Artificial Intelligence

Topic 8

Reinforcement Learning

- ◇ passive learning in a known environment
- ◇ passive learning in unknown environments
- ◇ active learning
- ◇ exploration
- ◇ learning action-value functions
- ◇ generalisation

Reading: Russell & Norvig, Chapter 20, Sections 1–7.

1. Reinforcement Learning

Previous learning examples

- supervised — input/output pairs provided
  eg. chess — given game situation and best move

Learning can occur in much less generous environments

- no examples provided
- no model of environment
- no utility function
  eg. chess — try random moves, gradually build model of environment and opponent

Must have *some* (absolute) feedback in order to make decision.

eg. chess — comes at end of game

$\Rightarrow$ called *reward* or *reinforcement*

Reinforcement learning — use rewards to learn a successful agent function

1. Reinforcement Learning

Harder than supervised learning

eg. reward at end of game — which moves were the good ones?

. . . but . . .

only way to achieve very good performance in many complex domains!

Aspects of reinforcement learning:

- *accessible* environment — states identifiable from percepts
  inaccessible environment — must maintain internal state
- model of environment known or learned (in addition to utilities)
- rewards only in terminal states, or in any states
- rewards components of utility — eg. dollars for betting agent or hints — eg. "nice move"
- *passive learner* — watches world go by
  active learner — act using information learned so far, use problem generator to explore environment

1. Reinforcement Learning

Two types of reinforcement learning agents:

utility learning

- agent learns utility function
- selects actions that maximise expected utitility

Disadvantage: must have (or learn) model of environment — need to know where actions lead in order to evaluate actions and make decision

Advantage: uses "deeper" knowledge about domain

Q-learning

- agent learns *action-value* function
  — expected utility of taking action in given state

Advantage: no model required

Disadvantage: shallow knowledge

— cannot look ahead

— can restrict ability to learn

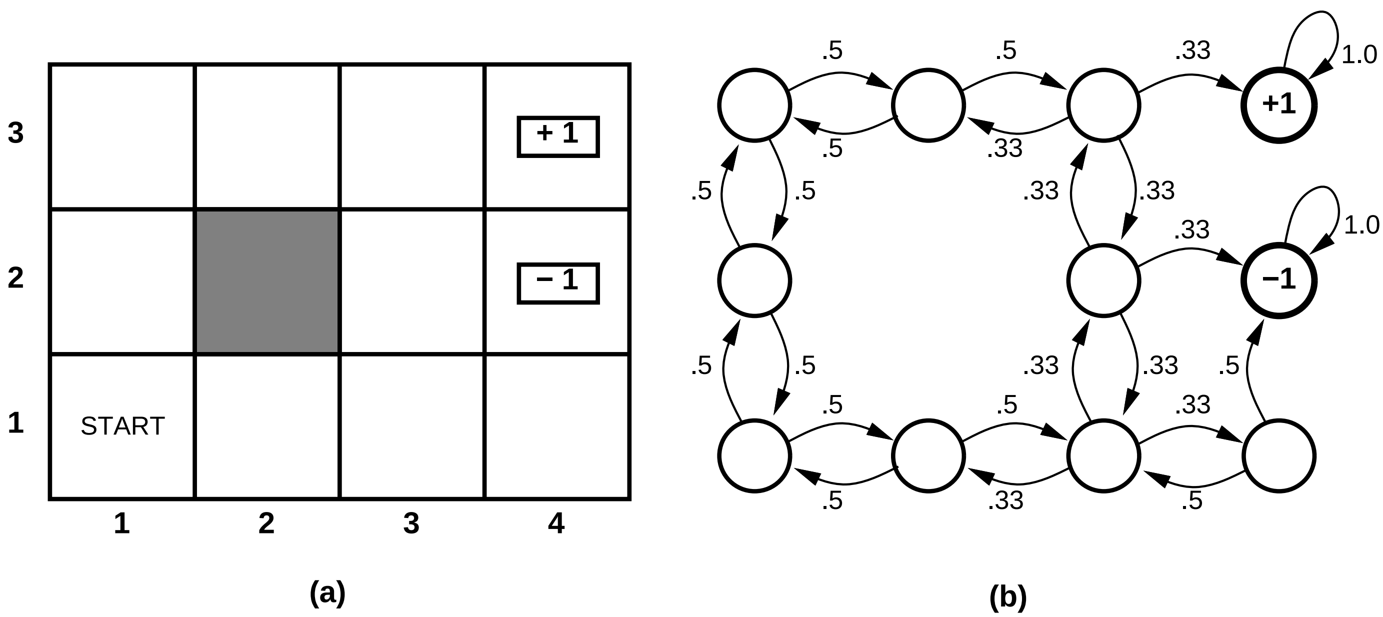
We start with utility learning. . .

2. Passive Learning in a Known Environment

Assume:

- accessible environment
- effects of actions known
- actions are selected *for* the agent $\Rightarrow$ *passive*
- known model M_{ij} giving probability of transition from state i to state j

Example:



(a) environment with utilities (rewards) of terminal states
(b) transition model M_{ij}

Aim: *learn utility values for non-terminal states*

2. Passive Learning in a Known Environment

Terminology

Reward-to-go $=$ sum of rewards from state to terminal state

additive utilitly function: utility of sequence is sum of rewards accumulated in sequence

Thus for additive utility function and state s:

expected utility of s = expected reward-to-go of s

Training sequence eg.

