Outline

- Logistics
- Review
- Wrapper Induction
  - LR & HLRT Biases
  - Sample Complexity (Theory, Practice)
  - Recognizer Corroboration
- Reinforcement Learning
  - Markov Decision Processes
  - Value Iteration & Policy Iteration
  - Q Learning of MDP Models from Behavioral Critiques

Logistics

- One Class to Go...
- Learning Problem Set
- Project Status

Defining a Learning Problem

A program is said to **learn** from experience ***E*** with respect to task ***T*** and performance measure ***P***, if it's performance at tasks in T, as measured by P, improves with experience E.

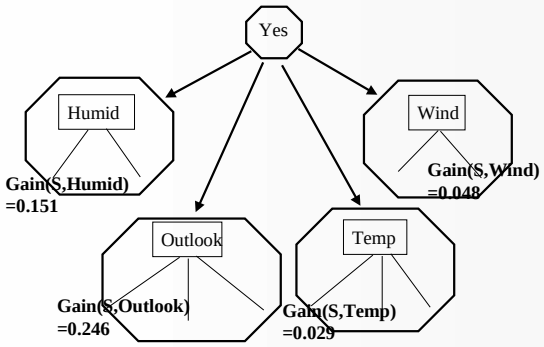
- Experience:
- Task:
- Performance Measure:
- Target Function
- Representation of Target Function Approximation
- Learning Algorithm

Which function first question?

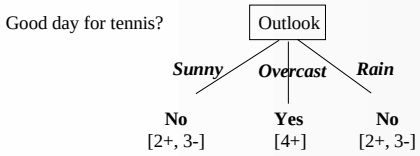
Concept Learning

- E.g. Learn concept "Good day for tennis"
  - Target Function has two values: T or F
- Represent concepts as decision trees
- Use ***hill climbing search***
- Thru space of ***decision trees***
  - Start with simple concept
  - Refine it into a complex concept as needed

Evaluating Attributes



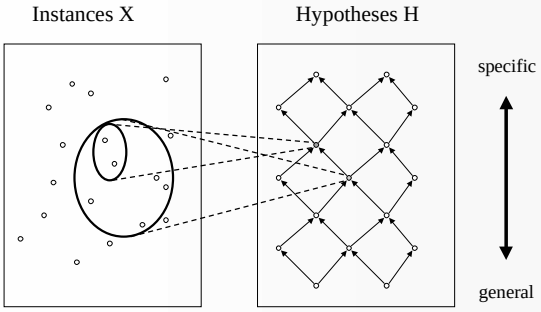
Resulting Tree ….



Summary: Learning = Search

- Target function = concept "edible mushroom"
  - Represent function as decision tree
  - Equivalent to propositional logic in DNF
- Construct approx. to target function via search
  - **Nodes:** decision trees
  - **Arcs:** elaborate a DT (making bigger + better)
  - **Initial State**: simplest possible DT (I.e. a leaf)
  - **Heuristic**: Information gain
  - **Goal**: No improvement possible ...
  - **Search Method**: ***hill climbing***

Correspondence
A hypothesis = set of instances



Version Space: Compact Representation

- Defn the ***general boundary*** G with respect to hypothesis space H and training data D is the set of **maximally general** members of H consistent with D
- Defn the ***specific boundary*** S with respect to hypothesis space H and training data D is the set of minimally general (**maximally specific**) members of H consistent with D

Training Example 3

<Rainy, Cold, High, Strong, Warm, Change> Good4Tennis=No

S_3 S_2 {<Sunny, Warm, ?, Strong, Warm, Same>}

G_3 {<Sunny,?,?,?,?,?>, <?,Warm,?,?,?,?>, <?,?,?,?,?,Same>}

↑

G_2 {<?, ?, ?, ?, ?, ?>}

Comparison

- Decision Tree learner searches a ***complete*** hypothesis space (one capable of representing any possible concept), but it uses an incomplete search method (hill climbing)
- Candidate Elimination searches an ***incomplete*** hypothesis space (one capable of representing only a subset of the possible concepts), but it does so completely.

