Lecture 20: Learning -4

Victor Lesser

CMPSCI 683
Fall 2004

Today's Lecture

- Review of Neural Networks
- Markov-Decision Processes
- Reinforcement learning

Back-propagation

- Gradient descent over network weight vector
- Easily generalizes to any directed graph
- Will find a local, not necessarily global error minimum
- Minimizes error over training examples — will it generalize well to subsequent examples?
- Training is slow — can take thousands of iterations.
- Using network after training is very fast

Applicability of Neural Networks

- Instances are represented by many attribute-value pairs
- The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes
- The training examples may contain errors
- Long training times are acceptable
- Fast evaluation of the learned target function may be required
- The ability of humans to understand the learned target function is not important

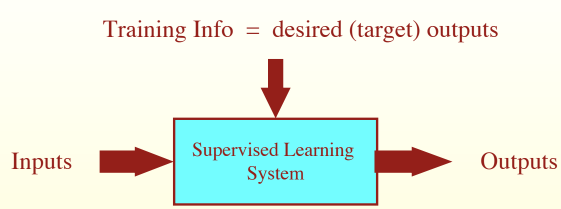
Problem with Supervised Learning

- Supervised learning is sometimes unrealistic: where will correct answers come from?
- In many cases, the agent will only receive a single reward, after a long sequence of actions.
- Environments change, and so the agent must adjust its action choices.
  - On-line issue

Reinforcement Learning

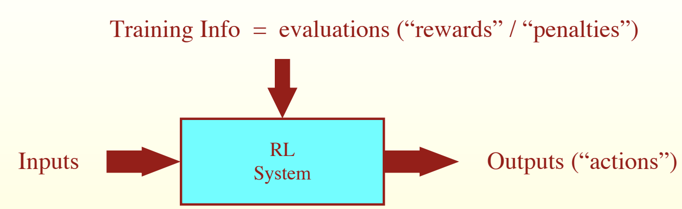
- Using feedback/rewards to learn a successful agent function.
- Rewards may be provided following each action, or only when the agent reaches a terminal state.
- Rewards can be components of the actual utility function or they can be hints (“nice move”, “bad dog”, etc.).

Supervised Learning



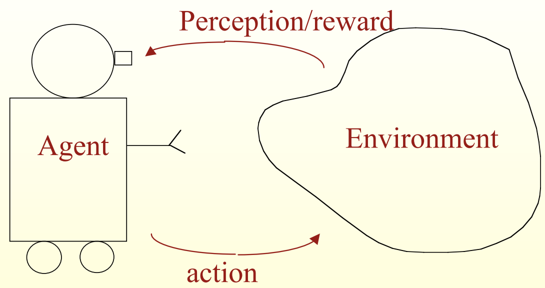
Objective: Minimize Error = (target output – actual output)

Reinforcement Learning



Objective: get as much reward as possible

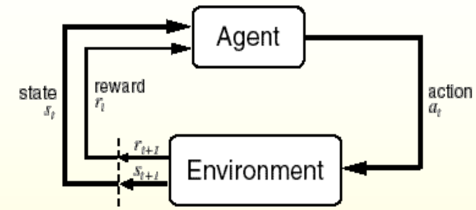
Reinforcement Learning



Utility(reward) depends on a sequence of decisions

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

The Agent-Environment Interface



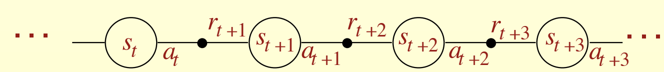
Agent and environment interact at discrete time steps: $t = 0, 1, 2, \ldots$

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}



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:

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

Partially Observable MDPs

Augmenting the completely observable MDP with the following elements:

- O - set of observations
- $P(o|s',a)$ - observation function
- Discrete probability distribution over starting states.

- Can be mapped into MDP
  - **Explodes state space**

Performance Criteria

- Specify how to combine rewards over multiple time steps.
- Finite-horizon and infinite-horizon problems.
- Additive utility = sum of rewards
- Using a discount factor
- Utility as average-reward per time step

Goals and Rewards

- **Is a scalar reward signal an adequate notion of a goal?—maybe not, but it is surprisingly flexible.**
- **A goal should specify** what **we want to achieve, not** how **we want to achieve it.**
- **A goal must be outside the agent's direct control—thus outside the agent.**
- **The agent must be able to measure success:**
  - explicitly;
  - frequently during its lifespan.

Returns/Utility of State/Reward to Go

Suppose the sequence of rewards after step t is:

$$r_{t+1}, r_{t+2}, r_{t+3}, \ldots$$

What do we want to maximize?