(1,1) $\rightarrow$(2,1) $\rightarrow$(3,1) $\rightarrow$(3,2) $\rightarrow$(3,1) $\rightarrow$(4,1) $\rightarrow$(4,2) [-1]
(1,1) $\rightarrow$(1,2) $\rightarrow$(1,3) $\rightarrow$(1,2) $\rightarrow \cdots \rightarrow$(3,3) $\rightarrow$(4,3) [1]
(1,1) $\rightarrow$(2,1) $\rightarrow \cdots \rightarrow$(3,2) $\rightarrow$(3,3) $\rightarrow$(4,3) [1]

Aim: use *samples* from training sequences to *learn* (an approximation to) expected reward for all states.

ie. generate an *hypothesis* for the utility function

Note: similar to sequential decision problem, except rewards initially unknown.

2.1 A generic passive reinforcement learning agent

Learning is iterative — successively update estimates of utilities

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]]
  $U \leftarrow$ UPDATE($U, e, percepts, M, N$)
  if TERMINAL?[e] **then** *percepts* $\leftarrow$ the empty sequence
  return the action *Observe*

Update

- after transitions, or
- after complete sequences

update function is one key to reinforcement learning

Some alternatives $\cdots \rightsquigarrow$

2.2 Naïve Updating — LMS Approach

From Adaptive Control Theory, late 1950s

Assumes:

observed rewards-to-go $\rightarrow$ actual expected reward-to-go

At end of sequence:

- calculate (observed) reward-to-go for each state
- use observed values to update utility estimates

eg, utility function represented by table of values — maintain running average...

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

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

2.2 Naïve Updating — LMS Approach

Exercise

Show that this approach minimises *mean squared error (MSE)* (and hence *root mean squared (RMS)* error) w.r.t. observed data.

That is, the hypothesis values x_h generated by this method minimise

$$\sum_i \frac{(x_i - x_h)^2}{N}$$

where x_i are the sample values.

For this reason this approach is sometimes called the *least mean squares (LMS)* approach.

In general wish to learn utility function (rather than table).

Have examples with:

- input value — state
- output value — observed reward

$\Rightarrow$ *inductive learning problem!*

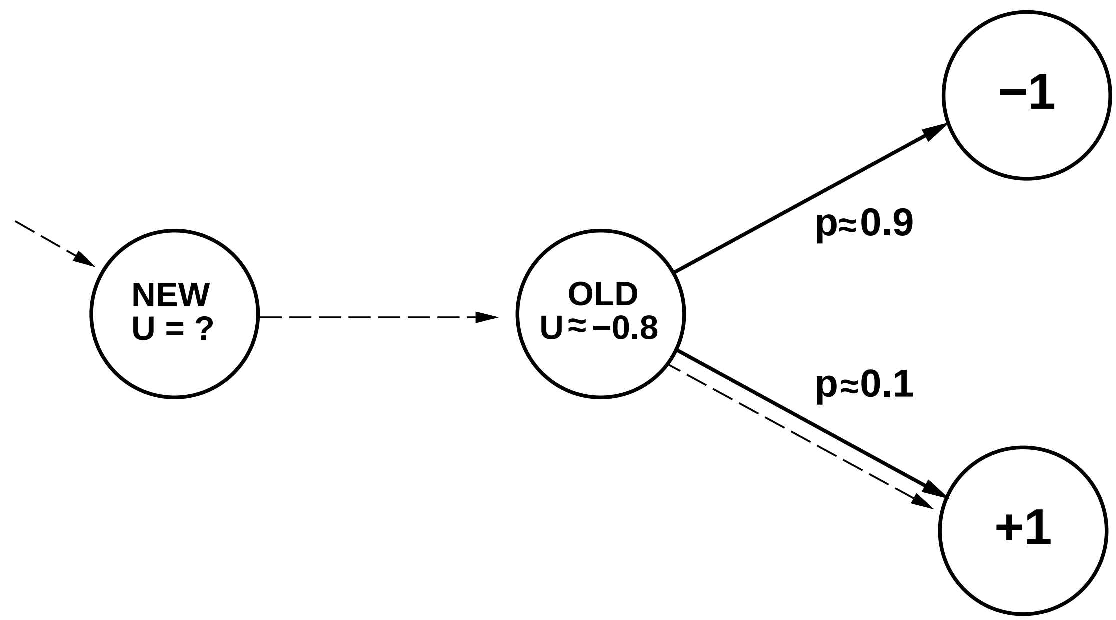
Can apply any techniques for inductive function learning — linear weighted function, neural net, etc. . .

2.2 Naïve Updating — LMS Approach

Problem:

LMS approach ignores important information
    ⇒ interdependence of state utilities!

