Reinforcement Learning

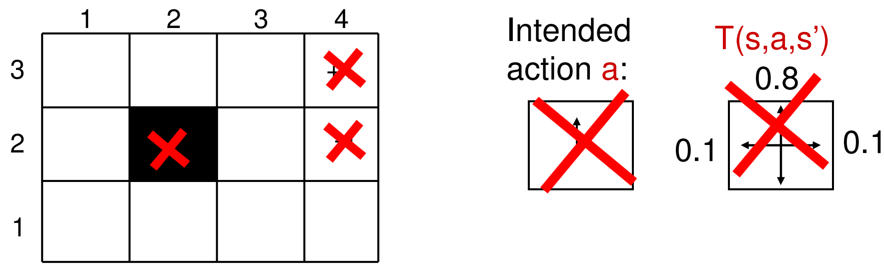
Reinforcement Learning

- R&N Chapter 21

- Demos and Data Contributions from

Vivek Mehta (vivekm@cs.cmu.edu)
Rohit Kelkar (ryk@cs.cmu.edu)

Reinforcement Learning



- Same (fully observable) MDP as before except:
  - We don't know the model of the environment
  - We don't know T(.,.,.)
  - We don't know R(.)
- Task is still the same:
  - Find an optimal policy

General Problem

- All we can do is try to execute actions and record the resulting rewards
  - World: You are in state 102, you have a choice of 4 actions
  - Robot: I'll take action 2
  - World: You get a reward of 1 and you are now in state 63, you have a choice of 3 actions
  - Robot: I'll take action 3
  - World: You get a reward of -10 and you are now in state 12, you have a choice of 4 actions
  - …………..

Learning from experience ….

Classes of Techniques

Reinforcement Learning

Model-Based

- Try to learn an explicit model of T(.,.,.) and R(.)

Model-Free

- Recover an optimal policy without ever estimating a model

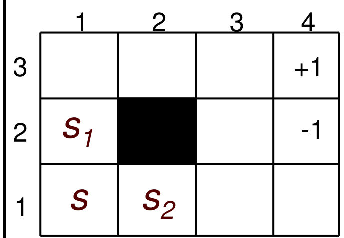
Model-Based

- If we knew a good estimate $T^{est}(.,.,.)$ of T(.,.,.) and R(.), we could evaluate the optimal policy by solving the fundamental MDP relations:

$$U^{est}(s) = R(s) + \gamma \max_a \left(\sum_{s'} T^{est}(s,a,s')\, U^{est}(s')\right)$$

$$\pi^*(s) = \operatorname{argmax}_a \left(\sum_{s'} T^{est}(s,a,s')\, U^{est}(s')\right)$$

Model Estimation



I observed a trajectory during which, when I moved Up from s = (1,1), I ended up in

s_1 = (1,2) 10 times

s_2 = (2,1) 2 times

T(s, Up, s_1) ~ 10/(10+2) = 0.83

T(s, Up, s_2) ~ 2/(10+2) = 0.17

Model-Based

- Move through the environment by executing a sequence of actions
- Evaluate T and R:
  - R(s) = Reward received when visiting state s
  - $T^{est}(s, a, s')$ ~ (# times we moved from s to s' on action a)/(# times we applied action a from s)
- This gives us an estimated model of the Markov system

Model-Based

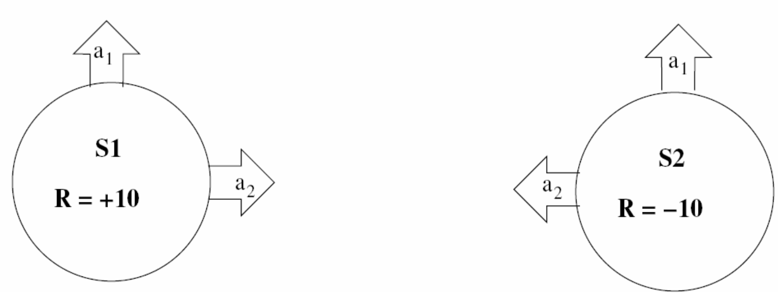
- Given T^{est} and R, we can now estimate the value at each state:

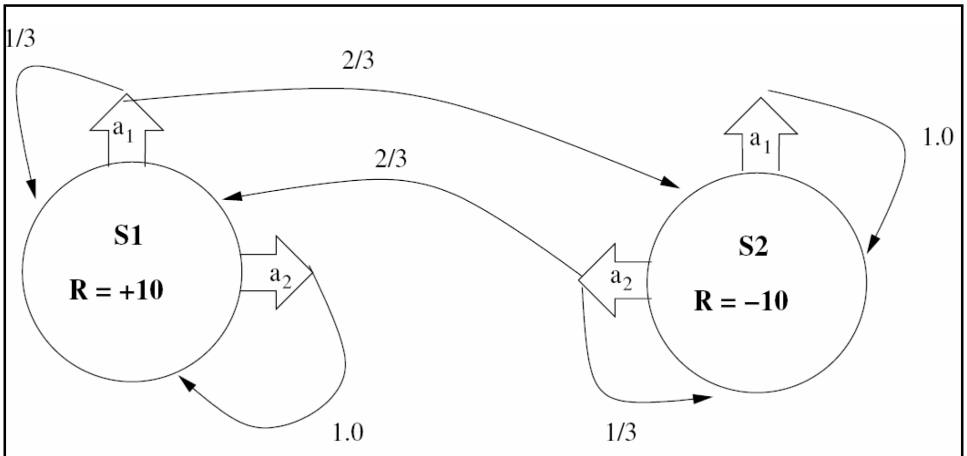
$$U^{est}(s)=R(s)+\gamma \max_a\left(\sum_{s'} T^{est}(s,a,s')\, U^{est}(s')\right)$$

  - Value iteration
  - Policy iteration
- This can be expensive if we do that at each step
- May require matrix inversion (size = number of states) or
- Many iterations of value iteration
- ***(Certainty Equivalent learning)***

Best Policy?

