Game Applications

Reinforcement Learning

Temporal Differences

Reading

- Chapter 11 of Anastasio

Example: Game Playing

- How to learn to play a game, say tic-tac-toe?
- **Supervised** learning approach:

  Listen to a **teacher** indicate whether a proposed move is good, or

  Observe an expert player play games; learn to mimic the good player.

Problems with Supervised Learning for Game Playing

- Need access to **expert**/teacher or many recorded game samples.
- The teacher might not be perfect.
- There is typically **no** play-by-play **target** value; the value is only assigned to a sequence of plays, ending with +1 (win) or -1 (loss).
- The environment is **non-deterministic**, so dealing with a single sequence is not enough.
- We don't necessarily have a **model** for the environment (e.g. the opponent).

Game Applications

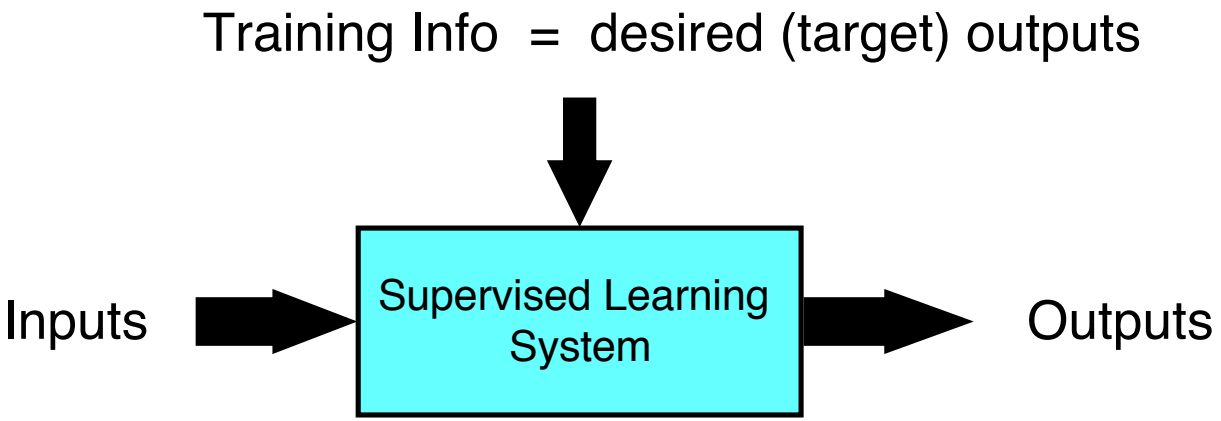
- BPPT (Backpropagation Through Time) seems potentially useable.
- Another approach is to use the "Temporal Difference" method, which is an example of "reinforcement learning".

General Learning Types

- **Supervised learning**: Training with desired answer given for each action
- **Unsupervised learning**: No desired answer; learns *similarities* between training patterns (e.g. clustering)
- **Reinforcement learning**:
  Generalization of Supervised Learning:

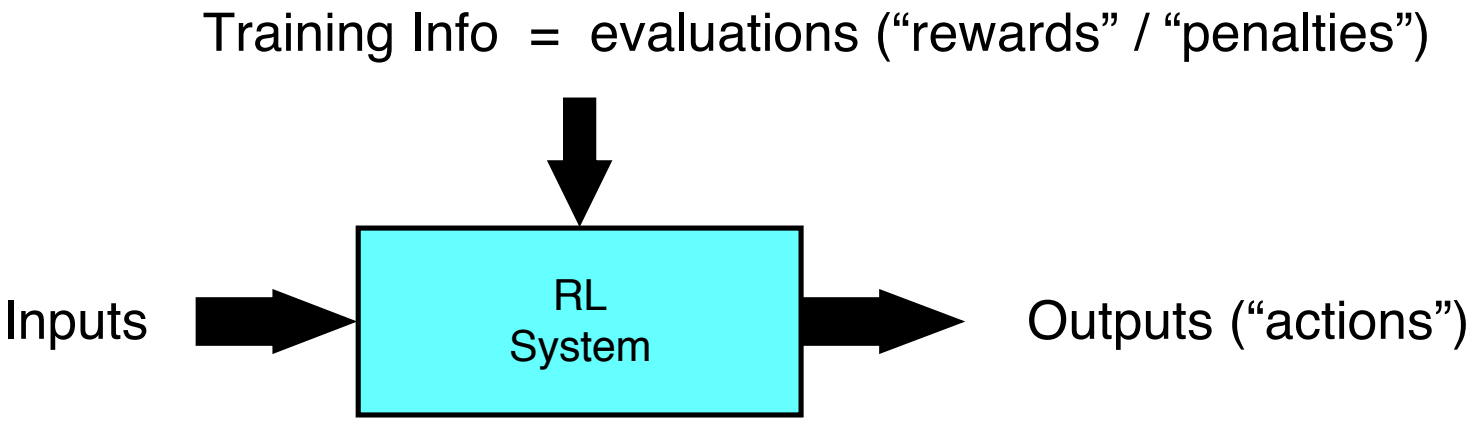
  Reward given later, not necessarily tied to specific action now

Supervised Learning



Error = (target output – actual output)

Reinforcement Learning



Objective: get as much reward as possible

Key Features of RL

(slide from Sutton and Barto)

- Learner is not told *which* actions to take
- Trial-and-Error search
- Possibility of delayed reward

  Sacrifice short-term gains for greater long-term gains
- Need to ***explore*** as well as ***exploit***
- Consider the whole problem of a goal-directed agent interacting with an uncertain environment

A Few Reinforcement Learning Examples

- **TD-Gammon**: Tesauro
  - world's best backgammon program
- **Elevator Control**: Crites & Barto
  - high performance down-peak elevator controller
- **Inventory Management**: Van Roy, Bertsekas, Lee&Tsitsiklis
  - 10–15% improvement over industry standard methods
- **Dynamic Channel Assignment**: Singh & Bertsekas, Nie & Haykin
  - high performance assignment of radio channels to mobile telephone calls

Reinforcement Learning for Games

- Reward often deferred until the end of the game.
- May need to deal with *stochastic* environment (transition probabilities).

Typical AI Model: MDP (Markov Decision Problem)

- Set of states, maybe very large
- Set of actions: Actions map states to sets of possible next states.
- Transition probabilities between states:

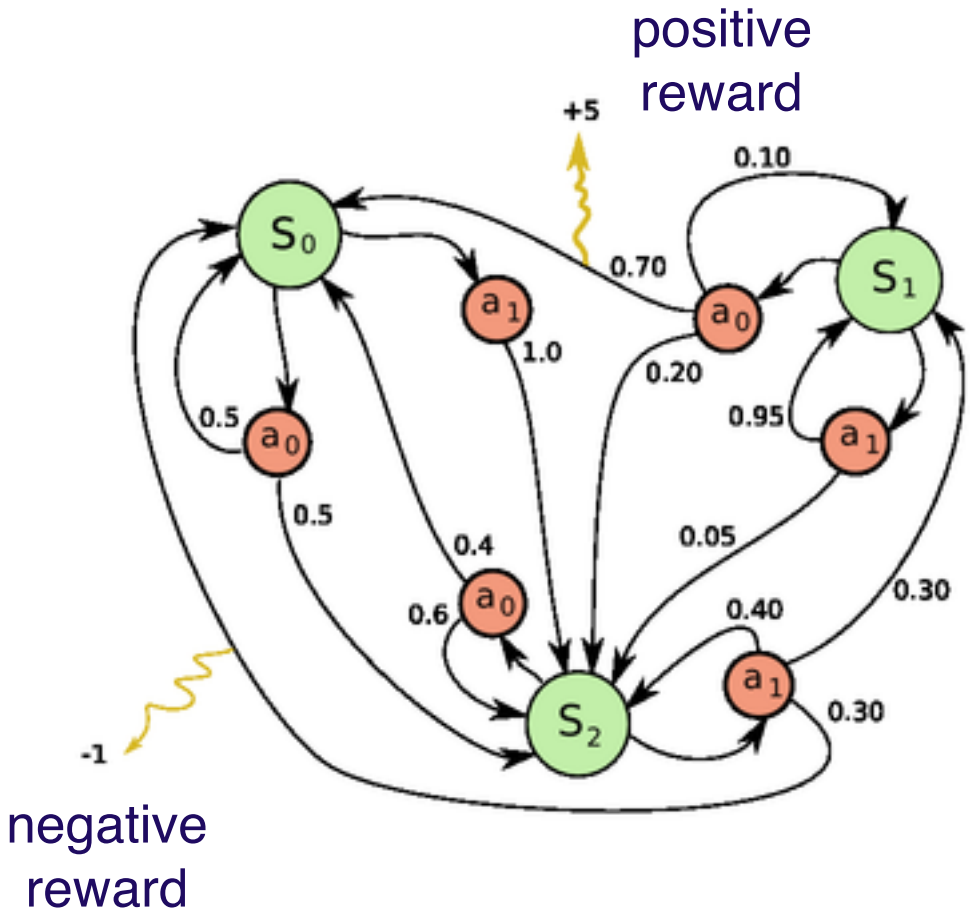
  M^{a}_{ij} = prob[transition from state i to state j,
  given **action** *a* is taken in state i]

- Reward R(i) (positive, negative, or 0) associated with each state i, or alt. $R_a(i, j)$ associated with state i, action a, going to state j.
- The objective is to accrue as much reward as possible.

State, Action, Outcome

- State: Joe is at bat.
- Action: Joe swings at pitch.
- Possible outcomes:
  - Strike: 50%
  - Ball: 30%
  - Hit: 20%

MDP with 3 states, 2 actions



http://en.wikipedia.org/wiki/Markov_decision_process

Problem: Find Best Policy

In any given state, decide which move to make, to get the most reward ultimately.

This is called the **policy** π and may be regarded as a mapping from states to actions:

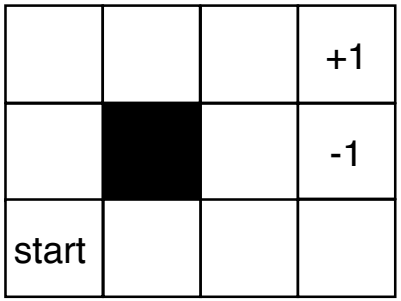
$\pi(s)$ = what action to take when in s.

Grid World Example (Russell & Norvig)

- In a grid world, the states are squares of a grid.
- For each pair of adjacent states i, j there is a probability of going from i to j by the (one and only) action: move.
- Markov property: The probability of being in a given state at step in the overall process is dependent only on the previous state and the transition probabilities (rather than the entire history).

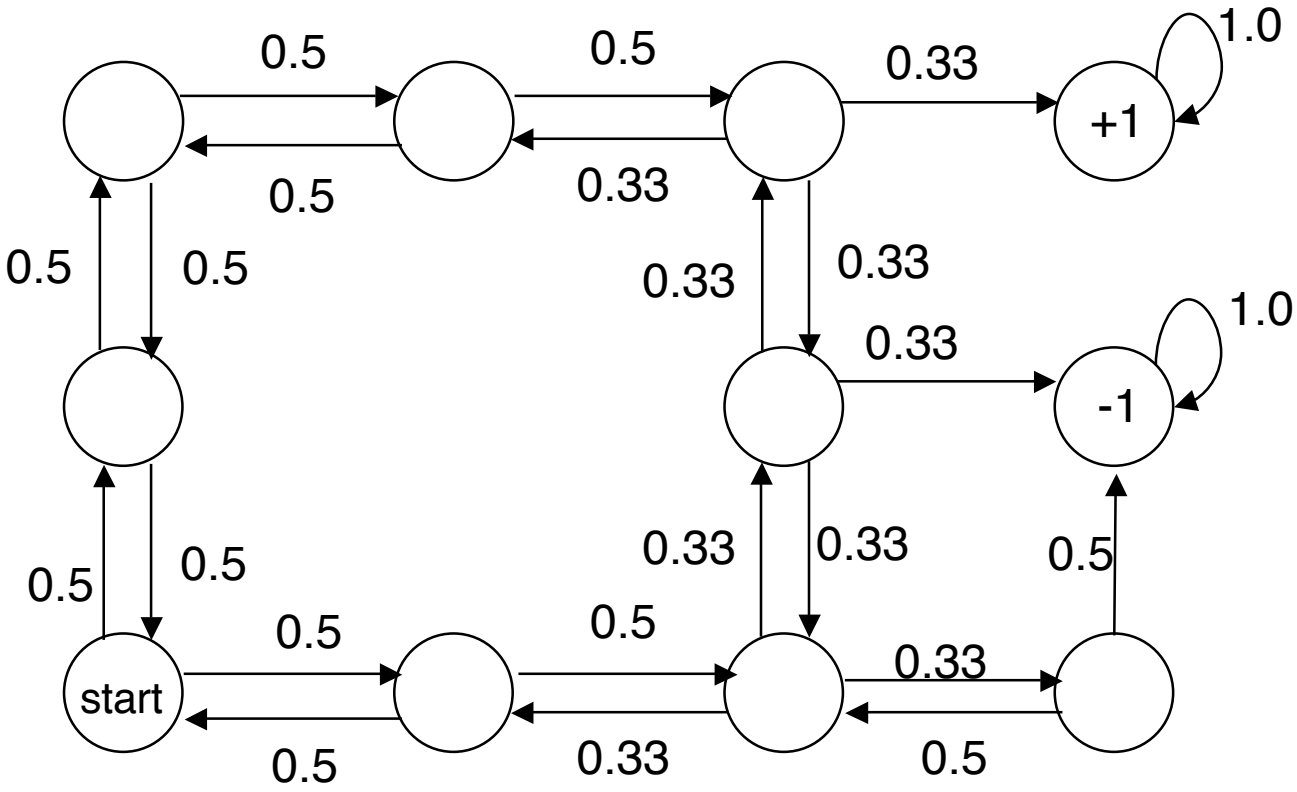
3x4 “Grid World” (Explicit States) Example (Norvig & Russell)

- Consider the following maze, with reward function 0 except as shown in the two boxes (with +1, -1).
- Assume a single action “move” with **equal probabilities** of moving any direction, except stay in +1 or -1 if reached.



