Game Applications

Reinforcement Learning

Temporal Differences

Game Applications

- BPPT seems potentially useable.
- Another approach is to use the "Temporal Difference" method.

Learning Types

- **Supervised learning**: Training with desired answer given for each action
- **Unsupervised learning**: No desired answer; learns *similarities* (clustering)
- **Reinforcement learning**: **Reward** given later, not necessarily tied to specific action

Example: Game Playing

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

  Listen to a **teacher** indicate good moves in various situations, or

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

Problems with Supervised

- Need access to expert or many recorded game samples.
- There is generally no play-by-play target value; the value is only assigned to a sequence of plays, ending with +1 (win) or -1 (lose).
- The teacher might not be perfect.

Reinforcement Learning

- Tries to address difficulties with supervised learning.
- Reward can be deferred until the end of the game.
- Can deal with stochastic environment (transition probabilities).

Typical AI Model: MDP (“Markov Decision Problem”)

- Set of states, maybe very large
- Set of actions
- Transition probabilities between states: M^a_{ij} = prob[transition from state i to state j given action *a* is taken in i]
- Reward R(i) (positive, negative, or 0) associated with each state i.
- The objective is to accrue as much reward as possible.

Utility Functions

- Desirability of moving to a given state s is expressed by a ***utility*** U(i).
- The utility is generally not given explicitly; it must be *learned*.
- Normally the ***expected*** utility is sought, since transitions can be probabilistic.
- Generally, given a choice of moves to several states, the state with ***highest expected utility*** will be chosen.

Additive Utility Function

- Assume that the sought utility function is *additive*:

  $U(i) = R(i) + \max_a(\sum_j M^a_{ij} \cdot U(j))$ Utility of state i

  where R(i) is the reward of state i, a is an action, and M^a_{ij} is the probability of going from state i to state j with action a.
- This is the *dynamic programming* equation (Richard Bellman).

A conceptual way to compute U

- Initialize U to R, the reward function
- While(not converged)

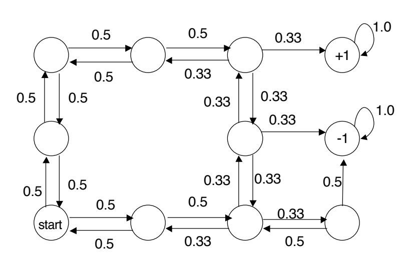
```
{
foreach state i, compute
   U'(i) = R(i) + max_a(Σ_j M^a_ij • U( j ))
replace U with U'
}
```

3x4 World 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.

			+1
			-1
start			

Markov State-Transition Probs



Computed Utilities using Dynamic Programming Eqn

-0.03	0.09	0.22	1.0
-0.15		-0.43	-1.0
-0.28	-0.40	-0.53	-0.76

Problem with this Method for Games

- There are generally too many states in a game to:
  - enumerate
  - compute their utilities

Reinforcement Learning using Temporal Differences

Temporal Difference Learning

- Through many trial runs, adjust the observed values of U so that they more closely agree with the dynamic programming equation. More specifically,
- if there is a **transition from i to j**, adjust U(i) so that it **better agrees** with U(j).
- Temporal Difference updating rule (with α the learning rate):

$$\Delta U(i) = \alpha \cdot (R(i) + U(j) - U(i))$$

Utilities Computed by TD vs. by Dynamic Programming

This was after 1 million cycles. The values were not yet stable.

TD	-0.01	0.05	0.14	1.0
DP	-0.03	0.09	0.22	
	-0.07		-0.5	-1.0
	-0.15		-0.43	
	-0.11	-0.16	-0.46	-0.76
	-0.28	-0.40	-0.53	-0.76

Game Playing

- In game playing it is generally infeasible to enumerate all states.
- However, we can apply the TD updating rule in the course of play.
- This will tend to explore regions of the state space that tend to occur in actual play.

More on TD method

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

Single- vs. Multi-Step Prediction

- In single-step prediction, the outcome is revealed right after the prediction.
- In multi-step prediction, the outcome is delayed for several or many steps.
- TD is applied in multi-step cases (in single-step it is identical to supervised learning).