$(\text{Start} = S_1, \text{Action} = a_1, \text{Reward} = 10, \text{End} = S_2)$
$(\text{Start} = S_2, \text{Action} = a_2, \text{Reward} = \text{-}10, \text{End} = S_1)$
$(\text{Start} = S_1, \text{Action} = a_2, \text{Reward} = 10, \text{End} = S_1)$
$(\text{Start} = S_1, \text{Action} = a_1, \text{Reward} = 10, \text{End} = S_1)$
$(\text{Start} = S_1, \text{Action} = a_2, \text{Reward} = 10, \text{End} = S_1)$
$(\text{Start} = S_1, \text{Action} = a_1, \text{Reward} = 10, \text{End} = S_2)$
$(\text{Start} = S_2, \text{Action} = a_1, \text{Reward} = \text{-}10, \text{End} = S_2)$
$(\text{Start} = S_2, \text{Action} = a_2, \text{Reward} = \text{-}10, \text{End} = S_2)$
$(\text{Start} = S_2, \text{Action} = a_2, \text{Reward} = \text{-}10, \text{End} = S_1)$





$\pi^*(S_1) = a_2$
$\pi^*(S_2) = a_2$

Problems

- Separates learning the model from using the model (not on-line learning)
- Expensive because entire set of equations is solved to find U^{est}
- How should the environment be explored? → No guidance until model is built
- Cannot handle changing environments

Solution

- Update U^{est} for state *s* *only* instead of solving for all the states

$$U^{est}(s) \leftarrow R(s) + \gamma \max_a (\sum_{s'} T^{est}(s,a,s') U^{est}(s'))$$

- Similar to one step of value iteration
- Terminology → Backup step
- Advantage:
  - Computation interleaved with exploration
  - Less computation at each step

Example: Model-Based Learning

- Update the current estimate of $U(s)$ = expected sum of future discounted reward using estimated T(.,.,.)

$$U^{est}(s) \leftarrow R(s) + \gamma \max_a (\sum_{s'} T^{est}(s,a,s') U^{est}(s'))$$

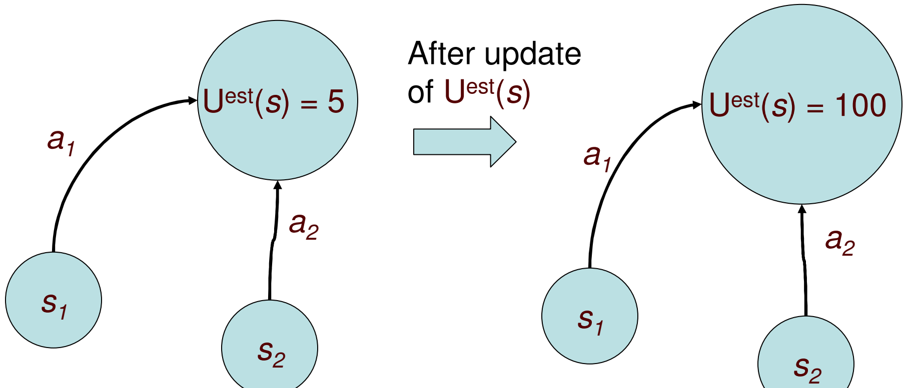
$$\pi(s) = \mathrm{argmax}_a (\sum_{s'} T^{est}(s,a,s') U^{est}(s'))$$

Two Problems

- Which states to update? U^{est} may have already converged for some states, so that the update does not make any difference
- How to explore the environment? We have not said how we generate the actions *a*

Which State to Update: Prioritized Sweeping

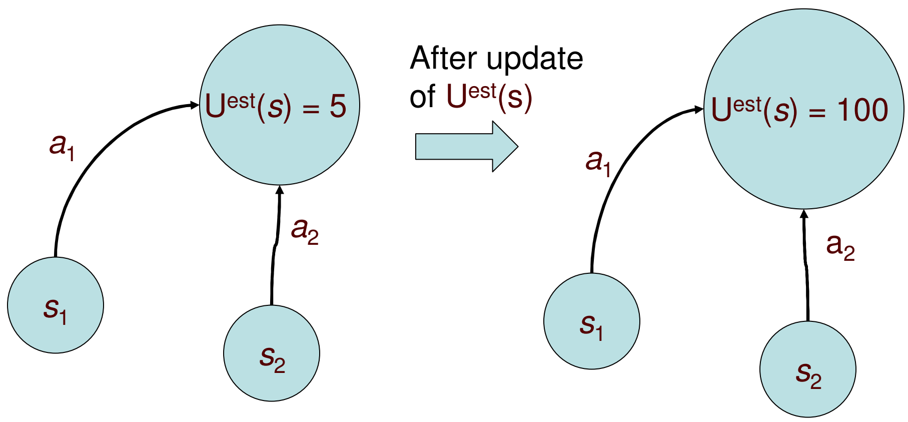
- Idea: Update the predecessors of the states that yield the largest change in U^{est}