Markov State-Transition Probs

based on equal probs. of any action in a state



Decision-Theoretic Criterion

Move to the state where the **expected value** of ultimate reward is the highest.

Utility Function =

Estimated Expected Value of Ultimate Overall Reward in a State

- Desirability of moving to a given state s is expressed by V(s).
- The **utility is not the reward**, but rather an ***estimate*** of the award that can be accrued from that state.
- The utility is generally not given explicitly; it must be computed or ***learned***.
- A general strategy: Given a choice of moves to several states, choose the state with ***highest utility***.

Best Policy using Utility Function

The goal is to choose a policy π that will maximize some cumulative function of the random rewards, typically the expected discounted sum over a potentially infinite horizon:

$$\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1}) \quad \text{(where we choose } a_t = \pi(s_t)\text{)}$$

where γ is the discount factor and satisfies $0 \leq \gamma < 1$. (For example, $\gamma = 1/(1+r)$ when the discount rate is r.) γ is typically close to 1.

Because of the Markov property, the optimal policy for this particular problem can indeed be written as a function of s only, as assumed above.

http://en.wikipedia.org/wiki/Markov_decision_process

Additive Utility Function

- Assume that the sought utility function V is *additive*, in that it obeys the relationship

Utility of state i

$$V(i) = R(i) + \max_{a}\left(\sum_{j} P^{a}_{ij} \cdot V(j)\right)$$

Expected utility of next state given a.

where

- R(i) is the reward of state i,
- a is an action, and
- P^{a}_{ij} is the probability of going from state i to state j with action a.

- This is the ***dynamic programming*** equation (Richard Bellman).

Possible Direct Computation of Utility Function

If for each state i the reward R(i) is known, and the transition probabilities are known, and there is just one action (e.g. move) then the expectations V(i) of ultimate reward satisfy

$$\forall i \quad V(i) = R(i) + \sum_j P_{ij} V(j)$$

In matrix form:

$$V = R + P^T V$$

Solving for Expected Values

In matrix form:

$$V = R + P^T V$$

$$(I - P^T)V = R$$

So

$$V = (I-P^T)^{-1}R$$

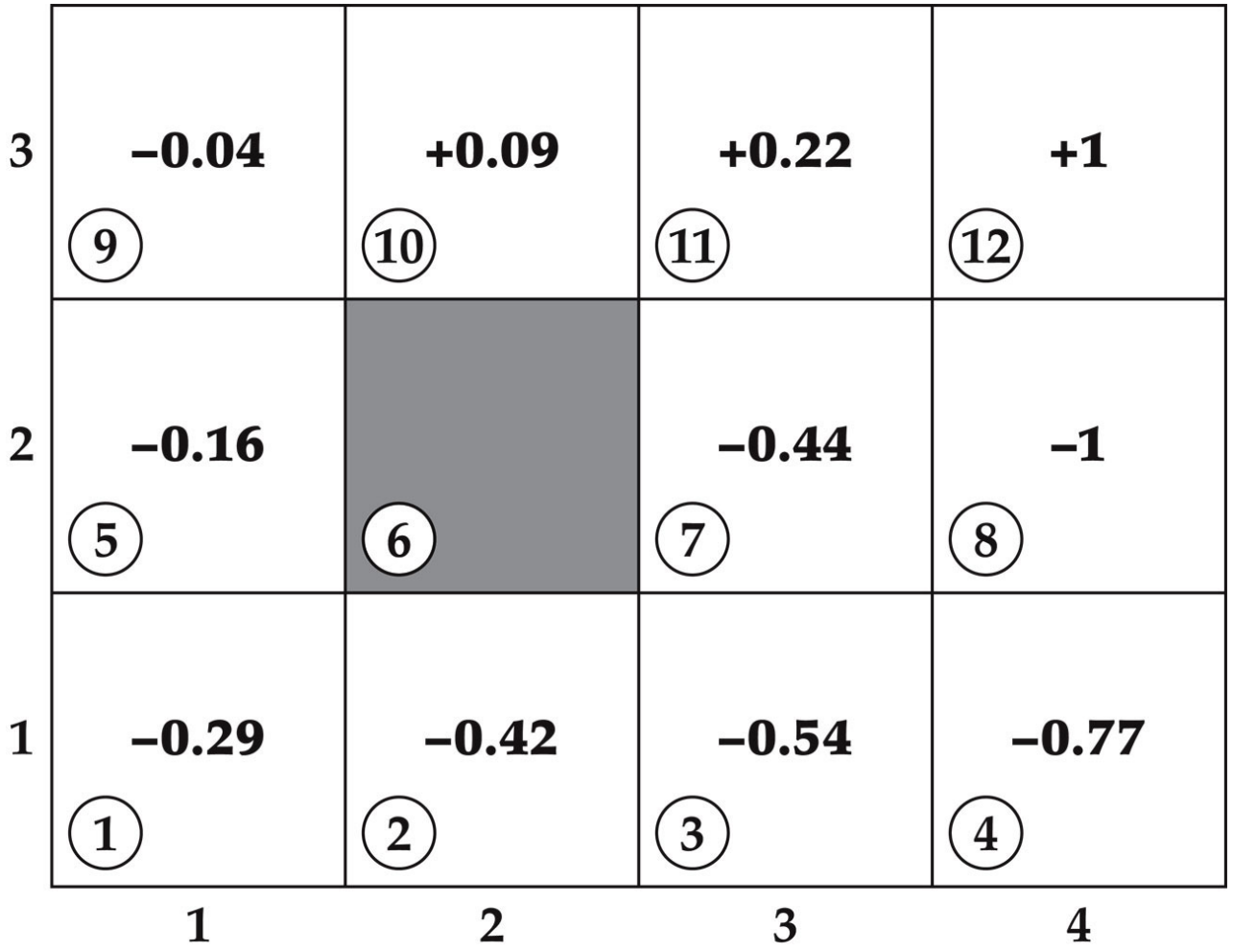
Computational issues:

Too many states in general case

P might not be known

```
% gridWorldSetUp.m establishes grid world and solves for exact values of utility

nStates=12;                    % set number of states
stateVec=(1:12)';              % set a state number vector
r=zeros(nStates,1);            % set a reinforcement vector
ProbMat=zeros(nStates);        % define a probability matrix
tsr=12;              % designate the terminal state of reward
tsp=8;               % designate the terminal state of punishment
intReSt=7;           % designate an intermediate state for reinforcement
r(tsr)=+1;           % set reinforcement of reward terminal state
r(tsp)=-1;           % set reinforcement of punishment terminal state
r(intReSt)=0;        % set intermediate reinforcement if desired

TM = ... % enter the transition matrix
[0  1  0  0  1  0  0  0  0  0  0  0
 1  0  1  0  0  0  0  0  0  0  0  0
 0  1  0  1  0  0  1  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  0  0  0
 1  0  0  0  0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  0  1  0
 0  0  0  1  0  0  1  0  0  0  0  0
 0  0  0  0  1  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  0  1  0  1  0
 0  0  0  0  0  0  1  0  0  1  0  0
 0  0  0  0  0  0  0  0  0  0  1  0];

% use the transition matrix to find the probability matrix
for j=1:nStates,                          % for each state
    indx=find(TM(:,j)~=0);                % find indices of allowed next states
    if isempty(indx), prob=0;             % if no next state assign zero prob
    else prob=1/sum(TM(:,j))  ; end       % else compute probability
    ProbMat(indx,j)=prob;                 % enter probability into prob matrix
end

% solve for the exact state values
exVals=inv(ProbMat'-eye(nStates))*(-r);
```

Exact Expected Values for the Grid World



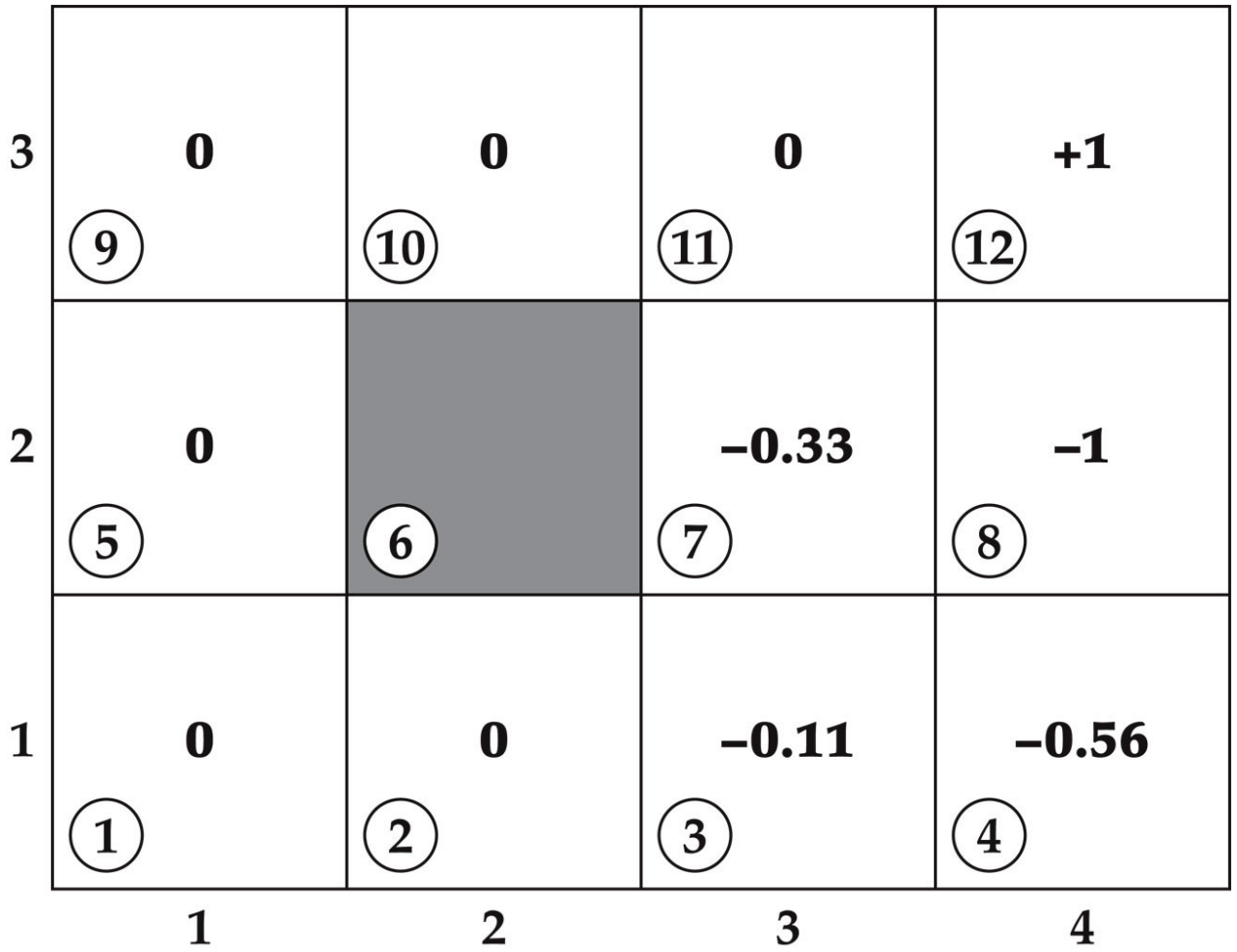
TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.2

A way to compute V: Iterative Dynamic Programming

- For each state i, initialize V(i) to R(i), the reward function.
- While(not converged)

```
{
in parallel for each state i,
   {
   Update V(i) ← R(i) + max( Σ_j P^a_ij • V( j ))
                          a
   }
}
```

After One Iteration of DP Update



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.3

```
% iterativeDynamicProg.m
% this script updates state values using the IDP algorithm
% note: script gridWorldSetUp must be run first

nEpo=100;                                % set number of epochs

v=zeros(nStates,1);                      % define estimated state values vector
v(tsr)=r(tsr); v(tsp)=r(tsp);            % set terminal state values
vEst=zeros(ceil(nEpo/10+1),nStates);     % define value hold array
rms=zeros(ceil(nEpo/10+1),1);            % define RMS error hold array
vEst(1,:)=v';                            % save initial state values
rms(1)=sqrt(mean((exVals-v).^2));        % save initial RMS error

for epo=1:nEpo,                          % do for all epochs
   tsf=0;                                % zero terminal state flag
   st=1;                                 % start epoch at state one
   while tsf==0,                         % while terminal state flag equals zero
       indx=find(TM(:,st)~=0);           % find indices of allowed next states
       prob=1/sum(TM(:,st));             % prob of transitions to each next state
       v(st)=r(st)+sum(prob*v(indx));    % update state value
       nIndx=length(indx);               % find the number of allowed next states
       choose=ceil(rand*nIndx);          % choose an index at random
       nextSt=indx(choose);              % choose the next state at random
       if nextSt==tsr | nextSt==tsp, tsf=1; end % check if terminal
       st=nextSt;                        % set current state to next state
   end                                   % end of while loop, end of one epoch

   if rem(epo,10)==0,                    % every ten epochs
      vEst(epo/10+1,:)=v';               % save value estimates
      rms(epo/10+1)=sqrt(mean((exVals-v).^2)); % save rms error
   end
end                                      % end of nEpo epochs
```

