
        reward-to-go ← reward-to-go + REWARD[e_i]
        U[STATE[e_i]] ← RUNNING-AVERAGE(U[STATE[e_i]], reward-to-go, N[STATE[e_i]])
     end
```

Figure 20.3 The update function for least mean square (LMS) updating of utilities.

we will show, it is an easy matter to use more powerful kinds of representations for the utility function, such as neural networks. Learning techniques for those representations can be applied directly to the observed data.

One might think that the LMS approach more or less solves the reinforcement learning problem—or at least, reduces it to one we already know a lot about. In fact, the LMS approach misses a very important aspect of the reinforcement learning problem, namely, the fact that the utilities of states are not independent! The structure of the transitions among the states in fact imposes very strong additional constraints: *The actual utility of a state is constrained to be the probability-weighted average of its successors' utilities, plus its own reward.* By ignoring these constraints, LMS-UPDATE usually ends up converging very slowly on the correct utility values for the problem. Figure 20.4 shows a typical run on the 4×3 environment in Figure 20.1, illustrating both the convergence of the utility estimates and the gradual reduction in the root-mean-square error with respect to the *correct* utility values. It takes the agent well over a thousand training sequences to get close to the correct values.