$U^{est}(s)$ has changed a lot and

$U^{est}(s_1) = R(s_1) + \ldots + T(s_1,a_1,s)U^{est}(s)$

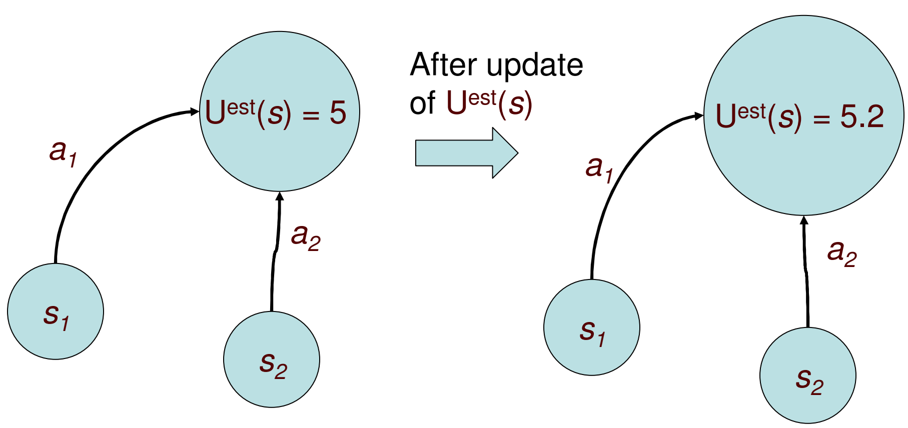
So $U^{est}(s_1)$ is probably going to change a lot too so we should update it right away



$U^{est}(s)$ has not changed a lot and

$U^{est}(s_1) = R(s_1) + \ldots + T(s_1,a_1,s)U^{est}(s)$

So $U^{est}(s_1)$ is probably not going to change a lot so it's not useful to waste time updating it



Prioritized Sweeping

- For each state: Remember Pred(s) = {visited states s' and action a such that a moves from s' to s}
- Store P = priority queue with "most promising" state first

1. s = top of the queue; U^{old} = current value of $U^{est}(s)$
2. Update the state value

$$U^{est}(s) \leftarrow R(s) + \gamma \max_a \left(\sum_{s'} T^{est}(s,a,s')\, U^{est}(s')\right)$$

3. $\Delta = |U^{old} - U^{est}(s)|$
5. For every predecessor (s_p, a_p) in Pred(s)
   - Add s_p to P with priority: $T^{est}(s_p, a_p, s)\Delta$

Prioritized Sweeping

- For each state: Remember Pred(s) = {visited states s' and action a such that a moves from s' to s}
- Store P = priority queue with "most promising" state first

1. s = top of the queue; U^{old} = current value of $U^{est}(s)$
2. Update the state value

$$U^{est}(s) \leftarrow R(s) + \gamma \max_a \left(\sum_{s'} T^{est}(s,a,s')\, U^{est}(s')\right)$$

3. $\Delta = |U^{old} - U^{est}(s)|$
5. For every predecessor (s_p, a_p) in Pred(s)
   - Add s_p to P with priority: $T^{est}(s_p, a_p, s)\Delta$

Δ small = boring state, no change in the value
Δ large = interesting state, new information is discovered

The value for s_p is likely to change if:
1. The value of its successor s has changed a lot (Δ is large) and
2. Some action is likely to move from s_p to s

Exploration Strategy

- In principle, we can compute a current estimate of the best policy:

$$\pi^*(s) = \text{argmax}_a \left(\sum_{s'} T^{est}(s,a,s')\, U^{est}(s')\right)$$

- What is then the best strategy for exploration?
  - Greedy: Always use $\pi^*(s)$ when in state s?
  - Random
  - Mixed: Sometimes use the best and sometimes use random

Why Not Obvious?

- *N*-armed bandit problem:
- We have *N* slot machines, each can yield $1 with some probability (different for each machine)
- In what order should we try the machines?
  - Stay with the machine with highest probability so far?
  - Random?
  - Something else?

Possible Solutions

- ε-greedy
  - Choose the (current) best one with probability 1-ε
  - Choose another one randomly with probability ε/(number of machines – 1)
- Boltzmann exploration
  - Choose machine *k* with Prob ~ $e^{-P_k/T}$
  - Decrease *T* as time passes

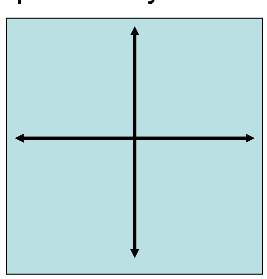
Remember the lectures on randomized search….

