Lecture 21: Learning - 5

Victor Lesser

CMPSCI 683
Fall 2004

Today's lecture

- **Continuation of Reinforcement learning**
  - Q-learning
- **Instance Based Learning**
  - Case-based learning

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, $t = 0, 1, 2, \dots$

$$\dots \; s_t \xrightarrow{a_t} \bullet \xrightarrow{r_{t+1}} s_{t+1} \xrightarrow{a_{t+1}} \bullet \xrightarrow{r_{t+2}} s_{t+2} \xrightarrow{a_{t+2}} \bullet \xrightarrow{r_{t+3}} s_{t+3} \xrightarrow{a_{t+3}} \dots$$

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 :

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

and each s,a pair: rewards

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

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

Optimal Value Functions and Policies

There exist optimal value functions:

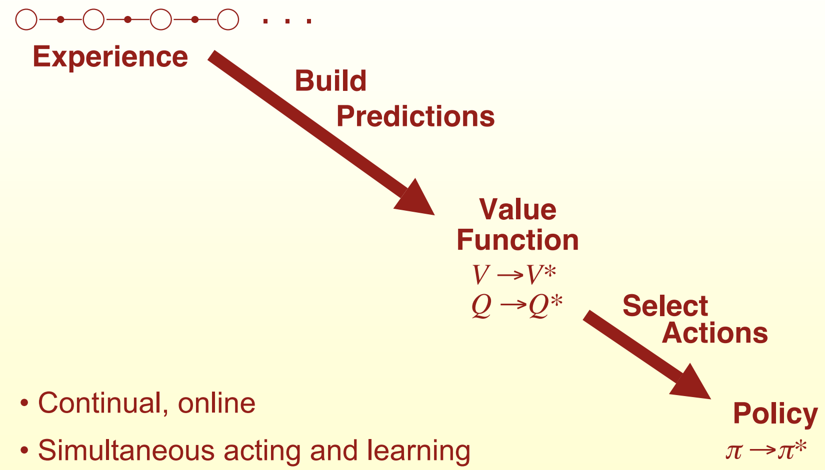
$$V^*(s) = \max_{\pi} V^{\pi}(s) \qquad Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

And corresponding optimal policies:

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

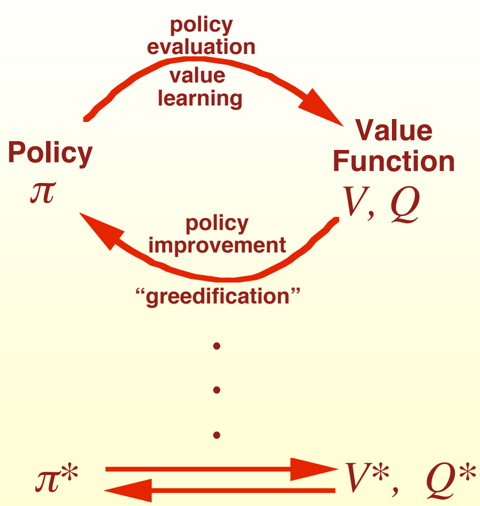
π^* **is the greedy policy with respect to** Q^*

What Many RL Algorithms Do



- Continual, online
- Simultaneous acting and learning

RL Interaction of Policy and Value



Passive Learning in a Known Environment

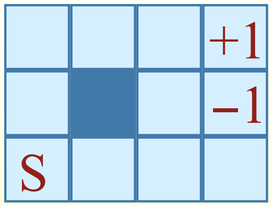
Given:

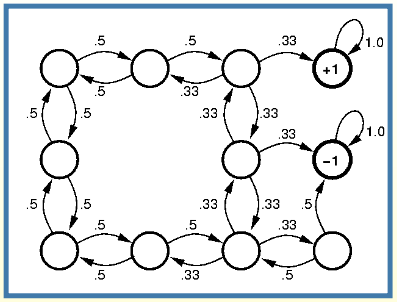
- A Markov model of the environment.
  - $P(s,s',a)$ – probability of transition from s to s' given a
  - $R(s,s',a)$ – expected reward on transition s to s' given a
- States, with probabilistic actions.
- Terminal states have rewards/utilities.

