Reinforcement Learning

So far, we had a well-defined set of training examples.
What if feedback is not so clear?
E.g., when playing a game, only after many actions
final result: win, loss, or draw.

Issue: learning via **delayed rewards / delayed feedback**.

One success: Tesauro's backgammon player (TD Gammon)
Start from random play; millions of games
World-level performance (changed game itself)

Imagine agent wandering around in environment.
How does it learn **utility** values of each state?

(i.e., what are good / bad states? avoid bad ones...)

Reinforcement learning will tell us how! Variations:

- environment accessible or inaccessible
- have model of environment and effects of action...or not
- rewards in terminal states only; or in any state
- agent can be passive (watch) or active (explore)

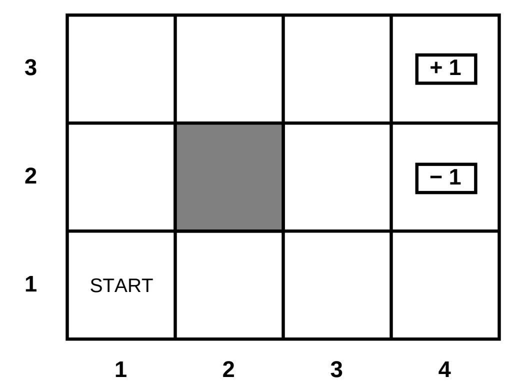
Reinforcement Learning for Backgammon

In backgammon: states = boards.
Only clear feedback in final states (win/loss).

We want to know **utility** of the other states.
Intuitively: utility = chance of winning

At first, we only know this for the end states.
Reinforcement learning: computes for intermediate states. Play by moving to maximum utility states!

back to simplified world . . .



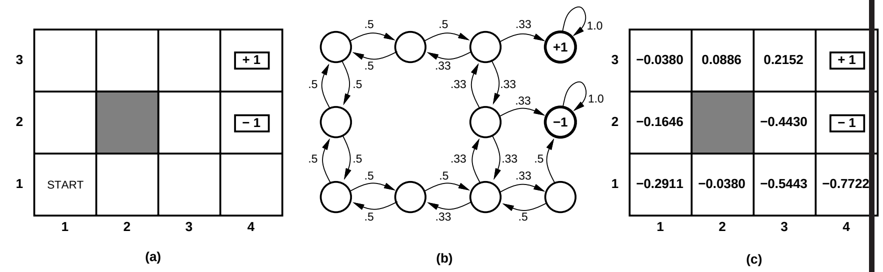
Passive Learning in a Known, Accessible Environment

Agent just wanders from state to state.
Each transition is made with a fixed probability.

Initially: only two known reward positions:

State (4,2) — a loss / poison / reward −1 (utility)
State (4,3) — a win / food / reward +1 (utility)

How does the agent learn about the utility, i.e., **expected value**, of the other states?



Three strategies:

(a) “Direct Sampling” (Adaptive control theory)
naive updating - LMS rule

(b) “Calculation” / “Equation solving”
dynamic programming

(c) “in between (a) and (b)”
Temporal Difference Learning — TD learning
used for backgammon

Naive updating (LMS approach)

Widrow and Hoff [1960]

(a) “Sampling” — agent makes random runs through environment; collect statistics on final payoff for each state (e.g. when at (2,3), how often do you reach +1 vs. −1?)

Learning algorithm keeps a running average for each state. Provably converges to true expected values (utilities).

U table of current utilities

e a unique state in the environment

percepts list of e's seen so far

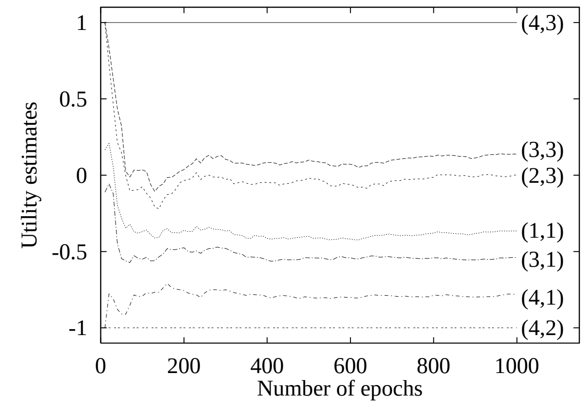
M model of environment

N table of visit frequencies

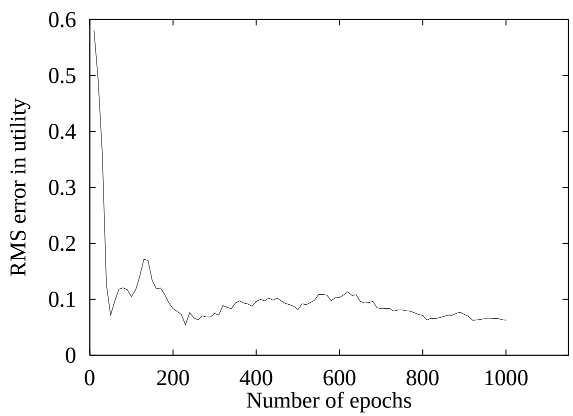
```
function LMS-UPDATE(U, e, percepts, M, N) returns an updated U

  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
```


Naive updating: direct utility estimation



Naive updating: direct utility estimation



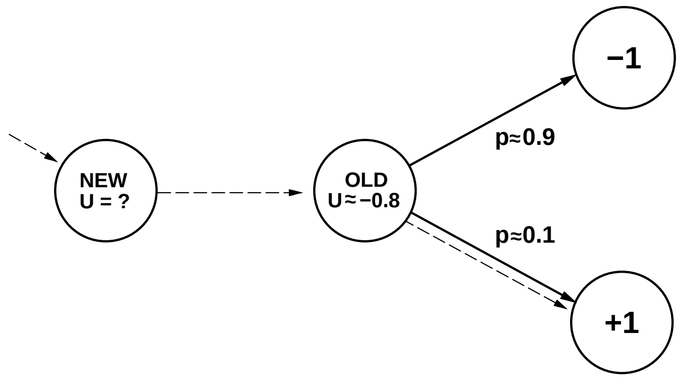
Problems

Ignores structure of transitions which impose 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.