Sutton's Derivation of TD

- 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 **outcome** of the sequence
- All values are real numbers
- Learner produces **predictions** that estimate the final outcome z:
  $P_1, P_2, P_3, \ldots., P_m$

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 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 previous 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 $x_1, x_2, x_3, \ldots., x_m$, is thus:

$$\sum_t \Delta w_t$$

Sutton's Derivation (5)

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

$$\Delta w_t = \underset{\text{learning rate}}{\alpha}\underbrace{(z - P_t)}_{\text{difference between outcome and prediction}}\underbrace{\nabla_w P_t}_{\text{gradient of prediction function}}$$

Sutton's Derivation (6)

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

  and we have the Widrow-Hoff rule:

$$\Delta w_t = \alpha\underbrace{(z - w^T x_t)}_{\text{difference between outcome and prediction}}\bullet \underbrace{x_t}_{\text{gradient of prediction function}}$$

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"

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

Sutton's Derivation (9)

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

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

Incremental change, slide 5: $\Delta w_t = \alpha(z - P_t)\nabla_w P_t$

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

from telescoping, slide 8

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

changing summation order, (see next slide)

$$= \sum_{t=1}^{m} \alpha \underbrace{(P_{t+1} - P_t)}_{\text{The "temporal difference"}} \sum_{k=1}^{t} \nabla_w P_k$$

factoring and changing indices

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 slide 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 prediction times the sum of the gradients computable at previous steps.

Sutton’s Derivation (11)

- If using backprop, 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), where λ is any value between 0 and 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 $0^0 = 1$):

$$\Delta w_t = \alpha \underbrace{(P_{t+1} - P_t)}_{\text{Change in value of prediction function}} \underbrace{\nabla_w P_t}_{\text{Gradient of } P_t}$$

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

Tic-Tac-Toe

- An example exists on turing:
  /cs/cs152/ttt
- It uses TD and backprop to learn to play tic-tac-toe.
- The source is courtesy of:
  http://www.geocities.com/chen_levkovich/tdprojectsources.html

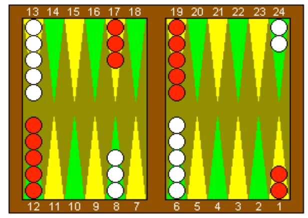
Case Study: Backgammon (Gerald Tesauro, 1995)

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.

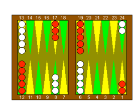
TD-gammon

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

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

Board Encoding Important (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 fo



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

Board Encoding Important (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 *it*.

Judgement superior to experts?

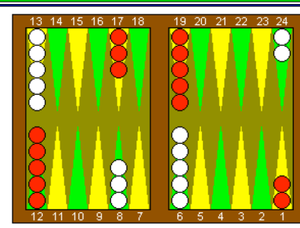


Figure 2: An illustration of the normal opening position in backgammon. TD-Gammon has sparked a near-universal conversion in the way experts play certain opening rolls. For example, with an opening roll of 4-1, most players have now switched from the traditional move of 13-9, 6-5 to TD-Gammon's preference,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

Table 2

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

Table 2: TD-Gammon's analysis of the two choices in Figure 2. 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?

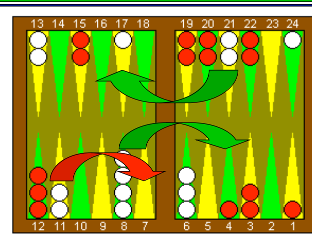


Figure 3: A complex situation where TD-Gammon's positional judgment is apparently superior to traditional expert thinking. White is to play 4-4. The obvious human play is 8-4*, 8-4, 11-7, 11-7. (The asterisk denotes that an opponent checker has been hit.) However, TD-Gammon's choice is the surprising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

Table 3

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

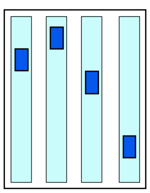
Table 3. TD-Gammon's analysis of the two choices in Figure 3. The estimated equity is the neural network's output at the 1-ply level (i.e., no lookahead). The rollout is actual outcome playing each position out 10,000 times to completion with different random dice sequences (see the appendix). Standard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

Other TD uses

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

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*