Problem:

- Learn expected utility of each state.

Example





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

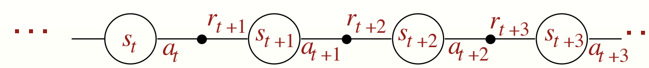
Learning Utility Functions

- 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.

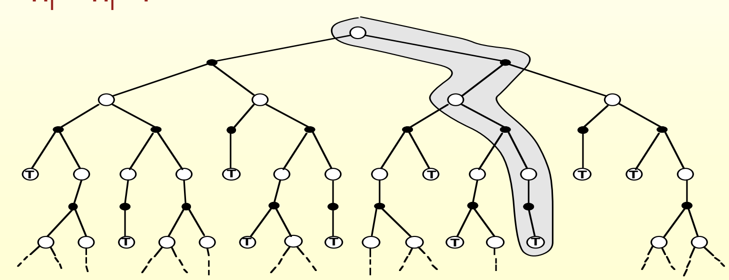
Naïve Updating

- 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).

Naïve Updating

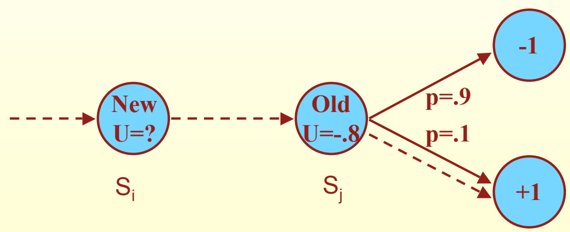


- **i =** s_t & j = s_{t+1}
- $U(i)_{n_i+1} = (R_i + U(i)_{n_i} * n_i + U(j)_{n_j})/ (n_i+1)$
- $n_i = n_i+1$



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:

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

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:
  - $\mathbf{U(i) = U(i) + \alpha\ (\underbrace{R(i) + U(j)}_{\text{New reward to go}} - U(i))}$
- The modification moves $U(i)$ closer to satisfying the original equation.

Rewriting the TD Equation with Discount Factor

Rewrite this

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

to get:

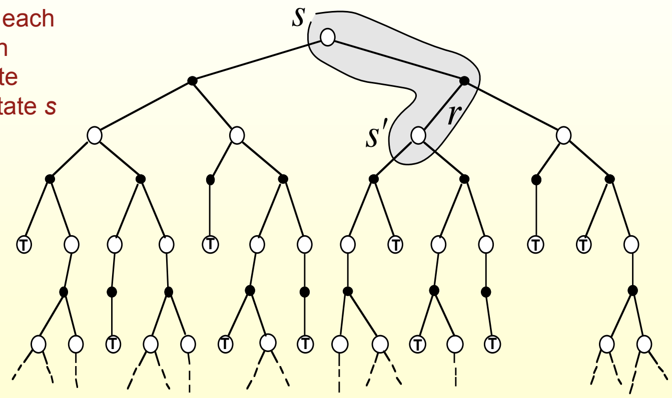
$$V(s) \leftarrow (1-\alpha) V(s) + \alpha \left[r + \gamma V(s') \right]$$