In general,

we want to maximize the **expected return**, $E\{R_t\}$, for each step t.

Episodic tasks: interaction breaks naturally into episodes, e.g., plays of a game, trips through a maze.

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T,$$

where T is a final time step at which a **terminal state** is reached, ending an episode.

Returns for Continuing Tasks

Continuing tasks: interaction does not have natural episodes

- Expected Return becomes infinite.

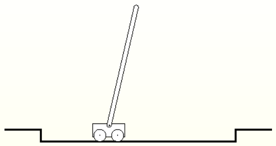
Discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where $\gamma, 0 \le \gamma \le 1$, is the **discount rate**.

shortsighted $0 \leftarrow \gamma \rightarrow 1$ farsighted

An Example



Avoid **failure:** the pole falling beyond a critical angle or the cart hitting end of track.

As an **episodic task** where episode ends upon failure:

reward = +1 for each step before failure

$\Rightarrow$ return = number of steps before failure

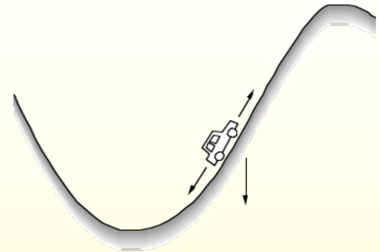
As a **continuing task** with discounted return:

reward = −1 upon failure; 0 otherwise

$\Rightarrow$ return is related to $-\gamma^k$, for k steps before failure

In either case, return is maximized by avoiding failure for as long as possible.

Another Example



Get to the top of the hill as quickly as possible.

reward = −1 for each step where **not** at top of hill

$\Rightarrow$ return = − number of steps before reaching top of hill

Return is maximized by minimizing number of steps reach the top of the hill.

Example: An Optimal Policy

A policy is a choice of what action to choose at each state

An Optimal Policy is a policy where you are always choosing the action that maximizes the "return"/"utility" of the current state

→	→	→	+1
↑		↑	−1
↑	←	←	←

.812	.868	.912	+1
.762		.660	−1
.705	.655	.611	.388

Actions succeed with probability 0.8 and *move at right angles with probability 0.1* (remain in the same position when there is a wall). Actions incur a small cost (0.04).

- What happens when cost increases?
- Why move to .655 instead of .611

Optimal Action Selection Policies

- Optimal policy defined by:

$$policy^*(i) = \arg\max_a \sum_j P(s_j \mid s_i, a) U(j)$$

$$U(i) = R(i) + \max_a \sum_j P(s_j \mid s_i, a) U(j)$$

- Can be solved using dynamic programming [Bellman, 1957]
  - **How to compute U(j) when it's definition is recursive**

Value Iteration [Bellman, 1957]

repeat

$U \leftarrow U'$

for each state i do

$$U'[i] \leftarrow R[i] + \max_a \sum_j P(s_j \mid s_i, a) U(j)$$

end

until $CloseEnough(U, U')$

Value Iteration & Finite-horizon MDPs

We can think of VI as maximizing our utility over some fixed horizon h. We will calculate $V_h(s)$, the maximum value attainable in h steps starting in state s (for all states s). We proceed backwards.

- $V_0(s) = 0$: no utility for taking no steps (or final val).
- $V_1(s) = \max_a R(s, a)$: with only one step, simply choose the action with the highest utility.
- $V_{t+1}(s) = \max_a \left(R(s, a) + \sum_{s'} Pr(s' \mid s, a) V_t(s') \right)$: take the action with the highest utility including the utility of the resulting state.

MDPs and Planning Under Uncertainty SP2-45 of 117 AAAI-99

Value Iteration Example

0.000	0.000	0.000	+1
0.000	0	0.000	-1
0.000	0.000	0.000	0.000

-0.04	-0.0	0.760	+1
-0.04	1	-0.04	-1
-0.04	-0.04	-0.04	-0.04

-0.08	0.560	0.832	+1
-0.08	2	0.464	-1
-0.08	-0.08	-0.08	-0.08

0.392	.738	0.890	+1
-0.12	3	0.572	-1
-0.12	-0.12	0.315	-0.12

0.577	0.819	0.906	+1
0.250	4	0.629	-1
-0.16	0.188	0.394	0.100

0.698	0.849	0.914	+1
0.472	5	0.648	-1
0.162	.313	0.492	0.185

0.809	0.86	0.918	+1
0.754	10	0.660	-1
0.67	0.590	0.577	0.351

0.812	0.868	0.918	+1
0.761	15	0.660	-1
0.704	0.653	0.606	0.378

0.812	0.868	0.918	+1
0.762	19	0.660	-1
0.705	0.655	0.611	0.388

Final Version

Issues with Value Iteration

- **Slow to converge**
- **Convergence occurs out from goal**
- **Information about shortcuts propagates out from goal**
- **Greedy policy is optimal before values completely settle.**
- **Optimal value function is a "fixed point" of VI.**