Note: DT learner works better in practice

Two kinds of bias

- Restricted hypothesis space bias
  - shrink the size of the hypothesis space
  - PAC framework
  - Sample complexity as f(hypothesis language expressiveness)
- Preference bias
  - ordering over hypotheses

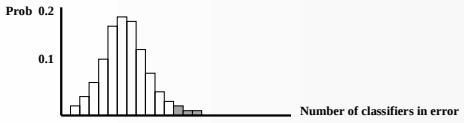
PAC Learning

- A learning program is program is probably approximately correct (with probability d and accuracy e) if given any set of training examples drawn from the distribution Pr, the program outputs a hypothesis f such that
- Pr(Error(f)>e) < d

- Key points:
  - Double hedge
  - Same distribution for training & testing

Ensembles of Classifiers

- Assume errors are independent
- Assume majority vote
- Prob. majority is wrong = area under biomial dist



- If individual area is 0.3
- **Area under curve for ≥11 wrong is 0.026**
- Order of magnitude improvement!

Constructing Ensembles

- Bagging
  - Run classifier *k* times on m examples drawn randomly with replacement from the original set of m examples
  - Training sets correspond to 63.2% of original (+ duplicates)
- Cross-validated committees
  - Divide examples into *k* disjoint sets
  - Train on *k* sets corresponding to original *minus* 1/k th
- Boosting
  - Maintain a probability distribution over set of training ex
  - On each iteration, use distribution to sample
  - Use error rate to modify distribution
    - Create harder and harder learning problems...

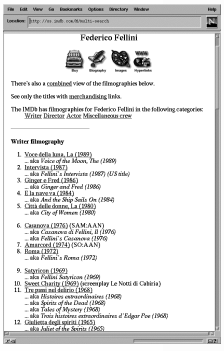
Review: Learning

- *Learning as **Search***
  - Search in the space of hypotheses
  - Hill climbing in space of decision trees
  - Complete search in conjunctive hypothesis representation
- *Notion of **Bias***
  - Restricted set of hypotheses (or preference order)
  - Strong bias means
    - ✔Greatly reduced sample complexity
    - ✘ Can't represent as many concepts
- Ensembles of classifiers:
  - Bagging, Boosting, Cross validated committees

Outline

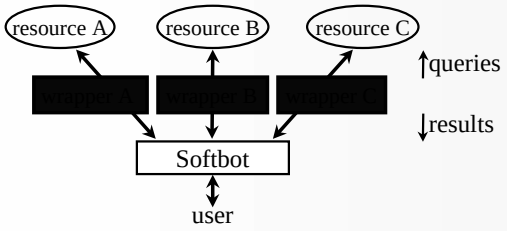
- Logistics
- Review
- Wrapper Induction
  - LR & HLRT Biases
  - Sample Complexity (Theory, Practice)
  - Recognizer Corroboration
- Reinforcement Learning
  - Markov Decision Processes
  - Value Iteration & Policy Iteration
  - Q Learning of MDP Models from Behavioral Critiques

Softbot Perception Problem



lots of
information
but
computers don't
understand
much of it

Strategy: ***Wrappers***



Scaling issues

Need custom wrapper for ***each*** resource.

```
<HTML><BODY BGCOLOR="FFFFFF" LINK=
"00009C" ALINK="00009C" VLINK="00009C"
TEXT= "000000">  <center>  <table><tr><td><NO
BR>  <NOBR><img src="/ypimages/b_r_hd_a.gif"
border=0 ALT="Switchboard Results" width=407
height=20 align=top><A HREF="/bin/cgiqa.dll?MEM
=1" TARGET ="_top"><img src="/ypimages/b_r_hd
_1.gif" border=0 ALT="People" width=54 height=20
align=top></A><A HREF="/bin/cgidir.dll?MEM=1"
TARGET="_top"><img src= "/ypimages/b_r_hd_2.gif"
border=0 ALT= "Business" width=62 height=24 align
=top></A><A HREF="/" TARGET="_top"><img src
="/ypimages /b_r_hd_3.gif" border=0 ALT="Home"
width=47 height=20 align=top></A></NOBR><br>
</td></tr></table>  </center><center><table border=0
width=576>  <tr><td colspan=2 align =center> <center>
```

→ useful information

But hand-coding is ***tedious***.