Comparison: Exact vs. Iterative

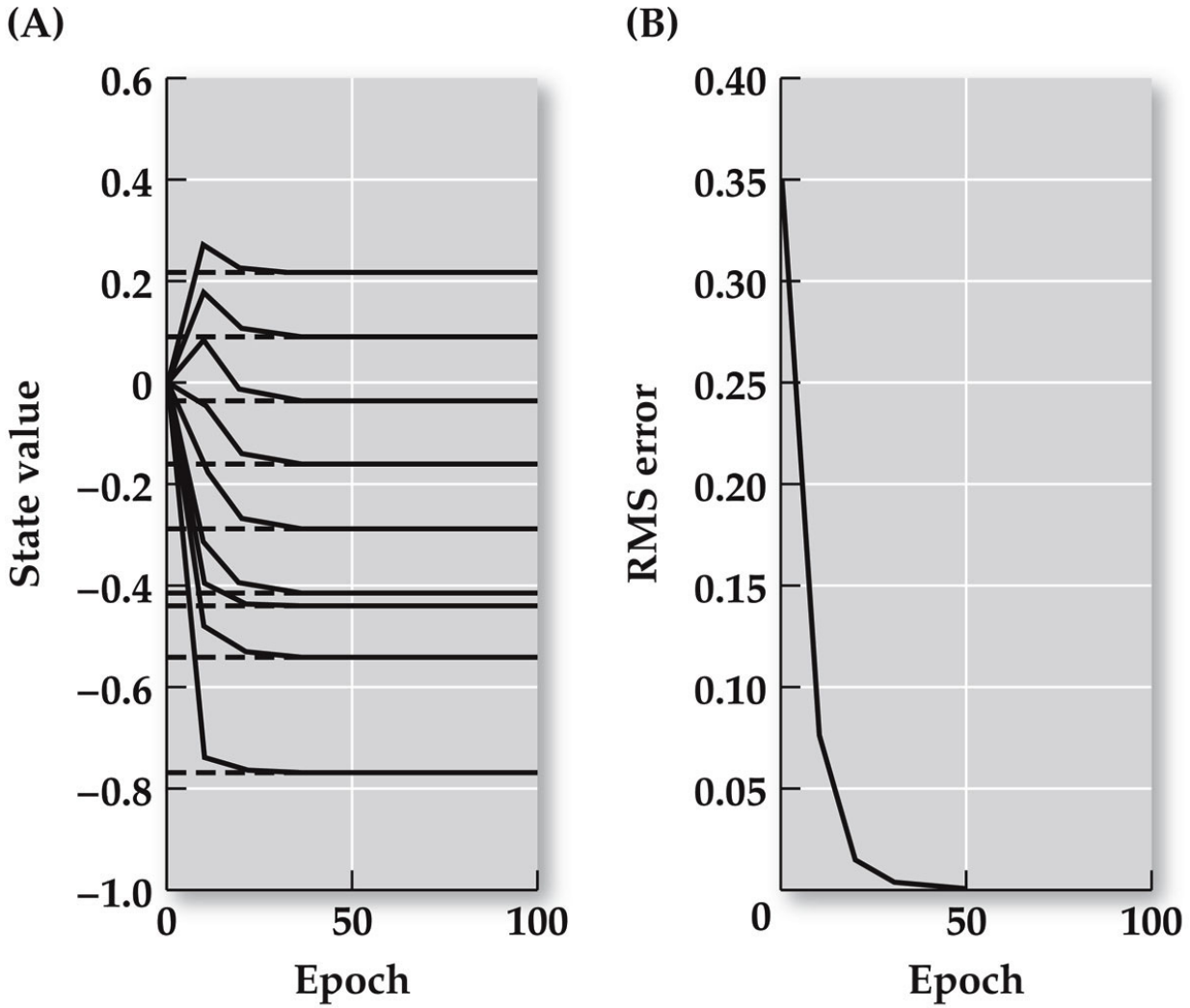
TABLE 11.1 **Exact gridworld state values compared with values estimated following 100 epochs of iterative dynamic programming**

State number	Exact value	Estimated value
1	−0.29	−0.29
2	−0.42	−0.42
3	−0.54	−0.54
4	−0.77	−0.77
5	−0.16	−0.16
6	0.00	0.00
7	−0.44	−0.44
8	−1.00	−1.00
9	−0.04	−0.04
10	+0.09	+0.09
11	+0.22	+0.22
12	+1.00	+1.00

State 6 is a disallowed state, and states 8 and 12 are terminal states of punishment and reward, respectively. Estimated state values match the exact values.

TUTORIAL ON NEURAL SYSTEMS MODELING, Table 11.1

Convergence of DP over 100 Iterations



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.4

Problem with DP Method for Games

- Probabilities aren't generally known.
- Thus ways to estimate utilities are sought that don't require knowing probabilities.

LMS (Least-Mean-Square) Estimation (section 11.2)

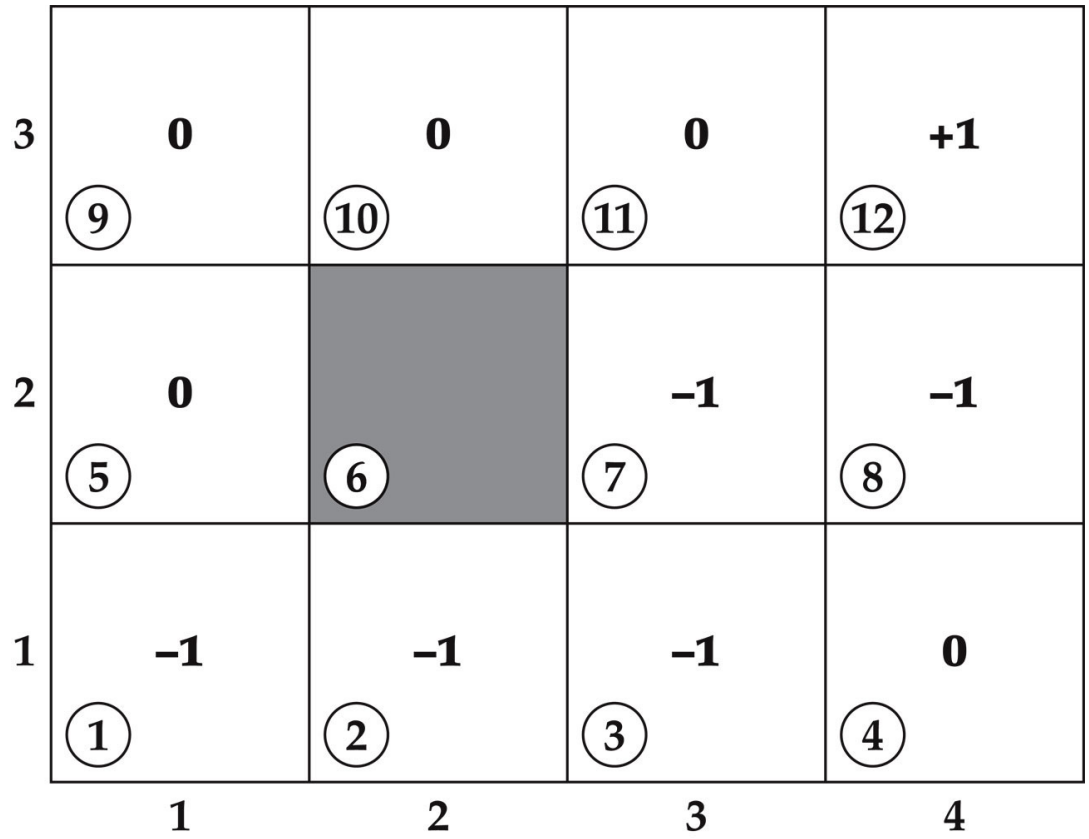
- Over a large number of epochs:
  - Generate a trajectory: a random sequence of connected states, resulting in some accumulated reward for the sequence.
  - Average the reward for the trajectory into the **accumulated** expected reward for each state.
  - (This averaging is done by maintaining a **counter** of the number of times a state is updated, in addition to the accumulated expected reward.)

LMS Trajectory

- The **sequence** of states visited has to be remembered.
- The accrued reinforcement is then distributed to the states in the trajectory.

```
% leastMeanSquares.m
% this script updates state values using the LMS algorithm
% note: script gridWorldSetUp must be run first

nEpo=1000;                                          % set number of epochs
v=zeros(nStates,1);                                 % define estimated state values vector
v(tsr)=r(tsr); v(tsp)=r(tsp);                       % set terminal state values
count=zeros(nStates,1);                             % define update count holding vector
vEst=zeros(ceil(nEpo/10+1),nStates);                % define value hold array
rms=zeros(ceil(nEpo/10+1),1);                       % define RMS error hold array
vEst(1,:)=v';                                       % save initial state values
rms(1)=sqrt(mean((exVals-v).^2));                   % save initial RMS error

for epo=1:nEpo,                                     % do for all epochs
   tsf=0;                                           % zero terminal state flag
   st=1;                                            % start epoch at state one
   trj=st;                                          % set the first state of the trajectory
   while tsf==0,                                    % while terminal state flag equals zero
       indx=find(TM(:,st)~=0);                      % find indices of allowed next states
       nIndx=length(indx);                          % find the number of allowed next states
       choose=ceil(rand*nIndx);                     % choose an index at random
       nextSt=indx(choose);                         % choose the next state at random
       if nextSt==tsr | nextSt==tsp, tsf=1; end     % check if terminal
       st=nextSt;                                   % set current state to next state
       trj=[trj st];                                % add new state to the state trajectory
   end                                              % end of trajectory
   lTrj=length(trj);                                % find the length (in states) of trajectory
   rTrj=zeros(lTrj,1);                              % set up the reinforcement trajectory
   trj=fliplr(trj);                                 % reverse the order of the state trajectory
   if trj(1)==tsr, rTrj(1)=r(tsr);                  % set end-state reward ...
   elseif trj(1)==tsp, rTrj(1)=r(tsp); end          % or punishment
   rTrj(find(trj==intReSt))=r(intReSt);             % intermediate reinforcement
   rtg=0;                                           % zero the reward-to-go
   for tr=1:lTrj,                                   % for each transition on (reversed) trajectory
       rtg=rtg+rTrj(tr);                            % increment the reward to go
       v(trj(tr))=v(trj(tr))+...                    % update the state values ...
           (rtg-v(trj(tr)))/(count(trj(tr))+1);     % via LMS
       count(trj(tr))=count(trj(tr))+1;             % increment the counter
   end                                              % end trajectory update loop
   if rem(epo,10)==0,                               % every ten epochs
       vEst(epo/10+1,:)=v';                         % save value estimates
       rms(epo/10+1)=sqrt(mean((exVals-v).^2));     % save rms error
   end
end                                                 % end of nEpo epochs
```

Grid World After One Step of LMS



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.5

Several Epochs of LMS Update

```
epoch = 1, trajectory = [1 5 9 10 11 12]
Vs =
     1     1     1     1
     1     0     0    -1
     1     0     0     0
counts =
     1     1     1     1
     1     0     0     0
     1     0     0     0

epoch = 2, trajectory = [1 5 9 10 11 7 8]
Vs =
     0     0     0     1
     0     0    -1    -1
     0     0     0     0
counts =
     2     2     2     1
     2     0     1     1
     2     0     0     0

epoch = 3, trajectory = [1 5 1 2 3 7 11 7 11 10 9 5 1 5 1 2 3 4 3 2 1 2 3 4 3 2 3 2 1 5 1 2 3 7 11 12]
Vs =
    0.3333    0.3333    0.6000    1.0000
    0.6667         0    0.5000   -1.0000
    0.7778    1.0000    1.0000    1.0000
counts =
     3     3     5     2
     6     0     4     1
     9     7     7     2

epoch = 4, trajectory = [1 5 1 2 1 5 9 10 11 10 9 5 1 5 9 5 9 5 1 2 3 2 1 2 3 4 3 7 3 2 1 2 3 7 8]
Vs =

   -0.4286   -0.2000    0.3333    1.0000
   -0.1667         0    0.0000   -1.0000
         0    0.0769    0.1667    0.3333
counts =

     7     5     6     2
    12     0     6     2
    16    13    12     3
```

Comparison Exact to LMS after 1000 Epochs

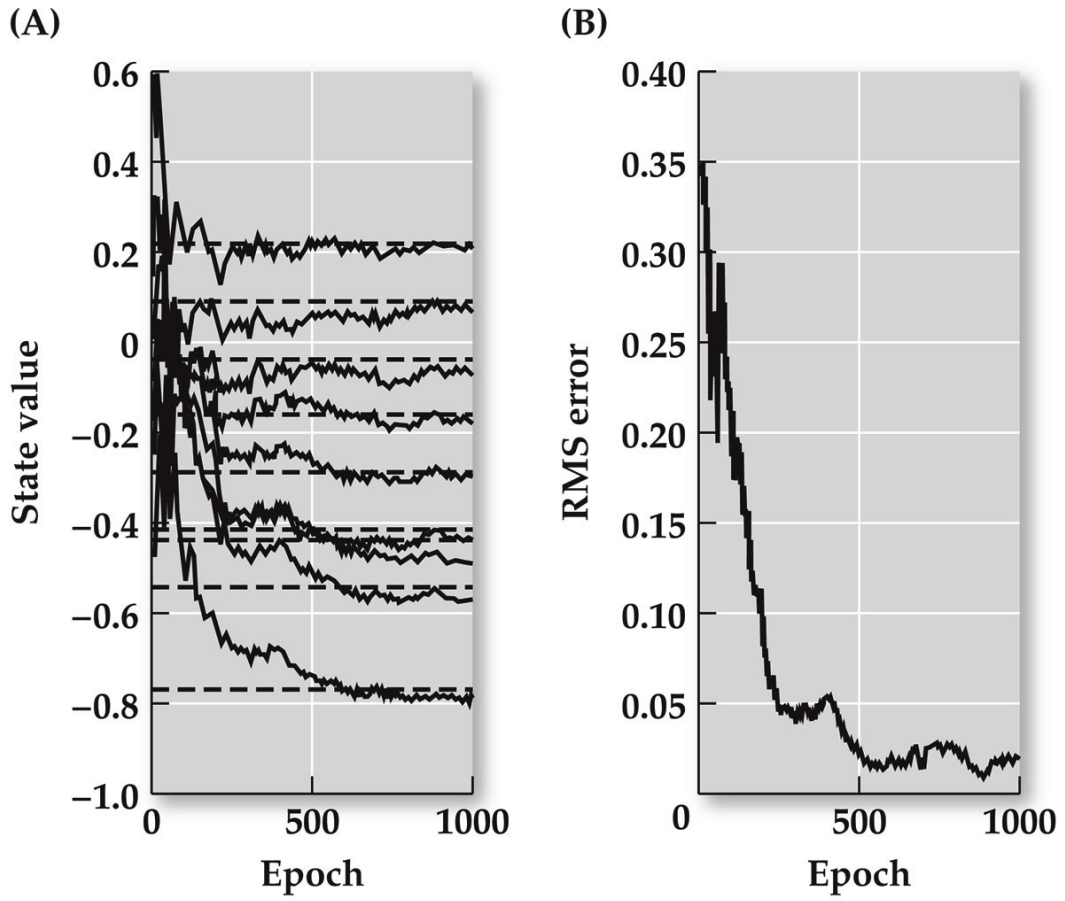
TABLE 11.3 **Exact gridworld state values compared with values estimated following 1000 epochs of temporal-difference learning**

State number	Exact value	Estimated value
1	−0.29	−0.31
2	−0.42	−0.45
3	−0.54	−0.62
4	−0.77	−0.86
5	−0.16	−0.18
6	0.00	0.00
7	−0.44	−0.47
8	−1.00	−1.00
9	−0.04	−0.06
10	+0.09	+0.05
11	+0.22	+0.25
12	+1.00	+1.00

State 6 is a disallowed state, and states 8 and 12 are terminal states of punishment and reward, respectively. Estimated state values are in good agreement with exact values.