Prioritized Sweeping

- **State value updates can be performed in any order in value iteration. This suggests trying to decide what states to update to maximize convergence speed.**
- **Prioritized sweeping is a variation of value iteration; more computationally efficient (focused).**
- **Puts all states in a priority queue in order of how much we think their values might change given a step of value iteration.**
- **Very efficient in practice (Moore & Atkeson, 1993).**

Policy Iteration

- **Solve infinite-horizon discounted MDPs in finite time.**
  - Start with value function V_0.
  - Let π_1 be greedy policy for V_0.
  - Evaluate π_1 and *let V_1 be the resulting value function.*
  - Let π_{t+1} be greedy policy for V_t
  - Let V_{t+1} be value of π_{t+1}.
- **Each policy is an improvement until optimal policy is reached (another fixed point).**
- **Since finite set of policies, convergence in finite time.**

Policy Iteration

$$\pi_1 \rightarrow V^{\pi_1} \rightarrow \pi_2 \rightarrow V^{\pi_2} \rightarrow \ldots \ \pi^* \rightarrow V^* \rightarrow \pi^*$$

Policy Evaluation step

"Greedification" step

Improvement is monotonic

<u>Generalized Policy Iteration</u>:
Intermix the two steps at a finer scale: state by state, action by action, etc.

Policy iteration [Howard, 1960]

repeat

$\Pi \leftarrow \Pi'$

$U \leftarrow ValueDetermination(\Pi)$

for each state i do

$$\Pi'[i] \leftarrow \arg\max_a \sum_j P(s_j \mid s_i, a) U(j)$$

end

until $\Pi = \Pi'$

Value determination

Can be implemented using:

Value Iteration:

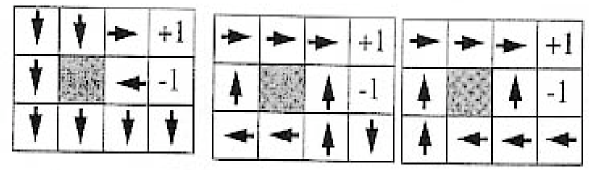
$$U_{t+1} = R(i) + \sum_j P(s_j \mid s_i, \Pi(i)) U_t(j)$$

or

Dynamic Programming:

$$U(i) = R(i) + \sum_j P(s_j \mid s_i, \Pi(i)) U(j)$$

Simulated PI Example



Fewer iterations than VI, but each iteration more expensive.

Source of disagreement among practioners: PI vs. VI.

Reinforcement Learning

- **Learning Model of Markov Decision Process**
- **Learning Optimal Policy**

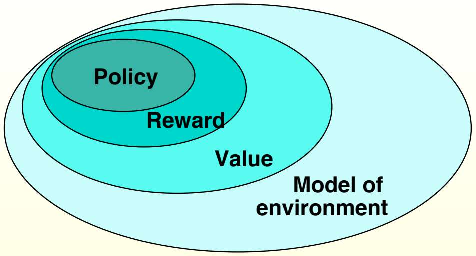
Key Features of Reinforcement Learning

- **Learner is not told which actions to take**
- **Trial-and-Error search**
- **Possibility of delayed reward**
  - Sacrifice short-term gains for greater long-term gains
- **The need to *explore* and *exploit***
- **Considers the whole problem of a goal-directed agent interacting with an uncertain environment**

What is Reinforcement Learning?

- Learning from interaction
- Learning about, from, and while interacting with an external environment
- Learning what to do
  - how to map situations to actions so as to maximize a numerical reward signal
- A collection of methods for approximating the solutions of stochastic optimal control problems

Elements of RL



- Policy: **what to do**
- Reward: **what is good**
- Value: **what is good because it *predicts* reward**
- Model: **what follows what**

Two basic designs

The agent may learn:

- Utility function on states (or histories) which can be used in order to select actions.
  - **Requires a model of the environment.**
- Action-value function for each state (also called Q-learning)
  - **Does not require a model of the environment.**

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.

Passive Learning in A Known Environment

Given:

- A Markov model of the environment.
- States, with probabilistic actions.
- Terminal states have rewards/utilities.

Problem:

- Learn expected utility of each state.