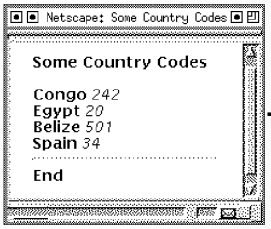
Wrapper Induction

[Kushmerick '97]

machine learning techniques
to automatically construct
wrappers from examples

→ wrapper procedure

Example



⟨ (Congo, 242)
(Egypt, 20)
(Belize, 501)
(Spain, 34) ⟩

LR wrappers: The basic idea

exploit ***fortuitous non-linguistic regularity***

<HTML><TITLE>Some Country Codes</TITLE>
<B>Congo</B> <I>242</I>
<B>Egypt</B> <I>20</I>
<B>Belize</B> <I>501</I>
<B>Spain</B> <I>34</I>
</BODY></HTML>

☞ Use **<B>**, **</B>**, **<I>**, **</I>** for parsing

Country/Code LR wrapper

procedure ExtractCountryCodes
  while there are more occurrences of <B>
    1. extract Country between <B> and </B>
    2. extract Code between <I> and </I>

☞ ***L***eft-***R***ight wrappers

procedure ExtractAttributes:
 while there are more occurrence of l_1
  1. extract 1st attribute between l_1 and r_1
  . . .
  K. extract Kth attribute between l_K and r_K

Observation

- **In principle**, a wrapper may be complex (an arbitrary procedure)
- **In this case**, it's very simple: 2k parameters
  - **<B>**
  - **</B>**
  - **<I>**
  - **</I>**
- **k = | Attributes |**

Assuming LR Nested-Loop Structure

Ubiquity!

wrapper class	useful?
LR	53 %
HLRT	57 %
OCLR	53 %
HOCLRT	57 %
N-LR	13 %
N-HLRT	50 %

total **70** %

"search.com" survey
AltaVista, WebCrawler, WhoWhere, CNN Headlines, Lycos, Shareware.Com, AT&T 800 Directory, ...

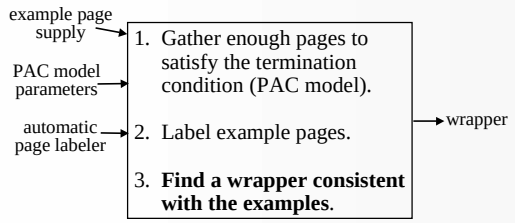
Inductive (example-driven) learning

examples → hypothesis

Thai food is spicy.
Vietnamese food is spicy.
German food isn't spicy.
→ Asian food is spicy.

→ wrapper

Wrapper induction algorithm



Step 3: Finding an LR wrapper

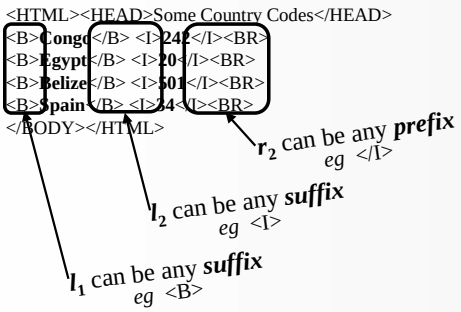
Example: Find **4** strings
⟨<B>, </B>, <I>, </I>⟩
⟨ l_1, r_1, l_2, r_2 ⟩

LR: Finding r_1

<HTML><HEAD>Some Country Codes</HEAD>
<B>**Congo**</B> <I>**242**</I>
<B>**Egypt**</B> <I>**20**</I>
<B>**Belize**</B> <I>**501**</I>
<B>**Spain**</B> <I>**34**</I>
</BODY></HTML>

r_1 can be any ***prefix***
eg </B or </B><

LR: Finding l_1, l_2 and r_2



Finding an LR wrapper: Algorithm

naïve algorithm
enumerate all combinations

```
for each candidate l1
  for each candidate r1
    . . .
      for each candidate lK
        for each candidate rK
          succeed if consistent
          with examples
```

O(S^{2K})

efficient algorithm
constraints are independent

```
for k = 1 to K
  for each candidate rk
    succeed if consistent with examples
for k = 1 to K
  for each candidate lk
    succeed if consistent with examples
```

O(**KS**)

S = length of examples
K = number of attributes

A problem with LR wrappers

Works for ...