TUTORIAL ON NEURAL SYSTEMS MODELING, Table 11.3

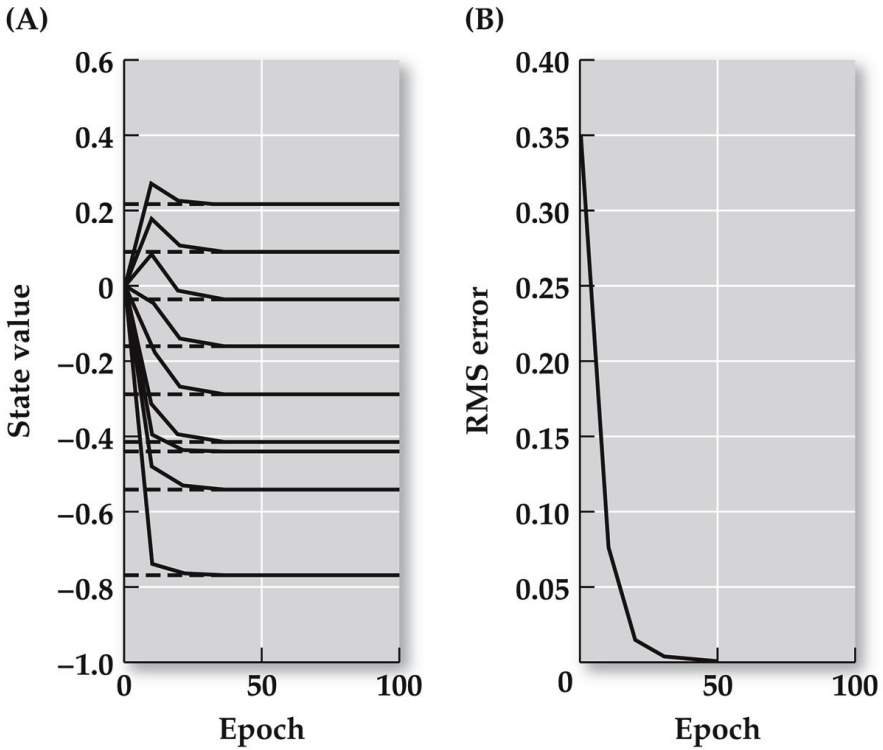
Convergence of LMS



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.6

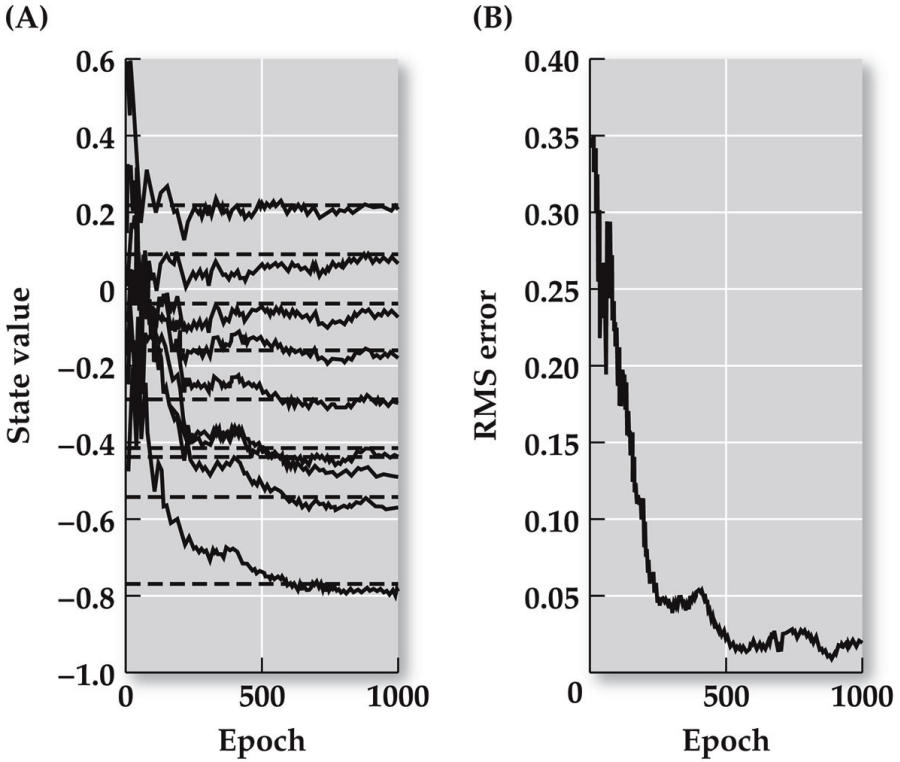
Convergence of DP vs. LMS

DP, 100 Epochs



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.4

LMS, 1000 Epochs



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.6

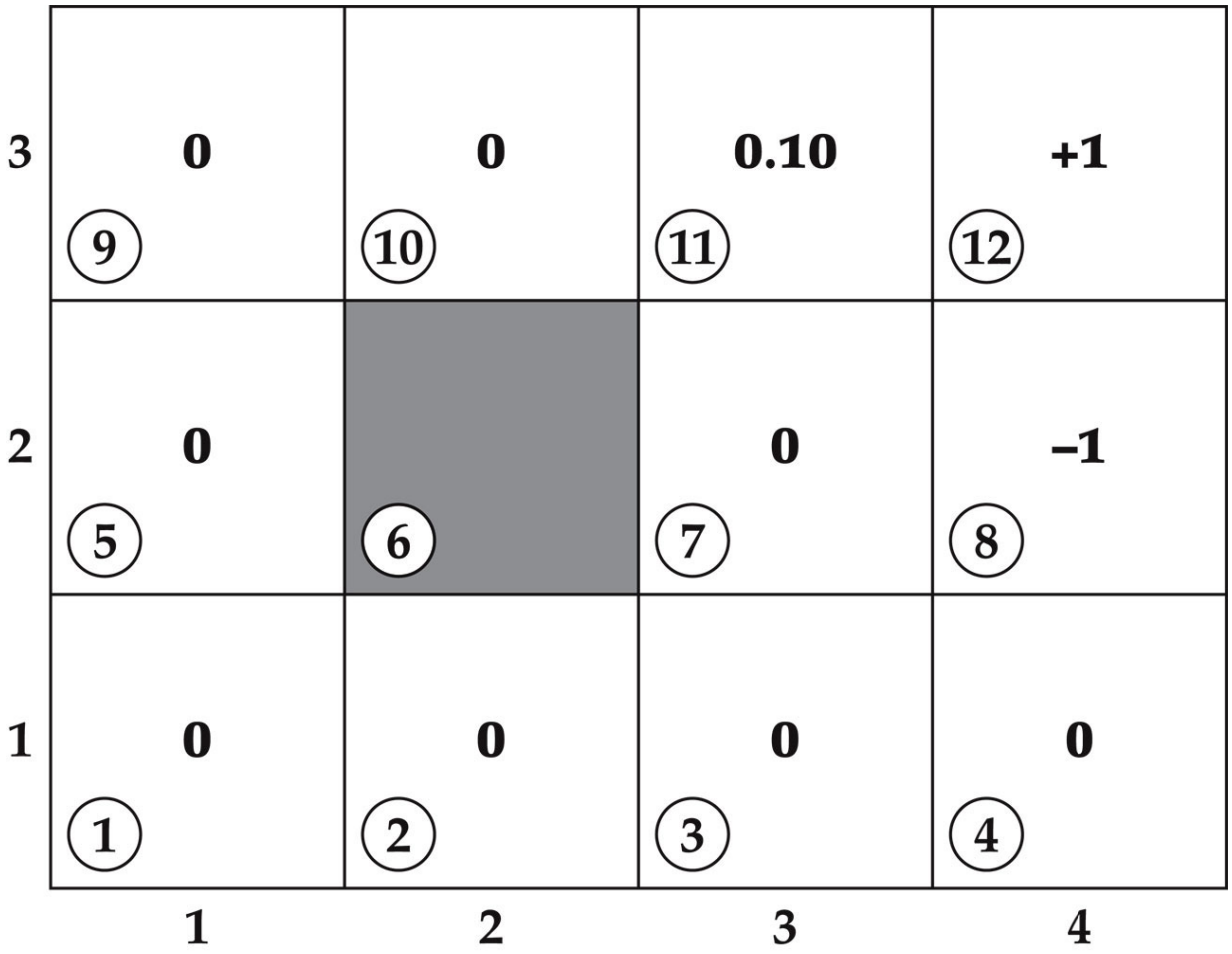
Temporal Difference (TD) Learning

- As with LMS, the probabilities are not assumed to be known.
- **Through many trial runs**, adjust the observed values of V so that they more closely agree with the dynamic programming equation.
- If we encounter a **transition from i to j**, adjust V(i) so that it **better agrees** with V(j), using a **discount rate** $\gamma \leq 1$ which tempers the current utility of future reward.
- Temporal Difference updating rule (with η the **learning rate**):

$$\Delta V(i) = \eta \bullet (R(i) + \gamma V(j) - V(i))$$

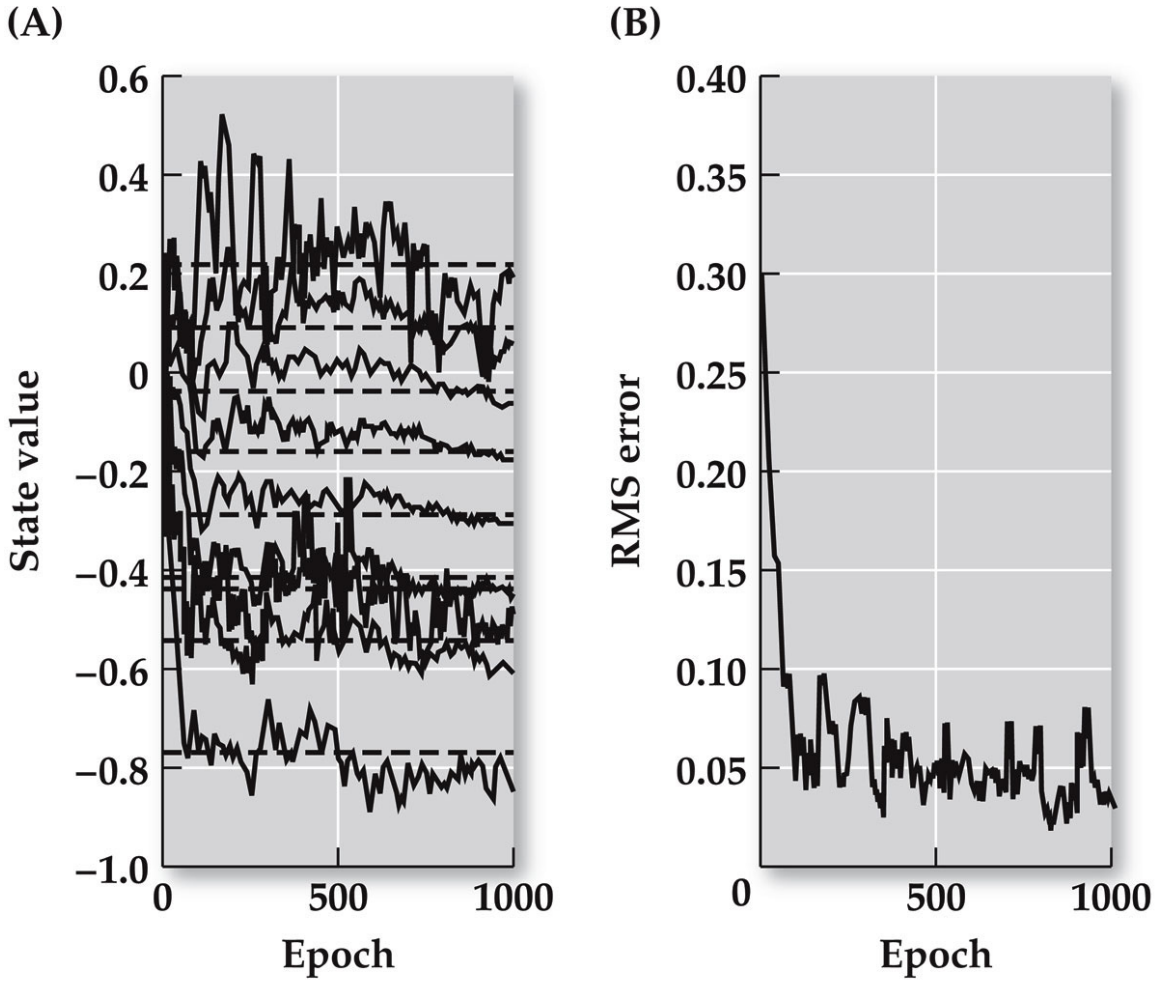
```
% temporalDifference.m
% this script updates state values using the TD algorithm
% note: script gridWorldSetUp must be run first

nEpo=1000;                                  % set number of epochs
a=0.1;                                      % set learning rate
dec=0.999;                                  % set learning rate decrement

v=zeros(nStates,1);                         % define estimated state values vector
v(tsr)=r(tsr); v(tsp)=r(tsp);               % set terminal state values
count=zeros(nStates,1);                     % define update count holding vector
vEst=zeros(ceil(nEpo/10+1),nStates);        % define value hold array
rms=zeros(ceil(nEpo/10+1),1);               % define RMS error hold array
vEst(1,:)=v';                               % save initial state values
rms(1)=sqrt(mean((exVals-v).^2));           % save initial RMS error

for epo=1:nEpo,                             % do for all epochs
   tsf=0;                                   % zero terminal state flag
   st=1;                                    % start epoch at state one
   while tsf==0,                            % while terminal state flag equals zero
       count(st)=count(st)+1;               % increment counter
       indx=find(TM(:,st)~=0);              % find indices of allowed next states
       nIndx=length(indx);                  % find the number of allowed next states
       choose=ceil(rand*nIndx);             % choose an index at random
       nextSt=indx(choose);                 % choose the next state at random
       v(st)=v(st)+(a*dec^count(st))*... % update value of ...
           (r(st)+v(nextSt)-v(st));      % current state using TD
       if nextSt==tsr | nextSt==tsp, tsf=1; end % check if terminal
       st=nextSt;                           % set current state to next state
   end                                      % end of while loop, end of one epoch

   if rem(epo,10)==0,                       % every ten epochs
      vEst(epo/10+1,:)=v';                  % save value estimates
      rms(epo/10+1)=sqrt(mean((exVals-v).^2)); % save rms error
   end                                      % end conditional
end                                         % end of nEpo epochs
```