Maze Example

Current optimal action for this state is Up

Move Up with probability 1-ε

Move Left with probability ε/3



Move Right with probability ε/3

Move Down with probability ε/3

Model-Free

- We are not interested in T(.,.,.), we are only interested in the resulting values and policies
- Can we compute something without an explicit model of T(.,.,.) ?

- First, let's fix a policy and compute the resulting values

Temporal Differencing

- Upon action $a = \pi(s)$, the values satisfy:

$$U(s) = R(s) + \gamma \sum_{s'} T(s,a,s')\, U(s')$$

For any s' successor of s, U(s) is "in between":

The new value considering only s':

$$R(s) + \gamma\, U(s')$$

and the old value

$$U(s)$$

Temporal Differencing

- Upon moving from s to s' by using action a, the new estimate of U(s) is approximated by:

  $$U(s) = (1-\alpha)\, U(s) + \alpha\, (R(s) + \gamma\, U(s'))$$

- *Temporal Differencing*: When moving from any state s to a state s', update:

  $$U(s) \leftarrow U(s) + \alpha\, (R(s) + \gamma\, U(s') - U(s))$$

Temporal Differencing

Current value

Discrepancy between current value and new guess at a value after moving to s'

$$\underbrace{U(s)}_{\text{Current value}} \leftarrow U(s) + \alpha\, \underbrace{(R(s) + \gamma\, U(s') - U(s))}_{\text{Discrepancy}}$$

The transition probabilities do not appear anywhere!!!

Temporal Differencing

Learning rate

$$U(s) \leftarrow U(s) + \alpha\,(R(s) + \gamma\,U(s') - U(s))$$

How to choose $0 < \alpha < 1$?

- Too small: Converges slowly; tends to always trust the current estimate of U
- Too large: Changes very quickly; tends to always replace the current estimate by the new guess

Temporal Differencing

How to choose $0 < \alpha < 1$?

- Start with large α

→ Not confident in our current estimate so we can change it a lot

- Decrease α as we explore more

→ We are more and more confident in our estimate so we don't want to change it a lot

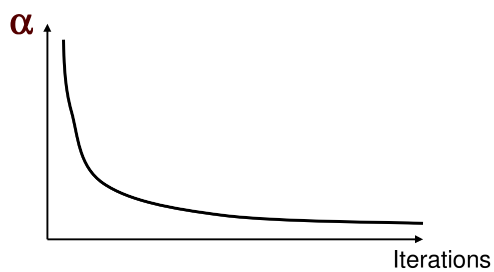
Temporal Differencing

Technical conditions:

$\sum \alpha(t)$ = infinity (α does not decrease too quickly)

$\sum \alpha^2(t)$ converges (but it does decrease fast enough)

Example: $\alpha = K/(K+t)$



Summary

- Learning exploring environment and recording received rewards
- Model-Based techniques
  - Evaluate transition probabilities and apply previous MDP techniques to find values and policies
  - More efficient: Single value update at each state
  - Selection of "interesting" states to update: Prioritized sweeping
- Exploration strategies
- Model-Free Techniques (so far)
- Temporal update to estimate values without ever estimating the transition model
- Parameter: Learning rate must decay over iterations

Temporal Differencing

Current value

Discrepancy between current value and new guess at a value after moving to *s'*

$$U(s) \leftarrow U(s) + \alpha\,(R(s) + \gamma\, U(s') - U(s))$$

The transition probabilities do not appear anywhere!!!

But how to find the optimal policy?

Q-Learning

- $U(s)$ = Utility of state *s* = expected sum of future discounted rewards
- $Q(s,a)$ = Value of taking action *a* at state *s* = expected sum of future discounted rewards after taking action *a* at state *s*

Q-Learning

(s,a) = "state-action" pair. Maintain table of $Q(s,a)$ instead of $U(s)$

- $U(s)$ = Ut… l sum of future di… ewards

- $Q(s,a)$ = Value of taking action a at state s = expected sum of future discounted rewards after taking action a at state s

Q-Learning

- For the optimal Q*:

$$Q^*(s,a) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q^*(s',a')$$

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$$

Q

Best value averaged over all possible states s' that can be reached from s after executing action a

Best expected value for state action (s,a)

• …nal Q*:

$$Q^*(s,a) = R(s) + \gamma \Sigma_{s'} T(s,a,s') \max_{a'} Q^*(s',a')$$

Reward at state s

$(s) = \text{arg}$…

Best value at the next state = Maximum over all actions that could be executed at the next state s'

Q-Learning: Updating Q without a Model

After moving from state s to state s' using action a:

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

Q-Learning: Updating Q without a Model

After moving from state *s* to state *s'* using action *a*:

Old estimate of Q(s,a)

Difference between old estimate and new guess after taking action *a*

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

New estimate of Q(s,a)

Learning rate $0 < \alpha < 1$

Q-Learning: Estimating the policy

Q-Update: After moving from state *s* to state *s'* using action *a:*

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

Policy estimation:

$$\pi(s) = \text{argmax}_a\, Q(s,a)$$

Q-Learning: Estimating the policy

Key Point: We do not use T(.,.,.) anywhere → We can compute optimal values and policies without ever computing a model of the MDP!

