Lecture 25: Learning 4

Victor R. Lesser
CMPSCI 683
Fall 2010

Final Exam Information

- Final EXAM on Th 12/16 at 4:00pm in Lederle Grad Res Ctr Rm A301
  - 2 Hours but obviously you can leave early!
- Open Book but no access to Internet
- Material from Lectures 12 -25
  - Lecture 14 will not be covered on exam
  - More operational than conceptual in that I will require you to carry out steps of an algorithm or inference process

Today's Lecture

- Reinforcement Learning

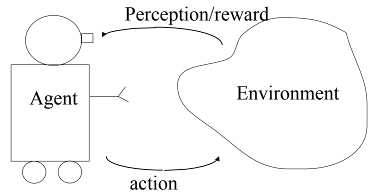
Problem with Supervised Learning

- Supervised learning is sometimes unrealistic: where will correct answers come from?
  - New directions emerging in the use of redundant information as a way of getting around the lack of extensive training data
- *In many cases, the agent will only receive a single reward, after a long sequence of actions/decisions.*
- *Environments change, and so the agent must adjust its action choices.*
  - *On-line issue*

Reinforcement Learning

- Using feedback/rewards to learn a successful agent function.
- Rewards may be provided following each action, or only when the agent reaches a terminal state.
- Rewards can be components of the actual utility function or they can be hints (“nice move”, “bad dog”, etc.).

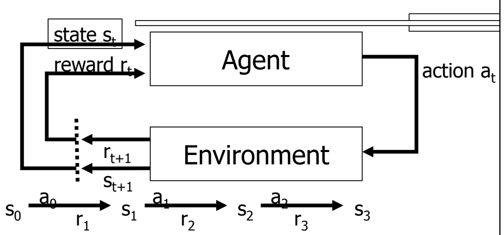
Reinforcement Learning



Utility(reward) depends on a sequence of decisions

How to learn best action (maximize expected reward) to take at each state of Agent

Reinforcement Learning Problem



Agent and environment interact at discrete time steps: $t = 0, 1, 2, K$

Agent observes state at step t: $s_t \in S$

produces action at step t: $a_t \in A(s_t)$

gets resulting reward: $r_{t+1} \in \Re$

and resulting next state: s_{t+1}

RL and Markov Decision Processes

- S - finite set of domain states
- A - finite set of actions
- $P(s'|s,a)$ - state transition function
- $r(s,a)$ - reward function
- S_0 - initial state
- The Markov assumption:

$$P(s_t \mid s_{t-1}, s_{t-2}, \ldots, s_1, a) = P(s_t \mid s_{t-1}, a)$$

RL Learning Task

Execute actions in the environment, observe results and

- Learn a policy $\pi(s) : S \rightarrow A$ from states $s_t \in S$ to actions $a_t \in A$ that maximizes the expected reward : $E[r_t+\gamma\, r_{t+1}+ \gamma^2 r_{t+2}+\ldots]$ from any starting state s_t
- $0<\gamma<1$ is the discount factor for future rewards
- Target function is $\pi(s) : S \rightarrow A$
- But there are no direct training examples of the form <s,a>, i.e., what action is the right one to take in state s
- *Training examples are of the form <<s,a,s’>,r>*

Key Features of Reinforcement Learning

- Learner is not told which actions to take
  - Learning about, from, and while interacting with an external environment
- Trial-and-Error search
- Possibility of delayed reward
  - Sacrifice short-term gains for greater long-term gains
- *The need to explore and exploit*
  - On-line Integrating performance and learning
- Considers the whole problem of a goal-directed agent interacting with an uncertain environment

Reinforcement Learning: Two Approaches

- Learning Model of Markov Decision Process
  - Learn model of operators transitions and their rewards
  - Compute optimal policy (value/policy iteration) based on model
- Learning Optimal Policy Directly
  - You don’t necessarily need to explicit learn MDP model in order to compute optimal policy

Two basic designs

- Utility-based agent learns a *Utility function* on states (or histories) which can be used in order to select actions
  - Must have a model of the environment
  - Know the result of the action (what state the action leads to)
- Q-learning agent learns an *Action-value function* for each state (also called Q-learning; does not require a model of the environment)
  - Does not need a model of the environment, only compare its available choices
  - Can not look ahead because do not know where their actions lead.

Utility function and action-value function

- Utility function denotes the reward for starting in state s and following policy π.

