Reinforcement Learning, cont.

Models an agent trying to learn “what to do” / “how to act” in a complex environment.

So far, we have seen passive learning
We learned utility values for a fixed world with fixed transitions (no real “actions”)

Key issue: delayed rewards / feedback.
(R&N chapter 20.)
Example environment next page.
Considered strategies:

(a) “Sampling” (Naive updating)
   Problem: slow.

(b) “Calculation” / “Equation solving”
   Problem: to many states.

(c) “in between (a) and (b)”
   (Temporal Difference Learning — TD learning)
   good compromise of (a) & (b).
Recap: Adaptive dynamic programming

Intuition:

Consider \( U(3, 3) \)

From figure we see that

\[
U(3, 3) = 0.33 \times U(4, 3) + 0.33 \times U(2, 3) + 0.33 \times U(3, 2)
\]

\[
= 0.33 \times 1.0 + 0.33 \times 0.0886 + 0.33 \times -0.4430
\]

\[
= 0.2152
\]

Equation solving!!

Slide CS472–5

Recap: Temporal difference learning

When observing a transition from \( i \) to \( j \),

bring \( U(i) \) value closer to that of \( U(j) \)

Use update rule:

\[
U(i) \leftarrow U(i) + \alpha(R(i) + U(j) - U(i))
\]

\( \alpha \) is the learning rate parameter

rule is called the temporal-difference or TD

equation. (we take the difference between successive states.)

Slide CS472–6
Issues:

What if environment is unknown?

What if agent can decide on actions?

How do we link this to game playing?

Active Learning

Let $M_{i,j}^a$ denote the probability of reaching state $j$ when executing action $a$ in state $i$

Consider: $M_{i,j}^a = 1$ for exactly one value of $j$ for each given $i$ and $a$.

what are the utility values of the states in that case?

This is e.g., the case in chess.

What is the “problem”?

Different in backgammon.
A **rational** agent will now optimize its **expected utility**. We get:

\[ U(i) = R(i) + \max_a \sum_j M_{ij}^a U(j) \]

Again, a weighted sum of the states you can reach. Pick action with maximal expected return.

**Slide CS472–9**

Note: \( R(i) \), rewards in end states; ground the model.

Standard TD equation can still be used!

\[ U(i) \leftarrow U(i) + \alpha (R(i) + U(j) - U(i)) \]

One more issue left . . . .

**Slide CS472–10**
How do we select actions?

How about: action with maximum expected utility?
(Note: using current estimates.)

Purely rational.

*What could be the problem?*

---

Issue: **Exploration.**

Always using optimal utility may get you stuck in some strategy found early on and that works reasonably well.

That way you don’t explore unknown, risky, but possibly much better strategies.

(It’s like finding your way around in a new city without exploring new many routes.)
Alternatives:
Pick random transitions.
   Very effective in exploration but
   bad for finding good utility values
   “i.e., short paths”

We’ll have to do something in between
i.e., a strategy that is encouraged
to explore unknown transitions
initially but later on converges to
maximal expected utility.

Pretty close to real life. :-)

Let’s consider an environment to test
the tradeoffs.

It’s a stochastic environment (R&N section 17.1).
Action (“move”) succeeds with 80%
but with 20% fail; move perpendicular.
(10% left / 10% right)
cost per step 0.04 (i.e., reward -0.04).
Note the optimal strategy.
Why does it make intuitive sense?
This strategy picks action with maximum expected utility at each square.

*But can we learn these utilities?*
Two strategies:

**Random**

**Greedy**
- Do what’s best according to current maximum expected utility.

See figure 20.10 from R&N, page 610.
As we might expect, with the random strategy we learn the environment and utilities corresponding to that strategy very well. (We explore the whole space. **But**, we “lose” a lot! (hit -1)

With the greedy strategy, we get a reasonable overall strategy (hit “-1” not very often) **But**, we don’t learn utilities very well. We often get stuck in path along bottom (to left).

**Slide CS472–21**

We can fix the problem by choosing a strategy in between random and greedy.

Basically uses an “exploration function”, which takes into account $N(a, i)$, the number of times action $a$ has been tried in state $i$.

It steers the search towards rarely tried actions.

After a while, when all actions in all states have been tried, the function converges to the standard utility function.

$$U^+(i) = R(i) + \max_{a} f(\sum_{j} M_{i,j}U^+(j), N(a, j))$$

**Slide CS472–22**
Utility estimates

Number of iterations

(4,3)  
(3,3)  
(2,3)  
(1,1)  
(3,1)  
(4,1)  
(4,2)  

RMS error, policy loss (exploratory policy)

Number of epochs

Slide CS472-23

Slide CS472-24
So far, we’ve learned utilities and then implemented some strategy for taking actions.

Why not learn directly “what to do in each state”? Can be done. We’ll learn the action-value function $Q(a, i)$, which gives us the value of action $a$ in state $i$. Again, we normally take the action with the highest $Q$ value. Why not always?

\[ U(i) = \max_a Q(a, i) \]

At equilibrium (analogous to equation for utilities):

\[ Q(a, i) = R(i) + \sum_j M_{i, j} \max_{a'} Q(a', j) \]

Again, look at $Q$ values of states $j$ as reached from $i$. Finally: TD $Q$-learning gives us:

\[ Q(a, i) \leftarrow Q(a, i) + \alpha (R(i) + \max_{a'} Q(a', j) - Q(a, i)) \]

after a transition from $i$ to $j$. 
TD-learning raises an interesting issue:
Why not just “learn what to do” by exploring and (Q) TD-learning?

Advantage: Need no model of environment.
(No knowledge representation!)

Current view: approach breaks down
when environment gets complex.
need to build some background model and use it.
(possibly in learning)

Practice
TD-learning and Q TD learning as we’ve seen so far
work nicely for small state spaces
(up to approx. 1000 states).
Note we use an explicit table to represent
utility function.
Interesting spaces are much, much bigger!!

10^{50} for backgammon
10^{2000} for automatic driving!
(input camera to actions)
   Aside: how would reinforcement learning approach
differ from neural net approach we saw before?
Solution

Two alternatives:
1) Use feature representation of states.
   Maps many states into same set of features
   (e.g. use 30 features to represent chess board.)
   TD learning learns utility function
   on those features (not of individual states).

2) Use function approximation of real utility function.
   Most used: use neural net to approximate and
   learn function. Input neural net: state of world
   (game); output: utility value.

These approaches give some of the best examples of
interesting learning applications.

For 1), we have Samuel’s checker player.

For 2), we have Tesauros’s backgammon player.
   world champion level; (almost) pure self play!
Fundamental Questions

Does good approx. utility (evaluation) function exist for chess / for go?
Can we learn it?
Note: Tradeoff between learning and (minimax) search.
In fact, a good utility function based on features has flavor of “pattern recognition”.
Patterns may be linked to deep strategies.
Reinforcement learning: Links problem solving and learning.

Slide CS472–31

The End

Slide CS472–32