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

Policy estimation:

$$\pi(s) = \text{argmax}_a \, Q(s,a)$$

Q-Learning: Convergence

- Q-learning guaranteed to converge to an optimal policy (Watkins)
- Very general procedure (because completely model-free)
- May be slow (because completely model-free)

(Start = S_1, Action = a_1, Reward = 10, End = S_2)

	S_1	S_2
a_1		
a_2		

(Start = S_2, Action = a_2, Reward = -10, End = S_1)

	S_1	S_2
a_1		
a_2		

(Start = S_1, Action = a_2, Reward = 10, End = S_1)

	S_1	S_2
a_1		
a_2		

(Start = S_1, Action = a_1, Reward = 10, End = S_1)

	S_1	S_2
a_1		
a_2		

(Start = S_1, Action = a_1, Reward = 10, End = S_2)

	S_1	S_2
a_1	5.0	0.0
a_2	0.0	0.0

(Start = S_2, Action = a_2, Reward = -10, End = S_1)

	S_1	S_2
a_1	5.0	0.0
a_2	0.0	-3.75

(Start = S_1, Action = a_2, Reward = 10, End = S_1)

	S_1	S_2
a_1	5.0	0.0
a_2	6.25	-3.75

(Start = S_1, Action = a_1, Reward = 10, End = S_1)

	S_1	S_2
a_1	9.0625	0.0
a_2	6.25	-3.75

$\pi^*(S_1) = a_1$

$\pi^*(S_2) = a_1$

Q-Learning: Exploration Strategies

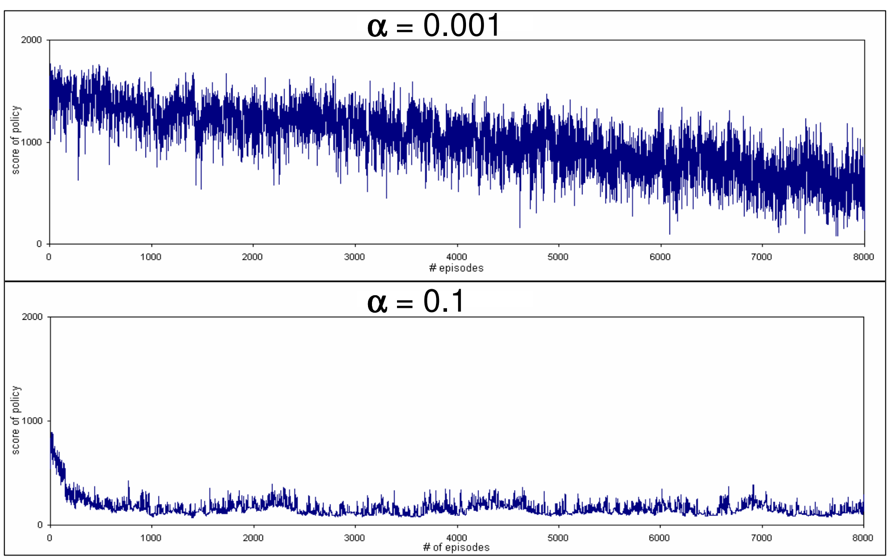
- How to choose the next action while we're learning?
  - Random
  - Greedy: Always choose the estimated best action $\pi(s)$
  - ε-Greedy: Choose the estimated best with probability $1-\varepsilon$
  - Boltzmann: Choose the estimated best with probability proportional to $e^{Q(s,a)/T}$

Evaluation

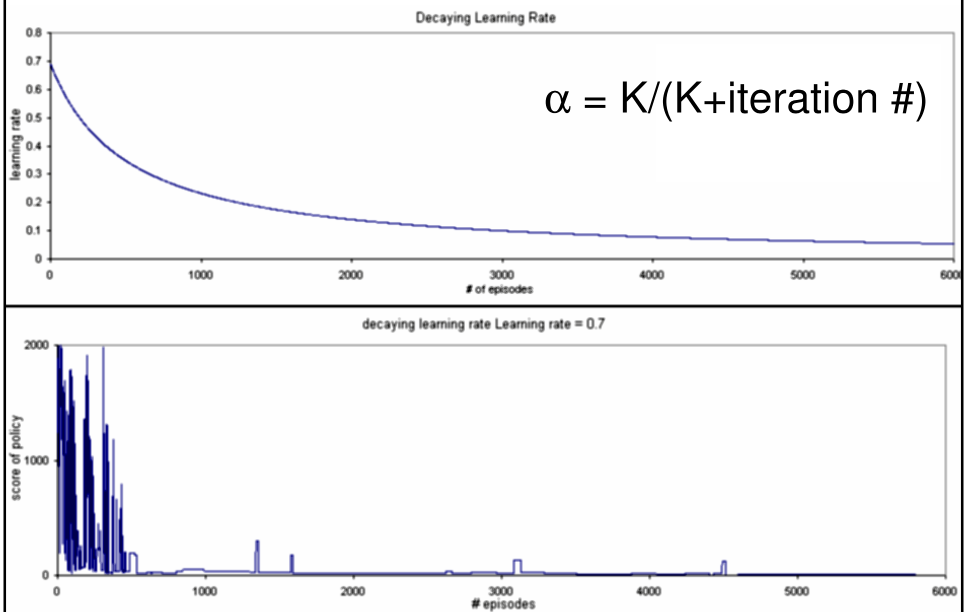
- How to measure how well the learning procedure is doing?
- U(*s*) = Value estimated at *s* at the current learning iteration
- U*(*s*) = Optimal value if we knew everything about the environment