Temporal Difference (TD) Learning

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

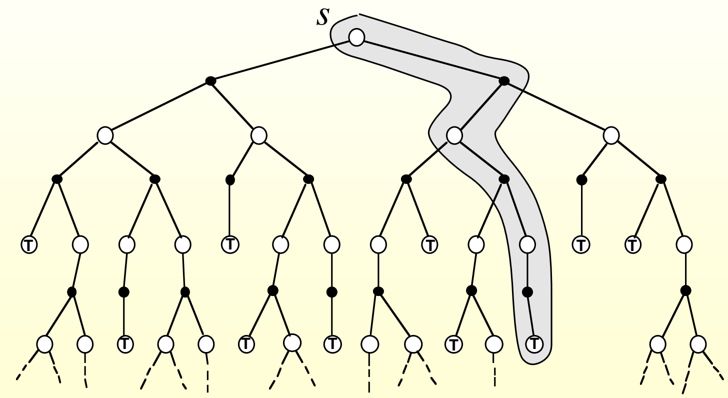
Sutton, 1988

After each action update the state *s*



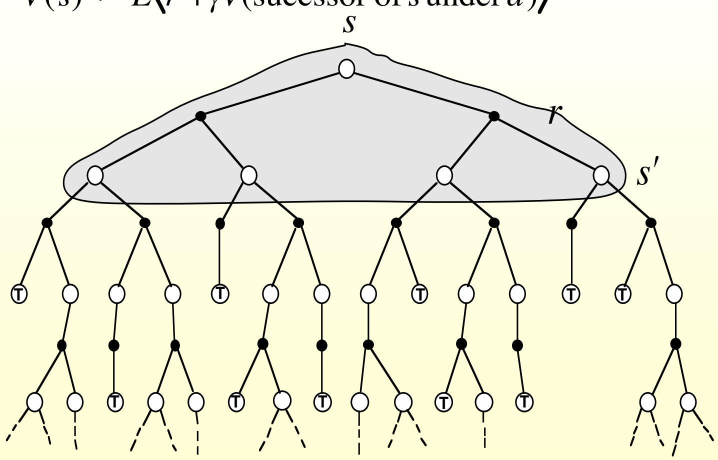
Simple Monte Carlo

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

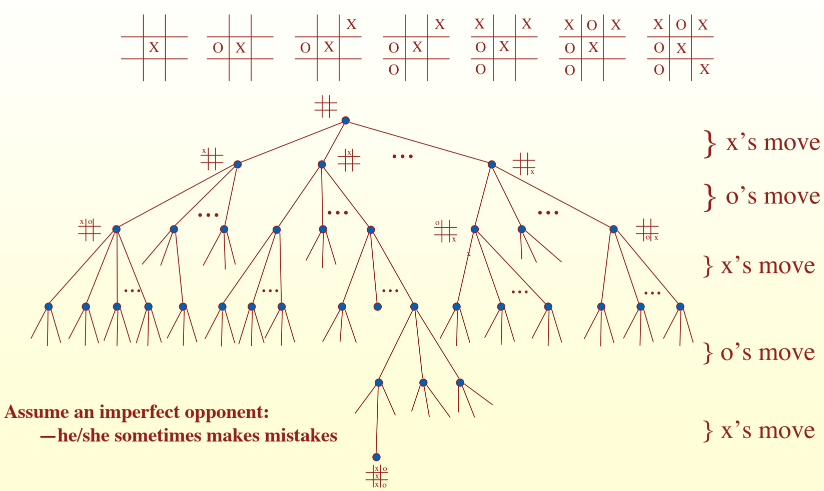


Adaptive/Stochastic Dynamic Programming

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



An Extended Example: Tic-Tac-Toe



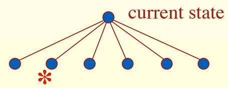
Assume an imperfect opponent:
—he/she sometimes makes mistakes

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	?
⋮	⋮	
x x x / o o	1	win
⋮	⋮	
x / x o / o	0	loss
⋮	⋮	
o x o / o x x / x o o	0	draw

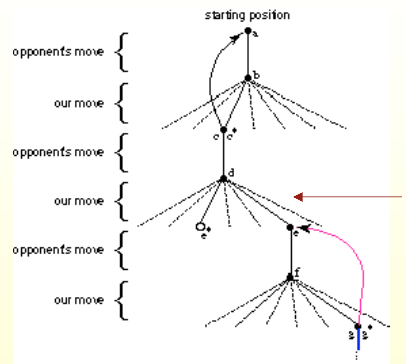
2. Now play lots of games.
To pick our moves, look ahead one step:

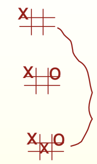


Just pick the next state with the highest estimated prob. of winning — the largest $V(s)$; a ***greedy*** move.