  $U^{\pi}(s) = r_t + \gamma\, r_{t+1} + \gamma^2\, r_{t+2} + \ldots = \Sigma_{i=0}\, \gamma^i\, r_{t+i}$

- Action value function denotes the reward for starting in state s, taking action a and following policy π afterwards.

$Q^{\pi}(s,a) = r(s,a) + \gamma\, r_{t+1} + \gamma^2\, r_{t+2} + \ldots = r(s,a) + \gamma U^{\pi}(\pi(s,a))$

Optimal Value Functions and Policies

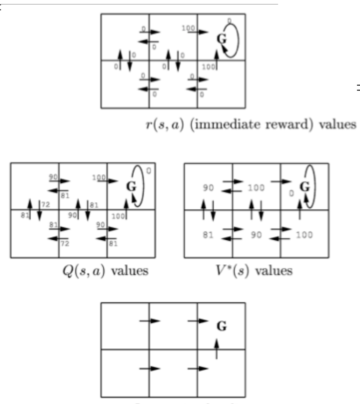
There exist optimal value functions:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \qquad Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

And corresponding optimal policies:

$$\pi^*(s) = \arg\max_{a} Q^*(s,a)$$

π^* is the greedy policy with respect to Q^*



$r(s,a)$ (immediate reward) values

$Q(s,a)$ values

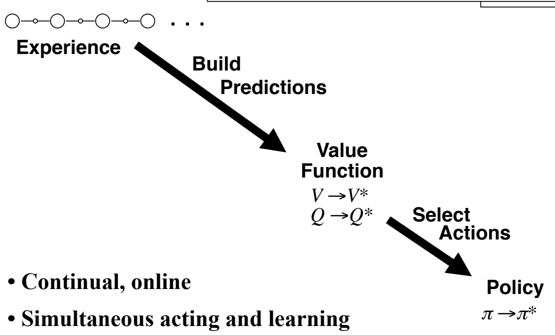
$V^*(s)$ values

One optimal policy

Passive versus Active learning

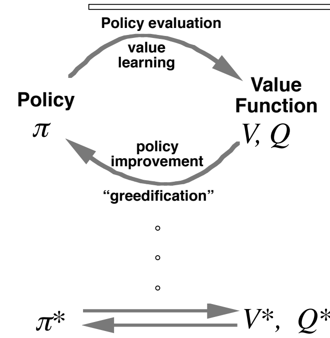
- A *passive learner* simply watches the world going by, and tries to learn the utility of being in various states.
- An *active learner* must also act using the learned information, and can use its problem generator to suggest explorations of unknown portions of the environment.

What Many RL Algorithms Do



- **Continual, online**
- **Simultaneous acting and learning**

RL Interaction of Policy and Value



Passive Learning in Known Environment

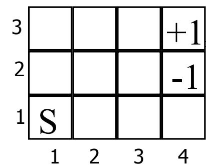
Given:

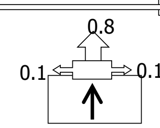
- A Markov model of the environment.
  - $P(s,s',a)$ – probability of transition from s to s' given a
  - $R(s,s',a)$ – expected reward on transition s to s' given a
- States, with probabilistic actions.
- Terminal states have rewards/utilities.

Problem:

- Learn expected utility of each state *V(s) or U(s)*.

Example





- Non-deterministic actions (transition model unknown to agent)
- Every state besides terminal states has reward -0.04
- Percepts tell you: [State, Reward, Terminal?]
- 3 Sequences of (state,action,reward)

$(1,1)_{-.04} \to (1,2)_{-0.4} \to (1,3)_{-0.4} \to (1,2)_{-0.4} \to (1,3)_{-0.4} \to (2,3)_{-0.4} \to (3,3)_{-0.4} \to (4,3)_{+1}$

$(1,1)_{-.04} \to (1,2)_{-0.4} \to (1,3)_{-0.4} \to (2,3)_{-0.4} \to (3,3)_{-0.4} \to (3,2)_{-0.4} \to (3,3)_{-0.4} \to (4,3)_{+1}$

$(1,1)_{-.04} \to (2,1)_{-0.4} \to (3,1)_{-0.4} \to (3,2)_{-0.4} \to (4,2)_{-1}$

Learning Utility Functions

- A training sequence is an instance of world transitions from an initial state to a terminal state.
- The additive utility assumption: utility of a sequence is the sum of the rewards over the states of the sequence.
- *Under this assumption, the utility of a state is the expected reward-to-go of that state.*