Markov Decision Processes (MDPs)

In RL, the environment is usually modeled as an MDP, defined by

S – set of states of the environment

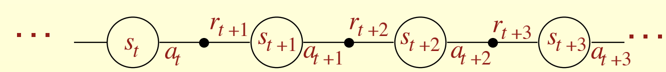
$A(s)$ – set of actions possible in state $s \in S$

$P(s,s',a)$ – probability of transition from s to s' given a

$R(s,s',a)$ – expected reward on transition s to s' given a

γ – discount rate for delayed reward

discrete time, $t = 0, 1, 2, \ldots$



The Objective is to Maximize Long-term Total Discounted Reward

Find a policy $\pi : s \in S \rightarrow a \in A(s)$ **(could be stochastic)**

that maximizes the value/utility (expected future reward) of each s **:**

$$V^{\pi}(s) = E\left\{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s_t = s, \pi \right\}$$

rewards

and each s,a **pair:**

$$Q^{\pi}(s,a) = E\left\{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s_t = s, a_t = a, \pi \right\}$$

These are called value functions (cf. evaluation functions in AI)

Optimal Value Functions and Policies

There exist optimal value functions:

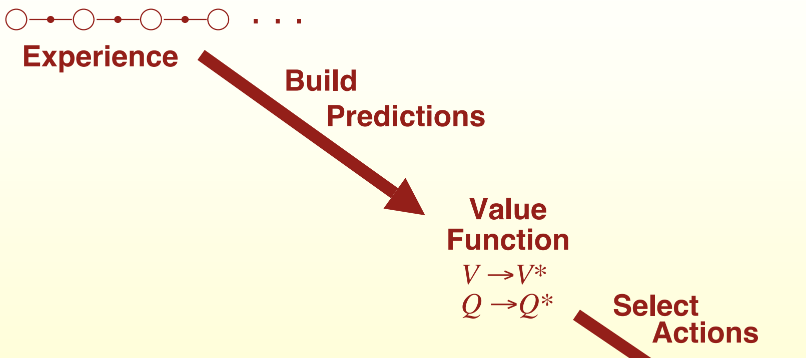
$$V^*(s) = \max_{\pi} V^{\pi}(s) \qquad Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

And corresponding optimal policies:

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

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

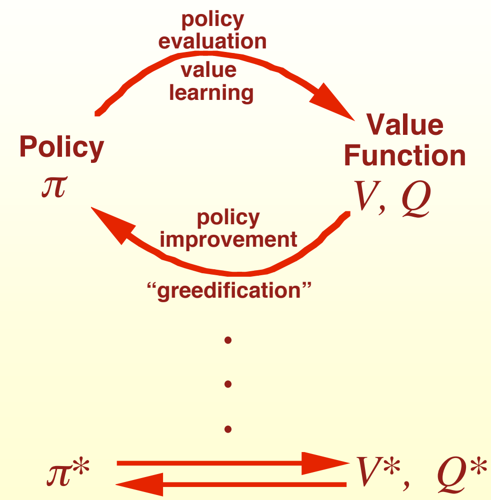
What Many RL Algorithms Do



Policy
$\pi \rightarrow \pi^*$

- Continual, online
- Simultaneous acting and learning

RL Interaction of Policy and Value



Passive Learning in a Known Environment

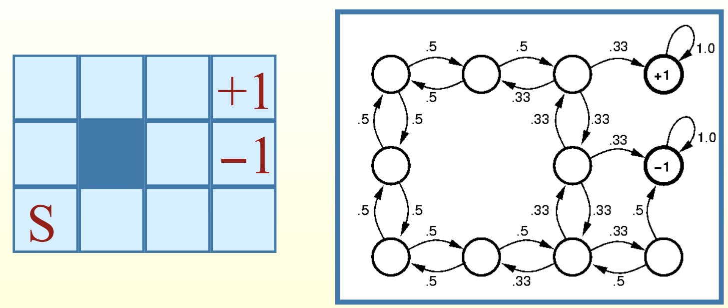
Given:

- A Markov model of the environment.
- States, with probabilistic actions.
- Terminal states have rewards/utilities.

Problem:

- Learn expected utility of each state.

Example



Percepts tell you:
[State, Reward, Terminal?]

Learning Utility Functions

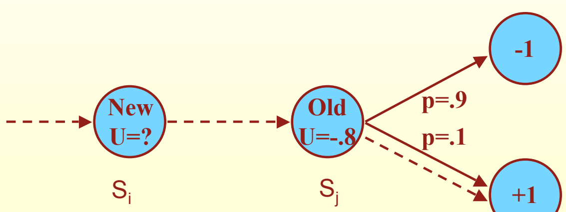
- A **training sequence** is an instance of world transitions from an initial state to a terminal state.
- The **additive utility assumption**: utility of a sequence is the sum of the rewards over the states of the sequence.
- Under this assumption, the utility of a state is the expected **reward-to-go**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

Naïve Updating

- 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.*
  - **Accumulate reward as you go back**
- Generates utility estimates that minimize the mean square error (LMS-update).