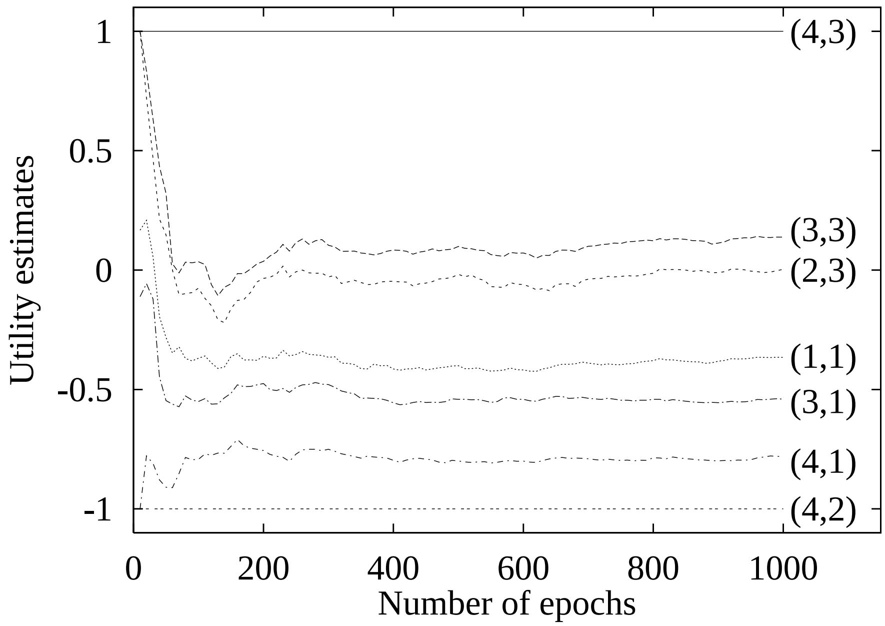
Example (Sutton 1998)

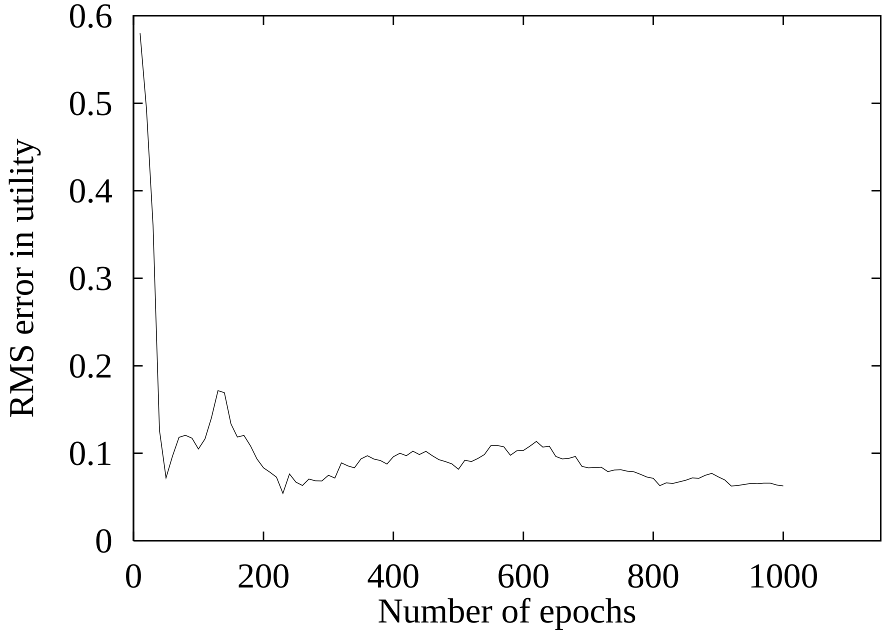


New state awarded estimate of $+1$. Real value ~ -0.8.

2.2 Naïve Updating — LMS Approach

Leads to slow convergence. . .





2.3 Adaptive Dynamic Programming

Take into account relationship between states...

utility of a state = probability weighted average of its successors' utilities + its own reward

Formally, utilities are described by set of equations:

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

(passive version of Bellman equation — no maximisation over actions)

Since transition probabilities M_{ij} known, once enough training sequences have been seen so that all reinforcements $R(i)$ have been observed:

- problem becomes well-defined *sequential decision problem*
- equivalent to *value determination* phase of policy iteration

$\Rightarrow$ above equation can be solved exactly

2.3 Adaptive Dynamic Programming

3	−0.0380	0.0886	0.2152	$\boxed{+1}$
2	−0.1646		−0.4430	$\boxed{-1}$
1	−0.2911	−0.0380	−0.5443	−0.7722
	1	2	3	4

Refer to learning methods that solve utility equations using dynamic programming as *adaptive dynamic programming (ADP)*.

Good benchmark, but intractable for large state spaces

eg. backgammon: 10^{50} equations in 10^{50} unknowns

2.4 Temporal Difference Learning

Can we get the best of both worlds — use contraints without solving equations for all states?

$\Rightarrow$ use observed transitions to adjust locally in line with constraints

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

α is *learning rate*

Called *temporal difference (TD) equation* — updates according to *difference* in utilities between *successive* states.

Note: compared with

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

— only involves observed successor rather than all successors

However, average value of $U(i)$ converges to correct value.

