L17: Approximate off-line MDP methods

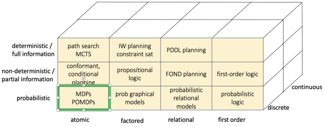
KAlg: 8.1, 8.2, 8.3, 8.6, 8.7

What you should know after this lecture

- How to use offline methods to find policies for continuous-space MDPs
- Some ideas about handling continuous-action MDPS
- Relationship to reinforcement learning

Probabilistic sequential decision-making

Probabilistic transitions
Atomic, discrete states
Full observability
Solution is a policy



	atomic	factored	relational	first order
deterministic / full information	path search MCTS	IW planning constraint sat	PDDL planning	
non-deterministic / partial information	conformant, conditional planning	propositional logic	FOND planning	first-order logic
probabilistic	MDPs POMDPs	prob graphical models	probabilistic relational models	probabilistic logic

- Agent can observe current state completely and correctly
- World dynamics are probabilistic and known to the agent
- Agent selects actions to maximize expected summed rewards over time
- Agent plans on-line to select next action based on current state (but still potentially thinking about a longer horizon)

Recall: Value iteration

```
ValueIteration($\mathcal{S}, \mathcal{A}, T, R, \gamma, \epsilon$)
Q(s, a) = 0 for $s \in \mathcal{S}$, $a \in \mathcal{A}$
while True:
    for $s \in \mathcal{S}$, $a \in \mathcal{A}$:
        $Q_{\text{new}}(s, a) = \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \max_{a'} Q(s', a')\right]$
    if $|Q - Q_{\text{new}}| < \epsilon$:
        return $Q_{\text{new}}$
    $Q = Q_{\text{new}}$
```

Continuous state-space MDPs

- There is no exact general-purpose algorithm.
- This problem (with a known model) is actually what most current applications of reinforcement-learning are trying to solve![1]
- There is a large collection of RL-like methods for addressing continuous MDPs, including
    - Value-iteration-like methods: fitted Q, approximate value iteration, approximate dynamic programming, neuro-dynamic programming, deep Q learning, ...
    - Policy-iteration-like methods: actor-critic RL methods represent both value function <u>and</u> policy; can apply to continuous action spaces as well.
    - Policy gradient methods: no value function at all—just gradient ascent in the space of expected value of executing the policy.

[1] RL was originally intended to be a model of how real creatures, e.g., honeybees, learn from small amounts of experience in small spaces.

Approximate value iteration

```
ApproximateVI($mdp$, $N$, $T$, $k$)
// N: number of sample states // T: number of iterations
// k: number of backup samples
$S = $ sampleStates($mdp$, $N$)
$Q = \{a : $ Fit($S, zeros(N)$)$\}$
for $t \in 1..T$:
    for $a \in mdp.\mathcal{A}$:
        $Y = [backup(s, a, Q, mdp, k)$  for $s \in S]$
        $Q[a] = $ Fit($S, Y$)
return $Q$
```

Becomes exact value iteration when:

- $mdp.\mathcal{S} = S$ is discrete
- $Q[a].predict(s_i) = y_i$: we remember our training data
- $backup(s, a, Q, mdp) =$ $\sum_{s' \in mdp.\mathcal{S}} mdp.T(s, a, s') \left[mdp.R(s, a, s') + \gamma \max_{a'} Q[a'].predict(s')\right]$

Approximate backup

When there are a large or infinite number of s' such that $P(s' \mid s, a) > 0$ we can't compute an exact backup.

- So, we sample k possible s' (can also sample r):

$$\mathit{backup}(s, a, Q, \mathit{mdp}) = \frac{1}{k} \sum_{\{(s',r) \sim \mathrm{TR}(s,a)\}_k} r + \gamma Q[a].\mathit{predict}(s')$$

- How to pick k? Computation time vs variance trade-off.
- Just need a generative model *mdp*.TR that we can call to get samples (but not explicit T)

Function approximation

Assume a function-approximation module with interface:

- $f = \textsc{Fit}(X, Y)$: takes a sequence of states, X, and a sequence of values Y and returns the state of an approximator
- $f.\textsc{predict}(x)$: approximator takes a query state x and returns a predicted value

KAlg chapter 8 talks about a lot of different ones. My favorite is kernel smoothing (also known as kernel regression):

- $\textsc{fit}(X, Y)$ just remembers X and Y
- Prediction is weighted combination of all the y values:

$$f.\textsc{predict}(x) = \frac{\sum_{i=1}^{N} k(x_i, x) y_i}{\sum_{i=1}^{N} k(x_i, x)}$$

where (if σ is big, this is average y; if small, nearest neighbor)

$$k(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$$

Neural networks!

Another way to meet this spec is a neural net

- $\text{FIT}(X, Y)$ does supervised regression (be sure not to have any output non-linearity) and returns weights θ
- $\theta.\text{PREDICT}(x)$ is just a forward pass on the trained network

To be incremental or not? We can choose:

- To re-initialize the network in each iteration of approximate value iteration.
- To train, in each iteration, starting from the previous Q networks' values.

Sampling states

In a low-dimensional problem, it is reasonable for sampleStates to generate an evenly-spaced grid of samples.
In high-dimensional problems, this is intractable. So:

- If the horizon is long and the reward is “sparse” (doesn’t give you any local signals about which parts of the space are better) then there’s nothing you can do.
- Otherwise, take the RL connection more seriously:
    - Assume an initial state or initial state distribution
    - Let π_Q be the “greedy” policy based on the current Q values
    - Gather new S on each iteration of avi, by starting at an initial state, and executing π_Q, but with some “exploration”, like:
        - with probability ϵ execute a random action instead of π_Q
        - try to take actions that will lead to previously un-visited parts of the state-space
    - Combine the states visited in this process with previous S (with some strategy for keeping S from growing too large, but also avoiding oscillations—replay buffer.)

Q Learning in simulation

This becomes Q-learning when you:

- Use your MDP model to build a simulator.
- Choose actions in a way that is mostly greedy wrt Q
- Update your Q values a small amount after each interaction

```
$\textsc{SimulatedQ}(mdp, T)$
$s = mdp.s_0$
$Q = \{a : \textsc{Fit}(S, zeros(N))\}$
for $t \in 1..T$:
    $a = \textsc{epsilonGreedy}(s, Q)$
    $s', r = mdp.\text{TR}(s, a)$
    $y = r + mdp.\gamma Q[a].predict(s')$
    $Q[a].\textsc{update}(s, y)$                // One neural network update
    $s = s'$
return $Q$
```

Handling continuous actions

- Value-iteration style:
    - Train a single function-approximator to represent $Q(s, a)$
    - Solve a continuous optimization problem to find $\pi_Q(s) = \mathsf{argmax}_a Q(s, a)$
    - Can be difficult in practice, both because optimization is hard and because it tends to find adversarial examples in your network. Can add a term that constrains it to be "close" to your training examples.
- Policy-iteration style:
    - Make a <u>policy network</u> π and <u>value network(s)</u> Q
    - Conceptually, alternate:
        - Use Q to generate (s, a) data for supervised training of the policy π_Q
        - Execute learned policy to get more data for training Q_π
    - In fact, you can do both in parallel, by fiddling with learning rates, etc.

Next time

- Introduction to POMDPs!