- ✔ **AltaVista**
  www.altavista.digital.com
- ✔ **Yahoo People Search**
  www.yahoo.com/search/people
- ✔ and many more

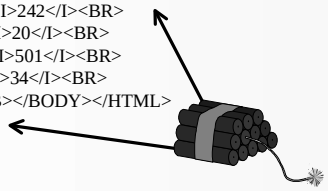
… but not

- ✘ **OpenText**
  search.opentext.com
- ✘ **Expedia World Guide**
  www.expedia.com/pub/genfts.dll
- ✘ and many more

The complication

Distracting text in head and tail

<HTML><TITLE>Some Country Codes</TITLE>
<BODY><B>Some Country Codes</B><P>
<B>Congo</B> <I>242</I>
<B>Egypt</B> <I>20</I>
<B>Belize</B> <I>501</I>
<B>Spain</B> <I>34</I>
<HR><B>End</B></BODY></HTML>



A solution: HLRT wrappers

*Ignore page's **head** and **tail***

<HTML><TITLE>Some Country Codes</TITLE> } head
<BODY><B>Some Country Codes</B> **<P>**
<B>Congo</B> <I>242</I>
<B>Egypt</B> <I>20</I>
<B>Belize</B> <I>501</I>
<B>Spain</B> <I>34</I>
<HR> <B>End</B></BODY></HTML> } tail

end of head

} body

start of tail

☞***H***ead-***L***eft-***R***ight-**T**ail wrappers

Country/Code HLRT wrapper

```
procedure ExtractCountryCodes
    skip past <P>
    while <B> before <HR>
        1. extract Country between <B> and </B>
        2. extract Code between <I> and </I>
```

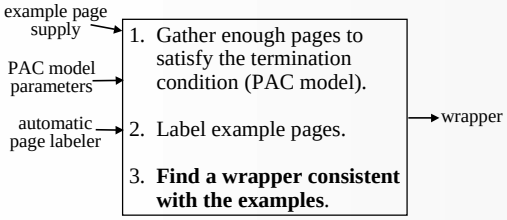
"Generic" HLRT wrapper

procedure ExtractAttributes:
skip past ***h***
while l_1 before ***t***
1. extract 1st attribute between l_1 and r_1
. . .
K. extract Kth attribute between l_K and r_K

HLRT wrapper ≡ **2*K*+2 strings**
$\langle h , t , l_1 , r_1 , \ldots, l_K , r_K \rangle$

head delimiter
tail delimiter
left delimiter
right delimiter
***K* = # attributes**

Wrapper induction algorithm



Step 3: Finding an HLRT wrapper

labeled pages ***wrapper***

$\langle h, t, l_1, r_1, \ldots, l_K, r_K \rangle$

Example: Find **6** strings
⟨<P>, <HR>, <B>, </B>, <I>, </I>⟩
⟨ ***h*** , ***t*** , l_1 , r_1 , l_2 , r_2 ⟩

HLRT: Finding r_1, l_2 and r_2

```
<HTML><TITLE>Some Country Codes</TITLE>
<BODY><B>Some Country Codes</B><P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR><B>End</B></BODY></HTML>
```

r_1 can be any ***prefix***

l_2 can be any ***suffix***

r_2 can be any ***prefix***

HLRT: Finding h, t, and l_1

h can be any ***substring ...***

```
<HTML><TITLE>Some Country Codes</TITLE>
<BODY><B>Some Country Codes</B><P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR><B>End</B></BODY></HTML>
```

t can be any ***substring ...***

l_1 can be any ***suffix ...***

… *such that* l_1 isn't confused by head or tail

Finding an HLRT wrapper: Algorithm

naïve algorithm
enumerate all combinations

```
for each candidate l1
  for each candidate r1
    . . .
      for each candidate lK
        for each candidate rK
          for each candidate h
            for each candidate t
              succeed if consistent
              with examples
```