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

RL Learning Rule for Tic-Tac-Toe





Take action O

Non-deterministic outcome based on opponent's move

"Exploratory" 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., $\alpha = .1$

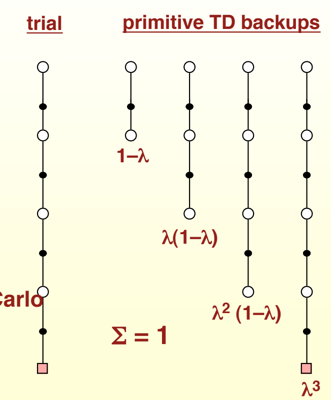
the ***step - size parameter***

More Complex TD Backups

e.g. TD (λ)

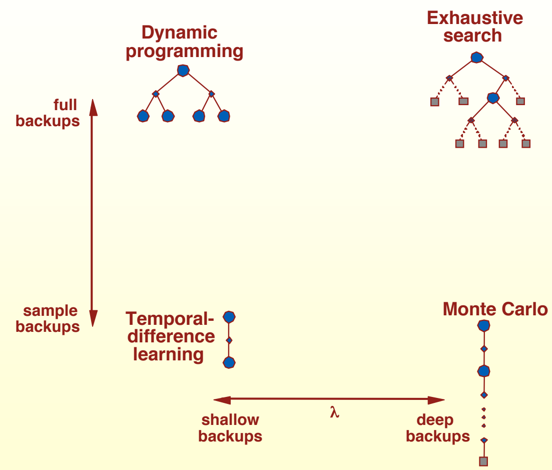
Incrementally computes a weighted mixture of these backups as states are visited

$\lambda = 0$ ⟶ Simple TD
$\lambda = 1$ ⟶ Simple Monte Carlo



Blending the backups

Space of Backups



Backup from all terminal states

Limitation of Learning V*

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

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

A problem:

- This works well if agent knows $\delta: S \times A \rightarrow S$ 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^*

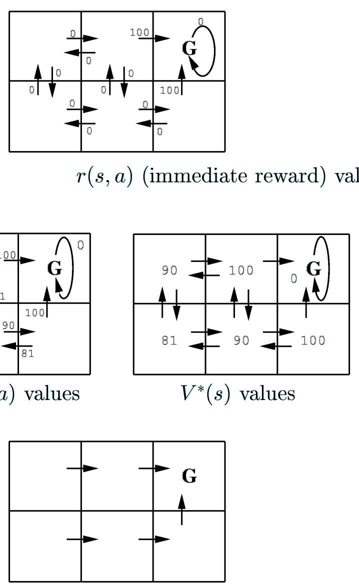
$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$

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 the agent will learn



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

$Q(s,a)$ values

$V^*(s)$ values

One optimal policy

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'

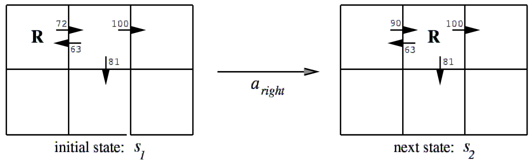
Q Learning for Deterministic Worlds

- 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

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

and

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

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- Temporal Difference Learning

***Q* learning: reduce discrepancy between successive *Q* estimates**

One step time difference:

$$Q^1(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

Two step time difference:

$$Q^2(s_t, a_t) \equiv r_t + \gamma\, r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

***N* step time difference:**

$$Q^n(s_t, a_t) \equiv r_t + \gamma\, r_{t+1} + \ldots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

Blend all of these:

$$Q^\lambda(s_t, a_t) \equiv (1-\lambda)[Q^1(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) \ldots]$$

The closer lamda is to the 1 the more important later differences

Q- Temporal Difference Learning, cont

$$Q^\lambda(s_t, a_t) \equiv (1-\lambda)[Q^1(s_t, a_t) + \lambda Q^2(s_t, a_t) + \lambda^2 Q^3(s_t, a_t) \ldots]$$

Equivalent expression:

$$Q^\lambda(s_t, a_t) = r_t + \gamma[(1-\lambda) \max Q(s_t, a_t) + \lambda Q^\lambda(s_{t+1}, a_{t+1})]$$

TD (λ) algorithm uses above training rule

- Sometimes converges faster than *Q* learning
- converges for learning V^* for any $0 \le \lambda \le 1$ (Dayan, 1992)
- Tesauro's TD-Gammon uses this algorithm

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, 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).