Direct Utility Estimation*

- Developed in the late 1950's in the area of adaptive control theory.
- Just keep a running average of rewards for each state.
- *For each training sequence, compute the reward-to-go for each state in the sequence and update the utilities.*

$(1,1)_{-.04} \rightarrow (1,2)_{-0.4} \rightarrow (1,3)_{-0.4} \rightarrow (1,2)_{-0.4} \rightarrow (1,3)_{-0.4} \rightarrow (2,3)_{-0.4} \rightarrow (3,3)_{-0.4} \rightarrow (4,3)_{+1}$

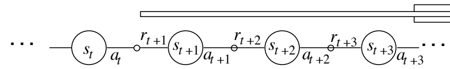
0.72 0.76 0.80 0.84 0.88 0.92 0.96 1.0

$U(1,1) = 0.72; \quad U(2,3) = 0.92; \quad U(3,3) = 0.96; \quad U(4,3) = 1.0;$

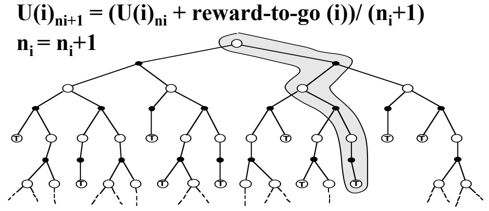
$U(1,2) = (0.76 + 0.84)/2 = 0.80$

$U(1,3) = (0.80 + 0.88)/2 = 0.84$

Direct Utility Estimation, cont



- i = s_t
- ***Reward-to-go (i) = sum of*** $r_{t+1} + r_{t+2} + \ldots\, r_{terminal}$
- **$U(i)_{n_i+1} = (U(i)_{n_i} + \text{reward-to-go }(i)) / (n_i+1)$**
- **$n_i = n_i + 1$**



Problems with Direct Utility Estimation

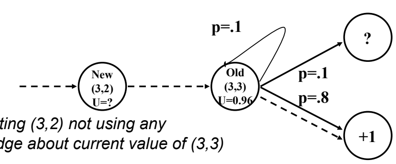
Converges very slowly because it ignores the relationship between neighboring states:

$(1,1)_{-.04} \rightarrow (1,2)_{-0.4} \rightarrow (1,3)_{-0.4} \rightarrow (1,2)_{-0.4} \rightarrow (1,3)_{-0.4} \rightarrow (2,3)_{-0.4} \rightarrow (3,3)_{-0.4} \rightarrow (4,3)_{+1}$

0.72 0.76 0.80 0.84 0.88 0.92 0.96 1.0

$U(1,1) = 0.72; \quad U(2,3) = 0.92; \quad U(3,3) = 0.96; \quad U(4,3) = 1.0;$

$(1,1)_{-.04} \rightarrow (1,2)_{-0.4} \rightarrow (1,3)_{-0.4} \rightarrow (2,3)_{-0.4} \rightarrow (3,3)_{-0.4} \rightarrow (3,2)_{-0.4} \rightarrow (3,3)_{-0.4} \rightarrow (4,3)_{+1}$



In updating (3,2) not using any knowledge about current value of (3,3)

Adaptive Dynamic Programming

Utilities of neighboring states are mutually constrained, Bellman equation:

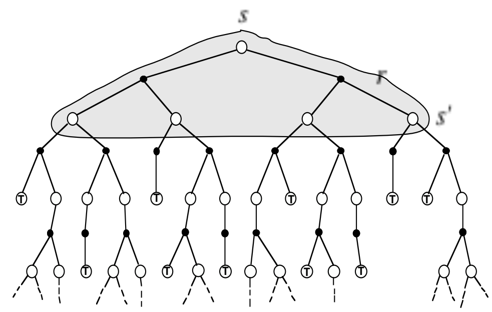
$U(s) = R(s) + \gamma\, \Sigma_{s'} P(s,a,s')\, U(s')$

Estimate P(s,a,s') from the frequency with which s' is reached when executing a in s.

Can use value iteration: initialize utilities based on the rewards and update all values based on the above equation.

Sometime intractable given a big state space.

Adaptive/Stochastic Dynamic Programming



TD: Temporal Difference Learning

- One of the first RL algorithms
- Learn the value of a *fixed* policy (no optimization; just prediction)
- Approximate the constraint equations without solving them for all states.

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s,\pi(s),s')\, U^{\pi}(s')$$

Problem: We don't know this.