efficient algorithm
constraints are mostly independent

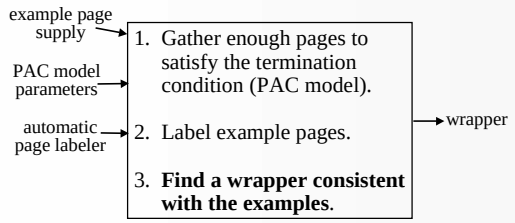
```
for k = 1 to K
  for each candidate rk
    succeed if consistent with examples
for k = 2 to K
  for each candidate lk
    succeed if consistent with examples
for each candidate h
  for each candidate t
    for each candidate l1
      succeed if consistent with examples
```

$O(S^{2K+2})$

S = length of examples
K = # attributes

$O(KS^2)$

Wrapper induction algorithm



Step 1. Termination condition

***Q**:* How many examples is enough?

A: Probabilistic model [Valiant, Kearns, …]

Want learned wrappers to be **"PAC"** (***Probably Approximately-Correct***)**:**

examine enough examples so that
with high probability,
the wrapper has ***high accuracy***.

PAC model

- Error of a hypothesis

$$E(h) \equiv \text{Prob}\left(\text{hypothesis } h \text{ is wrong on single instance selected randomly}\right)$$

- PAC criteria

$$\text{Prob}(E(h) > \varepsilon) < \delta$$

accuracy parameter $0 < \varepsilon < 1$

confidence parameter $0 < \delta < 1$

PAC model for HLRT

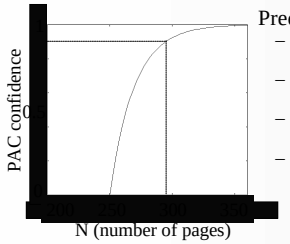
Theorem For ***any*** ε and δ, **if** wrapper ***w*** is consistent with a set of *N* examples such that

$$O(S^{5/3}) \cdot \left(1 - \frac{\varepsilon}{2}\right)^N < \delta$$

then ***w*** is PAC: Prob($E(\boldsymbol{w}) > \varepsilon$) < δ

N	=	number of examples
S	=	size of smallest example
ε	=	desired accuracy
δ	=	desired confidence

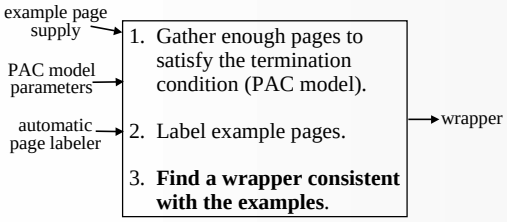
PAC model: Interpretation



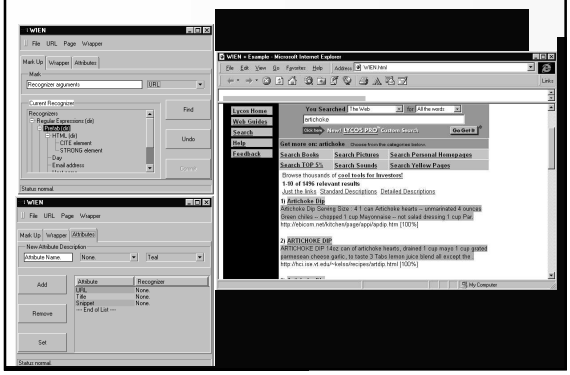
Predicted number of pages is

- *independent* of number of attributes
- *linear* in $1/\varepsilon$ (accuracy threshold)
- *logarithmic* in $1/\delta$ (confidence threshold)
- *logarithmic* in S (size of smallest example)

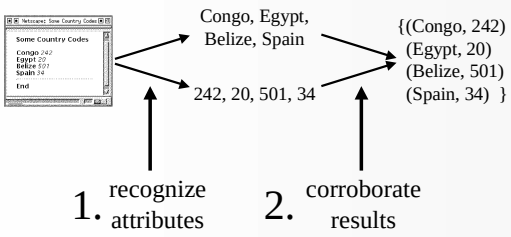
Wrapper induction algorithm



Step 2. WIEN: Manual page labeling



Automatic page labeling



Recognizers

A recognizer finds attribute instances

- *Regular expressions*
  telephone numbers, email addresses, URLs, dates, times, currency, countries, states, ISBN codes...
- *Indices, directories*
  companies, people, addresses, book titles
- *Natural language processing*