Getting the Degree of Abstraction Right

- **Time steps need not refer to fixed intervals of real time.**
- **Actions can be low level (e.g., voltages to motors), or high level (e.g., accept a job offer), "mental" (e.g., shift in focus of attention), etc.**
- **States can be low-level "sensations", or they can be abstract, symbolic, based on memory, or subjective (e.g., the state of being "surprised" or "lost").**

Memory-Based Learning

- **Encode specific experiences in memory rather than abstractions**
- **Carry out generalizations at the retrieval time rather than the storage time -- *lazy learning*.**
- **In their most general form:**
  - Based on partial match on a similarity metric, retrieve a set of cases/instances most "relevant" to the present context.
  - Adapt the retrieved experiences to new situations. This could be based on algorithms ranging from a simple k-nearest neighbor classification to chains of reasoning.

Instance-Based Learning

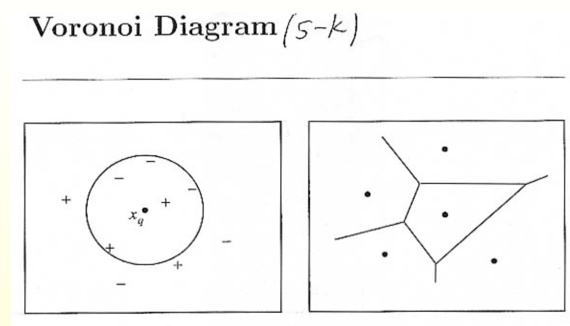
- **Key idea: just store all training examples $\langle x_i, f(x_i)\rangle$**
- **Nearest neighbor:**
  - Given query instance x_q, first locate nearest training example x_n, then estimate

$$\hat{f}(x_q) \leftarrow f(x_n)$$

- ***K*- Nearest neighbor**
  - Given x_q, take vote among its *k* nearest neighbors (if discrete-valued target function)
  - Take mean of *f* values of *k* nearest neighbors (if real-valued)

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

Voronoi Diagrams



On left, Positive and Negative Training Examples

- Nearest neighbor yes for x_q, 5-k classifies it as negative

On right, is decision surface for nearest neighbor, query in region will have same value

Distance-Weighted *k*NN

- **Might want weight nearer neighbors more heavily…**

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

- **Where**

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

- **And $d(x_q, x_i)$ is distance between x_q, x_i**
- **Note now it makes sense to use all training examples instead of just *k***
  - *Classification much slower*

Curse of Dimensionality

- Imagine instances described by 20 attributes, but only 2 are relevant to target function
- *Curse of dimensionality*: nearest neighbor is easily mislead when high-dimensional X
  - Similar to overfitting
- One approach:
  - Stretch *j*th axis by weight z_j, where $z_1, \ldots, z_n$ chosen to minimize prediction error
    - Length the axes that correspond to the more relevant attributes
  - Use cross-validation to automatically choose weights $z_1, \ldots, z_n$
    - Minimize error in classification
    - Setting z_j to zero eliminates this dimension all together

When to Consider Nearest Neighbor

- **Instances map to points in $\Re^n$**
  - Continuous real values
- **Less than 20 attributes per instance**
- **Lots of training data**
- **Advantages:**
  - Training is very fast
  - Robust to noise training data
  - Learn complex target functions
  - Don't lose information
- **Disadvantages:**
  - Slow at query time
  - Easily fooled by irrelevant attributes