- Modify U(s) whenever we see a transition from s to s' using the following rule:

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

Temporal Difference Learning cont.

- $U(s) = U(s) + \alpha\, (\underbrace{R(s) + \gamma\, U(s') - U(s)}_{\text{TD Error}})$
  - The modification moves U(s) closer to satisfying the original equation.
  - α: learning rate, can be a function $\alpha\,(N(s))$ that decreases as N(s) increases [number of times visting state s].
- Rewrite to get

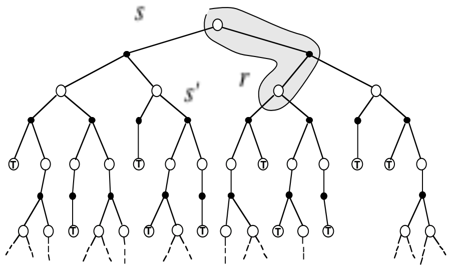
  $U(s) = (1-\alpha)\, U(s) + \alpha\, (R(s) + \gamma\, U(s'))$

Temporal Difference (TD) Learning

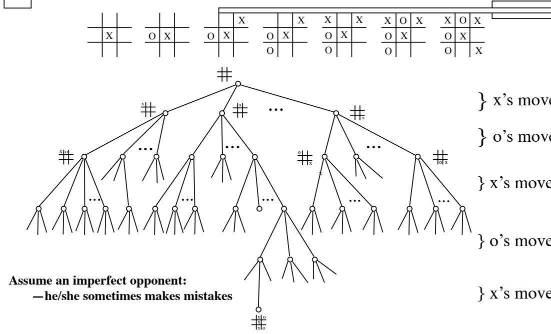
$$V(s) \leftarrow (1-\alpha)V(s) + \alpha\left[r + \gamma V(s')\right]$$

Sutton, 1988

After each action update the state



An Extended Example: Tic-Tac-Toe



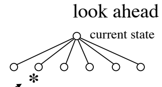
Assume an imperfect opponent:
—he/she sometimes makes mistakes

An RL Approach to Tic-Tac-Toe

1. Make a table with one entry per state:

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

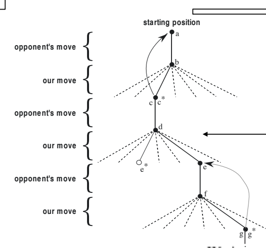
2. Now play lots of games.
To pick our moves,
look ahead one step:



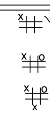
Just pick the next state with the highest estimated prob. of winning — the largest $V(s)$; a ***greedy*** move.

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

RL Learning Rule for Tic-Tac-Toe



"Exploratory" move



Take action O

Non-deterministic outcome based on opponent's 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***:

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

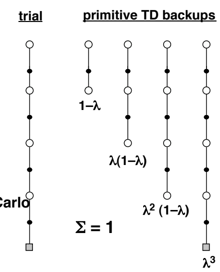
a small positive fraction, e.g., $\alpha = .1$
the ***step - size parameter***

More Complex TD Backups

e.g. TD (λ)

Incrementally computes a weighted mixture of these backups as states are visited

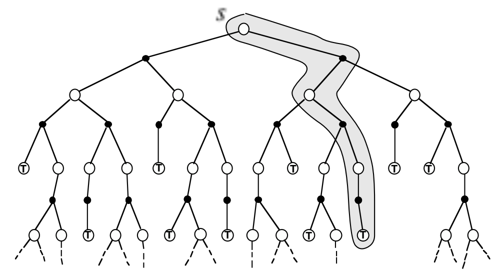
λ = 0 ⟶ Simple TD
λ = 1 ⟶ Simple Monte Carlo



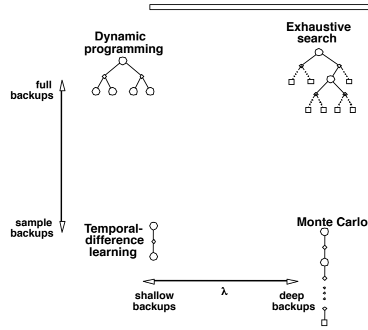
Blending the backups

Simple Monte Carlo

$$V(s) \leftarrow (1-\alpha)V(s) + \alpha\, REWARD(path)$$



Space of Backups



Backup from all terminal states

Limitation of Learning V* Deterministic Case

Choose best action from any state *s using learned* V*

$\pi^*(s)=\arg_a\max\,[r(s, a) + \gamma V^*(\delta(s, a))]$; deterministic case

A problem:

- This works well if agent knows δ: $S \times A \rightarrow S$ and r: $S \times A \rightarrow \Re$
- But when it doesn't, it can't choose actions this way

How Much To do we Need to Know To Learn

Q Learning for Deterministic Case

Define new function very similar to V^*

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

If agent learns Q, it can choose optimal action even without knowing r or δ!

$$\pi^*(s) = \arg_a \max[r(s,a) + \gamma V^*(\delta(s,a))]$$

$$\pi^*(s) = \arg_a \max Q(s,a)$$

Q is the evaluation function agent will learn

Training Rule to Learn Q for Deterministic Operators

Note Q and V^* closely related:

$$V^*(s) = \max_{a'} Q(s,a')$$

Which allows us to write Q recursively as

$$Q(s_t,a_t) = r(s_t,a_t) + \gamma V^*(\delta(s_t,a_t)))$$
$$= r(s_t,a_t) + \gamma \max_{a'} Q(s_{t+1},a')$$

Let $\hat{Q}$ denote learner's current approximation to Q. Consider training rule

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

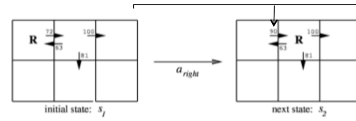
Where s' is the state resulting from applying action a in state s, and a' is the set of actions from s'

Q Learning for Deterministic Worlds

- For each s,a initialize table entry $\hat{Q}(s,a) \leftarrow 0$
- Observe current state s
- Do forever:
  - Select an action a and execute it
  - Receive immediate reward r
  - Observe the new state s'
  - Update the table entry for $\hat{Q}(s,a)$ as follows:

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

$\hat{Q}$ Updating



$$\begin{aligned}\hat{Q}(s_1,a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2,a') \\ &\leftarrow 0 + 0.9 \ \max\{63, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

notice if rewards non-negative, then

$$(\forall s,a,n) \ \ \hat{Q}_{n+1}(s,a) \geq \hat{Q}_n(s,a)$$

and

$$(\forall s,a,n) \ \ 0 \leq \hat{Q}_n(s,a) \leq Q(s,a)$$

Nondeterministic Q learning Case

What if reward and next state are non-deterministic?

We redefine V,Q by taking expected values

$$V^{\pi}(s) \equiv E[r_t + \gamma r_{t+1} + \gamma r_{t+2} + \ldots]$$

$$\equiv E[\sum_{i=0}^{\infty} \gamma^i r_{t+i}]$$

$$Q(s,a) \equiv E[r(s, a) + \gamma V^*(\delta(s,a))]$$

Nondeterministic Case, cont'd

Q learning generalizes to non-deterministic worlds
Alter training rule to

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \max_{a'} \hat{Q}_{n-1}(s', a')]$$

Where $\alpha_n = \frac{1}{1 + visits_n(s,a)}$

Can still prove convergence of $\hat{Q}$ to Q
[Watkins and Dayan, 1992]

Q-learning cont.

- Is it better to learn a model and a utility function, or to learn an action-value function with no model?
- This is a fundamental question in AI where much of the research is based on a knowledge-based approach.
- Some researchers claim that the availability of model free methods such as Q-learning means that the KB approach is unnecessary (or too complex).

What actions to choose?

- Problem: choosing actions with the highest expected utility ignores their contribution to learning.
- *Tradeoff between immediate good and long-term good* (exploration vs. exploitation).
  - A random-walk agent learns faster but never uses that knowledge.
  - A greedy agent learns very slowly and acts based on current, inaccurate knowledge.

What’s the best exploration policy?

- Give some weight to actions that were not tried very often in a given state, but counter that by knowledge that utility may be low.
  - Key idea is that in early stages of learning, estimations can be unrealistic low
- Similar to simulated annealing in that in the early phase of search more willing to explore

Practical issues - large State Set

- Too many states: Can define Q as a weighted sum of state features (factored state), or a neural net. Adjust the previous equations to update weights rather than updating Q.
  - Can have different neural networks for each action
  - This approach used very successfully in TD-Gammon (neural network).
- Continuous state-space: Can discretize it. Pole-balancing example (1968).

Reinforcement Learning Differs From Supervised Learning

- no presentation of input/output pairs
- agent chooses actions, receives reinforcement
- worlds are usually non-deterministic
- on-line performance is important
- system must explore the space of actions

End of Course

GOOD LUCK!!