Step further — replace α with function that decreases with number of observations

$\Rightarrow$ $U(i)$ *converges to correct value* (Dayan, 1992).

Algorithm $\cdots \rightsquigarrow$

2.4 Temporal Difference Learning

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

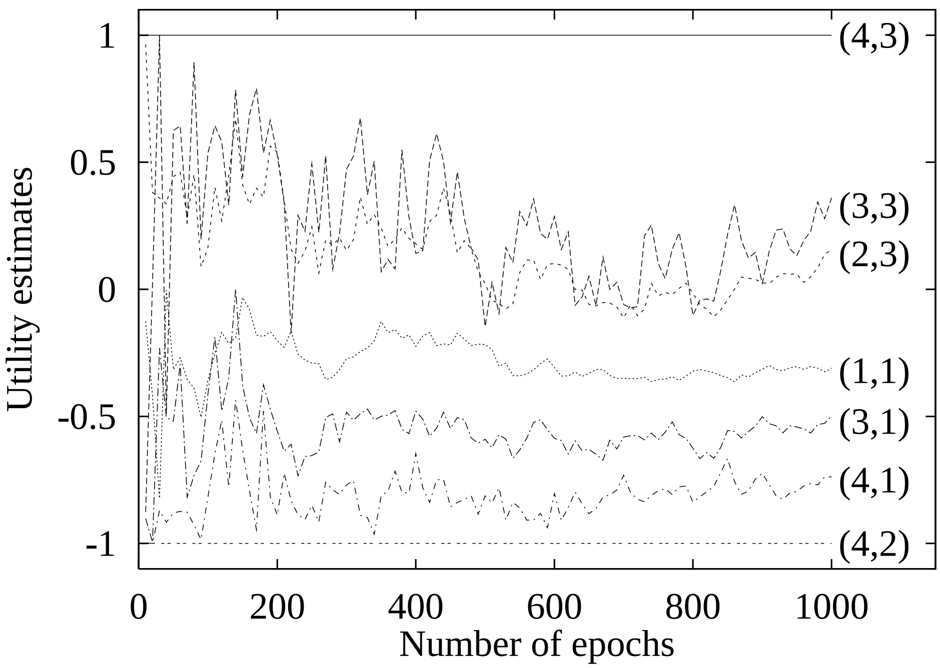
if TERMINAL?[e] **then**
U[STATE[e]] $\leftarrow$ RUNNING-AVERAGE(U[STATE[e]], REWARD[e], N[STATE[e]])
else if *percepts* contains more than one element **then**
$e' \leftarrow$ the penultimate element of *percepts*
$i,\ j \leftarrow$ STATE[e'], STATE[e]
$U[i] \leftarrow U[i] + \alpha(N[i])(\text{REWARD}[e'] + U[j] - U[i])$

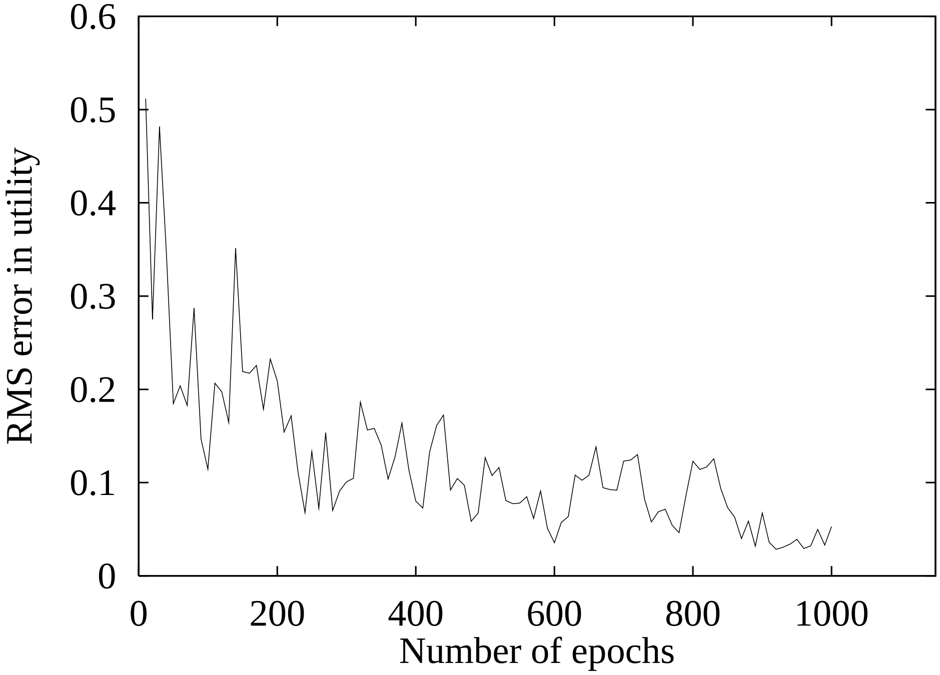
Example runs $\cdots \rightsquigarrow$

Notice:

- values more eratic
- RMS error significantly lower than LMS approach after 1000 epochs

2.4 Temporal Difference Learning





3. Passive Learning, Unknown Environments

- LMS and TD learning don't use model directly
  $\Rightarrow$ operate unchanged in unknown environment
- ADP requires estimate of model
- All utility-based methods use model for action selection

Estimate of model can be updated during learning by observation of transitions

- each percept provides input/output example of transition function

eg. for tabular representation of M, simply keep track of percentage of transitions to each neighbour

Other techniques for learning stochastic functions — not covered here.