Locally Weighted Regression: Approximating Real-Valued Function

- **Note *k*NN forms local approximation to *f* for each query point x_q**
- **Why not form an explicit approximation $\hat{f}(x)$ for region surrounding x_q**
  - Fit linear function to *k* nearest neighbors
    - $F(x)= w_0+w_1a_1+ \ldots\ldots w_na_n$
  - Fit quadratic….
  - Produces "piecewise approximation" to *f*
- **Several choices of error to minimize**
  - Squared error over *k* nearest to neighbors

$$E_1(x_q) \equiv \frac{1}{2x \in} \sum_{k \text{ nearest nbrs of } xq} (f(x) - \hat{f}(x))^2$$

  - Distance-weighted squared error over all neighbors

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 K(d(x_q, x))$$

  - …

Case-Based Reasoning

- **Can apply instance-based learning even when $X \neq \Re^n$**
- **Need different distance metric**
- **Case-Based Reasoning is instance-based learning applied to instances with symbolic logic descriptions**

```
((user-complaint error53-on-shutdown)
 (cpu-model PowerPC)
 (operating-system Windows)
 (network-connection PCIA)
 (memory 48meg)
 (installed-applications Excel Netscape VirusScan
 (disk 1gig)
 (likely-cause ???))
```

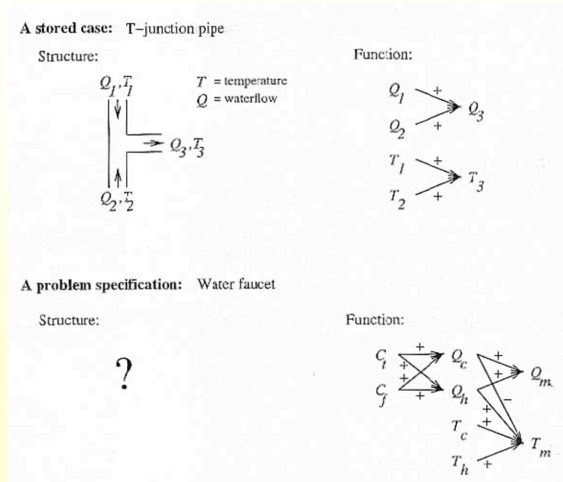

Ingredients of Problem-solving CBR

- **Key elements of problem solving CBR are:**
  - Cases represented as solved problems
  - Index cases under goals satisfied and planning problems avoided
  - Retrieve prior case sharing the most goals & avoiding the most problems
  - Adapt solution of prior case to solve a new case. May require re-solving problems and/or repairing solutions
  - Index new case and solution under goals satisfied and planning problems avoided

Case-Based Reasoning in CADET

- **CADET: 75 stored examples of mechanical devices**
  - Each training example: [qualitative function, mechanical structure]
  - New query: desired function
  - Target value: mechanical structure for this function
- **Distance metric: match qualitative function descriptions**
  - Size of largest subgraph between two function graphs

Case-Based Reasoning in CADET



Case-Based Reasoning in Chef

- **CHEF consists of six processes:**
  - **Problem anticipation**: the planner anticipates planning problems by noticing features in the current input that have previously participated in past planning problems
  - **Plan retrieval**: The planner searches for a plan that satisfies as many of its current goals as possible while avoiding the problems that it has predicted
  - **Plan modification**: The planner alerts the plans it has found to satisfy any goals from the input that are not already achieved
  - **Plan repair**: When a plan fails, the planner fixes the faculty plan by building up a casual explanation of why the failure has occurred and using it to find the different strategies for repairing it
  - **Credit assignment**: Along with repairing a failed plan, the planner wants to repair the characterization of the world that allowed it to create the failed plan in the first place. It does this by using the casual explanation of why the failure occurred to identify the features in the input that led to the problem and then mark them as predictive of it
  - **Plan storage**: The planner places successful plans in memory, indexed by the goals that they satisfy and the problems that they avoid

Beef-with-green-beans

A half pound of beef
Two tablespoons of soy sauce
One teaspoon of rice wine
A half tablespoon of corn starch
One teaspoon of sugar
A half pound of green bean
One teaspoon of salt
One chunk of garlic