- Need wrappers even with perfect recognizers***!!***
  - **wrappers** must be fast
  - while **recognizers** may be slow

Corroboration of Imperfect Recognizers

false negatives \ false positives	no	yes
no	*perfect*	*incomplete*
yes	*unsound*	~~*unreliable*~~

Corroboration practical
with ≥ ***1 perfect*** recognizers
& ***no unreliable*** recognizers

Corroboration: Example

Country *incomplete*
10-15
50-55

+

Code *perfect*
18-20
38-40
58-60

+

Capital *unsound*
5-7
19-25
22-28
42-48
44-49
59-65
62-68
70-75

→

Ctry	**Code**	**Capital**
10-15	18-20	22-28
?	38-40	42-48⊕44-49
50-55	58-60	62-68⊕70-75

Key: a country occurs from positions 50-55

compact representation of labels consistent with recognizers

Summary of results

wrapper class	useful?	learnable?
LR	53 %	$O(KS)$
HLRT	57 %	$O(KS^2)$
OCLR	53 %	$O(KS^2)$
HOCLRT	57 %	$O(KS^4)$
N-LR	13 %	$O(S^{2K})$
N-HLRT	50 %	$O(S^{2K+2})$

total **70** %

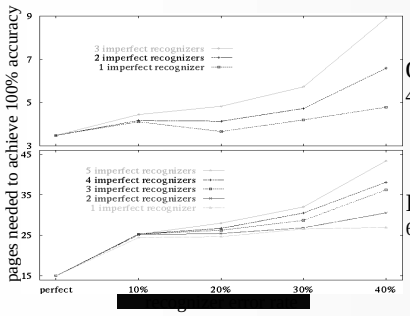
"search.com" survey
AltaVista, WebCrawler, WhoWhere, CNN Headlines, Lycos, Shareware.Com, AT&T 800 Directory, ...

time to automatically build wrappers
K = number of attributes
S = size of examples

Q: Is wrapper induction practical?

- Tested on several domains
  - OKRA email address locator
  - BigBook yellow-pages
  - AltaVista search engine
  - Corel stock photography catalog
- Measured
  - # pages needed for 100% accuracy on test suite as function of
  - recognizer error rates
- Overall performance
  - 0.2 CPU sec/attribute/KB → total ≈ 1 CPU minute
  - 4–44 pages needed for 100% accuracy

A: Yes



OKRA
4 attributes

BigBook
6 attributes

Kushmerick Contributions

Challenge: Lots of information, but computers don't understand most of it.

- Formalized wrapper construction as learning from examples
- Identified several wrapper classes: reasonably expressive, yet efficiently learnable
- Techniques for automatic page labeling

Outline

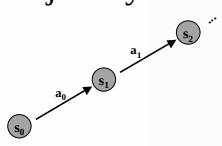
- Logistics
- Review
- Wrapper Induction
  - LR & HLRT Biases
  - Sample Complexity (Theory, Practice)
  - Recognizer Corroboration
- Reinforcement Learning
  - Markov Decision Processes
  - Value Iteration & Policy Iteration
  - Q Learning of MDP Models from Behavioral Critiques

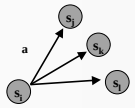
MDP Model of Agency

- Time is discrete, actions have no duration, and their effects occur instantaneously. So we can model time and change as $\{s_0, a_0, s_1, a_1, \ldots\}$, which is called a *history* or *trajectory*.
- At time *i* the agent consults a *policy* to determine its next action
  - the agent has "full observational powers": at time *i* it knows the entire history $\{s_0, a_0, s_1, a_1, \ldots, s_i\}$ accurately
  - policy might depend arbitrarily on the entire history to this point
- Taking an action causes a *stochastic transition* to a new state based on *transition probabilities* of the form $\text{Prob}(s_j \mid s_i, a)$
  - the fact that s_i and a are sufficient to predict the future is the *Markov assumption*

Transition Probabilities

Trajectory





Before executing a
What do you know?
$\text{Prob}(s_j \mid s_i, a)$,
$\text{Prob}(s_k \mid s_i, a)$,
$\text{Prob}(s_l \mid s_i, a)$, ...