4. Active Learning in Unknown Environments

Agent must decide which actions to take.

Changes:

- agent must include *performance element* (and *exploration element*) $\Rightarrow$ choose action
- model must incorporate probabilities *given action* — M_{ij}^a
- constraints on utilities must take account of choice of action

  $$U(i) = R(i) + \max_a \sum_j M_{ij}^a U(j)$$

  (Bellman's equation from sequential decision problems)

Model Learning and ADP

- Tabular representation — accumulate statistics in 3 dimensional table (rather than 2 dimensional)
- Functional representation — input to function includes action taken

ADP can then use value iteration (or policy iteration) algorithms

$\ldots \leadsto$

4. Active Learning in Unknown Environments

```
function Active-ADP-Agent(e) returns an action
    static: U, a table of utility estimates
            M, a table of transition probabilities from state to state
                for each action
            R, a table of rewards for states
            percepts, a percept sequence (initially empty)
            last-action, the action just executed

    add e to percepts
    R[State[e]] ← Reward[e]
    M ← Update-Active-Model(M, percepts, last-action)
    U ← Value-Iteration(U, M, R)
    if Terminal?[e] then percepts ← the empty sequence
    last-action ← Performance-Element(e)
    return last-action
```

Temporal Difference Learning

Learn model as per ADP.

Update algorithm...?

No change! Strange rewards only occur in proportion to probability of strange action outcomes

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

5. Exploration

How should performance element choose actions?

Two outcomes:

- gain rewards on current sequence
- observe new percepts for learning, and improve rewards on future sequences

trade-off between immediate and long-term good

— not limited to automated agents!

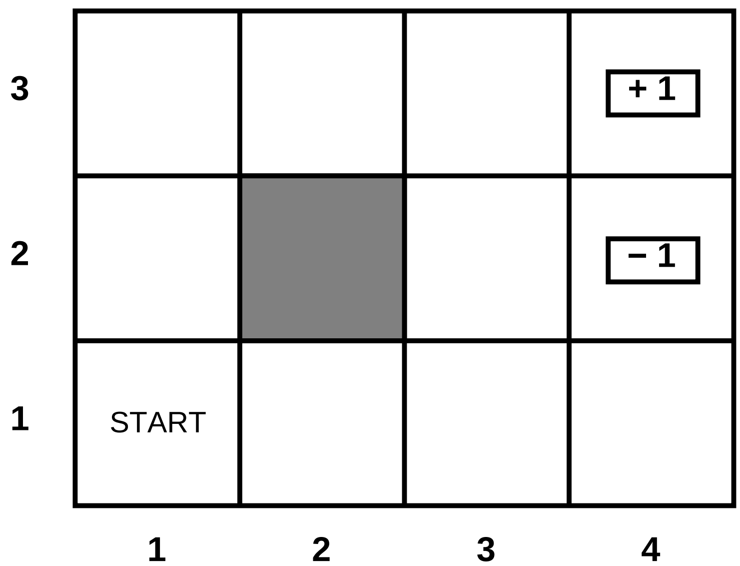
Non trivial

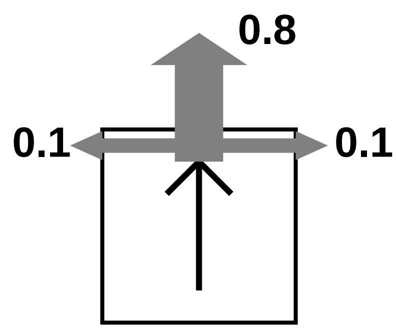
- too conservative $\Rightarrow$ get stuck in a rut
- too inquisitive $\Rightarrow$ inefficient, never get anything done

eg. taxi driver agent

5. Exploration

Example





Two extremes:

whacky — acts randomly in hope of exploring environment

⇒ learns good utility estimates

⇒ never gets better at reaching positive reward

greedy — acts to maximise utility given current estimates

⇒ finds a path to positive reward

⇒ never finds optimal route

Start whacky, get greedier?

Is there an optimal exploration policy?

5. Exploration

Optimal is difficult, but can get close. . .

— give weight to actions that have not been tried often, while tending to avoid low utilities

Alter constraint equation to assign higher utility estimates to relatively unexplored action-state pairs

$\Rightarrow$ optimistic "prior" — initially assume everything is good.

Let

$U^+(i)$ — optimistic estimate
$N(a,i)$ — number of times action a tried in state i

ADP update equation

$$U^+(i) \leftarrow R(i) + \max_a f(\sum_j M^a_{ij} U^+(j), N(a,i))$$

where $f(u,n)$ is *exploration function*.