Figure 11.7 Estimated state values for the gridworld after one epoch of temporal-difference learning



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.7

Figure 11.8 Estimated state values and root-mean-square error over 1000 epochs of temporal-difference learning in the gridworld



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.8

Utilities Computed by TD vs. by Dynamic Programming

TABLE 11.3 **Exact gridworld state values compared with values estimated following 1000 epochs of temporal-difference learning**

State number	Exact value	Estimated value
1	−0.29	−0.31
2	−0.42	−0.45
3	−0.54	−0.62
4	−0.77	−0.86
5	−0.16	−0.18
6	0.00	0.00
7	−0.44	−0.47
8	−1.00	−1.00
9	−0.04	−0.06
10	+0.09	+0.05
11	+0.22	+0.25
12	+1.00	+1.00

State 6 is a disallowed state, and states 8 and 12 are terminal states of punishment and reward, respectively. Estimated state values are in good agreement with exact values.

TUTORIAL ON NEURAL SYSTEMS MODELING, Table 11.3

Comparison Exact, IDP, LMS, TD for specified # epochs

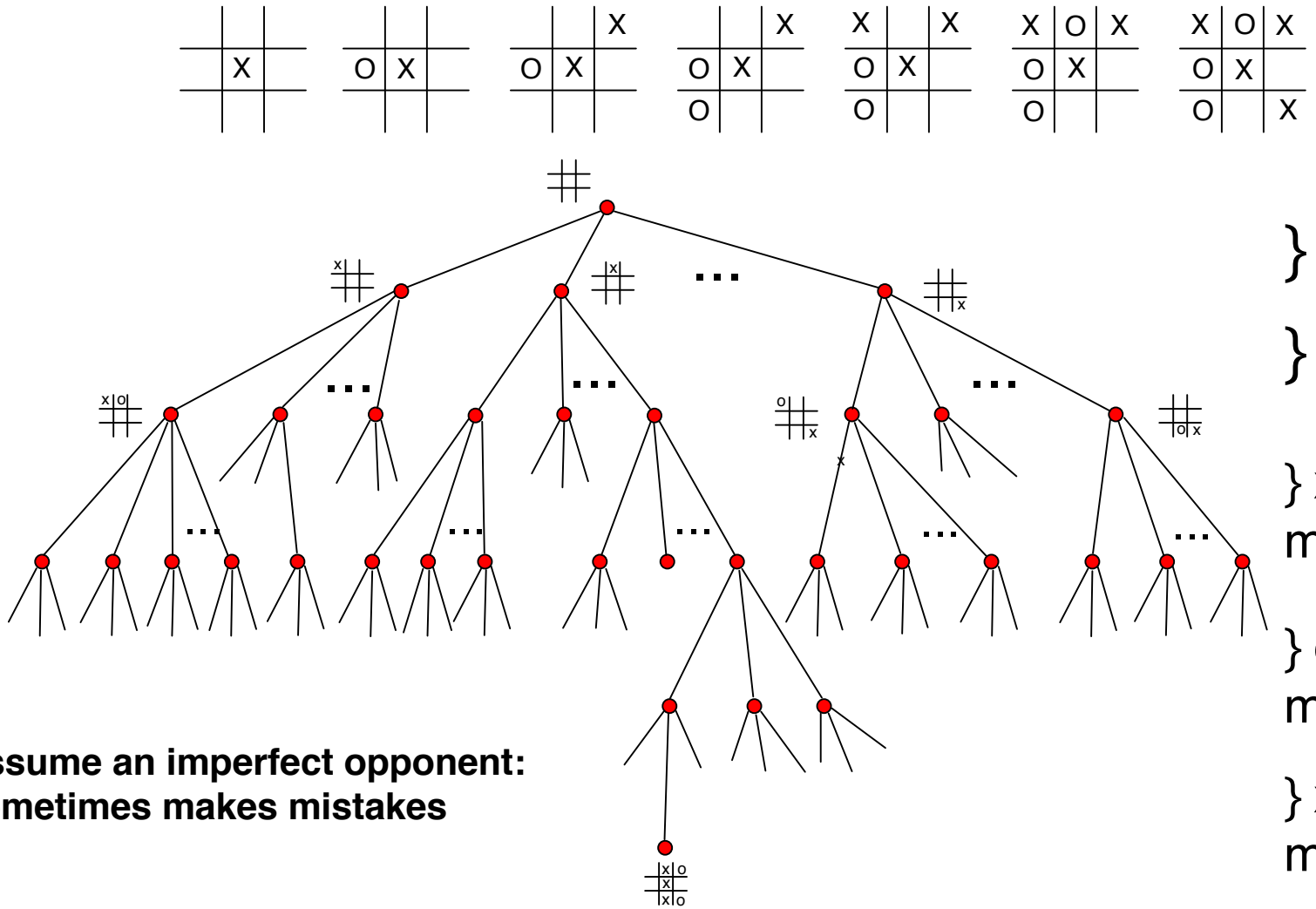
Exact value	IDP 100 Estimated value	LMS 1000 Estimated value	TD 1000 Estimated value
−0.29	−0.29	−0.31	−0.31
−0.42	−0.42	−0.45	−0.45
−0.54	−0.54	−0.62	−0.62
−0.77	−0.77	−0.86	−0.86
−0.16	−0.16	−0.18	−0.18
0.00	0.00	0.00	0.00
−0.44	−0.44	−0.47	−0.47
−1.00	−1.00	−1.00	−1.00
−0.04	−0.04	−0.06	−0.06
+0.09	+0.09	+0.05	+0.05
+0.22	+0.22	+0.25	+0.25
+1.00	+1.00	+1.00	+1.00

Game Playing

- In game playing it is generally infeasible to enumerate all states.
- However, the TD updating rule can be applied to states encountered **in the course of play**.
- We can also make “exploratory moves” along the way to make the utility estimates more robust.

Example: Tic-Tac-Toe

(slide from Sutton and Barto)



} x's move

} o's move

} x's move

} o's move

} x's move

Assume an imperfect opponent: sometimes makes mistakes

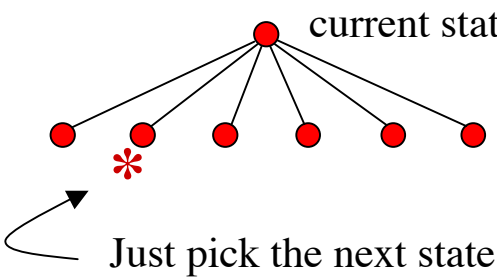
An RL Approach to Tic-Tac-Toe

(slide from Sutton and Barto)

1. Make a table with one entry per state:

State	$V(s)$ – estimated probability of winning	
	.5	?
	.5	?
⋮	⋮	
	1	win
⋮	⋮	
	0	loss
⋮	⋮	
	0	draw

2. Now play lots of games.
To pick our moves,
look ahead one step:

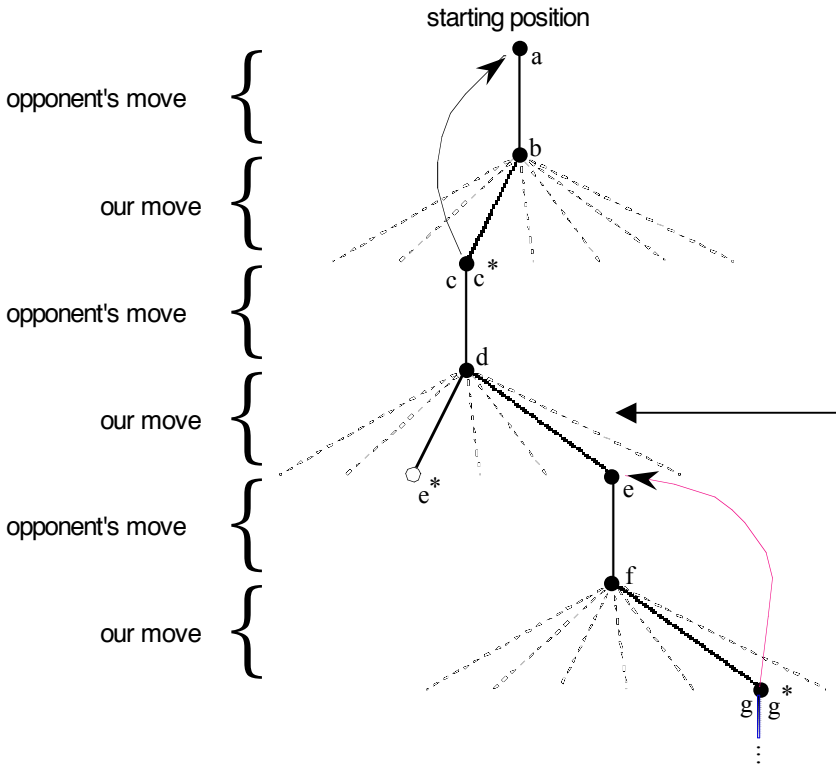


Just pick the next state with the highest estimated prob. of winning — the largest $V(s)$; a ***greedy*** move.

But 10% of the time pick a move at random; an ***exploratory move***.

RL Learning Rule for Tic-Tac-Toe

(slide from Sutton and Barto)



"Exploratory" move

s – the state before our greedy move

s' – the state after our greedy move

We increment each $V(s)$ toward $V(s')$ – a ***backup***:

$$V(s) \leftarrow V(s) + \alpha\left[V(s') - V(s)\right]$$

a small positive fraction, e.g., $\alpha = .1$

the ***step-size parameter***

More on TD method

- TD tries to train a utility function to **predict** the outcome from a state.
- The earliest known use (not by the name TD) was in Samuel's checker-playing program (1959).
- Richard Sutton expressed the general framework (1988).

Sutton's Derivation of TD
with Application to Neural Networks

- Observation-outcome sequence:

  $x_1, x_2, x_3, \ldots, x_m, z$

- x_t is the observation (input) at step t
- z is the net **outcome** of the sequence
- All values are real numbers

- Say that the learner's **predictions** that estimate the final outcome z are:

  $P_1, P_2, P_3, \ldots, P_m$

Sutton's Derivation (2)

- In general, a **prediction** P_t can be a function of all preceding observations, but for simplicity it can be assumed to just depend on the current observation x_t.
- (We could always include all previous observations as "part of" the current observation.)
- P_t can be regarded as the **utility** value in the earlier slides.

Sutton's Derivation (3)

- If computed by a neural net, P_t will also depend on some **weights** w, and could be written explicitly as

$$P_t = P(x_t, w)$$

- The learning rule will indicate how to update w.

- Let Δw_t be the weight change as a result of prediction P_t .

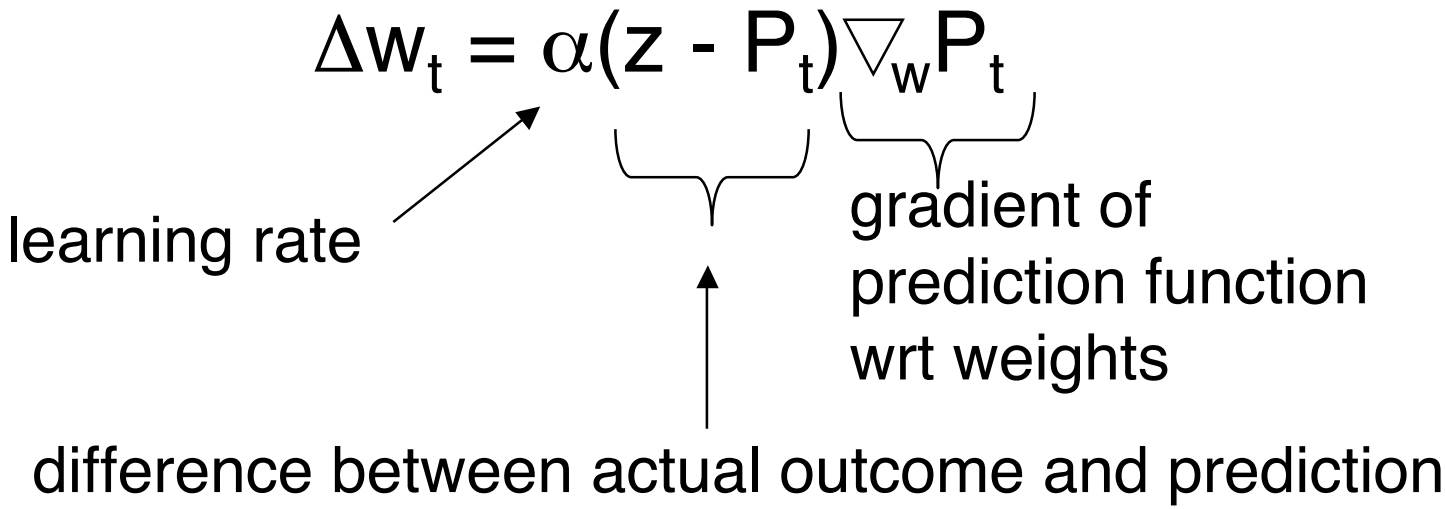
Sutton's Derivation (4)

- The **net change** in w over the entire observation sequence x_1, x_2, x_3, …., x_m, is thus:

$$\Sigma_t \Delta w_t$$

Sutton's Derivation (5)

- *Supervised* learning would pair each observation with the *expected final* outcome and train thus:



Sutton's Derivation (6)

- Example: If the prediction function were linear: $P_t(w, x_t) = \sum w(i) \cdot x_t(i)$ then

  $\nabla_w P_t = x_t$

  and we have the Widrow-Hoff rule:

$$\Delta w_t = \alpha(z - w^T x_t)\, x_t$$

$(z - w^T x_t)$: difference between outcome and prediction

x_t: gradient of prediction function

Sutton's Derivation (7)

- For an MLP network, rather than linear, the same update form can be used as with backpropagation. The gradient is just more complicated, as we know.
- The problem with supervised technique is that it **assumes knowledge of the final outcome**.
- Temporal differences **remove this assumption**, as shown next.