MDP Model (continued)

- The agent has a *value function* that determines how good its course of action is.
  - value function might depend arbitrarily on entire history: $v(\{s_0, a_0, s_1, a_1, ...\}) \in \Re$
- The agent's behavior is evaluated over a *finite horizon* or in the limit over an *infinite horizon*.
- The agent's task is to construct a ***policy*** that maximizes the expectation of the value function over the specified horizon.

Good News and Bad News

- The theory provides a good account of purely deliberative, purely reactive, and hybrid behaviors
- The assumption of ***full observability*** makes the problem much easier
- Without some additional simplifying assumptions about the value function, it's still much too hard

MDP Model (continued)

- First simplifying assumption: value function is *time separable:*

$$\boxed{v(\{s_0, a_0, ...,\}) = \sum_i r(s_i, a_i) \quad (or) \quad \sum_i (r(s_i) + c(a_i))}$$

- Discounting: rewards earned early are better than rewards earned late
  - because of the economics
  - because $\exists$ some chance that the agent will be terminated
- Infinite-horizon discounted problems

$$v(\{s_0, a_0, ...,\}) = \lambda^i \sum_{i=0}^{\infty} r(s_i, a_i)$$

Properties of the Model

- Assuming
  - full observability
  - bounded and stationary rewards
  - time-separable value function
  - discount factor
  - infinite horizon
- Optimal policy is *stationary*
  - Choice of action a_i depends only on s_i
  - Optimal policy is of the form $\pi(s) = a$
    - which is of fixed size |S|, regardless of the # of stages

Computing Optimal Policies

- We can define the expected value of being in state *s* and acting according to a fixed policy π

$$v_\pi(s) = r(s, \pi(s)) + \lambda \sum_{s'} \Pr(s' \mid s, \pi(s)) \times v_\pi(s')$$

- A fundamental result is that the optimal policy v*(s) is a solution to the following equation (the Bellman equation):

$$v^*(s) = \arg\max_a r(s, a) + \lambda \sum_{s'} \Pr(s' \mid s, a) \times v^*(s')$$

Policy Construction and Dynamic Programming

- This suggests a dynamic programming approach to solving the problem:
  - start with some $\mathbf{v_0(s)}$
  - compute $\mathbf{v_{i+1}(s)}$ using the recurrence relationship

$$v_{i+1}(s) = \arg\max_a r(s, a) + \lambda \sum_{s'} \Pr(s' \mid s, a) \times v_i(s')$$

  - stop when computation converges to $\left\| v_{n+1} - v_n \right\| < \frac{\varepsilon(1-\lambda)}{2\lambda}$
  - convergence guarantee is $\left\| v_{n+1} - v^* \right\| < \frac{\varepsilon}{2}$

Value Iteration and Its Variants

- Value Iteration is a straightforward implementation of the recursive optimality equation.
  - Initialize $\mathbf{v_0}$ to some nominal value.
  - Compute $\mathbf{v_{i+1}}$ from $\mathbf{v_i}$
  - Terminate when $\| \mathbf{v_{i+1}} - \mathbf{v_i} \|$ is close
- Several variants of value iteration try to get faster convergence by using new values of $v_{i+1}(s)$ as soon as they become available

Policy Iteration

- Note: value iteration never actually computes a policy: you can back it out at the end, but during computation it's irrel.
- Policy iteration as an alternative
  - Initialize $\pi_0(s)$ to some arbitrary vector of actions
  - Loop
    - Compute $v_{\pi i}(s)$ according to previous formula
    - For each state s, re-compute the optimal action for each state

$$\pi_{i+1}(s) = \arg\max_a r(s,a) + \lambda \sum_{s'} \Pr(s' \mid s, \pi_i(s)) v_{\pi i}(s)$$

    - Policy guaranteed to be at least as good as last iteration
    - Terminate when $\pi_i(s) = \pi_{i+1}(s)$ for every state s
- Guaranteed to terminate and produce an optimal policy. In practice converges faster than value iteration (not in theory)
- Variant: take updates into account as early as possible.