Note U^+ (not U) on r.h.s. — propagates tendency to explore from sparsely explored regions through densely explored regions

5. Exploration

$f(u, n)$ determines *trade-off* between "greed" and "curiosity"

$\Rightarrow$ should increase with u, decrease with n

Simple example

$$f(u, n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

where R^+ is optimistic estimate of best possible reward, N_e is fixed parameter

$\Rightarrow$ try each state at least N_e times.

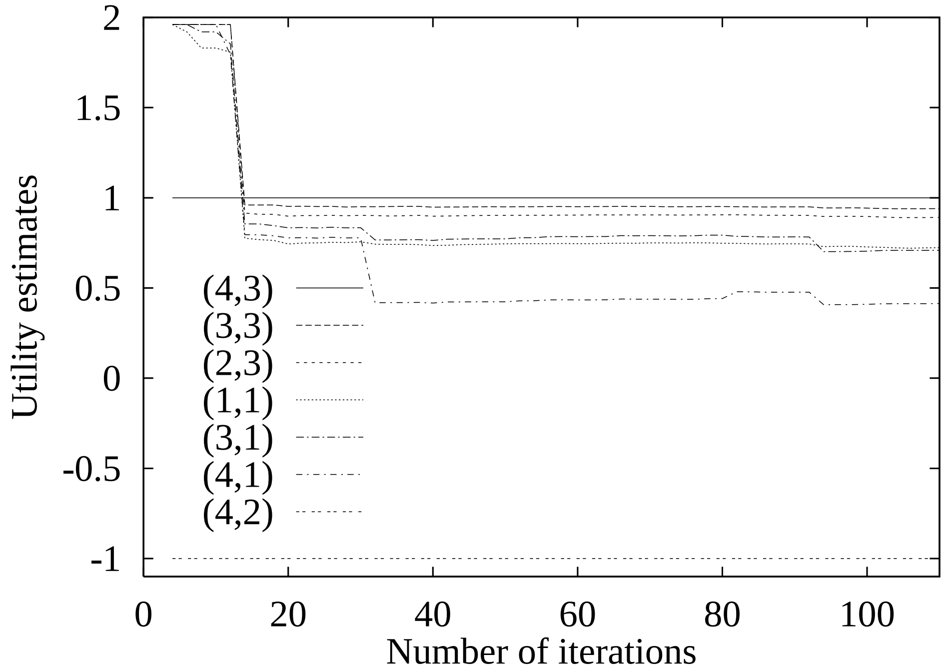
Example for ADP agent with $R^+ = 2$ and $N_e = 5 \cdots \leadsto$

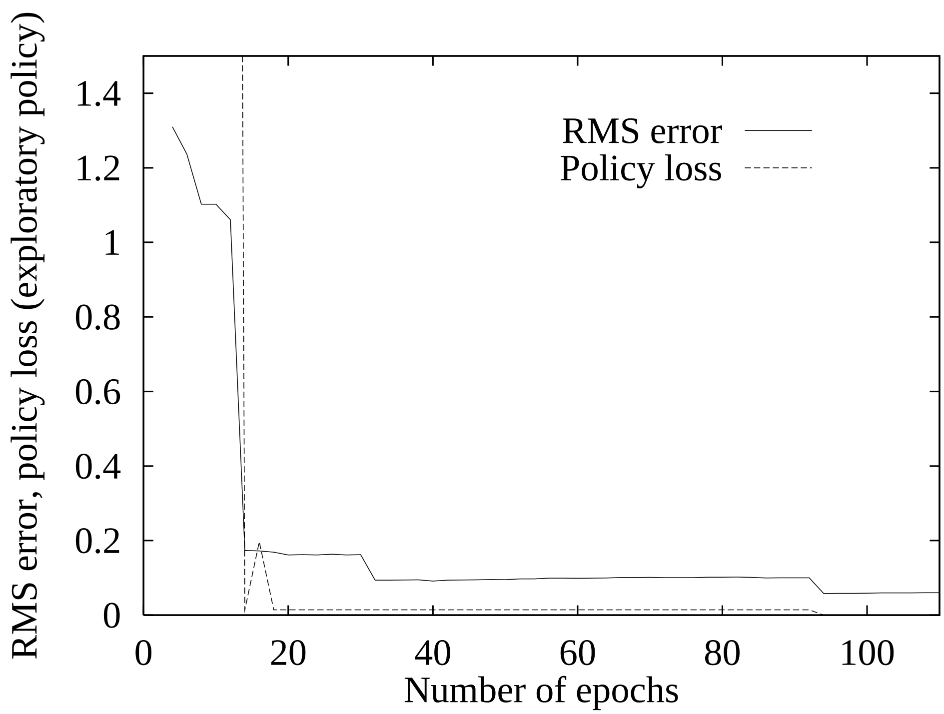
Note policy converges on optimal very quickly

(wacky — best policy loss $\approx$ 2.3
greedy — best policy loss $\approx$ 0.25)

Utility estimates take longer — after exploratory period further exploration only by "chance"

5. Exploration





6. Learning Action-Value Functions

Action-value functions

- assign expected utility to taking action a in state i
- also called *Q-values*
- allow decision-making without use of model

Relationship to utility values

$$U(i) = \max_a Q(a, i)$$

Constraint equation

$$Q(a, i) = R(i) + \sum_j M_{ij}^a \max_{a'} Q(a', j)$$

Can be used for iterative learning, but need to learn model.