$$\text{Error} = |U - U^*|$$

Constant Learning Rate

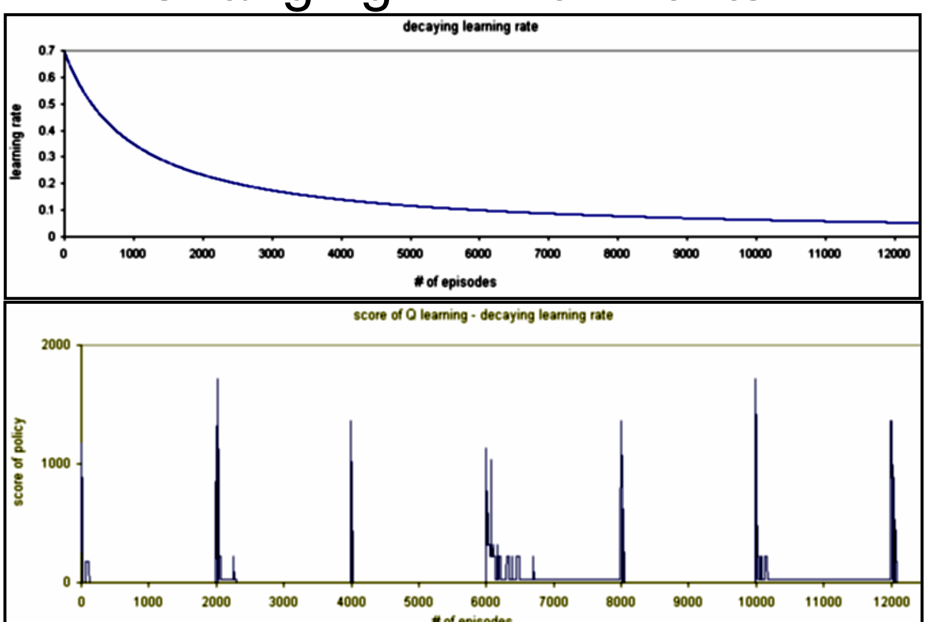


Decaying Learning Rate



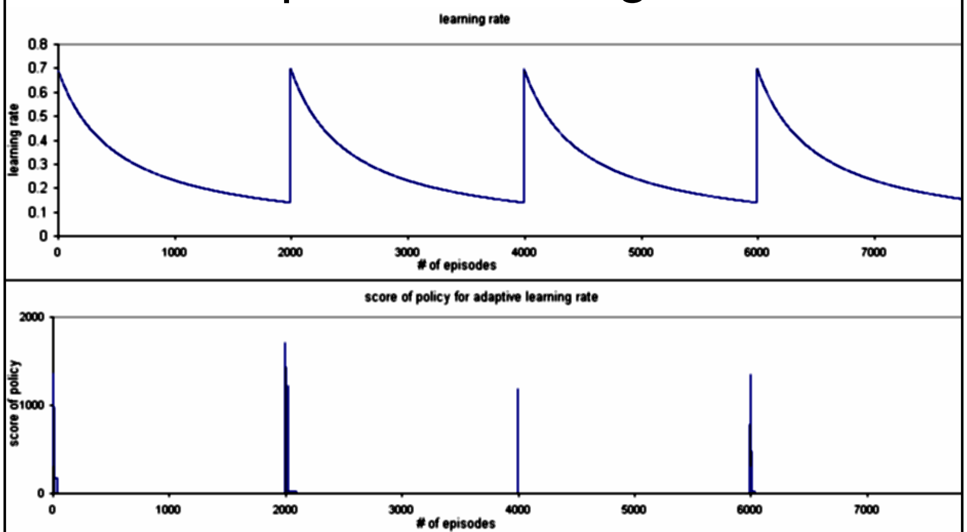
[Data from Rohit & Vivek, 2005]

Changing Environments



[Data from Rohit & Vivek, 2005]

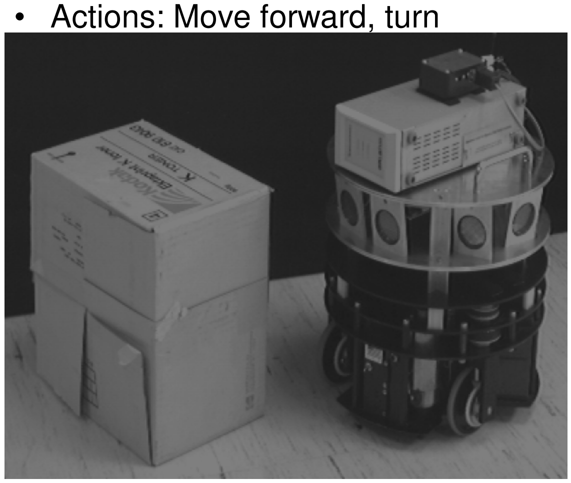
Adaptive Learning Rate



[Data from Rohit & Vivek, 2005]