Main effect: **slow convergence**.

Example where LMS does poorly



Dynamic programming

Consider $U(3,3)$

From figure we see that

$$U(3,3) = 0.33 \times U(4,3) + 0.33 \times U(2,3) + 0.33 \times U(3,2)$$
$$= 0.33 \times 1.0 + 0.33 \times 0.0886 + 0.33 \times -0.4430$$
$$= 0.2152$$

Check e.g. $U(3,1)$ yourself.

Utilities follow basic laws of probabilities:

write down equations; solve for unknowns.

Utilities follow from:

$$U(i) = R(i) + \sum_j M_{i,j} U(j) \qquad (\star)$$

(note: i, j over states.)
$R(i)$ is the reward associated with being in state i.
(often non-zero for only a few end states)
$M_{i,j}$ is the probability of transition from state i to j.

Dynamic programming style methods can be used to solve the set of equations.

Major drawback: number of equations and number of unknowns.

E.g. for backgammon: roughly 10^{50} equations with 10^{50} unknowns. Infeasibly large.

Temporal difference learning

Combine "sampling" with "calculation"

Or stated differently: TD-learning uses a sampling approach to solve the set of equations.

Consider the transitions, observed by a wandering agent.

Use the observed transitions to adjust the utilities of the observed states to bring them closer to the constraint equations.

Temporal difference learning

When observing a transition from i to j,
bring $U(i)$ value closer to that of $U(j)$

Use update rule:

$$U(i) \leftarrow U(i) + \alpha(R(i) + U(j) - U(i)) \quad (\star\star)$$

α is the **learning rate** parameter

rule is called the **temporal-difference** or **TD** equation (because we take the difference in utilities between successive states).

function TD-UPDATE($U, e, percepts, M, N$) **returns** the utility table U

```
if TERMINAL?[e] then
    U[STATE[e]] ← RUNNING-AVERAGE(U[STATE[e]], REWARD[e], N[STATE[e]])
else if percepts contains more than one element then
    e' ← the penultimate element of percepts
    i, j ← STATE[e'], STATE[e]
    U[i] ← U[i] + α(N[i])(REWARD[e'] + U[j] - U[i])
```


At first blush, the rule:

$$U(i) \leftarrow U(i) + \alpha(R(i) + U(j) - U(i)) \ (\star\star)$$

may appear to be a bad way to solve/approximate:

$$U(i) = R(i) + \sum_j M_{i,j} U(j) \ (\star)$$

Note that $(\star\star)$ brings $U(i)$ closer to $U(j)$ but in $(\star)$ we really want the **weighted** average over the neighboring states!

Issue resolves itself, because over time, we **sample** from the transitions out of i. So, successive applications of $(\star\star)$ average over neighboring states.

(keep α appropriately small)

Performance

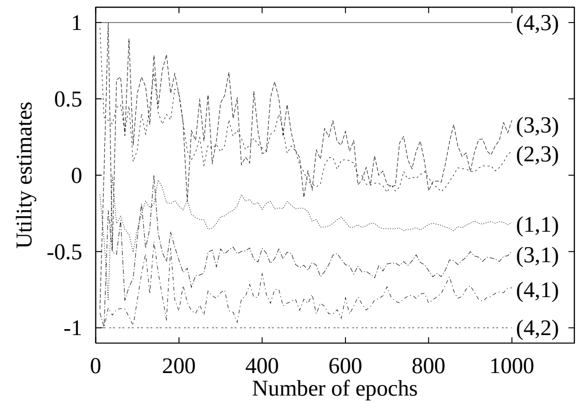
Runs noisier than Naive Updating (averaging), but smaller error.

In our 4x3 world, we get a root-mean-square error of less than 0.07 after 1000 examples.

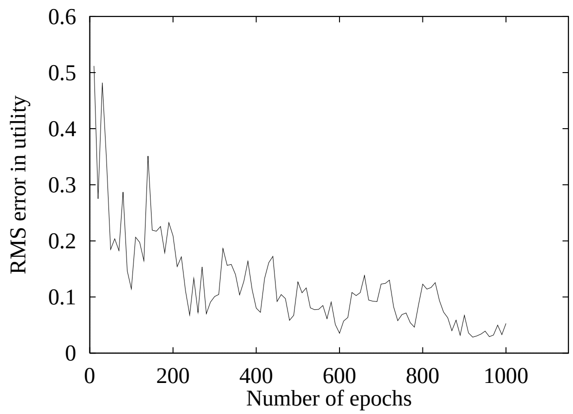
Also, note that compared to Dynamic Programming we only deal with *observed* states during sample runs.

I.e., in backgammon consider only a few hundreds of thousands of states out of 10^{50}. Represent utility function **implicitly** (no table) in neural network.

TD-learning



TD-learning



Reinforcement learning is a very rich area of study.

In some sense, touches on much of the core of AI. "How does an agent learn to take the right actions in its environment?"

In general, pick action that leads to state with highest utility as learned so far.

Extensions

— **Active learning** — exploration.

now and then make new (non utility optimizing move)

— **Learning action-value functions**

$Q(a, i)$ denotes value of taking action a in state i

we have: $U(i) = max_a Q(a, i)$

— **Generalization in reinforcement learning**

Use implicit representation of utility function

e.g. a neural network as in backgammon.

Input nodes encode board position;

activation of output node gives utility.