Alternative $\Rightarrow$ temporal difference learning

TD Q-learning update equation

$$Q(a, i) \leftarrow Q(a, i) + \alpha(R(i) + \max_{a'} Q(a', j) - Q(a, i))$$

6. Learning Action-Value Functions

Algorithm:

function Q-LEARNING-AGENT(e) **returns** an action
  static: Q, a table of action values
    N, a table of state-action frequencies
    a, the last action taken
    i, the previous state visited
    r, the reward received in state i

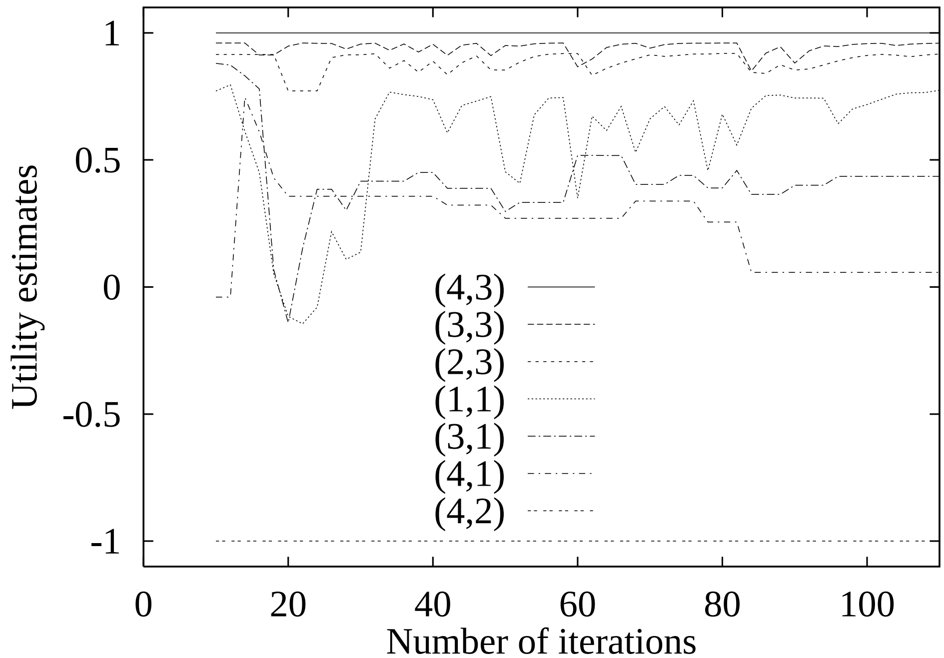
  j ← STATE[e]
  if i is non-null **then**
    $N[a, i] \leftarrow N[a, i] + 1$
    $Q[a, i] \leftarrow Q[a, i] + \alpha(r + \max_{a'} Q[a', j] - Q[a, i])$
  if TERMINAL?[e] **then**
    $i \leftarrow$ null
  else
    $i \leftarrow j$
    $r \leftarrow$ REWARD[e]
  $a \leftarrow \arg\max_{a'} f(Q[a', j], N[a', j])$
  return a

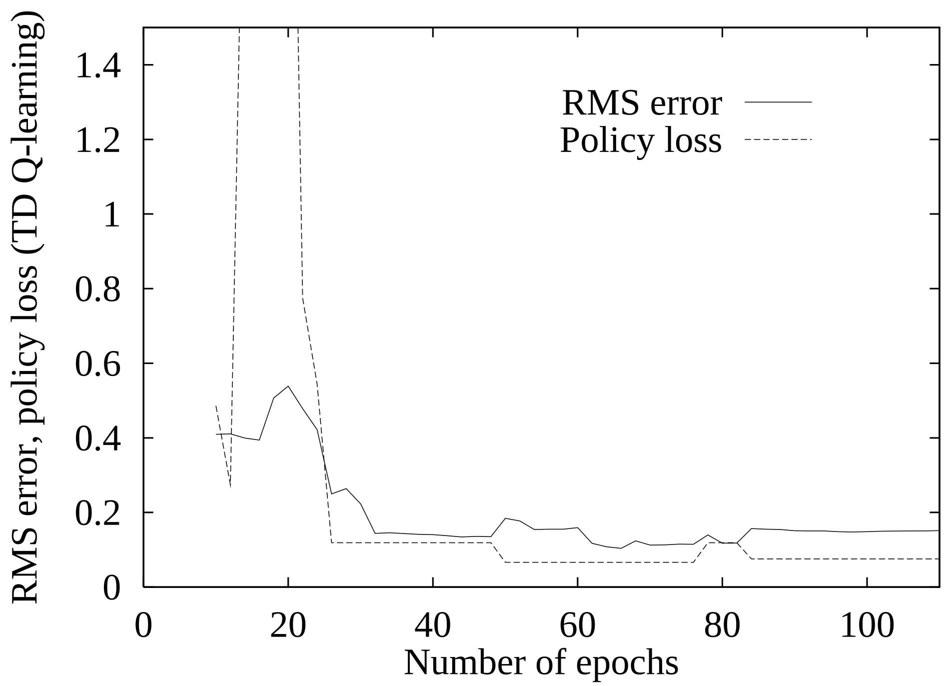
Example $\cdots \rightsquigarrow$

Note: slower convergence, greater policy loss

Consistency between values not enforced by model.