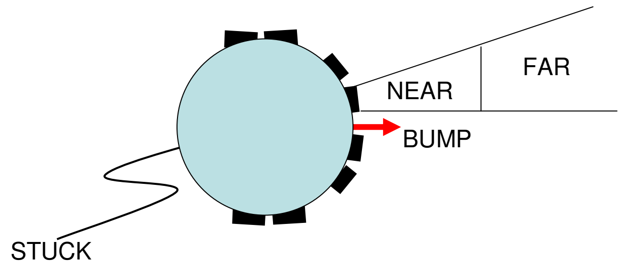
Example: Pushing Robot

- Task: Learn how to push boxes around.
- States: Sensor readings
- Actions: Move forward, turn



Example from Mahadevan and Connell, “Automatic Programming of Behavior-based Robots using Reinforcement Learning, Proceedings AAAI 1991

Example: Pushing Robot



- State = 1 bit for each NEAR and FAR gates x 8 sensors + 1 bit for BUMP + 1 bit for STUCK = 18 bits
- Actions = move forward or turn +/- 22^o or turn +/- 45^o = 5 actions

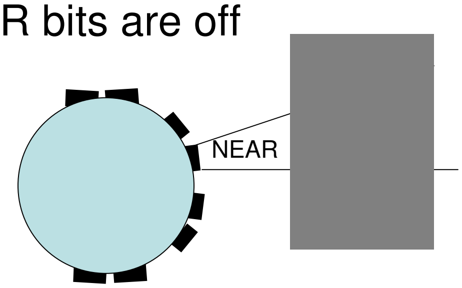
Example from Mahadevan and Connell, “Automatic Programming of Behavior-based Robots using Reinforcement Learning, Proceedings AAAI 1991

Learn How to Find the Boxes

- Box is found when the NEAR bits are on for all the front sonars.
- Reward:

R(*s*) = +3 if NEAR bits are on

R(*s*) = -1 if NEAR bits are off

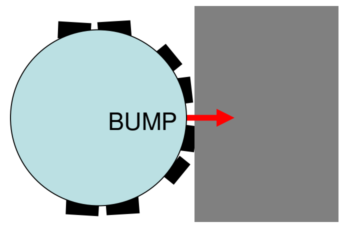


Learn How to Push the Box

- Try to maintain contact with the box while moving forward
- Reward:

R(s) = +1 if BUMP while moving forward

R(s) = -3 if robot loses contact

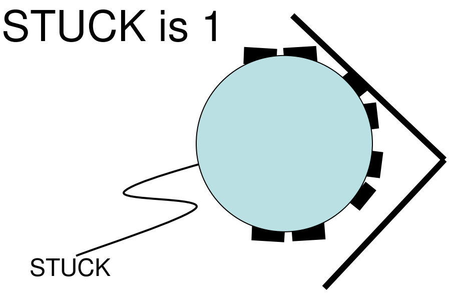


Learn how to Get Unwedged

- Robot may get wedged against walls, in which the STUCK bit is raised.
- Reward:

R(s) = +1 if STUCK is 0

R(s) = -3 if STUCK is 1



Q-Learning

- Initialize Q(*s*,*a*) to 0 for all state-action pairs
- Repeat:
  - Observe the current state *s*
    - 90% of the time, choose the action *a* that maximimizes Q(*s*,*a*)
    - Else choose a random action *a*
  - Update Q(*s*,*a*)

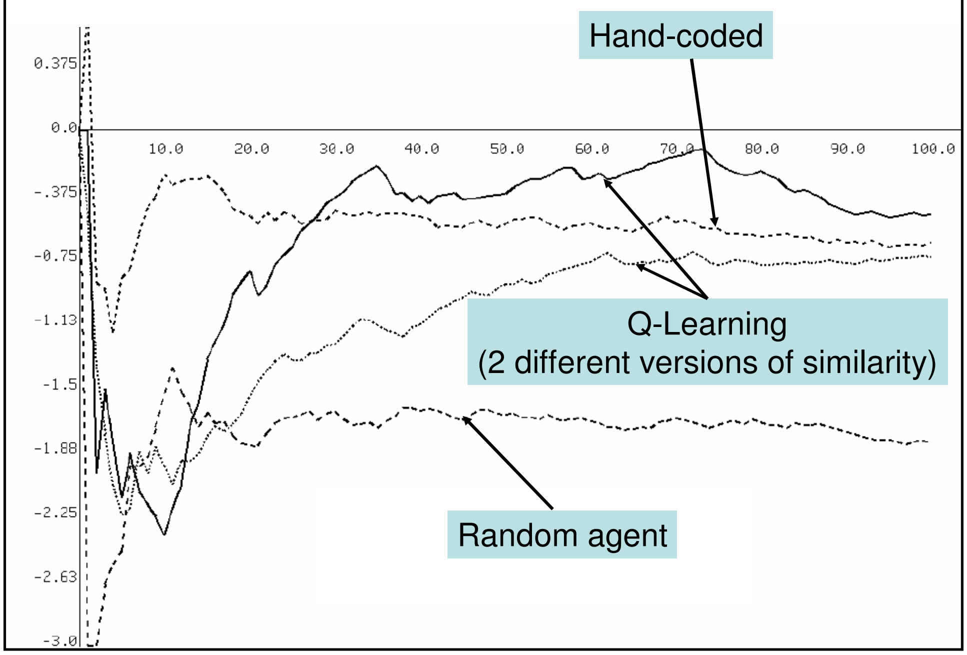
Q-Learning

- Initialize Q(*s*,*a*) to 0 for all state-action pairs
- Repeat:
  - Observe the current state *s*
    - 90% of the time, choose the action *a* that maximimizes Q(*s*,*a*)
    - Else choose a random action *a*
  - Update Q(*s*,*a*)

Improvement:
Update also all the states *s'* that are "similar" to *s*.