Problems with LMS-update

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



Adaptive Dynamic Programming

Utilities of neighboring states are mutually constrained:

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

Can apply dynamic programming to solve the system of equations (one eq. per state).

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

Temporal Difference Learning

Approximate the constraint equations without solving them for all states.

Modify U(i) whenever we see a transition from i to j using the following rule:

$$V(i) = V(i) + \alpha\,(R(i) + V(j) - V(i))$$

The modification moves V(i) closer to satisfying the original equation.

Rewriting the TD Equation with Discount Factor

Rewrite this

$$V(s) \leftarrow V(s) + \alpha\underbrace{\left[r + \gamma V(s') - V(s)\right]}_{\text{TD error}}$$

to get:

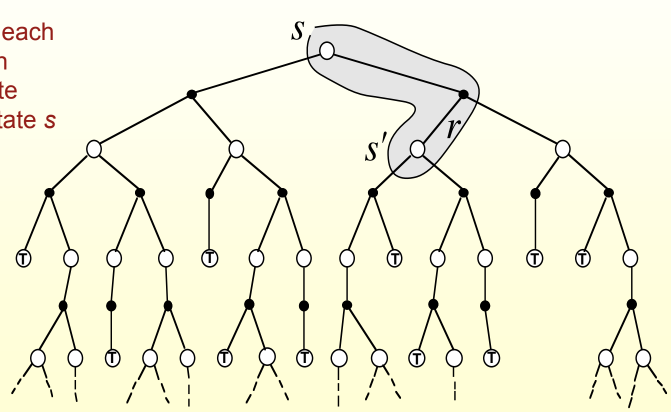
$$V(s) \leftarrow (1-\alpha)V(s) + \alpha\left[r + \gamma\, V(s')\right]$$

Temporal Difference (TD) Learning

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

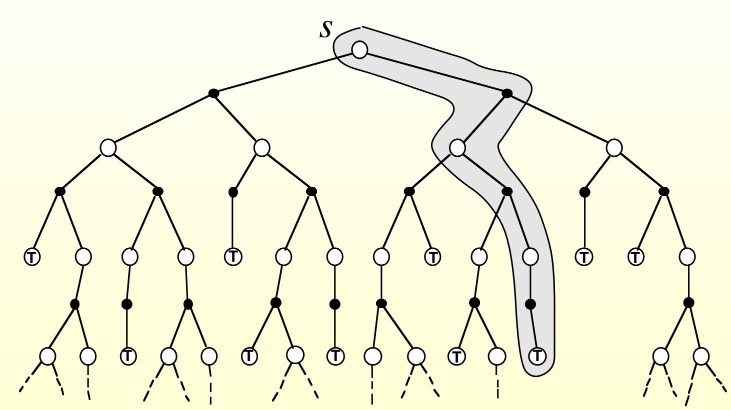
Sutton, 1988

After each action update the state *s*



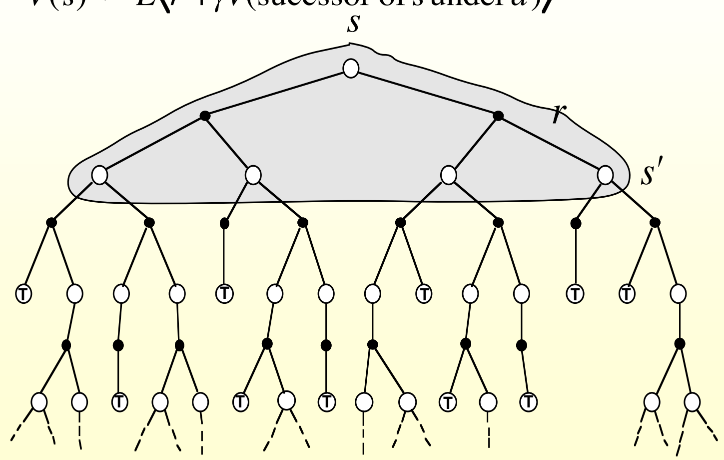
Simple Monte Carlo

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



Adaptive/Stochastic Dynamic Programming

$$V(s) \leftarrow E\langle r + \gamma V(\text{succssor of } s \text{ under } a)\rangle$$



Next Lecture

- **Q based Reinforecement Learning**
- **Additional Topics in Machine Learning**