Sutton's Derivation (8)

- Represent the error in a prediction $z - P_t$ as a sum of **changes** in predictions, using "telescoping"

$$z - P_t = \sum_{k=t}^{m} \left(P_{k+1} - P_k\right) \text{ where } P_{m+1} =_{def} z.$$

(z is the final outcome)

Sutton's Derivation (9)

- Now re-express the **net weight-change** for supervised learning:

$$\sum_{t=1}^{m} \Delta w_t = \sum_{t=1}^{m} \alpha \left(z - P_t\right) \nabla_w P_t$$

Incremental change, slide 5

$$\Delta w_t = \alpha (z - P_t) \nabla_w P_t$$

$$= \sum_{t=1}^{m} \alpha \sum_{k=t}^{m} \left(P_{k+1} - P_k\right) \nabla_w P_t$$

from telescoping (8)

$$= \sum_{k=1}^{m} \alpha \sum_{t=1}^{k} \left(P_{k+1} - P_k\right) \nabla_w P_t$$

changing summation order,

(see next slide)

$$= \sum_{t=1}^{m} \alpha \left(P_{t+1} - P_t\right) \sum_{k=1}^{t} \nabla_w P_k$$

factoring and changing indices

The "temporal difference"

Changing Summation Order

- Outer & inner summation:

  t = 1: k = 1, 2, 3, ..., m
  t = 2: k = 2, 3, ..., m
  t = 3: k = 3, 4, ..., m
  . . .
  t = m: k = m

$$\sum_{t=1}^{m} \sum_{k=t}^{m}$$

- Same as:

  k = 1: t = 1
  k = 2: t = 1, 2
  k = 3: t = 1, 2, 3
  . . .
  k = m: t = 1, 2, 3, ..., m

$$\sum_{k=1}^{m} \sum_{t=1}^{k}$$

Sutton's Derivation (10)

- From (9), the incremental weight change can be seen as

$$\Delta w_t = \alpha\left(P_{t+1} - P_t\right) \sum_{k=1}^{t} \nabla_w P_k$$

- In other words, weight change is based on the ***difference*** between current and previous predictions, times the sum of the gradients computed at previous steps.

Sutton's Derivation (11)

- If using backpropagation, for example, one would need to maintain a sum of the gradient values (weight changes) from previous steps.
- The method on the previous slides is called TD(1).
- For the *linear* case, TD(1) gives the same weight changes as Widrow-Hoff would.

Sutton's Derivation (12)

- TD(1) is **generalized** to TD(λ), $0 \leq \lambda \leq 1$.
- The value of λ is a **decay factor** indicating what portion of previous weight changes are to be added in.

$$\Delta w_t = \alpha\left(P_{t+1} - P_t\right) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k$$

- Lower values of λ give more weight to recent predictions.

Sutton's Derivation (13)

- Of special interest is TD(0) (note $0^0 = 1$):

$$\Delta w_t = \alpha(P_{t+1} - P_t)\nabla_w P_t$$

Change in value of prediction function

Gradient of P_t

- However, Bertsekas at MIT, 1995 showed by example that the TD(0) approximation can be inferior.

Possible Use of TD in Game-Playing

- For a given state of the game, **enumerate** the possible moves.
- Evaluate P_t (prediction of a win) for **each** state resulting from a possible move.
- Choose the move for which P_t is highest.
- Occasionally choose sub-optimal moves for purposes of exploration. (Opponents do not always play optimally.)

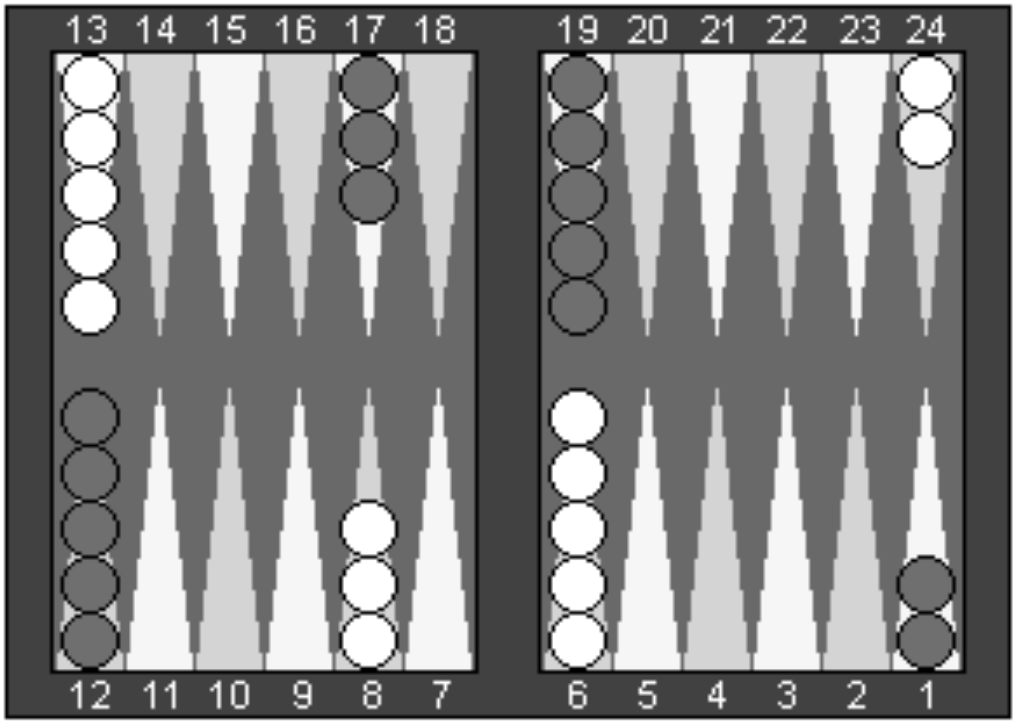
Case Study: Backgammon (Gerald Tesauro, 1995)

TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play

Gerald Tesauro
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
(tesauro@watson.ibm.com)

See also: http://www.research.ibm.com/massive/tdl.html

Backgammon Board



The normal opening position in backgammon

Summary of Backgammon

- Players roll dice and move their checkers from points according to the numbers shown on the dice.
- The sum of the number of points moved equals the number showing on the dice.
- Landing on another player's checker captures it.

Neurogammon

- Earlier program by the same author, 1989
- Trained using supervised learning (not TD):
  - 30,000 “expert opinions”
- Eventually augmented neural network with a traditional 2-ply AI search.

TD-gammon

- 2-layer network with:
  - 1 output (whether a proposed state is good or not)
  - 198 inputs
  - 40 or 80 hidden neurons
- Weight-update rule:

$$\Delta w_t = \alpha\left(P_{t+1} - P_t\right) \sum \lambda^{t-k} \nabla_w P_k$$

Board Encoding (1)

- 4 inputs encode the number of white pieces on each of 24 board points:
  - 0000: no pieces
  - 0001: one piece
  - 0011: two pieces
  - 0111: three pieces
  - x111: >3 pieces, x = (n-3)/2 for n pieces

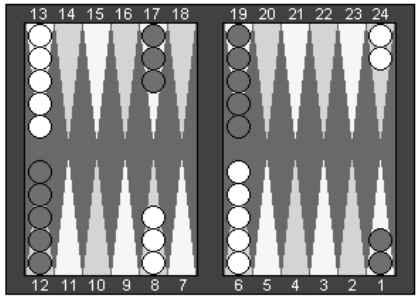


Figure 2. An illustration of the normal opening position in backgammon. TD-Gammon has sparked a near-universal conversion in the way experts play certain opening rolls. For example, with an opening roll of 4-1, most players have now switched from the traditional move of 13-9, 6-5, to TD-Gammon's preference, 13-9, 24-23. TD-Gammon's analysis is given in Table 2.

- 4x24 = 96 inputs for white + 96 for red

Board Encoding (2)

- Two more inputs encode number of pieces on the bar (n/2) for n pieces.
- Two more inputs encode the number of pieces removed (n/15).
- Two units encode whose turn to move.
- All unit inputs were roughly in the 0 to 1 range.

TD-gammon results world-class play

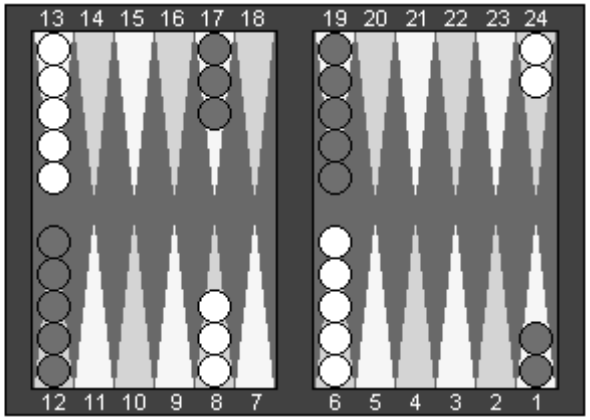
Program	Training Games	Opponents	Results
TDG 1.0	300,000	Robertie, Davis, Magriel	-13 pts/51 games (-0.25 ppg)
TDG 2.0	800,000	Goulding, Woolsey, Snellings, Russell, Sylvester	-7 pts/38 games (-0.18 ppg)
TDG 2.1	1,500,000	Robertie	-1 pt/40 games (-0.02 ppg)

Results of testing TD-gammon in play against world-class human opponents. Version 1.0 used 1-ply search for move selection; versions 2.0 and 2.1 used 2-ply search. Version 2.0 had 40 hidden units; versions 1.0 and 2.1 had 80 hidden units.

TD-gammon results

- In 1994, TD-Gammon was at the level of the best human players in the world.
- Expert players learned new strategy from the program.

Judgement superior to experts?



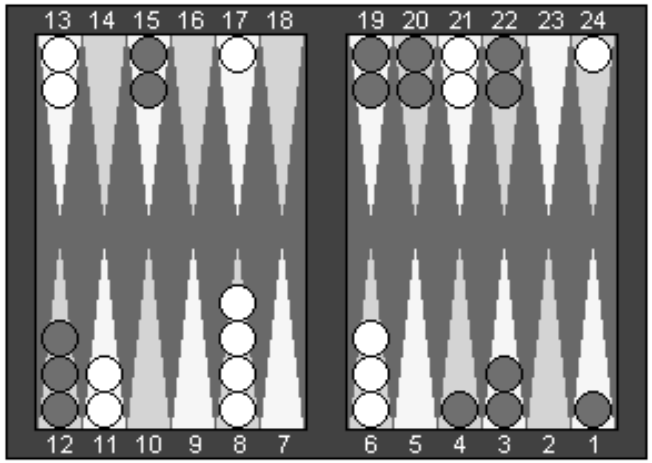
Move	Estimate	Rollout
13-9, 6-5 13-9, 24-23	−0.014 +0.005	−0.040 +0.005

An illustration of the normal opening position in backgammon.

TD-Gammon has sparked a near-universal conversion in the way experts play certain opening rolls. For example, with an opening roll of 4-1, most players have now switched from the traditional move of 13-9, 6-5 to TD-Gammon's preference, 13-9, 24-23.

TD-Gammon's analysis of the two choices: The estimated equity is the neural network's output at the 1-ply level (i.e. no lookahead). The rollout is actual outcome of playing each position out 10,000 times to completion with different random dice sequences. Standard deviation in the rollout results is approximately 0.01.

Judgement superior to experts?



Move	Estimate	Rollout
8-4*, 8-4, 11-7, 11-7	+0.184	+0.139
8-4*, 8-4, 21-17, 21-17	+0.238	+0.221

Table 3. TD-Gammon's analysis of the two choices in Figure 3. The estimated equity is the neural network's output at the 1-ply level (i.e., no lookahead). The rollout is actual outcome playing each position out 10,000 times to completion with different random dice sequences (see the appendix). Standard deviation in the rollout results is approximately 0.01.

Figure 3: A complex situation where TD-Gammon's positional judgment was apparently superior to traditional expert thinking. White is to play 4-4. The obvious human play is 8-4*, 8-4, 11-7, 11-7. (The asterisk denotes that an opponent checker has been hit.) However, TD-Gammon's choice is the surprising 8-4*, 8-4, 21-17, 21-17! TD-Gammon's analysis of the two plays is given in Table 3.