In this case: Similarity between *s* and *s'* is measured by the Hamming distance between the bit strings

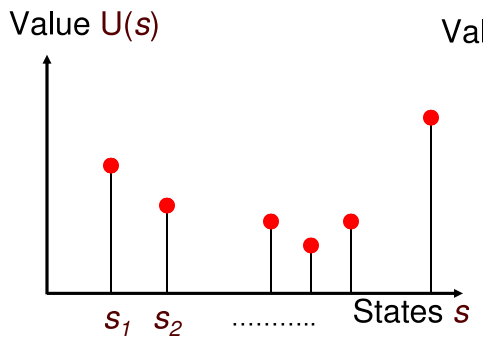
Performance



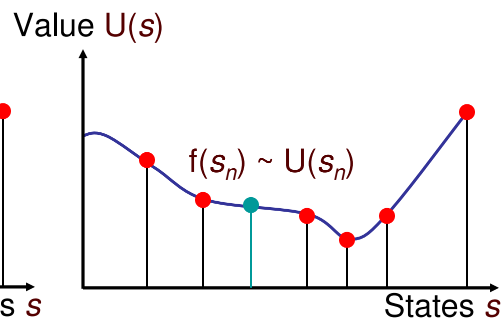
Generalization

- In real problems: Too many states (or state-action pairs) to store in a table
- Example: Backgammon → 10^{20} states!
- Need to:
  - Store U for a subset of states $\{s_1,..,s_K\}$
  - Generalize to compute U(s) for any other states s

Generalization



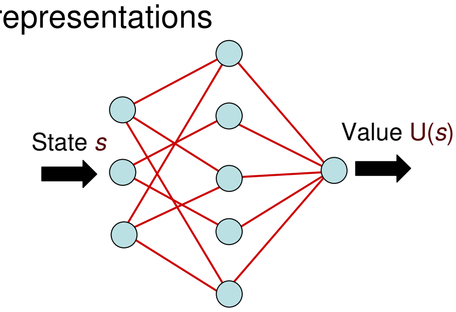
We have sample values of U for some of the states s_1, s_2



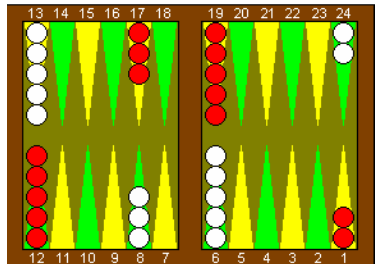
We interpolate a function f(.), such that for any query state s_n, f(s_n) approximates U(s_n)

Generalization

- Possible function approximators:
  - Neural networks
  - Memory-based methods
- …… and many others solutions to representing U over large state spaces:
  - Decision trees
  - Clustering
  - Hierarchical representations



Example: Backgammon



- States: Number of white and black checkers at each location

→ Order 10^{20} states!!!!

→ Branching factor prevents direct search

- Actions: Set of legal moves from any state

Example from: G. Tesauro. Temporal Difference Learning and TD-Gammon. Communications of the ACM, 1995

Example: Backgammon

- Represent mapping from states to expected outcomes by multilayer neural net
- Run a large number of "training games"
  – For each state *s* in a training game:
  – Update using temporal differencing
  – At every step of the game → Choose best move according to current estimate of U
- Initially: Random moves
- After learning: Converges to good selection of moves

Performance

- Can learn starting with no knowledge at all!
- Example: 200,000 training games with 40 hidden units.
- Enhancements use better encoding and additional hand-designed features

- Example:
  - 1,500,000 training games
  - 80 hidden units
  - -1 pt/40 games (against world-class opponent)

Example: Control and Robotics

- Devil-stick juggling (Schaal and Atkeson): Non-linear control at 200ms per decision. Program learns to keep juggling after ~40 trials. A human requires 10 times more practice.
- Helicopter control (Andrew Ng): Control of a helicopter for specific flight patterns. Learning policies from simulator. Learns policies for control pattern that are difficult even for human experts (e.g., inverted flight).

Summary

- Certainty equivalent learning for estimating future rewards
- Exploration strategies
- One-backup update, prioritized sweeping
- Model free (Temporal Differencing = TD) for estimating future rewards
- Q-Learning for model-free estimation of future rewards and optimal policy
- Exploration strategies and selection of actions

(Some) References

- S. Sutton and A.G. Barto. Reinforcement Learning: An Introduction. MIT Press.
- L. Kaelbling, M. Littman and A. Moore. Reinforcement Learning: A Survey. Journal of Artificial Intelligence Research. Volume 4, 1996.
- G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. Neural Computation 6(2), 1995.
- http://ai.stanford.edu/~ang/
- http://www-all.cs.umass.edu/rlr/