Summary of MDP Solution Techniques

All are variants of dynamic programming, starting at stage *0* and using an optimal policy for *n* stages to build an optimal policy for *n+1* stages

The use of this backup technique depends crucially on a time-separable value function.

Convergence guarantee depends crucially on discount factor.

Tractability depends crucially on full observability.

Current work:

using structured representations and approximation methods to avoid having to examine the entire state space

working with undiscounted "planning-like" problems

extension to models with partial observability

Reinforcement Learning

- Continue studying infinite-horizon discounted fully observable problems
- We make an implicit assumption that "models are expensive, trials are cheap."
- The problem is to learn the model parameters based only on observed state and reward information
  - **Transition probabilities**
  - **Reward function and discount factor**
  - **Optimal policy**
- Two main approaches:
  - learn the model then infer the policy
  - **learn the policy without learning the explicit model parameters**

Q Learning

- The premise: learn the optimal action *a* for state *s* directly
- The function *Q(s, a)* is (an estimate of) the expected future reward associated with executing *a* in state *s*:

$$\boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a}) \equiv \boldsymbol{r}(\boldsymbol{s},\boldsymbol{a}) + \lambda \sum_{\boldsymbol{s}'} \Pr(\boldsymbol{s}'|\,\boldsymbol{s},\boldsymbol{a})\boldsymbol{Q}(\boldsymbol{s}',\boldsymbol{a})$$

  - from Q(s,a) the optimal action $\pi^*(s)$ is obtained by taking the max
  - we want to learn this Q function directly
- Learning framework: repeatedly
  - Takes some action dictated by the Q function
  - Gets some reward r
  - Updates Q function appropriately

Q Learning (cont.)

- What is the appropriate update from estimated $\hat{Q}_n$ to the updated $\hat{Q}_{n+1}$
  - to ensure that for all *s* and *a*, $\hat{Q}_n(s,a)$ converges to Q(s,a) as n goes to infinity
- The key is to adjust the $\hat{Q}$ values gradually with each iteration:

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

  - where one possible function for α is

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

Learning rate

Convergence of Q update

- The $Q^{\wedge}$ update converges to the Q(s,a) function (and thus to an optimal policy choice) if
  - rewards are bounded and discounted
  - initial Q values are finite
  - each (s,a) pair is visited infinitely often
  - $0 \leq \alpha_n < 1$
  - $\alpha_n(s,a)$ decreases with the number of times (s,a) is visited

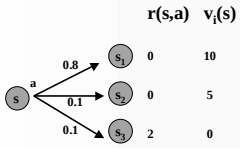
Summary of General MDP Model

- Input parameters:
  - A countable (finite) set of states, $S = \{s_1, \ldots, s_n\}$
  - A countable (finite) set of actions, $A = \{a_1, \ldots, a_m\}$
  - Action transitions: n^2m transition probabilities of the form $Prob(s_j \mid s_i, A)$
  - A value function of the form $v(\mathcal{H}) \rightarrow \Re$
    - mapping from *system trajectories* or *histories* into the real numbers
  - A fixed or infinite horizon N

Summary of Reinforcement Learning

- General problem is learning to act optimally based only on rewards accumulated from repeated trials
- Fundamental question is whether to learn the model explicitly
- Most techniques are based on the usual MDP formulation: full observability, infinite horizon, discounted total reward maximizing
- Most techniques guarantee convergence provided the state space is "fully explored"
  - if this is not the case---if the agent is to be "deployed" before training is complete, there is some advantage to exploration: acting suboptimally in order to learn more
  - the tradeoff between the expected value of exploration and expected value of acting optimally can be represented formally (though weakly)

Simple Backup



$$v_{i+1}(s) = \arg\max_a r(s,a) + \lambda \sum_{s'} \Pr(s' \mid s, a) \times v_i(s')$$

v_{i+1} =