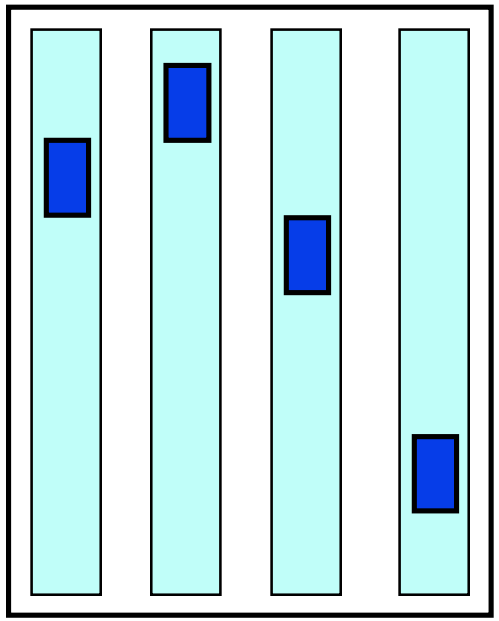
Other TD uses

- Checkers (A. Samuel)
- Go
- Othello
- Chess
- AHC (Adaptive Heuristic Critic): pole-balancing, etc. (Barto, Sutton, and Anderson)

Example: Elevator Dispatching

Crites and Barto, 1996

10 floors, 4 elevator cars



STATES: button states; positions, directions, and motion states of cars; passengers in cars & in halls

ACTIONS: stop at, or go by, next floor

REWARDS: roughly, –1 per time step for each person waiting

Conservatively about 10^{22} states

from R. S. Sutton and A. G. Barto: *Reinforcement Learning: An Introduction*

Further Background of TD

- Touretzky et al. mention that, in devising TD, Barto and Sutton extended the Rescorla-Wagner model [which is the same as the Widrow-Hoff rule and thus suffers from its linear limitations].

- Rescorla and Wagner, “A Theory of **Pavlovian** conditioning”, Classical Conditioning II: Theory and Research, Black and Prokasy (eds), Appleton-Century-Crofts, 1972.

http://en.wikipedia.org/wiki/Rescorla-Wagner_model

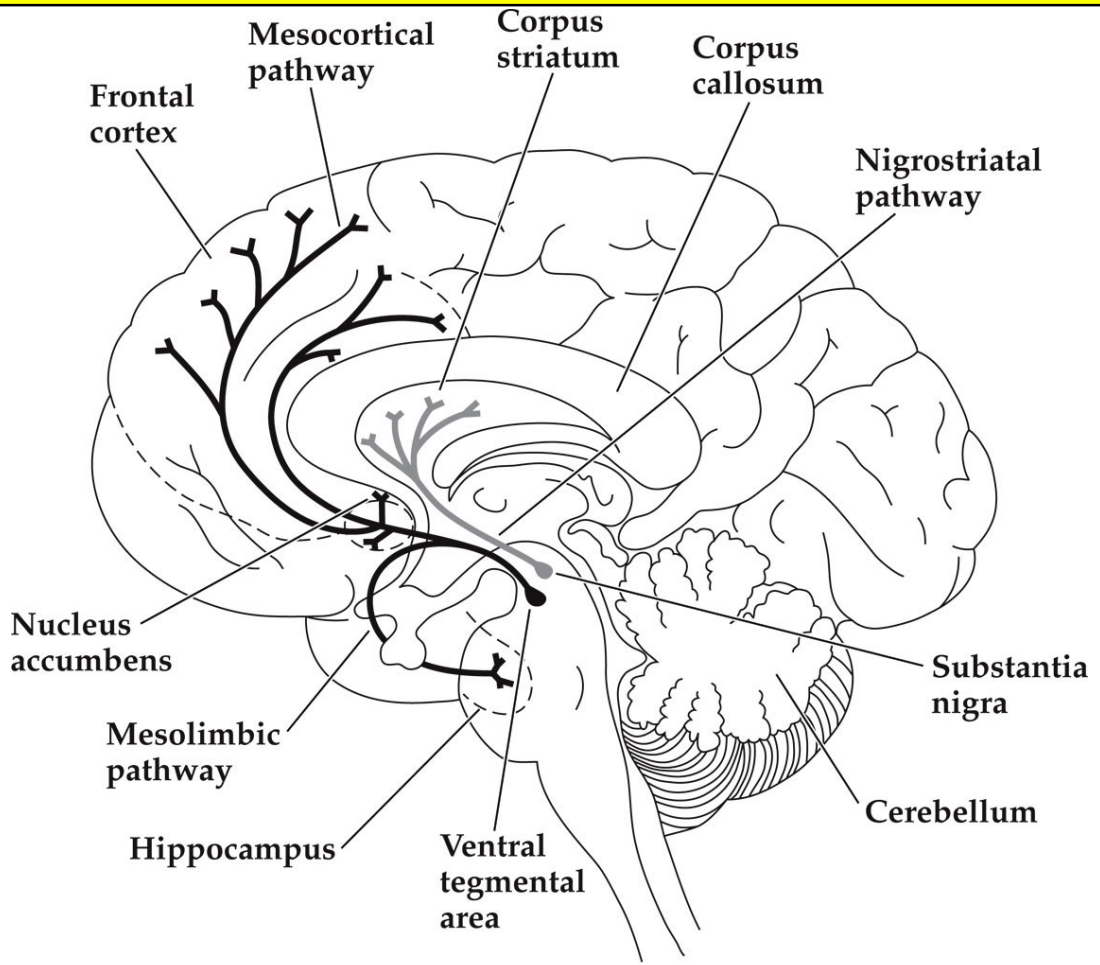
Biological Applications of TD

- A neural substrate of prediction and reward.
  Schultz W, Dayan P, Montague PR.
  Science. 1997 Mar 14; 275(5306):1593-9

The **capacity to predict future events** permits a creature to detect, model, and manipulate the causal structure of its interactions with its environment. Behavioral experiments suggest that **learning is driven by changes in the expectations about future salient events such as rewards and punishments**. Physiological work has recently complemented these studies by identifying dopaminergic neurons in the primate whose fluctuating output apparently signals changes or errors in the predictions of future salient and rewarding events. **Taken together, these findings can be understood through quantitative theories of adaptive optimizing control** [i.e. the TD model].

Figure 11.9 The midbrain dopamine pathways

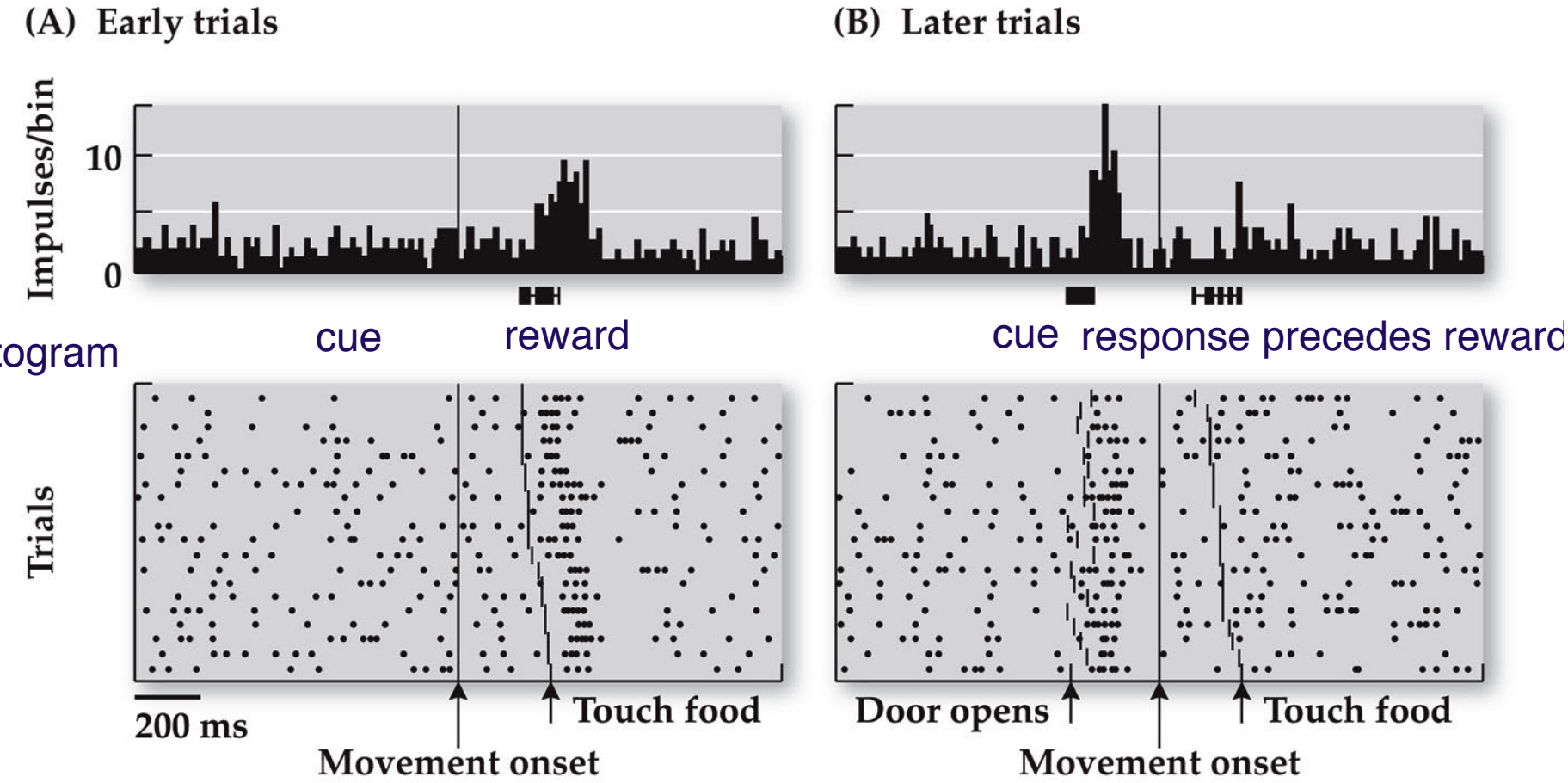
Stimulation of dopamine neurons becomes the reward.



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.9

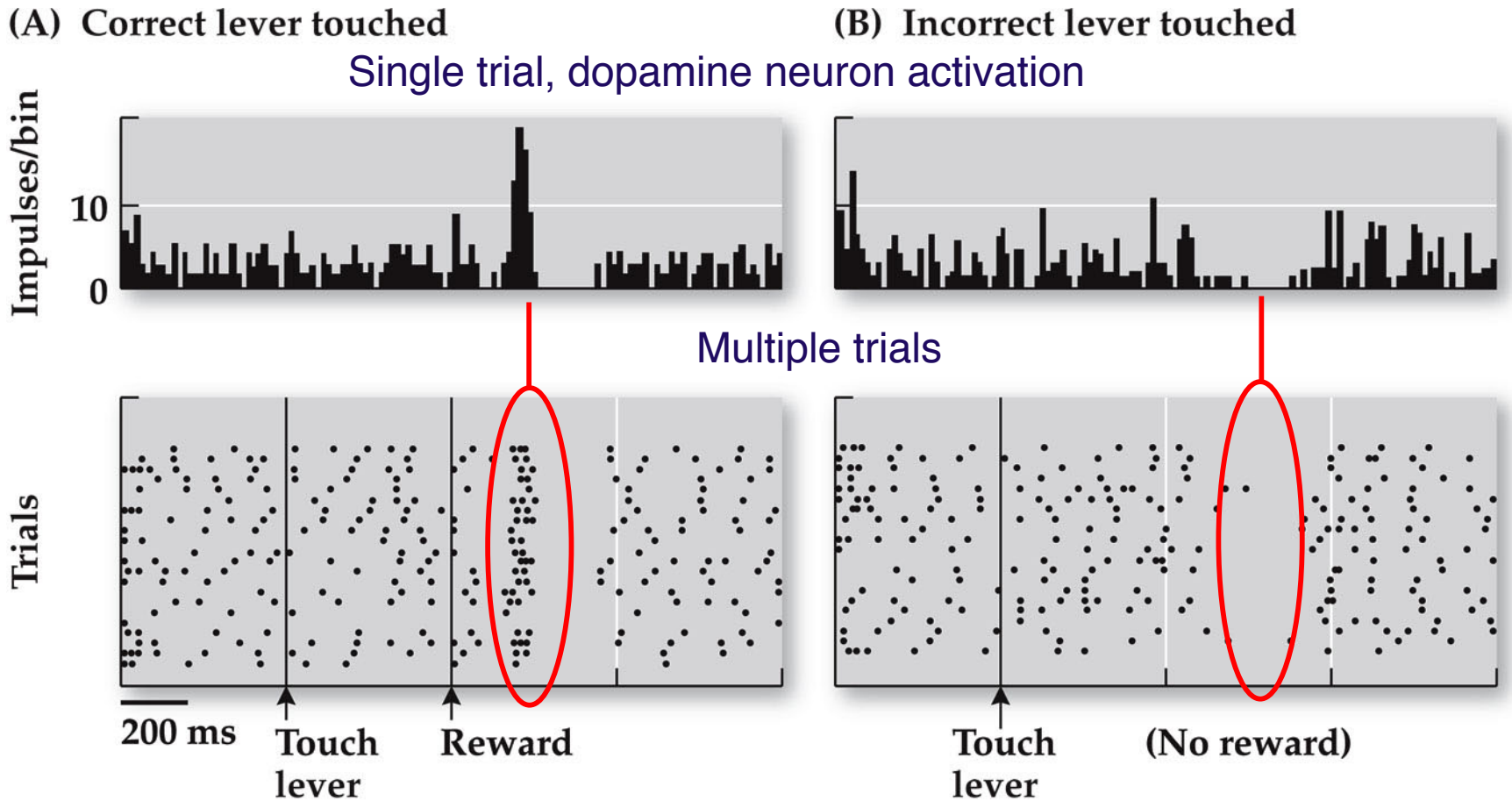
Dopamine neuron learns to predict future reward