6. Learning Action-Value Functions





7. Generalisation

So far, algorithms have represented hypothesis functions as tables — *explicit representation*

eg. state/utility pairs

OK for small problems, impractical for most real-world problems.

eg. chess and backgammon $\rightarrow 10^{50} - 10^{120}$ states.

Problem is not just storage — *do we have to visit all states to learn?*

Clearly humans don't!

Require *implicit representation* — compact representation, rather than storing value, allows value to be calculated

eg. weighted linear sum of features

$$U(i) = w_1 f_1(i) + w_2 f_2(i) + \cdots + w_n f_n(i)$$

From say 10^{120} states to 10 weights $\Rightarrow$ *whopping compression!*

But more importantly, returns estimates for unseen states

$\Rightarrow$ *generalisation!!*

7. Generalisation

Very powerful. eg. from examining 1 in 10^{44} backgammon states, can learn a utility function that can play as well as any human.

On the other hand, may fail completely...

> *hypothesis space must contain a function close enough to actual utility function*

Depends on

- type of function used for hypothesis
  eg. linear, nonlinear (neural net), etc
- chosen features

Trade off:

larger the hypothesis space

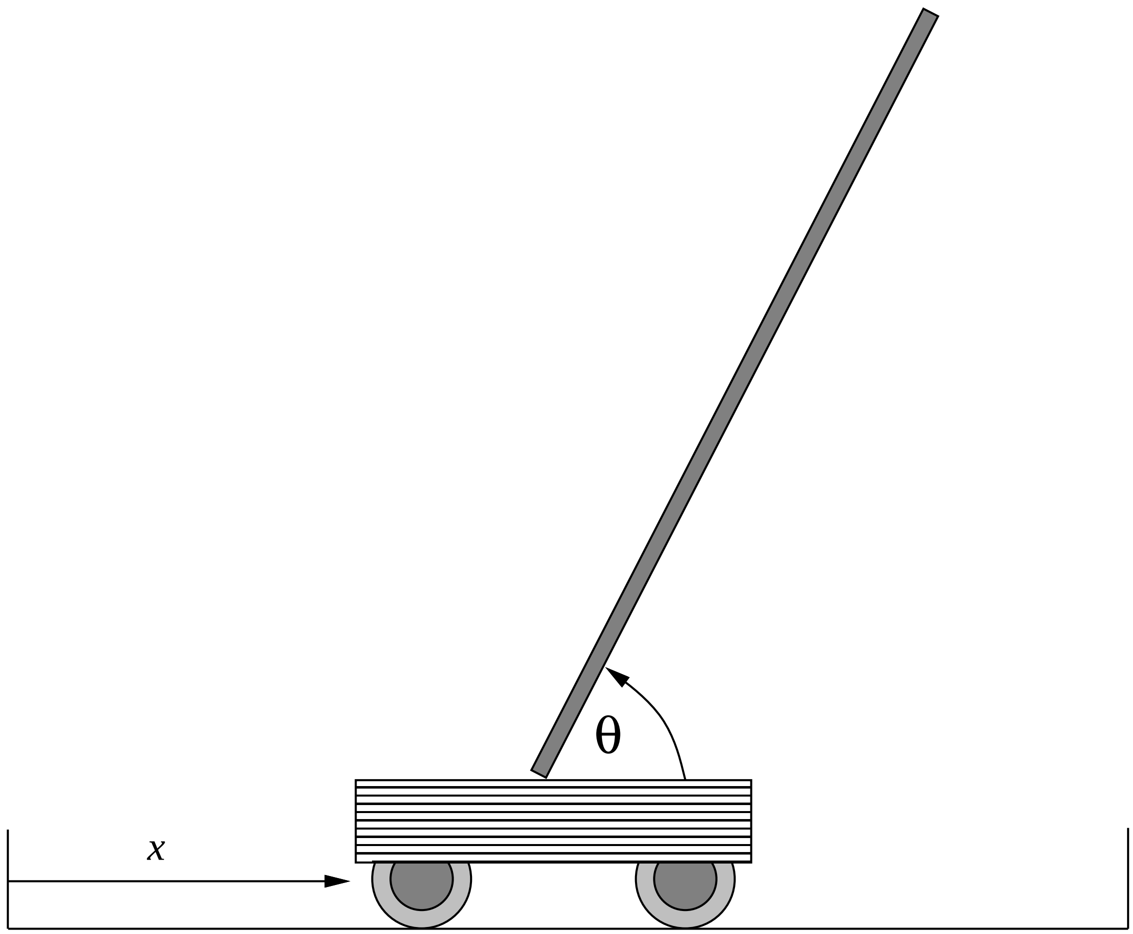
$\Rightarrow$ *better likelihood it includes suitable function*, but

$\Rightarrow$ *more examples needed*

$\Rightarrow$ *slower convergence*

7. Generalisation

And last but not least. . .



The End