- **Chop the garlic into pieces the size of matchheads**
- **Shred the beef**
- **Marinate the beef in the garlic, sugar, corn starch, rice wine and soy sauce**
- **Stir fry the spices, rice wine and beef for one minute**
- **Add the green bean to the spices, rice wine and beef**
- **Stir fry the spices, rice wine, green bean and beef for three minutes**
- **Add the salt to the spices, rice wine, green bean and beef**

Request for a New Recipe

Recipe for BEEF-AND-BROCOLI
Found nearest recipe is BEEF-WITH-GREEN-BEANS

Modifying recipe: BEEF-WITH-GREEN-BEANS
To satisfy: include broccoli in the dish

Placing some broccoli in recipe BEEF-WITH-GREEN-BEANS

-Considering ingredient-critic:
Before doing step: Stir fry the -Variable-
do: Chop the broccoli into pieces the size of chunks
-Ingredient critic applied

Chef alters old plans to satisfy new goals using a set of modification rules and a set of new objects

Check Whether New Recipe Works via Simulation

A half pound of beef
Two tablespoons of soy sauce
broccoli
One teaspoon of rice wine
A half tablespoon of corn starch

One teaspoon of sugar
A half pound of

One teaspoon of salt
One chunk of garlic

- **Chop the garlic into pieces the size of matchheads**
- **Shred the beef**
- **Marinate the beef in the garlic, sugar, corn starch, rice wine and soy sauce**
- **Chop the broccoli into pieces the size of chunks**
- **Stir fry the spices, rice wine and beef for one minute**
- **Add the broccoli to the spices, rice wine and beef**
- **Stir fry the spices, rice wine, broccoli and beef for three minutes**
- **Add the salt to the spices, rice wine, broccoli and beef**

The beef is now tender.
The dish now tastes savory.
The broccoli is not crisp.

The dish now tastes salty.
The dish now tastes sweet.
The dish now tastes like garlic.

Plan Repair

- ALTER-PLAN:SIDE-EFFECT: Replace the step that causes the violating condition with one that does not have the same side-effect but achieves the same goal
- SLPIT-AND-REFORM: Split the step into two separate steps and run them independently
- ADJUNT-PLAN:REMOVE: Add a new step to be run along with a step that causes a side-effect that removes the side-effect as it is created

Plan Storage

Indexing BEEF-AND-BROCCOLI under goals and problems:

If this plan is successful, the following should be true:
The beef is now tender.
The broccoli is now crisp.
Include beef in the dish.
Include broccoli in the dish.
Make a stir-fry dish.

The plan avoids failure exemplified by the state
“The broccoli is now soggy” in recipe BEEF-AND-BROCCOLI.

Problem Anticipation

Searching for plan that satisfies input goals-
- Include chicken in the dish.
- Include snow pea in the dish.
- Make a stir-fry dish.

Collecting and activating tests.
- Is the dish STYLE-STIR-FRY
- Is the item a MEAT
- Is the item a VEGETABLE
- Is the TEXTURE of item CRISP

Chicken+Snow pea+Stir Frying= Failure
"Meat sweats when it is stir-fried."
"Stir-frying in too much liquid makes crisp vegetables soggy."
Reminded of a failure in the BEEF-AND-BROCCOLI plan.
Failure= "The vegetable is now soggy"

Plan Retrieval

Driving down on: Make a stir-fry dish.
Succeeded-
Driving down on:
- Avoid failure exemplified by the state "The broccoli is now soggy" in recipe BEEF-AND-BROCCOLI

Succeeded
Driving down on: Include chicken in the dish
- Failed- Trying more general goal
  - Driving down on: Include meat in the dish
    - Succeeded

Driving down on: Include snow pea in the dish
- Failed-Trying more general goal
  - Driving down on: Include vegetable in the dish.
    - Succeeded

Found recipe→ REC9 BEEF-AND-BROCCOLI

Next Lecture

- **Analytical Learning (Explanation-Based Learning)**
  - First work done at Umass on learning rules of Baseball
- **A Quick Overview of Planning**