Single trial, **dopamine neuron activation**



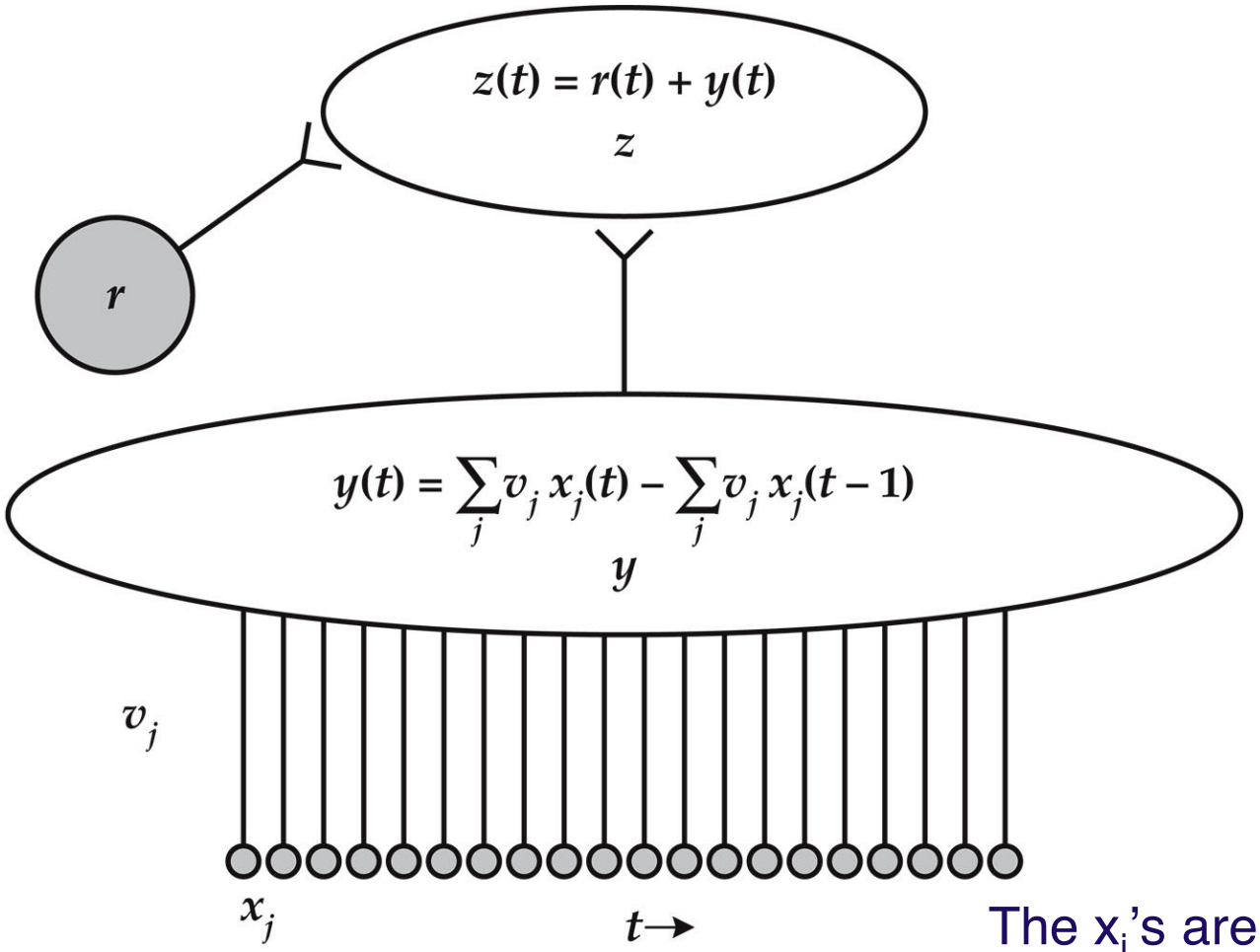
TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.10

Dopamine neuron enhanced based on monkey touching correct lever, but response later suppressed if no reward given.



TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.11

Figure 11.12 Temporal-difference learning implemented in a neural network model of the midbrain dopamine system



The x_j's are time-based.

TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 11.12

Explanation of the Model of Figure 11.12

- The model is set up for 50 time steps.
- On a given trial, the input units x_j are sequenced as follows:
  - The cue comes on at time 10, setting $x_1 = 1$, with all other $x_j = 0$.
  - At the next time step, $x_1 = 0$ and $x_2 = 1$.
  - and so on, for 20 time steps (20 input units).
- The weight v_j of input x_j represents the value of state j.
- Because only input is non-zero at a given time, the output y(t) represents the **difference** in overall value v(t) - v(t-1).
- The value r(t) represents **reward** at time t. It is 0, except possibly at time 30, when it is set to 1.
- The value z(t) = r(t) + y(t), which is the **temporal difference learning value** (used to update the v_j's):
  - $\Delta v_j = a x_j (t-1)\ z(t)$

Example Execution (for a simpler 5 input version): Trial 1

TABLE 11.4 **Some steps in the operation of a smaller version of the temporal-difference learning model of midbrain dopamine neurons** *(Part 1)*

(A) First trial, dopamine unit responds to reward and updates the value of input 5

	Time step t	Index j of input x_j	Value or input weight v_j	Difference unit y	Reward unit r	Dopamine unit z
	0		0	0	0	0
Cue	1	1	0	0	0	0
	2	2	0	0	0	0
	3	3	0	0	0	0
	4	4	0	0	0	0
	5	5	0 (goes to 1)	0	0	0
Reward	6		0	0	+1	+1

This smaller version of the midbrain dopamine model has only five input units. *Abbreviations*: t, time step ($t = 0,\ldots, 6$); j, index of active input unit x_j ($j = 1,\ldots, 5$); v_j, value (or weight of connection from input unit x_j to the difference unit y); y, difference unit; r, reinforcement unit; z, prediction error (dopamine) unit (see also Figure 11.12).

TUTORIAL ON NEURAL SYSTEMS MODELING, Table 11.4 (Part 1)

Example Execution: Trial 2

TABLE 11.4 **Some steps in the operation of a smaller version of the temporal-difference learning model of midbrain dopamine neurons** *(Part 2)*

(B) Second trial, dopamine unit responds to input 5 and updates the value of input 4

	Time step t	Index j of input x_j	Value or input weight v_j	Difference unit y	Reward unit r	Dopamine unit z
	0		0	0	0	0
Cue	1	1	0	0	0	0
	2	2	0	0	0	0
	3	3	0	0	0	0
	4	4	0 (goes to 1)	0	0	0
	5	5	1	+1	0	+1
Reward	6		0	−1	+1	0

This smaller version of the midbrain dopamine model has only five input units. *Abbreviations*: t, time step ($t = 0,\ldots, 6$); j, index of active input unit x_j ($j = 1,\ldots, 5$); v_j, value (or weight of connection from input unit x_j to the difference unit y); y, difference unit; r, reinforcement unit; z, prediction error (dopamine) unit (see also Figure 11.12).

***TUTORIAL ON NEURAL SYSTEMS MODELING*, Table 11.4 (Part 2)**

Example Execution: Trial 6

TABLE 11.4 **Some steps in the operation of a smaller version of the temporal-difference learning model of midbrain dopamine neurons** *(Part 3)*

(C) Sixth trial, dopamine unit responds to input 1, all input values have been updated

	Time step t	Index j of input x_j	Value or input weight v_j	Difference unit y	Reward unit r	Dopamine unit z
	0		0	0	0	0
Cue	1	1	1	+1	0	+1
	2	2	1	0	0	0
	3	3	1	0	0	0
	4	4	1	0	0	0
	5	5	1	0	0	0
Reward	6		0	–1	+1	0

This smaller version of the midbrain dopamine model has only five input units. *Abbreviations*: t, time step ($t = 0,\ldots, 6$); j, index of active input unit x_j ($j = 1,\ldots, 5$); v_j, value (or weight of connection from input unit x_j to the difference unit y); y, difference unit; r, reinforcement unit; z, prediction error (dopamine) unit (see also Figure 11.12).

***TUTORIAL ON NEURAL SYSTEMS MODELING*, Table 11.4 (Part 3)**

Example Execution: Reward Removed

TABLE 11.4 **Some steps in the operation of a smaller version of the temporal-difference learning model of midbrain dopamine neurons** *(Part 4)*

(D) Remove reward, dopamine unit responds to input 1, but goes negative on time step 6

	Time step t	Index j of input x_j	Value or input weight v_j	Difference unit y	Reward unit r	Dopamine unit z
	0		0	0	0	0
Cue	1	1	1	+1	0	+1
	2	2	1	0	0	0
	3	3	1	0	0	0
	4	4	1	0	0	0
	5	5	1	0	0	0
	6		0	−1	0	−1

This smaller version of the midbrain dopamine model has only five input units. *Abbreviations*: t, time step ($t = 0,\ldots, 6$); j, index of active input unit x_j ($j = 1,\ldots, 5$); v_j, value (or weight of connection from input unit x_j to the difference unit y); y, difference unit; r, reinforcement unit; z, prediction error (dopamine) unit (see also Figure 11.12).

TUTORIAL ON NEURAL SYSTEMS MODELING, Table 11.4 (Part 4)

TD Learning using a Robot

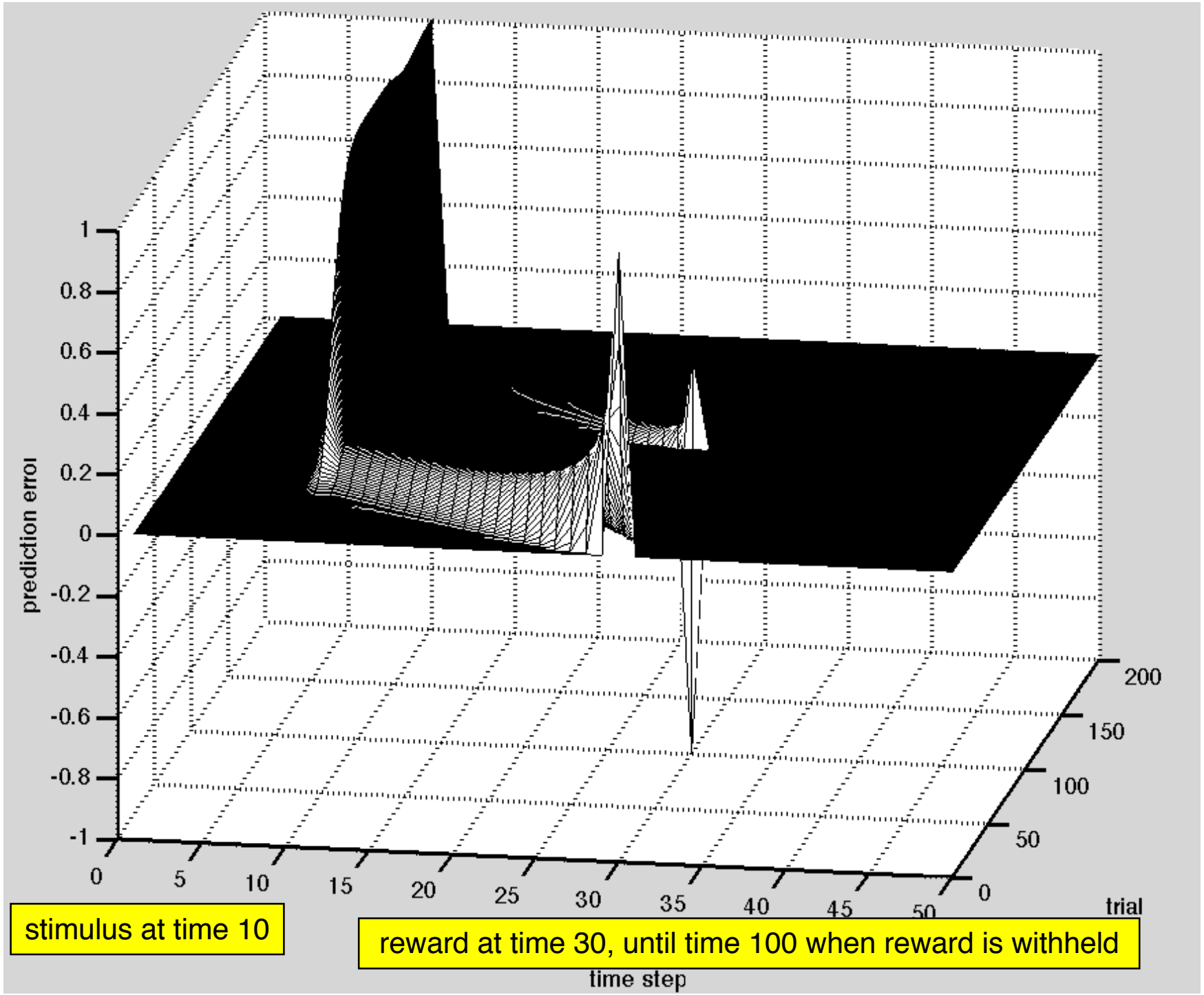
- Touretzky, Daw, and Tira-Thompson, Combining Configural and TD Learning on a Robot, Proc. 2nd Intl. Conf. on Development and Learning, 2002.
- Sony AIBO robot, model ERS-210
- Combine configural and TD learning in a classical conditioning model:
  - Solved the negative patterning problem
  - Discriminate sequences of stimuli
  - Exhibit second-order conditioning
  - Real-time interaction

http://www.youtube.com/watch?v=F7wtKynf58s

Multi-Trial Code

- The code on the next page compresses each trial into single elements of multiple vectors, each showing the time of cue onset, difference unit, dopamine response, etc.

```
% midbrainDopamine.m
% this script simulates the midbrain dopamine system

a=0.3;                                  % set learning rate
nTrials=200;                            % set number of trials
nTimes=50;                              % set number of time steps per trial

x=zeros(nTimes,1);                      % define input unit vector
y=zeros(nTimes,1);                      % define difference unit vector
v=zeros(nTimes,1);                      % define weight (value estimate) vector
r=zeros(nTimes,1);                      % define reward vector
z=zeros(nTimes,1);                      % define prediction unit vector
Tcourse=zeros(nTrials,nTimes);          % define time course hold array

qTime=10;                               % set time of cue
rTime=30;                               % set time of reward

x(qTime:rTime-1)=1;                     % set input responses

for c=1:nTrials,                        % for each learning trial
    r(rTime)=1;                         % set the reward at reward time
    if c==nTrials/2, r(rTime)=0; end    % withhold reward once
    y=[0; diff(v.*x)];                  % find the response of difference unit
    z=y+r;                              % find the response of prediction error unit
    v=v+a*x.*[z(2:nTimes);0];           % update the weights (values)
    Tcourse(c,:)=z';                    % save the prediction unit time course
end % end learning trial loop
```


prediction error
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
5
10
15
20
25
30
35
40
45
50
time step
0
50
100
150
200
trial
stimulus at time 10
reward at time 30, until time 100 when reward is withheld